Started on	Tuesday, 16 July 2019, 1:47 PM
State	Finished
Completed on	Thursday, 25 July 2019, 12:00 AM
Time taken	8 days 10 hours
Marks	19.70/20.00
Grade	**9.85** out of 10.00 (**99**%)

Information

Linux and C Basics

Goals

This lab introduces you to the C programming language and to the way in which C is supported under Linux in our labs. By the end of the lab you should:

- be able to use the shell to navigate the file system and run programs
- be able to edit, compile and build a simple C program
- be familiar with the structure of a C program containing only a main function
- be confident with int and float data types and expressions that manipulate them
- be reasonably competent with the printf and scanf functions for output and input
- be able to write a simple C program using all of the above.

Video Playlist

The video playlist for week 1 of the course is here. It is recommended that you watch all the videos before starting the lab but of course you can interleave video-watching with quiz-doing if you prefer.

Reference material

It is assumed that you have the text by King: "C Programming, a Modern Approach". The C programming section of ENCE260 follows the treatment in King fairly closely. However, because you are assumed to be a competent programmer already, we cover the early chapters of the text very quickly - this lab covers chapters 1 through 3. Note that at the end of each chapter in the text there is a Q & A section and a set of exercises. You are advised to read the Q & A sections, since they deal with lots of tricky little points that often confuse students. Also, the exercises in the text are very useful for testing your understanding of language details and improving your C skills.

The labs in this section of the course will be done within a Linux environment, so the lab includes a quick introduction to the bash shell. More information on Linux and the shell is available in the Debian on-line tutorial/reference manual (although you should ignore the chapter on Midnight Commander, as it's not installed) and of course via Google.

A bit about labs and quizzes

Labs in this section of the course are similar to those in COSC121. Each quiz is expected to be done in the same week as the material is covered in lectures. The close date for the quiz is typically in the middle of the following week. You must make your final submission of each quiz before the close date for the lab. As in COSC121, you can work on the quiz either in the labs or at home, although if you wish to work at home you'll need to have a GNU C compiler installed; see the lecture notes for your options. Of course, you don't have to complete the whole lab in a single sitting - you can log in and out of the quiz as often as you like, up until the time you've clicked *Submit all and finish*.

As in COSC121 the Precheck button gives you an unlimited number of free checks on the syntax of your submission and also checks various style and layout attributes. You should always make sure your code passes the Prechecks before clicking *Check*. The penalty regime for *Check* with most CodeRunner questions in labs is 0, 10%, 20%, ..., i.e. there is one free submissions and thereafter you pay 10% for each wrong submission. The penalties for multichoice questions are much higher - typically 30% to 50%.

The total of all your quizzes in ENCE260 is 10%. Roughly 4% of this will come from the six C quizzes, which are thus worth only around 0.7% each. However, please keep in mind at all times that it's the *learning* that's important in quizzes, not the marks. Copying other people's code might get you the marks (unless we catch you) but it certainly won't get you the programming skills. People who get 90% in the quizzes and assignments but then struggle to get 10% in the mid-semester test look very foolish - and they probably fail the course.

Question **1**

Correct

Mark 1.00 out of 1.00

Firstly a question to help us gather statistics on outcomes.

Which of the following best describes your programming background? If you have done COSC121 please select the first option, otherwise select one of the others. The *language* in which you've programmed (if you have) doesn't matter. All answers earn full marks.

Select one:

- I've done COSC121. ✔
- I haven't done any programming before.
- I've done only a little bit of programming (I've never written a program longer than about 40 lines of code).
- I've written at least one program longer than 40 lines of code but less than say 200 lines.
- I've written at least one program of over 200 lines in length.

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Information

Using the bash shell

This information panel explains some of the more important capabilities provided by the *bash* shell and available through a Linux terminal window. Knowing how to use the shell is very important in all sections of ENCE260. Please don't feel tempted to skip over it ("*I don't want to know all this old fashioned rubbish; I'll just use a GUI all the time"*). You might save half an hour now but the price will be many wasted hours later.

Most of the material in this lab is introduced in the following two videos. It is recommended that you watch the videos with a computer at hand, trying out the various commands as they are introduced. Please don't skip these videos - there's lots of useful information in them that isn't present in the text below, including keyboard shortcuts that can save you a lot of time.

- Don't bash bash
- More Linux stuff

1. The terminal

Unix was originally designed to allow multiple users to connect to a central machine, each user sitting at a terminal. A terminal was initially a mechanical teletype - a sort of electric typewriter designed for sending telegrams around the world.

Each terminal is really two devices: a keyboard, which sends typed characters down a wire to the computer, and a printer, which displays characters received down a wire from the computer. Usually the computer is programmed to echo each keyboard character back to the printer but not always (e.g., when passwords are entered). Many or most Linux programs take their input by default from a so-called "standard input" stream of characters, which defaults to the keyboard, and they send their output to a so-called "standard output" stream, which defaults to the screen. Standard input can easily be redirected to take data from a disk files rather than the keyboard. Similarly, standard output can be redirected to a disk file rather than to the screen.

