CME 211 C++ Exam

Due Thursday December 6th, 2018 at 11:59 P.M. PDT

1 Short-Answer (31 points total)

For the following section of questions, please record your answers in a `short_answers.md` file. I'd recommend budgeting approximately 30 minutes for this portion; ideal answers will be succinct.

1.1 Vocabulary (14 points)

(A: 4 points)

Describe what is happening in *each* statement below using a single word or phrase.

```
int n;
n = 0;
int f(void);
int f(void) { /* insert sub-routine contents here. */ }
```

(B: 3 points)

Explain in one sentence what is meant when we say that C++ is *statically typed*?

Explain in another sentence what is meant when we say that C++ is *strongly typed*?

In another single sentence, reconcile your previous answer with the following code snippet, which compiles *without error*.

```
int pi = 3.14;
```

(C: 2 points)

Write a one-line C++ declaration for a statically allocated array.

Separately, describe what is meant by the sub-phrase "statically allocated" within *this* context: in particular, what exactly is static?

(D: 2 points)

Explain in a single sentence the behavior of the following function definition.

```
int Sum(int n) {
  int sum;
  for (int i = 0; i < n; i++)
    sum += i;
  return sum;
}
```

In a separate sentence, describe the *simplest* improvement to the program.

(E: 3 points)

What's the name for the statement on line 3 below?

```
1   struct S {
2     int s;
3     S() { s = 42; }
4   };
```

What does this statement return, and when would it get used in a program?

1.2 Data Types (8 points)

(A: 3 points)

Suppose you wish to assign a unique identifier to each student in *this* course. What data type might you consider using?

What about a unique identifier for each individual in the US population today?

Lastly, what about a unique identifier for each individual in the world population today?

(B: 1 point)

What kind of warning might you expect to earn (assuming appropriate flags enabled) when compiling the following snippet of code? Assume that we have already used `#include<vector>`.

```
std::vector<int> v = {0, 1, 2, 4, 8, 16, 32, 64};
for (int i = 0; i < v.size(); i++) {}
```

(C: 2 points)

Is the value of the following variable well-defined? If so: what is it; if not: why? Assume we have already used `#include<string>` and that the statement is contained within a `main` sub-routine.

```
std::string s;
```

(D: 2 points) Consider the following user-interface for an inner-product sub-routine:

```
double Dot(std::vector<double> x, std::vector<double> y);
```

Recall that an inner-product is defined for two vectors with equal length n as $\langle x, y\rangle = \sum_{i=1}^{n} x_i y_i$. What modifications (if any) would you recommend for the interface to the function `Dot`? Explain your reasoning and write out any supporting C++ code.

1.3 C-Style Strings (3 points, where up to one point can count as bonus)

(A: 3 points) Suppose you are given a method to compute the length of a null-terminated byte string.

```
unsigned strlen(const char *str) {
  unsigned n = 0;
  while(*str++) n++;
  return n;
}
```

First, consider the input argument `const char *str`. What is `const` refer to, here? Is it the pointer or the character that is constant? Hint: this program compiles without error.

Consider the expression `*str++`, explain what is happening here in at most two detailed sentences. (Note: if it's helpful, the order-of-evaluation for the expression may be disambiguated by `*(str++)`.)

Separately, what is the computational *work* required by this algorithm as a function of the input argument size? (How should we even measure the input argument size?)

1.4 Compilers (2 points)

(A: 2 points) You wrote the following code in an interface file, `foo.hpp`.

```
int Foo(void);
```

In an implementation file `foo.cpp`, you define the sub-routine.

```
#include "foo.hpp"
int Foo(void) { return 42; }
```

You decide to call this in a `main` sub-routine.

```
#include "foo.hpp"
int main() {
  int x = Foo();
}
```

You compile the program as follows, but are greeting with an error:

```
g++ -std=c++11 main.cpp -o main
In function ‘main’:
  main.cpp:(.text+0x9): undefined reference to ‘Foo()’
  collect2: error: ld returned 1 exit status
```

What *type* of error is this? How can we easily rectify this? (Provide concrete syntax)

1.5 Memory and Segfaults (4 points)

(A: 4 points) Assume the following code is contained within a `main` sub-routine.

```
int arr[3];
arr[3] = 0;
std::cout << arr[3] << '\n';
```

In a single word (or phrase) describe the outcome of the above expressions; is (any) output guaranteed? What should we expect as output, and how reliable is this expectation?

