1

Lists, Stacks, and Queues

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the importance of using the right data structure in any application
- Implement various built-in data structures, depending on the problem, to make application development easier
- Implement a custom linear data structure suited for given situations if the ones provided by C++ are not good enough for the use case
- Analyze real-life problems where different types of linear data structures are helpful and decide which one will be the most suitable for a given use case

This chapter describes the importance of using the right data structures in any application. We will learn how to use some of the most common data structures in C++, as well as built-in and custom containers, using these structures.

Introduction

The management of data is one of the most important considerations to bear in mind while designing any application. The purpose of any application is to get some data as input, process or operate on it, and then provide suitable data as output. For example, let's consider a hospital management system. Here, we could have data about different doctors, patients, and archival records, among other things. The hospital management system should allow us to perform various operations, such as admit patients, and update the joining and leaving of doctors of different specialties. While the user-facing interface would present information in a format that is relevant to the hospital administrators, internally, the system would manage different records and lists of items.

A programmer has at their disposal several structures to hold any data in the memory. The choice of the right structure for holding data, also known as a **data structure**, is crucial for ensuring reliability, performance, and enabling the required functionalities in the application. Besides the right data structures, the right choice of algorithms to access and manipulate the data is also necessary for the optimal behavior of the application. This book shall equip you with the ability to implement the right data structures and algorithms for your application design, in order to enable you to develop well-optimized and scalable applications.

This chapter introduces basic and commonly used linear data structures provided in C++. We will look at their individual designs, pros, and cons. We will also implement said structures with the help of exercises. Understanding these data structures will help you to manage data in any application in a more performant, standardized, readable, and maintainable way.

Linear data structures can be broadly categorized as contiguous or linked structures. Let's understand the differences between the two.

Contiguous Versus Linked Data Structures

Before processing the data in any application, we must decide how we want to store data. The answer to that question depends on what kind of operations we want to perform on the data and the frequency of the operations. We should choose the implementation that gives us the best performance in terms of latency, memory, or any other parameter, without affecting the correctness of the application.

A useful metric for determining the type of data structure to be used is algorithmic complexity, also called **time complexity**. Time complexity indicates the relative amount of time required, in proportion to the size of the data, to perform a certain operation. Thus, time complexity shows how the time will vary if we change the size of the dataset. The time complexity of different operations on any data type is dependent on how the data is stored inside it.

Data structures can be divided into two types: contiguous and linked data structures. We shall take a closer look at both of them in the following sections.

Contiguous Data Structures

As mentioned earlier, **contiguous data structures** store all the elements in a single chunk of memory. The following diagram shows how data is stored in contiguous data structures:

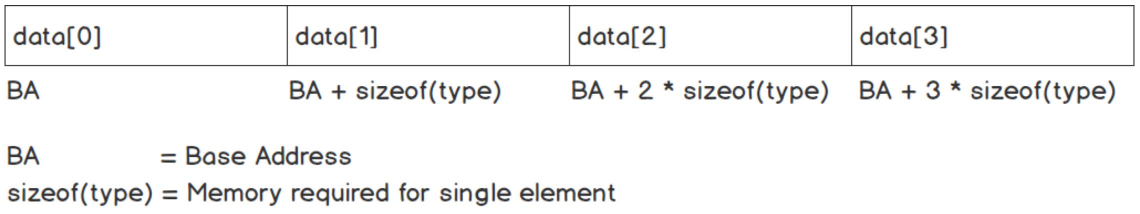


Figure 1.1: Diagrammatic representation of contiguous data structures

In the preceding diagram, consider the larger rectangle to be the single memory chunk in which all the elements are stored, while the smaller rectangles represent the memory allocated for each element. An important thing to note here is that all the elements are of the same type. Hence, all of them require the same amount of memory, which is indicated by `sizeof(type)`. The address of the first element is also known as the **Base Address** (**BA**). Since all of them are of the same type, the next element is present in the `BA + sizeof(type)` location, and the one after that is present in `BA + 2 * sizeof(type)`, and so on. Therefore, to access any element at index `i`, we can get it with the generic formula: `BA + i * sizeof(type)`.

In this case, we can always access any element using the formula instantly, regardless of the size of the array. Hence, the access time is always constant. This is indicated by O(1) in the Big-O notation.

The two main types of arrays are static and dynamic. A static array has a lifetime only inside its declaration block, but a dynamic array provides better flexibility since the programmer can determine when it should be allocated and when it should be deallocated. We can choose either of them depending on the requirement. Both have the same performance for different operations. Since this array was introduced in C, it is also known as a C-style array. Here is how these arrays are declared:

- A static array is declared as `int arr[size];`.
- A dynamic array in C is declared as `int* arr = (int*)malloc(size * sizeof(int));`.
- A dynamic array is declared in C++ as `int* arr = new int[size];`.

A static array is aggregated, which means that it is allocated on the stack, and hence gets deallocated when the flow goes out of the function. On the other hand, a dynamic array is allocated on a heap and stays there until the memory is freed manually.

Since all the elements are present next to each other, when one of the elements is accessed, a few elements next to it are also brought into the cache. Hence, if you want to access those elements, it is a very fast operation as the data is already present in the cache. This property is also known as cache locality. Although it doesn't affect the asymptotic time complexity of any operations, while traversing an array, it can give an impressive advantage for contiguous data in practice. Since traversing requires going through all the elements sequentially, after fetching the first element, the next few elements can be retrieved directly from the cache. Hence, the array is said to have good cache locality.

