y: Alex Sink(HardCoded†)
Shouts to: ZiffDuck and Cerebus for proofing this, all my friends on Hotline, Logik, and Odin and PaiChan for helping to spread info about the zine.

Alright, now for you right brain thinkers, this will be perfect. We are going to go in some logical order(not HC's random thinking). Although I do suggest reading my other article to get a feel of the language.

Lesson 1-Hello World

What you need:

-An ANSI compliant compiler (opposed to K & R), and knowledge of how to basically use it(read the documentation that came with it)

-Ascii text editor-vi, wordpad, simpletext. (unless your compiler is an IDE, in which case it will be built in)

-Basic knowledge of your OS

Where to start:

The classic program, Hello World. You make this by creating a document, called helloworld.c, with your text editor and putting the following in the document:

```
#include <stdio.h>

int main(void)


printf("Hello World! \n");

return 0;
```

Now that you have that, save the file, and input the file(put it) into your compiler. Now you should have the program Helloworld! Run it and it should print on your screen:

Hello World!

and return you to where you were.

Explanation:

First off, in C, the number 0 means that the statement is false. If it is returned to your OS through the return statement, it exits the program and tells your OS that everything went well (a different meaning usage though). So, logically, 1 or 2 would evaluate to be true in a statement (in C, you will be forced to substitute and evaluate things many times), but if used with return, would exit the program and tell your OS that something went wrong.

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

This is a Preprocessor command that tells your compiler to add(include a copy of) the pre-written code, "stdio.h" to your file. The #include line causes the preprocessor to include a copy of "stdio.h" at this point in your code. The header file (that's what the *.h means-any header file) is provided by the C system. This file contains the code for printf(), along with other basic functions, so we added it.

```
int main(void)
```

This is the one and only function that your computer looks for by default, so everything is done through this statement. Since I did not specify the return value(what it returns to the higher function, in this case, the OS) of the function, it will automatically be type int (basic integer), meaning that I must return a value This is the heart and soul of your program. "A function," you ask? A function is just a set of commands that you can easily access without having to write them all over again. Also these functions take variables in and work with them; the way you input the variables is by putting them in-between the ()'s. More on functions in the next article.

This means that everything inside the brackets is treated like one line of code. This is important, because functions only take one line of code, so in almost all cases with functions, you will have to use brackets.

```
printf("Hello World");
```

This uses the standard printf() function found in stdio.h. This statement calls the printf() function. What you put inside the ()'s is what is worked on by the function. In this case, 'printf()' prints it's arguments directly on the screen. So the text Hello World is printed onto the screen. This is ended with a semicolon, like almost all statements in C.

```
return 0;
```

This exist the program, and returns 0 to your OS, telling it that the program exited well.

Just signals the end of the 's everything inside them is treated like one statement by the function main().

Lesson 2 - Variables

What's a variable?-

A variable is a place in memory for a number (you can say letter too, though it's not entirely correct). The programmer can name this anything that the system has not already named (some variables are already defined by your OS) and may use it however they wish. You assign the value to a variable after it has been declared by using this syntax:

```
vartype varname = Value;
```

The name of the variable is switched with the value of it is it's name when it is used in mathematical functions. This also happens when using a variable in statements. Variables are always switched when using them, except when you specifically specify otherwise.

There are seven basic types of variables in C: char, int, short, long, signed, unsigned, and float. Sometimes some of these are mixed to give you different output(e.g. unsigned long), but not all of them can be mixed.

Char:

This is a one byte variable. Normally it holds 256 values, which can represent any ascii character (any normal characters). This variable type is mostly used with characters (duh).

So this will hold either an 'e' or the value of 234. But alas, the letters are but numbers that correspond to letters. Each letter on the keyboard has a respective ascii equivalent -e.g. "0" is 48, "C" is 67. You can view the ascii code for a character by simply doing a printf("%d", charname); and comparing it to printf("%c",charname); This is also an integral (integer) type, so it can't be a fraction or a decimal. If it evaluates as such, it will chop off the fraction or decimal. This type of variable is declared by:

char varname;

Int:

This is a very common type and it is the base for many other types of variables. It is considered the natural or usual type of variable. It is small enough to be efficient, but large enough to not easily run over(if you have the highest value, and add one, you will get the lowest value, and it will start counting up from the lowest value). The exact size of this variable type varies from machine to machine. Remember that half of all of the variables are positive, and half of them are negative.

