ENG1008 C Programming

Topic 2

- **Structured Programming**

"Good programmers write code that humans can understand."
Martin Fowler

Objectives

- To use basic *problem-solving* techniques
- To develop *algorithms* based on *top-down program flow*
- To use `if` selection statement and the `if…else` selection statement for condition/selection
- To use `while` repetition statement for repeating the execution of a block of code
- To use *assignment*, *increment* and *decrement* operators
- To implement *counter-controlled* repetition and *sentinel-controlled* repetition

Algorithms

- **Solution** of computing problem
  - Involves running a set of operations in a specific sequence
  - The procedure to solve the problem
    - involves *operations* to be executed
    - the *order* in which these operations are being executed
    - this is the **Algorithm**
- This is derived from having a *thorough understanding* of the *problem* and coming up with an *execution plan*
- The *sequence* or *order of execution* is crucial and important

Algorithms

- Correctly specifying the order in which the operations should execute is important
- Consider the “rise-and-shine algorithm” followed by one student for getting out of bed and going to school:
  - Get out of bed,
  - take off pajamas,
  - take a shower,
  - get dressed,
  - eat breakfast, and
  - Take a bus to school.
- This gets the student to school well prepared for classes.

Algorithms

- Suppose the steps are performed in a slightly different order
  - Get out of bed,
  - take off pajamas,
  - get dressed,
  - take a shower,
  - eat breakfast,
  - carpool to work.
- In this case, our student shows up for school soaking wet
- Specifying the order in which statements should execute in a computer program is called **program control**

Pseudocode

- **Pseudocode**
  - Helps the programmer to *think through the solution* **before** starting to *write the program* in a programming language (e.g. C)
  - Helps in the *development* of the Algorithm
  - English like
    - not a programming language
    - English language constructs modeled to *look like* statements available in most programming languages
    - not to be executed/run on the computer system
  - Can be *easily converted* to the actual program
  - *No fixed syntax* for most operations is required

Pseudocode

- **Pseudocode**
  - Only contains **action statements** and the **program flow**
  - Steps presented in a *structured* manner (numbered, indented, and so on)
  - Emphasis is on *process*, not notation
  - Well-understood forms allow *logical reasoning* about algorithm behavior
  - *Definitions* (e.g. int i) are not executable statements
    - instructs the compiler to reserve memory space
    - does not cause any action

Pseudocode

Step	Operation
1	Get values for *gallons used, starting mileage, ending mileage*
2	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
3	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
4	Print the value of *average miles per gallon*
5	Stop

Step	Operation
1	Get values for *gallons used, starting mileage, ending mileage*
2	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
3	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
4	Print the value of *average miles per gallon*
5	If *average miles per gallon* is greater than 25.0 then
6	Print the message 'You are getting good gas mileage'
	Else
7	Print the message 'You are NOT getting good gas mileage'
8	Stop

