Assignment 1: C++ and Unix

1 Introduction

1.1 Assignment Goals

This assignment introduces C++ and how to interact with the shell and the file system in C++ programs.

1.2 Group Size

For this assignment, the maximum group size is 2.

1.3 Due Date

This assignment is due Mar 17 2021 at 11:59pm.

This assignment has a sliding late window of partial credit for late assignments. The late window ends at precisely 18/03/2021 16:59. If you submit after the due date but before the close of the late window, your assignment grade will be scaled from 100% to 50% on a linear time scale from the due time to the window end.

1.4. Identifying authorship

Each author should be identified on a separate comment line with the following format:

// Copyright year person1name email1@sdu.edu
// Copyright year person2name email2@sdu.edu

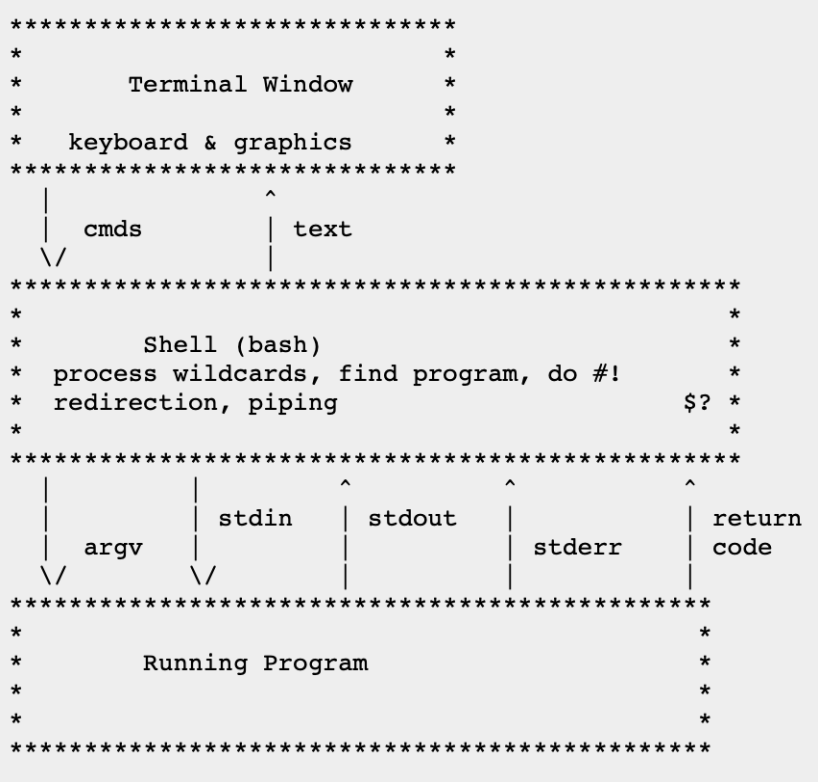
where personXname is replaced by your name (in any format).
The line must start with // and must end with your full SDU email.

2 Background

In unix, the shell or terminal provides an environment in which the program can interact with the operating system and user. One of the ways it does this is to provide input and output facilities to the terminal. The other way is by passing programs configuration options on the command line. These are called “command-line arguments.”

2.1 Programs, Shell, Terminal

The following diagram summarizes the way the terminal window, the shell program, and the running program interact with each other in a Unix system such as the devbox.



2.2 Input and Output

The input and output “streams” are

- standard input (stdin): the program can read whatever is “typed” at the terminal
- standard output (stdout): normal output from the program goes here.
- standard error (stderr): when errors occur during execution of a program, the program normally sends the error messages here.

By default, both stdout and stderr will appear together on the terminal, but they can be redirected to illustrate the difference

2.3 Command-line arguments

Most unix programs reserve stdin and stdout for data. Configuration options are typically specified as arguments specified on the “command line.”
Normally, ls lists all files in the current directory. With command line arguments, this behavior can be modified:
Examples:
List the files in the directory /media

ls /media
Compile splitargs.cpp and make an executable called splitargs
g++ splitargs.cpp -o splitargs
In this example, there are three arguments:

- argument 1 is the name of the program to compile
- argument 2 is a symbol that g++ recognizes which means “the next argument is the name of the output”
- argument 3 is the name of the executable file to create

2.4 Executable programs in Unix

There are several different types of “applications” or “programs” in all operating systems. Here, we focus on how these work in Linux / Unix environments.
The word “executable” is both a noun (thing) and an adjective (description).
A “true” executable is a file containing machine code. It will be loaded into memory by the OS and the code will be executed.
When C++ programs are compiled, the result is a true executable program.

