C Language Reference

Jeffrey La Favre

December 4, 2015

Variables and Constants

Variables are used in a program to store values. You could think of them as containers, where you can put numbers and characters. The value stored in a variable can be digits representing a number or it can be a character (a letter of the alphabet). In your robot programming you will use variables to store numbers for the most part.

As the name *variable* implies, the value that is stored in the container can change as the program runs. This is what distinguishes a variable from a constant. A constant does not change in value.

The names of variables are created by the programmer (that is you). There are some rules you need to follow and there are some words that you can't use. A variable name cannot have any spaces, it needs to start with a letter or the underscore character (_), and can contain letters and numbers. It is recommended that you use lower case letters for variables, although sometimes a few letters can be upper case to help make the name more recognizable. The reason for using lower case letters is that by convention, constants usually are all in upper case. It is easier for you to distinguish variables and constants if the constants are in upper case and variables in lower case.

Variable and constant names are case sensitive. If you decide to name a variable **tempC** at the start of your code in a declaration but then later refer to it as **tempc**, your program will have an error, because it won't recognize **tempc** as a valid variable name.

Let's take an example. Suppose in your program you want to calculate the circumference of a circle. In order to do that, you will need to use the value of π, which is a constant (a constant never changes value). So at the beginning of your program you define a constant and give it the name of PI (you used upper case letters) and assigned a value of 3.14159 to PI. Now you only need to use PI in your code instead of 3.14159 each time you want to multiply by π.

Suppose you want a variable to store the value of the circumference once you have done a calculation. You decide to name the variable **circum**, being short for circumference. You could have selected the whole word, **circumference** or just **c** for the name if you wanted. Or you could call it **b** or just about anything else. It does not matter. But you decided on **circum** because it is shorter than the whole word and won't take up as much space in your code. And you did not want to use **c** because you were worried you might forget what it represents. In any case, the point here is that you can use any name you want, as long as it conforms to the rules and is not a **reserved word**, already used in the C programming language.

Here is a list of reserved words, that you can't use as variable names: auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.

Before you can use a variable you must declare it. By declaring the variable you are giving it a name and letting the computer know it is a variable. If you don't declare the variable before you try to use it in your code, then an error will occur when you try to compile the code and you will not be successful in creating your program.

When you declare a variable you must also declare its **data type**. The most likely that you will use are integer (**int**) and floating point number (**float**). Integers are whole numbers and floating point numbers can have places to the right of the decimal point. Suppose you wanted to create an integer type variable named **ticks** and a floating point type named **distance**. Here is how you would declare the variables:

int ticks;

float distance;

Here are some additional data types: char = character – a single byte, *i.e.*, just one character can be stored; short = short integer; long = long integer; double = double-precision floating point. The table below lists some data types and gives the range of values that can be stored in an ActivityBot robot program.

Data Type	Range of Values for ActivityBot robot
signed char	-127 to 127
char	0 to 255
int	-2,147,483,647 to 2,147,483,647
unsigned int	0 to 4,294,967,295
long	same as int
unsigned long	same as unsigned int
float	approx: $3.4X10^{-38}$ to $3.4X10^{38}$ with 6 digits of precision

You might be wondering why the char (character) data type is listed with a range of 0 to 255. At the machine level, computers only work with numbers. We have to assign characters to numbers. There are various schemes of assignment. A common one is known as ASCII for short. ASCII stands for American Standard Code for Information Interchange. An ASCII code is the numerical representation of a character such as '**a**' or '**=**' or an action of some sort, like escape (the Esc key on your Windows computer keyboard). The ASCII value for lower case **a** is 97, for **=** the value is 61, for escape the value is 27. Since the char data type can range from 0 to 255, then there is room for 256 different characters and actions. The ASCII table available at http://www.asciitable.com/ lists the 256 characters and actions under its system.

