

Fall Semester 2018

# Aid Management Application (AMA)

Version 3.4.1

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 a disaster area. The application tracks the quantity of items needed, tracks the quantity on hand, and stores the information in a file for future use.

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

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

To complete this project you will need to create several classes that encapsulate your solution.

## OVERVIEW OF THE CLASSES TO BE DEVELOPED

The classes used by the application are:

### **Date**

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

### **Error**

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

### **Good**

A class that manages a non-perishable good object.

### **Perishable**

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

### **iGood**

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

- read itself from the console or write itself to the console
- save itself to a text file or load itself from a text file
- compare itself to a unique C-style string identifier
- determine if it is greater than another good in the collating sequence
- report the total cost of the items on hand
- describe itself
- update the quantity of the items on hand
- report its quantity of the items on hand
- report the quantity of items needed
- accept a number of items

Using this class, the client application can

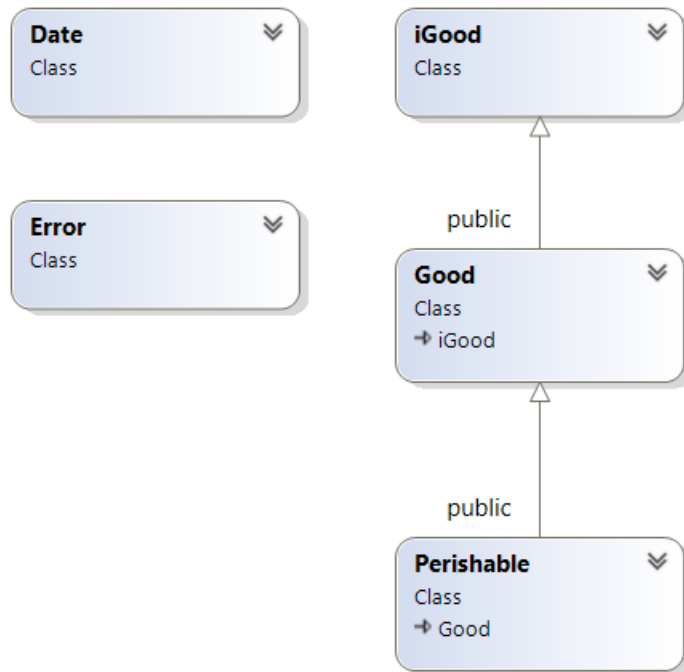
- save its set of iGoods to a file and retrieve that set at a later time
- read individual item specifications from the keyboard and display them on the screen
- update information regarding the number of each good on hand

## THE CLIENT APPLICATION

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

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

## PROJECT CLASS DIAGRAM



## PROJECT DEVELOPMENT PROCESS

The Development process of the project consists of 5 milestones and therefore 5 deliverables. Shortly before the due date of each deliverable a tester program and a script will be provided for testing and submitting the deliverable. The approximate schedule for deliverables is as follows

- Due Dates (at 11:59pm on each day)
  - The Date module Due: November 2<sup>nd</sup>, 11 days
  - The Error module Due: November 9<sup>th</sup>, 7 days
  - The Good module Due: November 21<sup>st</sup>, 12 days
  - The iGood interface Due: November 23<sup>rd</sup>, 2 days
  - The Perishable module Due: November 28<sup>th</sup>, 3 days

## SUBMISSION INSTRUCTIONS

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

Note that by the end of the semester you **MUST have submitted a fully functional project to pass this subject**. If you fail to do so, you will fail the subject. If you do not complete the final milestone by the end of the semester and your total average, without your project's mark, is above 50%, your professor *may* record an "INC" (incomplete mark) for the subject. With the release of your transcript you will receive a new due date for completion of your project. The maximum project mark that you will receive for completing the project after the original due date will be "49%" of the project mark allocated on the subject outline.

## FILE STRUCTURE OF THE PROJECT

Each class belongs to its own module. Each module has its own header (.h) file and its own implementation (.cpp) file. The name of each file without the extension is the name of its class.

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

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

## MILESTONE 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:

```
const int min_year = 2018
const int max_year = 2038
const int min_date = 751098
```

### Private members:

#### Data:

The year – a four digit integer between `min_year` and `max_year`

The month of the year – a value between 1 and 12 inclusive

The day of the month – a value between 1 and the number of days in the month (see the `mday(int,int)` member function described below) – Note that February has 29 days in a leap year.

This class uses 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
```

### Member functions:

**int** mdays(int month, int year) **const**; (this query has already been implemented for you and is provided)

This query returns the number of days in the specified month of the specified year.

**void** errCode(int errorCode);

This function sets the error state variable to one of the values listed above.

### Public members:

#### 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:

$$= \text{year} * 372 + \text{month} * 13 + \text{day}$$

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 modifier

`int errorCode() 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 an `std::ostream` object in the following format: YYYY/MM/DD, and then returns a reference to the `std::ostream` object.

### Helper functions:

`operator<<`

This operator works with an `std::ostream` object as the left operand to print a date to the console.

`operator>>`

This operator works with an `std::istream` object as the left operand to read a date from the console.

Call your `read` and `write` member functions on the right operand in these operators; DO NOT use friends for these global operator overloads.

Include the prototypes for these two helper operators in the header file. Place their prototypes immediately after the class definition.

### Testing:

Test your code in Visual Studio using the tester program supplied as the main module.

## MILESTONE 1 SUBMISSION

Upload **Date.h** and **Date.cpp** with the tester program to your matrix account. Compile and rerun your code on matrix and make sure everything works properly.

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

```
~profname.proflastname/submit 244_ms1 <ENTER>
```

and follow the instructions.

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

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