Practice Midterm 3

CS106B

_____

(Print name legibly)

_____

(SUID **number**)

Exam Instructions

There are **3** questions worth a total of **100** points. Write all answers directly on the exam paper. This printed exam is closed-book and closed-device; you may refer only to our provided reference sheet. You are required to write your SUID number in the blank at the top of each odd-numbered page.

C++ Coding Guidelines

Unless otherwise restricted in the instructors for a specific problem, you are free to use any of the CS106B libraries and classes. You don't need *#include* statements in your solutions; just assume the required header files (*vector.h*, *strlib.h*, etc.) are visible. You do not need to declare prototypes. You are free to create helper functions unless the problem states otherwise. Comments are not required, but when your code is incorrect, comments could clarify your intentions and help the graders award partial credit.

The Stanford University Honor Code (2023 Revision)

The Honor Code is an undertaking of the Stanford academic community, individually and collectively. Its purpose is to uphold a culture of academic honesty.

Students will support this culture of academic honesty by neither giving nor accepting unpermitted academic aid in any work that serves as a component of grading or evaluation, including assignments, examinations, and research.

Instructors will support this culture of academic honesty by providing clear guidance, both in their course syllabi and in response to student questions, on what constitutes permitted and unpermitted aid. Instructors will also not take unusual or unreasonable precautions to prevent academic dishonesty.

Students and instructors will also cultivate an environment conducive to academic integrity. While instructors alone set academic requirements, the Honor Code is a community undertaking that requires students and instructors to work together to ensure conditions that support academic integrity.

In signing below, I acknowledge, accept, and agree to abide by the Honor Code.

_____

(signature)

This cover sheet has been revised to reflect the format you should expect to see on this quarter's exam.

1. **Chaos and Quirkiness (C++, ADTs, and Big-O) (40 pts)** Consider the following functions:

```
void chaosAdd(Vector<char>& v, char ch)
{
   Vector<char> reversed;

   while (!v.isEmpty()) {
      reversed.add(v.remove(v.size() - 1));
   }

   v.add(ch);

   for (int i = 0; i < reversed.size(); i++) {
      v.add(reversed[i]);
   }
}

char chaosRemove(Vector<char>& v)
{
   if (v.size() == 0) {
      error("chaosRemove() is fail. YIKES!");
   }

   char retval = v.remove(0);
   Vector<char> reversed;

   while (!v.isEmpty()) {
      reversed.insert(0, v.remove(0));
   }

   for (int i = 0; i < reversed.size(); i++) {
      v.add(reversed[i]);
   }

   return retval;
}


Vector<char> createChaosVector(string str) {
   Vector<char> v;
   for (char ch: str) {
      chaosAdd(v, ch);
   }
   return v;
}

void destroyChaosVector(Vector<char>& v) {
   while (!v.isEmpty()) {
      cout << chaosRemove(v);
   }
   cout << endl;
}

void causeChaos(string str) {
   Vector<char> v = createChaosVector(str);
   destroyChaosVector(v);
}
```

This problem is continued on the following page.

Please be an **absolute champ** and ensure all your answers for all parts of this problem (including those on the following pages) are neatly confined within the given boxes.

General Advice:

As you work through these problems, consider what each loop in *chaosAdd()* and *chaosRemove()* is doing at a high level. That will make this problem go faster than tracing through those loops line-by-line over and over again.

a) Give the vector that would be returned by *createChaosVector*(“*abcdxyz*”). In your response, use curly brace notation. For example, if the vector would contain ‘h’ at index 0 and ‘i’ at index 1, your response would be {‘h’, ‘i’}.

b) Give a string, *str*, that would cause *createChaosVector(str)* to return {‘w’, ‘o’, ‘r’, ‘l’, ‘d’}.

c) What would a call to *causeChaos(“abcdxyz”)* ultimately print to the screen?

d) Give a string, *str*, that would cause *causeChaos(str)*, to print *abcd* to the screen.

e) Suppose we use only *chaosAdd()* and *chaosRemove()* – and no other functions – to add and remove elements from a vector. With respect to the order in which elements are added and removed, will we have effectively implemented a *stack*, a *queue*, or *neither*? In answering this question, focus only on the order in which elements are processed, not the names of the functions or the lack of other functionality, such as the lack of specialized *isEmpty()* or *size()* functions. Select **exactly one** option:

☐ Stack

☐ Queue

☐ Neither

We will not grade anything written outside the boxes on this page.

