CS102: Loops

A **loop** is a sequence of statements that will be executed repeatedly zero or more times. A loop can be executed a set number of times, or as long as some condition is met. Sometimes you might want to think of a loop as being executed until some condition is met. Of course, looping until one condition is met is the same as looping as long as the opposite of the condition is met. For instance, if you loop until "x" equals 5, that is the same as looping as long as "x" does not equal 5. This should become clearer when we look at examples. Each single repetition of the loop is known as an **iteration** of the loop.

There are three loop statements in C. We will look at them one at a time. The first is the "while" statement. The general format is:

```
while (expression)
  statement;
```

The statement will be executed as long as the expression remains true, or until a special command is encountered to end the loop. Of course, the inner statement can be a compound statement, and it generally will be.

Remember previously we looked at two ways to write a program that prints the phrase "Hello World!" to the screen three times on three separate lines. Here is one way to do it with a "while" statement:

```
#include <stdio.h>

int main(void)
{
  int x = 3;

  while (x > 0)
  {
    printf("Hello World!\n");
    x = x - 1;
  }

  return 0;
}
```

This "while" loop will undergo three iterations. During each, the phrase "Hello World!" will be printed on a separate line. Why does it execute three times? The variable "x" is initialized above the loop with the value of 3. The loop will repeat as long as the expression "x > 0" is true. At the end of each loop iteration, "x" is decreased by 1. After three iterations, "x" will have the value of 0, and the expression will no longer be true, so the loop will end.

Note that there is no semicolon after the right parenthesis ending the expression that "while" is checking. If there were, it would mean that the program would repeat the null statement until the condition was not true. If the condition starts off true, it will stay true, and we will fall in to an infinite loop. Infinite loops will be discussed more shortly.

Remember we have learned about the decrement operators. Two other ways that this code could be written are:

```
#include <stdio.h>

int main(void)
{
  int x = 3;

  while (x-- > 0)
    printf("Hello World!\n");

  return 0;
}
```

Here, since the decrement operator is placed after the variable, the old value of the variable is used to compare against 0. The first time through, this value is 3, then 2, then 1. The fourth time, it is 0 so we exit the loop.

And:

```
#include <stdio.h>

int main(void)
{
  int x = 3;

  while (--x >= 0)
    printf("Hello World!\n");

  return 0;
}
```

Now, the decrement operator is placed before the variable, so the value of "x" is decreased and then its value is used. The first time through, this value is 2, then 1, then 0. The fourth time, it is –1 so we exit the loop.

In all of these cases, we know ahead of time that the loop will be executed 3 times (i.e., it will go through 3 iterations).

Here is an example of code in which we don’t know in advance how many iterations there will be:

```
#include <stdio.h>

int main(void)
{
  int x, y;

  printf("Enter two numbers: ");
  scanf("%d %d", &x, &y);
  while (y != 0)
  {
    printf("%d / %d = %d\n", x, y, x/y);

    printf("Enter two numbers: ");
    scanf("%d %d", &x, &y);
  }

  return 0;
}
```

This code repeatedly asks the user to enter two integers. As long as the second number is not zero, the program prints the result of dividing the first number by the second (using integer division, so the fractional part will be dropped). If the second number is 0, the program ends. (If we try to divide something by 0, this causes a run time error; i.e., the program crashes!)

There are a couple of things to note here. First, this is the first time we've used "scanf" to have the user enter more than one number at a time. This is similar to "printf" in that each instance of "%d" indicates that an integer variable can be entered. When "scanf" is reading numbers, whitespace is ignored, so the user can separate his or her numbers with a space, many spaces, a tab, a newline, or any other whitespace. If, however, the user enters text, the result could be unpredictable.

Second, note we had to repeat two of the lines in this program twice. Why? Obviously, we need to have these lines at least once in the loop, so they will be repeated during each iteration. However, we need them once before the loop so that we can check the value the first time. What if we initialize the value of "y" to be non-zero? Can we then skip the instance of the two lines before the loop? Not as we have things here. The problem is, if we then have the two lines at the start of the loop (before the actual division), the division will take place after the user enters a 0 as the second value, which will cause an error. If the two lines are at the end of the loop (after the division), then the first instance of the division will take place on bogus values, not entered by the user.

