

Interview Review Chart

Karl Solomon

December 27, 2024

1 Parallelism

1.1 Concepts

Concurrency When two or more tasks can start/run/complete in overlapping time periods. This does not necessarily mean they'll be running in overlapping time periods. Examples include:
RTOS

Parallelism When tasks run at the same time (e.g. on a multi-core CPU)

Multithreading When multiple tasks are running on a CPU. This can be implemented truly parallel where each task has access to separate HW/core. However, more common in desktop applications is SMT.

SMT (Simultaneous Multithreading) Multiple threads share 1 core. The thread instructions are pipelines s.t. they run mostly in-parallel, and when one is waiting for I/O the other can run uninhibited, however since only one thread can access a dedicated HW block at any given time they are not truly parallel.

1.2 Implementation

- Concurrency and Multithreading

- Basic concepts of threads and processes
- Thread synchronization mechanisms (mutexes, semaphores, locks)
- Race conditions:
 1. Multiple threads accessing a shared resource
 2. At least one thread writes to the resource
 3. Lack of proper synchronization
- deadlocks

```
1 std::mutex m1, m2;  
2 // Must have 2 locks where they are locked in different orders  
  in different locations/threads  
3 thread1: m1.lock(); m2.lock(); m1.unlock(); m2.unlock();  
4 thread2: m2.lock(); m1.lock(); m2.unlock(); m1.unlock();
```

- Atomic operations

```
1 std::atomic<T> var; // where T is a primitive type  
2 var = val;  
3 var.load(val);  
4 var.store(val);  
5 var.wait(); // waits until the value changes
```

```

6 T current = var.exchange(val); // writes val to var and gets
  previous/current value of var
7 // Compare-and-swap. Atomically compares object.value with
  that of expected. If bitwise-equal then replaces former
  with desired. Otherwise loads actual value into expected (
  via load operation)
8 bool res = var.compare_exchange_strong(expected, desired); //
  preferred when don't expect high contention and cost of
  retrying is significant. Simpler, but slower.
9 bool res = var.compare_exchange_weak(expected, desired); //
  preferred if you're anyways retrying in a loop or cost of
  retrying is low. More efficient, but more complex.

```

```

1 // Weak usage
2 std::atomic<int> value{0};
3 int expected = 0;
4 int desired = 1;
5 while(!value.compare_exchange_weak(expected, desired)){
6     ; // handle spurious failures
7 }
8
9 // Strong usage
10 if(value.compare_exchange_strong(expected, desired)){
11     // op successful
12 }
13 else
14 {
15     // op failed
16 }

```

- Thread-Safe Data Structures:

- Concurrent collections
- Concurrent collections (e.g., ConcurrentQueue, ConcurrentBag)
- Lock-free data structures
- Understanding the differences between thread-safe and non-thread-safe collections

- Design Patterns for Concurrency:

- Producer-Consumer pattern
- Readers-Writer pattern
- Thread pool pattern

- Language-Specific Concurrency Features for C++:

- std::thread
- <atomic>

- Callback Mechanisms:

- Function pointers

```

1 #include <stdio.h>
2
3 // Define a struct with function pointers for arithmetic
  operations
4 typedef struct {
5     int (*add)(int, int);
6 } ArithmeticOperations;
7

```

```

8 // Define the functions for arithmetic operations
9 int add(int a, int b) { return a + b; }
10
11 int main() {
12     // Initialize the struct with function pointers
13     ArithmeticOperations ops;
14     ops.add = add;
15     // Use the function pointers to perform operations
16     int x = 10, y = 5;
17     printf("Add: %d + %d = %d\n", x, y, ops.add(x, y));
18     return 0;
19 }

```

- Delegates (in languages that support them)
- Lambda expressions

- Performance Considerations:

- Understanding the overhead of different synchronization mechanisms
Generally best practice to measure performance in single-threaded vs multithreaded/parallel environments. Since there is overhead with creating/cleaning up threads/processes it can make your program run slower in smaller data sets.
- Balancing thread safety with performance

- Testing Multithreaded Code:

- Techniques for writing unit tests for concurrent code
- Tools for detecting race conditions and deadlocks
 - * **Helgrind:** Part of Valgrind suite. Checks for race conditions, but slow.
 - * **ThreadSanitizer:** Compiler flag in llvm/clang. Faster than Helgrind.
 - * **RacerD:** Meta's C++-specific concurrent static analyzer. Good for target code-bases.
 - * **Clang Static Analyzer:** Detects some simple conditions.

- Distributed Systems Concepts:

While not directly related to this problem, understanding concepts like eventual consistency and distributed locking can be beneficial

- Algorithms for Concurrent Operations:

- Compare-And-Swap (CAS) operations
- Lock-free algorithms

- Memory Models:

- Understanding memory barriers and volatile variables
- Cache coherence issues in multi-core systems

Cache Coherence The process of ensuring that data is stored in multiple caches within a multiprocessor system is consistent and synchronized.

This ensures that all processors have a *consistent* view of shared memory. Cache coherence protocols manage the flow of data between caches, updating cache lines and tracking the status of shared data. This can be complicated because it requires balancing performance and coherence overhead. The 2 main types of protocols are Directory-based and Snoop-based.

Directory-based The sharing status of a block of physical memory is kept in just one location (the directory). The directory can also be distributed to improve scalability. Communication is established using point-to-point requests through the interconnection network.

Snoop-based Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. Caches are all accessible via some broadcast medium (a bus or switch), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access. Requires broadcast, since caching information is at processors. This is useful for small-scale machines.

Point of Coherency (PoC) Point at which all agents in a system which can access memory are guaranteed to see the same data.

Migration Data is migrated to the local cache levels.

Replication The same data is replicated across all caches.

Assume Snoop-based protocol. There are 2 ways to maintain coherence:

1. **Write Invalidate Protocol:** Ensure that a processor has exclusive access to a data item before it writes that item. This is most common protocol.
2. **Write Broadcast/Update:** All cached copies are updated simultaneously. This requires more bandwidth. When multiple updates happen to the same location, unnecessary updates are done. However, this is a lower latency between write/read.

