Problem Solving and Program Design - Chapter 5

Cory L. Strope
Chapter 5

5.1 Repetition in Programs
5.2 Counting Loops and the While Statement
5.3 Computing a Sum or a Product in a Loop
5.4 The for Statement
5.5 Conditional Loops
5.6 Loop Design
5.7 Nested Loops
5.8 Do While Statement and Flag-Controlled Loops
5.10 How to Debug and Test
5.11 Common Programming Errors
Repetition in Programs

Just as the ability to make decisions (if-else selection statements) is an important programming tool, so too is the ability to specify the repetition of a group of operations. When solving a general problem, it is sometimes helpful to write a solution to a specific case. Once this is done, ask yourself:

- Were there any steps that I repeated? If so, which ones?
- Do I know how many times I will have to repeat the steps?
- If not, how did I know how long to keep repeating the steps?
Counting Loops

The loop shown below in pseudocode is called a **counter-controlled loop** (or **counting loop**) because its repetition is managed by a loop control variable \( lcv \) whose value represents a count.

Set \( lcv \) to an initial value of 0
while the \( lcv < \) final_value {
    Block of program code;
    Increase \( lcv \) by 1;
}

Counter-controlled loop are used when we know how many loop repetitions to use before loop execution.
This example while loop computes and displays the gross pay for seven employees. The loop body is a compound statement (between { and }). The loop repetition condition controls the while loop.

```c
count_emp = 0;  // Set counter to 0;
while (count_emp < 7) { // If count_emp < 7, do stmts
    printf("Hours> ");
    scanf("%d",&hours);
    printf("Rate> ");
    scanf("%lf",&rate);
    pay = hours * rate;
    printf("Pay is \$%6.2f\n", pay);
    count_emp = count_emp + 1; /* Increment count_emp */
}
printf("\nAll employees processed\n");
```
While Statement

Syntax of the while Statement:

- Initialize the lc
  - Without initialization, the lc value is meaningless.
- Test the lc before the start of each loop repetition
- Update the lc during the iteration
  - Ensures that the program progresses to the final goal

```c
count = 0;
while(count < 1) {
    printf("Count = %d\n", count + 1);
}
```

If any of these are skipped it may produce an infinite loop.
General While Loops

It is better to generalize variables when possible. For example:

```c
int num_emp=0, count_emp=0;
printf("How many employees> ");
scanf("%d", &num_emp);
while(count_emp < num_emp) {
    . . .
    count_emp = count_emp + 1;
}
```

Using `num_emp` instead of the constant 7 allows our code to be more general.
While loop examples

\[ \sum_{i=1}^{100} i \]

is equivalent to \( 1 + 2 + 3 + \ldots + 100 \). Using the while loop:

```c
sum = 0;
lcv = 1;
while (lcv <= 100) {
    sum = sum + lcv;
    lcv = lcv + 1;
}
printf("Sum is %d", sum);
```
While loop examples

\[ \prod_{i=1}^{100} i \]

is equivalent to \(1 \times 2 \times \ldots \times 100\). The while loop can be used similarly as for addition:

```java
product = 1;
lcv = 1;
while (lcv <= 100) {
    product = product * lcv;
    lcv = lcv + 1;
}
```
Compound Assignment Operators

- Expressions such as:
  \[\text{variable} = \text{variable} \; \text{op} \; \text{expression};\]
  where \text{op} is a C operator such as \(+, -, *, /,\) etc. occur frequently.

- C has a set of shortcuts:
  Instead of writing \(x = x + 1\) we can write \(x += 1\).

- Similarly, \(-=, *=, /=,\) and \(%=\) are used in the same way.

```c
product = 1; lcv = 1;
while(lcv <= 100) {
    product *= lcv;
    lcv += 1;
}
```

Or, for a more complex example:

\[n = n \ast (x + 1); \equiv n\ast = x + 1;\]
for loops

- Program Style
- Increment and Decrement Operators
- Increment and Decrement Other Than 1
for loops

- A better way to construct a counting loop is to use the for statement.
  - while loops always run the risk of infinite loops.
- C provides for statements as another form for implementing loops.
- As before we need to initialize, test, and update the loop control variable.