Linked Data Structures

Linked data structures hold the data in multiple chunks of memory, also known as nodes, which may be placed at different places in the memory. The following diagram shows how data is stored in linked data structures:

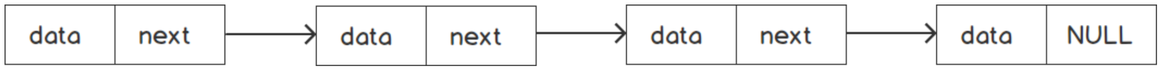


Figure 1.2: Linked data structures

In the basic structure of a linked list, each node contains the data to be stored in that node and a pointer to the next node. The last node contains a **`NULL`** pointer to indicate the end of the list. To reach any element, we must start from the beginning of the linked list, that is, the head, and then follow the next pointer until we reach the intended element. So, to reach the element present at index **`i`**, we need to traverse through the linked list and iterate **`i`** times. Hence, we can say that the complexity of accessing elements is O(*n*); that is, the time varies proportionally with the number of nodes.

If we want to insert or delete any element, and if we have a pointer to that element, the operation is really small and quite fast for a linked list compared to arrays. Let's take a look at how the insertion of an element works in a linked list. The following diagram illustrates a case where we are inserting an element between two elements in a linked list:

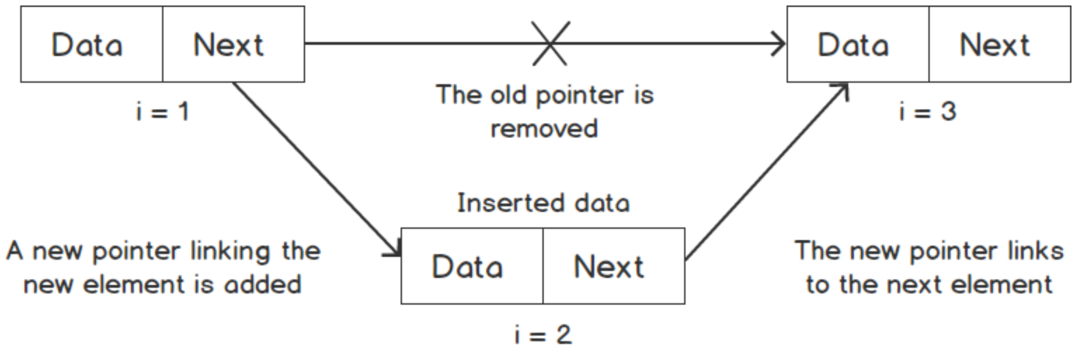

Data
Next
Data
Next
The old pointer is removed
i = 1
i = 3
Inserted data
A new pointer linking the new element is added
Data
Next
The new pointer links to the next element
i = 2

Figure 1.3: Inserting an element into a linked list

For insertion, once we've constructed the new node to be inserted, we just need to rearrange the links so that the next pointer of the preceding element ($i = 1$) points to the new element ($i = 2$) instead of its current element ($i = 3$), and the next pointer of the new element ($i = 2$) points to the current element's next element ($i = 3$). In this way, the new node becomes part of the linked list.

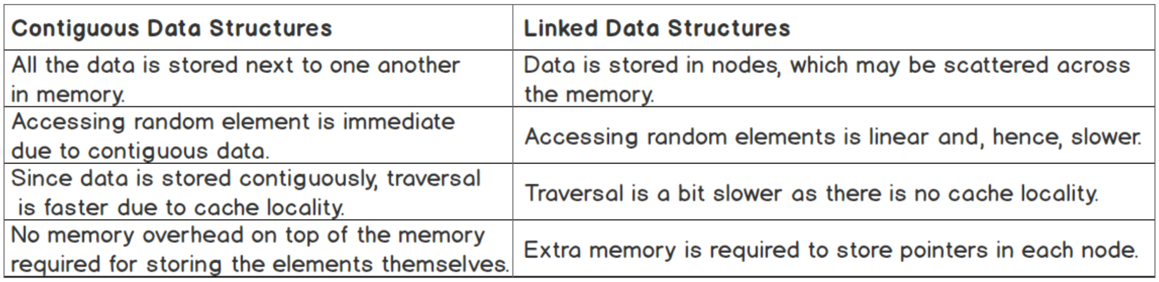
Similarly, if we want to remove any element, we just need to rearrange the links so that the element to be deleted is no longer connected to any of the list elements. Then, we can deallocate that element or take any other appropriate action on it.

A linked list can't provide cache locality at all since the elements are not stored contiguously in memory. Hence, there's no way to bring the next element into the cache without actually visiting it with the pointer stored in the current element. So, although, in theory, it has the same time complexity for traversal as an array, in practice, it gives poor performance.

The following section provides a summary of the comparison of contiguous and linked data structures.

Comparison

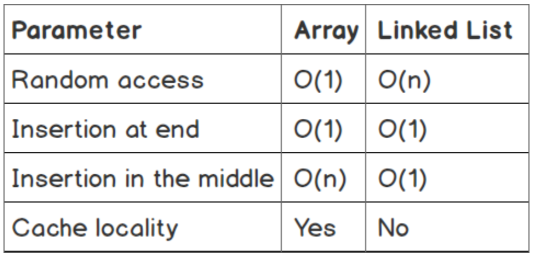
The following table briefly summarizes the important differences between linked and contiguous data structures in general:



Contiguous Data Structures	Linked Data Structures
All the data is stored next to one another in memory.	Data is stored in nodes, which may be scattered across the memory.
Accessing random element is immediate due to contiguous data.	Accessing random elements is linear and, hence, slower.
Since data is stored contiguously, traversal is faster due to cache locality.	Traversal is a bit slower as there is no cache locality.
No memory overhead on top of the memory required for storing the elements themselves.	Extra memory is required to store pointers in each node.