- Practice Problems:

- Implement a thread-safe singleton

```
1 #include <iostream>
2 #include <mutex>
3
4 class Singleton {
5 public:
6     // Delete copy constructor and assignment operator to
        // prevent creation of additional instances through
        // copying
7     Singleton(const Singleton&) = delete;
8     Singleton& operator=(const Singleton&) = delete;
9
10    // Static method to get the instance of the singleton
11    static Singleton& getInstance() {
12        static Singleton instance; // Static Local variables
        // are guaranteed to be thread-safe in C++11 and later
13        return instance;
14    }
15
16    void showMessage() { std::cout << "Singleton instance
        accessed!" << std::endl; }
17
18 private:
19    // Private constructor to prevent instantiation
20    Singleton() { std::cout << "Singleton instance created!"
        << std::endl; }
21 };
22
23 int main() {
24     // Access the singleton instance
25     Singleton& singleton = Singleton::getInstance();
26     singleton.showMessage();
```

```

27
28     return 0;
29 }

```

– Create a simple producer-consumer queue

```

1  #include <chrono>
2  #include <condition_variable>
3  #include <iostream>
4  #include <mutex>
5  #include <queue>
6  #include <thread>
7
8  class ThreadSafeQueue {
9  public:
10     void enqueue(int item) {
11         std::lock_guard<std::mutex> lock(mutex_);
12         queue_.push(item);
13         cond_var_.notify_one();
14     }
15
16     int dequeue() {
17         std::unique_lock<std::mutex> lock(mutex_);
18         cond_var_.wait(lock, [this] { return !queue_.empty(); });
19         int item = queue_.front();
20         queue_.pop();
21         return item;
22     }
23
24 private:
25     std::queue<int> queue_;
26     std::mutex mutex_;
27     std::condition_variable cond_var_;
28 };
29
30 void producer(ThreadSafeQueue& queue, int numItems) {
31     for (int i = 0; i < numItems; ++i) {
32         std::this_thread::sleep_for(std::chrono::milliseconds
33             (100)); // Simulate work
34         queue.enqueue(i);
35         std::cout << "Produced: " << i << std::endl;
36     }
37 }
38
39 void consumer(ThreadSafeQueue& queue, int numItems) {
40     for (int i = 0; i < numItems; ++i) {
41         int item = queue.dequeue();
42         std::cout << "Consumed: " << item << std::endl;
43     }
44 }
45
46 int main() {
47     ThreadSafeQueue queue;
48     const int numItems = 10;
49
50     std::thread producerThread(producer, std::ref(queue),
51         numItems);
52     std::thread consumerThread(consumer, std::ref(queue),
53         numItems);
54
55     producerThread.join();
56     consumerThread.join();

```

```

55     return 0;
56 }

```

– Implement a basic thread pool

```

1  #include <execution> // Required for std::execution::seq
2  #include <iostream>
3  #include <numeric>
4  #include <random>
5  #include <thread>
6  #include <vector>
7
8  void accumulateRandomNumbers(int threadID) {
9      static std::mutex m;
10     std::random_device rd;
11     std::mt19937 gen(rd());
12     std::uniform_real_distribution<double> dis(0.0, 1.0);
13
14     std::vector<double> numbers(1024 * 1024);
15     for (auto& num : numbers) {
16         num = dis(gen);
17     }
18
19     double sum = std::reduce(std::execution::seq, numbers.
20         begin(), numbers.end());
21     std::lock_guard<std::mutex> lock(m);
22     std::cout << "Thread " << threadID << " accumulated sum: "
23         << sum << std::endl;
24 }
25
26 int main() {
27     const int numThreads = std::thread::hardware_concurrency()
28         ; // Number of threads in the pool
29     std::vector<std::thread> threadPool;
30
31     // Create and launch threads
32     for (int i = 0; i < numThreads; ++i) {
33         threadPool.emplace_back(accumulateRandomNumbers, i);
34     }
35
36     // Join threads to the main thread
37     for (auto& thread : threadPool) {
38         thread.join();
39     }
40
41     return 0;
42 }

```

– Solve classic concurrency problems like the dining philosophers problem

```

1  #include <chrono>
2  #include <condition_variable>
3  #include <iostream>
4  #include <mutex>
5  #include <thread>
6  #include <vector>
7
8  class DiningPhilosophers {
9  public:
10     DiningPhilosophers(int numPhilosophers)
11         : numPhilosophers_(numPhilosophers), states_(
12             numPhilosophers, State::THINKING) {}

```

```

13     void philosopher(int id) {
14         while (true) {
15             think(id);
16             pickUpForks(id);
17             eat(id);
18             putDownForks(id);
19         }
20     }
21
22 private:
23     enum class State { THINKING, HUNGRY, EATING };
24
25     void think(int id) {
26         std::cout << "Philosopher " << id << " is thinking."
27             << std::endl;
28         std::this_thread::sleep_for(std::chrono::milliseconds
29             (1000));
30     }
31
32     void eat(int id) {
33         std::cout << "Philosopher " << id << " is eating." <<
34             std::endl;
35         std::this_thread::sleep_for(std::chrono::milliseconds
36             (1000));
37     }
38
39     void pickUpForks(int id) {
40         std::unique_lock<std::mutex> lock(mutex_);
41         states_[id] = State::HUNGRY;
42         cond_var_.wait(lock, [this, id] { return canEat(id);
43             });
44         states_[id] = State::EATING;
45     }
46
47     void putDownForks(int id) {
48         std::unique_lock<std::mutex> lock(mutex_);
49         states_[id] = State::THINKING;
50         cond_var_.notify_all();
51     }
52
53     bool canEat(int id) {
54         int left = (id + numPhilosophers_ - 1) %
55             numPhilosophers_;
56         int right = (id + 1) % numPhilosophers_;
57         return states_[id] == State::HUNGRY && states_[left]
58             != State::EATING && states_[right] != State::EATING
59             ;
60     }
61
62     int numPhilosophers_;
63     std::vector<State> states_;
64     std::mutex mutex_;
65     std::condition_variable cond_var_;
66 };
67
68 int main() {
69     const int numPhilosophers = 5;
70     DiningPhilosophers diningPhilosophers(numPhilosophers);
71     std::vector<std::thread> threads;
72
73     for (int i = 0; i < numPhilosophers; ++i) {
74         threads.emplace_back(&DiningPhilosophers::philosopher,
75             &diningPhilosophers, i);
76     }
77 }

```

```

67     }
68
69     for (auto& thread : threads) {
70         thread.join();
71     }
72
73     return 0;
74 }

```

Write a few test cases in addition to the solution. Remember, for interviews, it's not just about knowing the solutions, but also being able to explain your reasoning, discuss trade-offs, and analyze the performance and correctness of your solutions. Lastly, be prepared to write code on a whiteboard or in a simple text editor. Practice implementing these concepts without relying on an IDE's features.

