Chapter 6
Arrays and Containers

6.1 Introduction to Data Structures

When designing a computational approach to solving a problem, there are two important considerations that need to be made: the *algorithm* and the *data structures* that are used to implement the algorithm.

An **algorithm** is a well-defined set of instructions that can be followed to solve a problem. In the previous two chapters, we introduced the concept of time complexity and how it can be used to measure the efficiency of an algorithm. However, the performance of an algorithm not only relies on its implementation, but also the data structures it is used on. A **data structure** is defined by a collection of data, the relationships among them, and the functions and operations that can be applied to these data values. Each data structure provides an interface that defines how its data can be accessed and organizes this data in memory in a manner that efficiently supports this access.

A **container** is a type of data structure that can be used to store other objects. A container manages the memory used by the objects it holds. There are several types of containers that we will cover in this class; you do not need to know any of these details just yet, but this provides a preview of what is to come.

- A *sequence container* is a container that allows its elements to be accessed sequentially. Examples of sequence containers include
  - `std::array<>` *(chapter 6)*
  - `std::vector<>` *(chapter 7)*
  - `std::list<>` *(chapter 8)*
  - `std::forward_list<>` *(chapter 8)*
  - `std::deque<>` *(chapter 9)*
- An *associative container* is a container that stores its data in sorted order and allows elements to be quickly searched in $\Theta(\log(n))$ time. Examples of associative containers include
  - `std::map<>` *(chapter 18)*
  - `std::multimap<>` *(chapter 18)*
  - `std::set<>` *(chapter 18)*
  - `std::multiset<>` *(chapter 18)*

- An *unordered associative container* is a container where elements have no defined order. Unordered associative containers are usually implemented using hash tables, and they support fast lookup (i.e., average-case $\Theta(1)$ time). Examples include
  - `std::unordered_map<>` *(chapter 17)*
  - `std::unordered_multimap<>` *(chapter 17)*
  - `std::unordered_set<>` *(chapter 17)*
  - `std::unordered_multiset<>` *(chapter 17)*
- A *container adaptor* is an interface that is adapted for specific purposes, and they usually restrict the functionality of a standard container. Examples of container adapters include
  - `std::queue<>` *(chapter 9)*
  - `std::stack<>` *(chapter 9)*
  - `std::priority_queue<>` *(chapter 10)*

Both data structures and algorithms play an important role in the design and development of any program, and throughout this course, you will learn how to identify the best data structures and algorithms that can be used to approach a given problem. In this chapter, we will begin this process by discussing the most basic of sequential containers: the array.

6.2 Array Fundamentals

※ 6.2.1 *C-Style Arrays*

An **array** is a fixed-size container of objects that are stored contiguously in memory. Each element of an array must have the same type. The following code provides an example of how to declare and use a C-style array:

```
1  int32_t arr[] = {0, 1, 5, 3};
2  arr[2] = 2;
3  int32_t* ptr = arr;
4  ptr = &arr[1];
```

Line 1 initializes an array using an initializer list. In this case, `arr` is declared as an array of size 4 with data values 0, 1, 5, and 3.

Remark: C++11 introduced the concept of *uniform initialization*, which allows initialization to be done with one common syntax. This initialization uses curly braces and can be used to initialize anything from primitive types to larger objects. For example, the expression `int32_t num{281}` initializes the variable `num` to `281`.

There are a few notable types of initialization, of which include default and value initialization. Atomic types in C++ are undefined when *default initialized*. With curly braces, however, these types can be *value initialized* to have an initial value:

Default Initialization:	**Value Initialization:**
`int32_t x;  // x is undefined`	`int32_t x{};  // x is 0`
`int32_t* y;  // y is undefined`	`int32_t* y{};  // y is nullptr`
`int32_t z[5];  // z has undefined data`	`int32_t z[5]{};  // z is init with 0's`

Curly braces cannot be used in narrowing initializations. For example, `int32_t num = 281.57` does a narrowing conversion from `double` to `int32_t` by truncating the decimal, so `num` ends up storing `281`. However, using braces to initialize `num` (i.e., the expression `int32_t num{281.57}`) would cause a compiler error. The same applies for arrays.

Square brackets (`operator[]`) can be used to access or modify any element in the array in $\Theta(1)$ time. This is because `arr[i]` does the same thing as `*(arr + i)` (pointer arithmetic and dereference), which can be done in constant time.

Line 3 of the above code creates a pointer to the array. Notice that, when we assign the value of `arr` to `ptr`, we end up assigning *pointer* to the first element of the array. This is an attribute of C-style arrays: with a few exceptions (such as `sizeof()`), an expression for an array decays into a pointer to the first element of the array upon behaviors such as assignment or usage as a function argument.

Arrays do not provide bounds checking, so error checking is the responsibility of the programmer. This is because error checking adds additional overhead to performance, which may not be desired if speed is important. As a result, the programmer must add in checks to ensure that the program is not accessing memory it should not be accessing.

In the above example, we initialized a *non-dynamic array*, whose size is *fixed* at compile time. If you wanted to insert more elements in the array, you would have to reallocate the data into a bigger array elsewhere in memory, since the original array has a fixed size of 4. We will cover dynamic arrays (which automatically resize to fit the data) later in this chapter.

※ 6.2.2 *The STL Array (std::array)*

The C++ standard template library (STL) provides the `std::array<>` container in the `<array>` library that can be declared as follows:

```
std::array<TYPE, SIZE> arr;
```

For example, the following declaration creates a C++ array of size 4 with the elements 0, 1, 5, and 3:

```
std::array<int32_t, 4> arr = {0, 1, 5, 3};
```

Or, using curly brace initialization:

```
std::array<int32_t, 4> arr{0, 1, 5, 3};
```

The STL `std::array<>` is a wrapper over the standard C-style array, and it provides several additional features that are missing from the C array. First, the `std::array<>` gives you access to useful member functions (such as `.size()`) and iterators (which will be covered in chapter 11). The following table summarizes a few of these member functions (do not worry too much about these for now):

Function	Behavior
`.size()`	Returns the size of the array
`.empty()`	Returns a Boolean specifying whether the array is empty
`.front()`	Returns the first element of the array
`.back()`	Returns the last element of the array
`.begin()`	Returns a random access iterator to the first element in the array
`.end()`	Returns a random access iterator to the position one past the last element in the array
`.cbegin()`	Returns a *constant* random access iterator to the first element in the array
`.cend()`	Returns a *constant* random access iterator to the position one past the last element in the array
`.rbegin()`	Returns a *reverse* iterator to the last element in the array
`.rend()`	Returns a *reverse* iterator to the position one before the first element in the array
`.crbegin()`	Returns a constant reverse iterator to the last element in the array
`.crend()`	Returns a constant reverse iterator to the position one before the first element in the array

The following code illustrates an example of `std::array<>` usage, contrasting how array size is determined between a C-style array and a `std::array<>`. For C-style arrays, the size is determined by taking the size of all the elements in the array and dividing it by the size of a single element. However, the `std::array<>` provides a member variable `.size()` that does this work for you.

```
1  // How to find size of standard C-array
2  int32_t c_arr[] = {0, 1, 5, 3};
3  size_t c_arr_size = sizeof(c_arr) / sizeof(c_arr[0]);  // stores 4
4
5  // How to find size of C++ std::array<>
6  std::array<int32_t, 4> cpp_arr = {0, 1, 5, 3};
7  size_t cpp_arr_size = cpp_arr.size();                  // stores 4
```

Additionally, unlike the standard C-style array, a `std::array<>` can be treated like a fundamental type, supporting features such as direct assignment or copy using `operator=`, as well as value semantics (i.e., they can be passed into or returned from a function by value).

In general, if you ever find the need to use a fixed-sized array in a C++ program, you should choose a `std::array<>` over a C-style array due to its cleaner interface and additional features. However, we will not be delving too deeply into the usage of this data container here, since a `std::vector<>` is often a better choice in most situations (which is another container type that will be covered in the next chapter).

※ 6.2.3 *Common Off-By-One Errors*

When looping through arrays, it is important to include the correct bounds for what you want to do. Recall that bounds checking is not done automatically and is the responsibility of the programmer! Since arrays in C++ use 0-indexing, the following loop would go off the end of the array. This is because the last element of the array has index `SIZE - 1`, where `SIZE` is the size of the array. To fix this issue, the <= in the loop definition should be replaced with a <.

```
// INCORRECT
for (size_t i = 0; i <= SIZE; ++i) {
  arr[i] = i;
}
```

```
// CORRECT
for (size_t i = 0; i < SIZE; ++i) {
  arr[i] = i;
}
```

If you are using both the element at the current index (index `i`) and the next index (index `i + 1`) in the body of your loop, you will need to stop the loop before it reaches index `SIZE - 1`. This is because `i + 1 = SIZE` if `i` reaches `SIZE - 1`, and `SIZE` is not a valid index!

```
// INCORRECT
for (size_t i = 0; i < SIZE; ++i) {
  arr[i] = arr[i + 1];
}
```

```
// CORRECT
for (size_t i = 0; i < SIZE - 1; ++i) {
  arr[i] = arr[i + 1];
}
```

If you are accessing index `i - 1` in the body of your loop, you may need an additional iteration if you want to visit every element in the array. This can be done using <= instead of < when defining the bounds of the loop. In addition, you should start iterating at index 1, since `i - 1` would not be a valid index if `i` were 0.

```
// INCORRECT
for (size_t i = 1; i < SIZE; ++i) {
  arr[i - 1] = i;
}
```

```
// CORRECT
for (size_t i = 1; i <= SIZE; ++i) {
  arr[i - 1] = i;
}
```

6.3 Storing Multidimensional Data in Arrays

The following initializes a fixed one-dimensional (1-D) array of size 12:

```
int32_t arr_1D[12] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

In memory, the array is stored like this, since elements are contiguous in memory (the address numbers were arbitrary chosen, and they are separated by 4 because that is the size of a 32-bit integer):

0	1	2	3	4	5	6	7	8	9	10	11
0x100	0x104	0x108	0x10c	0x110	0x114	0x118	0x11c	0x120	0x124	0x128	0x12c

Now, suppose we want to represent two-dimensional data in array. To do so, we can declare a fixed-size 2-D array, using the following syntax:

```
int32_t arr_2D[num_rows][num_cols];
```

For example, the following initializes a 2-D array with 3 rows and 4 columns:

```
int32_t arr_2D[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
```

The first row of the `arr_2D` matrix would have the elements 0, 1, 2, and 3; the second row would have the elements 4, 5, 6 and 7; the third and final row would have the elements 8, 9, 10, and 11.

To access an element in a 2-D array, use the `[]` operator twice, where the first `[]` contains the row index and the second `[]` contains the column index. For example, `arr_2D[1][2]` retrieves the element in row 1, column 2 of the matrix (which is `6`).

How is this 2-D array stored in memory? Because memory chips are 1-dimensional, the 2-D array is laid out in memory *as if it were a 1-D array* (in row-major order). That is, the 2-D array is stored like this in memory:

0	1	2	3	4	5	6	7	8	9	10	11
0x100	0x104	0x108	0x10c	0x110	0x114	0x118	0x11c	0x120	0x124	0x128	0x12c

When you try to access an element of a 2-D array, the compiler actually converts the 2-D row and column indices into the corresponding index of a 1-D array. The conversion formula is:

$$\text{1-D\_index} = \text{2-D\_row\_index} \times \text{num\_columns} + \text{2-D\_column\_index}$$

For example, when `arr_2D[1][2]` is called, the compiler converts the 2-D indices into the index $1 \times 4 + 2 = 6$. It then accesses index 6 of the underlying 1-D array that is stored in memory to get the correct value.

Additionally, if you wanted to treat a one-dimensional array as a two-dimensional one, you can use the following equations to convert a 1-D index into its corresponding 2-D ones:

$$\text{2-D\_row\_index} = \text{1-D\_index} \,/\, \text{num\_columns}$$
$$\text{2-D\_column\_index} = \text{1-D\_index} \,\%\, \text{num\_columns}$$

Note: the % symbol represents the modulo operation, which finds the remainder when one number is divided by another. For example, 11 % 5 = 1, because dividing 11 by 5 yields a remainder of 1.

Example 6.1 A 2-D array is initialized with 14 rows and 11 columns. Suppose that the element at row 7, column 9 is accessed. What is the index of this element in the underlying 1-D array in memory?

To solve this, we will use the conversion formula from a 2-D index to a 1-D index:

$$\text{1-D\_index} = \text{2-D\_row\_index} \times \text{num\_columns} + \text{2-D\_column\_index} = 7 \times 11 + 9 = 77 + 9 = \mathbf{86}$$

Example 6.2 A 1-D array is initialized with a size of 126. This 1-D array stores two-dimensional data with 7 rows and 18 columns. What is the row and column index of the element at index 59 of the 1-D array?

To solve this, we will use the conversion formula from a 1-D index to a 2-D index:

$$\text{2-D\_row\_index} = \text{1-D\_index} \,/\, \text{num\_columns} = 59/18 = \mathbf{3}$$
$$\text{2-D\_column\_index} = \text{1-D\_index} \,\%\, \text{num\_columns} = 59 \,\%\, 18 = \mathbf{5}$$

The element at index 59 corresponds with the element at row 3, column 5 of the 2-D array. Note that integer division is used to calculate the row index, since the decimal is truncated.

In summary, regardless of the dimension of the data, all arrays look like 1-D arrays in memory. As a result, any *n*-dimensional array can be converted to a one-dimensional array using arithmetic operations.

6.4 Heap-Allocated Arrays

In the previous examples, we declared an array on the stack. Stack-allocated arrays are automatically deallocated once they go out of scope. However, there is also a limitation to stack-allocated arrays — in order to declare an array on the stack, its size must be a constant value that is known at compile time. This is an issue if you do not know what the size of an array should be until you actually receive the data during runtime. Consider the following code, which takes in a size value from the user and attempts to initialize an array with that size:

```
1  size_t size;
2  std::cin >> size;
3  int32_t arr[size];
```

The problem with this code is that `size` is a variable whose value is determined at runtime, not at compile time. This is not allowed per the C++ standard.[1] Although some compilers support this behavior, you should not be declaring an array on the stack if you do not know its expected size at compile time, as doing so could cause issues (for example, if the user inputs an absurdly large value for `size`, your program would exceed the amount of stack space available and immediately crash).

Instead, it is preferable to declare variable-size arrays on the heap. The heap is a portion of your computer's memory that is not managed automatically (compared to a stack), so you are responsible for deallocating any memory that you allocate on the heap. To allocate memory on the heap, you should use the keyword `new`; to deallocate memory, you should use the keyword `delete`. The following code takes in a size value at runtime and initializes an array with that size on the heap:

```
1  size_t size;
2  std::cin >> size;
3  int32_t* arr = new int32_t[size];
4  /* do stuff with array */
5  delete[] arr;
```

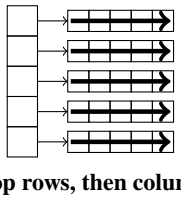
After you are done with an array that is declared on the heap using `new`, you must deallocate it yourself. Failure to do so would lead to a memory leak, where memory that is no longer accessible still remains in the heap. To delete an array that is allocated on the heap, you must follow the `delete` keyword with square brackets (i.e., `delete[]`, as shown on line 5 of the code above).

You can declare a 2-D array with variable size in a similar fashion. Since the 2-D array is dynamically allocated on the heap during runtime rather than during compile time, we will need to first allocate the outer dimension (rows) and then the inner dimension (columns):

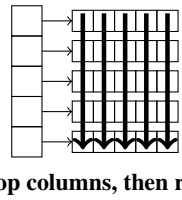
```
1  size_t rows, cols;
2  std::cout << "Enter the number of rows: " << std::endl;
3  std::cin >> rows;
4  std::cout << "Enter the number of columns: " << std::endl;
5  std::cin >> cols;
6  int32_t** arr = new int32_t*[rows];
7  for (size_t r = 0; r < rows; ++r) {
8    arr[r] = new int32_t[cols];
9  } // for r
10 int32_t val = 0;
11 for (size_t r = 0; r < rows; ++r) {
12   for (size_t c = 0; c < cols; ++c) {
13     arr[r][c] = val++;
14   }  // for c
15 } // for r
16 /* do stuff with the array */
17 for (size_t r = 0; r < rows; ++r) {
18   delete[] arr[r];
19 } // for r
20 delete[] arr;
```

Line 6 of the code above initializes an array of pointers to other arrays, and each row is individually initialized in the `for` loop on lines 7-8. This initialization is shown on the top of the next page.

Lines 11-13 initialize the individual elements in the 2-D array itself. When looping through a multidimensional array, it is faster to loop through the outermost dimension (rows) in the outermost loop. This is due to something known as caching (which is covered in EECS 370 and not this class): essentially, it is much more efficient to sequentially access items that are closer together in memory than items that are farther apart. This idea is shown in the figure below.

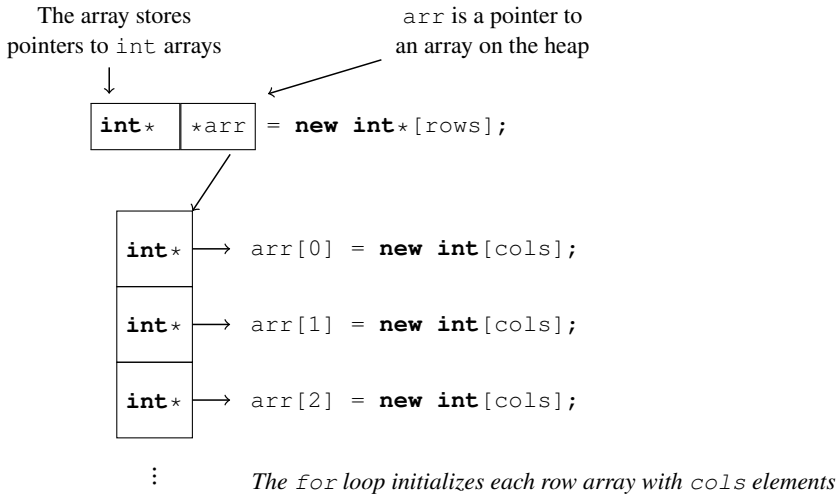


Loop rows, then columns
Faster since memory accesses are contiguous



Loop columns, then rows
Slower since memory accesses are not contiguous

[1] Variable length stack arrays were permitted back in C99, but they were discontinued in later versions of the language. If you want to specify an array on the stack using a variable (e.g., `int arr[size]`), you would have to explicitly initialize the variable and declare it using the `constexpr` keyword, which specifies that is possible to evaluate the value of the variable at compile time.



The `for` loop initializes each row array with `cols` elements

Since we called `new` to initialize this 2-D array, we will have to deallocate it after we are done using it. The `delete` operations must be called in the opposite order in which we called `new`. Since we initialized the array of row pointers (`arr`) before the individual row arrays (`arr[0]`, `arr[1]`, ...), we have to iterate through `arr` and delete all of the row arrays before we can delete `arr`. This is shown on lines 17-20.

To review, we have discussed two different ways to initialize an array. One way is to declare the array on the stack. Stack-allocated arrays are automatically deallocated when they go out of scope, and accessing an element only requires one access to memory regardless of dimension (since the compiler does math to obtain the correct memory location of an object). However, arrays on the stack are limited in size and require the programmer to know the dimensions of the array during compilation. As a result, stack-allocated arrays are not very versatile.

To address the issue of variable-sized input, a dynamically allocated pointer-based array can be initialized instead. There are a few caveats to watch for when an array is declared on the heap. With a heap-allocated array, an element would require more than one memory access for larger dimensions: in the 2-D array above, we would first need to find the pointer to the correct row array, and then the address of the element we want within the row array itself. Furthermore, arrays that are allocated on the heap must be deallocated by the programmer before the program ends. Nonetheless, dynamically allocated arrays are more flexible and should be used if the array size cannot be determined until runtime.

Working with heap-allocated arrays can be a messy process at times. Luckily, in your projects, you will not be using C-style arrays. Instead, you will be using the C++ `std::vector<>`, which abstracts away these details and provides a cleaner interface for storing data in an array-like format. A vector gives you all the functionality of a dynamically allocated array, but it handles all the dynamic memory allocation and deallocation for you. In addition, it can also grow in size based on the amount of data it is required to store, unlike a fixed-size array. Vectors will be discussed in more detail in the next chapter.

Remark: In general, when dynamic memory is allocated, whoever called `new` is also responsible for calling `delete`. When working with dynamic memory, make sure to add in a `delete` whenever you call `new`, before you do any other work! This ensures that you don't forget to deallocate memory, which can lead to memory leaks that can be difficult to track down.

In addition, after deleting a pointer to dynamic memory, it is good practice to set it to `nullptr`:

```
1   delete ptr;
2   ptr = nullptr;
```

This acts as a safeguard against unintentional uses of a deleted value, and it also protects you from double deletes (i.e., `delete nullptr` is safe and does nothing, but calling `delete ptr` twice could crash your program).

Note: To avoid all the hassle of `new` and `delete`, you can use smart pointers defined in the `<memory>` library to manage your memory for you (`std::unique_ptr<>`, `std::shared_ptr<>`, `std::weak_ptr<>`). Smart pointers are essentially wrappers over normal pointers that provide enhanced functionality and automatic memory management. However, smart pointers are *not* allowed in this class (although they are still good to know, especially for upper-level classes - see chapter 27, but only after the class is done).

6.5 Copying with Pointers

Suppose you are given two arrays, and you want to copy the data from one array (the source array) to the other (the destination array). Assume that the destination array is large enough to fit all the data in the source array. You are given three pointers:

```
1   int32_t* first;   // pointer to the first element of the source array
2   int32_t* last;    // pointer one past the last element of the source array
3   int32_t* dest;    // pointer to the first position of the destination array
```

To copy over elements from the first array into the second, you can use the following process:

1. Dereference the value of `first` and assign it to the value of `dest` (i.e., `*dest = *first`).
2. Increment `first` to move the next element of the source array (i.e., `first++`).
3. Increment `dest` to move to the next position of the destination array (i.e., `dest++`).
4. Repeat for all the elements in the array, stopping the loop when `first` reaches `last`.

This process can be converted into either of the following loops (both a `while` loop and a `for` loop are shown):

```
while (first != last) {
  *dest++ = *first++;
}
```

```
for (; first != last; ++first, ++dest) {
  *dest = *first;
}
```

For the `while` loop implementation, because the postfix operator is used (`++` after the variable name), the incrementation is done *after* the dereference and assignment. Essentially, the `while` loop runs until `first == last`, each time copying the value pointed to by `first` into the position pointed to by `dest`. The loop terminates once `first` reaches `last`, which points one past the end of the source array. Note that `first != last` is used instead of `first < last`, since only `first != last` is guaranteed to work on all types of containers.

This method can be extended across different containers using iterators, which will be covered in chapter 11. Using pointers and iterators to copy elements between containers is preferable to using indexing, since they are more generic and allow the same code to be used for multiple container types, even ones that do not support indexing.

In addition, there exists a function that can be used to copy a block of memory from one location to another. This function is `memcpy()`, and there is a good chance you will have to use it in a later class, even if you do not use it during this class. To use this function, you have to include the `<cstring>` library.

`void* memcpy(void* dest, const void* source, size_t num);`

Copies `num` bytes from the object pointed to by `source` to the object pointed to by `dest`. The underlying type of the objects pointed to by `source` and `dest` are irrelevant, as the data is copied over as raw bytes. If the objects pointed to by `source` and `dest` overlap, or if either point to an invalid address or `nullptr`, the behavior of this function is undefined. (A `void*` is a generic pointer that has no type associated with it, and can be used to reference an object of any data type.)

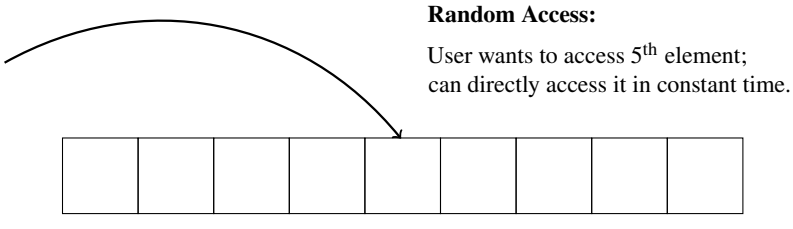
An example of `memcpy()` is shown below:

```
int32_t src[] = {1, 2, 3, 4, 5};
int32_t dst[5];
std::memcpy(dst, src, sizeof(src));
for (size_t i = 0; i < 5; ++i) {
  std::cout << dst[i] << " ";        // prints 1 2 3 4 5
} // for i
```

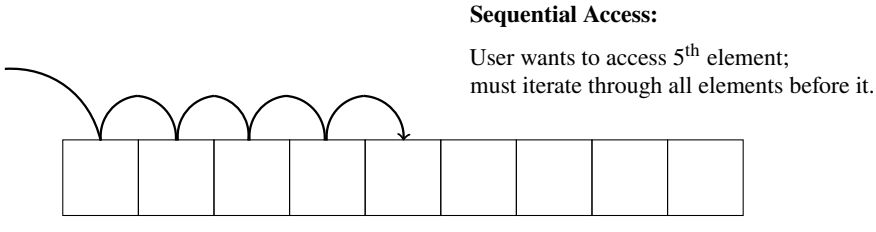
The `sizeof()` function returns the number of bytes that a data type or object occupies in memory. Here, `sizeof(src)` calculates the total size of the `src` array so that `memcpy()` can copy over the correct number of bytes to the `dst` array.

6.6 Sequential and Random Access

As mentioned in the first section of this chapter, a **container** is an object that stores a collection of data types. Different containers can have different properties. In this section, we will discuss two different access methods for iteration that may be supported by a container of data: random access and sequential access. With **random access**, you are able to access any arbitrary element in a sequence of data as efficiently as any other element at any point in time, regardless of the size of the sequence. On the other hand, with **sequential access**, the elements in a sequence of data are visited in a predetermined order, where each element is accessed directly after its predecessor. An array is an example of a container that supports random access, since any item in an array can be directly accessed in $\Theta(1)$ time using `operator[]`.



On the other hand, a container does *not* support efficient random access if its elements must be accessed in a predeteremined sequence. Such containers only support efficient sequential access. For instance, if a user requests to access an element in a container that only supports sequential access, the entire container must be read from the beginning to the element that is desired; there is no way to calculate the memory location of a random element directly. As such, there is no guarantee that an element in such a container can accessed as easily as any other. An example of a sequential container is a linked list (covered in chapter 8). Without any supporting data structures, accessing an arbitrary n^{th} element of a linked list would require you to iterate through the first n elements in the container beforehand.

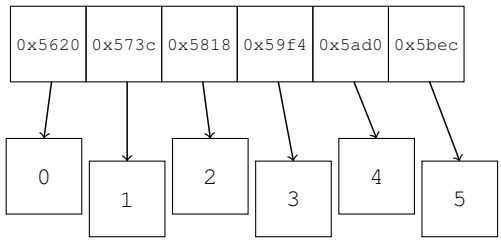


6.7 Data Storage and Retrieval

Suppose you wanted to create a container to store a data type. How would you store the data? You have two options: you could store the *values* of the data elements themselves in the container, or you could store *pointers* to your data values in your container.

Storing by Value:

Storing by Pointer:



Containers that store data by value have full ownership over the memory used to store their data, and they are responsible for managing their own memory. If you want to insert or delete elements from a container that stores data by value, you should ask the container to do it for you via some sort of internal operation (such as calling an `insert()` member function). Containers that store values are the most common, but they can be expensive to copy if the objects they store are large.

On the other hand, you could store pointers to your data in the container instead. If you do this, the container no longer has direct ownership over the data, and it is not responsible for allocating or deallocating the memory for the data values themselves. Instead, the container is only responsible for the *pointers* to the data. This can be dangerous, since the container no longer provides direct protection: the data in the container can be modified elsewhere without permission from the container itself. A container of pointers, for example, can easily be invalidated if something else deletes what its pointers are pointing to. That being said, containers of pointers can be useful in certain cases, such as with containers of C-strings (`char*`) and shared data (e.g., having one master container that stores the data and smaller containers that access this data in different ways using pointers; this ensures that your data only exists in memory once).

There are also several considerations that need to made when getting values out of a container. Suppose you wanted to retrieve an element from a container (e.g., by calling `operator[]`). How should the element be returned to the user? There are three ways that this can be done.

First, the container could return a *copy* of the element that the user wants. For example, if the user requests the element at the fifth index of a container, the container could return a separate copy of this element to the user. However, this method is inefficient: if the objects in a container were large, accessing elements in the container would be slow since a copy would have to be made every time the user requests an element.

An alternative option would be to return a *pointer* to the element that is requested. Since an address to an object is returned, no copies are needed. However, this option is unsafe, as it gives the user the ability to do things that they shouldn't be allowed to do! With a pointer to the contents of a container, a user could modify anything they wanted, iterate off the end of the container and/or access memory they shouldn't be able to access, or even delete the contents of the data entirely! As a result, returning pointers to private data generally isn't ideal.

The third option would be to return a *reference* to the element that was requested. This option is typically the best choice. Because references don't require copies, returning by reference is faster than returning by value. In addition, references are safer than pointers because they cannot be modified — a user cannot use a reference to maliciously access other memory without permission. Furthermore, if you want to prevent the user from modifying the contents of the data you give them, you can return a `const` reference to the requested element instead of a normal, non-`const` reference.

What to Store in a Container

	Value	Pointer
Example	`T data;`	`T* data;`
Data Ownership	Container	Container or other object
Drawbacks	Large objects can take time to copy	Unsafe, can be modified elsewhere or invalidated
Usage	Most common	Used for `char*`, shared data

What to Retrieve from a Container

	Value	Pointer, Const Pointer	Reference, Const Reference
E.g.,	`T get_elt(size_t idx);`	`T* get_elt(size_t idx);`	`T& get_elt(size_t idx);`
Notes	Costly for copying large objects	Unsafe, can be used to access prohibited memory	Usually a good choice; safer than pointers and faster than copies

6.8 Implementing an Array-Based Container

※ 6.8.1 *A Blueprint of the Custom Array-Based Container*

In this section, we will combine the concepts of the previous sections and begin implementing an array-based container that can be used to manage a dynamically allocated array. This container will not only have the functionality of a standard C-style array — it will also have features that abstract away memory allocation work from the user. Recall that standard C-style arrays have fixed size; for instance, if you wanted to add a fifth element to an array that can only store at most four elements, you would have to reallocate a new, larger array in memory to be able to add that element. It is also cumbersome to copy or assign standard C-style arrays, since doing so would require deep copies of all the elements to be made. Our goal is to implement a container that can handle this work for the user, so that users of the container will not have to worry about memory allocation and deallocation when trying to insert an element or copy an array.

Since our container is a custom object, we will define it using a C++ `class`. Here is a blueprint for this custom `Array` object that supports limited functionality. We will be adding to this as we continue through this section.

