C++ news

Oxiane INSTITUT

Summary

- New Features in C++17
  - if constexpr.
  - Structured Bindings.
  - Inline Variables.
  - Fold Expressions.
  - Filesystem Library.
  - std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types.
  - Parallel Algorithms.
  - std::byte: New type to represent bytes.
  - Uniform Initialization Enhancements.
- New Features in C++20
  - Concepts: Allows defining constraints for templates, improving readability and safety of templates.
  - Ranges.
  - Coroutines.
  - Modules.
  - Three-Way Comparison (Spaceship Operator <=>).
  - Calendar and Time Zone Library.
  - Expanded constexpr.
  - constinit.

Summary

- New Features in C++23
  - Pattern Matching.
  - std::expected.
  - Executor Framework.
  - Networking TS.
  - Reflection.
  - Improved support for SIMD

C++ 17

Oxiane INSTITUT

if constexpr

The `if constexpr` construct was introduced to provide a way to perform compile-time conditional branching. Unlike the regular `if` statement, `if constexpr` is evaluated at compile time. If the condition evaluates to `true`, the associated statement is compiled; otherwise, it is discarded.

Syntax and Basic Usage

```
if constexpr (condition) {
    // Compile-time branch if condition is true
} else {
    // Compile-time branch if condition is false
}
```

if constexpr

Example 1: Type Traits

```
#include <iostream>
#include <type_traits>
template<typename T>
void print_type(T value) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "The type is integral: " << value << '\n';
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "The type is floating-point: " << value << '\n';
    } else {
        std::cout << "The type is neither integral nor floating-point.\n";
    }
}
int main() {
    print_type(42);         // Outputs: The type is integral: 42
    print_type(3.14);       // Outputs: The type is floating-point: 3.14
    print_type("Hello");    // Outputs: The type is neither integral nor floating-point.
    return 0;
}
```

if constexpr

Example 2: Enabling Specific Member Functions

`if constexpr` can also be used to enable or disable member functions based on conditions known at compile time.

```
#include <iostream>
#include <type_traits>

template<typename T>
class MyClass {
public:
    void foo() {
        if constexpr (std::is_same_v<T, int>) {
            std::cout << "Specialized behavior for int.\n";
        } else {
            std::cout << "General behavior.\n";
        }
    }
};

int main() {
    MyClass<int> intObj;
    intObj.foo(); // Outputs: Specialized behavior for int.
    MyClass<double> doubleObj;
    doubleObj.foo(); // Outputs: General behavior.
    return 0;
}
```

if constexpr

Use Cases

1. **Template Metaprogramming**: `if constexpr` simplifies and enhances template metaprogramming by providing a cleaner way to handle different cases based on types or compile-time constants.
2. **Optimizations**: By excluding code branches that are not needed based on compile-time conditions, `if constexpr` can help optimize the compiled binary by removing unnecessary code paths.
3. **Error Checking and Static Assertions**: It can be used for static checks, allowing different error messages or static assertions based on the compile-time conditions.
4. **Library Design**: In designing generic libraries, `if constexpr` allows for more flexible and efficient handling of a wide range of types and use cases without sacrificing performance or code readability.

if constexpr

Comparison with SFINAE

Before `if constexpr`, SFINAE (Substitution Failure Is Not An Error) was a common technique to achieve similar results. However, SFINAE can be more complex and harder to maintain. `if constexpr` offers a more straightforward syntax and better readability, making it a preferred choice for many modern C++ metaprogramming tasks.

Limitations

- The condition in `if constexpr` must be a compile-time constant.
- The branches must be valid code paths even if they are not taken, meaning they should not cause compilation errors on their own.

if constexpr

Use Case: Compile-Time Array Size Optimization

```
#include <iostream>
#include <array>

// Simple insertion sort implementation
template<typename T, std::size_t N>
void insertion_sort(std::array<T, N>& arr) {
    for (std::size_t i = 1; i < N; ++i) {
        T key = arr[i];
        std::size_t j = i;
        while (j > 0 && arr[j - 1] > key) {
            arr[j] = arr[j - 1];
            --j;
        }
        arr[j] = key;
    }
}
// Placeholder for more complex sorting (like quicksort)
template<typename T, std::size_t N>
void quicksort(std::array<T, N>& arr) {
    // Implementation of a more efficient sort for larger arrays
    // For demonstration, we'll assume it's a complex operation.
    std::sort(arr.begin(), arr.end());  // Using STL sort for simplicity
}
template<typename T, std::size_t N>
void sort_array(std::array<T, N>& arr) {
    if constexpr (N <= 10) {
        // Use insertion sort for small arrays
        insertion_sort(arr);
    } else {
        // Use quicksort for larger arrays
        quicksort(arr);
    }
}
int main() {
    std::array<int, 5> smallArray = {5, 2, 4, 3, 1};
    sort_array(smallArray);
    for (const auto& el : smallArray) {
        std::cout << el << ' ';
    }
    std::cout << '\n';  // Outputs: 1 2 3 4 5
    std::array<int, 15> largeArray = {10, 7, 2, 3, 1, 4, 9, 8, 5, 6, 15, 12, 14, 13, 11};
    sort_array(largeArray);
    for (const auto& el : largeArray) {
        std::cout << el << ' ';
    }
    std::cout << '\n';  // Outputs: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    return 0;
}
```

Structured Bindings

Structured bindings in C++17 provide a convenient way to unpack objects into individual named variables. This feature is particularly useful for working with tuples, pairs, and other data structures that group multiple values together. By using structured bindings, you can decompose these structures into distinct variables with clear, descriptive names, making the code more readable and expressive.

Syntax and Basic Usage

The syntax for structured bindings involves using the `auto` keyword followed by square brackets that list the variable names:

```
auto [var1, var2, var3] = some_tuple;
```

Structured Bindings

Example 1: Decomposing a Tuple

One of the simplest and most common use cases is decomposing a tuple into separate variables.

```
#include <iostream>
#include <tuple>

int main() {
    std::tuple<int, double, std::string> data = {1, 3.14, "C++17"};

    auto [id, value, text] = data; // Structured binding

    std::cout << "ID: " << id << "\n";
    std::cout << "Value: " << value << "\n";
    std::cout << "Text: " << text << "\n";

    return 0;
}
```

Structured Bindings

Example 2: Iterating Over a Map

Structured bindings can also be useful when iterating over containers like maps, where each element is a key-value pair.

```
#include <iostream>
#include <map>

int main() {
    std::map<int, std::string> students = {
        {1, "Alice"},
        {2, "Bob"},
        {3, "Charlie"}
    };

    for (const auto& [id, name] : students) {
        std::cout << "Student ID: " << id << ", Name: " << name << '\n';
    }

    return 0;
}
```

Structured Bindings

Use Cases

1. **Destructuring Data**: When dealing with complex data types, structured bindings allow you to destructure and access data members directly, enhancing code readability.
2. **Iterating Over Containers**: They are particularly useful in range-based for loops, making it easier to access elements like key-value pairs in maps.
3. **Function Return Values**: Functions that return tuples or pairs can have their return values unpacked directly into variables, simplifying the handling of multiple return values.
4. **Working with Complex Objects**: They can be used to unpack objects like structs or classes with multiple public members, making the code cleaner and more concise.

Structured Bindings

Advantages of Structured Bindings

- **Improved Readability**: By assigning meaningful names to the elements of a tuple or pair, code becomes more self-explanatory.
- **Reduced Boilerplate**: Reduces the need for explicit type declarations and separate statements for accessing tuple or pair elements.
- **Enhanced Flexibility**: Works seamlessly with various standard library types like `std::pair`, `std::tuple`, and user-defined types that have suitable decomposition declarations.

Structured Bindings

Example 3: Unpacking Custom Structs

Structured bindings can also be applied to user-defined types if they provide the appropriate support.

```
#include <iostream>
#include <string>

struct Person {
    std::string name;
    int age;
};

int main() {
    Person person{"John Doe", 30};

    auto [personName, personAge] = person; // Unpacking the struct

    std::cout << "Name: " << personName << ", Age: " << personAge << '\n';

    return 0;
}
```

Structured Bindings

Limitations and Considerations

- **Scope**: Variables declared with structured bindings are limited to the scope in which they are declared.
- **Mutable Elements**: For non-const structured bindings, the elements must be mutable. If the source data is const, the bound variables will also be const.
- **Copy vs. Reference**: By default, structured bindings copy the elements. To avoid copies, you can use references:

```
auto& [var1, var2, var3] = some_tuple; // Bind by reference
```

Inline Variables

C++17 Inline Variables

The C++17 standard introduced the concept of **inline variables**, which allows the definition of variables with external linkage in header files without violating the One Definition Rule (ODR). This feature is particularly useful for defining constants and other global variables that should be accessible across multiple translation units without causing linker errors.