This variable is normally stored in 2 bytes(on 16 bit machines) or 4 bytes(32 bit machines). On 32 bit machine, this type stores 2 to the 32nd numbers in total, or about 4 billion values, 2 billion positive, 2 billion negative.

Int varname;

Short:

A variable type that is rarely used unless storage is really of concern. But if that is really the case, you most likely want to use a more advanced method of variable storage altogether. However, it is useful for making small variables that you don't want to get large, but then you have to make sure to do error checking so it does not, when given the highest value + 1, go from the highest value to the lowest value when you add one to the highest

value. And you can use unsigned shorts, which will eliminate that, but you will still go back to zero.

short varname;

Long:

Yet another take off of int. It is usually larger, but is not always. However, you can usually use it when you need a longer value.

long varname;

Signed:

Every variable is assumed to be of type signed. What does this mean? It just means that it represents both positive and negative values, and makes space for both. You say the same thing if you put signed before any variable type than if you just put the variable type. It is the assumed modifier of variables.

signed vartype varname;

Unsigned:

This type specifically tells your computer to ONLY hold positive values. This is very important if you never want your program to roll over to negative values, or just if you want to use a higher number when only using positive values. E.g, all 4 billion values in a 32 bit int will be set aside for positive values, and you will never get any negative values.

unsigned varname;

Float:

This is the outcast of the whole group. Many modifiers (short, long, etc) cannot be used on it. You cannot have a short float or an unsigned float. Only float, double or long double are allowed types of variables. These also hold decimal places, unlike the other variables. Also, because of this, it stores numbers in scientific notation too. You must be very careful when assigning values to variables of this type, for you must follow the conventions for scientific notation and floating point values that your compiler provides.

Your computer holds these values by using three basic parts: the integer value(whole numbers), the floating point value(the decimals), and the exponent parts. An example of this would be "123.45678e-22" where 123 is the integer part, the decimal point is... well the decimal point, 45678 is the floating point part, and e-22, being the e-xponent part, multiplies everything by 10 to the -22nd power(shifts the decimal point to the left 22 places)--this is scientific notation. For the most part, you must have all three types(this is a common usage), but you do not always have to have all three parts. You do not necessarily need to add the integer or exponent part to these(or you can just add one), but you are required to have the decimal point and a floating point value(".0" would be a valid float). Or if you need REALLY big numbers you can use just the integer and exponent parts("1000e38" would be valid).

How high can it count exactly?

It once again depends on your machine type and your compiler. However, normally, a float stores 6 decimal places, a double stores about 15 places, and a long double does not usually store more than a double. A float can usually hold values from -38 to 38 in the exponent part, while a double usually stores exponents from -308 to 308. But, this is very rough. Finding the exact values is a whole field of study, so just play around with the numbers until you get it right, but beware, often when storing the value, your computer will automatically convert the number to scientific notation, and then shift it over to the right one more place, making the integer value 0. Just don't depend on these being EXACTLY correct, because they won't always turn out that way.

```
float varname;

double varname;

long double varname;
```

P.S.-see the attached file for a program that will find out how much you can put in your variables (I left out floats though.).

HUH?
Ok, if you're looking at the screen kinda funny, it's understandable. I'm going to assign values to variables while creating them (yes, you can do this too!).

```
int ProgDi = 1;

short HC = 1;

long Logik = 1;
```

```
signed int sInt = 1;

signed sInt2 =1;

unsigned int uInt = 500;

char charVal = 'O';

float fex = 1.03e22;

double dex = 3.3333333e307;
```

This would not make a complete program , but if placed inside the main() function, and you add return 0; at the end, it would make a program. This is what is usually referred to as psudocode. This will not print anything (I explain how to do that next), though.

Ok, I want to print this stuff!