*The for statement designates a specific place for the initialization, testing, and update components.*
**for loop Example**

\[ \sum_{i=0}^{100} i \]

sum = 0;
for (lcv = 0; lcv <= 100; lcv++) {
    sum = sum + lcv;
}

**Notation:** lcv++ is another notation, equivalent to lcv = lcv + 1 and lcv += 1.
For clarity, the book usually places each expression of the `for` heading on a separate line. If all three expressions are very short, however, they will be placed on one line. The body of the `for` loop is indented just as the `if` statement.
Increment and Decrement Operators

The counting loops that we have seen have all included assignment expressions of the form

- `counter = counter + 1`
- `counter++`
- `counter += 1`

This will add 1 to the variable `counter`. If we use a `-(--)` instead of a `+(++)`, it will subtract 1 to the variable `counter`. 
The increment (+=) and decrement -= operators will also work for values greater than 1.

- **Increment operations**: \( \text{sum} = \text{sum} + x \) or \( \text{sum} += x \), will take the value of \( \text{sum} \), add \( x \) to it, and then assign the new value to \( \text{sum} \).

- **Decrement operations**: \( \text{temp} = \text{temp} - x \) or \( \text{temp} -= x \), will take the value of \( \text{temp} \), subtract \( x \) from it and then assign the new value to \( \text{temp} \).
Conditional Loops

- The exact number of loop repetitions we need to run for a loop will not always be known before loop execution begins.

- Stick-Men: Do we need to exit the program if the user inputs a value other than 1 or 2 for the number of heads, or sets of arms? (or legs..?)

Initialization step? Test? Update action?
Example

Create a program that prompts the user for a value and multiplies it by temp and stores the result in temp until the user enters a 0.

In pseudocode:

assign temp the value of 1
prompt the user for a value
while value does not equal 0
    assign temp the value temp times value
    prompt the user for a value
output the value of temp
Example

Code:

temp = 1;
printf("Enter a value, 0 will stop the program> ");
scanf("%d", &value);
while(value != 0) {
    temp = temp * value;
    printf("Enter a value, 0 will stop the program> ");
    scanf("%d", &value);
}
printf("The product is %d", temp);
Loop Design

To this point, we have been analysing the actions a loop performs.

Now, we also want to design our own loops:

- Sentinel-Controlled Loops
- Using a `for` Statement to Implement a Sentinel Loop
- Endfile-Controlled Loops
- Infinite Loops on Faulty Data
Sentinel-Controlled Loops

- Often we don’t know how many data items the loop should process when it begins execution.
- **Sentinel-Controlled Loops** continue to read data until a unique data value is read, called the “sentinal value”.
- The sentinal value should be a value that could not normally occur as data.
- Reading the sentinal value signals the program to stop reading and processing new data.
- Examples: Product of a list of numbers, with ‘0’ stopping the loop.

1. Get a line of data
2. While the sentinal value not encountered
   3. Process the data line (compute)
4. Get another line of data
Using a for statement

Because the for statement combines the initialization, test, and update in once place, some programmers prefer to use it to implement sentinel-controlled loops.

```c
printf("Enter first score (or %d to quit)> ", sentinel);
for(scanf("%d", &score);
    score != sentinel;
    scanf("%d", &score)) {
    sum += score;
    printf("Enter next score (%d to quit)> ", sentinel);
}
```
Endfile-Controlled Loops

In Section 2.7, we discussed writing programs to run in batch mode using data files.

- A data file is always terminated by an endfile (EOF) character that can be detected by the `scanf` and `fscanf` functions.
- Therefore you can write a “batch” program that processes a list of data of any length without requiring a special sentinel value at the end of the data file.

