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

There are two categories for the types of goods that need to be shipped:

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

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 application are:

Date

A class that holds the expiry date of the perishable items.

Error

A class that tracks the error state of its client. Errors may occur during data entry and user interaction.

Good

A class that manages a non-perishable good object.

Perishable

A class that manages a perishable good object. This class inherits the structure of the “Good” class and manages a date.

iGood

An interface to the Good hierarchy. This interface exposes the features of the hierarchy available to the application. Any “iGood” class can

- read itself from the console or write itself to the console
- save itself to a text file or load itself from a text file
- compare itself to a unique C-style string identifier
- determine if it is greater than another good 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

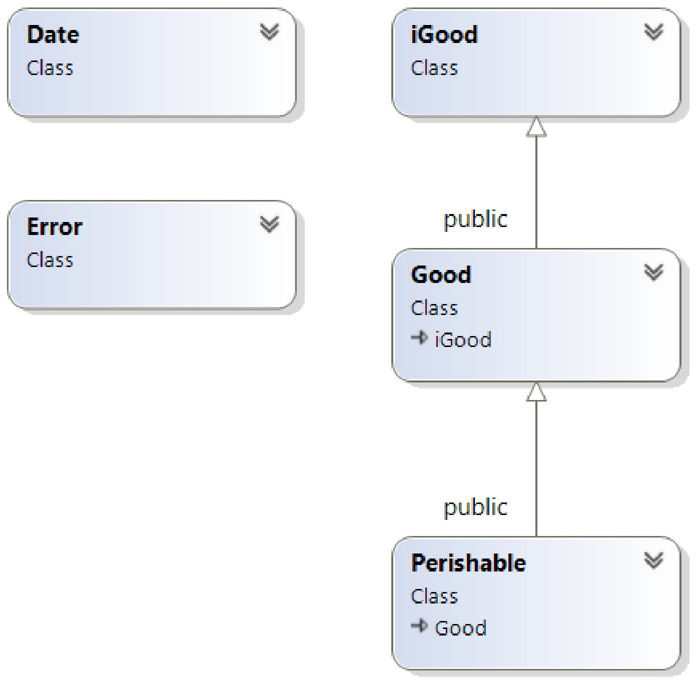
- save its set of iGoods to a file and retrieve that set at a later time
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each good on hand

THE CLIENT APPLICATION

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

- list the Goods
- display details of a Good
- add a Good
- add items of a Good
- update the items of a Good
- delete a Good
- sort the set of Goods

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 (at 11:59pm on each day)

The Date module	Due: November 2nd, 11 days
The Error module	Due: November 9th, 7 days
The Good module	Due: November 21st, 12 days
The iGood interface	Due: November 23rd, 2 days
The Perishable module	Due: November 28th, 3 days

SUBMISSION INSTRUCTIONS

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

Note that by the end of the semester you **MUST have submitted 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 belongs to its own module. Each module has its own header (.h) file and its own implementation (.cpp) file. The name of each file without the extension is the name of its class.

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

All the code developed for this application belongs to the **`aid`** namespace.

MILESTONE 3: THE GOOD CLASS

The `Good` class is a concrete class that encapsulates the general information for an aid Good.

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

Your `Good` class uses an `Error` 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 Good – 10.
- Maximum number of characters in the user's name descriptor for a Good length – 75.
- The current tax rate – 13%.

Private members:

Data members:

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

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

- `const char* name() const`

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

- `const char* sku() const`

  This query returns the address of the C-style null-terminated string that holds the sku of the Good.

- `const char* unit() const`

  This query returns the address of the C-style null-terminated string that holds the unit of the Good.

- `bool taxed() const`

  This query returns the taxable status of the Good.

- `double itemPrice() const`

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

- `double itemCost() const`

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

- `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 `Error` object to the current object.

- `bool isClear() const`

  This query returns true if the `Error` 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 Good type. The default value is 'N'. This function

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

- `Seven argument Constructor`

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

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

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

- `Copy Constructor`

  This constructor receives an unmodifiable reference to a **Good** object and copies the object referenced to the current object.

- `Copy Assignment Operator`

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

- `Destructor`

  This function deallocates any memory that has been dynamically allocated for the current object.

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

This query receives a reference to an `std::fstream` object and an optional bool and returns a reference to the `std::fstream` object. This function

- inserts into the `std::fstream` object the character that identifies the Good type as the first field in the record.
- inserts into the `std::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 `std::fstream` object and returns a reference to that `std::fstream` object. This function

- extracts the fields for a single record from the `std::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 `std::ostream` object and a `bool` and returns a reference to the `std::ostream` object. If the current object is in an error state, this function displays the error message. If the current object is empty, this function does not display anything further and returns. If the current object is not empty, this function inserts the data fields for the current object into the `std::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). If the name of the object is greater than 74 characters, this function only displays the first 74 characters:

- **`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 `std::istream` object and returns a reference to the `std::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 encounters 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 `std::istream` object (calling `std::istream::setstate(std::ios::failbit)`) and sets the error object to the error message noted in brackets.

  If the `std::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 function that reports failure of an `std::istream` object is `std::istream::fail()`.

  If the `std::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 `std::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 `std::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 Good on hand, taxes included.

- `void quantity(int)`

  This modifier that receives an integer holding the number of units of the `Good` that are on hand. If this number is positive-valued this function resets the number of units that are on hand to the number received; otherwise, this function does nothing.

- `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 Good that are needed.

- `int quantity() const`

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

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

  This query receives the address of a C-style null-terminated string holding a Good 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 Good&) const`

  This query receives an unmodifiable reference to a **`Good`** object and returns true if the name of the current object is greater than the name of the referenced **`Good`** 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 **`Good`** 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).

Helper functions:

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

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

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

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

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

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

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

Once you have implemented all of the functions for this class, compile your `Good.cpp` and `Error.cpp` files with the tester files provided using Visual Studio. The provided files should compile without error. The executable version should read and append text to the **`ms3.txt`** file. Test your executable to make sure that it runs successfully.

MILESTONE 3 SUBMISSION

Upload `Good.h`, `Good.cpp`, `Error.h`, `Error.cpp` and the tester files to your matrix account. Compile and rerun your code and make sure everything works properly.

Then run the following command from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 244_ms3 <ENTER>
```

and follow the instructions.

Please note that a successful submission does not guarantee full credit for this milestone.

If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.