

Workshop #1: Modules

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

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

- Part 1 (LAB): A step-by-step guided workshop, worth 50% of the workshop's total mark

Please note that the part 1 section is **not to be started in your first session of the week**. You should start it on your own before the day of your class and join the first session of the week to ask for help and correct your mistakes (if there are any).

- Part 2 (DIY): A Do It Yourself type of workshop that is much more open-ended and is worth 50% of the workshop's total mark.
- *reflection*: non-coding part, to be submitted together with *DIY* part. The reflection doesn't have marks associated with 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).
- Submissions of part 2 that do not contain the *reflection* (that is the **non-coding part**) are not considered valid submissions and are ignored.

Due Dates

Depending on the section you are enrolled in, the submission due day of the week may shift a day or two. Please choose the "-due" option of the submitter program to see the exact due date of your section:

```
~profname.proflastname/submit 2??/wX/pY -due<ENTER>
```

- Replace ?? with your subject code (00 or 44)
- Replace X with Workshop number: [1 to 10]
- Replace Y with the part number: [1 or 2]

Overall workshop due days

- day 1: Workshop open for preview

(If you need to check your program with the submitter, you can use `-feedback` option to test the execution without submission)

- day 2: submission opens for both parts 1 and 2
- day 5: (end of day) Part 1 due
- day 8: (end of day) Part 2 due
- day 9: (end of day) submissions rejected

If at the deadline (end of day 8) 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%.**

Citation

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.

For work that is done entirely by you (ONLY YOU)


If the file contains only your work or the 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.

For work that is done partially by you.

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 is 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.

- Add the citation to the file in which you have the borrowed code
- In the 'reflect.txt' submission of part 2 (DIY), add exactly what is added to which file and from where (or whom).

 This **Submission Policy** only applies to the workshops. All other assessments in this subject have their own submission policies.

If you have helped someone with your code

If you have helped someone with your code. Let them know of these regulations and in your 'reflect.txt' of part 2 (DIY), write exactly which part of your code was copied and who was the recipient of this code.

By doing this you will be clear of any wrongdoing if the recipient of the code does not honour these regulations.

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`: the 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 follows to check for possible memory leaks (assuming your executable name is `ws`):

```
valgrind --show-error-list=yes --leak-check=full --show-leak-kinds=all --track-origins=yes  
ws
```

- `--show-error-list=yes`: show the list of detected errors
- `--leak-check=full`: check for all types of memory problems
- `--show-leak-kinds=all`: show all types of memory leaks identified (enabled by the previous flag)
- `--track-origins=yes`: tracks the origin of uninitialized values (`g++` must use `-g` flag for compilation, so the information displayed here is meaningful).

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`.

Note: All the code written in workshops and the project must be implemented in the `sdds` namespace, unless instructed otherwise.

LAB (50%)

Shopping List is a program that keeps track of your shopping list up to 15 items. You can add items to the list, remove and check the items you bought. Also, you can remove all the checked items and clear the list.