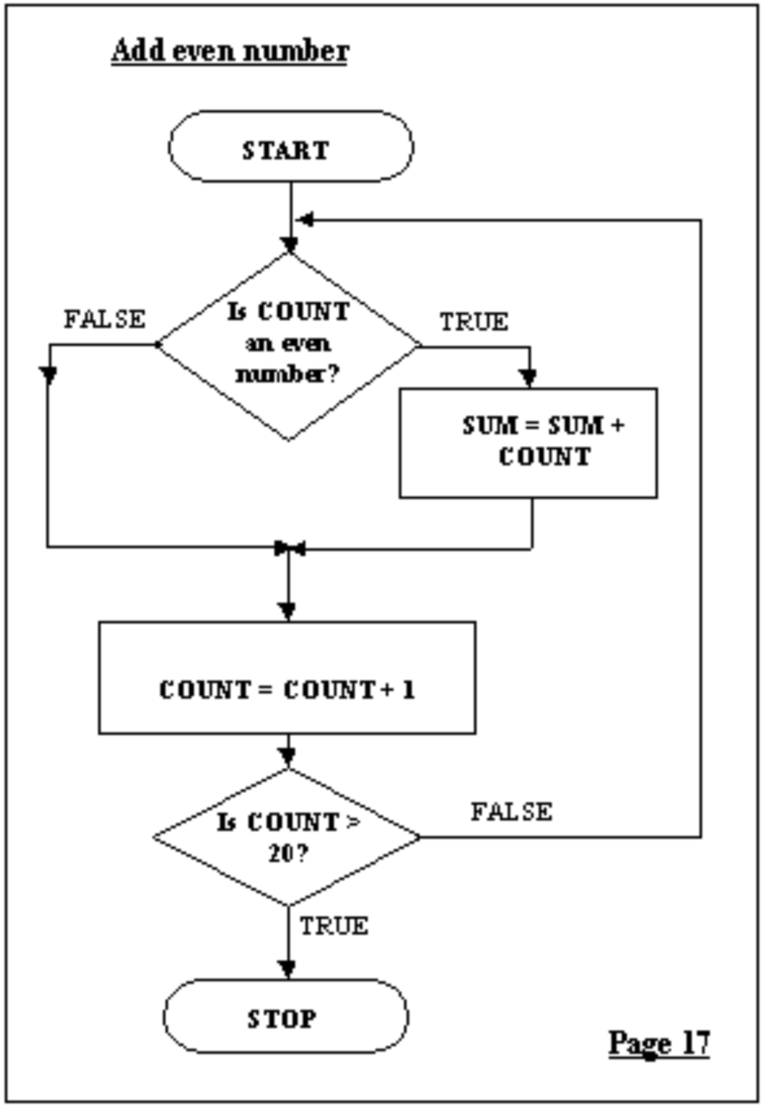
Pseudocode

Step	Operation
1	*response* = *Yes*
2	While (*response* = *Yes*) do Steps 3 through 11
3	Get values for *gallons used, starting mileage, ending mileage*
4	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
5	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
6	Print the value of *average miles per gallon*
7	If average miles per gallon > 25.0 then
8	Print the message 'You are getting good gas mileage'
	Else
9	Print the message 'You are NOT getting good gas mileage'
10	Print the message 'Do you want to do this again? Enter Yes or No'
11	Get a new value for *response* from the user
12	Stop

Flowcharts

- **Flowcharts**
  - *Graphical* representation of an algorithm or part of it
  - Drawn using *special symbols* (e.g. diamonds, rectangles, circles)
  - Symbols connected by *arrows* - *flowlines*
    - not a programming language
    - not to be executed/run on the computer system
  - Similar to *pseudocode*
  - Useful for program development or to represent the algorithm
  - *Illustrates clearly* the **program flow**

Flowcharts



Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Extracted from https://www.smartdraw.com/flowchart/flowchart-symbols.htm

Flowcharts

- **Flowcharts**
  - *Begin* is the first symbol and is represented by the *rounded rectangle symbol*
  - *Actions* are represented by *rectangle* symbols
  - *Decisions* are represented by *diamond* symbols
  - *Connector* symbols – *small circular symbols* are used to connect different portion of the algorithm

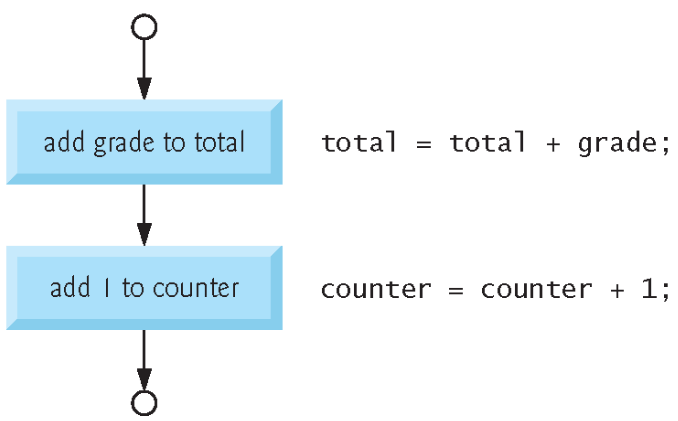


Fig. 3.1 | Flowcharting C's sequence structure.

