MODULES

Workshop 1 Parts 1 and 2
(v0.9)

In process of doing your first workshop, in part 1 you are to sub-divide a program into modules, compile each module separately and construct an executable from the results of each compilation. In Part 2 you are to write a modular program based on your knowledge of ipc144 subject.

LEARNING OUTCOMES

Upon successful completion of this workshop, you will have demonstrated the abilities to:

- organize source code into modules, using header and implementation files;
- compile and link modular programs;
- distinguish the contents of a header and an implementation file;
- describe to your instructor what you have learned in completing this workshop.

SUBMISSION POLICY

The workshop is divided into two coding parts and one non-coding part:

- *Part 1*: worth 50% of the workshop's total mark, is due on **Wednesday at 23:59:59** of the week of your scheduled lab.

- *Part 2*: worth 50% of the workshop's total mark, is due on **Sunday at 23:59:59** of the week of your scheduled lab. Submissions of *part 2* that do not contain the *reflection* are not considered valid submissions and are ignored.

- *reflection*: non-coding part, to be submitted together with *Part 2*. The reflection doesn't have marks associated to it but can incur a **penalty of max 40% of the whole workshop's mark** if your professor deems it insufficient (you make your marks from the code, but you can lose some on the reflection).

If at the deadline the workshop is not complete, there is an extension of **one day** when you can submit the missing parts. **The code parts that are submitted late receive 0%.** After this extra day, the submission closes; if the workshop is incomplete when the submission closes (missing at least one of the coding or non-coding parts), **the mark for the entire workshop is 0%**.

Every file that you submit must contain (as a comment) at the top **your name**, **your Seneca email**, **Seneca Student ID** and the **date** when you completed the work.

If the file contains only your work, or work provided to you by your professor, add the following message as a comment at the top of the file:

I have done all the coding by myself and only copied the code that my professor provided to complete my workshops and assignments.

If the file contains work that is not yours (you found it online or somebody provided it to you), **write exactly which part of the assignment are given to you as help, who gave it to you, or which source you received it from.** By doing this you will only lose the mark for the parts you got help for, and the person helping you will be clear of any wrongdoing.

COMPILING AND TESTING YOUR PROGRAM

All your code should be compiled using this command on `matrix`:

```
g++ -Wall -std=c++11 -g -o ws file1.cpp file2.cpp ...
```

- `-Wall`: compiler will report all warnings
- `-std=c++11`: the code will be compiled using the C++11 standard
- `-g`: the executable file will contain debugging symbols, allowing *valgrind* to create better reports
- `-o ws`: the compiled application will be named `ws`

After compiling and testing your code, run your program as following to check for possible memory leaks (assuming your executable name is `ws`):

```
valgrind ws
```

To check the output, use a program that can compare text files. Search online for such a program for your platform, or use *diff* available on `matrix`.

PART 1 (50%)

ORIGINAL SOURCE CODE (THE WORDSTAT PROGRAM)

`wordStat` is a program that reads a text file from standard input and analyzes and reports the number of words and their occurrences in the text file.

To feed the text file into the program through standard input we need to use the redirection operator (recall ULI101) of the operating system (that is " < ").

Using command line, if the executable of the program is called "ws", and we need to analyze a text file called "text.txt", we would do so with the following command in **Matrix** (or another Linux command line interface):

```
ws < text.txt <ENTER>
```

See the instructions below to find out how to setup your **Visual Studio** to do the same thing.

