Understood. Let's explore some Miscellaneous C++ Features that are quite useful in modern C++ development.

VII. Miscellaneous C++ Features

- **Lambda Expressions (C++11):**
  - **Explanation:** Provide a concise way to define anonymous function objects (closures) directly in the code where they are used. They are particularly useful for short, one-off functions, often used with STL algorithms.
  - **Syntax:** [capture-list](parameter-list) -> return-type { function-body }
    - **capture-list:** Specifies which variables from the surrounding scope are accessible inside the lambda and how (by value or by reference).
    - **parameter-list:** Similar to the parameter list of a regular function.
    - **return-type:** Optional. If omitted, it is usually deduced by the compiler.
    - **function-body:** The code to be executed when the lambda is called.

- # Examples:

```C++
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    int factor = 2;

    // Lambda that multiplies each number by factor (capture by value)
    std::vector<int> multiplied;
    std::transform(numbers.begin(), numbers.end(),
std::back_inserter(multiplied),
                   [factor](int n) { return n * factor; });

    std::cout << "Multiplied numbers: ";
    for (int num : multiplied) {
        std::cout << num << " "; // Output: 2 4 6 8 10
    }
    std::cout << std::endl;

    int sum = 0;
    // Lambda that adds each number to sum (capture by reference)
    std::for_each(numbers.begin(), numbers.end(),
                   [&sum](int n) { sum += n; });
    std::cout << "Sum of numbers: " << sum << std::endl; // Output: 15

    // Lambda with auto return type deduction
    auto isEven =(int n) { return n % 2 == 0; };
    std::cout << "Is 4 even? " << isEven(4) << std::endl; // Output: 1 (true)

    // Lambda with explicit return type
    auto add =(int a, int b) -> int { return a + b; };
    std::cout << "Sum of 5 and 7: " << add(5, 7) << std::endl; // Output: 12

    return 0;
}
```

- **Capture Modes:**
  - ``: Nothing captured.
  - [=]: Capture all automatic variables from the surrounding scope by value.
  - [&]: Capture all automatic variables from the surrounding scope by reference.
  - [var]: Capture var by value.
  - [&var]: Capture var by reference.
  - [=, &var]: Capture all by value, but capture var by reference.
  - [&, var]: Capture all by reference, but capture var by value.
  - [this]: Capture the this pointer (available in member functions).
- **Function Objects (Functors) - Revisited:**
  - **Explanation:** As introduced earlier, function objects are objects of classes that overload the function call operator operator(). Lambdas are essentially syntactic sugar for creating small, often unnamed, function object classes.
  - **More Complex Example:**

```
C++
#include <iostream>
#include <vector>
#include <algorithm>
```

```
class PowerOf {
private:
    int exponent;
public:
    PowerOf(int exp) : exponent(exp) {}
    int operator()(int base) const {
        int result = 1;
        for (int i = 0; i < exponent; ++i) {
            result *= base;
        }
        return result;
    }
};

int main() {
    std::vector<int> bases = {2, 3, 4};
    std::vector<int> squares(bases.size());
    std::transform(bases.begin(), bases.end(), squares.begin(), PowerOf(2));
    std::cout << "Squares: ";
    for (int num : squares) {
        std::cout << num << " "; // Output: 4 9 16
    }
    std::cout << std::endl;

    std::vector<int> cubes(bases.size());
    std::transform(bases.begin(), bases.end(), cubes.begin(), PowerOf(3));
    std::cout << "Cubes: ";
    for (int num : cubes) {
        std::cout << num << " "; // Output: 8 27 64
    }
    std::cout << std::endl;

    return 0;
}
```

- ## Benefits of Function Objects over Raw Functions:

        - **State:** Function objects can have member variables, allowing them to maintain state across multiple calls (as seen in the PowerOf example).
        - **Type Safety:** They have a specific type, which can be useful in template programming.
        - **Inline Potential:** The compiler might have more opportunities to inline the code of a function object compared to a function pointer.
- **std::optional (C++17):**
    - **Explanation:** A template class that represents an optional value. It can either contain a value of its underlying type or indicate that no value is present. This is useful for situations where a function might not always return a meaningful result.
    - **Usage:** Include the <optional> header.
    - **Example:**

```C++
#include <iostream>
#include <optional>
#include <string>

std::optional<int> tryParseInt(const std::string& str) {
```

```
    try {
        return std::stoi(str);
    } catch (const std::invalid_argument&) {
        return std::nullopt; // Indicates no value
    } catch (const std::out_of_range&) {
        return std::nullopt;
    }
}

int main() {
    std::string input1 = "123";
    std::string input2 = "abc";

    std::optional<int> result1 = tryParseInt(input1);
    if (result1.has_value()) {
        std::cout << "Parsed integer: " << result1.value() << std::endl; // Output: 123
    } else {
        std::cout << "Failed to parse integer from '" << input1 << "'." << std::endl;
    }

    std::optional<int> result2 = tryParseInt(input2);
    if (result2) { // Can be used in a boolean context
        std::cout << "Parsed integer: " << result2.value() << std::endl;
    } else {
        std::cout << "Failed to parse integer from '" << input2 << "'." << std::endl; // Output:
Failed to parse integer from 'abc'.
    }

    // Accessing with value_or provides a default value if no value is present
    int value = result2.value_or(-1);
    std::cout << "Value or default: " << value << std::endl; // Output: -1

    return 0;
}
```

