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

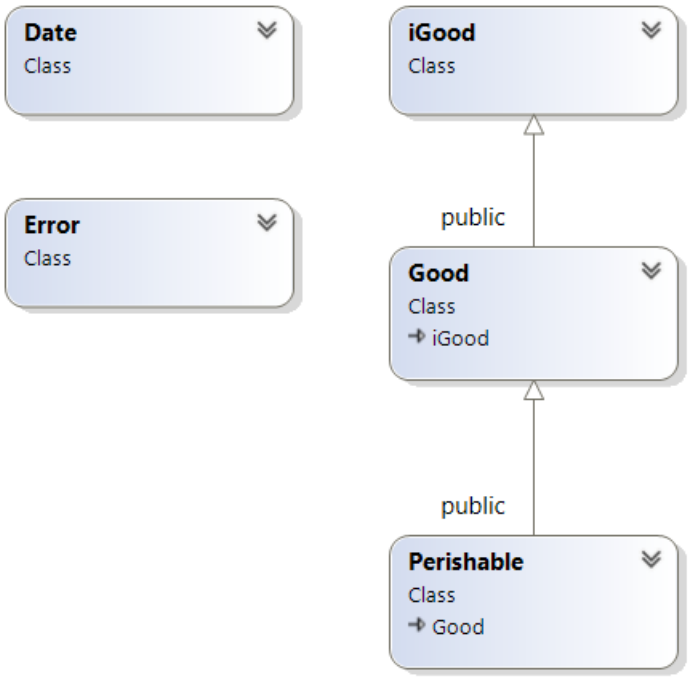
- save its set of iGoods to a file and retrieve that set at a later time
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each good on hand

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 5: THE GOOD HIERARCHY

This milestone creates the `Good` hierarchy and is the last milestone in the project.

The first step to creating the `Good` hierarchy is to derive your `Good` class from your `iGood` interface. Your `Good` class is a concrete class that encapsulates information for perishable goods.

Upgrade your `Good` module to share select functions with the other classes in the hierarchy:

- In the class definition:
  - derive `Good` from interface `iGood`
  - change the prototypes for the following functions to receive a reference to any object in the hierarchy:
    - `bool operator>(const iGood&) const`
    - `double operator+=(double&, const iGood&)`
    - `std::ostream& operator<<(std::ostream&, const iGood&)`
    - `std::istream& operator>>(std::istream&, iGood&)`
- In the class implementation:
  - change the signatures of these functions to receive a reference to any object in the hierarchy:
    - `bool operator>(const iGood&) const`
    - `double operator+=(double&, const iGood&)`
    - `std::ostream& operator<<(std::ostream&, const iGood&)`
    - `std::istream& operator>>(std::istream&, iGood&)`

THE PERISHABLE CLASS

The second step to completing the `Good` hierarchy is to derive the `Perishable` class from your `Good` class. Your `Perishable` class is a separate concrete class that encapsulates information for perishable goods.

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

Your `Perishable` class uses a `Date` object, but does not need its own `Error` object (the `Good` base class handles all error processing).

Private data member:

A `Date` object holds the expiry date for the perishable good.

Public member functions:

Your design includes the following public member functions:

- `No argument Constructor`

This constructor passes the file record tag for a perishable good ('P') to the base class constructor and sets the current object to a safe empty state.

- **`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 modified `fstream` object. This function stores a single file record for the current object. This function
  - calls its base class version passing as arguments a reference to the `fstream` object and a false flag. The base class inserts the common data into the `fstream` object
  - inserts a comma into the file record
  - appends the expiry date to the file record.
  - If `newLine` is true, this function inserts a newline character ('\n') before exiting. In this case, the file record created will look something like:

    `P,1234,water,1.5,0,1,liter,5,2018/03/28`<NEWLINE>

  - If `newLine` is false, this function does not insert a newline character ('\n') before exiting. In this case, the file record created will look something like:

    `P,1234,water,1.5,0,1,liter,5,2018/03/28`

  Note that the first field in the file record is 'P'. This character was passed to the base class at construction time and is inserted by the base class version of this function.

