1. FUNC6: Generics, Templates (25 min)
    a.

```cpp
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class Kontainer {
    public:
        void add (T element) {
            if (elements.size() < MAX_SIZE) {
                elements.push_back(element);
            } else {
                cout << "CONTAINER IS AT CAPACITY" << endl;
            }
        }

        T findMin() const {
            if (elements.empty()) {
                cout << "ELEMENTS IS EMPTY" << endl;
            }

            T minElement = numeric_limits<T>::max();

            for (auto& element : elements) {
                if (element < minElement) {
                    minElement = element;
                }
            }

            return minElement;
        }

    private:
        const int MAX_SIZE = 100; // Maximum capacity of the
container
        vector<T> elements;        // Vector to hold the elements
};
```

```
int main() {
    Kontainer<int> intContainer;
    intContainer.add(10);
    intContainer.add(5);
    intContainer.add(15);

    cout << "Minimum in intContainer: " << intContainer.findMin() <<
endl;
```

b.

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class Kontainer {
    public:
        void add (T element) {
            if (elements.size() < MAX_SIZE) {
                elements.push_back(element);
            } else {
                cout << "CONTAINER IS AT CAPACITY" << endl;
            }
        }

        T findMin() const {
            if (elements.empty()) {
                cout << "ELEMENTS IS EMPTY" << endl;
            }

            T minElement = numeric_limits<T>::max();

            for (auto& element : elements) {
                if (element < minElement) {
                    minElement = element;
                }
            }

            return minElement;
        }

    private:
        const int MAX_SIZE = 100; // Maximum capacity of the
container
        vector<T> elements;        // Vector to hold the elements
};

int main() {
    Kontainer<int> intContainer;
    intContainer.add(10);
    intContainer.add(5);
    intContainer.add(15);

    cout << "Minimum in intContainer: " << intContainer.findMin() <<
endl;
```

c.

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

public class Kontainer<T extends Comparable<T>> {
    private static final int MAX_SIZE = 100; // Maximum capacity of
the container
    private List<T> elements;               // List to hold the
elements

    // Constructor
    public Kontainer() {
        elements = new ArrayList<>();
    }
```

2. FUNC7: Generics, Templates, Duck Typing (10 min)
   If designing a new language, I would choose the Generics approach, blending compile-time safety with runtime type information, as seen in C# generics with reification. Generics strike a balance between performance and flexibility by ensuring strong compile-time type-checking to minimize runtime errors while retaining the ability to access type information at runtime when needed. They provide clearer error messages

compared to templates and avoid the runtime fragility of duck typing. Unlike templates, generics reduce code bloat and improve maintainability by using a single compiled representation of a generic class or function. Additionally, generics offer a simpler programming model for developers, avoiding the complexity of template specialization while still being flexible enough for most use cases. This combination of type safety, performance, and ease of use makes generics an ideal choice for modern programming languages.

3. FUNC8: Parametric Polymorphism (5 min)
   Dynamically-typed languages cannot support parametric polymorphism in the traditional sense because parametric polymorphism relies on type constraints enforced at compile time, which dynamically-typed languages lack. In statically-typed languages, parametric polymorphism allows a function or class to operate on any type while ensuring that the type remains consistent and constrained by generic type parameters. This ensures that type mismatches are caught during compilation.

4. FUNC9: Duck Typing in Statically Typed Languages (5 min)
   In statically-typed languages that use templates, such as C++, we can achieve behavior similar to duck typing by relying on the fact that templates don't actually require explicit types on certain required operations or methods. The compiler determines whether the type is valid by checking if it supports the operations invoked within the template code, which mimics the "if it quacks like a duck" philosophy of duck typing.

5. FUNC10: Templates, Generics (15 min)
   a. This void pointer form of code can simplify the syntax since only one implementation is needed, making it useful when templates aren't available. However, it comes at the cost of losing type safety, as the compiler cannot check types, increasing the risk of runtime errors due to incorrect casting. Additionally, manual casting and type handling make the code more error-prone and less readable. On the other hand, C++ templates enforce compile-time type safety, which prevents type mismatches and reduces runtime errors. Templates also make code more readable and maintainable, as the types are explicit and managed automatically, without requiring manual casting. They also allow for compile-time optimizations, generating more efficient code.

   b. This is similar to generics in languages like Java in that both allow for the storage of multiple types in a single container without specifying exact types at compile time. However, the key difference is that void pointers in C++ lose type safety, requiring manual casting and risking runtime errors, while Java generics provide type safety, ensuring type correctness at compile time and eliminating the need for casting. Java's generics are also more flexible, offering constraints and type checks, while void pointers are more prone to errors and less readable.

c. Use templates ☠️