2 Operating Systems

2.1 Caches

TLB The TLB is a small, fast, and fast-access memory that is used to translate virtual memory addresses into physical memory addresses. a.k.a. Translation Lookaside Buffer.

2.2 Basic Concepts

- **TLB: Translate Lookaside Buffer**
- **Processes and Threads**
 - Process creation and termination
 - Thread lifecycle and management
- **Memory Management**
 - Virtual memory
 - Paging and segmentation
- **File Systems**
 - File system structure
 - File operations and permissions

3 C

3.1 Preprocessor

```

1 # // stringizes the macro parameter
2 #define stringify(x) #x
3 #define foo 1
4 stringify(foo) // --> evaluates to "foo", NOT "1"

```

```

1 ## // concatenates the macro parameter
2 #define COMMAND(NAME) {#NAME, NAME ## _command}
3 struct command commands[] = {
4     COMMAND(quit), // equivalent to {quit_command}
5     COMMAND(help), // equivalent to {help_command}
6 }

```


- predefined macros
 - `__FILE__`
 - `__LINE__`
 - `__DATE__`
 - `__TIME__`
 - `__STDC_VERSION__`
 - `__cplusplus`
- item2
- item3
- item4

3.2 Peripherals

• I2C

SDA is data, SCL is clock. PURs typically in the 1-4.7k range. Too weak = slow comm and errors. Clocks are usually 100k-1MHz. Addr can be 7 or 10 bit. This is rate-limiter for number of slaves, though line impedance would increase for each slave. Here are some use usage examples:

1. Master sends START and slave Addr
2. Master sends data to slave
3. Master terminates with a STOP

1. Master sends START and slave Addr
2. Master sends data to slave
3. Master sends repeatedSTART and either sends more data to slave or receives data from slave.
4. Master sends STOP

• SPI

Serial Peripheral Interface (SPI) is a synchronous serial communication protocol used for short-distance communication, primarily in embedded systems. It uses a master-slave architecture with a single master and multiple slaves. Communication is full-duplex, and it requires four wires: MOSI, MISO, SCLK, and SS.

• UART

Universal Asynchronous Receiver-Transmitter (UART) is a hardware communication protocol that uses asynchronous serial communication with configurable speed. It is commonly used for communication between microcontrollers and peripherals. UART requires only two wires: TX (transmit) and RX (receive).

• USB

Universal Serial Bus (USB) is an industry-standard for short-distance digital data communications. It supports plug-and-play installation and hot swapping. USB is used for connecting peripherals such as keyboards, mice, printers, and external storage devices to computers.

• HDMI

High-Definition Multimedia Interface (HDMI) is a proprietary audio/video interface for transmitting uncompressed video data and compressed or uncompressed digital audio data from an HDMI-compliant source device to a compatible display device. It is commonly used for connecting devices like TVs, monitors, and projectors.

4 C++

4.1 <algorithm>

- **batchOperations**

– `for_each`, `ranges::for_each`, `for_each_n`, `ranges::for_each_n`

```
1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2
3 // Use std::for_each to print each element
4 std::for_each(numbers.begin(), numbers.end(), [](int n) { std
  ::cout << n << " "; });
5
6 // Use std::for_each_n to print the first 3 elements
7 std::for_each_n(numbers.begin(), 3, [](int n) { std::cout << n
  << " "; });
8
9 // Use std::ranges::for_each to print each element
10 std::ranges::for_each(numbers, [](int n) { std::cout << n << "
  "; });
11
12 // Use std::ranges::for_each_n to print the first 3 elements
13 std::ranges::for_each_n(numbers.begin(), 3, [](int n) { std::
  cout << n << " "; });
```

- **Search Operations**

– `all_of`, `any_of`, `none_of`

```
1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 // Use std::all_of to check if all elements are positive
3 bool allPositive = std::all_of(numbers.begin(), numbers.end(),
  [](int n) { return n > 0; });
4 // Use std::any_of to check if any element is greater than 4
5 bool anyGreaterThanFour = std::any_of(numbers.begin(), numbers
  .end(), [](int n) { return n > 4; });
6 // Use std::none_of to check if no elements are negative
7 bool noneNegative = std::none_of(numbers.begin(), numbers.end
  (), [](int n) { return n < 0; });
```

– `ranges::contains`, `ranges::contains_subrange`

```
1 std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2
3 // Check if the range contains the value 5
4 bool containsFive = std::ranges::contains(numbers, 5);
5 // Check if the range contains the value 11
6 bool containsEleven = std::ranges::contains(numbers, 11);
7 // Define a subrange to check
8 std::vector<int> subrange = {4, 5, 6};
9 // Check if the range contains the subrange
10 bool containsSubrange = std::ranges::contains_subrange(numbers
  , subrange);
11 // Define another subrange to check
12 std::vector<int> nonExistentSubrange = {7, 8, 11};
13 // Check if the range contains the non-existent subrange
14 bool containsNonExistentSubrange = std::ranges::
  contains_subrange(numbers, nonExistentSubrange);
```

- find, find_if, find_if_not, ranges::find, ranges::find_if, ranges::find_if_not

```
1 #include <algorithm>
2 #include <iostream>
3 #include <ranges>
4 #include <vector>
5
6 int main() {
7     std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9,
8                               10};
9
10    // Use std::find to find the first occurrence of 5
11    auto it = std::find(numbers.begin(), numbers.end(), 5);
12
13    // find the first even number
14    it = std::find_if(numbers.begin(), numbers.end(), [](int n
15    ) { return n % 2 == 0; });
16    // find the first odd number
17    it = std::find_if_not(numbers.begin(), numbers.end(), [](
18    int n) { return n % 2 == 0; });
19    // find the first occurrence of 5
20    auto range_it = std::ranges::find(numbers, 5);
21    // find the first even number
22    range_it = std::ranges::find_if(numbers, [](int n) {
23    return n % 2 == 0; });
24    // find the first odd number
25    range_it = std::ranges::find_if_not(numbers, [](int n) {
26    return n % 2 == 0; });
27 }
```