Let's do a code example to illustrate the use of variables. We want to calculate the circumference of a circle. In order to do the calculation we need to know the radius of the circle. The formula is circumference = $2\pi r$, where r is the radius. To do this in a C language program let's make two variables, one named **c**, which will be used to store the value of the circumference and one named **r**, which will store the value of the radius. Let's take a look at the code that would accomplish this computation on the next page.

```
#include <stdio.h>

#define PI 3.14159

int c, r;

int main()

{                       //start of main() code block – (a comment starts with //)

    r = 30;

    c = 2 * PI * r;

    printf("circumference = %d", c);

}                       //end of main() code block
```

When the above code is compiled and run, the output on the computer screen will be: "circumference = 188". The first line, **#include <stdio.h>**, lets the computer know that the standard input/output library is to be included. That library contains the **printf()** function in it and the program won't work unless we name this library (in this case for a program made using Dev-C++, for robot programing the library name is different). The second line is the definition of the constant PI, which starts with **#define** then the name we want to use for the constant, **PI**, then the value for the constant. You do not use an equal sign here. Don't do it like this: #define PI = 3.14159. Also notice that there is no semicolon (;) marking the end of a #define line or a #include line.

The next line is **int c, r;** This is the declaration for two variables named c and r. The data type is integer (int). You can name as many variables as you want on the line, separating them with commas and be sure to end the statement with a semicolon (;).

Every C program must include the **main()** function. Most of your code usually is located inside the **main()** function. Program execution starts with the first statement inside the **main()** function. If the **main()** function contains more than one line under it, that is, more than one **statement**, then the statements must be enclosed in curly braces **{}**. Notice that the first line below **int main()** has the beginning brace **{** and the last line has the ending brace **}**. This lets the computer know that all lines between **{** and **}** belong to the **int main()** function.

Inside the **main()** function we find the statement **r = 30;** which causes a value of 30 to be stored in the variable named **r**. Then the next statement is the computation: **2** is multiplied by **PI** and the result is multiplied by the value stored in **r**. The result of the complete computation is stored in the variable named **c**.

The next statement codes for the display of the value stored in variable **c** (see the **printf()** function below for an explanation of how it works).

Array Variables

If you need to create several variables that are related to each other, it can be an advantage to use an array of variables. For example, suppose you want to track the number of wheel ticks for the left wheel as the robots operates. You need to store a number at various locations in the execution sequence of the program. You decide to name the array variable **leftTicks**. Here is one method of declaring the array:

int leftTicks[4];

The **[4]** immediately after the name of the variable lets the computer know that this is to be an array variable. Furthermore, the array will contain four variables for storing values. The actual names of the variables will be: **leftTicks[0]**, **leftTicks[1]**, **leftTicks[2]**, **leftTicks[3]**. It should be evident that this system is an efficient method of declaring a set of variables. Let's compare it to an alternative, declaring 4 separate variables:

int leftTicks1, leftTicks2, leftTicks3, leftTicks4;

It should evident that **int leftTicks[4];** is a more efficient declaration. And the array becomes even more efficient as the array size increases. What if you needed to create 20 variables? It would be fairly easy using an array:

int leftTicks[20];

Using the alternative, you would need to declare 20 variable names. That could be tedious.

Suppose you want to store 6 numbers in an array. I will give you two methods for this. Let's create an array of variables with the name **dist**, which is short for some distance value. Here is the first method:

int dist[] = {1, 2, 3, 5, 7, 11};

The above line declares an array variable named **dist** and assigns to it the numbers listed between curly braces **{** and **}**. I think you will admit this is efficient. Note in this case that we don't specify the number of variable elements to include in the array. There is no number between the brackets **[]**. It is not necessary to make that specification because there are 6 values listed between the braces **{}**. In other words, this will be an array of 6 elements. Suppose we want to retrieve the value 5, which is stored in the fourth element of the array. This is how you would refer to the element in code: **dist[3]**. Why did I place a 3 inside the brackets instead of a 4? Let me name all the elements in sequence so you can understand: **dist[0]**, **dist[1]**, **dist[2]**, **dist[3]**, **dist[4]**, **dist[5]**. The first element in the array has an index number of 0, not 1. Therefore, the fourth element in the array has an index number of 3.

