1 History of C

Early programming was done in assembly. Assembly is writing directly in machine code (numbers that the computer understands as commands), but assembly use words in place of numbers to make it easier for the programmer to read and write. There were a few early languages developed, but they all had problems in both design and implementation. Something better was needed.

C was developed at Bell Laboratories in 1972 by Dennis Ritchie. C originated from the B programming language with the purpose of creating a language that was "high level", machine independent, and still able to access the machines low level systems. Both C and Unix were written in assembly, but both were quickly re-written into C.

C compilers spread throughout many universities in the US and the rest of the world. This caused major compatibility problems (start of the compiler wars) and in 1983, the American National Standards Institute (ANSI) formed a committee with the purpose of standardizing C. The result is commonly known as ANSI C.

For more history

http://cm.bell-labs.com/cm/cs/who/dmr/chist.html

2 I know Java, so why is C important?

One of the biggest reasons anyone should learn C is that it is ubiquitous. The C language is the most widely used programming language, thus large amounts of code are already written in C. As a programmer your job maybe to maintain (update, debug, add features) to existing code. Many older programs need to be rewritten in modern programming languages and being able to read a C program can help speed up the process. Their are also many programs that simply cannot be upgraded to a modern language (operating system kernels).

As hard as C's creator tried, C is not a high level language. Most computer scientist today would call C a low level or mid level language. This makes the language more challenging to write in, but means you can write any program in C. The low level to C makes it fast. Most languages have a hard time competing with the speed of C programs. C is also the best choice for writing hardware level code such as device drivers and microprocessors found in everyday machines and appliances.

C is the common ancestor to the vast majority of programming languages available today. Having a strong understanding of C will give you a better understanding of how other high level languages work. Some languages are built on top of C, these include C++, objective C and D. Many of high level languages can use C libraries to extend their functionality.

Last but no least knowing C can help you get a job.

For more information

http://www.cprogramming.com/whyc.html

3 Hello World

There is some crazy convention in the world of computers that states: The first program you present to a student **must be** "hello world".

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main( int argc, char **argv )
5 {
6     printf( "Hello World!\n" ); /* English */
7
8     printf( "Hallo Welt!\n" ); /* German */
9
```

```
10      printf( "Bonjour Tout le Monde!\n" ); /* French */
11
12      printf( "Hola Mundo!\n" ); /* Spanish */
13
14      printf( "Ciao Mondo!\n" ); /*Italian */
15
16      return( 0 );
17  }
```

The output of this program is:

```
Hello World!
Hallo Welt!
Bonjour Tout le Monde!
Hola Mundo!
Ciao Mondo!
```

Lets break this down

Include Macro

On lines 1 and 2 we have what is called a macro. Macros are not C code, but code for the compiler. In this case we are using the **include** macro. Include tells the compiler to bring in code from another file, this is similar to the import statement in Java. The angled brackets tell the compiler to look in the compilers include paths (this will be discussed later in the course). In the example below the include call is using quotes. This tells the compiler to look for the file using the current files path and the base. The example also shows how we can use directory notion to point to a file in a different directory.

```
1  #include "my_code.h"
2  #include "../network_stuff/ftp.h"
```

The file **stdlib** is the standard C library. This contains lots of useful code for programming in C. The file **stdio** is the standard input and output library. This is useful for getting and sending information. This includes the console (or screen in laymen's terms) and files.

Main Function

On line 4 in the hello world program we have what is called a **function definition**. This line defines a function in C (more on functions later). The function definition is as follows:

```
[Return Type]  [Function Name] ( [ Parameter list ] )
{
[function body]
}
```

This tells the C compiler that there is a function called main that returns an integer, and it takes an integer that we will call argc and a char ** call argv. A char** is an array of of strings, more bout this in another section.

Functions in C are like functions in mathematics. They have both an input and an output. We can think of our main function like so: $x = F(y, z)$. F takes two values called y and z, and returns a value called x. Functions do not have to return a value in C.

All C programs must have a main function. The main function is called by the operating system to start the program. The function definition on line 4 in the hello world demo is the correct and proper definition for a C main function. Some compilers will accept other variations, but you will be marked down if you do not define your main function as such.

Return Type

In C functions can only return one value, or nothing at all. The possible return types are any of the built-in types, and structures. Built in types will be discussed in a later section.

Function Name

Function names must be unique, start with a letter, and can only contain letters, numbers and underscores. Other then these rules you can name functions anything. When programming you should use meaningful names of functions.

Parameter List

The parameter list contains zero or more parameters that a function takes as input. A parameter can be any of the built-in types, arrays, and structures. Functions can also take a variable number of parameters using ellipses (...).

Function Body

The function body is the work that the function does. The function body is contained inside of a block. In C their are commands (macros, function definitions), statements, and blocks.

Blocks

Blocks are simply containers that can hold command, statements, other blocks, or even nothing. You start a block by using a left curly brace **{** and close on by using the right curly brace **}**. Commonly you use tabs to offset the contents of a block for readability.

Statements

Statements are a single C instruction that ends in a semicolon **;**. Instructions do not have to be on one line. C ignores white space (except for the need to put spaces between words). You can also have compound instructions.

Printf

Inside the main function we have five printf instructions. Printf is a function the is inside stdio.h. This function is used to print to the console. On these five lines the program prints hello world in five languages. The firsts parameter for printf is always a string. All strings start and end with a quotation mark. White space is not ignored in a string, and they are single lined. Printf will print the strings contents to the console for us. These instructions **call** the printf function. The instruction ends with a semicolon.

Comments

After the printf instruction we have what is called a comment. Comments are ignore by compilers and only exist for programmers to leave notes or comments in code. The C language only has one style of comments, but most compilers can access two style of comments. For this class both styles can be used.

Multi-line comments

The first style is the multi-line comment. This is the style we see in the hello world program. Multi-line comments start with a a slash star (/*)and end with a star slash (*/). Multi-line comments cannot be inside another multi-line comment. Example:

```
1 /*
2     Author:   [Name]
3     Created: [Date File Created]
4     File Name: [File Name]
5     Description:
6             [Description of what this file contains]
7
8     Notes:
9             [Developmental Notes]
```

```
10
11      Bugs:
12              [List of bugs and their status ie.. FIXED, OPEN, CLOSED.. ]
13
14  */
15
16  /******************
17   *  The compiler only cares about the /* everything inside is ignored!
18   *
19   *****************/
20
21   /* this is a one line usage of a multi-line comment */
```

Single Line Comments

The second commenting style is a single line comment. A single line comment starts with two slashes and ends on a new line. You put these at the end of instructions. Older compilers may throw errors when seeing single line comments, make sure you understand you know what your compiler can support. Single line comments can be nested in multi-line comments. Example:

```
1  // Single comment
2
3  printf( "Hello World!\n" );  // English
4  /*
5      // single line in a multi=line!
6
7      inside the multi-line but not a single!
8
9      // another single!
10 */
```

Return Statement

Following the five printf statements on line 16 in the hello world example is the **return** statement. The return statements is used to exit a function, and if the function have a return type, it returns the return value. A return statement can appear anywhere in the function, functions that do not return a value do need to have a return, and functions can have several return statements.

Returning from the main function exits the program. The return value in main is used to indicate to the operating system if the program ran successfully. A return of zero means success, non-zero means an error occurred.

4 Compiling

Compiling is the process of turning one thing into another. In order to make our C code into a program we need to compile our code into something the computer understands. We do this with a compiler. This document will use GCC on OS X 10.6.8. To compile the hello world program we do the following:

- Writing your program in a text file and save it to disk.
- Open up the terminal and change into the directory where your file is saved.
- Compile the program with the following line:

  ```
  gcc hello.c -o hello
  ```

- Run the program:

```
./hello
```

If you entered the program correctly you should get the same output as listed in section 3.

Compiling Many Files

When writing a program in C, you do not have to put all of the program code in one file. A program can (and normally does) contain many files. Large projects can easily have hundreds or even thousands of files. To compile multiple, simply list all of the files you want compiled into the program. This works well for small programs with only a few files, but large projects become a problem.

The first problem is keeping track of all of the files that need to be compiled. Many developers solved this problem by writing scripts, or small programs to compile the project for them. When new files are added to the project, all the program has to do is update the build script to include the new files.

The second problem that large projects have is compile time. Compiling the hello world program only takes a few seconds, but a large program can take hours to compile. Early programmers realised that when working with large projects, only a few files changed in between compiles. In order to reduce the time it took to compiler there projects they would only compile the files that changed. C makes this easy to do. This is done with linking and libraries.

Linking

C files are compiled individually. Each C file gets compiled into what is called an **object file**. An object file contains all of the compiled code for a single C file, and the information needed to link file. Linking is the process of combining object files and libraries into a programs binary file. This process is much simpler and a lot faster then compiling. GCC will call the linker for us.

Libraries

Libraries are a collection of object files into a single file. Libraries contain commonly used functions and **data structures** (more about this in another section). The purpose of a library is to give developers access to quality reusable code. The hello world program used the standard C library so it can print to the console. Libraries reduce development time and make the life of a programmer that much easier.

Using GCC to make Object Files

Using GCC to make object files is simple. The following command will compile hello.c into an object file, and then link the object file into the hello binary file.

