Chapter 4
Loops

4.1 Chapter Overview

Many times when we write programs, we want a block of statements to execute more than once. Simply copying and pasting that block of code many times is not convenient, hard to maintain if you want to change something in that block of code, and may not be generic enough. This handout gives a summary about iterative execution (loops). There are three different types of loops (`for`, `while`, and `do-while`). Booleans are also discussed in this handout.

4.2 Background

This program reads two integers from the user and prints the sum.

```
#include <stdio.h>
int main() {
    int num1, num2, sum;

    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);

    return 0;
}
```

What if we want to allow the user to enter two integers 5 times, printing the sum each time?

```
#include <stdio.h>
int main() {
    int num1, num2, sum;

    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);

    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);

    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);

    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);

    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);

    return 0;
}
```

What if we want to allow the user to enter two integers 25 times, printing the sum each time? What if we want to process a thousand pairs? Or, what if the number of pairs is unknown until the user tells the program how many pairs to process, or the user provides a sentinel value (i.e. a special value, such as -1) to indicate when done with input? (Some examples are shown in section 4.8 of these notes).

Loops are a control structure that controls the flow of execution based on evaluating some condition. If the condition is true, the code inside the loop will execute. After each iteration of the loop, the condition is checked again to see if it is still true. If the condition is false, the loop is exited and execution of the program continues with the code immediately after the loop.

In C, an expression that has the value of 0 is considered to be false. An expression that has a value of anything other than 0 is considered to be true.

A loop's condition expression can be ANY **Boolean** expression. A Boolean expression has a truth value of either true or false. They can be a relational expression or a logical expression. Examples:

```
while (count < 10)
{
      // Do something
      // update 'count'
}


while (count < 3 && done != 0)
{
       // Do something
       // update 'count' and 'done'
}
```

Relational Operators

Operator	**Meaning**
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Be careful not to confuse == with =. The == is a relational operator that compares two items to see if they are equal to each other. The = is an operator that assigns a value to a variable. So a == 2 tests to see if the value of a is equal to 2. Whereas, a = 2 assigns the value of 2 to the variable a.

Relational operators have lower precedence than all other arithmetic operators. So, a < b + c is the same as a < (b + c).

Logical Operators

Operator	**Meaning**
&&	Logical AND
||	Logical OR
!	Logical NOT

! Logical NOT - unary operator; reverses the truth/falsity of its condition

Boolean Values

P	Q	P || Q	P && Q	!P
true	true	true	true	false
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

4.3 For Loop

```
1 for (init_expression; loop_condition; loop_expression) {
2     statements;
3 }
4 ...
```

The above listing shows the syntax of a `for` loop. The `init_expression` is executed only once at the beginning before the loop execution starts. This is where the **index** variable is initialized. If this variable has already been initialized before the for loop, then you can just put a semi-colon in this spot for the `init_expression`. If the variable being initialized in the `init_expression` is also declared here, it is a **local** variable and cannot be used outside of the `for` loop because it is only known within the loop.

The `loop_condition` is then evaluated. If it is true, the block of statements `statements` is executed, followed by the `loop_expression`. Then the `loop_condition` is evaluated again. The block of statements followed by the `loop_expression` keep executing as long as the `loop_condition` is true. Once the `loop_condition` is false, the loop body does not execute and execution resumes from Line 4. The body of a `for` loop may not execute if from the beginning the result of the `loop_condition` is false.

Nesting loops is sometimes required to accomplish a given goal. A nested `for` loop is a `for` loop within another `for` loop. The proper use of indentation becomes even more important with more sophisticated program constructs, such as nested loops.

4.4 While Loop

```
1 init_expression;
2 while (loop_condition) {
3     statements;
4     loop_expression;
5 }
6 ...
```

The above listing shows the syntax of a `while` loop. This loop executes a block of statements represented by `statements` in Line 3 as long as the `loop_condition` is true. Line 1 indicates that the variables used in the loop should be initialized somewhere in the program before the `while` loop. In Line 2, the `loop_condition` is checked, if the result is true, the block of statements between the curly braces is executed. If the result is false, this block is not executed and the statements following the loop (Line 6) are executed. One of the problems you can run into when writing loops is **infinite loops**. In this case, the loop will run infinitely as the `loop_condition` will always return true. To avoid this problem, the body of the loop needs to change the variables involved in the `loop_condition` so it will eventually be evaluated to false and the loop execution terminates. Changing the variables involved in the `loop_condition` is indicated by `loop_expression` in Line 4. Because the execution of the loop body depends on the result of the `loop_condition`, that body may not be executed at all. For example, if from the beginning the `loop_condition` is false, the loop body will not be executed and execution will resume at Line 6.

