SUMMER SEMESTER 2019

Aid Management Application (AMA)

Version 3.7.0

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

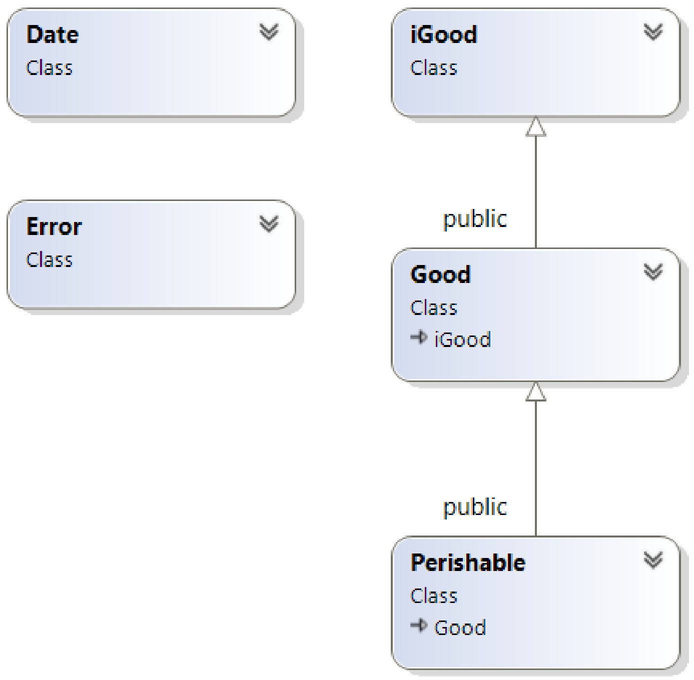
- save its set of iGoods to a file and retrieve that set at a later time
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each good on hand

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

Each milestone is due at 23:59:59 of the specified day:

• The `Date` module	Due: July 11th (11 days)
• The `Error` module	Due: July 18th (7 days)
• The `Good` module	Due: July 30th (12 days)
• The `iGood` interface	Due: August 1st (2 days)
• The `Perishable` module	Due: August 4th (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 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 1: the Date Module

To 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` object and printable by an `std::ostream` object using the following format: `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:

- `min_year` with the value 2018. This represents the minimum acceptable year for a valid date.
- `max_year` with the value 2038. This represents the maximum acceptable year for a valid date.
- `min_date` with the value 751098. This represents the date 2018/12/30 (see formula below).

Class Members:

Add to the class attributes to store the following information:

- Year: a four digit integer between the limits defined above.
- Month: an integer with the values between 1 and 12 inclusive.
- Day of the Month: an integer with a value between and the maximum number of days in the month. Use the function `int mdays(int year, int month)` to find out how many days are in a given month for a given year. Note that February can have 28 or 29 days depending on the year.
- A comparator value for comparing the date stored in the current object with the date stored in another `Date` object. Your constructors set this comparator value and your public relational operators use it to compare 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 of this class 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
DAY_ERROR  2  -- Day value is invalid
MON_ERROR  3  -- Month value is invalid
YEAR_ERROR 4  -- Year value is invalid
PAST_ERROR 5  -- Date value is invalid
```

Add to the class the following private functions:

- `void errCode(int errorCode)`: This function sets the error state variable to one of the values listed above.
- `int mdays(int year, int month) const`: This function returns the number of days in a given month for a given year. Use the implementation below:

```
int Date::mdays(int year, int mon) const
{
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, -1 };
  int month = mon >= 1 && mon <= 12 ? mon : 13;
  month--;
  return days[month] + int((month == 1)*((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0));
}
```

Add to the class the following public functions:

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 in that order. This constructor checks if each number is in range, in the order of year, month and day and value. 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 and sets the comparator value to 0.

Use the following formula to set the comparator value for a valid date: `year * 372 + month * 31 + day`

For the date received to be valid its comparator value must be greater than or equal to `min_date` and all other conditions must be met.

RELATIONAL 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 the objects 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 MODIFIERS

- `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 (if `istr` fails, the function `istr.fail()` returns `true`), this function sets the error state to **`CIN_FAILED`** and does NOT clear `istr`.

  If your `read()` function reads the numbers successfully, and the read values are valid, it stores them into the current object's instance variables. Otherwise, your function does not change the current object.

  Regardless of the result of the input process, your function returns a reference to the `std::istream` object.

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

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

Free Helper Functions:

- `operator<<`: Prints the date to the first parameter (use `Date::write()`).
- `operator>>`: Reads the date from the first parameter (use `Date::read()`).

You can add as many private members as your design requires. Do not add extra public members.

Submission

If not on matrix already, upload `Date.h`, `Date.cpp` and `ms1.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_ms1<ENTER>
```

and follow the instructions.