- **std::variant (C++17):**
  - **Explanation:** A template class that can hold a

value of one of several specified types. It's a type-safe union. You need to include the <variant> header.

- **Usage:** You specify the possible types that the variant can hold as template arguments.
- **Example:**

```C++
#include <iostream>
#include <variant>
#include <string>

int main() {
    std::variant<int, double, std::string> data;

    data = 10;
    std::cout << "Value: " << std::get<int>(data) << std::endl; // Output: 10

    data = 3.14;
    std::cout << "Value: " << std::get<double>(data) << std::endl; // Output: 3.14

    data = "Hello";
    std::cout << "Value: " << std::get<std::string>(data) << std::endl; // Output: Hello

    // Trying to get the wrong type will throw std::bad_variant_access
    try {
        std::cout << std::get<double>(data) << std::endl;
    } catch (const std::bad_variant_access& e) {
        std::cerr << "Error: " << e.what() << std::endl; // Output: Error: Bad variant access
    }

    // You can also use index() to get the index of the currently held type
    std::cout << "Current index: " << data.index() << std::endl; // Output: 2 (for string)

    // Using std::visit to process the variant based on its type
    auto visitor =(auto arg) {
```

```
        std::cout << "Visited value: ";
        if constexpr (std::is_same_v<decltype(arg), int>) {
            std::cout << "integer " << arg << std::endl;
        } else if constexpr (std::is_same_v<decltype(arg), double>) {
            std::cout << "double " << arg << std::endl;
        } else if constexpr (std::is_same_v<decltype(arg), std::string>) {
            std::cout << "string '" << arg << "'" << std::endl;
        }
    };
    std::visit(visitor, data); // Output: Visited value: string 'Hello'

    return 0;
}
```

- **std::any (C++17):**
  - **Explanation:** A template class that can hold a value of any type (including void). It provides a way to opt out of static type checking when needed. You need to include the <any> header.
  - **Usage:** You can assign values of different types to an std::any object. To access the stored value, you need to know its actual type and use std::any_cast.
  - **Example:**

```
C++
#include <iostream>
#include <any>
#include <string>

int main() {
    std::any value;

    value = 42;
```

```
    std::cout << "Value: " << std::any_cast<int>(value) << std::endl; // Output: 42

    value = 3.14;
    std::cout << "Value: " << std::any_cast<double>(value) << std::endl; // Output: 3.14

    value = std::string("World");
    std::cout << "Value: " << std::any_cast<std::string>(value) << std::endl; // Output:
World

    // Trying to cast to the wrong type will throw std::bad_any_cast
    try {
        std::cout << std::any_cast<int>(value) << std::endl;
    } catch (const std::bad_any_cast& e) {
        std::cerr << "Error: " << e.what() << std::endl; // Output: Error: bad any cast
    }

    // You can check the type using type()
    std::cout << "Type of value: " << value.type().name() << std::endl;

    // You can also check if a value is held
    if (value.has_value()) {
        std::cout << "Value is present." << std::endl; // Output: Value is present.
    }

    return 0;
}
```

- **Type Aliases (using) (C++11):**
  - **Explanation:** Provide a way to give a new name to an existing type. This can improve code readability and make it easier to change types later. using can also be used for template type aliases, which were more cumbersome with typedef.

- # Examples:

```C++
#include <iostream>
#include <vector>
#include <string>

int main() {
    using Integer = int;
    Integer myInt = 10;
    std::cout << "Integer value: " << myInt << std::endl;

    using StringList = std::vector<std::string>;
    StringList names = {"Alice", "Bob", "Charlie"};
    for (const auto& name : names) {
        std::cout << name << " ";
    }
    std::cout << std::endl;

    // Template type alias
    template <typename T>
    using VectorOf = std::vector<T>;

    VectorOf<double> doubles = {1.1, 2.2, 3.3};
    for (double d : doubles) {
        std::cout << d << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

- # Comparison with typedef:

```C++
// typedef for a function pointer
typedef void (*FunctionPtr)(int);