Here is a second method of assigning values to the array:

```
int dist[6];

dist[0] = 1, dist[1] = 2, dist[2] = 3, dist[3] = 5, dist[4] = 7, dist[5] = 11;
```

I think you would agree that this second method is not as efficient as the first. This line:

int dist[6];

declares the variable with the name **dist** and specifies that it will contain 6 elements. Then the next line lists out each element, with its index number, and assigns a value to it.

Input and Output Functions

Displaying text on a computer screen

printf() or for ActivityBot robot **print()** is a function used to output text. In other words, this function causes text to be displayed in the program window on the computer screen.

Example:
printf("Hello World!");
This line of code would result in the display of this on the computer screen: **Hello World!**

Any text you put between the double quotes is the text that will be displayed on the screen. There will be occasions when you want the text display to include some information stored in a variable. You can't just type in the variable name as part of the text between double quotes. That would just print the name of the variable on the screen. Here is how you would code to include a display of values stored in variables:

printf("The temperature is %d degrees F which is the same as %d degrees C.", degF, degC);

In the above statement there are two decimal placeholders (**%d**). The placeholders are the locations in the text where the values of the variables will appear. Notice that after the end quote the code has this: **, degF, degC** . These are the variable names which hold the values to be displayed. The first one listed is **degF** and therefore, its value will appear where the first **%d** appears in the text. Suppose the value stored in **degF** was 68 and the value 20 was stored in **degC**. Then our text displayed on the screen would look like this:

The temperature is 68 degrees F which is the same as 20 degrees C.

The variable names **degF** and **degC** are just the names a programmer might create to store a temperature value in degrees F and degrees C.

The **%d** placeholder must be used with variables of the integer (int) type. If you have variables of the floating point type (float) then you need to use **%f** as the placeholder. If you just use **%f**, then many places to the right of the decimal point will be displayed. If that is not what you want, then you need to modify the placeholder. Suppose you want to just display numbers with only two places to the right of the decimal point. Then use this placeholder: **%.2f** The **.2** part of the placeholder indicates that you want 2 places to the right of the decimal point to be displayed. If the number of places in the value stored is more than two, the number is rounded off to the nearest two places.

Usually your robot programs for autonomous control won't be sending text to the computer screen because the robot won't be connected to the computer (although you could do it wirelessly with the appropriate accessories). Nevertheless, sending text to the computer, while the robot is connected by wire, is helpful in checking some of the systems on the robot. In addition, if you are writing programs in C that are just for running on a computer, then of course the **printf()** function will be very important. Many programs need to have text output on a display screen to be of any value to the user.

Retrieving input from the keyboard

scanf() or for ActivityBot **scan()**, is a function that can be used to get input from a keyboard. Suppose you are writing a program to do some calculations. We can take our previous example of calculating the circumference of a circle. But this time we will not assign a specific value for the radius, but will ask the user of the program to input the radius. Now the program becomes more useful. Let's take a look at some example code:

```
#include <stdio.h>

#define PI 3.14159

int c, r;

int main()

{

    printf("Enter the value of the radius and then press the Enter key\n ");

    scanf("%d ", &r);  //the program halts here, waiting for user to input value

    c = 2 * PI * r;

    printf("The circumference of a circle with a radius of %d is %d.", r,  c);

}
```

The statement
scanf("%d ", &r);
retrieves the value input from the keyboard. A placeholder (**%d**) must be used inside the double quotes. Following that is the specification for the location to store the retrieved input. This is indicated by putting an ampersand (**&**) character directly in front of the variable name. Therefore, **&r** means store the input in the variable named **r**.