- **`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 data fields for a single file record from the `fstream` object. This function
  - calls its base class version passing as an argument a reference to the `fstream` object
  - loads the expiry date from the file record using the **`read()`** function of the `Date` object
  - extracts a single character from the `fstream` object.

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

  This query receives a reference to an `ostream` object and a `bool` flag and returns a reference to the modified `ostream` object. The flag identifies the output format. This function
  - calls its base class version passing as arguments a reference to the `ostream` object and the `bool` flag.
  - if the current object is in an error or safe empty state, does nothing further.

- if the current object is not in an error or safe empty state, inserts the expiry date into the `ostream` object.
- if `linear` is true, adds the expiry date on the same line for an outcome that looks something like this:

```
1234   |water                |   1.50|   1|liter      |   5|2018/12/30
```

- If `linear` is false, this function adds a new line character followed by the string "Expiry date: " and the expiry date for an outcome something like this:

```
Sku: 1234
Name (no spaces): water
Price: 1.50
Price after tax: N/A
Quantity on hand: 1 liter
Quantity needed: 5
Expiry date: 2018/12/30
```

This function does not insert a newline after the expiry date in the case of linear output (`linear` is true) or the case of line-by-line output (`linear` is false).

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

  This modifier receives a reference to an `istream` object and returns a reference to the modified `istream` object. This function populates the current object with the data extracted from the `istream` object.

  This function calls its base class version passing as its argument a reference to the `istream` object. If the base class object extracts data successfully, this function prompts for the expiry date and stores it in a temporary `Date` object. The prompt looks like with a single space after the colon:

  `Expiry date (YYYY/MM/DD):`

  If the temporary Date object is in an error state, this function stores the appropriate error message in the base class' error object and sets the state of the `istream` object to a failed state. The member function that sets it to a failed state is (`istream::setstate(std::ios::failbit)`). The messages that correspond to the error codes of a `Date` object are:

CIN_FAILED:	**Invalid Date Entry**
YEAR_ERROR:	**Invalid Year in Date Entry**
MON_ERROR:	**Invalid Month in Date Entry**
DAY_ERROR:	**Invalid Day in Date Entry**

  If the `istream` object is not in an error state, this function copy assigns the temporary `Date` object to the instance `Date` object. The member function that reports failure of an `istream` object is `istream::fail()`.

- **`const Date& expiry() const`**

This query returns the expiry date for the perishable good.

After you have implemented the **Perishable** class completely, compile your implementation files with the tester files provided. Your implementation files should compile with no error, use your interface and read and append text to the `good.txt` and `goodShortFile.txt` files.

