Containers library

The Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queues, lists and stacks. There are two(until C++11)three(since C++11) classes of containers:

- sequence containers,
- associative containers, and
- unordered associative containers, (since C++11)

each of which is designed to support a different set of operations.

Sequence containers

Sequence containers implement data structures which can be accessed sequentially. -

- array (static contiguous memory),
- vector (dynamic contiguous memory),
- forward_list (singly linked list),
- list (doubly linked list),
- deque (double ended queue)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(log\ n)$ complexity).

- set - collection of unique keys, **sorted** by keys
- multiset -same as set but multiple keys of same value are allowed
- map - collection of key-value pairs, sorted by keys, keys are unique
- multimap - same as map but multiple keys of same value are allowed

Unordered associative containers **(since C++11)**

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ average, $O(n)$ worst-case complexity).

- unordered_set - collection of unique keys, **hashed** by keys
- unordered_multiset -same as set but multiple keys of same value are allowed
- unordered_map - collection of key-value pairs, **hashed** by keys, keys are unique
- unordered_multimap - same as map but multiple keys of same value are allowed

Container adaptors

Container adaptors provide a different interface for sequential containers.

stack: adapts a container to provide stack (LIFO data structure)
queue: adapts a container to provide queue (FIFO data structure)
priority_queue: adapts a container to provide priority queue

flat_set: (C++23): adapts a container to provide a collection of unique keys, sorted by keys
flat_map:(C++23):adapts two containers to provide a collection of key-value pairs, sorted by unique keys
flat_multiset: (C++23): adapts a container to provide a collection of keys, sorted by keys
flat_multimap: (C++23): adapts two containers to provide a collection of key-value pairs, sorted by keys

C++ Containers and operations

https://www.sandordargo.com/blog/2023/11/15/most-important-complexities

First, let's see what are the most important containers you'll likely deal with in a coding interview, what are the underlying data structures and what are the related complexities. My goal is not to give you a deep analysis, just to provide you with the most necessary information, then you can do your own research.

`std::array`

`std::array` is a fixed-size array, storing objects in contiguous memory locations.

- accessing the first element: with `front()` which has a complexity of `O(1)`
- accessing the last element: with `back()` which has a complexity of `O(1)`
- accessing a random element: with `at()` or with `operator[]` both have a complexity of `O(1)`

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>

int main()
{
    // Construction uses aggregate initialization
    std::array<int, 3> a1{{5, 4, 3}};
          // Double-braces required prior to C++11
    std::array<int, 3> a2 = {1, 2, 3}; // Double braces never
required
```

```
    // Container operations are supported
    std::sort(a1.begin(), a1.end());
    for (const auto& k : a1)
        std::cout << k << ' ';
    std::cout << '\n';

    // Ranged for loop is supported
    std::array<std::string, 2> a3{"E", "\u018E"};
    for (const auto& s : a3)
        std::cout << s << ' ';
    std::cout << '\n';

    // Deduction guide for array creation (since C++17)
    [[maybe_unused]] std::array a4{3.0, 1.0, 4.0}; //
std::array<double, 3>

    // Behavior of unspecified elements is the same as with built-in
arrays
    [[maybe_unused]] std::array<int, 2> a5; // No list init, a5[0]
and a5[1]
                                            // are default
initialized
    [[maybe_unused]] std::array<int, 2> a6{}; // List init, both
elements are value
                                              // initialized, a6[0] =
a6[1] = 0
    [[maybe_unused]] std::array<int, 2> a7{1}; // List init,
unspecified element is value
                                               // initialized, a7[0]
= 1, a7[1] = 0
}
```

`std::list`

`std::list` is a container that supports fast insertion and removal, but doesn't support fast random access. It is usually implemented as a doubly-linked list. `std::forward_list` is similar, but implemented with a singly-linked list, so it's more space efficient, but it supports iteration only in one direction

- accessing the first element: with `front()` which has a complexity of `O(1)`

- accessing the last element: with `back()` which has a complexity of `O(1)` (not supported by `std::forward_list`)
- accessing a random element: not supported
- inserting at the front: with `push_front()` which has a complexity of `O(1)`
- inserting at the back: with `push_back()` which has a complexity of `O(1)` (not supported by `std::forward_list`)
- inserting at a random location: with `insert()` which has a complexity of `O(1)` for one element, and complexity of `O(n)` for multiple elements, where `n` is the number of elements to be inserted (`insert_after` for `std::forward_list`)
- removing an item from the front: with `pop_front()` which has a complexity of `O(1)`
- removing an item from the back: with `pop_back()` which has a complexity of `O(1)` (not supported by `std::forward_list`)
- removing an item from a random location: with `erase()` which has a complexity of `O(1)` for one element, and a complexity of `O(n)` for multiple elements, where `n` is the number of elements to be erased (`erase_after` for `std::forward_list`)