Suppose your robot is configured with a wireless device to communicate with your computer. Then it would be possible to control the robot from the computer keyboard. You could write a program, using the **scan()** function, that causes the robot to turn left when the L key is pressed, turn right when the R key if pressed, drive forward when the F key is pressed and drive backward when the B key is pressed.

Math Operations

The **binary arithmetic operators** are: **+** (add), **-** (subtract), ***** (multiply), **/** (divide) and **%** (modulus). Suppose you have a variable named **answer**. Then this is how you could code calculations:

answer = 4 + 5	result of calculation, 9, is stored in variable named answer
answer = 5 – 4	result of calculation, 1, is stored in variable named answer
answer = 5 * 4	result of calculation, 20, is stored in variable named answer
answer = 10/5	result of calculation, 2, is stored in variable named answer
answer = 10%3	result of calculation, 1, is stored in variable named answer

In the case of modulus, use the % character instead of / in a divide operation and the answer will be the remainder. In our example, 10 divided by 3 yields a quotient of 3 with a remainder of 1.

Relational and Logical Operators

The **relational operators** are: **> >= < <=**

Here are examples of their use:

(a > b) here we are asking this: is **a** greater than **b**? Suppose that the variable **a** is storing a value of 3 and the variable **b** is storing a value of 2. In this case then, the answer is TRUE, **a** is greater than **b**. The symbol (**>**) means greater than

(a >= b) here we are asking if **a** is greater than OR equal to **b**. If **a** equals 3 and **b** equals 2, then the answer would be TRUE. Also, if **a** equals **3** and **b** equals **3**, the answer would still be TRUE. If **a** equals **2** and **b** equals **3**, then the answer would be FALSE.

(a < b) here we are asking if **a** is less than **b**. If **a** equals 2 and **b** equals 3, then the answer would be TRUE. The symbol (**<**) means less than

(a <= b) here we are asking if **a** is less than OR equal to **b**. If **a** equals 2 and **b** equals 3, or if **a** equals 3 and **b** equals 3, then the answer would be TRUE.

The **equality operators** are: **==** **!=**

Here are examples:

(a == 10) in this case we are asking if **a** is equal to 10. Why don't we just do it this way (a = 10). In the latter case we are not posing a question, but rather assigning a value of 10 to **a**.

(a != 10) in this case we are asking if **a** is NOT equal to 10.

Two **other logical operators** are: **&& ||**

&& means AND, || means OR

Here are examples:

(a ==3) && (b ==4) this compound question would be TRUE if **a** equals 3 AND **b** equals 4.

(a == 3) || (b==4) this compound question would be TRUE if **a** equals 3 OR **b** equals 4 OR **a** equals 3 and **b** equals 4

Making Decisions - Loops

Loops are a very important part of programming in C language. Loops provide a way to repeat a block of code. The loop can be configured to repeat indefinitely, until the program is closed. Alternatively, the loop can be configured to repeat until some condition is met or not met.

The while() loop

The **while() loop** is one type of loop available. The **while()** loop continues to loop until the condition specified in the parentheses () evaluates to FALSE. The number 1 is equivalent to TRUE and the number 0 is equivalent to FALSE. If you write your code like this: **while(1)**, then the while loop will continue to run as long as the program is open. You could write this: **while(0)**, but it would make no sense, because the loop would never run. The other alternative is to place a condition inside the parentheses which can at times evaluate to TRUE and at other times can evaluate to FALSE. In that case you want the code block in the loop to run only while the condition is TRUE. Here is an example code snippet (*i.e.*, not a complete program):

```
int i = 1;

while(i < 6)

{                                    //begin of while() code block

        printf("The loop is running\n ");

        i = i + 1;

}                                    //end of while() code block, now program goes back to while() line
```

The above code will cause the computer to display 5 lines of text, each containing “The loop is running.” Why does it print 5 lines? Because the code block inside the while loop runs 5 times. Each time after a line is printed to the screen, the variable named i is incremented by a value of one (add 1 to i). When the computer gets to the bottom of the loop, it goes back to the line containing **while(i < 6)** and checks to see if i is still less than 6. The value of i starts out as 1 and by the time 5 lines have been printed, the value of i will be 6. Then the next time the value of i is evaluated at the top of the loop, it no longer is less than 6 and the loop stops running.