This problem is continued on the following page.

Notation Requirement:

When answering the following questions, use ***n*** if you need a variable to denote the size of the parameter being passed to the function (the length of the vector or string).

General Advice:

As you work through these problems, consider how the index passed to a vector's *insert()* or *remove()* affects the runtime of those function calls.

f) Using Big-O notation, give the **best-** and **worst-case** runtimes for the *chaosRemove()* function.

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

g) Using Big-O notation, give the **best-** and **worst-case** runtimes for the *createChaosVector()* function.

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

h) Using Big-O notation, give the **best-** and **worst-case** runtimes for the *destroyChaosVector()* function. You may assume that printing a single character to the screen with *cout* is an O(1) operation.

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

i) Using Big-O notation, give the **best-** and **worst-case** runtimes for the *causeChaos()* function.

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

We will not grade anything written in the empty space above.

This problem is continued on the following page.

```
void quirkyAdd(Vector<char>& v, char ch)
{
   Vector<char> reversed;

   while (!v.isEmpty()) {
      reversed.add(v.remove(v.size() - 1));
   }

   // Key Change: This for-loop now happens before we call v.add(ch).
   // Everything else about this function is the same as chaosAdd().
   for (int i = 0; i < reversed.size(); i++) {
      v.add(reversed[i]);
   }


   v.add(ch);
}

char quirkyRemove(Vector<char>& v)
{
   // Key Change: Removing from a different index (of our own choosing).
   char retval = v.remove( ??? );


   // The rest of this function is the same as chaosRemove().
   Vector<char> reversed;

   while (!v.isEmpty()) {
      reversed.insert(0, v.remove(0));
   }

   for (int i = 0; i < reversed.size(); i++) {
      v.add(reversed[i]);
   }

   return retval;
}

Vector<char> quirkyCreate(string str) {
   Vector<char> v;
   for (char ch: str) {
      quirkyAdd(v, ch);
   }
   return v;
}
```

Consider the functions above, which are slight twists on *chaosAdd()*, *chaosRemove()*, and *createChaosVector()*.

j) Give the vector that would be returned by *quirkyCreate*(“*abcdxyz*”). In your response, use curly brace notation. For example, if the vector would contain ‘h’ at index 0 and ‘i’ at index 1, your response would be {‘h’, ‘i’}.

k) Suppose we use only *quirkyAdd()* and *quirkyRemove()* – and no other functions – to add and remove elements from a vector. In order to get the vector to behave like a **queue**, what index do we need to remove from in *quirkyRemove()*?

2. Problem Solving with ADTs (40 pts)

Moira's Teashop of Wonderment and Whimsy serves phenomenal teas with unusual names. They are working on revising their *extensive* menu of teas and would like to be able to quickly replace certain common phrases across all the tea descriptions on their menu. To do that, they have embedded variables in their tea descriptions, but they need your help writing functions that will allow them to quickly update all occurrences of those variables across their menu.

For the purposes of this problem, a "variable" in a tea description is just a string made up entirely of uppercase characters – and possibly one or more underscores – enclosed in angled brackets. For example, in the following tea description, the "variables" are <*OOLONG_ORIGIN*> and <*OOLONG_CAFFEINE*>:

An oolong tea from <OOLONG_ORIGIN>. Full-bodied, sweet, and floral. <OOLONG_CAFFEINE>

a) Your first task is to write a function that takes a vector of strings – each with a "variable," followed by a single hash symbol (#), followed by the text that the teashop wants that variable to be replaced with on their menu – and generates a map of variable strings to replacement text strings. We guarantee each string will contain **exactly one** hash symbol.