Figure 1.4: Table comparing contiguous and linked data structures

The following table contains a summary of the performance of arrays and linked lists regarding various parameters:



Parameter	Array	Linked List
Random access	O(1)	O(n)
Insertion at end	O(1)	O(1)
Insertion in the middle	O(n)	O(1)
Cache locality	Yes	No

Figure 1.5: Table showing time complexities of some operations for arrays and linked lists

For any application, we can choose either data structure or a combination of both, based on the requirements and the frequencies of the different operations.

Arrays and linked lists are very common and are extensively used in any application to store data. Hence, the implementation of these data structures must be as bug-free and as efficient as possible. To avoid reinventing the code, C++ provides various structures, such as **`std::array`**, **`std::vector`**, and **`std::list`**. We will see some of them in more detail in upcoming sections.

Limitations of C-style Arrays

Though C-style arrays do the job, they are not commonly used. There are a number of limitations that indicate the necessity of better solutions. Some of the major limitations among those are as follows:

- Memory allocation and deallocation have to be handled manually. A failure to deallocate can cause a memory leak, which is when a memory address becomes inaccessible.
- The **`operator[]`** function does not check whether the argument is larger than the size of an array. This may lead to segmentation faults or memory corruption if used incorrectly.
- The syntax for nested arrays gets very complicated and leads to unreadable code.
- Deep copying is not available as a default function. It has to be implemented manually.

To avoid these issues, C++ provides a very thin wrapper over a C-style array called **`std::array`**.

std::array

`std::array` automates the allocation and deallocation of memory. **`std::array`** is a templatized class that takes two parameters – the type of the elements and the size of the array.

In the following example, we will declare **`std::array`** of **`int`** of size **`10`**, set the value of any of the elements, and then print that value to make sure it works:

```
std::array<int, 10> arr;        // array of int of size 10


arr[0] = 1;                     // Sets the first element as 1

std::cout << "First element: " << arr[0] << std::endl;


std::array<int, 4> arr2 = {1, 2, 3, 4};

std::cout << "Elements in second array: ";

  for(int i = 0; i < arr.size(); i++)

    std::cout << arr2[i] << " ";
```

This example would produce the following output:

```
First element: 1
Elements in second array: 1 2 3 4
```

As we can see, **`std::array`** provides **`operator[]`**, which is same as the C-style array, to avoid the cost of checking whether the index is less than the size of the array. Additionally, it also provides a function called **`at(index)`**, which throws an exception if the argument is not valid. In this way, we can handle the exception in an appropriate manner. So, if we have a piece of code where we will be accessing an element with a bit of uncertainty, such as an array index being dependent on user input, we can always catch the error using exception handling, as demonstrated in the following example.

```
try
{
    std::cout << arr.at(4);    // No error
    std::cout << arr.at(5);    // Throws exception std::out_of_range
}
catch (const std::out_of_range& ex)
{
    std::cerr << ex.what();
}
```

Apart from that, passing **`std::array`** to another function is similar to passing any built-in data type. We can pass it by value or reference, with or without **`const`**. Additionally, the syntax doesn't involve any pointer-related operations or referencing and de-referencing operations. Hence, the readability is much better compared to C-style arrays, even for multidimensional arrays. The following example demonstrates how to pass an array by value:

```
void print(std::array<int, 5> arr)
{
    for(auto ele: arr)
    {
        std::cout << ele << ", ";
    }
}
std::array<int, 5> arr = {1, 2, 3, 4, 5};
print(arr);
```

This example would produce the following output:

```
1, 2, 3, 4, 5
```

We can't pass an array of any other size for this function, because the size of the array is a part of the data type of the function parameter. So, for example, if we pass **`std::array<int, 10>`**, the compiler will return an error saying that it can't match the function parameter, nor can it convert from one to the other. However, if we want to have a generic function that can work with **`std::array`** of any size, we can make the size of the array templatized for that function, and it will generate code for all the required sizes of the array. So, the signature will look like the following:

```
template <size_t N>

void print(const std::array<int, N>& arr)
```

Apart from readability, while passing **`std::array`**, it copies all the elements into a new array by default. Hence, an automatic deep copy is performed. If we don't want that feature, we can always use other types, such as reference and **`const`** reference. Thus, it provides greater flexibility for programmers.

In practice, for most operations, **`std::array`** provides similar performance as a C-style array, since it is just a thin wrapper to reduce the effort of programmers and make the code safer. **`std::array`** provides two different functions to access array elements – **`operator[]`** and **`at()`**. **`operator[]`**, is similar to C-style arrays, and doesn't perform any check on the index. However, the **`at()`** function provides a check on the index, and throws an exception if the index is out of range. Due to this, it is a bit slower in practice.

