C++ Review

Karl Solomon

January 2, 2025

Contents

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**

```
std::array<int, 3> arr; // uninitialized (whatever was in memory before)
std::array<int, 3> arr = {}; // initialized as 0s
std::array<int, 3> arr1 = {1, 2, 3};
std::array<int, 3> arr2{1, 2, 4};
```

```
arr1.fill(0); // fills array with 0s
arr1.swap(arr2); // swaps contents of arr1 and arr2
```

  - **vector**

```
std::vector<int> v;
v.capacity(); // size of currently allocated memory
v.shrink_to_fit(); // releases unused memory
v.reserve(100); // pre-allocates 100 elements
v.clear(); // erases all elements
v.erase(v.begin()); // erases first element
v.push_back(1); // adds 1 to the end
v.rbegin(); // reverses iterator
std::erase_if(v, [](int x) { return x > 10; }); // removes all elements > 10
std::vector<Pair<int,int>> classV;
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.

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

int main() {
    // Create a priority queue (max-heap by default)
    std::priority_queue<int> pq;

    // Insert elements into the priority queue
    pq.push(10);
    pq.push(5);
    pq.push(20);
    pq.push(15);

    // The priority queue will arrange elements in descending order
    // Result: pq = {20, 15, 10, 5}

    // Access and remove elements from the priority queue
    while (!pq.empty()) {
        int top = pq.top();   // Access the top element
        pq.pop();             // Remove the top element
        // Result: top = 20, then 15, then 10, then 5
    }

    return 0;
}
```

  - **flat_set**
  - **flat_map**
  - **flat_multiset**
  - **flat_multimap**

3 Modern C++

- C++11
  - **Alias Templates**

```cpp
#include <iostream>
#include <map>
#include <string>
#include <vector>

// Alias template for a vector of a specific type
template <typename T>
using Vector = std::vector<T>;

// Alias template for a map with string keys and a specific value type
template <typename V>
using StringMap = std::map<std::string, V>;

int main() {
    // Using the alias template for a vector of integers
    Vector<int> intVector = {1, 2, 3, 4, 5};
    std::cout << "Vector of integers: ";
    for (const auto& elem : intVector) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // Using the alias template for a map with string keys and integer values
    StringMap<int> ageMap = {{"Alice", 30}, {"Bob", 25}, {"Charlie", 35}};
    std::cout << "Map of ages: ";
    for (const auto& pair : ageMap) {
        std::cout << pair.first << ": " << pair.second << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

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

```cpp
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.

```cpp
struct Base
{
    virtual void foo();
};
struct A : Base
{
    void foo() final; // Base::foo is overridden and A::foo is the final override
    void bar() final; // Error: bar cannot be final as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B {}; // Error: B is final
```

- **initializer list**

```cpp
/*
 * In this program:
 *  Vector Initialization: A ‘std::vector‘ is initialized using an initializer list, which provides a concise way to
 * initialize containers with a list of values. Class Constructor: The ‘MyClass‘ constructor takes an
 * ‘std::initializer_list<int>‘ as a parameter, allowing objects of ‘MyClass‘ to be initialized with a list of integers.
 *  Function Parameter: The ‘printList‘ function takes an ‘std::initializer_list<std::string>‘ as a parameter,
 * demonstrating how initializer lists can be used to pass a variable number of arguments to a function. This program
 * demonstrates the flexibility and convenience of using initializer lists in various contexts in C++.
 */

#include <initializer_list>
#include <iostream>
#include <vector>

class MyClass {
 public:
    MyClass(std::initializer_list<int> list) {
        for (auto elem : list) {
            data_.push_back(elem);
        }
    }

    void print() const {
        for (auto elem : data_) {
            std::cout << elem << " ";
        }
        std::cout << std::endl;
    }

 private:
    std::vector<int> data_;
};

void printList(std::initializer_list<std::string> list) {
    for (const auto& elem : list) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;
}

int main() {
    // Initializing a vector using an initializer list
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::cout << "Vector elements: ";
    for (int v : vec) {
        std::cout << v << " ";
    }
    std::cout << std::endl;

    // Using initializer list in a class constructor
    MyClass myObject = {10, 20, 30, 40, 50};
    std::cout << "MyClass elements: ";
    myObject.print();

    // Passing an initializer list to a function
    std::cout << "String list: ";
    printList({"Hello", "World", "from", "initializer", "list"});

    return 0;
}
```

- **iota**

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

- **lambdas**

```cpp
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

int main() {
    // Basic lambda with no capture
    auto greet = []() { std::cout << "Hello, World!" << std::endl; };
    greet();

    // Lambda with capture by value
    int a = 10;
    auto captureByValue = [a]() { std::cout << "Captured by value: " << a << std::endl; };
    captureByValue();

    // Lambda with capture by reference
    int b = 20;
    auto captureByReference = [&b]() {
        b += 10;
        std::cout << "Captured by reference: " << b << std::endl;
    };
    captureByReference();
    std::cout << "Modified b: " << b << std::endl;

    // Lambda with explicit return type
    auto add = [](int x, int y) -> int { return x + y; };
    std::cout << "Sum: " << add(3, 4) << std::endl;

    // Generic lambda
    auto multiply = [](auto x, auto y) { return x * y; };
    std::cout << "Product: " << multiply(3, 4.5) << std::endl;

    // Lambda with STL algorithms
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::for_each(numbers.begin(), numbers.end(), [](int n) { std::cout << n << " "; });
    std::cout << std::endl;

    // C++23: Deducing 'this' in lambdas
    struct Counter {
        int count = 0;
        auto increment() {
            return [this]() {
                ++count;
                std::cout << "Count: " << count << std::endl;
            };
        }
    };

    Counter counter;
    auto inc = counter.increment();
    inc();
    inc();

    return 0;
}
```

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

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

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

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

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

```
#include <random>
// default_random_engine
// philox4x64 -> philox_engine
// random_device = non-deterministic generator based on hardware entropy
std::random_device rd;
rd.entropy(); // estimate of random number device entropy. Deterministic entropy = 0.
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**

```cpp
#include <iostream>
#include <string>
#include <tuple>

int main() {
    // Create a tuple with different types
    std::tuple<int, std::string, double> person = std::make_tuple(25, "Alice", 68.5);

    // Access elements of the tuple using std::get
    int age = std::get<0>(person);
    std::string name = std::get<1>(person);
    double weight = std::get<2>(person);

    // Modify elements of the tuple
    std::get<0>(person) = 30;
    std::get<2>(person) = 70.0;
    // Use std::tie to unpack tuple into variables
    int newAge;
    std::string newName;
    double newWeight;
    std::tie(newAge, newName, newWeight) = person;

    std::cout << "Unpacked Name: " << newName << ", Unpacked Age: " << newAge << ", Unpacked Weight: " << newWeight
           << std::endl;

    // Use std::ignore to unpack only specific elements
    std::tie(std::ignore, newName, std::ignore) = person;
    std::cout << "Unpacked Name with ignore: " << newName << std::endl;

    return 0;
}
```

  - **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 algoithm 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**
  - **constinit**
  - **string formatting**
  - **template concepts**
  - **coroutines**
  - **modules**

- C++23
  - **print/println**

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

    ```
    #include <bit>
    std::byteswap(T n) noexcept; // T can be any integer value
    ```
  - **flat_map/flat_set**

4 Concepts

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

  RAII `Resource Acquisition Is Initialization`