For example, suppose your function receives the following vector:

```
{
    "<OOLONG_ORIGIN>#the mountainous region of Nantou, Taiwan",
    "<OOLONG_CAFFEINE>#Caffeine content: medium.",
    "<ANY_TEMP>#Available hot or iced."
}
```

In that case, your function should return the following map:

```
{
    "<ANY_TEMP>" : "Available hot or iced.",
    "<OOLONG_CAFFEINE>" : "Caffeine content: medium.",
    "<OOLONG_ORIGIN>" : "the mountainous region of Nantou, Taiwan"
}
```

We will not grade anything written in the empty space above.

There is space for you to write your function on the following page.

Write your function in the space below, with the following restrictions:

- Your function must be **iterative** (not recursive).
- You must use the function signature given below.
- You cannot write any **helper functions**.
- If a variable occurs more than once in the *varStrings* vector, throw an error. Recall that the syntax for throwing an error is simply *error(msg)*, where *msg* is a string containing an error message.
- You should make appropriate use of library functions to solve this problem. A solution that does so should fit comfortably in the space below.

```
Map<string, string> createVarDescriptionsMap(Vector<string>& varStrings) {

}
```

We will not grade anything outside the box above.

This problem is continued on the following page.

b) Your second task is to write a function that takes in a vector of tea descriptions and builds a map of the indices where any variables occur in those descriptions. For this part of the problem, a single tea description will be represented as a vector of strings, where each string contains a single word, punctuation mark, or variable. Any variables that occur in a tea description are guaranteed to be isolated in their own string within the vector representation of that description. For example, suppose we have the following two tea descriptions:

> "A creamy chocolate rooibos tea. Organic, herbal, caffeine free. <ANY_TEMP>"
> "A full-bodied, sweet, floral oolong. <OOLONG_CAFFEINE> <ANY_TEMP>"

The vector representations of those descriptions would be as follows:

> {"A", "creamy", "rooibos", "tea", ".", "Organic", ",", "herbal", ",", "caffeine", "free", ".", "<ANY_TEMP>"}
> {"A", "full-bodied", ",", "sweet", ",", "floral", "oolong", ".", "<OOLONG_CAFFEINE>", "<ANY_TEMP>"}

Those vectors will be passed to you in a vector of vectors, like so:

```
{
    {"A", "creamy", "rooibos", "tea", ".", "Organic", ",", "herbal", ",", "caffeine", "free", ".", "<ANY_TEMP>"},
    {"A", "full-bodied", ",", "sweet", ",", "floral", "oolong", ".", "<OOLONG_CAFFEINE>", "<ANY_TEMP>"}
}
```

Note that we have the following variable occurrences in those vectors:

- <ANY_TEMP> occurs in vector 0 at index 12.
- <ANY_TEMP> occurs in vector 1 at index 9.
- <OOLONG_CAFFEINE> occurs in vector 1 at index 8.

From this, your goal is to construct a map of type `Map<string, Map<int, int>>`. The ***keys*** in this map are the variable strings that occur within the tea description vectors passed to this function. So, in the example above, our keys will be "<ANY_TEMP>" and "<OOLONG_CAFFEINE>". The ***values*** in the map are of type `Map<int, int>`. In these "sub-maps," the ***keys*** are the indices of the vectors that contain a particular variable, and the ***values*** are the indices within those vectors where the variables occur. The resulting map for the example given above is as follows:

```
{
    "<ANY_TEMP>" :              ← This key maps us to the sub-map on the following lines.
        {                       ← This is the beginning of our sub-map associated with "<ANY_TEMP>"
            0:12,               ← This is one entry in our sub-map, mapping 0 (key) to 12 (value).
            1:9                 ← This is a second entry in our sub-map, mapping 1 (key) to 9 (value).
        }                       ← This is the end of our sub-map associated with "<ANY_TEMP>"

    "<OOLONG_CAFFEINE>" :       ← This key maps us to the sub-map on the following lines.
        {                       ← This is the beginning of our sub-map associated with "<OOLONG_CAFFEINE>"
            1:8                 ← This is the single entry in our sub-map, mapping 1 (key) to 8 (value).
        }                       ← This is the end of our sub-map associated with "<OOLONG_CAFFEINE>"
}
```

A Note About Maps:

You might be wondering about that `Map<string, Map<int, int>>` type. Here's how that works: given a key, *k1* (which must be a string), this map unlocks a sub-map. If we feed the key *k2* (which must be an integer) to that sub-map, we will get the integer that maps to (assuming *k1* and *k2* are valid keys for those maps). For example, in the map shown on the previous page, *map*["*<ANY_TEMP>*"] gives us the first sub-map we see in that diagram, and *map*["*<ANY_TEMP>*"][1] gives us the 9.

