Attributes, local static variables and recursion

```
/*
    . Topics:
        . attributes
        . function local static variables
        . recursion:
            . momoization
            . iterative solutions
        . Recursive lambda functions (C++23)

*/
```

Attributes

```
/*
    . Attributes:
        . Attributes in C++ are a powerful way to communicate extra information to the compiler
          without affecting the behavior of the program directly.
        . Attributes from earlier standards like [[noreturn]], [[deprecated]], and [[nodiscard]]
          help with code readability, diagnostics, and error prevention.
        . Newer attributes in C++20 ([[likely]], [[no_unique_address]]) and C++23 ([[assume]],
          [[nodiscard("reason")]]) focus on further improving performance and providing clearer guidance.
        . Compilers have varying levels of support for this. Clang is a great compiler if your prioritize
          clear error messages.


*/
```

Attributes

```cpp
// [[noreturn]] (C++11)  tells the compiler that a function will not return
// control to the caller. It can be useful in functions that terminate the program or throw exceptions.
//This is useful when writing functions that handle fatal errors or terminate the program.
// The compiler will generate warnings if any code after a [[noreturn]] function is treated as reachable.
export [[noreturn]] void exit_program() {
    std::exit(1);  // Exits the program without returning
}

int main(){

    //[[noretun]]
    fmt::println("Exiting the program");
    exit_program();// This call may not return
    fmt::println("Program ends properly");
}
```

Attributes

```
//[[deprecated]] (Introduced in C++11, but C++14 added the ability to provide a message)
//Indicates that a function, class, or variable is deprecated, warning the user that the
// feature should not be used anymore. When updating libraries, marking old APIs as deprecated
// alerts users without breaking code immediately.
export [[deprecated("Use new_function() instead")]]
void old_function() {
    fmt::println("This is the old function.");
}

export void new_function() {
    fmt::println("This is the new function.");
}

int main(){

    //[[deprecated]]
    old_function(); // This will generate a warning
    new_function(); // This will not generate a warning
}
```

Attributes

```cpp
//[[nodiscard]] (C++17)
//This attribute warns the user if the return value of a function is discarded.
//It's useful for functions where the return value should not be ignored, such as error codes.
// Helps catch logical errors in cases where ignoring a return value could result
//in bugs, e.g., ignoring error codes in functions.
export [[nodiscard]] int calculate_value() {
    return 42;
}

int main(){

    //[[nodiscard]]
    calculate_value(); // This will generate a warning
    int result = calculate_value(); // This will not generate a warning
}
```

Attributes

```
//[[fallthrough]]
//In a switch statement, this attribute tells the compiler that falling through
// to the next case is intentional.
//If the attribute is not present, the compiler will generate a warning (or an error,
//depending on the compiler and settings) if a case falls through to the next one without a break statement.
// When designing complex control structures with switch, this attribute ensures
//readability and prevents unintentional fallthrough.

export void handle_switch(int value) {
    switch (value) {
        case 1:
            fmt::println("Handling 1");
            //[[fallthrough]];
        case 2:
            fmt::println("Handling 2");
            break;
        default:
            fmt::println("Handling default");
    }
}

int main(){

    //[[fallthrough]]
    handle_switch(1);
}
```

Attributes

```
//[[likely]] and [[unlikely]]   (C++20)
//These attributes indicate to the compiler the expected code execution path, potentially
// improving branch prediction. Optimizing performance-critical code where certain branches
// are more probable than others, such as handling errors in large input sets.

export int process_value(int value) {
    if (value == 42) [[likely]] {
        return value * 2;
    } else [[unlikely]] {
        return value / 2;
    }
}

int main(){
    //[[likely]] and [[unlikely]]
    auto value1 = process_value(42);
    auto value2 = process_value(43);
    fmt::println("Value1: {}, Value2: {}", value1, value2);
}
```

