/\*

Course\_Name: OOP 345

Section: A

Title: Workshop 6 Part 2

Module: Reflect.txt

Student\_Name: Zhaokai\_Guan

Student\_ID: 130988215

\*/

For this week’s content, I was exposed to some more details about standard library. More importantly, the concept of containers and iterators from the STL was introduced by my prof. I have overheard that this part is extremely important to those who are willing to use C++ in their future careers. And I am very glad I could have the chance to practice the knowledge I learned during the lecture.

1. Why do you need to deallocate the dynamically allocated memory in your vector?

The best way to understand this is comparing the WS04 and this workshop. In WS04, we created a pointer of an array and build the class based on the class relationships. In that workshop we had to manage the dynamically allocated memory pointed by the pointer of pointers. But in the workshop, as we learned, the STL containers manage the memory internally, which means we do not need to worry about the memory allocation for the pointer of pointers.

BUT!!! Remember each element inside this vector is a vehicle pointer!!!

Each element of the vector m\_vehicle can be pointing to a chunk of dynamically allocated memory.

class Autoshop {

std::vector<Vehicle\*> m\_vehicle{};

…

So that is why we need to deallocate every single element’s memory.

Autoshop::~Autoshop(){

// Understand the STL from here//

// Level 1 is managed by the STL lib//

// But remember m\_vehicle is vector of object pointers//

// They are pointing to some dynamic memory//

// So, you need to deallocate those Level2 memories pointed by the vector elements//

for (auto & i:m\_vehicle){

delete i;

}

}

1. 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?

My loadData function is using the try and catch pattern to collect any potential exceptions, which we learned from the week 5.

Here is how I designed the logic for error catching.

If try triggers any exceptions, catch will collect the exceptions respectively based on the exception’s type.

For example, if it is a invalid data contained inside one of the record, the one argument constructor will throw the std::string, and the first catch will collect that message and print it out onto the screen.

Same theory applies to the char errTag. Only difference is that it is generated from the utility module.

try {

sdds::Vehicle\* aVehicle = sdds::createInstance(file);

if (aVehicle)

as += aVehicle;

}

catch (std::string err) {

std::cerr << err << std::endl;

}

catch (char errType) {

std::cerr << "Unrecognized record type: ["<<errType<<"]\n";

}

If I can not use this, I assume my old friend dummy object will come into play. Here is how I would implement it.

a. Create a Car constructor takes char as argument, and the parameter name would be errorTag. I would set the m\_condtion to the type tag, and the m\_topSpeed is set to 0.

b. Inside the createInstance function from the utilities module, if the type is unknown, call the Car constructor I declared above with the unsigned int 1. And Ignore the current line until the new line character.

c. Inside the Car constructor (istream& in), if there is invalid record, the m\_Conditon of current object will be set to ‘!’. And the remaining of the line will be ignored.

sdds::Vehicle\* aVehicle = sdds::createInstance(file);

if (aVehicle.condtion()==!)

std::cerr<<“invalid record\n”

delete aVehicle;

else if(aVehicle.topSpeed() == 0)

std::cerr << "Unrecognized record type: ["<<aVehicle.Condition()<<"]\n";

delete aVehicle;

else

as += aVehicle;

Compared to the try and catch pattern, this pattern is slow and stupid. We should avoid this kind of ugly design.

1. 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!**

First of all, we need to know Vehicle class is an interface aka pure abstract class. YOU CAN NOT INSTANTIATE ANY OBJECTS FROM AN ABSTRACT CLASS. Don’t even need to mention a pure abstract class.

class Vehicle

{

public:

virtual double topSpeed() const = 0;

virtual void display(std::ostream&) const = 0;

virtual std::string condition() const = 0;

virtual ~Vehicle() {};

};

But you can declare a pointer under an abstract class, and let it point to the objects on the different inheritance hierarchy. This is the backbone of our workshop’s design.