Throughout the rest of this problem, trust the *map*[*k1*][*k2*] syntax. If you use it, it will probably work exactly as you expect it to, even for adding items to new or existing maps. Also, recall that *map*[*key*] automatically adds the key to the map – no need for manual initialization. It Just Works™.

We will not grade anything written in the empty space above.

There is space for you to write your function on the following page.

In the box below, write a function that builds a map as described on the previous pages, with the following restrictions:

- Your function must be **iterative** (not recursive).
- You must use the function signature given below.
- You cannot write any **helper functions**.
- You may assume that no tea description contains the same variable multiple times.
- Treat any string that begins with "<" and ends with ">" as a variable; don't waste time checking whether it also contains only uppercase letters and/or underscores.
- You should make appropriate use of library functions to solve this problem. A solution that does so should fit comfortably in the space below.

```
Map<string, Map<int, int>> createVarLocationsMap(Vector<Vector<string>>& teaDescriptions) {

}
```

NOTE!

*You must **either** write your createVarLocationsMap() function in the box below **or** on the previous page – **not both!** We will only grade one of these createVarLocationsMap() boxes. We will default to grading the one on the previous page unless you **boldly cross it out**, in which case we will grade this page instead. The box on this page is not meant to be an overflow space for a solution that does not fit on one page; it is here in case something goes catastrophically wrong on the previous page and you need somewhere else to rewrite your solution from scratch.*

See note above! Do not let your solution from the previous page overflow onto this page!

```
Map<string, Map<int, int>> createVarLocationsMap(Vector<Vector<string>>& teaDescriptions) {
```

c) Your final task for this problem is to use the maps you built above to update a vector of tea descriptions. Write a function that takes a *teaDescriptions* vector, a *varDescriptionsMap* constructed by the function you wrote in part (a) of this problem, and a *varLocationsMap* constructed by the function you wrote in part (b), and updates the tea descriptions accordingly. The function signature is as follows:

```
void updateVariables(Vector<Vector<string>>& teaDescriptions,
                     Map<string, string>& varDescriptionsMap,
                     Map<string, Map<int, int>>& varLocationsMap);
```

The parameters are as follows:

- `Vector<Vector<string>>& teaDescriptions`

  A vector of tea descriptions, as described in part (b) of this problem. For example:

  ```
  {
    {“A”, “creamy”, “rooibos”, “tea”, “.”, “Organic”, “,”, “herbal”, “,”, “caffeine”, “free”, “.”, “<ANY_TEMP>”},
    {“A”, “full-bodied”, “,”, “sweet”, “,”, “floral”, “oolong”, “.”, “<OOLONG_CAFFEINE>”, “<ANY_TEMP>”}
  }
  ```

- `Map<string, string>& varDescriptionsMap`

  A map produced by the *createVarDescriptionsMap()* function from part (a). Each key is a (unique) variable string, and each value is a string we will use to replace all occurrences of that variable in our tea descriptions. For example:

  ```
  {
    “<ANY_TEMP>” : “Available hot or iced.”,
    “<OOLONG_CAFFEINE>” : “Caffeine content: medium.”,
    “<OOLONG_ORIGIN>” : “the mountainous region of Nantou, Taiwan”
  }
  ```