- find_last, find_last_if, find_last_if_not, find_end, ranges::find_end

```
1 #include <algorithm>
2 #include <iostream>
3 #include <ranges>
4 #include <vector>
5
6 std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 5,
7                          6, 7};
8
9 // find the last occurrence of 5
10 auto lastFive = std::ranges::find_last(numbers, 5);
11 // find the last even number
12 auto lastEven = std::ranges::find_last_if(numbers, [](int n) {
13     return n % 2 == 0; });
14 // find the last odd number
15 auto lastOdd = std::ranges::find_last_if_not(numbers, [](int n
16     ) { return n % 2 == 0; });
17
18 // Define a subrange to find
19 std::vector<int> subrange = {5, 6, 7};
20
21 // find the last occurrence of the subrange
22 auto lastSubrange = std::find_end(numbers.begin(), numbers.end
23     (), subrange.begin(), subrange.end());
24
25 // find the last occurrence of the subrange
26 auto lastSubrangeRange = std::ranges::find_end(numbers,
27     subrange);
```

- find_end, ranges::find_end, find_first_of, ranges::find_first_of
- adjacent_find, ranges::adjacent_find

- count, count_if, ranges::count, ranges::count_if
- mismatch, ranges::mismatch, equal, ranges::equal
- search, search_n, ranges::search, ranges::search_n
- ranges::starts_with, ranges::ends_with

• Fold Operations

- ranges::fold_left, ranges::_fold_left_first, ranges::fold_left_with_iter, ranges::fold_left_first_with_iter
- ranges::fold_right, ranges::fold_right_last

• Copy Operations

- copy, copy_if, ranges::copy, ranges::copy_if
- copy_n, ranges::copy_n
- copy_backwards, ranges::copy_backwards
- move, ranges::move
- move_backward, ranges::move_backward

• Swap Operations

- swap
- swap_ranges
- ranges::swap_ranges
- iter_swap

• Transform Operations

- transform, ranges::transform

```

1  #include <algorithm>
2  #include <execution>
3  #include <iostream>
4  #include <vector>
5
6  int main() {
7      // Original vector
8      std::vector<int> l1 = {1, 2, 3, 4, 5};
9      std::vector<int> l2 = std::vector<int>(l1.size(), 0);
10     std::vector<int> l3 = std::vector<int>(l1.size(), 0);
11
12     // simple transform (1 input, 1 output)
13     std::transform(l1.begin(), l1.end(), l2.begin(), [](int a)
14         { return a * 10; });
15
16     // transform (2 inputs, 1 output)
17     int multiplier = 2;
18     std::transform(std::execution::par_unseq, l1.begin(), l1.
19         end(), l2.begin(), l3.begin(),
20         [multiplier](int a, int b) { return (
21             multiplier * a) + b; });
22
23     return 0;
24 }
```

- replace, replace_if, ranges::replace, ranges::replace_if
- replace_copy, ranges::replace_copy, replace_copy_if, ranges::replace_copy_if

• Generation Operations

- fill, fill_n, ranges::fill, ranges::fill_n
- generate, generate_n, ranges::generate, ranges::generate_n

- **Removing Operations**

- remove, remove_if, ranges::remove, ranges::remove_if
- remove_copy, remove_copy_if, ranges::remove_copy, ranges::remove_copy_if
- unique, unique_copy, ranges::unique, ranges::unique_copy

- **Order-Changing Operations**

- reverse, ranges::reverse, reverse_copy, ranges::reverse_copy
- rotate, rotate_copy, ranges::rotate, ranges::rotate_copy
- shift_left, shift_right, ranges::shift_left, ranges::shift_right
- shuffle, random_shuffle, ranges::shuffle

- **Sampling Operations**

- sample, ranges::sample

- **Random Number Generation**

- ranges::generate_random

- **Partitioning Operations**

- is_partitioned, ranges::is_partitioned
- partition, ranges::partition
- partition_copy, ranges::partition_copy
- stable_partition, ranges::stable_partition
- partition_point, ranges::partition_point

- **Sorting Operations**

- sort, ranges::sort
- stable_sort, ranges::stable_sort
- partial_sort, ranges::partial_sort
- partial_sort_copy, ranges::partial_sort_copy
- is_sorted, ranges::is_sorted
- is_sorted_until, ranges::is_sorted_until
- nth_element, ranges::nth_element

- **Binary Search Operations (on partitioned ranges)**

- lower_bound, ranges::lower_bound, upper_bound, ranges::upper_bound
- equal_range, ranges::equal_range
- binary_search, ranges::binary_search

- **Set Operation (on sorted ranges)**

- includes, ranges::includes
- set_union, ranges::set_union
- set_intersection, ranges::set_intersection
- set_difference, ranges::set_difference
- set_symmetric_difference, ranges::set_symmetric_difference

- **Merge Operations (on sorted ranges)**

- merge, ranges::merge
- inplace_merge, ranges::inplace_merge

• Heap Operations

- push_heap, ranges::push_heap, pop_heap, ranges::pop_heap

```

1  std::vector<int> heap = {3, 1, 4, 1, 5, 9, 2, 6};
2
3  // Convert the vector into a heap
4  std::make_heap(heap.begin(), heap.end());
5  // Result: heap = {9, 6, 4, 1, 5, 3, 2, 1}
6
7  // Using std::push_heap to add a new element and maintain heap
   property
8  heap.push_back(7);
9  std::push_heap(heap.begin(), heap.end());
10 // Result: heap = {9, 7, 4, 6, 5, 3, 2, 1}
11
12 // Using std::pop_heap to remove the largest element and
   maintain heap property
13 std::pop_heap(heap.begin(), heap.end());
14 heap.pop_back();
15 // Result: heap = {7, 6, 4, 1, 5, 3, 2, 1}
16
17 // Using std::ranges::push_heap to add a new element and
   maintain heap property
18 heap.push_back(8);
19 std::ranges::push_heap(heap);
20 // Result: heap = {8, 7, 4, 6, 5, 3, 2, 1}
21
22 // Using std::ranges::pop_heap to remove the largest element
   and maintain heap property
23 std::ranges::pop_heap(heap);
24 heap.pop_back();
25 // Result: heap = {7, 6, 4, 1, 5, 3, 2, 1}

