## Chapter 11 Practice Exercises

**Disclaimer:** These practice questions are not official and may not have been vetted for course material. You may use these for practice, but you should prioritize official resources from current staff for a more accurate depiction of what you need to know for assignments and exams.

1. Algorithms in the standard template library (STL) often work on iterator ranges. In the STL, what is the standard convention for determining which elements belong in an iterator range, given two iterators `first` and `last`?
   - **A)** `first` is inclusive, `last` is inclusive: `[first, last]`
   - **B)** `first` is inclusive, `last` is exclusive: `[first, last)`
   - **C)** `first` is exclusive, `last` is inclusive: `(first, last]`
   - **D)** `first` is exclusive, `last` is exclusive: `(first, last)`
   - **E)** None of the above

2. Which of the following statements is **FALSE** about the STL?
   - **A)** Using the STL can make your code more concise and readable
   - **B)** The STL allows reuse of algorithms with different data structures
   - **C)** The STL can help minimize explicit dynamic memory usage
   - **D)** The implementation of STL algorithms is proprietary and never visible to the public
   - **E)** None of the above

3. Consider the following statements regarding the STL's sorting algorithm, `std::sort()`.

```
template <typename RandomAccessIterator, typename Compare>
void std::sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```
Sorts the elements in the range `[first, last)` in ascending order, or using the comparator `comp` if specified.

   - **I.** The worst-case time complexity of `std::sort()` is $\Theta(n \log(n))$, given input size $n$
   - **II.** `std::sort()` can be used to sort any container that supports iterators
   - **III.** `std::sort()` can be used to sort a vector of strings

   Which of the above statements are **TRUE**?
   - **A)** I only
   - **B)** III only
   - **C)** I and II only
   - **D)** I and III only
   - **E)** I, II, and III

4. Which of the following statements is **FALSE**?
   - **A)** Input iterators and output iterators can only read and write in the forward direction
   - **B)** Bidirectional iterators share the same functionalities as forward iterators, but they also decrement
   - **C)** Forward iterators can access the same value more than once
   - **D)** The reverse iterator returned by `.rend()` points to the first element in a container
   - **E)** None of the above

5. Consider the following snippet of code:

```
1  int main() {
2    int32_t my_ints[] = {15, 21, 43, 40, 28, 50, 54, 35};
3    for (auto& val : my_ints) {
4      ++val;
5    } // for val
6
7    std::vector<int32_t> vec(my_ints, my_ints + 7);
8    std::sort(vec.begin(), vec.begin() + 4, std::greater<int32_t>());
9    std::reverse(vec.begin(), vec.end());
10   std::sort(vec.begin(), vec.begin() + 3, std::less<int32_t>());
11
12   auto it = vec.insert(vec.begin() + 5, 3, 19);
13   vec.erase(vec.begin() + 3, it);
14   std::swap(vec[1], vec[4]);
15   for (auto val : vec) {
16     std::cout << val << " ";
17   } // for val
18 } // main()
```

   What does this code output?
   - **A)** `29 19 55 19 51 19 41 44`
   - **B)** `36 19 55 19 51 19 22 41 44`
   - **C)** `16 19 51 19 36 19 29 41 44`
   - **D)** `29 19 55 22 51 19 19 41 44`
   - **E)** `29 19 55 19 51 41 44`

6. Consider the following snippet of code, which stores the total number of messages sent by EECS 281 staff members on the class Discord server (as of August 27, 2023):