Control Structures

- **Sequential execution**
  - Code in the program are *normally executed sequentially*
  - Unless redirected, the computer will execute the C statements one after another in the order they were written
  - There are specific C statements that are able to *transfer the control / change the program flow*
    - enables the programmer to specify the next statement to be executed instead of the next one in sequence

Control Structures

- **Selection statements**
  - The **if** selection statement (Section 3.5) will perform an action if a condition is true or skip that action if the condition is false (*single selection*)
  - The **if…else** selection statement (Section 3.6) will perform an action if a condition is true and perform another action if the condition is false (*double selection*)
  - The **switch** selection statement (Chapter 4) will perform different sets of actions depending on the value of an expression (*multiple selection*)

Control Structures

- **Repetition statements** (Chapter 4)
  - The **while** statement
  - The **do…while** statement
  - The **for** statement
- C has only seven **control** statements: *sequence*, three types of *selection* and three types of *repetition*
- The different types of control statements are used together in a C program to implement the required algorithm
- There is a *single entry* and *single exit* for these control structures – this help to build clear programs

If statement

- **Selection** statement are used to choose among alternative course of action
  - The flowchart of Fig. 3.2 illustrates the **single-selection if** statement for

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

  - The **diamond** symbol – the *decision* symbol, means that a decision must be made

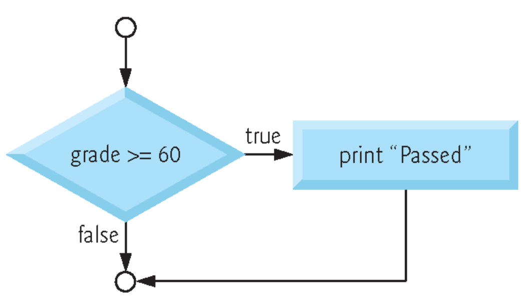


Fig. 3.2 | Flowcharting the single-selection `if` statement.

Pseudocode

if the grade obtained is more than or equal to 60
    print "Passed"

If statement

- The *decision symbol* contains an **expression**, such as a condition, that can be either **true** or **false**
  - Decisions can be based on conditions containing *relational or equality* operators
  - A decision can be based on *any* expression
  - If the expression evaluates to ***zero***, it is treated as ***false***; If it evaluates to ***nonzero***, it is treated as ***true***
- *Single entry/single exit* statement

If…else statement

- The **if…else** selection statement allows you to specify that different actions are to be performed when the *condition is true* and *when it is false*

- The flowchart of Fig. 3.3 illustrates the double-selection if…else

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

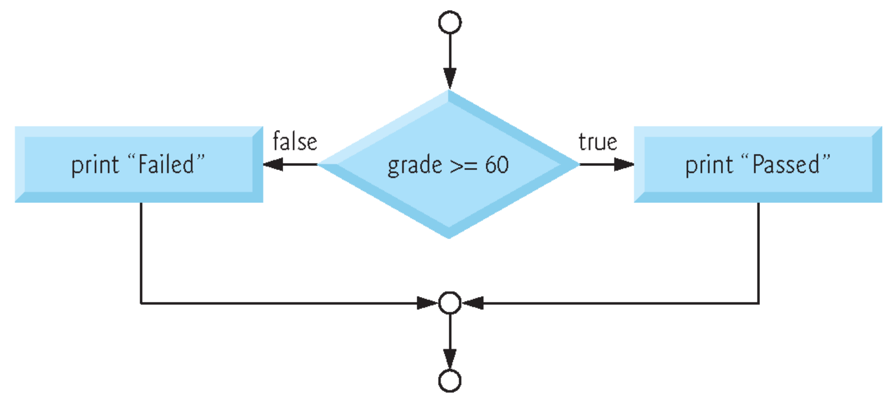


Fig. 3.3 | Flowcharting the double-selection `if…else` statement.