```

- make_heap, ranges::make_heap, sort_heap, ranges::sort_heap

```

1  std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6};
2
3  // Using std::make_heap to create a max-heap from the numbers
   vector
4  std::make_heap(numbers.begin(), numbers.end());
5  // Result: numbers = {9, 6, 4, 1, 5, 1, 2, 3}
6
7  // Using std::sort_heap to sort the heap
8  std::sort_heap(numbers.begin(), numbers.end());
9  // Result: numbers = {1, 1, 2, 3, 4, 5, 6, 9}
10
11 // Reset the numbers vector for ranges example
12 numbers = {3, 1, 4, 1, 5, 9, 2, 6};
13
14 // Using std::ranges::make_heap to create a max-heap from the
   numbers vector
15 std::ranges::make_heap(numbers);
16 // Result: numbers = {9, 6, 4, 1, 5, 1, 2, 3}
17
18 // Using std::ranges::sort_heap to sort the heap
19 std::ranges::sort_heap(numbers);
20 // Result: numbers = {1, 1, 2, 3, 4, 5, 6, 9}

```

- is_heap, ranges::is_heap, is_heap_until, ranges::is_heap_until

```

1 std::vector<int> numbers = {9, 6, 4, 1, 5, 1, 2, 3};
2
3 // Using std::is_heap to check if the numbers vector is a heap
4 bool isHeap = std::is_heap(numbers.begin(), numbers.end());
5 // Result: isHeap = true
6
7 // Using std::is_heap_until to find the first position where
  the heap property is violated
8 auto heapEnd = std::is_heap_until(numbers.begin(), numbers.end
  ());
9 // Result: heapEnd points to numbers.end(), indicating the
  entire range is a heap
10
11 // Using std::ranges::is_heap to check if the numbers vector
  is a heap
12 bool isHeapRanges = std::ranges::is_heap(numbers);
13 // Result: isHeapRanges = true
14
15 // Using std::ranges::is_heap_until to find the first position
  where the heap property is violated
16 auto heapEndRanges = std::ranges::is_heap_until(numbers);
17 // Result: heapEndRanges points to numbers.end(), indicating
  the entire range is a heap
18
19 // Modify the vector to violate the heap property
20 numbers = {9, 6, 4, 10, 5, 1, 2, 3};
21
22 // Re-check using std::is_heap
23 isHeap = std::is_heap(numbers.begin(), numbers.end());
24 // Result: isHeap = false
25
26 // Re-check using std::is_heap_until
27 heapEnd = std::is_heap_until(numbers.begin(), numbers.end());
28 // Result: heapEnd points to numbers.begin() + 3, where the
  value 10 violates the heap property
29
30 // Re-check using std::ranges::is_heap
31 isHeapRanges = std::ranges::is_heap(numbers);
32 // Result: isHeapRanges = false
33
34 // Re-check using std::ranges::is_heap_until
35 heapEndRanges = std::ranges::is_heap_until(numbers);
36 // Result: heapEndRanges points to numbers.begin() + 3, where
  the value 10 violates the heap property

```

• Min/Max Operations

– max, min, ranges::max, ranges::min

```

1 int a = 10;
2 int b = 20;
3
4 // Using std::max to find the maximum of two values
5 int maxVal = std::max(a, b);
6 // Result: maxVal = 20
7
8 // Using std::min to find the minimum of two values
9 int minVal = std::min(a, b);
10 // Result: minVal = 10
11
12 std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
13
14 // Using std::ranges::max to find the maximum value in a range

```

```

15 int maxInRange = std::ranges::max(numbers);
16 // Result: maxInRange = 9
17
18 // Using std::ranges::min to find the minimum value in a range
19 int minInRange = std::ranges::min(numbers);
20 // Result: minInRange = 1

```

– max_element, min_element, ranges::max_element, ranges::min_element

```

1 std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
2
3 // Using std::max_element to find the maximum element in a
  range
4 auto maxElement = std::max_element(numbers.begin(), numbers.
  end());
5 // Result: *maxElement = 9
6
7 // Using std::min_element to find the minimum element in a
  range
8 auto minElement = std::min_element(numbers.begin(), numbers.
  end());
9 // Result: *minElement = 1
10
11 // Using std::ranges::max_element to find the maximum element
   in a range
12 auto maxElementRanges = std::ranges::max_element(numbers);
13 // Result: *maxElementRanges = 9
14
15 // Using std::ranges::min_element to find the minimum element
   in a range
16 auto minElementRanges = std::ranges::min_element(numbers);
17 // Result: *minElementRanges = 1

```

– minmax, ranges::minmax, minmax_element, ranges::minmax_element

```

1 int a = 10;
2 int b = 20;
3
4 // Using std::minmax to find the minimum and maximum of two
  values
5 auto minmaxPair = std::minmax(a, b);
6 // Result: minmaxPair.first = 10, minmaxPair.second = 20
7
8 std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
9
10 // Using std::minmax_element to find the minimum and maximum
   elements in a range
11 auto minmaxElements = std::minmax_element(numbers.begin(),
  numbers.end());
12 // Result: *minmaxElements.first = 1, *minmaxElements.second =
   9
13
14 // Using std::ranges::minmax to find the minimum and maximum
   in a range
15 auto minmaxRange = std::ranges::minmax(numbers);
16 // Result: minmaxRange.min = 1, minmaxRange.max = 9
17
18 // Using std::ranges::minmax_element to find the minimum and
   maximum elements in a range
19 auto minmaxRangeElements = std::ranges::minmax_element(numbers
  );
20 // Result: *minmaxRangeElements.min = 1, *minmaxRangeElements.
  max = 9
21 //

```



```

22 // TODO (ksolomon): make sure usage is correct. why does
    minmax return a pair, whereas ranges::minmax return a tuple
    ?
23 //

```

– clamp, ranges::clamp

```

1  int value = 15;
2  int lowerBound = 10;
3  int upperBound = 20;
4
5  // Using std::clamp to constrain the value within the range [
    lowerBound, upperBound]
6  int clampedValue = std::clamp(value, lowerBound, upperBound);
7  // Result: clampedValue = 15
8
9  // Using std::clamp to constrain a value below the lower bound
10 int belowLower = 5;
11 int clampedBelow = std::clamp(belowLower, lowerBound,
    upperBound);
12 // Result: clampedBelow = 10
13
14 // Using std::clamp to constrain a value above the upper bound
15 int aboveUpper = 25;
16 int clampedAbove = std::clamp(aboveUpper, lowerBound,
    upperBound);
17 // Result: clampedAbove = 20
18
19 // Using std::ranges::clamp to constrain the value within the
    range [lowerBound, upperBound]
20 int clampedValueRanges = std::ranges::clamp(value, lowerBound,
    upperBound);
21 // Result: clampedValueRanges = 15
22
23 // Using std::ranges::clamp to constrain a value below the
    lower bound
24 int clampedBelowRanges = std::ranges::clamp(belowLower,
    lowerBound, upperBound);
25 // Result: clampedBelowRanges = 10
26
27 // Using std::ranges::clamp to constrain a value above the
    upper bound
28 int clampedAboveRanges = std::ranges::clamp(aboveUpper,
    lowerBound, upperBound);
29 // Result: clampedAboveRanges = 20