Here is a sample execution of the program

LAB Execution example

```
-->>> My Shopping List <<<--  
1-[ ]Oranges qty:(4)  
2-[ ]Apples qty:(4)  
3-[ ]Bananas qty:(10)  
4-[ ]Frozen Strawberries qty:(1)  
5-[X]Milk 3% qty:(2)  
6-[ ]Milk Skim qty:(1)  
7-[ ]Lundry Detergent liquic qty:(1)  
8-[ ]Lundry Detergent pods qty:(1)
```

```

0- [ ]Lundry Detergent pods qty:(1)
-----
1- Toggle bought Item
2- Add Shopping Item
3- Remove Shopping Item
4- Remove bought Items
5- Clear List
0- Exit
> 1
Item number: 3
-->>> My Shopping List <<<--
1-[ ]Oranges qty:(4)
2-[ ]Apples qty:(4)
3-[X]Bananas qty:(10)
4-[ ]Frozen Strawberries qty:(1)
5-[X]Milk 3% qty:(2)
6-[ ]Milk Skim qty:(1)
7-[ ]Lundry Detergent liquic qty:(1)
8-[ ]Lundry Detergent pods qty:(1)
-----
1- Toggle bought Item
2- Add Shopping Item
3- Remove Shopping Item
4- Remove bought Items
5- Clear List
0- Exit
> 4
Removing bought items, are you sure?
(Y)es/(N)o: y
-->>> My Shopping List <<<--
1-[ ]Oranges qty:(4)
2-[ ]Apples qty:(4)
3-[ ]Frozen Strawberries qty:(1)
4-[ ]Milk Skim qty:(1)
5-[ ]Lundry Detergent liquic qty:(1)
6-[ ]Lundry Detergent pods qty:(1)
-----
1- Toggle bought Item
2- Add Shopping Item
3- Remove Shopping Item
4- Remove bought Items
5- Clear List
0- Exit
> 2
Item name: Tooth Paste
Quantity: 3
-->>> My Shopping List <<<--
1-[ ]Oranges qty:(4)
2-[ ]Apples qty:(4)
3-[ ]Frozen Strawberries qty:(1)
4-[ ]Milk Skim qty:(1)
5-[ ]Lundry Detergent liquic qty:(1)
6-[ ]Lundry Detergent pods qty:(1)
7-[ ]Tooth Paste qty:(3)
-----
1- Toggle bought Item
2- Add Shopping Item
3- Remove Shopping Item
4- Remove bought Items

```

```
5- Clear List
0- Exit
> 0
```

Step 1: *Test the Program*

On Windows, In Visual Studio

- Open Visual Studio 2022 and create an Empty C++ Windows Console Project:



- Move the files `wlp1.cpp` and `shoppinglist.csv` into the project's folder. Use Windows explorer to do this (or your favorite file manager). This step is important to keep all the files related to a project in a single place on disk.
- Add `wlp1.cpp` file to your project:
 - Open Solution Explorer (click on `View` » `Solution Explorer`)
 - Right-click on `Source Files`
 - Select `Add` » `Existing Item...`
 - Select `wlp1.cpp` from the file browser
 - Click on `Ok`
- Run the program by selecting `Debug` » `Start Debugging` or pressing the `F5` key on your keyboard.

On Linux, in your `matrix` account

- Connect to Seneca with [Global Protect VPN](#)
- Create a folder that will contain all your projects for this term's work. Name it `cpp_projects`. Inside this folder create another folder named `wlp1`.
- Upload `wlp1.cpp` and `shoppinglist.csv` files into the `~/cpp_projects/wlp1`.
- Using an `ssh` client (e.g., putty), go into that folder (`cd ~/cpp_projects/wlp1`) and compile the source file (see above for an explanation of the compilation command flags):

```
g++ wlp1.cpp -Wall -std=c++11 -o ws
```

- Run and test the execution:

```
ws
```

Step 2: Create the Modules

Using Visual Studio, in the Solution Explorer, add five new modules to your project:

- `shoppingListApp` - a module to hold the `main()` function and its relative functions and constant value (see below). This module should have only an implementation file (`*.cpp`).
- `File` - a module to hold the functions and global variables related to file processing. This module should have a header (`*.h`) and an implementation file (`*.cpp`).
- `ShoppingList` - a module to hold the direct shopping list related functions, global variables and constants. This module should have a header (`*.h`) and an implementation file (`*.cpp`).
- `ShoppingRec` - a module to hold the shopping record related functions, variables, constants and the `ShoppingRec` structure. This module should have a header (`*.h`) and an implementation file (`*.cpp`).
- `Utils` - a module to hold the general utility functions for the applications. This module may be moved to other workshops and assignments if needed. This module should have a header (`*.h`) and an implementation file (`*.cpp`).

Header files

Add `File.h`, `ShoppingList.h`, `ShoppingRec.h` and `Utils.h` to the project (in *Solution Explorer*, right-click on `Header Files` » `Add` » `New Item` and add a header file).

Make sure you add the compilation safeguards and also have all the C++ code in the last four modules in a namespace called `sdds`.

Compilation Safeguards

Compilation safeguards refer to a technique to guard against multiple inclusion of header files in a module. 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, see below 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.