```
1  int main() {
2    std::vector<std::pair<std::string, int32_t>> total_messages;
3    total_messages.emplace_back("slime", 70885);
4    total_messages.emplace_back("doubledelete", 34197);
5    total_messages.emplace_back("toafu", 30666);
6    total_messages.emplace_back("denalz", 15391);
7    total_messages.emplace_back("khuldraeseth", 25628);
8    total_messages.emplace_back("a-32", 24450);
9    std::sort(total_messages.begin(), total_messages.end());
10
11   auto rit = total_messages.rbegin();
12   for (int32_t i = 0; i < 4; ++i) {
13     ++rit;
14   } // for i
15
16   std::cout << rit->first << '\n';
17 } // main()
```

What does this code output?

**A)** `a-32`
**B)** `denalz`
**C)** `doubledelete`
**D)** `slime`
**E)** `toafu`

7. You have a vector of size 3, with three values A, B, and C. Given a comparator `comp`, you know that `comp(A, B)` returns `false`, `comp(B, C)` returns `true`, and `comp(A, C)` returns `true`. If you sort this vector using `std::sort()`, where `comp` is passed in as an argument, what are the final contents of the vector after the sorting is complete?

**A)** A, B, C
**B)** B, A, C
**C)** B, C, A
**D)** C, A, B
**E)** C, B, A

8. Which of the following segments of code produces a *different* outcome than all the others?

**A)**
```
std::vector<int32_t> a = {1, 2, 3, 4, 5};
std::vector<int32_t> b(a.size());
std::reverse_copy(a.begin(), a.end(), b.begin());
```

**B)**
```
std::vector<int32_t> a = {1, 2, 3, 4, 5};
std::vector<int32_t> b(a.size());
std::reverse_copy(a.rbegin(), a.rend(), b.begin());
std::reverse(b.begin(), b.end());
```

**C)**
```
std::vector<int32_t> a = {1, 2, 3, 4, 5};
std::vector<int32_t> b(a.size());
std::reverse_copy(a.rbegin(), a.rend(), b.rbegin());
```

**D)**
```
std::vector<int32_t> a = {1, 2, 3, 4, 5};
std::vector<int32_t> b(a);
std::reverse(b.begin(), b.end());
```

**E)**
```
std::vector<int32_t> a = {1, 2, 3, 4, 5};
std::vector<int32_t> b(a.rbegin(), a.rend());
std::reverse(b.rbegin(), b.rend());
```

9. Consider the following code, which attempts to multiply all the elements in a vector by two:

```
1  int main() {
2    std::vector<int32_t> vec = {1, 2, 3, 4, 5};
3    auto it = vec.begin();
4    while (it != vec.end()) {
5      it = it *= 2;
6    } // while
7  } // main()
```

The expected contents of `vec` after this code runs to completion is `[2, 4, 6, 8, 10]`. However, this code has a bug. Which of the following changes would fix this bug?

**A)** Change line 5 to `it *= 2;`
**B)** Change line 5 to `*it = *it * 2;`
**C)** Change line 5 to `*it = *it++ * 2;`
**D)** Change line 5 to `*it++ = *it * 2;`
**E)** More than one of the above

10. Suppose you had two `InputIterators`, `it1` and `it2`. Which of the following expressions are valid?

    **I.** `*it1++`
    **II.** `--it1`
    **III.** `it1 != it2`

    **A)** I only
    **B)** III only
    **C)** I and II only
    **D)** I and III only
    **E)** II and III only

11. You are given an iterator `it`. If the expression `*(it + 3)` is valid, what category of iterator is `it`?
    **A)** `it` is an `InputIterator`
    **B)** `it` is an `OutputIterator`
    **C)** `it` is a `BidirectionalIterator`
    **D)** `it` is a `ForwardIterator`
    **E)** `it` is a `RandomAccessIterator`

12. Suppose you had an algorithm that accepts `BidirectionalIterators` as arguments:

