

# Lab 1: Introduction to C Programming Environment

---

## Learning outcomes

---

- Understand and discuss programming environment
- Understand steps involved in writing a C program
- Understand program submission environment

## Prerequisites

---

None if you're using lab computers. If you're using your personal Windows 10 machine, you must install the WSL environment and programming tools. Details are provided on the course's home page.

## Introduction

---

A computer's operating system is a resource manager consisting of a set of programs that coordinate *safe*, *efficient*, and *abstract* access by users and application programs to system resources such as CPU, memory, input/output (I/O) devices including disks, printers, keyboards, monitors, network interfaces, and so on. *Safe* coordination means that the operating system allows only one application program to access a particular system resource at a time. For example, the operating system will allow only one application program to send data to the printer at a time. Data from other programs is spooled until the printer has completed its previous job. After the conclusion of the current print job, the operating system transmits the spooled data to the printer using a first-in-first-out priority. *Efficient* coordination means that programs waiting for their input or output operations to be completed (such as reading from or writing to files) have their execution suspended to make way for the execution of other programs. The operating system provides *abstractions* (such as files rather than disk locations, or network sockets rather than network interface addresses) that isolate application programs and users from low-level hardware details. Finally, operating systems provide *system utilities* or *commands* that allow users to accomplish universal functions such as file maintenance, printing, redirecting file input and file output.

Computers in DigiPen labs have the Windows Subsystem for Linux (WSL) that provides a Linux environment directly on Windows 10. For tutorials and assignments in this course, you'll be editing, compiling, linking, and executing your code in Linux.

You might be wondering why this introductory course in programming doesn't just use Windows and what is to be gained using a Linux environment. There are three main reasons why you should learn Linux.

- First, Linux is everywhere. Whether you are aware of it or not, you're already using Linux every moment of every day. If you've an Android cell phone, you're using a variant of Linux. If you've an Apple cell phone, you're using iOS which is a variant of Unix from which Linux itself is derived. The web and internet are powered by servers running Linux. In-vehicle infotainment systems, airplane in-flight entertainment systems, consumer electronics including smart TVs, DVRs, voice controlled devices such as Amazon Echo and Google Home, home security systems, and numerous industrial applications such as robots, assembly lines, and mass-transit systems are entirely powered by Linux.

- Second, Linux is free. Linux is an open-source operating system, with source code distributed by the [Free Software/Open Source](#) community. Anyone who receives it can make changes and redistribute it. No one company or individual owns Linux, which was developed, and is still being debugged and improved, by thousands of corporate-supported and volunteer programmers.
- Third, it is good for your career. Because Linux is free, versatile, and present everywhere, experience in Linux is a necessary requirement for most software developers and engineers. The basic exposure to Linux gained in this course will begin your journey in gaining and developing job skills that are in demand and well paid.

Since Linux on Windows PCs is implemented directly on top of Windows, part of using Linux effectively is learning to use Windows effectively. This tutorial begins with an introduction to the command line features of the Windows operating system. The aim is to get you comfortable using textual commands to get your work done in Windows. The tutorial continues with a short introduction to Linux and the bash shell for doing useful work in Linux. Next, the tutorial introduces the tools that you'll repeatedly use as a software developer. You'll be introduced to the process of creating a program written in C programming language: using Visual Studio Code to edit a C source file, using GCC C compiler toolchain `gcc` to compile the source file into an object file, link this object file with the C standard library to create an executable program, and then executing this C program. The final aim of this tutorial is to expose you to the nitty-gritty details of submitting programming assignments to a server that will automatically grade your program.

## Shells and Command-line interfaces

---

Most operating systems support two forms of interaction with users: command-line interfaces (CLIs) for textual input using a keyboard and graphical user interfaces (GUIs) for interactions involving a mouse device and a keyboard. A [shell](#) is a text-based CLI program that acts as a *command interpreter* - it takes commands from the keyboard and passes them to the operating system for execution, and then displays the results of these commands on the screen. Using terse textual commands, users can launch programs such as word processors, browsers, games, run a string of commands as a single command, and, in general, interact more efficiently with the rest of the system. Users can pipe the output of one command to another command. Finally, tasks can be automated by encapsulating the commands required to execute the tasks into scripts and then running the scripts through the shell. On the other hand, GUI-based applications provide direct user interaction but rely on menus, buttons, a keyboard and a mouse device to get anything done. This means that powerful tools require complicated interfaces so that the user can access with a mouse all the features of the application. The more features an application has, the more complicated the interface gets.

Ordinary users of computers prefer to interact with computers and the operating system controlling these computers using a GUI. In conjunction with a keyboard and mouse, a GUI provides such users a visually intuitive interface to view, control, manipulate, and toggle through multiple folders and programs. Almost all of us, irrespective of the operating system loaded on our computer, prefer using the GUI associated with that operating system. However, GUIs are neither convenient nor efficient in many scenarios that arise in software development. Instead, programmers prefer using a shell to interact with the operating system because it increases productivity and is an important part of the tool chain that every programmer must know.

Consider a directory with a single file that has an image encoded in the JPEG format. Suppose you wish to convert the JPEG image to PNG format. You would launch GUI-based image editor, open the file, change certain settings in the editor, and then convert it to the destination PNG format. These actions could potentially take about 2 minutes of your time. Now, suppose the directory contains ten JPEG image files. Although you would find the manual conversion of each file

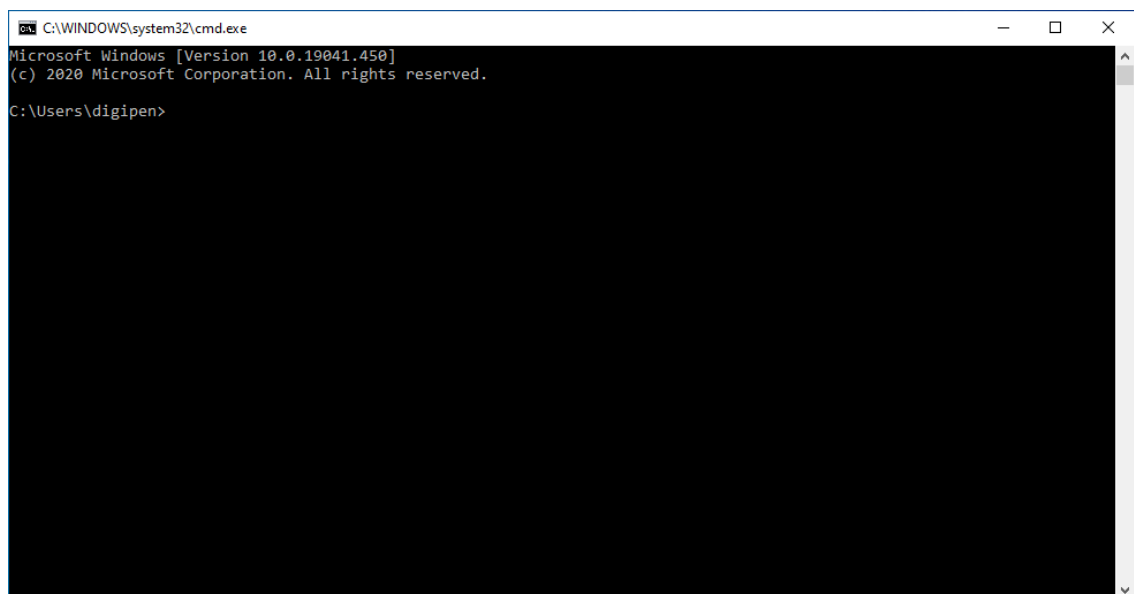
cumbersome, the entire process would require less than 30 minutes. What if the directory contains hundreds or even thousands of JPEG image files? The amount of time required to manually convert a large number of files using the GUI will considerably slow down your productivity for days on end. In this particular situation, you could type a single command in the shell to perform the conversions in a fraction of the time required by the GUI-based manual method. As a software developer, you will be frequently called upon to automate manual tasks, and in such scenarios, knowing how to use a shell will considerably improve your productivity.

Another advantage of using the shell is the ability to collect together a set of textual commands into a simple program called a *script*. You can author scripts whenever you wish to automate certain tasks and run the script using just one command whenever required. This is very useful if you have some specific need or require a specific behavior with some tool, which is not implemented by any graphical application. A perfect example of automating software development tasks is provided by the *make* utility. Suppose your software contains hundreds or even thousands of source files (not inconceivable as your software grows both in scope and ambition). Individually compiling such a large number of source files is a giant undertaking. Instead, you'd specify the set of tasks to be executed by the C compiler as shell commands in a text file called *makefile*. When you type `make` in the shell, the commands in the *makefile* are executed. Although, depending on your computer's configuration, you may have to wait a while for these thousands of source files to be compiled and linked into an executable program.

