Chapter 3

Structured Program Development

Chapter 3 - Structured Program Development

Outline

3.1 Introduction
3.2 Algorithms
3.4 Control Structures
3.5 The If Selection Statement
3.6 The If…Else Selection Statement
3.7 The While Repetition Statement
3.8 Formulating Algorithms: Case Study 1 (Counter-Controlled Repetition)
3.9 Formulating Algorithms with Top-down, Stepwise Refinement: Case Study 2 (Sentinel-Controlled Repetition)
3.11 Assignment Operators
3.12 Increment and Decrement Operators

3.1 Introduction

- Before writing a program:
  - Have a thorough understanding of the problem
  - Carefully plan an 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
  - The order in which these actions are to be executed
- Program control
  - Specify order in which statements are to be executed

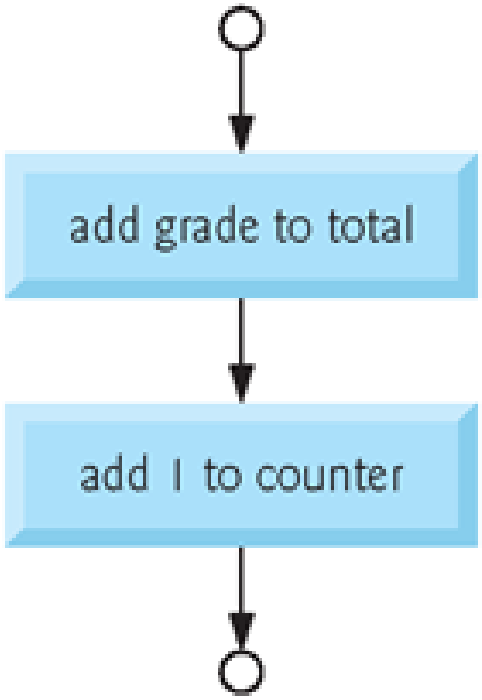
3.4 Control Structures

- Sequential execution
  - Statements executed one after the other in the order written
- Transfer of control
  - If you use the ***goto*** statement, the next statement executed will be the transferred location, NOT the next location in sequence
  - Overuse of `goto` statements led to many problems

- All programs written in terms of 3 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`

3.4 Control Structures

Figure 3.1 Flowcharting C's sequence structure.




```
total = total + grade ;
```

```
counter = counter + 1 ;
```

3.5 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

3.5 The `if` Selection Statement

- Statement in C:

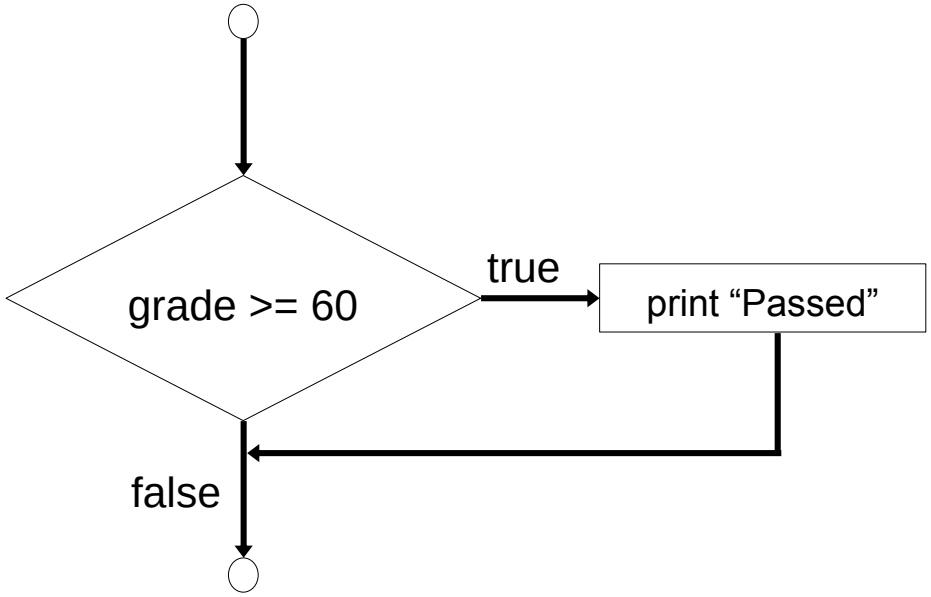
```
if ( grade >= 60 )
   printf( "Passed\n" );
```