> **NOTE**: Installation guides for preparing your computer for the subject can be found in this playlist:
> https://www.youtube.com/playlist?list=PLxB4x6RkyIosAh1of4FnX7-g2fk0MUeyc
> **OR:**
> **https://tinyurl.com/244setup**

RETRIEVE THE ORIGINAL PROGRAM:

Workshop Steps:
(if you have not followed the Installation guides to prepare your computer go to **Using ZIP DOWNLOAD**)
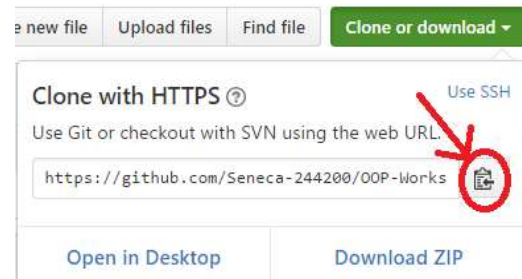
USING GIT:

a. Open https://github.com/Seneca-244200/BTP-Workshops and click on "Clone or download" Button; this will open "Clone with HTTPS" window.

b. Copy the https URL by clicking on the button on the right side of the URL:

c. Open File Explorer on your computer and select or create a directory for your workshops.

Now Clone (download) the original source code of `w1p1.cpp` (Workshop1) from GitHub in one of the following three ways: (methods 1 and 2 are preferred)

1. Using `TortoiseGit`:

    a. Right click on the selected directory and select "Git Clone":
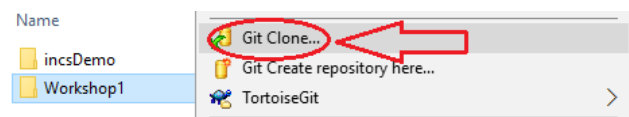
    This will open the "Git Clone" window with the URL already pasted in the "URL" text box; if not, paste the URL manually.

    b. Click on OK.

    This will create on your computer a clone (identical directory structure) of the directory on `Github`. Once you have cloned the directory, you can open the directory `BTP-Workshops/WS01` and start doing your workshop.
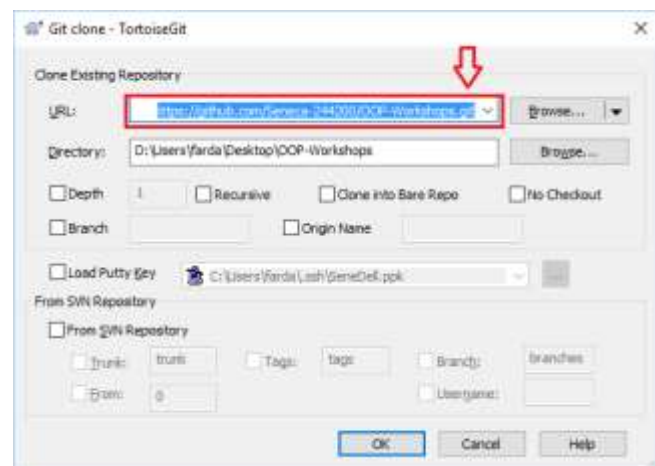
Note that you will repeat this process for all workshops and milestones of your project in this subject.

> **NOTE**: If your professor makes any changes to the workshop, you can right click on the cloned repository directory and select **Tortoise-Git/pull** to update and sync your local workshop to the one on **Github** without having to download it again. Note that this will only apply the changes made and will not affect any work that you have done on your workshop.

2. Using the command line:

   a. Open the `git` command line on your computer.
   b. Change the directory to your workshops directory.
   c. Issue the following command at the command prompt in your workshops directory:

`git clone` https://github.com/Seneca-244200/BTP-Workshops.git`<ENTER>`

> **NOTE**: The URL for all the workshops are the same throughout the semester. The only thing that changes, is the workshop number.

This will create on your computer a clone (identical directory structure and content) of the `BTP-Workshops` directory on `Github`. Once you have cloned the directory, you can open subdirectory `BTP-Workshops/WS01` and start doing your workshop. Note that you will repeat this process for all workshops and milestones of your project in this subject.