As mentioned earlier, iterating over an array is a very common operation. **`std::array`** provides a really nice interface with the help of a range for loops and iterators. So, the code for printing all the elements in an array looks like this:

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};

for(auto element: arr)

{

    std::cout << element << ' ';

}
```

This example would show the following output:

```
1 2 3 4 5
```

In the preceding example, when we demonstrated printing out all of the elements, we iterated using an index variable, where we had to make sure that it was correctly used according to the size of the array. Hence, it is more prone to human error compared to this example.

The reason we can iterate over **std::array** using a range-based loop is due to iterators. **std::array** has member functions called **begin()** and **end()**, returning a way to access the first and last elements. To move from one element to the next element, it also provides arithmetic operators, such as the increment operator (**++**) and the addition operator (**+**). Hence, a range-based **for** loop starts at **begin()** and ends at **end()**, advancing step by step using the increment operator (**++**). The iterators provide a unified interface across all of the dynamically iterable STL containers, such as **std::array**, **std::vector**, **std::map**, **std::set**, and **std::list**.

Apart from iterating, all the functions for which we need to specify a position inside the container are based on iterators; for example, insertion at a specific position, deletion of elements in a range or at a specific position, and other similar functions. This makes the code more reusable, maintainable, and readable.

> **Note**
>
> For all functions in C++ that specify a range with the help of iterators, the **start()** iterator is usually inclusive, and the **end()** iterator is usually exclusive, unless specified otherwise.

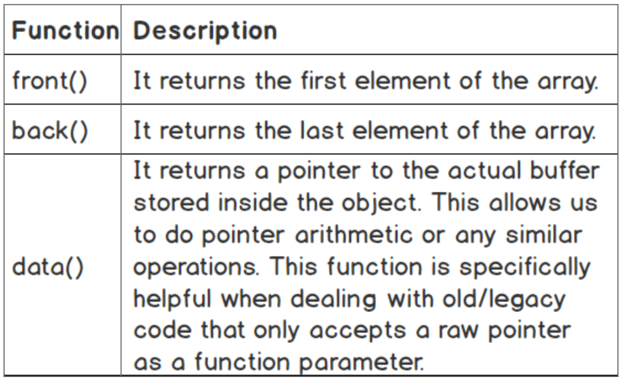
Hence, the **array::begin()** function returns an iterator that points to the first element, but **array::end()** returns an iterator just after the last element. So, a range-based loop can be written as follows:

```
for(auto it = arr.begin(); it != arr.end(); it++)
{
    auto element = (*it);
    std::cout << element << ' ';
}
```

There are some other forms of iterators, such as **const_iterator** and **reverse_iterator**, which are also quite useful. **const_iterator** is a **const** version of the normal iterator. If the array is declared to be a **const**, its functions that are related to iterators, such as **begin()** and **end()**, return **const_iterator**.

reverse_iterator allows us to traverse the array in the reverse direction. So, its functions, such as the increment operator (**++**) and **advance**, are inverses of such operations for normal iterators.

Besides the **`operator[]`** and **`at()`** functions, **`std::array`** also provides other accessors, as shown in the following table:



Function	Description
front()	It returns the first element of the array.
back()	It returns the last element of the array.
data()	It returns a pointer to the actual buffer stored inside the object. This allows us to do pointer arithmetic or any similar operations. This function is specifically helpful when dealing with old/legacy code that only accepts a raw pointer as a function parameter.

Figure 1.6: Table showing some accessors for std::array

The following snippet demonstrates how these functions are used:

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
std::cout << arr.front() << std::endl;       // Prints 1
std::cout << arr.back() << std::endl;        // Prints 5
std::cout << *(arr.data() + 1) << std::endl; // Prints 2
```

Another useful functionality provided by **`std::array`** is the relational operator for deep comparison and the copy-assignment operator for deep copy. All size operators (**<**, **>**, **<=**, **>=**, **==**, **`!=`**) are defined for **`std::array`** to compare two arrays, provided the same operators are also provided for the underlying type of **`std::array`**.

C-style arrays also support all the relational operators, but these operators don't actually compare the elements inside the array; in fact, they just compare the pointers. Therefore, just the address of the elements is compared as integers instead of a deep comparison of the arrays. This is also known as a **shallow comparison**, and it is not of much practical use. Similarly, assignment also doesn't create a copy of the assigned data. Instead, it just makes a new pointer that points to the same data.

> **Note**
>
> Relational operators work for **`std::array`** of the same size only. This is because the size of the array is a part of the data type itself, and it doesn't allow values of two different data types to be compared.

In the following example, we shall see how to wrap a C-style array, whose size is defined by the user.

Exercise 1: Implementing a Dynamic Sized Array

Let's write a small application to manage the student records in a school. The number of students in a class and their details will be given as an input. Write an array-like container to manage the data, which can also support dynamic sizing. We'll also implement some utility functions to merge different classes.

