CS102: Selection Statements

A **Boolean expression** is an expression that is either true or false. (Related to this, logical data is data that has the value of true or false.) Standard C does not have a Boolean data type. However, it does use Boolean expressions. If an expression is true, it is given the value of 1. If an expression is false, it is given the value of 0. Similarly, if a numeric expression is used as a Boolean expression, any non-zero value will be considered true, while a zero value will be considered false.

One of the most important statements in C is the "if" statement. Here is the general format:

```
if (expression)
  statement;
```

In this case, the statement within the "if" statement will only be executed if the expression is true.

Here are a few very simple programs using an "if" statement:

Program #1:

```
#include <stdio.h>

int main(void)
{
  if (1)
    printf("Hello World!\n");
  return 0;
}
```

Program #2:

```
#include <stdio.h>

int main(void)
{
  if (-55)
    printf("Hello World!\n");
  return 0;
}
```

Program 3:

```
#include <stdio.h>

int main(void)
```

```
{
  if (0)
    printf("Hello World!\n");
  return 0;
}
```

The first two programs above will print "Hello World!" to the screen, since non-zero values are considered true. The third program will do nothing, since a zero value is considered false.

Here are two more examples, with more sensible (but still pretty useless) Boolean expressions:

Program 4:

```
#include <stdio.h>

int main(void)
{
  if (7 > 5)
    printf("Hello World!\n");
  return 0;
}
```

Program 5:

```
#include <stdio.h>

int main(void)
{
  if (5 > 7)
    printf("Hello World!\n");
  return 0;
}
```

The first of these two programs will print "Hello World!" to the screen, since the expression is true, but the second will not.

Not that there is no semicolon after the right parenthesis following the Boolean expression; if there were, then the semicolon would represent the end of a statement, and we would be saying "if an expression is true, do nothing".

In order to show some semi-useful examples of the "if" statement, we're going to introduce another function from the "stdio" library (the same library that contains "printf") called "scanf". The "scanf" function can be used to have the user enter a value for a variable. You will not understand how this works until much later.

```
#include <stdio.h>

int main(void)
{
  int x;

  printf("Please enter an integer: ");
  scanf("%d", &x);
  if (x > 10)
    printf("Hello World!\n");
  return 0;
}
```

This program allows the user to type a number and will print "Hello World!" if and only if that number is greater than 10.

There are a few things to note here.

1) Notice there is no '\n' at the end of the string in the first printf statement. Therefore, the user will be entering the number on the same line that the string is printed. Of course, you could include a '\n' if you want; it's just a matter of taste.
2) VERY IMPORTANT: Notice the '&' to the left of the variable "x" in the call to "scanf". What follows is a brief explanation of why this is there, but you won't really understand it until you have learned about pointers and function calls. Normally, when you pass a variable to a function, the value of the variable is copied to another location in memory, and this copy is used by the function. Because of this, it is impossible for functions to change the value of the variable that was passed. The '&' sign (called the "address-of" operator) tells the computer that we are NOT passing the value of "x", but rather the memory address of "x" (the location in memory where the value of "x" is stored). The function "scanf" can write a new value into this memory location, and therefore produce the effect of changing the value of "x". It is a very common error to forget to include the '&' sign to the left of a variable passed to "scanf". Sometimes, it is a hard error to catch for beginner programmers. The value of the variable won't be changed, and "scanf" may wind up writing to some random location in memory, which could have weird effects.
3) The function "scanf" sometimes has unpredictable behavior when the user does not enter what he or she is supposed to. For instance, with the above program, if the user enters a string instead of an integer, the value of "x" will be unpredictable.

The '>' is an example of a **relational operator**. A relational operator takes two operands and compares them to each other, resulting in a value of true (1) or false (0). The '>' is just one relational operator provided by C. There are six relational operators:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal

Here is an example of the use of the equal operator:

```
#include <stdio.h>

int main(void)
{
  int x;

  printf("Please enter an integer: ");
  scanf("%d", &x);
  if (x == 10)
    printf("Hello World!\n");
  return 0;
}
```

You can probably guess that this program prints "Hello World!" to the screen if and only if the integer entered by the user is 10.

