Homework 6 – Spring 2023 (Due: May 17, 2023)

In this homework, you'll explore more concepts from function palooza: returning results, error handling, and a bit about parameter passing.. Some questions have multiple, distinct answers which would be acceptable, so there might not be a "right" answer: what's important is your ability to justify your answer with clear and concise reasoning that utilizes the appropriate terminology discussed in class. Each question has a time estimate; you'll know you're ready for the exam when you can solve them roughly within their time constraints.

We understand, however, that as you learn these questions may take more time. For that reason, only **starred red** questions need to be completed when you submit this homework (the rest should be used as exam prep materials). Note that for some multi-part questions, not all parts are marked as red so you may skip unstarred subparts in this homework.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and handwritten solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work.

1. ** Consider the following program that looks suspiciously like Python (but we promise you, it isn't!):

```
def foo(a):
  a = 3
  bar(a, baz())

def bar(a, b):
  print("bar")
  a = a + 1

def baz():
  print("baz")
  return 5

a = 1
foo(a)
print(a)
```

Assume that in this language, formal parameters are mutable.

a) ** (2 min.) Suppose you know this language has pass-by-value semantics. What would this program print out?

b) ** (2 min.) Suppose you know this language has pass-by-reference semantics. What would this program print out?

c) ** (2 min.) Suppose you know this language has pass-by-object reference semantics. What would this program print out?

d) ** (2 min.) Suppose you know this language has pass-by-need semantics. What would this program print out?

2. ** (10 min.) Consider the following C++ struct:

```
template <typename T>
struct Optional {
  T *value;
};
```

If `value` is `nullptr`, then we interpret the optional as a failure result. Otherwise, we interpret the optional as having some value (which is pointed to by `value`).

Next, consider two different implementations of a function that finds the first index of a given element in an `int` array:

```
Optional<int> firstIndexA(int arr[], int size, int n) {
  for (int i = 0; i < size; i++) {
    if (arr[i] == n)
      return Optional<int> { new int(i) };
  }
  return Optional<int> { nullptr };
}
```

```
int firstIndexB(int arr[], int size, int n) {
  for (int i = 0; i < size; i++) {
    if (arr[i] == n)
      return i;
  }
  throw std::exception();
}
```

Compare our generic `Optional` struct with C++'s native exception handling (throwing errors). Discuss the tradeoffs between each approach, and the different responsibilities that consumers of either API must adopt to ensure their program can handle a potential failure (i.e. element not found). Also discuss which approach is more suitable for this use case, and why.

3. **For this problem, you'll need to consult C++'s exception hierarchy. Consider the following functions which uses C++ exceptions:

```
void foo(int x) {
 try {
   try {
     switch (x) {
      case 0:
       throw range_error("out of range error");
      case 1:
       throw invalid_argument("invalid_argument");
      case 2:
       throw logic_error("invalid_argument");
      case 3:
       throw bad_exception();
      case 4:
       break;
     }
   }
   catch (logic_error& le) {
     cout << "catch 1\n";
   }
   cout << "hurray!\n";
 }
 catch (runtime_error& re) {
   cout << "catch 2\n";
 }
 cout << "I'm done!\n";
}
```

```
void bar(int x) {
  try {
    foo(x);
    cout << "that's what I say\n";
  }
  catch (exception& e) {
   cout << "catch 3\n";
   return;
  }
  cout << "Really done!\n";
}
```

Without running this code, try to figure out the output for each of the following calls to bar():

a) ** (2 min.) `bar(0);`

b) **(2 min.) `bar(1);`

c) (2 min.) `bar(2);`

d) **(2 min.) `bar(3);`

e) (2 min.) `bar(4);`

4. For this problem, you are to write two versions of a container class called *Kontainer* that holds up to 100 elements, and allows the user to add a new element as well as find/return the minimum element, regardless of the data type of the elements (e.g., integers, floating-point numbers, strings, etc.). Don't worry about the user overflowing the container with items. You will be implementing this class using templates in C++ *and* generics in a statically-typed language of your choice (e.g., Java, Swift, Kotlin, Go, Haskell, etc.).

a) Show your implementation of the Kontainer class using C++ templates:

b) Show your implementation of the Kontainer class using generics in a statically-typed language of your choice - feel free to use any online resources you like if you choose a different language for your solution:

Hint: If you haven't programmed in another statically-typed language other than C++ before, there are tons of online examples and online compilers which make this easy! Here are links to a few online compilers: Kotlin, Java, Swift.

c) Try creating a Kontainer with a bunch of doubles and Strings. Show your code here:

d) Highlight the key advantages and disadvantages of each approach based on your actual coding experience while solving a-c above. What was easy, what was difficult? What things did you run into that you wouldn't have expected.