Excellent! Let's proceed to the fascinating world of Advanced C++ Concepts.

V. Advanced C++ Concepts

- **Templates:**
  - **Explanation:** Templates are a powerful feature in C++ that allow you to write generic code that can work with different data types without having to write separate versions for each type. They provide a way to achieve compile-time polymorphism.
  - **Function Templates (Example):**

```C++
#include <iostream>

template <typename T> // 'typename' can also be 'class' here
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    std::cout << "Max of 5 and 10: " << max<int>(5, 10) << std::endl; // Explicit instantiation with int
    std::cout << "Max of 2.5 and 1.7: " << max(2.5, 1.7) << std::endl;   // Type deduction for double
    std::cout << "Max of 'a' and 'c': " << max('a', 'c') << std::endl;   // Type deduction for char
    return 0;
}
```

  - **Class Templates (Example):**

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

```cpp
#include <string>

template <typename T>
class Container {
private:
    T data;
public:
    Container(T value) : data(value) {}
    void display() const {
        std::cout << "Data: " << data << std::endl;
    }
    T getValue() const {
        return data;
    }
};

int main() {
    Container<int> intContainer(42);
    intContainer.display();
    std::cout << "Value: " << intContainer.getValue() << std::endl;

    Container<std::string> stringContainer("Hello");
    stringContainer.display();
    std::cout << "Value: " << stringContainer.getValue() << std::endl;
    return 0;
}
```

- **Template Argument Deduction and Explicit Instantiation:**
  - **Template Argument Deduction:** The compiler often can automatically deduce the type arguments for a function template based on the types of the arguments passed in the function call (as seen in the

max(2.5, 1.7) example above).

- **Explicit Instantiation:** You can explicitly specify the type arguments when calling a function template (e.g., max<int>(5, 10)). You can also explicitly instantiate a template for a specific type outside of its use to force compilation for that type.

```C++
template int max<int>(int a, int b); // Explicit instantiation
```

- **Metaprogramming with Templates (Brief Introduction):** A technique of using templates to perform computations at compile time rather than runtime. This can lead to more efficient code as some calculations are done before the program even starts. Template metaprogramming often involves recursion through templates.

```C++
#include <iostream>

template <int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template <> // Template specialization for the base case
struct Factorial {
    static const int value = 1;
```

```
};

int main() {
    constexpr int result = Factorial::value; // Calculated at compile time
    std::cout << "Factorial of 5: " << result << std::endl;
    return 0;
}
```

- Variadic Templates (C++11): Templates that can take a variable number of arguments of any type. They are often used in conjunction with recursion or fold expressions to process the arguments.

```
C++
#include <iostream>

template <typename T>
void print(T arg) {
    std::cout << arg << std::endl;
}

template <typename T, typename... Args>
void print(T arg, Args... args) {
    std::cout << arg << " ";
    print(args...); // Recursive call
}

int main() {
    print(1, 2.5, "hello", 'c');
    return 0;
}
```

- Concepts (C++20): Provide a way to specify constraints on template parameters. They allow you to define what requirements a type must satisfy to be used with a particular

template. This leads to more readable error messages and more robust template code.

```C++
#include <iostream>
#include <concepts>

template <typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

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

int main() {
    std::cout << "Sum of integers: " << add(5, 3) << std::endl;
    std::cout << "Sum of doubles: " << add(2.5, 1.7) << std::endl;
    // std::cout << add("hello", "world") << std::endl; // Error: "hello" is not Numeric
     return 0;
}
```

- Standard Template Library (STL):
  - Explanation: A powerful set of template classes and functions that provide common data structures and[1] algorithms. It significantly simplifies and speeds up C++ development by providing ready-to-use components.
  - Containers: Objects that store collections of other objects.
    - Sequence Containers: Store elements in a linear order.
      - std::vector: Dynamically resizable array.

Efficient for random access and adding/removing elements at the end.

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

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    numbers.push_back(6);
    std::cout << "Size: " << numbers.size() << std::endl;
    std::cout << "Element at index 2: " << numbers[2] << std::endl;
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

- std::list: Doubly linked list. Efficient for inserting and deleting elements anywhere in the list.

```C++
#include <iostream>
#include <list>

int main() {
    std::list<std::string> names = {"Alice", "Bob", "Charlie"};
    names.push_front("David");
    names.push_back("Eve");
    for (const std::string& name : names) {
        std::cout << name << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

- std::deque: Double-ended queue. Supports efficient insertion and deletion at both the

front and the back.