Key Concept and Syntax

Before C++17, defining a global variable in a header file could lead to multiple definitions across different translation units, causing linker errors. To avoid this, the usual practice was to declare the variable `extern` in the header and define it in a corresponding source file. With C++17's inline variables, you can define a variable in a header file, and the compiler ensures that there is only one definition across all translation units.
The syntax for declaring an inline variable is to use the `inline` keyword before the variable definition:

```
inline constexpr int example = 42;
```

Inline Variables

Example 1: Global Constants

A common use case for inline variables is defining global constants in header files. Before C++17, to avoid multiple definitions, constants were often defined as `static` or `constexpr`. With inline variables, you can define these constants directly in the header without worrying about multiple definitions.

```
// constants.h
#ifndef CONSTANTS_H
#define CONSTANTS_H

inline constexpr int MAX_CONNECTIONS = 100;
inline constexpr double PI = 3.14159265358979323846;

#endif // CONSTANTS_H
```

```
// main.cpp
#include <iostream>
#include "constants.h"

int main() {
    std::cout << "Max connections: " << MAX_CONNECTIONS << '\n';
    std::cout << "Value of PI: " << PI << '\n';
    return 0;
}
```

Inline Variables

Example 2: Shared Configuration Across Modules

```
// config.h
#ifndef CONFIG_H
#define CONFIG_H
inline const std::string defaultFilePath = "/etc/myapp/config.txt";
inline const int maxRetries = 5;
#endif // CONFIG_H
```

```
// logger.cpp
#include <iostream>
#include "config.h"

void logConfigPath() {
    std::cout << "Config file path: " << defaultFilePath << '\n';
}

// main.cpp
#include <iostream>
#include "config.h"

void attemptOperation() {
    for (int i = 0; i < maxRetries; ++i) {
        std::cout << "Attempt " << i + 1 << " of " << maxRetries << '\n';
        // Operation logic...
    }
}
int main() {
    logConfigPath();
    attemptOperation();
    return 0;
}
```

Inline Variables

Advantages of Inline Variables

1. **Single Definition Across Translation Units**: Inline variables guarantee that there is only one definition across all translation units, preventing linker errors.
2. **Convenient for Header-Only Libraries**: They make it easier to develop header-only libraries where constants or global variables need to be shared across multiple files.
3. **Simplified Syntax**: Inline variables eliminate the need for separate declarations and definitions (`extern` in headers and definitions in source files), simplifying code management.
4. **Consistent Values**: They ensure that a single value is used throughout the application, reducing the risk of discrepancies.

Inline Variables

Example 3: Template Specializations

```
// template_config.h
#ifndef TEMPLATE_CONFIG_H
#define TEMPLATE_CONFIG_H

template <typename T>
inline constexpr T epsilon = T(0); // General case

template <>
inline constexpr double epsilon<double> = 1e-9; // Specialization for double

#endif // TEMPLATE_CONFIG_H
```

```
// calculations.cpp
#include <iostream>
#include "template_config.h"

void checkEpsilon() {
    std::cout << "Epsilon for double: " << epsilon<double> << '\n';
}

int main() {
    checkEpsilon();
    return 0;
}
```

Inline Variables

Considerations and Best Practices

- **Use with Caution**: Inline variables are a powerful tool, but they should be used judiciously. Overuse can lead to overly complex and hard-to-maintain code, especially in large projects.
- **Namespace Use**: To avoid name collisions, inline variables are often placed inside namespaces.
- **Compatibility**: While inline variables simplify certain scenarios, they require C++17 or newer. Ensure your build environment supports this standard.

Fold expressions

Fold expressions in C++17 simplify the process of writing variadic template functions by providing a concise way to apply binary operators to parameter packs. A fold expression allows you to apply an operator to a series of arguments, either left-to-right or right-to-left, without manually writing the recursive template logic previously required.

Key concept and syntax

The general syntax for a fold expression is:

```
(... operator pack)  // Unary left fold
(pack operator ...)  // Unary right fold
(init operator ... operator pack)  // Binary left fold with initial value
(pack operator ... operator init)  // Binary right fold with initial value
```

Fold expressions

Example 1: summing values with unary left fold

One of the simplest and most illustrative use cases for fold expressions is summing a list of numbers.

```
#include <iostream>

template<typename... Args>
auto sum(Args... args) {
    return (args + ...);
}

int main() {
    std::cout << sum(1, 2, 3, 4, 5) << '\n';  // Outputs: 15
    std::cout << sum(10, 20, 30) << '\n';    // Outputs: 60
    return 0;
}
```

Fold expressions

Example 2: logical AND with unary right fold

Fold expressions can also be used with logical operators. For instance, you might want to check if all arguments satisfy a certain condition using a logical AND.

```
#include <iostream>

template<typename... Args>
bool all_true(Args... args) {
    return (args && ...);
}

int main() {
    std::cout << std::boolalpha;
    std::cout << all_true(true, true, true) << '\n';   // Outputs: true
    std::cout << all_true(true, false, true) << '\n';  // Outputs: false
    return 0;
}
```

Fold expressions

Use cases

1. **Aggregation operations**: Fold expressions can perform aggregation operations like summing numbers, multiplying, finding the minimum/maximum, and more.
2. **Logical operations**: They can simplify logical checks across multiple conditions, such as checking if all or any conditions are met (`&&` or `||`).
3. **Container operations**: They can apply operations on elements in containers, such as combining strings or concatenating lists.
4. **Custom operations**: Fold expressions can work with user-defined types and custom binary operators, allowing for flexible and reusable code patterns.

Fold expressions

Example 3: concatenating strings

Fold expressions are also useful for concatenating a variable number of strings or string-like objects.

```
#include <iostream>
#include <string>

template<typename... Args>
std::string concatenate(Args... args) {
    return (std::string(args) + ...); // Unary left fold
}

int main() {
    std::string result = concatenate("Hello", " ", "World", "!");
    std::cout << result << '\n';  // Outputs: Hello World!
    return 0;
}
```

Fold expressions

Advantages of fold expressions

1. **Conciseness**: They eliminate the need for manually writing recursive templates to unpack parameter packs, making the code more concise and easier to understand.
2. **Safety**: By leveraging compile-time guarantees, fold expressions reduce the likelihood of runtime errors associated with manual unpacking of parameter packs.
3. **Generality**: Fold expressions work with all types that support the specified operator, including user-defined types and standard library types.

Considerations and best practices

- **Operator support**: The types used in fold expressions must support the specified operator. For example, to use the `+` operator, the types must have a defined `operator+`.
- **Order of evaluation**: Be mindful of the order of operations. For instance, left folds and right folds can produce different results with non-associative operators.
- **Initial values**: When necessary, provide an initial value to avoid issues with empty packs or to ensure proper handling of the first element.

Filesystem Library

The C++17 standard introduced the filesystem library, which provides a set of facilities for performing operations on file systems and their components, such as paths, regular files, and directories. The library offers a convenient and standardized way to manipulate file system paths, query file attributes, and perform file system operations, making it easier to write portable and efficient file-related code.

Key components and concepts

The filesystem library is part of the `<filesystem>` header and is contained within the `std::filesystem` namespace. Some of the key components include:

- **`std::filesystem::path`**: A class that represents file and directory paths.
- **`std::filesystem::directory_entry`**: Represents an entry in a directory, such as a file or another directory.
- **`std::filesystem::directory_iterator`**: Allows iteration over the contents of a directory.
- **`std::filesystem::recursive_directory_iterator`**: Similar to `directory_iterator`, but it recursively iterates through directories.
- **`std::filesystem::file_status`**: Represents the status of a file, including its type and permissions.

Filesystem Library

Example 1: basic file and directory operations

```
#include <iostream>
#include <filesystem>
namespace fs = std::filesystem;
int main() {
    fs::path filePath = "example.txt";
    // Check if file exists
    if (fs::exists(filePath)) {
        std::cout << filePath << " exists.\n";
    } else {
        std::cout << filePath << " does not exist.\n";
    }
    // Create a directory
    fs::path dirPath = "example_directory";
    if (!fs::exists(dirPath)) {
        fs::create_directory(dirPath);
        std::cout << "Directory created: " << dirPath << '\n';
    }
    // Iterate through files in a directory
    for (const auto& entry : fs::directory_iterator(dirPath)) {
        std::cout << "Found: " << entry.path() << '\n';
    }
    return 0;
}
```

Filesystem Library

Example 2: working with paths

The `std::filesystem::path` class provides a rich interface for manipulating file paths. This includes operations like concatenation, extraction of components, and normalization.

```
#include <iostream>
#include <filesystem>

namespace fs = std::filesystem;

int main() {
    fs::path path1 = "C:/Program Files";
    fs::path path2 = "MyApp";
    fs::path fullPath = path1 / path2 / "app.exe";

    std::cout << "Full path: " << fullPath << '\n';
    std::cout << "Filename: " << fullPath.filename() << '\n';
    std::cout << "Parent path: " << fullPath.parent_path() << '\n';
    std::cout << "Extension: " << fullPath.extension() << '\n';

    return 0;
}
```

Filesystem Library

Use cases

1. **File and Directory Management**: Creating, renaming, moving, and deleting files and directories.
2. **Path Manipulation**: Constructing, decomposing, and normalizing file paths in a platform-independent manner.
3. **File System Queries**: Checking file existence, file size, permissions, and other attributes.
4. **Directory Traversal**: Iterating over files and subdirectories within a directory, including recursive traversal.