First of all, you must remember that all these types of variables just set apart different amounts of space for 1's and 0's. You may read(and therefore print) this data any way that you wish, but usually you want to print it out by the correct type. However, sometimes you may want to print it out in another form. You may want to find the ascii value of a letter by printing out it in number form for example. Before I get started, an example:

```
printf("%d", ProgDi);
```

The printf function takes two values in this case: the control string(the first part, "%d"- which contains the conversion character for decimal integers), a comma to separate, and other arguments(a variable in this case: ProgDi). If you want to print more than one value, you would do it like this:

```
printf("%d %d", ProgDi, ProgDi);
```

Just separate the variables with a comma.

Now that you have seen me print a number in decimal form, how would you print other formats, like characters? Well, you use "printf()"'s conversion characters (the letter that is directly after the "%" character) and what they display the variable as:

c-will display it as a character

d-decimal integer(you've seen this before)

e-floating point number in scientific notation

f- as a floating point number

g- in e or f format, whatever is shorter

s-as a string(characters)

Usage of these:

printf("%(control character) ..." , varname);

All the following information is rather trivial, and you will mainly need it for reference:

One other major point in using these variables is that some people(you always have to read other people's code) like to use field widths. These are placed directly after the "%" character, but directly before the conversion character. This width is normally(by default) set to one. However, if you set it to a different value, you can get extra spacing between variables. Some people find this very convenient, but it can also get quite confusing after a while. It is just important to understand how this works.

Basic usage:

printf("%(field width)(control character)", varname);

Also, when using "printf()" or "scanf()", remember that there are special keys that mean certain things when used after the escape character (not escape key), "\". The escape character is used to escape the usual meaning of the character that follows it. Like the "\n" character that I have been using. It doesn't print out "\n" that is because it is a special, escape character, which prints a newline it is often referred to as the newline character, for it is considered one character. Here are the special characters in a list:

\a- alert- causes the bell in your computer to chime

\\-backslash- prints a backslash in printf's(or anything like printf()'s) text.

\b-backspace-prints a backspace character and moves the cursor back

\r-carrage return...literally...not just any way of making a new line

\"-double quote-prints double quotes in a printf() or similar statement

\t-horizontal tab-this tabs the text

\n-newline-your buddy. It prints a new line.

\0-null character-represents absolute nothingness, or the end of a string (more on that next issue)

\'-single quote- allows a single quote inside a printf() or similar statement

\v-vertical tab-tab down

\?-question mark...could be useful sometimes (not in printf()'s though...not sure when...)

Example:

```
/* a quick program to show the use of these odd characters */
#include <stdio.h>

int main(void)

char testchar1 = 'b', testchar2 = 'B', testchar3 = '4';
/* printf(" "the fundamentals of printf" ");   does not work...too many "s this line is
commented out here */

printf(" \"The fundamentals of printf(): \" \v \n");
```

```
printf("\? wtf does this do?   \a \a \a \n");

printf("with 1 as field width testchar1 :%1c What it looks like normally:%c \n", testchar1,
testchar1);

printf("with 1 vs 10 field width:%c-vs:%10c-\n", testchar1, testchar1);

printf("prints ascii values of testchars:%d%5d%5d\n", testchar1,testchar2,testchar3);

printf("look spacing of 1,2,and 5:%c%2c%5c", testchar1, testchar2,testchar3);

return 0;
```

What if I want to get basic input?

You have to use a function called scanf()
you use it in a kind of like you use printf(), but of course, it will have a different effect. Here is the basic usage:

```
scanf( "%(scanf control character) ), &varname );
```

It is very similar to printf(), except of course, you get a value put in the variable instead of printing it, and you have to use scanf()'s conversion characters. One main difference, the "&" right before the variable name. Why do you have that? It is the address operator. I will explain this in the next article in detail, but you just need to know that you must put this directly in front of your variable name, or nothing will be done to the variable.

The scanf() conversion characters:

c- character

d- decimal integer

s- string /*we will get into strings next issue */

f- floating point number(float)

lf- floating point number(double)

When you use these characters, the input will be stored in the variable as the type that the control character specifies, so be careful when putting these; you don't want to be using %c to put a value in an integer type variable.