Important: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

MILESTONE 2: THE ERROR MODULE

The `Error` class manages the error state of client code and encapsulates the last error message.

Any client can define and store an `Error` object. If a client encounters an error, the client can set its `Error` object to an appropriate message. The client specifies the length of the message.

The `Error` object reports whether or not any error has occurred. The `isClear()` query on the object reports if an error has occurred. If an error has occurred, the object can display the message associated with that error. The object can be send its message to an `std::ostream` object.

This milestone does not use the `Date` class.

The class `Error` manages a resource and doesn't support copying operations.

Complete your implementation of the `Error` class based on the following information:

DATA MEMBER:

- A pointer that holds the address of the message, if any, stored in the current object.

PUBLIC MEMBER FUNCTIONS:

- `explicit Error(const char* errorMessage = nullptr)`: This constructor receives the address of a C-style null terminated string that holds an error message.

  If the address is `nullptr`, this function puts the object into a safe empty state.

  If the address points to an empty message, this function puts the object into a safe empty state.

  If the address points to a non-empty message, this function allocates memory for that message and copies the message into the allocated memory.

- A destructor that de-allocates any memory that has been dynamically allocated by the current object.
- `void clear()`: This function clears any message stored by the current object and initializes the object to a safe empty state.
- `bool isClear() const`: This query reports returns true if the current object is in a safe empty state.
- `void message(const char* str)`: This function stores a copy of the C-style string pointed to by `str`.

  De-allocates any memory allocated for a previously stored message.

If `str` points to a non-empty message, this function allocates the dynamic memory needed to store a copy of `str` (remember to include 1 extra byte for the null terminator) and copies the message into that memory.

If `str` points to an empty message, this function puts the current object in a safe empty state.

- `const char* message() const;`

  If the current object is not in a safe empty state, this query returns the address of the message stored in the object.

  If the current object is in a safe empty state, this query returns `nullptr`.

FREE HELPER OPERATOR:

- `operator<<`: This operator sends an `Error` message, if one exists, to an `std::ostream` object and returns a reference to the `std::ostream` object.

  If no message exists, this operator does not send anything to the `std::ostream` object and returns a reference to the `std::ostream` object.

SUBMISSION

If not on matrix already, upload `Error.h`, `Error.cpp` and `ms2.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_ms2<ENTER>
```

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

Milestone 3: The Good Module

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:

- `max_sku_length`: Maximum number of characters in a SKU (stock keeping unit) – 7.
- `max_unit_length`: Maximum number of characters in the units' descriptor for a `Good` – 10.
- `max_name_length`: Maximum number of characters in the user's name descriptor for a `Good` length – 75.
- `tax_rate`: The current tax rate – 13%.

Private Data:

- 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:

- `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` without tax.
- `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
  - stores the character received in an instance variable;
  - sets the current object to a safe recognizable empty state.

- Seven Arguments Constructor: This constructor receives the following values in its seven parameters in the following order:

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

- `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.

SUBMISSION

If not on matrix already, upload `Error.h`, `Error.cpp`, `Good.h`, `Good.cpp` and `ms3.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_ms3<ENTER>
```

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

MILESTONE 4: THE IGOOD INTERFACE

The `iGood` class is an interface that exposes the `Good` hierarchy to client applications. This class is abstract and cannot be instantiated. You will add and develop concrete classes of the hierarchy in the following milestone.

You do not need the `Date`, `Error` and `Good` classes for this milestone.

Save your definition of the `iGood` interface in a header file named `iGood.h`.

