Part D – Inheritance

# Derived Classes & Custom I/O Operators

Workshop 7

*(V1.0 – on the day of your lab, before submission, make sure there are no new updates or corrections)*

In this workshop, you will work with classes that make up hierarchal structure. The base or parent class will be a Vehicle that holds common attributes of a vehicle then the child class Car will be derived from the parent class. In addition to this hierarchy, we will define custom input/output operators for these classes.

## Learning Outcomes

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

- Inherit a derived class from a base class
- Shadow a base class member function with a derived class member function
- Access a shadowed member function that is defined in a base class
- Utilize custom input/output operators with these classes
- To describe to your instructor what you have learned in completing this workshop and the project milestones.

## Submission Policy

This workshop has only in lab section.

> *The DIY parts are now replaced with the project milestones under the **OOP-Project** repository.*

Workshop 7 is to be submitted during the workshop period from the lab. To get the mark for this workshop you must attend the lab.

Start the workshop few days earlier and come to the workshop sessions with questions and attendance.

Remember: you must be present at the lab in order to get credit for the workshop.
If you attend the lab period and cannot complete the workshop during that period, ask your instructor for permission to complete after the period.

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.

## CITATION AND SOURCES

When submitting Workshops, Project and assignment deliverables, a file called **sources.txt** must be present. 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.*
*Then add your name and your student number as signature*

*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.*
*You need to mention the workshop name or assignment name and also mention the file name and the parts in which you received the code for help.*
*Finally add your name and student number 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.

## LATE SUBMISSION PENALTIES:

Late submission: 1 to 6 days -50% after that submission rejected.

- If the content of sources.txt is missing, the mark for the whole workshop will be **0.**

## WORKSHOP DUE DATES

You can see the exact due dates of all assignments by adding -due after the submission command:
Run the following script from your account (use your professor's Seneca userid to replace profname.proflastname, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

**~profname.proflastname/submit 244/NXX/WS07/lab -due**<ENTER>

## COMPILING AND TESTING YOUR PROGRAM

All your code should be compiled using this command on matrix:
**g++ -Wall -std=c++11 -o ws (followed by your .cpp files)**
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>**

# Utils Module

For those of you who would like to reuse their previously developed functions and classes in later workshops, an empty module is added to the workshop called `Utils` (`Utils.cpp` and `Utils.h`). These files are empty and will be compiled with your workshop.
If you don't have any interest in reusing your previous functions, leave these files as they are, and they will not have any effect in the workshop.
If you have anything that you would like to add to the project, place their code in the `".cpp"` file and the definitions and prototypes in the `".h"` file. This header file will be included in the tester program.
In any case when submitting your work, make sure that these two files (empty or not) are submitted along with your workshop.

# Vehicle Module

The Vehicle module is a class that holds the information about a vehicle

To accomplish the above follow these guidelines:

Design and code a class named **`Vehicle.`**

Follow the usual rules for creating a module: (i.e. Compilation safeguards for the header file and the sdds namespace).

Create the following constants in the **Vehicle** header:

MIN_SPEED with a value of 100.
This constant integer number is used to set the speed limit to minimum.

MAX_SPEED with a value of 140

This constant integer number is used to check the speed limit.

## PRIVATE MEMBERS:

A Vehicle will have the following data members:

- `m_speed`
  This is an integer value that represents the speed of a vehicle.
- `m_noOfSeats`
  This is an **integer value that** represents the number of seats of a vehicle.

## PUBLIC MEMBERS:

Upon instantiation, a `Vehicle` object receives no information or information on all of the above two values. If the speed is greater than or equal to MIN _SPEED and if the number of seats is greater than zero, then the object accepts them. Other-wise, the object assumes a safe empty state.

```
bool finetune();
```

This member function finetune a vehicle's speed if it crosses the maximum speed limit. If the speed of a vehicle is more than the MAX_SPEED then it will set the speed to MIN_SPEED. Otherwise, instead of performing the above, print out the following:

```
"The car cannot be tuned"<newline>
```

If the fine tune was successful, return **true**. Otherwise return **false**.

```
ostream& display(ostream& os) const;
```

This member function will display the current details of the Vehicle. This public member function receives a reference to an ostream object. **Review the materials on custom I/O** for reference if difficulties arise.

