Summer 2017.

Aid Management Application (AMA)

When disaster hits an area, the most important thing is to be able provide the people affected with what they need as quickly and as efficiently possible.

Your job for this project is to prepare an application that manages the list of goods needed to be shipped to the area. The application should be able to keep track of the quantity of items needed, the quantity on hand, and store them in a file for future use.

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

- Non-Perishable products, such as blankets and tents, that have no expiry date, we refer to these type of products as AMA_product.
- Perishable products, such as food and medicine, that have an expiry date, we refer to these products as AMA_Perishable.

To accomplish this task you need to create several classes to encapsulate the problem and provide a solution for this application.

CLASSES TO BE DEVELOPED

The classes needed for 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.

Streamable

This interface (a class with “only” pure virtual functions) enforces the classes that inherit from it to be *streamable*. Any class derived from “Streamable” can read from or write to the console, or can be saved to or loaded from a text file.

Using this class the list of items can be saved into a file and retrieved later, and individual item specifications can be displayed on screen or read from keyboard.

Product

A class inherited from Streamable, containing general information about an item, like the name, Stock Keeping Unit (SKU), price etc.

AMA_Product

A class for non-perishable items that is inherited from the “Product” class and implements the requirements of the “Streamable” class (i.e. implements the pure virtual methods of the Streamable class)

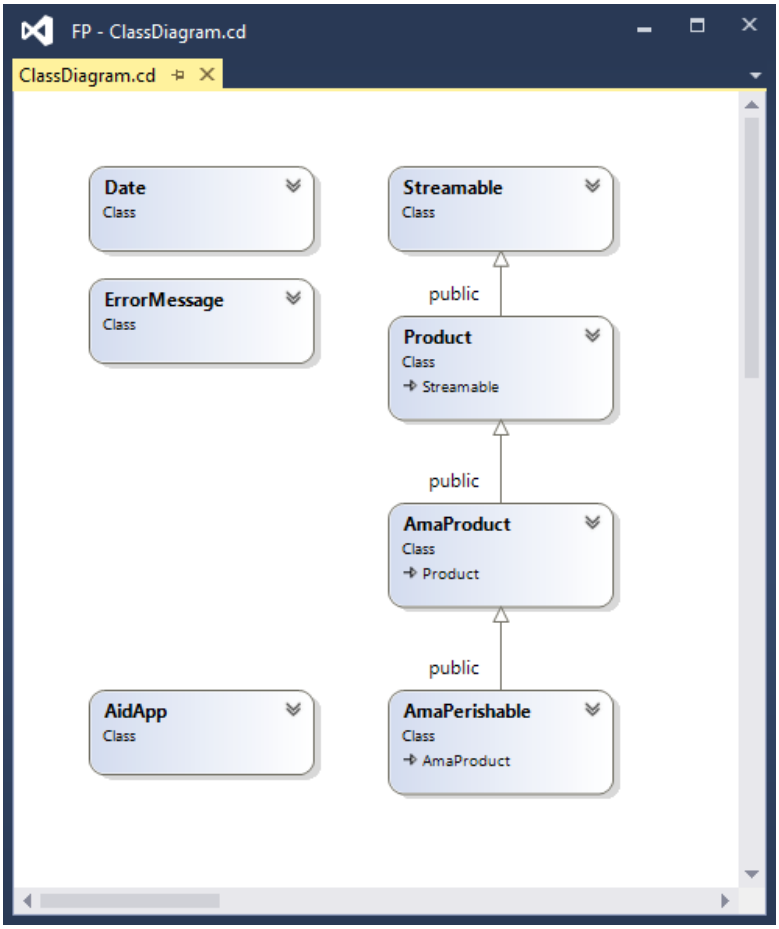
AMA_Perishable

A class inherited from the “AMA_Product” that provides expiry date for Perishable items.

AidApp

The main application class that is essentially the manager class for the NFI and Perishable items. This class provides the user with a user-interface to list, add and update the items saved in a data file.

