Fall 2016.

Wedding Planner Application

OOP244 Assignment Milestone 4
(Due Nov 22nd 23:59)

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, Stock Keeping Unit (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& conInput(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.h` and `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* Error);
```

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.

MILESTONE 3: THE GOOD CLASS

Note: You should NOT have more than one return statement in a function. This rule (having one point of entry to and one point of exit out of a function) was established during the structured programming era decades ago and is not allowed in your code.

THE READWRITABLE INTERFACE

The ReadWritable class enforces inherited classes to implement functions that work with fstream and iostream objects. This class is already implemented and the code is in the file ReadWritable.h under OOP_MS3 repository on github. There is no cpp file for this module, since it is an interface and all the functions (methods) in this class are pure virtual.

Pure virtual member functions (methods):

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

1- Store
   Is a constant member function (does not modify the owner) and receives and returns a reference to an std::fstream object.
   In future milestones children of ReadWritable will implement this method for instances that can be stored in a file.
2- Load
   Receives and returns a reference to an std::fstream object.
   In future milestones children of ReadWritable will implement this method for instances that can be read from a file.
3- display
   Is a constant member function and returns a reference to an std::ostream object. This function receives two arguments: the first is a reference to an std::ostream object and the second is a bool argument called linear.
   In future milestones children of ReadWritable will implement this method for instances that can be printed on the screen in either of two formats:
   Linear: the object information is printed in a single line
   Form: the object information is printed in several lines like a form.
4- conInput
   Returns and receives a reference to an std::istream object.
   In future milestones children of conInput will implement this method for instance that receive input from the console.

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

THE GOOD CLASS

Create a class called Good. The class Good is responsible for encapsulating a general ReadWritable item.

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

The class Good is implemented under the ict namespace. Code the Good class in the Good.cpp and Good.h files provided in OOP_MS3 repository on github:
https://github.com/Seneca-244200/OOP_MS3

You do not need the Date class for this milestone.

Good 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 Good
price_: Double
Holds the Price of the Good
taxed_: Boolean
This variable will be true if this item is taxed
quantity_: Integer
Holds the on hand (current) quantity of the item.
qtyNeeded_: Integer
Holds the quantity needed to purchase

Public member variables and constructors

Constructor:

Good is constructed by passing 5 values to the constructor:
the SKU, the Name, the price, the Quantity needed and if the good is taxed or not.
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 the value for the good being taxed is not provided, it will set the taxed_ flag to the default value “true”

Dynamic memory allocation necessities

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

Accessors

Setters:

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

- **sku**
- **price**
- **name**
- **taxed**
- **quantity**
- **qtyNeeded** (quantity Needed)

All the above setters return void.

Getters:

Create the following getter functions to return the values or addresses of the member variables:

- **sku**, returns a constant character pointer
- **price**, returns a double
- **name**, returns a constant character pointer
- **taxed**, returns a boolean
- **quantity**, returns an integer
- **qtyNeeded** (quantity Needed), returns an integer

Also:

- **cost**, returns a double

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

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 Good, if they are the same, it will return true or else, it will return false. This operator cannot modify the owner.

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

This operator will add the received integer value to the quantity on hand of the Good, returning the sum.

Non-Member operator overload:

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

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

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

Non-member IO operator overloads:

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

Note: operator<<, displays the Good in "linear" format. (i.e, linear is true)

Make sure the prototype of the functions are in **Good.h**.

MILESTONE 3 SUBMISSION

If not on matrix already, upload **Good.h, Good.cpp, GoodTester.cpp, ReadWritable.h** and **wpgeneral.h** 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_ms3 <ENTER>

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

MILESTONE 4: THE ONSHELF AND CUSTOMMADE CLASSES

(DUE NOV 22ND 23:59)

Before starting milestone 4 modify the store function prototype in ReadWritable Class to:

```
virtual std::fstream& store(std::fstream& file, bool addNewLine = true)const = 0;
```

OnShelf Class

Implement the OnShelf class in OnShelf.h and OnShelf.cpp as a class derived from a Good class.
Essentially, OnShelf is a ReadWritable Good class that is not abstract.

An OnShelf is a Good designed to work with the Wedding Planner Application.

Private member variable

```
char recTag_;
```
Holds a single character to tag the records as CustomMade (‘**C’**) or OnShelf (‘**O’**) Good in a file.

Protected member variable

OnShelf class has only one protected member variable of type Error, called **err**_.

Constructor:

OnShelf 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 ‘O’ as the default value for the argument.

Public member functions

OnShelf implements all four pure virtual methods of the class ReadWritable (the signatures of the functions are identical to those of ReadWritable) as follows:

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

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

```
sku, name, price, taxed, quantity, quantity needed
```
and if addNewLine is true , it will end them with a new line. Then it will return the file argument out.

Example:

```
0,1234,box,123.45,1,1,5<Newline>
```

`fstream& OnShelf::load(fstream& file)`
Using the operator>>, ignore and getline methods of istream, OnShelf 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 **“O,” (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& OnShelf::display(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, display(), prints the Good class in different formats:

Linear is true:

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

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

Sku: left justified in MAX_SKU_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
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
Quantity Needed: 5
NO NEW LINE
```

Or the following if the Good is not taxed:

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

Afterwards, write returns the ostr argument.

`istream& OnShelf::conInput(istream& istr):`

Receives the values using istream (the istr argument) exactly as the following:

```
Sku: 1234<ENTER>
Name: box<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 a 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 Good class (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, conInput will return the istr argument.

CustomMade Class

Implement the CustomMade class in CustomMade.h and CustomMade.cpp to be derived from an OnShelf class. Essentially, CustomMade is an OnShelf class with an delivery date.

Private member variables

CustomMade class has one private member variable:

- A Date, called delivery_

Constructor:

CustomMade has only one default constructor invokes the OnShelf constructor passing the value 'C' for the recTag argument.

Public member functions

Public Accessors (setters and getters)

const Date& delivery()const;

returns a constant reference to delivery_ member variable.

void delivery(const Date &value);

Sets the delivery_ attribute to the incoming value.

Virtual method implementations

CustomMade re-implements all four virtual methods of the OnShelf.

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

Calls the parent's store passing the file and a "false" value as arguments and then writes a comma and the delivery 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:

C,1234,water,1.5,0,1,5,2017/10/12<NEWLINE>

`fstream& load(fstream& file)`

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

`ostream& display(ostream& ostr, bool linear)const:`

Calls the display of the parent passing ostr and linear as arguments. Then if err_ is clear and Good is not empty:

if linear is true, it will just print the delivery otherwise it will first go to new line and then print:

"Expriy date: " and then print the delivery date.

The outcome will be like this:

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

OR:

```
Sku: 1234
Name: water
Price: 1.50
Price after tax: N/A
Quantity On Hand: 1
Quantity Needed: 5
Expiry date: 2017/10/12
```
NO NEW LINE

Afterwards, write returns the ostr argument.

`istream& conInput(istream& istr);`
It will call the parent's conInput 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 delivery date of the object to the temporary Date object read from the console.

At end, conInput will return the istr argument.

MILESTONE 4 SUBMISSION

If not on matrix already, upload wpgeneral.h, Date.h, Date.cpp, Error.h, Error.cpp, ReadWritable.h, Good.h, Good.cpp, OnShelf.h, OnShelf.cpp, CustomMade.h,

CustomMade.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 oop_ms4 <ENTER>

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