

Winter 2016.  
OOP244 Assignment

# Aid Management Application (AMA) V3.0

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 and quantity in 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 form 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)

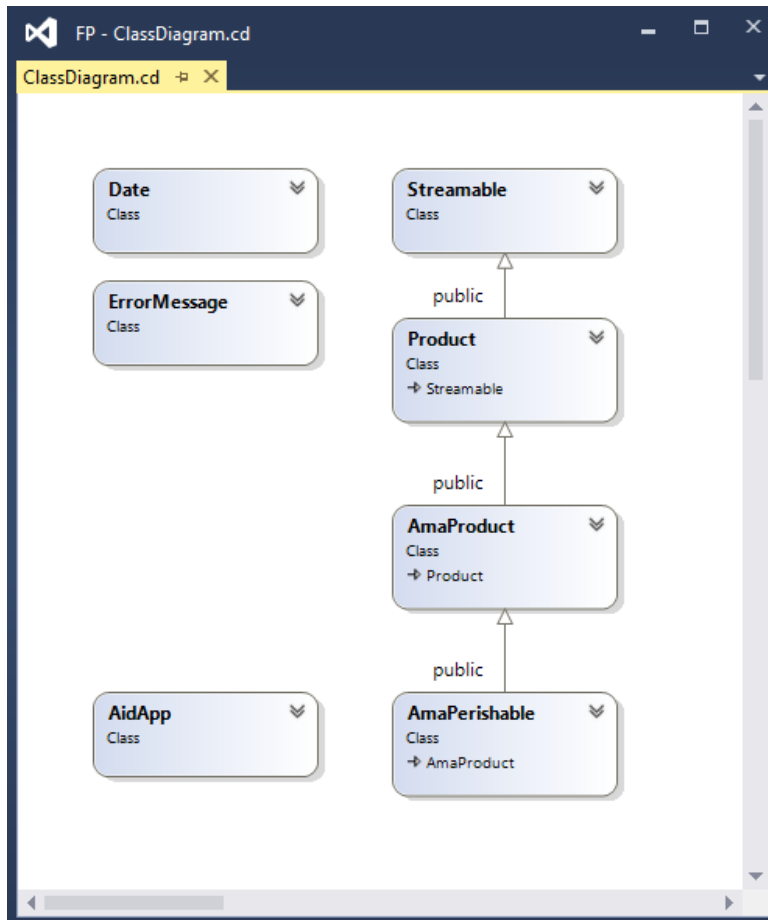
### AMA\_Perishable

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) + 35 days
  - The Date class. Due: KO + 5 days
  - The ErrorMessage class Due: KO + 8 (3 days)
  - The Streamable class Due: KO + 9 (1 day)
  - The Product class Due: KO + 15 (6 days)
  - The AMA product classes Due: KO + 25 (10 days)
  - The AidApp class. Due: KO + 35 (10 days)

## 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:

<code>TAX (0.13)</code>	The tax value for the NFI items
<code>MAX_SKU_LEN (7)</code>	The maximum size of a SKU
<code>DISPLAY_LINES (10)</code>	Product lines to display before each pause
<code>MIN_YEAR (2000)</code>	The min and max for year to be used for date validation
<code>MAX_YEAR (2030)</code>	
<code>MAX_NO_RECS (2000)</code>	The maximum number of records in the data file.

This header file should get included where 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 find out if the date value is valid or not, and which part is erroneous if so. The possible error values should be defined in date 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 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`. No validation required.

#### 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()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 is 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 is not in range, it will set the `readErrorCode_` 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 returns 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`, respectively, from the console.

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 for milestone 1 from [https://github.com/Seneca-244200/OOP-FP\\_MS1.git](https://github.com/Seneca-244200/OOP-FP_MS1.git)

and implement the `Date` class.

Compile and test you code with 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:

```
Sections SAA and SBB:
~edgardo.arvelaez/submit ms1 <ENTER>
Section SCC and SDD:
~fardad.soleimanloo/submit ms1 <ENTER>
Section SEE and SFF:
~eden.burton/submit ms1 <ENTER>
```

and follow the instructions.

## MILESTONE 2: THE ERRORMESSAGE CLASS

Clone/download milestone 2 from [https://github.com/Seneca-244200/OOP-FP\\_MS2.git](https://github.com/Seneca-244200/OOP-FP_MS2.git)

and implement 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 return \*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:

**Sections SAA and SBB:**

~edgardo.arvelaez/submit ms2 <ENTER>

**Section SCC and SDD:**

~fardad.soleimanloo/submit ms2 <ENTER>

**Section SEE and SFF:**

~eden.burton/submit ms2 <ENTER>

and follow the instructions.

## MILESTONE 3: THE STREAMABLE INTERFACE

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

Code the Streamable class in the Streamable.h file provided in OOP-FP\_MS3 repository (No CPP file) on github: [https://github.com/Seneca-244200/OOP-FP\\_MS3.git](https://github.com/Seneca-244200/OOP-FP_MS3.git)

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:

```
Sections SAA and SBB:  
~edgardo.arvelaez/submit ms3 <ENTER>  
Section SCC and SDD:  
~fardad.soleimanloo/submit ms3 <ENTER>  
Section SEE and SFF:  
~eden.burton/submit ms3 <ENTER>
```

and follow the instructions.