Note that there are TWO '=' characters in a row. It is a very common error to use just one. This type of error is often very hard to catch! The problem is that with just one '=' sign, the program would still be valid. An assignment statement is also given the value that is assigned to the variable. So:

```
if (x = 10)
  statement
```

will always assign the value of 10 to "x" and then result in the inner statement being executed (since 10 evaluates to true)!

```
if (x = 0)
  statement
```

will always assign the value of 0 to "x" and the inner statement will not be executed (since 0 evaluates to false)!

It may look like x is being compared to 10 or 0 because of the "if" statement, but this is not the case.

Always remember to use "==" for comparisons in an "if" statement or other Boolean expressions.

We've listed the relational operators above. C also has three **logical operators**:

!	not
&&	Logical and
||	Logical or

Consider this program:

```
#include <stdio.h>

int main(void)
{
  int x;

  printf("Please enter an integer: ");
  scanf("%d", &x);
  if ((x >= 10) && (x <= 20))
    printf("Hello World!\n");
  return 0;
}
```

This program prints "Hello World!" to the screen if and only if the integer entered by the user is greater than or equal to 10 and less than or equal to 20.

Now consider:

```
#include <stdio.h>

int main(void)
{
  int x;

  printf("Please enter an integer: ");
  scanf("%d", &x);
  if ((x < 10) || (x > 20))
    printf("Hello World!\n");
  return 0;
}
```

This program prints "Hello World!" to the screen if and only if the integer entered by the use if less than 10 or greater than 20.

It is possible to have multiple statements executed when the expression checked by an "if" statement is true. Here is an example:

```
#include <stdio.h>

int main(void)
{
  int x;

  printf("Please enter an integer: ");
  scanf("%d", &x);
```

```
  if (x > 10)
  {
    printf("The number you entered is greater than 10.\n");
    printf("%d is still a positive number.\n", x-10);
  }
  return 0;
}
```

This program has the user enter a number. If and only if the number is greater than 10, the program will inform the user that the number is greater than 10, and also that the number minus 10 is still positive. One thing to notice here is that the value being passed to "printf" is the outcome of an expression, which is perfectly valid. The expression is evaluated and only the result is sent to the function.

A collection of zero or more statements inside of left and right curly braces is known as a **compound statement**, or a **block**. A compound statement can be used anyplace that a single statement can be used!

There are several common statements that, like the "if" statement, make compound statements useful, but you could place one anywhere that you can place a single statement. Here is a silly example:

```
#include <stdio.h>

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

  x = 10;
  y = 100;
  z = 1000;

  {
    printf("The value of x is %d.\n", x);
    printf("The value of y is %d.\n", y);
    printf("The value of z is %d.\n", z);
  }
}
```

You can probably guess what this program prints to the screen, but you may be asking, "What is the use of having a compound statement in the middle of this function?". The answer is, there is no use! You might as well just have three single statements here. The compound statement is confusing, and this is sloppy code, but it is valid and it works. The example is only being shown to explain that a compound statement can be used anywhere that a single statement can be used. Remember back to the generic example of the "if" statement:

```
if (expression)
  statement;
```

Because C was designed in such a way that compound statements can always replace single statements, an "if" statement can automatically accept a compound statement to execute when the expression is true!

Every function (including "main") has a **function body** that is a compound statement.

Every compound statement can optionally include local declarations after the opening curly brace!

Here is an example that also introduces a new library:

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

int main(void)
{
  float x;

  printf("Please enter a positive number: ");
  scanf("%f", &x);
  if (x > 0)
  {
    float y;

    y = sqrt(x);
    printf("The square root of %f is approximately %f.\n",
x, y);
  }

  return 0;
}
```

One of the things we're doing here is introducing a new library, the math library. Like the standard input/output library which provides the function "printf", the math library includes functions which are often useful, including the function "sqrt" which takes a floating point number as input and returns a floating point number as output. In order to use the math library, the compiler has to link it with your source code.

Here things get a little confusing! There are some provided libraries that the compiler will link with automatically. The "stdio" library is one of those libraries. There are other libraries that you have to tell the compiler to link with explicitly! The math library is one of these libraries (although some compilers might also link to it automatically). In order to force the compiler to link with the math library, you have to use the "-l" parameter followed by the letter "m" (with no space in between) when you call the compiler. For

example, suppose the above code is stored in a file called "sqrt.c", and you want the executable file to be named "sqrt". Then you can compile by typing:

