Homework 6 – 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?

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:

```cpp
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:

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

```cpp
int firstIndexB(int arr[], int size, int n) {
  for (int i = 0; i < n; 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. **Consider the following C++ program:

```cpp
// 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.

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.

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.

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.

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?

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

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

```
class Cat {
public:
  Cat() { what_do_cats_say_ = "Meeeeoooowwww"; }
  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?

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.

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.

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 {
  ...
}
```

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?

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?