Pseudocode

if the grade obtained is more than or equal to 60
    print "Passed"
else
    print "Failed"

If…else statement

- **Nested if…else** statement for multiple cases
  - Placing if…else statement inside if…else statements

Pseudocode

```
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"
```

C Program

```c
if (grade >= 90) {
   printf("A")
} // end if
else {
   if (grade >= 80) {
      printf("B")
   } // end if
   else {
      if (grade >= 70) {
         printf("C")
      } // end if
      else {
         if (grade >= 60) {
            printf("D")
         } // end if
         else {
            printf("F")
         } // end else
      } // end else
   } // end else
} // end else
```

If…else statement

- Nested if…else statement for multiple cases
  - *re-writing*

```
if (grade >= 90)
   printf("A");
else if (grade >= 80)
   printf("B");
else if (grade >= 70)
   printf("C");
else if (grade >= 60)
   printf("D");
else
   printf("F");
```

`If…else` statement

- The if selection statement *expects only one statement* in its body—if you have only one statement in the **`if`**'s body, you *do not need* to enclose it in *braces*

- To include *several statements* in the body of an **`if`**, you must enclose the set of statements in *braces* **{** *and* **}**

```
if (grade >= 60)
    printf("Passed.");
else {
    printf("Failed.");
    printf("Try again.");
}
```

- ternary
  Conditional operator (?:)
  - It takes 3 operands – 1st operand is a condition, 2nd operand is the value if the condition is true and 3rd operand is for the value if the condition is false
    - true false
      `printf(grade >= 60 ? "Passed" : "Failed");`
  - The 2nd and 3rd operands can also be actions to be executed
    - true false
      `grade >= 60 ? printf("Passed") : printf("Failed");`

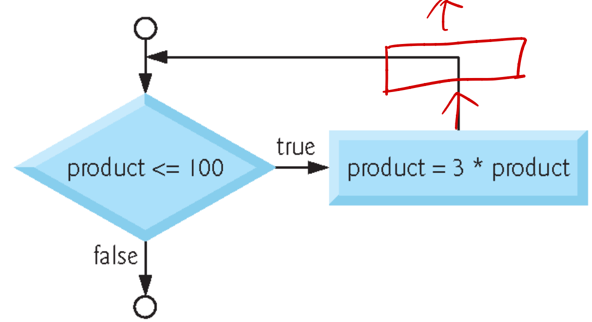
`while` statement

- A **repetition** statement (also called an iteration statement) allows you to specify that an *action is to be repeated* while *some condition remains true*
  - This action will be *performed repeatedly* while the condition remains *true*
  - The flowchart of Fig. 3.4 illustrates the *flow of control* in the **`while`** repetition statement

```
while (product <= 100) {
   product = 3 * product;
}
```

Pseudocode

while the value of product is less than or equal to 100
   multiply product by 3



Flowcharting the `while` repetition statement.