```C++
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq = {2, 3, 4};
    dq.push_front(1);
    dq.push_back(5);
    std::cout << "Front: " << dq.front() << ", Back: " << dq.back() << std::endl;
    return 0;
}
```

- std::forward_list (C++11): Singly linked list. More memory-efficient than std::list but only allows forward iteration.
- std::array (already covered in Fundamentals): Fixed-size array wrapper.

- Associative Containers: Store elements in a sorted manner (based on keys).
  - std::set: Stores unique elements in a sorted order.

```C++
#include <iostream>
#include <set>

int main() {
    std::set<int> uniqueNumbers = {3, 1, 4, 1, 5, 9, 2, 6};
    for (int num : uniqueNumbers) {
        std::cout << num << " "; // Output: 1 2 3 4 5 6 9
    }
    std::cout << std::endl;
    return 0;
}
```

        - std::multiset: Stores elements in a sorted order and allows duplicate elements.
        - std::map: Stores key-value pairs where keys are unique and sorted.

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

int main() {
    std::map<std::string, int> ages;
    ages["Alice"] = 30;
    ages["Bob"] = 25;
    ages["Charlie"] = 35;
    std::cout << "Age of Bob: " << ages["Bob"] << std::endl;
    return 0;
}
```

        - std::multimap: Stores key-value pairs where keys are sorted and can be duplicated.
    - Unordered Associative Containers (C++11): Store elements in an unordered manner for potentially faster average-case access (using hash tables).
        - std::unordered_set, std::unordered_multiset, std::unordered_map, std::unordered_multimap.
- Adaptors: Provide a different interface to existing containers.
    - std::stack: LIFO (Last-In, First-Out) data

structure, typically implemented using std::deque.

```C++
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
     s.push(10);
    s.push(20);
    s.push(30);
    std::cout << "Top element: " << s.top() << std::endl; // Output: 30
    s.pop();
    std::cout << "Top element after pop: " << s.top() << std::endl; // Output: 20
     return 0;
}
```

- std::queue: FIFO (First-In, First-Out) data structure, typically implemented using std::deque.
- std::priority_queue: Elements are stored such that the largest (by default) element is always at the top.

- Algorithms: Functions that operate on ranges of elements (often provided by containers). They perform various operations like sorting, searching, transforming, etc. You usually need to include the <algorithm> header.

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

```
int main() {
    std::vector<int> nums = {5, 2, 8, 1, 9};
    std::sort(nums.begin(), nums.end());

std::cout << "Sorted numbers: ";for (int num : nums) {std::cout << num << " "; // Output: 1 2 5 8 9}std::cout << std::endl;     auto it = std::find(nums.begin(), nums.end(), 8);
    if (it != nums.end()) {
        std::cout << "Found 8 at index: " << std::distance(nums.begin(), it) << std::endl;
    }

    std::vector<int> squaredNums(nums.size());
    std::transform(nums.begin(), nums.end(), squaredNums.begin(),(int n){ return n * n; });
    std::cout << "Squared numbers: ";
    for (int num : squaredNums) {
        std::cout << num << " "; // Output: 1 4 25 64 81
    }
    std::cout << std::endl;
    return 0;
}
```

- Iterators: Objects that provide a way to access elements within a container. They act like pointers but are more general and work with different types of containers. Common iterator types include input, output, forward, bidirectional, and random access iterators.

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

int main() {
    std::vector<int> data = {10, 20, 30};
    for (std::vector<int>::iterator it = data.begin(); it != data.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    for (auto it = data.cbegin(); it != data.cend(); ++it) { // Constant iterator
```

```
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    for (auto& element : data) { // Range-based for loop uses iterators internally
        std::cout << element << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

- Function Objects (Functors): Objects that can be called like functions (they overload the function call operator `operator()`). They can store state and often used with STL algorithms to customize behavior.

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

class MultiplyBy {
private:
    int factor;
public:
    MultiplyBy(int f) : factor(f) {}
    int operator()(int num) const {
        return num * factor;
    }
};

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};
    std::vector<int> multipliedNums(nums.size());
    std::transform(nums.begin(), nums.end(), multipliedNums.begin(), MultiplyBy(3));
    std::cout << "Multiplied numbers: ";
    for (int num : multipliedNums) {
        std::cout << num << " "; // Output: 3 6 9 12 15
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

    - Allocator Class: Defines how memory is allocated and deallocated for a container. You can provide custom allocators if you have specific memory management requirements, but the default allocator is sufficient for most cases.
- Exception Handling:
    - Explanation: A mechanism to handle runtime errors (exceptions) that occur d[2]uring program execution. It allows you to gracefully recover from errors and prevent program crashes.
    - try, catch, throw (Example):

```
C++
#include <iostream>
#include <stdexcept>

int divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero!");
    }
    return a / b;
}

int main() {
    int x = 10, y = 0;
    try {
        int result = divide(x, y);
        std::cout << "Result: " << result << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << std::endl; // Output: Error: Division by zero!
```

```
    }
    std::cout << "Program continues after the try-catch block." << std::endl;
    return 0;
}
```

- Standard Exception Hierarchy: C++ provides a hierarchy of standard exception classes derived from `std::exception` (defined in `<stdexcept>`). Common ones include `std::runtime_error`, `std::logic_error`, `std::out_of_range`, `std::bad_alloc`, etc.
- Custom Exception Classes: You can create your own exception classes by inheriting from `std::exception`[3] or one of its derived classes. It's good practice to override the `what()` virtual member function to provide a custom error message.

