Winter 2017.

Point Of Sale Application

Final Project

Milestone 4

(V1.1) replaced second load function with write

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 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.

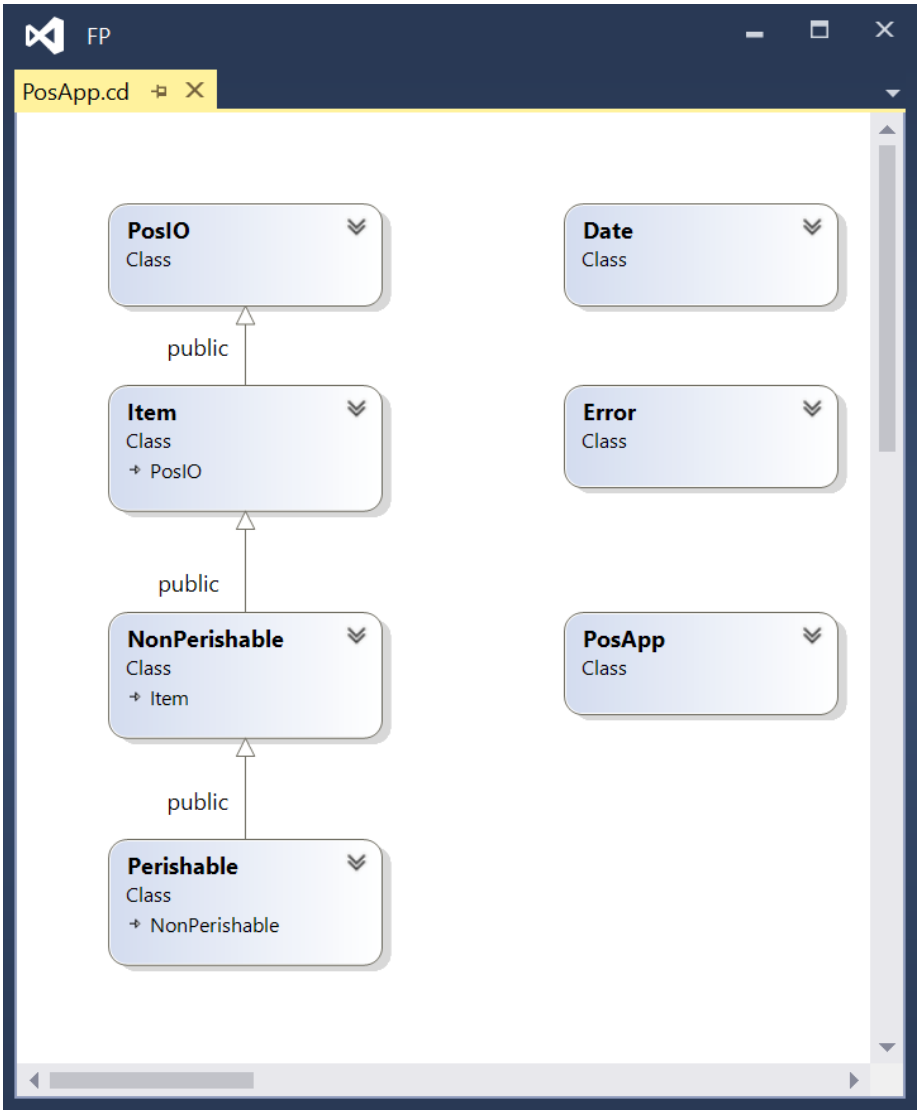
Using this class the list of items can be saved to a file and retrieved later, and individual item specifications can be displayed on screen (in detail or as a bill item) or read from keyboard.

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

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:

```
TAX (0.13)              The tax rate for the goods
MAX_SKU_LEN (7)         The maximum size of an SKU code

MIN_YEAR (2000)         The min year used to validate year input
MAX_YEAR (2030)         The max year used to validate year input

MAX_NO_ITEMS (2000)      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__
```

MILESTONE 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/BTP-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-funtions (methods):

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

Reads the date in the following format: YYYY/MM/DD (e.g. 2015/03/24) from the console if _date only 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
200_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 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 by **m_message.**

```
void clear();
```
de-allocates the memory pointed 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 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 **200_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 200_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 **iostream** objects.

