**Workshop 6: STL Containers**

In this workshop, you store polymorphic objects in an STL container.

You are going to create an application that simulates an autoshop that sells various types of vehicles. This specific application, will focus on **cars** and **racecars**.

**Learning Outcomes**

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

* manage polymorphic objects using the vector container of the STL
* move a dynamically allocated object into a container
* code a range-based iteration on the objects in a container
* report and handle an exception

**Submission Policy**

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

* *Part 1*: worth 0% of the workshop's total mark, is optional and designed to assist you in completing the second part.
* *Part 2*: worth 100% 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 does not 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).

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 wrong doing.

**Compiling and Testing Your Program**

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

/usr/local/gcc/9.1.0/bin/g++ -Wall -std=c++17 -g -o ws file1.cpp file2.cpp ...

* -Wall: compiler will report all warnings
* -std=c++17: the code will be compiled using the C++17 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.

If, when you try to compile/submit the workshop on matrix, you encounter an error regarding /lib64/libstdc++.so.6: version 'CXXABI\_1.3.9 not found', add the following line at the end of your .bashrc file (this is a hidden text file located in the home folder--~--of your matrix account):

export LD\_LIBRARY\_PATH=/usr/local/gcc/9.1.0/lib64:$LD\_LIBRARY\_PATH

After you edit the file, logout and login again--this problem should go away.

**Part 1 (0%)**

The first portion of this workshop consists of modules:

* w6 (supplied)
* Vehicle (supplied)
* Car
* Autoshop
* Utilities

Enclose all your source code within the sdds namespace and include the necessary guards in each header file.

**Car Module**

This module defines a class that holds information about a single car.

***Private Data***

Design and code a class named Car, that inherits the interface Vehicle (supplied), and that should be able to store the following information (for each attribute, you can chose any type you think it's appropriate--you must be able to justify the decisions you have made):

* **maker**
* **condition**: a car can be **new**, **used**, or **broken** in need of repairs.
* **top speed**

***Public Members***

* a custom constructor that receives an object of type std::istream& as parameter. This constructor should be able to read a single record from the stream, extract all the information about a single car and store it in the attributes. Each record has the following format:
* TAG,MAKER,CONDITION,TOP\_SPEED
  + TAG, is a single character representing the type of vehicle: c or C for car. Any other tag is considered invalid.
  + MAKER, the name of the company that makes the car
  + CONDITION, a single character: n for **new**, u for **used**, and b for **broken**. Any other character is considered invalid.
  + TOP\_SPEED, an integer containing the maximum speed the vehicle can achieve. If the content of this field is not a number, then the field is considered invalid.
  + any space at the beginning/end of each field should be removed.
* std::string condition() const: a query that returns new, used or broken, representing the condition of the car
* double topSpeed() const
* void display(std::ostream& out) const, a query that inserts in the first parameter the content of the car object in the format | MAKER | CONDITION | TOP\_SPEED |, where
  + MAKER, field of size 10
  + CONDITION, field of size 6
  + TOP\_SPEED, field of size 6, with two significant digits after the period
  + see alignment in the sample output.

**Add any other function that is required by your design!**

For the first part, assume there are no invalid records in the file.

**Autoshop module**

This module defines a class that holds information about an autoshop that sells various types of vehicles.

***Private Data***

Design and code a class named Autoshop that should be able to store the following information:

* std::vector<Vehicle\*> m\_vehicles: a vector that store all the vehicles available at this autoshop.

***Public Members***

* Autoshop& operator +=(Vehicle\* theVehicle): adds the vehicle received as parameter into the m\_vehicles vector.
* void display(std::ostream& out) const: iterates over the vehicles stored in m\_vehicles and prints them into the parameter (use Vehicle::display()) using the format:
* --------------------------------
* | Cars in the autoshop! |
* --------------------------------
* VEHICLE
* VEHICLE
* ...
* --------------------------------

**To iterate over the elements of the vector, use STL iterators!**

**Utilities module**

This module should contain a single function that creates instances on the Vehicle hierarchy:

Vehicle\* createInstance(std::istream& in);

This function should extract data from the parameter; if the first non-blank character is c or C, the this function should **dynamically** create an instance of type Car passing the stream to the constructor, and return it to the client.

If the first non-blank character is anything else (or there are no more characters to extract), this function should return nullptr.

Because the input file contains two types of delimiters (\n for records, and , for the fields in a record), you can use the class std::stringstream (utilization of this class is not mandatory, the extraction can be achieved without using it).

When implementing the createInstance function, consider the following STL class:

* [std::stringstream](https://en.cppreference.com/w/cpp/io/basic_stringstream/basic_stringstream)

**w6 Module (supplied)**

The tester module for this portion has been supplied. **Do not modify the existing code!**

When doing the workshop, you are encouraged to write your own tests, focusing on a single implemented feature at the time, until you get all functionality in place.

**Sample Output**

When the program is started with the command (the file cars.txt is provided):

w6.exe cars.txt

the output should look like the one from the sample\_output.txt file.

Command Line:

--------------------------

1: w6.exe

2: cars.txt

--------------------------

--------------------------------

| Cars in the autoshop! |

--------------------------------

| Toyota | new | 157.00 |

| Honda | broken | 145.00 |

| Chrysler | new | 173.00 |

--------------------------------

**Test Your Code**

To test the 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 g++ compiler (available at /usr/local/gcc/9.1.0/bin/g++) 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 345\_w6\_p1

and follow the instructions.

