Fall Semester 2017

Aid Management Application (AMA)

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

This project prepares an application that manages the list of goods needed to be shipped to the area. The application should track the quantity of items needed, track the quantity on hand, and store the information in a file for future use.

The types of goods needed to be shipped are divided into two categories;

- Non-Perishable products, such as blankets and tents, which have no expiry date. We refer to products in this category as NonPerishable 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 the solution to this problem and to provide a solution for this application.

OVERVIEW OF CLASSES TO BE DEVELOPED

The classes used by this 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

This interface (a class with “only” pure virtual functions) sets the requirements that derived classes must implement. These requirements have been set by the AidApp class. Any class derived from “Product” 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 application can

- save its set of Products 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

NonPerishable

A class for non-perishable products that implements the requirements of the "Product" interface (i.e. implements its pure virtual methods)

Perishable

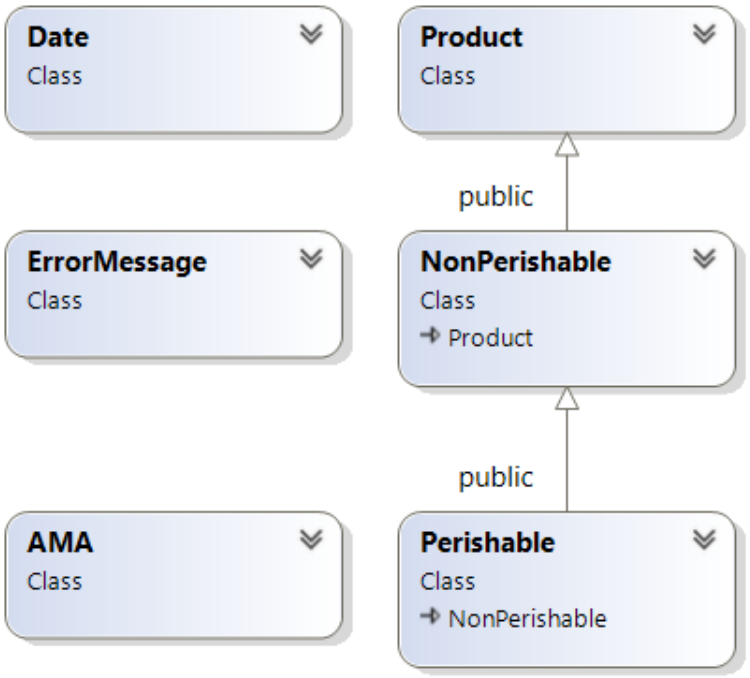
A class for perishable products that inherits from the "NonPerishable" class and provides an expiry date.

AidApp

The main application class manages the set of Products and provides the user with an interface 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 to you for testing and submitting the deliverable. The approximate schedule for deliverables is as follows

- Due Dates (Revised)
  - The Date class — Due: Nov 27th, 12 days
  - The ErrorMessage class — Due: Dec 4th, 7 days
  - The Product interface — Due: Dec 6th, 2 days
  - The NonPerishable class — Due: Dec 18th, 12 days
  - The Perishable class — Due: Dec 20th, 2 days
  - The AidApp class — Cancelled

GENERAL PROJECT SUBMISSION

In order to earn credit for the whole project, all milestones must be completed and assembled 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 the final milestone of your project is not completed by the end of the semester and your total average, without the 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 **`sict`** namespace.

MILESTONE 4: THE NON-PERISHABLE CLASS

Derive a class named `NonPerishable` from the `Product` interface. Your `NonPerishable` class is a concrete class that encapsulates the general information for a `Product`.

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

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

Pre-defined constants:

Define the following as constants in the namespace:

- Maximum sku length – 7.
- Maximum unit length – 10.
- Maximum name length – 75.
- Tax Rate – 13%.

Private members:

Data members:

A character indicates the type of 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 max_sku_length.

A pointer to a character array in dynamic memory that holds the name of the product.

A character array that holds the unit for the product (stock keeping unit) – the maximum number of characters excluding the null byte is max_unit_length.

An integer that holds the current quantity on hand of the product.

An integer that holds the quantity needed of the product.

A double that holds the price of the product before taxes.

A bool that is true if the product is taxable.

An `ErrorMessage` object that holds the error state of the current object.

Protected member functions:

Your design includes the following protected member functions.

- `void name(const char*)` - receives the address of a C-style null terminated string that holds the name of the product and stores the name in dynamically allocated

memory. This function replaces any name previously stored. If the address in the incoming parameter is `nullptr`, this function removes the name of the product, if any, from memory.