  - C code corresponds closely to the pseudocode

- Diamond symbol (decision symbol) in flowcharts
  - Indicates decision is to be made
  - Contains an expression that can be `true` or `false`
  - Test the condition, follow appropriate path

3.5 The `if` Selection Statement

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



3.6 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

3.6 The `if...else` Selection Statement

- C code:

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

Ternary Conditional Operator

- Ternary conditional operator (`? :`)
  - Takes three arguments (condition, value if `true`, value if `false`)

Condition `?` *Yes-part* `:` *No-part*

  - Our pseudocode could be written:

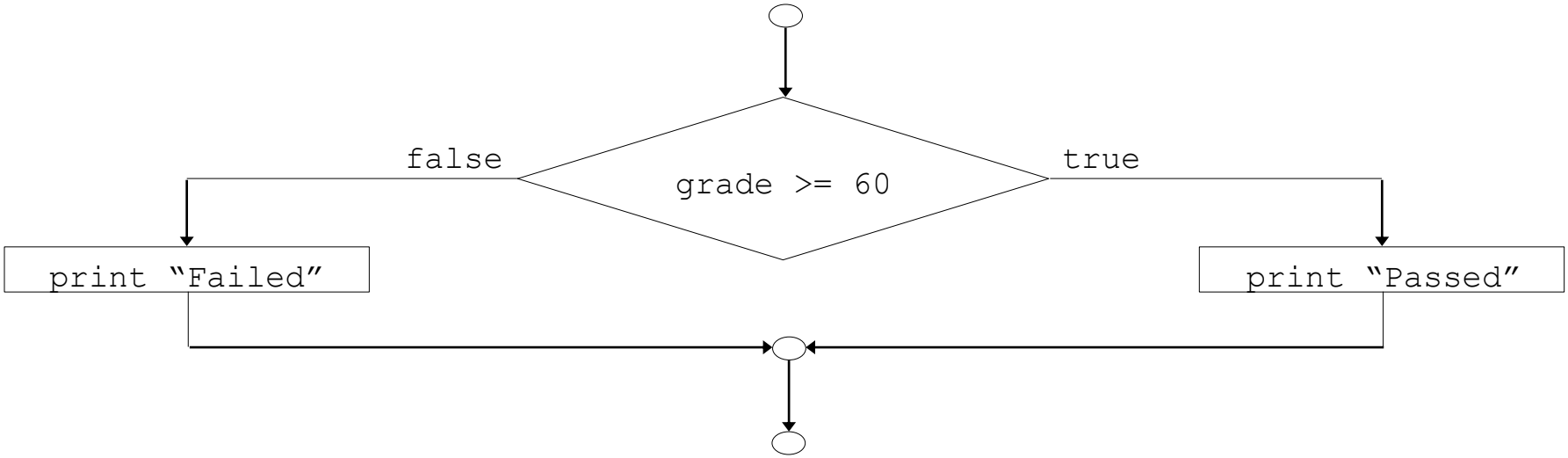
```
printf( "%s\n", grade >= 60 ? "Passed" : "Failed" );
```

  - Or it could have been written:

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

3.6 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

3.6 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"
```

3.6 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

3.6 The `if...else` Selection Statement

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

Example1:

Determining whether a number is odd or even

```c
#include <stdio.h>

int main()
{
 int a;

 printf("Bir sayı giriniz : ");
 scanf("%d", &a);

 if ( a % 2   ==   0 )
       printf("Çift sayıdır\n");
 else
       printf("Tek sayıdır\n");
}
```

Example2

```
#include <stdio.h>

int main()
{
      if (-1)
          printf("SONUC TRUE\n");
      else
          printf("SONUC FALSE\n");
}
```

A decision can be made on any expression.

Zero means **false**

Nonzero means **true**

Examples:

-1 is true

1 is true

0 is false

Program Output