When a user connects to Unix through a terminal, a special program called a *shell* is started. [Unix has various layers of software, which were traditionally visualized as being spherical, like layers of an onion. The innermost bit of the sphere was called the kernel and the outermost layer, which was the one the user interacted with, was called the shell.] The standard default shell in Linux is called `bash`. The original Unix shell, called just *sh*, was written by Steve Bourne at Bell Labs. A succession of more elaborate shells followed, such as *csh*, *tcsh* and *ksh*. When the GNU/Linux developers rewrote the shell code (which they did to avoid any licensing disputes) they called it the "Bourne again shell", or `bash`.

The shell receives typed input from the user, one line at a time. A line of input to the shell is assumed to be a command to the computer, and the shell has the job of interpreting the command and sending the response back to the user. All commands begin with a command word, which is either the name of a command that is "built in" to the shell or is the name of a file on disk that contains an executable program. The rest of the command line modifies how the command should run, e.g., by specifying data file(s) and command options. Although modern Unices, such as Linux, now have extra software layers that provide a GUI ("Graphical User Interface") for desktop use, much of the functionality can still be accessed only through shell commands. Most system-level management of Linux based servers like web servers (around 70% of the world's web servers are Linux-based) is via the command line access. Users usually access the shell through a terminal emulator program. In the version of Linux supplied in the lab the terminal emulator can be found on the screen of applications by typing "terminal" into the search window. You might want to drag the icon onto the desktop to make it more readily available.

Start a terminal now, and use it to experiment with the commands that follow.

2. Syntax of a typical bash command line

A typical shell command line is of the form

```
cmd -o1 -o2 --long1 --long2 file1 file2 < inputfile > outputfile
```

where:

- **`cmd`** is either a command that's built in to the shell or the name of a file in one of the directories that bash searches to find executable code.
- **`-o1`** and **`-o2`** are "short" options, usually consisting of a single letter perhaps followed by a digit e.g.
  - **`-a`** for listing all files including system files (used by the `ls` or "list directory" command)
  - **`-O2`** for optimisation level 2 compilation (used by the C compiler).
- **`--long1`** and **`--long2`** are longer more-readable versions of options provided by many commands as alternatives to the hard-to-remember one-letter version, e.g. **`--all`** is equivalent to **`-a`**
- **`file1`**, **`file2`** etc are files to be processed by the command
- **`< inputfile`** is used to redirect so-called "standard input" to come from the given input file rather than the keyboard. For example

  ```
  python3 myprog.py < data.txt
  ```

  runs the python3 program *myprog.py* but takes standard input from *data.txt* rather than the keyboard.
- **`> outfile`** is used to redirect so-called "standard output" to go to the given output file rather than the screen. For example

  ```
  ls -l a.txt b.txt > dirlist
  ```

runs the built-in shell command **`ls`** ("list directory") to list in a long format the details of files **`a.txt`** and **`b.txt`**, sending the output to a file **`dirlist`**.

3. Filenames

File and directory names should not contain spaces!

Filenames beginning with a period are system files and are generally hidden unless specifically requested. For example, `.bashrc` (in your home directory) is the bash script that runs whenever bash starts. Two special cases are '`.`', which denotes the current directory, and '`..`' which denotes the parent of the current directory. [A *directory* in Linux is the equivalent of a *folder* in Windows.]

When naming files, usually a full *filepath* can be given. This is made up of a series of directory names separated by forward-slash characters ('/' - NB *not* the backwards slash used by Windows) with the actual file name at the end. Filepaths are absolute if they start with a '/' (which denotes the so-called root directory), otherwise they are relative to the current directory. For example, `/usr/bin/awk` is the file `awk` in the absolute directory `/usr/bin`, whereas `usr/fred.c` or the equivalent `./usr/fred.c` is the file `fred.c` located in the `usr` subdirectory of the current directory.

A command line filepath can be written as a pattern containing one or more '*' or '?' characters: a '*' matches any sequence of characters while a '?' matches any single characters. Pattern-type filepaths are expanded by the shell in a process called 'globbing' to a sequence of actual filepaths. For example, if the current directory contains just two text files `a.txt` and `b.txt`, the directory listing command

```
ls -l a.txt b.txt > dirlist
```

could be written more compactly as

```
ls -l *.txt > dirlist
```

Another useful filepath expansion performed by the shell is the replacement of '~' (tilde) with your home directory path so that, for example,

```
ls ~
```

lists your home directory regardless of what directory you're currently in.

4. Some useful commands

The following commands should be enough to get you through most of the first half of ENCE260. To find out what options are available for each command, you can usually use the *man* command (which prints the relevant page of the user manual, hence the name), e.g.

```
man less
```

to find out about the *less* command. Unfortunately, however, some of the commands, like *cd* are built into bash, so *man cd* for example doesn't work. To find about these commands you can do *man bash* but - be warned - you'll likely drown in the output. A better solution in such cases is to use Google (say) to search for: *linux cd command.*

And now ... here are the commands you should know:

`info`	Get info on a command (a sometimes-useful alternative to `man`)
`ls`	List specified directory/files (default: current directory)
`cd`	Change current directory
`cp`	Copy files
`mv`	Move (or rename) files
`rm`	Remove files (careful! No prisoners taken)
`mkdir`	Make a new directory
`rmdir`	Remove a directory
`less`	Display the contents of a file
`set`	Set an environment variable (affects subsequent commands)
`man`	Display a man page
`apropos`	Search the manual for a keyword (same as `man -k`
`vim`	In-terminal editor ("VI-iMproved"). You'll hate it.
`gcc`	Run the GNU C compiler (see later)

Question **2**

Correct

Mark 1.00 out of 1.00

Choose the bash command (just its name) that you would use to perform the given operation. Some of the available commands won't be used. The penalty for resubmitting this question is 33%.