```
gcc hello.c -c
gcc hello.o -o hello
```

The -c option tells GCC to only compile the code, skipping the linker. By default GCC calls the object file the same as the c file, except it changes the extension from c to o. The next step is to link the object file into a binary. The is done by calling GCC again but we tell it to use the object file. The -o option tells GCC what we want to call the output files name.

Build Tools

In order to make to take advantage of linking and libraries many programmers use build tools. The primary build tool using in C development are **makefiles**. Makefiles are a script used by the program make to compile large C programs (make can be used for almost any type of scripting not just C). Writing makefiles can be a tedious and difficult task to get right, to make things easier we will use **cmake** to generate a makefile. CMake will be covered in another section, for now use GCC from the command line.

5 Variables

"A variable is a symbolic name given to some known or unknown quantity or information, for the purpose of allowing the name to be used independently of the information it represents." - Wikipedia

The information is contained in a computers memory. To access this the computer needs to know where it is in memory. Memory is separated into **bytes** (blocks of 8-bits), and each byte is given a numerical address. A variable is a symbolic name given to that memory address.

Variables in C must start with a letter and can only contain letters, numbers, and underscores. Variables in C have another important piece of information attached to them, **data type**. The data types tells the compiler how big the variable in bytes is, and how to use the variable.

C Data Types

In C there are only four basic data types:

- **char** A single byte of data. This is an integer type (1, 2, 3).
- **int** The standard integer type size. Normally 4 bytes.
- **float** Single-precision floating point value (-12.34, 3.14).
- **double** Double-precision floating point value. Two floats put together.

Various **qualifiers** can be added to these basic types.

Creating a variable can be created by giving its data type, the variable name and a semicolon. Multiple variable of the same type can be listed together by using a coma to separate them. By default variables are **not** initialized. This means they have whatever value was already memory where they are located. This can lead to what are know as **bugs**. To prevent this variables should be initialized to a known value. This is done by a equals sign and a value after the varible name. Example:

```
1 int age;  /* This could be set to any value */
2 float pi = 3.14f;
3 double speed_of_light = 299792458.0;
4 char day = 23, month = 2, year = 1984;  /* three variables on one line */
```

Numeric Values

In the previous example several variables were initialized to numeric values. There are three numeric value types integers, single-precision, and double-precision. Integer values are positive or negative whole numbers(-2, 0, 34). Integers can also be expressed in octal and hex. Octal numbers start with a zero (0776, 0124). Hex numbers start with a 0x (0xFFEE00, 0x01).

Single-precision numbers are positive or negative numbers that include a decimal point and end with the letter f (3.14f, -0.0f). Double-precision numbers are like single-precision numbers except they **do not** end with the letter f (123124.12423423, -123.43). Having a difference between single and double-precision numbers allows the compiler to optimize their storage.

Signed and Unsigned Integer Types

By default char and int are signed. This means that they can represent negative numbers. In order to represent negative numbers, the most significant bit is used as the sign bit. This means signed numbers have less bits available and have a smaller max and min. Both char and int can be set to signed or unsigned with a qualifier. This is done by adding the word signed or unsigned before the data type. Floating point numbers are always signed.

Type	Range
signed char	-127 to 127
unsigned char	0 to 255
signed int	2,147,483,648 to 2,147,483,647
unsigned int	0 to 4,294,967,295

Int Size Qualifier

In order to get integers of other sizes, a size qualifier is used. The three size qualifiers are **short**, **long** and **long long**. When using a size qualifier, int is implied and can be omitted. The two following declarations are equivalent.

```
1 unsigned short int numTickets;
2 unsigned short numTickets;
```

A **short int** is two bytes, or 16-bits. A **long long int** eight bytes or 64-bits. There is a problem when it comes to a **long int**. The size of a long int depends on the machine, on some machines its 4 bytes, and other its 8 bytes. Int size types can be problematic but the C library solves this problem with the file inttypes.h.

inttypes.h

The file **inttypes.h** gives a program access to set of integer types with a known size. You will be required to use this library in all assigned projects. Inttypes have eight data types for integers that you will use.

Type	Sign	Bits	Bytes	Minimum	Maximum
int8_t	Signed	8	1	-128	127
uint8_t	Unsigned	8	1	0	255
int16_t	Signed	16	2	32,768	32,767
uint16_t	Unsigned	16	2	0	65,535
int32_t	Signed	32	4	2,147,483,648	2,147,483,647
uint32_t	Unsigned	32	4	0	4,294,967,295
int64_t	Signed	64	8	9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	Unsigned	64	8	0	18,446,744,073,709,551,615

To use these types just include inttypes.h

```
1 #include <inttypes.h>
```

Prior to 2010, Microsoft Visual Studio did not follow the C standard, and did not include inttypes.h and stdint.h. If your compiler does not have these files try this project:

`http://code.google.com/p/msinttypes/`

Constants

Constants are variables that cannot be changed. The compiler enforces that these variables are read only. Constants are created by adding the const qualifier to the variables declaration. Constants need to be initialized when they are declared.

```
1 const float Pi = 3.14f;
```

Where to Declare Variables

Variables in C can only be declared globally (outside of any functions), or at the top of a functions. Variables must be declared before any code. Example:

```
1
2 int GlobalVariable = 0; // This is correct!
3
4 int main( int argc, char **argv )
5 {
6     int LocalVariable = 1; // This is correct!
7     char Local2, Local3; // Also correct
8
9     printf( "HI!" );
```

```
10
11      float BadVariable; // This is wrong, should before the printf
12
13      return( 0 );
14  }
```

More Information on C Data Types

`http://en.wikipedia.org/wiki/C_variable_types_and_declarations`