Soyeon Kim, 106292803

Homework 7

CS 131, Fall 2024
Carey Nachenberg

FUNC6: Generics, Templates

Part A

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

template <typename T>

class Kotainer {
 private:
   std::vector<T> elements;
   static const int MAX_SIZE = 100;

 public:
   Kotainer() {}

   void add(const T& element) {
     if (elements.size() < MAX_SIZE) {
       elements.push_back(element);
     }
   }

   T getMin() const {
     return *std::min_element(elements.begin(), elements.end());
   }
};
```

Part B

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Kotainer<T extends Comparable<T>> {
 private List<T> elements;
 private static final int MAX_SIZE = 100;

 public Kotainer() {
   elements = new ArrayList<>();
 }
 public void add(T element) {
   if (elements.size() < MAX_SIZE) {
     elements.add(element);
   }
 }
 public T getMin() {
   return Collections.min(elements);
 }
}
```

Part C

```
  public static void main(String[] args) {
    Kotainer<Double> kotainer_double = new Kotainer<>();
    kotainer_double.add(1.0);
    kotainer_double.add(2.0);
    kotainer_double.add(3.0);
    System.out.println(kotainer_double.getMin());

    Kontainer<String> kotainer_string = new Kotainer<>();
    kotainer_string.add("cat");
    kotainer_string.add("dog");
    kotainer_string.add("elephant");
    System.out.println(kotainer_string.getMin());
  }
```

FUNC7: Generics, Templates, Duck Typing

(a) + no runtime overhead, since it's resolved at compile-time
- make a copy for every type, so if it's used for many different types, it may increase compilation time and code sizes.

(b) + ensure type safety, and short compilation time without code duplication
- less flexible than templates

(c) + very flexible and simple code
- errors caught at runtime, and slower performance

FUNC8: Parametric Polymorphism

This is because dynamically typed languages do not know the type until runtime, so they cannot enforce type constraints.

FUNC9: Duck Typing in Statically Typed Languages

Templates can be similar to duck-typing because a template is instantiated with a specific type only when it is used. This means that the compiler does not check if the template is type-safe for all possible types like Generics, but only for the types that are actually used. This provides flexibility similar to duck typing in dynamically typed language, however, any type incompatibility will result in a compile-time error rather than a runtime error.

FUNC10: Templates, Generics

Part A

It is a flexible and simple way to store different types of values without creating multiple copies like template. This can be faster and more straightforward, however, it is very dangerous because different types have different sizes, and void pointers do not carry type information. Implicit casting is required when retrieving values, and it leads to undefined behavior.

Part B

Both approaches allow storing a variety of types without knowing their specific type at compile time. However, the C++ Holder class is not type-safe, unlike Java generics, which enforce type safety through compile-time checking. In contrast, the C++ Holder class relies on runtime type checking, making it more error-prone.

Part C

We can add templates with type constraints.

```
template <typename T, typename = typename
std::enable_if<std::is_pointer<T>::value>::type>
```