Sprint 1 Guidance

CSE 1325 – Spring 2020 – ELSA

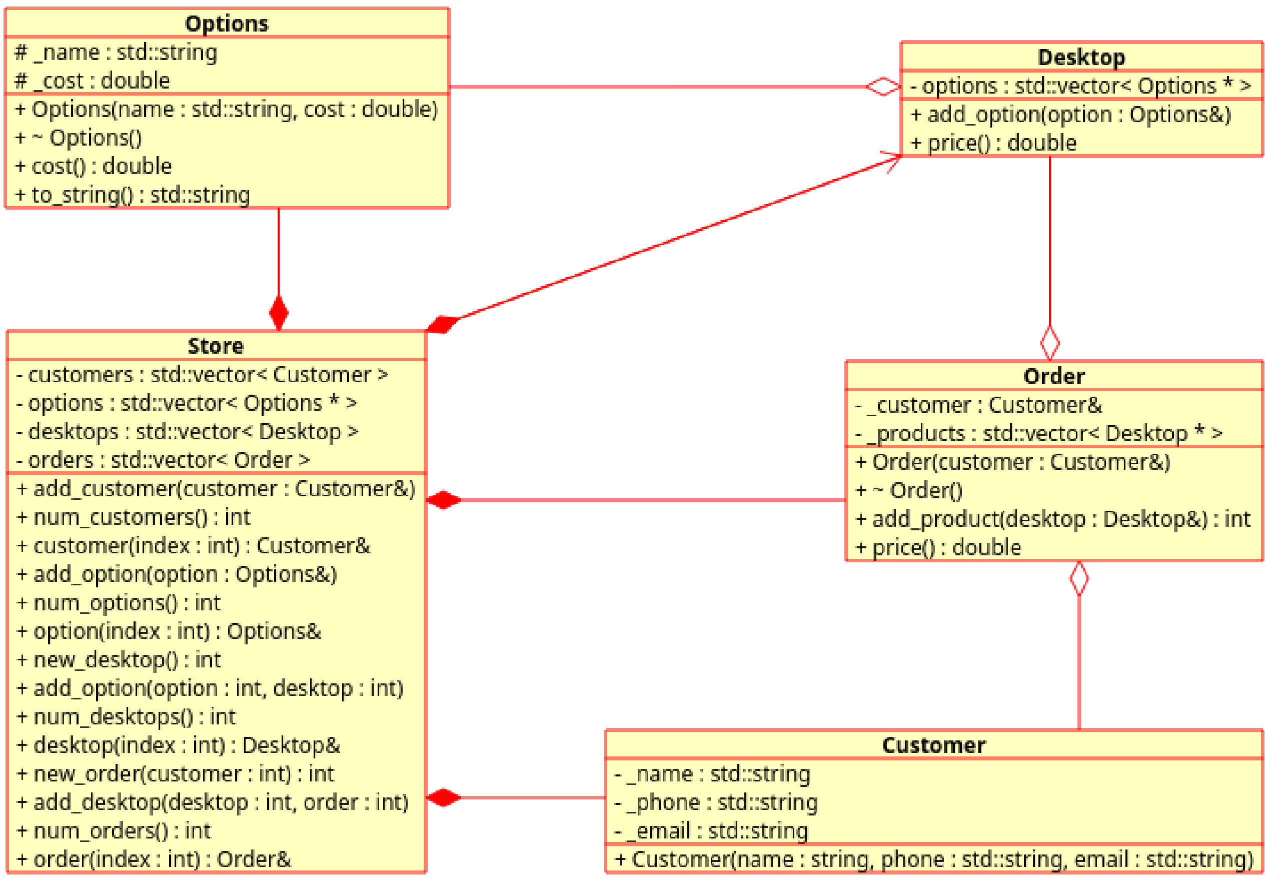
IMPORTANT: Do NOT use full_credit, bonus, or extreme_bonus subdirectories for the final project. Keep all files to be graded in the cse1325/elsa directory on GitHub for the duration of the final project.

