Homework 7 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

Templates allow compile-time type checking of functions or classes that instantiate the template with a specific type(s). The drawback however is that because a concrete version of the template is created for each type combination used, the resulting binary is bloated with lots of code repetition, fully fledged function/class definitions for *each* instantiation.

On the other hand, generics eliminate this code repetition by having code that uses them reference the same generic definition. However, this means that the generic code cannot make any assumptions about what types may be used with it, so its type checking is hyper-conservative, raising compiler errors for code that programmers can know is safe but cannot be deduced likewise by the compiler. This is solved by using **bound types**, giving the generic more flexibility (and readability) but at the cost of having to define a separate **interface(s)** to bind to.

Duck typing is different from both in that it's strictly a runtime concept. Templates and generics still guarantee type safety because all necessary checks are run at compile-time. By the nature of duck typing, it is impossible to know whether an operand(s) supports a certain operation until the runtime environment checks for it when the operation is attempted. Duck typing is the most flexible of the three and does not add code bloat/complexity but at the cost of looser type safety and runtime bugs that may be harder to diagnose.

If I were designing a language, I would consider the class of problems my language aims to solve. If it's meant to be a scripting language or a language geared towards fast development time, a more flexible approach like duck typing can make it easier to get code up and running. If my language is more geared towards robust applications for which bugs are much less forgiving, I would use templates or generics. The choice *between* templates and generics is a hard one, but I think it comes down to factors such as:

- Templates being more flexible at compile time but taking more time to compile (and larger executable size) due to the creation of concrete definitions for each instantiation. However, after this step, the templated code is treated as ordinary code with no indirection or abstraction, so the performance is comparable to if the code weren't templated in the first place. This is good for software where performance is the utmost priority.
- Generics being arguably more readable and faster at compile time (and more OOP-centric, supporting bound types with **interfaces**), but there is a runtime cost as there is a necessary level of indirection where code that uses the generic must jump around to the generic code in memory as opposed to having their own definitions. This can be better for software that needs to be maintainable and easily extendible (such as one that has a lot of updates).

Question 2

Dynamically typed languages cannot support parametric polymorphism because variables do not have type information associated with them; rather, they are simply symbols bound to values in memory that DO have

type information. Thus, it doesn't make sense for a dynamically typed language to adhere to a template/generic because if a variable needed to be a certain type, it can be assigned to a value of that type without any checks anyway. Some such languages like Python support **type hints** to improve developer workflow and integrate with linters, but these are ignored by the interpreter and do not affect execution.

Question 3

It wouldn't be true duck typing, but you can accomplish a similar effect with templates by wrapping the method to be duck typed (the operation to "check for" in the operand(s)) as a templated function. Then, classes that support that method (by defining it themselves) can be passed into the wrapper that then calls the method on its behalf. This is similar to duck typing in dynamically typed languages because the compile-time type checking will validate that an object passed into the wrapper actually defines the duck-typed method and raises an error otherwise, and because its parameterized, the wrapper can accept *any* class type as long as it implements the wrapped method.

```
#include <iostream>
using namespace std;

template <typename T>
void callQuack(T const &obj)
{
    obj.quack();
}

class Duck
{
public:
    void quack() const { cout << "Quack!\n"; };
};

class DuckPerson
{
public:
    void quack() const { cout << "I say quack!\n"; };
};

class Goose
{
public:
    void honk() const { cout << "Honk!\n"; };
};

int main()
{
    Duck d;
    DuckPerson p;
    Goose g;
    callQuack(d); // Quack!
    callQuack(p); // I say quack!
    // Compiler error if not commented out:
    // error: 'const class Goose' has no member named 'quack'
```

```
    callQuack(g);
}
```

The pro is that we have simulated the flexibility of true duck typing in a statically typed setting, meaning we can reduce code repetition while catching bugs at compile time. However, there is an extra level of indirection (the `callQuack` method from the above example) that makes the code less readable and maintainable as developers have to remember to call the standalone `callQuack` function instead of the more intuitive `obj.quack()`, also contradicting the mission of OOP to group data and behavior together in the first place.

Question 4

Question 4a

The pro is that there is no code duplication that results from instantiating a template with concrete types, resulting in a smaller executable size. It's also arguably more flexible as we can mix types like `std::string` and `int` without having to define some supertype to accommodate both via polymorphism. The con however is that it is much more error-prone. The developer uses **explicit casts** to operate with values of a specific type, which tells the compiler to interpret those bytes as that type regardless if it makes sense, forsaking type checking that could've been provided by the compiler. Furthermore, there is more burden on the developer to remember to explicitly cast with the right type every time and in the right places, which can lead to less readable and more buggy code.

Question 4b

This approach is similar to generics in that the "generic" code only occurs once in the program's code. All code *using* the generic simply use that code in their own specialized way. For generics, the type declaration is effectively replaced with the concrete type and for points, the `void` pointer is casted toa a pointer to the desired type.

