**Homework 6 Solutions**

This homework covers the following topics from class:

- Function palooza, part 2
    - More on Parameters (Positional vs. Named Parameters, Optional Parameters and Default Parameters, Variadic Functions), Returning Values and Error Handling (error objects, optionals, assertions/invariants)
- Function palooza, part 3
    - Error Handling Part 2 (Exceptions, Panics), First-class Functions (Lambdas/Closures Across Languages)

## DATA9: Binding Semantics, Parameter Passing, Pass By Need (8 min)

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.

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

**Solution:**

**baz**
**bar**
**1**

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

**Solution:**

**baz**
**bar**
**4**

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

**Solution:**

**baz**
**bar**
**1**

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

**Solution:**

This one is tricky! For this problem (and in this class), we define "need" by whether or not we use the return value of the function. The core observation is that we never use the return value of baz() in bar() (as bar() ignores its second argument). So, in a pass-by-need semantics language (like Haskell), we'd never call baz().

You might wonder why we call foo() if we don't use its return value! In this case, we're *not* using parameter passing semantics at all, so whether or not it's "needed" is not a concern. But, if this function *also* used need semantics for variable binding, etc. – then, we woulnd't use foo() or bar() (since we use none of their return values).

**bar**
**1**

## FUNC1: Default Parameters (5 min)

(5 min.) Khoi has decided to create a new programming language called Lit and allow functions to have default parameters in any position (not just for the last N arguments as in C++ and Python). For example, he says he wants this to be a valid function definition:

```
func addNumbers(x: Int = 10, y: Int, z: Int = 40) -> Int {
  return x + y + z
}
```

He knows that if he uses this approach, there may be ambiguities, e.g.:

```
func main() {
  result := addNumbers(20, 30) // is this x=20, y=30, z=40 or x=10, y=20, z=30?
```

```
}
```

So he needs some advice.  He wants to know what to require for function calls to ensure there's no ambiguity as to which arguments go to which parameters.  How would you design function calls to eliminate ambiguity as to which arguments are being passed to which formal parameters? For example, if you required all arguments to be named, this would eliminate any ambiguity:

```
result := addNumbers(y: 20, z: 30)
```

Give at least two additional approaches that might work. Be creative, there's no right answer!

**Solution:**

One solution would be to have an approach where arguments are first bound to non-default parameters in the order they are provided. After all non-default parameters are filled, the remaining arguments could be associated with parameters after the last non-default parameter. For example:

```
result := addNumbers(20, 30)
```

In this case, 20 would be associated with the first non-default formal parameter of y, then 30 would fill in parameter z.

Another solution would be to introduce a special syntax for specifying when a default value should be used explicitly, such as an underscore `_` or a keyword like `default`. For example:

```
result := addNumbers(_, 30, 40)
```

or something like this:

```
result := addNumbers(, 30, 40)
```

This would clearly indicate that the first argument should use its default value, but the third argument should be replaced.

## FUNC2: Lambdas, Closures (5 min)

(5 min.) Consider the follow JavaScript program which passes a lambda to the *callLambda* function:

```
function callLambda(action) {
  action();
  action();
}

function main() {
  let counter = 0;

  // Pass a lambda to another function
  callLambda(() => {
    counter++;
    console.log(`Counter: ${counter}`);
  });

  console.log(`Final Counter in main: ${counter}`);
}

main();
```

This code outputs:

```
Counter: 1
Counter: 2
Final Counter in main: 2
```

What must be happening in JavaScript to enable this functionality? How can the calls to the lambda expression (e.g., action()) possibly modify variables defined in main? Specifically, explain how you think JavaScript closures work to enable this behavior.

Solution:

When the lambda function is defined in the main function, it captures the surrounding lexical environment, which includes the variable counter. This captured environment is retained by the lambda even when it is passed to and executed inside the callLambda function.

Furthermore, the counter variable is captured by reference, not by value. This means that any modification to counter variable within the lambda affects the counter variable in the main function (they're the same variable!). When callLambda invokes the lambda (action()), the lambda increments counter and prints its value.

The lambda retains access to the counter variable in main throughout the lambda's lifetime. Therefore, when action() is called multiple times in callLambda, it continues to modify main's same counter variable, resulting in the printed values of 1 and 2, and finally 2 as the value of counter in main.

## FUNC3: Error Handling, Optionals, Exceptions (10 min)

(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.

**Solution:**

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`.

## FUNC4: Results, Optionals, Errors, Exceptions (8 min)

In this problem, you will choose the most appropriate error handling mechanism for each scenario and give a brief explanation justifying your choice.

For each problem, you may choose from:

- Error Objects
- Status Objects
- Result Objects
- Assertions
- Exceptions
- Panics

Part A: You have been asked to write a function that accepts a URL, validates its format, and if it is properly formatted, extracts and returns its domain. It's unimportant to know why the URL was malformed. What error handling mechanism would you choose and why?

Solution:

Since it's unimportant to know why the URL is malformed, a status object is appropriate. The function can return a simple success or failure status, with the success path returning the domain and the failure path indicating that the URL was invalid without any need for detailed error information. This approach keeps the implementation lightweight and avoids the complexity of dealing with detailed error messages.

Part B: You are building a module to load the configuration data for the flight control software that operates a rocket engine. If the configuration is invalid the rocket will fail catastrophically. What error handling mechanism would you choose and why?

Solution:

In this case, the severity of the failure (a rocket failing catastrophically) suggests that panicking is appropriate. A panic ensures that the program halts immediately, as continuing with invalid configuration data would have dire consequences. It also signals a critical failure that should never occur during normal operation (and which should be investigated immediately).

Part C: You are building an internal API that will only be called by other software components you've built and will not accessed by external users. The API can only operate correctly on inputs of size 1 to 1000 elements, with other sizes causing the API to result in undefined behavior. What error handling mechanism would you choose and why?

Solution:

Since this is an internal API, assertions are a good choice to ensure that only valid inputs are passed in. Passing an invalid input to such a component suggests a logic error in the calling component and should immediately terminate the program. An assertion would immediately catch such mistakes during testing and debugging, preventing them from reaching production.

Part D: You are building a database engine which reads and writes to cloud storage systems. These cloud storage systems have a 99.9% uptime but do occasionally fail, resulting in the transaction having to be repeated from scratch. What error handling mechanism would you choose and why?

Solution:

Exceptions are suitable because the cloud storage failure is an external factor that can happen intermittently. Exceptions allow the error to be caught and the transaction retried without terminating the program. This mechanism is ideal for rare transient issues that require recovery rather than immediate program termination.

## FUNC5: Exceptions (10 min)

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";
}
```

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

```
bar(0);

bar(1);
```

```
    bar(2);

    bar(3);

    bar(4);
```

**Solution:**

**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.**

**`bar(0)` produces:**

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

**`bar(1)` produces:**

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

**`bar(2)` produces:**

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

**bar(3) produces:**

```
      catch 3
```

**bar(4) produces:**

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