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

class MyCustomError : public std::runtime_error {
public:
    MyCustomError(const std::string& message) : std::runtime_error(message) {}
};

void processData(int value) {
    if (value < 0) {
        throw MyCustomError("Value cannot be negative.");
    }
    std::cout << "Processing value: " << value << std::endl;
}
```

```
int main() {
    try {
        processData(10);
        processData(-5);
    } catch (const MyCustomError& e) {
        std::cerr << "Custom error caught: " << e.what() << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Standard runtime error caught: " << e.what() << std::endl;
    }
    return 0;
}
```

- Stack Unwinding: When an exception is thrown, the program searches up the call stack for a matching `catch` handler. During this process, the destructors of any automatic objects (local variables) in the scopes being exited are called. This is important for resource management (RAII).
- Exception Safety (No-Throw, Strong, Basic Guarantees): Refers to the level of guarantee a function provides in the face of exceptions.
  - No-throw guarantee: The function will not throw any exceptions (often indicated with `noexcept`).
  - Strong exception safety: If an exception is thrown, the program state remains as it was before the function was called

(transactional behavior).
        - Basic exception safety: If an exception is thrown, the program state is valid, and no resources are leaked, but the state might not be exactly as it was before the call.
    - Rethrowing Exceptions: A `catch` block can rethrow the exception it caught (or a different exception) using the `throw;` statement (without specifying an exception object). This is often used when a handler needs to do some local cleanup but wants a higher-level handler to deal with the error.
- Move Semantics and Rvalue References (C++11):
    - Explanation: Introduced to improve performance by avoiding unnecessary copying of objects, especially when dealing with temporary objects (rvalues).
    - Rvalue References (`&&`): A new type of reference that can bind to rvalues (temporary objects or values that are about to expire). This allows you to "move" resources from an rvalue to another object instead of making a deep copy.

- # Move Constructor and Move Assignment Operator (Transfer of Ownership - Example):

```
C++
#include <iostream>
#include <string>
#include <utility> // For std::move

class MyString {
private:
    char* data;
    size_t length;
public:
    MyString(const char* str = "") : length(std::strlen(str)) {
        data = new char[length + 1];
        std::strcpy(data, str);
        std::cout << "Constructor for: " << data << std::endl;
    }
    ~MyString() {
        std::cout << "Destructor for: " << (data ? data : "(null)") << std::endl;
        deletedata;
    }
    MyString(const MyString& other) : length(other.length) { // Copy constructor (deep copy)
        data = new char[length + 1];
        std::strcpy(data, other.data);
        std::cout << "Copy constructor for: " << data << std::endl;
    }
    MyString(MyString&& other) noexcept : data(other.data), length(other.length) { // Move
constructor
        other.data = nullptr;
        other.length = 0;
        std::cout << "Move constructor for: " << data << std::endl;
    }
    MyString& operator=(const MyString& other) { // Copy assignment
```

std::cout << "Copy assignment for: " << other.data << std::endl;if (this != &other) {deletedata;length = other.length;4data = new char[length + 1];std::strcpy(data, other.data);}return *this;}MyString& operator=(MyString&& other) noexcept { // Move assignmentstd::cout << "Move assignment for: " << other.data << std::endl;if (this != &other) {deletedata;data = other.data;length = other.length;other.data = nullptr;other.length = 0;}return this;}const char getString() const { return data; }};

```
MyString createString() {
    MyString temp("Temporary String");
    return temp; // Move constructor will be used (RVO might also occur)
```

```
}

int main() {
    MyString str1 = createString();
    MyString str2 = std::move(str1); // Explicitly move from str1
    std::cout << "str2: " << str2.getString() << std::endl;
    // std::cout << "str1: " << str1.getString() << std::endl; // Might be empty or invalid

    return 0;
}
```

    - Perfect Forwarding (std::forward): A mechanism that allows you to forward arguments to another function while preserving their original value category (whether they are lvalues or rvalues). This is crucial when writing generic functions or templates that need to work with both types of arguments efficiently.
- Concurrency and Multithreading:
    - Explanation: The ability of a program to execute multiple tasks (threads) seemingly in parallel. This can improve performance on multi-core processors.
    - Threads (std::thread - Example):