Example:

```
/*   be careful when you use this program!   There is no way to verify that you have entered the correct type of variable(no error checking), so if you give it the wrong stuff, it will give you odd answers
*/
#include<stdio.h>

int main(void)


/* declaring variables */


int decimalint=0; /* it is good coding practice to set all your variables to zero, they make take on different values otherwise */

char charscan = ' '; /*setting it equal to a space */

float floater = 0.0; /* same principal once again */

double doubler = 0.0;

/* work */

printf("now scanning in your text for: an integer, a character, a float, and then a double all seperated by the /n or enter key");

scanf("%d\n%c\n%f\n%lf", &decimalint,&charscan,&floater,&doubler);

printf("%d - %c - %f - %lf: is what you put in.",decimalint,charscan,floater,doubler);

return 0;
```

Lesson 3 Controlling your program:

Before, your program ran in a very linear, understandable way. Then, we get into changing flow of your program so that it does not run in a line, but allows you to create a program that can have many possible outputs. This is of course a very fundamental part of programming, and it often causes quite a few headaches, for you must envision every possible case. But this is what makes any program interesting.

Keep in mind when programming these that they are eventually broken into one of two situations, 1 or 0, true or false. This is what you need to work out in your head what the statements will work out to: true or false.

The "If()" statement:

When you run the "if()" function, it evaluates what is in the parenthesis. If it evaluates to be true(non-zero), then the statement block (the stuff in the brackets afterward) is executed once. If the value in the parenthesis is 0, or false, the statement block is not executed. You may also have many if() statements one after another, or nested inside one another (an "if()" inside an "if()")

Basic usage:

```
if( varname   or number )
```

```
if(varname or number)/* an example of if nesting(putting an if inside an if) */
```

```
yourcommands;
```

Also, when using if(), you may also want something to happen if the if() evaluates as anything else. The answer to this is the else() function. If the if() evaluates as false, then the else() evaluates as true and executes.

Basic Usage:

```
If( varname or number )

foo;

else

bar;
```

Example:

```
#include <stdio.h>

int main(void)

int vtrue = 1;

int vfalse = 0;
```

```
if(vtrue)



printf("vtrue runs in this case! \n");



else



printf("vtrue isn't evaluated as true in this case \n");



/* or if you want to be lazy... this is the same as the first if, but it only works when you have
only one statement to execute if the if() or else() is evaluated as true. */

if(vtrue)

printf("lazy way of vtrue works\n");

else

printf("lazy way of vtrue doesn't work\n");
```

```
if(1)

printf(" 1 works for this...\n");

else

printf(" this shouldn't work \n");

if(vfalse)

printf("this shouldn't work\n");

else

printf("vfalse is false\n");

if(0)

printf("this shouldn't work\n");

else

printf("0 is false\n");
```

```
if(0   || vfalse)   /* if 0 or vfalse is true, run this -see the end of the chapter for more of this sort of thing*/


printf("this shouldn't work\n");


else


printf("niether 0 or vfalse evaluate as true\n");


return 0;
```

The While() statement

The while() statement (or loop-whatever) is very similar to if(). It executes only if the statement inside the parenthesis is true (non-zero). However, the main thing that separates it from if() is that it keeps executing the code in the statement block (the stuff in between the " "s) until the statement evaluates as false. So you must have some way of making the statement false sometime, so be sure to have a way of doing just that when using these.

Basic Usage:

```
while( statement )


statements;
```

You may also reverse these statements in a strange, yet very useful, way. It is the Do ... while() loop. This is very similar, but the code in the while's statement block is executed with the do statement, and if the while() statement evaluates as true, the whole thing repeats itself, like a normal while statement.

Example:

```
#include<stdio.h>

int main(void)


int i=0,n=5;

while(i++ < n)   /* the ++ means i = i +1, or add one to i...commonly called the auto-incrementer */


printf("value of i:%d \n", i);

do




printf("You should only see this once, because the while() is false\n");



while(i++ < n);

return 0;
```

For() statement

This is basically an extension of if(). It only executes if the statement that is being evaluated is true (any non-zero number). However, it has a control feature that allows you to run the statement block a certain numbers of times.

```
for( i = 0; i < 500; i++ ) /* it goes initialize counter; evalstatement, increment/decrement */




statements;
```

What does this do exactly?