```
#include <algorithm>
#include <iostream>
#include <list>

int main()
{
    // Create a list containing integers
    std::list<int> l = {7, 5, 16, 8};

    // Add an integer to the front of the list
    l.push_front(25);
    // Add an integer to the back of the list
    l.push_back(13);

    // Insert an integer before 16 by searching
    auto it = std::find(l.begin(), l.end(), 16);
```

```
    if (it != l.end())
        l.insert(it, 42);

    // Print out the list
    std::cout << "l = { ";
    for (int n : l)
        std::cout << n << ", ";
    std::cout << "};\n";
}
```

std::forward_list

`std::forward_list` is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list. Compared to `std::list` this container provides more space efficient storage when bidirectional iteration is not needed.

Important functions: front, pop_front, push_front, swap, empty, clear, insert_after, emplace_front, emplace_after

std::vector

`std::vector` is a dynamically sized sequence container, where the elements are stored contiguously. Random access is cheap, as well as operations at the end, unless reallocation is required.

- accessing the first element: with `front()` which has a complexity of `O(1)`
- accessing the last element: with `back()` which has a complexity of `O(1)`
- accessing a random element: with `at()` or with `operator[]` both have a complexity of `O(1)`
- inserting at the front: with `insert()` which has a complexity of `O(n+m)` where `n` is the number of elements to insert and `m` is the size of the container
- inserting at the back: with `push_back()` which has a complexity of amortized `O(1)`
- inserting at a random location: with `insert()` which has a complexity of `O(n+m)` where `n` is the number of elements to insert and `m` is the distance between the elements to insert and the end of the container

- removing an item from the front: with `erase()` which has a complexity of `O(n+m)` where `n` is the number of elements erased (calls to the destructor) and `m` is the number of assignments to make - the size of the elements left in the vector
- removing an item from the back: with `pop_back()` which has a complexity of `O(1)`
- removing an item from a random location: with `erase()` which has a complexity of `O(n+m)` where `n` is the number of elements erased (calls to the destructor) and `m` is the number of assignments to make - at least the number of elements after the last erased item, worst the size of the whole container left

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

int main()
{
    // Create a vector containing integers
    std::vector<int> v = {8, 4, 5, 9};

    // Add two more integers to vector
    v.push_back(6);
    v.push_back(9);

    // Overwrite element at position 2
    v[2] = -1;

    // Print out the vector
    for (int n : v)
        std::cout << n << ' ';
    std::cout << '\n';
}
```

std::deque

`std::deque` (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.

As opposed to `std::vector`, the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays, with additional

bookkeeping, which means indexed access to deque must perform two pointer dereferences, compared to vector's indexed access which performs only one.

The storage of a deque is automatically expanded and contracted as needed. Expansion of a deque is cheaper than the expansion of a `std::vector` because it does not involve copying of the existing elements to a new memory location. On the other hand, deques typically have large minimal memory cost; a deque holding just one element has to allocate its full internal array (e.g. 8 times the object size on 64-bit libstdc++; 16 times the object size or 4096 bytes, whichever is larger, on 64-bit libc++).

The complexity (efficiency) of common operations on deques is as follows:

- Random access - constant $O(1)$.
- Insertion or removal of elements at the end or beginning - constant $O(1)$.
- Insertion or removal of elements - linear $O(n)$.

Important methods are front(), back(), pop_front() and pop_back().

```
#include <deque>
#include <iostream>

int main()
{
    // Create a deque containing integers
    std::deque<int> d = {7, 5, 16, 8};

    // Add an integer to the beginning and end of the deque
    d.push_front(13);
    d.push_back(25);

    // Iterate and print values of deque
    for (int n : d)
        std::cout << n << ' ';
    std::cout << '\n';
}
```

Associative containers - set, map, multiset, multimap

std::set

`std::set` is an associative container that contains a sorted set of unique objects of type `Key`. Sorting is done using the key comparison function *Compare*. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as Red–black trees

Noted functions: empty, size, max_size, count, erase, find, insert, emplace, clear

```cpp
#include <set>
#include <iostream>

int main()
{
    std::set<int> c = {1, 2, 3, 4, 1, 2, 3, 4};

    auto print = [&c]
    {
        std::cout << "c = { ";
        for (int n : c)
            std::cout << n << ' ';
        std::cout << "}\n";
    };
    print();

    std::cout << "Erase all odd numbers:\n";
    for (auto it = c.begin(); it != c.end();)
    {
        if (*it % 2 != 0)
            it = c.erase(it);
        else
            ++it;
    }
    print();

    std::cout << "Erase 1, erased count: " << c.erase(1) << '\n';
    std::cout << "Erase 2, erased count: " << c.erase(2) << '\n';
    std::cout << "Erase 2, erased count: " << c.erase(2) << '\n';
    print();
}
```

std::map

`std::map` is a sorted associative container providing search, removal and insertion at a logarithmic complexity. They are usually implemented as red-black trees.