```
template <typename BidirectionalIterator>
void foo(BidirectionalIterator first, BidirectionalIterator last);
```

    Using the rules for bidirectional iterators covered in this chapter, which of the following types of iterators could you pass into the `foo()` function without any issues? *Select all that apply.*
    **A)** `ForwardIterator`
    **B)** `BidirectionalIterator`
    **C)** `InputIterator`
    **D)** `OutputIterator`
    **E)** `RandomAccessIterator`

13. Which of the following iterator types support(s) decrementation using `operator--`? *Select all that apply.*
    **A)** `ForwardIterator`
    **B)** `BidirectionalIterator`
    **C)** `InputIterator`
    **D)** `OutputIterator`
    **E)** `RandomAccessIterator`

14. Which of the following iterator types support(s) comparison using `operator<`? *Select all that apply.*
    **A)** `ForwardIterator`
    **B)** `BidirectionalIterator`
    **C)** `InputIterator`
    **D)** `OutputIterator`
    **E)** `RandomAccessIterator`

15. Which of the following best matches each of the specified containers with its iterator category?

    **A)**
```
std::vector<>: ForwardIterator
std::list<>:   ForwardIterator
std::deque<>:  ForwardIterator
```

    **B)**
```
std::vector<>: ForwardIterator
std::list<>:   BidirectionalIterator
std::deque<>:  ForwardIterator
```

    **C)**
```
std::vector<>: BidirectionalIterator
std::list<>:   ForwardIterator
std::deque<>:  BidirectionalIterator
```

    **D)**
```
std::vector<>: RandomAccessIterator
std::list<>:   BidirectionalIterator
std::deque<>:  BidirectionalIterator
```

    **E)**
```
std::vector<>: RandomAccessIterator
std::list<>:   BidirectionalIterator
std::deque<>:  RandomAccessIterator
```

16. Consider the following code. What are the contents of `vec` after the code runs to completion?

```
int main() {
  std::vector<int32_t> vec = {19, 33, 52, 17, 37, 58, 39, 44};
  std::sort(vec.rbegin(), vec.rbegin() + 6);
} // main()
```

**A)** `[17, 19, 33, 37, 39, 52, 58, 44]`
**B)** `[17, 19, 33, 37, 52, 58, 39, 44]`
**C)** `[19, 58, 52, 44, 39, 37, 33, 17]`
**D)** `[19, 33, 58, 52, 44, 39, 37, 17]`
**E)** `[58, 52, 37, 33, 19, 17, 39, 44]`

17. Consider the following code. What are the contents of `vec` after the code runs to completion?

```
struct Compare {
  bool operator() (const int32_t lhs, const int32_t rhs) const {
    return std::abs(lhs - 281) < std::abs(rhs - 281);
  } // operator()()
};