```
1  template <typename T>
2  class Array {
3    T* data;          // Array data
4    size_t length;    // Array size
5  public:
6    Array(size_t len = 0) : length{len} {
7      data = (len ? new T[len] : nullptr);
8    } // Array()
9
10   size_t size() const {
11     return length;
12   } // size()
13 };
```

Let's take a look at what this `Array` object currently does. There are currently two private member variables that are managed by the `Array`. The `data` variable on line 3 points to a heap-allocated array that contains the `Array` container's data. The `length` variable on line 4 stores the number of elements in the `Array`. On line 6, we have the `Array` constructor; the user can initialize the `Array` to have any size they want by passing in a value for `len`. For example, if the following code were run, the constructor would initialize the value of `length` to `5` and set `data` to a dynamically allocated array of size 5 (`data = new T[len]`).

```
Array<int32_t> a{5};            // initialize Array called 'a' with length 5
```

If the user creates an `Array` without specifying `len`, then the default argument value is used (`len = 0`).

```
Array<int32_t> b;               // len not specified, so b has a length 0
```

If `len` has a value of 0, then `data` is set to `nullptr` on line 7 (`data = nullptr`), and the value of `length` is also set to 0 on line 6. The assignment expression on line 7 (using the ternary operator) ensures that if `len` is a non-zero value, it is set to `new T[len]`; otherwise, it is set to `nullptr`.

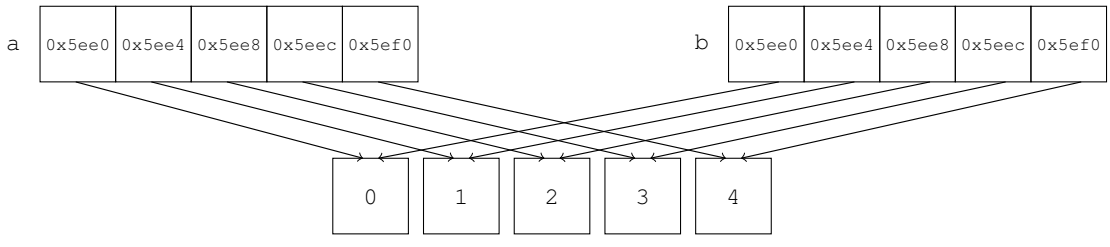
Lastly, we have the `size()` method on line 9, which returns the length of the `Array`. Since this member function does not modify any member variables of the `Array` class, it is declared as `const`. For example, the following code would print out `5`:

```
Array<int32_t> a{5};                    // initialize Array called 'a' with length 5
std::cout << a.size() << std::endl;     // prints 5
```

These features are what we have so far with our `Array` implementation. However, there is a problem! Suppose we ran the following code:

```
Array<int32_t> a{5};                   // initialize Array called 'a' with length 5
Array<int32_t> b = a;                  // initialize 'b' with contents of 'a'
```

To initialize `b`, all the member variables of `a` are copied over one by one. However, since the `data` member variable is an array of pointers, the pointers are copied over and not the data objects themselves! Thus, both `a` and `b` end up sharing the same data.



When the contents of `a` is modified, the contents of `b` would also change, since both containers point to the same memory. In addition, if the memory of `a` is deallocated, so would the memory of `b`! This is not intended behavior. If you try to access an element of `b` after `a` is deallocated, you would be accessing invalid memory; on the other hand, if you try to deallocate `b` after `a` is deallocated, you would end up with a double delete that could crash your program! To prevent these errors from happening, we will need to implement a custom assignment operator that copies the data objects themselves rather than just the pointers.

※ 6.8.2 *The Big Three*

In general, if you are designing a class that manages dynamic memory as its data, you should also define the following "Big Three" as member functions of your class to handle dynamic memory:

1. Destructor (e.g., `~Array()`)
2. Copy constructor (e.g., `Array(const Array& other)`)
3. Overloaded assignment operator (e.g., `Array& operator=(const Array& rhs)`)

Remark: You may instead see this rule of the "Big Three" as the rule of the "Big Five." There are two additional methods that are often included, but you do not need to implement them for this course:

4. Move constructor (e.g., `Array(Array&& other)`)
5. Overloaded move assignment operator (e.g., `Array& operator=(Array&& rhs)`)

As a brief introduction, the move constructor and assignment operator can be used to transfer ownership of data from one object to another in certain situations (rather than make a copy of the data object, assign the copy to the new object, and discard the original object). The double ampersand (`&&`) indicates that an object is an *rvalue reference*, which is a temporary, unnamed object that is created during the execution of a C++ program. A useful mnemonic is to think of the "r" in "rvalue" as representing "right-hand side," since rvalues can never go on the left-hand side of an assignment — this is because rvalues are temporary and often cannot last beyond the statement they are created in.

```
int32_t a = 10;       // '10' is an rvalue
int32_t b = 20;       // '20' is an rvalue
int32_t c = a + b;    // 'a + b' is an rvalue
int32_t d = a;        // 'a' NOT an rvalue, is a named user-defined variable
```

The move constructor and move assignment operator accept an rvalue as an argument (e.g, `&&other`) and thus can only be invoked if an rvalue is present. This is because it is harmless to transfer an rvalue's ownership since they are temporary objects that would otherwise be destructed. If an instance of an object is ever assigned or copy constructed from an rvalue, and a move constructor and move assignment operator are defined, then that custom object will claim ownership of the rvalue's data during construction (instead of making a copy of the rvalue and assigning the custom object to the copy). This can speed things up if an object is large and can take time to copy.

If no rvalue is present, then a move constructor or move assignment operator will NOT be automatically invoked. This is because non-rvalues are not temporary and may be used later on, so it is not safe to transfer ownership of their data. If you want to move such a variable, you must explicitly call the `std::move()` function to show that you intend to transfer the ownership of that variable. For example, consider the following code:

```
Array<int32_t> a{5};
Array<int32_t> b = std::move(a);
```

Here, the `Array` object `a` essentially transfers ownership of its data to the `Array` object `b`. This speeds up the assignment process since a separate copy of `a` does not need to be made. However, after the move is complete, `a` ends up in an unspecified state and should no longer be used until it is assigned a new value. *You do not need to know about move semantics for 281, but we will go over it more in section 6.9.*

Let's build upon our implementation of the `Array` class to include these methods.

```
1  template <typename T>
2  class Array {
3    T* data;          // Array data
4    size_t length;    // Array size
5  public:
6    // Constructor
7    Array(size_t len = 0) : length{len} {
8      data = (len ? new T[len] : nullptr);
9    } // Array()
10
11   // Destructor
12   ~Array();
13
14   // Copy Constructor
15   Array(const Array& other);
16
17   // Overloaded Assignment Operator
18   Array& operator=(const Array& rhs);
19
20   // Move Constructor
21   Array(Array&& other) = delete;
22
23   // Overloaded Move Assignment Operator
24   Array& operator=(Array&& rhs) = delete;
25
26   size_t size() const {
27     return length;
28   } // size()
29 };
```

We will not be implementing the move constructor and the overloaded move assignment operator for now, so we have disabled them by setting their member functions equal to the `delete` keyword on lines 21 and 24. In general, setting any member function to `delete` suppresses the function and disallows it from being invoked.

※ 6.8.3 *Implementing the Destructor*

Let's start by implementing the destructor, which deallocates all dynamic memory owned by the `Array` container. This is done by calling `delete[]` on the heap-allocated `data` array. Since both lines of code take 1 step (assuming the type `T` does not have a destructor on its own), the overall time complexity of the `Array` destructor is $\Theta(1)$.

