

# C++ Review

Karl Solomon

December 30, 2024

## Contents

<b>I</b>	<b>C++</b>	<b>1</b>
<b>1</b>	<b>Classes</b>	<b>1</b>
<b>2</b>	<b>Containers</b>	<b>1</b>
<b>3</b>	<b>Modern C++</b>	<b>3</b>
<b>4</b>	<b>Concepts</b>	<b>10</b>

## Part I

# C++

## 1 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

## 2 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** Abstract data type that provides efficient access to the highest priority element. This is basically c++'s implementation of a Max Heap.

```

1 #include <iostream>
2 #include <queue>
3 #include <vector>
4
5 int main() {
6     // Create a priority queue (max-heap by default)
7     std::priority_queue<int> pq;
8
9     // Insert elements into the priority queue
10    pq.push(10);
11    pq.push(5);
12    pq.push(20);
13    pq.push(15);
14
15    // The priority queue will arrange elements in descending
    order
16    // Result: pq = {20, 15, 10, 5}

```

```

17
18     // Access and remove elements from the priority queue
19     while (!pq.empty()) {
20         int top = pq.top(); // Access the top element
21         pq.pop();          // Remove the top element
22         // Result: top = 20, then 15, then 10, then 5
23     }
24
25     return 0;
26 }

```

- flat\_set
- flat\_map
- flat\_multiset
- flat\_multimap

### 3 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 }
```

```

21     }
22
23     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;

```

```

13     auto captureByValue = [a]() { std::cout << "Captured by
14         value: " << a << std::endl; };
15     captureByValue();
16
17     // Lambda with capture by reference
18     int b = 20;
19     auto captureByReference = [&b]() {
20         b += 10;
21         std::cout << "Captured by reference: " << b << std::
22             endl;
23     };
24     captureByReference();
25     std::cout << "Modified b: " << b << std::endl;
26
27     // Lambda with explicit return type
28     auto add = [](int x, int y) -> int { return x + y; };
29     std::cout << "Sum: " << add(3, 4) << std::endl;
30
31     // Generic lambda
32     auto multiply = [](auto x, auto y) { return x * y; };
33     std::cout << "Product: " << multiply(3, 4.5) << std::endl;
34
35     // Lambda with STL algorithms
36     std::vector<int> numbers = {1, 2, 3, 4, 5};
37     std::for_each(numbers.begin(), numbers.end(), [](int n) {
38         std::cout << n << " "; });
39     std::cout << std::endl;
40
41     // C++23: Deducing 'this' in lambdas
42     struct Counter {
43         int count = 0;
44         auto increment() {
45             return [this]() {
46                 ++count;
47                 std::cout << "Count: " << count << std::endl;
48             };
49         }
50     };
51
52     Counter counter;
53     auto inc = counter.increment();
54     inc();
55     inc();
56
57     return 0;
58 }

```

**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.h>
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` See <algorithm> document for copious examples using ranges.

- **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 Concepts

- **Types**
- **RAII**

**RAII** Resource Acquisition Is Initialization