Compilation Safeguards: <https://www.youtube.com/watch?v=EGak2R7QdHo>

Implementation Files

Add `shoppingListApp.cpp`, `File.cpp`, `ShoppingList.cpp`, `ShoppingRec.cpp` and `Utils.cpp` to the project (in *Solution Explorer*, right-click on `Source Files` » `Add` » `New Item` and add a C++ file).

Step 3: The Main Module

Because it will contain the `main()` function, we will refer to `shoppingListApp` module as the **main** module.

At the top of the file `shoppingListApp.cpp`, add these include and namespace statements:

```
#include <iostream>
#include <cstdlib>
#include "File.h"
#include "ShoppingList.h"
#include "Utils.h"

using namespace std;
using namespace sdds;
```

then add the definition of a constant:

```
// set to false when compiling on Linux
const bool APP_OS_WINDOWS = true;
```

From the code that has been provided to you, add the definition (implementation) of the following functions (with copy/paste):

- `main()`
- `listMenu()`

This module doesn't have a header file.

Step 4: The Other Modules

With copy/paste from the code provided, copy the functions into various modules, splitting them as describe below. Put the **declaration** of the function in headers and the **definition** in implementation files. Don't forget to add the compilation safeguard to **every** header and to make sure that each compilation safeguard is unique in the entire project.

The module `ShoppingRec` should contain:

- `ShoppingRec` custom type (`struct`)
- `getShoppingRec()`
- `displayShoppingRec()`
- `toggleBoughtFlag()`
- `isShoppingRecEmpty()`
- the constants `MAX_QUANTITY_VALUE` and `MAX_TITLE_LENGTH`

The module `ShoppingList` should contain:

- `loadList()`
- `displayList()`
- `removeBoughtItems()`
- `removeItem()`
- `saveList()`
- `clearList()`
- `toggleBought()`
- `addItemToList()`
- `removeItemfromList()`
- `listIsEmpty()`
- the constant `MAX_NO_OF_RECS`
- the variables `recs[]` and `noOfRecs`

The module `Utils` should contain:

- `flushkeys()`
- `ValidYesResponse()`
- `yes()`
- `readCstr()`
- `readInt()`

The module `File` should contain:

- `openFileForRead()`
- `openFileForOverwrite()`
- `closeFile()`
- `freadShoppingRec()`
- `fwriteShoppintRec()`
- the constant `SHOPPING_DATA_FILE` and the variable `sfptr`

Guideline for Creating Modules

- **Inclusions**

Avoid unnecessary random includes and only include a header file in another file in which the header file functions are called or the header file definitions are used. A file should include **everything** it needs, and nothing more.

- **Custom Types** (`struct` definitions)

Structure definitions must be kept in the header files to be visible to all the modules using it.

- **Global Variables**

Global variable must be in implementation files to be kept invisible to other modules. If you add global variables in header files, linking errors will occur.

- **Global Constants**

The constants are to be added to the file they are used in; if they are used in a header file, they must be added to the header file otherwise they must be added to the implementation file they are used in.

Prefer `const` instead of `#define` for the rest of the term.

- **Namespaces**

All your code (in headers and implementation files) must be surrounded by the `sdds` namespace. The only function that is not to be added in a namespace is `main()` (C++ standard demands that `main()` must be in the global namespace).

:no_entry:Important: Never put the `using namespace ...` statement in a header file. Do some research and explain in the *□ reflection* what could go wrong if `using namespace ...` is added in a header file.

:no_entry:Important: In the headers you can define only *custom types* and *constants*; the functions should only be declared in headers and defined in implementation files. Defining functions and variables in headers will (almost) always lead to linking errors (errors from the *third stage* of compilation).

Step 5: Testing

Windows

To ensure its correctness, every application must be tested. It is recommended that the project you are working on should **always** be in a compilable state; after adding/editing some code (a small function or a portion of a function), compile and test that the new code performs as intended.

To test that you have done the above steps correctly, compile each module separately. To compile a module independent of other modules, right-click on the implementation file in the *Solution Explorer* and select `Compile` from the menu. This process executes the first two stages of compilation and will reveal if there are **syntactic** errors in that specific module.