The output should look something like this:

```
--Good test:
----Taxed validation test:
Enter the following:
 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): a

 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): a

Passed!
 Message should be:  Only (Y)es or (N)o are acceptable
 Your Error message: Only (Y)es or (N)o are acceptable
Press enter to continue ...

----Price validation test:
Enter the following:
 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): y
 Price: abc

 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): y
 Price: abc

Passed!
 Message should be:  Invalid Price Entry
 Your Error message: Invalid Price Entry
Press enter to continue ...

----Quantity validation test:
Enter the following:
 Sku: abc
 Name (no spaces): abc
 Unit: abc
```

```
 Taxed? (y/n): y
 Price: 10
 Quantity on hand: abc

 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): y
 Price: 10
 Quantity on hand: abc

Passed!
 Message should be:  Invalid Quantity Entry
 Your Error message: Invalid Quantity Entry
Press enter to continue ...

----Quantity needed validation test:
Enter the following:
 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): y
 Price: 10
 Quantity on hand: 10
 Quantity needed: abc

 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): y
 Price: 10
 Quantity on hand: 10
 Quantity needed: abc

Passed!
 Message should be:  Invalid Quantity Needed Entry
 Your Error message: Invalid Quantity Needed Entry
Press enter to continue ...

----Display test, the output of the Program and yours must match:
Enter the following:
 Sku: 1234
 Name (no spaces): box
 Unit: kg
 Taxed? (y/n): y
 Price: 123.45
 Quantity on hand: 1
 Quantity needed: 5

 Sku: 1234
 Name (no spaces): box
```

```
 Unit: kg
 Taxed? (y/n): y
 Price: 123.45
 Quantity on hand: 1
 Quantity needed: 5

-Compare the output of the Program and your output:
--Linear------------
 Program: 1234   |box                | 139.50|   1|kg        |   5|
   Yours: 1234   |box                | 139.50|   1|kg        |   5|
--Form Display------
--Program:
 Sku: 1234
 Name (no spaces): box
 Price : 123.45
 Price after tax : 139.50
 Quantity on hand : 1 kg
 Quantity needed : 5
--Yours:
 Sku: 1234
 Name (no spaces): box
 Price: 123.45
 Price after tax: 139.50
 Quantity on Hand: 1 kg
 Quantity needed: 5
Press enter to continue ...

----Storage and loading test, the output of the Program and yours must match:
--Store Good, program:
N,1234,box,kg,1,123.45,1,5
N,1234,box,kg,1,123.45,1,5
--Store Good, yours:
N,1234,box,kg,1,123.45,1,5
N,1234,box,kg,1,123.45,1,5
--Load Good:
 Program: 1234   |box                | 139.50|   1|kg        |   5|
   Yours: 1234   |box                | 139.50|   1|kg        |   5|

--Perishable Good test:
----Expiry date Validation test:
Enter the following:
 Sku: abc
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): n
 Price: 10
 Quantity on hand: 10
 Quantity needed: 10
 Expiry date (YYYY/MM/DD): 10/1/1

 Sku: abc
```

```
 Name (no spaces): abc
 Unit: abc
 Taxed? (y/n): n
 Price: 10
 Quantity on hand: 10
 Quantity needed: 10
 Expiry date (YYYY/MM/DD): 10/1/1

Passed!
 Message should be:  Invalid Year in Date Entry
 Your Error message: Invalid Year in Date Entry
Press enter to continue ...

----Display test, the output of the Program and yours must match:
Enter the following:
 Sku: 1234
 Name (no spaces): water
 Unit: liter
 Taxed? (y/n): n
 Price: 1.5
 Quantity on hand: 1
 Quantity needed: 5
 Expiry date (YYYY/MM/DD): 2018/12/30

 Sku: 1234
 Name (no spaces): water
 Unit: liter
 Taxed? (y/n): n
 Price: 1.5
 Quantity on hand: 1
 Quantity needed: 5
 Expiry date (YYYY/MM/DD): 2018/12/30

-Compare the output of the Program and your output:
--Linear------------
 Program: 1234   |water               |   1.50|   1|liter     |   5|2018/12/30
   Yours: 1234   |water               |   1.50|   1|liter     |   5|2018/12/30
--Form Display------
--Program:
 Sku: 1234
 Name (no spaces): water
 Price : 1.50
 Price after tax : N/A
 Quantity on hand : 1 liter
 Quantity needed : 5
 Expiry date : 2018/12/30
--Yours:
 Sku: 1234
 Name (no spaces): water
 Price: 1.50
 Price after tax:  N/A
```

```
 Quantity on Hand: 1 liter
 Quantity needed: 5
 Expiry date: 2018/12/30

Press enter to continue ...

----Storage and loading test, the output of the Program and yours must match:
--Store Perishable, program:
P,1234,water,liter,0,1.5,1,5,2018/12/30
P,1234,water,liter,0,1.5,1,5,2018/12/30
--Store Perishable, yours:
P,1234,water,liter,0,1.5,1,5,2018/12/30
P,1234,water,liter,0,1.5,1,5,2018/12/30
--Load Perishable:
 Program: 1234   |water               |   1.50|   1|liter     |   5|2018/12/30
   Yours: 1234   |water               |   1.50|   1|liter     |   5|2018/12/30
Press any key to continue . . .
```

MILESTONE 5 SUBMISSION

If not on matrix already, upload **Date.h**, **Date.cpp**, **Error.h**, **Error.cpp**, **iGood.h**, **Good.h**, **Good.cpp**, **Perishable.h**, **Perishable.cpp**, **ms5_Allocator.cpp** and **ms5_tester.cpp** the tester to your matrix account. Compile and run 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_ms5 <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.