Whenever you have a forwarding reference in the following format:

```
template <typename T>
void foo(T&& arg);
```

the provided reference collapsing rules make it possible to determine whether `foo()` was called on an lvalue or an rvalue. If an lvalue reference of type `Thing` were passed to `foo()`, then `foo(T&&)` gets deduced as `foo(Thing&&)`, which resolves to `foo(Thing&)` (and thus the compiler knows that `foo()` must have been called on an lvalue reference). On the other hand, if an rvalue reference of type `Thing` were passed to `foo()`, then `foo(T&&)` gets deduced as `foo(Thing&&&&)`, which resolves to `foo(Thing&&)` (and thus the compiler knows that `foo()` must have been called on an rvalue reference).

Perfect forwarding and forwarding references are quite valuable: they can be used to avoid excessive copying, and they also allow you to write methods that support many different input value types without having to implement multiple overloads (e.g., one version that accepts lvalues, another version that accepts rvalues, etc.). One of the best use cases of perfect forwarding occurs when an object needs to be moved through multiple function calls between its point of creation and its destination. This strategy is employed by many standard library functions to simplify the process of object creation (one notable example we will cover in the next chapter is `std::vector<>::emplace_back(Args&&... args)`, which takes in a set of constructor arguments and forwards it to the constructor of the object to be created, which is then directly constructed at the back of the vector).

Chapter 6 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. Which of the following statements is/are **TRUE** about arrays in C++?
   - **I.** The size of an array may be changed during runtime.
   - **II.** Elements in an array are contiguous in memory.
   - **III.** Given two arrays of the same type `a1` and `a2`, the expression `a1 = a2` performs an element-by-element copy from `a2` into `a1`.

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

2. Suppose you had a 2-D array with 7 rows and 9 columns. You are told that an element *k* is located at row index 5 and column index 6. Zero indexing is used. If this 2-D array were instead represented as a 1-D array, what would the index of *k* be?
   A) 11
   B) 30
   C) 42
   D) 51
   E) 60

3. Consider the following snippet of code:

```
1   int main () {
2     constexpr size_t sz = 5;
3     int32_t arr[sz];
4     int32_t eecs = 281;
5
6     for (int& i : arr) {
7       i = eecs++;
8     } // for i
9
10    for (int32_t j = 0; j < sz; ++j) {
11      arr[j] = arr[j + 1];
12    } // for j
13
14    for (int32_t k = 1; k <= sz; ++k) {
15      arr[k - 1] = k;
16    } // for k
17  } // main()
```

   The above code has a bug. On which line is this bug located?
   A) Line 3
   B) Line 6
   C) Line 7
   D) Line 10
   E) Line 14

4. Which of the following statements is **FALSE**?
   A) If a container stores pointers to dynamic memory, only the copy constructor needs to be overloaded to support deep copies
   B) A drawback of storing data by value in a container is that large objects may be expensive to copy
   C) If pointers to data are stored in a container, data ownership may not be exclusive to that container
   D) It is typically safer to return an element in a container by reference than it is to return a pointer to the element
   E) None of the above

5. What is the worst-case time complexity of copying an array of n elements to another array, if you use the most efficient algorithm?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

6. The copy-swap method is a technique that is primarily designed for implementing/overloading which one of the following?
   A) Constructor
   B) Copy Constructor
   C) Destructor
   D) Assignment Operator
   E) None of the above

7. Which of the following statements is **TRUE** regarding the storage of data in a container?
   A) A container that stores primitive types by value has the drawback of taking a long time to copy, and is thus not often used
   B) Returning a pointer to data stored inside a container could be dangerous, as you have no control over what someone could do with that pointer
   C) Storing data by pointer in a container has the unique drawback in that its data cannot be initialized nor modified
   D) If you need to store data that is shared by multiple parts of your program, then this shared data must be stored in a container by value
   E) None of the above

8. What is the time complexity of inserting an element at the *back* of an array, assuming that the array has enough capacity to support the new element without reallocation?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

9. What is the time complexity of inserting an element at the *front* of an array, assuming that the array has enough capacity to support the new element without reallocation?
   A) $\Theta(1)$
   B) $\Theta(\log(n))$
   C) $\Theta(n)$
   D) $\Theta(n\log(n))$
   E) $\Theta(n^2)$

10. Which of the following statements is **TRUE** regarding sequential and random access?
    I. Sequential access processes elements by starting at the beginning of a container, whereas random access directly finds an element without sequentially accessing all previous elements.
    II. There exist container interfaces that do not support random access at all.
    III. Random access is possible in arrays because their elements are homogeneous in size and stored contiguously in memory.

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

11. Templated container classes often include some type of "get element" operation that can be used to access an element in the container. Which function header prototype is typically the best choice for a `get_element()` method for a templated container that can hold any type `TYPE`, assuming that you do **NOT** want the container's data to be modified?
    A) `const TYPE& get_element(int);`
    B) `TYPE& get_element(int);`
    C) `TYPE get_element(int);`
    D) `const TYPE* get_element(int);`
    E) None of the above

12. Which of the following statements is **FALSE**?
    A) Having both a `const` and non-`const` implementation of `operator[]` can help the compiler optimize code for speed
    B) If only a non-`const` version of `operator[]` is implemented for type `T`, then `operator[]` would fail on a `const` object of type `T`
    C) If `operator[]` does not modify the object it is called on, then the `const` version will always be run
    D) `operator[]` should return a reference to the element that is accessed
    E) None of the above

