Algorithm Library Stuff

Matt Warner

1 History

Pre C++11 (before lambdas were introduced) implementations of the algorithm library were achieved with custom function objects. Which might look something like this:

```
struct Stats {
    int size = 0;
    int sum_of_elements = 0;

    // Custom function object that overrides the call operator
    void operator()(int v) {
        size++;
        sum_of_elements+= v;
    }

    std::vector<int> vec{1,2,3,4,5,6,7};
    // Using for_each to get vector size and sum of elements
    Stats results = std::for_each(vec.begin(), vec.end(), Stats{});
    std::cout << results.size << " | " << results.sum_of_elements << std::endl;
    // Prints -> 7 | 28
} ;
```

In this example, we are using **for_each** with a custom function object which is used to fill the temporary object created with **Stats{}** with the data obtained from each element in the vector. The temporary object is then being assigned to **results**, which holds the accumulated values.

Now, Since the release of C++11, we can use a lambda function in replace of a custom function object to acheive the same results while offering implementations that are far less verbose, which makes these implementations easier to read and understand.

```
int size = 0;
int sum_of_all_elements = 0;
std::vector<int> vec{1,2,3,4,5,6};
std::for_each(vec.begin(),vec.end(),[&](int elem){ ++size; sum_of_all_elements+=elem;});
```

C++17 standard introduced parallel algorithms that provide an easy way to speed up processing with minimal effort. All you need to do is to specify the desired execution model, and the library will take care of parallelizing the execution.

```
std::atomic<int> size = 0, sum_of_all_elements = 0;
std::vector<int> vec{1,2,3,4,5,6};

std::for_each(std::execution::par_unseq, data.begin(), data.end(),[&](int elem) {
    ++size;
    sum_of_all_elements+=elem;
});
```

Finally, the C++20 standard introduced a significant re-design in the form of **ranges** and **views**.

range versions of algorithms can now operate on ranges instead of ***begin*** and ***end*** iterators and views provide lazily evaluated versions of algorithms and utilies.

```
uint16_t size = 0, sum_of_all_element = 0;
std::vector<int> data = {1,2,3,4,5,6,7};
std::ranges::for_each(data, [&](int elem) {
    size++;
    sum_of_all_elements+= elem;
});
```

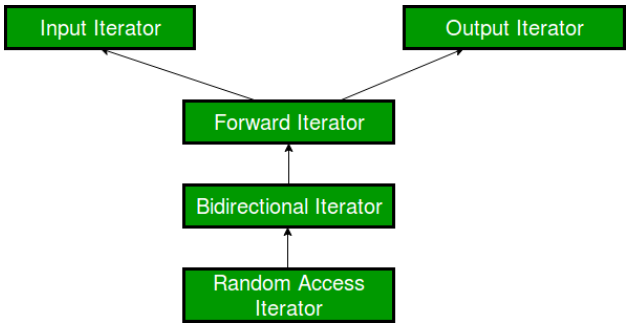
2 Iterators

Algorithms operate on data structures, which poses a issue. How do you abstract the implementation details of specific data structurese and allow the algorithm to work with any data structure that satisfies the algorithm's requirements?

The C++ standary library solution to this problem are iterators and ranges. Iterators encapsulate implementation details of data structure traversal and simultaneously expose a set of operations possible on the given data structure in constant time and space.

A range is then denoted by a pair of iterators, or more generally, since C++20, an iterator and a sentinel. In mathematical terms, a pair of iterators it1,it2 denotes a range [it1, it2), that is, the range includes the element referenced by it1 and ends before the element referenced by it2.

To reference the entire content of a data structure, we can use the begin() and end() methods that return an iterator to the first element and an iterator one past the last element, respectively. Hence, the range [begin, end) contains all data structure elements.



From the above hierarchy, it can be said that random-access iterators are the strongest of all iterator types.