It is possible, however, to rewrite the code using the lines only once as follows:

```
#include <stdio.h>
```

```
int main(void)
{
  int x, y;

  while (1)
  {
    printf("Enter two numbers: ");
    scanf("%d %d", &x, &y);
    if (y == 0)
      break;

    printf("%d / %d = %d\n", x, y, x/y);
  }

  return 0;
}
```

When you use "while (1)" (or any other non-zero constant), the loop should continue until some special statement inside the loop stops it. Remember, when a non-zero constant is used as a Boolean expression, it is interpreted as true. We've already seen the use of "break" inside of a "switch" statement. Now, we see it used inside of a "while" statement. When the "break" statement is reached, the computer jumps to the first statement after the "while" loop, regardless of whether or not the condition (i.e., expression) being checked by the loop is true.

Now, the program enters the loop no matter what and asks the user to enter the two numbers at the start of each iteration of the loop. When the user enters 0 as the second number, the loop is exited, and the program ends.

Now consider the following silly program:

```
#include <stdio.h>

int main(void)
{
  while (1)
  {
    printf("Hello World!\n");
  }

  return 0;
}
```

What do you think this program does? It prints "Hello Wold!" forever! This is called an **infinite loop**. Here, we created one on purpose, but normally, they are created by accident. (The compound statement within the loop is optional. That is, if each iteration consists of a single statement, you do not have to make it a compound statement.) What do you do when you are caught in an infinite loop? You press Ctrl-C on your keyboard. Pressing Ctrl-C will halt the execution of your C program! What happens if you use "while(0)" in your program? The loop is skipped no matter what. It is useless, but valid.

Here are five ways to exit a loop:

1) The condition the loop depends on is not met at the time of the check.
2) A "break" statement is encountered.
3) A statement that ends the current function, such as "return", is encountered.
4) The program crashes.
5) The user presses Ctrl-C.

Here is another example of a program that uses "while":

```
#include <stdio.h>

int main(void)
{
  int x, digit, sum_digits;

  printf("Enter a positive integer: ");
  scanf("%d", &x);

  sum_digits = 0;
  while (x > 0)
  {
    digit = x % 10;
    sum_digits = sum_digits + digit;
    x = x / 10;
  }

  printf("The sum of the digits is %d!\n", sum_digits);

  return 0;
}
```

This program asks the user to enter a number, and if it is positive, it will sum the digits. This sum is stored in the variable "sum_digits", which is initialized to zero. The user enters a number, and the value is stored in "x". As long as "x" is greater than zero, we mod it by 10, which gets the right-most digit of the number, and we add this digit to "sum_digits". Then we divide "x" by 10. Remember, when we apply integer division, the fractional part is cut off, so in effect, we are removing the right-most digit of "x" (which we have already added to the sum).

What if we wanted to include the original number in the print statement at the end? For instance, "The sum of the digits of 107 is 8!" or something like that. Right now, we can't do it, because "x" will equal zero by the time we get to the "printf" statement. If we want to be able to print this message, we will have to make a copy of the value, and either use the original in the loop (in which case we print the copy) or use the copy in the loop (in which case we print the original).

The next loop statement we are going to look at is the "do…while" statement. It is similar to the "while" statement, but the condition is checked at the end. The general format is:

```
do
  statement
while (condition);
```

For example, here is the program to print "Hello World!" three times using a "do…while" statement:

```
#include <stdio.h>

int main(void)
{
  int x = 3;

  do
  {
    printf("Hello World!\n");
    x = x - 1;
  } while (x > 0);

  return 0;
}
```

Notice now that there is a semicolon after the right parenthesis ending the expression while is checking. If you forget it, you will get a compiler error when you try to compile the program.

The first time the condition is checked, "x" will be 2, and the loop has already gone through one iteration. After three iterations, "x" will be 0, and the loop will end.

