Summer Semester 2018

Aid Management Application (AMA)

Version 3.4

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.

ErrorState

A class to keep track of the error state of client code. Errors may occur during data entry and user interaction.

Product

A class for managing non-perishable products.

Perishable

A class for managing perishable products. This class 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

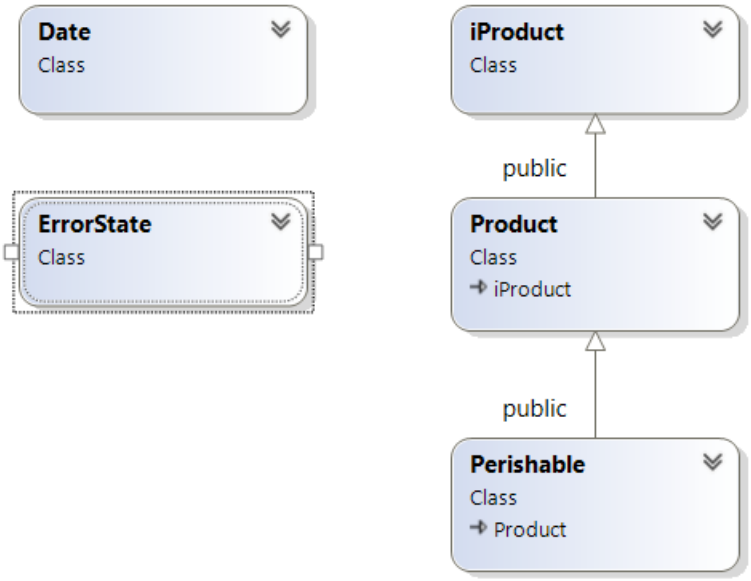
- save its set of iProducts to a file and retrieve that set later
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each product on hand

THE CLIENT APPLICATION

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: July 13th, 11 days
  - The ErrorState class — Due: July 20th, 7 days
  - The Product class — Due: August 1st, 12 days
  - The iProduct interface — Due: August 3rd, 1 day
  - The Perishable class — Due: August 8th, 3 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 3: THE PRODUCT CLASS

The `Product` class is a concrete class that encapsulates the general information for an AMA product.

Define and implement your `Product` class in the **AMA** namespace. Store your class definition in a file named **Product.h** and your implementation in a file named **Product.cpp**.

Your `Product` class uses an `ErrorState` object, but does not need a `Date` object.

Pre-defined constants:

Define the following as namespace constants:

- Maximum number of characters in a sku (stock keeping unit) – 7.
- Maximum number of characters in the units' descriptor for a product – 10.
- Maximum number of characters in the user's name descriptor for a product length – 75.
- The current tax rate – 13%.

Private members:

Data members:

- A character that indicates the type of the product – for use in the file record
- A character array that holds the product's sku (stock keeping unit) – the maximum number of characters excluding the null byte is defined by the namespace constant.
- A character array that describes the product's unit – the maximum number of characters excluding the null byte is defined by the namespace constant.
- A pointer that holds the address of a C-style string in dynamic memory containing the name of the product.
- An integer that holds the quantity of the product currently on hand; that is, the number of units currently on hand.
- An integer that holds the quantity of the product needed; that is, the number of units needed.
- A double that holds the price of a single unit of the product before any taxes.
- A bool that identifies the taxable status of the product; its value is true if the product is taxable.
- An `ErrorState` object that holds the error state of the `Product` object.

Protected member functions:

Your design includes the following protected member functions.

- `void name(const char*)`

  This function receives the address of a C-style null-terminated string that holds the name of the product. This function

  - stores the name in dynamically allocated memory
  - replaces any name previously stored
  - If the incoming parameter holds the **`nullptr`** address, this function removes the name of the product, if any, from memory.

- `const char* name() const`

  This query returns the address of the C-style string that holds the name of the product. If the product has no name, this query returns **`nullptr`**.

- `const char* sku() const`

  This query returns the address of the C-style string that holds the sku of the product.

- `const char* unit() const`

  This query returns the address of the C-style string that holds the unit of the product.

- `bool taxed() const`

  This query returns the taxable status of the product.

- `double price() const`

  This query returns the price of a single item of the product.

- `double cost() const`

  This query returns the price of a single item of the product plus any tax that applies to the product.

- `void message(const char*)`

  This function receives the address of a C-style null-terminated string holding an error message and stores that message in the `ErrorState` object.

- `bool isClear() const`

  This query returns true if the `ErrorState` object is clear; false otherwise.

Public member functions:

Your design includes the following public member functions:

- `Zero-One argument Constructor`

  This constructor optionally receives a character that identifies the product type. The default value is 'N'. This function

- stores the character receives in an instance variable
- sets the current object to a safe recognizable empty state.

- `Seven argument Constructor`

  This constructor receives in its seven parameters values in the following order:

  - the address of an unmodifiable C-style null terminated string holding the sku of the product
  - the address of an unmodifiable C-style null terminated string g holding the name of the product
  - the address of an unmodifiable C-style null terminated string holding the unit for the product
  - an integer holding the quantity of the product on hand – defaults to zero
  - a Boolean value indicating the product's taxable status – defaults to true
  - a double holding the product's price before taxes – defaults to zero
  - an integer holding the quantity of the product needed – defaults to zero

  This constructor allocates enough memory to hold the name of the product. Note that a protected function has been declared to perform this task.

- `Copy Constructor`

  This constructor receives an unmodifiable reference to a `Product` object and copies the object referenced to the current object.

- `Copy Assignment Operator`

  This operator receives an unmodifiable reference to a `Product` object and replaces the current object with a copy of the object referenced.

