

Winter 2017.

Point Of Sale Application

Final Project

Milestone3 (V1.0)

Your job for this project is to prepare an application that manages the list of items stocked in a store for sale. Your application keeps track of the quantity of items in the store, saved in a file and updates their quantity as they are sold.

The types of items kept in store are Perishable or Non-perishable.

- Perishables: Items that are mostly food and vegetable and have expiration date.
- Non-perishables: Items that are for household use and don't have expiry date.

To prepare the application you need to create several classes that encapsulate the different tasks at hand.

CLASSES TO BE DEVELOPED

The classes required by your application are:

Date	A class that manages date and time.
Error	A class that keeps track of the errors occurring during data entry and user interaction.
PosIO	<p>An abstract class that enforces iostream read and write functionality for the derived classes. An instance of any class derived from "PosIO" can read from or write to the console, or be saved to or retrieved from a text file.</p> <p>Using this class the list of items can be saved to a file and retrieved later, and individual item specifications can be displayed on screen (in detail or as a bill item) or read from keyboard.</p>
Item	A class derived from PosIO, containing general information about an item in the store, like the name, Stock Keeping Unit (SKU) number, price, etc.
NonPerishable	A class derived from the "Item" class that implements the requirements of the "PosIO" class. (i.e. implements the pure virtual methods of the PosIO class).
Perishable	A class holding information for a Perishable item derived from the "NonPerishable" class that re-implements the requirements of the "PosIO"

PosApp

The class that manages Perishable and Non-Perishable items in a file. This class manages the listing, adding and updating the data file as the items are bought or sold in the store.



PROJECT DEVELOPMENT PROCESS

Your development work on this project has 5 milestones and therefore is divided into 5 deliverables. Shortly before the due date of each deliverable a tester program and a script will be provided to you. Use this tester program to test your solution and use the “submit” command for each of the deliverables as you do for your workshop.

Since the design of this project is an ongoing process, you may have to make minor changes to the previous milestones if there is a bug or incorrect design specs, when a new milestone is published.

The approximate schedule for deliverables is as follows

- | | |
|--|----------------------------------|
| • Date and Error class | 8 days Due: Mar 15 th |
| • PosIO class | 1 day Due: Mar 16 th |
| • Item class | 6 days Due: Mar 22 nd |
| • Perishable and NonPerishable classes | 7 days Due: Mar 29 th |
| • PosApp class. | 9 days Due: Apr 7 th |

FILE STRUCTURE FOR THE PROJECT

Each class will have its own header (.h) file and implementation (.cpp) file. The names of these files should be the class name.

In addition to the header files for each class, create a header file called “POS.h” that defines general values for the project, such as:

<code>TAX (0.13)</code>	The tax rate for the goods
<code>MAX_SKU_LEN (7)</code>	The maximum size of an SKU code
<code>MIN_YEAR (2000)</code>	The min year used to validate year input
<code>MAX_YEAR (2030)</code>	The max year used to validate year input
<code>MAX_NO_ITEMS (2000)</code>	The maximum number of records in the data file.

Include this header file wherever you use these values.

Enclose all the code developed for this application within the **ict** namespace.

Make sure all your header files guard against multiple inclusions by adding the following commands at the very beginning of each header file:

```
#ifndef ICT_HeaderFileName_H_
#define ICT_HeaderFileName_H_
```

and adding the following command to the very end of each header files:

```
#endif
```

The “HeaderFileName” in the first two commands is replaced with the name of your header file; for example, if your header file name is PosApp.h then the commands will be:

```
#ifndef ICT_POSAPP_H__
#define ICT_POSAPP_H__
```

MLESTONE 1:

Preliminary task

To kick-start the first milestone download the Visual Studio project, or individual files for milestone 1 from <https://github.com/Seneca-244200/OOP-Milestone1>

THE DATE CLASS

The Date class encapsulates a single date and time value in the form of five integers: year, month, day, hour and minute. The date value is readable by an istream and printable by an ostream using the following format: **YYYY/MM/DD, hh:mm** or **YYYY/MM/DD** if the class is to hold only the date without the time. (i.e. if `m_dateOnly` is true; see “`bool m_dateOnly;`”)

Complete the implementation of the Date class under the following specifications:

Member Data (attributes):

```
int m_year;
```

Year; a four digit integer between MIN_YEAR and MAX_YEAR, as defined in “POS.h”

```
int m_mon;
```

Month of the year, between 1 to 12

```
int m_day;
```

Day of the month, note that in a leap year February has 29 days, (see `mday()` member function)

```
int m_hour;
```

A two digit integer between 0 and 23 for the hour of a day.

```
int m_min;
```

A two digit integer between 0 and 59 for the minutes passed the hour

```
int m_readErrorCode;
```

Error code which identifies the validity of the date and, if erroneous, the part that is erroneous.