Can we compile this program (with warning flags)?

Is the program likely to run, or should we expect a segmentation fault?

Is there a tool or compiler option we could enable to facilitate spotting these phenomena?

2 Programming (80 points total, inclusive of 10 writeup points)

Our objective is to write a C++ program which emulates an inventory management system and attempts to facilitate a requested transaction/purchase.[1]

2.1 Overview

Inventory Valuation At a minimum, in order to invoke our program we require as input a set of inventory be provided (via an input file). If this is the only input given to the `main` program (see section 2.7), then our sole job will be to compute the sum total number of items in the inventory as well as the total valuation of all goods.

Request-for-Purchase A client may *optionally* make a request-for-purchase, in which case the user of the program must specify (in addition to the above inventory file) (i) a particular item (possibly contained in our inventory), (ii) a non-negative quantity to request, and (iii) a real-number/value describing how much cash the client will tend in the transaction. Each of these can be fed in directly as command line arguments to the program. We remark that as inventory managers we seek to generate as much revenue as possible by selling as many items as we have in stock in order to fulfill the request (or whatever portion of it the client can actually afford). As certificate, we must provide a "receipt" specifying how much change shall be returned to the client (if any) in the form of console output (see section 2.2). See section 2.3.1 for how to calculate change.

Assumptions on Inputs We *can* assume that the input file used to specify inventory is well-formatted as described below in section 2.2. We can assume that the `item-requested`, is a sequence of characters, and that `qty-requested` and `lump-sump` will be a well-formatted integer and real-number respectively.

Edge Cases We *cannot* assume that we stock the item that is requested! Further, we *cannot* assume that the quantity of items requested or the lump-sum provided will be non-negative, nor can we assume that the funds provided are sufficient for the quantity specified. If the item isn't in stock, we will assume we can't sell anything at all to the client. If the client does not provide sufficient funds, we should sell them the maximal quantity that they can afford (possibly zero). If the client attempts to request a *negative quantity of goods* (i.e. they are attempting to *sell us* inventory!), *or* if the client attempts to provide a *negative amount of funds* (i.e. they are attempting to rob us!), then in either case we should print some sort of informative message and exit the program gracefully.

Usage We will provide a `main` program which fixes the interface for your inventory management system. The inventory manager can be invoked as follows:

```
./main inventory_fname [item-requested] [qty-requested] [lump-sum]
```

[1]Recommended time: 2hr 30m. Allotment: 30 min to read the instructions and *understand* the problem/-solution; 60 min to actually type out code; 30 min to debug code; and 30 min to justify/write-up `README.md`.

2.2 Data Formats

Input: Inventory File Format We associate with each `item` a `price` and `quantity`.

```
item_1, x_1, n_1
...
item_N, x_N, n_N
```

For each item (or line in our input), we anticipate a *comma separated* tuple describing: (i) `item_i` the name of the item (e.g. "candy bar" or "water") (ii) x_i the price of the item (a real number with two significant digits of decimal precision,[2] e.g. `0.95` denotes 95¢), and (iii) n_i the integer quantity of the item that we will stock. We may assume that the input file is reasonably well formatted, entirely consistent with the description above. Additionally, we may assume that the same item-name does *not* appear more than once in an input file. Note that the item-name may contain spaces; consider using functionality from `stringstream`.

Console Output: `CashRegister` Format We will only deal with bills in the following denominations: $\{1, 5, 10, 20\}$, and our change system will consist of pennies, nickels, dimes, and quarters. If we return any change to a customer, we should print the summary to console in a manner consistent with what is specified below:

```
20 : x1
10 : x2
5  : x3
1  : x4
0.25 : x5
0.10 : x6
0.05 : x7
0.01 : x8
```

Here, each line item describes a particular quantity of money. The first item on each line describes the magnitude of the denomination, and the second quantity (following a colon delimiter) describes the non-negative quantity of the item available as part of the funds. E.g. if `x4` were `3`, this encodes three one-dollar bills are to be returned to the customer.