- accessing an element: with `at()` or with `operator[]` both have a complexity of `O(log n)` where `n` is the size of the container
- inserting an element at a random location: with `insert()` or with `operator[]` both have a complexity of `O(log n)` where `n` is the size of the container. With `insert()` you can insert multiple elements, and then the complexity becomes

`O(m * log n)`, where `m` is the number of elements to insert. `insert()` can also take a position as a hint where to insert. If the insertion happens there then the complexity is amortized `O(1)` otherwise `O(log n)`

- removing an item: with `erase()` which has a complexity of amortized `O(1)` if the erasure happens with an iterator, otherwise it's `O(log(n) + m)` where `n` is the size of the container and `m` is the number of elements to remove
- finding an element: with `find()` which has a complexity of `O(log n)` where `n` is the size of the container

```
#include <iostream>
#include <map>
#include <string>
#include <string_view>

void print_map(std::string_view comment, const std::map<std::string, int>& m)
{
    std::cout << comment;
    // Iterate using C++17 facilities
    for (const auto& [key, value] : m)
        std::cout << '[' << key << "] = " << value << "; ";

// C++11 alternative:
//  for (const auto& n : m)
//      std::cout << n.first << " = " << n.second << "; ";
//
// C++98 alternative:
//  for (std::map<std::string, int>::const_iterator it = m.begin(); it != m.end(); ++it)
//      std::cout << it->first << " = " << it->second << "; ";

    std::cout << '\n';
}

int main()
{
    // Create a map of three (string, int) pairs
    std::map<std::string, int> m{{"CPU", 10}, {"GPU", 15}, {"RAM", 20}};

    print_map("1) Initial map: ", m);
```

```
    m["CPU"] = 25; // update an existing value
    m["SSD"] = 30; // insert a new value
    print_map("2) Updated map: ", m);

    // Using operator[] with non-existent key always performs an insert
    std::cout << "3) m[UPS] = " << m["UPS"] << '\n';
    print_map("4) Updated map: ", m);

    m.erase("GPU");
    print_map("5) After erase: ", m);

    std::erase_if(m, [](const auto& pair){ return pair.second > 25; });
    print_map("6) After erase: ", m);
    std::cout << "7) m.size() = " << m.size() << '\n';

    m.clear();
    std::cout << std::boolalpha << "8) Map is empty: " << m.empty() << '\n';
}
```

std::multiset

`std::multiset` is an associative container that contains a sorted set of objects of type Key. Unlike set, multiple keys with equivalent values are allowed.

```
 std::multiset<int> c = {1, 2, 3, 4, 1, 2, 3, 4};

    auto print = [&c]
    {
        std::cout << "c = { ";
        for (int n : c)
            std::cout << n << ' ';
        std::cout << "}\n";
    };
    print();
```

find - Finds an element with key equivalent to `key`. If there are several elements with the requested key in the container, any of them may be returned.

std::multimap

`std::multimap` is an associative container that contains a sorted list of key-value pairs, while permitting multiple entries with the same key.

Unordered associative containers (c++11 onwards) - unordered_set, unordered_map, unordered_multiset, unordered_multimap

std::unordered_set

`std::unordered_set` is an associative container that contains a set of unique objects of type `Key`. Search, insertion, and removal have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its value. This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into.

Container elements may not be modified (even by non const iterators) since modification could change an element's hash and corrupt the container.

Apart from usual functions, it also has bucket and hash related functions such as bucket_count (number of buckets), bucket_size (items in a bucket with index n), bucket (bucket for a key), hash_function (function used to hash the key)

std::unordered_map

`std::unordered_map` is an unsorted associative container optimized for search, removal and insertion which come at a constant time complexity. `std::unordered_map` is a hash map internally.

- accessing an element: with `at()` or with `operator[]` both have a complexity of `O(1)` on average and `O(n)` at worst where `n` is the size of the container
- inserting an element at a random location: with `insert()` or with `operator[]` both have a complexity of `O(1)` on average and `O(n)` at worst, where `n` is the size of the map. If `m` elements are inserted then the average case is `O(m)` and the worst case is `O(m * n + n)`
- removing an item: with `erase()` which has a complexity of amortized `O(1)` if the erasure happens with an iterator, otherwise, on average it's `O(m)` where `m` is the number of elements to remove, worst case it's `O(n)` where `n` is the size of the container
- finding an element: with `find()` it's `O(1)` on average and in the worst case it's `O(n)` where `n` is the size of the container

std::unordered_multiset

`std::unordered_multiset` is an associative container that contains set of possibly non-unique objects of type Key.

std::unordered_multimap

`std::unordered_multimap` is an unordered associative container that supports equivalent keys (an unordered_multimap may contain multiple copies of each key value) and that associates values of another type with the keys.