Example of specifying a range using two iterators

```
std::vector<int> data = {1,2,3,4,5,6,7};

std::vector<int>::iterator it1 = data.begin();
std::vector<int>::iterator it2 = it1 + 2;

std::for_each(it1, it2, [](int elem) {
  std::cout << elem << ", ";
});
// Prints: 1, 2,
```

2.1 Iterator categories

The set of operations that are possible in constant time and space defines the following categories of iterators (and consequently ranges)

- **input/output iterator**: read/write each element once, advance
  data streams, e.g. writing/reading data to/from a network socket.
- **forward iterator** Can iterate forward only
  singly-linked list, e.g. std::forward_list
- **bidirectional iterator** Can iterate foward and backward
  doubly-linked list, e.g. std::list, std::map, std::set

- **random access iterator** Can jump to and compare with any other position.
  $=, *, ++, !=, --, -=, <, <=, \ldots[], -$
  std::vector, std::array, std::deque, raw arrays, strings
- **contiguous iterator** random access iterator + the storage of elements is contiguous
  arrays, e.g. std::vector

random access iterator vs bidirectional iterator

```
std::vector<int> arr = {1,2,3,4,5,6,7};
std::vector<int>::iterator it1 = arr.begin();
it1 += 5;
++it1;
ssize_t dst1 = it1 - arr.begin();
// dst == 6
std::list<int> lst = {1,2,3,4,5,6,7};
std::list<int>::iterator it2 = lst.begin();
// it2+=5; Would not compile
std::advance(it2, 5); // Ok, linear advance by 5 steps
++it2;
// it2 - it.begin(); Would not compile
ssize_t dst2 = std::distance(lst.begin(), it2); // Ok, linear calc.
```

input/output iterators

This category of iterators is split into two types, the first type being **input iterators**. They are considered to be the weakest as well as the simplest among all the iterators available, based upon their functionality and what can be achieve using them. They are the iterators that can be used in sequential input operations, where each value pointed by the iterator is read-only once and then the iterator is incremented.

Input iterators can be used only with single-pass algorithms, i.e., algorithms in which we can go to all the locations in the range at most once, like when we have to search or find any element in the range, we go through the locations at most once.

An input iterator can be compared for equality with another iterator. Since, iterators point to some location, the two iterators will be equal if they point to the same location. So, the following two expressions are valid:

```
A == B
A != B
```

An input iterator can be dereferenced, using the operator* and operator-¿ as an rvalue to obtain the value stored at the position being pointed to by the iterator

An input iterator can be incremented, so that it refers to the next element in the sequence, using operator++()

Note:-

The fact that we can use input iterators with increment doesn't mean that we can decrement. Input iterators are unidirectional and can only move forward.

Input iterators are also swappable, meaning we can exchange or swap the values they point to.

After understanding its features and deficiencies, it is very important to learn about its practical implementations as well. As told eariler, input iterators are used only when we want to access elements and not when we have to assign elements to them. The following example demonstrates this:

```
// Definition of std::find() template
InputIterator find (InputIterator first, InputIterator last, const T&amp; val)
{
  while (first != last) {
    if (*first==val) { return first; }
    ++first;
  }
  return last;
}
```

So, this is the practical implementation of input iterators, single-pass algorithms where only we have to move sequentially and access the elements and check for equality.

Output iterators are the exact opposite of input iterators, as they perform the opposite function of input iterators. They can be assigned values in a sequence, but cannot be used to access values.

Just like with input iterators, output iterators can be used only with single-pass algorithms, i.e., algorithms in which we can go to all the locations in the range at most once, such that these locations can be dereferened or asigned value only once.

Unlike input iterators, output iterators cannot be compared for equality with another iterator. So, the following two expressions are invalid if A and B are output iterators:

```
A == B  // Invalid
A != B // Invalid
```

An input iterator can be dereferenced as an rvalue, using operator* and →, whereas an output iterator can be dereferenced as an lvalue to provide the location to store the value. So, the following two expressions are valid if A is an output iterator:

```
*A = 1; // Dereferencing using *
A->m_v = 7; // Assigning a memeber variable m_v
```

An output iterator can be incremented, so that it refers to the next element in sequence, using operator++(). So, the following two expressions are valid if A is an output iterator.

```
A++
++A
```

The value pointed to by these iterators can be exchanged or swapped.