Display a file on the terminal	less ✔
Create a new directory	mkdir ✔
Show the contents of a directory	ls ✔
Delete a file	rm ✔
Make a backup copy of a file	cp ✔
Rename a file	mv ✔

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

Suppose you wanted to save the output from an `ls` command ("list directory") to a file `directory.txt`. Which of the following commands would do that?

As for most single-answer multichoice questions in this quiz, the penalty for a wrong submission is 50%. If you don't know the answer, perhaps you should re-read the information section rather than guessing?

Select one:

- `ls <directory.txt`
- `ls --out=directory.txt`
- `ls -o directory.txt`
- `ls directory.txt`
- `ls >directory.txt` ✔ Yay!

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Question **4**

Correct

Mark 1.00 out of 1.00

In a long-format directory listing (use *man ls* to find out which options are required to get a long-format directory listing), how can you identify which files are directories?

Select one:

- The file name ends with a slash.
- The line begins with a 'd' for 'directory'. ✔
- The line begins with an 'f' for 'folder'.
- The file attributes contain one or more '*'s, denoting it's not an ordinary file.

Correct

Marks for this submission: 1.00/1.00.

Question **5**

Correct

Mark 1.00 out of 1.00

What word does the letter 'w' in the Linux command `pwd' stand for? [Use the *man* command to find out what the *pwd* command does.]

Answer: working ✔

Correct

Marks for this submission: 1.00/1.00.

Question **6**

Correct

Mark 1.00 out of 1.00

What is the effect of the command "`cd .`"? Note that the full stop is part of the command.

Select one:

- Change to the parent directory.
- Change to the home directory.
- Nothing. ✔
- Play the current audio CD.

Correct

Marks for this submission: 1.00/1.00.

Question **7**

Correct

Mark 1.00 out of 1.00

Suppose you're writing a program called *myprog* which reads data from the keyboard and writes its output to the screen. You're finding it annoying to debug this program, because every time you run it you have type in all the input before finding out whether the output is correct or not.

A better approach might be to enter all the input into a file called, say, *in.txt*, and then run your program so that it reads from that file instead of the keyboard. Which of the following *bash* commands would you use to run your program, assuming you don't modify your program to take input from somewhere other than the keyboard?

Select one:

- `./myprog -i in.txt`
- `./myprog <in.txt` ✔
- `./myprog --input=in.txt`
- `./myprog in.txt`
- `./myprog >in.txt`

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Information

We'll now look at one particularly important Linux command, *gcc*. This runs the GNU C Compiler, the program that translates your C source program into machine code.

Firstly we need a simple program to practice with. We'll use the classic *Hello World* program, which you're not expected to understand yet.

Type the command *nano hello.c* to start the simple text editor called *nano* in a terminal, editing a (new) file *hello.c*. Then copy and paste the following code into the window (using *Edit > Paste* from the menu or a middle-mouse-button-click or *SHIFT-CTRL-V,* **not** *CTRL-V*). Yes we know it has "issues" but please just copy and paste it exactly as it is rather than trying to fix it.

```
#include <stdio.h>
int main(void)
{
    printf('Hello world!\n');
}
```

Exit from *nano* (CTRL-X), saving the program (Y) and confirming the file name (Enter). Then type the following commands into the terminal, observing what happens after each one.

```
gcc -o hello hello.c
./hello
rm hello
gcc -o hello -Werror hello.c
./hello
```

Question **8**

Correct

Mark 1.00 out of 1.00

Select the correct statements from the following list. As with most multianswer multichoice questions, the resubmission penalty is 33.3%.

Select one or more:

- *gcc* is the C equivalent of Python, i.e., it runs C programs for you.
- When *gcc* prints warning messages, there is a strong probability that the program is broken and won't run correctly. ✔ Indeed
- Despite the warning messages, *gcc* still outputs an executable program, which you can run if you choose. ✔ Yes, but choosing to do so is probably unwise.
- The occurrence of a "segmentation fault" when you run your program is generally considered to be an undesirable outcome. ✔ Obviously (?)!
- Anyone who ignores warning messages from the C compiler is a dodo who deserves what they get. ✔ You have been warned.
- Including the -Werror flag on the command line is A Good Thing To Do because it stops you from proceeding after you've got warning messages from the compiler. ✔ Indeed! [I sure hope you didn't take several goes at this one.]

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Information

What just happened?!

As you've realised (I hope), I've given you a bad program that the C compiler compiles, albeit with warnings, to generate a seriously broken program that crashes with a segmentation fault when it's run.

