Homework 7 Solutions

This homework covers the following topics from class:

- Function palooza part 4, OOP palooza, part 1
    - Polymorphism (Subtype, Ad-hoc, Parametric - Generics vs. Templates)
    - OOP Intro, OOP History

FUNC6: Generics, Templates (25 min)

For this problem, you are to write two versions of a container class called *Kontainer* that holds up to 100 elements, and allows the user to add a new element as well as find/return the minimum element, regardless of the data type of the elements (e.g., integers, floating-point numbers, strings, etc.). Don't worry about the user overflowing the container with items. You will be implementing this class using templates in C++ *and* generics in a statically-typed language of your choice (e.g., Java, Swift, Kotlin, Go, Haskell, etc.).

Part A: (10 min.) Show your implementation of the Kontainer class using C++ templates:

Solution:

```
template <typename T>
class Kontainer {
private:
    T elements_[100];
    int count_;

public:
    Kontainer() : count_(0) {}

    // don't worry about overflowing, as the spec says
    void add_element(const T& element) {
        elements_[count_++] = element;
    }

    // assumes there's at least one element
    T get_min() const {
        T min = elements_[0];
        for (int i = 1; i < count_; i++) {
            if (elements_[i] < min) {
                min = elements_[i];
            }
        }

        return min;
    }
};
```

Part B: (10 min.) Show your implementation of the Kontainer class using generics in a statically-typed language of your choice - feel free to use any online resources you like if you choose a different language for your solution:

Hint: If you haven't programmed in another statically-typed language other than C++ before, there are tons of online examples and online compilers which make this easy! Here are links to a few online compilers: Kotlin, Java, Swift.

Solution:

In Matt's slides, he has solutions in Haskell, Rust, and TypeScript.

```
// Here's a solution in C# (Generated by ChatGPT)
// Note: this is a bounded generic; it requires any type used with
// it to support a comparison interface (IComparable)
public class Kontainer<T> where T : IComparable<T>
{
    private T[] elements;
    private int count;
    public Kontainer() {
        elements = new T[100];
        count = 0;
    }

    public void add_element(T element) {
        elements[count++] = element;
    }

    public T get_min() {
        T min = elements[0];
        for (int i = 1; i < count; i++) {
            // the CompareTo() method is part of
            // the IComparable interface
            if (elements[i].CompareTo(min) < 0) {
                min = elements[i];
            }
        }
        return min;
    }
}
```

Part C: (5 min.) Now try creating two Kontainers of type double and type String with your generic class. Add some values to each Kontainer.

Show your code here:

Solution:

```
// C# Kontainer (generated by ChatGPT!)
class Program
{
    static void Main(string[] args)
    {
        Kontainer<double> doubleContainer = new Kontainer<double>();
        doubleContainer.add_element(3.14);
        doubleContainer.add_element(2.718);
        doubleContainer.add_element(1.618);
        doubleContainer.add_element(0.577);
        Console.WriteLine("Minimum element: " +
                           doubleContainer.get_min());

        Kontainer<string> stringContainer = new Kontainer<string>();
        stringContainer.add_element("apple");
        stringContainer.add_element("banana");
        stringContainer.add_element("carrot");
        stringContainer.add_element("avocado");
        Console.WriteLine("Minimum element: " +
                           stringContainer.get_min());
    }
}
```

FUNC7: Generics, Templates, Duck Typing (10 min)

(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?

Solution:

The template approach:

Benefits:

- Performance: Templates allow for compile-time polymorphism, which can lead to more efficient code execution at run-time. Why? The compiler generates specialized versions of the template function or class for each type used, resulting in optimized code.

- Earlier errors: Many template errors are reported during the compilation process rather than at run-time. helping developers identify and fix issues at an early stage.

Drawbacks:

- Code size: The use of templates can lead to code bloat. Each instantiation of a template with a different type creates a separate copy of the parameterized code, increasing the size of the compiled executable.
- Compilation time: Every time you use a templated function/class with a new type, the compiler generates a new version of that function/class and then compiles it. So if you use a templated class/function with many types, this can significantly increase compilation time.
- More confusing errors: Since errors during compilation are detected on generated code (after the parameterized type(s) are applied to the template to yield a concrete version of the templated code), compilation errors can sometimes be more cryptic.

The generics approach:

Benefits:

- Code size: A generic is compiled once during the creation of a single object file, regardless of how many types the generic is parameterized with in the rest of the code base. This can reduce the overall size of the codebase relative to templates.
- Clarity of errors: Generic type errors are reported during the compilation process, providing clear error messages.

- Compilation time: Since a generic is compiled only once regardless of how many different types it's used with, this can improve compilation time relative to templates.

Drawbacks:

- Generics are generic: An unbounded generic is not able to leverage any operations that are specific to a particular type of object (e.g., asking an object to quack()). Its code must be compatible with all object types, using only operations like assignment. Even operations as simple as comparisons are banned in generics unless they are bounded with a type. So this limits the capabilities of generic functions/classes.
- Performance: Bounded generics rely on dynamic dispatch, which can introduce performance overhead. Runtime determination of an object's type is necessary to dispatch to the appropriate method implementation.