Code the **PosIO** class in the **PosIO.h** file provided in milestone2 repository (No CPP file) on github: https://github.com/Seneca-244200/BTP-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 **200_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 200_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 BTP-Milestone3 repository on github:

https://github.com/Seneca-244200/BTP-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 200_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.

MILESTONE 4: THE NONPERISHABLE AND PERISHABLE CLASSES

Part one: NonPerishable

Before starting this, please download/clone the files provided from https://github.com/Seneca-244200/BTP-Milestone4 then add all the classes you implemented in previous milestones.

NonPerishable Class

Implement the NonPerishable class as a class derived from an Item class.
Essentially, NonPerishable is an Item class that is not abstract.

Private member variables

NonPerishable class adds only one private member variable of type Error, called **m_err**.

Constructor:

No constructors are crated for this class

Protected member functions

```
bool ok() const;
```

ok returns true if m**_err** is clear and has no error message.

```
void error(const char* message);
```

error sets the error message of **m_err** to the incoming argument "**message**".

```
virtual char signature()const;
```

Returns the record tag of the NonPershable class to be written in a file, that is **'N'**;
(it returns 'N')

Public member functions

NonPerishable implements all four pure virtual methods of the class PosIO (the signatures of the functions are identical to those of PosIO).

```
std::fstream& save(std::fstream& file)const
```

Using the operator<< of ostream first writes the **`signature()`** and a comma into the **file** argument, then without any formatting or spaces writes all the member variables of Item, comma separated, in following order:

```
sku, name, price, taxed, quantity
```

and ends them with a new line. Then it will return the file argument out.

Example:

```
N,1234,Candle,1.23,1,38
```

```
std::fstream& load(std::fstream& file)
```

Using operator>>, ignore and getline methods of istream, NonPerishable reads all the fields form the file and sets the member variables using the setter methods. When reading the fields, load assumes that the record does not have the "**N,**" at the beginning, so it starts the reading from the sku.

No error detection is done.
At the end the file argument is returned.

Hint: create temporary variables of type double, int and string and read the fields one by one, skipping the commas. After each read, set the member variables using setter methods.

```
ostream& write(ostream& ostr, bool linear)const
```
(V1.1) correction

If the object is not **ok(),** it simply prints the m_err using ostr and returns ostr. If the object is **ok()** (No Error) then depending on the value of linear, write(), prints the Item in different formats:

If Linear is true:

Prints the Item values separated by Bar "|" character in following format:

```
1234   |Candle              |   1.23| TN|  38|    52.82|
```

SKU: left justified in MAX_SKU_LEN characters
Name: left justified 20 characters wide, trimmed to 20 if longer than 20 characters.
Price: right justified, 2 digits after decimal point 7 chars wide
Taxed: space, "T" if it is taxed, space if it is not taxed and then the **signature() character**
Quantity: right justified 4 characters wide
Cost: total cost, considering quantity and tax; same format as price, 9 chars wide
One Bar and **NO NEW LINE**

If Linear is false:

Prints one member variable per line in following layout.
All formats are like linear layout, with no width restriction, except Name, which occupies one line; 80 chars.

```
Name:
Candle
Sku: 1234
Price: 1.23
Price after tax: 1.39
Quantity: 38
Total Cost: 52.82 <Newline>

OR if not taxed

Sku: 1234
Name: Candle
Price: 1.23
Price after tax: N/A
Quantity: 38
Total Cost: 46.74 <Newline>
```

Afterwards, write returns the ostr argument.

```
std::istream& read(std::istream& is)
```

Receives the values using istream (the istr argument) exactly as the following:

```
Non-Perishable Item Entry:
Sku: 1234<ENTER>
Name:
Candle<ENTER>
Price: 1.23<ENTER>
Taxed: y<Enter>
Quantity: 38<ENTER>
```

if **istr** is in a **fail** state, then the function exits doing nothing other than returning istr. If at any stage istr fails (cannot read), the error message will be set to the proper error message and the rest of the entry is skipped and nothing is set in the Item (also no error message is displayed). Here are the possible error messages:

fail at Price Entry:	**Invalid Price Entry**
fail at Taxed Entry:	**Invalid Taxed Entry, (y)es or (n)o**
fail at Quantity Entry:	**Invalid Quantity Entry**

When validating Taxed Entry, if the character entered is not a valid response to be consistent with an istream failure, manually set the istr to failure mode by calling this function:

```
istr.setstate(ios::failbit);
```

Since the rest of the member variables are text, istr cannot fail on them, therefore there are no error messages designated for them. Make sure at the end of the Entry you do not read the last new line or flush the keyboard.