- `const char* name() const;` - 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`.
- `double cost() const` - returns the price of a single item of the product plus any tax that applies to the product.
- `void message(const char*)` - receives the address of a C-style string holding an error message and stores that message in the **`ErrorMessage`** object.
- `bool isClear() const` - returns true if the **`ErrorMessage`** 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, stores the character in an instance variable and sets the current object to a safe recognizable empty state.
- `Seven argument Constructor` - receives in its parameters seven values in the following order:
    - the address of an unmodifiable C-string holding the product sku
    - the address of an unmodifiable C-string holding the name of the product
    - the address of an unmodifiable C-string holding the unit for the product
    - an integer holding the quantity of the product on hand – defaults to zero
    - a Boolean value indicating whether or not the product is taxable – defaults to true
    - a double holding the price of the product before taxes, if any apply – defaults to zero
    - an integer holding the quantity needed – defaults to zero

  This constructor allocates enough memory to hold the name of the product. Note that a protected function is available for this.
- `Copy Constructor` - receives an unmodifiable reference to a **`NonPerishable`** object and copies the object referenced to the current object.

- `Copy Assignment Operator` - receives an unmodifiable reference to a **`NonPerishable`** object and replaces the current object with a copy of the object referenced.

- `Destructor` - deallocates any memory that has been dynamically allocated.

- `std::fstream& store(std::fstream& file, bool newLine=true) const` – a query that receives a reference to an `fstream` object and an optional bool and returns a reference to the `fstream` object. This function inserts a record with comma separated fields containing the data for the current object into the `fstream` object. The first field in the record is the character that identifies the product type. If the bool parameter is true, this function inserts a newline at the end of the record.

- `std::fstream& load(std::fstream& file)` – a modifier that receives a reference to an `fstream` object and returns a reference to that `fstream` object. This function extracts the fields for a record from the `fstream` object, creates a temporary object from the field data and copy assigns that object to the current object.

- `std::ostream& write(std::ostream& os, bool linear) const` – a query that 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 separated by a vertical bar character (‘|’). If the `bool` parameter is true, output is on a single line with the field widths as shown in parentheses:

  - `sku – (max_sku_length)`
  - `name – (20)`
  - `cost – (7)`
  - `quantity – (4)`
  - `unit – (10)`
  - `quantity needed – (4)`

  If the bool parameter is false, output is on separate lines with the following descriptors (a single space follows each colon:

  - `Sku:`
  - `Name:`
  - `Price:`
  - either of:
    - `Price after tax:`
    - `N/A`
  - `Quantity On Hand:`
  - `Quantity Needed:`

- `std::istream& read(std::istream& is)` – a modifier that 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. The error messages generated are shown in brackets:
    - Sku: <input value – C-style string>
    - Name: <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 this function encounters an error on the 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 this function encounters an error on the Qunatity input, it sets the error object to the error message noted in brackets. If 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 accepts all input successfully, it stores the input values in the current object.

- `bool operator==(const char*) const` – a query that receives the address of an unmodifiable C-style string and returns true if the string is identical to the sku of the current object; false otherwise.

- `double total_cost() const` – a query that returns the total cost of all items of the product on hand, taxes included.

- `void quantity(int)` – a 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.

- `bool isEmpty() const` - returns true if the object is in a safe empty state; false otherwise.

- `int qtyNeeded() const` – a query that returns the number of units of the current object that are needed.

- `int quantity() const` – a query returns the number of units of the current object that are on hand.

- `bool operator>(const char*) const` - receives the address of a C-style string holding a product sku and returns true if the string is greater than the sku of the current object as the string comparison functions define 'greater than'; false otherwise.

- `int operator+=(int)` – a modifier that 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).

- `bool operator>(const Product&) const` – a query that 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 `Product` object; false otherwise.

The following helper functions support your **`NonPerishable`** class:

- `std::ostream& operator<<(std::ostream&, const Product&)` – a helper that receives a reference to an `ostream` object and an unmodifiable reference to a `Product` object and returns a reference to the `ostream` object. Implementations of this function will insert a **`Product`** record into the `ostream`.

- `std::istream& operator>>(std::istream&, Product&)` – a helper that receives a reference to an `istream` object and an modifiable reference to a `Product` object and returns a reference to the `istream` object. Implementations of this function will extract the `Product` record from the `istream`.

- `double operator+=(double&, const Product&)` – a helper that receives a reference to a `double` and an unmodifiable reference to a `Product` object and returns a `double`. Implementations of this function will add the total cost of the `Product` object to the `double` received and return the updated `double`.

- `Product* CreateProduct()` – a helper that creates a **`NonPerishable`** object in dynamic memory and returns its address.

After you have implemented this concrete class completely, compile your `NonPerishable.cpp` file with the `ProductTester.cpp` provided. These two files should compile with no error, use your interface and read and append text to the `product.txt` file.

MILESTONE 4 SUBMISSION

If not on matrix already, upload **Product.h, NonPerishable.h, NonPerishable.cpp, ErrorMessage.h, ErrorMessage.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_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.