Define the possible error values in the Date header-file as follows:

```
NO_ERROR    0    -- No error - the date is valid
CIN_FAILED  1    -- istream failed on accepting information using cin
YEAR_ERROR  2    -- Year value is invalid
MON_ERROR   3    -- Month value is invalid
DAY_ERROR   4    -- Day value is invalid
HOUR_ERROR  5    -- Hour value is invalid
MIN_ERROR   6    -- Minute value is invalid
```

`bool m_dateOnly;`

A flag that is true if the object is to only hold the date and not the time. In this case the values for hour and minute are zero.

Private Member functions (private methods):

`int value()const; (this function is already implemented and provided)`

This function returns a unique integer number based on the date-time. You can use this value to compare two dates. If the value() of one date-time is larger than the value of another date-time, then the former date-time (the first one) follows the second.

`void errCode(int errorCode);`

Sets the `m_readErrorCode` member variable to one of the possible values listed above.

`int mdays()const; (this function is already implemented and provided)`

This function returns the number of days in the month based on `m_year` and `m_mon` values.

`void set(); (this function is already implemented and provided)`

This function sets the date and time to the current date and time of the system.

`void set(int year, int mon, int day, int hour, int min);`

Sets the member variables to the corresponding arguments and then sets the `m_readErrorCode` to `NO_ERROR`.

Constructors:

No argument constructor: Sets the `m_dateOnly` attribute to false and then sets the date and time to the current system's date and time using the `set()` function.

Three argument constructor: This constructor sets the `m_dateOnly` attribute to true and then accepts three integer arguments to set the values of `m_year`, `m_mon` and `m_day` and sets `m_hour` and `m_min` to zero. It also sets the `m_readErrorCode` to `NO_ERROR`.

Five argument constructor: This constructor sets the `m_dateOnly` attribute to false and then accepts five integer arguments to set the values of `m_year`, `m_mon`, `m_day`, `m_hour` and `m_min`. It also sets the `m_readErrorCode` to `NO_ERROR`. The last argument of this constructor (int min) should have a default value of "0" so the constructor can be called with four arguments too.

Public member-functions (methods) and operators:

Relational operator overloads:

```
bool operator==(const Date& D)const;
bool operator!=(const Date& D)const;
bool operator<(const Date& D)const;
bool operator>(const Date& D)const;
bool operator<=(const Date& D)const;
bool operator>=(const Date& D)const;
```

These operators return the result of comparing the left operand to the right operand. These operators use the `value()` member function in their comparison. For example, `operator<` returns true, if `this->value()` is less than `D.value()`; otherwise returns false.

Accessor or getter member functions (methods):

`int` `errCode()``const`;

Returns the `m_readErrorCode` value.

`bool` `bad()``const`;

Returns true if `m_readErrorCode` is not equal to zero.

`bool` `dateOnly()``const`;

Returns the `m_dateOnly` attribute.

`void` `dateOnly(bool value)`;

Sets the `m_dateOnly` attribute to the “value” argument. Also if the “value” is true, then it will set `m_hour` and `m_min` to zero.

IO member-functions (methods):

`std::istream&` `read(std::istream& istr = std::cin)`;

Reads the date in the following format: YYYY/MM/DD (e.g. 2015/03/24) from the console if `_dateOnly` is true or in the following format: YYYY/MM/DD, hh:mm (e.g. 2015/03/24, 22:15) if `_dateOnly` is false. This function does not prompt the user. If the `istream(istr)` object fails at any point, this function sets `m_readErrorCode` to `CIN_FAILED` and does NOT clear the `istream` object. If the `istream(istr)` object reads the numbers successfully, this function validates them. It checks that they are in range, in the order of year, month and day (see the general header-file and the `mday()` function for acceptable ranges for years and days respectively). If any number is not within range, this function sets `m_readErrorCode` to the appropriate error code and omits any further validation. Irrespective of the result of the process, this function returns a reference to the `istream(istr)` object.

`std::ostream&` `write(std::ostream& ostr = std::cout)``const`;

This function writes the date to the `ostream(ostr)` object in the following format: YYYY/MM/DD, if `m_dateOnly` is true or YYYY/MM/DD, hh:mm if `m_dateOnly` is false. Then it returns a reference to the `ostream(istr)` object.

Non-member IO operator overloads: (Helpers)

After implementing the `Date` class, overload the `operator<<` and `operator>>` to work with `cout` to print a `Date`, and `cin` to read a `Date`, respectively, from the console.

Use the `read` and `write` member functions. DO NOT use friends for these operator overloads.

Include the prototypes for these helper functions in the `date` header file.