## Task 1: Windows command prompt

Windows provides a command shell (variously called *Command Processor*, *Command Prompt*, `cmd` prompt, or `cmd.exe`) that allows users to directly interact with the Windows operating system. Complete the following tasks to get some practice using the Windows shell.

1. The easiest way to open Windows command shell is to press `WIN+R` keyboard shortcut to open the `Run` dialog box. The `WIN` key contains the Windows logo and is located to the left of `Space Bar`. In `Run` dialog box, enter `cmd.exe` and hit `Enter`. Supposing your student login is `digipen`, the command shell will look like this:



```
CA\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.19041.450]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Users\digipen>
```

2. To use Windows command shell, enter a valid command along with any optional parameters followed by `Enter` key. The Windows command shell then executes the command as entered and performs the task or function it is designed to perform in Windows. Ok, let's try some typing! Execute command `ver`. The help pages for the command can be read by typing command `help` followed by command name followed by `Enter`:

```
1 | C:\Users\digipen>help ver
```

3. Now, type command `VER`.
4. Was the output for commands `ver` and `VER` different? What does this tell you about the case sensitivity of Windows shell? Windows is not case-sensitive. This means that command `help` is equivalent to `HELP` or `Help` or `HElp`. Similarly, a file `plot4.R` is equivalent to `PLOT4.R` and to `pLOT4.r` and all other combinations of uppercase and lowercase characters.
5. Next, type command `time`. The shell displays the current time and if you've administrative privileges on your computer, the shell also allows you to change the current time.
6. Use `help` command to obtain additional information about `time` command. How would you tell Windows to just output current time, without the prompt for changing current time?
7. Now hit up-arrow key `↑`. Watch how your previous command returns. This means that Windows shell has *command history*. Now, try using left-arrow `←` and right-arrow `→` keys. These two keys allow you to position the text cursor anywhere in the command line allowing the easy correction of typing mistakes.
8. A convenient way to view your command history is to hit `F7` key which will display a pop-up window containing a list of recently executed shell commands. You can scroll through the list using `↑` and `↓` keys.
9. Another way to view your command history is to type the following command in the shell:

```
1 | C:\Users\digipen>doskey /HISTORY
```

10. You can customize your Windows shell window by typing `ALT+SPACEBAR+P` to open the shell's *Properties*. Clicking *Font* tab allows you to change character fonts. Clicking *Colors* tab will allow you to change the shell's background and text colors. Clicking *Options* tab will allow you to change command history size.

## Task 2: Files and directories

A *file* is the primary unit of storage on your computer. Every file has a name, which generally indicates the type of information the file contains. As your collection of files grows from a few hundred to thousands, a hierarchical [file system](#) is required to locate and access specific files. Think of a file system as a type of index used by the operating system for all the data contained in a storage device.

Every operating system uses *directories* (also known as *folders* in Windows) to organize its files, the files of application programs, and users' files such as images from a digital camera. These directories are organized in a hierarchical tree structure, starting from the *root directory*. In addition to files, directories may contain other directories. A directory within another directory is called a *subdirectory*. Subdirectories may in turn contain sub-subdirectories and files. By creating subdirectories, you can better categorize and organize your files.

Just as a directory is a group of files, a *drive*, which is always represented by a drive letter (C, D, and so on), is a group of directories. Drives are usually associated with physical disk drives. A single physical drive could be logically divided into multiple drives.

Strictly speaking, all directories are subdirectories, except for the *root directory*. The root directory is the starting point from which all other directories branch out - in other words, the root directory is the root of the tree structure used to organize directories. The root directory does not have a name. Instead, it is represented by a backslash `\`. When the current working directory of the shell is root directory, the shell command prompt appears as `C:\>`. This prompt indicates that you're in the root directory of drive `C`.

In practical terms, a hierarchical filesystem means the following. When you first start Windows shell, you will be at a point in your file system known as your *home directory*. Within that directory, you can make subdirectories, and within them you can make sub-subdirectories, and so on. So, your filesystem has a tree-like shape that is a directory in the operating system's filesystem. This hierarchical filesystem helps you to organize your files. For example, suppose you are taking two courses labeled CS100 and MAT140. You could make directories with these names, and then keep all your files for a given course in that directory. You may have already started looking for a summer internship, so you might create a directory named `Internships`, and then keep all your resume's, cover letters, and other related documents in that directory.

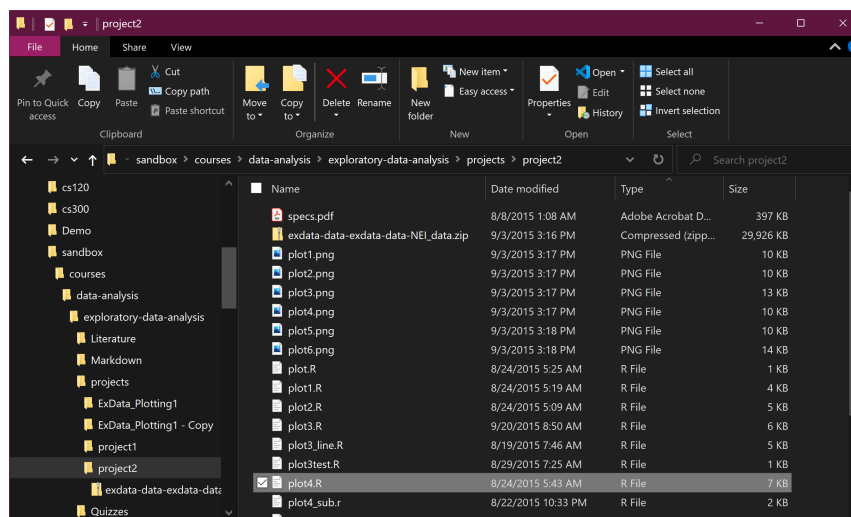
The *current working directory* of the shell is described in the shell's prompt: `drive\current-directory>`. Supposing your student login name is `digipen`, your Windows command shell is launched in directory `C:\Users\digipen`. Now, complete the following tasks:

1. What happens when you type command `dir`? `dir` stands for *directory* which in GUI environments such as Windows is also referred to as a *folder*. Are you able to see all the files and directories in the home directory?
2. Use command `help` to identify the different parameters (or command extensions) available to `dir`. In particular, investigate `/A`, `/O`, `/S`, and `/W` extensions.
3. Close the shell by typing `exit`.
4. If you're using your personal machines, use your favorite technique (here's [one](#)) to create a folder called `sandbox` in `C:` drive. Such a folder pre-exists on all lab machines.

## Task 3: Paths

A *path* is the course that leads from the root directory of a drive to the file you want to use. The *pathname* is the string of characters starting with the drive letter and including the names of directories, subdirectories and the name of the file to be accessed separated by backslash `\` character. A pathname can be specified in two ways:

- Absolute pathname: An *absolute pathname* begins from the root directory of a drive. For example, the path to access file `plot4.R` in the hierarchical tree structure from the following picture will be specified at the command prompt as `C:\sandbox\courses\data-analysis\exploratory-data-analysis\projects\project2\plot4.R`. This is the absolute pathname to file `plot4.R`. The first letter and colon `C:` represent the drive the file is stored on. The first backslash represents root directory `sandbox`. The second backslash separates `sandbox` directory from a subdirectory `courses`. The third backslash separates `courses` subdirectory from a subdirectory `data-analysis` and so on.



- Relative pathname: A *relative pathname* is the course that leads from the current working directory. For example, if the current working directory is `C:\sandbox\courses`, then the relative path to access file `plot4.R` is `data-analysis\exploratory-data-analysis\projects\project2\plot4.R`.

Now, complete the following tasks:

1. Changing directories: Launch the Windows shell. How will you navigate from your home directory to a new location? To change directories, use command `chdir` (for *Change Directory*). The period character `.` indicates *current working directory* while a string of two period characters `..` indicates *parent directory*. Use command `chdir ..` to move from the current working directory to the parent directory. Use `chdir` to change your current working directory to `C:\sandbox`. If you're having trouble, don't forget the built-in help which can be accessed like this:

```
1 C:\Users\digipen>help chdir
```

2. Creating a directory: Read the built-in help information on command `mkdir` (using command `help`). In the current working directory `C:\sandbox`, create a new directory named `Fruit`. Further, create a subdirectory named `Mango` in directory `Fruit`. Change your current working directory to `Mango`. Does your command prompt look like this?

```
1 C:\sandbox\Fruit\Mango>
```

Change back to the parent directory `Fruit`. The simplest way is to use the two periods `..` shortcut in your command.