Perform the following steps to complete the exercise:

1. First, include the required headers:

```
#include <iostream>
#include <sstream>
#include <algorithm>
```

2. Now, let's write a basic templated structure called **`dynamic_array`**, as well as primary data members:

```
template <typename T>
class dynamic_array
{
    T* data;
    size_t n;
```

3. Now, let's add a constructor that takes the size of the array and copies it:

```
public:
dynamic_array(int n)
{
    this->n = n;
    data = new T[n];
}

    dynamic_array(const dynamic_array<T>& other)
  {
    n = other.n;
    data = new T[n];

    for(int i = 0; i < n; i++)
    data[i] = other[i];
  }
```

4. Now, let's add **operator[]** and **function()** in the **public** accessor to support the access of data directly, in a similar way to **std::array**:

```
T& operator[](int index)
{
    return data[index];
}

const T& operator[](int index) const
{
    return data[index];
}

T& at(int index)
{
    if(index < n)
    return data[index];
    throw "Index out of range";
}
```

5. Now, let's add a function called **size()** to return the size of the array, as well as a destructor to avoid memory leaks:

```
size_t size() const
{
    return n;
}

~dynamic_array()
{
    delete[] data;   // A destructor to prevent memory leak
}
```

6. Now, let's add iterator functions to support range-based loops to iterate over **dynamic_array**:

```
T* begin()
{
    return data;
}

const T* begin() const
{
    return data;
}

T* end()
{
    return data + n;
}
const T* end() const
{
    return data + n;
}
```

7. Now, let's add a function to append one array to another using the **+** operator. Let's keep it as a **friend** function for better usability:

```
friend dynamic_array<T> operator+(const dynamic_array<T>& arr1, dynamic_
array<T>& arr2)
{
    dynamic_array<T> result(arr1.size() + arr2.size());
    std::copy(arr1.begin(), arr1.end(), result.begin());
    std::copy(arr2.begin(), arr2.end(), result.begin() + arr1.size());

    return result;
}
```

8. Now, let's add a **to_string** function that takes a separator as a parameter with the default value as "**,**":

```
std::string to_string(const std::string& sep = ", ")
{
  if(n == 0)
    return "";
  std::ostringstream os;
  os << data[0];

  for(int i = 1; i < n; i++)
    os << sep << data[i];

  return os.str();
}
};
```

9. Now, let's add a **`struct`** for students. We'll just keep the name and the standard (that is, the grade/class in which the student is studying) for simplicity, and also add **`operator<<`** to print it properly:

```
struct student
{
    std::string name;
    int standard;
};

std::ostream& operator<<(std::ostream& os, const student& s)
{
    return (os << "[Name: " << s.name << ", Standard: " << s.standard <<
"]");
}
```

10. Now, let's add a **`main`** function to use this array:

```
int main()
{
    int nStudents;
    std::cout << "Enter number of students in class 1: ";
    std::cin >> nStudents;

dynamic_array<student> class1(nStudents);
for(int i = 0; i < nStudents; i++)
{
    std::cout << "Enter name and class of student " << i + 1 << ": ";
    std::string name;
    int standard;
    std::cin >> name >> standard;
    class1[i] = student{name, standard};
}

// Now, let's try to access the student out of range in the array
try
{
    class1[nStudents] = student{"John", 8};  // No exception, undefined
behavior
    std::cout << "class1 student set out of range without exception" <<
std::endl;
```

```
        class1.at(nStudents) = student{"John", 8};  // Will throw exception
    }
    catch(...)
    {
    std::cout << "Exception caught" << std::endl;
    }

    auto class2 = class1;  // Deep copy

        std::cout << "Second class after initialized using first array: " <<
    class2.to_string() << std::endl;

        auto class3 = class1 + class2;
        // Combines both classes and creates a bigger one

        std::cout << "Combined class: ";
        std::cout << class3.to_string() << std::endl;

        return 0;
    }
```

11. Execute the preceding code with three students – **`Raj(8)`**, **`Rahul(10)`**, and **`Viraj(6)`** as input. The output looks like the following in the console:

    ```
    Enter number of students in class 1 : 3
    Enter name and class of student 1: Raj 8
    Enter name and class of student 2: Rahul 10
    Enter name and class of student 3: Viraj 6
    class1 student set out of range without exception
    Exception caught
    Second class after initialized using first array : [Name: Raj, Standard:
    8], [Name: Rahul, Standard: 10], [Name: Viraj, Standard: 6]
    Combined class : [Name: Raj, Standard: 8], [Name: Rahul, Standard: 10],
    [Name: Viraj, Standard: 6], [Name: Raj, Standard: 8], [Name: Rahul,
    Standard: 10], [Name: Viraj, Standard: 6]
    ```

    Most of the functions mentioned here have a similar implementation to that of **`std::array`**.

Now that we have seen various containers, we shall learn how to implement a container that can accept any kind of data and store it in a common form in the following exercise.

Exercise 2: A General-Purpose and Fast Data Storage Container Builder

In this exercise, we will write a function that takes any number of elements of any type, which can, in turn, be converted into a common type. The function should also return a container having all the elements converted into that common type, and it should also be fast to traverse:

1. Let's begin by including the required libraries:

```
#include <iostream>
#include <array>
#include <type_traits>
```

2. First, we'll try to build the signature of the function. Since the return type is a container that is fast to traverse, we'll go ahead with **`std::array`**. To allow any number of parameters, we'll use variadic templates:

```
template<typename ... Args>
std::array<?,?> build_array(Args&&... args)
```

   Considering the requirement that the container should be fast to traverse for the return type, we can choose an array or a vector. Since the number of elements is known at the compile time based on the number of parameters to the function, we can go ahead with **`std::array`**.