It is customary to place the "while" part of the "do…while" statement after the right curly brace ending the loop statements, but of course, this is not necessary, and you can place it on the next line if you prefer.

In practice, the "do…while" statement is not used commonly. Because the check is at the end of the loop, it can only be used when you know the loop will be executed at least one time. One case where it might be useful to use a "do…while" loop is when you want the user to enter a specific type of data, and you want to keep prompting the user until he or she gets it right. For example, consider the following loop (assume "x" is an integer variable):

```
do
```

```
{
  printf("Enter a number from 1 to 100: ");
  scanf("%d", &x);
} while ((x < 1) || (x > 100));
```

If the user does not enter a number in the correct range, the "while" condition will be met, and the loop will undergo another iteration, prompting the user again. Only after the user enters a number from 1 to 100 will the expression be false and the loop end. (If the user enters text, however, this can cause unpredictable behavior.)

As with the "while" statement, if a "break" statement is encountered within a "do…while" statement, the program jumps to the first statement after the loop.

We're now going to move on to the next loop statement, the "for" statement. The general format of the "for" statement is:

```
for (initialization; condition; update)
  statement;
```

The initialization is usually an assignment of a variable to some starting value, and the update is often an assignment that changes this variable. The statement will be executed as long as the condition, which is an expression, is true. All three fields are optional (although the separating semicolons are necessary). If the initialization or update is left out, they are considered null statements (statements that do nothing). If the condition is left out, it is considered to be always true, and the loop will continue until a statement is reached to break out of the loop.

Anything that can be done with a "for" statement can also be done with an equivalent "while" statement. The conversion is simple:

```
initialization;
while (condition)
{
  statement;
  update;
}
```

However, sometimes the "for" statement is more readable, and it is quite commonly used.

Here is the program to print "Hello World!" to the screen three times on three separate lines using a "for" statement:

```
#include <stdio.h>

int main(void)
{
  int x;
  for (x = 0; x < 3; x++)
    printf("Hello World!\n");

  return 0;
}
```

Here, "x" is initialized to 0, and incremented by 1 after each iteration of the loop. After 3 iterations, the value of "x" will be 3, and the condition "x < 3" will no longer be true so the loop will end.

It is also possible to have the initialization and updates consist of multiple statements separated by commas. For example, here is an example that prints the first 10 powers of 2 to the screen:

```
#include <stdio.h>

int main(void)
{
  int power, result;

  for (power = 1, result = 2; power <= 10; power++, result
= result * 2)
  {
    printf("2 to the power of %d equals %d.\n", power,
result);
  }

  return 0;
}
```

Here we are initializing "power" to be 1 and "result" to be 2. At the end of each iteration of the loop, we increase "power" by 1 and multiply "result" by 2. We continue as long as "power" is less than or equal to 10.

As with "while" statements and "do…while" statements, a "break" statement encountered within a for loop will skip to the line following the end of the "for" loop.

Sometimes, within compound statements that are part of loops, there will be additional loops. Loops within loops are referred to as **nested loops**. Sometimes you may hear people refer to **outer loops** (the loop encountered first) and **inner loops** (any loops embedded within an outer loop).

We have already looked at a program which sums the digits of a number. Let's say we want to print out all numbers from 1 to 1000 such that the sum of the digits equals 5. Here is the code:

```
#include <stdio.h>

int main(void)
{
  int x, sum_digits, digit, temp;

  for (x = 1; x <= 1000; x++)
  {
    temp = x;
    sum_digits = 0;
    while (temp > 0)
    {
      digit = temp % 10;
      sum_digits = sum_digits + digit;
      temp = temp / 10;
    }

    if (sum_digits == 5)
      printf("%d\n", x);
  }

  return 0;
}
```

Here, using the "for" loop, we loop through the numbers from 1 to 1000. For each number, we use a nested "while" loop to sum the digits of the number, and check if this sum equals 5. If so, we print out the number.