PROJECT CLASS DIAGRAM



PROJECT DEVELOPMENT PROCESS

The Development process of the project is divided into 6 milestones and therefore six deliverables. Shortly before the due date of each deliverable a tester program and a script will

be provided to you to test and submit each of the deliverables. The approximate schedule for deliverables is as follows

- Due: Kickoff (KO) + 33 days
  - The Date class. Due: Jul 8th, 5 days
  - The ErrorMessage class Due: Jul 11th, 3 days
  - The Streamable class Due: Jul 12th, 1 day
  - The Product class Due: Jul 18th, 6 days
  - The AMA product classes Due: Jul 27th, 9 days
  - The AidApp class. Due: Aug 5th, 9 days

GENERAL PROJECT SUBMISSION

This is the first of several milestones and in order to complete 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 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 give you an "INC" (incomplete mark) for the subject. With the release of your transcript you will receive a new due date to complete your project for a maximum mark of "49%" and your total mark for the subject will be calculated accordingly.

FILE STRUCTURE OF THE PROJECT

Each class will have its own header file and cpp file. The names of these files should be the same as the class name.
Example: Class **Date** has two files: **Date.h** and **Date.cpp**

In addition to header files for each class, create a header file called "general.h" that will hold the general defined values for the project, such as:

```
TAX (0.13)              The tax value for the NFI items
MAX_SKU_LEN (7)         The maximum size of a SKU

DISPLAY_LINES (10)      Product lines to display before each pause

MIN_YEAR (2000)         The min and max for year to be used for date validation
MAX_YEAR (2030)

MAX_NO_RECS (2000)      The maximum number of records in the data file.
```

This header file should get included were these values are needed.

Note that all the code developed for this application should be in **`sict`** namespace.

MILESTONE 1: THE DATE CLASS

The Date class encapsulates a date value in three integers for year, month and day, and is readable by istreams and printable by ostream using the following format for both reading and writing: YYYY/MM/DD

Complete the implementation of Date class using following information:

Member Data:

`int year_;` Holds the year; a four digit integer between MIN_YEAR and MAX_YEAR, defined in "general.h"

`int mon_;` Month of the year, between 1 to 12

`int day_;` Day of the month, note that in a leap year February is 29 days, (see mday() member function)

`int readErrorCode_;` This variable holds an error code with which the caller program can reference to find out if the date value is valid, and if not, which part is erroneous. The following possible error values should be defined in the date class header-file as follows:

```
NO_ERROR   0  -- No error the date is valid
CIN_FAILED 1  -- istream failed when entering information
YEAR_ERROR 2  -- Year value is invalid
MON_ERROR  3  -- Month value is invalid
DAY_ERROR  4  -- Day value is invalid
```

Private Member functions:

`int value()const;` (this function is already implemented and provided)

This function returns a unique integer number based on the date. This value is used to compare two dates. (If the value() of date one is larger than date two, then date one is after/more recent than date two).

`void errCode(int errorCode);`

Sets the readErrorCode_ member-variable to one of the values mentioned above.

Constructors:

No argument (default) constructor: sets year_, mon_ and day_ to "0" and readErrorCode_ to NO_ERROR.

Three argument constructor: Accepts three arguments to set the values of year_, mon_ and day_ attributes. It also sets the readErrorCode_ to NO_ERROR.

Public member-functions and operators

Comparison Logical operator overloads:

```
bool operator==(const Date& D)const;
bool operator!=(const Date& D)const;
bool operator<(const Date& D)const;
bool operator>(const Date& D)const;
bool operator<=(const Date& D)const;
bool operator>=(const Date& D)const;
```

These operators return the comparison result of the return value of the value() function applied to left and right operands (The Date objects on the left and right side of the operators).
For example `operator<` returns true if `this->value()` is less than `D.value()` or else it returns false.

`int mdays(int mon)const;` (this function is already implemented and provided)

Returns the number of days in a month.

Accessor or getter member functions:

`int errCode()const;` Returns the readErrorCode_ value.
`bool bad()const;` Returns true if readErrorCode_ is not equal to zero.