The Exceptional Laptops and Supercomputers Always (ELSA) store offers the coolest (ahem) deals in computing technology for the savvy computer geek and their lucky friends. Each computer can be hand-crafted to match the technologist's exact needs, with a growing selection of convenient predefined configurations already purchased by your discerning peers (and competitors).

They now belatedly seek to automate their storefront, replacing paper forms and ink pens with the miracle of modern computing technology. Your goal is to prove that you can implement their store management system and thus win the contract to build it, with all of the associated fame and cash.

Your implementation is permitted to vary from any class diagram provided with the final project as needed without penalty, as long as you correctly implement both the letter of and the intent of the feature list. **If you have questions about the acceptability of changes that you are considering, contact a TA or the professor first!**

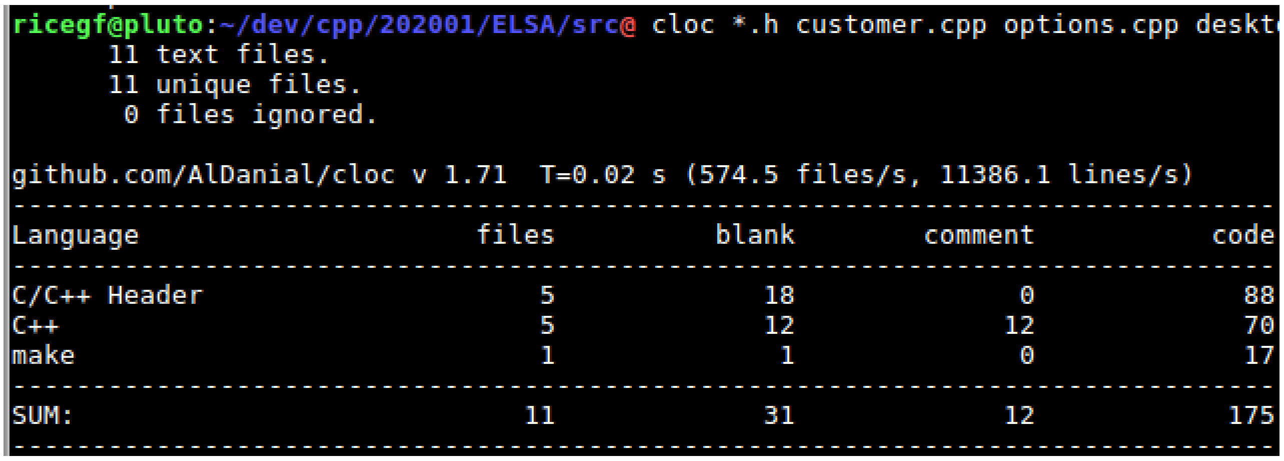
This class diagram is for sprint 1 only:



Options
# _name : std::string
# _cost : double
+ Options(name : std::string, cost : double)
+ ~ Options()
+ cost() : double
+ to_string() : std::string
Desktop
- options : std::vector< Options * >
+ add_option(option : Options&)
+ price() : double
Store
- customers : std::vector< Customer >
- options : std::vector< Options * >
- desktops : std::vector< Desktop >
- orders : std::vector< Order >
+ add_customer(customer : Customer&)
+ num_customers() : int
+ customer(index : int) : Customer&
+ add_option(option : Options&)
+ num_options() : int
+ option(index : int) : Options&
+ new_desktop() : int
+ add_option(option : int, desktop : int)
+ num_desktops() : int
+ desktop(index : int) : Desktop&
+ new_order(customer : int) : int
+ add_desktop(desktop : int, order : int)
+ num_orders() : int
+ order(index : int) : Order&
Order
- _customer : Customer&
- _products : std::vector< Desktop * >
+ Order(customer : Customer&)
+ ~ Order()
+ add_product(desktop : Desktop&) : int
+ price() : double
Customer
- _name : std::string
- _phone : std::string
- _email : std::string
+ Customer(name : string, phone : std::string, email : std::string)

Note carefully the open-diamond aggregation lines indicating references or pointers (for Desktop's Options and for Order's Customer and Desktops) versus the closed-diamond composition lines showing where the actual objects are *stored* – in the Store class (desktops are stored on the heap but still managed by the Store class and merely referenced by the Order class).

