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

In this homework, you'll explore more concepts from function palooza: returning results, error handling, first class functions, lambdas/closures and capture strategies, and polymorphism (ad-hoc, parametric - templates and generics). We'll also explore OOP topics from this week's lectures . 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?

baz
bar
1

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

baz
bar
4

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

baz
bar
1

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

This one is tricky! Note that bar doesn't return anything, and thus foo never "uses" bar's value, so it never gets called. And, foo doesn't return anything either – so we wouldn't even call foo! Implicitly, we're assuming that we determine "need" with function return values.

bar
1

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.

There are a few different concepts to consider:

Recoverability. There is no way that invoking `firstIndexA`, by itself, could crash your program. However, since `firstIndexB` throws an exception, your program would crash if you neglected to handle the error properly (i.e. using a catch statement). Thus, you could argue that `firstIndexA` is more appropriate for programs where crashing must be avoided at all costs.

Incentive to handle errors. Counter to the point made in the previous paragraph, because `firstIndexA` cannot crash your program, its users may be tempted to ignore the case in which an error occurs (otherwise known as gracefully failing). However, `firstIndexB` *forces* you to explicitly handle the error (otherwise your program would crash). From this perspective, throwing exceptions may be better from a code quality standpoint.

Overall, however, we'd generally argue that `firstIndexA` is more appropriate for this use case. It is not unreasonable to assume that consumers of this API would want to use this function to check if an element exists in an array. It is also not unreasonable to assume that the element we are searching for may not exist in the array. As a result, potentially subjecting your program to a crash (even if it may force clients to properly handle error states) is probably a bit too heavy-handed. It makes more sense to return an `Optional` and handle the case

when it is `nullptr`.

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():

The important thing to understand here is that when you throw an exception, it will be caught by a catch clause IFF the catch clause specifies either the same exception name (e.g., I throw a `logic_error`, and I catch a `logic_error` as shown in the `foo()` function), or if your catch specifies a superclass of the thrown exception (e.g., I throw a `range_error` and we catch a `runtime_error`, which is a superclass of `range_error`). If a catch block attempts to catch an unrelated exception, then it will be ignored, and the current function will be terminated. This will continue until a compatible catch addresses the exception, or the program terminates.

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

```
catch 2
I'm done!
that's what I say
```

```
Really done!
```

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

```
catch 1
hurray!
I'm done!
that's what I say
Really done!
```

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

```
catch 1
hurray!
I'm done!
that's what I say
Really done!
```

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

```
catch 3
```

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

```
hurray!
I'm done!
that's what I say
Really done!
```

4. (10 min) 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:

```
template <typename T>
class Kontainer {
private:
    T elements_[100];
    int count_;

public:
    Kontainer() : count_(0) {}

    // don't worry about overflowing, as the spec says
    void add_element(const T& element) {
      elements_[count_++] = element;
    }

    // assumes there's at least one element
    T get_min() const {
        T min = elements_[0];
        for (int i = 1; i < count_; i++) {
            if (elements_[i] < min) {
                min = elements_[i];
            }
        }

        return min;
    }
};
```

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.

In Matt's slides, he has solutions in Haskell, Rust, and TypeScript.

```
// Here's a solution in C# (Generated by ChatGPT)
// Note: this is a bounded generic; it requires any type used with
// it to support a comparison interface (IComparable)
public class Kontainer<T> where T : IComparable<T>
{
    private T[] elements;
    private int count;

    public Kontainer() {
        elements = new T[100];
        count = 0;
    }

    public void add_element(T element) {
        elements[count++] = element;
    }

    public T get_min() {
        T min = elements[0];
        for (int i = 1; i < count; i++) {
            // the CompareTo() method is part of
            // the IComparable interface
            if (elements[i].CompareTo(min) < 0) {
                min = elements[i];
            }
        }

        return min;
    }
}
```

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

```
// C# Kontainer (generated by ChatGPT!)
class Program
{
    static void Main(string[] args)
    {
        Kontainer<double> doubleContainer = new Kontainer<double>();
        doubleContainer.add_element(3.14);
        doubleContainer.add_element(2.718);
        doubleContainer.add_element(1.618);
        doubleContainer.add_element(0.577);
        Console.WriteLine("Minimum element: " +
                          doubleContainer.get_min());

        Kontainer<string> stringContainer = new Kontainer<string>();
        stringContainer.add_element("apple");
        stringContainer.add_element("banana");
        stringContainer.add_element("carrot");
        stringContainer.add_element("avocado");
        Console.WriteLine("Minimum element: " +
                          stringContainer.get_min());
    }
}
```

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.

This is subjective, and up to each student.