Thread safety

1. All container functions can be called concurrently by different threads on different containers. More generally, the C++ standard library functions do not read objects accessible by other threads unless those objects are directly or indirectly accessible via the function arguments, including the this pointer.
2. All `const` member functions can be called concurrently by different threads on the same container. In addition, the member functions `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `data()`, `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()`, and, except in associative containers, `operator[]`, behave as `const` for the purposes of thread safety (that is, they can also be called concurrently by different threads on the same container). More generally, the C++ standard library functions do not modify objects unless those objects are accessible, directly or indirectly, via the function's non-const arguments, including the this pointer.
3. Different elements in the same container can be modified concurrently by different threads, except for the elements of `std::vector<bool>` (for example, a vector of `std::future` objects can be receiving values from multiple threads).
4. Iterator operations (e.g. incrementing an iterator) read, but do not modify the underlying container, and may be executed concurrently with operations on other iterators on the same container, with the const member functions, or reads from the elements. Container operations that invalidate any iterators modify the container and cannot be executed concurrently with any operations on existing iterators even if those iterators are not invalidated.
5. Elements of the same container can be modified concurrently with those member functions that are not specified to access these elements. More generally, the C++ standard library functions do not read objects indirectly accessible through their arguments (including other elements of a container) except when required by its specification.
6. In any case, container operations (as well as algorithms, or any other C++ standard library functions) may be parallelized internally as long as this does not change the user-visible results (e.g. `std::transform` may be parallelized, but not `std::for_each` which is specified to visit each element of a sequence in order).

Algorithms

If you use raw loops and you understand the containers, you don't have to deal with these. A surprising "advantage" of using raw loops - please, prefer algorithms!

Otherwise, most probably you understand what standard algorithms do. Think about them and you'll be able to come up with their complexities in most cases. Let's have a look at some algorithms:

- `all_of`/ `any_of` / `none_of` have at most `O(n)` complexity where `n` is the size of the range the algorithm is applied on
- `count_if` has a complexity of `O(n)` where `n` is the size of the range the algorithm is applied on
- `find` / `find_if` have a complexity of `O(n)`. They need at most `n` applications of `operator==` or a predicate where `n` is the length of the range passed in
- `replace` / `replace_if` have a complexity of `O(n)`. They need `n` applications of `operator==` or of a predicate where `n` is the length of the range passed in and at most `n` assignments
- `copy` / `copy_if` have a complexity of `O(n)`. `copy` does `n` assignments where `n` is the length of the passed-in range, for `copy_if` we also have to think about the application of the predicate, while the number of assignments might be smaller.
- `transform` also has a complexity of `O(n)`. It performs exactly `n` applications of the operation, where `n` is the length of the passed-in range.
- `generate` has a complexity of `O(n)` as it invokes `n` times the generator function and also performs the same amount of assignments.
- `remove_if` has a complexity of `O(n)` as it performs `n` applications of `operator==` or of a predicate where `n` is the length of the range passed in.
- `swap` has a complexity of `O(1)` if applied on single values and `O(n)` if applied on arrays where `n` is the size of the arrays to be swapped

- `reverse` performs exactly half as many swaps as the size of the range to be reversed, therefore the complexity is `O(n)`
- `rotate` also has a complexity of `O(n)`.

Quite boring, right? But boredom brings simplicity to your calculations.

adjacent_find

Searches for two adjacent elements that are either equal or satisfy a specified condition.

```
// alg_adj_fnd.cpp
// compile with: /EHsc
#include <list>
#include <algorithm>
#include <iostream>

// Returns whether second element is twice the first
bool twice (int elem1, int elem2 )
{
    return elem1 * 2 == elem2;
}

int main()
{
    using namespace std;
    list<int> L;
    list<int>::iterator Iter;
    list<int>::iterator result1, result2;

    L.push_back( 50 );
    L.push_back( 40 );
    L.push_back( 10 );
    L.push_back( 20 );
    L.push_back( 20 );

    result2 = adjacent_find( L.begin( ), L.end( ), twice );
    if ( result2 == L.end( ) )
        cout << "There are not two adjacent elements where the "
            << "second is twice the first." << endl;
    else
    {
        cout << "There are two adjacent elements where "
```

```
            << "the second is twice the first.\n"
            << "They have values of " << *(result2++)
            << " & " << *result2 << "." << endl;
    }
}
```

all_of

Returns `true` when a condition is present at each element in the given range.

```
int main()
{
    using namespace std;

    list<int> li { 50, 40, 10, 20, 20 };
    list<int>::iterator iter;

    cout << "li = ( ";
    for (iter = li.begin(); iter != li.end(); iter++)
        cout << *iter << " ";
    cout << ")" << endl;

    // Check if all elements in li are even.
    auto is_even = [](int elem){ return !(elem % 2); };
    if (all_of(li.begin(), li.end(), is_even))
        cout << "All the elements are even numbers.\n";
    else
        cout << "Not all the elements are even numbers.\n";
}
```