3. Deleting a directory: Read the built-in help information on command `rmdir`. Delete directory `Mango` to simplify your directory structure. Further, delete directory `Fruit`. Were you successful? Can you delete a directory if the Windows shell is in that directory? If you were unsuccessful the first time, you'll have to step up to the parent directory `C:\sandbox` and then use command `rmdir` to delete subdirectory `Fruit`.
4. Close the Windows shell by typing `exit` at the command prompt.

## Task 4: More file management

This task continues the trip through Windows file management using the Windows shell. In particular, you'll learn how to startup or execute an application program from the shell, work with files, and use *wildcards* to manipulate groups of files.

1. Launch the Windows shell and change your current working directory to `C:\sandbox` and create a subdirectory named `test` in this current working directory.
2. Visual Studio Code (referred to as Code in this document) is a free, modern, multi-platform, and open-source source code editor for a variety of popular programming languages that is installed on all DigiPen lab desktops. Avoid using Notepad (also from Microsoft and also installed on all Windows computers) because it was never designed to be a source code editor. Since Code will be used frequently, you will find it convenient to launch the editor by typing `code` followed by `Enter` in the shell. Even better, to set Code's current working directory to your current working directory, provide Code the current directory using the period character `.`:

```
1 | C:\sandbox>code .
```

3. Use the editor to create and save a text file `foo.001` in directory `C:\sandbox\test`. Type any text you want into the file.
4. Read the built-in help information on command `copy` (using command `help copy` in the shell). Using command `copy`, make two copies of file `foo.001` named `foo.002` and `foo.003`.
5. Using command `copy`, append files `foo.001`, `foo.002`, and `foo.003` and name the destination file `foo.004`.
6. Read the built-in help information on command `rename`. Using command `rename`, rename file `foo.004` as `boo.004`.
7. Make a copy of `boo.004` called `boo.final`. At this point, you'll have files `foo.001`, `foo.002`, `foo.003`, `boo.004`, and `boo.final` in current working directory `C:\sandbox\test`.
8. If you want to carry out a task for a group of files whose names have something in common, you can use one or more *wildcards* to specify groups of files. Two wildcards are commonly used: question mark `?` and asterisk `*`. Wildcard `?` represents *zero or a single* character that a group of files have in common. Wildcard `*` represents *zero or more* characters that a group of files have in common. Using command `mkdir`, create subdirectory `Math` in current working directory `C:\sandbox\test`. Read the built-in help information on command `move`. Using command `move` and wildcard `*`, move all files from current working directory to subdirectory `Math`. Use command `dir` to confirm the current working directory is now empty of files.
9. From current working directory `C:\sandbox\test`, use command `move` and wildcard `*` to move all files in subdirectory `Math` to the current working directory. Use command `dir` to confirm that subdirectory `Math` is now empty of files.
10. Read the built-in help information on command `erase` or command `del` (they are the same command). Then, use command `erase` to delete file `foo.003`.
11. Use command `erase` and wildcard `?` to delete files `foo.001` and `foo.002`. Further, use command `erase` and wildcard `*` to delete the remaining two files `boo.004` and `boo.final`.
12. Use the two periods shortcut `..` in your pathname to move from the current working directory to `C:\sandbox`. Delete subdirectory `test`.
13. Close the Windows shell by typing `exit` at the command prompt.



## Task 5: I/O streams and redirection operators

Operating systems use a powerful abstraction called *streams* to provide a flexible, portable, and efficient means of input and output (I/O). Regardless of where I/O originates or is destined, it is dealt as streams of characters. A *stream* is a sequence of bytes that may be associated with a file or a physical device such as a printer, keyboard, or monitor. A *text stream* is a sequence of characters divided into lines; each line consists of zero or more characters followed by a newline character. The contents of a text stream can be read or viewed using a text editor. A *binary stream* is a sequence of unprocessed bytes that are unreadable using a text editor. The stream abstraction allows programs to read input data and write output data from a variety of devices without having to handle the details of each individual device.

By default, every program is provided three I/O streams to interact with its environment: *standard input* (`stdin`) for the program to read input, *standard output* (`stdout`) for the program to write output, and *standard error* (`stderr`) for the program to write error or diagnostic messages. Conceptually, think of `stdin` as a pipe through which input information flows from keyboard device to the program, `stdout` as a pipe through which output information flows from the program to computer screen, and `stderr` as a pipe through which error and diagnostic information flows from the program to computer screen.

*Redirection operators* can be used to redirect input to `stdin` stream and output from `stdout` stream to different devices than the default devices. More specifically, redirection operator `<` can redirect input into `stdin` stream from default keyboard device to a file. Likewise, redirection operator `>` can redirect output of `stdout` and `stderr` streams from default computer screen to a file. Complete the following tasks to learn about redirection of input and output streams:

1. Read [this](#) table to understand the various operators that redirect `stdin`, `stdout`, and `stderr` streams.
2. Launch Windows shell and change current working directory to `C:\sandbox`. Download file `stats-win.exe` from the lab webpage. Your browser may issue a warning about downloading and saving this executable file; ignore the warning because I can vouch that the executable is neither a virus nor a malware. Instead, this program reads zero or more integer values from `stdin` (keyboard device), computes [mean and median](#) of these values and writes these two statistics to `stdout` (computer screen). Next, copy the downloaded file into `C:\sandbox`.
3. Run the program from command prompt (by typing program name `stats-win.exe` followed by `Enter` key). The program will wait for the user to enter zero or more integers. The user can signal to the program that no further input is to be expected using `Ctrl+D` combo. The `Ctrl+D` combo is a special character in Windows called the *End-of-File* character `EOF` to signal to programs that no further input is to be expected. You enter a sequence of integers 10, 5, 20, 12, 3, 8 followed by `Ctrl+D` followed by `Enter` key (not the commas please - just the numbers). The program will stop reading from `stdin` as soon as it reads `Ctrl+D`. Next, the program computes mean and median of input integers as floating-point values and prints their values to `stdout`. The entire exercise is illustrated below with line 3 showing the program's output for the input on line 2:

```

1 C:\sandbox>stats-win.exe
2 10 5 20 12 3 8 Ctrl+D
3 9.67 9.00
4
```



4. Launch Code from command prompt to create file `input.txt`, type integers 10, 5, 20, 12, 3, 8 into the file on a single line or multiple lines, and save the file (in current working directory `c:\sandbox`). The simplest way to create file `input.txt` is like this:

```
1 | C:\sandbox>code input.txt
```

The Windows shell provides command `type` to display contents of text file(s). Enter `type input.txt` at the command prompt to verify that you entered the integers correctly.

5. Program `stats-win.exe` can now be provided input in text file `input.txt` rather than default keyboard device by redirecting the input of the program's `stdin` stream using *input redirection* symbol `<` followed by `Enter` key.

```
1 | C:\sandbox>stats-win.exe < input.txt
2 | 9.67 9.00
3 |
```

As seen above, the redirection generates the same result as when you manually type integral values in the shell. An implicit *End-of-File* character is generated when the program reads the last character in file `input.txt`. Input redirection plays an useful role in two scenarios. First, it removes the tedium of typing lots of values. Second, it allows programs to be automated without waiting for user interaction. You'll be relying on input redirection to write programs that read data from files throughout this semester. Therefore, make sure to completely and thoroughly understand the useful technique of input redirection.

6. In a similar fashion, it is possible to redirect the output of the program's `stdout` stream from the default computer screen to a file using *output redirection* symbol `>`:

```
1 | C:\sandbox>stats-win.exe > output.txt
2 | 10 5 20 12 3 8 Ctrl+D
3 |
```

In this case, you'll type the input sequence of integer values using the keyboard device while the output is redirected from default computer screen to file `output.txt`. Confirm the output file's contents using command `type`:

```
1 | C:\sandbox>type output.txt
2 | 9.67 9.00
3 |
```

Note that if file `output.txt` already exists, it'll be overwritten.