As told eariler, output iterators are used only when we want to assign elements and not when we have to access elements. The following STL algorithm can show this fact:

```
// Definition of std::move()
template
OutputIterator move(InputIterator first, InputIterator last, OutputIterator result) {
  while (first != last) {
    *result = std::move(*first);
    ++result;
    ++first;
  }
  return result;
}
```

Here, since the result is the iterator to the resultant container, to which elements are assigned, we cannot use input iterators and have made use of output iterators at their place, whereas for accessing elements, input iterators are used which only needs to be incremented and accessed. After looking at the features of output iterators, we should also look at their deficiencies as well, which are mentioned in the following points:

- **Only assigning, no accessing**: One of the biggest deficiencies is that we cannot access the output iterators as rvalue. So, an output iterator can only modify the element to which it points to by being used as the target for an assignment.

```
int main () {
  std::vector<int> v1 = {1,2,3,4};

  std::vector<int>::iterator i1;
  for (i1 = v1.begin(), i1 != v1.end(), ++i1) {
    *i1 = 1;
  }
  // v1 becomes 1 1 1 1
  return 0;
}
```

The above example shows assigning values using an output iterator, however, if we do something like this:

```
a = *i1; // where a is a variable
```

That is not allowed for output iterators, as they can only be the target in assignment.

Note:-

If you test this above code, it will work since vector's iterators are random access iterators

- **Cannot be decremented**: Just like we can use operator++() with output iterators for incrementing them, we cannot decrement them.
- **Use in multi-pass algorithms**: Since it is unidirectional and can only move forward, such iterators cannot be used in multi-pass algorithms, in which we need to move through the container multiple times.
- **Relational Operators**: Just like output iterators cannot be used with equality operators, it also can not be used with other relational operators like =
- **Arithmetic Operators**: Similar to relational operators, they also can't be used with arithmetic operators like +,- and so on. This means that output iterators can only move in one direction.

forward iterators

Forward iterators are considered to be the combination of input as well as output iterators. It provides support to the functionality of both of them. It permits values to be both accessed and modified.

A forward iterator can be compared for equality with another iterator. Since, iterators point to some location, the two iterators will be equal if they point to the same location. So, the following two operations are valid for forward iterators

```
A == B
A != B
```

A forward iterator can be incremented, so that it refers to the next element in sequence, using operator++()

Note:-

The fact that we can use forward iterators with increment operator doesn't mean that operator–() can also be used with them. Remember that forward iterators are unidirectional and can only move in the forward direction.

The value pointed to by these iterators are swappable as well, this is shown in the example code below:

```
template <typename T, typename ForwardIt>
void replace (ForwardIt first, ForwardIt last, const T& old_value, const T& new_value) {
  while (first != last) {
    if (*first == old_value) {
      *first = new_value;
    } ++it;
  }
}
```

Here, we have made use of forward iterators, as we need to make use of the feature of both input as well as output iterators.

bidirectional iterators

The concept **bidirectional_iterator** refines **forward_iterator** by adding the ability to move an iterator backward. Therefore, they allow for forward advancement as well as the ability to move backward.

It is to be noted that containers like **std::list**, **std::map**, **std::multimap**, **std::set** and **std::multiset** support bidirectional iterators.

Usability: Since, forward iterators can be used in multi-pass algorithms, i.e., algorithms which involve processing the container several times in various passes, therefore bidirectional iterators can also be used in multi-pass algorithms

Equality / Inequality Comparison: A Bidirectional iterator can be compared for equality with another iterator. Since, iterators point to some location, so the two iterators will be equal only when they point to the same position, otherwise not. So, the following two expressions are valid if A and B are Bidirectional iterators:

```
A == B // Checking for equality
A != B // Checking for inequality
```

Dereferencing: Because an input iterator can be dereferenced, using operator * and → as an rvalue and an output iterator can be dereferenced as an lvalue, so forward iterators being the combination of both can be used for both the purposes, and similarly, bidirectional opeartors can also serve both the purposes.

```
 // Definition of std::reverse_copy()
template OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator
↪  last, OutputIterator result)
{
  while (first != last) {
   *result++ = *--last;
  }
  return result;
}
```

Incrementable: A bidirectional iterator can be incremented, so that it refers to the next element in sequence, using operator++(). So, the following two expressions are valid if A is a bidirectional iterator.

```
A++ // Using post increment operator
++A // Using pre increment operator
```

Decrementable: This is the feature which differentiates a Bidirectional iterator from a forward iterator. Just like we can use operator++() with bidirectional iterators for incrementing them, we can also decrement them.