Now remove `wlp1.cpp` from the project. You can do this by right-clicking on the filename in *Solution Explorer* and selecting `Remove` in the menu; this will remove the file from the project, but not delete it from disk.

To test that the logic (functionality) is correct, you must compile the **entire** project and run it (press `F5`).

Matrix (Linux)

Upload all the module files and the data file `shoppinglist.csv` to your `matrix` account. Modify `shoppingListApp.cpp` and set the `APP_OS_WINDOWS` constant value to `false`:

```
// set to false when compiling on Linux
const bool APP_OS_WINDOWS = false;
```

Compile the source code using the following command:

```
g++ shoppingListApp.cpp File.cpp ShoppingList.cpp ShoppingRec.cpp Utils.cpp -Wall -std=c++11
-o ws
```

Run the program using the command

```
ws
```

and make sure that everything still works properly.

LAB Submission (part 1)

To test and demonstrate the execution of your program upload all source-code and input files to your `matrix` account, compile your program (see the command compilation command above) and use the same data as shown in the [LAB Execution example](#) . Make sure that everything runs properly and there are no compilation errors or warnings.

Then, run the following command from your account

- replace `profname.proflastname` with your professor's Seneca userid
- replace `??` with your subject code (200 or 244)
- replace `#` with the workshop number
- replace `X` with the workshop part number (1 or 2)

```
~profname.proflastname/submit 2??/w#/pX
```

and follow the instructions.

:warning: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. Re-submissions will attract a penalty.

DIY (50%)

In this part you are to create *"Dictionary"*: a program that loads a collection of words and their definitions from a file and allows a client to interact with the dictionary using certain functions. A word can have multiple definitions.

The Input File

The words in the dictionary will be loaded from a file. The file has a very well defined structure and you can assume that it will always be correct (no error checking is required for reading from the file).

Each word looks like this:

```
fine
adjective: of high quality.
adjective: (of a thread, filament, or person's hair) thin.
noun: very small particles found in mining, milling, etc.
adverb: in a satisfactory or pleasing manner; very well.
```

On the first line, it's the word we are defining. The next lines contain the definitions for the word, one per line. Each line with a definition has the following structure (the elements are in sequence):

- `\t` as a first character signals that it's a definition
- the type of the word (*noun*, *verb*, *adjective*, etc.); this is an array of characters that doesn't contain blanks
- `:` a separator between the type of the word and the definition
- the rest of the characters in the line are part of the definition for the word.

Two consecutive words in the dictionary will be separated by **exactly** one blank line.

The file can have maximum 100 words with their definitions. Each word can have maximum 8 definitions. A word that is defined and the type of the word can have maximum 64 characters, and the text of a definition can have maximum 1024 characters. Use these numbers when you design your code.

Dictionary Module

Create a module named `dictionary` that contains any structures/functions/global variables that are useful when clients interact with a dictionary. Your code must have at least the following (but you can add more, as your design requires):



```

/// <summary>
/// Load from a file a set of words with their definition. Any previous
/// existing dictionary is discarded, regardless of the result of the load.
/// </summary>
/// <param name="filename">The name of the file containing the dictionary.</param>
/// <returns>0 if data has been loaded from the file, non-zero otherwise
/// (null parameter, empty parameter, missing file).</returns>
int LoadDictionary(const char* filename);

/// <summary>
/// Save all the words existing in the dictionary into a file. The functions
/// "LoadDictionary" should be able to load from this after the save finished.
/// </summary>
/// <param name="filename">The name of the file where to save.</param>
void SaveDictionary(const char* filename);

/// <summary>
/// Search in the dictionary for the word and display all the definitions found.
///
/// Print "NOT FOUND: word [X] is not in the dictionary." if
/// the word doesn't exist in dictionary.
/// </summary>
/// <param name="word">The word to search and display.</param>
void DisplayWord(const char* word);

/// <summary>
/// Adds a word to the dictionary.
///
/// If the dictionary already contains the word, add a new definition for it.
/// </summary>
/// <param name="word">The word to add to the dictionary.</param>
/// <param name="type">The type of the word (noun, verb, adjective, etc.)</param>
/// <param name="definition">The definition for the word.</param>
void AddWord(const char* word, const char* type, const char* definition);

/// <summary>
/// Searches in the dictionary for a word and update its definition and type.
/// If the word contains multiple definitions, print the message
/// "The word [X] has multiple definitions:" followed by a list of existing
/// definitions, and asks user which one to update. This function assumes that
/// the user's input is correct and doesn't require validation.
///
/// If the word is not in the dictionary, this function does nothing.
/// </summary>
/// <param name="word">The word to update.</param>
/// <param name="type">The new type of the definition.</param>
/// <param name="definition">The new definition.</param>
/// <returns></returns>
int UpdateDefinition(const char* word, const char* type, const char* definition);

