CS102: Variables and Expressions

The topic of **variables** is one of the most important in C or any other high-level programming language. We will start with a simple example:

```
#include <stdio.h>

int main(void)
{
  int x;

  x = 5;
  printf("The value of x is %d.\n", x);
  return 0;
}
```

In this example, "x" is the name of a variable. The line "int x;" is an example of a local declaration. We are declaring that "x" is the name of a variable of type "int" which stands for "integer". So "x" can take on integer values. When the computer sees a declaration like this, it sets aside a space in memory to store the value of the variable. The line "x = 5;" sets the value of "x" to 5. We've previously seen "printf" used to print only strings. The "%d" is a special code that causes the "printf" function to fill in to this part of the string with an integer value specified after the string. If an integer variable appears after the string, the value of that variable is displayed. You could also include a constant. If the value 5 is used instead of the variable "x", the output of this program would be the same. Either way, it causes the string "The value of x is 5." to be printed on its own line.

Technically speaking, a variable is a named memory location. Every variable has a **type**, which defines the possible values that the variable can take, and an **identifier**, which is the name by which the variable is referred.

In order to use a variable in a C program, you must first declare it. You declare a variable with a **declaration**. As in the example shown above, a declaration of a variable consists of the variable's type followed by the variable's name (identifier). There must be whitespace between the type and the name. It is also possible to declare multiple variables on one line, as we'll see later.

You probably have some notion of a variable from algebra. In algebra, most variables are single letters, like "x" or "y". These are acceptable variable names in C also, but when programming in C, you often you use several characters to create descriptive variable names. Here are the rules that bind possible variable names, or identifiers, in C:

1) The first character must be an alphabetic character (lower-case or capital letters) or an underscore '_'.
2) All characters must be alphabetic characters, digits, or underscores.
3) The first 31 characters of the identifier are significant. Identifiers that share the same first 31 characters may be indistinguishable from each other.
4) Cannot duplicate a reserved keyword (i.e., a word that has a special meaning in C).

Examples of valid identifiers include:
x
Name
first_name
name2
_maxInt

Examples of invalid identifiers include:
2name /* Can't start with number */
first name /* Can't use ' ' */
first-name /* Can't use '-' */
int /* Can't use reserved word */

The type, or data type, of a variable determines a set of values that the variable might take and a set of operations that can be applied to those values.

Five standard data types provided for by C are:

- **int** – used for variables that can take on numeric integer values.
- **char** – used for variables that can take one character from the computer's "alphabet" as a value. Most computers use the American Standard Code for Information Interchange (ASCII) alphabet.
- **float** – used for variables that can take on numeric values with a fractional part.
- **double** – similar to "float", allowing greater precision
- void – used mainly for functions that don't take any parameters or functions that don't return a value.

In addition to these basic types, there are **modifiers** that you can apply to the types. For example, you can specify that a variable is a signed int (one that can take both positive and negative values, which is the default) or an unsigned int (one that can only take positive values).

Remember that a variable is a named memory location. The memory of a computer is divided into bytes. A byte is a sequence of 8 bits. A bit is a 0 or a 1. The number of bytes that a variable occupies in memory determines the number of possible values that it can take. C does not specify the size of an integer variable. Most computers today use 32-bit integers (4 bytes), although in the past, many used 16 bit integers (two bytes). One way to control the size of an integer variable is to specify that it is a "short int" (16 bits) or a "long int" (32 bits).

Let's say that an integer uses 4 bytes, or 32 bits, of memory. Then there are 2^32 possible values that this variable can take (corresponding to the 2^32 possible combinations of 0's and 1's). This leads to the following facts:

- A 32 bit integer can have 2^32 = 4,294,967,296 possible values.
- A signed 32 bit integer can have values ranging from –2,147,483,648 to 2,147,483,647.

- An unsigned 32 bit integer can gave values ranging from 0 to 4,294,967,295.
- A 16 bit integer can have 2^16 = 65,536 possible values.
- A signed 16 bit integer can have values ranging from –32,768 to 32,767.
- An unsigned 16 bit integer can have values ranging from 0 to 65,535.

A character, or "char", occupies just one byte of memory. Remember 1 byte = 8 bits, so there are only 2^8 = 256 possible characters. As mentioned above, the "alphabet" used by most computers is the **ASCII** encoding scheme. There are actually only 128 characters defined in standard ASCII, corresponding to the numbers from 0 to 127. So characters with values between 128 and 255 won't be consistent from computer to computer.

Here are some of the important ranges within the ASCII encoding scheme:
48 – 57: the digits '0' through '9'
65 – 90: the capital letters 'A' through 'Z'
97 – 122: the lowercase letters 'a' through 'z'

Here is a simple, useless example of a program using a character variable:

```
#include <stdio.h>

int main(void)
{
  char x;

  x = 'Q';
  printf("The value of x is %c.\n", x);
  return 0;
}
```

This program prints out the string "The value of x is Q." on its own line.

