Homework 6 with Solutions – Fall 2022 (Due: Nov. 23, 2022)

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?

```
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. **Consider the following C++ program:

```
// Interface for a basic shape in C++ using pure virtual functions
class IShape {
  virtual int get_area() const = 0;
};

// Rectangle class which implements the shape interface
class Rectangle: public IShape {
  Rectangle(int width, int height) { ... }
  int get_area() const { ... }
};

void operate_on_shape(IShape &s) {          // compiles fine
  cout << "the shape's area is: " << s.area();
}

int main() {
  Rectangle r(10,20);       // compiles fine
  IShape *s;                // compiles fine
  IShape s;                 // fails to compile.
}
```

a) **(5 min.) In class, we learned that when you define a class X, it also defines a new type of the same name X, and that when you define an interface I, it also defines a new type I.

Class types can be used to define objects, object references/pointers, and references. But interface types **cannot** be used to define objects - they only can be used to define object references/pointers and references.

Explain why we can't define concrete objects using an interface type, but can use interface types to define object references, references and pointers.

By definition, an interface is just a set of function prototypes/declarations, and does NOT include actual { bodies } of code. Therefore it's impossible to instantiate an object with an interface type, since if I did do so and tried to call one of its functions (like `get_area()`), there would be no function { body } to call! On the other hand, if I have a class that implements all of the functions of an interface, then this is a concrete class, and I may instantiate objects with it.

b) **(5 min.) Does the concept of interfaces make sense in dynamically typed languages? Explain why or why not. If so, find one use-case where interfaces make sense. If not, explain how dynamically typed languages accomplish similar functionality without interfaces.

No - interfaces are not used in dynamically-typed languages. They are only used in statically type languages, where we want to define a function that can operate on objects that implement a particular interface, e.g.:

```
void operate_on_shape(IShape &shape) { … }
```

The above function indicates that it can operate on any object that implements the `IShape` interface. Since variables/parameters don't have types in dynamically-typed languages, there's no way to specify that a parameter must implement a particular interface like IShape. In dynamically typed languages, we simply use duck typing to accomplish the same thing.

c) **(7 min.) When would we want to use interface inheritance rather than traditional hybrid inheritance (which inherits both an implementation and an interface)? Give at least two motivating use cases.

There are many examples, but here are a few:

1. Enabling an object to be comparable to other objects by having it implement an `ICompare`-style interface.

2. Enabling a container object to support iteration over the items it holds.

5. This problem focuses on class variables and methods. Consider the following Python program which defines a class that contains a class variable (`shared_count`).

```
class Foo():
  def initial_value():
    print("generating initial value")
    return 42

  shared_count = initial_value()   # initialize class variable

  def inc(self):
   Foo.shared_count += 1

print("creating a and b")
a = Foo()
b = Foo()
a.inc()
print(a.shared_count)
print(b.shared_count)
```

a) (3 min.) First, without running this program, what would you expect this program to print out? If you're unsure of what would be printed, run the program and find out.

We'd expect it to print 43 and 43.

b) (5 min.) Give an explanation of why the output is the way it is. Specifically, focus on what actions happen during creation of the class, vs. what actions happen during the creation of an object. Use this reasoning to explain why/when/how many times initial_value() is called. What can you conclude about the initialization of class variables?

Since `shared_count` is a class-variable, a single copy of the variable is created and initialized once when the class is first created. This is why the "generating initial value" output is produced just once when the class is defined, *before* we print out "creating a and b." Later when we increment our class variable, this changes the single variable that is shared by all of the class instances (a and b, in this example). So when we print out the values of `a.shared_count` and `b.shared_count`, they're the same.

c) (4 min.) Why is an instance function able to access the members of an object but a class/static function is not?

A class/static function has no access to individual objects. All it knows about is the overall class and any class-level variables and class-level methods that are general to the class. An instance method, on the other hand, is always given access to a specific object with the `this`/`self` pointer/object reference so it can operate on that object. Class/static functions have no such argument that lets them operate on a particular object.

6. (5 min.) Consider the code below.