The duck typing approach:

Benefits:

- Flexibility: Duck typing is more flexible. I can use objects of entirely unrelated types with a function/class and the code can work without complex typing rules. This can simplify code and allow objects of different types to be used interchangeably if they support the required operations (e.g., quack())..
- Compile time: For compiled languages, compilation can be much faster since all error checking of duck typing occurs at runtime.

Drawbacks:

- Error detection: Errors are not detected until run-time which may complicate debugging a program.
- Performance: Duck typing incurs some runtime performance overhead due to dynamic method dispatch and type checking during execution.

One possible answer (inspired by ChatGPT): If I had to pick one approach, I would choose the generics approach. Generics strike a balance between code size and good performance while providing early type-checking during compilation. They allow for code reuse and provide clear error messages during the compilation process. While there may be some performance overhead compared to templates, the flexibility and ease of use offered by generics make them a suitable choice for most scenarios.

FUNC8: Parametric Polymorphism (5 min)

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

Solution:

Dynamically-typed languages are programming languages where variables are not explicitly assigned a type and a variable can thus refer to values of different types at runtime. Parametric polymorphism, on the other hand, requires that a function or class can be defined in such a way that it can have type(s) specified/parameterized for one or more variables used by that function or class (e.g., vector<int>, where int is the parameterized type) prior to execution. Since variables are typeless in dynamically-typed languages, it's impossible to assign parameterized types to variables, precluding the use of parametric polymorphism.

FUNC9: Duck Typing in Statically Typed Languages (5 min)

(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)?

Solution:

Since a template can be used to define a class/function that can operate on any type T (e.g, a duck or a doctor), so long as the operations (e.g., quack( )) used by that class/function on each templated variable are consistent with that type T, a single templated class/function can essentially operate on objects of completely different/unrelated types. This gives us duck-typing-like functionality:

```
template <typename T>
class MakeItQuack {
public:
  void process(T &v) {
    v.quack();
  }
};

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

class Doctor {
public:
  void quack() { cout << "This cures insomnia!\n"; }
};

int main() {
  Duck daffy;
  MakeItQuack<Duck> quack1;
  quack1.process(daffy);   // causes a duck to quack

  Doctor fauchi;
  MakeItQuack<Doctor> quack2;
  quack2.process(fauchi);  // causes a doctor to quack
}
```

In the above example, MakeItQuack operates on Ducks and Doctors making them quack(), even though both classes are entirely unrelated and not derived from some common base type.

Templates are similar to duck typing in that a single function/class can operate on many different types of variables that are unrelated, so long as those variables support the proper operations.

Templates are different than duck typing in that with templates, the compiler can validate that all operations performed by the templated class/function are all valid at compile time, since the statically-typed language can verify that every parameterized type (e.g., Nerd or Duck) used with the template supports all required operations used by the template (e.g., quack()). These compile-time errors can assist in writing correct code. In contrast, with duck typing, since variables have no types, type checking must occur at runtime at the time the operation (e.g., quack()) is performed, leading to more bugs/issues that can't be detected prior to execution.

FUNC10: Templates, Generics (15 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.

Part A: (5 min.) Discuss the pros and cons of the approach shown above vs. C++ templates.

Solution:

Pros:

- this class can hold/work with any type of variable

Cons:

- there's no type checking possible so the user could try to add/extract incompatible objects (e.g., add a string, then try to extract and treat that string as if it were a Dog object).

- since the Holder class has no idea what type of objects/values it holds, or even if all the values it holds are of the same type, it's impossible for it to perform operations on those held objects (e.g., asking them to quack(), study(), etc.). With a template, since the parameterized type is known, the templated function/class can use any operations it likes so long as they're compatible with the templated type.

Part B: (5 min.) How is this approach similar or different to generics in languages like Java?

Solution:

In the above approach, the Holder class has no way of knowing what type of data it holds and no way of type checking that all items added to it are of a consistent type. Since it does not know the type of the values/objects it holds, it must treat them as generic values and can't make any assumptions about what methods/operations they support. Similarly, in Java generics, the generic function/class can make no assumptions about the type of the variables used with that class (unless the generic is a bounded generic), and therefore it is also limited in the set of operations it can perform on values that are processed by the generic (like our C++ code above).

On the other hand, when we instantiate a generic (e.g., ArrayList<Dog> x = new ArrayList<Dog>), the compiler can at least type-check all operations that use x. For example it can verify that only Dog objects are added to x (e.g., x.add(new Dog("fido"));), and code that extracts an item from x doesn't try to treat the extracted object as anything but a dog, e.g., a String. Our Holder class can't enforce this kind of checking.

Part C (5 min): What change might we make to the above program to make it work more like a bounded generic class?

Solution:

We could change the code so instead of using a void *, it used another concrete type like a Mammal *. This would now limit the Holder class to processing only Mammal-related objects (People, Dogs, Raccoons), but also it would allow the code to call any methods that are present in a Mammal class or any of its superclasses (e.g., breathe(), have_live_births(), etc.).. It would then be equivalent to a bounded generic in a language like Java.