Recursion, variadic templates and fold expressions

```
/*
     . Topics:
       . Template recursion
       . Variadic function templates
       . Variadic class templates
       . Fold expressions

*/
```

Template recursion

```
/*
    . Exploring template recursion
        . bare recursion function that executes at run time
        . template recursion that executes at compile time
        . constexpr function that computes the factorial of a number at compile time
*/
```

Template recursion

```cpp
//Factorial
//Bare recursion
export int factorial_bare(int n) {
    if (n <= 1) return 1;
    return n * factorial_bare(n - 1);
}

// Template recursion - primary template
export template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

// Specialization for the base case
template<>
struct Factorial<0> {
    static constexpr int value = 1;
};


//constexpr function
export constexpr int factorial(int n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}
```

Template recursion

```
fmt::println("Factorial of 5 using bare recursion: {}", templates_1::factorial_bare(5));
fmt::println("Factorial of 5 using template recursion: {}", templates_1::Factorial<5>::value);
fmt::println("Factorial of 5 using constexpr function: {}", templates_1::factorial(5));
```

Template recursion: Demo time!

```
fmt::println("Factorial of 5 using bare recursion: {}", templates_1::factorial_bare(5));
fmt::println("Factorial of 5 using template recursion: {}", templates_1::Factorial<5>::value);
fmt::println("Factorial of 5 using constexpr function: {}", templates_1::factorial(5));
```

Variadic function templates

```
/*
    . Variadic Function templates:
        . Functions that can accept an arbitrary number of arguments.
    . How It Works:
        . Base Case:
            . The base case function takes two parameters of type T and returns their sum.
            . This base case handles the situation where there are exactly two arguments left to process.

        . Variadic Template:
            . The variadic template function uses a template parameter pack (Args...) to accept
              an arbitrary number of arguments. It adds the first argument (first) to the
              result of calling add recursively on the remaining arguments (args...). The
              recursion eventually reduces the problem to the base case of adding two parameters.

        . This function template can handle any number of arguments and sum them up correctly.

    . Visualization:
                    add(1, 2, 3, 4)
                    |
                    +-- 1 + add(2, 3, 4)
                            |
                            +-- 2 + add(3, 4)
                                |
                                +-- Base case: add(3, 4)
                                        |
                                        +-- 3 + 4 = 7
        Final result:   1   +   2    +    7    =   10

*/
```

Variadic function templates

```
namespace impl_1
{
    // Base case to add two numbers
    export template <typename T>
    T add(T a, T b) {
    return a + b;
    }

    export template <typename T, typename... Args>  // Args is a template parameter pack
    T add(T first, Args... args) {                  // Args... is a function parameter pack
        return (first + add(args...));              // args is a parameter pack expansion.
        // return add(first, add(args...));
    }

} // namespace impl_1
```

Variadic function templates

```
namespace impl_2
{
    //Another implementation using the size of operator together with if constexpr
    //The compiler still generates different overloads for each number of arguments.
    export template <typename T, typename... Args>
    T add2(T first, Args... args) {
        /*
            .1 First Call, 2nd Call, 3rd Call, 4th Call are evaluated here.
            At the last step, first will be 4 and args will be empty, so we return 4.

        */

        if constexpr (sizeof...(args) == 0) {
            return first;
        } else {
            return first + add2(args...);
        }
    }

} // namespace impl_2
```