7. If output file `output.txt` already exists and you wish to append new results to the file, use append operator `>>` instead. Exercise the append operator like this:

```
1 | C:\sandbox>stats-win.exe >> output.txt
2 | 10 5 20 12 3 8 a
3 |
```

Confirm the output file's contents using command `type`:

```
1 C:\sandbox>type output.txt
2 9.67 9.00
3 9.67 9.00
4
```

8. It is sometimes useful to redirect both input and output streams simultaneously, like this:

```
1 C:\sandbox>stats-win.exe < input.txt > output.txt
```

Confirm input and output redirections are working correctly:

```
1 C:\sandbox>type output.txt
2 9.67 9.00
3
```

9. Delete all files in `C:\sandbox` and close the Windows shell by typing `exit` at the command prompt.

## Task 6: Piping

*Piping* is a form of redirection where a program's output is further processed by directing the first program's output as input of another program. Complete the following tasks to put piping into practice.

1. Launch a Windows shell and set current working directory to `C:\sandbox`. At command prompt, enter command `help`. The output of command `help` is too large to fit into the shell's window and most of the output whizzes by too fast and remains unread. Command `more` takes text as input and formats the text so that it is presented to the user one screen at a time. The concept of piping can now be used to redirect command `help`'s output as input to command `more` which presents its input in presentable single-window chunks. Exercise command `more` like this:

```
1 C:\sandbox>help | more
```

Use the spacebar to update the contents of the shell's window when using the `more` command. Use the `q` button if you want to quit from the `more` command.

2. Download file `printints-win.exe` from the lab webpage (again ignore any warnings from your browser) and copy the file into current working directory `C:\sandbox`. This program takes as input an integer value from the command line and prints an equivalent number of randomly generated numbers to `stdout`. Run this program at the command prompt to print 1000 randomly generated integer values like this:

```
1 C:\sandbox>printints-win.exe 1000
```

You're unable to read the 1000 numbers printed to `stdout` because they whiz by too fast. By piping the program's output as input to command `more`, you can examine the numbers at leisure one terminal screen at a time:

```
1 C:\sandbox>printints-win.exe 1000 | more
```

- Piping is a useful feature because it allows different programs to be chained together to create more powerful programs. Let's look at another example of the usefulness of stream redirection and piping. Generate a large set of random numbers in text file `random.txt` like this:

```
1 C:\sandbox>printints-win.exe 1000 > random.txt
```

The mean and median of these 1000 random values can be computed like this:

```
1 C:\sandbox>stats-win.exe < random.txt
```

All of this can be done in a single step using piping:

```
1 C:\sandbox>printints-win.exe 1000 | stats-win.exe
```

- Delete all files in `C:\sandbox` using `erase` or `del` commands and close the Windows shell by typing `exit` at the command prompt.

## Linux

Just like Windows and macOS, Linux is a popular operating system originating from another popular operating system called Unix. On a historical note, C was invented in the early 1970's to implement the Unix operating system. Unix has now been popular for more than 50 years. Today, without variants of Unix systems, the Web and Internet would come to a screeching halt. Cellphone calls could not be made, electronic commerce would grind to a halt, and there would have never been console games, or "Jurassic Park" or any other CGI movie.

Unix was first fabricated in 1969 at AT&T Bell Labs (where the transistor was invented in 1947) as the operating system for a [time-sharing](#) computer that allowed multiple users to run multiple programs that could be (almost) simultaneously executed on the CPU. Later in 1973, Ken Thompson and Dennis Richie succeeded in rewriting Unix in a new high-level programming language called C which itself was invented by them for the sole purpose of authoring Unix. This is an important event because all other previous operating systems were written in assembler in order to extract maximum performance from hardware and were therefore difficult to port to computers made by other vendors. Because Unix was authored in a high level programming language, it became easier to port Unix across different hardware architectures and CPUs. During the 1980s, the multi-user, multi-tasking, hardware-independent nature of Unix made the computer industry adopt Unix as an operating system suitable for all computers.

Starting in 1975, AT&T made Unix widely available and at minimal cost to educational institutions making Unix popular in computer science departments. This led to a ripple effect - computer science students whose lab work had pioneered these new applications of technology were attaining management and decision-making positions inside the burgeoning technology industry and amongst its customers. And they wanted to continue using Unix and making it more versatile by adding additional features. Many variants of Unix were introduced by various hardware vendors. Starting in 1974, the Berkeley campus of the University of California emerged as the single most important academic hot-spot in Unix development with a variant called Berkeley Unix. In 1983, Berkeley Unix added TCP/IP networking features and other major enhancements at the behest of the Defense Advanced Research Projects Agency to power servers for the [ARPANET](#) network. After the addition of TCP/IP, everything changed. The ARPANET network became the precursor to the Internet. And the networking tools and system utilities in Unix became an important reason for the popularity of Unix as servers for the Internet.

The increasing popularity of different flavors of Unix marketed by computer vendors such as IBM, DEC, and HP made it more complex to port applications across hardware, operating systems, and vendors. The [POSIX](#) standard was developed in the 1980s to resolve the portability issue. POSIX does not define the operating system, it only defines the [application programming interface](#) (API) between an application and an operating system, tools, and programs for software compatibility between variants of Unix and with other operating systems. Programmers have the freedom to write their operating system and application anyway they want as long as the interface between the two is honored. Because POSIX is independent of hardware, operating system or vendor, it's easier to achieve application portability.

Later in 1991, high licensing costs of Unix led to the development of an open source version of Unix called Linux. It has now become the best-known and most used variant of Unix. Aside from Windows powered computers and devices, almost every other device in the computing world is powered by an operating system that is a variant of Unix. The list includes Linux, Android, iOS, Apple desktop operating system macOS, Chrome OS, and Orbis OS (used in PlayStation 4). Large hardware vendors such as IBM, HP, SGI, Sun, DEC, Compaq had their own versions of Unix. Except for IBM's AIX, the rest are more or less extinct and have been replaced with Linux. This document will use Linux in a general sense rather than referring to a particular variant.

## Design features of Linux

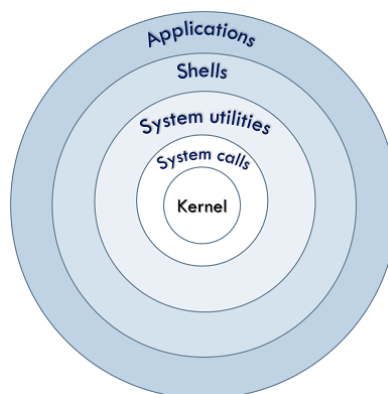
One of the many consequences of the pace of hardware and software development, is that most of what one knows becomes obsolete over every 18 months or so. Linux does not abolish this phenomenon, but does do a good job of containing it. There's a bedrock of unchanging basics: command line languages such as Korn shell, Bourne shell, high-level programming language C, system calls, software development utilities such as *make*, and networking libraries that one can actually keep using for decades. Elsewhere it is impossible to predict what will be stable; even entire operating systems cycle out of use. Thus the loyalty Linux commands. Much of Linux's stability and success are attributed to design decisions made back at the beginning; decisions that have been proven sound over and over. The following is a list of design features that were incorporated in the initial developmental philosophy of Linux that have played a critical role in its popularity.

- Linux is designed as a multi-user, multi-tasking operating system with multiple users having multiple tasks running simultaneously. This was not true with early versions of Windows and Apple's Mac OS.
- Linux is not specific to a particular type of computer hardware; instead, it is a hardware independent operating system. Linux is still the only operating that can present a consistent, documented API across a heterogeneous mix of computers, vendors, and special-purpose hardware. It is the only operating system that can scale from embedded chips and handhelds, up through desktop machines, through servers, and all the way to database back ends, and special-purpose number-crunching supercomputers.
- Linux is flexible for users because it supports multiple styles of user interface: plain text using a terminal and through a Graphical User Interface (GUI) as seen in Linux and macOS. Users can type plain text containing terse commands and messages to interact with Linux. This means that Linux can be implemented as a lightweight operating system without carrying the complexity overhead built into GUI-based operating systems such as Windows 10 and macOS. Note that this is often seen as a weakness because it increases the perceived complexity of the system to end users. The purpose of this tutorial is reduce this perceived complexity of Linux-like systems.
- Linux has become the one indispensable core technology of the entire Internet and its modern face, the World Wide Web. To function effectively as an Internet expert, an understanding of Linux and its components are indispensable.

- Linux was designed to be a software development environment consisting of a toolkit that contains many system utilities that improve the productivity of programmers. Each system utility is specialized to perform only one task very well. The output of every utility is expected to become the input of another, yet unknown, utility. This ability of Linux to glue together utilities mean that components of its basic toolkit can be combined to produce useful effects that designers of individual utilities never anticipated. This allows programmers to rapidly prototype new tools, toolkits, and programs by getting something small working as soon as possible and modifying it incrementally until it is finished.
- The open-source community is most active in the Linux world. High-quality, open-source software and development tools are usually, equal to, or superior to, their proprietary equivalents. See [here](#) for a comprehensive list that compares open-source and proprietary software.