any_of

Returns `true` when a condition is present at least once in the specified range of elements.

```
   // Check if there's an even element in li.
    auto is_even = [](int const elem){ return !(elem % 2); };
    if (any_of(li.begin(), li.end(), is_even))
        cout << "There's an even element in li.\n";
```

binary_search

Tests whether there's an element in a sorted range that is equal to a specified value or that is equivalent to it in a sense specified by a binary predicate.

```
   list<int> List1;

    List1.push_back( 50 );
    List1.push_back( 10 );
    List1.push_back( 30 );
    List1.push_back( 20 );
    List1.push_back( 25 );
    List1.push_back( 5 );

    List1.sort();

   // default binary search for 10
    if ( binary_search(List1.begin(), List1.end(), 10) )
        cout << "There is an element in list with a value equal to 10."
```

copy

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a forward direction.

```
vector<int> v1, v2; //v2 has 11 elements

// To copy the first 3 elements of v1 into the middle of v2
    copy( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 4 );
```

copy_backward

Assigns the values of elements from a source range to a destination range, iterating through the source sequence of elements and assigning them new positions in a backward direction.

```
   // To copy_backward the first 3 elements of v1 into the middle of v2
    copy_backward( v1.begin( ), v1.begin( ) + 3, v2.begin( ) + 7 );
```

copy_if

In a range of elements, copies the elements that are `true` for the specified condition.

```
   list<int> li{ 46, 59, 88, 72, 79, 71, 60, 5, 40, 84 };
   list<int> lo(li.size()); // lo = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

   // is_odd checks if the element is odd.
    auto is_odd = [](int const elem) { return (elem % 2); };
    // use copy_if to select only odd elements from li
    // and copy them to lo, starting from lo's begin position
    auto oc = copy_if(li.begin(), li.end(), lo.begin(), is_odd);
    lo.resize(std::distance(lo.begin(), oc));  // shrink lo to new size
```

copy_n

Copies a specified number of elements.

```
int main()
{
    using namespace std;
    string s1{"dandelion"};
    string s2{"badger"};

    cout << s1 << " + " << s2 << " = ";

    // Copy the first 3 letters from s1
    // to the first 3 positions in s2
    copy_n(s1.begin(), 3, s2.begin());

    cout << s2 << endl;
}
```

count

Returns the number of elements in a range whose values match a specified value.

```
    vector<int>::iterator::difference_type result;
    result = count(v1.begin(), v1.end(), 10);
    cout << "The number of 10s in v2 is: " << result << "." << endl;
```

count_if

Returns the number of elements in a range whose values satisfy a specified condition.

```
    vector<int>::iterator::difference_type result1;
    result1 = count_if(v1.begin(), v1.end(), [](auto x) {return x > 10;});
    cout << "The number of elements in v1 greater than 10 is: "
         << result1 << "." << endl;
```

equal

Compares two ranges element by element for equality or equivalence in a sense specified by a binary predicate.

```
bool   b = equal(v1.begin(), v1.end(), v3.begin(), v3.end());
```

fill

Assigns the same new value to every element in a specified range.

```
   // Fill the last 5 positions with a value of 2
    fill( v1.begin( ) + 5, v1.end( ), 2 );
```

fill_n

Assigns a new value to a specified number of elements in a range beginning with a particular element.

```
   // Fill the first 3 positions with a value of 1, saving position.
    auto pos = fill_n( v.begin(), 3, 1 );
```

find

Locates the position of the first occurrence of an element in a range that has a specified value.

find_first_of

Searches for the first occurrence of any of several values within a target range.

```
   // Searching v1 for a match to L1 under the binary predicate twice
    vector<int>::iterator result2;
    vector<int> v1, v2;
    result2 = find_first_of ( v1.begin( ), v1.end( ), v2.begin( ),
                              v2.end(), twice );

// Return whether second element is twice the first
bool twice ( int elem1, int elem2 )
{
    return 2 * elem1 == elem2;
}
```

find_if

Locates the position of the first occurrence of an element in a range that satisfies a specified condition.

```
// Function to use as the UnaryPredicate argument to find_if() in this
example
bool is_odd_int(int i) {
    return ((i % 2) != 0);
}

   // call <algorithm> std::find_if()
    auto p = find_if(first, last, is_odd_int);
```

for_each

Applies a specified function object to each element in a forward order within a range and returns the function object.

