Winter Semester 2018

Aid Management Application (AMA)

Version 3.2

When disaster hits a populated area, the most critical task is to provide immediately affected people with what they need as quickly and as efficiently as possible.

This project completes an application that manages the list of goods that need to be shipped to the disaster area. The client application tracks the quantity of items needed, tracks the quantity on hand, and stores the information in a file for future use.

The types of goods that need to be shipped are of two categories;

- Non-Perishable products, such as blankets and tents, which have no expiry date. We refer to products in this category as Product objects.
- Perishable products, such as food and medicine, that have an expiry date. We refer to products in this category as Perishable.

To complete this project you will need to create several classes that encapsulate your solution.

OVERVIEW OF THE CLASSES TO BE DEVELOPED

The classes used by the client application are:

Date

A class to be used to hold the expiry date of the perishable items.

ErrorMessage

A class to keep track of the errors occurring during data entry and user interaction.

Product

A class for non-perishable products.

Perishable

A class for perishable products that inherits the structure of the “Product” class and manages an expiry date.

iProduct

An interface to the Product hierarchy. This interface exposes the features of the hierarchy available to the client application. Any “iProduct” class can

- read itself from or write itself to the console
- save itself to or load itself from a text file
- compare itself to a unique C-string identifier
- determine if it is greater than another product in the collating sequence
- report the total cost of the items on hand
- describe itself
- update the quantity of the items on hand
- report its quantity of the items on hand
- report the quantity of items needed
- accept a number of items

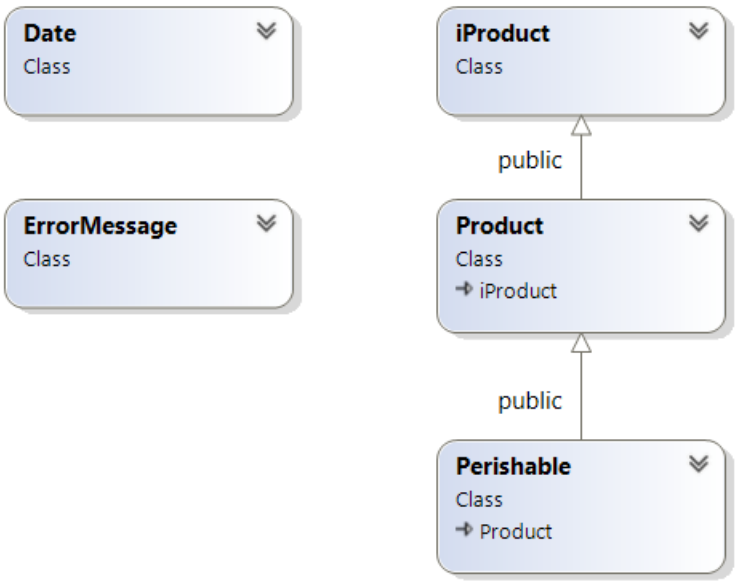
Using this class, the client application can

- save its set of iProducts to a file and retrieve that set later
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each product on hand

The client application manages the iProducts and provides the user with options to

- list the Products
- display details of a Product
- add a Product
- add items of a Product
- update the items of a Product
- delete a Product
- sort the set of Products

PROJECT CLASS DIAGRAM



PROJECT DEVELOPMENT PROCESS

The Development process of the project consists of 5 milestones and therefore 5 deliverables. Shortly before the due date of each deliverable a tester program and a script will be provided for testing and submitting the deliverable. The approximate schedule for deliverables is as follows

- Due Dates
  - The Date class — Due: Mar 16th, 11 days
  - The ErrorMessage class — Due: Mar 23th, 7 days
  - The Product class — Due: Apr 11th, 12 days
  - The iProduct interface — Due: Apr 12th, 1 day
  - The Perishable class — Due: Apr 14th, 2 days

GENERAL PROJECT SUBMISSION

In order to earn credit for the whole project, you must complete all milestones and assemble them for the final submission.

Note that at the end of the semester you **MUST submit a fully functional project to pass this subject**. If you fail to do so, you will fail the subject. If you do not complete the final milestone by the end of the semester and your total average, without your project's mark, is above 50%, your professor may record an "INC" (incomplete mark) for the subject. With the release of your transcript you will receive a new due date for completion of your project. The maximum project mark that you will receive for completing the project after the original due date will be "49%" of the project mark allocated on the subject outline.