1. Get first data value and save input status.
2. While input status !EOF
3. Process data value (compute)
4. Get next data value, save in input status.
```c
#include <stdio.h>

int main(void) {
    FILE *inp;
    int sum = 0, score, input_status;

    inp = fopen("scores.dat", "r");

    input_status = fscanf(inp, "%d", &score);
    while (input_status != EOF) {
        printf("%5d \n", score);
        sum += score;
        input_status = fscanf(inp, "%d", &score);
    }
    printf("\nSum of exam scores is %d\n", sum);
    fclose(inp);
    return 0;
}
```
Infinite Loop on Faulty Data

If we used a 7o instead of 70 for example 5.10:

1. Program would accepts all integers of the input. ‘7’ is a score of type int
   
   input buffer = 7 0

2. Value not the sentinel, so add 7 into sum.
   
   input buffer = 0

3. Program reads ‘o’ → defaults to value of 0, does not read ‘o’.
   
   input buffer = 0

4. Value not the sentinel, add 0 into sum.
   
   input buffer = 0

5. Program reads ‘o’ → defaults to value of 0, does not read ‘o’.
   
   input buffer = 0

...
Nested Loops

Like if statements, loops can also be nested.

- Nested loops consist of an outer loop with or more inner loops.
- Each time the outer loop is repeated, the inner loops are reentered.
- The inner loop control expressions are reevaluated, and all required iterations are performed.
Example

```c
lcv1 = 0;
sum = 0;
while (lcv1 < 100) {
    lcv2 = 0;
    while (lcv2 < 100) {
        sum = sum + 1;
        lcv2++;
    }
    lcv1++;
}
```
The do-while Statement and Flag-Controlled Loops

- do-while statement
- flag-controlled loops
• The for statement and the while statement evaluate conditions before the first execution of the loop body.

• In most cases, this pretest is desirable;
  • Prevents the loop from executing when there are no data items to process
  • Prevents execution when the initial value of the loop control variable is outside the expected range.

• Situations involving interactive input, when we know that a loop must execute at least one time, often use a do-while loop.
Do While Example

do {
    printf("Enter a letter from A through E> ");
    scanf("%c", &letter_choice);
} while (letter_choice < 'A' || letter_choice > 'E');
Flag-Controlled Loops

- Sometimes a loop repetition condition becomes so complex that placing the full expression in its usual spot is awkward.
- In many cases, the condition may be simplified by using a flag.
  - A **flag** is a type `int` variable used to represent whether or not a certain event has occurred.
  - A flag has one of two values: 1 (`true`) and 0 (`false`).
How to Debug and Test Programs

- Using Debugger Programs
- Debugging without a Debugger
- Off-by-One Loop Errors
- Testing
A debugger helps you to debug a C program.

 Allows you to execute your program one statement at a time.
  
  Through single-step execution, you can trace your programs execution and observe the effect of each C statement on variables you select.

 If your program is very long, you can separate your program into segments by setting breakpoints at selected statements.
  
  The debugger executes all statements up to the breakpoint
  
  You can examine the variables and values at this point to make sure your program is working.
Debugging without a Debugger

• Use several `printf` statements to output the values of your variables to make sure they have the correct value in them as your program executes.

• It is often helpful to print out the value of your loop control variable to make sure you are incrementing it and will not enter an infinite loop.
Off-by-One Loop Errors

Loop boundaries - the initial and final values of the loop control variable.

- A fairly common logic error in programs with loops is a loop that executes one more time or one less time than required.
  - If a sentinel-controlled loop performs an extra repetition, it may erroneously process the sentinel value along with the regular data.
  - while(val != SENTINEL) {
    scanf("%d",&val);
    variable op val; }

- If a loop performs a counting operation, make sure that the initial and final values of the loop control variable are correct and that the loop repetition condition is right.
  - The sum of 1...100, is not for(i = 1; i < 100; i++) sum += i; i <= 100 should be used.
After all errors have been corrected and the program appears to execute as expected, the program should be tested thoroughly to make sure it works.

For a simple program, make enough test runs to verify that the program works properly for representative samples of all possible data combinations.
Common Programming Errors

• if and while statement can be confused, since they have similar appearance,

• Remember to initialize loop control variable as to prevent infinite loops.

• Infinite loops are bad. Avoid infinite loops. Just avoid them.

• Remember to use { and } around the code of the loop statements.

• Be careful about the loop conditions, if we only want positive results then (result != 0) would not work since result might become negative without ever being 0.

• do-while loops always executes once and then tests the condition.

• With the compound assignment operators, the parentheses are assumed to be around any expression that is the second operand,
  • a *= b + c; equals a = a * (b + c); not a = a*b + c;.