02_03_C Structured Program Development

Objectives

In this chapter, you'll:

- Use basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` selection statement and the `if…else` selection statement to select actions.
- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Learn structured programming.
- Use increment, decrement and assignment operators.

Why an ampersand in the `scanf` ?

- This is because parameters in C functions/procedures are passed by value, and the only way to pass by reference is to pass the reference (address, pointer to) the variable.

- In C, if we want to pass parameters by reference the address of (&) operator should be used.

- The **`scanf()`** function reads formatted input and has to put this input into something like a variable.

- What we are doing, in fact, is passing the address of such variable to the **`scanf()`** function so it can put the input (from the keyboard for example) directly in this variable.

```c
// Fig. 3.6: fig03_06.c
// Class average program with counter-controlled iteration.
#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 iteration. (Part 1 of 2.)

```
    // 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 iteration. (Part 2 of 2.)

```c
// Fig. 3.8: fig03_08.c
// Class-average program with sentinel-controlled iteration.
#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 iteration. (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 iteration. (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 iteration. (Part 3 of 3.)

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

// function main begins program execution
int main( void )
{
   // initialize variables in definitions
   unsigned int passes = 0; // number of passes
   unsigned int failures = 0; // number of failures
   unsigned int student = 1; // student counter
   int result; // one exam result

   // process 10 students using counter-controlled loop
   while ( student <= 10 ) {

      // prompt user for input and obtain value from user
      printf( "%s", "Enter result ( 1=pass,2=fail ): " );
      scanf( "%d", &result );

```

Fig. 3.10 | Analysis of examination results. (Part 1 of 4.)

```c
        // if result 1, increment passes
        if ( result == 1 ) {
            passes = passes + 1;
        } // end if
        else { // otherwise, increment failures
            failures = failures + 1;
        } // end else

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

    // termination phase; display number of passes and failures
    printf( "Passed %u\n", passes );
    printf( "Failed %u\n", failures );

    // if more than eight students passed, print "Bonus to instructor!"
    if ( passes > 8 ) {
        puts( "Bonus to instructor!" );
    } // end if
} // end function main
```

Fig. 3.10 | Analysis of examination results. (Part 2 of 4.)

```
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
```

Fig. 3.10 | Analysis of examination results. (Part 3 of 4.)

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

Fig. 3.10 | Analysis of examination results. (Part 4 of 4.)

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.

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

```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
```

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

```
5
5
6

5
6
6
```

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

Increment and Decrement Operators

- The three assignment statements in Fig. 3.10

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with *assignment operators* as

```
passes += 1;
failures += 1;
student += 1;
```

with *preincrement operators* as

```
++passes;
++failures;
++student;
```

or with *postincrement operators* as

```
passes++;
failures++;
student++;
```

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

Fig. 3.14 | Precedence and associativity of the operators encountered so far in the text.

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

// function main begins program execution
int main( void )
{
   int integer1; // first number to be entered by user
   int integer2; // second number to be entered by user

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

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

   int sum; // variable in which sum will be stored
   sum = integer1 + integer2; // assign total to sum

   printf( "Sum is %d\n", sum ); // print sum
} // end function main
```

Fig. 2.5 | Addition program. (Part 1 of 2.)

Secure C Programming

Arithmetic Overflow

- Figure 2.5 presented an addition program which calculated the sum of two **`int`** values 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>`**.
- You can see your platform's values for these constants by opening the header **`<limits.h>`** in a text editor.
- It's considered a good practice to ensure that before you perform arithmetic calculations, 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 general, counters that should store only non-negative values should be declared with unsigned before the integer type.

- You can determine your platform's maximum unsigned **`int`** value with the constant **`UINT_MAX`** from **`<limits.h>.`**

Secure C Programming (Cont.)

- The class-average program in Fig. 3.6 could have declared as `unsigned int` the variables `grade`, `total` and `average`.
- Grades are normally values from 0 to 100, so the total and average should each be greater than or equal to 0.
- We declared those variables as ints because we can't control what the user actually enters—the user could enter negative values.
- Worse yet, the user could enter a value that's not even a number. (We'll show how to deal with such inputs later).

Secure C Programming (Cont.)

- Sometimes sentinel-controlled loops use invalid values to terminate a loop.
- For example, the class-average program of Fig. 3.8 terminates the loop when the user enters the sentinel -1 (an invalid grade), so it would be improper to declare variable grade as an unsigned int.
- As you'll see, the end-of-file (EOF) indicator—which is introduced in the next chapter and is often used to terminate sentinel-controlled loops—is also a negative number.

Secure C Programming (Cont.)

`scanf_s` and `printf_s`

- The C11 standard's Annex K introduces more secure versions of **`printf`** and **`scanf`** called **`printf_s`** and **`scanf_s`**. Annex K is designated as optional, so not every C vendor will implement it.

- Microsoft implemented its own versions of **`printf_s`** and **`scanf_s`** prior to the publication of the C11 standard and immediately began issuing warnings for every **`scanf`** call.

- The warnings say that **`scanf`** is deprecated—it should no longer be used—and that you should consider using **`scanf_s`** instead.