Homework 7 – Spring 2023 (Due: May 24, 2023)

In this homework, you'll continue to explore parametric polymorphism as well as some of the fundamental OOP concepts.

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. ** (10 min.) Discuss the benefits and drawbacks of (a) the template approach, (b) the generics approach, and (c) the duck typing approach for implementing generic functions/classes. Consider such things as: performance, compilation time, clarity of errors, code size, and type-checking (when does it occur - at runtime or compile-time). If you were designing a new language and had to pick one approach, which one would you choose and why?

2. ** (5 min.) Explain why dynamically-typed languages can't support *parametric polymorphism*.

3. ** (5 min.) Explain how we can accomplish something like duck-typing in statically-typed languages that use templates. How is this similar to duck typing in dynamically typed languages, and how is it different (e.g., are there any pros/cons of either approach)?

4. (10 min.) Consider the following C++ container class which can hold pointers to many types of values:

```
class Holder {
private:
  static const int MAX = 10;
  void *arr_[MAX]; //  array of void pointers to objs/values
  int count_;
public:
  Holder() {
   count_ = 0;
  }
  void add(void *ptr) {
   if (count_ >= MAX) return;
   arr_[count_++] = ptr;
  }
  void* get(int pos) const {
   if (pos >= count_) return nullptr;
   return arr_[pos];
  }
};

int main() {
 Holder h;
 string s = "hi";
 int i = 5;
 h.add(&s);
 h.add(&i);

 // get the values from the container
 std::string* ps = (std::string *)h.get(0);
 cout << *ps << std::endl;  // prints: hi
 int* pi = (int *)h.get(1);
 cout << *pi << std::endl;  // prints: 5
}
```

This class does not use C++ templates, yet by using void pointers (void *) which are a generic type of pointer that can legally point to any type of value, it allows us to store a variety of (pointers to) values in our container object. This is how we used to do things before C++ added templates.

a. Discuss the pros and cons of the approach shown above vs. C++ templates.

b. How is this approach similar or different to generics in languages like Java?

c. What change might we make to the above program to make it work more like a bounded generic class?

5. ** (5 min.) As we learned in class, some of the original OOP languages didn't support classes, just objects. As we know, classes serve as "blueprints" for creating new objects, enabling us to easily create many like objects with the same set of methods and fields. In a language without classes, like JavaScript, how would you go about creating uniform sets of objects that all have the same methods/fields?

6. (5 min.) By convention, member variables in Python objects are typically accessed directly without using accessor or mutator methods (i.e., getters and setters) - even by code external to a class. What impact does this have on encapsulation and why might Python have chosen this approach? Are there ever cases when you should use getters/setters in Python rather than allowing direct access to member variables? If so, when?

7. ** (5 min.) We learned in class that when we define a new interface X, this also results in the creation of a new *type* called X. So in the following code, the definition of the abstract IShape class (C++'s equivalent of an interface) defines a new type called IShape:

```
// IShape is an abstract base class in C++, representing an interface
class IShape {
public:
  virtual double get_area() const = 0;
  virtual double get_perimeter() const = 0;
};
```

Now consider the code below:

```
int main() {
 IShape *ptr;  // works!

 IShape x;     // does it work?
}
```

Can the *IShape* type be used to instantiate concrete objects like x or only pointers like ptr? Why?

8. ** (5 min.) Why aren't interfaces used/needed in dynamically-typed languages?

9. (5 min.) In Java, the *protected* keyword has a different meaning than in languages like C++. Here's an example of two unrelated classes that are defined in the same "package" (a package is in some ways like a namespace in C++):

foo.java

```
package edu.ucla.protected_example;

class Foo {
    protected int myProtectedVariable;

    public Foo() {
        myProtectedVariable = 42;
    }
}
```

bar.java

```
package edu.ucla.protected_example;

class Bar {
    public void accessProtectedVariable(Foo f) {
        System.out.println("Accesses Foo's protected var: " +
                           f.myProtectedVariable);
    }
}
```

Inspecting the above code, we can see that the Bar class is able to access the protected members of the Foo class, even though the two classes are unrelated (except by virtue of the fact that they are part of the same package). How does this differ from the semantics of the *protected* keyword in C++, and what are the pros and cons of Java's approach to *protected* relative to C++?

10. ** (5 min.) Give an example of where you'd prefer interface inheritance instead of traditional subtype inheritance. Why? Give an example of when you'd want to do the opposite.