You're not really expected to understand why, yet -- we'll look at a correct C program very soon. However, if you would like to correct the program just for your personal satisfaction, change the single quotes to double quotes in the call to the *printf* function. You should now be able to compile and run the program without any warnings, segmentation faults or other ugly behaviour.

Now ... let's move on to something different before we return to looking at our first C program.

Using *geany*

Although we encourage you to get familiar with the terminal interface to Linux (which you will have to use in the second term, anyway), most of the labs in the first term will use the geany mini-IDE, which provides you with *Compile*, *Build* and *Execute* buttons. Each of these can be programmed to execute the appropriate bash command, as explained below.

Launch geany and create a new empty file called prog1.c somewhere in your new ence260 directory tree. Save the empty file, quit from geany then right-click the file in the file browser (i.e., the program you get when you, say, double-click your home directory on the desktop). Select *Properties* and choose the *Open With* tab. Select g*eany* (which makes it the program to be used by default to open C programs) and click *Close.*

If you now double-click *prog1.c* in the file browser it should open in geany. Do that now.

Use the *Build > Set Build Commands* menu command to set up the *Compile*, *Build* and Execute commands, which are just shell commands with a bit of extra syntax to insert the current filename, with or without its extension, into the command. Check that they are set as follows; if not, fix them:

Compile: `gcc -std=c99 -Werror -Wall -g -c "%f"`
Build: `gcc -std=c99 -Werror -Wall -g -o "%e" "%f"`
Execute: `"./%e"`

Stuff you really do need to know:

Yes, I know it's tempting to skip over the following, but each of the following bullet points is relevant to the rest of the course, and you'll waste time later if you try to save time now. [Which I know from bitter experience a large percentage of you will do. Sigh.]

- **`gcc`** is the GNU C compiler, which translates source files into machine code.
- The **`-g`** option on the compile and build commands tells the compiler to output extra information into the executable file, such as the source code line numbers, which is useful when we need to wheel out the heavy-duty debugging tools such as *valgrind*.
- **`-std=c99`** ensures that any programs you write are ISO C99 compliant.
- **`-Wall`** tells the compiler to output all its warning messages.
- **`-Werror`** ensures that any warnings are treated as errors, preventing you from proceeding when the compiler is warning you of defects in your program. If you omit this option, which is not the default, the compiler will accept all sorts of nonsense code, and you may waste hours of unnecessary debugging time. Also, be warned that the quiz server is set to reject any C-code that generates warning messages.
- The special string **`"%f"`**, meaningful only to *geany*, means "use the source file name here".
- The special string **`"%e"`**, meaningful only to geany, means "use the source file name without the extension here".
- The **`-c`** option on the *compile* command tells the compiler to do just the compilation step, not a full compile-and-build. If the source file is, say, `prog.c` the output will be a new file called `prog.o`; this is the "relocatable object code" that needs to be combined with various library code (start-up code, printf etc) before it can be executed. With the *Build* command set as above, you won't generally need to use the *Compile* command, except to do a quick syntax check, as the *Build* command includes compilation.
- The *Build* command given above lacks the `-c` option and is therefore a full "compile-then-build" command, which generates an executable program. By default, the output executable program would be called `a.out`, which isn't a very helpful name. Therefore we also add in a `-o` option which must be immediately followed by the filename to use for the executable file. For example `-o fred` would result in the executable output file being called `fred`, and `-o "%e"` results in the executable file having the same name as the source file but without the `.c` extension. Thus, for example, building the file `prog1.c` will yield an executable program called `prog1`.
- The command to run when the *Execute* button is pressed is simply the name of the executable file to be run, as specified in the build command; it must be prefixed by "`./`" because by default Linux will not look for executables in the current directory unless so instructed.

Question **9**

Correct

Mark 1.00 out of 1.00

Select all the correct statements regarding the setup of geany from the list below, assuming the above instructions in the lab spec on how to configure it for this course have been carried out. Also assume that a C program is open for editing. [Note: this question is intended to be answered entirely on the basis of the above material. However, if you wish to experiment to confirm your answers, you may wish to postpone answering this question until after question 9, when you should know more about compiling programs with geany.]

Select one or more:

- The *Build* button will not build an executable file unless the *Compile* button has been pressed first.
- The *Execute* button will not generally work unless the *Build* button has been used at least once. ✔
- In the Compile and Build commands, the `-g` option tells the compiler to generate code that will use a GUI (Graphical User Interface).
- Geany will not build an executable if the attempt to compile the source code generates warning messages, since these will be treated as errors. ✔ That's correct, but only because we set the -Werror flag in the command line. It's not the default behaviour.
- In the build command, the `-o` option is followed by the name of the executable file to generate. ✔ That's right. Remember that the two must go together - students often insert extra flags in between the two and wonder why it doesn't work.
- If you were to omit the `-c` option from the *Compile* command, it would act like the *Build* command except that it would generate an executable file called *a.out.* ✔

Correct

Marks for this submission: 1.00/1.00.