IO member-funtions

`std::istream& read(std::istream& istr);`

Reads the date in the following format: YYYY?MM?DD (e.g. 2016/03/24 or 2016-03-24) from the console. This function will not prompt the user. If the istream `(istr)` fails at any point, it will set the `readErrorCode_` to `CIN_FAILED` and will NOT clear the istream object. If the numbers are successfully read in, it will validate them to be in range, in the order of year, month and day (see general header-file and mday() method for acceptable ranges for years and days respectively. Month can be between 1 and 12 inclusive). If any of the numbers are not within range, the `readErrorCode_ will be set` to the appropriate error code and stop further validation. Irrespective of the result of the process, this function will return the incoming istr argument.

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

Writes the date using the ostr argument in the following format: YYYY/MM/DD, then return the ostr.

Non-member IO operator overloads:

After implementing the Date class, overload the operator<< and operator>> to work with cout to print a Date, and cin to read a Date, from/to the console respectively.

Use the read and write methods and DO NOT use friends for these operator overloads.

Make sure the prototype of the functions are in Date.h.

Preliminary task

To kick-start the first milestone, clone/download milestone 1 from the repository and implement the Date class.

Compile and test you code using the four tester programs starting from tester number 1 up to 4.

MILESTONE 1 SUBMISSION

If not on matrix already, upload **general.h, Date.h, Date.cpp** and the four testers 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_ms1 <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.

MILESTONE 2: THE ERRORMESSAGE CLASS

The ErrorMessage class encapsulates an error message in a dynamic C-style string and also is used as a flag for the error state of other classes.

Later in the project, if needed in a class, an ErrorMessage object is created and if an error occurs, the object is set a proper error message.
Then using the **isClear()** method, it can be determined if an error has occurred or not and the object can be printed using **cout** to show the error message to the user.

Private member variable (attribute):

ErrorMessage has only one private data member (attribute):

```
char* message_;
```

Constructors:

No Argument Constructor, (default constructor):

`ErrorMessage();`

Sets the **message_** member variable to **nullptr.**

Constructors:

`ErrorMessage(const char* errorMessage);`

Sets the **message_** member variable to **nullptr** and then uses the **message()** setter member function to set the error message to the **errorMessage** argument.

`ErrorMessage(const ErrorMessage& em) = delete;`

A deleted copy constructor to prevent an ErrorMessage object to be copied.

Public member functions and operator overloads (methods):

`ErrorMessage& operator=(const ErrorMessage& em) = delete;`

A deleted assignment operator overload to prevent an ErrorMessage object to be assigned to another.

`ErrorMessage& operator=(const char* errorMessage);`

Sets the message_ to the **errorMessage** argument and returns the current object (*this) by:

- De-allocating the memory pointed by **message_**
- Allocating memory to the same length of **errorMessage + 1** and keeping the address in **message_** data member.
- Copying **errorMessage** c-string into **message_.**
- Returning *this.
  You can accomplish this by reusing your code and calling the following member functions:
  Call **clear()** and then call the setter **message()** function and retrun *this.

`virtual ~ErrorMessage();`

de-allocates the memory pointed by **message_.**

`void clear();`

de-allocates the memory pointed by **message_** and then sets **message_** to **nullptr.**

```
bool isClear()const;
```

returns true if **message_** is **nullptr.**

```
void message(const char* value);
```

Sets the **message_** of the ErrorMessage object to a new value by:

- de-allocating the memory pointed by **message_.**
- allocating memory to the same length of **value + 1** keeping the address in **message_** data member.
- copying **value** c-string into **message_.**

```
const char* message()const;
```

returns the address kept in **message_**.

Helper operator overload:

Overload **`operator<<`** so the ErrorMessage can be printed using **`cout`**.

If ErrorMessage **isClear,** Nothing should be printed, otherwise the c-string pointed by **message_** is printed.

MILESTONE 2 SUBMISSION

If not on matrix already, upload **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 200_ms2 <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.

MILESTONE 3: THE STREAMABLE INTERFACE

The Streamable class is provided to enforce inherited classes to implement functions to work with fstream and iostream objects.