```

• Lexicographical Operations

- lexicographical_compare, ranges::lexicographical_compare
- lexicographical_compare_three_way

• Permutation Operations

- next_permutation, ranges::next_permutation, previous_permutation, ranges::previous_permutation, is_permutation, ranges::is_permutation

```

1  std::vector<int> numbers = {1, 2, 3};
2  std::vector<int> otherNumbers = {3, 2, 1};
3
4  // Using std::next_permutation to get the next lexicographical
    permutation
5  std::next_permutation(numbers.begin(), numbers.end());
6  // Result: {1, 3, 2}

```

```

7
8 // Using std::ranges::next_permutation to get the next
  lexicographical permutation
9 std::ranges::next_permutation(numbers);
10 // Result: {2, 1, 3}
11
12 // Using std::previous_permutation to get the previous
  lexicographical permutation
13 std::previous_permutation(numbers.begin(), numbers.end());
14 // Result: {1, 3, 2}
15
16 // Using std::ranges::previous_permutation to get the previous
  lexicographical permutation
17 std::ranges::previous_permutation(numbers);
18 // Result: {1, 2, 3}
19
20 // Using std::is_permutation to check if two sequences are
  permutations of each other
21 bool isPermutation = std::is_permutation(numbers.begin(),
  numbers.end(), otherNumbers.begin());
22 // Result: true
23
24 // Using std::ranges::is_permutation to check if two sequences
  are permutations of each other
25 bool isPermutationRanges = std::ranges::is_permutation(numbers
  , otherNumbers);
26 // Result: true

```

• Numeric Operations

– `iota`, `ranges::iota`

```

1 // Using std::iota to fill a vector with sequential values
2 std::vector<int> numbers(10);
3 std::iota(numbers.begin(), numbers.end(), 1); // Fills with
  values starting from 1
4
5 // Using std::ranges::iota to fill another vector with
  sequential values
6 std::vector<int> moreNumbers(10);
7 std::ranges::iota(moreNumbers, 11); // Fills with values
  starting from 11

```

– `accumulate`, `reduce`, `transform_reduce`

```

1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2
3 // Using std::accumulate to sum the elements
4 int sum = std::accumulate(numbers.begin(), numbers.end(), 0);
5
6 // Using std::reduce to sum the elements (C++17)
7 int sumReduce = std::reduce(std::execution::seq, numbers.begin
  (), numbers.end(), 0);
8
9 // Using std::transform_reduce to compute the sum of squares
10 int sumOfSquares = std::transform_reduce(numbers.begin(),
  numbers.end(), 0, std::plus<>(), [](int n) { return n * n;
  });

```

– `inner_product`

```

1 std::vector<int> vector1 = {1, 2, 3};
2 std::vector<int> vector2 = {4, 5, 6};
3

```

```

4 // Using std::inner_product to compute the inner product of
  vector1 and vector2
5 int result = std::inner_product(vector1.begin(), vector1.end(),
  , vector2.begin(), 0);
6 // result = 32

```

– adjacent_difference

```

1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> partialSums(numbers.size());
3
4 // Using std::partial_sum to compute the partial sums of the
  numbers vector
5 std::partial_sum(numbers.begin(), numbers.end(), partialSums.
  begin());
6 // partialSums: [1, 3, 6, 10, 15]

```

– partial_sum

```

1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> partialSums(numbers.size());
3
4 // Using std::partial_sum to compute the partial sums of the
  numbers vector
5 std::partial_sum(numbers.begin(), numbers.end(), partialSums.
  begin());
6 // partialSums = [1,3,6,10,15]

```

– exclusive_scan, inclusive_scan, transform_exclusive_scan, transform_inclusive_scan

```

1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> exclusiveScanResult(numbers.size());
3 std::vector<int> inclusiveScanResult(numbers.size());
4 std::vector<int> transformExclusiveScanResult(numbers.size());
5 std::vector<int> transformInclusiveScanResult(numbers.size());
6
7 // Using std::exclusive_scan to compute exclusive prefix sums
8 std::exclusive_scan(numbers.begin(), numbers.end(),
  exclusiveScanResult.begin(), 0);
9 // Result: {0, 1, 3, 6, 10}
10
11 // Using std::inclusive_scan to compute inclusive prefix sums
12 std::inclusive_scan(numbers.begin(), numbers.end(),
  inclusiveScanResult.begin());
13 // Result: {1, 3, 6, 10, 15}
14
15 // Using std::transform_exclusive_scan to compute exclusive
  prefix sums of squares
16 std::transform_exclusive_scan(numbers.begin(), numbers.end(),
  transformExclusiveScanResult.begin(), 0, std::plus<>(),
  [](int n) { return n * n; });
17 // Result: {0, 1, 5, 14, 30}
18
19
20 // Using std::transform_inclusive_scan to compute inclusive
  prefix sums of squares
21 std::transform_inclusive_scan(numbers.begin(), numbers.end(),
  transformInclusiveScanResult.begin(), std::plus<>(),
  [](int n) { return n * n; });
22 // Result: {1, 5, 14, 30, 55}
23

```

• Uninitialized Memory Operations

4.2 Classes

- **Class Definition**

- Syntax and structure
- Access specifiers: `public`, `private`, `protected`

- **Inheritance**

- Single and multiple inheritance
- Virtual inheritance

- **Polymorphism**

- Function overloading
- Operator overloading
- Virtual functions and abstract classes

4.3 Containers

- Sequence

- **array**

```
1 std::array<int, 3> arr; // uninitialized (whatever was in
   memory before)
2 std::array<int, 3> arr = {}; // initialized as 0s
3 std::array<int, 3> arr1 = {1, 2, 3};
4 std::array<int, 3> arr2{1, 2, 4};
5 arr1.fill(0); // fills array with 0s
6 arr1.swap(arr2); // swaps contents of arr1 and arr2
```