`while` statement

- The statement(s) contained in the **`while`** repetition statement constitute the *body of the while loop*
- The **`while`** statement body may be a *single statement* or *compound statements*
- The condition will eventually become *false*
- At this point, the repetition terminates, and the first pseudocode statement *after* the repetition structure is executed (i.e. program execution continues with the next statement after while)

```
int product = 3;
while (product <= 100) {
   product = 3 * product;
   printf("%d\n", product);
}
```

```
9
27
81
243

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

Loop (Flowcharts)

- **Loop** refers to repetition of block of instructions
  - Often, the real power of a computer comes from performing a calculation many times
  - Looping takes advantage of the power of computers
- The **while** statement:

  While ("a true/false condition") do step i to step j

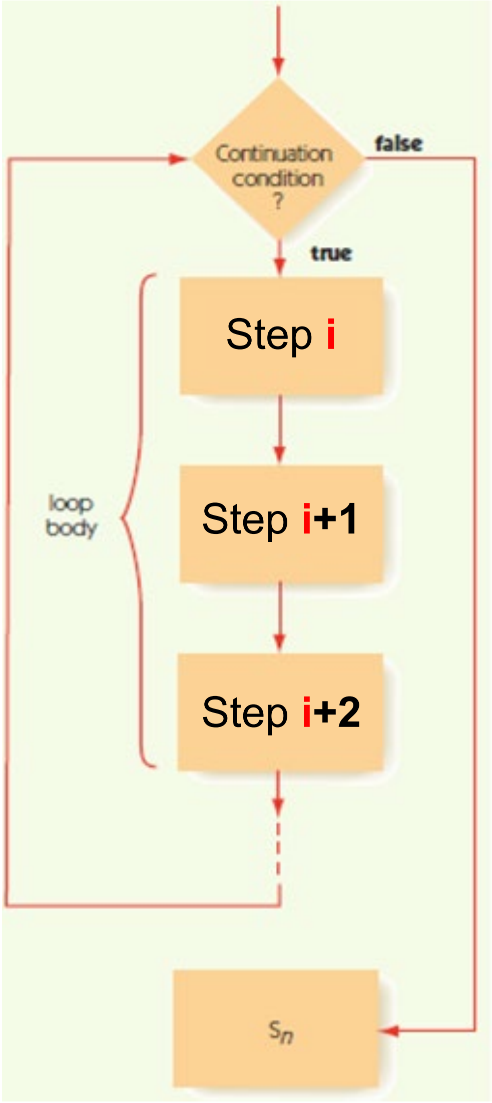
  Step i: operation

  Step i+1: operation

  .

  .

  Step j: operation
- If the continuation condition never becomes false, then we will forever be trapped in an **infinite loop**



Loop (Pseudocode)

- Extending the previous example to print whether the driver is getting good gas mileage or not, this time we print only while the user respond with a Yes:

Step	Operation
1	*response* = *Yes*
2	While (*response* = *Yes*) do Steps 3 through 11
3	Get values for *gallons used, starting mileage, ending mileage*
4	Set value of *distance driven* to (*ending mileage* – *starting mileage*)
5	Set value of *average miles per gallon* to (*distance driven* ÷ *gallons used*)
6	Print the value of *average miles per gallon*
7	If average miles per gallon > 25.0 then
8	Print the message 'You are getting good gas mileage'
	Else
9	Print the message 'You are NOT getting good gas mileage'
10	Print the message 'Do you want to do this again? Enter Yes or No'
11	Get a new value for *response* from the user
12	Stop

Third version of the average miles per gallon algorithm

❖ Formulating Algorithms

Formulating Algorithms **Case Study 1**: Counter-Controlled Repetition

- To illustrate how algorithms are developed, we solve several variations of a class-averaging problem.
- Consider the following problem statement:
  - *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.*
- The class average is equal to the sum of the grades divided by the number of students.
- The **algorithm** for solving this problem on a computer must input each of the grades, perform the averaging calculation, and print the result.

Formulating Algorithms Case Study 1: Counter-Controlled Repetition

- To use Pseudocode to *list the actions to execute* and specify *the order* in which these actions should execute
  - To use a counter-controlled repetition to input the grades one at a time 10
  - To use a variable counter to specify the number of times a set of statements should execute. In this case, repetition terminates when the counter exceeds 10

```
Set total to zero
Set grade counter to one        // initialize

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 3.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

Formulating Algorithms Case Study 1: Counter-Controlled Repetition

- In the algorithm,
  - A total is a variable used to add up the sum of a series of values.
  - A counter is a variable used to count (in this case to count the number of grades entered).
  - Counter is declared as an *unsigned int* (i.e. takes non-negative values 0 and higher) as it counts from 1 to 10
  - Variables *totals* or *sum* should normally be initialized to zero before being used in a program; otherwise the sum would include the *previous value stored in the total's memory location*.
  - *Counter* variables are normally initialized to zero or one, depending on their use.
  - An uninitialized variable contains a "garbage" value—the value last stored in the memory location reserved for that variable.
  - The averaging calculation in the program produced an integer result of 81 even though 817 divided by 10 = 81.7 (to be worked on later – floating point numbers).