Code the Streamable class in a file called Streamable.h.

You do not need the Date or ErrorMessage class for this milestone.

Pure virtual member functions (methods):

Streamable class, being an interface, has only four pure virtual member functions (methods) with following names:

```
1- fstream& store(fstream& file, bool addNewLine = true)const
        Is a constant member function (does not modify the owner) and receives
        and returns references of std::fstream.
        In future milestones children of Streamable will implement this method,
        when they are to be stored in a file.
2- fstream& load(std::fstream& file)
        Receives and returns references of std::fstream.
        In future milestones children of Streamable will implement this method,
        when they are to be read from a file.
3- ostream& write(ostream& os, bool linear)const
        Is a constant member function and returns a reference of std::ostream.
        write() receives two arguments: the first is a reference of std::ostream
        and the second is a bool argument called linear.
        In future milestones children of Streamable will implement this method
        when they are to be printed on the screen in two different formats:
        Linear: the class information is to be printed in one line
        Form: the class information is to be printed in several lines like a
        form.
4- istream& read(istream& is)
        Returns and receives references of std::istream.
        In future milestones children of Streamable will implement this method
        when their information is to be received from console.
```

As you already know, these functions only exist as prototypes in the class declaration in the header file.

After implementing this class, compile it with Myfile.cpp, MyFile.h and StreamableTester.cpp. The program should compile with no error and using the tester program you will be able to read and append text to the streamable.txt file.

MILESTONE 3 SUBMISSION

If not on matrix already, upload **Streamable.h, MyFile.h, MyFile.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.

MILESTONE 4: THE PRODUCT CLASS

Create a class called Product. The class Product is responsible for encapsulating a general Streamable Product.

Although the class Product is a Streamable (inherited from Streamable), it will not implement any of the pure virtual member functions, therefore it remains abstract.

The class Product is implemented under the sict namespace. Code the Product class in the Product.cpp and Product.h files.

You do not need the Date class for this milestone.

Product Class specs:

Private Member variables:

sku_: Character array, MAX_SKU_LEN + 1 characters long
This character array holds the SKU (barcode) of the items as a string.
name_: Character pointer
This character pointer points to a dynamic string that holds the name of the Product
price_: Double
Holds the Price of the Product
taxed_: Boolean
This variable will be true if this Product is taxed
quantity_: Integer
Holds the on hand (current) quantity of the Product.
qtyNeeded_: Integer
Holds the needed quantity of the Product.

Public member functions and constructors

No argument Constructor;

This constructor sets the Product to a safe recognizable empty state. All number values are set to zero in this state.

Five argument Constructor;

Product is constructed by passing 5 values to the constructor:
the SKU, the Name, if the Product is taxed or not, , the Price and the Needed Quantity.
The constructor:

- Copies the SKU into the corresponding member variable up to MAX_SKU_LEN characters.
- Allocates enough memory to hold the name in the name_ pointer and then copies the name into the allocated memory pointed to by the member variable name_.
- Sets quantity on hand to zero.
- Sets the rest of the member variables to the corresponding values received by the arguments.
- If value for Product being taxed is not provided, it will set the taxed_ flag to the default value “true”

Copy Constructor;

See below:

Dynamic memory allocation necessities

Implement the copy constructor and the operator= so the Product is copied from and assigned to another Product safely and without any memory leak. Also implement a virtual destructor to make sure the memory allocated by name_ is freed when Product is destroyed.

Accessors

Setters:

Create the following setter functions to set the corresponding member variables:

- **sku**
- **price**
- **name**
- **taxed**
- **quantity**
- **qtyNeeded**

All the above setters return void.

Getters (Queries):

Create the following constant getter functions to return the values or addresses of the member variables: (these getter methods do not receive any arguments)

- **sku**, returns constant character pointer
- **price**, returns double
- **name**, returns constant character pointer

- **taxed,** returns boolean
- **quantity,** returns integer
- **qtyNeeded,** returns integer

Also:

- **cost,** returns double

Cost returns the cost of the Product after tax. If the Product is not taxed the return value of cost() will be the same as price.

