Part B – Foundations

Classes and Privacy

Workshop 3
v1.0 (release)

In this workshop, you will use *classes, access levels* and *member functions* to create encapsulated objects. We will be doing so via the use of a **class** that represents a Train carrying Cargo.

Learning Outcomes

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

- to define a class type
- to privatize data within the class type
- to instantiate an object of class type
- to access data within an object of class type through public member functions
- to use standard library facilities to format data inserted into the output stream
- to describe to your instructor what you have learned in completing this workshop

Submission Policy

The workshop has into two components:

- **lab**: to be completed before the end of the lab period and submitted from the lab;
- **DIY**: to be completed within **5 days** after the day of your lab;

The *lab* section is to be completed after the workshop is published, and before the end of the lab session. The *lab* must be submitted during the workshop period from the lab.

If you attend the lab period and cannot complete the *lab* portion of the workshop during that period, ask your instructor for permission to complete the *lab* portion after the period.

You must be present at the lab for the duration of the entire session in order to get credit for the *lab* portion.

If you do not attend the workshop, you can submit the *lab* section along with your *DIY* section (see Submission Penalties below). The *DIY* portion of the lab is due on the day that is **5 days** after your scheduled *lab* workshop (by 23:59 or 11:59PM) (even if that day is a holiday).

The *DIY* section of the workshop is a task that utilizes the concepts you have done in the **lab** section. This section is open ended with no detailed instructions other than the required outcome.

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.

Ask your professor if there are any additional requirements for your specific section.

Reflection, Citation and Sources

After the workshop fully is completed (meaning the **lab** was already submitted and the **diy** is ready to be submitted), 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. Add any other comments you wish to make.

There is a section below prior to the **diy submission procedure** that provides some questions and expectations you should aim to answer at a minimum for the reflection. Aiming higher than that will be to your benefit as a poorly done reflection can incur a **mark reduction** (see Submission Penalties below).

This reflection is a mandatory part of the workshop submission.

When submitting any deliverables, create a file called **sources.txt** in the project folder. This file will be submitted with your work automatically.

You are to write either of the following statements in the file "**sources.txt**":

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

NAME (STUDENT#), SENECA EMAIL

OR:

Write exactly which part of the code of the workshops or the assignment are given to you as help and who gave it to you or which source you received it from.

Finally add your name, student number and Seneca email as signature.

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.

SUBMISSION PENALTIES:

If you do not follow the submission the policies, the following penalties will be applied:

- If the **lab** is submitted past the due date but before the rejection date, it will be receive **0%.**
- If the submitted **reflection** is deemed to be insufficient in depth or does not demonstrate a strong understanding of the core concepts used in the Workshop, a **maximum reduction of 40%** may be applied the overall mark of the Workshop.
- If any of *lab*, *DIY* or *reflection* is missing on rejection date, the total mark of the workshop will be **0%**.

To see the dues for a deliverable, use:

```
~cornel.barna/submit 200/NXX/WSXX/lab -due<ENTER>
~cornel.barna/submit 200/NXX/WSXX/diy -due<ENTER>
```

COMPILING AND TESTING YOUR PROGRAM

Compile all your code using this command on matrix:

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

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

```
valgrind ws<ENTER>
```

LAB - 50%

Create an empty module called **Train** to represent Trains that carry Cargo. As you code this module, remember to make use of compilation safeguards as well as enclose each variable, function, structure or identity inside the **sdds** namespace. Review Workshop 1/2 for details.

Firstly, define the following constants in the **Train** header:

`MAX_NAME` with a value of 30.
This constant number represents the maximum length for a **Train's** name excluding the *nullbyte* (\0).

`MAX_DESC` with a value of 20.
This constant number represents the maximum length for a **Cargo's** description excluding the *nullbyte* (\0).

STRUCTURES / CLASSES

Create a **struct** called **Cargo** that has the following data members in it:

- `description` – A statically allocated **character array** that will hold the description of a Cargo. Its length is determined by MAX_DESC.
- `weight` – A **double** that represents the weight of a Cargo.

Create a **class** called **Train** that has the following **private** data members in it:

- `name` – A statically allocated **character array** that will hold the name of a Train. Its length is determined by MAX_NAME.
- `id` – An **integer** that represents the identifying number of a Train.
- `cargo` – A **pointer** of **Cargo** type. It will be used to represent the Cargo a Train is carrying by utilizing dynamic memory.

TRAIN PUBLIC MEMBERS

In order to interact with the **private** data members of the **Train** class, public member functions are needed. The following function prototypes should be placed in the Train **header** and their definitions in the respective Train **implementation** files (.cpp).

```
void setTrain(const char*, int)
```

This function takes in two parameters of which the first is a **constant character pointer** that will be used to set the **name** data member of the Train and the second is an integer used to set the **id** of the Train.