- **vector**

```
1 std::vector<int> v;
2 v.capacity(); // size of currently allocated memory
3 v.shrink_to_fit(); // releases unused memory
4 v.reserve(100); // pre-allocates 100 elements
5 v.clear(); // erases all elements
6 v.erase(v.begin()); // erases first element
7 v.push_back(1); // adds 1 to the end
8 v.rbegin(); // reverses iterator
9 std::erase_if(v, [](int x) { return x > 10; }); // removes all
   elements > 10
10 std::vector<Pair<int, int>> classV;
11 classV.emplace_back(10, 1); // create Pair object and push to
   back
```

- **inplace_vector**

- **deque**

- Associative

- **Set**
 - **Map**
 - **Multiset**
 - **Multimap**

- Unordered Associative

- **unordered_set**
 - **unordered_map**

- `unordered_multiset`
- `unordered_multimap`

- **Adaptors**

- `stack`
- `queue`
- `priority_queue`
- `flat_set`
- `flat_map`
- `flat_multiset`
- `flat_multimap`

4.4 Modern C++

- C++11

- **Alias Templates**

```

1 #include <iostream>
2 #include <map>
3 #include <string>
4 #include <vector>
5
6 // Alias template for a vector of a specific type
7 template <typename T>
8 using Vector = std::vector<T>;
9
10 // Alias template for a map with string keys and a specific
    value type
11 template <typename V>
12 using StringMap = std::map<std::string, V>;
13
14 int main() {
15     // Using the alias template for a vector of integers
16     Vector<int> intVector = {1, 2, 3, 4, 5};
17     std::cout << "Vector of integers: ";
18     for (const auto& elem : intVector) {
19         std::cout << elem << " ";
20     }
21     std::cout << std::endl;
22
23     // Using the alias template for a map with string keys and
        integer values
24     StringMap<int> ageMap = {{ "Alice", 30}, {"Bob", 25}, {"
        Charlie", 35}};
25     std::cout << "Map of ages: ";
26     for (const auto& pair : ageMap) {
27         std::cout << pair.first << ": " << pair.second << " ";
28     }
29     std::cout << std::endl;
30
31     return 0;
32 }

```

- **atomic**

Well-defined behavior in the event of RMW race condition. Accesses to atomics may establish inter-thread synchronization and order non-atomic accesses.

```

1 atomic_bool b; // same as std::atomic<bool> b;

```

- **auto**
- **constexpr**
- **final**

- * Specifies that a class cannot be inherited from.
- * When used in a virtual function, specifies that the function cannot be overridden by a derived class.
- * **final** is also a legal variable/function name. Only has special meaning in member function declaration or class head.

```

1 struct Base
2 {
3     virtual void foo();
4 };
5 struct A : Base
6 {
7     void foo() final; // Base::foo is overridden and A::foo is
                        // the final override
8     void bar() final; // Error: bar cannot be final as it is
                        // non-virtual
9 };
10
11 struct B final : A // struct B is final
12 {
13     void foo() override; // Error: foo cannot be overridden as
                            // it is final in A
14 };
15
16 struct C : B {}; // Error: B is final

```

- **initializer list**

```

1  /*
2   * In this program:
3   *   Vector Initialization: A 'std::vector' is initialized
4   *   using an initializer list, which provides a concise way to
5   *   initialize containers with a list of values. Class
6   *   Constructor: The 'MyClass' constructor takes an
7   *   'std::initializer_list<int>' as a parameter, allowing
8   *   objects of 'MyClass' to be initialized with a list of
9   *   integers.
10  *   Function Parameter: The 'printList' function takes an 'std
11  *   ::initializer_list<std::string>' as a parameter,
12  *   demonstrating how initializer lists can be used to pass a
13  *   variable number of arguments to a function. This program
14  *   demonstrates the flexibility and convenience of using
15  *   initializer lists in various contexts in C++.
16  */
17
18 #include <initializer_list>
19 #include <iostream>
20 #include <vector>
21
22 class MyClass {
23 public:
24     MyClass(std::initializer_list<int> list) {
25         for (auto elem : list) {
26             data_.push_back(elem);
27         }
28     }
29
30     void print() const {

```

```

24         for (auto elem : data_) {
25             std::cout << elem << " ";
26         }
27         std::cout << std::endl;
28     }
29
30 private:
31     std::vector<int> data_;
32 };
33
34 void printList(std::initializer_list<std::string> list) {
35     for (const auto& elem : list) {
36         std::cout << elem << " ";
37     }
38     std::cout << std::endl;
39 }
40
41 int main() {
42     // Initializing a vector using an initializer list
43     std::vector<int> vec = {1, 2, 3, 4, 5};
44     std::cout << "Vector elements: ";
45     for (int v : vec) {
46         std::cout << v << " ";
47     }
48     std::cout << std::endl;
49
50     // Using initializer list in a class constructor
51     MyClass myObject = {10, 20, 30, 40, 50};
52     std::cout << "MyClass elements: ";
53     myObject.print();
54
55     // Passing an initializer list to a function
56     std::cout << "String list: ";
57     printList({"Hello", "World", "from", "initializer", "list"});
58
59     return 0;
60 }

```

– iota

```

1 void iota(ForwardIterator begin, ForwardIterator end, T v); //
    fills range [first-last] with sequentially increasing
    values starting at v in begin

```

– lambdas

```

1 #include <algorithm>
2 #include <functional>
3 #include <iostream>
4 #include <vector>
5
6 int main() {
7     // Basic lambda with no capture
8     auto greet = []() { std::cout << "Hello, World!" << std::
9         endl; };
10    greet();
11
12    // Lambda with capture by value
13    int a = 10;
14    auto captureByValue = [a]() { std::cout << "Captured by
15        value: " << a << std::endl; };
16    captureByValue();

```

```

16 // Lambda with capture by reference
17 int b = 20;
18 auto captureByReference = [&b]() {
19     b += 10;
20     std::cout << "Captured by reference: " << b << std::
        endl;
21 };
22 captureByReference();
23 std::cout << "Modified b: " << b << std::endl;
24
25 // Lambda with explicit return type
26 auto add = [](int x, int y) -> int { return x + y; };
27 std::cout << "Sum: " << add(3, 4) << std::endl;
28
29 // Generic lambda
30 auto multiply = [](auto x, auto y) { return x * y; };
31 std::cout << "Product: " << multiply(3, 4.5) << std::endl;
32
33 // Lambda with STL algorithms
34 std::vector<int> numbers = {1, 2, 3, 4, 5};
35 std::for_each(numbers.begin(), numbers.end(), [](int n) {
36     std::cout << n << " "; });
37 std::cout << std::endl;
38
39 // C++23: Deducing 'this' in lambdas
40 struct Counter {
41     int count = 0;
42     auto increment() {
43         return [this]() {
44             ++count;
45             std::cout << "Count: " << count << std::endl;
46         };
47     };
48
49     Counter counter;
50     auto inc = counter.increment();
51     inc();
52     inc();
53
54     return 0;
55 }

```

capture comma-separated list of variables which are captured/modified by the lambda. Captures cannot have same name as input parameters.