- **isEmpty** returns bool

isEmpty return true if the Product is in a safe empty state.

All the above getters are constant methods, which means they CANNOT modify the owner.

Member Operator overloads:

Operator== : receives a constant character pointer and returns a Boolean.

This operator will compare the received constant character pointer to the SKU of the Product, if they are the same, it will return true or else, it will return false.

Operator+= : receives an integer and returns an integer.

This operator will add the received integer value to the quantity of the Product, returning the sum.

Operator-= : receives an integer and returns an integer.

This operator will reduce the quantity of the Product by the integer value returning the quantity after reduction.

Non-Member operator overload:

Operator+= : receives a double reference value as left operand and a constant Product reference as right operand and returns a double value;

This operator multiplies the cost of the Product by the quantity of the Product and then adds that value to the left operand and returns the result.

Essentially this means this operator adds the total cost of the Product on hand to the left operand, which is a double reference, and then returns it.

Non-member IO operator overloads:

After implementing the Product class, overload the operator<< and operator>> to work with ostream (cout) to print a Product to, and istream (cin) to read a Product from, the console. Use the write() and read()methods of Streamable class to implement these operator overloads.

Make sure the prototype of the functions are in Product.h.

MILESTONE 4 SUBMISSION

If not on matrix already, upload **general.h, Streamable.h, Product.h, Product.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_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.

MILESTONE 5: THE AMAPRODUCT AND AMAPERISHABLE CLASSES

AmaProduct Class

Implement the AmaProduct class in AmaProduct.h and AmaProduct.cpp as a class derived from a Product class.
Essentially, AmaProduct is a Streamable Product class that is not abstract.

An AmaProduct is a Product designed to work with the Aid Management Application.

Private member variables

```
char fileTag_;
```
Holds a single character to tag the records as Perishable or non-Perishable product in a file.

```
char unit_[11];
```
Unit of Measurement (i.e. Kg, Liters, …)

Protected member variables

AmaProduct class has only one protected member variable of type ErrorMessage, called **err_**.

Constructor:

AmaProduct has only one constructor that receives the value for the filetag_ member variable and if this value is not provided, it will use the character 'N' as the default value for the argument.

Public member functions

const char* unit()const;

returns a constant pointer to the **unit_** member variable.

void unit(const char* value);

Copies the incoming value string into the **unit_** string.
Make sure copying does not pass the size of the **unit_** array.

AmaProduct implements all four pure virtual methods of the class Streamable (the signatures of the functions are identical to those of Streamable).

`fstream& AmaProduct::store(fstream& file, bool addNewLine)const`:

Using the operator<< of ostream first writes the fileTag_ member variable and a comma into the **file** argument, then without any formatting or spaces writes all the member variables of Product, comma separated, in following order:

```
sku, name, price, taxed, quantity, unit, quantity needed
```

and if addNewLine is true , it will end them with a new line. Then it will return the file argument out.

Example:

```
N,1234,box,123.45,1,1,kg,5<Newline>
```

`fstream& AmaProduct::load(fstream& file)`

Using the operator>>, ignore and getline methods of istream, AmaProduct reads all the comma separated fields form the current record in the file and sets the member variables using the

setter methods. When reading the fields, load assumes that the record does not have the **"N," (the filetag_)** at the beginning, so it starts the reading from the sku.

No error detection is done.
At the end the file argument is returned.

Hint: create temporary variables of type double, int and string and read the fields one by one, skipping the commas. After each read, set the member variables using setter methods.