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

// The function object multiplies an element by a Factor
template <class Type>
class MultValue
{
private:
    Type Factor;   // The value to multiply by
public:
    // Constructor initializes the value to multiply by
    MultValue ( const Type& value ) : Factor ( value ) {
    }

    // The function call for the element to be multiplied
    void operator( ) ( Type& elem ) const
    {
        elem *= Factor;
    }
};

int main()
{
    using namespace std;
    vector<int> v1;
    vector<int>::iterator Iter1;

    // Constructing vector v1
    int i;
    for ( i = -4 ; i <= 2 ; i++ )
    {
        v1.push_back( i );
    }

    cout << "Original vector v1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;

    // Using for_each to multiply each element by a Factor
    //for_each ( v1.begin( ), v1.end( ), MultValue<int> ( -2 ) );
    for_each ( v1.begin( ), v1.end( ), [](int& k) { k = k * -2;} );
```

```
    cout << "Multiplying the elements of the vector v1\n "
            << "by the factor -2 gives:\n v1mod1 = ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}

/*
Original vector v1 = ( -4 -3 -2 -1 0 1 2 ).
Multiplying the elements of the vector v1
 by the factor -2 gives:
 v1mod1 = ( 8 6 4 2 0 -2 -4 ).
*/
```

generate

Assigns the values generated by a function object to each element in a range.
The function object is invoked for each element in the range and doesn't need to return the same value each time it's called. It is called with no arguments to generate the values to be assigned to each of the elements in the range.

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

int main()
{
    using namespace std;

    // Assigning random values to vector integer elements
    vector<int> v1 ( 5 );
    vector<int>::iterator Iter1;

    generate ( v1.begin( ), v1.end( ), rand );

    cout << "Vector v1 is ( " ;
    for ( Iter1 = v1.begin( ) ; Iter1 != v1.end( ) ; Iter1++ )
        cout << *Iter1 << " ";
    cout << ")." << endl;
}
```

is_sorted

Returns `true` if the elements in the specified range are in sorted order.

remove

Eliminates a specified value from a given range without disturbing the order of the remaining elements. Returns the end of a new range free of the specified value.

```
   // Remove elements with a value of 7
    auto new_end = remove ( v1.begin( ), v1.end( ), 7 );

    // To change the sequence size, use erase
    v1.erase (new_end, v1.end( ) );

Always use erase() after remove() to resize.
```

replace/replace_if

Examines each element in a range and replaces it if it matches a specified value.

```
   replace_if(v.begin(), v.end(), [](int k){ return k > 6; }, 70);
```

reverse

Reverses the order of the elements within a range.

```
   // Reverse the elements in the vector
    reverse (v1.begin( ), v1.end( ) );
```

sort

Arranges the elements in a specified range into a nondescending order or according to an ordering criterion specified by a binary predicate.

```
   // To sort in descending order. specify binary predicate
    sort( v1.begin( ), v1.end( ), greater<int>( ) );

// Return whether first element is greater than the second
bool UDgreater ( int elem1, int elem2 )
{
    return elem1 > elem2;
}

    // A user-defined (UD) binary predicate can also be used
    //sorts in Desc order as above
    sort( v1.begin( ), v1.end( ), UDgreater );
```

swap

This exchanges the values of two objects. = `swap( v1, v2 );`

transform

Applies a specified function object to each element in a source range or to a pair of elements from two source ranges. Then, it copies the return values of the function object into a destination range.

```
   // Using transform to multiply each element by a factor of 5
    transform ( v1.begin( ), v1.end( ), v2.begin( ), MultValue<int> ( 5 )
);
```

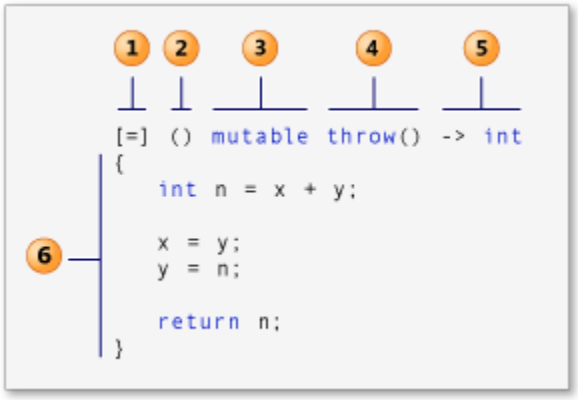
unique

Removes duplicate elements that are next to each other in a specified range.

Conclusion

In this article, we talked about the complexity analysis of operations on containers and of algorithms which are so often make important part of a software developer job interview. We discussed some hints on how to approach such questions if you neglected complexity analysis during most of your preparation for interviews. Finally, we quickly went through the most important complexities of C++ containers and standard algorithms so that you can have the most basic characteristics that you'd need at a job interview. Good luck!

Lambdas
https://learn.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170



1. *capture clause* (Also known as the *lambda-introducer* in the C++ specification.) [&, =]
2. *parameter list* Optional. (Also known as the *lambda declarator*)
3. *mutable specification* Optional.
4. *exception-specification* Optional.
5. *trailing-return-type* Optional.
6. *lambda body*.