Prior to setting the values straightaway, we need to validate the parameters. If either of the provided parameters are invalid then set the Train's data members to **default values**. We can define invalid as being:

1. If the character pointer is *nullptr*.
2. If the string pointed to by the character pointer is the *empty string*.
3. If the provided integer for the id is *less than 1*.

These default values should indicate that the Train has not been set with valid values. A value for integer data could be **0** or a **negative number**. A possible value for character data could be the **nullbyte**.

If a **valid** set of values are provided via the parameters then simply set the Train's *name* and *id* respectively to those values.

At end set the value of the **cargo** data member of the Train to **nullptr**. *This setting of nullptr is used to indicate that there is no cargo being carried by the Train.*

```
bool isEmpty() const
```

This function will return a **true** or **false** value depending on if the Train is in an **empty state**. Consider using the empty state values from the above setTrain function here as the test for if the Train is empty.

`void display() const`
This function will display the details of a Train in the following manner:

- First it will print out:
  `"***Train Summary***" <newline>`

- If the Train is **empty** it will only print out the following line:
  `"This is an empty train." <newline>`

  If the Train **isn't empty** then it will print out the name and id of the train:
  `"Name: [name] ID: [id]" <newline>`

- Following the name and id (if non-empty train), if there is **cargo** on the train then print out the cargo's description and weight:
  `"Cargo: [description] Weight: [weight]" <newline>`

  Otherwise simply print out:
  `"No cargo on this train." <newline>`

`void loadCargo(Cargo)`
This function will take in a parameter of **Cargo** type and use it to set the values of the Train's **cargo** data member. First allocate dynamic memory for a single instance of Cargo to the **cargo** pointer data member then copy over the values from the parameter to it.

`void unloadCargo()`
This function will deallocate any memory that may have been allocated to the **cargo** pointer data member and then set the value of the pointer to **nullptr** (the empty state).

LAB MAIN MODULE

```
/*****************************************************************
// Workshop 3: Classes & Privacy
// File TrainTester.cpp
// Version 1.0
// Date       2020/01/19
```

```
// Author Michael Huang
// Description
// Tests Train module
//
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
//
/////////////////////////////////////////////////////////////////
*********************************************************************/

#include <iostream>
#include "Train.h"
#include "Train.h" // intentional
using namespace std;
using namespace sdds;

ostream& line(int len, char ch) {
	for (int i = 0; i < len; i++, cout << ch);
	return cout;
}
ostream& number(int num) {
	cout << num;
	for (int i = 0; i < 9; i++) {
		cout << " - " << num;
	}
	return cout;
}

int main() {

  sdds::Cargo c1{ "Boxes", 55.55 };
  sdds::Cargo c2{ "Flowers", 66.666 };
  sdds::Cargo c3{ "Ore", 77.7777 };

  cout << "Create a Train and attempt to set it (empty state)" << endl;
  line(64, '-') << endl; number(1) << endl;
  sdds::Train t1;
  t1.setTrain(nullptr, 1);
  sdds::Train t2;
  t2.setTrain("", 1);
  sdds::Train t3;
  t3.setTrain("Birthday Train", -5);
  if (t1.isEmpty() && t2.isEmpty() && t3.isEmpty())
    cout << "Success! Each Train is in empty state" << endl;

  cout << "\nDisplay an empty Train" << endl;
  line(64, '-') << endl; number(2) << endl;
  t1.display();

  cout << "\nSet each Train to a non empty state" << endl;
  line(64, '-') << endl; number(3) << endl;
  t1.setTrain("Birthday Train", 1);
  t2.setTrain("Choo Choo Train", 2);
  t3.setTrain("Hype Train", 3);

  if (!t1.isEmpty() && !t2.isEmpty() && !t3.isEmpty())
    cout << "Success! Each Train is in non empty state" << endl;
```

```
  cout << "\nDisplay each non empty Train" << endl;
  line(64, '-') << endl; number(4) << endl;
  t1.display(); cout << endl;
  t2.display(); cout << endl;
  t3.display(); cout << endl;

  cout << "\nLoad Cargo onto a Train and Display" << endl;
  line(64, '-') << endl; number(5) << endl;
  t1.loadCargo(c1);
  t2.loadCargo(c2);
  t3.loadCargo(c3);
  t1.display(); cout << endl;
  t2.display(); cout << endl;
  t3.display(); cout << endl;

  cout << "\nUnload Cargo from each Train and Display" << endl;
  line(64, '-') << endl; number(6) << endl;
  t1.unloadCargo();
  t2.unloadCargo();
  t3.unloadCargo();
  t1.display(); cout << endl;
  t2.display(); cout << endl;
  t3.display(); cout << endl;

  return 0;
}
```