`ostream& AmaProduct::write(ostream& os, bool linear)const.`

If the **err_** member variable is not clear (use isClear member function). It simply prints the err_ using ostr and returns ostr. If the **err_** member variable is clear (No Error) then depending on the value of linear, write(), prints the Product in different formats:

Linear is true:

Prints the Product values separated by Bar "|" character in following format:

```
1234   |Box                 | 139.50|   1|kg        |   5|
```

Sku: left justified in MAX_UPC_LEN characters
Name: left justified 20 characters wide (truncated if longer than 20 chars)
Cost: (not the price) right justified, 2 digits after decimal point 7 chars wide
Qty on hand: right justified 4 characters wide
Unit: left justified 10 characters wide
Quantity needed: right justified 4 characters wide
NO NEW LINE

Linear is false:

Prints one member variable per line in following format:

```
Sku: 1234
Name: box
Price: 123.45
Price after tax: 139.50
Quantity On Hand: 1 kg
Quantity Needed: 5
NO NEW LINE
```

Or the following is the product is not taxed:

```
Sku: 1234
Name: box
Price: 123.45
Price after tax: N/A
Quantity On Hand: 1 kg
Quantity Needed: 5
NO NEW LINE
```

Afterwards, write returns the ostr argument.

`istream& AmaProduct::read(istream& istr)`:
Receives the values using istream (the istr argument) exactly as the following:

```
Sku: 1234<ENTER>
Name: box<ENTER>
Unit: kg<ENTER>
Taxed? (y/n): y<ENTER>
Price: 123.45<ENTER>
Quantity On hand: 1<ENTER>
Quantity Needed: 5<ENTER>
```

if **istr** is in a **fail** state, then the function exits doing nothing other than returning istr.

When entering the Taxed field, check the character entered, if it is one of 'Y','y','N' or 'n' then clear (flush) the keyboard, otherwise set the message of **err_** object to "**Only (Y)es or (N)o are acceptable"** and the rest of the entry is skipped.
Also to make the error handling is consistent with istream's fail flag, call the following function:
`istr.setstate(ios::failbit);`
This will manually put the istream in failure state. By doing this, the error handling will be consistent with istream's error detection.

If at any stage istr fails (cannot read), **err_** should be set to the proper error message and the rest of the entry is skipped and nothing is set in the Product (also no error message is displayed).
Here are the possible error messages:

fail at Price Entry:	**Invalid Price Entry**
fail at Quantity Entry:	**Invalid Quantity Entry**
fail at Quantity Needed Entry:	**Invalid Quantity Needed Entry**

Since the rest of the member variables are text, istr cannot fail on them, therefore there are no error messages designated for them. Make sure at the end of the Entry you do not read the last new line or flush the keyboard.

At end, read will return the istr argument.

AmaPerishable Class

Implement the AmaPerishable class in AmaPerishable.h and AmaPerishable.cpp to be derived out of an AmaProduct class. Essentially, AmaPerishable is an AmaProduct class that with an expiry date.

Private member variables

AmaPerishable class has one private member variable:

- A Date, called expiry_

Constructor:

AmaPerishable has only one default constructor invokes the AmaProduct constructor passing the value 'P' for the fileTag argument.

Public member functions

Public Accessors (setters and getters)

const Date& expiry()const;

returns a constant reference to expiry_ member variable.

void expiry(const Date &value);

Sets the expiry_ attribute to the incoming value.

Virtual method implementations

AmaPerishable re-implements all four virtual methods of the AmaProduct.

`fstream& store(fstream& file, bool addNewLine = true)const`:

Calls the parent's **store** passing the file and "false" as arguments and then writes a comma and the expiry date into the file. If the addNewLine argument is true, it will write a newline into the file.
The outcome will be something like this being written to the file:
P,1234,water,1.5,0,1,liter,5,2017/10/12<NEWLINE>

`fstream& load(fstream& file)`
Calls the parent's load passing the file as the argument and then calls the read method of the expiry_ object passing the file as the argument and then ignores one character (reads one character from the file and dumps it).

`ostream& write(ostream& ostr, bool linear)const`:
Calls the write of the parent passing ostr and linear as arguments. Then if err_ is clear and product is not empty:

if linear is true, it will just print the expiry otherwise it will first go to new line and then print:
"Expriy date: " and the print the expiry date.
The outcome will be like this:

```
1234   |water                |   1.50|   1|liter     |   5|2017/10/12
```

OR:

```
Sku: 1234
Name: water
Price: 1.50
Price after tax: N/A
Quantity On Hand: 1 liter
Quantity Needed: 5
Expiry date: 2017/10/12
```

NO NEW LINE

Afterwards, write returns the ostr argument.