FILE STRUCTURE OF THE PROJECT

Each class has its own header (.h) file and its own implementation (.cpp) file. The name of each file is the name of its class.

Example: Class **Date** is defined in two files: **Date.h** and **Date.cpp**

All of the code developed for this application should be enclosed in the **`AMA`** namespace.

MILESTONE 1: THE DATE CLASS

To kick-start this project, clone/download milestone 1 from the course repository and code the missing parts of the **Date** class.

The **Date** class encapsulates a date that is readable by an **std::istream** and printable by an **std::ostream** using the following format for both reading and writing: YYYY/MM/DD, where YYYY refers to a four-digit value for the year, MM refers to a two-digit value for the month and DD refers to a two-digit value for the day in the month.

Complete the implementation of the **Date** class using following specifications:

Pre-defined constants:

Pre-define the limits on the years to be considered acceptable:

```
const int min_year = 2000
const int max_year = 2030
```

Private members:

Data:

The year – a four digit integer between **min_year** and **max_year**

The month of the year – a value between 1 and 12 inclusive

The day of the month – a value between 1 and the number of days in the month (see the **mday(int,int)** member function described below) – Note that February has 29 days in a leap year.

The comparator value to be used for comparing the date stored in the current object with the date stored in another Date object. Your constructors set this value and your public operators use it to compare two dates. (If the value of date one is larger than the value of date two, then date one is more recent than date two; that is, date one is after date two).

The error state which the client can reference to determine if the object holds a valid date, and if not, which part of the date is in error. The possible error states are integer values *defined* as macros in the **Date** class header:

```
NO_ERROR   0  -- No error - the date is valid
CIN_FAILED 1  -- istream failed on information entry
YEAR_ERROR 2  -- Year value is invalid
MON_ERROR  3  -- Month value is invalid
DAY_ERROR  4  -- Day value is invalid
```

Member functions:

```
int mdays(int month, int year) const; (this query is already
      implemented and provided)
```

This query returns the number of days in **month** of **year**.

```
void errCode(int errorCode);
```

This function sets the error state variable to one of the values listed above.

Public members:

Constructors:

No argument (default) constructor: initializes the object to a safe empty state and sets the error state to **NO_ERROR**. Use 0000/00/00 as the date for a safe empty state and set the comparator value to 0.

Three argument constructor: accepts in its parameters integer values for the year, month and day. This constructor checks if each number is in range, in the order of year, month and day. If any of the numbers are not within range, this function sets the error state to the appropriate error code and stops further validation. (Use the **mday(int,int)** member function to obtain the number of days in the received month for the received year. The month value can be between 1 and 12 inclusive). If all of the data received is valid, this constructor stores the values received in the current object, calculates the comparator value, and sets the error state to **NO_ERROR**. If any of the data received is not valid, this constructor initializes the object to a safe empty state, sets the comparator value to 0 and sets the error state to **NO_ERROR**.

Use the following formula to set the comparator value for a valid date:

= year * 372 + month * 13 + day

Operators

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

These comparison operators return the result of comparing the current object as the left-hand side operand with another Date object as the right-hand side operand if the two objects are not empty. If one or both of them is empty, these operators return false.

For example **operator<** returns true if the Date stored in the current object is before the date stored in **rhs**; otherwise, this operator returns false.

Queries and modifier

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

This query returns the error state as an error code value.

`bool bad() const;`

This query returns true if the error state is not `NO_ERROR`.

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

This function reads the date from the console in the following format: YYYY?MM?DD (e.g. 2016/03/24 or 2016-03-24). This function does not prompt the user. If **`istr`** fails at any point, this function sets the error state to **`CIN_FAILED`** and does NOT clear **`istr`**. If **`istr`** has failed, a call to **`istr.fail()`** returns true. If your **`read()`** function reads the numbers successfully, Regardless of the result of this input process, this function returns a reference to the **`std::istream`** object.

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

This query writes the date to an **`std::ostream`** object in the following format: YYYY/MM/DD, and then returns a reference to the **`std::ostream`** object.

Helper functions:

`operator<<`

This operator works with an **`std::ostream`** object to print a date to the console.

`operator>>`

This operator works with an **`std::istream`** object to read a date from the console.

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

Include the prototypes for these two operators in the header file. Place their prototypes after the class definition.

Testing:

Test you code using the tester program supplied as the main module.

MILESTONE 1 SUBMISSION

If not on matrix already, upload **Date.h** and **Date.cpp** with the four testers 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_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.