A brief description of each class follows. Don't be intimidated – all but the Store class are principally just a few one-liner method implementations, and even Store isn't that much code. The suggested solution comes in at 175 lines of non-comment non-blank lines of code total (excluding main.cpp).



```
ricegf@pluto:~/dev/cpp/202001/ELSA/src@ cloc *.h customer.cpp options.cpp deskt
      11 text files.
      11 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.71  T=0.02 s (574.5 files/s, 11386.1 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C/C++ Header                     5             18              0             88
C++                              5             12             12             70
make                             1              1              0             17
-------------------------------------------------------------------------------
SUM:                            11             31             12            175
-------------------------------------------------------------------------------
```

Class Overviews

Customer

This is a simple, invariant class that simple holds a customer's personal information. Access the information in text format by defining the operator<< friend function for it.

Options

The Options class represents parts of the computer that the Customer can select, such as CPU model, RAM size, disk size and technology, and so on. For Sprint 1, we simply track the name and cost of each option. You'll want the operator<< function for it.

ASIDE: You may notice that both Desktop and Store use pointers to track Option instances, a not so subtle indication that this will become the base class for derived classes storing technology-specific information (maybe GHz for CPU or Gbytes for RAM) to be accessed polymorphically. The to_string *virtual* method exists to enable polymorphic behavior for operator<<, as we'll see.

Desktop

This class merely stores the collection of Options that comprise the desktop. The add_option method contributes one Option reference to the vector. The price method simply returns the sum of the Option costs. You'll want an operator<< for Desktop, too (picking up a pattern?).

Order

Order is an *association class* (remember our class relationships discussion in Lecture 07?) binding a Customer (supplied via the constructor) to one or more Desktops (added via repeated calls to the add_product method) that the Customer is purchasing. The price() method returns the sum of the Desktop product prices from the _products vector. You'll want an operator<< for Order as well (definitely a pattern!).

Store

Store maintains the 4 vectors holding the instances of the other classes. Note that options is a vector of *pointers* to Option objects, which should be created on the heap. The other vectors can hold the objects themselves.

The Customer and Option vectors use a 3-method interface:

- add_customer / add_option, which just pushes the parameter onto the vector
- num_customers / num_options, which returns the size() method of the vector
- customer / option, which returns a reference to the object in the vector at the index provided as a parameter.

With this interface, it's trivial to create a 3-term for loop that iterates over each vector element.

The Desktop and Order vectors use a 4-method interface:

- new_desktop / new_order, which pushes a new object onto the vector *and returns its index*. The index is used to add options to the new desktop or desktops to the new order.
- add_option / add_desktop, which is given *the index* of the option to add to the desktop *at the provided index* / the *index* of the desktop to add to the order *at the provided index*. Thus, these two methods are used to add an *existing* option / desktop to an *existing* desktop / order.
- num_desktops / num_orders and desktop / order work exactly like num_customers and customer, enabling a 3-term for loop that iterates over each vector element.

At last, Store doesn't need an operator<<! It simply contains via composition far too much data to simply **`std::cout << store;`** !

Main

An example main.cpp is provided **cse1325-prof/elsa/sprint1/main.cpp** that interfaces with the Store class (and manipulates the other classes as needed) to implement a basic menu-driven command line interface. However, if you have time, I encourage you to write your own interface. It's simply very good experience. But be forewarned – it's a 1-sprint wonder, to be replaced next sprint with windows and mouse interactions.

Where We’re Going

The classes comprise the model (in the model-view-controller pattern sense) for our computer store. The command line interface will be replaced starting in sprint 2 with a graphical (windowed) user interface based on the gtkmm library. Later sprints will also add file I/O (to save and load the store) and polymorphism (to store and retrieve additional information about specific Options).

Note that sprint 1 isn’t due until March 17 due to spring break, making it the rare 2-week sprint / homework assignment.