Variadic function templates

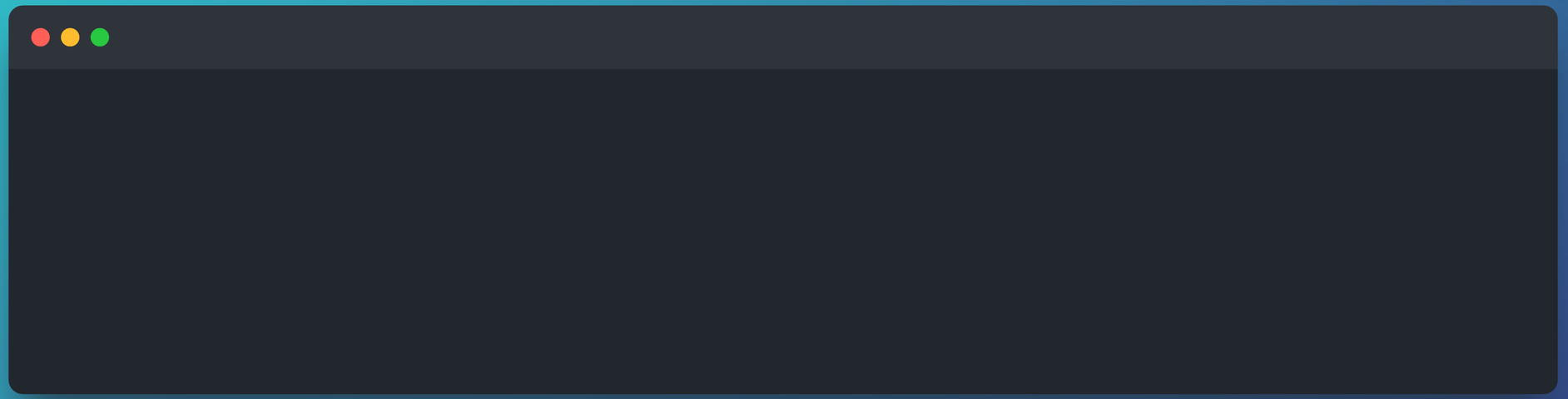
```
. Call stack
    ---------------
    | add2(1, 2, 3, 4)    |
    |      (first = 1)    |
    | args = (2, 3, 4)    |
    |---------------------|
    | add2(2, 3, 4)       |
    |      (first = 2)    |
    | args = (3, 4)       |
    |---------------------|
    | add2(3, 4)          |
    |      (first = 3)    |
    | args = (4)          |
    |---------------------|
    | add2(4)             |
    |      (first = 4)    |
    | args = ()           |
    |---------------------|
```

```
. Stack unwinding:
    ---------------
    | add2(4)           |
    | returns 4         |
    |-------------------|
    | add2(3, 4)        |
    | returns 3 + 4 = 7|
    |-------------------|
    | add2(2, 3, 4)     |
    | returns 2 + 7 = 9|
    |-------------------|
    | add2(1, 2, 3, 4)  |
    | returns 1 + 9 = 10|
    |-------------------|
```

Variadic function templates

```
//Do the sums from 1 element up to 5 elements
//auto value = variadic_function_templates::add(1);    //This will not compile,we need at least two parameters.
fmt::println("Sum of 2 elements: {}", templates_2::impl_1::add(1, 2));
fmt::println("Sum of 3 elements: {}", templates_2::impl_1::add(1, 2, 3));
fmt::println("Sum of 4 elements: {}", templates_2::impl_1::add(1, 2, 3, 4));
fmt::println("Sum of 5 elements: {}", templates_2::impl_1::add(1, 2, 3, 4, 5));
fmt::println("Sum of 5 elements using add2: {}", templates_2::impl_2::add2(1, 2, 3, 4, 5));
```

Variadic function templates: Demo time!



Variadic class templates.

```
/*
    . variadic class templates.
        . They also use template recursion combined with inheritance to work
        . We lay out the problem by setting up a simple tuple named bag.
            . We do this and test it in the simple namespace.
        . We would want to set up bags that can store any number of elements of different types.
            . We do this and test it in the wild namespace.

*/
```

Variadic class templates.

```
namespace simple{
    export template<typename T1, typename T2, typename T3>
    struct Bag {
        T1 first;
        T2 second;
        T3 third;
        Bag(T1 a, T2 b, T3 c) : first(a), second(b), third(c) {}
    };
} // namespace simple
```

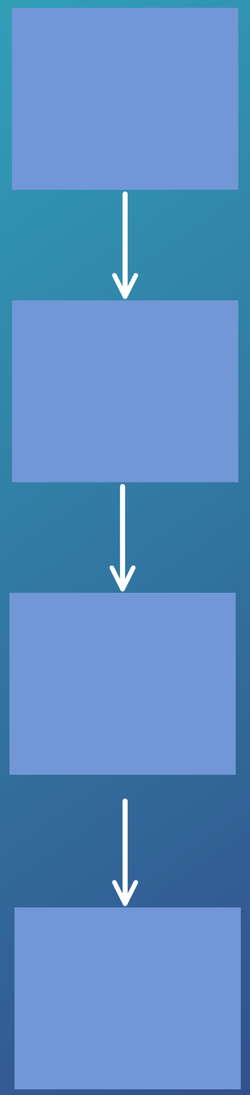
Variadic class templates.

```cpp
//Creating a Bag with different types: Simple
templates_3::simple::Bag<int, double, std::string> bag1(1, 3.14, "Hello");
templates_3::simple::Bag<std::string, double, std::string> bag2("The sum is: ", 3.14, "Hello");

fmt::println("\nSimple Bags: ");
fmt::println("Bag1: {}, {}, {}", bag1.first, bag1.second, bag1.third);
fmt::println("Bag2: {}, {}, {}", bag2.first, bag2.second, bag2.third);
```