Question 4c

Instead of `void` pointers, which can accommodate *any* type, we can store pointers to our custom supertype from which the types we want to accommodate derive:

```
#include <iostream>
using namespace std;

class StringOrInt{};

// Boxed types for the sake of demonstration.
class String : public StringOrInt
{
public:
    String(string const &str) : str_(str) {}
    string str_;
};

class Int : public StringOrInt
{
```

```
public:
    Int(int num) : num_(num) {}
    int num_;
};

class BoundedHolder
{
private:
    static const int MAX = 10;
    StringOrInt *arr_[MAX];
    int count_;

public:
    BoundedHolder()
    {
        count_ = 0;
    }
    void add(StringOrInt *ptr)
    {
        if (count_ >= MAX)
            return;
        arr_[count_++] = ptr;
    }
    StringOrInt *get(int pos) const
    {
        if (pos >= count_)
            return nullptr;
        return arr_[pos];
    }
};
```

Now `BoundedHolder` will only accept pointers to objects of types that subclass `StringOrInt`.

Question 5

Based on my knowledge, I know that JavaScript supports **prototypal inheritance**, where instead of objects deriving their state from classes, objects are created from existing objects. This approach of "cloning" an object from an existing object allows one to create a uniform set of objects that all have the same methods/fields. Because the objects are still independent of each other, their methods/fields can then be mutated individually, similar to how instances of the same class share the same blueprint but may have different concrete values for its state.

Question 6

This approach violates encapsulation because outside agents can see and even mutate fields of a class that make up its internal state. Python might have chosen this approach because it's a lot simpler for quick/small projects, and Python as a language is geared towards fast development time anyway. As code scales however, it would be more maintainable to use getters/setters rather than allowing direct access to member variables. This helps enforce (by developers, not by the language) encapsulation and a cleaner separation between the public interface and implementation details.

Question 7

The `IShape` type CANNOT be used to instantiate concrete objects like `x`. This is because an interface by definition does not provide any implementation for the prototypes it declares. Thus, a concrete instance of `IShape` would not make sense let alone be functional. *Pointers* like `ptr` to `IShape`s works just fine because it represents a point to some valid concrete instance that happens to **derive** (i.e. **implement**) the interface, so that instance is functional with concrete implementations for all abstract methods.

Question 8

Interfaces aren't needed in dynamically typed languages because by definition type checking is not done until runtime, so there is often no way to determine *before* runtime if a type conforms to a certain interface by implementing specific abstract methods. There are ways to simulate this AT runtime (such as the `abc` module in Python), but such mechanisms are not enforced by the language itself. Furthermore, dynamically typed languages often make use of **duck typing**, where the presence and validity of a method call is checked at runtime instead of before runtime, allowing for more flexible and self-contained code that does not rely on some interface construct. Many also support dynamically addition/removal of fields and/or methods of an object after it's been instantiated from a class, effectively modifying its public interface, making the concept of an interface definition less useful.

Question 9

Skipped.

Question 10

An example where interface inheritance is preferable over traditional subtype inheritance is when there is no logical "default" behavior for a supertype. For example, if I wanted to encompass all shapes such as `Rectangle`s and `Circle`s with a supertype `Shape` with shared behavior like `get_area()` and `get_perimeter()`, these methods make sense for the concrete subclasses but have no meaning for a `Shape` alone.

```
class Shape {
public:
    virtual double get_area() const = 0;
    virtual double get_perimeter() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double width, double height) :
        width_(width), height_(height) {}
    double get_area() const override { return width_ * height_; }
    double get_perimeter() const override { return 2 * width_ + 2 * height_; }
private:
    double width_;
    double height_;
};
```

```
class Circle : public Shape {
public:
    Circle(double radius) : radius_(radius) {}
    double get_area() const override { return PI * radius_ * radius_; }
    double get_perimeter() const override { return 2 * PI * radius_; }
private:
    double radius_;
};
```

The area of a rectangle is given by width * height and that of a circle is given by PI * radius^2. We know that a shape should *have* some way to compute the area, but a general shape itself doesn't have a sensible formula for its own area. Thus, it makes more sense to model it as an interface for subclasses to implement rather than a fully-fledged base class with concrete implementations.

On the other hand, it makes sense to use traditional subtype inheritance when there IS sensible "default" behavior for the general version of the types. Suppose we have a bunch of `Monster`s in some video game, and their hit points (HP) is modeled with some function of their `level`. An exception is `BossMonster`s, which have *more* hit points per level:

```
class Monster {
public:
    Monster(unsigned level) : level_(level) {}
    virtual unsigned get_max_HP() const { return 2 * level_ + 10 };
private:
    unsigned level_;
};

class BossMonster : public Monster {
public:
    virtual unsigned get_max_HP() const override {
      return 15 * Monster::get_max_HP();
    }
};
```

In this example, the generic `Monster` "knows" how to compute its own max HP, so it can provide a concrete implementation. Subclasses like `BossMonster` can then override that with their own implementation as needed.