// using alias for the same function pointer
using FunctionPtrAlias = void (*)(int);
```

```
// typedef for a template (more awkward)
template <typename T>
struct MyContainer { /* ... */ };
// typedef MyContainer<int> IntContainer; // Doesn't work directly

// using alias for a template (cleaner)
template <typename T>
using MyIntContainer = MyContainer<T>;
MyIntContainer<int> container;
```

- **Attributes (C++11 onwards):**
  - **Explanation:** Provide a way to specify additional information about entities (like variables, functions, classes) to the compiler. Attributes can be used for various purposes, such as hinting to the compiler about optimization, suppressing warnings, or indicating specific properties.
  - **Syntax:** [[attribute]] or [[namespace::attribute]]
  - **Examples:**

```
C++
#include <iostream>

[[noreturn]] // Indicates that this function does not return
void terminate_program() {
    std::cout << "Terminating..." << std::endl;
    std::exit(1);
}

[[deprecated("Use a newer function instead")]]
void old_function() {
```

```
    std::cout << "This is the old function." << std::endl;
}

int main() {
    // terminate_program(); // Program will exit here

    old_function(); // Compiler might issue a deprecation warning

    return 0;
}
```

- **Common Attributes:**
  - [[noreturn]]: Indicates that a function does not return to the caller.
  - [[deprecated]]: Marks an entity as deprecated, potentially with a message.
  - [[fallthrough]]: Indicates that a case label in a switch statement is intentionally falling through to the next case.
  - [[maybe_unused]]: Indicates that a variable might intentionally not be used, suppressing compiler warnings.
  - [[nodiscard]]: Indicates that the return value of a function should not be ignored.
- **Compile-time Assertions (static_assert) (C++11):**
  - **Explanation:** Allow you to perform assertions

at compile time. If the condition specified in static_assert is false, the compiler will issue a diagnostic message and stop compilation. This is useful for checking conditions that can be determined during compilation, such as type properties or constant expressions.

- **Syntax:**
  static_assert(boolean-constant-expression, optional-message);
- **Example:**

```C++
#include <iostream>
#include <type_traits> // For std::is_integral

template <typename T>
T process(T value) {
    static_assert(std::is_integral<T>::value, "Type T must be an integer type.");
    return value + 5;
}

int main() {
    std::cout << process(10) << std::endl; // OK
    // std::cout << process(3.14) << std::endl; // Compilation error due to static_assert

    static_assert(sizeof(int) == 4, "Integer size is not 4 bytes on this platform.");

    return 0;
}
```

- **Special Member Functions - Revisited (Modern C++ Emphasis):**

- **Explanation:** These are member functions that the compiler can automatically generate if you don't define them explicitly. Understanding their behavior, especially with move semantics in C++11 and later, is crucial.
- **The Six Special Member Functions (as of C++11):**
    1. **Default Constructor:** Called when an object is created with no arguments.
    2. **Destructor:** Called when an object is destroyed.
    3. **Copy Constructor:** Called when an object is initialized from another object of the same type.
    4. **Copy Assignment Operator:** Called when an object is assigned the value of another object of the same type.
    5. **Move Constructor (C++11):** Called when an object is initialized from an rvalue reference of the same type (transfers resources).
    6. **Move Assignment Operator (C++11):** Called when an object is assigned the value of an rvalue reference of the same type

(transfers resources).

- **Compiler Generation Rules (Simplified for modern C++):**
  - If you don't declare any constructors, the compiler will generate a public default constructor.
  - If you don't declare a copy constructor, it will be generated if needed (performs member-wise copy).
  - If you don't declare a move constructor, it will be generated if needed (performs member-wise move if possible).
  - Similar rules apply to the copy assignment, move assignment, and destructor.
  - **Rule of Zero:** If your class manages no resources (like raw pointers to dynamically allocated memory), you should generally rely on the compiler-generated special member functions. This is often the best approach in modern C++ by using RAII (e.g., smart pointers, containers).
  - **Rule of Five (now Rule of Six with C++11):** If you need to define any of the destructor,

copy constructor, copy assignment, move constructor, or move assignment, you likely need to define all of them to ensure proper resource management. However, with the Rule of Zero, you often don't need to define any of these manually.

- **= default and = delete:** You can explicitly tell the compiler to generate the default version of a special member function using = default, or to prevent its generation using = delete.

```C++
class NonCopyable {
public:
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;
};
```

This concludes our exploration of Miscellaneous C++ Features. These features enhance the power and flexibility of the language. Next, we will briefly touch upon C++ Standard Library Overview. Let me know if you have any questions about any of these topics.[1]

Sources

1. https://netfiles.pw/cpp-11-cpp14-cpp17-cpp20-features/