Filesystem Library

Example 3: querying file attributes

The filesystem library allows querying various attributes of files and directories, such as size, last modification time, and file permissions.

```
#include <iostream>
#include <filesystem>
#include <chrono>
#include <iomanip>

namespace fs = std::filesystem;

int main() {
    fs::path filePath = "example.txt";

    if (fs::exists(filePath)) {
        std::cout << "File size: " << fs::file_size(filePath) << " bytes\n";

        // Get the last write time
        auto ftime = fs::last_write_time(filePath);
        std::time_t cftime = decltype(ftime)::clock::to_time_t(ftime);
        std::cout << "Last write time: " << std::put_time(std::localtime(&cftime), "%F %T") << '\n';

        // Get file permissions
        fs::file_status status = fs::status(filePath);
        auto perm = status.permissions();
        std::cout << "Permissions: "
                  << ((perm & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
                  << ((perm & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
                  << ((perm & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
                  << '\n';
    } else {
        std::cout << "File does not exist.\n";
    }

    return 0;
}
```

Filesystem Library

Advantages of the filesystem library

1. **Platform Independence**: The library abstracts away platform-specific details, providing a consistent interface across different operating systems.
2. **Ease of Use**: Simplifies common file and directory operations, making them more accessible and less error-prone.
3. **Safety**: Functions in the filesystem library often return `std::error_code` or throw exceptions, providing clear error reporting and handling mechanisms.
4. **Efficiency**: Optimized for typical file system operations, the library provides efficient access to file system metadata and operations.

Considerations and best practices

- **Error Handling**: Be aware that many filesystem operations can throw exceptions or return error codes. Proper error handling is crucial, especially in production environments.
- **Security**: When working with file paths, consider potential security issues such as path traversal vulnerabilities. Always validate and sanitize inputs that involve file paths.
- **Performance**: Be mindful of the performance implications of file system operations, especially when dealing with large directories or files. Use appropriate iterators and caching strategies as needed.

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

C++17 introduced several utility classes that significantly enhance the language's capabilities in handling optional values, variant types, and generic data storage. These classes—`std::optional`, `std::variant`, and `std::any`—provide a standardized and type-safe way to manage and manipulate these data patterns, simplifying code and reducing errors.

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

`std::optional`

`std::optional` represents a type that may or may not contain a value. It provides a way to explicitly handle cases where a value may be absent, avoiding the pitfalls of using raw pointers or sentinel values like `nullptr` or special numbers.

```
#include <iostream>
#include <optional>
std::optional<int> find_even(int num) {
    if (num % 2 == 0) {
        return num;
    }
    return std::nullopt;
}

int main() {
    auto result = find_even(4);
    if (result) {
        std::cout << "Found even number: " << *result << '\n';
    } else {
        std::cout << "No even number found.\n";
    }
    result = find_even(3);
    if (result) {
        std::cout << "Found even number: " << *result << '\n';
    } else {
        std::cout << "No even number found.\n";
    }
    return 0;
}
```

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

Use cases for `std::optional`

1. **Nullable Values**: Replaces nullable pointers and special sentinel values, providing a clear and type-safe way to represent optional values.
2. **Error Handling**: Can be used to indicate failure without using exceptions or error codes, especially in contexts where a failure simply means the absence of a value.
3. **Default Arguments**: Offers a way to handle default or optional arguments in functions without overloading.

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

`std::variant`

`std::variant` is a type-safe union that can hold one value out of a specified set of types. It provides an alternative to traditional unions with added safety features, ensuring that only one type is active at any time.

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

using VarType = std::variant<int, float, std::string>;

void print_variant(const VarType& var) {
    std::visit([](const auto& value) { std::cout << value << '\n'; }, var);
}

int main() {
    VarType v = 42;
    print_variant(v);  // Outputs: 42

    v = 3.14f;
    print_variant(v);  // Outputs: 3.14

    v = std::string("Hello, world!");
    print_variant(v);  // Outputs: Hello, world!

    return 0;
}
```

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

Use cases for `std::variant`

1. **Type-Safe Unions**: Replaces traditional unions with type safety, preventing issues like accessing the wrong type.
2. **Heterogeneous Collections**: Useful for managing collections or return values that can vary in type.
3. **Tagged Unions**: Provides a type-safe alternative to manual type tagging in unions.

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

`std::any`

`std::any` is a type-erased container for single values of any type. It can store any copy-constructible type, allowing for the storage and retrieval of values without knowing their type at compile time.

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

void print_any(const std::any& a) {
    if (a.type() == typeid(int)) {
        std::cout << std::any_cast<int>(a) << '\n';
    } else if (a.type() == typeid(double)) {
        std::cout << std::any_cast<double>(a) << '\n';
    } else if (a.type() == typeid(std::string)) {
        std::cout << std::any_cast<std::string>(a) << '\n';
    } else {
        std::cout << "Unknown type\n";
    }
}

int main() {
    std::any a = 10;
    print_any(a);  // Outputs: 10
    a = 3.14;
    print_any(a);  // Outputs: 3.14
    a = std::string("Hello, world!");
    print_any(a);  // Outputs: Hello, world!
    return 0;
}
```

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

Use cases for `std::any`

1. **Type Erasure**: Allows for the storage of objects of any type without knowing the type at compile time.
2. **Plugin Systems**: Useful in systems where objects of unknown types at compile time need to be stored and manipulated.
3. **Generic Containers**: Can be used to store objects of heterogeneous types in a single container.

std::optional, std::variant, std::any: Utility classes for managing optional values, variants, and generic types

Advantages of these utility classes

1. **Safety and Clarity**: These classes provide safer and clearer alternatives to traditional techniques like raw pointers, unions, and manual type management.
2. **Type Safety**: `std::optional` and `std::variant` ensure type safety by providing clear and explicit interfaces for handling multiple types or the absence of a value.
3. **Versatility**: `std::any` offers great flexibility by allowing storage of any type, making it useful for cases where type information is not available at compile time.

Considerations and best practices

- **Performance**: `std::variant` and `std::any` involve some overhead due to type checking and type erasure, respectively. They should be used judiciously in performance-critical code.
- **Error Handling**: Proper handling of type retrieval, especially with `std::any_cast`, is essential to avoid exceptions due to bad casts.
- **Usage Context**: While these utilities are powerful, they should be used in appropriate contexts where their benefits outweigh the overhead and complexity.

Parallel algorithms

C++17 introduced parallel algorithms to the Standard Template Library (STL), enabling developers to execute standard algorithms in parallel, sequentially, or with vectorization. This feature leverages modern hardware capabilities, such as multi-core processors and SIMD (Single Instruction, Multiple Data) instructions, to improve performance for computationally intensive tasks.

Key concept and syntax

Parallel algorithms are part of the `<algorithm>` header and are invoked by passing an execution policy as the first argument. The execution policies defined in C++17 are:

- `std::execution::seq`: Specifies sequential execution. It behaves like the traditional STL algorithms.
- `std::execution::par`: Specifies parallel execution. The algorithm may run concurrently across multiple threads.
- `std::execution::par_unseq`: Specifies parallel and vectorized execution. The algorithm may use both multiple threads and SIMD instructions.

Parallel algorithms

Example 1: parallel `std::for_each`

The `std::for_each` algorithm can be parallelized to apply a function to each element in a range concurrently.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::for_each(std::execution::par, data.begin(), data.end(), [](int& n) {
        n *= 2; // Double each element
    });

    for (int n : data) {
        std::cout << n << ' ';
    }
    std::cout << '\n';  // Outputs: 2 4 6 8 10 12 14 16 18 20

    return 0;
}
```

Parallel algorithms

Use cases for parallel algorithms

1. **Data Processing**: Ideal for operations on large datasets, such as applying transformations or filters, where parallel processing can significantly reduce execution time.
2. **Scientific Computing**: Useful in scientific applications that require heavy numerical computations, leveraging multiple cores for tasks like simulations or statistical analyses.
3. **Image and Signal Processing**: Accelerates processing tasks that involve manipulating large arrays of data, such as applying filters or transformations.

Parallel algorithms

Example 2: parallel `std::transform`

The `std::transform` algorithm applies a function to each element in a range and stores the result in another range. Using parallel execution can accelerate this process.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <execution>

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::vector<int> results(data.size());

    std::transform(std::execution::par, data.begin(), data.end(), results.begin(), [](int n) {
        return n * n; // Square each element
    });

    for (int n : results) {
        std::cout << n << ' ';
    }
    std::cout << '\n';  // Outputs: 1 4 9 16 25 36 49 64 81 100

    return 0;
}
```

Parallel algorithms

Advantages of parallel algorithms

1. **Performance**: By leveraging multiple cores, parallel algorithms can significantly reduce the time required for data processing tasks.
2. **Simplicity**: Parallel algorithms provide a simple and standardized way to exploit parallelism without needing to manually manage threads.
3. **Safety**: The standard parallel algorithms are designed to prevent common concurrency issues, such as data races, making them safer than manually implemented parallel solutions.