```
SONUC TRUE
```

3.7 The `while` Repetition Statement

- Repetition structure
  - Programmer specifies an action to be repeated while some condition remains `true`
  - Psuedocode:

    Set Product to 1

    While Product is less than or equal to 1000

    Multiply Product by 2

    Print Product

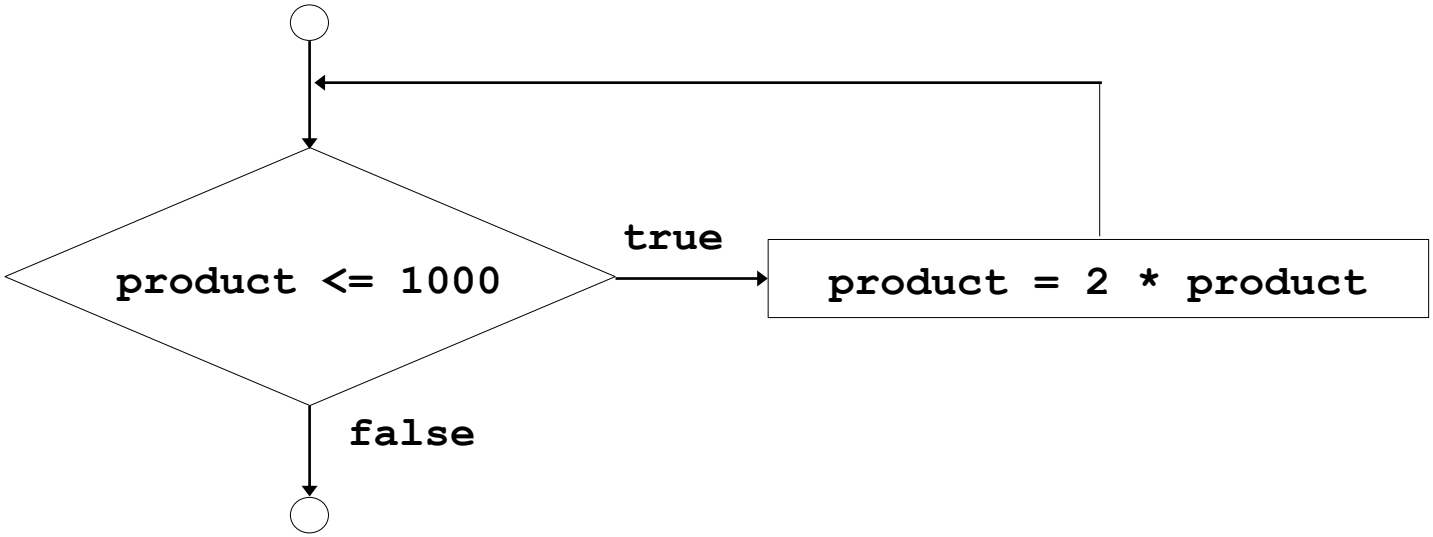
  - `while` loop repeated until condition becomes `false`

Product = 2*2*2*2* . . . . *2 = 2^n = 1024

3.7 The `while` Repetition Statement

- Example:

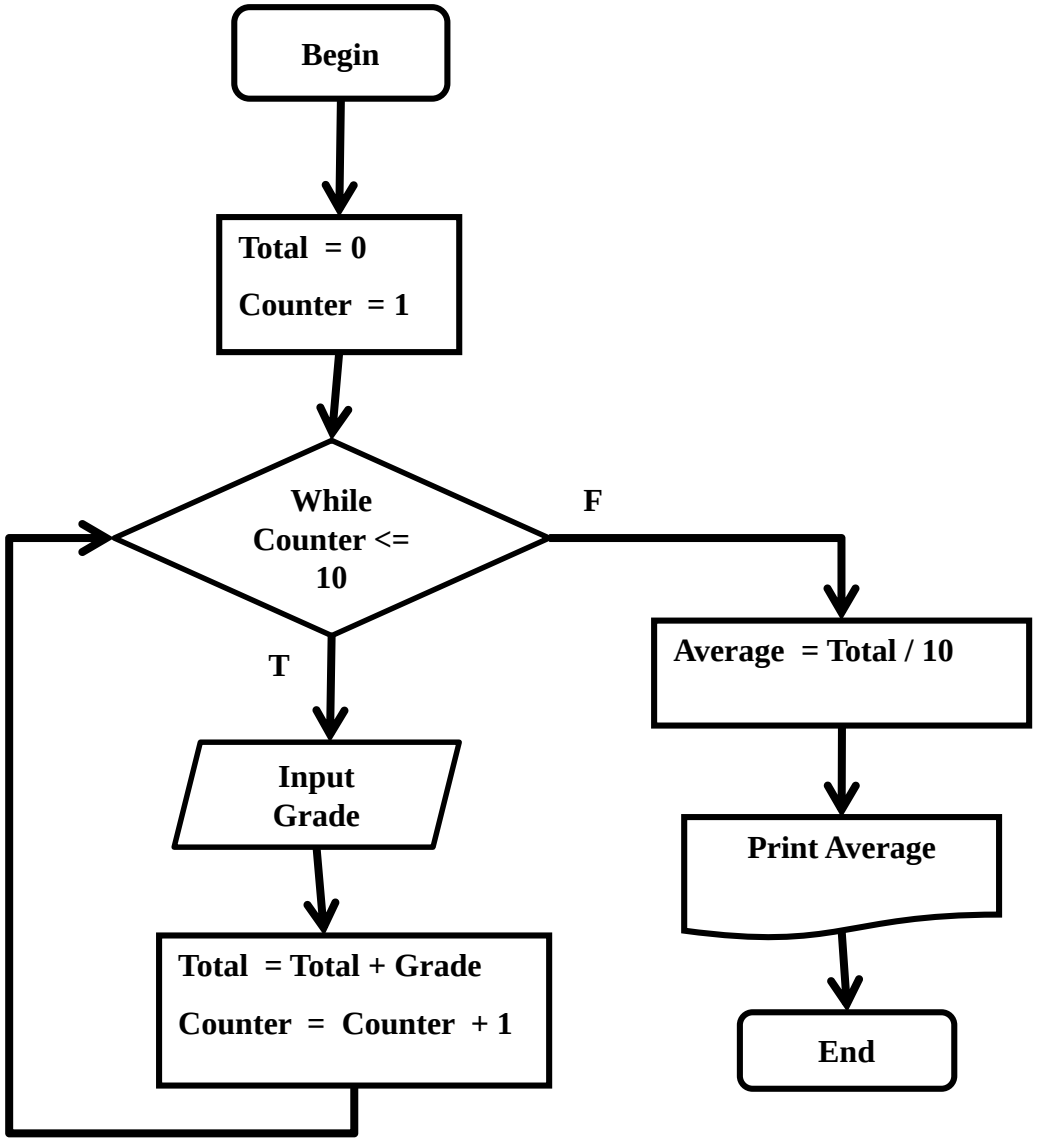
```
int product = 1;
while ( product <= 1000 )
       product = 2 * product;
```



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.

Flow Chart
(Counter-Controlled Repetition)



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

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

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

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

   // termination phase
   average = total / 10; // integer division, fraction is lost!
   printf( "Class average is %d\n", average ); // display result
}
```