```
1  ~Array() {
2    delete[] data;   // delete dynamic memory owned by container
3    data = nullptr;  // safeguard against usage/double delete
4  } // ~Array()
```

※ 6.8.4 *Implementing the Copy Constructor*

The copy constructor is also fairly straightforward. A copy constructor has no return value, has the same name as the class it is defined in, and accepts a *reference* to the object it wants to copy from (the reference is important: without it, a copy of the object would have to be made before the copy constructor can even run, which is not allowed). In the case of our `Array` object, we want all the elements in the `data` array to be copied over, and not just the `data` pointer. To do this, the body of the copy constructor must explicitly copy over each data element one by one:

```
1  Array(const Array& other) {
2    data = new T[other.length];
3    length = other.length;
4    // make copies of all the elements in the data array
5    for (size_t i = 0; i < length; ++i) {
6      data[i] = other.data[i];
7    } // for i
8  } // Array()
```

We can simplify this code by initializing `data` and `length` in a member-initializer list instead of the function body:

```
1  Array(const Array& other)
2    : data{new T[other.length]}, length{other.length} {
3    // make copies of all the elements in the data array
4    for (size_t i = 0; i < length; ++i) {
5      data[i] = other.data[i];
6    } // for i
7  } // Array()
```

Since every element in the `Array` must be visited when running the copy constructor, the time complexity of the copy constructor is $\Theta(n)$, where n represents the number of elements in the `Array`.

※ 6.8.5 *Implementing the Overloaded Assignment Operator and the Copy-Swap Method*

The overloaded assignment operator is where things get a little bit interesting. For the `Array` class, the goal of an overloaded assignment operator would be copy all the data elements from the source `Array` to the destination `Array`. The following implementation does just that:

```
1  Array& operator=(const Array& rhs) {
2    // delete current data
3    delete[] data;
4    // set length and data pointer
5    length = rhs.length;
6    data = new T[length];
7    // copy contents of rhs over
8    for (size_t i = 0; i < length; ++i) {
9      data[i] = rhs.data[i];
10   } // for i
11   return *this;
12 } // operator=()
```

However, this implementation would fail if someone tried to assign an `Array` object to itself (e.g. `a = a`). To fix this, we must add a check to make sure `rhs` is not the current `Array` that we want to assign to (by checking whether the memory addresses are the same on line 3 below):

```
1  Array& operator=(const Array& rhs) {
2    // don't do anything if self-assigning
3    if (this == &rhs) {
4      return *this;
5    } // if
6    // delete current data
7    delete[] data;
8    // set length and data pointer
9    length = rhs.length;
10   data = new T[length];
11   // copy contents of rhs over
12   for (size_t i = 0; i < length; ++i) {
13     data[i] = rhs.data[i];
14   } // for i
15   return *this;
16 } // operator=()
```

This version of the overloaded assignment operator is better and works for most cases. However, it is not the most ideal way to do an assignment. First, we need to do an additional check to see if `this == &rhs` every time we try to use the operator, despite the fact that self-assignment nearly never happens (if the programmer knows what they are doing). Second, if the program somehow fails to allocate memory on line 10, the program would throw an exception and terminate the operation; since we deallocated the array on line 7, all the original data is lost and the `Array` data would point to deleted memory (the `length` member variable would be messed up as well). If this happens, our `Array` would be corrupted! We do not want this — if the assignment somehow fails, we still want to retain the state of our original data.

A better method for implementing the assignment operator is known as the **copy-swap method**. The copy-swap method creates a *temporary copy* of the new `Array` and swaps the contents of the old `Array` with the contents of the new `Array`. The old contents are then automatically deallocated when they go out of scope. This fixes the two problems with the initial approach: no self-assignment check is necessary (which makes our code cleaner), and the original data is retained until an assignment is successfully made. If assignment fails and an exception is thrown, the failed operation does not have any side effects on the existing data (this safety is known as a *strong exception guarantee*). To implement copy-swap, we will first implement a `swap` function in our `Array` class that allows us swap the contents of two `Array` objects:

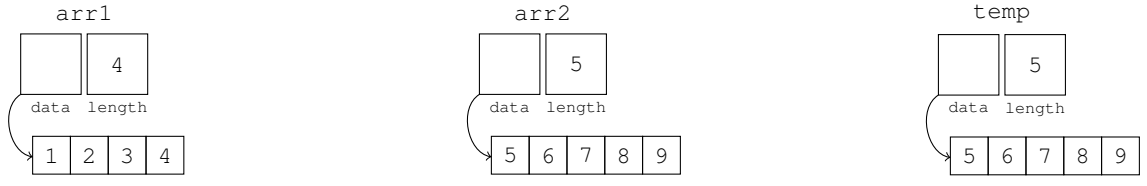
```
1  void swap_array(Array& other) {
2    std::swap(data, other.data);
3    std::swap(length, other.length);
4  } // swap_array()
```

We will then use this method in our overloaded assignment operator function:

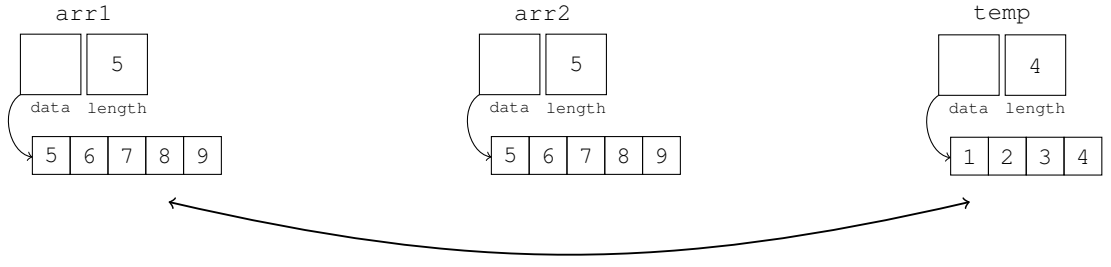
```
1  Array& operator=(const Array& rhs) {
2    // create temporary Array that stores the contents of the new Array
3    // a deep copy is made since this uses the copy constructor
4    Array temp{rhs};
5    // swap the original data with temp's data
6    swap_array(temp);
7    // the old data is now stored in temp, and the
8    // updated data is now stored in the current Array (this)
9    return *this;
10 } // operator=()
11 // temp goes out of scope, so the old data is cleaned up by destructor
```

A visualization of the copy-swap process is shown below, depicting what happens when `arr1 = arr2` is executed:

On line 4 of the function body of `operator=()`, we make a deep copy of `arr2` and assign it to a local variable with the name `temp`.

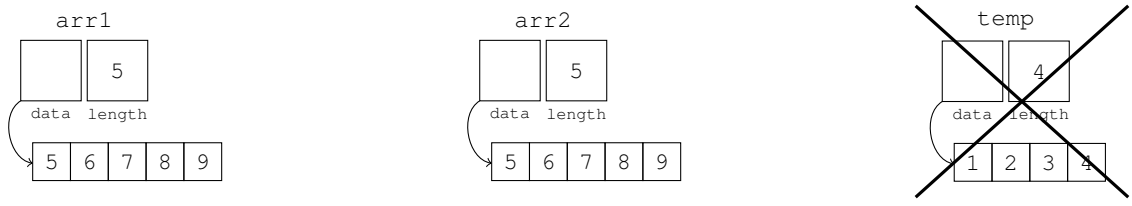


On line 6, the contents of `arr1` (the current `Array` that `operator=` is being invoked on) and `temp` are swapped.



At this point, `arr1` contains the contents of `arr2`, which is what we wanted from the assignment. If someone tried to self-assign an object to itself (`arr1 = arr1`), the copy-swap approach would just swap the `Array` with a copy of its own data, so there is no need to check this condition every time an assignment is made. In addition, if the program had failed to allocate new memory, it would have thrown an exception on line 4, before the original `Array` could be modified on line 6. This ensures that our implementation of `operator=` does not corrupt our data if there was not enough memory to make a copy.

On line 9, we return `*this`, which is a reference to the `Array` that we just assigned to (in this case, `arr1`). After line 10, the function goes out of scope, and all local variables are automatically deallocated. In this case, `temp` (which contains the old data of `arr1`) is automatically cleaned up by the `Array` destructor.



Since the assignment operator does a deep copy, its time complexity is also $\Theta(n)$, where n is the size of `rhs`.

※ 6.8.6 *Implementing the Subscript Operator*

Now that we have implemented the Big Three, we will implement `operator[]` so that our `Array` supports indexing. As covered previously, it is generally ideal to return references to elements that are requested, since they are faster than copies but safer than pointers. Thus, `operator[]` will be implemented to return a reference to the requested data element.

The implementation of `operator[]` below is fairly straightforward. It checks to see if the provided index is valid, and then indexes the correct position of the `Array` object's underlying `data` array. If the index is not valid, the operation throws a `std::runtime_error` exception (since we do not want to return a reference to something the user can unintentionally modify).

```
1  T& operator[](size_t idx) {
2    if (idx < length) {
3      return data[idx];
4    } // if
5    throw std::runtime_error{"Invalid index provided to operator[]"};
6  } // operator[]()
```

Even with this code, we are not done; there is still an issue we need to address. However, this issue does not arise from the implementation, since it handles array indexing correctly. For instance, the output for the following code would be `"0 1 2 3 4"`.