2.4.1 Executable permissions

The word “executable” as an adjective refers to a permission setting on a file, which indicates to the operating system that this file is intended to be run as an executable (the noun).
The permissions for a file are viewed with ls like this:
ls -l
There are nine (3x3) permission flags: three access modes (read, write, and execute) for three entities (user, group, other).
and they can be changed using chmod like this:
chmod 755 a.out
or
chmod o+x recent
Type man ls or man chmod to see all the options for these programs.

2.4.2 Scripts As Executables

A text file containing bash commands or python commands can also be executed, in one of two ways, explicitly or implicitly.

2.4.2.1 Explicit scripting

In this method, the scripting program is directly called from the terminal, and the name of the script is provided to that program as a command line argument.
Here are examples:
bash my_commands.sh
or
python my_commands.py

in which case the files my_commands.sh and my_commands.py do not have to have executable permisions (the programs bash and python have the permissions!)

2.4.2.2 Implicit scripting

If you want your script to look more like a “real program” (which we call implicit scripting to distinguish it from the case above) here is how it is done:

1. Change the name of the program to my_commands since the Unix convention is that executables do not have extensions.
2. Change the permissions of the program using chmod usually to 755
3. Add the following line to the beginning of your program

#!/usr/bin/env python
This is called a shebang statement and it tells the OS to run the rest of the program through the interpreter executable that follows.
The program env looks up the location of the program python and runs it.
Alternatively, you can also add
#!/home/ece-student/anaconda3/bin/python
but this is less portable.

2.5 Quiz

1. Does the shell listen to your keystrokes or does it wait until you hit the enter / return key? Prove your answer with an example.

3 The Assignment

3.1 Command Line Argument Handling (2 points)

Write a C++ program splitargs.cpp that outputs the first four arguments to stdout and any additional arguments, if any, to stderr.
Each argument should appear on a line by itself.
So, for example, if the program is run like this:

splitargs one two 3 four five six

then the output to stdout will be

one
two
3
four

and the output to stderr will be

five
six

2.1.1 Hints

Caution: both python and C++ include an extra argument as “argument 0”. This extra argument is the name of the program. These programs should not print out argument 0.

3.2 Numerical Functions (5 points)

Your assignment is to write the following functions:

bool is_happy(int x)
double product_of_positives(vector<double>)
int product_of_positives(vector<int>)
vector<int> proper_divisors(int n)
string add(const string& num1, const string& num2)

3.2.1 The is_happy function

return True if x is a happy number, otherwise False
From: http://en.wikipedia.org/wiki/Happy_number
Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers, while those that do not end in 1 are unhappy numbers (or sad numbers).

3.2.2 The product_of_positives function

return the product of all the positive numbers in the sequence

3.2.3 The proper_divisors function

return all proper divisors of n in a tuple and in numerical order

3.2.4 The add function

Write a function that adds two non-negative integers represented as strings, returning the result as a string. This is the function signature:
string add(const string& num1, const string& num2)

The values num1 and num2 can represent arbitrarily long integers, so all the calculations must be done without using integer types (except for calculating each digit.)
There is no need to check for or deal with strings that do not represent non-negative integers.

3.2.5 Submission and file structure

Your submission must be in the form of file called “numbers.cpp”
For your convenience, the functions are defined in a starter file here:
numbers_template.cpp
For the numbers assignment, you should include <vector> and <string> but no other libraries.
Here is an example of how to test your program:
testing_numbers.cpp

3.3 Oldest files (3 points)

Write a C++ program oldest.cpp which outputs the oldest files in the current directory, starting with the file modified furthest in the past, and working towards the newer files. Your program should accept one command line argument which represents the number of files to list. Here is an example:
> oldest 3
nine
eight
seven
[Assume there are nine files, one being the most recent and nine being the oldest]

For this part and only for part 3.3 you can use other libraries.

3.4 No brackets

You must not use brackets, i.e. [ ] anywhere in these programs.
Use the at method instead.