3. Now, we must provide the type of the elements and the number of elements for **`std::array`**. We can use the **`std::common_type`** template to find out what the type of elements inside **`std::array`** will be. Since this is dependent on arguments, we'll provide the return type of the function as a trailing type:

```
template<typename ... Args>
auto build_array(Args&&... args) -> std::array<typename std::common_
type<Args...>::type, ?>
{
    using commonType = typename std::common_type<Args...>::type;
    // Create array
}
```

4. As shown in the preceding code, we now need to figure out two things – the number of elements, and how to create the array with **commonType**:

   ```
   template< typename ... Args>
   auto build_array(Args&&... args) -> std::array<typename std::common_
   type<Args...>::type, sizeof...(args)>
   {
       using commonType = typename std::common_type<Args...>::type;
       return {std::forward<commonType>(args)...};
   }
   ```

5. Now, let's write the **main** function to see how our function works:

   ```
   int main()
   {
       auto data = build_array(1, 0u, 'a', 3.2f, false);
       for(auto i: data)
           std::cout << i << " ";
       std::cout << std::endl;
   }
   ```

6. Running the code should give the following output:

   ```
   1 0 97 3.2 0
   ```

   As we can see, all final output is in the form of float, since everything can be converted to float.

7. To test this further, we can add the following inside the **main** function and test the output:

   ```
   auto data2 = build_array(1, "Packt", 2.0);
   ```

   With this modification, we should get an error saying that all the types can't be converted to a common type. The exact error message should mention that template deduction has failed. This is because there is no single type in which we can convert both the string and number.

Builder functions, such as the one we have created in this exercise, can be used when you are not sure about the type of data, yet you need to optimize efficiency.

There are a lot of useful features and utility functions that **std::array** doesn't provide. One major reason for this is to maintain similar or better performance and memory requirements compared to C-style arrays.

For more advanced features and flexibility, C++ provides another structure called **std::vector**. We will examine how this works in the next section.

std::vector

As we saw earlier, **`std::array`** is a really good improvement over C-style arrays. But there are some limitations of **`std::array`**, where it lacks functions for some frequent use cases while writing applications. Here are some of the major drawbacks of **`std::array`**:

- The size of **`std::array`** must be constant and provided at compile time, and fixed. So, we can't change it at runtime.
- Due to size limitations, we can't insert or remove elements from the array.
- No custom allocation is possible for **`std::array`**. It always uses stack memory.

In the majority of real-life applications, data is quite dynamic and not a fixed size. For instance, in our earlier example of a hospital management system, we can have more doctors joining the hospital, we can have more patients in emergencies, and so on. Hence, knowing the size of the data in advance is not always possible. So, **`std::array`** is not always the best choice and we need something with dynamic size.

Now, we'll take a look at how **`std::vector`** provides a solution to these problems.

std::vector – Variable Length Array

As the title suggests, **`std::vector`** solves one of the most prominent problems of arrays – fixed size. **`std::vector`** does not require us to provide its length during initialization.

Here are some of the ways in which we can initialize a vector:

```
std::vector<int> vec;
// Declares vector of size 0

std::vector<int> vec = {1, 2, 3, 4, 5};
// Declares vector of size 5 with provided elements

std::vector<int> vec(10);
// Declares vector of size 10

std::vector<int> vec(10, 5);
// Declares vector of size 10 with each element's value = 5
```

As we can see from the first initialization, providing the size is not mandatory. If we don't specify the size explicitly, and if we don't infer it by specifying its elements, the vector is initialized with the capacity of elements depending on the compiler implementation. The term "size" refers to the number of elements actually present in the vector, which may differ from its capacity. So, for the first initialization, the size will be zero, but the capacity could be some small number or zero.

We can insert elements inside the vector using the **push_back** or **insert** functions. **push_back** will insert elements at the end. **insert** takes the iterator as the first parameter for the position, and it can be used to insert the element in any location. **push_back** is a very frequently used function for vectors because of its performance. The pseudocode of the algorithm for **push_back** would be as follows:

```
push_back(val):
    if size < capacity
    // If vector has enough space to accommodate this element
    - Set element after the current last element = val
    - Increment size
    - return;

    if vector is already full
    - Allocate memory of size 2*size
    - Copy/Move elements to newly allocated memory
    - Make original data point to new memory
    - Insert the element at the end
```

The actual implementation might differ a bit, but the logic remains the same. As we can see, if there's enough space, it only takes O(1) time to insert something at the back. However, if there's not enough space, it will have to copy/move all the elements, which will take O(*n*) time. Most of the implementations double the size of the vector every time we run out of capacity. Hence, the O(*n*) time operation is done after n elements. So, on average, it just takes one extra step, making its average time complexity closer to O(1). This, in practice, provides pretty good performance, and, hence, it is a highly used container.

For the **insert** function, you don't have any option other than to shift the elements that come after the given iterator to the right. The **insert** function does that for us. It also takes care of reallocation whenever it is required. Due to the need to shift the elements, it takes O(*n*) time. The following examples demonstrate how to implement vector insertion functions.

Consider a vector with the first five natural numbers:

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

> **Note**
>
> Vector doesn't have a **push_front** function. It has the generic **insert** function, which takes the iterator as an argument for the position.

The generic **insert** function can be used to insert an element at the front, as follows:

```
vec.insert(int.begin(), 0);
```

Let's take a look a few more examples of the **push_back** and **insert** functions:

```
std::vector<int> vec;
// Empty vector {}

vec.push_back(1);
// Vector has one element {1}

vec.push_back(2);
// Vector has 2 elements {1, 2}

vec.insert(vec.begin(), 0);
// Vector has 3 elements {0, 1, 2}

vec.insert(find(vec.begin(), vec.end(), 1), 4);
// Vector has 4 elements {0, 4, 1, 2}
```

As shown in the preceding code, **push_back** inserts an element at the end. Additionally, the **insert** function takes the insertion position as a parameter. It takes it in the form of an iterator. So, the **begin()** function allows us to insert an element at the beginning.