There are two ways to specify a character in C. A simple character that appears on the keyboard can usually be specified by just typing the character within single quotes, like in this example. The other way is to specify it with a number, the number whose ASCII code represents the character. Remember that the range from 65 to 90 in ASCII represents the capital letters. This means that 81 represents 'Q'. So an equivalent statement to "x = 'Q';" would be "x = 81". Because x is declared to be a "char", the value 81 represents the letter Q.

Notice that instead of "%d", the "printf" string now contains a "%c", which means that the value that gets printed in its place should be considered a character. If a "%d" were used in the program, it would print "The value of x is 81.".

In C, the '=' symbol is the **assignment operator**. It causes the computer to evaluate the **expression** to the right of the '=' and assign the value of the expression to the variable to the left of the '='. So let's say that x is an integer. The statement "x = 5;" will set the

value of the variable named "x" to 5. This is different than the use of '=' in algebra, in which it is used to express a fact. In algebra, if you see the following two lines:

```
x = 5;
x = 7;
```

They would express a contradiction. In C, this is perfectly valid. The first statement sets the value of x to 5. The second resets the value of x to 7. The value of x is no longer 5. In fact, if these two statements appear right next to each other in a C program with nothing in between, the first of the two statements is useless, but perfectly valid.

The expression to the right of the '=' doesn't have to be a single constant. It can be a mathematical expression, possibly containing variables and/or constants, and also containing **arithmetic operators**. In the following examples, we're going to assume that x and y are integer variables. The following statement is valid:

```
x = 2 * (3 + 99) / 20;
```

Since x is declared to be an integer, and the expression to the right of '=' evaluates to 204 / 20 which is 10.2, the final value of x will be 10. The fractional part of the expression is cut when performing integer division.

The following statement is also valid:

```
x = y + 5;
```

In this case, the value of x will be 5 more than the value of y.

Here is another valid statement:

```
x = x + 1;
```

In algebra, if you ever derive a formula like this, it either means that you made a mistake or that there was a contradiction between other formulas somewhere. In C, this is a perfectly valid and very common statement. Remember, the expression to the right of the equal sign is evaluated first, and then its value is then assigned to the variable to the left. So if x equals five when this statement is first reached, the expression to the right of '=' evaluates to 6, and the value of 6 is assigned back to x.

Here is an example of a non-valid statement:

```
x + 1 = 10;
```

In algebra, you could solve this and discover that x = 9. In C, this is not valid. You can not assign a value to the quantity "x+1". Only variables can appear to the left of '='. (Technically speaking, there are other so-called l-values that can appear to the left of the assignment operator, but typically l-values are variables.)

Now consider one of the previous programs with an extra line:

```
#include <stdio.h>

int main(void)
{
  char x;

  x = 'Q';
  x = x + 9;
  printf("The value of x is %c.\n", x);
  return 0;
}
```

Since "x" is a character, the statement "x = ‘Q’;" is equivalent to "x = 81". The new value of x after the statement "x = x + 9;" will be 90, which is the ASCII code for a capital Z. So this program prints "The value of x is Z." to the screen on its own line.

There are some special characters that have special symbolic sequences set aside to represent them. We've already seen one case of this; the newline character is represented by ‘\n’. It is also number 10 in the ASCII chart. So if x is a character, then the lines "x = ‘\n’;" and "x = 10;" are equivalent. Some other special characters are:

\t	tab
\b	backspace
\a	bell
\’	single quote
\"	double quote
\\	backslash

Here is a simple example using a float:

```
#include <stdio.h>

int main(void)
{
  float x;

  x = 2.5;
  x = x * 3;
  printf("The value of x is %f.\n", x);
  return 0;
}
```

Note the "%f" in the printf string, which means that the value that gets printed in its place should be considered a float. This may not print exactly what you expect. This program

will print the string "The value of x is 7.500000." on its own line. It is possible to control the format of how floats get printed, but that is a later topic.

When you have multiple variables of the same type, you can declare them with multiple lines or one line. The following three lines:

```
int first;
int second;
int third;
```

are equivalent to the following one line:

```
int first, second, third;
```

Variable in C are not necessarily initialized automatically. You should always initialize a variable before using its value. The following program has unpredictable behavior:

```
#include <stdio.h>

int main(void)
{
  int x;

  /* x is not initialized, its value is not predictable */
  printf("The value of x is %d.\n", x);
  return 0;
}
```

Since x is never initialized, there is no way of predicting its value.

Variables can be initialized while they are declared. So the following code at the start of a function:

```
int x;
x = 5;
```

is equivalent to:

```
int x = 5;
```

If you declare multiple variables with one line and want to initialize all of them, you must specify all of their values. For example:

```
int x = 0, y = 0, z = 0;
```

This is different than:

```
int x, y, z = 0; // Only z is initialized
```

Another operator that C recognizes is known as the modulus operator. This operator returns the remainder when one operand is divided by another. Here's an example:

```
#include <stdio.h>

int main(void)
{
  int operand1 = 22, operand2 = 5, quotient, remainder;

  quotient = operand1 / operand2;
  remainder = operand1 % operand2;
  printf("When %d is divided by %d the quotient is %d and
the remainder is %d.\n", operand1, operand2, quotient,
remainder);
  return 0;
}
```

This program displays the string "When 22 is divided by 5 the quotient is 4 and the remainder is 2." to the screen (or standard output) on its own line.