The definition of your `iGood` interface includes the following pure virtual member functions:

- `std::fstream& store(std::fstream& file, bool newLine = true) const`: This query will receive a reference to an `std::fstream` object and an optional bool and return a reference to the `std::fstream` object. The bool argument will specify whether or not a newline should be inserted after each `iGood` record. Implementations of this function will insert the `Good` records into the `std::fstream` object.
- `std::fstream& load(std::fstream& file):` This modifier will receive a reference to an `std::fstream` object and return a reference to that `std::fstream` object. Implementations of this function will extract `iGood` records from the `std::fstream` object.
- `std::ostream& write(std::ostream& os, bool linear) const`: This query will receive a reference to an `std::ostream` object and a `bool` and return a reference to the `std::ostream` object. The `bool` argument will specify whether or not the records should be listed on a single line or on separate lines. Implementations of this function will insert the `iGood` record for the current object into the `std::ostream` object.
- `std::istream& read(std::istream& is)`: This modifier will receive a reference to an `std::istream` object and returns a reference to the `std::istream` object. Implementations of this function will extract the `iGood` record for the current object from the `std::istream` object.
- `bool operator==(const char*) const`: This query will receive the address of an unmodifiable C-style null-terminated string and return true if the string is identical to the stock keeping unit of an `iGood` record; false otherwise.
- `double total_cost() const`: This query will return the cost of a single unit of an `iGood` with taxes included.
- `const char* name() const`: This query will return the address of a C-style null-terminated string containing the name of an `iGood`.

- `void quantity(int)`: This modifier will receive an integer holding the number of units of an `iGood` that are currently available. This function will set the number of units available.
- `int qtyNeeded() const`: This query will return the number of units of an `iGood` that are needed.
- `int quantity() const`: This query will return the number of units of an `iGood` that are currently available.
- `int operator+=(int)`: This modifier will receive an integer identifying the number of units to be added to the `iGood` and return the updated number of units currently available.
- `bool operator>(const iGood&) const`: This query will receive an unmodifiable reference to an `iGood` object and return true if the current object is greater than the referenced `iGood` object; false otherwise.

The following helper functions support your interface:

- `std::ostream& operator<<(std::ostream&, const iGood&)`: This helper function will receive a reference to an `std::ostream` object and an unmodifiable reference to an `iGood` object and return a reference to the `std::ostream` object. Implementations of this function will insert the `iGood` record for the referenced object into the `ostream` object.
- `std::istream& operator>>(std::istream&, iGood&)`: This helper function will receive a reference to an `std::istream` object and a reference to an `iGood` object and return a reference to the `std::istream` object. Implementations of this function will extract the `iGood` record for the referenced object from the `std::istream` object.
- `double operator+=(double&, const iGood&)`: This helper function will receive a reference to a `double` and an unmodifiable reference to an `iGood` object and return a `double`. Implementations of this function will add the total cost of the `iGood` object to the `double` received and return the updated value of the `double`.
- `iGood* CreateProduct(char tag)`: This helper function will return the address of a `iGood` object or `nullptr`.

Once you have defined this interface, compile the tester files provided on Visual Studio. These files should compile without error. The executable code should use your interface to append data fields to and read data fields from the `ms4.txt` file.

SUBMISSION

If not on matrix already, upload `iGood.h`, `ms4_Allocator.cpp`, `ms4_MyGood.h`, `ms4_MyGood.cpp` and `ms4.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_ms4<ENTER>
```

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

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 prototypes 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:

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

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<ENDL>
    ```

  - 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**
PAST_ERROR:	**Invalid Expiry Date 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()`.

NOTE: if you encounter a mismatch between your output and that of the tester program, confirm that the `Good::read()` function leaves the `istream` input buffer empty on a reading of the input that generates no errors.

- `const Date& expiry() const`: This query returns the expiry date for the perishable product.

FREE HELPER FUNCTION

In `Perishable.cpp`, implement the following function as a free helper:

- `iGood* CreateProduct(char tag)`: This helper function will dynamically create and return the address of an `iGood` object:
    - if the parameter is `'N'` or `'n'`, this function creates a `Good` object
    - if the parameter is `'P'` or `'p'`, this function creates a `Perishable` object
    - returns `nullptr` otherwise.

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` and `ms5.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca `userid` to replace `profname.proflastname`):

```
~profname.proflastname/submit 200_ms5<ENTER>
```

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.