Chapter 3- Part 2
Structured Program Development in C

C How to Program

Type this program in Codeblocks

```
#include <stdio.h>

int main()
{
    int product = 3;

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

    return 0;
}
```

Self-Review exercise

- Write a program that calculates the sum of the integers from 1 to 10.
- Define variables sum and x of type int.
- Use the while statement to loop through the calculation and increment statements.
- The loop should terminate when the value of x becomes 11.

Introduction

- Before writing a program to solve a particular problem, we must have a thorough understanding of the problem and a carefully planned solution approach.

Formulating Algorithms
Counter-Controlled Repetition

- This technique uses

- A variable called counter to count the number of times a set of statements should execute.

- A variable called total to accumulate the sum of a series of values.

Counter-Controlled Repetition
Case Study

- 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.

Counter-Controlled Repetition (Cont.)
Case Study

- We use counter-controlled repetition to input the grades one at a time.

```
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>

// 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
```

Fig. 3.6 | Class-average problem with counter-controlled repetition. (Part 1 of 2.)

```c

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

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

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

Fig. 3.6 | Class-average problem with counter-controlled repetition. (Part 2 of 2.)

Counter-Controlled Repetition (Cont.) Case Study

- The algorithm mentions a total and a counter.
- A total is a variable used to accumulate 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.
- Variables used to store totals must be initialized to zero before being used in a program
- An uninitialized variable contains a "garbage" value—the value last stored in the memory location reserved for that variable.

Formulating Algorithms
Sentinel-Controlled Repetition

- Use a special value called a sentinel value to indicate “end of data entry.”

- The user types in data until …
- 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.

Sentinel-Controlled Repetition Case Study

- Consider the following problem:
  - *Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*
- Notice that 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?

Sentinel-Controlled Repetition (Cont.)
Case Study

- 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.

Sentinel-Controlled Repetition (Cont.) Case Study

Top-Down, Stepwise Refinement

- We approach the class-average program with a technique called top-down, stepwise refinement.

- We begin with a pseudocode representation of the top:

  - *Determine the class average for the quiz*

- The top is a single statement that conveys the program's overall function.
- To refine the top statement always ask yourself what do I need to make this statement true in terms of pseudocode

Sentinel-Controlled Repetition (Cont.)
Case Study

Top-Down, Stepwise Refinement

- We now begin the refinement process.
- We divide the top statement into a series of smaller tasks and list these in the order in which they need to be performed.
- This results in the following first refinement.
  - *Initialize variables*
    Input, sum, and count the quiz grades
    Calculate and print the class average

Sentinel-Controlled Repetition (Cont.) Case Study

- The first pseudocode statement
  - *Initialize variables*

- Can be refined as follows:
  - *Initialize total to zero*
    Initialize counter to zero

Sentinel-Controlled Repetition (Cont.)
Case Study

- Let's refine the second pseudocode statement
  - *Input, sum, and count the quiz grades*

- The user will enter legitimate grades in one at a time.
- After the last legitimate grade is typed, the user will type the sentinel value.
- The program will test for this value after each grade is input and will terminate the loop when the sentinel is entered.

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)

Sentinel-Controlled Repetition (Cont.) Case Study

- The third pseudocode statement
  - *Calculate and print the class average*

  may be refined as follows:
  - *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").

```
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.

Sentinel-Controlled Repetition (Cont.) Case Study

- 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 program introduces the data type `float` to handle numbers with decimal points (called floating-point numbers) and introduces a special operator called a cast operator to handle the averaging calculation.

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

// function main begins program execution
int main( void )
{
   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.)