Variadic class templates: any number of types.

```cpp
namespace wild{

    // Base case for recursion: an empty Bag
    export template<typename... Types>
    class Bag;  // Forward declaration

    // Specialization for the empty Bag
    export template<>
    class Bag<> {
        // Empty base case
    };

    // Recursive case: inherit from Bag of the tail
    export template<typename Head, typename... Tail>
    class Bag<Head, Tail...> : public Bag<Tail...> {
    public:
        // Constructor initializes the current value and passes the rest to the base class
        Bag(Head v, Tail... tail) : Bag<Tail...>(tail...), m_value(v) {}

        // Getter for the current value
        Head get_value() const { return m_value; }

        // Getter for the tail
        const Bag<Tail...>& get_tail() const { return *this; }
    public:
        Head m_value;
    };

} // namespace wild
```

Variadic class templates: any number of types.



Bag<> --> empty base case

Bag<std::string> --> stores std::string ("Hello, world!")

Bag<double, std::string> --> stores double (3.14)

Bag<int, double, std::string> --> stores int (42)

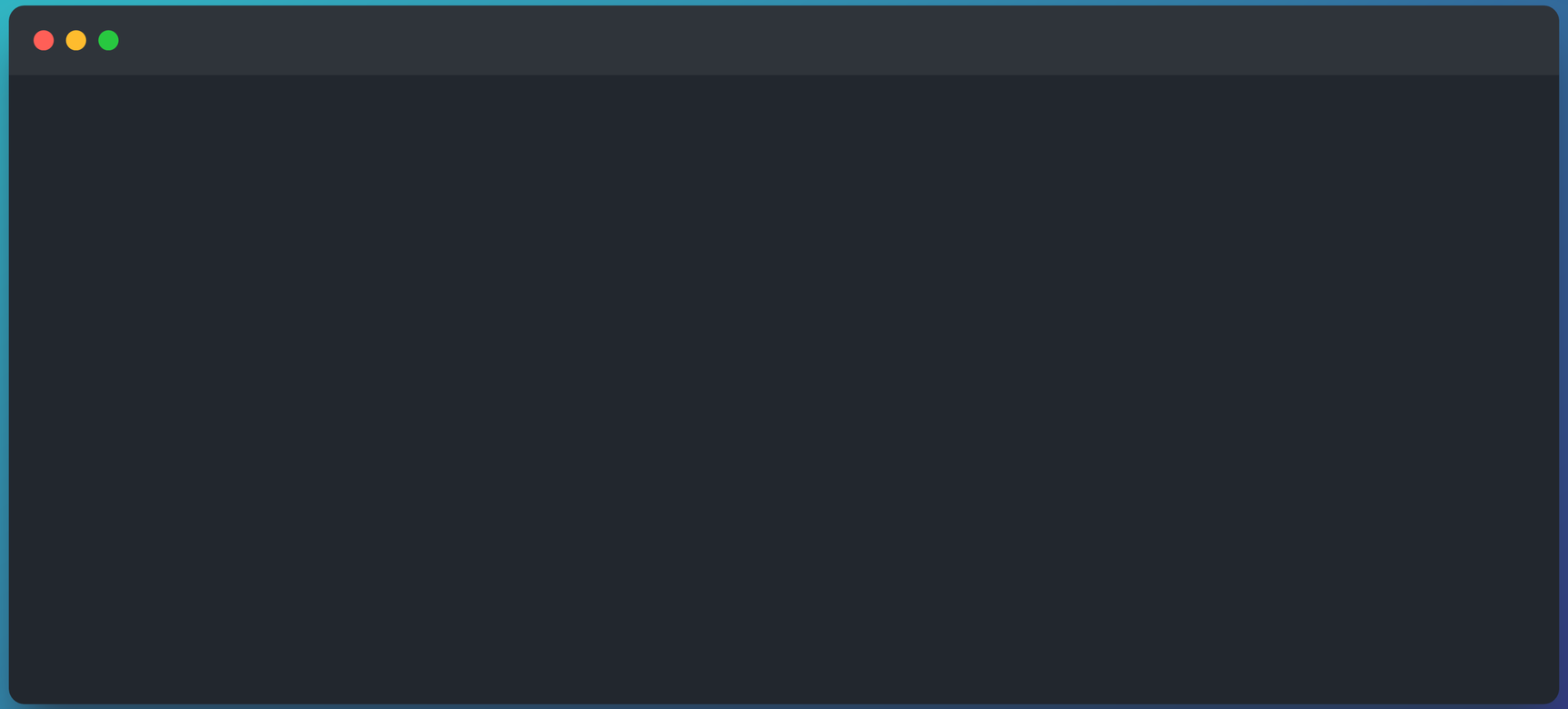
Variadic class templates: any number of types.

```
/*
    . Accessing the Values:
        . Here is code that is declaring a bag and accessing the value:
            Bag<int, double, std::string> myBag(42, 3.14, "Hello, world!");
            std::cout << myBag.get_value() << std::endl; // Prints: 42
            std::cout << myBag.get_tail().get_value() << std::endl; // Prints: 3.14
            std::cout << myBag.get_tail().get_tail().get_value() << std::endl; // Prints: Hello, world!
        . We access the first value by just calling the get_value method.
        . To access the others, we have to somehow jump to the base class and call get_value there.
        . We do this by casting the current object to the base class and then calling get_value.
                . This is what we do in the get_tail method.
                . Notice that it returns a reference to the base class.
                    . The syntax is weird but that's what it does.
        . You can't keep going up and calling get_value indefinitely though.
            . You can go up to just before the base case.


*/
```

