Tutorial 1: C Revision

Each tute will typically consist of six sections:

- General review questions/discussion
- New programming concepts
- A section on compilation
- A section on compiler/general programming errors
- A section on debugging
- Exercises

Review: Basics

What is the difference between a function declaration and definition?

A function (or variable, or struct) declaration simply tells the compiler that there is something that exists with a given type, and a given name, ie:

```
int funcThatReturnsOne();
```

tells the compiler that a function exists with the name func and it returns type int. The compiler will not always need to know everything about this function in order to use it, this means that you can write code that the compiler will understand without needing to give it all the details.

By contrast, a definition will tell the compiler all the details about the function (or struct). In this case it would mean providing a function body, ie:

```
int funcThatReturnsOne() { return 1; }
```

What typically goes into a header and source file and why?

A header file has the extension .h and typically (but not always) contains only declarations, while source files typically contain definitions. This is not always the case, but it has a few advantages:

Code-bloat: C (and C++) follow the One Definition Rule, whereby a definition can only appear in one compilation unit (.o file). The compiler will simply copy-paste header to wherever they are needed, if you put definitions into the header, they will appear in multiple .o files, increasing the size of compilation units. It can also be annoying to track down if you’ve defined the same function multiple times in different headers.

Reduced compile time: if the compiler only needs to know about the declaration in another area of compilation, giving it the full definition is a waste of time. This may not be an issue for small projects, but long compile times are typically the biggest complaint against C/C++ and is very noticeable in large projects.

Portability: by separating declarations and definitions, all definitions for a project can be compiled ahead of time and assembled into libraries, so that the user only needs to have the library file and the header files which contains its definitions (this means the user doesn’t need to waste time always recompiling the library as well).

Circular dependency problems cannot be avoided if header files only contain definitions (which is a nice segway to the next question)

What is forward declaration?

A forward declaration is a means of solving circular dependency problems. Consider the following two header files:

```
#ifndef HEADER_1
#define HEADER_1

#include “header2.h”

struct A
{
	B *b;
};

#endif


#ifndef HEADER_2
#define HEADER_2

#include “header1.h”

struct B
{
	A *a;
};
```

```
    #endif
```

header1 needs to know about struct B, which is contained in header2, while header2 needs to know about struct A, which is contained in header1. The include guards here will not prevent the file from being included again, but it will prevent the contents from being used and triggering recursive includes. This means that header2 will complain that A (or B) is an unknown type.

To fix this problem, you need to use a technique called forward declaration:

```
    #ifndef HEADER_1
    #define hEADER_1

    struct B;

    struct A
    {
        B *b;
    };

    #endif


    #ifndef HEADER_2
    #define HEADER_2

    struct A;

    struct B
    {
        A *a;
    };

    #endif
```

Now B can be declared as part of struct A and A can be declared as part of struct B. For this to work they must be pointers, and none of their members can be used (since no knowledge of the implementation is available). To use them you need to include both headers in any source file that wants to use either struct.

New Programming Concepts: None (yet)

Compiling: Executables

What is a C executable?

A C (or C++) executable is a file that contains a set of instructions in machine code that tell the CPU what to do. The machine code must match the instruction set of the particular CPU/OS and therefore any program that needs to be run on a different CPU architecture of OS should be recompiled (although the OS may use an exe format that will work on different CPUs). Instruction sets typically map to assembly instructions, the only difference being that an instruction will be a set of ones and zeros rather than an assembly command (different CPU/OS configurations will typically expect a different set of ones and zeros for a given instruction).

Each OS will have its own executable format. The programs we will compile use the ELF format (Executable and Linkable Format), which is the standard format for Linux/Unix. We will look at the ELF executable in more detail later.

You can see the assembly instructions for any given program by compiling with the flag -S.

How can you compile a C program to produce an executable that you can run on the linux servers?

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

How do you compile a c++ program on the linux servers provided?