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

        // get next grade from user
        printf( "%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.)

```
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.)

Converting Between Types Explicitly and Implicitly

- Averages do not always evaluate to integer values.
- Often, an average is a value such as 7.2 or –93.5 that contains a fractional part.
- These values are referred to as floating-point numbers and can be represented by the data type `float`.
- Line 12
  - `float average;`

Converting Between Types Explicitly and Implicitly(Cont.)

- However, the average in Line 38 is the result of the calculation `total / counter` which is an integer because `total` and `counter` are both integer variables.

- Dividing two integers results in integer division in which any fractional part of the calculation is truncated.

- Because the calculation is performed *first*, the fractional part is lost *before* the result is assigned to `average`.

Converting Between Types Explicitly and Implicitly(Cont.)

- 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 task.
- Line 38
  - `average = ( float ) total / counter;`
- includes the cast operator (`float`), which creates a temporary floating-point copy of its operand.

Converting Between Types Explicitly and Implicitly(Cont.)

- 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`.

Converting Between Types Explicitly and Implicitly(Cont.)

- C evaluates arithmetic expressions only in which the data types of the operands are *identical*.
- To ensure that the operands are of the *same* type, the compiler performs an operation called implicit conversion on selected operands.
- For example, in an expression containing the data types `unsigned int` and `float`, copies of `unsigned int` operands are made and converted to `float`.

Converting Between Types Explicitly and Implicitly(Cont.)

- Cast operators are available for *most* data types—they're formed by placing parentheses around a type name.
- Each cast operator is a unary operator, i.e., an operator that takes only one operand.
- Line 38
  - `average = ( float ) total / counter;`

Formating Floating-Point Numbers

- Line 41
  - `printf("Average is %.2f\n", 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.
- If the `%f` conversion specifier is used (without specifying the precision), the default precision of 6 is used—exactly as if the conversion specifier `%.6f` had been used.

Formating Floating-Point Numbers

- When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions.
- The value in memory is unaltered.
- For instance:
  - ```
    printf( "%.2f\n", 3.446 ); /* prints 3.45 */
    printf( "%.1f\n", 3.446 ); /* prints 3.4  */
    ```
    ```

Notes on Floating-Point Numbers

- Floating-point numbers are not always "100% precise"
- When we divide 10 by 3, the result is 3.3333333… with the sequence of 3s repeating infinitely.
- 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*.

Assignment Operators

- C provides several assignment operators for abbreviating assignment expressions.
- For example, the statement
  - `c = c + 3;`
- can be abbreviated with the addition assignment operator += as
  - `c += 3;`

Assignment Operators (Cont.)

- 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;*
- Figure 3.11 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

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
- the following expressions are the same
  - `c = c + 1;`
  - `c += 1;`
  - `c++;`

Increment and Decrement Operators (Cont.)

- If increment or decrement operators are placed before a variable, it 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.

- If increment or decrement operators are placed after a variable, it 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 1.

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

Common Programming Error 3.8

Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error, e.g., writing `++(x + 1)`.

```
// 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
```

Fig. 3.13 | Preincrementing and postincrementing. (Part 1 of 2.)

Secure C Programming

Arithmetic Overflow

- Figure 2.5 presented an addition program which calculated the sum of two int values (line 18) with the statement

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

- Even this simple statement has a potential problem—adding the integers could result in a value that's *too large* to store in an int variable.
- This is known as arithmetic overflow and can cause undefined behavior, possibly leaving a system open to attack.

Secure C Programming (Cont.)

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants `INT_MAX` and `INT_MIN`, respectively, which are defined in the header `<limits.h>`

- It’s considered a good practice to ensure that before you perform arithmetic calculations like the one in line 18 of Fig. 2.5, they will not overflow.

- The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline “INT32-C.”

Secure C Programming (Cont.)

Unsigned Integers

- In Fig. 3.6, line 8 declared as an `unsigned int` the variable counter because it's used to count only *non-negative values*.
- In general, counters that should store only non-negative values should be declared with `unsigned` before the integer type.
- Variables of `unsigned` types can represent values from 0 to approximately twice the positive range of the corresponding signed integer types.
- You can determine your platform's maximum unsigned `int` value with the constant `UINT_MAX` from `<limits.h>`

Maximum and minimum values depend on the computer you run the program

type	bits	from	to
int	16	−32,768	32,767
unsigned int	16	0	65,535

Self Review Exercise – Gas Mileage

- A driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful.
- Develop a program that will input all the miles driven and gallons used for each tankful.
- The program should calculate and display the miles per gallon obtain for each tankful.
- After processing all input information the program should calculate and print the combined miles per gallon obtained for all tankfuls.