`istream& read(istream& istr)`:

It will call parent's read passing istr as argument.

Then if **err_** is clear it will print:

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

then it will read the date from the console into a temporary Date object.

If Expiry (Date) Entry fails then, depending of the error code stored in the Date object, set the error message in **err_** to:

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**

Then to be consistent with istream failure, manually sets the istr to failure mode by calling this function:

```
istr.setstate(ios::failbit);
```

If nothing has failed, then it will set the expiry date of the object to the temporary Date object read from the console.

At end, read will return the istr argument.

MILESTONE 5 SUBMISSION

If not on matrix already, upload **general.h, Date.h, Date.cpp, ErrorMessege.h, ErrorMessege.cpp, Streamable.h, Product.h, Product.cpp, AmaProduct.h, AmaProduct.cpp, AmaPerishable.h, AmaPerishable.cpp** and the tester files 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_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.

MILESTONE 6: THE AID MANAGEMENT APLICATION

For your final milestone for this project, create a class called "AidApp".

AidApp is a class that uses the previously created classes in this project to give the user the capability to store and retrieve Perishable and Non-Perishable product information from and to a file.

The AidApp class has several private member functions and only two public functions.

A description for each function is provided below. Those that are more complex will be complimented with suggested pseudo code to help you implement the function. You may use the pseudo code as suggested, revise it to improve on the logic, or implement your own logic.

Code the AidApp class in files **AidApp.h** and **AidApp.cpp**.

AidApp class

Private member variables:

```
char filename_[256];
```

Holds the name of the text file used to store the product information.

```
Product* product_[MAX_NO_RECS];
```

An array of Product pointers, MAX_NO_RECS long. (i.e. Each element of this array is a product pointer).

```
fstream datafile_;
```

An fstream instance used to create and access a file.

```
int noOfProducts_;
```

Number of Products (perishable or non-perishable) pointed to by the product_ array.

The Constructor

The AidApp constructor receives a constant char string called filename and then:
1- Copies filename to filename_ member variable
2- Sets all the product_ elements to nullptr
3- Sets noOfProducts_ to zero
4- Loads the Records (calls the member function to do this)

Private member functions:

Copy and assignment prevention

Make sure the AidApp cannot get copied or assigned to another AidApp.

`void pause()const;`

Prints: `"Press Enter to continue..."<NEWLINE>` then waits for the user to hit enter. If the user hits any other key, the key is ignored. Only the ENTER key will terminate this function.

`int menu();`

Menu() displays the menu as follows and waits for the user to select an option.

```
Disaster Aid Supply Management Program
1- List products
2- Display product
3- Add non-perishable product
4- Add perishable product
5- Add to quantity of purchased products
6- Sort products
0- Exit program
> _
  ^ The cursor should be positioned here when the menu is printed
```

- If the selection is valid, menu() will return the selection otherwise it will return -1
- The standard input buffer (keyboard) must be cleared before the function exits.

`void loadRecs();`

Opens the data file for reading. If the file does not exist, it will create an empty file, close the file and exit.

Otherwise, read the data file and store each record to the pointer array of Products accordingly (overwriting the old ones).

Note: Prevent memory leaks by deleting each Product element pointer in the array before overwriting/replacing the element with a new Product pointer.