```
class Cat {
public:
  Cat() { what_do_cats_say_ = "Meeeeooooowwww"; }
  void talk() {
     cout << "I say: " << this->what_do_cats_say_;
  }
private:
  string what_do_cats_say_;
};

Cat tom, ferguson;
tom.talk();       // this in the talk() method refers to "tom"
ferguson.talk(); // this in the talk() method refers to
"ferguson"
```

When you call an instance method (e.g., tom.talk()), somehow the body of the called method has a binding between the **this/self** variable and the original object (tom). How does the this/self object reference or pointer get into the member function from the point of the function call?

Hint: Think a bit about how the language actually might implement this - does it set a global variable called "this" to the address of the object (tom)? Something else?

When you do a call like this:

```
tom.talk()
```

This is taking the pointer/object reference to the value referred to by the tom variable, and sending it as an (often implicit) first parameter to the called method. So here's what would be happening:

```
talk(&tom);  // tom's address is secretly passed to talk()
```

The talk method similarly has a hidden first parameter called this:

```
void talk(Cat *this) { … }
```

In some languages like Python and Go, the parameter is explicitly named (e.g. "self"), but in other languages like C++ it's implicit.

7. **(5 min.) Joi wants to build a stack of integers class, basing it on a C++ vector, but is not sure whether she should **inherit** from vector and add new stack-specific functions:

```
class Stack: public vector<int> {
public:
     void push_top(int val) { ... }
     int pop_top() { ... }
};
```

Or use **composition** with **delegation**:

```
class Stack {
public:
  void push_top(int val) { ... }
  int pop_top() { ... }
private:
  public vector<int>  items_;
};
```

Which approach is better? Explain why.

Composition with delegation would be the preferred approach. We don't expect or want a `Stack` to support all of the methods that a `vector` supports. Many of them (e.g., indexing a specific item in the vector) make no sense for a `Stack`. So the interface of a vector is larger and non-overlapping with that of a `Stack`. As such, we'd want to have a `Stack` implement its own public interface, and use a vector for its implementation, without exposing the vector's interface. This would be done with composition (defining a member variable of type vector) and delegation (calling vector's methods from within stack's methods).

8. **

a) **(5 min.) Explain why a class that implements multiple interfaces doesn't suffer from the same problems as a class that uses multiple subtype/hybrid inheritance.

Since an interface has no code associated with it, a class can implement multiple interfaces and there will be no chance that duplicated member data or function bodies will be inherited and shared (i.e., creating a diamond pattern).

b) (5 min.) What happens if a class like C below implements two different interfaces that define the same exact function prototype? Does this pose a problem? If so, why? If not, why not?

```
interface A {
  void foo(int bar);
}

interface B {
  void foo(int bar);
}

class C implements A, B {
  ...
}
```

This does not pose a problem. Since both foo() functions have the same interface (return type and parameters/types), even though the C class inherits the function prototype twice - once from A and once from B, you'd only provide a single implementation of the function inside of class C, e.g.:

```
class C {
  void foo(int bar) { /* only one of these, not two */ }
};
```

So the single foo() implementation in class C would satisfy both interfaces.

9. **

a) **(8 min.) C++ and C# chose to use static method binding by default. In other words, to get dynamic binding functionality you have to explicitly declare a method as "virtual" - otherwise it defaults to statically bound. In contrast, languages like Java and Python only support dynamic binding (all methods are virtual). What are the pros and cons of each approach?

Default is no virtual (like C++):

Pros: Improves performance, since we don't have to deal with expensive dynamic binding and indirect function-pointer-based calls through v-tables.

Cons: May result in bugs when a non-virtual method is overridden, resulting in incorrect behavior when polymorphism/dynamic binding is used.

Default is all virtual (like Java):

Pros: Reduces the likelihood of bugs, since every method is virtual and when overridden, will work with dynamic binding.

Cons: Slows down the program, since every method call must use a vtable (or similar structure) to determine what method to call.

b) **(6 min.) Consider the following C++ class:

```
class Dog {
public:
  virtual void bark() { ... }
  void sit() { ... }
  static void fetch() { ... }
};
```

Which of the above methods, if any, would be referenced in C++'s vtable. Why?

Vtables only include pointers to virtual functions. They do not include instance methods or static methods. And if we did this:

```
class Retriever: public Dog {
public:
  void bark() { ... }
  void sit() { ... }
  static void fetch() { ... }
};
```

Even though we left out the "virtual" keyword for `bark()`, the method is still virtual because it was marked virtual in the base class. So `bark()` in `Retriever` would be placed in `Retriever`'s vtable.