You can change a `for` loop to a `while` loop and vice versa by following the syntax structure provided in the respective listings. `Init_expression`, `loop_condition` , and `loop_expression` need not be changed in the process.

Determining when to use a `while` loop versus when to use a `for` loop is simply a developer's choice. In general, when executing a block a code for a known number of times, a `for` loop is used. When the number of iterations is not known, a `while` loop is normally used (e.g., reading input from the user until he/she enters a valid input).

4.5 Do-while Loop

A `do-while` loop is another way to repeat the execution of a block of code. The only difference between `do-while` and the other loops (`for` and `while`) is that the body of a `do-while` loop is executed at least once, while the body of a `for` or `while` loop may never execute if the `loop_condition` is false from the beginning.

```
1 do {
2      statements;
3 } while (loop_condition);
4 ...
```

The above listing shows how to write a `do-while` loop. From the listing, you can see that the first thing that executes is `statements`. (This guarantees that the loop body will execute at least once.) Then the `loop_condition` is evaluated. If it is true, the loop body is executed again. If it is false then the statements following the loop (Line 4) are executed. When writing a `do-while` loop, do not forget to modify the variables involved in the `loop_condition` ; otherwise, you will run into an infinite loop.

4.6 Why indent, you ask?

```
/*  Program 4.5, page 53
program to calculate the triangular number
of whatever number the user inputs, giving the user 5 tries;
this demonstrates nested for loops  */
#include <stdio.h>
int main (void) {
int n, number, triangularNumber, counter;
for (counter = 1; counter <= 5; ++counter)  {
printf (“What triangular number do you want? “);
scanf (“%i”, &number);
triangularNumber = 0;
for (n = 1; n <= number; ++n)
triangularNumber += n;
printf (“Triangular number %i is %i \n\n”, number, triangularNumber);}
return 0; }
```

4.7 Redirection

You can use redirection to **send output to a file** instead of to the screen (stdout). For example:

```
      ./a.out > results.txt
```

The file results.txt will either be created or written over. If you want to append to an existing file, use two greater than signs:

```
      ./a.out >> results.txt
```

Any printf statements, or fprintf statements that specify stdout as the first argument, will be written to the file specified by the redirection. (If you have any fprintf statements that specify stderr as the first argument - - those will still be printed to the screen.)

You can also use redirection to **specify that input come from a file** instead of the keyboard (stdin).

```
      ./a.out < nums.txt
```

Whenever your program executes a scanf, or an fscanf from stdin, the input will come from the file specified by the redirection.

You can combine the two on the same command line:

```
      ./a.out < nums.txt > results.txt
```

When you design a program to be used with redirection, you should leave out the prompts since the data is coming from a file and not the user (you do not need to prompt the user for input or provide instructions to the user).

4.8 Looping Examples

These examples show a few different ways of handling an arbitrary number of pairs of numbers.

1. A while loop when **asking user how many pairs** of numbers will be entered. How could you rewrite this using a for loop instead?

```
int howMany, num1, num2, sum;

printf("How many pairs of numbers will you be entering?");
scanf("%d", &howMany);

while(howMany > 0) {
    fscanf(stdin, "%d %d", &num1, &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);
    howMany--;
}
```

2. A **sentinel controlled loop**. This will not be a viable approach if any of the numbers in the pairs may be the same value as the sentinel.

```
int num1, num2, sum;

printf("Enter positive numbers for your pairs, one at a time. \
        Enter a single 0 when finished.");
scanf("%d", &num1);

while(num1 != 0) {
    fscanf(stdin, "%d", &num2);
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);
    scanf("%d", &num1);
}
```