int main() {
  std::vector<int32_t> vec = {100, 150, 200, 250, 300, 350, 400, 450, 500};
  std::sort(vec.begin(), vec.end(), Compare());
} // main()
```

**A)** `[250, 200, 150, 100, 500, 450, 400, 350, 300]`
**B)** `[300, 350, 400, 450, 500, 100, 150, 200, 250]`
**C)** `[500, 100, 450, 150, 400, 200, 350, 250, 300]`
**D)** `[300, 250, 350, 200, 450, 150, 500, 100, 400]`
**E)** `[300, 250, 350, 200, 400, 150, 450, 100, 500]`

18. You are given an unsorted vector of a million integers, and you want to find the $281^{\text{th}}$ largest element. In terms of asymptotic time complexity, which of the following STL algorithms would be most efficient for this task?
    **A)** `std::find()`
    **B)** `std::find_if()`
    **C)** `std::sort()`
    **D)** `std::partial_sort()`
    **E)** `std::nth_element()`

19. Which of the following statements is **TRUE** about `std::remove()`?
    **A)** A call to `std::remove()` (and nothing else) can reduce the size of the container it is called on
    **B)** A call to `std::remove()` (and nothing else) can reduce the capacity of the container it is called on
    **C)** A call to `std::remove()` will always return an iterator that points to the last element not removed
    **D)** A call to `std::remove()` should be followed with a call to the container's `.erase()` to clean up the data that was removed
    **E)** More than one of the above

20. Consider the following vector of integers:

| 2 | 3 | 4 | 4 | 4 | 5 | 6 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

    Suppose you called `std::lower_bound()` on the iterator range `[vec.begin(), vec.end())` using a target value of 4. Which of the following iterators is returned?
    **A)** An iterator pointing to index 1
    **B)** An iterator pointing to index 2
    **C)** An iterator pointing to index 3
    **D)** An iterator pointing to index 4
    **E)** An iterator pointing to index 5

21. Consider the same vector as in question 20. Suppose you called `std::upper_bound()` on the iterator range `[vec.begin(), vec.end())` using a target value of 4. Which of the following iterators is returned?
    **A)** An iterator pointing to index 1
    **B)** An iterator pointing to index 2
    **C)** An iterator pointing to index 3
    **D)** An iterator pointing to index 4
    **E)** An iterator pointing to index 5

22. Consider the same vector as in question 20. Suppose you called `std::upper_bound()` on the iterator range `[vec.begin(), vec.end())` using a target value of 6. Which of the following iterators is returned?
    **A)** An iterator pointing to index 4
    **B)** An iterator pointing to index 5
    **C)** An iterator pointing to index 6
    **D)** An iterator pointing to index 7
    **E)** None of the above

23. Which one of the following statements is **TRUE**?
    **A)** An iterator typically holds an address, and incrementing the iterator using `operator++` always increments that address
    **B)** For any valid STL container `c` that supports iterators, you can use the expression `c.end() - c.begin()` to return the same value as `c.size()`
    **C)** An iterator that points to an element in an STL container is guaranteed to remain valid throughout the entire lifetime of that container, until the container goes out of scope
    **D)** For any valid STL container `c`, the iterator returned by `c.end()` refers to the last valid element in `c`
    **E)** None of the above

24. What is the worst-case time complexity of `std::lower_bound()`, given that the container is sorted, supports random access iterators, and contains $n$ elements?
    **A)** $\Theta(1)$
    **B)** $\Theta(\log(n))$
    **C)** $\Theta(n)$
    **D)** $\Theta(n\log(n))$
    **E)** $\Theta(n^2)$

25. Given an *unsorted* vector of integers of size $n$, what is the best attainable average-case time complexity of finding the three largest elements in the vector?
    **A)** $\Theta(1)$
    **B)** $\Theta(\log(n))$
    **C)** $\Theta(n)$
    **D)** $\Theta(n\log(n))$
    **E)** $\Theta(n^2)$

26. Which of the following STL containers is *least* likely to invalidate existing iterators, pointers, and references when new elements are added?
    **A)** `std::vector<>`
    **B)** `std::deque<>`
    **C)** `std::array<>`
    **D)** `std::list<>`
    **E)** All of the above containers are equally likely to invalidate existing iterators, pointers, and references

27. Implement the STL's `unique_copy()` function according to its official interface and description.

```
template <typename ForwardIterator, typename OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last, OutputIterator result);
```

`unique_copy()` copies elements from the range `[first, last)` to a range beginning with `result`, except that in a consecutive group of duplicate elements only the first one is copied. The function then returns an iterator to the position one past the end of the range to which the elements are copied. For example, if you execute the following code:

```
int32_t data[6] = {1, 3, 3, 1, 1, 0};
int32_t output[6];
unique_copy(data, data + 6, output);
```

you would end up with the following contents of `output`:

| 1 | 3 | 1 | 0 | | |
|---|---|---|---|---|---|

You may **NOT** use any STL algorithms or functions, and the complexity of this function should be linear on the number of elements in the input range. For elements in the range, exactly `last - first` applications of `operator==()` and at most `last - first` assignments are performed.

28. Implement the STL's `set_difference()` function according to its official interface and description.