How could this apply to code for running a robot? Well, suppose our variable named i actually contains the value sent by a robot sensor that is equivalent to the distance to a wall in millimeters. The robot is supposed to follow along the wall but keep at a distance of no less than 6 mm from the wall. Furthermore, suppose we substitute the **printf()** function in the code with some code that makes the robot steer away from the wall. Then our **while()** loop would prevent the robot from getting too close to the wall. Whenever the robot gets closer than 6 mm, then the **while()** loop starts running and the robot steers away from the wall until the robot is at least 6 mm away from the wall.

The for() loop

The second loop type is the **for() loop**. This is a more compact form of a loop, but perhaps a bit more difficult to understand until you get used to using it. This is the format for a **for()** loop:

for(expression 1; expression 2; expression 3)

Any of the three expressions can be omitted, but both semicolons (;) must remain in place. Usually expressions 1 and 3 are assignments or calls to a function and expression 2 is used as a relational expression. So let’s see an example:

```
for(i = 1; i < 6; i++ )
```

The first expression, **i = 1**, assigns a value of 1 to the variable named i (in this case I am not showing the code that created the variable, which must come before the **for()** line). The next expression, **i < 6**, is a conditional expression. The **for()** loop will run as long as the value of i remains less than 6. The third expression, **i++** , is used to increment the value of i by one (the ++ after i just means add one to it). Let’s take our previous **while()** example and change it to a **for()** loop:

```
int i;     //create an integer type variable named i, but do not assign a value yet

for(i = 1; i < 6; i++)

        printf("The loop is running\n ");
```

The above code prints 5 lines, just like the example **while()** code. In a real sense both example loops accomplish the same thing. The difference is that the **for()** loop takes fewer lines to accomplish the same result. Another advantage of the **for()** loop applies when you have many lines of code inside the loop. In that case it is easier to see all the expressions controlling the loop because they are placed in the parentheses of **for()**.

Making Multiple Decisions – if statements

The **while()** and **for()** loops provide a means for making decisions. If the specified condition is TRUE, then the loop runs, otherwise, the loop does not run. There are only two decision choices, TRUE or FALSE. However, there is another way to make decisions that allows more than just two different outcomes. This is accomplished with **if()** statements. Consider the code example below:

```
if(a == 10)                    //i.e., if a is equal to 10

Put code here you want to run if a equals 10;

else

put code here you want to run if a does not equal 10;
```

The code above only makes one TRUE or FALSE decision, just like the loops. However, take a look at this example:

```
if(a == 10)

put code here you want to run if a equals 10;

else if(a == 9)

put code here you want to run if a equals 9;

else if(a == 8)

put code here you want to run if a equals 8;

else

put code here you want to run if a does not equal 10, 9 or 8;
```

You can test as many conditions as you wish by using more **else if()** lines. In this sequence only the first decision uses **if()**. It is not necessary to use **else** to finish up the sequence. If you don't want the program to do anything if the other conditions do not test to be TRUE, then you can just leave off the else. You can't use more than one else in the sequence and it must come last. Use none or one. All other decision tests must be **else if()**.

Let's consider another example, where variable **a** has already been assigned a value of 10:

```
if(a == 10)                    //i.e., if a is equal to 10

put code here you want to run if a equals 10;
```

```
else if(a > 9)                    //i.e., if a is greater than 9

put code here you want to run if a is greater than 9;

else if(a == 8)

put code here you want to run if a equals 8;
```

Now I will ask you a trick question. Which lines of code will run? The **if(a == 10)** is TRUE and **else if(a > 9)** is also TRUE. Will the code under each run? The answer is NO. In the **if – else if – else** sequence, only the code under the first TRUE condition will run. Since **if(a ==10)** comes before **else if(a < 9)**, the code under **if(a == 10)** is the only code that will run. After that the program escapes from the sequence and does not check any further to see if additional conditions might be TRUE.