```
1  Array<int> a{5};  // init Array of size 5
2  for (size_t i = 0; i < a.size(); ++i) {
3    a[i] = i;
4    std::cout << a[i] << " ";
5  } // for i
```

The issue arises when we try to run something like this:

```
6  const Array<int> b = a;
7  for (size_t i = 0; i < b.size(); ++i) {
8    std::cout << b[i] << " ";
9  } // for i
```

If you try to compile this, you would get the following error:

```
error:  passing 'const Array' as 'this' argument discards qualifiers [-fpermissive]
```

Why did this happen? Recall from chapter 1 that a `const` object *cannot* call a non-`const` member function! Since `b` is defined as a `const Array`, it is prohibited from calling `operator[]` since its definition is non-`const`, which is why the indexing on line 8 fails. For this to work, we also need to implement a `const` version of `operator[]`:

```
1  const T& operator[](size_t idx) const {
2    if (idx < length) {
3      return data[idx];
4    } // if
5    throw std::runtime_error{"Invalid index provided to operator[]"};
6  } // operator[]()
```

Both versions of `operator[]` need to be defined for indexing to be supported on `const` and non-`const` `Array` objects. The non-`const` definition of `operator[]` would be invoked for a non-`const` `Array`, while the `const` definition would be invoked for a `const` `Array`.

Remark: The previously implemented member functions of `Array` are assumed to be implemented *within* the class definition. For example, we wrote `operator[]` as if it were implemented within the `Array` definition:

```
template <typename T>
class Array {
  T* data;
  size_t length;
public:
  ...
  const T& operator[](size_t idx) const {
    if (idx < length) {
      return data[idx];
    } // if
    throw std::runtime_error{"Invalid index provided to operator[]"};
  } // operator[]()
  ...
};
```

If you want to define a member function *outside* the class definition, you will need to use the scope resolution operator (`::`) to identify the class that the member function belongs to (with template declarations as needed). The scope resolution operator goes before the function name, but after the return type. Examples using the member functions we have implemented so far are shown below:

```
template <typename T>
class Array {
  T* data;
  size_t length;
public:
  // define member function headers within the class, but
  // implement them outside the class using scope resolution (::)
  Array(size_t len = 0);
  ~Array();
  Array(const Array& other);
  void swap_array(Array& other);
  Array& operator=(const Array& rhs);
  T& operator[](size_t idx);
  const T& operator[](size_t idx) const;
  size_t size() const;
};

template <typename T>
Array<T>::Array(size_t len) : length{len} {
  data = (len ? new T[len] : nullptr);
} // Array()

template <typename T>
Array<T>::~Array() {
  delete[] data;
  data = nullptr;
} // ~Array()

template <typename T>
Array<T>::Array(const Array& other)
  : data{new T[other.length]}, length{other.length} {
  for (size_t i = 0; i < length; ++i) {
    data[i] = other.data[i];
  } // for i
} // Array()

template <typename T>
void Array<T>::swap_array(Array& other) {
  std::swap(data, other.data);
  std::swap(length, other.length);
} // swap_array()

template <typename T>
Array<T>& Array<T>::operator=(const Array& rhs) {
  Array temp{rhs};
  swap_array(temp);
  return *this;
} // operator=()

template <typename T>
T& Array<T>::operator[](size_t idx) {
  if (idx < length) {
    return data[idx];
  } // if
  throw std::runtime_error{"Invalid index provided to operator[]"};
} // operator[]()
```

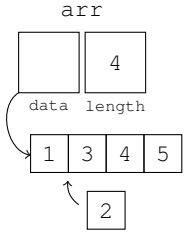
```
58  template <typename T>
59  const T& Array<T>::operator[](size_t idx) const {
60    if (idx < length) {
61      return data[idx];
62    } // if
63    throw std::runtime_error{"Invalid index provided to operator[]"};
64  } // operator[]()
65
66  template <typename T>
67  size_t Array<T>::size() const {
68    return length;
69  } // size()
```

※ 6.8.7 *Implementing Insert*

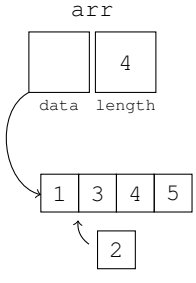
What if you wanted to implement a member function that could insert an element anywhere into the `Array` container? This process may seem simple at first, but you will quickly realize it is complicated for two primary reasons:

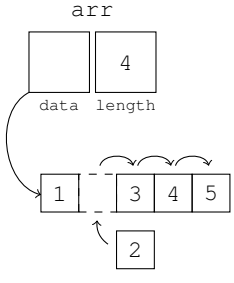
1. Elements in an array-based container must be stored contiguously in memory.
2. Once you allocate a `data` array, its size cannot change; if you want more space, you must allocate a new array and move the data over.

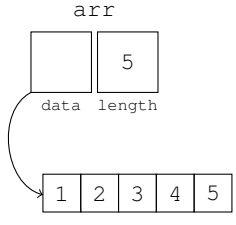
Consider the following `Array`. Suppose you wanted to insert the number `2` between `1` and `3`:



Because elements in an array-based container must be contiguous in memory, the element `2` must end up *directly after* the element `1` and *directly before* the element `3` after the insertion. For this to happen, every element after the insertion point must be shifted one to the right to create space for the `2`.







We could attempt to write an insert function as follows. This function takes in two arguments: the index of insertion and the value to insert. The function then shifts all the elements after the insertion point and inserts the value at the specified index (e.g., `insert(1, 2)` would insert `2` at index `1` of the `Array`). The function returns `true` if the element was successfully inserted and `false` if it was not.

```
1   template <typename T>
2   bool insert(size_t idx, T val) {
3     if (idx < length) {
4       for (size_t i = length - 1; i > idx; --i) {
5         data[i] = data[i - 1];
6       } // for i
7       data[idx] = val;
8       return true;
9     } // if
10    return false;
11  } // insert()
```

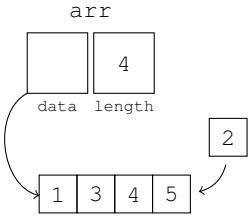
If we analyze this function's complexity, we can conclude the following (assuming that `idx` is valid and the array has n elements):

- The *best-case* time complexity is $\Theta(1)$: this occurs when an element is inserted at the very end of the array (since no shifting is needed).
- The *worst-case* time complexity is $\Theta(n)$: this occurs when an element is inserted at the very beginning of the array (since all elements after it must be shifted).
- The *average-case* time complexity is $\Theta(n)$: if we consider all possible cases, the average number of elements we have to shift is $n/2$ (and $\Theta(n/2) = \Theta(n)$ after dropping coefficients).

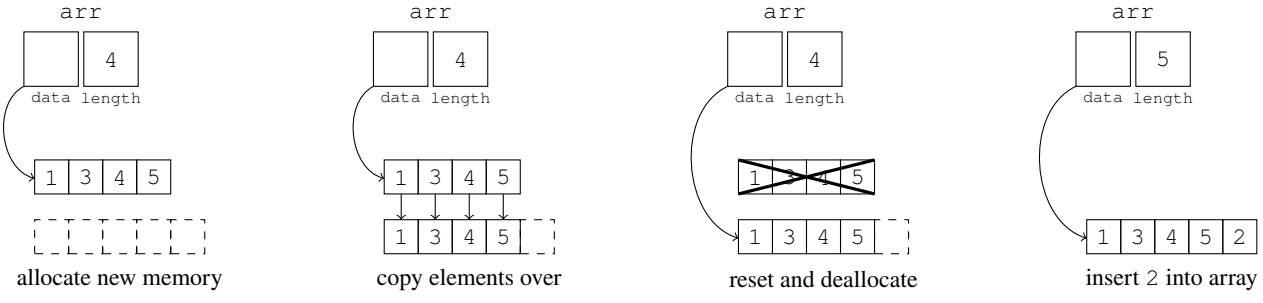
However, this insertion method does not work in all cases because of the other issue: once an array is allocated, its size is fixed! For instance, if `data` is initialized to a heap-allocated array of size 4, that array cannot expand in size. As a result, we would not even be able to insert `2` using

the implementation above. If we wanted our underlying `data` array to grow with the size of the data, we would have to allocate a larger array elsewhere and move our current data there.

Let's consider a simpler insertion example. Instead of inserting `2` between `1` and `3`, let's suppose we wanted to insert `2` at the end of the array. The array is initialized to have a size of 4 (e.g., `Array<int32_t> arr{4};`).



Since the underlying `data` array was initialized to hold 4 elements (e.g., `data = new int32_t[4];`), its size is fixed at 4 and cannot be changed. If we want to store more than 4 elements, we would have to allocate an entirely new, larger array so that the 5th element can be added in. One way to do this is to allocate an entirely new array that can hold 5 elements, copy over the original 4 elements, deallocate the old array, and add in the 5th element, as shown:



This approach, however, is not the most performant. This is because the copying process takes $\Theta(n)$ time for n elements, and you would need to make a copy every time an element is inserted beyond the array's capacity! For instance, if you had an array with 1,000 elements, adding the 1,001st and 1,002nd elements would require two array allocations and 2,001 copies (1,000 on the first allocation; 1,001 on the second).

To reduce the number of copies needed while still ensuring space is available for additional elements, we could plan in advance and allocate more memory than needed. Instead of incrementing the capacity by 1 every time, we could prepare for future insertions by *doubling* the capacity of the array whenever reallocation is done. For example, instead of increasing our array capacity from 4 to 5, we could increase it from 4 to 8:

If we do this, adding the 6th, 7th, and 8th elements would not require the array to be copied. When the 9th element is added, the array is reallocated and its elements are copied over, but the new array capacity would double to 16 — this would allow us to insert the 10th to 16th elements in constant time without any additional copying overhead.

If we double on reallocation, we would allocate more space than we need, and the size and capacity of the underlying data array may be different. Thus, we will need to keep track of both the size and capacity values. The capacity would tell us when we would have to grow our array and reallocate, while the size would tell us the index of the last valid element (e.g., if the array had a capacity of 8 but only 5 valid elements, the user of the array should not be accessing indices 5 to 7). Using this implementation, we can redefine our `Array` member variables as follows:

```
1  template <typename T>
2  class Array {
3    T* data;            // pointer to Array data
4    size_t size;        // number of valid elements in 'data' array
5    size_t capacity;    // capacity of underlying 'data' array
6    ...
7  };
```

Here, the `size` member variable represents the number of valid elements that are in the array, while the `capacity` member variable represents the maximum number of elements that the underlying array can hold. In the previous example, the `size` of the array would be `5` and the capacity of the array would be `8`.

Could we plan further in advance by tripling or quadrupling the array capacity with every new allocation? We could, but this would be wasteful when it comes to memory. Although tripling or quadrupling capacity could reduce the number of copies we have to make, there is no guarantee that all this memory will be used, and the improvement in performance is asymptotically insignificant.

We have gotten quite far with our custom-implemented `Array` class in this section, implementing a container that abstracts away the work of memory allocation from the user and supports several features that are not present with just a standard C-style array. However, there is still more we can do. For instance, we can implement a member function that can delete elements from our container, or a member function that can manually change the capacity of the underlying `data` array.

However, we will not be implementing any more member functions in our custom `Array` class. First, several of these features can get quite complicated. Insertion and deletion, for instance, both require elements in the array to be shifted. More importantly, however, doing so would not be necessary — most of the work has already been done for us, so we do not need to spend time reinventing the wheel! C++ already provides us with an array-based container that manages dynamic memory and performs all of the functionalities we have implemented in this section (and much more!). This container is known as the `std::vector<>`, which we will cover in the next chapter.

6.9 Lvalues, Rvalues, and Move Semantics (⁎)

In this section, we will be briefly discussing the concept of **move semantics**, which allows resources to be transferred between objects instead of copied. Move semantics can be used to optimize the performance of your code, as it allows you to avoid making unnecessary copies of temporary objects that you know are not going to exist for much longer.

Remark: The material in this section is beyond the scope of this class, and you will *not* need to know it for projects or exams. That being said, you are still allowed to take advantage of move semantics in your projects or lab assignments if you know how they work. If you don't know how move semantics work and don't have the time or commitment to read this section, that's perfectly fine as well! The runtime benchmarks that you are required to meet for coding assignments in this class are based off of instructor solutions that do not take advantage of these optimizations.

※ 6.9.1 *Lvalues and Rvalues* (⁎)

To begin, we must first introduce the concepts of lvalues and rvalues. **Lvalues**, historically known as "left-hand" values since they can go on the left-hand side of an assignment, are named objects with an identifiable location in memory (because of this, lvalues are also known as *locator* values). **Rvalues**, historically known as "right-hand" values since they can only go on the right-hand side of an assignment, are temporary, unnamed objects that cannot be assigned to (i.e., they cannot go on the left-hand side of an assignment). For example, consider the following:

```
int32_t x = 281;
```

There are two components to this assignment. On the left-hand side, we have the integer variable `x`, which is named and has an identifiable address in memory. Thus, the variable `x` is an lvalue. On the right-hand side, we have the number `281`, which is a temporary numeric value that has no identifiable location in memory until it is assigned. Thus, `281` is an rvalue.

Lvalues can be assigned to rvalues or other lvalues, but rvalues cannot be assigned to at all. For instance, the following is okay since both `x` and `y` are lvalues, and thus can be assigned to:

```
int32_t x = 281;
int32_t y = x;   // OK, x and y are both lvalues
```

However, the following is not okay, since `281` is an rvalue, and thus cannot be assigned to:

```
int32_t x = 281;
281 = x;         // NOT OK, 281 is an rvalue
```

If you return a value from a function that returns by value (instead of by reference), then that value will be returned as an rvalue. In the following code, `get_favorite_class()` returns an rvalue, which is then assigned to the lvalue `x`:

```
1  int32_t get_favorite_class() {
2    static int32_t fav_class = 281;
3    return fav_class;
4  } // get_favorite_class()
5
6  int main() {
7    int32_t x = get_favorite_class();  // x is lvalue, get_favorite_class() is rvalue
8  } // main()
```

Because `get_favorite_class()` is an rvalue here, the following would not work:

```
get_favorite_class() = 370;  // not OK, get_favorite_class() is an rvalue
```

However, this would work if the `get_favorite_class()` returned by reference instead of by value. If a function returns by reference, it would return an lvalue reference, which can be assigned to:

```
1  int32_t& get_favorite_class() {
2    static int32_t fav_class = 281;
3    return fav_class;
4  } // get_favorite_class()
5
6  int32_t main() {
7    get_favorite_class() = 370;  // OK, get_favorite_class() is an lvalue reference
8  } // get_favorite_class()
```

You cannot construct an lvalue reference from an rvalue; if you declare an lvalue reference, you must assign it with another lvalue:

```
int32_t& x = 281;  // not OK, since 281 is an rvalue
```

There is an exception, however: an lvalue reference *can* be assigned to an rvalue if the lvalue reference is declared as `const`:

```
const int32_t& x = 281;  // OK, since x is a const lvalue reference
```

This distinction is important. For instance, consider the following two functions:

```
void foo(std::string& s) {
  /* ... do stuff ... */
} // foo()
```

```
void bar(const std::string& s) {
  /* ... do stuff ... */
} // bar()
```

Because the `foo()` function takes in a non-`const` lvalue reference, it can only be called on other lvalues. However, the `bar()` function takes in a `const` lvalue reference, which allows it to be called on both lvalues and rvalues:

```
std::string str1 = "EECS";
std::string str2 = "281";

foo(str1);          // OK, str1 is lvalue
foo(str2);          // OK, str2 is lvalue
foo(str1 + str2);   // NOT OK, str1 + str2 is rvalue
foo("EECS 281");    // NOT OK, "EECS 281" is rvalue

bar(str1);          // OK, str1 is lvalue
bar(str2);          // OK, str2 is lvalue
bar(str1 + str2);   // OK, str1 + str2 is rvalue, but lvalue reference parameter is const
bar("EECS 281");    // OK, "EECS 281" is rvalue, but lvalue reference parameter is const
```

So, why do we care so much about lvalues and rvalues, and how can they help us optimize our code? The reason is that we can operate on lvalues and rvalues with different levels of care. If we are given an lvalue, we have to be careful with how we work with its data, since it is entirely possible that something else will need to use it after we are done. However, if we are given an rvalue, we know that the object we are working on is temporary, and that we can abuse it however we want without worrying about consequences down the road. This gives us the ability to make optimizations that we could not have done with lvalues, such as stealing an object's resources and distributing them somewhere else.

This leads us to the concept of **rvalue references**. An rvalue reference is denoted with a double ampersand (`&&`), and can only take on temporary rvalue objects. For example, consider the following function:

```
void baz(std::string&& s) {
  /* ... do stuff ... */
} // baz()
```

Unlike the previous two functions, this function only accepts rvalues (since the string is passed in with a double ampersand).

```
std::string str1 = "EECS";
std::string str2 = "281";

baz(str1);          // NOT OK, str1 is lvalue
baz(str2);          // NOT OK, str2 is lvalue
baz(str1 + str2);   // OK, str1 + str2 is rvalue
baz("EECS 281");    // OK, "EECS 281" is rvalue
```

Rvalue references allow us to perform different optimizations depending on whether an object is temporary or not. If we have a function that can perform special optimizations on rvalues, we can simply write an overloaded version of that function that takes in an rvalue reference, as shown:

```
void qux(const std::string& s) {
  // runs if input is an lvalue
  // even though rvalues can be accepted as const lvalue references
  // rvalues will always be sent to the && version if one exists
  ...
} // qux()

void qux(std::string&& s) {
  // runs if input is a temporary rvalue
  // can do rvalue optimizations (e.g., safely steal data from s)
  ...
} // qux()

int main() {
  std::string str1 = "EECS";
  std::string str2 = "281";

  qux(str1);          // first version runs (const& s)
  qux(str2);          // first version runs (const& s)
  qux(str1 + str2);   // second version runs (&&s)
  qux("EECS 281");    // second version runs (&&s)
} // main()
```

※ 6.9.2 *Implementing the Move Constructor* (✻)

One practical use case of rvalue references comes up with move semantics, which allows us to avoid unnecessary copies of temporary rvalue objects. As an example, consider the following implementation of a `String` class:

```
1   class String {
2     char* data;       // pointer to char array on heap
3     size_t length;    // length of String
4   public:
5     // constructor from char*
6     String(const char* str) {
7       length = strlen(str);
8       data = new char[length + 1];
9       std::cout << "Char Ctor: Heap allocation made" << std::endl;
10      for (size_t i = 0; i < length; ++i) {
11        data[i] = str[i];
12      } // for i
13      data[length] = '\0';
14    } // String()
15
16    // copy constructor from another String
17    String(const String& other)
18      : data{new char[other.length + 1]}, length{other.length} {
19      std::cout << "Copy Ctor: Heap allocation made" << std::endl;
20      for (size_t i = 0; i < length; ++i) {
21        data[i] = other.data[i];
22      } // for i
23      data[length] = '\0';
24      std::cout << "Copy Ctor: String successfully copied" << std::endl;
25    } // String()
26    /* ... other member functions ...*/
27  };
```

Now, suppose we have a `Student` object that stores a `String` object internally, as shown below:

```
29  class Student {
30    String name;
31  public:
32    // constructor
33    Student(const String& name_in)
34      : name{name_in} {}
35    /* ... other member functions ... */
36  };
```

Let's instantiate a `Student` object:

```
38  int main() {
39    Student s{"Alice"};
40  } // main()
```

If we try to run `main()`, the following would get printed:

```
Char Ctor: Heap allocation made
Copy Ctor: Heap allocation made
Copy Ctor: String successfully copied
```

What happened here? Even though we created a single `Student` object that stores a single `String` object, we somehow ended up making two heap allocations and a copy operation. *This is because we ended up copying a temporary rvalue!* When the `Student` object was constructed on line 39, the string `"Alice"` was first converted into a temporary `String` object (an rvalue), which resulted in the first heap allocation (line 8). Then, this rvalue was passed into the constructor and assigned into the `name` member variable of the `Student` object. This invokes the `String` copy constructor, which makes a *copy* of the temporary rvalue that we just created, allocates new memory for this copy (line 18), and then copies the contents of the rvalue to this new copy. The rvalue is then destructed after this is all done.

Obviously, this is inefficient: we ended up making two heap allocations and a copy operation for no reason, when we could have just constructed a `String` object using a single heap allocation. This is where move semantics come in; instead of doing this excess work, we can *transfer* the data from the first `String` we constructed to the `name` variable of the `Student` object. By doing so, only a single heap allocation is needed: when the `String` is first constructed on line 39.

For this to work, we will need to overload the standard copy constructor to accept rvalue references. A constructor that accepts an rvalue of the same type is known as a **move constructor**, and it is invoked when an object is initialized to an rvalue. Unlike the copy constructor, the move constructor does not need to allocate new memory and copy the `String` over. Instead, since we know the `String` passed into the move constructor is an rvalue, we can safely take ownership of its data without the need to make a new copy, as the rvalue will get destroyed afterward.

The move constructor is implemented below:

```
1   class String {
2     char* data;
3     size_t length;
4   public:
5     /* ... previously defined members not shown to save space ... */
6     // move constructor
7     String(String&& other) noexcept
8       : data{other.data}, length{other.length} {
9       other.data = nullptr;
10      other.length = 0;
11      std::cout << "Move Ctor: Data transferred from rvalue" << std::endl;
12    } // String()
13    /* ... other member functions ...*/
14  };
```

After the move constructor "steals" the data from the rvalue that was passed in (via a shallow copy on line 8), it sets the rvalue's `data` pointer to `nullptr`. **This step is important, since the rvalue's data will get destructed after the move constructor completes! If you don't set the rvalue's data pointer to `nullptr`, the data you stole will also end up getting deleted when the rvalue gets cleaned up!**

Remark: Move constructors (and move assignment operators, which we will discuss later) should be declared as `noexcept`. This essentially indicates that the move constructor (and the move assignment operator) should not throw any exceptions. The `noexcept` keyword was covered back in chapter 1 (in the exceptions section).

We will also add a member function to the `Student` class to handle rvalue references for the move constructor:

```
1   class Student {
2     String name;
3   public:
4     // constructor (for lvalues)
5     Student(const String& name_in)
6       : name{name_in} {}
7     // constructor (for rvalues)
8     Student(String&& name_in)
9       : name{name_in} {}
10    /* ... other member functions ... */
11  };
```

If we run the code again and check the output, this is what we get:

```
Char Ctor: Heap allocation made
Copy Ctor: Heap allocation made
Copy Ctor: String successfully copied
```

Nothing changed! Why did this happen? Shouldn't line 9 invoke the move constructor, since `name_in` is passed in as an rvalue reference? Well, it turns out there is a catch! It is true that the constructor on lines 8-9 only runs if the `String` that is passed is an rvalue reference. However, once the `String` is passed into this constructor, it gains a name (`name_in`) and an identifiable location in memory (i.e., you can take its address). Thus, the rvalue reference in the function parameter *behaves like an lvalue reference inside the function!* For the move constructor to be invoked, you would have to explicitly cast `name_in` back into an rvalue in the `String` constructor. This can be done using the `std::move()` function, which is added to line 9 below.

```
1   class Student {
2     String name;
3   public:
4     // constructor (for lvalues)
5     Student(const String& name_in)
6       : name{name_in} {}
7     // constructor (for rvalues)
8     Student(String&& name_in)
9       : name{std::move(name_in)} {}
10    /* ... other member functions ... */
11  };
```

After running this modified code, we get the following output, which is what we want. The `String` object is only allocated once, and its data is *transferred* to the `name` member of the `Student` object instead of copied.

```
Char Ctor: Heap allocation made
Move Ctor: Data transferred from rvalue
```

Remark: The `std::move()` function casts its argument to an rvalue; it does not actually physically move anything in memory. When you call `std::move()` on an object, you are essentially telling the compiler that it can treat that object as an rvalue, and that its data can be safely stolen without repercussions (the actual "stealing" is done in the move constructor or move assignment operator). If you call `std::move()` on an object to transfer its data, it ends up in an unspecified state, and you should *not* use it until it is assigned a new value.

※ 6.9.3 *Implementing the Overloaded Move Assignment Operator* (✻)

The rules for transferring ownership of data also apply to the assignment operator. The standard overloaded assignment operator implementation (`operator=`) has to make a copy of the object it is passed in before assigning it (i.e., if you want to assign `a = b`, you have to make a copy of `b` and then assign it to `a`). However, if you are assigning an object from a rvalue, you can take advantage of rvalue optimizations by implementating the **overloaded move assignment operator**. The implementation of this operator is very similar to the implementation of the move constructor: simply overload the operator to take in an rvalue reference, and then steal all the data associated with this rvalue. An implementation using the above `String` object is shown below:

```
class String {
  char* data;
  size_t length;
public:
  /* ... previously defined members not shown to save space ... */
  // move assignment operator
  String& operator=(String&& rhs) noexcept {
    // don't do anything if self-assigning
    if (this == &rhs) {
      return *this;
    } // if
    // delete original data
    delete[] data;
    // steal length and data pointer from rvalue
    length = rhs.length;
    data = rhs.data;
    // clear rvalue's data so the data you stole is
    // not deleted when the rvalue gets destructed
    rhs.length = 0;
    rhs.data = nullptr;
  } // operator=()
  /* ... other member functions ...*/
};
```

Much like the standard assignment operator, we can get rid of the self-assignment check by using `std::swap()` to swap the contents of an object with the contents of the rvalue it is being assigned.

```
class String {
  char* data;
  size_t length;
public:
  /* ... previously defined members not shown to save space ... */
  // move assignment operator
  String& operator=(String&& rhs) noexcept {
    std::swap(this->length, rhs.length);
    std::swap(this->data, rhs.data);
    return *this;
  } // operator=()
  /* ... other member functions ...*/
};
```

Remark: Before C++11 introduced move semantics, the `std::swap()` function was essentially implemented like this:

```
template <typename T>
void swap(T& a, T& b) {
  T temp = a;
  a = b;
  b = temp;
} // swap()
```

However, now that we have covered move semantics, we can build a better `std::swap()` function that does not perform unnecessary copies. Instead of making copies of the data we want to swap, we can use move construction and move assignment to transfer ownership of data instead. This is essentially how `std::swap()` is implemented in C++11 and later:

```
template <typename T>
void swap(T& a, T& b) {
  T temp = std::move(a);
  a = std::move(b);
  b = std::move(temp);
} // swap()
```

※ 6.9.4 *Return Value Optimization* (✻)

You should avoid using `std::move()` to return a local object from a function, even if it may seem like an optimization. This is because it suppresses something known as **return value optimization (RVO)**. Consider the following function:

```
1  std::string gen_big_string(size_t n) {
2    std::string big_string;
3    /* ... generate big string ... */
4    return big_string;
5  } // gen_big_string()
6
7  int main() {
8    std::string str = gen_big_string(281);
9  } // main()
```

It may be tempting to call `std::move()` on `big_string` when it is returned, since we do not want the `big_string` to be copied upon return. However, compilers are smart enough to detect situations like this! If a function returns a local object by value, and the object's type matches the return type of the function, the compiler may use RVO to build the object *directly in its intended destination in memory*. That is, instead of building a temporary copy of `big_string` and then copying (or moving) it to the memory address of `str`, RVO allows the contents of `big_string` to be constructed *directly* at the memory address of `str`. If RVO occurs, no copying or moving needs to be done! Because the copy/move operation is elided (i.e., omitted) by the compiler under the specified conditions, the RVO procedure is formally known as a *copy elision* optimization.

However, consider what happens if we attempt to call `std::move()` on a local object before we return it from a function:

```
1  std::string gen_big_string(size_t n) {
2    std::string big_string;
3    /* ... generate big string ... */
4    return std::move(big_string);
5  } // gen_big_string()
6
7  int main() {
8    std::string str = gen_big_string(281);
9  } // main()
```

This may seem innocuous, but it may actually be detrimental to the performance of your code! If you call `std::move()` on a local object before you return it from a function, you are no longer returning a local object of the same type as the return type of the function. Instead, you are returning an rvalue reference to that object (in this case, `std::string&&`), which violates the conditions required for RVO. As a result, the compiler is unable to take advantage of copy elision and cannot construct `big_string` directly in its intended destination. Instead, it has to build a separate instance of the `big_string` object in the stack frame of `gen_big_string()` and then transfer its data into the memory address of `str` in the stack frame of `main()`. This is why you should not call `std::move()` when returning a local object that is eligible for RVO: by doing so, you would prevent your compiler from performing copy elision optimizations!

※ 6.9.5 *Perfect Forwarding and Forwarding References* (✻)

Lastly, we will briefly discuss a concept related to move semantics known as **perfect forwarding**. With perfect forwarding, a templated function accepts a special type of reference known as a **forwarding reference** (denoted using a double ampersand `&&`), which can be used in conjunction with `std::forward<>` to preserve the original value category of its argument (i.e., whether it is an lvalue or rvalue). An example of perfect forwarding is shown below, where the templated function `foo()` takes in an argument and passes it into another function `bar()`.

```
1  template <typename T>
2  void foo(T&& arg) {
3    bar(std::forward<T>(arg));
4  } // foo
```

If `arg` gets passed into `foo()` as an lvalue reference, it also gets passed to the `bar()` method as an lvalue reference. If `arg` gets passed into `foo()` as an rvalue reference, it also gets passed to the `bar()` method as an rvalue reference.

At this point, you might be wondering: what is the purpose of the `std::forward<>` method on line 3? The use of `std::forward<>` is needed here because, as we mentioned previously, all function parameters behave like lvalues, even if they were initially passed in as an rvalue (since the function argument itself is a named object). Hence, the variable `arg` on line 2 will always be an lvalue even if `foo()` was called on an rvalue. What `std::forward<>` does here is that it performs a *conditional* cast on `arg` based on the value category of the argument that was passed into `foo()`. If `foo()` was called on an lvalue reference, then `std::forward<>` does not need to do anything (since the function parameter `arg` is already an lvalue), and `bar()` is thereby called on the same lvalue reference. However, if `foo()` was called on an rvalue reference, `std::forward<>` casts `arg` into an rvalue before passing it into `bar()`. This ensures that an rvalue that is passed into `foo()` will also be passed as an rvalue into `bar()`.

While we will not go into too much detail here, `std::forward<>` is able to detect whether `foo()` was invoked on an lvalue or an rvalue using a process known as *reference collapsing* when deducing templates. When the compiler deduces templates involving references:

- An argument of type `T& &` resolves to `T&` (lvalue reference).
- An argument of type `T& &&` resolves to `T&` (lvalue reference).
- An argument of type `T&& &` resolves to `T&` (lvalue reference).
- An argument of type `T&& &&` resolves to `T&&` (rvalue reference).

Whenever you have a forwarding reference in the following format:

```
1  template <typename T>
2  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).