13. Consider the following code, which provides a partial definition of an `Array` object.

```
1  struct Array {
2    double* data;
3    size_t length;
4    Array(size_t len = 0) : length{len} {
5      data = len ? new double[len] : nullptr;
6    } // Array()
7
8    double& operator[](size_t idx) {
9      if (idx < length) {
10       return data[idx];
11     } // if
12     throw runtime_error("invalid index");
13   } // operator[]
14
15   const double& operator[](size_t idx) const {
16     if (idx < length) {
17       return data[idx];
18     } // if
19     throw runtime_error("invalid index");
20   } // operator[] const
21 };
22
23 // special operator<< for Array
24 std::ostream& operator<<(std::ostream& os, const Array& a) {
25   for (size_t i = 0; i < a.length; ++i) {
26     os << a[i] << " ";
27   } // for i
28   return os;
29 } // operator<<()
30
31 int main() {
32   Array a{3};
33   a[0] = 5;
34   a[1] = 4;
35   a[2] = 3;
36   for (size_t i = 0; i < std::min(a[0], a[a.length - 1]); ++i) {
37     std::cout << a << " ";
38   } // for i
39 } // main()
```

While running `main()`, how many times is the `const` version of `operator[]` called?
    A) 9
    B) 11
    C) 15
    D) 17
    E) 20

14. Consider the following implementation of an `Array` class's copy constructor:

```
1  template <typename T>
2  class Array {
3    T* data;
4    size_t length;
5  public:
6    Array(const Array arr)
7      : data{new T[arr.length]}, length{arr.length} {
8      for (size_t i = 0; i < length; ++i) {
9        data[i] = arr.data[i];
10     } // for i
11   } // Array()
12 };
```

Does the above copy constructor work as expected? If not, which line(s) of the above code is/are implemented incorrectly?
    A) The copy constructor as written is correctly implemented and works without issue
    B) No, there is an issue on line 6
    C) No, there is an issue on line 7
    D) No, there is an issue on line 8
    E) More than one of (B), (C), and (D)

Chapter 6 Exercise Solutions

1. **The correct answer is (B).** Only statement II is true, since elements in an array are stored contiguously in memory. Statement I is false because arrays are fixed-size and cannot be changed at runtime. Statement III is false because `operator=` (without an explicit overload) will only perform a shallow copy of the array that does not copy each of the individual elements over.

2. **The correct answer is (D).** Using the formula for calculating the index, we get row × num_columns + column = $5 \times 9 + 6 = 45 + 6 = 51$.

3. **The correct answer is (D).** When `j` has a value of `sz - 1`, the program attempts to access `arr[sz]`, which is invalid. To fix this, `j` must be less than `sz - 1` rather than just `sz`.

4. **The correct answer is (A).** Statement (A) is false, since the copy constructor is not the only thing that needs to be overloaded for the container to manage its dynamic memory properly. You will also need to implement the other members of the big three (destructor and overloaded assignment operator).

5. **The correct answer is (C).** To copy an array of n elements, you cannot do better than $\Theta(n)$ since you have to visit each element at least once to copy it. There is no need to do any work beyond $\Theta(n)$ either; as the number of elements you have to copy grows, the number of operations you need will also grow linearly.

6. **The correct answer is (D).** Copy-swap is a technique devised to implement the overloaded assignment operator by leveraging existing implementations of the copy constructor and destructor. With copy swap, the assignment of one entity to another creates a copy of the original entity, swaps this object so that it becomes the object being assigned to, and deallocates the previous object.

7. **The correct answer is (B).** Statement (B) is true, since the user may be able to do anything with the data via the pointer without any knowledge of the container that holds it. Statement (A) is false, since primitive types are easier to copy if they are stored by value (since they are more lightweight than copying a pointer, which is what passing by reference does behind the scenes — see section 1.2.2). Statement (C) is false, since data can be initialized and modified in a container of pointers. Statement (D) is false, since shared data is better stored as pointers or references, so that all the parts of the program can access the same data (as opposed to storing by value, which requires a copy).

8. **The correct answer is (A).** Assuming no reallocation is necessary, the time complexity of inserting an element to the back of an array is constant (since it can be placed at the very back of the array without having to touch any of the other elements).

9. **The correct answer is (C).** Assuming no reallocation is necessary, the time complexity of inserting an element to the front of an array is linear, since you would have to shift all existing elements over by one to open up a spot at the beginning for the new element.

10. **The correct answer is (E).** All of the statements are true. Statement I provides the definition of sequential vs. random access. Statement II is true because certain containers (such as lists) do not support random access at all. Statement III is true because these two conditions allow arrays to identify the memory position of any of its elements using simple pointer arithmetic.

11. **The correct answer is (A).** To return an element from a container that should not be modified, it is best to return a const reference. The const specified indicates the return value should not be modified, and the reference avoids making a copy of elements you want to return in the container.

12. **The correct answer is (C).** The `const` version of `operator[]` is run if the container it is called on is specified as `const`. Even if a call to `operator[]` does not modify a value in the container, the non-`const` version of `operator[]` would still be called if the container itself is not `const`.

13. **The correct answer is (A).** The `const` version of `operator[]` is called if the `Array` it is invoked on is `const`. This happens in the overloaded `operator<<` method, which calls `operator[]` on the array 3 times. Since the overloaded operator is invoked 3 times on line 37 (since `std::min(a[0], a[a.length - 1])` is 3), the `const` version of `operator[]` is called a total of $3 \times 3 = 9$ times.

14. **The correct answer is (B).** The issue here is that the parameter of the copy constructor is passed in by value instead of by reference (i.e., `Array` instead of `Array&`). Passing the array by value involves making a copy, which invokes the copy constructor we are trying to implement, causing an error.