Formulating Algorithms Case Study 1: Counter-Controlled Repetition

SINGAPORE INSTITUTE OF TECHNOLOGY

```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 )
{
   unsigned int counter; // number of grade to be entered next
   int grade; // grade value
   int total; // sum of grades entered 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( "%s", "Enter grade: " ); // prompt for input
      scanf( "%d", &grade ); // read grade from user
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter
   } // end while

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

   printf( "Class average is %d\n", average ); // display result
} // end function main
```

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

```
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
```

Fig. 3.6 | Class-average problem

Formulating Algorithms with Top-Down, Stepwise Refinement **Case Study 2**: Sentinel-Controlled Repetition

- Consider the following problem:
  - *Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*
- In the first class-average example, the number of grades (10) was known in advance.
- In this example, the program must process an **arbitrary number** of grades.
- How can the program determine when to **stop** the input of grades? How will it know when to calculate and print the class average?

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- One solution is to use a special value called a sentinel value (also called a signal value, a dummy value, or a flag value) to indicate "end of data entry".
  - Grades are keyed in until all legitimate grades have been entered.
  - The user then types the sentinel value to indicate "*the last grade has been entered.*"
  - Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions isn't known before the loop begins executing.
  - Clearly, the sentinel value must be chosen so that it *cannot be confused with an acceptable input value*.
  - Because grades on a quiz are normally nonnegative integers, **–1** is an *acceptable sentinel value* for this problem.
  - Thus, a run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1 (*–1 is the sentinel value, so it should not enter into the averaging calculation*).

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- We approach the class-average program with a technique called top-down, stepwise refinement, a technique that is essential to the development of well-structured programs.
  - Begin with the pseudocode representation of the top

    Determine the class average for the quiz

  - The top conveys the program's overall function and is in effect a complete representation of the program
  - However, it rarely conveys sufficient amount of detail (to code)
  - *Refinement process*: divide the top into smaller tasks and to list them in the order it needs to be performed

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- **1st refinement**

  Initialise variables

  Input, sum, and count the quiz grades

  Calculate and print the class average

- **2nd refinement** :
  Start to work on specific variables – we need:
  - running total of the numbers
  - a count of the number of grades processed
  - a variable to receive the value of each grade, and
  - a variable for the calculated average

Pseudocode statement ***Initialise variables*** may be refined as

Initialise total to zero

Initialise counter to zero

Note that the variables to receive each grade and the calculated average do not need to be initialised as their values will be overwritten

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- Pseudocode statement: ***Input, sum, and count the quiz grades***
  - Needs a **repetition** structure that repeatedly *reads in each grade*
  - *Sentinel-controlled repetition* is used as the number of grades is not known at the start
  - The user will enter the valid grades one at a time
  - *After the last valid grade* has been entered, the *sentinel value* is entered (i.e. the user will type the sentinel value)
  - The program will *test* for this *sentinel value after each grade value* is *input*, and will *terminate* the loop when the *sentinel* is typed

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)

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- Notice that in **pseudocode**, *braces are not used* for the statements that form the body of the while statement; simply **indent** all these statements in the while loop indicates they all belong to the while

- ***Calculate and print the class average***

  may be refined as

  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"

- Notice that we're being careful here to test for the possibility of *division by zero*—a fatal error that if undetected would cause the program to fail (often called "crashing").