- `Destructor`

  This function deallocates any memory that has been dynamically allocated.

- `std::fstream& store(std::fstream& file, bool newLine=true) const`

  This query receives a reference to an `fstream` object and an optional bool and returns a reference to the `fstream` object. This function
  - inserts into the `fstream` object the character that identifies the product type as the first field in the record.

  - inserts into the `fstream` object the data for the current object in comma separated fields.
  - if the bool parameter is true, inserts a newline at the end of the record.

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

  This modifier receives a reference to an `fstream` object and returns a reference to that `fstream` object. This function
  - extracts the fields for a single record from the `fstream` object
  - creates a temporary object from the extracted field data
  - copy assigns the temporary object to the current object.

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

  This query receives a reference to an `ostream` object and a `bool` and returns a reference to the `ostream` object. This function inserts the data fields for the current object into the `ostream` object in the following order and separates them by a vertical bar character (**'|'**). If the `bool` parameter is true, the output is on a single line with the field widths as shown below in parentheses:

  - **sku – (maximum number of characters in a sku)**
  - **name – (20)**
  - **cost – (7)**
  - **quantity – (4)**
  - **unit – (10)**
  - **quantity needed – (4)**

  If the `bool` parameter is false, this function inserts the fields on separate lines with the following descriptors (a single space follows each colon):

  - **Sku:**
  - **Name (no spaces):**
  - **Price:**
  - either of:
    - **Price after tax:**
    - **N/A**
  - **Quantity on hand:**
  - **Quantity needed:**

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

  This modifier receives a reference to an `istream` object and returns a reference to the `istream` object. This function extracts the data fields for the current object in the following order, line by line. This function stops extracting data once it has encountered an error. The error messages are shown in brackets. A single space follows each colon:
  - **Sku:** <input value – C-style string>
  - **Name (no spaces):** <input value – C-style string>

- **Unit:** <input value – C-style string>
- **Taxed? (y/n):** <input character – y,Y,n, or N> [“Only (Y)es or (N)o are acceptable”]
- **Price:** <input value – double> [“Invalid Price Entry”]
- **Quantity on hand:** <input value – integer> [“Invalid Quantity Entry”]
- **Quantity needed:** <input value – integer> [“Invalid Quantity Needed Entry”]

If this function encounters an error for the Taxed input option, it sets the failure bit of the `istream` object (calling `istream::setstate(std::ios::failbit)`) and sets the error object to the error message noted in brackets.

If the `istream` object is not in a failed state and this function encounters an error on accepting Price input, it sets the error object to the error message noted in brackets. The member function that reports failure of an `istream` object is `istream::fail()`.

If the `istream` object is not in a failed state and this function encounters an error on the Quantity input, it sets the error object to the error message noted in brackets.

If the `istream` object is not in a failed state and this function encounters an error on the Quantity needed input, it sets the error object to the error message noted in brackets.

If the `istream` object has accepted all input successfully, this function stores the input values accepted in a temporary object and copy assigns it to the current object.

- `bool operator==(const char*) const`

  This query receives the address of an unmodifiable C-style null-terminated string and returns true if the string is identical to the sku of the current object; false otherwise.

- `double total_cost() const`

  This query that returns the total cost of all items of the product on hand, taxes included.

- `void quantity(int)`

  This modifier that receives an integer holding the number of units of the `Product` that are on hand. This function resets the number of units that are on hand to the number received.

- `bool isEmpty() const`

  This query returns true if the object is in a safe empty state; false otherwise.

- `int qtyNeeded() const`

  This query that returns the number of units of the product that are needed.

- `int quantity() const`

This query returns the number of units of the product that are on hand.

- `bool operator>(const char*) const`

  This query receives the address of a C-style null-terminated string holding a product sku and returns true if the sku of the current object is greater than the string stored at the received address (according to how the string comparison functions define 'greater than'); false otherwise.

- `bool operator>(const Product&) const`

  This query receives an unmodifiable reference to a `Product` object and returns true if the name of the current object is greater than the name of the referenced `Product` object (according to how the string comparison functions define 'greater than'); false otherwise.

- `int operator+=(int)`
  This modifier receives an integer identifying the number of units to be added to the `Product` and returns the updated number of units on hand. If the integer received is positive-valued, this function adds it to the quantity on hand. If the integer is negative-valued or zero, this function does nothing and returns the quantity on hand (without modification).

The following helper functions support your `Product` class:

- `std::ostream& operator<<(std::ostream&, const Product&)`

  This helper receives a reference to an `ostream` object and an unmodifiable reference to a `Product` object and returns a reference to the `ostream` object. Your implementation of this function will insert a `Product` record into the `ostream`.

- `std::istream& operator>>(std::istream&, Product&)`

  This helper receives a reference to an `istream` object and a reference to a `Product` object and returns a reference to the `istream` object. Your implementation of this function extracts the `Product` record from the `istream`.

- `double operator+=(double&, const Product&)`

  This helper receives a reference to a `double` and an unmodifiable reference to a `Product` object and returns a `double`. Your implementation of this function adds the total cost of the `Product` object to the `double` received and returns the updated `double`.

Once you have implemented all of the functions for this class, compile your `Product.cpp` and `ErrorState.cpp` files with the `MyProduct.cpp` and tester files provided. The provided files

should compile without error. The executable version should read and append text to the **`ms3.txt`** file.

MILESTONE 3 SUBMISSION

If not on matrix already, upload **`Product.h`**, **`Product.cpp`**, **`MyProduct.h`**, **`MyProduct.cpp`**, **`ErrorState.h`**, **`ErrorState.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.