Attributes

```
//[[assume]]  (C++23) //This attribute allows programmers to inform the compiler that a certain
//condition will always hold. The compiler can then make optimizations based on this assumption,
//but if the condition fails, the behavior is undefined. Use this attribute in highly performance-sensitive
//code where you want to help the compiler optimize based on known truths.
export void process_data(int value) {
    [[assume(value > 0)]];
    fmt::println("Value is greater than 0: {}", value);
}

int main(){
    //[[assume]]
    process_data(5);
}
```

Attributes

```cpp
// [[nodiscard("Use the return value to check for errors")]] . C++23 adds the ability to provide a message
// with [[nodiscard]] to give more context to the warning. This can be useful for explaining why the return
// value should not be ignored.
export [[nodiscard("This result must be used for important logic.")]]
int compute_important_value() {
    return 100;
}

int main(){
    //[[nodiscard]] with message
    compute_important_value();
}
```

Function local static variables

```
//Static variables in functions: This allows us to keep track of the number of times a function has been called.
void user_login() {
    static size_t login_count = 0;
    ++login_count;

    fmt::println("Welcome back! This is your login attempt number: {}", login_count);
}

int main() {
    //Function local static variables

    user_login();
    user_login();
    user_login();
}
```

Recursion

The Fibonacci sequence is defined as:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2) for n > 1
```

fib(10) visualized:

```
fib(10)
├── fib(9)
│   ├── fib(8)
│   │   ├── fib(7)
│   │   │   ├── fib(6)
│   │   │   │   ├── fib(5)
│   │   │   │   │   ├── fib(4)
│   │   │   │   │   │   ├── fib(3)
│   │   │   │   │   │   │   ├── fib(2)
│   │   │   │   │   │   │   │   ├── fib(1) = 1
│   │   │   │   │   │   │   │   └── fib(0) = 0
│   │   │   │   │   │   │   └── fib(1) = 1
│   │   │   │   │   │   └── fib(2) (already computed above)
│   │   │   │   │   └── fib(3) (already computed above)
│   │   │   │   └── fib(4) (already computed above)
│   │   │   └── fib(5) (already computed above)
│   │   └── fib(6) (already computed above)
│   └── fib(7) (already computed above)
└── fib(8) (already computed above)
```

Raw recursion

```
/*
    This example is simple and readable, but recursion can be inefficient for
    large n due to repeated calculations. In practice, you should use an iterative
     solution or memoization for better performance.
*/
export constexpr int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Memoization

```
//Can we make this function constexpr?
export int fibonacci_memo(int n) {
    static std::vector<int> memo(100, -1); // Initialize with -1 to indicate uncomputed values

    if (n == 0) return 0;
    if (n == 1) return 1;

    if (memo[n] != -1) return memo[n]; // Return memoized value if available

    memo[n] = fibonacci_memo(n - 1) + fibonacci_memo(n - 2); // Compute and store the value
    return memo[n];
}
```

Memoization

```
//Iterative solution
 export constexpr int fibonacci_iterative(int n) {
     if (n == 0) return 0;
     if (n == 1) return 1;

     int prev = 0, curr = 1, next;
     for (int i = 2; i <= n; ++i) {
         next = prev + curr;
         prev = curr;
         curr = next;
     }
     return curr;
 }
```

Recursive lambdas

```cpp
//Recursive lambdas (C++ 23)
//Failed attempt
/*
    auto fibonacci_lambda = [](int n) {
    if (n < 2) { return n; }
    return fibonacci_lambda(n -1) + fibonacci_lambda(n -2);  // Compiler error!
};
*/
//We need a way to name the lambda, so we can access the name inside the body.
// C++23 explicit object parameters come to the rescue.
// For now think of this just as a piece of syntax.
// We'll learn more about explicit object parameters later on.
auto fibonacci_lambda = [](this auto& self, int n) {
    if (n < 2) { return n; }
    return self(n -1) + self(n -2);
};

auto value = fibonacci_lambda(10);
fmt::println("Fibonacci of 10 is: {}", value);
```