Parallel algorithms

Example 3: `std::reduce` with parallel execution

`std::reduce` is a parallel version of `std::accumulate` that can sum up a range of values. It can be particularly useful for large datasets where a parallel reduction can speed up the computation.

```
#include <iostream>
#include <vector>
#include <numeric>
#include <execution>

int main() {
    std::vector<int> data(1000000, 1); // A large vector with one million elements all set to 1

    // Calculate the sum using parallel execution
    int sum = std::reduce(std::execution::par, data.begin(), data.end());

    std::cout << "Sum: " << sum << '\n';  // Outputs: Sum: 1000000

    return 0;
}
```

Parallel algorithms

Considerations and best practices

- **Overhead**: For small datasets, the overhead of setting up parallel execution may outweigh the performance benefits. Parallel algorithms are most beneficial for large datasets or computationally expensive operations.
- **Thread Safety**: While parallel algorithms handle many concurrency issues, it's essential to ensure that the operations performed in parallel are thread-safe, particularly if they involve shared resources.
- **Hardware Limitations**: The actual performance gain from parallel algorithms depends on the hardware, such as the number of available cores and support for SIMD instructions.

Limitations

1. **Deterministic Execution**: Unlike sequential algorithms, parallel algorithms may produce results in different orders due to non-deterministic execution. This is especially relevant for algorithms like `std::for_each` where the operation may have side effects.
2. **Exceptions Handling**: If exceptions are thrown during parallel execution, they may terminate the entire operation, and handling these exceptions can be complex.

std::byte: New type to represent bytes

C++17 introduced `std::byte` as a new data type to represent raw byte data. The primary purpose of `std::byte` is to provide a type-safe way of manipulating byte-oriented data, distinguishing it from character types and integer types. This separation helps prevent unintentional misuse of byte data as numbers or characters, enhancing type safety and code clarity.

Key concept and syntax

`std::byte` is an enumeration type defined in the `<cstddef>` header. Unlike traditional integral types, `std::byte` does not implicitly convert to or from integer types, preventing accidental arithmetic operations that might lead to unintended consequences.

```
#include <iostream>
#include <cstddef> // For std::byte

int main() {
    std::byte b = std::byte{0x1F}; // Initialize with a value
    std::cout << "Byte value: " << std::to_integer<int>(b) << '\n';
    // Bitwise operations
    b = b | std::byte{0x10};
    std::cout << "After OR operation: " << std::to_integer<int>(b) << '\n';

    b = b & std::byte{0xF0};
    std::cout << "After AND operation: " << std::to_integer<int>(b) << '\n';
    return 0;
}
```

std::byte: New type to represent bytes

Key features of `std::byte`

1. **Type Safety**: Unlike `char` or `unsigned char`, `std::byte` does not represent a character or numerical value. It is purely a collection of bits, making it clear that operations on it are bitwise rather than arithmetic.
2. **No Implicit Conversion**: `std::byte` does not implicitly convert to or from integer types, reducing the risk of accidental misuse.
3. **Bitwise Operations**: `std::byte` supports bitwise operations, such as AND (`&`), OR (`|`), XOR (`^`), and NOT (`~`). These operations are explicitly defined for `std::byte` and require explicit casting to or from integer types using `std::to_integer`.

Use cases for `std::byte`

1. **Memory and Buffer Management**: `std::byte` is ideal for representing raw memory or buffer data, where the actual content is not interpreted as a specific type (e.g., numbers or characters).
2. **Networking and Binary Protocols**: In applications involving low-level networking, binary protocols, or file formats, `std::byte` can be used to handle byte streams without implying any specific data type.
3. **Type-Safe Low-Level Operations**: It helps in performing low-level operations, such as byte-wise manipulation or bit-field extraction, in a type-safe manner.

std::byte: New type to represent bytes

Example: Using `std::byte` in a buffer

```
#include <iostream>
#include <vector>
#include <cstddef>

int main() {
    std::vector<std::byte> buffer(10);

    // Set specific bytes
    buffer[0] = std::byte{0x01};
    buffer[1] = std::byte{0x02};
    buffer[2] = std::byte{0x03};

    // Access and manipulate bytes
    for (size_t i = 0; i < buffer.size(); ++i) {
        std::cout << "Byte " << i << ": " << std::to_integer<int>(buffer[i]) << '\n';
    }

    // Example of modifying a specific byte
    buffer[0] = buffer[0] | std::byte{0x0F};
    std::cout << "Modified first byte: " << std::to_integer<int>(buffer[0]) << '\n';

    return 0;
}
```

std::byte: New type to represent bytes

Considerations and best practices

- **Explicit Casting**: When you need to interpret or manipulate `std::byte` values as integers, use `std::to_integer`. This requirement makes operations explicit and prevents accidental misuse.
- **Avoid Misuse as Char**: Do not use `std::byte` as a character type. For text data, use `char`, `unsigned char`, or other character types to avoid confusion.
- **Bitwise Operations**: Use `std::byte` for tasks that naturally involve bitwise operations or when you want to emphasize the raw nature of the data.

Limitations

- **No Arithmetic**: `std::byte` does not support arithmetic operations, reflecting its role as a pure byte data type rather than a numeric type.
- **Increased Verbosity**: The lack of implicit conversion and the requirement for explicit casting can increase verbosity, but this trade-off enhances safety and clarity.

Uniform initialization enhancements

C++17 introduced several enhancements to the uniform initialization syntax, which was originally introduced in C++11. Uniform initialization provides a consistent syntax for initializing objects, aggregates, and arrays using curly braces `{}`. This syntax is intended to reduce ambiguity and improve type safety by preventing implicit narrowing conversions. The enhancements in C++17 aim to further unify and simplify initialization patterns, making the language more consistent and expressive.

Uniform initialization enhancements

1. Copy Elision

In C++17, mandatory copy elision was introduced, which means that the compiler is required to omit the copy or move operation in certain initialization contexts, even if the copy constructor or move constructor has side effects. This enhancement simplifies the language and can improve performance by eliminating unnecessary copies.

```
#include <iostream>

struct Example {
    Example() { std::cout << "Constructed\n"; }
    Example(const Example&) { std::cout << "Copy Constructed\n"; }
    Example(Example&&) { std::cout << "Move Constructed\n"; }
};

Example createExample() {
    return {}; // Copy elision in action
}

int main() {
    Example e = createExample(); // No copy or move construction
    return 0;
}
```

Uniform initialization enhancements

2. Aggregate Initialization with `=`

C++17 clarifies the use of aggregate initialization using the `=` sign. It ensures that aggregates (structs or arrays with public data members and no user-provided constructors) can be initialized using the `{}` syntax with the `=` sign.

```
#include <iostream>
#include <string>

struct Data {
    int id;
    std::string name;
};

int main() {
    Data d = {1, "John"}; // Aggregate initialization with '='
    std::cout << d.id << ": " << d.name << '\n';
    return 0;
}
```

Uniform initialization enhancements

3. Direct List Initialization

C++17 allows direct list initialization for non-aggregate types, ensuring that the initialization is as consistent and predictable as possible across different types.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5}; // Direct list initialization
    for (int n : vec) {
        std::cout << n << ' ';
    }
    std::cout << '\n';
    return 0;
}
```

Uniform initialization enhancements

Advantages of uniform initialization

1. **Consistency**: Provides a single, consistent syntax for initializing objects, reducing the likelihood of errors and confusion.
2. **Prevention of Narrowing Conversions**: The `{}` initialization syntax disallows narrowing conversions, enhancing type safety. For example, trying to initialize an `int` with a `double` value using `{}` would result in a compile-time error.
3. **Clarity and Explicitness**: Makes the initialization process explicit, clearly indicating the values being used for initialization.

Use cases for uniform initialization

1. **Initializing Aggregates and Arrays**: Provides a clear and concise way to initialize aggregates (such as structs and arrays) with specific values.
2. **Avoiding Ambiguities**: Helps avoid ambiguities that can arise with traditional initialization methods, such as constructor overload resolution issues.
3. **Safe Initialization**: Ensures safe initialization by preventing issues like narrowing conversions and uninitialized variables.

Uniform initialization enhancements

Example: Preventing Narrowing Conversions