- If the vehicle`s speed is greater than OR equal to MIN_SPEED and the number of seats is GREATER THAN zero, then print the speed and number of seats as follows:
  Format:
  `"|[speed]|[noOfSeats]"<newline>`
  Sample output:
  `|120|5`

  After that we also have to check if the vehicle`s speed is more than the MAX_SPEED, then print out the following:

  `"Car has to be fine-tuned for speed limit" <newline>`
- If vehicle is in empty state, then it will print:
  `"This Car is not initiated"<newline>`
- Lastly at the end of the function **return the parameter os**.
  **Refer to the sample output for details.**

```
istream& input(istream& in);
```
The member function facilitates setting the values of a Vehicle through the use of user input. This member function receives a reference to an `istream` object and returns a reference to the modified `istream` object.

The order in which input is supplied is as follows:

1. Create a temporary **int variable** to store a **speed**.
2. Print out the following:
   `"Enter the Vehicle`s speed: "`
3. Accept user input from the **in** parameter into this temporary **integer variable**.
4. Create a temporary **integer variable** to store number of seats.
5. Print out the following:
   `"Enter no of seats: "`
6. Accept user input from the **in** parameter into this temporary **integer variable**.

7. Utilize these two variables to then set the current **Vehicle** object. Consider the use of the 2-arg constructor here with temporary objects to set the current Vehicle.

### Operator Overload Helpers

```
ostream& operator<<(ostream& os, const Vehicle& v);
```

This helper receives a reference to an `ostream` object and an unmodifiable reference to a Vehicle object and returns a reference to the `ostream` object

```
istream& operator>>(istream& in, Vehicle& V);
```

This helper receives a reference to an `istream` object and a reference to a Vehicle object and returns a reference to the `istream` object.

## CAR MODULE

The **Car** module will be a child of **Vehicle**. It will include some new data members and functions on top of what was established in Vehicle.

Create a class called **Car** that derives from **Vehicle**.

Create the following constants in the **Car** header:

`MAX_CHAR` with a value of 20.
This constant **integer** number is used to determine the max length of the type for a car.

`MIN_YEAR` with a value of 2000

This constant integer number is used to check the minimum valid registration year of a car.

### PRIVATE MEMBERS:

A Car will have the following data members:

- `m_carType`

- This is a **char array** with a max length of MAX_CHAR representing the type of the car. Recall the need for a **null byte**.
- `m_regYear`
  This is an **integer value that** represents a registration year of a car. This value cannot be less than MIN_YEAR.

## PUBLIC MEMBERS:

**Constructors**

Car objects will be making use of **constructors** to handle their creation. There are two **constructors** that are required:

1. Default constructor (no argument)– This constructor should set a Car object to **a safe empty state**. Only the **data members new** to Car will need to be set. Rely on the **base class's default constructor** to set the data members inherited from **Vehicle**.

1. 4 Argument Constructor – This constructor will take in 4 parameters:
   a. A **const char pointer** that represents the type of the Car.
   b. An **integer value that** represents the registration year of a Car which can not be less than MIN_YEAR.
   c. An **integer value** that represents the speed of a Car.
   d. An **integer value that** represents the number of seats of a Car.

**Utilize the 2 arguments constructor from Vehicle when coding this 4-arg constructor.**

### Other Members

```
void finetune();
```

This member function will fine tune a car only if the registration year is a valid year. It relies on the Vehicle finetune() function to fine tune the car`s speed limit.

- If the registration year of a vehicle is less than the MIN_YEAR then it print out the following:

  `"Car cannot be tuned and has to be scraped"<newline>`

- If the registration year is valid then it will fine tune the car's speed limit to MIN_VALUE by calling Vehicle's finetune() function and will print out the following command if it returns true,

```
"This car is finely tuned to default speed"<newline>
```

```
ostream& display(ostream& os) const;
```

This member function will display the current details of a Car. It works similar as the Vehicles **display** function and calls that function. Firstly, it checks if the Car is in an empty state.

If the Car is in an **empty** state the following will be printed:

```
"Car is not initiated yet"<newline>
```

If the Car isn't empty, then it will print like the following sample output:

```
sedan                2019|120|5
```

**Print Car type width as 20. The words in blue are elements new to Car while the red are from Vehicle. Consider how you can call Vehicle's display to make this occur.** Lastly at the end of the function **return the parameter os**.
**Refer to the sample output for details.**

```
istream& input(istream& in);
```

The member function facilitates setting the values of a Car through the use of user input. It goes through a process very similar to that of the Vehicle **input** function.

The order in which input is supplied is as follows:

1. Create a temporary **character array** to store a **carType.** Use MAX_CHAR to determine the size.
2. Print out the following:

   ```
   "Enter the car type: "
   ```

3. Accept user input from the **in** parameter into this temporary variable. Consider the use of the **getline**() functions as we are extracting a string of a certain length.
4. Create a temporary **int** variable to store the **registration year**.
5. Print out the following:

   ```
   "Enter the car registration year: "
   ```

6. Accept user input from the **in** parameter into this temporary variable.
7. Copy temp variable to the data member (by following validation if any).