2.3 Inventory Class

Tracking Inventory: Implementation Options We need a way to track inventory; there are many equally valid choices for implementations. Consider choosing from:

1. Define a (helper) `Item` class (storing item-name, associated price, and quantity in-stock) and use this to define a data-attribute to track our entire inventory via `vector<Item>`,
2. Use one `map` to relate item-names to associated prices, and another map from item-names to associated quantities (in-stock), or
3. Use a single `map` from item-names to a tuple (or struct) of length two (describing price and quantity respectively).[3]

[2] We will assume that our pricing system does *not* utilize fractional cents, i.e. finest granularity is a penny.

[3] You might wonder: why not use a `set` of `items` if an `item` object tracks both price and quantity in

Please Don't Note that another option that we ask you *not* implement is to have three separate attributes (each a vector) describing the items, prices, and quantities where we maintain the invariant that the ordering between attributes is consistent throughout the lifetime of the object. But this is too easy, since it only uses `vector`'s on atomic types or `std::string`. We can instead remove the assumption of these separate attributes being ordered if we adhere to an implementation recommended. This tests either (i) our ability to nest containers (one we create) or (ii) our ability to use containers beyond simple `vector`s.

2.3.1 Fulfilling a Request For Purchase

We require that you implement an `AttemptExport` method as part of your `Inventory` class, which attempts to export (or sell) a select quantity of a *particular* item to a customer. This method should return an `int` describing the number of items that will be transacted, based on the method described below in the paragraph on Making Change (2.3.1); if the order cannot be fulfilled, the method shall return zero. If a transaction occurs, apart from returning an appropriate non-zero value the method should keep our internal inventory/state up-to-date. Note: you may find it helpful to define additional (intermediary) attributes that are mutated by the `AttemptExport` method, even if not explicitly specified in the instructions.

Price set by Policy of Maximal Fulfillment and Client Budget It is the responsibility of the cash-register (or inventory-holder) to fulfill the order to the fullest extent possible. I.e. if a customer desires 5 units of an item, but we only have 3 in stock, then we shall sell them 3 units of said item. If, on the other hand, the client can only afford 4 units, then we shall sell them 4 units of said item. We emphasize that our approach is simple and greedy: sell as many goods as the customer desires and can afford. The register must have a way of communicating the final charge of the *purchased* items as an input to the next step.

Making Change We simply assume that the customer tends us a set of funds in a lump sum and that it is our responsibility to make change using our `CashRegister` object.[4] Note that in an *arbitrary* coin-system, tending exact change using the fewest number of coins (across denominations) is a hard problem. However, in the US coin system, it *is true* that if we simply greedily choose the largest denominator not exceeding the amount (remaining) to be returned as change, that we can yield a uniquely optimal solution. a simple iterative approach is enough to solve this sub-problem. If any non-zero amount of change is required to be given back to the customer, we shall print this information to console in a format for funds described above in section 2.2 paragraph on Console Output `CashRegister` Format.

stock, since in this case we only need to associate one element in our container with each unique type of item we stock and (some type of) a `set` is perfect for this. However, as we'll learn in CME 212, whether we use an *ordered* or *unordered* set, we will be tasked with overloading either a comparison operator or hashing function respectively, and this is beyond the scope of what we expect you to know for this course.

[4]In practice, when the customer provides funding for the purchase they tend some amount of physical cash using a combination of denominations or a set of funds (which we *could* describe using a `CashRegister` object, for example) but we can ignore this detail for now.

Simplifying Assumption: Infinite Funds To simplify things, we may assume that that as holders of inventory we have access to a cash-register with sufficient funds to make change for any transaction requested; in particular, assume we have an infinitude of each denomination available to us. I.e., as many pennies, quarters, ones, etc. as we need for any transaction.