## Linux components

The following figure presents a highly simplified architecture of a Linux operating system and shows how Linux succeeds in presenting users and application programs with a uniform interface without regard to the details of the underlying hardware.



**Kernel:** At the center of the figure is the *kernel*, the core of the operating system, that is loaded into memory during system start up (the boot process). The kernel is in direct control of the underlying resources or hardware of the computer. The kernel provides low-level device, memory and CPU management functions such as dealing with interrupts from hardware devices, sharing the CPU among multiple programs, allocating memory for programs, and so on. Hardware-independent kernel services are provided to *application programs* (such as web browsers and word processors) and *system utility programs* (simple but useful programs that are a part of the operating system and only perform a specific task and no more, for example, a program that finds files that contain certain text) through the *system call* interface. Each system call implements a basic service such as allocating memory to a program, enforcing file permissions, creating a new file, deleting an old file, start a program's execution, or open a logical network connection to another computer, and so on.

**Shell:** Linux supports two forms of interaction with users: command line shells for textual input and GUIs for interactions involving a mouse and a keyboard. As explained earlier, a *shell* is a text-based command-line interface program that acts as a command interpreter for users - it takes commands from the keyboard and passes them to the kernel for execution, and then displays the results of these commands on the computer screen. A Linux system contains different shells such as Bourne shell, Korn shell, and Bourne Again shell (*bash*) with each shell having its own strengths and weaknesses. On the other hand, GUI-based applications provide direct user interaction but rely on menus, buttons, a keyboard and a mouse device to get anything done. This means that powerful tools require complicated interfaces so that the user can access almost all the features of the application with a mouse. The more features an application has, the more complicated the interface gets.

**System utilities:** In Linux, system utilities are designed to be powerful tools that do a single task extremely well. Every variant of Linux provides a standard set of several hundred utilities for accomplishing universal functions such as printing, text editing, file maintenance, maintaining file permissions, and so on. For example, the program called `grep` (short for *Global Regular Expression Print*) finds text inside files. The utility `wc` (short for *Word Count*) prints statistics related to the text in a file including the counts of number of words, lines and bytes. Users can often solve problems by interconnecting these tools instead of writing a large monolithic application program.

**Application programs:** Most versions of Linux provide several open-source applications as standard. Examples include text editors, latex - a powerful typesetting language, image viewers, C and C++ compilers.

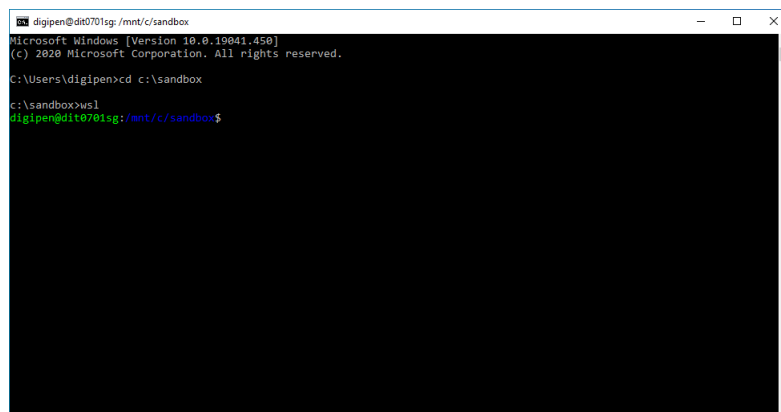
## Task 7: Starting Linux bash shell

The Windows Subsystem for Linux (WSL) lets developers run a Linux environment directly and with minimal overhead on Windows so that most Linux facilities such as command-line tools, utilities, and applications can be executed unmodified. WSL provides a comprehensive collection of Linux utilities and programs such as `sed`, `awk`, `grep`, `make` for programmers to use Linux toolchains in Windows. Most importantly, the [GNU Compiler Collection](#) (GCC) - a collection of free, high-quality, open-source compilers and interpreters for C, C++, Python, Fortran, and other languages - can be easily installed in WSL. Think of WSL as an environment that will allow you to execute Windows applications in Linux and write Linux code in Windows. You might use WSL in a practical setting to provide a convenient environment in Windows to build applications for a variety of platforms including embedded chips and handhelds, desktop machines, database back ends for e-commerce, and special-purpose number-crunching supercomputers. Note that a program built in Linux using WSL is a native Linux program meaning that it can be ported to other Linux environments. However, the Linux program will not execute in plain Windows. Complete the following tasks for an introduction to Linux.

1. Begin by launching a Windows shell and change your directory so that the current working directory is `C:\sandbox` - you've practiced this in earlier tasks. To launch the Linux command shell, run command `ws1`:

```
1 | C:\sandbox>ws1
```

Windows will launch the Linux shell. Again, it is easy to see because your command prompt has changed:



```

digipen@dt0701sg: /mnt/c/sandbox
Microsoft Windows [Version 10.0.19041.450]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\digipen>cd c:\sandbox
c:\sandbox>ws1
digipen@dt0701sg: /mnt/c/sandbox$

```

Remember command `ws1` as it is the command you use to switch into Linux from Windows. Very conveniently, you will still be in the same directory `C:\sandbox` as you were in Windows. To get out of Linux and return back to Windows, simply type `exit` in the Linux shell. If you happen to type `exit` in the Windows shell, it will close the command prompt

and you'll have to reopen it.

Notice the directory you're in after switching to Linux: in Windows you were in directory `C:\sandbox` while in Linux the directory is `/mnt/c/sandbox`. When you're in Linux, the entire C drive for Windows is located in `/mnt/c`. Also notice that directories are separated by a forward slash `/` in Linux, while Windows uses a backslash `\`.

You should see a *shell prompt* consisting of your user name (say `digipen`) followed by `@` followed by the name of the lab desktop (say `dit0701sg`) followed by a colon `:` followed by Linux path followed by a dollar sign `$` followed by a space:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$
```

- Now type some nonsense characters and hit `Enter` key:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ @#$$
```

You should get an error message from Linux operating system because it cannot find an utility named `@#$$`:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ @#$$
2 | @#$$: command not found
```

- Linux has many shells. You can determine the system's default shell using command `echo $SHELL` and the current shell being used using command `echo $0`:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ echo $SHELL
2 | /bin/bash
3 | digipen@dit0701sg:/mnt/c/sandbox$ echo $0
4 | -bash
```

The results of the two commands indicates that both default and current shells are *bash* shells.

- Just like the Windows shell, *bash* shell has *command history*. Pressing up-arrow key `↑` makes the previous command appear. Pressing down-arrow key `↓` will bring back the blank line. Say you wish to type command `cd` with a lengthy pathname:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ cd ../game/graphics/assets/shadowmaps
```

Just as with the Windows shell, command `cd` changes directories in a Linux bash shell. Suppose, though, that you mistype the command as

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ cd ../game/graphi fx/assets/shadowmaps
```

Also, suppose you have not yet hit the `Enter` key. Then, you can change the `graphi fx` to `graphics` as follows: use left-arrow key `←` to go to `f`; hit `Delete` key to remove `f` and `x`; hit `c` and `s` keys; and then hit `Enter` key to process the command.

- If you've already hit `Enter` key and then noticed that you had a typing error, you can use `CTRL+p` to go back to your previous command, and then use the method described earlier to remove the typing error. While `CTRL+p` behaves similar to up-arrow key `↑`, you can replicate the behavior of down-arrow key `↓` with `CTRL+n`. You can use `CTRL+p` to go back through the



command history and `CTRL+n` to go forward. You can use `CTRL+p` or `↑` to repeat without change a previous command.

- One more bash shell convenience is *auto filename completion*. Suppose, for example you've a file named `jack.and.jill.went.up.the.hill`, which you'd like to copy to a file named `jj`. In *bash* shell, the command is

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ cp jack.and.jill.went.up.the.hill jj
```

But rather than explicitly typing the long filename, suppose that this is the only file in the current working directory which begins with the sequence of characters `jac`. You can type

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ cp jac
```

and then hit `Tab` key. The shell will then auto complete that file name for you, so that the command line will look like this

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ cp jack.and.jill.went.up.the.hill
```

You'd continue typing destination filename `jj` from the end of auto filename completion and then hit `Enter`.

- Command `history` prints the command history as a list. Each item in the list consists of an numerical id (starting from 1) followed by the command name. Suppose for example, the tenth item in the list is `10 cp jack.and.jill.went.up.the.hill jj`. Rather than typing the entire command, you could have the bash shell automatically execute the command by typing:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ !10
```

followed by `Enter` key. If you want the previous command to be re-executed, just type `!!` followed by `Enter` key.

