## Random thoughts about learning a programming language:

Learning a programming language is like learning a spoken language.  The more you practice, the better you get, and the fewer mistakes you make.

Programming languages are easier than spoken languages because there are many fewer words to learn.

Programming languages are harder than spoken languages because if you make a minor mistake –or even a major one --in a spoken language you may still be understood.  In a computer language even a small mistake can get you into trouble; sometimes BIG trouble.

Learning and working with a computer language may cause you to increase your usage of profanity in your spoken language.

## Random notes about the C language

/* comment  */   this is the standard method of entering a comment.  It may be on a single line or it may extend over many lines.

// this is a single line comment it ends at the end of the line.  (This non-standard and does not work on all compilers)

***

Just about all programming languages have some equivalent of the function. In other languages they may be called subroutines or procedures.   A function is a block of code which accomplishes some task. You may or may not pass one or more variables (called parameters) to the function.  The function may or may or may not return a value.  A function may be "called" more than once during the execution of a program.  One of the challenges of learning C is learning which functions are available, which libraries they are in, and how to use them.

## The print() and printf() functions:

print()  appears to be able to print text, but you can't embed formatting

printf means print formatted.  Embedded codes control formatting of variables.

printf("This is a printf statement %d\n", c);      // the %d indicates a decimal value is inserted here. The value inserted will be the value of the variable c.   – also the \n means go to a new line.

printf("This is a printf statement %9.2f \n", c);     //%9.2f means a floating point number is inserted here. The number will be formatted in 9 spaces with two spaces following the decimal point.  The value to be inserted will be the value of the variable c.

***

## Extremely simple simple program:

```
/*
Hello Message.c
Display a hello message in the serial terminal.
http://learn.parallax.com/propeller-c-start-simple/simple-hello-message
*/

#include "simpletools.h"                // Include simpletools header

int main()                              // main function
{
print("Hello World \n");                // print out a message
print("This is your computer speaking!");
}
```

Notes:
1. both kinds of comments
2. #include directive is needed for the print() function
3. main() function
4. use of { and }
5. use of \n
6. every command ends with a ;

***

HOME (1)

HOME character (1) sends SimpleIDE Terminal's cursor to top-left "home" position.

CRSRXY (2)

CRSRXY character (2) sends cursor to a certain number of spaces over (X) and returns (Y) down from SimpleIDE Terminal's top-left HOME position. This character has to be followed immediately by the X and Y values when transmitted to the SimpleIDE Terminal.

CRSRLF (3)

CRSRLF character (3) sends the SimpleIDE Terminal's cursor one column (space) to the left of its current position.

CRSRRT (4)

CRSRRT character (4) sends the SimpleIDE Terminal's cursor one column (space) to the right of its current position.

CRSRUP (5)

CRSRUP character (5) sends the SimpleIDE Terminal's cursor one row (carriage return) upward from its current position.

CRSRDN (6)

CRSRDN character (6) sends the SimpleIDE Terminal's cursor one row (carriage return) downward from its current position.

BEEP (7)

BEEP character (7) makes the system speaker in some computers beep when received by SimpleIDE Terminal.

BKSP (8)

BKSP character (8) sends the SimpleIDE Terminal's cursor one column (space) to the left of its current position and erases whatever character was there.

TAB (9)

TAB character (9) advances the cursor to the right by a tab's worth of spaces.

NL (10)

NL character (10) sends the SimpleIDE Terminal's cursor to the leftmost character in the next line down.

LF (10)

LF is same as NL.

CLREOL (11)

CLREOL character (11) erases all SimpleIDE Terminal characters to the right of the cursor.

CLRDN (12)

CLRDN character (12) erases all SimpleIDE Terminal characters below the cursor.

CR (13)

CR character (13) sends SimpleIDE Terminal's cursor one row downward.

CRSRX (14)

CRSRX character (14) positions SimpleIDE Terminal's cursor X characters from the its left edge.

CRSRY (15)

CRSRY character (15) sends SimpleIDE Terminal's cursor Y rows to the from its top edge.

CLS (16)

CLS character (16) clears SimpleIDE's screen, erasing all characters and placing the cursor in the top-left corner.

***

```
#include "simpletools.h"
int main(void)
{
int n;
print("Hello ");
pause(1000);
print(HOME);  // this is how you use the special codes
pause(1000);
print("Here is the second line");

return 0;
}
```

## Variable types in C

A variable is simply a name given to a memory location. The contents of that memory location may be changed while the program is running. The C language provides many types of variables. The four most common types of variables are listed in the table below.

| Type of variable | How variable x is declared | printf or scanf codes | Description |
|---|---|---|---|
| Character | char x | %c | A single character |
| String (of characters) | char x[] | %s | A group of characters |
| Integer | int x | %d | An integer number |
| Floating point | float x | %f | A number containing a decimal point |
| Double precision | double x | %lg | more digits of precision |
| Scientific notation | double x | %le | Scientific (e) notation |

Note: the "l" in lg and le is a lower case letter l and not the digit 1.