```
gcc –o sqrt sqrt.c –lm
```

After you compile, of course, you can run the executable program by typing "./sqrt" at the Linux command prompt.

More specifically, when the parameter "–l<name>" is used with the compiler, the compiler looks for a file named "lib<NAME>.a" in a specific directory. The name of the math library is "libm.a".

You may be wondering why you have to have the "#include <math.h>" line at the top of the program if you are going to tell the compiler to link with the library using "-lm" anyway. You'll be able to understand this better after learning about functions. The quick answer is that in order to call a function whose code does not appear above the line that calls it, you need to have a function prototype statement above the code that calls it. The "#include" directives are actually including these prototype statements for functions in the appropriate libraries.

Now look back at the code. We're using "scanf" to have the user enter a floating point number. Like with "printf", the "%f" symbol is used to represent a float.

We then use an "if" statement to ensure the number is positive, since we can't take the square root of a negative number. If we try, we would get a run-time error! This might cause the program to crash, or it might cause the function to return a weird value.

If the number the user enters is positive, we fall in to the compound statement guarded by the "if" statement. At the start of this compound statement, we declare a local floating point variable "y". We set the value of "y" to the value of the square root of "x" and print it out.

Now we could have declared "y" at the top of the function main along with "x" (either as part of the same declaration or with a separate declaration). Either way would be valid, but the convention for standard C is to declare all the variables used in a function at the beginning of the function. (This is not true of other languages, including C++, and ultimately, it is a matter of preference.) Some standard C programmers consider this program to have bad style, but it is being shown here to demonstrate that local variables can be declared at the start of any compound statement.

This brings us to another important definition: The **scope** of a variable is the portion of a program in which you can use that variable. Some people prefer to say that the scope of a variable is the region of the program in which the variable is visible.

When you declare a local variable at the beginning of a block, the scope of that variable is that block. If you try to use the variable outside of the block, you will get a compiler

error. If you declare a variable at the beginning of a function, you can use the variable anywhere in that function. In the example above, the variable "y" could only be used inside the compound statement which is guarded by the "if".

You will learn more about scope when you learn about functions.

Sometimes, you want to execute certain code if a certain condition is true, and do something else otherwise. For this, C provides the "if…else" statement. The general format is:

```
if (expression)
  statement1;
else
  statement2;
```

As you might guess, statement1 is only executed if the expression in parentheses is true, and statement2 is executed otherwise.

Here is an example:

```
#include <stdio.h>

int main(void)
{
  int x, y;

  printf("Please enter an integer: ");
  scanf("%d", &x);
  if (x >= 0)
    y = x;
  else
    y = -x;
  printf("The absolute value of %d is %d.\n", x, y);
  return 0;
}
```

You can probably figure out what this does.

Of course, the statements after the "if" or "else" clauses could be compound statements. The following is a valid statement (assume "y", "z", "count1", "count2", and "diff" are int variables):

```
if (y > z)
{
  count1 = count1 + 1;
  diff = y - z;
}
```

```
else
{
  count2 = count2 + 1;
  diff = z - y;
}
```

This might be part of some program storing the number of times that "y" is greater than "z" in "count1" and the number of times that "z" is greater than "y" in "count2", and also doing something with the difference.

The statements within the "if" or "else" clauses might themselves be "if" or "if…else" statements! If so, the inner "if" or "if…else" statement is said to be nested. For example:

```
if (y > 10)
  if (z < 5)
    x = 10;
  else
    x = 5;
else
  x = 99;
```

Here's a case of code that could be confusing:

```
if (x > 10)
  if (x < 20)
    printf("The value of x is between 10 and 20.\n");
else
  printf("What do we know about x?");
```

The problem about this code is that it is indented badly. It appears that the "else" clause matches the first "if" clause and that the second message would only get printed if "x" is less than or equal to 10. However, in actuality, an "else" clause is always matched with the most recent available "if" clause. (Remember, C ignores indentation and other white space.) So the second message is printed if "x" is greater than 10 and greater than or equal to 20 (i.e., simply greater than or equal to 20). The code should be formatted as follows to avoid confusion:

```
if (x > 10)
  if (x < 20)
    printf("The value of x is between 10 and 20.\n");
  else
    printf("What do we know about x?");
```

If you wanted the behavior of the code to be what the first of these two examples appears to be doing, you can force this behavior with appropriate curly braces as follows:

```
if (x > 10)
```

```
{
  if (x < 20)
    printf("The value of x is between 10 and 20.\n");
}
else
  printf("What do we know about x?");
```

Now, the else clause must match the first "if" clause, since the second "if" statement is within its own compound statement, ended by the right curly brace. (Note: It is perfectly valid to have a compound statement with just one statement. In fact, you can have an empty compound statement, represented "{}", but this won't serve any purpose.)

Now, we are going to look at the else-if statement, used when there are more than just two possible states that we want to distinguish. Here is an example:

```
#include <stdio.h>

int main(void)
{
  int x, y;

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

  if (x <= 0)
    printf("That number is not positive!\n");
  else if (x < 10)
    printf("That number has one digit.\n");
  else if (x < 100)
    printf("That number has two digits.\n");
  else if (x < 1000)
    printf("That number has three digits.\n");
  else
    printf("That number has four or more digits.\n");

  return 0;
}
```

It should be pretty obvious what this program is doing. We only get to a particular case if all the ones before it were not true. The final "else" clause takes care of everything that has not fallen in to one of the other cases. This program will print out only one message.

The "if" statement is an example of a **selection statement** because the program selects one path to follow depending on certain conditions. A simple "if" statement follows the "if" clause if an expression is true and skips it otherwise. An "if…else" statement and those with "else-if" clauses are more complex.

Another selection statement in C is the "switch" statement. Here is an example:

```
#include <stdio.h>

int main(void)
{
  char c;

  printf("Enter a character: ");
  scanf("%c", &c);

  switch(c)
  {
  case 'a': case 'A':
  case 'e': case 'E':
  case 'i': case 'I':
  case 'o': case 'O':
  case 'u': case 'U':
    printf("%c is always a vowel!\n", c);
    break;
  case 'y': case 'Y':
    printf("%c is sometimes a vowel!\n", c);
    break;
  default:
    printf("%c is never a vowel!\n", c);
    break;
  }

  return 0;
}
```

The beginning of this program should be clear; the program has the user enter a character. The expression inside the parenthesis after the "switch" is an expression. It doesn’t have to be a single variable. Whatever it is, the expression is evaluated. In this case, the value of the expression is just the value of the variable.

A "switch" statement is followed by a series of "case" statements inside curly braces. If the value of the evaluated expression matches the value being checked by a "case" statement, the code following that "case" statement (or set of "case" statements) is executed. There can be one statement or multiple statements following each case. If there are multiple statements, you do not need to make them a compound statement, but you can if you want to.

When the computer reaches a "break" statement, it skips to the end of the "switch" statement. In other words, the computer jumps to the statement following the right curly brace that ends the switch statement. The "break" statement has different meanings

depending on how it is used; This is how it is used with a "switch" statement, but you will see other uses of it later.

If there is no "break" statement at the end of a series of statements, the execution will continue with the next set of statements. For instance, if the first "break" statement in the above code was omitted, and the user typed a vowel, the program would first display the "…always a vowel!" message and then display the "…sometimes a vowel!" message. When the next "break" was encountered, the switch statement would end.

If no "case" statement is matched, and there is a "default" statement in the switch statement, the code following it is executed. The "default" statement is optional; without it, if no case is matched, the switch statement is skipped.

The expression following each "case" statement must be a constant expression; i.e., it cannot contain variables. No two "case" statements can be checking for the same value or you will get a compiler error.

The spacing and indentation of the "switch" and "case" statements are, of course, left for the programmer to decide. If you have multiple "case" statements together, they could all be on one line, all be on separate lines, or mixed like in the example above. Whatever you think is clear and readable.

The "switch" statement is not very commonly used. It can always be replaced with a series of else-if statements. You can only use "switch" and "case" statements if you are checking an expression against a finite amount of constant, integral or character values! Even when you can use "switch" efficiently, it is just a matter of personal preference whether you decide to do it this way of with else-if statements. Sometimes, as in the example above, a "switch" statement is probably more readable, as otherwise you would probably need to use very long expressions.