Now that we have learned about the normal insertion functions, let's take a look at some better alternatives, available for vectors, compared to the **`push_back`** and **`insert`** functions. One of the drawbacks of **`push_back`** and **`insert`** is that they first construct the element, and then either copy or move the element to its new location inside the vector's buffer. This operation can be optimized by calling a constructor for the new element at the new location itself, which can be done by the **`emplace_back`** and **`emplace`** functions. It is recommended that you use these functions instead of normal insertion functions for better performance. Since we are constructing the element in place, we just need to pass the constructor parameters, instead of the constructed value itself. Then, the function will take care of forwarding the arguments to the constructor at the appropriate location.

`std::vector` also provides **`pop_back`** and **`erase`** functions to remove elements from it. **`pop_back`** removes the last element from the vector, effectively reducing the size by one. **`erase`** has two overloads – to remove the single element provided by the iterator pointing to it, and to remove a range of elements provided by the iterator, where the range is defined by defining the first element to be removed (inclusive) and the last element to be removed (exclusive). The C++ standard doesn't require these functions to reduce the capacity of the vector. It depends entirely on the compiler implementation. **`pop_back`** doesn't require any rearranging of elements, and hence can be completed very quickly. Its complexity is O(*1*). However, **`erase`** requires the shifting of the elements, and hence takes O(*n*) time. In the following exercise, we shall see how these functions are implemented.

Now, let's take a look at the example about removing elements from a vector in different ways:

Consider a vector with 10 elements – **`{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`**:

```
vec.pop_back();
// Vector has now 9 elements {0, 1, 2, 3, 4, 5, 6, 7, 8}

vec.erase(vec.begin());
// vector has now 7 elements {1, 2, 3, 4, 5, 6, 7, 8}

vec.erase(vec.begin() + 1, vec.begin() + 4);
// Now, vector has 4 elements {1, 5, 6, 7, 8}
```

Now, let's take a look at some other useful functions:

- **`clear()`**: This function simply empties the vector by removing all of the elements.
- **`reserve(capacity)`**: This function is used to specify the capacity of the vector. If the value specified as the parameter is greater than the current capacity, it reallocates memory and the new capacity will be equal to the parameter. However, for all other cases, it will not affect the vector's capacity. This function doesn't modify the size of the vector.
- **`shrink_to_fit()`**: This function can be used to free up the extra space. After calling this function, size and capacity become equal. This function can be used when we are not expecting a further increase in the size of the vector.

Allocators for std::vector

`std::vector` resolves the drawback of **`std::array`** regarding custom allocators by allowing us to pass an allocator as a template parameter after the type of data.

To use custom allocators, we follow certain concepts and interfaces. Since a vector uses allocator functions for most of its behaviors related to memory access, we need to provide those functions as part of the allocator – **`allocate`**, **`deallocate`**, **`construct`**, and **`destroy`**. This allocator will have to take care of memory allocation, deallocation, and handling so as not to corrupt any data. For advanced applications, where relying on automatic memory management, mechanisms can be too costly, and where the application has got its own memory pool or similar resource that must be used instead of default heap memory, a customer allocator is very handy.

Therefore, **`std::vector`** is a really good alternative to **`std::array`** and provides a lot more flexibility in terms of its size, growth, and other aspects. Asymptotically, all the similar functions of an array have the same time complexity as a vector. We usually pay extra performance cost only for the extra features, which is quite reasonable. For an average case, the performance of a vector is not very far from an array. Hence, in practice, **`std::vector`** is one of the most commonly used STL containers in C++ because of its flexibility and performance.

std::forward_list

So far, we've only seen array-like structures, but, as we saw, insertion and deletion in the middle of the data structures are very inefficient operations for contiguous data structures. And that's where linked-list-like structures come into the picture. A lot of applications require frequent insertion and deletion in the middle of a data structure. For example, any browser with multiple tabs can have an extra tab added at any point in time and at any location. Similarly, any music player will have a list of songs that you can play in a loop, and you can also insert any songs in the middle. In such cases, we can use a linked-list structure for good performance. We'll see the use case of a music player in *Activity 1*, *Implementing a Song Playlist*. Now, let's explore what kind of containers C++ provides us with.

The basic structure of a linked list requires us to have a pointer and to manage memory allocation and deallocation manually using the **`new`** and **`delete`** operators. Although it is not difficult, it can lead to bugs that are difficult to trace. Hence, just like **`std::array`** provides a thin wrapper over C-style arrays, **`std::forward_list`** provides a thin wrapper over a basic linked list.

The purpose of **`std::forward_list`** is to provide some additional functionality without compromising performance compared to a basic linked list. To maintain performance, it doesn't provide functions to get the size of the list or to get any element but the first one directly. Hence, it has a function called **`front()`** to get the reference to the first element, but nothing like **`back()`** to access the last element. It does provide functions for common operations, such as insertion, deletion, reverse, and splice. These functions don't affect the memory requirements or performance over basic linked lists.

Additionally, just like **`std::vector`**, **`std::forward_list`** can also take a custom allocator as the second template parameter if required. Hence, we can easily use it for advanced applications that benefit from custom memory management.

