# Variable templates and aliases

```
/*
    . Topics
        . Variable templates and alias templates:
            .#1: Variable templates

            .#2: Alias templates
                .  using declarations
                . typedefs are old C++ and we ignore them
*/
```

# Variable templates

```
//#1: A simple constant for Pi that works with different types
export template<typename T>
constexpr T PI = T(3.1415926535897932385);
```

# Variable templates

```cpp
//#1: A simple constant for Pi that works with different types
export template<typename T>
constexpr T PI = T(3.1415926535897932385);


//Using variable template
fmt::println("Using variable template");
auto pi_float_v = PI<float>;
auto pi_double_v = PI<double>;
auto pi_long_double_v = PI<long double>;
fmt::println("PI<float>: {}", pi_float_v);
fmt::println("PI<double>: {}", pi_double_v);
fmt::println("PI<long double>: {}", pi_long_double_v);
```

# Variable templates

```cpp
//#2: A variable template that simplifies checking if a type is integral
template<typename T>
constexpr bool is_integral_v = std::is_integral<T>::value;

export void check_integer(){
    // Testing with different types
    fmt::println("Is int integral? {}", is_integral_v<int>);           // true
    fmt::println("Is double integral? {}", is_integral_v<double>);     // false
    fmt::println("Is char integral? {}", is_integral_v<char>);         // true
    fmt::println("Is float integral? {}", is_integral_v<float>);       // false

    //Doing it raw
    fmt::println("Is int integral? {}", std::is_integral<int>::value);      // true
    fmt::println("Is double integral? {}", std::is_integral<double>::value); // false
    fmt::println("Is char integral? {}", std::is_integral<char>::value);    // true
    fmt::println("Is float integral? {}", std::is_integral<float>::value);  // false

}
```

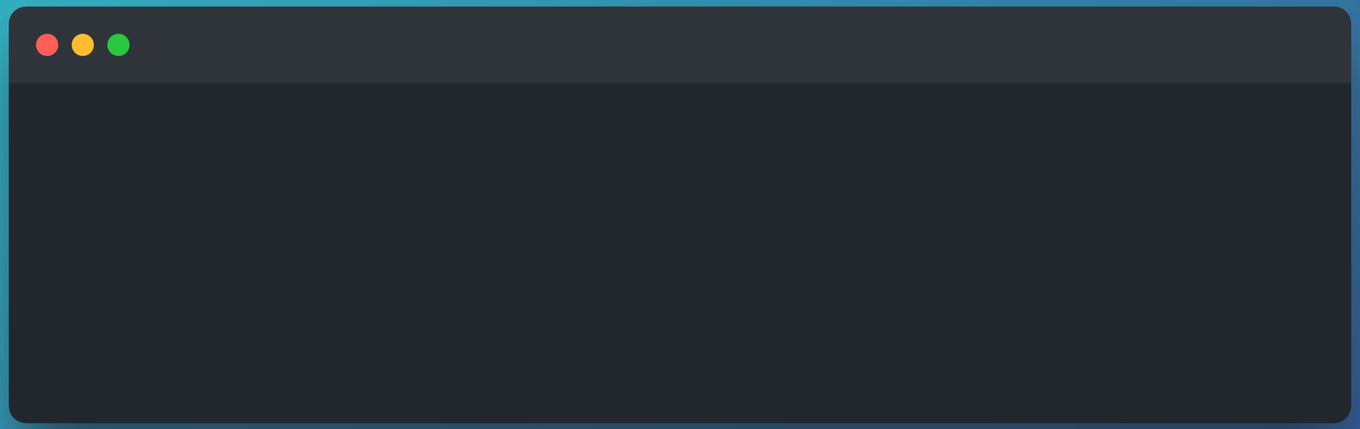
# Variable templates

```
//#3: Generic default value for different types
template<typename T>
constexpr T default_value = T{};

// Specializations for specific types. Yes, variable templates can be specialized.
// DO THIS: Comment out the specializations and see what happens.
template<>
constexpr int default_value<int> = 42;

template<>
constexpr const char* default_value<const char*> = "Hello, world!";

export void print_default_values(){
    fmt::println("Default value for int: {}", default_value<int>);
    fmt::println("Default value for double: {}", default_value<double>);
    fmt::println("Default value for char: {}", default_value<char>);
    fmt::println("Default value for float: {}", default_value<float>);
    fmt::println("Default value for const char*: {}", default_value<const char*>);
}
```

# Variable templates

```
// #4: Compute the factorial at compile-time using a variable template
export template<int N>
constexpr int factorial = N * factorial<N - 1>;

template<>
constexpr int factorial<0> = 1;
```

# Variable templates: Demo time!



# Alias templates

```
/*
        . Renaming templates
*/
```

# Alias templates

```
// KeyValueStore class template
export template <typename Key, typename Value>
class KeyValueStore {
public:
    void insert(const Key& key, const Value& value) {
        store[key] = value;
    }

    Value get(const Key& key) const {
        return store.at(key);
    }

private:
    std::map<Key, Value> store;
};

//using declarations
// Alias template for std::vector<T>
export template <typename T>
using Vec = std::vector<T>;

// Alias template for KeyValueStore with std::string as key
export template <typename Value>
using StringKeyStore = KeyValueStore<std::string, Value>;
```

# Alias templates

```
fmt::println("Using alias templates: ");
Vec<int> vec{1,2,3,4,5};
for(const auto& elem: vec){
    fmt::print("{} ", elem);
}
fmt::print("\n");

StringKeyStore<int> store;
store.insert("one", 1);
store.insert("two", 2);
store.insert("three", 3);
fmt::println("Value of one: {}", store.get("one"));
fmt::println("Value of two: {}", store.get("two"));
fmt::println("Value of three: {}", store.get("three"));
```