Winter 2017.

Point Of Sale Application

Final Project

(V1.1) See foot note

Your job for this project is to prepare an application that manages the list of items stored 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 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 to keep track of the errors occurring during data entry and user interaction.

PosIO A 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.

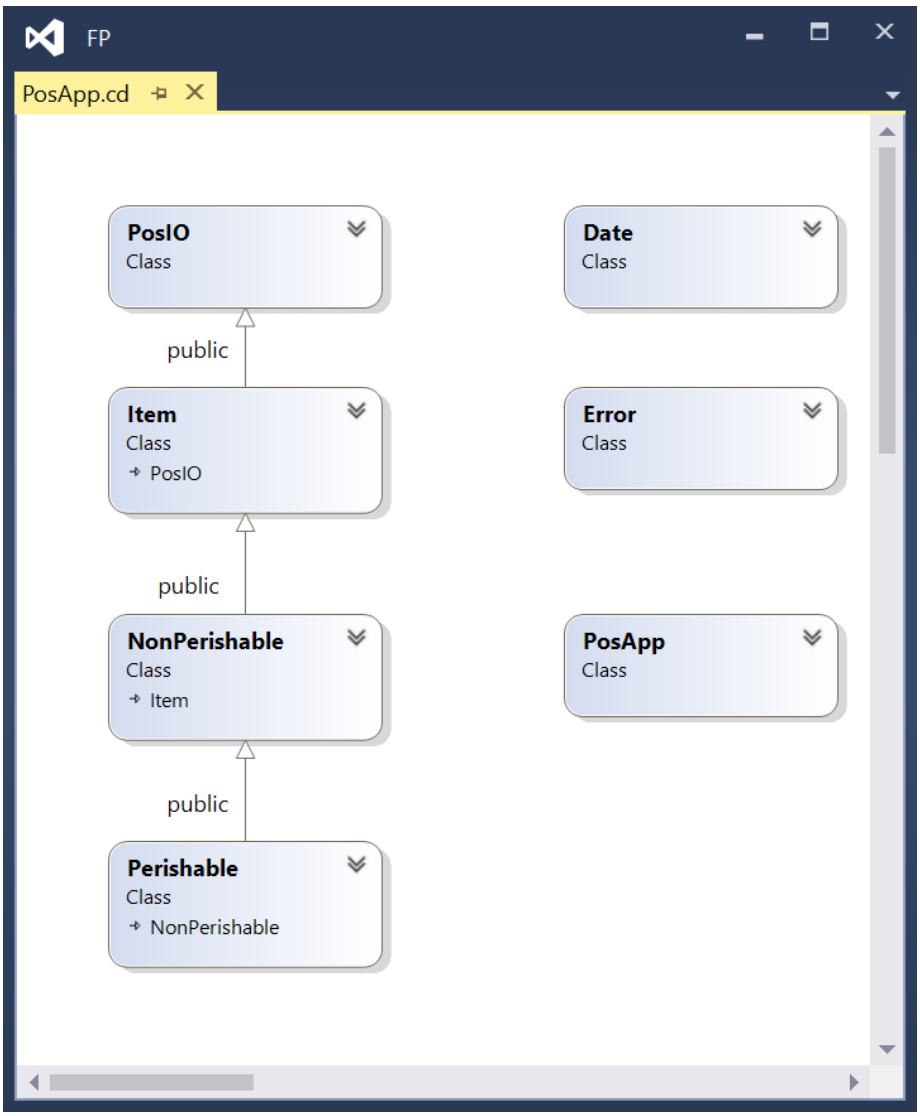
Using this class the list of items can be saved to a file and retrieved later, and individual item specifications can be displayed on screen (in detail or as a bill item) or read from keyboard.

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 sict namespace.

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

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

And adding the following command to the very end of your header files:

```
#endif
```

The "HeaderFileName" in the first two commands are 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/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 the 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 defined 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

244_ms1_tester.cpp Tests all the above.

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

THE ERROR CLASS

The Error class encapsulates an error message in a dynamic C-style string and also is used as 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 a proper error message.
Then using the **isClear()** method, it can be determined if an error has occurred or not and the object can be printed using **cout** to show 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 an Error object to be copied.

Public member functions and operator overloads (methods):

```
Error& operator=(const Error& em) = delete;
```
A deleted assignment operator overload to prevent an Error object to be assigned to another.

```
void operator=(const char* errorMessage);
```
Sets the m_message to the **errorMessage** argument and returns nothing (V1.1) by:

- De-allocating the memory pointed by **m_message**
- Allocating memory to the same length of **errorMessage + 1** and keeping the address in **m_message** data member.
- Copying **errorMessage** c-string 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 to the same (length of **value)** + 1 keeping the address in **m_message** data member.
- copying **value** c-string into **m_message.**

```
operator const char*() const;
```
returns the address kept 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 using **`cout`**.

If Error **isClear,** Nothing should be printed, otherwise the c-string pointed 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 everything works properly.

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

(V1.1): corrected "`void operator=(const char* errorMessage)`" description:
return *this was changed to return nothing;