- These bash shell conveniences save a lot of typing and don't distract from the programming process that you're concentrating on. Whenever you use text editors, email utility programs, image viewers, and so on, check for these conveniences.
- Linux is case sensitive. This means that a file named `cs120.txt` is different from `Cs120.txt` or `CS120.TXT`. Similarly, commands are also case sensitive. The correct command is `history` while typing `HISTORY` into bash shell followed by `Enter` will make the shell print `HISTORY: command not found`.
- Delete files in `/mnt/c/sandbox` and quit the Linux bash shell by typing `exit` at the command prompt. Type `exit` again at the command prompt to close the Windows shell.

## Task 8: Linux filesystem

Just like Windows, the Linux operating system is also built around the concept of a hierarchical *file system* which is used to store the information constituting the system's long-term state. This state includes the operating system kernel itself, executable files for commands and utilities supported by the operating system, configuration information, user data, and various special files for providing controlled access to system hardware and operating system functions. This task is concerned with using the *bash* shell to navigate across a Linux file system and to create, rename, and delete files and directories.

1. Launch Windows command shell by typing `cmd.exe` in `Run` dialog box obtained by pressing `WIN+R` keyboard shortcut. Supposing your user name is `digipen`, the Windows command shell is created in directory `C:\Users\digipen`. Now use command `wt` to switch from Windows to Linux bash shell.
2. When you're in Linux, the entire C drive for Windows is located in `/mnt/c`. Therefore, your current Windows directory `C:\Users\digipen` will be identified in Linux as `/mnt/c/Users/digipen`. Type command `pwd` (for *Print Working Directory*) in the `bash` shell to find your current working directory:

```
1 digipen@dit0701sg:/mnt/c/Users/digipen$ pwd
2 /mnt/c/Users/digipen
```

The `bash` shell will print your current working directory as `/mnt/c/Users/digipen`. Notice that Linux uses forward-slash separator `/` in path names while Windows uses backslash separator `\`.

3. In Linux, `man` (for *manual* pages) is the command used to interface with the online reference and manual pages. Use command `man` to find out what commands `ls` and `cd` do and what options (or flags or extensions) are available.
4. Using command `cd`, change your current working directory to `c:/sandbox`, like this:

```
1 digipen@dit0701sg:/mnt/c/Users/digipen$ cd /mnt/c/sandbox
```

5. Just as with Windows, bash shells use period character `.` to indicate the current working directory while a string of two period characters `..` indicate the parent directory. Use two period characters to change your current working directory to `/mnt/c`.
6. Also just as with Windows, Linux recognizes wildcards such as `?` and `*` to specify groups of files. Recall that wildcard `?` represents *zero or a single* character that a group of files have in common, while wildcard `*` represents *zero or more* characters that a group of files have in common.
7. Change your current working directory to `/mnt/c/sandbox`.
8. In your current working directory, create a directory labeled `CS120` using command `mkdir`. Confirm the directory was created using command `ls`.
9. Using commands `mkdir` and `cd`, create subdirectories `01`, `02`, and `03` in directory `CS120`. You should now have the following hierarchies: `CS120/01`, `CS120/02`, and `CS120/03`.
10. Change your current working directory to `CS120/01`. Launch Code just as you would in a Windows shell. If this is being done the very first time, the operating system will install certain software components related to Code. Once Code launches, create a text file named `foo.txt`. Insert any legible text you wish into this file and shutdown Code.
11. On many occasions, you'd prefer to read contents of text files without resorting to an editor such as Code. Linux provides two convenient commands `cat` and `more` to do exactly that. You can display contents of a text file using command `cat`:

```
1 digipen@dit0701sg:/mnt/c/sandbox/CS120/01$ cat foo.txt
```

If the file consists of more lines of text than can fit your shell window, you can display its contents one screen at a time by piping the output of command `cat` to command `more`:

```
1 | digipen@dit0701sg:/mnt/c/sandbox/CS120/01$ cat foo.txt | more
```

12. Change current working directory from `CS120/01` to `CS120/02`. Next, use command `cp` to copy file `foo.txt` from subdirectory `01` to the current directory as a file named `bar.txt`. If you're confused about command `cp`, use command `man cp | more` to get more information about `cp`.
13. From the current working directory, use command `mv` to move the file `bar.txt` to file `foobar.txt` in directory `CS120/03`. Remember to use command `man mv | more` if you're unable to complete this task.
14. Change your current working directory to `CS120/03` and use command `ls` to check if file `foobar.txt` exists. The file should exist if the previous steps were completed correctly. The contents of the file can be checked using command `cat`.
15. To delete hierarchies `CS120/01`, `CS120/02`, and `CS120/03`, change your current working directory to `/mnt/c/sandbox` and use command `rm` like this:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ rm CS120
2 | rm: cannot remove 'CS120': Is a directory
```

As seen from the error message, `rm` is the correct command to delete files and directories but to remove directories, command `rm` must be provided additional options. Use command `man` (for manual) to research command `rm`:

```
1 | digipen@dit0701sg:/mnt/c/sandbox$ man rm | more
```

16. After removing the `CS120` directory hierarchy, quit the Linux shell by typing `exit` at the command prompt. Type `exit` again at the command prompt to close the Windows shell.

## Task 9: Getting started with gcc

The purpose of this portion of the tutorial is to expose you to compiler tools that will be used in this and other programming courses. Lab computers provide [GNU Compiler Collection](#) (GCC) - a free collection of compilers and interpreters for C, C++, Python, Fortran, Go, and other languages. GCC is portable - it runs on most platforms available today, and can produce output for many types of processors. GCC is not only a native compiler - it can also cross-compile any program, producing executable files for a different system from the one used by GCC itself. This allows software to be compiled for embedded systems such as entertainment devices and game consoles which are not capable of running a compiler. All of this is good news because you will be learning and using a C compiler that is widely used in a variety of academic, industrial, and professional settings.

1. Begin by launching a Windows shell, changing your directory so that current working directory is `C:\sandbox`, and then launching the Linux bash shell. Using command `mkdir`, create directory `Tab01` and change the current working directory in bash shell to directory `Tab01`. Launch Code from the bash shell and edit *source file* `prog1.c` with the following C code. I urge you to not copy-and-paste!!! It is important that you read-and-type and potentially make mistakes that can be explained and corrected. This is the only way to gain experience when you're starting out as a beginner with little or no exposure to programming!!!

```
1 | /*
```

```

2  This is the smallest legal C code.
3  The program doesn't take any input from the user.
4  Nor, does the program print any output to the user.
5  In other words, the program does nothing useful.
6  These lines between the tokens on lines 1 and 9 constitute
7  a comment. Comments are meant for human readers of the file
8  and are not consumed by the compiler.
9  */
10
11 /*
12 Every C program requires a function called main that is the entry
13 point of the program. That is, your computer's CPU will begin
14 executing the program from the first line (statement) of function
15 main. In this main function, the statement simply returns the
16 integer value 0 back to the operating system indicating
17 successful program execution.
18 */
19 int main(void) {
20     return 0;
21 }
22

```

Notice empty line 22. Every legal C source file must have such an empty line at the end.

2. Preprocessing, compiling, assembling, and linking are the various stages involved in converting the source code in `prog1.c` to an executable program. Rather confusingly, most programmers refer to these distinct stages using the umbrella term *compiling* (confusing because compiling is also one of the stages) or *compiler driver* or *compiler toolchain*. `gcc` is the name of the command that implements these distinct steps using the GCC C compiler. Compile file `prog1.c` with `gcc`, like this:

```

1  digipen@dit0701sg:/mnt/c/sandbox/lab01$ gcc -std=c11 -pedantic-errors -
    wstrict-prototypes -Wall -Wextra -Werror prog1.c -o prog1.out

```

This compiles source code in `prog1.c` to executable machine code and stores it in executable file `prog1.out`. The output file for the machine code is specified using `-o` option. This option is usually given as the last argument on the command line. If it is omitted, the output is written to a default executable file called `a.out`. Note that if a file with the same name as the executable file already exists in the current directory it will be overwritten.

Option `-std=c11` specifies the base ISO C11 standard. Options `-wstrict-prototypes`, `-Wall`, and `-Wextra` individually turn on different types of compiler diagnostic warnings while options `-pedantic-errors` and `-Werror` convert these warnings to full-fledged errors. Options are not required for `gcc` to produce warnings and errors for obvious syntax related issues. The options listed above produce diagnostic warnings in non-obvious situations that could potentially lead to hard-to-detect bugs. Further, options `-std=c11` and `-pedantic-errors`, and `-wstrict-prototypes` aim to make the best possible attempt to ensure C code conforms as a subset of C++. Since compiler warnings are an essential aid in detecting problems when programming in C and C++, you are required to **always use** these options! The biggest cause of students getting a C grade rather than an A grade can be attributed to not following this advice. An itemized listing of these options is detailed in this [section](#).

3. Compiling `prog1.c` should not produce any warnings nor errors if the program was typed exactly as shown. Source code which does not produce any warnings is said to *compile cleanly*. To run the program, type the pathname of the executable like this:

```

1 digipen@dit0701sg:/mnt/c/sandbox/lab01$ ./prog1.out
2 digipen@dit0701sg:/mnt/c/sandbox/lab01$

```

This loads executable file `prog1.out` into memory and causes the CPU to begin executing the instructions contained within it. The period character `.` indicates *current working directory* and path `./prog1.out` refers to file `prog1.out` in the current working directory. Since the program doesn't do anything, you'll not see any output generated by it.

Recall that when you executed Windows executable `stats-win.exe` in the Windows shell, you did not have to provide the path to the current directory. Instead, you were able to execute the command like this:

```

1 C:\sandbox>stats-win.exe < input.txt
2 9.67 9.00
3

```

This is because unlike Linux bash shell, the Windows shell automatically specifies a path to the current directory.

## Task 10: Second C program

Let's now write a classic program that prints `Hello world` to `stdout`.

1. Create a source file `prog2.c` and type the following C source code:

```

1 // while the tokens /* and */ can be used to specify multi-line
2 // comments, token // is used to specify single-line comment.
3 #include <stdio.h> // contains function prototype of printf
4
5 int main(void) {
6     printf("Hello world\n");
7     return 0;
8 }
9

```

2. Use instructions from the previous task to compile source file `prog2.c` and store the executable in `prog2.out`.
3. Executing program `prog2.out` should generate the following output:

```

1 digipen@dit0701sg:/mnt/c/sandbox/lab01$ ./prog2.out
2 Hello world
3 digipen@dit0701sg:/mnt/c/sandbox/lab01$

```

Characters `\n` in string `"Hello world\n"` represents a newline. Note the two lines of output generated by the program - the sequence of characters in string `"Hello world"` followed by a newline. If you get compilation errors, then you must have typed the source code incorrectly. Go back and review the source code to make sure there are no syntax errors.

## Task 11: Debugging - finding errors in program

As mentioned earlier, compiler warnings are an essential aid when programming in C. This is demonstrated in the sequence of steps below.

1. Create a source file `prog3.c` and type the following C source code:

```

1  /*
2  This program will print the text "DigiPen's zip code is 13960".
3  Rather than printing a constant value, we store the zip code in
4  variable called zcode so that whenever the zip code changes, the
5  printf function will print the updated zip code.
6  */
7
8  #include <stdio.h> // contains function prototype of printf
9
10 int main(void) {
11     // zip is a variable; 139660 is an integer literal
12     // here, we're defining the variable and also initializing
13     // the variable to have value 139660
14     int zip = 139660;
15
16     printf("DigiPen's zip code is %d.\n", zip);
17     return 0;
18 }
19

```

2. Compile the program and store the executable in `prog3.out`.
3. Executing program `prog3.out` should generate the following output:

```

1  digipen@dit0701sg:/mnt/c/sandbox/lab01$ ./prog3.out
2  DigiPen's zip code is 139660.
3  digipen@dit0701sg:/mnt/c/sandbox/lab01$

```

4. Change the initializer for variable `zip` to various integral values (such as your hometown zip code), recompile, run the executables, and observe the outputs generated by these programs.
5. To demonstrate that compiler warnings are an essential aid to debugging C/C++ programs, a subtle bug is introduced in the code that makes function `printf` misbehave while the code itself is successfully compiled by `gcc`. On line 16 (of the code above), change the *integer* format `%d` to a *floating-point* (that is, having fractions) format `%f`. This will introduce an error into the program's behavior because the programmer should specify her intent to print an integral zip code by using integer format `%d`. However, the programmer has mistakenly typed the format specifier as `%f` which will lead to function `printf` printing floating-point values rather than integer values.
6. Compile `prog3.c`. Because of the subtle error you've introduced, `gcc` generates some diagnostic messages before successfully compiling the source file:

```

1 digipen@dit0701sg:/mnt/c/sandbox/lab01$ gcc -std=c11 prog3.c -o prog3.out
2 prog3.c: In function 'main':
3 prog3.c:16:34: warning: format '%f' expects argument of type 'double',
  but argument 2 has type 'int' [-Wformat=]
4     16 |     printf("DigiPen's zip code is %f.\n", zip);
5         |                                     ~^      ~~~
6         |                                     |      |
7         |                                     double int
8         |                                     %d
9 digipen@dit0701sg:/mnt/c/sandbox/lab01$

```

7. Suppose you ignore these diagnostic messages and execute the program. The program generates incorrect output:


```

1 digipen@dit0701sg:/mnt/c/sandbox/lab01$ ./prog3.out
2 DigiPen's zip code is 0.000000.
3 digipen@dit0701sg:/mnt/c/sandbox/lab01$

```

The incorrect format specifier `%f` causes the output to be corrupted because function `printf` is passed an integer instead of a floating-point number. Integers and floating-point numbers are stored in different formats in memory, leading to a spurious result. The actual output of your program depends on the specific platform and environment and may differ from the one shown above.

8. The moral of the story here is to always read and correct diagnostic messages from `gcc`. In this instance, the warning message indicates a format string has been used incorrectly in file `prog3.c` at line 16. Messages produced by `gcc` always have the form `file:line-number:message`. The compiler distinguishes between *error messages*, which prevent successful compilation, and *warning messages* which indicate possible problems but do not stop the program from compiling. In this case, the correct format specifier should have been `%d` because variable `zip` is of type `int`. Page 37 of the text contains a partial list of allowed format specifiers. The entire list is available [here](#).

Although the compiler has flagged warning messages for the text in line 16, it still compiles the incorrect program `prog3.c` into an executable. This is a problem because the incorrect executable could be distributed for use by others. For example, you could upload this buggy program as your submission for an assignment causing your grade to be a . What you would really want is for `gcc` to convert these warnings to errors. Compared to warnings, errors are catastrophic - the compiler cannot or will not generate an executable. This is possible by using an additional option `-Werror`:

```

1 digipen@dit0701sg:/mnt/c/sandbox/lab01$ gcc -std=c11 prog3.c -o
  prog3.out
2 prog1.c: In function 'main':
3 prog1.c:16:34: error: format '%f' expects argument of type 'double', but
  argument 2 has type 'int' [-Werror=format=]
4     16 |     printf("DigiPen's zip code is %f.\n", zip);
5         |                                     ~^      ~~~
6         |                                     |      |
7         |                                     double int
8         |                                     %d
9 cc1: all warnings being treated as errors
10 digipen@dit0701sg:/mnt/c/sandbox/lab01$

```



Notice that `gcc` has converted the diagnostic warning message to an error message and quit the compilation process without creating the executable file.

The moral of the story is to always use the `-werror` option when you compile. The best defensive technique you could adopt is to always use all these options `-std=c11`, `-pedantic-errors`, `-strict-prototypes`, `-Wall`, `-Wextra`, and `-werror` every time you compile, irrespective of whether you're programming for fun or for your project or when submitting assignments. Although this tutorial has not illustrated the use of options other than `-werror`, their uses will be necessary in more complex programs.

## Task 12: Submission and automatic evaluation

Create a source file `q.c` and type the code on page 15 of Lecture 2's presentation deck.

### Testing

Compile and link your source file `q.c` using the full suite of required `gcc` options:

```
1 | digipen@dit0701sg:/mnt/c/sandbox/tut02$ gcc -std=c11 -pedantic-errors -
   | wstrict-prototypes -Wall -Wextra -werror q.c -o q.out
```

If your source file doesn't compile and link successfully, use the diagnostic messages from `gcc` to determine the errors in your source file, repair those errors, and recompile. Repeat this process until you've successfully generated executable `q.out`. Now, run executable `q.out` like this to display the prompt `Enter base and power:` on ``stdout`:

```
1 | digipen@dit0701sg:/mnt/c/sandbox/tut02$ ./q.out
2 | Enter base and power:
```

Provide input to the program by typing integer values 3 and 4 followed by the `Enter` key:

```
1 | digipen@dit0701sg:/mnt/c/sandbox/tut02$ ./q.out
2 | Enter base and power: 3 4
```

The program will generate the following output:

```
1 | 3 ^ 4 is: 81
```

Test the program's behavior by evaluating the exponents of a variety of base numbers.

### Submission and automatic evaluation

1. In the course web page, click on **Lab 1 Submission Page**. Next, click on the **Submission** button: you'll see something like this:

csd1120f20-b.sg

[Dashboard](#) / [Courses](#) / [Singapore Campus](#) / [Computer Science](#) / [csd1120f20-b.sg](#) / [14 September - 20 September](#) / [Lab 1](#)

Description Submissions list Similarity Test activity

Submission Edit Submission view Grade Previous submissions list

▼ Submission

Comments

q.c

Choose a file... Maximum size for new files: 5MB

You can drag and drop files here to add them.

Submit Cancel

2. Drag and drop your submission file `q.c` and click the **Submit** button followed by the **Continue** button. The automatic grading will begin: your submission `q.c` will first be compiled and linked to create an executable file. Next, the executable is run and its output is compared with the correct output:

csd1120f20-b.sg

[Dashboard](#) / [Courses](#) / [Singapore Campus](#) / [Computer Science](#) / [csd1120f20-b.sg](#) / [14 September - 20 September](#) / [Lab 1](#)

Description Submissions list Similarity Test activity

Submission Edit Submission view Grade Previous submissions list

Submitted on Thursday, 26 August 2021, 10:04 PM ([Download](#)) ([Evaluate](#))

Automatic evaluation[-]

**Proposed grade: 100 / 100**

**Compilation[-]**  
 Compiling your submitted source file q.c ...  
 Linking object files with C standard library ...  
 Executable successfully created

**Comments[-]**  
[\[+\]Summary of tests](#)

3. The automatic grader will assign a grade for this and future programming assignments and labs based on the whether your source code successfully compiles, successfully links, and successfully generates the correct output. If you're not satisfied with your grade, you should go back and repeat the entire process: amend your source code to ensure compilation and linking are successful, and that your program generates the correct output. You can repeat this process as many times as necessary up until the submission deadline.
4. All submissions will also provide a set of rubrics (think of a rubric as a grading rule) specifying how your grade will be computed. The rubrics for this submission are:
1. *F* grade if `q.c` doesn't compile with the full suite of `gcc` options.
  2. *F* grade if `q.c` doesn't link to create an executable.
  3. *F* grade if the program's output is not correct.
  4. *A+* grade if program's output is correct output.

## Review

After completing this tutorial, you should be comfortable with the basic knowledge and skills required to use the programming environment for this course, that is, navigate through your computer's filesystem using both Windows shell and Linux bash shell; edit, compile, and link source files using the `gcc` compiler toolchain; and submitting source code for automatic grading. In particular,

1. You should be able to launch the Windows shell and implement basic file management.
2. You should be able to launch and use Code to edit source code from Windows shell.
3. You should be able to launch the Linux bash shell from the Windows shell and implement basic file management.
4. You should be able to launch and use Code to edit source code from Linux bash shell.
5. You understand the abstraction of streams. You know the meaning of `stdin` (standard input) and `stdout` (standard output) streams.
6. You understand the meaning of stream redirection. You know how to redirect program output to a file and redirect program input from a file.
7. You understand the idea of pipes and piping so that individual programs can be strung together to create a toolchain.
8. You understand the edit-compile-execute cycle using `gcc` compiler toolchain. You know the purpose of `gcc` issuing warnings and you know the various options available to programmers. You understand option `-o` used with `gcc` that allows programmers to provide names other than `a.out` to executables.
9. You understand how to write, compile, and link code that uses function `printf` to write specific text to `stdout`.
10. You understand the steps involved in submitting source code for tutorials and programming assignments.
11. You should have practiced the following Windows shell and Linux bash shell commands:

What does command do?	Windows shell	Linux bash shell
Provide information on commands	<code>help</code>	<code>man</code>
List files	<code>dir</code>	<code>ls</code>
Copy files	<code>copy</code>	<code>cp</code>
Delete files	<code>erase</code> and <code>del</code>	<code>rm</code>
Move files	<code>move</code>	<code>mv</code>
Rename files	<code>ren</code> and <code>move</code>	<code>mv</code>
Print contents of text files	<code>type</code>	<code>cat</code>
Display output one screen at a time	<code>more</code>	<code>more</code> or <code>less</code>
Create directory	<code>mkdir</code> or <code>md</code>	<code>mkdir</code>
Delete directory	<code>rmdir</code> or <code>rd</code>	<code>rm</code>
Change directory	<code>chdir</code> or <code>cd</code>	<code>cd</code>
Print current working directory	<code>chdir</code> or <code>cd</code>	<code>pwd</code>

What does command do?	Windows shell	Linux bash shell
Create links between files	<code>mklink</code> (not covered)	<code>ln</code>
Clear shell screen	<code>cls</code>	<code>clear</code>
Time of day	<code>time</code>	<code>time</code>
Date	<code>date</code>	<code>date</code>

## GCC C compiler options

This is an itemized listing of `gcc` options that are required whenever you write code. Your programming submissions will always be compiled using *all* of the listed options.

- `-std=c11`: Information about the ISO C11 standard is available in the [C11 N1570 standard draft](#).
- `-pedantic-errors`: Gives an error when base standard C11 requires a diagnostic message to be produced. C89 allows the declaration of a variable, function argument, or structure member to omit the type specifier, implicitly defaulting its type to `int`. Although, legal in C89, this is considered illegal in C99, C11, and C++:

```

1  #include <stdio.h>
2  int main(void) {
3      static x = 10; // notice the lack of type specifier int
4      printf("x: %d\n", x);
5      return 0;
6  }
7

```

However, compiling this code with a C11 compiler, as in: `gcc -std=c11 tester.c` elicits only a warning message. However, compiling the same code with the `-pedantic-errors` option produces an error.

According to Section 5.1.1.3 of the [C11 N1570 standard draft](#),

1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.<sup>9)</sup>

9. The intent is that an implementation should identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. It may also successfully translate an invalid program.

Based on the above text, `-pedantic-errors` cannot be solely used to check programs for strict ISO C conformance. The flag finds some non-ISO practices, but not all - only those for which ISO C *requires* a diagnostic, and some others for which diagnostics have been added.

- `-Wall`: This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning). Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary

or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning. Some of them are enabled by `-wextra` but many of them must be enabled individually. The entire list of warning flags enabled by `-Wall` for GCC can be found [here](#).

- `-wextra`: This enables some extra warning flags that are not enabled by `-Wall`. The entire list can be found [here](#).
- `-Werror`: Converts diagnostic messages generated by the base compiler and warnings generated by flags `-Wall` and `-wextra` to errors. This is a necessary feature for generating cleanly compiled code that doesn't generate any warning messages.
- `-Wstrict-prototypes`: It is legal in all ISO C standards to specify a function declaration as:

```
1 | int f();
```

C89, C99, and C11 compilers read the above declaration as `f` is a function that takes unknown number of parameters with unknown types and return a value of type `int`. This means the following code in a source file `test.c` will compile and link with undefined behavior when executed.

```
1 | #include <stdio.h>
2 |
3 | int foo();
4 |
5 | int main(void) {
6 |     int x = 2, y = 4, z = 6;
7 |     printf("%d + %d = %d\n", x, y, foo(x, y));
8 |     printf("%d + %d + %d = %d\n", x, y, z, foo(x, y, z));
9 |     return 0;
10 | }
11 |
12 | int foo(int i, int j) {
13 |     return i + j;
14 | } // don't forget to add empty line after this
```

The code will compile with GCC and Clang (and also with Microsoft Compiler) without any diagnostic messages:

```
1 | $ gcc -std=c11 -pedantic-errors -Wall -wextra -Werror test.c
```

However, the same code will not compile with a ISO C++ compiler:

```
1 | $ g++ -std=c++11 test.c
```

since the declaration `int f();` in ISO C++ compilers indicates that `f` is a function that takes no parameters and returns a value of type `int`.

To ensure compatibility with ISO C++ standards, the `-Wstrict-prototypes` must be used with GCC and Clang to ensure that `test.c` doesn't successfully compile.