- `Map<string, Map<int, int>>& varLocationsMap`

  A map produced by the *createVarLocationsMap()* function from part (b). Each key is a (unique) variable string, and each value is a sub-map. In each sub-map, each key is the index of some vector in our *teaDescriptions* vector that contains the given variable string, and each value is the specific index within that vector where the variable occurs. For example:

  ```
  {
    “<ANY_TEMP>” : { 0:12, 1:9 },
    “<OOLONG_CAFFEINE>” : { 1:8}
  }
  ```

After calling this function with the example parameters above, our *teaDescriptions* would become:

```
{
  {“A”, “creamy”, “rooibos”, “tea”, “.”, “Organic”, “,”, “herbal”, “,”, “caffeine”, “free”, “.”, “Available hot or iced.”},
  {“A”, “full-bodied”, “,”, “sweet”, “,”, “floral”, “oolong”, “.”, “Caffeine content: medium.”, “Available hot or iced.”}
}
```

Notice that after updating our tea descriptions, some strings will contain multiple words. That's fine.

Additional Notes and Requirements:

- Your *updateVariables()* function must **not** iterate over or search through the *teaDescriptions* vector. Instead, it should rely on the *varLocationsMap* to tell it what indices to go to when modifying strings in the *teaDescriptions* vector (and its sub-vectors). (Although you may not iterate over *teaDescriptions*, you may iterate over other variables if needed.)

- Your *updateVariables()* function should assume *varLocationsMap* contains valid indices. Do not check whether those indices are out of bounds. However, you should check before overwriting a string that it actually matches the variable that *varLocationsMap* indicates we should be overwriting at that particular location. If not, throw an error. For example, using the *teaDescriptions* on the previous page, if *varLocationsMap* told us that "<ANY_TEMP>" could be found in vector 0 at index 3, we would throw an error, because that string is "tea", not "<ANY_TEMP>".

- It's fine if *varDescriptionsMap* contains keys that are not in *varLocationsMap*. However, if *varLocationsMap* contains a key that is not in *varDescriptionsMap*, you should throw an error.

We will not grade anything written in the empty space above.
There is space for you to write your function on the following page.

Write your function in the space below, with the following restrictions:

- Your function must be **iterative** (not recursive).
- You must use the function signature given below.
- You cannot write any **helper functions**.
- You should make appropriate use of our library functions to solve this problem. A solution that does so should not require all the space provided below.

```
void updateVariables(Vector<Vector<string>>& teaDescriptions,
                     Map<string, string>& varDescriptionsMap,
                     Map<string, Map<int, int>>& varLocationsMap) {
```

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.

The exam continues on the following page.

3. Recursive Coding (20 pts)

Write a function that takes a string (*str*) and an integer (*k*) and prints all permutations of length *k* that can be made of the characters in *str* and also returns the total number of strings printed. The function signature is as follows:

```
int kPermute(string str, int k)
```

For example, *kPermute("ab", 3)* should print the following 8 strings (and should return 8 when finished):

aaa
aab
aba
abb
baa
bab
bba
bbb

You may assume there are no repeated characters in *str* and that $k \geq 0$. If *str* is empty and $k > 0$, you should throw an error, since we cannot make any strings of length $k > 0$ if there are no characters to choose from.

For your reference, here is the *permute()* function we saw in class. Read over this code as a starting point. You can borrow code and structure from *permute()* when writing your *kPermute()* function.

```
void permute(string soFar, string rest)
{
   if (rest == "")
   {
      cout << soFar << endl;
      return;
   }

   for (int i = 0; i < rest.length(); i++)
   {
      string newRest = rest.substr(0, i) + rest.substr(i + 1);
      permute(soFar + rest[i], newRest);
   }
}

void permute(string str)
{
   permute("", str);
}
```

We will not grade anything written on this page.
There is space for you to write your function on the following page.

Write your function in the space below, with the following restrictions:

- Your function must be **recursive** to earn credit.
- You must use the function signature given below for the initial call to this function.
- You **can** (and should) write at least one helper function.
- Please pass all parameters **by value**, even in your helper function.
- Don't forget this function is supposed to return an integer.
- Don't worry about the order of the function definitions (i.e., even though your helper function will go below the definition of *kPermute()*, you don't need a functional prototype).

```
int kPermute(string str, int k) {

}
```

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.