```
template <typename ForwardIterator1, typename ForwardIterator2,
          typename OutputIterator, typename Compare>
OutputIterator set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2,
                              OutputIterator result,   Compare comp);
```

The `set_difference()` function constructs a sorted range beginning in the location pointed to by `result` with the set difference of the sorted range `[first1, last1)` with respect to the sorted range `[first2, last2)`. The difference of two sets is formed by the elements that are present in the first set, but not in the second one. The elements copied by the function always come from the first range, in the same order. For containers supporting multiple occurrences of a value, the difference includes as many occurrences of a given value as in the first range, minus the number of matching elements in the second, preserving order. The elements are compared using the comparator `comp`. Two elements, `a` and `b`, are considered equivalent if `!comp(a, b) && !comp(b, a)`. For example, given the vectors:

```
first = [5, 10, 15, 20, 25]
second = [10, 20, 30, 40, 50]
```

the set difference constructed at `result` is `[5, 15, 25]` (elements in the first range not in the second). You may **NOT** use any STL algorithms or functions, and the complexity of this function should be linear on the number of elements in the input ranges.

29. Implement the STL's `minmax_element()` function according to its official interface and description.

```
template <typename ForwardIterator, typename Compare>
std::pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last,
                                                           Compare comp);
```

The `minmax_element()` function returns a pair with an iterator pointing to the element with the smallest value in the provided iterator range `[first, last)` as the `first` element, and an iterator pointing to the largest value in the range as the `second` element. The comparisons are performed using the `comp` comparator. If more than one equivalent element has the smallest value, the first iterator points to the *first* of such elements. If more than one equivalent element has the largest value, the second iterator points to the *last* of such elements.

For example, given the vector `vec = [3, 7, 6, 9, 5, 8, 2, 4]`, running the function with `vec.begin()` as the first argument, `vec.end()` as the second argument, and `operator<` as the comparator, a pair would be returned with an iterator to `2` as the first element and an iterator to `9` as the second element. You may **NOT** use any STL algorithms or functions, and your implementation must run in linear time on the number of elements in the input range.

30. Implement the STL's `replace_if()` function according to its official interface and description.

```
template <typename ForwardIterator, typename Predicate>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);
```

The `replace_if()` function replaces all elements in the range `[first, last)` for which `pred` returns `true` with `new_value`. For example, suppose we have a functor, `IsEven`, that returns `true` for integers that are even:

```
struct IsEven {
  bool operator() (const int32_t val) const {
    return val % 2 == 0;
  } // operator()()
};
```

In this case, running the following code would replace all even numbers in the input iterator range with the value `281`:

```
std::vector<int32_t> nums = {100, 105, 110, 115, 120, 125, 130, 135, 140, 145};
replace_if(nums.begin(), nums.end(), IsEven(), 281);
// contents of vector are now [281, 105, 281, 115, 281, 125, 281, 135, 281, 145]
```

You may **NOT** use any STL algorithms or functions, and your implementation must run in linear time on the number of elements in the input range.

---

## Chapter 11 Exercise Solutions

1. **The correct answer is (B).** When working with STL iterator ranges, the standard convention is that `first` is inclusive while `last` is exclusive.

2. **The correct answer is (D).** Many implementations of STL algorithms are public for anyone to view, so option (D) is false. Options (A), (B), and (C) are all true: the STL can make code more concise and readable, since you can just include and use a pre-implemented function that satisfies your needs; the STL allows reuse of algorithms with different data structures through the use of iterators; and the STL can minimize explicit dynamic memory usage by handling memory allocation behind the scenes for you.

3. **The correct answer is (D).** The STL algorithm library's `std::sort()` algorithm takes in two random access iterators. Containers that have iterators that do not support random access (such as lists) cannot be sorted using `std::sort()`. Statements I and III are true: `std::sort()` runs in worst-case $\Theta(n \log(n))$ time, and it can be used to sort a vector of strings (in fact, if a container supports random access iterators, `std::sort()` can sort the container as long as the type of its elements supports `operator<` or can be compared using the passed in comparator argument).

4. **The correct answer is (D).** The reverse iterator returned by `.rend()` points to the theoretical position *before* the first element in a container, and not the last element itself (you can think of this as analogous to `.end()`, which points to the position one past the end, and not the last element itself). All of the other options are true.

5. **The correct answer is (A).** On line 2, we create the following array:

`[15, 21, 43, 40, 28, 50, 54, 35]`

On line 4, we increment each value in the array by one.

`[16, 22, 44, 41, 29, 51, 55, 36]`

On line 7, we range construct a vector using all values from `my_ints` to `my_ints + 7`, not including the value of `my_ints + 7` since end is exclusive:

`[16, 22, 44, 41, 29, 51, 55]`

On line 8, we sort the vector from `vec.begin()` to `vec.begin() + 4`, not including the value at `vec.begin() + 4` since end is exclusive. Since `std::greater<>` is used, the elements in this range are sorted in descending order:

`[44, 41, 22, 16, 29, 51, 55]`

On line 9, we reverse the entire vector. All elements are included since `vec.end()` points one past the end of the vector:

`[55, 51, 29, 16, 22, 41, 44]`

On line 10, we sort the vector from `vec.begin()` to `vec.begin() + 3`, not including the value at `vec.begin() + 3` since end is exclusive. Since `std::less<>` is used, the elements in this range are sorted in ascending order:

`[29, 51, 55, 16, 22, 41, 44]`

On line 12, we insert three values before `vec.begin() + 5`, each initialized to a value of 19. `it` points to the first `19` (in bold):

`[29, 51, 55, 16, 22, 19, 19, 19, 41, 44]`

On line 13, we erase all values from `vec.begin() + 3` to `it`, not including the value at `it` itself:

`[29, 51, 55, 19, 19, 19, 41, 44]`

On line 14, we swap the element at index 1 with the one at index 4 to give us our answer: `[29, 19, 55, 19, 51, 19, 41, 44]`

6. **The correct answer is (B).** Pairs are sorted by their `first` value, and then their `second` value in the case of a tie. Thus, the call to `std::sort()` on line 9 sorts the elements in the following order:

   `a-32 denalz doubledelete khuldraeseth slime toafu`

   On line 11, we initialize a reverse iterator, which points to the last element in the sorted vector, or the pair associated with `"toafu"`.

   `a-32 denalz doubledelete khuldraeseth slime toafu`

   We then increment the reverse iterator four times, which moves it toward the front of the sorted vector by four positions:

   `a-32 denalz doubledelete khuldraeseth slime toafu`

   `a-32 denalz doubledelete khuldraeseth slime toafu`

   `a-32 denalz doubledelete khuldraeseth slime toafu`

   `a-32 denalz doubledelete khuldraeseth slime toafu`

   The value of this pair's `first` value is then printed out, which is `denalz`.

7. **The correct answer is (B).** If a comparator is passed in, the order of elements after sorting is determined using the following rule: given two elements that are passed into the comparator, a return value of `true` indicates that the first element comes before the second in sorted order; otherwise, it comes after. In this case, `comp(A, B)` indicates that `A` comes after `B`, `comp(B, C)` indicates that `B` comes before `C`, and `comp(A, C)` indicates that `A` comes before `C`. The final order would therefore be B, A, C.

8. **The correct answer is (E).** All of the options except for (E) inserts the elements of `a` into `b` in reverse order (i.e., `[5, 4, 3, 2, 1]`). Note that the iterator range determines how the algorithm views the order of elements: if you pass in a reverse iterator range, the algorithm essentially sees and acts upon the data in reverse order.

9. **The correct answer is (D).** Options (A) and (B) do not work since the iterator is never incremented, so it can get stuck in the `while` loop without ever exiting. For our desired result, we also want to multiply the value of the iterator by 2 first, before assigning it (which overwrites the original value). This is not done in option (C), which increments before assigning, thereby assigning to the wrong value (which yields the result `[1, 2, 4, 8, 16]` instead).

10. **The correct answer is (D).** Input iterators support increments, dereferencing, and `!=`. However, they do not support decrements. Thus, only I and III are valid.