> **NOTE**: If your professor makes any changes to the workshop, you can issue the command **git pull<ENTER>** in the cloned repository directory to update and sync your local workshop to the one on Github without having to download it again. Note that this will only apply the changes made and will not affect any work that you have done on your workshop.
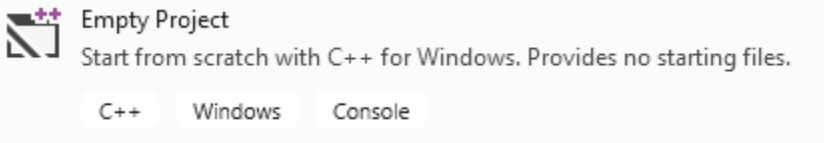
USING ZIP DOWNLOAD:

a. Open
https://github.com/Seneca-244200/BTP-Workshops and
click on "Clone or download"
button and click on
"Download ZIP".
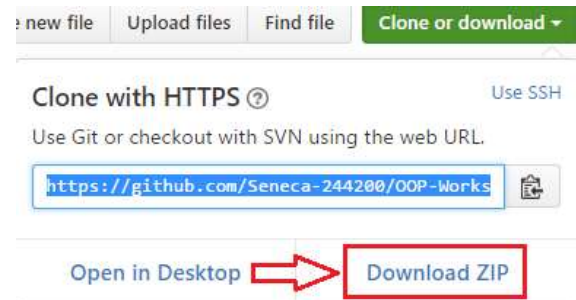
b. This will download to your
computer a zipped file copy of the workshop repository in Github.
You can extract this file to where you want to do your workshop.

> NOTE: Note that, if your professor makes any changes to the
> workshop, to get them you have to separately download another
> copy of the workshop and manually apply the changes to your
> working directory to **make sure nothing of your work is overwritten
> by mistake**.

STEP 1: TEST THE PROGRAM

1. On Windows, using Visual Studio (VS)

a. Open Visual studio 2019 and create an Empty C++ Windows Console
Project:

b. In VS, (if not open already) open Solution Explorer *(click on
View/Solution Explorer)* and then add w1p1.cpp file to your project:
   - Right click on "Source Files"
   - Select "Add/Existing Item"
   - Select w1p1.cpp from the file browser
   - Click on "Ok"

c. To add the command line arguments needed to feed the
"FoxInSocks.txt" file to the program through the standard input do
the following:
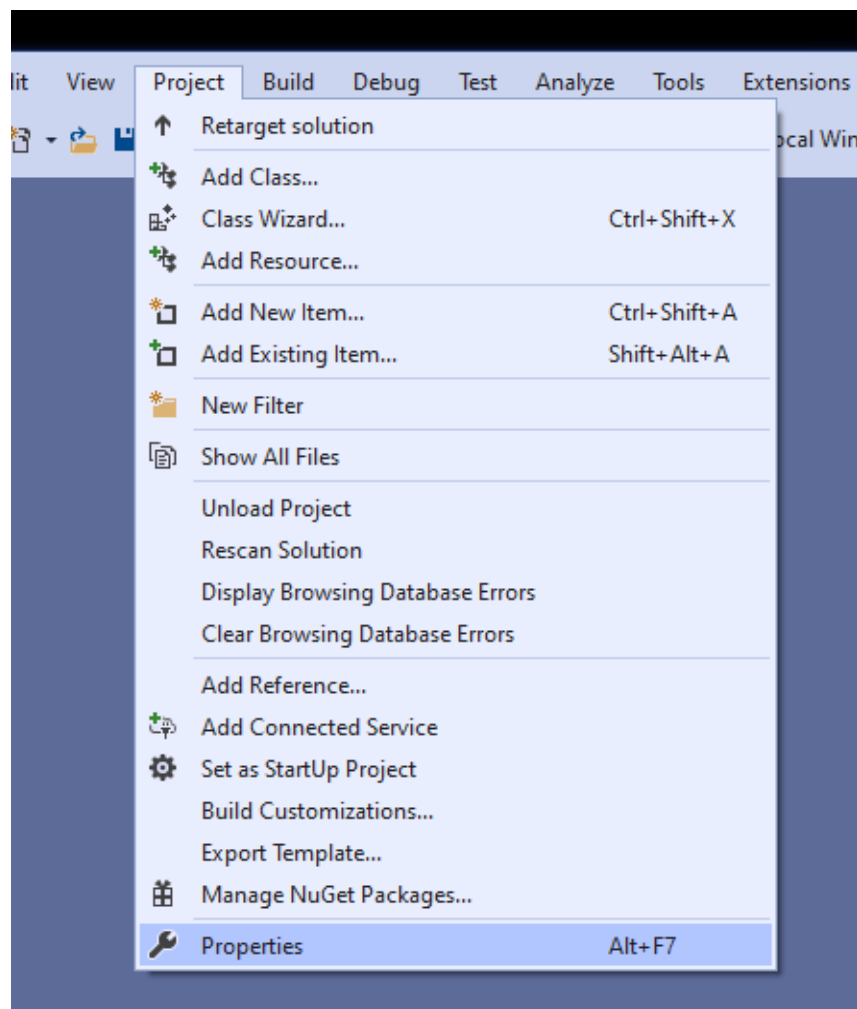
In "Project" menu click on "Properties":

