# Fall 2016.

# **Wedding Planner Application**

OOP244 Assignment Milestone 2

V1.2 (changed UPC to SKU, changed due dates and classes to implement)

When planning a wedding, the most important thing is to provide all required items on time in order to ensure that the entire ceremony progresses smoothly.

Your job for this project is to prepare an application that manages the list of goods required for a wedding and the delivery date of those goods, if applicable. Your application keeps track of the quantity of goods needed and the quantity on hand, and stores this information in a file for future use.

The types of goods needed for a wedding are divided into two categories;

* OnShelf: Items that can be purchased from a store and are available when requested.
* CustomMade: Items that are made to order and will only be ready at some future date.

To prepare the application you need to create several classes that encapsulate the different tasks at hand.

**Classes to be developed**

The classes required by your application are:

**Date** A class that manages a date.

**Error** A class to keep track of the errors occurring during data entry and user interaction.

**ReadWritable** *(This class is already implemented and provided)*This interface (a class with “only” pure virtual functions) enforces the classes that inherit from it to be *Read and Writable*. Any class derived from “ReadWritable” 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.

**Good** A class derived from ReadWritable, containing general information about an item needed for the wedding, like the name, Universal Product Code (SKU), price, etc.

**OnShelf** A class holding information for OnShelf items derived from the Good class that implements the requirements of the ReadWritable class (i.e. implements the pure virtual methods of the ReadWritable class)

**CustomMade** A class derived from the OnShelf class that holds a delivery date.

**WPlanner** The class that manages OnShelf and CustomMade goods. This class manages the listing, adding and updating the goods for a wedding.

**Project Development Process**

Your development work on this project has five milestones and therefore is divided into five deliverables. Shortly before the due date of each deliverable a tester program will be provided to you. Use this tester program to test your solution and use the script to submit each of the deliverables. The approximate schedule for deliverables is as follows

* Date class Due: Nov 3rd
* Error Class Due: Nov 9th
* Good class Due: Nov 15th
* OnShelf and CustomMade classes Due: Nov 21th
* WPlanner class. Due: Nov 29th

**File Structure For the project**

Each class will have its own module; a header (.h) file and an implementation (.cpp) file. The names of these files should be the same as the class name.

In addition to the header files for each class, create a header file called wpgeneral.h that defines general values for the project, such as:

TAX (0.13) The tax rate for the goods

MAX\_SKU\_LEN (4) The maximum size of a SKU code

MIN\_YEAR (2000) The min year used to validate year input

MAX\_YEAR (2030) The max year used to validate year input

MAX\_NO\_RECS (2000) The maximum number of records in the data file.

Include this header file wherever you use these values.

Enclose all the code developed for this application within the ict namespace.

**Milestone 1: the Date class**

The Date class encapsulates a single date value in the form of three integers: year, month and day. The date value is readable by an istream and printable by an ostream using the following format: YYYY/MM/DD (the separators do not have to be “/”)

Complete the implementation of the Date class under the following specifications:

## **Member Data (attributes):**

**int** year\_; Year; a four digit integer between MIN\_YEAR and MAX\_YEAR, as defined in wpgeneral.h

**int** mon\_; Month of the year, between 1 and 12

**int** day\_; Day of the month; note that in a leap year February has 29 days, (see mday() member function)

**int** readErrorCode\_; Error code which identifies the validity of the date and, if erroneous, it identifies the part that is incorrect. Define the possible error values in the Date header-file as follows:

NO\_ERROR 0 -- No error - the date is valid

CIN\_FAILED 1 -- istream failed on accepting 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 (private methods):**

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

This function returns a unique integer number based on the date. You can use this value to compare two dates. If the value() of one date is larger than the value of another date, then the former date (the first one) follows the second.

**void** errCode(**int** errorCode);

Sets the readErrorCode\_ member variable to one of the possible values listed above.

## **Constructor:**

This constructor accepts three arguments to set the values of year\_, mon\_ and day\_. It also sets the readErrorCode\_ to NO\_ERROR.

## **Public member-functions (methods) and operators:**

Relational 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 result of comparing the left operand to the right operand. These operators use the value() member function in their comparison. For example operator< returns true if this->value() is less than D.value(); otherwise returns false.

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

This function returns the number of days in the month based on year\_ and mon\_ values.

**Accessor or getter member functions (methods):   
int** errCode() **const**; Returns the readErrorCode\_ value.

**bool** bad() **const**; Returns true if readErrorCode\_ is not equal to zero.

**IO member-funtions (methods):**

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

Reads the date in following format: YYYY/MM/DD (e.g., 2015/03/24) from the console. This function does not prompt the user. If the istream (i.e., istr) object fails at any point, this function sets readErrorCode\_ to CIN\_FAILED and does **NOT** clear the istream object. If the istream object reads the numbers successfully, this function validates them. It checks that they are in range, in the order of year, month and day (see the wpgeneral header-file and the mday() function for acceptable ranges for years and days respectively). If any number is not within range, this function sets readErrorCode\_ to the appropriate error code and omits any further validation. Irrespective of the result of the process, this function returns a reference to the istream (i.e., istr) object. std::ostream& write(std::ostream& ostr) **const**;

This function writes the date to the ostream (i.e., ostr) object in the following format: YYYY/MM/DD, then returns a reference to the ostream object.

**Non-member IO operator overloads: (Helpers)**

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 member functions. **DO NOT** use friends for these operator overloads.

Include the prototypes for these helper functions in the date header file.

# **Preliminary task**

To kick-start the first milestone clone or download the Visual Studio project, or individual files for milestone 1 from <https://github.com/Seneca-244200/OOP_MS1>.

Start your development and test your implementation with tester number 1 and work your way up to tester number 4. Then compile your code with the main tester (oop\_ms1\_tester.cpp) and make sure your code passes all the tests.

If not on matrix already, upload your Date.cpp, Date.h, wpgeneral.hand oop\_ms1\_tester.cpp 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 oop\_ms1 <ENTER>**

Following the instructions, test and demonstrate execution of your program.

**Milestone 2: the ERROR CLASS**

Clone/download milestone 2 from <https://github.com/Seneca-244200/OOP-FP_MS2.git>

and implement the Error class.

The Error 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 Error 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):**

Error has only one private data member (attribute):

**char\* message\_;**

# **Constructors:**

No Argument Constructor, (default constructor):

**Error();**

Sets the **message\_** member variable to **nullptr.**

Constructors:

**Error(const char\* Error);**

Sets the **message\_** member variable to **nullptr** and then uses the **message()** setter member function to set the error message to the **Error** argument.

**Error(const Error& em) = delete;**

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

# **Public member functions and operator overloads (methods):**

**Error& operator=(const Error& em) = delete;**

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

**Error& operator=(const char\* errorMessage);**

Sets the message\_ to the **Error** argument and returns the current object (\*this) by:

* De-allocating the memory pointed by **message\_**
* Allocating memory to the same length of **Error + 1** and keeping the address in **message\_** data member.
* Copying **Error** 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 ~Error();**

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 Error 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 Error can be printed using **cout**.  
If Error **isClear,** Nothing should be printed, otherwise the c-string pointed by **message\_** is printed.

**Milestone 2 SUBMISSION**

If not on matrix already, upload **Error.h, Error.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 oop\_ms2 <ENTER>**

Following the instructions, test and demonstrate execution of your program.