11. **The correct answer is (E).** Of the provided iterators, only random access iterators support pointer arithmetic, as performed with `it + 3`.

12. **The correct answers are (B) and (E).** Any iterator type that is higher in the hierarchy (e.g., supports at least the same operations) than the iterator type required by the function argument can also be passed into the function. In this case, random access iterators are the only category that is higher that bidirectional iterators, so both of these iterator types can be passed into the function.

13. **The correct answers are (B) and (E).** Bidirectional and random access iterators are the only two that support decrementation.

14. **The correct answer is (E).** Only random access iterators support pointer comparison using `operator<`.

15. **The correct answer is (E).** Vectors and deques support random access iterators, while lists only support bidirectional iterators.

16. **The correct answer is (D).** The value at `vec.rbegin()` is `44`, and the value of `vec.rbegin() + 6` is `33`. The call to sort sorts all the values between these two values, excluding the value pointed to by the end iterator (e.g., the values of `[52, 17, 37, 58, 39, 44]` are sorted). However, since we passed in reverse iterators, the `std::sort()` function call essentially views the elements in reverse order, so the first value in sorted order is placed at the position of `vec.rbegin()` and the last value in sorted order is placed at the position of `vec.rbegin() + 5`. The ends up rearranging all the values after `33` in reverse sorted order (from highest to lowest), giving us a final vector of `[19, 33, 58, 52, 44, 39, 37, 17]`.

17. **The correct answer is (E).** Using this comparator, a value `A` comes before `B` if `std::abs(A - 281)` is less than `std::abs(B - 281)`. As a result, the vector ends up getting sorted in order of ascending distance from 281, which gives us the outcome in option (E).

18. **The correct answer is (E).** As its name implies, `std::nth_element()` can be used to find the $n^{\text{th}}$ value in sorted order without having to sort the entire container, which takes $\Theta(n)$ time. Options (A) and (B) can be used to find a target value, but it is not the best option if you want to find an element in an unsorted vector given its sorted position. Similarly, while (C) and (D) can make finding the $n^{\text{th}}$ easier by sorting the vector, these would require your algorithm to take $\Theta(n \log(n))$ time.

19. **The correct answer is (D).** Only option (D) is true, per the erase-remove idiom, since a call to `std::remove()` does not physically erase elements from its container. Options (A) and (B) are false for this reason: the STL algorithm alone does not reduce the structure of the underlying container. Option (C) is false because `std::remove()` returns an iterator the next available position after the removal, and not the last element not removed itself.

20. **The correct answer is (B).** A call to `std::lower_bound()` on a sorted range returns an iterator to the first element that does not compare less than the target argument of `val` (or the end of the given iterator range if this does not exist). In this case, this would be the first `4` in the vector, which is located at index 2.

21. **The correct answer is (E).** A call to `std::upper_bound()` on a sorted range returns an iterator to the first element that compares greater than the target argument of `val` (or the end of the given iterator range if this does not exist). In this case, this would be the value of `5`, which is located at index 5.

22. **The correct answer is (E).** A call to `std::upper_bound()` on a sorted range returns an iterator to the first element that compares greater than the target argument of `val` (or the end of the given iterator range if this does not exist). In this case, there is no value that compares greater than `6`, so `vec.end()` is returned.

23. **The correct answer is (E).** None of the statements are true. (A) is false because iterators provide an interface to the elements of a container and is not a perfect substitute for a memory address; incrementing an iterator does not always increment that element's address (e.g., for a list, the next element is not guaranteed to located in the next memory address). (B) is false because subtracting two iterators can only be done if a container supports random access iterators, and not just any type of iterator. (C) is false due to the presence of iterator invalidation; for containers like vectors, it is possible for the underlying data to be moved around in memory as the container grows in size. (D) is false because `.end()` returns the one past the end iterator, which points to the position directly after the last valid element.

24. **The correct answer is (B).** For a sorted container that supports random access iterators, `std::lower_bound()` can be done using a binary search, which takes $\Theta(\log(n))$ time on the number of elements $n$.

25. **The correct answer is (C).** This can be solved in linear time by using `std::nth_element()` to identify the third largest element in the vector, and then doing another linear pass to identify the two other values that are larger.

26. **The correct answer is (D).** The `std::list<>` is the only container of the ones provided that guarantees that validity of iterators throughout the lifetime of the element in the container. The other containers do not provide this guarantee (e.g., a vector's elements may be reallocated in memory after new elements are added).

27. One potential solution is shown below. In this implementation, we keep track of two adjacent iterators that iterate over the range in tandem. If the front iterator has a value that is different from the back iterator, then we write the value of the front iterator to the output iterator that is passed in; otherwise, the value of the front iterator is ignored and the two iterators are incremented without writing it to the output.