In "Properties" window click on "Debugging" and then in "Command Arguments" section enter: **"< FoxInSocks.txt"** and click on ok.
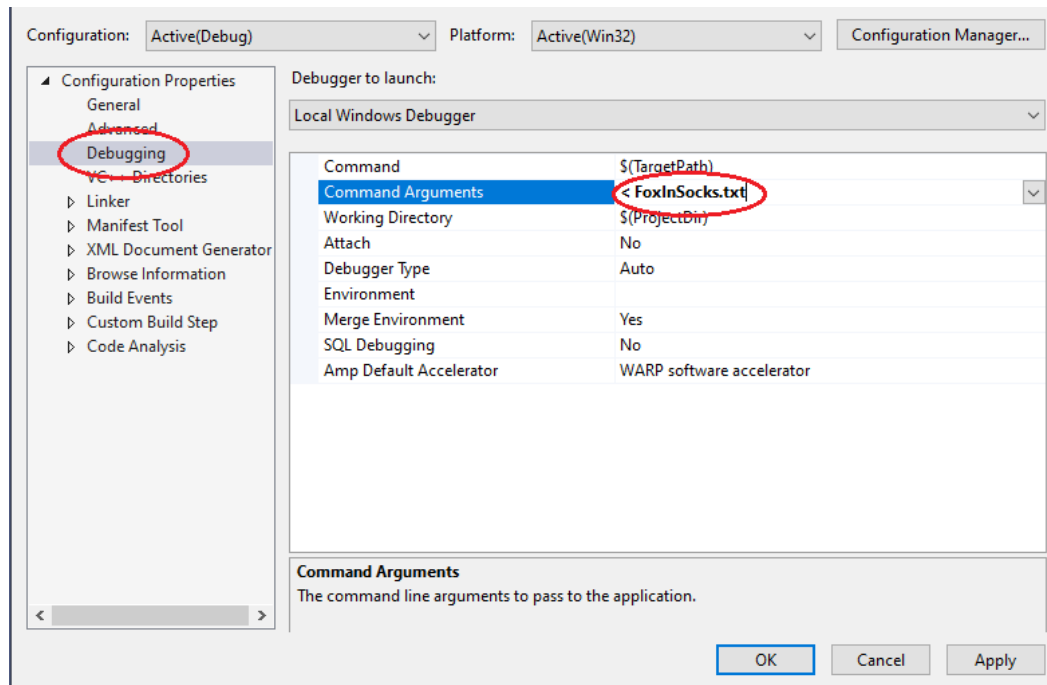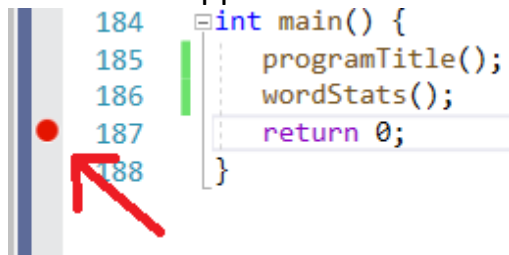

d. Compile and run the program by selecting "Debug/Start without Debugging" or pressing "Ctrl-F5".