A couple of things to notice here: First, you must remember to reset "sum_digits" to 0 at the beginning of each iteration of the "for" loop. Otherwise, the sum of the digits of each number will get added to the sum of the digits of all previous numbers!

Second, notice that we copy the value of "x" to the variable "temp" at the beginning of each iteration of the "for" loop and use "temp" within the "while" loop. This is necessary for two reasons. One is that if we changed the value of "x" (kept dividing it by 10 until it was 0), then when we reached the end of each iteration, "x" would be 0, and when we update it by incrementing by 1, "x" would be 1. The value of "x" would never reach 1000, and we’d be in an infinite loop. The second reason is that we want to print out the value of "x" after the "while" loop if the sum of the digits happens to be 5.

Here is an example of using nested loops to print out a multiplication table:

```
#include <stdio.h>

int main(void)
{
  int row, col;
  printf("\t0\t1\t2\t3\t4\t5\t6\t7\t8\t9\n");
  for (row = 0; row <= 9; row++)
  {
    printf("%d", row);
    for (col = 0; col <= 9; col++)
    {
      printf("\t%d", row*col);
    }
    printf("\n");
  }

  return 0;
}
```

Using the '\t' symbol within a string passed to "printf" causes a tab to be printed, and the computer skips to the start of the next 8 character column. The '\t' symbol is a special way of representing the tab character, in the same way that the '\n' symbol is a special way of representing the newline character. The first "printf" in this program skips the first column (since no characters appear to the left of the first tab), and then prints out column headers 0 through 0 in the next 9 columns.

We then loop through 9 rows of the table. At the beginning of each row, we print the row number. Then, we loop through 9 columns, and for each, we tab over to the next column and print the product of the row number and column number. After the inner "for" loop (at the end of each iteration of the outer "for" loop), we print a '\n', which causes the program to move the cursor to the start of the next line.

If you encounter a "break" statement in the middle of a nested loop, the control of the program jumps to the first statement after the innermost loop surrounding the "break" statement. For instance, let's say in the previous program you only want to print out the half of the multiplication table that is below the diagonal line from the top left to the bottom right. There is no need to print the result of 2*7 if you are going to print the result of 7*2 anyway!

We can make this adjustment by adding one "if" statement to our program:

```
#include <stdio.h>

int main(void)
{
  int row, col;

  printf("\t0\t1\t2\t3\t4\t5\t6\t7\t8\t9\n");
  for (row = 0; row <= 9; row++)
  {
    printf("%d", row);
    for (col = 0; col <= 9; col++)
    {
      printf("\t%d", row*col);
      if (col == row)
        break;
    }
    printf("\n");
  }

  return 0;
}
```

Now, for each row, once the column equals the row, we skip the rest of the inner "for" loop and jump to the line that prints the newline character. We then move on to the next row.

The last statement we will look at related to loops is the "continue" statement. The "continue" statement causes the rest of the current iteration of the current loop to be skipped. If it is a "for" loop, the updates that occur at the end of each iteration still occur. If the condition of the loop is still met, the next iteration begins; otherwise, the loop ends.

For example, let’s say we want to print out all numbers from 1 to 20 that are NOT divisible by 5. One way to do this is:

```
#include <stdio.h>

int main(void)
{
  int x;

  for (x = 1; x <= 20; x++)
  {
    if (x % 5 == 0)
      continue;

    printf("%d\n", x);
  }

  return 0;
}
```

The first thing to note here is the test for divisibility by 5. Remember that the "%" is the modulus operator; it returns the remainder when the first operand is divided by the second operand. If the remainder when "x" is divided by 5 is 0, then "x" is divisible by 5. When x is not divisible by 5, the condition of the "if" statement is false, so we reach the "printf" statement and print out the number. When "x" is 5, 10, or 15, the condition of the "if" statement is true, so we "continue", or skip, to the end of the current iteration of the "for" loop, still do the update "x++", and start the next iteration of the loop. When "x" is 20, we skip to the end of the current iteration of the "for" loop, still do the update "x++", but now x will be 21, so the condition of the "for" loop is no longer met, and we end the loop.