***This part represents a milestone in completing the workshop and is not marked!***

**Part 2 (100%)**

The second part of this workshop upgrades your solution to include one more module and update some existing modules:

* Racecar
* Car (to be updated)
* Autoshop (to be updated)
* Utilities (to be updated)
* w6 (to be updated)

**Racecar Module**

Add a Racecar module to your project. A Racecar is a Car that for a short period of time can boost its top speed by a certain percent (this class should inherit Car).

***Private Data***

* double m\_booster: the percentage (stored as a number between 0 an 1) by which this Racecar can boost its top speed.

***Public Members***

* Racecar(std::istream& in): calls the constructor from Car to build the subobject, and then it extracts the last field from the stream containing the booster bonus. The input format for a racecar is TAG,MAKER,CONDITION,TOP\_SPEED,BOOSTER
* void display(std::ostream& out) const: calls Car::display() to print the information about the car, and adds \* at the end. The format should be | MAKER | CONDITION | TOP\_SPEED |\*
* double topSpeed() const: returns the top speed of the car, including any booster effects.

This class should not have access to the attributes of the parent class.

**Car Module**

Update the Car module to handle invalid records, and generate exceptions when an invalid record is detected. This requires you to modify the custom constructor to detect the following situations:

* the token representing the condition the car is empty (no characters or only blanks):
  + consider that the car is **new**
* the token representing the condition of the car is a different character than n, u, or b:
  + generate an exception to inform the client that this record is invalid
* the token representing the top speed is not a number:
  + generate an exception to inform the client that this record is invalid

No need to change anything else.

**Utilities module**

Update the createInstance to build an instance of type Racecar if the first non-blank character extracted from the stream is r of R.

If the first non-blank character is not c, C, r, or R this function should generate an exception, informing the client that this record contains information about an unknown vehicle.

If there is no more information to be extracted from the stream, this function should return nullptr.

**Autoshop Module**

Update this module to include two additional public functions.

***Public Members***

* a destructor. This function should iterate over the objects stored in the vector, and delete each one of them (note that the first portion has a memory leak because the dynamically allocated vehicles were not deleted anywhere).
* void select(T test, std::list<const Vehicle\*>& vehicles): a **template** function that iterates over the vehicles stored in the m\_vehicles, and adds to the second parameter all vehicles for which the test is true. The first parameter (test) can be a lambda expression, a pointer to a function, or a functor that matches the prototype:

bool func(const sdds::Vehicle\*);

**Since this is a template function, it must be implemented in the header!** The class is not a template.

**w6 Module (partially supplied)**

This module has some missing parts. The missing parts are marked with TODO, describing what code you should add and where. **Do not modify the existing code, only add what is missing!**

**Sample Output**

When the program is started with the command (the files are provided):

w6.exe dataClean.txt dataMessy.txt

the output should look like the one from the sample\_output.txt file.

Command Line:

--------------------------

1: w6.exe

2: dataClean.txt

3: dataMessy.txt

--------------------------

--------------------------------

| Cars in the autoshop! |

--------------------------------

| Toyota | new | 157.00 |

| Jaguar | used | 295.20 |\*

| Honda | broken | 145.00 |

| Porche | used | 365.40 |\*

| Chrysler | new | 173.00 |

--------------------------------

Invalid record!

Invalid record!

Unrecognized record type: [t]

--------------------------------

| Cars in the autoshop! |

--------------------------------

| Toyota | new | 157.00 |

| Jaguar | used | 295.20 |\*

| Honda | broken | 145.00 |

| Porche | used | 365.40 |\*

| Chrysler | new | 173.00 |

| Alfa Romeo | new | 176.00 |

| Ford | new | 162.00 |

| Red Bull | broken | 346.56 |\*

| Ferrari | new | 367.33 |\*

--------------------------------

--------------------------------

| Fast Vehicles |

--------------------------------

| Porche | used | 365.40 |\*

| Red Bull | broken | 346.56 |\*

| Ferrari | new | 367.33 |\*

--------------------------------

--------------------------------

| Vehicles in need of repair |

--------------------------------

| Honda | broken | 145.00 |

| Red Bull | broken | 346.56 |\*

--------------------------------

**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 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. Include in your explanation—**but do not limit it to**—the following points:

* Why do you need to deallocate the dynamically allocated memory in your vector?
* When building an object from the input file, different things can go wrong: the record represents an unknown type of vehicle, or record contains invalid data. How can the function loadData() detect what went wrong? Can you think at another solution than the one you have implemented?
* In this workshop you had to use std::list<sdds::Vehicle\*> and std::vector<sdds::Vehicle\*> to store a collection of vehicles as pointers. Could you have used std::list<sdds::Vehicle> and std::vector<sdds::Vehicle> instead? **Justify your answer!**

To avoid deductions, refer to code in your solution as examples to support your explanations.

**Submission**

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

Upload the source code and the reflection file to your matrix account. Compile and run your code using the latest version of the g++ compiler (available at /usr/local/gcc/9.1.0/bin/g++) 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 345\_w6\_p2

and follow the instructions.

**⚠️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.