3. End-of-input controlled loop. This uses the return value from scanf to **check the number of items entered by the user**. If the user enters one number and hits the return key, and then hits the return key again, the value returned by the scanf will be 1, thereby exiting the while loop.

```
int num1, num2, sum;
printf("Enter positive numbers for your pairs, one at a time. |
        Press the return key an extra time when finished.");

while(scanf("%d %d", &num1, &num2) == 2) {
    sum = num1+ num2;
    fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);
}
```

4. End-of-input controlled loop. This uses the return value from scanf to **check the number of items read from the file**. This example **assumes the use of redirection** when running the program, so the input values are contained in a file. For example:

```
./a.out < input.txt
```

If the end of the file is reached, a special flag called **EOF** is returned. This example also shows an if statement, which is the next chapter in the book and will be discussed in more detail.

```
int num1, num2, sum;

while(scanf("%d", &num1) != EOF) {
    if (scanf("%d", &num2) != EOF {
        sum = num1+ num2;
        fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);
    }
}
```

5. End-of-file controlled loop. This example **detects end-of-file with the feof function**. This feof function returns a true or false, which can then be tested in your program. To use for standard input, you would specify `stdin` as the argument to the `feof` function. This also **assumes the use of redirection** when running the program, such as:

```
./a.out < input.txt
```

```
int num1, num2, sum;

scanf("%d", &num1);
while(!feof(stdin)) {
    scanf("%d", &num2);
    if (!feof(stdin)){
        sum = num1+ num2;
        fprintf(stdout, "%d  + %d = %d\n",  num1, num2,  sum);
        scanf("%d", &num1);
    }
}
```

6. **Counting example.** Suppose we want to count the number of passing grades that are greater than or equal to 70. This example assumes the grades are contained in a file and that redirection will be used when run, such as:

```
./a.out < grades.txt
```

```
int passCount = 0, grade;

scanf("%d", &grade);
while(!feof(stdin)) {
    if (grade >= 70)
        passCount++;
    scanf("%d", &grade);
}

printf("The number of passing grades 70 or above is: %d\n", passCount);
```

7. **Accumulating example**. Sometimes we want to maintain a sum or total of some values in a looping program. In the example below, this sum is used to calculate the average of all the grades. This example assumes the grades are contained in a file and that redirection will be used when run, such as:

```
       ./a.out < grades.txt
```

```
int sum = 0, sumCount = 0, grade, average;

scanf("%d", &grade);
while(!feof(stdin)) {
    sum += grade;     // running sum
                      // same as:   sum = sum + grade;
    sumCount++;       // the total number of grades
    scanf("%d", &grade);
}

average = sum / sumCount;
printf("The average grade is: %d\n", average);
```

8. **Searching example**. Continuing with our grades example, suppose we wanted to search the set of grades for a specific grade. This also uses redirection to get the target grade as well as the rest of the grades from a file, with the target grade listed first.

```
       ./a.out < grades.txt
```

```
int grade, target, found = 0;

scanf("%d", &target);

scanf("%d", &grade);
while(!feof(stdin)) {
    if (grade == target)
        found = 1;
    scanf("%d", &grade);
}

if (found)
    printf("Target grade %d was found.\n", target);
else
    printf("Target grade %d was not found.\n", target);
```

9. **Searching improvement**. This is the above example with the enhancement that the loop will end once the target grade has been found. You can add a loop counter with a print statement after the loop to print the number of iterations that occurred to this program and the above program to compare the number of iterations from each program.

```
int grade, target, found = 0;

scanf(“%d”, &target);

scanf(“%d”, &grade);
while(!feof(stdin) && !found) {
    if (grade == target)
        found = 1;
    scanf(“%d”, &grade);
}

if (found)
    printf(“Target grade %d was found.\n”, target);
else
    printf(“Target grade %d was not found.\n”, target);
```

10. **Finding extremes**. Still using the set of grades, let’s say we wanted to know the highest grade and the lowest grade.

```
int grade, maxGrade, minGrade;

scanf(“%d”, &grade);
maxGrade = grade;
minGrade = grade;

while(!feof(stdin)) {
    if (grade > maxGrade)
        maxGrade = grade;
    if (grade < minGrade)
        minGrade = grade;
    scanf(“%d”, &grade);
}

printf(“Highest grade is %d, lowest is %d\n”, maxGrade, minGrade);
```