Information

prog1.c

Using *geany* type the program below into a new file called prog1.c [Yes, shock horror, you're expected to actually type it in yourself. When learning new programming languages, copy and paste is *not* your friend.] Try to understand each line as you enter it. Numbers in circles, are references to the notes that follow the program. The code in the function body should be indented by four spaces - the tab key *should* achieve that but if you're working at home and/or not using *geany* to edit code you might need to adjust your editor preferences to replace tabs with a 4-character indent.

A 14 minute video explaining the following program is available here.

```
/* Your first mindless C program */    (1)

#include <stdio.h>    (2)

int main(void)    (3) (4) (5)
{
    // First the declarations    (1) (6)
    int number1;    (7)
    int number2;
    int total;

    // Now some code    (8)
    number1 = 10;
    number2 = 20;
    total = number1 + number2;
    printf("The sum of %d and %d is %d\n", number1, number2, total);
    (9) (10) (11) (12)
    return 0;
}
```

Note the following (with reference to the ringed numbers):

1. In C99, the version of C taught in this course, there are two sorts of comments: those delimited by /* and */, which can run over multiple lines, and those starting with '//' which run only to the end of the line. You should however be warned that the older but more portable ANSI C89 standard supports only the first type of comment, so if you envisage porting code between different operating systems it is safer to stick to C89, where the -std=c99 flag would be replaced by the -ansi flag on the compile and build command lines.
2. The line `#include <stdio.h>` is a C preprocessor command. The C preprocessor, `cpp`, is a separate program that is called by `gcc` before it starts compiling in earnest. `cpp` copies a file from its standard input to its standard output, transforming it according to preprocessor commands embedded within the input file itself. The commands are identified by a '#' character at the start of the line. These are not Python-style comment lines! The command `#include <stdio.h>` causes the C preprocessor to insert into the source file at this point an exact copy of the file `stdio.h`. This file contains declarations of various "standard I/O" functions, such as the `printf` function. The enclosing of the name in angle brackets is a syntactic convention that indicates that the file is found in the system include directory (or a list of such directories). In our case, `stdio.h` is to be found in the directory `/usr/include`.
3. A C program consists simply of a set of functions, one of which must be the specially-named *main* function. Unlike Python, you can't have "global code" that lies outside of any function declarations.
4. Unlike Python, C does not place any semantic significance on indentation, white space etc, although newlines shouldn't appear in the middle of strings. Instead, semicolons are used to terminate statements and opening and closing braces are used to define program blocks. However, this does not mean that you can forget all the nice use of indentation that you learned in COSC121! You are still expected to lay out your code as though indentation was semantically significant.
5. The `main` function is called when the program is executed. It is defined to return an int value which should be 0 if the program executes correctly without errors. A non-zero return value indicates an error has occurred. The return value is often checked by shell scripts (i.e., files containing a sequence of shell commands, like "batch" files in Windows) to abort the script's execution if a single step fails. You may come across example C programs in which the main function is of type `void` (meaning it doesn't return a value) or in which its return type is omitted.
6. Each function (or, actually, each code block) starts by *declaring* all the variables that the function/block will use. Space for these is allocated on the stack when the function starts. This *static typing* is very different from the dynamic typing you are used to in a language like Python where the type of data bound to a variable can change during execution. Each *declaration* specifies the type of data the variable will contain (in this case int) and the name of the variable. [In C99, you can actually mix up the declarations with the statements, but we ask that you don't do this in ENCE260; it's important to understand the difference between declarations and statements and keeping them separate in this way is pedagogically useful.]
7. C's `int` type isn't quite the same as Python's. In C, `int`s are a fixed size (unspecified by the standard!) - 32 bits on our lab machines.
8. Lines of actual executable code must come after the declarations. In this example there are three assignment statements, a call to the `printf` function and a `return` statement.

9. The `printf` function outputs text to "standard output", which appears in the terminal window unless redirected as indicated earlier. If you're running the program from within geany a temporary window is opened for displaying standard output and for echoing keyboard input.
10. The first parameter to printf is a template or pattern string enclosed in double quote characters. You can't use single quotes for delimiting strings - they're used for char (character) constants only (see later labs). The output from printf is the template string with the various format specifiers - in this case %d - replaced with corresponding arguments, i.e., the first format specifier is replaced by the first argument after the template, the second by the second argument, etc. The last character in the template is usually the newline character, written as '\n', to ensure that the output cursor moves to the start of the next line.
11. Format specifiers start with a '%' character, have optional field width and precision specifiers (introduced in later labs) and finish with a character specifying the type of conversion or formatting to apply to the argument. In this case, %d specifies that the argument should be formatted as a decimal integer. Other types of format specifiers you'll meet in due course are s for string and f for floating point numbers, plus (more rarely) e for exponential floating point, o for octal and x for hexadecimal.
12. If the template argument contains no format specifiers, it is output exactly as is. There should be no other arguments to the function in that case. For example

    ```
    printf("Hello world\n");
    ```

    However, in such cases an alternative function *puts*, or "put string", may be preferable. For example

    ```
    puts("Hello world");
    ```

    Unlike *printf*, *puts* always adds a newline to the end of the string it prints.