***

### A sample program:

```
/*
This complete program asks for two numbers, adds them, and prints out the sum
*/
#include "simpletools.h"
int main()                    // Main function
{
float a,b,c;
print("Please enter a number: ");
a = getFloat();
print("Please enter another: ");
b = getFloat();
c=a*b;
printf("%f times %f equals %f \n",a,b,c);
return 0;
}
```

*** Another program using scanf() rather than getFloat()

```
/**
* Accept and do arithmetic with floating point
*/
#include "simpletools.h"

int main(void)
{
double a,b;
pause(1000);
printf("Enter a number ");
scanf("%le",&a);
printf("You entered %le \n",a);
b=100*a;
printf("Answer is %le\n", b);
return 0;
}
```

This program fragment shows how character strings might be used:

```
   char str1[20], str2[30];
   printf("Enter name: ");
   scanf("%s", &str1);

   printf("Enter your website name: ");
   scanf("%s", &str2);

   printf("Entered Name: %s\n", str1);
   printf("Entered Website:%s\n", str2);
```

***

## Launching other cogs:

An amazing feature of the Propeller processor is the fact that it actually has 8 processors (cogs). Each cog can be working on a different function. Launching a function into another cog is likely to involve the use of global variables to communicate between/among processes.

Here is a fragment:

```
static volatile int t, n; // Declare Global vars for cogs to share before the main()
unsigned int stack[40 + 25];
// Stack vars for other cog minimum size= 40 longs plus some extra for variables
int main() // main function
{
int t = 50;
int n = 2;
```

```
// now launch adder function into another cog
cogstart(&adder, NULL, stack, sizeof(stack));
```

(the above is only a fragment, not a complete program)

Stack size: Be liberal with extra stack space for prototyping, and if in doubt, 40 to whatever value you calculate. I think the units here are bytes.

## Did You Know?

## Global Variables in Library Source Files

A global variable is accessible to all functions in every source file where it is declared. To avoid problems:

**Initialization** — if a global variable is declared in more than one source file in a library, it should be initialized in only one place or you will get a compiler error.

**Static** — use the static keyword to make a global variable visible only to functions within the same source file whenever possible. This removes any potential for conflict with variables of the same name in any other library source files or user application code. You can use static and volatile together.

**Volatile** — Use the volatile keyword for global variables that need to be used by functions running in different cogs. This keeps the C compiler's size optimizer from removing code that affects other functions' ability to read or write to that variable from another cog. You can use static and volatile together.

**Naming** — if a programmer happens to give a global variable in their application code the same name as a non-static global variable in a library, the names will conflict and give unexpected results. To help avoid this, name your variables in a libName_varName format. If you run into a mystery bug when writing an application, it is worth checking the documentation for the libraries you are using to see if you have a variable name conflict.

Static variable –one whose lifetime extends across the entire run of the program.

Volatile variable - the **volatile** keyword prevents the compiler from applying certain optimizations which it might have otherwise applied because ordinarily it is assumed variables cannot change value "on their own." In the Propeller setting, with multiple cogs working, a shared variable can be set by one cog and changed by another, so it needs to be volatile.

Another, easier method would be the following:

```
#include "simpletools.h"                   // Library include
void blink();                              // Forward declaration
int main()                                 // Main function
{
  cog_run(&blink, 10);        // Run blink in other cog
}  //end of main
void blink()                               // Blink function for other cog
{
  while(1)                                 // Endless loop for other cog
  {
    high(26);                              // P26 LED on
    pause(100);                            // ...for 0.1 seconds
    low(26);                               // P26 LED off
    pause(100);                            // ...for 0.1 seconds
  }  //end of infinite loop
}   //end of blink()
```

cog_run is designed to make launching application level functions (typically from the main file). All you have to do is pass a pointer to a function with no return value or parameters along with the number for extra memory to reserve. The value returned can be used to shut down the process and free up memory and a cog later by passing it to cog_end.

**Stack Size - how much?** 10 is the bare minimum value you would want to use for the **stackSize**. If you were to add more instructions to the blink function's code block, you would need to increase it. Add 1 for every local variable used, 2 for each function called, and 1 for each parameter and each return value used by the functions called. (I think the units here are longs.)

Returns: *coginfo Address of memory set aside for the cog. Make sure to save this value in a variable if you intend to stop the process later with cog_end or check which cog the process was launched into with cog_num.
***

## Libraries

Simpletools is for lots of stuff, just about everything

abdrive is for the activity bot servos and encoders

abcalibrate (obviously) is the calibration routine

servo.h for Parallax standard and Parallax continuous servos

***

## Tutorials: http://www.cprogramming.com/tutorial/c/lesson1.html

Printf() and scanf() tutorial: https://www.cs.utah.edu/~zachary/isp/tutorials/io/io.html

Printf( codes http://personal.ee.surrey.ac.uk/Personal/R.Bowden/C/printf.html

Or http://www.cdf.toronto.edu/~ajr/209/notes/printf.html

***

**Videos:** PLAN: assign movie for HW BEFORE the class covers the material.

Propeller Activity Board intro https://www.youtube.com/watch?v=xoYvCP2Ghs4 (2:47)

Why should I learn C? https://www.youtube.com/watch?v=37GJTKHn2ec (2:26)

Intro Programming in Propeller C https://www.youtube.com/watch?v=IL2WjB03eU8 (4:36)

How a compiler works (silent view of all stages) (1:29) Watch only the first video, not whole list.

https://www.youtube.com/watch?v=2dan4hJlOv0&list=PLDD1161DD92CA9F3E&index=1

Compiler vs Interpreter https://www.youtube.com/watch?v=kmQUB-5cEgM (3:36)