TESTER PROGRAMS:

There are 6 tester programs to test your implementation of the Date and the Error class:

01-DefValTester.cpp	Tests defined values
02-ConstructorTester.cpp	Tests Date constructors and relative operations
03-LogicalOperator.cpp	Tests the logical operator overloads
04-DateErrorHandling.cpp	Tests the error handling in Date
05-ErrorTester.cpp	Tests the Error class
244_ms1_tester.cpp	Tests all the above.

The last program does all the tests, but for your convenience the full test is broken down into 5 five separate files (the first five) to help you test your code in small steps earlier throughout development.

THE ERROR CLASS

The Error class encapsulates error processing using a dynamic C-style string and a flag for the error state of other classes.

Later in the project, if needed in a class, an Error object is created and if an error occurs, the object is set to a proper error message.

Then, calling the **isClear()** method can determine if an error has occurred or not and printing the object to **cout** shows the error message to the user.

Private member variable (attribute):

Error has only one private data member (attribute):

```
char* m_message;
```

Constructors:

No Argument Constructor, (default constructor):

```
Error();
```

Sets the **m_message** member variable to **nullptr**.

Constructors:

```
Error(const char* errorMessage);
```

Sets the **m_message** member variable to **nullptr** and then uses the **message()** setter member function to set the error message to the **errorMessage** argument.

```
Error(const Error& em) = delete;
```

A deleted copy constructor to prevent copying of an Error object.

Public member functions and operator overloads (methods):

Error& operator=(const Error& em) = delete;

A deleted assignment operator overload to prevent assignment of an Error object to another.

void operator=(const char* errorMessage);

Copies the string pointed to by **errorMessage** into **m_message** and returns nothing by:

- De-allocating the memory pointed to by **m_message**
- Allocating memory of the length of **errorMessage + 1** and storing its address in the **m_message** data member.
- Copying the C-string pointed to by **errorMessage** into ***m_message**.

You can accomplish this by reusing your code and calling the following member functions:

Call **clear()** and then call the setter **message()** function.

virtual ~Error();

de-allocates the memory pointed to by **m_message**.

void clear();

de-allocates the memory pointed to by **m_message** and then sets **m_message** to **nullptr**.

bool isClear()const;

returns true if **m_message** is **nullptr**.

void message(const char* value);

Sets the **m_message** of the Error object to a new value by:

- de-allocating the memory pointed to by **m_message**.
- allocating memory of the length of **value + 1** and storing the address in the **m_message** data member.
- copying the C-string pointed to by **value** into ***m_message**.

operator const char*() const;

returns the address stored in **m_message**.

operator bool()const;

Exactly like **isClear()**; returns true if **m_message** is **nullptr**

Helper operator overload:

Overload **operator<<** so the Error can be printed to an **ostream** object.

If Error **isClear**, nothing is printed, otherwise the C-string pointed to by **m_message** is printed.

MILESTONE 1 SUBMISSION

If not on matrix already, upload **Error.h**, **Error.cpp**, **Date.h**, **Date.cpp** and **244_ms1_tester.cpp** to your matrix account. Compile and run your code and make sure that everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professor's Seneca userid)

```
~profname.proflastname/submit 244_ms1 <ENTER>
```


and follow the instructions.

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.

MILESTONE 2: THE POSIO INTERFACE

The **PosIO** class enforces inherited classes to implement functions that work with **fstream** and **istream** objects.

Code the **PosIO** class in the **PosIO.h** file provided in milestone2 repository (No CPP file) on github: <https://github.com/Seneca-244200/OOP-Milestone2>

You do not need the Date or Error class for this milestone.

Pure virtual member functions (methods):

The **PosIO** class, being an interface, has only pure virtual member functions (methods). These functions have the following names:

1- **std::fstream& save(std::fstream& file)const**

Is a query (does not modify the owner) and receives and returns references of **std::fstream**.

*In future milestones children of **PosIO** will implement this method, when they are to be saved in a file.*

2- **std::fstream& load(std::fstream& file)**

Is a modifier that receives and returns references to an **std::fstream** object.

*In future milestones children of **PosIO** will implement this method, when they are to be read from a file.*

3- **std::ostream& write(std::ostream& os, bool linear)const**

Is a query and returns a reference to a **std::ostream** object. This function receives two arguments: the first is a reference to an **std::ostream** object and the second is a **bool**.

*In future milestones children of **PosIO** will implement this method when they are to be printed on the screen in two different formats:*

***linear**: the class information is to be printed in one line*

4- **std::istream& read(std::istream& is)**

Returns and receives references to an **std::istream** object.

*In future milestones children of **PosIO** will implement this method when their information is to be received from console.*

As you already know, these functions only exist as prototypes in the class declaration in the header file. Read this: <https://scs.senecac.on.ca/~oop244/pages/content/abstr.html#pur>

After implementing this class, compile it with **Myfile.cpp**, **MyFile.h** and **244_ms2_tester.cpp**. The program should compile with no error and using the tester program you will be able to read and append text to the **PosIO.txt** file.