Compiling and running prog1.c

Compile the program `prog1.c` by clicking the *Compile* button in the toolbar. Observe in the *Compiler* tab of the bottom pane that the command

```
gcc -std=c99 -Werror -Wall -g -c "prog1.c"
```

is executed. [If that's not the command that's executed, did you set the *Compile* command as explained in the g*eany* section earlier?] `gcc` is the GNU C compiler.

Observe in the file-manager window that a new file, `prog1.o`, has appeared. This is a relocatable object code file. It contains the compiled machine code in a form that allows it to be relocated to a suitable place in memory when the executable file is built.

Now click the *Build* button. This time the executed command should be

```
gcc -std=c99 -Werror -Wall -g -o "prog1" "prog1.c"
```

This again executes the C compiler, `gcc`, except that the `-o` flag tells it that after compiling the program it should also build the required executable file `prog1`. The building is actually done by a separate program, `ld` ("ell dee"), which is the so-called link editor, or just linker for short. `ld` merges `prog1.o` with the various library code required, such as `printf`. The output is an executable file, `prog1`; observe with the file browser that `prog1` has appeared.

Run the program by clicking the *Execute* button in geany or by typing the command

```
./prog1
```

in a terminal window. Verify that the screen output is as expected.

Question **10**

Correct

Mark 1.00 out of 1.00

Match up the given file names on the left with the descriptive comments on the right. Some of the descriptive comments will not be used.

prog1	User's executable program ✔
cpp	Pre-processor ✔
ld (that's ell-dee not eye-dee!)	Linker ✔
gcc	C compiler ✔
geany	IDE ✔
prog1.c	Source code ✔
stdio.h	Contains declaration of printf ✔
prog1.o	Relocatable object code ✔

Correct

Marks for this submission: 1.00/1.00.

Question **11**

Correct

Mark 1.00 out of 1.00

Assuming the same geany settings as before, what happens if you enclose the string parameter to *printf* in single quote characters rather than double quote characters?

Select one:

- The program won't compile. ✔
- Nothing changes -- the program compiles and runs exactly as before.
- The program compiles, with warning messages, but prints only the first character ('T') of the string.
- The program compiles with warning messages but gives a segmentation fault when executed.

Correct

Marks for this submission: 1.00/1.00.

Question **12**

Correct

Mark 1.00 out of 1.00

The program below should print the value of 60,000 * 60,000. What value does it actually print? [It's assumed you're doing this on a COSC lab machine. You'll probably get the same answer on other machines, but it's not totally guaranteed.]

Your answer should be just a number, containing only '+', '-' and digit (0 - 9) characters.

The resubmission penalty is 10%.

```
#include <stdio.h>

int main(void)
{
    int n = 60000;
    printf("%d\n", n * 60000); // Print a big number
    return 0;
}
```

Answer: -694967296 ✔

This is the answer you should get on a COSC lab machine. If you got something else, you're either not using a machine that has 32-bit ints as standard or your program is wrong.

This demonstrates the limitations of fixed-length 32-bit integers.

Correct

Marks for this submission: 1.00/1.00.

Question **13**

Correct

Mark 1.00 out of 1.00

Why does the previous program print an answer that is mathematically incorrect?

Note: this question is arguably slightly unfair since you haven't yet learnt about number representation, which will be covered soon in the computer architecture section. However, the feedback from the previous question will give you some idea of the likely answer and you might be able to deduce more from the knowledge that an *n*-bit number has only 2^n different states, roughly half of which are used for positive numbers and half for negative numbers. The penalty for a wrong submission is set to 10% and the feedback from a wrong answer should give you enough further information so you don't pay any further penalties.

If you wish to give yourself a head start on number representation, see here, for example.

Select one:

- Because there's an error in the GNU C compiler.
- Because *ints* on our machines can be at most 2^{31} - 1 and values above that "wrap" to negative numbers. ✔ Yes, well done. This is called "integer overflow".
- Because *ints* on our machines can be at most 2^{32} - 1 and values above that "wrap" to negative numbers.
- Because *ints* on our machines can be at most 2^{63} - 1 and values above that "wrap" to negative numbers.
- Because *ints* on our machines can be at most 2^{64} - 1 and values above that "wrap" to negative numbers.

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Question **14**

Correct

Mark 1.00 out of 1.00

Write a C program that simply prints

```
Hello ENCE260!
```

to standard output. Develop and test it in *geany*, then copy the code into the answer box below.

Note:

1. As in COSC121, your code is subject to certain style checks before it is run. The only ones likely to affect you in this question are:
   - The opening and closing braces of a function body should be on lines by themselves, in column 0
   - Function body lines should be indented by 4 spaces

   Others will be introduced as they become relevant.
2. In ENCE260 most CodeRunner questions display a *Precheck* button, which allows you to check for free that your code satisfies any style constraints and that the code compiles correctly. You are strongly advised to always click *Precheck* before clicking *Check.* If the precheck fails, fix your code then precheck it again.
3. The usual penalty regime for ENCE260 is 0, 10%, 20%, ... i.e. you get one free submission.