```
1  template <typename ForwardIterator, typename OutputIterator>
2  OutputIterator unique_copy(ForwardIterator first, ForwardIterator last, OutputIterator result) {
3    // check for empty range
4    if (first == last) {
5      return result;
6    } // if
7
8    *result++ = *first;  // write first value to output
9    ForwardIterator prev = first++;
10
11   while (first != last) {
12     if (!(*first == *prev)) {
13       *result++ = *first;
14     } // if
15     prev = first++;
16   } // while
17
18   return result;
19 } // unique_copy()
```

28. One potential solution is shown below. In this implementation, we iterate over the two iterator ranges (which we will denote as `A` and `B`, where `A` is the first range) until one reaches the end. When doing so, we compare the values at the two iterators. If `comp(A, B)`, we write the value of the iterator corresponding to `A` to the result, and then increment this iterator (note that `comp` determines how the iterator ranges are sorted). If `comp(B, A)`, we write the value of the iterator corresponding to `B` to the result, and then increment this iterator. Otherwise, the values are the same, so we increment both iterators. Lastly, if we reach the end of the second iterator range before the first one, we will write all remaining elements in the first iterator range to the result.

```
template <typename ForwardIterator1, typename ForwardIterator2,
          typename OutputIterator, typename Compare>
OutputIterator set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2,
                              OutputIterator result,   Compare comp) {
  while (first1 != last1 && first2 != last2) {
    if (comp(*first1, *first2)) {
      *result++ = *first1++;
    } // if
    else if (comp(*first2, *first1)) {
      ++first2;
    } // else if
    else {
      ++first1;
      ++first2;
    } // else
  } // while

  // write all remaining elements in the first iterator range to the result
  // if there are still elements left after first2 reaches last2
  while (first1 != last1) {
    *result++ = *first1++;
  } // while

  return result;
} // set_difference()
```

29. One potential solution is shown below. In this implementation, we iterate over the given iterator range and keep track of the largest and smallest values we have encountered so far. At each element, we update the min value if the element is smaller than the best encountered, and we update the max value if the element is larger than the best encountered. Then, we return a pair containing the min and max.

```
template <typename ForwardIterator, typename Compare>
std::pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first,
                                                           ForwardIterator last, Compare comp) {
  ForwardIterator min = first, max = first;
  for (; first != last; ++first) {
    if (comp(*first, *min)) {
      min = first;
    } // if
    if (!comp(*first, *max)) {
      max = first;
    } // if
  } // for

  return {min, max};
} // minmax_element()
```

30. This can be implemented by iterating over the iterator range, checking if each element satisfies `pred`, and replacing with `new_value` if it does. An implementation of this solution is shown below:

```
template <typename ForwardIterator, typename Predicate>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value) {
  for (; first != last; ++first) {
    if (pred(*first)) {
      *first = new_value;
    } // if
  } // for
} // replace_if()
```