Workshop 1

Across Translation Units

In this workshop, you implement aspects of linkage, storage duration, namespaces, guards, and operating system interfaces.

LEARNING OUTCOMES

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

- to declare a local variable that remains in memory for the lifetime of the program
- to guard a class definition from repetition
- to access a variable defined in a different translation unit
- to receive program arguments from the command line
- to upgrade code to accept and manage a user-defined string of any length

SUBMISSION POLICY

The *in-lab* section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. If you do not attend the workshop, you can submit the *in-lab* section along with your *at-home* section (see penalties below). The *at-home* portion of the lab is due on the day that is four days after your scheduled in-lab workshop (23:59:59) (even if that day is a holiday).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.

You are responsible to back up your work regularly.

Late Submission Penalties:

- *In-lab* portion submitted late, with *at-home* portion: **0** for *in-lab*. Maximum of 7/10 for the entire workshop.
- If any of *in-lab*, *at-home* or *reflection* portions is missing, the mark for the workshop will be **0**/10.

SPECIFICATIONS – IN LAB

This workshop consists of three modules:

- **w1**
- **process** (supplied)
- **String**

Enclose all your source code within the **sict namespace** and include the necessary guards in each header file. For the in-lab part, your **String** class should store no more than 3 characters excluding the null terminator. The output from your executable running Visual Studio with the following command line arguments should look like

```
Command Line : C:\Users\...\Debug\w1 in_lab.exe oop345 btp305
3: oop
4: btp
```

w1 Module

Your **w1 module** consists of a **main()** function and an unmodifiable variable named **INITIAL** of global scope that defines the integer value for the next item in a list, in this case a value of 3.

The **main()** function

- receives a set of standard command line arguments
- echoes the set of arguments to standard output separated by a single space on one line
- inserts the error message **"***Insufficient number of arguments***"** if the user has not entered any string on the command line and passes control to the operating system with a return value of 1.
- calls the **process** function for each string and passes the address of that string as the only argument.
- passes control to the operating system with a return value of 0.

process Module

The **process** module is supplied and you do not need to change its header or implementation files. The implementation file defines a single function named **process()** that receives the address of an unmodifiable C-style null terminated string, constructs a **String** object from the C-style string and inserts that object into standard output followed by a newline.

String Module

Your **String** module defines an unmodifiable integer named MAX of value 3 along with a class named **String** that holds a C-style null-terminated string of up to MAX characters excluding the null byte terminator. The class includes the following member and helper functions:

- a one-argument constructor that receives the address of an unmodifiable C-style null-terminated string and copies the string at that address into an instance variable. Check for receipt of the null address and store an empty string in that case.

- a query named **display()** that receives a reference to an **std::ostream** object and inserts the string stored in the instance variable.

- a helper non-friend operator that inserts the saved string into the left operand. This operator inserts an item number followed by a colon and a single space as shown in the output example above. Your function uses a local variable to store the current item number and starts with the value of an external global variable named **INITIAL** defined in another translation unit. Your function inserts the string associated with the right operand and increments the current item number after the insertion.

Do not use the **string** class of the standard library in this workshop. Use the **cstring** functions.

In-Lab Submission (30%)

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

Upload your source code to your matrix account. Compile and run your code using the latest version of the gcc compiler and make sure that everything works properly.

Then, run the following command from your account: (replace profname.proflastname with your professor's Seneca userid)

~profname.proflastname/submit 345XXX_w1_lab<ENTER>

and follow the instructions. Replace **XXX** with the section letter(s) specified by your instructor.

SPECIFICATIONS – AT HOME

For this part of the workshop, upgrade your **String** class to manage a C-style null-terminated string of any length. Do not change your other modules.

Reflection

Study your final solution, 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. Explain in your own words what you have learned in completing this workshop. Include in your explanation but do not limit it to the following points (40%):

- the difference between internal and external linkage using examples from your code
- the execution effect of the code in the **process()** function that creates the **String** object which is inserted into standard output (which member function(s) does this code call)
- the changes that you made in upgrading your **String** class.

Include all corrections to the Quiz you have received (30%).

At-Home Submission (70%)

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

Upload your source code to your matrix account. Compile and run your code using the latest version of the gcc compiler and make sure that everything works properly.

Then, run the following command from your account: (replace profname.proflastname with your professor's Seneca userid)

~profname.proflastname/submit 345XXX_w1_home<ENTER>

and follow the instructions. Replace **XXX** with the section letter(s) specified by your instructor.