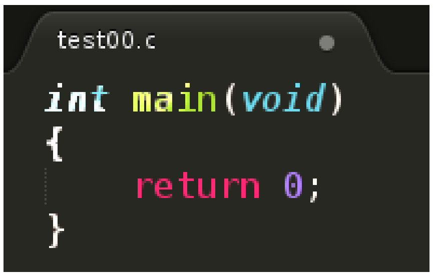
FP3405 – Workshop Task – Program Compilation Step-by-Step

Here we write a small program in C and practice compiling it. C is a programming language for writing programs.

All programs start executing from the 'main' procedure. In C / C++, procedures (collections of commands with a scope) are called functions.

Here is our main function, - it takes no arguments (hence 'void'), and returns an integer (a number, e.g. …, -7, -6, …, 0, 1, 2, …, 99, 100, …)



```
test00.c
int main(void)
{
    return 0;
}
```

…Can't copy-paste it? That was intentional… get typing!

Type it in using a text editor (e.g. Sublime Text, available under ~/soft/) as '`test00.c`'. Be tidy, save it in a folder!

Simple Compile

Compile it using the `gcc` (GNU C Compile) command:

```
$ gcc test00.c -o test00
```

Here we end up with an executable program called `test00`.

Explanation:

A breakdown of the above command (no need to run these commands here)…

```
$ gcc test00.c -o test00
```

…program name is `gcc` (GNU C Compile or something like that).

```
$ gcc test00.c -o test00
```

…1st argument to gcc is the file name of the C code you want to compile.

```
$ gcc test00.c -o test00
```

…2nd and 3rd arguments indicate the output ('`-o`') and filename of the output.

[Extra: Try leaving the 2nd and 3rd arguments off, what do we get?]

Got errors? Try again; double check what you wrote. Still got errors? Ask the Lecturer.

What kind of file is the output? (Remember the command from the *'Command Line Familiarisation'* Workshop Task? Hint: Try `file`.)
What attributes does the created file have? (I.e. `rwx`. Try '`ls -l`')
Can we execute (run) the created file?

Execute / Run

Attempt to execute the resultant file:

```
$ test00
```

Command not found? What? It's right there!

The file we just made is not in the 'search path'. (When did we last hear about the path? Hint: *'Command Line Familiarisation'* Workshop Task.)
But we don't need to put it in the 'search path' (PATH environment variable), we can just tell it to look for the file in this directory: You need to precede the program with a dot slash to do that…

```
$ ./test00
```

…this ignores the PATH variable, and just looks for it in the current working directory (as shown by '`pwd`'), i.e. the one you're currently in.

The program, as expected, did nothing, - just returned to the place from which it was called.

The compilation goes all the way from your source code (*.c) to the executable object file, but we can break down the stages to see what really happens.

Compiler Steps

The C / C++ compiler actually performs through 4 stages:

1. The 'Pre-processor'
2. Compilation (to assembly language)
3. Assemble (to machine code)
4. Link (to get the executable)

We'll skip 1. for now as our source code contains not pre-processor directives.

```
[$ gcc -E test00.c]
```

Let's **compile** to assembly language (the target is x86_64 – as given by '`uname -a`'):

```
$ gcc -S test00.c
```

…what file does this create (`ls`)? Have a look at it (`cat`)…

```
$ cat test00.s
```

Our program has been transformed into the .s file.
Looks a bit messy. Actually, we just want to see the assembly instructions, so let's ignore any line beginning with white space followed by a dot character. We'll do this by using a pipe ('|') to send the output of cat to a second program called 'grep' (General Regular Expression Parser)…

```
$ cat test00.s | grep -v ' *\.'
```

…that looks cleaner. Your whole program becomes this; just 5 machine (assembly) instructions.

```
$ file test00.s
```

…pretty handy that '`file`' command. It often knows what's what.

Let's **assemble** to machine code:

```
$ gcc -c test00.s
```

...what file does this create (`ls`)? Have a look at it (`cat`)...

```
$ cat test00.o
```

...uh oh. That wasn't healthy.

```
$ file test00.o
```

...it's something: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped.
...an 'object' file. Contains code, data, and other information.

An executable file in Linux and some other *nix systems is stored in an **ELF** file format.

Look at the file sizes of the .c, the .s, and the .o. The .o is quite big in comparison.

Two useful tools to have a look at this file...

```
$ hexdump -C test00.o
$ objdump  <some stuff here, like a switch, and the filename>
```

But we can't execute this .o file. ☹
[Extra: Try to execute it (`./test00.o`), but first you would have to make it executable (use `chmod` with '`+x`' argument on the file).]

Let's **link** to get the executable code (same as what we had originally):

```
$ gcc test00.o -o bananaman
```

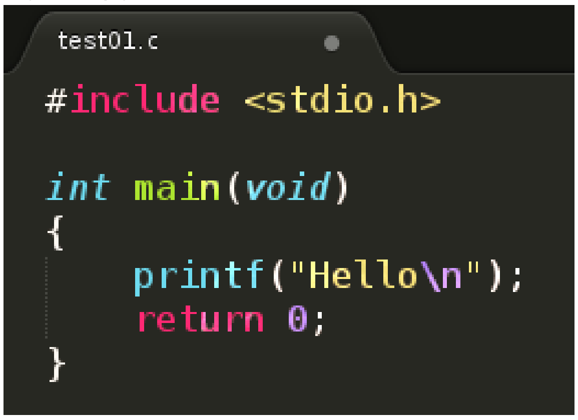
...what file does this create (`ls`)? Have a look at it (`cat`, or `hexdump` and `objdump`)

Hey, why's our program so big? Where'd all that extra stuff come from and what does it do? Why'd it suddenly get so complex? [Discussion.]

Try to execute it (`./bananaman`)

Again

Now, let's try again with a slightly improved program – one with some output to the screen, you can output any string you like.



```
#include <stdio.h>

int main(void)
{
    printf("Hello\n");
    return 0;
}
```

Type it in as before, this time call it ‘`test01.c`’. Save it in the same folder.

Repeat all stages as above. Not any differences.

We have a tool to help us look at the differences

```
$ diff test00.c test01.c
```

…only shows us the differences between the two files.

Do the same for the .s files. How about for the .o files and the executable files?

Additional practice: Use the following on a variety of files:

```
$ hexdump -C test00.o
$ objdump  <some stuff here, like a switch, and the filename>
$ readelf  <some stuff here, like a switch, and the filename>
$ nm       <some stuff here, like a switch, and the filename>
```

Self-assessment, and feedback to the lecturer

This was:	Easy | Fairly easy | So-so | Fairly tricky | Difficult
The material was:	Entirely new to me | some new / forgotten | Trivial

Common names for *nix non-alphanumeric characters:

( )	Parenthesis, parens
{ }	Braces
[ ]	Brackets, square brackets
-	Dash
/	Slash
\	Backslash, escape [sequence]
~	Tilde
#	Hash
$	String, dollar
|	Pipe
>	[Re-direct [standard] out]
<	[Re-direct [standard] in]
=	Assign, equals (non-mathematically)