EXECUTION EXAMPLE

```
Create a Train and attempt to set it (empty state)
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
Success! Each Train is in empty state

Display an empty Train
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
***Train Summary***
This is an empty train.

Set each Train to a non empty state
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
Success! Each Train is in non empty state

Display each non empty Train
----------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
***Train Summary***
Name: Birthday Train ID: 1
No cargo on this train.

***Train Summary***
Name: Choo Choo Train ID: 2
```

```
No cargo on this train.

***Train Summary***
Name: Hype Train ID: 3
No cargo on this train.


Load Cargo onto a Train and Display
---------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
***Train Summary***
Name: Birthday Train ID: 1
Cargo: Boxes Weight: 55.55

***Train Summary***
Name: Choo Choo Train ID: 2
Cargo: Flowers Weight: 66.666

***Train Summary***
Name: Hype Train ID: 3
Cargo: Ore Weight: 77.7777


Unload Cargo from each Train and Display
---------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
***Train Summary***
Name: Birthday Train ID: 1
No cargo on this train.

***Train Summary***
Name: Choo Choo Train ID: 2
No cargo on this train.

***Train Summary***
Name: Hype Train ID: 3
No cargo on this train.
```

IN-LAB SUBMISSION

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

If not on matrix already, upload `Train module` and the `TrainTester.cpp` program to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~cornel.barna/submit 200/NXX/WS03/lab<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.

DIY (DO IT YOURSELF) – 50%

In the DIY, we will be updating our **Train** module to provide a stronger sense of encapsulation while also creating additional functions. As you work on the DIY, it is recommended to look over the provided main program to gain hints on how to implement the requirements.

Firstly, define the following two new constants in the **Train** header:

`MAX_WEIGHT` with a value of 700.555.
This constant **double** number represents the maximum weight for a **Cargo**.

`MIN_WEIGHT` with a value of 40.444.
This constant **double** number represents the minimum weight for a **Cargo**.

UDPATE CARGO STRUCT TO CLASS

Change the **Cargo** struct to a **class**. Follow the normal notions of how a class should be accessed (private data members & public member functions). This change may cause previous functionality from the **lab** portion to no longer work as is. Address this by creating public member functions that can get and set the data members of Cargo (description and weight). This may then require you to modify functions in Train that previously interacted with Cargo to use your new Cargo functions.

You may also create any private member functions needed to accomplish your design.

In addition, one public member function for Cargo is required:

```
void init(const char*, double)
```
This function will take in a **constant character pointer** that will point to a string literal (a description for the Cargo) and a **double** (the weight of the Cargo). It will then use those parameters to set the values of a Cargo object. Minimal validation is needed only to make sure that when copying the value of the **description** to the data member, it doesn't exceed MAX_DESC in length and that the **weight** is within the range of MIN_WEIGHT and MAX_WEIGHT. If a weight given through the double parameter is outside of that range simply set the **weight** of the Cargo to MIN_WEIGHT instead.

UPDATE / NEW TRAIN FUNCTIONS

```
void display() const
```
Update the display function so that when displaying the weight of the Cargo (if any), the value of the weight is limited to **2 decimal places of precision**.

```
bool swapCargo(Train&)
```
This function will take in a **reference** to another Train object and will swap the Cargo between the two. For example, if a Train A was carrying Cargo A and Train B was carrying Cargo B, this function would result in Train A with Cargo B and vice versa. If a swap was successful return **true** and **false** otherwise.

A swapping of Cargo can not occur unless both the **Train** object calling swapCargo and the **Train** provided in the parameter actually carry Cargo. If this is the case then the function doesn't do anything except return a Boolean value.

```
bool increaseCargo(double)
```
This function will take in a **double** number and use it to increase the Train's Cargo's **weight**. If the weight of the Cargo was successfully increased (meaning the weight actually changed values) then return **true** and **false** otherwise. When increasing the weight, this function should still respect the value of **MAX_WEIGHT** and not exceed it.

If the Train isn't carrying any Cargo then this function doesn't do anything except return a Boolean value.

```
bool decreaseCargo(double)
```

Similar to the increaseCargo function except it decrease the Train's Cargo weight instead. It will return **true** if the Cargo's **weight** was successfully decreased and **false** otherwise. Again, this function should respect the value of MIN_WEIGHT and not exceed it when attempting to decrease Cargo weight.

If the Train isn't carrying any Cargo then this function doesn't do anything except return a Boolean value.