Capture list

- * & = capture all used variables by reference
- * = = capture all used variables by copy
- * varName = by-copy
- * varName... = by-copy pack-expansion
- * varName initializer = by-copy w/ initializer
- * &varName = by-reference
- * &varName... = by-reference pack-expansion
- * &varName initializer = by-reference w/ initializer
- * this = by-reference capture of current object
- * *this = by-copy capture of current object
- * ... = by-copy capture of all objects w/ pack expansion
- * &... initializer = by-reference w/ initializer and pack expansion


```

1 // If the capture-default is &, subsequent simple captures
  must not begin with &.
2 [&] {}; // OK: by-reference capture default
3 [&, i] {}; // OK: by-reference capture, except i is
  captured by copy
4 [&, &i] {}; // Error: by-reference capture when by-
  reference is the default
5 [&, this] {}; // OK, equivalent to [&]
6 [&, this, i] {}; // OK, equivalent to [&, i]

```

```

1 // If the capture-default is =, subsequent simple captures
  must begin with & or be *this(since C++17) or this(since C
  ++20) .
2 [=] {}; // OK: by-copy capture default
3 [=, &i] {}; // OK: by-copy capture, except i is captured by
  reference
4 [=, *this] {}; // until C++17: Error: invalid syntax
5 // since C++17: OK: captures the enclosing S2
  by copy
6 [=, this] {}; // until C++20: Error: this when = is the
  default
7 // since C++20: OK, same as [=]

```

- **mutex**
- **override**
- **random**

```

1 #include <stdlib>
2 int rand(); // returns integer in [0, RAND_MAX]

```

```

1 #include <random>
2 // default_random_engine
3 // philox4x64 -> philox_engine
4 // random_device = non-deterministic generator based on
  hardware entropy
5 std::random_device rd;
6 rd.entropy(); // estimate of random number device entropy.
  Deterministic entropy = 0.
7 std::uniform_real_distribution<double> dist(0.0, 1.0);

```

Distribution list

- * uniform
 - int
 - real (double)
- * bernoulli
 - bernoulli
 - binomial
 - negative binomial
 - geometric
- * Poisson
 - poisson
 - exponential
 - gamma
 - weibull
 - extreme_value
- * Normal

- normal
- lognormal
- chi_squared
- cauchy
- fisher_f
- student_t
- * Sampling
 - discrete
 - piecewise_constant
 - piecewise_linear
 - item4
- **range-based for**
- **thread**
- **trailing return type** auto main() --> int {return 0;}

- C++14

- **Variable Templates**
- **Generic Lambdas**

- C++17

- **tuple**

```

1 #include <iostream>
2 #include <string>
3 #include <tuple>
4
5 int main() {
6     // Create a tuple with different types
7     std::tuple<int, std::string, double> person = std::
        make_tuple(25, "Alice", 68.5);
8
9     // Access elements of the tuple using std::get
10    int age = std::get<0>(person);
11    std::string name = std::get<1>(person);
12    double weight = std::get<2>(person);
13
14    // Modify elements of the tuple
15    std::get<0>(person) = 30;
16    std::get<2>(person) = 70.0;
17    // Use std::tie to unpack tuple into variables
18    int newAge;
19    std::string newName;
20    double newWeight;
21    std::tie(newAge, newName, newWeight) = person;
22
23    std::cout << "Unpacked Name: " << newName << ", Unpacked
        Age: " << newAge << ", Unpacked Weight: " << newWeight
24        << std::endl;
25
26    // Use std::ignore to unpack only specific elements
27    std::tie(std::ignore, newName, std::ignore) = person;
28    std::cout << "Unpacked Name with ignore: " << newName <<
        std::endl;
29
30    return 0;
31 }

```

- **execution policies**

seq used to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution must be sequential. This is used by default when no execution policy is specified.

par Indicates that a parallel algorithm MAY be parallelized. Synchronization techniques (e.g. mutexes) may be used.

par_unseq A parallel algorithm MAY be parallelized, vectorized, and moved between threads. Vectorization MUST not use any vectorization-unsafe operations (e.g. mutexes and `std::atomic`)

unseq An algorithm's execution MAY be vectorized. Synchronization techniques MUST NOT be used. Since C++20 (the rest of the policies were introduced in C++17).

- C++20

- **Modules**

- **Coroutines**

- **Ranges**

Extension/Generalization of algorithms and iterator libraries to make them less error-prone. Ranges are an abstraction of the following:

```
* [begin, end) iterator pair : ranges::sort()
* begin + [0, size) : views::counted()
* [begin, predicate) : views::take_while() (conditionally-terminated sequences)
* [begin, ..) : unbounded (e.g. views::iota())
```

`std::views` // shorthand for `std::ranges::views` TODO: do more usage/investigation on these

- **Midpoint**

Can be used on any arithmetic type, excluding `bool`. Can be used on objects as long as they are not incomplete types. Returns half the sum of the two inputs, no overflow occurs (this is the main reason to use STL rather than custom implementation). Inputs must point to elements in same object, else behavior is undefined. In case of decimal in average, rounds down.

- **using enum**

- **constexpr**

- **string formatting**

- **template concepts**

- **coroutines**

- **modules**

- C++23

- **print/println**

```
1 #include <print>
2 std::print("{0} {2}{1}!", "Hello", 23, "C++");
3 std::println(); // adds newline to std::print();
```

- **byteswap**

```
1 #include <bit>
2 std::byteswap(T n) noexcept; // T can be any integer value
```

- **flat_map/flat_set**

4.5 Concepts

- **Types**

- **RAII**

RAII Resource Acquisition Is Initialization

- **item3**

- **item4**