8. To accept user input to set the other data members inherited from Vehicle, **call vehicle's input function here**.
9. Lastly return the parameter **in**.

## Operator Overload Helpers

```
ostream& operator<<(ostream& os, const Car& C);
```

This helper receives a reference to an `ostream` object and an unmodifiable reference to a Car object and returns a reference to the `ostream` object;

```
istream& operator>>(istream& in, Car& C);
```

This helper receives a reference to an `istream` object and a reference to a Car object and returns a reference to the `istream` object.

In-Lab Main Module

```
// OOP244 Workshop 7: Inheritance
// File: VehicleTester.cpp
// Version: 1.0
// Date: 02/28/2020
// Author: Nargis Khan
// Description:
// This file tests lab section of your workshop
////////////////////////////////////////////
// ----------------------------------------------------------
// Name            Date            Reason
// Nargis
////////////////////////////////////////////////////////////////

#include<iostream>
#include "Vehicle.h"
#include "Car.h"
using namespace std;
using namespace sdds;

int main() {
   cout << "-----------------------------------------" << endl;
   cout << "*** Checking Car default constructor ***" << endl;
   Car c1;
   c1.display(cout);

   cout << "-------------------------------------------------" << endl;
   cout << "*** Checking Car 4 arg constructor (invalid ***)" << endl;
```

```
    Car c2("", 2016, 100, 5);
    Car c3("SUV", 1999, -120, 8);
    Car c4("sports", 1998, 110, 0);
    c2.display(cout);
    c3.display(cout);
    c4.display(cout);
    cout << "-----------------------------------------------" << endl;

    cout << "*** Checking Car 4 arg constructor (valid) ***" << endl;
    Car c5("SEDAN", 2016, 120, 5);
    Car c6("SUV", 2018, 110, 8);
    Car c7("SPORTS", 2020, 130, 2);
    c5.display(cout);
    c6.display(cout);
    c7.display(cout);
    cout << "-----------------------------------------------" << endl;

    cout << "*** Checking custom input and output operator ***" << endl;
    cin >> c1;
    cin.ignore(2000, '\n');
    cout << c1;
    cout << "-----------------------------------------------" << endl;

    cout << "*** Checking Car finetune ***" << endl;
    c1.finetune();
    cout << c1;
    cout << "-----------------------------------------------" << endl;

    cout << "*** Checking Car finetune for nonempty car ***" << endl;
    cin >> c4;
    c4.finetune();
    cout << c4;

    return 0;

}
```

## EXECUTION EXAMPLE RED VALUES ARE USER ENTRY

```
-----------------------------------------------
*** Checking Car default constructor ***
Car is not initiated yet
-----------------------------------------------
*** Checking Car 4 arg constructor (invalid ***)
Car is not initiated yet
Car is not initiated yet
Car is not initiated yet
-----------------------------------------------
*** Checking Car 4 arg constructor (valid) ***
SEDAN               2016|120|5
SUV                 2018|110|8
```

```
SPORTS              2020|130|2
--------------------------------------------------
*** Checking custom input and output operator ***
Enter the car type: sedan
Enter the car registration year: 2019
Enter the Vehicle`s speed: 150
Enter no of seats: 5
sedan               2019|150|5
Car has to be fine tuned for speed limit
--------------------------------------------------
*** Checking Car finetune ***
This car is finely tuned to default speed
sedan               2019|100|5
--------------------------------------------------
*** Checking Car finetune for nonempty car ***
Enter the car type: sports
Enter the car registration year: 1999
Enter the Vehicle`s speed: 120
Enter no of seats: 2
Car cannot be tuned and has to be scraped
Car is not initiated yet
```

## 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 sources.txt, Vehicle module, Car module and the VehicleTester.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.):

**~profname.proflastname/submit 244/NXX/WS07/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.

# 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 and how you solved them. Add any other comments you wish to make.

The reflection should also contain any thoughts and learning from the final project milestones when possible.

## Reflection Submission

You can submit your reflection 4 to 6 days after your lab session. Upload reflect.txt to your matrix account. Then, run the following command from your account (use your professor's Seneca userid to replace profname.proflastname, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~profname.proflastname/submit 244/NXX/WS07/reflect <ENTER>
```

and follow the instructions generated by the command.

To see the when you can submit your reflection and when the submission closes, (like any other submission) add **-due** to the end of your reflection submission command:

```
~profname.proflastname/submit 244/NXX/WS07/reflect -due<ENTER>
```