```

Implement every function listed above in the namespace `sdds`.



Word module

Create a module named `word` that contains functionality related to a single word (e.g., to display it to the

screen, to save it into a file, to reset it to a default state, etc.).

Hints

The hints below are just guides; you can create a design that accomplishes the task using different features.

- Create a custom type (`struct`) to hold information about a word (the string representing the word, the number of definitions the word has, the definitions, etc.). 
- Create a custom type (`struct`) to hold information about a dictionary (how many word are there, a collection with all the words, etc.).
- Use arrays when you need to store a collection of objects of the same type.
- See which tasks must be performed multiple times; put that code in a function and call the function when needed. 
- Keep the functions simple; a function should do only one thing. If a function becomes too long, break it into multiple simpler functions (a good rule of thumb is *"a function should be at most a screen big"*, so a reader doesn't have to scroll to see the entire function).
- Use the provided sample output for hints about how to present the information to the user.
- Make sure you read and understand the provided code **before** implementing your solution.
- Check the C course for a refresh in file and string manipulation. Consider the following functions:
 - `std::fopen` - <https://en.cppreference.com/w/cpp/io/c/fclose>
 - `std::fclose` - <https://en.cppreference.com/w/cpp/io/c/fopen>
 - `std::fgets` - <https://en.cppreference.com/w/cpp/io/c/fgets>
 - `std::fgetc` - <https://en.cppreference.com/w/cpp/io/c/fgetc>
 - `std::fscanf` - <https://en.cppreference.com/w/cpp/io/c/fscanf>
 - `std::scanf` - <https://en.cppreference.com/w/cpp/io/c/scanf>
 - `std::printf` - <https://en.cppreference.com/w/cpp/io/c/printf>
 - `std::strcpy` - <https://en.cppreference.com/w/cpp/string/byte/strcpy>
 - `std::strncpy` - <https://en.cppreference.com/w/cpp/string/byte/strncpy>
 - `std::strcmp` - <https://en.cppreference.com/w/cpp/string/byte/strcmp>
 - `std::strncmp` - <https://en.cppreference.com/w/cpp/string/byte/strcmp>

Tester Program

The `main` module and the input files are provided. Read the code and make sure you understand it before attempting to implement anything. **Do not change the existing code.** If the program is implemented correctly, the output should look like the one from the `sample_output.txt` file.

□ Reflection

Study your final solutions for each deliverable of the workshop, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. **This should take no less than 30 minutes of your time and the result is suggested to be at least 150 words in length.**

Create a **text** file named `reflect.txt` that contains your **detailed** description of the topics that you have learned in completing this particular workshop and mention any issues that caused you difficulty and how you solved them. To avoid deductions, refer to code in your solution as examples to support your explanations.

You may be asked to talk about your reflection (as a presentation) in class.

DIY Submission (part 2)

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

Upload your source code to your `matrix` account. Compile and run your code using the `g++` compiler as shown above and make sure that everything works properly.

Then, run the following command from your account

- replace `profname.proflastname` with your professor's Seneca userid
- replace `??` with your subject code (200 or 244)
- replace `#` with the workshop number
- replace `X` with the workshop part number (1 or 2)

```
~profname.proflastname/submit 2??/w#/pX
```

and follow the instructions.

:warning: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. Re-submissions will attract a penalty.