At end, read will return the istr argument.

Part Two: Perishable Class

Perishable Class

Implement the **Perishable** class to be derived out of a **NonPerishable** class. Essentially, **Perishable** is a **NonPerishable** class with an expiry date.

Private member variables

Perishable class has one private member variables:

- A Date, called m_expiry (date only mode)

Protected member function

Perishable class has one protected member function that overrides the `signature() method of NonPerishable class;`

```
char signature()const;
```

Returns the record tag of the Pershable class to be written in a file, that is **'P'**;
(it returns 'P')

Constructor:

Create a no-argument default constructor and set the m_expiry attribute to date only mode.

Public member functions:

Pure virtual method implementations

Perishable implements all four pure virtual methods of the class PosIO. (the signatures of the functions are identical to those of PosIO).

```
fstream& Perishable::save(fstream& file)const
```

Calls the **NonPerishable** class's **save()** method, and then using operator<< writes a comma and the expiry date into the **file** argument. As a result the following will be written in the **file**:

```
sku, name, price, taxed, quantity, expiry date
```

Then it will return the **file** argument.
Example:

```
P,1234,4L Milk,3.99,0,2,2015/12/10
```

`fstream& Perishable::load(fstream& file)`
Calls the **NonPerishable** class's **load()** method by passing the **file** argument to it. Then ignores one character from the **file** and using operator>> reads the date from the **file** into the **m_expiry** attribute.

No error detection is done.
At the end the **file** argument is returned.

`ostream& Perishable::write(ostream& os, bool linear)const`
Calls **NonPerishable** class's **write**() passing **os** argument to it.

The if the class is **ok()** and **linear** argument is false, using **os** it will print:
`>Expiry date: <`
and **m_expiry** attribute and then goes to new line.

Afterwards, write returns the **ostr** argument.

`istream& Perishable::read(istream& istr) fardad was here`

First this function will add "Perishable " to the prompt of **NonPerishable::read()** function by printing:

`>Perishable <`

Next it will call **NonPerishable** class's **read**() by passing the **istr** to it.

Finally to get the expiry date it will prompt:

`>Expiry date (YYYY/MM/DD): <`

And using **istr** it will get the expiry date from user into **m_expiry** attribute.

If expiry entry fails then, depending of the error code stored in the Date class (**m_expriy**) , it will set the error by calling the **error()** function to one of the following:

CIN_FAILED: **Invalid Date Entry**

YEAR_ERROR: **Invalid Year in Date Entry**

MON_ERROR: **Invalid Month in Date Entry**

DAY_ERROR: **Invalid Day in Date Entry**

After setting the error message to be consistent with an **istream** failure, manually set the **istr** to failure mode by calling this function:

```
istr.setstate(ios::failbit);
```

Make sure at the end of the Entry you do not read the last new line or flush the keyboard.

At end, **read** will return the **istr** argument.

MILESTONE 4 SUBMISSION

After Compiling and testing your **Date.cpp**, **Date.h**, **PosIO.h**, **POS.h**, **Item.cpp**, **Item.h**, **Perishable.cpp**, **Pershable.h**, **NonPerishable.cpp** and **NonPerishable.h** with the provided tester programs:

01-NPErrHandling.cpp
02-NPDisplayTest.cpp
03-NPSaveLoad.cpp
04-PerErrHandling.cpp
05-PerDateErrHandling.cpp
06-PerDisplayTest.cpp
07-PerSaveLoad.cpp

And making sure that they all work properly, upload Date.cpp, Date.h, PosIO.h, POS.h, Item.cpp, Item.h, Perishable.cpp, Pershable.h, NonPerishable.cpp and NonPerishable.h and ms4tester.cpp to your matrix account. ms4tester.cpp does all the 7 tests in one file. 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 200_ms4 <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.