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

```
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"
```

Fig. 3.7 | Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- Although only integer grades are entered,
  - The *averaging calculation* is likely to produce a number with a decimal point.
  - The *type int* cannot represent such a number.
  - The data type float is used to handle numbers with decimal points (called floating-point numbers).
  - A special operator called a cast operator is used to handle the *averaging calculation*.
  - The variable average is defined to be of *type float* (line 12) to capture the fractional result of our calculation.
  - However, the result of the calculation *total / counter* is an *integer* because *total and counter are both integer variables*.

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- continued,
  - Dividing two integers results in integer division in which any fractional part of the calculation is truncated (i.e., lost).
  - Because the calculation is performed *first*, the *fractional part* is lost *before* the result is assigned to average.
  - To produce a *floating-point calculation with integer values*, we must create temporary values that are floating-point numbers.
  - C provides the unary cast operator to accomplish this:

    average = **(float) total / counter;**
  - The **cast operator (float)** creates a *temporary floating-point copy* of its operand, total.
  - The value stored in total is still an integer.

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- continued,
  - Using a cast operator in this manner is called explicit conversion.
  - The calculation now consists of a *floating-point value (the temporary float version of total) divided by the unsigned int value stored in counter*.
  - C evaluates arithmetic expressions whereby operands are identical; to ensure this, the compiler performs an operation called implicit conversion on selected operands.
  - a *copy of counter* is made and *converted to* **float**, the calculation is performed and the result of the floating-point division is assigned to average.

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

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

// function main begins program execution
int main( void )
{
   unsigned 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( "%s", "Enter grade, -1 to end: " ); // prompt for input
   scanf( "%d", &grade ); // read grade from user

```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 1 of 3.)

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

```
   // 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( "%s", "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
      puts( "No grades were entered" );
   } // end else
} // end function main
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 2 of 3.)

Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

```
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
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 3 of 3.)

Floating Point Numbers

- Although **floating-point numbers** are not always “100% precise,” they have numerous applications.
  - When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643.
  - Another way floating-point numbers develop is through division.
  - The computer allocates only a *fixed* amount of space to hold such a value, so the stored floating-point value can be only an *approximation*.
  - Figure 3.8 uses the printf conversion specifier %.2f (line 41) to print the value of average.
  - The f specifies that a floating-point value will be printed.
  - The .2 is the precision with which the value will be displayed — with 2 digits to the right of the decimal point (*default is %.6f*)
  - The value in memory is not changed.

Assignment Operators

- Several assignment operators can be used to abbreviate assignment expressions.
  - c = c + 3; can be abbreviated as c += 3
  - The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator

- Any statement of the form
  - *variable = variable operator expression;*
  - where *operator is one of the binary operators +, -, *, / or % can be written in the form*
    - *variable operator= expression;*

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

Fig. 3.11 | Arithmetic assignment operators.

Increment and Decrement Operators

- C also provides the unary **increment** operator, **++**, and the unary **decrement** operator, **--**
  - If a variable c is to be *incremented by 1*, the increment operator **++** can be used rather than the expressions c = c + 1 or c += 1
  - If a variable c is to be *decremented by 1*, the decrement operator **--** can be used rather than the expressions c = c - 1 or c -= 1
- If increment or decrement operators are *placed before* a variable (i.e., prefixed), they're referred to as the preincrement or predecrement operators, respectively
- If increment or decrement operators are *placed after* a variable (i.e., postfixed), they're referred to as the postincrement or postdecrement operators, respectively

Increment and Decrement Operators

- **Preincrementing** (**predecrementing**) a variable causes the variable to be incremented (decremented) by 1, then the **new value** of the variable is used in the expression in which it appears
- **Postincrementing** (**postdecrementing**) the variable causes the **current value** of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by

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 | Increment and decrement operators

Preincrementing and postincrementing

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

```
5
5
6

5
6
6
```

Fig. 3.13 | Preincrementing and postincrementing. (Part 2 of 2.)