Limitations: Although Bidirectional iterators can be used with equality operator, they cannot be used with other relational operators like $\geq$ or $=$. Additionally, bidirectional iterators cannot use arithmetic, i.e., the following operations are not valid:

```
A + 1 // Not allowed
A - 1 // Not allowed
A += 2; // Not allowed
A = 5; // Not allowed
...
```

Random Access Iterators

Random-access iterators are a refined version of a **birdirectional_iterator**. They can be used to access elements at an arbitrary offset position relative to the element they point to, offering the same functionality as pointers.

Due to there support of constant time advancement, random access iterators are the most complete iterators in terms of functionality.

Note:-

All pointer types are also valid random-access iterators.

It is to be noted that containers like **std::vector**, **deque** support random-access iterators. This means that if we declare normal iterators for them, those iterators will be random access iterators

contiguos iterators (As of C++20)

The contiguous iterator concept refines random access iterators by providing a guarantee the denoted elements are stored contiguously in memory.

3 Naming and common behavior

There are a few common naming patterns that can be seen with STL algorithms.

3.1 Counted variants "_n"

Counted variants of algorithms accept the range specified using the start iterator and the number of elements (instead of begin and end). This behaviour can be a convenient alternative when working with input and output ranges, where we often do not have an explicit end iterator.

Examples: `std::for_each_n, std::copy_n`

Note:-

While `std::search_n` does follow the naming, it does not follow the same semantics. The _n here refers to the number of instances of the searched element.

3.2 Copy variants "_copy"

Copy variants of in-place algorithms do not write their output back to the source range. Instead, they ouput the result to one or more output ranges, usually defined by a single iterator denoting the first element to be written to (the number of elements is implied from the source range). The copy behaviour allows these variants to operate on immutable ranges.

Examples: `std::remove_copy, std::partial_sort_copy`

3.3 Predicate variants "_if"

Predicate variants of algorithms use a predicate to determine a "match" instead of comparing against a value. The standard also has one instance of _if_not variant that inverts the predicate logic (**false** is treated as a match.)

Examples: `std::find_if, std::replace_if`

3.4 Restrictions on invocable

Many algorithms can be customized using an invocable. However, with a few exceptions, the invocable is not permitted to modify elements of the range or invalidate iterators. On top of that, unless explicitly noted, the algorithms do not guarantee any particular order of invocation.

These restrictions in practice mean that the passed invocable must be regular. The invocable must return the same result if invoked again with the same arguments. This definition permits accessing a global state such as a cache but does not permit invocables that change their result based on their internal state (such as generators).

4 Sorting

4.1 sort

As the name implies, the **sort** function sorts a container. Heres an example:

```
std::vector<int> vec{4,5,7,8,4,2,5,8,3};

std::sort(vec.begin(), vec.end());
```

Pretty straighforward for primative data types like *char, int, double*. However, sorting strings requires more thought. In the example above, we simply called **std::sort** using **vec.begin()** and **vec.end()** and didn't bother using any sorting criteria. Including such criteria is unnecessary in the above example. But what if we had something like this:

```
std::vector<std::string> vec {
    "Hello", "Berlin", "are", "cities", "some", "Here", "Cologne", "LA", "London"
};
```

With this vector, only including **begin()** and **end()** will sort the array, but wont account for upper and lower case characters. Therefore, the output would look like this:

```
Berlin Cologne Hello Here LA London are cities some
```

In order to account for upper and lowercase characters we can use a lambda to add a sorting criteria:

```
std::sort(vec.begin(),vec.end(),[](std::string lhs, std::string rhs) {
    std::lexigraphical_compare(lhs.begin(), lhs.end(), rhs.begin(), rhs.end(), [] (char
    ↪ lhs, char rhs) {
        return std::toupper(lhs) < std::toupper(rhs);
    });
});
```

In the above example, we are using a lambda as a third parameter to compare two string objects, in the body of the lambda, we are calling another algorithm that performs lexigraphical compare on the two string objects in order to get an implementation of std::sort that is case-insensative.