```
#include <iostream>

void process(int value) {
    std::cout << "Value: " << value << '\n';
}

int main() {
    // int a = {2.5}; // Error: narrowing conversion
    int b{2.5}; // Error in C++11 and later: narrowing conversion
    process(b);

    return 0;
}
```

Uniform initialization enhancements

Considerations and best practices

- **Default Initialization**: Be aware of how default initialization behaves with uniform initialization. For example, `int x{};` zero-initializes `x`, whereas `int x;` leaves `x` uninitialized.
- **Aggregate Initialization**: When using uniform initialization for aggregates, ensure that all members are initialized or have default values to avoid undefined behavior.
- **Constructor Overloading**: If a class has overloaded constructors, be mindful of how uniform initialization interacts with them, as it might lead to different behavior compared to traditional constructor calls.

Limitations

- **Non-Aggregate Classes with Private Members**: Uniform initialization cannot be used to initialize non-aggregate classes with private members directly from outside the class. In such cases, constructors must be used.
- **Complex Overloads**: With complex overloads, especially involving initializer lists and constructors, it might sometimes be challenging to predict which constructor will be called. It's essential to understand the precedence and resolution rules.

C++ 20

Oxiane INSTITUT

Concepts: Defining constraints for templates

C++20 introduced **concepts**, a powerful feature that allows developers to specify constraints on template parameters. Concepts improve the expressiveness, readability, and safety of templates by ensuring that template arguments meet specific requirements. This feature helps catch errors at compile time, making generic programming more robust and easier to understand.

Basic syntax

```
template<typename T>
concept ConceptName = /* condition involving T */;

template<ConceptName T>
void function(T param) {
    // Function implementation
}
```

Alternatively, a concept can be used with `requires` syntax directly in the template declaration:

```
template<typename T>
requires ConceptName<T>
void function(T param) {
    // Function implementation
}
```

Concepts: Defining constraints for templates

Example 1: Defining and using a concept

```
#include <iostream>
#include <concepts>

template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template<Addable T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(3, 4) << '\n';           // Outputs: 7
    std::cout << add(2.5, 3.5) << '\n';       // Outputs: 6
    // std::cout << add("hello", "world") << '\n'; // Error: const char* is not Addable
    return 0;
}
```

Concepts: Defining constraints for templates

Advantages of concepts

1. **Improved Readability**: Concepts provide a clear and concise way to specify the requirements of template parameters, making the code easier to understand.
2. **Early Error Detection**: By specifying constraints, concepts enable the compiler to catch errors at the point of template instantiation, providing more informative error messages.
3. **Self-Documenting Code**: Concepts act as documentation, explicitly stating the requirements for template arguments, which aids in code maintenance and usage.
4. **Overload Resolution**: Concepts can help in resolving overloads by constraining which function templates are viable, leading to more precise function selection.

Concepts: Defining constraints for templates

Use cases for concepts

1. **Constrained Algorithms**: Defining constraints for algorithm templates ensures that only suitable types can be used, preventing misuse and potential runtime errors.
2. **Generic Data Structures**: Concepts can define the necessary operations and types required by template-based data structures, ensuring correct usage.
3. **Interface Specification**: Concepts can serve as a specification for interfaces in generic programming, similar to interfaces in other programming languages.

Concepts: Defining constraints for templates

Example 2: Combining concepts with `requires` expressions

```
#include <iostream>
#include <concepts>
template<typename T>
concept Incrementable = requires(T a) {
    { ++a } -> std::same_as<T&>;
    { a++ } -> std::same_as<T>;
};
template<typename T>
concept Decrementable = requires(T a) {
    { --a } -> std::same_as<T&>;
    { a-- } -> std::same_as<T>;
};
template<typename T>
concept Bidirectional = Incrementable<T> && Decrementable<T>;
template<Bidirectional T>
void manipulate(T& value) {
    ++value;
    --value;
    std::cout << value << '\n';
}
int main() {
    int x = 10;
    manipulate(x); // Outputs: 10
    // Uncommenting the next line would cause a compile-time error because `double` is not Bidirectional
    // double y = 5.5;
    // manipulate(y);
    return 0;
}
```

Concepts: Defining constraints for templates

Considerations and best practices

- **Granularity**: Concepts should be defined with the right level of granularity. Too coarse concepts can be less useful, while too fine-grained concepts can be overly restrictive.
- **Use Descriptive Names**: Concepts should be named clearly to indicate what requirements they express, improving readability and maintainability.
- **Backwards Compatibility**: While concepts provide a powerful new tool, not all existing codebases will immediately adopt them. Consider how concepts will interact with existing templates and code.

Limitations

- **Concept Complexity**: Defining complex concepts can sometimes lead to intricate and difficult-to-understand constraints, especially for beginners.
- **Compiler Support**: While most modern compilers support concepts, some older compilers may not, potentially limiting portability.

Ranges

C++20 introduced the Ranges library, providing a new, more expressive way to work with sequences of data. Ranges build upon and enhance the existing STL iterators and algorithms by introducing range-based views, actions, and a more consistent and safer way to manipulate data. This modern iteration model allows developers to write cleaner, more readable, and more maintainable code.

1. **Ranges**: Represent sequences of elements, typically defined by a pair of iterators or by an iterable object.
2. **Views**: Lazily-evaluated transformations of ranges, providing a way to modify or filter data without creating copies.
3. **Actions**: Operations that can modify a range in place.
4. **Range Algorithms**: Algorithms that operate on ranges instead of iterators.

Basic Syntax

A range can be any object that can be iterated over, similar to how iterators work but encapsulated in a more user-friendly interface. Views are created using a pipe (`|`) syntax, allowing for a fluent and readable transformation of ranges.

Ranges

Example 1: Basic usage of ranges and views

```
#include <iostream>
#include <vector>
#include <ranges>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    // Create a view that filters even numbers and squares them
    auto even_squares = numbers | std::ranges::views::filter([](int n) { return n % 2 == 0; })
                                | std::ranges::views::transform([](int n) { return n * n; });

    // Iterate over the view and print the results
    for (int n : even_squares) {
        std::cout << n << ' ';
    }
    std::cout << '\n';  // Outputs: 4 16 36 64 100

    return 0;
}
```

Ranges

Advantages of the Ranges Library

1. **Lazy Evaluation**: Views are evaluated lazily, avoiding unnecessary computations and memory allocations. Operations are only performed when the data is accessed.
2. **Composability**: The pipe (`|`) syntax allows chaining multiple operations together in a clear and concise way, making the code more readable.
3. **Safety and Expressiveness**: The Ranges library provides a safer interface by reducing iterator manipulation errors. It also offers a rich set of expressive tools for transforming and querying data.
4. **Consistency**: Ranges offer a consistent approach to iteration, replacing the mix of iterator-based and range-based algorithms with a unified model.

Use Cases for Ranges

1. **Data Filtering and Transformation**: Easily filter and transform sequences of data using a declarative style.
2. **Range-Based Algorithms**: Utilize algorithms that operate directly on ranges, making code more concise and avoiding common pitfalls associated with iterators.
3. **Functional Programming Style**: Embrace a more functional style of programming with operations like `map`, `filter`, and `reduce` using views and actions.

Ranges

Example 2: Composing multiple views

```
#include <iostream>
#include <vector>
#include <ranges>

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    // Compose views: filter even numbers, transform by squaring, then take the first three
    auto result = data | std::ranges::views::filter([](int n) { return n % 2 == 0; })
                       | std::ranges::views::transform([](int n) { return n * n; })
                       | std::ranges::views::take(3);

    for (int n : result) {
        std::cout << n << ' ';
    }
    std::cout << '\n';  // Outputs: 4 16 36

    return 0;
}
```

Ranges

Considerations and Best Practices

- **Efficiency**: While views are efficient due to lazy evaluation, be aware of the complexity of the operations being performed, especially when chaining many views together.
- **Temporary Lifetimes**: Be cautious when creating views from temporary containers, as the underlying data must outlive the view.
- **Compatibility**: Some older codebases may not yet fully integrate with the Ranges library, requiring careful consideration when refactoring existing code.

Limitations

- **Complexity in Debugging**: The lazy and compositional nature of ranges can sometimes make debugging more complex, as operations are deferred until the data is accessed.
- **Learning Curve**: There is a learning curve associated with understanding and effectively using the new range-based approach, especially for developers accustomed to traditional iterators.

Coroutines

Key Concepts and Syntax

Coroutines in C++ are functions that can suspend their execution to be resumed later. They are defined using the special keywords `co_await`, `co_yield`, and `co_return`. A function becomes a coroutine if it uses any of these keywords.

Basic Syntax

To define a coroutine, the function must return a coroutine type, which is a type supporting the coroutine API. This typically involves:

1. **Promise Type**: Defines the behavior of the coroutine.
2. **Awaitable and Awaiter**: Types used with `co_await` to suspend and resume execution.
3. **Coroutine Handle**: Represents a handle to the coroutine, allowing control over its execution.

Coroutines

Example 1: Basic Coroutine with `co_await`