For example:

Result
`Hello ENCE260!`

Answer: (penalty regime: 0, 10, ... %)

```
#include <stdio.h>
int main(void)
{
    printf("Hello ENCE260!\n");
}
```

	Expected	Got	
✔	`Hello ENCE260!`	`Hello ENCE260!`	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00.

Information

celsius.c

Create another C program file called `celsius.c` in *geany*. Enter the following code. But before you do ...

... note the spelling of *celsius*. It is **not** *celcius*.

This program introduces several new features of C as explained in the notes below. Please read each of these carefully and/or watch the 12-minute video on the program here.

```
/*********************************************
 * celsius.c (King Chapter 2, page 24)
 * Converts a Fahrenheit temperature to Celsius
 * Slightly modified by RJL
 *********************************************/

#include <stdio.h>
#include <stdlib.h>     ①

#define FREEZING_PT 32.0      ② ③
#define SCALE_FACTOR (5.0 / 9.0)

int main(void)
{
    float fahrenheit = 0;    ④ ⑤ ⑥ ⑦ ⑧
    float celsius = 0;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);  ⑨
    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent: %.1f\n", celsius);  ⑩
    return EXIT_SUCCESS;
}
```

Key points (with reference to the circled numbers):

1. This program includes a new file `stdlib.h`, for reasons that will become clear very shortly.
2. The program introduces another C preprocessor command - `#define`. The line `#define FREEZING_PT 32.0` results in the C pre-processor replacing every subsequent occurrence of the token `FREEZING_PT` with the string "`32.0`", which is all characters following the `FREEZING_PT` token within the `#define` line. Similarly, the token `SCALE_FACTOR` gets replaced by the string "`(5.0 / 9.0)`". It is important to realise that this is just a raw textual replacement, occurring before the C compiler proper gets invoked. Thus, by the time the program gets passed to the compiler proper, the line beginning

   ```
   celsius = ...
   ```

   will have been converted to:

   ```
   celsius = (fahrenheit - 32.0) * (5.0 / 9.0);
   ```

3. When the replacement text for a *#define* contains operators, you should enclose the replacement text in parentheses to ensure that arithmetic operations take place in the right order.
4. Although it is legal in C to declare multiple variables on a single line, e.g. *int fahrenheit, celsius*, the ENCE260 style guideline requires that you use a separate declaration for each variable, as it makes them much easier for the reader to find and so you can attach explanatory comments to each one later, if you choose.
5. As stated earlier, we ask that you put all declarations before all statements in a function. This isn't actually a *requirement* of C99 but it is a requirement of C89 and is recommended when you are still learning the fundamentals.
6. When declaring new variables in C, you are strongly advised to specify their initial values too (e.g., `int i = 0`). Note that this is still a *declaration* (with an initialiser) *not* an assignment statement. In C, if you inadvertently use an uninitialised variable before assigning it a value, you get a random value from memory. This makes the output of the program unpredictable and debugging difficult. Programs may work one day but fail the next, or work on home machines but not when we test them on the quiz server. This problem can't occur in Python or Java.
7. This program introduces the `float` data type. This is a 32-bit floating point number, with about 7 decimal digits of accuracy. That's much less accurate than Python's `float` type, which is a 64-bit representation. If you want the extra accuracy (and why wouldn't you, unless you want huge arrays of floating point numbers?) you should instead declare your variables as being of type `double`, although this does create a slight extra hassle with the `printf` format specifiers - see later.
8. The other basic C type to be introduced in due course is type `char`, which is essentially an 8-bit integer used to hold a single ASCII character. There is no boolean type; we'll see later how to get around that.

9. The `scanf` function, declared in `stdio.h`, is an input function that can be loosely thought of as "`printf` running backwards". This particular example reads a `float` value into the variable `fahrenheit`. Like `printf`, its first parameter is a format string, which is a template to be matched against the data read from the standard input. The precise matching rules are a bit non-intuitive, e.g., white space in the format string matches arbitrary amounts of white space in the input. Type *man scanf* for details. Subsequent *scanf* parameters are the names of the variables into which the numeric data read from the input will be placed. Each variable name must be preceded by '&' for reasons that are inexplicable at this stage in the course. Just accept it for now!
10. Note that standard input is by default read from the keyboard but as explained earlier in this lab, it can be redirected from the command line. For example, a command like `./celsius < mydata.txt` would run the celsius program, feeding it the text data from the file `mydata.txt` as standard input. In *geany* you can redirect standard input and output in the same way, by setting an appropriate execute command via the *Build > Set Build Commands* menu item.
11. When using floating-point format output specifiers, it is usual to include a precision specifier, indicating how many digits should follow the decimal point. For example, '`%.1f`' indicates that a float value is to be formatted with a precision of 1 decimal digit. If a fixed field width of, say, 6 characters were required (e.g., for tabular output), we'd use instead a format specifier that includes a field width too, e.g. '`%6.1f`'.

Build and run `celsius` several times, entering different data and experimenting with the code, until you think you understand it properly.

Question **15**

Correct

Mark 1.00 out of 1.00

Program *celsius.c* includes *stdlib.h*. If this include were omitted a compiler error would occur because a symbol is undeclared. What is that symbol? [Its name is required, not its value.]