DIY Main Module

```
/*****************************************************************
// Workshop 3: Classes & Privacy
// File TrainTester2.cpp
// Version 1.0
// Date       2020/01/19
// Author Michael Huang
// Description
// Tests updated Train module
//
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
//
/////////////////////////////////////////////////////////////////
*****************************************************************/

#include <iostream>
#include "Train.h"
#include "Train.h" // intentional
using namespace std;
using namespace sdds;

ostream& line(int len, char ch) {
	for (int i = 0; i < len; i++, cout << ch);
	return cout;
}
ostream& number(int num) {
	cout << num;
	for (int i = 0; i < 9; i++) {
		cout << " - " << num;
	}
	return cout;
}

int main() {

  Cargo c1, c2;
  c1.init("Boxes", -5000);
  c2.init("Flowers", 5000);
  Train t1, t2;
```

```
  cout << "\nSet each Train to a non empty state, load cargo and display" << endl;
  line(64, '-') << endl; number(1) << endl;
  t1.setTrain("Birthday Train", 1);
  t2.setTrain("Choo Choo Train", 2);
  t1.loadCargo(c1);
  t1.display();
  cout << endl;
  t2.display();

  cout << "\nIncrease cargo weight in a train and display" << endl;
  line(64, '-') << endl; number(2) << endl;
  if (t1.increaseCargo(50)) cout << "t1 cargo was correctly increased" << endl;
  if (t1.increaseCargo(999)) cout << "t1 cargo was correctly increased to MAX_WEIGHT" <<
endl;
  if (!t1.increaseCargo(999)) cout << "t1 cargo was correctly not increased" << endl;
  cout << endl;
  t1.display();

  cout << "\nDecrease cargo weight in a train with no cargo and display" << endl;
  line(64, '-') << endl; number(3) << endl;
  if (!t2.decreaseCargo(25)) cout << "t2 doesn't have cargo was correctly not touched" <<
endl;
  t2.loadCargo(c2);
  if (!t2.decreaseCargo(25)) cout << "t2 after loading cargo was correctly not decreased
below MIN_WEIGHT" << endl;
  t2.increaseCargo(50);
  if (t2.decreaseCargo(10)) cout << "t2 decreased weight correctly" << endl;
  cout << endl;
  t2.display();

  cout << "\nSwap cargo between train t1 and t2" << endl;
  line(64, '-') << endl; number(4) << endl;
  t1.swapCargo(t2);
  t1.display();
  cout << endl;
  t2.display();

  cout << "\nUnload Cargo from each Train and attempt to swap" << endl;
  line(64, '-') << endl; number(5) << endl;
  t1.unloadCargo();
  t2.unloadCargo();
  if (!t1.swapCargo(t2)) cout << "correctly did not attempt to swap train without cargo"
<< endl;
  cout << endl;
  t1.display();
  cout << endl;
  t2.display();

  return 0;
}
```

EXECUTION EXAMPLE

```
Set each Train to a non empty state, load cargo and display
---------------------------------------------------------------
```

```
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
***Train Summary***
Name: Birthday Train ID: 1
Cargo: Boxes Weight: 40.44

***Train Summary***
Name: Choo Choo Train ID: 2
No cargo on this train.

Increase cargo weight in a train and display
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
t1 cargo was correctly increased
t1 cargo was correctly increased to MAX_WEIGHT
t1 cargo was correctly not increased

***Train Summary***
Name: Birthday Train ID: 1
Cargo: Boxes Weight: 700.55

Decrease cargo weight in a train with no cargo and display
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
t2 doesn't have cargo was correctly not touched
t2 after loading cargo was correctly not decreased below MIN_WEIGHT
t2 decreased weight correctly

***Train Summary***
Name: Choo Choo Train ID: 2
Cargo: Flowers Weight: 80.44

Swap cargo between train t1 and t2
----------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
***Train Summary***
Name: Birthday Train ID: 1
Cargo: Flowers Weight: 80.44

***Train Summary***
Name: Choo Choo Train ID: 2
Cargo: Boxes Weight: 700.55

Unload Cargo from each Train and attempt to swap
----------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
correctly did not attempt to swap train without cargo

***Train Summary***
Name: Birthday Train ID: 1
No cargo on this train.

***Train Summary***
Name: Choo Choo Train ID: 2
No cargo on this train.
```

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 file named `reflect.txt` that contains your **detailed description of the topics that you have learned** in completing this workshop and mention any issues that caused you difficulty. Some possible points of discussion for the reflection — **but do not limit it to just these** —:

1. What is the purpose of an empty state? What are possible empty state values?
2. What is the difference between structs and classes in C++?
3. What is the notion of class privacy and why is it important?
4. Describe what you have learned in this workshop

DIY SUBMISSION

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

If not on matrix already, upload `TrainTester2.cpp` program and the updated `Train` module to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

~cornel.barna/submit 200/NXX/WS03/diyENTER>

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.