```
#include <iostream>
#include <future>
#include <coroutine>
#include <thread>

struct Awaitable {
    std::future<void> future;

    bool await_ready() const noexcept { return future.wait_for(std::chrono::seconds(0)) == std::future_status::ready; }
    void await_suspend(std::coroutine_handle<> handle) {
        std::thread([this, handle] {
            future.wait();
            handle.resume();
        }).detach();
    }
    void await_resume() const noexcept {}
};

std::future<void> asyncTask() {
    std::cout << "Starting async task...\n";
    std::this_thread::sleep_for(std::chrono::seconds(2)); // Simulating delay
    std::cout << "Async task completed.\n";
}

Awaitable asyncFunction() {
    co_await asyncTask();
    std::cout << "Continuation after async task.\n";
}

int main() {
    auto result = asyncFunction();
    std::this_thread::sleep_for(std::chrono::seconds(3)); // Wait for the async task to complete
    return 0;
}
```

Coroutines

Advantages of Coroutines

1. **Asynchronous Programming**: Coroutines enable writing asynchronous code that looks and behaves similarly to synchronous code, simplifying error handling and logic flow.
2. **Lazy Evaluation**: They allow for the implementation of generators and iterators that produce values on demand, rather than computing everything upfront.
3. **Efficiency**: Coroutines can improve performance by avoiding unnecessary thread creation and context switching, using cooperative multitasking instead.

Use Cases for Coroutines

1. **Asynchronous I/O**: Efficiently handle I/O operations without blocking the main thread, useful in networking and GUI applications.
2. **Generators**: Create sequences of values lazily, producing each value on demand.
3. **Stateful Iteration**: Implement iterators that maintain state across function calls without complex state management logic.

Coroutines

Example 2: Coroutine as a Generator with `co_yield`

```
#include <iostream>
#include <coroutine>

template<typename T>
struct Generator {
    struct promise_type {
        T current_value;
        std::suspend_always yield_value(T value) noexcept {
            current_value = value;
            return {};
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        Generator get_return_object() { return Generator{std::coroutine_handle<promise_type>::from_promise(*this)}; }
        void return_void() {}
        void unhandled_exception() { std::exit(1); }
    };
    using handle_type = std::coroutine_handle<promise_type>;
    handle_type coro;
    Generator(handle_type h) : coro(h) {}
    ~Generator() { if (coro) coro.destroy(); }

    T next() {
        coro.resume();
        return coro.promise().current_value;
    }

    bool done() { return !coro || coro.done(); }
};

Generator<int> countUpTo(int n) {
    for (int i = 0; i <= n; ++i) {
        co_yield i;
    }
}

int main() {
    auto gen = countUpTo(5);
    while (!gen.done()) {
        std::cout << gen.next() << ' '; // Outputs: 0 1 2 3 4 5
    }
    std::cout << '\n';
    return 0;
}
```

Coroutines

Considerations and Best Practices

- **Resource Management**: Properly manage resources, especially when coroutines interact with external resources like file handles or network connections.
- **Coroutine Lifetime**: Be mindful of the coroutine's lifetime and ensure that objects used within the coroutine outlive its execution.
- **Error Handling**: Handle exceptions within coroutines appropriately, ensuring they do not propagate in unexpected ways.

Limitations

- **Complexity**: The use of coroutines introduces additional complexity in understanding control flow, especially when managing coroutine state and lifetimes.
- **Compiler and Tooling Support**: While modern compilers support coroutines, not all tooling (e.g., debuggers) may fully support coroutine inspection and debugging.
- **Performance Considerations**: While coroutines can be efficient, improper use or excessive coroutine creation can lead to performance overhead.

Modules

Key Concepts and Syntax

Modules in C++ provide a way to package and import code in a more structured manner compared to header files. A module is defined by a `module` declaration and can export certain parts of its content, making them available to other translation units.

Basic Syntax

- **Module Declaration**: Starts with the `module` keyword, followed by the module's name.
- **Export Declaration**: Indicates which parts of the module are made available for other translation units.
- **Import Declaration**: Uses the `import` keyword to use a module's exported content in another translation unit.

Modules

Example 1: Defining and using a simple module

```
// math_operations.cppm
module math_operations;

export int add(int a, int b) {
    return a + b;
}

export int multiply(int a, int b) {
    return a * b;
}
```

```
// main.cpp
import math_operations;
#include <iostream>

int main() {
    std::cout << "3 + 4 = " << add(3, 4) << '\n';       // Outputs: 3 + 4 = 7
    std::cout << "3 * 4 = " << multiply(3, 4) << '\n';  // Outputs: 3 * 4 = 12
    return 0;
}
```

Modules

Advantages of Modules

1. **Improved Compilation Speed**: Modules can significantly reduce compilation times by eliminating the need to reparse headers and by enabling better caching of compiled interface units.
2. **Better Encapsulation**: Modules provide a clearer boundary for interfaces and implementations, preventing unintended dependencies and reducing the risk of macro collisions.
3. **Simplified Dependency Management**: The `import` mechanism replaces the textual inclusion of header files, making dependencies more explicit and easier to manage.
4. **Enhanced Type Safety**: Modules eliminate the preprocessor's ability to interfere with types and declarations, leading to safer and more predictable code.

Use Cases for Modules

1. **Library Development**: Modules are particularly beneficial for large libraries, where they can dramatically reduce build times and improve the encapsulation of library internals.
2. **Large Codebases**: In large projects with many dependencies, modules help manage complexity by clearly defining interfaces and reducing compile-time dependencies.
3. **Preventing Macro Conflicts**: Modules isolate code from the effects of macros, reducing the risk of unintended behavior due to macro definitions.

Modules

Example 2: Internal and external partitioning in modules

```
// complex_operations.cppm
module;
#include <cmath> // This include is private to the module implementation
module complex_operations;
export double calculate_magnitude(double real, double imag) {
    return std::sqrt(real * real + imag * imag);
}
export module complex_operations_helpers;
export double calculate_phase(double real, double imag) {
    return std::atan2(imag, real);
}
```

```
// main.cpp
import complex_operations;
import complex_operations_helpers;
#include <iostream>
int main() {
    double real = 3.0, imag = 4.0;
    std::cout << "Magnitude: " << calculate_magnitude(real, imag) << '\n'; // Outputs: Magnitude: 5
    std::cout << "Phase: " << calculate_phase(real, imag) << '\n';        // Outputs: Phase: 0.927295
    return 0;
}
```

Modules

Considerations and Best Practices

- **Backward Compatibility**: While modules provide many benefits, existing codebases heavily reliant on traditional headers may require significant refactoring to adopt modules.
- **Module Boundaries**: Carefully design module boundaries to balance encapsulation and accessibility. Expose only what is necessary and keep internal details hidden.
- **Performance Considerations**: Although modules can improve compile times, the initial setup and partitioning of modules might require some additional effort and consideration of performance trade-offs.

C++20 modules offer a transformative way to organize and manage code, providing benefits in terms of compilation speed, encapsulation, and overall code clarity. They represent a significant departure from the traditional header and source file separation, aiming to streamline the development process and improve the maintainability of large codebases. As the ecosystem evolves to fully support modules, they are expected to become a foundational feature for modern C++ development.

Three-Way Comparison (Spaceship Operator `<=>`)

Key Concepts and Syntax

The three-way comparison operator `<=>` returns a value of type `std::strong_ordering`, `std::weak_ordering`, `std::partial_ordering`, or a custom comparison category depending on the types involved. These types provide a standardized way of representing the result of a comparison:

- **`std::strong_ordering`**: For total orderings where comparisons are always well-defined.
- **`std::weak_ordering`**: For types where equality and less-than comparisons are well-defined, but not necessarily substitutable.
- **`std::partial_ordering`**: For partial orderings where some values may be incomparable.

Basic Syntax

```
#include <compare>

struct MyType {
    int value;

    // Automatically generates all six comparison operators (<, >, <=, >=, ==, !=)
    auto operator<=>(const MyType&) const = default;
};
```

Three-Way Comparison (Spaceship Operator `<=>`)

Example 1: Implementing the spaceship operator