MILESTONE 2 SUBMISSION

If not on matrix already, upload **PosIO.h**, **MyFile.h**, **MyFile.cpp** and the tester to your matrix account. Compile and run your code and make sure everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 244_ms2 <ENTER>
```

and follow the instructions.

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.

MILESTONE 3: THE ITEM CLASS

Create a class called Item. The class Item is responsible for encapsulating a general **PosIO** item.

Although the class Item is a **PosIO** (inherited from **PosIO**) it will not implement any of the pure virtual member functions, therefore it remains abstract.

The class `Item` is implemented under the `ict` namespace. Code the `Item` class in the `Item.cpp` and `Item.h` files provided in OOP-Milestone3 repository on github:

<https://github.com/Seneca-244200/OOP-Milestone3>

You do not need the `Date` class for this milestone.

Item Class specs:

Private Member variables:

m_sku: C-style null-terminated character array, `MAX_SKU_LEN + 1` characters long

This character array holds the SKU (barcode) of the items as a string.

m_name: Character pointer

This character pointer points to a dynamically allocated string that holds the Item's name

m_price: Double

This variable holds the Item's Price

m_taxed: Boolean

This variable is true if this item is taxed, false otherwise

m_quantity: Integer

This variable holds the Item's on-hand (current) quantity.

Public member functions and constructors

No argument Constructor;

This constructor sets the item to a safe recognizable empty state. All numeric values are set to zero in this state.

Four argument Constructor;

This constructor receives 4 values: the SKU, the Name, the price and the tax status.

The constructor:

- Copies the SKU into the corresponding member variable up to `MAX_SKU_LEN` characters.
- Allocates enough memory to hold the name in the `m_name` pointer and then copies the name stored at the received address into the allocated memory at `m_name`.
- Sets the quantity on hand to zero.
- Sets the remaining member variables to the received parameter values.
- If the tax-status value is not provided, set the `m_taxed` flag to the default value `"true"`

Copy Constructor;

See below:

Dynamic memory allocation necessities

Implement the copy constructor and the `operator=` so that the `Item` referenced in the parameter is copied from and assigned to another `Item` safely and without any memory leak. Also, implement a virtual destructor to ensure that any memory allocated for `m_name` is freed when the `Item` is destroyed.

In `operator=`, if the current object is being set to another `Item` that is in a safe empty state, the copy assignment operation is to be ignored.

Accessors

Modifiers (Setters):

Create the following modifiers (setter functions) to set the corresponding member variables:

- **sku**
- **price**
- **name**
- **taxed**
- **quantity**

All the above modifiers return **void**.

Queries (Getters):

Create the following queries to return the values or unmodifiable addresses of the member variables: (these getter methods do not receive any arguments and they are not to change the owner)

- **sku**, returns the address of an unmodifiable character string
- **price**, returns a double
- **name**, returns the address of an unmodifiable character string
- **taxed**, returns a bool
- **quantity**, returns an integer

Also:

- **cost**, returns double
Cost returns the after-tax cost of the item. If the Item is not taxed, the return value is the price.
- **isEmpty**, returns bool
isEmpty return true if the Item is in a safe empty state.

Note: All the above queries are constant methods, which means they CANNOT modify the owner.

Member Operator overloads:

operator== : receives the address of an unmodifiable character string and returns a bool.

This operator compares the character string at the received address to the Item's SKU, if they are the same, it will return **true**; otherwise **false**. Note that this operator cannot modify the owner.

operator+= : receives an integer and returns an integer.

This operator adds the received value to the Item's quantity on hand and returns the sum.

operator-= : receives an integer and returns an integer.

This operator reduces the Item's quantity on hand by the received value and returns the quantity on hand after reduction.

Non-Member operator overload:

operator+= : receives a modifiable reference to a **double** as the left operand and an unmodifiable reference to an **Item** as the right operand and returns a double value.

This operator multiplies the cost of the **Item** by the quantity of the **Item** and then adds that value to the left operand and returns the result.

Essentially, this operator adds the total cost of the referenced **Item** to the left operand and then returns it.

Non-member IO operator overloads:

After implementing the **Item** class, overload the **operator<<** and **operator>>** to print **Item** objects to **ostream** console objects (cout), and to read **Item** objects from **istream** console objects (cin). Use the **write()** and **read()** methods of the **PosIO** class to implement these operator overloads.

Make sure that the prototypes of the functions are in **Item.h**.

If not on matrix already, upload [POS.h](#), [PosIO.h](#), [Item.h](#), [Item.cpp](#) and the tester to your matrix account. Compile and run your code and make sure everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 244_ms3 <ENTER>
```

and follow the instructions.

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.