Now let's take a look at a complete **if – else if – else** sequence.

```
int a = 10;

if(a == 10)

        printf("a equals 10");

else if(a == 9)

        printf("a equals 9");

else if(a == 8)

        printf("a equals 8");

else

        printf("a does not equal 10 or 9 or 8 ");
```

In the above code the computer will print this on its screen: "a equals 10." If we changed the value of variable **a** to 8 in the declaration (int a = 8;), then the computer would print "a equals 8." If we changed the value of **a** to 7, then the computer would display "a does not equal 10 or 9 or 8."

As the code is written above, the computer runs through the sequence only one time. In many cases we will want our program to run through the sequence more than one time. In those cases we can just put the code inside a **while()** loop like the code on the next page.

```
while(1)
{           //start of while() code block
int a = 10;
if(a == 10)
            printf("a equals 10");
else if(a == 9)
            printf("a equals 9");
else if(a == 8)
            printf("a equals 8");
else
            printf("a does not equal 10 or 9 or 8");
}           //end of while() code block, execution now returns to the while(1) line
```

The **while()** code above really does not make much sense. It would just cause the line "a equals 10" to repeatedly print out to the screen as long as the program runs. But suppose the value of the variable **a** changes as the program is running. That could happen if **a** is storing a value from a robot sensor, for example. Furthermore, suppose the code for each decision is not to print something, but to direct the robot to move in a certain fashion. Here is an example. Suppose we want a robot to follow a wall at a distance of 10 centimeters (cm). The code is below.

```
int a;                  //variable a stores the distance the robot is from the wall
while(1)                //loop runs indefinitely
{                       //start of while() code block
a = ping_cm(17);        //ping_cm(17) gets a distance measurement from an ultrasonic sensor
if(a == 10)
            put code here that makes robot drive straight forward;
else if(a < 10)
            put code here that makes robot steer away from wall;
else if(a > 10)
            put code here that makes robot steer toward the wall;
}                       //end of while() code block
```

One last thing about **if-else if-else**. Suppose you want to add more than one line of code under each decision point. Then you must enclose those lines in curly braces **{ }**. Like this:

```
if(a==10)

{

        put some code here;

        put another line of code here;

}

else

 {

        put code line here;

        another code line here;

}
```

Nested if() structures

There may be a need in your coding to create a **nested** structure of **while()** or **for()** loops or **if-else** structures. Nesting is a term used when you put a decision structure inside another decision structure. Here we will give the example of a nested if-else structure. You must take care in creating a nested structure or else (no pun intended) the program may fail due to a coding error. You must pay particular attention to the use of curly braces **{}** to nest one structure inside another. Examine the nested structure on the next page.

```
#include <stdio.h>
  int a = 3;
  int main()
  {                                   //start of main() function
    if (a < 5)                        //start of upper level if-else
    {                                 // enclose the nested if() in braces {}, this line is start of the nesting
          if (a == 4)                          //this if() is a nested if
        printf("a equals 4");
          else                                 //this else belongs to the second if(), the nested if()
          printf("a is less than 4");
    }                                 //this line is the end of the nest
    else                              //this else belongs to the first if()
    printf("a is greater than 4");
  }                                   //end of main() function
```

In the above code the text output will be “a is less than 4” if the variable **a** stores a value of 3. Let’s follow the sequence of events. Upon entering the **main()** function the program first encounters the line: **if (a < 5)** and since 3 is less than 5, the program goes directly to the nested if. In this second if, the argument is **a == 4**, which is asking if **a** is equal to 4. The answer is FALSE, so the program moves next to the else that is also nested with the second if and there we find the code that will display “a is less than 4.”

Notice that the code for the nested if() is indented. While indenting is not required, it is a good method of making the nested structure more noticeable to the human reader.