After reading all the records, close the file.

```
Pseudo code:
Set readIndex to zero
Open the file for reading (use ios::in)
if the file is in fail state it means there is no file on the disk, then
  clear the failure
  close the file
  open the file for writing (ios::out) to create the file
  close thefile
otherwise
  until reading fails loop
     deallocate the memory pointed by product pointer at readindex
     read one char character to identify type of Product into Id character
     if Id character is P
       Dynamically create a Perishable product and hold it in product pointer at readIndex
     if Id character is N
       Dynamically create a Non-perishable product and hold it in product pointer at readIndex
     if either P or N is read
        skip the comma in the file
        load the product from the file (using its load method)
        add one to read index
  continue the loop
set number of products to readIndex
close the datafile
```

`void saveRecs();`

- Opens the file for writing
- Loops through the product_ array up to noOfProducts_ and stores them in the datafile_
- Closes the file

`void listProducts()const;`

- First it will print the following title :

```
 Row | SKU    | Product Name        | Cost   | QTY| Unit     |Need| Expiry
-----|--------|---------------------|--------|----|----------|----|----------
```

- Then loops through the products_ array up to noOfProducts_ and prints the row number (four spaces wide, right justified), followed by a bar/pipe character (|) surrounded by two spaces
- Then prints the current Product in the loop followed by a newline

- When the number of printed items reaches 10, it will pause until the user hits the Enter key.
- With each loop iteration, it will calculate the total cost of the products in a double value using the operator+= implemented by the Product class.
- When the list is done, a line of dashes will be printed:

```
---------------------------------------------------------------------
```

- Finally, the total cost will be printed:

```
Total cost of support: $9999.99
```

The total cost value is printed with a Dollar sign at left and two digits after decimal point

`int SearchProducts(const char* sku)const;`

Loops through the product_ up to noOfProducts_ and checks each of them for the same SKU as the incoming argument using the operator== implemented by the Product class. If a match is found it will return the index of the found Product in the product_ array, otherwise it will return -1.

`void addQty(const char* sku);`

Updates the quantity on hand for a product.
updateQty() searches for the Product with the same sku as the incoming argument. If not found it will display:

```
"Not found!"<NEWLINE>
```

If found, it will display the Product in non-linear format and then asks for an integer for quantity purchased:

```
"Please enter the number of purchased items: "
```

If it cannot read the integer it prints:

```
"Invalid quantity value! "<NEWLINE>
```

If it can read the integer, it makes sure the amount is less than or equal to the amount required (i.e. less than qtyNeeded() - quantity()). If it is less than or equal, it will add the value to the quantity on hand of the product using operator+= implemented by the Product class. If the value is not less than or equal the amount needed, it will only accept the amount required and prints a message to return the extra:

```
"Too many items; only 999 is needed, please return the extra 99 items. "< NEWLINE >
```

Lastly, all records will be saved back to the file and a message displayed:

"Updated!" <NEWLINE>

Make sure after the entry the keyboard is flushed.

`void addProduct(bool isPerishable);`

Depending on the value of the argument being true or false, create a Perishable or Non-perishable Product and get the values from the user and add it to the end of the product_ array and save the records (call saveRecs()). If there is an error, display the Product and exit the function (this will show the error message).

`int run();`

Display the menu, receive the user's selection, and do the action requested (follow with a pause using the pause() function), and repeat (redisplay the menu…) until the user selects zero to exit.

```
1- List products
```
List the products.
```
2- Display product
```
Ask for a sku using this prompt
`"Please enter the SKU: "`
(receive input from console) then search for the Product.
If found, display the Product information in non-liner format
Otherwise display:
`"Not found!"`
```
3- Add non-perishable product
```
Add a Non-Perishable product to the system using the addProduct function
Load the records.
```
4- Add perishable product
```
Add a Perishable product to the to the system using the addProduct function
Load the records.
```
5- Add to quantity of purchased products
```
Ask for a sku using this prompt
`"Please enter the SKU: "`
(receive input from console) and then add to the quantity using the addQty() function.
```
6- Sort products
```
Use the template provided in sort.h and add a method to the AidApp class to sort the products. In this method call the sort function template and pass the product array and number of items.

Save the products to the file so the file records will also be sorted and print `"Sorted!"`.

```
0- Exit program
```

The program will terminate printing:
`"Goodbye!!"`

In case of invalid menu selection the program will print:
`"===Invalid Selection, try again.==="`
Followed by a pause (pause() function) before redisplaying the menu.
The Run function will return 0 when it ends.

MILESTONE 6 SUBMISSION

TBA