Answer: EXIT_SUCCESS ✔

Correct

Marks for this submission: 1.00/1.00.

Question **16**

Correct

Mark 0.90 out of 1.00

In the program *celsius.c*, if a semicolon were erroneously added to the end of the line

```
#define FREEZING_PT 32.0
```

then the line beginning "celsius = ..." would be converted by the preprocessor into:

Select one:

- `celsius = (fahrenheit - 32.0) * (5.0 / 9.0);;`
  i.e., there would an extra semicolon at the end of the line.
- `celsius = (fahrenheit - 32.0;) * (5.0 / 9.0);`
  i.e., there would an extra semicolon embedded in the line. ✔
- `celsius = (fahrenheit - 32.0) * (5.0 / 9.0);`
  i.e., the same line as if the semicolon were omitted.
- No conversion would take place -- the C preprocessor would reject the #define line.

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.90/1.00**.

Question **17**

Correct

Mark 1.00 out of 1.00

Suppose a C program contained the following:

```
#define THING 4 + 2
int silly = THING * THING;
```

What would be the value of 'silly'?

Answer: 14 ✔

No, it doesn't give the answer 36! Because the parentheses around the expression were omitted, the raw textual replacement resulted in the line
int silly = 4 + 2 * 4 + 2;

Correct

Marks for this submission: 1.00/1.00.

Question **18**

Correct

Mark 1.00 out of 1.00

What is the output from the following two lines of C? [Try to predict the output rather than run the code.]

```
float f = 5.0 / 9.0;
printf("f = %.1f\n", f);
```

Select one:

- f = 5.0 / 9.0
- f = 0.5555555
- f = .5555555
- f = 0.5
- f = .5
- f = 0.6 ✔ Values are rounded.
- f = .6

Correct

Marks for this submission: 1.00/1.00.

Question **19**

Correct

Mark 1.00 out of 1.00

How does the output from celsius.c change if the SCALE_FACTOR is defined by the line

```
#define SCALE_FACTOR (5 / 9)
```

?

Select one:

- Answers are always 0.0 ✔ Yes, that's right. The division operator in C, when applied to integers, is more-or-less equivalent to Python's // operator. However, if either operand is a float or double, the result will be a float or double.
- Answers are integer values, rounded down.
- The output does not change.
- Answers are integer values, rounded to the nearest integer.

Your answer is correct.

Correct

Marks for this submission: 1.00/1.00.

Question **20**

Correct

Mark 0.80 out of 1.00

Write a C program that prompts the user "How many miles? " (noting the extra space after the question mark), reads their response as a floating-point number and prints out the number of kilometres to 2 decimal digits of accuracy. 1 mile = 1.609344 kilometers.

When run interactively, the terminal output for the first test would be something like the following (where `[rjl83@cssecs3 ~]$` is your bash command prompt):

```
[rjl83@cssecs3 ~]$ ./myprog
How many miles? 1
That's 1.61 km.
[rjl83@cssecs3 ~]$
```

In this interactive mode, the characters you type on the keyboard are echoed to the output.

When run on the quiz server, however, the input is taken from a data file, not the keyboard. To simulate this, use the editor to create a test file `data.txt` that contains the input data that you type when running the program interactively, i.e. the single line

```
1
```

You can then run your program using this input file as data as follows:

```
[rjl83@cssecs3 ~]$ ./myprog < data.txt
How many miles? That's 1.61 km.
[rjl83@cssecs3 ~]$
```

Note that neither the input data nor the terminating newline appears in the output when running non-interactively. This is different from Python questions in COSC121 where we simulated the keyboard echo by using our own version of the *input* function. We can't do that in C.

A new style rule:

- Identifiers must contain 3 or more characters, except for *i, j, k, m, n, s, t, x, y, z*. (And possibly other exceptions introduced as required for specific questions, e.g. to represent common physical concepts, such as *v* for velocity.)

For example:

Input	Result
1	How many miles? That's 1.61 km.
2.2	How many miles? That's 3.54 km.

Answer: (penalty regime: 0, 10, ... %)

```
#include <stdio.h>

#include <stdlib.h>

#define SCALE_FACTOR 1.609344

int main(void)
{
    float miles = 0;
    float kilometres = 0;
    printf("How many miles? ");

    scanf("%f", &miles);
    kilometres = (miles * SCALE_FACTOR);
    printf("That's %.2f km.", kilometres);
    //printf("Celsius equivalent: %.f", celsius);
    return EXIT_SUCCESS;
}

```

	Input	Expected	Got	
✔	1	How many miles? That's 1.61 km.	How many miles? That's 1.61 km.	✔
✔	2.2	How many miles? That's 3.54 km.	How many miles? That's 3.54 km.	✔
✔	123.456	How many miles? That's 198.68 km.	How many miles? That's 198.68 km.	✔

Passed all tests! ✔

Correct

Marks for this submission: 1.00/1.00. Accounting for previous tries, this gives **0.80/1.00**.

◄ ENCE260 C Summary (pdf from Learn) Jump to... Quiz 2: Expressions and Flow Control ►