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};       // OK
    [&, &i]{};      // ERROR: i preceded by & when & is the default
    [=, this]{};    // ERROR: this when = is the default
    [=, *this]{ }; // OK: captures this by value. See below.
```

```
    [i, i]{};       // ERROR: i repeated
}
```

A capture followed by an ellipsis is a pack expansion, as shown in this variadic template example:

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

To use lambda expressions in the body of a class member function, pass the `this` pointer to the capture clause to provide access to the member functions and data members of the enclosing class.

When you use the capture clause, we recommend that you keep these points in mind, particularly when you use lambdas with multi-threading:

- Reference captures can be used to modify variables outside, but value captures can't. (`mutable` allows copies to be modified, but not originals.)
- Reference captures reflect updates to variables outside, but value captures don't.
- Reference captures introduce a lifetime dependency, but value captures have no lifetime dependencies. It's especially important when the lambda runs asynchronously. If you capture a local by reference in an async lambda, that local could easily be gone by the time the lambda runs. Your code could cause an access violation at run time.

Generalized capture (C++14)

In C++14, you can introduce and initialize new variables in the capture clause, without the need to have those variables exist in the lambda function's enclosing scope. The initialization can be expressed as any arbitrary expression; the type of the new variable is deduced from the type produced by the expression. This feature lets you capture move-only variables (such as `std::unique_ptr`) from the surrounding scope and use them in a lambda.

C++

Copy

```
pNums = make_unique<vector<int>>(nums);

//...

      auto a = [ptr = move(pNums)]()

        {

           // use ptr

        };
```

Examples:

```
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto
variable.
    auto f1 = [](int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

The Microsoft C++ compiler binds a lambda expression to its captured variables when the expression is declared instead of when the expression is called.

```
// declaring_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>
```

```
int main()
{
    using namespace std;

    int i = 3;
    int j = 5;

    // The following lambda expression captures i by value and
    // j by reference.
    function<int (void)> f = [i, &j] { return i + j; };

    // Change the values of i and j.
    i = 22;
    j = 44;

    // Call f and print its result.
    cout << f() << endl;
}

Output is: 47 <<<< 3 + 44
```

This example declares a lambda expression that returns the sum of two integers and calls the expression immediately with the arguments `5` and `4`:

```
// calling_lambda_expressions1.cpp
// compile with: /EHsc
#include <iostream>

int main()
{
    using namespace std;
    int n = [] (int x, int y) { return x + y; }(5, 4);
    cout << n << endl;
}
```

This example passes a lambda expression as an argument to the `find_if` function. The lambda expression returns `true` if its parameter is an even number.

```
// calling_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
```

```
    using namespace std;

    // Create a list of integers with a few initial elements.
    list<int> numbers;
    numbers.push_back(13);
    numbers.push_back(17);
    numbers.push_back(42);
    numbers.push_back(46);
    numbers.push_back(99);

    // Use the find_if function and a lambda expression to find the
    // first even number in the list.
    const list<int>::const_iterator result =
        find_if(numbers.begin(), numbers.end(),[](int n) { return (n % 2)
== 0; });

    // Print the result.
    if (result != numbers.end()) {
        cout << "The first even number in the list is " << *result << "."
<< endl;
    } else {
        cout << "The list contains no even numbers." << endl;
    }
}

Output: The first even number in the list is 42.
```

Nested Lambda: You can nest a lambda expression inside another one, as shown in this example. The inner lambda expression multiplies its argument by 2 and returns the result. The outer lambda expression calls the inner lambda expression with its argument and adds 3 to the result.

```
// nesting_lambda_expressions.cpp
// compile with: /EHsc /W4
#include <iostream>

int main()
{
    using namespace std;

    // The following lambda expression contains a nested lambda
    // expression.
    int timestwoplusthree = [](int x) { return [](int y) { return y * 2;
}(x) + 3; }(5);

    // Print the result.
```

```
    cout << timestwoplusthree << endl;
}
```

`Higher order Lambda Functions:` Many programming languages support the concept of a *higher-order function.* A higher-order function is a lambda expression that takes another lambda expression as its argument or returns a lambda expression. You can use the function class to enable a C++ lambda expression to behave like a higher-order function. The following example shows a lambda expression that returns a `function` object and a lambda expression that takes a `function` object as its argument.

```
// higher_order_lambda_expression.cpp
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    // another lambda expression that adds two numbers.
    // The returned lambda expression captures parameter x by value.
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The following code declares a lambda expression that takes another
    // lambda expression as its argument.
    // The lambda expression applies the argument z to the function f
    // and multiplies by 2.
    auto higherorder = [](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}
```

`Lambda in a Function:` You can use lambda expressions in the body of a function. The lambda expression can access any function or data member that the enclosing

function can access. You can explicitly or implicitly capture the `this` pointer to provide access to functions and data members of the enclosing class.

```
// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [*this](int n) { cout << n * _scale << endl; });
}

// capture "this" implicitly
void ApplyScale(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [=](int n) { cout << n * _scale << endl; });
}

// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
    // and the scale value to the console.
    void ApplyScale(const vector<int>& v) const
    {
        for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale <<
endl; });
    }
```

```
private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}
```