Many compilers exist for producing executables from C++ files. Assuming you are used to using the GNU C++ Compiler, it can be invoked using the command g++. You can type man g++ into the terminal to see a description of how the compiler works, and what options it can takes as command line args (or simply look it up online). For this subject we will be using the C++ version of this compiler, which is invoked with the command g++, ie:

```
g++ -o exe_name hello_world.c -Wall -pedantic -std=c++14
```

The flag -o expects the name of the executable directly after it. Source files can be given anywhere. Note that the g++ compiler can be used to compile C code (as above), and if no executable name is given, the default name a.out is used.

What is the first step in the compilation of a program (ignoring the preprocessor)?

First the compiler will parse all code that is provided to it to check its syntax. The syntax of the program is the set of rules and symbols that the compiler expects, the ‘grammar’ of the programming language is usually stored and checked against source code using regular expressions. All tokens within a source code file will usually be transformed into a hierarchical syntax tree for easy traversal at later stages of compilation.

Errors: Syntax Errors

What is a syntax error?

A syntax error is an error that occurs at compile time while the compiler is parsing the source code. Any symbols that do not appear in the grammar of the programming language cannot be properly handled by the compiler and therefore cannot be properly places in the syntax tree. A syntax error will usually prevent an executable from being produced.

What are some examples of common syntax errors?

An easy syntax error to introduce is to misspell a data type or variable (or anything really), or you can remove the semicolon (so many possibilities here, see how many you can think of).

Debugging: stderr

What is stderr?

stderr is a file stream that is typically used for printing errors to the console/terminal.

How can stderr be used for debugging and how does it differ from stdout?

stderr can be used to print strings to the console in the same way as stdout can be used with printf. The main difference is that stderr is not buffered and will print its data to the console immediately, whereas stdout will not.

The easiest way of debugging a program is using print statements to log what the program is doing at different stages. This can be used as a basic way of determining where a program crashes in the absence of any other debugging system.

Exercises:

What is wrong with the following:
(a) if (a == 2); { return a; }

(b) for (int i = 0, j = 2; i < 100 & j < 1024; I++, j *= 2) { printf(“*”); }

(a) The program contains a semicolon after the if statement, meaning the return statement will always be called.
(b) Lots of syntax errors. Firstly I (from I++) is not the same as i (from int i), so it is treated as a new variable by the compiler which has not been declared. It needs to be changed to i. Everything else will compile correctly, however the code contains a logic error (depending on how you look at it). The & is a bitwise & rather than a logical &, so the statement will perform a bitwise and between the boolean result of i < 100, j < 1024 (since those operators have higher precedence, they will be evaluated first, then the bitwise will be performed). The bitwise result will be evaluated as a boolean, and the loop will continue until it is 0 (or false). If this was not the intention, it should be changed to &&.

Write a program to calculate the nth fibonacci number, input from the command line.

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

static int fib(int n)
{
        return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv)
{
        if (argc != 2)
        {
                fprintf(stderr, “Program usage: `./exe n` where n is the nth fibonacci number”);
                return EXIT_SUCCESS;
        }

        int n;
        if (sscanf(argv[1], “%d”, &n) == EOF)
        {
                fprintf(stderr, “Command line arg was not an int\n”);
                return EXIT_SUCCESS;
        }

        fprintf(stdout, “%d\n”, fib(n));
        return EXIT_SUCCESS;
}
```

Better solution from paul:

```
#include <cstdlib>
#include <iostream>

static int fib(int n)
{
	return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv)
{
	if (argc != 2)
	{
		fprintf(stderr, “Program usage: `./exe n` where n is the nth fibonacci number”);
		return EXIT_SUCCESS;
	}

	int n;
	n = stoi(argv[1]);

	std::cout << “the result is “ << n << std::endl;
	return EXIT_SUCCESS;
}
```

What is the following code doing and what will the output be?

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

int main(int argc, char **argv)
{
	int x = 10;
	while ( x --> 0) // x goes to 0
		fprintf(stdout, “%d “, x);
	return EXIT_SUCCESS;
}
```

For this question it might be easier to rewrite the while statement as:

```
while ( (x--) > 0)
```

The issue is that the program tests the value of x then subtracts 1 which means the final output statement will print -1.