4.2 stable_sort

stable sort is another STL algorithm that can be used to sort a container. **stable_sort** is very much like **sort** except for a few key differences. For starters, when using **stable_sort**, it is guarenteed that the semantically equivalent values will have their order preserved. Where as the same cannot be said for sort. However, the implication of this is that **std::stable_sort** cannot be performed quite as efficiently in terms of execution time.

Shown below is the two sorting algorithms time complexity:

std::sort: $O(N \cdot \log(N))$

std::stable_sort: $O(n \cdot \log^2(N))$ Knowing when to implement **stable_sort** instead of **sort** can be tricky. A good example of when to use it can be seen in the below example:

```
struct Neighbour {
   int floor;
   string name;
   Neighbour(int f, string n) : floor(f), name(n) {}
};
std::vector<Neighbour vec = {
Neighbour(1,"Bob"),
Neighbour(2, "Annie"),
Neighbour(3, "Peter")
Neighbour(4, "Bob")
Neighbour(5, "Larry")
};
```

If you now want to sort your list alphabetically, and you use

```
std::sort(vec.begin(),vec.end(), [](Neighbor a, Neighbour b){
    return a.name < b.name;
});
```

The results might be:

- 2 Annie
- 1 Bob
- 4 Bob
- 5 Larry
- 3 Peter

Or:

- 2 Annie
- 4 Bob
- 1 Bob
- 5 Larry
- 3 Peter

With **stable_sort**, it ensures that you always will get the first result. With the duplicates in the same order they were in the initial list.

4.3 partial_sort

Another sorting algorithm is **partial_sort**. This will sort only a sub-part of a container, rather than the entire thing. It rearanges the elements in the range [**first,last)**, in such a way that the elements before middle are sorted in ascending order, whereas the elements after middle are left without any specific order.

```
std::vector<int> vec = {10,45,60,78,23,21,3};
vector<int>::iterator it;
// Using std::partial_sort
std::partial_sort(vec.begin(), vec.begin + 1, vec.end());

// Displaying the smallest element after applying
// std::partial_sort
it = vec.begin();
std::cout << *it;
```

5 Finding stuff

5.1 find_if

the function call **find_if** can be used on a container like a vector to find a element in the container that satifies a given condition. For example, consider this code snippet that returns true if there is a element in the array greater than 10.

```
std::vector<int> coll{1,2,3,4,5,11};
auto it = std::find_if(coll.begin(), col.end(), [](int elem) { return elem > 10 ;});
return (it != coll.end()) ? 1 : 0;
```

5.2 find

We can also use **find** to find a specific elem in the container. Unlike **find_if**, which is checks for a condition, **find** uses **operator==** to find a given element. For example:

```

std::vector coll{1,2,3,4,5,6};
auto it = std::find(coll.begin(), coll.end(), 2);
if (*it.base() == 2) { return true; }
```

5.3 stable_partition

We can use **stable_partition** to partition a container. For example, lets say we have an array:

```
std::vector<int> coll{1,0,1,0,1,0,1,0};
```

And we wanted to partition the array such that all the zeros are on the left side of the array and everything else on the right. Using **stable_partition**, we could write:

```
std::stable_partition(coll.begin(),coll.end(),[](int elem) { return elem == 0 ;});
```

5.4 count_if

We can use **count_if** to perform some counting on a container. For example with vectors, if we wanted to count the number of odd elements:

```
std::vector coll{1,2,3,4,5,6};

// Using a lambda function to get the odd elements
int num = std::count_if(coll.begin(), coll.end(),[](int elem){return (elem % 2 != 0);});

std::cout << num << std::endl;
```