```
#include <iostream>
#include <compare>

struct Point {
    int x;
    int y;

    auto operator<=>(const Point&) const = default; // Defaulted three-way comparison
};

int main() {
    Point p1{1, 2};
    Point p2{2, 1};

    if (p1 < p2) {
        std::cout << "p1 is less than p2\n";
    } else if (p1 > p2) {
        std::cout << "p1 is greater than p2\n";
    } else {
        std::cout << "p1 is equal to p2\n";
    }

    return 0;
}
```

Three-Way Comparison (Spaceship Operator `<=>`)

Advantages of the Three-Way Comparison Operator

1. **Unified Comparison Interface**: The `<=>` operator provides a single, unified interface for all types of comparisons, reducing boilerplate code and potential errors.
2. **Consistency**: Ensures consistent and correct implementation of all comparison operators, including `<`, `>`, `<=`, `>=`, `==`, and `!=`.
3. **Performance**: Allows the compiler to optimize comparisons better, potentially improving performance.
4. **Expressiveness**: The use of comparison categories (`std::strong_ordering`, etc.) provides a clear and expressive way to define different kinds of orderings.

Use Cases for the Spaceship Operator

1. **Sorting and Searching**: Facilitates the implementation of data structures that require ordering, such as sorted containers or searching algorithms.
2. **Custom Types**: Provides an easy and efficient way to implement comparisons for custom user-defined types.
3. **Automatic Generation**: In many cases, defaulting the `<=>` operator automatically generates all necessary comparison logic, simplifying development.

Three-Way Comparison (Spaceship Operator `<=>`)

Example 2: Custom three-way comparison logic

```
#include <iostream>
#include <compare>
#include <string>

struct Person {
    std::string name;
    int age;
    auto operator<=>(const Person& other) const {
        if (auto cmp = age <=> other.age; cmp != 0) return cmp;
        return name.compare(other.name) <=> 0;
    }
    bool operator==(const Person& other) const = default;
};

int main() {
    Person alice{"Alice", 30};
    Person bob{"Bob", 30};
    if (alice < bob) {
        std::cout << "Alice is less than Bob\n";
    } else if (alice > bob) {
        std::cout << "Alice is greater than Bob\n";
    } else {
        std::cout << "Alice is equal to Bob\n";
    }
    return 0;
```

Three-Way Comparison (Spaceship Operator `<=>`)

Considerations and Best Practices

- **Use Default Where Possible**: Whenever the default comparison is suitable, prefer using `= default` to automatically generate the necessary operators.
- **Custom Ordering Logic**: For complex types or specific ordering requirements, carefully implement the `<=>` operator to ensure all fields are correctly considered.
- **Type Categories**: Understand the differences between `std::strong_ordering`, `std::weak_ordering`, and `std::partial_ordering` to use them appropriately.

Limitations

- **Partial Orderings**: Not all types can be fully ordered, especially when dealing with floating-point numbers where comparisons can involve undefined or indeterminate results.
- **Compiler Support**: Full support for the three-way comparison operator requires a modern compiler that adheres to the C++20 standard.

Calendar and Time Zone Library

Key Concepts and Features

1. `std::chrono::time_zone` and `std::chrono::zoned_time`: These classes allow handling of time zones and conversions between different time zones.
2. `std::chrono::sys_days` and `std::chrono::local_days`: Represent calendar dates in the Gregorian calendar, facilitating date arithmetic and manipulation.
3. **Date and Time Parsing/Formatting**: Functions to parse and format dates and times according to different locales and formats.
4. **Leap Seconds and Daylight Saving Time**: Comprehensive handling of complexities like leap seconds and daylight saving time transitions.

```
#include <iostream>
#include <chrono>
#include <iomanip>
int main() {
    using namespace std::chrono;
    // Get current time in system clock (UTC)
    auto now = system_clock::now();
    // Convert to calendar date (year/month/day)
    auto today = floor<days>(now);
    year_month_day ymd{today};
    std::cout << "Current date (UTC): " << ymd << '\n';
    // Convert to a specific time zone
    auto tz = locate_zone("America/New_York");
    zoned_time zt{tz, now};
    std::cout << "Current time in New York: " << zt << '\n';
    return 0;
}
```

Calendar and Time Zone Library

Advantages of the Calendar and Time Zone Library

1. **Comprehensive Time Zone Support**: Provides accurate and up-to-date information about time zones, including daylight saving time rules and historical changes.
2. **Unified API for Date and Time Manipulation**: Offers a consistent and intuitive set of tools for handling dates and times, making it easier to perform operations like addition, subtraction, and comparison.
3. **Locale-Sensitive Parsing and Formatting**: Supports formatting and parsing of dates and times according to various locales, making it easier to work with internationalization.
4. **Accuracy and Precision**: Handles edge cases such as leap seconds and transitions in daylight saving time, ensuring accurate time calculations.

Use Cases

1. **Global Applications**: Essential for applications that operate across multiple time zones, such as scheduling systems, international meetings, or global financial applications.
2. **Event Logging and Monitoring**: Accurate timestamping in different time zones is crucial for logging systems and monitoring tools.
3. **User Interfaces and Reports**: Displaying times and dates in a user-friendly format according to user preferences or locale.

Calendar and Time Zone Library

Considerations and Best Practices

- **Time Zone Database**: The library depends on an up-to-date time zone database. Ensure that your environment keeps this data current to avoid inaccuracies due to changes in time zone rules.
- **Use Strong Typing**: Leverage the strong typing provided by the library (e.g., `year_month_day`, `zoned_time`) to avoid errors and improve code clarity.
- **Understand Time Conversions**: Be aware of the implications of converting between time zones, especially around daylight saving time transitions.

Expanded constexpr, constinit

Expanded `constexpr`

Key Concepts and Features

1. **Virtual Function Calls**: C++20 allows `constexpr` functions to contain virtual function calls, provided that the calls can be resolved at compile time.
2. **`try` and `catch` Blocks**: Exception handling with `try` and `catch` blocks is permitted in `constexpr` functions, but only for exceptions thrown within the constexpr context that can be evaluated at compile time.
3. **Standard Library Extensions**: A broader range of standard library types and functions are now usable in `constexpr` contexts, including certain operations with `std::vector` and other standard containers, provided they do not involve dynamic memory allocation.

Expanded constexpr, constinit

Example: Virtual Function Calls and Exception Handling

```
#include <iostream>
#include <string>
#include <stdexcept>

class Base {
public:
    virtual constexpr int getValue() const = 0;
};

class Derived : public Base {
public:
    constexpr int getValue() const override { return 42; }
};

constexpr int safeDivide(int a, int b) {
    if (b == 0) throw std::invalid_argument("Division by zero");
    return a / b;
}

int main() {
    constexpr int result = Derived().getValue();
    std::cout << "Result: " << result << '\n'; // Outputs: Result: 42

    try {
        constexpr int div = safeDivide(10, 2);
        std::cout << "Division Result: " << div << '\n'; // Outputs: Division Result: 5

        // Uncommenting the following line will cause a compile-time error due to division by zero
        // constexpr int error = safeDivide(10, 0);
    } catch (const std::invalid_argument& e) {
        std::cout << "Error: " << e.what() << '\n';
    }

    return 0;
```

Expanded constexpr, constinit

`constinit`

Key Concepts

1. **Compile-Time Initialization Guarantee**: `constinit` ensures that a variable is initialized during program startup, not at first use.
2. **Static and Thread Storage Duration**: It is used with variables that have static or thread-local storage duration.

Expanded constexpr, constinit

Example: `constinit` Usage

```
#include <iostream>

struct S {
    int value;
    constexpr S(int v) : value(v) {}
};

constinit S global_instance(42);

int main() {
    std::cout << "Global instance value: " << global_instance.value << '\n'; // Outputs: Global instance value: 42
    return 0;
}
```

Expanded constexpr, constinit

Advantages of Expanded `constexpr` and `constinit`

1. **Improved Compile-Time Computation**: The expanded `constexpr` capabilities allow for more sophisticated compile-time logic, reducing runtime overhead and improving efficiency.
2. **Safer Static Initialization**: The `constinit` keyword prevents lazy initialization of static variables, eliminating certain classes of bugs related to uninitialized data in multi-threaded programs.
3. **Enhanced Error Checking**: `constexpr` functions with exception handling and extended standard library support enable more rigorous compile-time error checking and validation.

Considerations and Best Practices

- **Static Initialization Order**: Be mindful of initialization order issues when using `constinit`, especially when multiple static variables depend on each other.
- **Avoiding Dynamic Memory**: Remember that `constexpr` still does not allow dynamic memory allocation (`new` and `delete`).
- **Exception Handling in `constexpr`**: While `try` and `catch` are allowed, use them judiciously, as only certain exceptions can be handled at compile time.

C++ 23

Oxiane INSTITUT

Pattern Matching

Key Concepts and Features

1. **Pattern Matching Syntax**: Pattern matching in C++23 allows you to match against specific patterns in a data structure and execute code based on the match. This can simplify complex `if-else` chains and `switch` statements.
2. **Deconstruction of Complex Types**: Pattern matching enables the extraction and deconstruction of data within tuples, structs, variants, and other aggregate types.
3. **Exhaustiveness Checking**: The compiler can ensure that all possible cases are handled, which helps in writing more robust and error-free code.