Break it down (the way you work with all logic):

for()

Just the function call describing what to do with the information inside the brackets.

i =0;

This created the variable i. It is assumed that the i is an integer, when you use it in a for() statement. The first ";" space is used for declaring integers that will be used for counting the number of loops. This number will be used in the second part of the for() statement usually, and usually it is incremented(added to) in the third part.

i < 500;

A basic mathematical statement. You should all know about inequalities from math class. this is the part of the for() statement that is evaluated. If this middle part is true, then the statement executes. This is like what goes in the parentheses in the if() statements.

i++

WHAT?

This is incrementing (adding to) the variable by one.

The statements:

i = i + 1;

i++;

Are exactly the same, except the second one is easier to read, and to write. (programmers are lazy) I'll talk about using this operator next time in more detail. (I am lazy). But all you have to know is that i++ adds one to i, whatever i(the variable) may be.

Also, a very important feature of the for() statement is when you use it with nothing being evaluated like this:

```
for(;;;)


Statements;
```

This keeps on executing what is inside of the for statement, until it finds a break, return 0 or an ExitToShell. This is very useful when you are making a game, and you want to keep repeating the same algorithm.

Example:

```
#include<stdio.h>

main()


/* this is what you would have to do with an if() statement */


int i =0;

if(i <500)




printf("I love C! \n");


i++;




/* with a for() statement */


for( i =0; i < 500; i++)
```

```
printf("I love C! \n");



/* an infinite loop terminated by a break(exits any loop) */

for(;;;)

printf("you are inside an infinite loop.   If I didn't call the break;   this would not exit.\n");

break;


return 0;
```

The for() loop can be a real convenience.

The Switch: statement

Switch compares one variable against certain stated conditions, or "case s".   It's like using many if()'s that are evaluating if a lot of variables are equal to one thing.

Example:

```
#include<stdio.h>

main()


char answer = ' ';

printf("do you like cookies?   y/n?");

scanf("%c", &answer);
```

```
switch(answer)


case 'y':

printf( "cookies are good!   :-) " );

break;

case 'n':

printf( "too bad :-(" );

break;

default:

printf( "I don't understand you" );


return 0;
```

This can be rather confusing at first, but it is really useful. Basically, you put the variable into the switch(xxx). Then you make instances for each possible case, and make sure that if none of the cases are evaluated are true, then there is a default action. You can have as many cases as you want.

So, the variable goes to the switch, and the switch looks for all of the case's and sees if any of them are equal to what was put in the switch() parentheses. Then it keeps on executing code, even if it goes into another case statement, until it sees a break. Then it exits the

switch(). Not only does it do this, but the switch() statement, because it makes you provide a default: switch, it provides error checking, which can be a real pain to create using other methods.

Ways to evaluate multiple statements in one go:

I will discuss this in more detail in the next issue, when discussing operators in more detail. You can put these in between two statements(inside the same parenthesis) to evaluate two values at the same time.

Usage:

statement1 && statement2 -the logical AND operator. Evaluates as true only when both values are true

statement1 || statement2 - the logical OR. If statement 1 OR 2 is true, then the whole statement is true.

!statement - the opposite of the statement. If it it true, then it evaluates as false and visa-versa

statement1 == statement2 - this evaluates as true if the two statements are equal. Be careful of doing the following statement instead.

statement1 = statement2- This is the ASSIGNMENT operator! You just assigned the value in statement2 to what is in statement1. Not quite what you meant...

statement1 != statement2 - you may use the "=" like this, when you have another operator there, like in this statement. It will be true if the two variables are not equal

statement1 >= statement2 - Will be true if statement1 is less than or equal to statement2, you can use inequalities like this, or you can use them by themselves.

Summary:

These change the way your program runs. With these commands, you will be able to write a simple text based game(and those can be great fun). The if() statements only executes it's code if the value inside the parenthesis is true(non-zero). The for() statement does three things before it executes it's code. It performs the initialization of variables, it evaluates the statements inside the 2nd part executing only if this value is true, and then it performs some sort of operation on the variable(usually incrementing).

References that I used for this article:

-Brain(very useful)

-A Book on C fourth edition by Al Kelley and Ira Pohl -- great book. More useful than all the other books that I have read by far. Copyright 1998 by Addison Wesley Longman, Inc.
ISBN: 0-201-18399-4

Other References

-Hoganbooks.com(online books!!!)