```
C++
#include <iostream>
#include <thread>
#include <chrono>

void task(const std::string& message) {
    std::cout << "Thread started: " << message << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "Thread finished: " << message << std::endl;
```

```
}

int main() {
    std::thread t1(task, "Task 1");
    std::thread t2(task, "Task 2");

    std::cout << "Main thread continues..." << std::endl;

    t1.join(); // Wait for t1 to finish
    t2.join(); // Wait for t2 to finish

    std::cout << "All threads finished." << std::endl;
    return 0;
}
```

- Mutexes (std::mutex): Used to protect shared resources from being accessed by multiple threads simultaneously, preventing race conditions.
- Locks (std::lock_guard, std::unique_lock): RAII wrappers for mutexes that automatically lock the mutex when they are created and unlock it when they go out of scope. std::unique_lock provides more flexibility than std::lock_guard (e.g., deferred locking, timed locking).
- Condition Variables (std::condition_variable): Allow threads to wait until a certain condition is true. They are typically used in conjunction with a mutex.
- Futures and Promises (std::future, std::promise):

Provide a way to get the result of an asynchronous operation. A promise can set a value (or an exception) that can later be retrieved by a future.

- Atomic Operations (std::atomic): Provide operations on primitive data types that are guaranteed to be performed indivisibly, even in the presence of multiple threads. This avoids the need for explicit locking in some simple cases.
- Synchronization Primitives (Semaphores, Barriers, Readers-Writer Locks): More advanced synchronization tools for controlling access to shared resources and coordinating the execution of multiple threads. These are often provided by the operating system or external libraries.
- C++ Concurrency Memory Model (Brief Mention): Defines how memory operations in different threads interact with each other, especially in terms of visibility and ordering. It's a complex topic that ensures consistent behavior in concurrent programs.

    - Task-Based Parallelism with `std::async`: Provides a higher-level way to run tasks asynchronously (potentially in parallel) and get their results using futures.
- Namespaces (Nested, Unnamed):
    - Nested Namespaces: You can define namespaces within other namespaces to create a hierarchical organization of your code.

```C++
namespace Outer {
    namespace Inner {
        void someFunction() {
            std::cout << "Function in Outer::Inner namespace." << std::endl;
        }
    } // namespace Inner
} // namespace Outer

int main() {
    Outer::Inner::someFunction();
    return 0;
}
```

    - Unnamed Namespaces (already covered in Fundamentals): Provide internal linkage for the symbols they contain, making them local to the current translation unit (source file).
- Preprocessor Directives:
    - Explanation: Instructions to the C++ preprocessor, which runs before the actual

compilation. They are typically used for including header files, defining macros, and conditional compilation.

- #include: Includes the contents of another file (usually a header file) into the current source file.
  - #include <filename>: Used for standard library headers (searches in standard include directories).
  - #include "filename": Used for user-defined headers (searches in the current directory and other project-specific directories).
- #define: Creates a macro (a symbolic name or parameterized replacement for a sequence of characters).

```C++
#define PI 3.14159
#define SQUARE(x) ((x) * (x)) // Use parentheses to avoid unexpected behavior
```

- Conditional Compilation Directives (#ifdef, #ifndef, #else, #endif): Allow you to conditionally compile parts of your code based on whether certain macros are defined.

```C++
#include <iostream>

#define DEBUG // Define the DEBUG macro
```

```cpp
int main() {
#ifdef DEBUG
    std::cout << "Debugging mode is enabled." << std::endl;
#else
    std::cout << "Debugging mode is disabled." << std::endl;
#endif

#ifndef RELEASE
    std::cout << "This is not a release build." << std::endl;
#endif

    return 0;
}
```

This concludes our exploration of Advanced C++ Concepts. We have covered a lot of ground, from templates and the STL to exception handling, concurrency, and preprocessor directives. The next section will focus on Input/Output in C++. Let me know if you have any questions about any of these advanced topics.

Sources

1. https://github.com/HatsuneQixeo/cpp-experiment
2. https://blog.csdn.net/qq_33243369/article/details/131222773
3. https://algocademy.com/blog/c-data-types-a-comprehensive-guide-for-beginners-and-advanced-programmers/
4. https://velog.io/@29th_challenger/STL
5. https://github.com/HH-Tushar/Problem_Solving
6. http://www.interviewbyte.in/interview_prep/interview/Cpp/cpp_functors
7. https://hellointern.in/blog/top-interview-questions-and-answers-for-java-developer-internship-54
8. https://blog.csdn.net/qwsbcjendb/article/details/133706481
9. https://blog.nexotips.com/tag/cpplogic/feed/
10. https://github.com/Prushchuk/operator-overload