Helper Method Specifications Recall that `AttemptExport` is a core method whose interface/behavior is well-specified in section 2.3.1.

- The `Print` method for a `Inventory` object should output to console (for each item) the item's name followed by an *indented* newline which states both the price of the item and the quantity in stock (themselves separated by a comma followed by a space).
- The `SummarizeTransaction` shall be responsible for two details: (i) it shall *return* as value the revenue generated from the transaction, and (ii) it shall *print* to console (as side-effect) the change being returned to the customer (if any is deserved), in the format described in section 2.2 paragraph "Console Output: `CashRegister` Format".
- The `TotalStock` method (used in the `std::printf` statement) should return a non-negative *integer* describing how many items there are in-stock at the moment; i.e. we wish to account for not only unique items, but their multiplicities.
- We ask that the `Value` method (also used in the `std::printf`) return the total value (possibly containing a fractional component) of our inventory when considering the prices across all goods in stock

2.4 `CashRegister` Class

The `Print` method should output to console (for each denomination in our set) the number of items returned of corresponding value as change. This class should be capable of greedily seeks to pull money from a (imaginary) cash-register in such a way that the sum of money returned is exactly equal to the input argument provided to the constructor. The object should be capable of tracking how many bills of each type are used, and we leave the design decision of how to implement this entirely up to you! Note: we never use a `CashRegister` in our `main.cpp` file, but it is still useful for internal implementation in say, `SummarizeTransaction`.

2.5 Summary of Requirements (70 points)

The following should all be methods as part of your `Inventory` and `CashRegister` classes that you will implement, and these points are separate from the 20 points available to earn as part of constructing the core-data structures backing `Inventory` class. (Note that you should decide which methods belong to each class and how they interface with each other, and please consider carefully the return types specified as well as the types of input arguments.)

1. `Inventory Class` (36 points specified below, plus 20 possible for design)
   - (4 points) A `Print` method.
   - (18 points) An `AttemptExport` method.
   - (2 points) A `SummarizeTransaction` method.
   - (6 points *each*) A `TotalStock` *and* a `Value` method.
2. A `CashRegister` class (4 points specified below, plus 10 possible for design),
   - (4 points) A `Print` method, itself implicitly invoked by `Inventory::SummarizeTransaction`.

2.6 Writeup (10 additional points)

Focus on Explaining Design Considerations Please place your write-up a separate markdown file, which we ask that you label as `README.md`. In your writeup, you should justify which implementation you chose to back your `Inventory`. You should also discuss whether the keyword `new` appears in your program, and why this is appropriate. Did you consider how methods accept arguments (e.g. use of `const` references)? Public/private attributes?

Discuss an aspect of your program that you are proud of. If you know of ways your program could be further improved, you may describe any known shortcomings and corresponding fixes. Lastly, if you found that you had to modify `main.cpp` (highly discouraged!) in order to "get a running program" then please also justify any modifications made.

Include as part of your `README` the command used to compile your program.

2.7 `main.cpp`

Lastly, we emphasize that your program *should embrace* an Object Oriented Paradigm. To aid you in your development, we provide a `main` program which we ask you *do not* modify.

```cpp
#include <cstdio>
#include <iostream>
#include <string>
// Optional: #include "Item.hpp"
#include "Inventory.hpp"
int main(int argc, char *argv[]) {
    if (argc < 2) {
        std::cout << "Usage:\n\t./main inventory_fname [requested_item] [requested_qty] [lump_sum]\n";
        return 0;
    }
    Inventory i(argv[1]);
    i.Print();
    if (argc >= 3) {
        std::cout << "Handling an export request." << std::endl;
        std::string item(argv[2]);
        int   qty   = std::atoi(argv[3]);
        float funds = std::atof(argv[4]);
        if (i.AttemptExport(item, qty, funds))
          std::cout << i.SummarizeTransaction() << '\n';
    }
    std::printf("Total number of items in inventory: %d, valued at %g\n", i.TotalStock(), i.Value());
}
```