`Using Lambda with templates:` Because lambda expressions are typed, you can use them with C++ templates. The following example shows the `negate_all` and `print_all` functions. The `negate_all` function applies the unary `operator-` to each element in the `vector` object. The `print_all` function prints each element in the `vector` object to the console.

```
// template_lambda_expression.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

// Negates each element in the vector object. Assumes signed data type.
template <typename T>
void negate_all(vector<T>& v)
{
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });
}

// Prints to the console each element in the vector object.
template <typename T>
void print_all(const vector<T>& v)
{
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });
}
```

```
int main()
{
    // Create a vector of signed integers with a few elements.
    vector<int> v;
    v.push_back(34);
    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all():" << endl;
    print_all(v);
}

Output:
34
-43
56
After negate_all():
-34
43
-56
```

`Handling Exceptions:` The body of a lambda expression follows the rules for both structured exception handling (SEH) and C++ exception handling. You can handle a raised exception in the body of a lambda expression or defer exception handling to the enclosing scope. The following example uses the `for_each` function and a lambda expression to fill a `vector` object with the values of another one. It uses a `try`/`catch` block to handle invalid access to the first vector.

```
// eh_lambda_expression.cpp
// compile with: /EHsc /W4
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    // Create a vector that contains 3 elements.
    vector<int> elements(3);

    // Create another vector that contains index values.
    vector<int> indices(3);
    indices[0] = 0;
```

```
    indices[1] = -1; // This is not a valid subscript. It will trigger an
exception.
    indices[2] = 2;

    // Use the values from the vector of index values to
    // fill the elements vector. This example uses a
    // try/catch block to handle invalid access to the
    // elements vector.
    try
    {
        for_each(indices.begin(), indices.end(), [&](int index) {
            elements.at(index) = index;
        });
    }
    catch (const out_of_range& e)
    {
        cerr << "Caught '" << e.what() << "'." << endl;
    };
}
```

`Managed C++/CLI:` The capture clause of a lambda expression cannot contain a variable that has a managed type. However, you can pass an argument that has a managed type to the parameter list of a lambda expression. The following example contains a lambda expression that captures the local unmanaged variable `ch` by value and takes a System.String object as its parameter.

```
// managed_lambda_expression.cpp
// compile with: /clr
using namespace System;

int main()
{
    char ch = '!'; // a local unmanaged variable

    // The following lambda expression captures local variables
    // by value and takes a managed String object as its parameter.
    [=](String ^s) {
        Console::WriteLine(s + Convert::ToChar(ch));
    }("Hello");
}
```

Function Objects vs. Lambdas

When you write code, you probably use function pointers and function objects to solve problems and perform calculations, especially when you use C++ Standard Library algorithms. Function pointers and function objects each have advantages and disadvantages—for example, function pointers have minimal syntactic overhead but do not retain state within a scope, and function objects can maintain state but require the syntactic overhead of a class definition.

A lambda combines the benefits of function pointers and function objects and avoids their disadvantages. Like a function object, a lambda is flexible and can maintain state, but unlike a function object, its compact syntax doesn't require an explicit class definition. By using lambdas, you can write code that's less cumbersome and less prone to errors than the code for an equivalent function object.

The following examples compare the use of a lambda to the use of a function object. The first example uses a lambda to print to the console whether each element in a `vector` object is even or odd. The second example uses a function object to accomplish the same task.

Example 1: Using a Lambda

This example passes a lambda to the for_each function. The lambda prints a result that states whether each element in a `vector` object is even or odd.

```
// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
   // Create a vector object that contains 9 elements.
   vector<int> v;
   for (int i = 1; i < 10; ++i) {
      v.push_back(i);
   }

   // Count the number of even numbers in the vector by
   // using the for_each function and a lambda.
   int evenCount = 0;
   for_each(v.begin(), v.end(), [&evenCount] (int n) {
```

```
        cout << n;
        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++evenCount;
        } else {
            cout << " is odd " << endl;
        }
    });

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
         << " even numbers in the vector." << endl;
}
```

```
Output:
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

Comments

In the example, the third argument to the for_each function is a lambda. The `[&evenCount]` part specifies the capture clause of the expression, `(int n)` specifies the parameter list, and remaining part specifies the body of the expression.

Example 2: Using a Function Object

Sometimes a lambda would be too unwieldy to extend much further than the previous example. The next example uses a function object instead of a lambda, together with the for_each function, to produce the same results as Example 1. Both examples store the count of even numbers in a `vector` object. To maintain the state of the operation, the `FunctorClass` class stores the `m_evenCount` variable by reference as a member variable. To

perform the operation, FunctorClass implements the function-call operator, operator(). The Microsoft C++ compiler generates code that is comparable in size and performance to the lambda code in Example 1. For a basic problem like the one in this article, the simpler lambda design is probably better than the function-object design. However, if you think that the functionality might require significant expansion in the future, then use a function object design so that code maintenance will be easier.

For more information about the operator(), see Function Call. For more information about the for_each function, see for_each.

```
// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};
```

```
int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

```
Output:
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```