C from first principles

This is a C programming tutorial for those who know nothing about programming but want to understand fully the reasons why things are done, not just rote learning. Please feel free to update sections that aren't clear. This is a work in progress.
– *fluke*

Getting set up

This section needs expanding
For this tutorial we will use GCC and the standard Unix command line tools.

- Windows: Install MinGW and MSYS
- Mac OS X: Install XCode
- Linux: You should already have gcc installed

A brief history of C

C was created by Denis Ritchie at Bell Labs (then part of AT&T) in 1972/73 for porting the Unix operating system to the PDP-11 (an early minicomputer). C became popular at universities which used Unix due it coming with the complete source code in C.

For more information, see:

- http://en.wikipedia.org/wiki/C_(programming_language)#History
- http://en.wikipedia.org/wiki/Unix
- http://en.wikipedia.org/wiki/BSD

How C gets from your brain to your computer

Computers understand only machine code, a series of numbers encoding the instructions to execute. For instance, if you wanted to load the value 42 into memory address 3 on a PIC 18F series microcontroller, you'd give it these instructions:

```
0000 1110 0010 1010
0110 1110 0000 00011
```

Not very easy to understand, is it? If we associated each instruction with a short name (mnemonic) then it would be easier to write down:

```
        MOVLW    42
        MOVWF    3
```

But we'd still have to translate it by hand before giving it to the microcontroller. Even better would be

if we had a computer program to do the translation. This sort of program is known as an assembler and the instructions we give it known as assembly language. We could also define symbols to refer to the addresses and constants in the program so we wouldn't have to remember what was in each numeric address:

```
answer  EQU     3

        MOVLW   42
        MOVWF   answer
```

Taking the idea an step further, we could enhance the assembler program so that it took a higher level language. One where we give it a more abstract input, one that didn't depend on the target microcontroller. This way the same program could be used for more than one type of microcontroller, or even desktop computers as well. This sort of program is known as a compiler, it takes a high level language, translates it into assembly language and then calls the assembler to generate machine code. So our program would simply be:

```
    int answer = 32;
```

Working with the Unix command line

The usual way of interacting with a Unix system is via the command line, also known as the shell. Commands are given as a name of a program (or internal command of the shell), then a series of arguments. Common commands are:

- `ls [dir name]` List files in given directory (or current directory if no directory given).
- `cp [source files …] [destination]` Copy the source file(s) to the destination.
- `mv [source files …] [destination]` Move the source file(s) to the destination. Also used for renaming files.
- `gcc` (run the GNU C compiler, more details given later.)
- *./program name* runs a program in the current directory. You have to give the path as the current directory is not usually searched when looking for commands.

Unlike Windows, a Unix shell expands wildcards such as * into multiple filenames itself. So the program doesn't know if you used wildcards or not. Be careful of leaving off the destination argument to `cp` or `mv`. If you said

```
mv *.c
```

and there were only 2 files matching `*.c`, then the first file would overwrite the second.

Lesson 1: Your first C program

To start with, open up a text editor (such as notepad on Windows or nano on Linux) and type this in:

```
#include <stdio.h>

```

```
int main(int argc, char **argv)
{
  printf("hello, world\n");
  return 0;
}
```

Save the file as `hello.c` in your home directory.

Now let's examine each line to find out what it does:

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

Tells the preprocessor include the file `stdio.h`. This has the definition for the `printf` function and other functions for input or output. The angle brackets around the name tell it to look in the standard system include paths. If we had used double quotes around the name, it would have looked in the same directory as the C source file instead.

```

```

A blank line. You can have these anywhere you want, whitespace between tokens is ignored.

```
int main(int argc, char **argv)
```

Declares a function named `main`. This is the function called by the operating system when it starts up (though the compiler may insert setup code before it gets to `main`). The function returns an `int` and takes 2 paramters, an `int` named `argc` (which holds the count of the command line arguments) and a pointer to a pointer to a `char` named `argv` (the text of the command line arguments). Pointers and how they relate to strings of charaters will be explained later.

```
{
```

An open brace starts the body of the function.

```
  printf("hello, world\n");
```

Print `hello, world` and then a newline to the standard output device (usually the screen). Statements are terminated by semi-colons.

```
  return 0;
```

Return from the function with 0 as the result. A non-zero result would signal to the operating system that this program had an error.

```
}
```

A close brace ends the function.

Now that we know what it does, let's compile it. Open up a shell, change into your home directory if you're not there already and run this command:

```
gcc -Wall -g -o hello hello.c
```

This runs the gcc program to compile your source file into something your computer can understand. The options to gcc are:

- `-Wall` tells gcc to print all warnings it has about your program (such as mixing types, which is legal in C but possibly a mistake)
- `-g` includes debugging information in the executable
- `-o hello` names the output file `hello`. If you don't specify this, it will be named something like `a.out`
- `hello.c` the source file

If gcc prints any errors or warnings, check that you typed the program in correctly and saved it in the right place and try compiling again. Once you have no errors, run the program with

```
./hello
```

and you should see it print

```
hello world
```

Lesson 2: Scalar Types and printf

Every piece of data in C has a “type”. This lesson deals only with scalars, which is a fancy way of saying a data point with one value, as opposed to arrays and structures which have multiple values.

Numbers in C can be either integer (having no fractional part) or floating point (having a fractional part and a wider range, but a limited precision).