This is the first example in which the "printf" string includes more than one variable. It's pretty straightforward how this works (although the way that this is handled within the actual "printf" function will be understood only after covering functions, and even then not really in its entirety in this introductory class). All instances of special symbols in the string get replaced with values given by the corresponding expressions appearing after the string, in order.

Two other common operators are the **increment** and **decrement** operators. The following statements (assuming that "x" is an integer):

```
++x;
x++;
```

are both equivalent to:

```
x = x + 1;
```

while:

```
--x;
x--;
```

are both equivalent to:

```
x = x -1;
```

You will often see these operators used to increment or decrement a variable that is also being used as part of an expression. If the operator appears before the variable, the variable's final value is used in the expression. If the operator appears after the variable, the variable's original value is used in the expression. For example, assuming that "x" and "y" are both integers:

```
x = 5;
y = x++;
```

After these two statements, y will equal 5 and x will equal 6.

```
x = 5;
y = ++x;
```

After these two statements, y will equal 6 and x will equal 6.

```
x = 5;
y = x--;
```

After these two statements, y will equal 5 and x will equal 4.

```
x = 5;
y = --x;
```

After these two statements, y will equal 4 and x will equal 4.

Notice that some of the expressions have been mixing variables with constants. So far, when we've used numerical constants, we just specified the number outright.

Sometimes, you may want to code common constants, such as pi, so that you don't have to type in the value multiple times. There are at least two ways to do this. One is to use the "const" qualifier in the declaration of a variable. For example:

```
const float pi = 3.14159;
```

Of course, this is not the exact value of pi, but an approximation!

The variable "pi" can now be used in expressions like any other float variable, but the value can never be changed. If your program tries to change it, the compiler will notice and give you an error.

Here is an example of the use of a constant in a program:

```
#include <stdio.h>

int main(void)
```

```
{
  const float pi=3.14159;
  int radius = 5;

  printf("The approximate area of a circle with radius %d
is %f.\n", radius, pi*radius*radius);
  return 0;
}
```

This program displays "The approximate area of a circle with radius 5 is 78.539753." to the screen.

Another way to do it is to use the preprocessor directive "#define". The "#define" preprocessor directive causes the compiler to replace all instances of specific text with other text. Here is an example of its use:

```
#include <stdio.h>

#define PI 3.14159

int main(void)
{
  int radius = 5;

  printf("The approximate area of a circle with radius %d
is %f.\n", radius, PI*radius*radius);
  return 0;
}
```

You might be surprised that the output of this program is not exactly identical to that of the last program (the last digit of the output was different when I ran these examples); I am not going to get into the specific reasons for that here, but it has to do with the finite precision of floating point values.

Two other things to note here: First, there is no semicolon at the end of the "#define" line. If there were, the semicolon would get added along with the 3.14159 wherever you see PI in the code, and this would cause an error. Second, using "#define" can lead to some weird errors. For instance, let’s say you define "PI" like here, and then later you try to name a variable "SPICE". The PI would be replaced with 3.14159 by the compiler, and this would lead to an error.

Two other topics related to expressions are those of **precedence** and **associativity**.

The precedence of operators determines the order in which order different operators are evaluated when they occur in the same expression. Operators of higher precedence are applied before operators of lower precedence.

Let us reconsider the statement we saw earlier (where "x" is an integer variable):

```
x = 2 * (3 + 99) / 20;
```

The precedence of multiplication and division are higher than the precedence of addition, but the precedence of parentheses is highest of all. So the value of "x" will be 10.

If the statement was written without the parentheses:

```
x = 2 * 3 + 99 / 20;
```

Then the value of "x" will be 10 (remember that integer division truncates).

The associativity of operators determines the order in which operators of equal precedence are evaluated when they occur in the same expression. Most operators have a left-to-right associativity, but some have right-to-left associativity.

Here are some more examples. Assume that x is an int:

```
x = 5 - 2 * 7 - 9;
```

The '*' has a higher precedence than '-' so it is evaluated first, and the statement is equivalent to:

```
x = 5 - 14 - 9;
```

The minus has left-to-right associativity, so the statement is equivalent to:

```
x = -18;
```

Also, the assignment operator (represented by the "=" sign) has lower precedence than either addition or multiplication or any other mathematical operator, and this is how C enforces the rule that the expression to the right of the '=' gets evaluated first and then the resulting value gets assigned to the variable to the left of the '='.

The '=' itself is one of the few operators that has right-to-left associativity. The following is actually a valid C statement, assuming that "x", "y", and "z" are integers:

```
x = y = z = 7;
```

This statement assigns the value of 7 to "z" first, then to "y", and then to "x".

If '=' was given left-to-right associativity, the old value of "y" would be assigned to "x", then the old value of "z" would be assigned to "y", and then the value of 7 would be assigned to "z". This is pretty counter-intuitive, and that's why '=' was given right-to-left associativity!