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

- 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)**

3.9 Formulating Algorithms with 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:

    1. ***Initialize variables***
    2. ***Input, sum and count the quiz grades***
    3. ***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

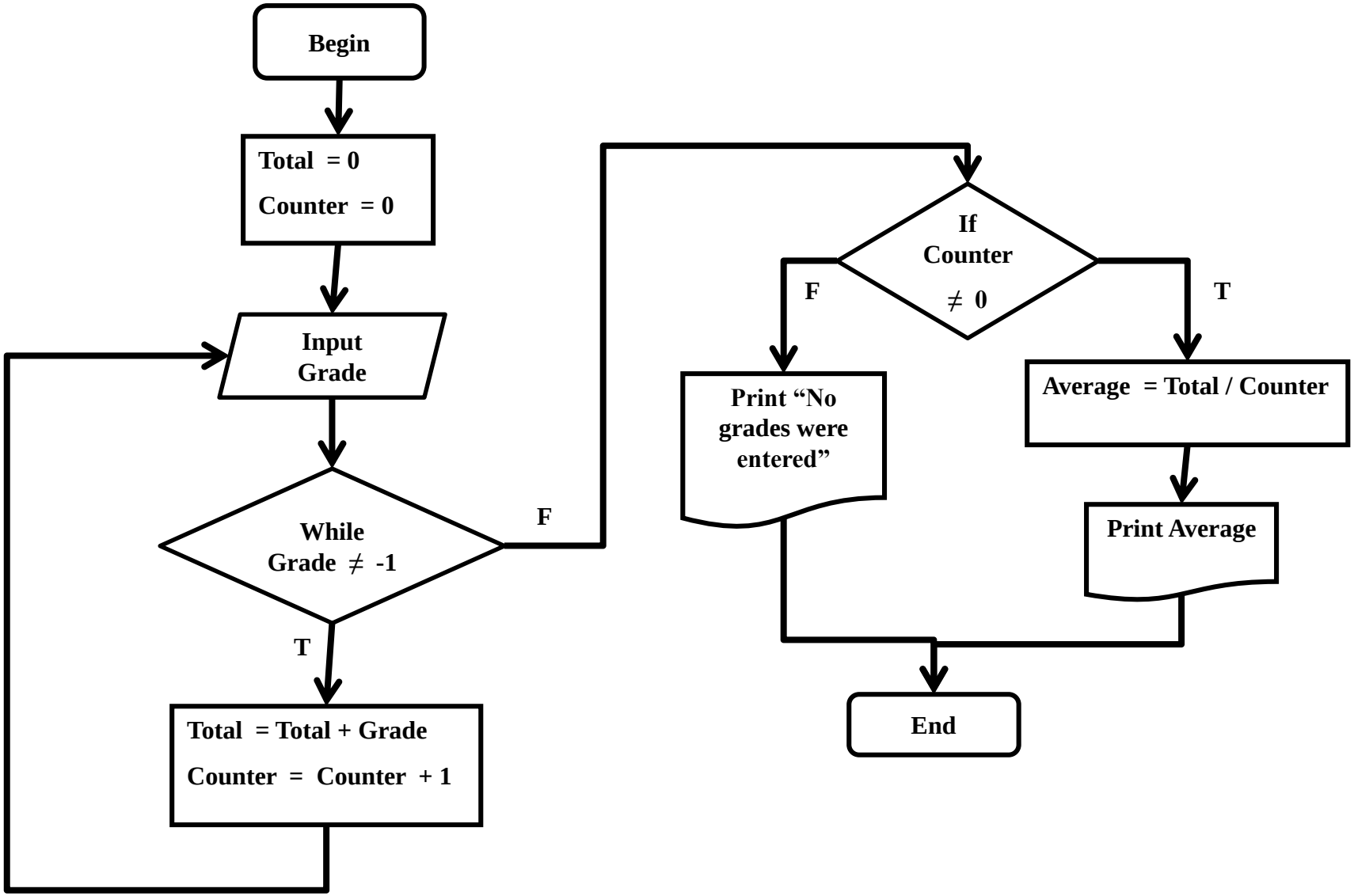
Final Pseudoce

1.1. Initialize total to zero
1.2. Initialize counter to zero

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

3.1. If the counter is not equal to zero
    3.1.1. Set the average to the total divided by the counter
    3.1.2. Print the average
3.2. else
    3.2.1. Print "No grades were entered"

Flow Chart
(Sentinel-Controlled Repetition)



Part 1 of 2

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

int main() {
   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

   // 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
   }
```

Part 2 of 2

```
   // 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
} // end main
```

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
```

Program Output

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

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)
h %= 9      (h = h % 9)
```

3.11 Assignment Operators

Assume:

int c = 3, d = 5, e = 4, f = 6, h = 12;

Assignment operator	Sample expression	Explanation	Assigns
+=	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
%=	h %= 9	h = h % 9	3 to h

Fig. 3.11 Arithmetic assignment operators.

3.12 Increment and Decrement Operators

- Increment operator (++)
  - Can be used instead of `a+=1`
- Decrement operator (`--`)
  - Can be used instead of `a-=1`

- Preincrement and Predecrement
  - Operator is used before the variable (`++a` or `--a`)
  - Variable is changed first, then the expression is evaluated

- Postincrement and Postdecrement
  - Operator is used after the variable (`a++` or `a--`)
  - Expression executes before the variable is changed

3.12 Increment and Decrement Operators

- If variable a equals 5, then

```
printf( "%d", ++a );
```

Prints 6

```
printf( "%d", a++ );
```

Prints 5

  - In either case, a now has the value of 6

- Standalone usage: When variable not in an expression
  - Preincrementing and postincrementing have the same effect

```
++a;
printf( “%d”, a );
```

  - Has the same effect as

```
a++;
printf( “%d”, a );
```

3.12 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.

Fig. 3.12 The increment and decrement operators

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

int main() {
   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;  // reassign 5 to c
   printf( "%d\n", c );   // print 5
   printf( "%d\n", ++c ); // preincrement then print 6
   printf( "%d\n", c );   // print 6
}
```

Program Output

```
5
5
6

5
6
6
```

3.12 Increment and Decrement Operators

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

Fig. 3.14 Precedence of the operators encountered so far.