Note that doing so the program output will open and close automatically when it is done. This happens because a file is being redirected as standard input.

To pause the program at the end of the main function, so you can see the output do the following:

Move the mouse over the left side of the "return" statement in the main function and add a breakpoint by clicking on the gray area and a red circle appears.

Now you can run the program by selecting "Debug/Start Debugging" or pressing the "F5" button.

This will run the program in debugging mode and pause the execution at the breakpoint (which is the end of the function main). Now you can see the console window with the output before it closes.

2. On Linux, in your `matrix` account.

    e. Upload `w1p1.cpp` and `FoxInSocks.txt` to your matrix account (Ideally to a designated directory for your workshop solutions). Then, enter the following command to compile the source file and create an executable called `w1`:

> **`g++ w1p1.cpp -Wall -std=c++11 -o ws`**`<ENTER>`

- `-Wall`: display all warnings
- `-std=c++11`: compile using C++11 standards
- `-o w1`: name the executable w1

    f. Type the following to run and test the execution:

> **`ws < FoxInSocks.txt`**`<ENTER>`

STEP 2: CREATE THE MODULES

1. On Windows, using Visual Studio (VS)

In solution explorer, add three new modules to your project:

- `WordStat;` A module to hold the main() function and its relative functions. (see below)
- `Word;` A module to hold the functions related to Word processing and analysis.
- `Utils;` A module to hold the general helper functions which have no direct relation to the word analysis in the program.

The `WordStat` module has an implementation (`.cpp`) file but no header file. The Word and Utils modules have both implementation (`.cpp`) and header (`.h`) files:

- Add `Word.h` and `Utils.h` to the "Header Files" directory (right click on "Header Files" and select "Add/New Item" and add a header file)

Make sure you add the **compilation safeguards*** and also have all the code in Word and Utils modules in a <u>namespace</u> called "**sdds**".

IMPORTANT: Note that, you are not allowed to use the "**using**" statement in a header file.

For example, in a header file you are not allowed to write:
```
using namespace std;
```

---

* compilation safeguards refer to a technique to guard against multiple inclusion of header files. It does so by applying macros that check against a defined name:

```
#ifndef NAMESPACE_HEADERFILENAME_H // replace with relevant names
#define NAMESPACE_HEADERFILENAME_H

// Your header file content goes here

#endif
```

If the name isn't yet defined, the **#ifndef** will allow the code to proceed onward to then define that same name. Following that the header is then included. If the name is already defined, meaning the file has been included prior (otherwise the name wouldn't have been defined), the check fails, the code proceeds no further and the header is not included again.

Compilation safeguards prevent multiple inclusions of a header **in a module**. They do not protect against including the header again in a different module (remember that each module is compiled independently from other modules).

Additionally, here is an instructional **video** showing how the compiler works and why you need these safeguards in all of your header files. **Do note that this video describes the intent and concept behind safeguards, the naming scheme isn't the standard for our class. Follow the standard for safeguards as described in your class**. https://www.youtube.com/watch?v=EGak2R7QdHo

- Add `WordStat.cpp`, `Word.cpp` and `Utils.cpp` to the "Source Files" directory (right click on "Source Files" and select "Add/New Item" and add a C++ file)

  The content of WordStat.cpp file should be exactly as following:

```
#include "Word.h"
using namespace sdds;

int main() {
   programTitle();
   wordStats();
   return 0;
}
```

Separate the rest of the functions in `w1p1.cpp` and copy them into Word and Utils modules as you find fit. Copy the body of the functions into the **cpp** files and the prototype into the **header** files.