Variadic class templates: any number of types.

```cpp
//Creating a Bag with different types: Wild
templates_3::wild::Bag<int, double, std::string> bag3(1, 3.14, "Hello");
templates_3::wild::Bag<std::string, double, std::string, int> bag4("The sum is: ", 3.14, "Hello", 42);

//Printing bag3
fmt::println("\nBag3: ");
fmt::println("First value: {}", bag3.get_value());
fmt::println("Second value: {}", bag3.get_tail().get_value());
fmt::println("Third value: {}", bag3.get_tail().get_tail().get_value());

//Printing bag4
fmt::println("\nBag4: ");
fmt::println("First value: {}", bag4.get_value());
fmt::println("Second value: {}", bag4.get_tail().get_value());
fmt::println("Third value: {}", bag4.get_tail().get_tail().get_value());
fmt::println("Fourth value: {}", bag4.get_tail().get_tail().get_tail().get_value());
```

Variadic class templates: any number of types - Demo time!



Fold expressions

```
/*
    . Fold expressions:
        .  fold expressions allow you to apply a binary operator to a parameter pack, expanding it
           in various ways depending on the type of fold expression.
        . We have two forms:
            . Unary fold expressions, which don't have an initial value.
            . Binary fold expressions, which have an initial value.

*/
```

Fold expressions

```cpp
// Unary left fold: (1,2,3) becomes ((1 + 2) + 3)
export template <typename... Args>
auto add_unary_left_fold(Args... args) {
    return ( ... + args);
}

//  Unary right fold; (1,2,3) becomes (1 + (2 + 3))
export template <typename... Args>
auto add_unary_right_fold(Args... args) {
    return (args + ...);
}

//  Binary left fold: (1,2,3) becomes 0 + (1 + (2 + 3))
export template <typename... Args>
auto add_binary_left_fold(Args... args) {
    return (0 + ... + args);
}

//  Binary right fold; (1,2,3) becomes ((1 + (2 + 3)) + 0)
export template <typename... Args>
auto add_binary_right_fold(Args... args) {
    return (args + ... + 0);
}
```

Fold expressions

. Comparisons:

Aspect	Unary Left	Unary Right	Binary Left	Binary Right
Syntax	(... op args)	(args op ...)	(init op ...)	(args op ... op init)
Direction	Left	Right	Left	Right
Example	((1 + 2) + 3)	(1 + (2 + 3))	0 + ((1 + 2) + 3)	(1 + (2 + (3 + 0)))
No Args	No (error)	No (error)	Yes	Yes
Initial Value	None	None	Required	Required

Fold expressions

```
//Trying out fold expressions
fmt::println("Unary left fold add 1,2,3: {}", templates_4::add_unary_left_fold(1, 2, 3, 4, 5));
fmt::println("Unary right fold add 1,2,3: {}", templates_4::add_unary_right_fold(1, 2, 3, 4, 5));
fmt::println("Binary left fold add 1,2,3: {}", templates_4::add_binary_left_fold(1, 2, 3, 4, 5));
fmt::println("Binary right fold add 1,2,3: {}", templates_4::add_binary_right_fold(1, 2, 3, 4, 5));


//Trying out empty parameter packs
//Unary folds must have at least one parameter, otherwise, you'll get a compiler error.
/*
//fmt::println("Unary left fold add: {}", templates_4::add_unary_left_fold()); Error
//fmt::println("Unary right fold add: {}", templates_4::add_unary_right_fold()); Error
fmt::println("Binary left fold add: {}", templates_4::add_binary_left_fold());
fmt::println("Binary right fold add: {}", templates_4::add_binary_right_fold());
fmt::println("Unary left fold add 1: {}", templates_4::add_unary_left_fold(1));
```