```
match (expression) {
    case pattern1:
        // Code for pattern1
        break;
    case pattern2:
        // Code for pattern2
        break;
    // More cases...
    default:
        // Code if no patterns match
}
```

Pattern Matching

Example: Pattern Matching with Variants

```
#include <iostream>
#include <variant>

std::variant<int, double, std::string> getValue() {
    return 42;
}

int main() {
    auto value = getValue();
    match (value) {
        case int i:
            std::cout << "Integer: " << i << '\n';
            break;
        case double d:
            std::cout << "Double: " << d << '\n';
            break;
        case std::string s:
            std::cout << "String: " << s << '\n';
            break;
        default:
            std::cout << "Unknown type\n";
    }
    return 0;
}
```

Pattern Matching

Advantages of Pattern Matching

1. **Conciseness**: Pattern matching reduces the verbosity of `if-else` or `switch` statements by providing a more declarative syntax.
2. **Readability**: It enhances code readability, making it clear what types or structures are being matched and how they are processed.
3. **Safety**: The compiler can enforce exhaustive handling of all possible patterns, reducing the chances of unhandled cases and runtime errors.

Use Cases for Pattern Matching

1. **Handling Variants and Tuples**: Simplifies the handling of `std::variant`, `std::tuple`, and other data structures that can hold multiple types or values.
2. **State Machines**: Useful in implementing state machines where different states are represented as different patterns.
3. **Deconstructing Data**: Allows for the deconstruction of complex data structures in a more intuitive and less error-prone manner.

std::expected

1. **Success and Error Representation**: `std::expected` encapsulates the result of an operation that can either succeed (holding a value) or fail (holding an error). This makes the error handling explicit in the type system.
2. **No Exception Handling**: Unlike exceptions, `std::expected` uses regular control flow and doesn't involve stack unwinding, making it a safer and more predictable alternative in some scenarios.
3. **Simple Interface**: The interface of `std::expected` is designed to be simple and intuitive, with methods to check the status and retrieve the contained value or error.

std::expected

```
#include <iostream>
#include <expected>

std::expected<int, std::string> divide(int a, int b) {
    if (b == 0) {
        return std::unexpected{"Division by zero"};
    }
    return a / b;
}
int main() {
    auto result = divide(10, 2);
    if (result) {
        std::cout << "Result: " << *result << '\n'; // Outputs: Result: 5
    } else {
        std::cout << "Error: " << result.error() << '\n';
    }
    auto errorResult = divide(10, 0);
    if (errorResult) {
        std::cout << "Result: " << *errorResult << '\n';
    } else {
        std::cout << "Error: " << errorResult.error() << '\n'; // Outputs: Error: Division by zero
    }
    return 0;
}
```

std::expected

Advantages of `std::expected`

1. **Explicit Error Handling**: `std::expected` makes error handling a first-class citizen in the type system, reducing the risk of ignoring errors.
2. **No Exceptions**: It provides a way to handle errors without relying on exceptions, which can simplify code and improve performance, especially in systems where exception handling is expensive or undesirable.
3. **Clear Code Flow**: By returning `std::expected`, functions clearly indicate that they may fail, making the potential for failure explicit in the API.

Use Cases for `std::expected`

1. **Error-Prone Operations**: Ideal for operations that are prone to failure, such as file I/O, network operations, or arithmetic computations that might involve invalid inputs (e.g., division by zero).
2. **Function Return Values**: When a function might fail and you want to avoid the overhead and complexity of exception handling, `std::expected` provides a straightforward alternative.
3. **Complex Error Information**: Allows returning complex error information (such as error codes or messages) along with the result, improving the ability to diagnose and respond to failures.

Executor Framework

1. **Executors**: Executors are objects responsible for managing the execution of tasks. They determine when and on which thread a task should run.
2. **Schedulers**: Schedulers are a higher-level abstraction that can be used to produce executors or manage the execution timing and sequencing of tasks.
3. **Custom Execution Policies**: The Executor Framework allows developers to create custom execution policies, defining specific behaviors for task execution, such as threading models, priority handling, or resource management.
4. **Integration with Existing Asynchronous Frameworks**: Executors are designed to work seamlessly with existing asynchronous frameworks like `std::future`, `std::async`, and coroutines, providing a more consistent and flexible approach to task management.

Executor Framework Example

```
#include <iostream>
#include <execution>
#include <thread>

void printMessage(const std::string& message) {
    std::cout << message << " from thread: " << std::this_thread::get_id() << '\n';
}

int main() {
    std::execution::static_thread_pool pool(4); // Create a thread pool with 4 threads
    auto executor = pool.executor(); // Get the executor from the pool

    executor.execute([] { printMessage("Hello, world!"); });
    executor.execute([] { printMessage("Task 2 executed"); });

    pool.wait(); // Wait for all tasks to complete

    return 0;
}
```

Executor Framework

Advantages of the Executor Framework

1. **Separation of Concerns**: By decoupling task submission from task execution, the Executor Framework allows for greater flexibility and control over how tasks are managed and executed.
2. **Consistent Interface**: Provides a consistent interface for task execution, reducing the complexity of managing asynchronous work across different parts of an application.
3. **Scalability**: Executors can be tailored to specific needs, such as handling large numbers of tasks efficiently or ensuring that tasks are executed with specific timing or resource constraints.
4. **Integration with Modern C++ Features**: Executors work seamlessly with coroutines and other asynchronous constructs, making it easier to manage asynchronous workflows in C++.

Use Cases for the Executor Framework

1. **Thread Pool Management**: Easily manage a pool of threads to handle a large number of tasks, such as in server applications or real-time systems.
2. **GUI and Event-Driven Applications**: Executors can manage tasks in response to events or user actions, ensuring that tasks are executed in the appropriate context (e.g., on the main UI thread).
3. **High-Performance Computing**: Executors allow for fine-grained control over task execution, which is essential in high-performance and parallel computing environments.

Networking TS (Technical Specification)

1. **Asynchronous Networking**: The library supports asynchronous I/O operations, enabling efficient handling of multiple network connections without blocking threads.
2. **Portable Sockets API**: The Networking TS provides a portable and type-safe API for working with sockets, ensuring that network code can run across different platforms with minimal changes.
3. **Integration with Executors**: The Networking TS is designed to work seamlessly with the C++ Executor Framework, allowing network operations to be managed using executors for better control over task execution.

Networking TS (Technical Specification)

Example: Basic TCP Server

```
#include <iostream>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

int main() {
    boost::asio::io_context io_context;

    tcp::acceptor acceptor(io_context, tcp::endpoint(tcp::v4(), 8080));
    tcp::socket socket(io_context);

    std::cout << "Waiting for connection...\n";
    acceptor.accept(socket);

    std::string message = "Hello, client!";
    boost::asio::write(socket, boost::asio::buffer(message));

    std::cout << "Message sent to client.\n";
    return 0;
}
```

Reflection

1. **Compile-Time Reflection**: Allows querying type information at compile time, enabling metaprogramming and code generation based on type traits.
2. **Type Inspection**: Programs can inspect the properties of types, such as their members, base classes, and attributes, which can be used for serialization, debugging, or generating bindings.
3. **Improved Metaprogramming**: Reflection capabilities enhance metaprogramming by providing a way to write more generic and reusable code based on type characteristics.

Reflection

Example: Simple Type Inspection

```
#include <iostream>
#include <type_traits>

template<typename T>
void print_type_info() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Type is integral.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Type is floating-point.\n";
    } else {
        std::cout << "Type is neither integral nor floating-point.\n";
    }
}

int main() {
    print_type_info<int>();    // Outputs: Type is integral.
    print_type_info<double>(); // Outputs: Type is floating-point.
    print_type_info<std::string>(); // Outputs: Type is neither integral nor floating-point.
    return 0;
}
```

Improved Support for SIMD

1. **Standardized SIMD Operations**: C++23 provides a standardized way to write SIMD operations, ensuring portability and efficiency across different hardware architectures.
2. **SIMD-Friendly Algorithms**: The standard library now includes algorithms that are optimized for SIMD, making it easier to write high-performance code without resorting to platform-specific intrinsics.
3. **Vectorized Execution**: SIMD support allows loops and algorithms to be automatically vectorized, leading to significant performance improvements in data-parallel tasks.

Improved Support for SIMD

```
#include <iostream>
#include <vector>
#include <immintrin.h> // Intel intrinsics for SIMD
void simd_add(const std::vector<float>& a, const std::vector<float>& b, std::vector<float>& result) {
    size_t size = a.size();
    for (size_t i = 0; i < size; i += 8) {
        __m256 va = _mm256_loadu_ps(&a[i]); // Load 8 floats from a
        __m256 vb = _mm256_loadu_ps(&b[i]); // Load 8 floats from b
        __m256 vr = _mm256_add_ps(va, vb);  // Perform SIMD addition
        _mm256_storeu_ps(&result[i], vr);   // Store the result
    }
}
int main() {
    std::vector<float> a(16, 1.0f);
    std::vector<float> b(16, 2.0f);
    std::vector<float> result(16);
    simd_add(a, b, result);
    for (float v : result) {
        std::cout << v << ' '; // Outputs: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
    }
    std::cout << '\n';
    return 0;
}
```

Conclusion