To test that you have done this correctly, you can compile each module separately, by right clicking on `Utils.cpp` or `Word.cpp` and select `compile` from the menu. If the compilation is successful, most likely you have done it correctly.

> **NOTE**: The equivalent of this on matrix is to add **-c** to the compile command:
>
> **g++ Utils.cpp –Wall -std=c++11 –c**<ENTER>
>
> This will only compile `Utils.cpp` and will not create an executable.

Now remove `w1p1.cpp` from the project. You can do this by right clicking on the filename in solution explorer and selecting remove in the menu (make sure you do not delete this file but only remove it).

Compile and run the project (as you did before in Step 1) and make sure everything works.

2. On Linux, in your `matrix` account.

Upload the five files from earlier (Utils.cpp, Utils.h, Word.cpp, Word.h and WordStat.cpp) to your matrix account and compile the source code using the following command.

g++ Utils.cpp Word.cpp WordStat.cpp -Wall -std=c++11 -o ws<ENTER>

Run the program like before with the **FoxInSocks.txt** and make sure that everything still works properly.

SUBMISSION

If not on matrix already, upload `Utils.cpp`, `Utils.h`, `Word.cpp`, `Word.h`, `WordStat.cpp`, to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account (use your professor's Seneca userid to replace `profname.proflastname`.

~profname.proflastname/submit 200/w1/p1 <ENTER>

and follow the instructions generated by the command and your program.

> **IMPORTANT**: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

PART 2 (50%)

MARK STATS

Write an application that receives series of marks for an assessment. The number of marks can be no less than 5 or more than 50. Each mark can have a value between 0 and 100 (inclusive).

Your application should ask for the name of the assessment (which will not be more than 40 characters), number of marks and then receive all the marks by prompting row of entry for each mark. Then the application should sort the marks from highest to lowest and print the following information:

- Name of the assessment <NEWLINE>
- The comma separated list of the sorted marks <NEWLINE>
- the Average of all the marks <NEWLINE>
- the number of passing marks (marks above or equal to 50) <NEWLINE>

All your number entries must be fool proof.

Your code must reside in the following three modules:

Markstats.cpp: (no header file)

This module must hold only the following code:

Marks.cpp:

This module should hold all your mark related functions

Utils.cpp:

This module should hold all the helper function which no not have any specific logical relation to a mark. (you can use this module in your future workshops).

NOTE: If you are using any libraries form C language in a C++ file, the header files are included as follows:
instead of `#include <stdio.h>` use `#include <cstdio>`.
This rule (removing **.h** from the end and adding a **c** to the beginning) applies to all C header files.

You can find useful information in the following notes from previous semester:

Input functions (Validation section, for fool-proof data entry)
https://ict.senecacollege.ca/~ipc144/pages/content/formi.html

Algorithmes (Sort algorithme)
https://ict.senecacollege.ca/~ipc144/pages/content/sorts.html

For an assignment called "Midterm Test" with mark values as:
34,89,55,23,50,79,100,60,72

Your program must produce the following output: (user entries are in RED)

```
Mark Stats Program.

Please enter the assessment name: Midterm Test
Please enter the number of marks: 3
Invalid value (5<=value<=50), try again: three
Invalid Number, try again: 3marks
Invalid trailing characters, try again: 9
Please enter 9 marks (0<=Mark<=100):
1> 200
Invalid value (0<=value<=100), try again: 34
2> 89
3> 55
4> 23
5> 50
6> 79
7> 100
8> 60
9> 72
Thank you...
Assessment Name: Midterm Test
----------------
100, 89, 79, 72, 60, 55, 50, 34, 23
Average: 62.4
Number of Passing Marks: 7
```

SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload `Utils.cpp`, `Utils.h`, `Marks.cpp`, `Marks.h`, `MarkStats.cpp`, `reflect.txt` to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account (use your professor's Seneca userid to replace `profname.proflastname`:

~profname.proflastname/submit 200/w1/p1 <ENTER>

and follow the instructions generated by the command and your program.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.