Workshop 6

STL Containers

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

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 *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

The solution to the in-lab part of this workshop consists of three modules:

- **w6** (supplied)
- **Sale**
- **Product**

Enclose all your source code within the **sict namespace** and include the necessary guards in each header file. The output from your executable running Visual Studio with the following command line argument should look like

```
Command Line : C:\Users\...\Debug\in_lab.exe Sales.dat

Product No      Cost
     10012     34.56
     10023     45.67
     10234     12.32
     10056     67.54
     10029     54.12
     10034     96.30
     Total    310.51
```

The input for testing your solution is stored in a user-prepared file. The name of the file is specified on the command line as shown in red above. The file is supplied with this workshop. The contents of this file are

```
10012 34.56
10023 45.67
10234 12.32
10056 67.54
10029 54.12
10034 96.30
```

Each record in the file consists of a product number and its price.

Product Module

Design and code an inheritance hierarchy named **iProduct** for holding data on products for sale. Your interface to the hierarchy uses the field width (**int FW**) defined outside the translation unit and specifies the following public member function requirements:

- **double price() const** – a query that returns the price of a product.
- **void display(std::ostream& os) const** – a query that displays the information about the product. Use the field width defined in the supplied **w6** module.

Your interface also includes prototypes to meet the following global function requirements:

- **operator<<(std::ostream& os, const iProduct& p)** – a helper that inserts the product information into the stream **os**.
- **iProduct* readRecord(std::ifstream& file);** – a function that reads a single record from file object **file**, allocates memory for a product in the hierarchy, stores the information read into the product object and returns its address.

A concrete class named **Product** implements this interface and includes

- a constructor that receives and stores the product number and its price before tax.

Sale Module

Design and code a class named **Sale** for managing a sequence of **iProduct** objects. Your class design includes the following public member functions:

- A single argument constructor that receives the address of an unmodifiable C-style null terminated string that contains the name of the file holding records of **iProduct** type. This function opens the file, reads one record from the file, moves that record into an STL container and keeps reading records until there are none left to be read. If this function fails to open the file, it throws an exception with an appropriate message.
- **void display(std::ostream& os) const** – a query that displays the products that make up the sale and the total price in the format shown above. In coding this function use a range based for and the field width specified outside the translation unit.
- A destructor that deallocates the memory for your **iProduct** objects.

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_w6_lab<ENTER>

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

SPECIFICATIONS – AT HOME

The at-home part of this workshop updates the following modules to include tax:

- **Product** (from in-lab)
- **Sale** (from in-lab)

The output from your executable running Visual Studio with the following command line argument should look like

```
Command Line : C:\Users\...\Debug\at_home.exe Sales.dat

Product No      Cost Taxable
     10012     34.56
     10023     45.67 HST
     10234     12.32 PST
     10056     67.54
     10029     54.12 HST
     10034     96.30
     Total    324.47
```

The input for testing your solution is stored in a user-prepared file. The name of the file is specified on the command line as shown in red above. The file is supplied with this workshop. The contents of this file are

```
10012 34.56
10023 45.67 H
10234 12.32 P
10056 67.54
10029 54.12 H
10034 96.30
```

Each record in the file consists of a product number, its price and optionally its taxable status.

Product Module

Derive a class named **TaxableProduct** from your **Product** class to add taxation functionality to the hierarchy. Your derived class includes

- an enumeration list of tax types and a class variable with corresponding rates
    - HST 0.13
    - PST 0.08

Your derived class also includes the following upgrades:

- A three-argument constructor that receives and stores the product number, its price and its tax status.
- **double price() const** – a query that returns the price of a product tax included.
- **void display(std::ostream& os) const** – a query that displays the information about the product and appends the tax status as shown in the example above.
- **iProduct* readRecord(std::ifstream& file)** – this function reads a single record from file object **file**, allocates memory for a product in the hierarchy, stores the information read into the product object and returns its address. The information includes the product number and price and optionally the tax status as shown in the sample input above. If the record does not include a tax status character this function allocates memory for a **Product** object. If the record has a tax status character this function allocates memory for a **TaxableProduct** object.

Sale Module

Upgrade the **Sale** module to include **"Taxable"** in the heading for displaying the product data as shown in the sample output above.

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 reason for using the vector container rather than any other available in the STL.
- Why do you need to deallocate the dynamically allocated memory in your vector?
- How the range-based for simplifies coding in this case.

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

Include all corrections to the Quiz(zes) 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_w6_home<ENTER>

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