Inserting and Deleting Elements in forward_list

`std:: forward_list` provides the **`push_front`** and **`insert_after`** functions, which can be used to insert an element in a linked list. Both of these are slightly different compared to insertion functions for vectors. **`push_front`** is useful for inserting an element at the front. Since **`forward_list`** doesn't have direct access to the last element, it doesn't provide a **`push_back`** function. For insertion at a specific location, we use **`insert_after`** instead of **`insert`**. This is because inserting an element in a linked list requires updating the next pointer of the element, after which we want to insert a new element. If we provide just the iterator, where we want to insert a new element, we can't get access to the previous element quickly, since traversing backward is not allowed in **`forward_list`**.

Since this is a pointer-based mechanism, we don't really need to shift the elements during insertion. Hence, both of the insertion functions are quite a bit faster compared to any array-based structures. Both the functions just modify the pointers to insert a new element at the intended position. This operation is not dependent on the size of the list and therefore has a time complexity of O(1). We shall take a look at the implementation of these functions in the following exercise.

Now, let's see how we can insert elements in a linked list:

```
std::forward_list<int> fwd_list = {1, 2, 3};


fwd_list.push_front(0);
// list becomes {0, 1, 2, 3}


auto it = fwd_list.begin();


fwd_list.insert_after(it, 5);
// list becomes {0, 5, 1, 2, 3}


fwd_list.insert_after(it, 6);
// list becomes {0, 6, 5, 1, 2, 3}
```

`forward_list` also provides **`emplace_front`** and **`emplace_after`**, which is similar to **`emplace`** for a vector. Both of these functions do the same thing as insertion functions, but more efficiently by avoiding extra copying and moving.

`forward_list` also has **`pop_front`** and **`erase_after`** functions for the deletion of elements. **`pop_front`**, as the name suggests, removes the first element. Since it doesn't require any shifting, the operation is quite fast in practice and has a time complexity of O(1). **`erase_after`** has two overloads – to remove a single element (by taking an iterator to its previous element), and to remove multiple elements in a range (by taking an iterator to the element before the first element of the range and another iterator to the last element).

The time complexity of the **`erase_after`** function is linear to the number of elements that are erased because the deletion of elements can't be done via deallocating just a single chunk of memory. Since all the nodes are scattered across random locations in memory, the function needs to deallocate each of them separately.

Now, let's see how we can remove the elements from the list:

```
std::forward_list<int> fwd_list = {1, 2, 3, 4, 5};


fwd_list.pop_front();
// list becomes {2, 3, 4, 5}


auto it = fwd_list.begin();


fwd_list.erase_after(it);
// list becomes {2, 4, 5}


fwd_list.erase_after(it, fwd_list.end());
// list becomes {2}
```

Let's explore what other operations we can do with **`forward_list`** in the following section.

Other Operations on forward_list

Apart from the **`erase`** functions to delete elements based on its position determined by iterators, **`forward_list`** also provides the **`remove`** and **`remove_if`** functions to remove elements based on their values. The **`remove`** function takes a single parameter – the value of the elements to be removed. It removes all the elements that match the given element based on the equality operator defined for the type of the value. Without the equality operator, the compiler doesn't allow us to call that function and throws a compilation error. Since **`remove`** only deletes the elements based on the equality operator, it is not possible to use it for deletion based on other conditions, since we can't change the equality operator after defining it once. For a conditional removal, **`forward_list`** provides the **`remove_if`** function. It takes a predicate as a parameter, which is a function taking an element of the value type as a parameter, and a Boolean as the return value. So, all the elements for which the predicate returns true are removed from the list. With the latest C++ versions, we can easily specify the predicate with lambdas as well. The following exercise should help you to understand how to implement these functions.

Exercise 3: Conditional Removal of Elements from a Linked List Using remove_if

In this exercise, we'll use the sample information of a few Indian citizens during the elections and remove ineligible citizens, based on their age, from the electoral roll. For simplicity, we'll just store the names and ages of the citizens.

We shall store the data in a linked list and remove the required elements using **`remove_if`**, which provides a way to remove elements that meet a certain condition, instead of defining the positions of the elements to be removed:

1. Let's first include the required headers and add the **`struct citizen`**:

```
#include <iostream>
#include <forward_list>

struct citizen
{
    std::string name;
    int age;
};

std::ostream& operator<<(std::ostream& os, const citizen& c)
{
    return (os << "[Name: " << c.name << ", Age: " << c.age << "]");
}
```

2. Now, let's write a **`main`** function and initialize a few citizens in a **`std::forward_list`**. We'll also make a copy of it to avoid having to initialize it again:

```
int main()
{
  std::forward_list<citizen> citizens = {{"Raj", 22}, {"Rohit", 25},
{"Rohan", 17}, {"Sachin", 16}};

  auto citizens_copy = citizens;

  std::cout << "All the citizens: ";
  for (const auto &c : citizens)
      std::cout << c << " ";
  std::cout << std::endl;
```

3. Now, let's remove all of the ineligible citizens from the list:

```
citizens.remove_if(
    [](const citizen& c)
    {
        return (c.age < 18);
    });

std::cout << "Eligible citizens for voting: ";
for(const auto& c: citizens)
    std::cout << c << " ";
std::cout << std::endl;
```

   The **remove_if** function removes all the elements for which the given predicate is true. Here, we've provided a lambda since the condition is very simple. If it were a complicated condition, we could also write a normal function that takes one parameter of the underlying type of list and returns a Boolean value.

4. Now, let's find out who'll be eligible for voting next year:

```
citizens_copy.remove_if(
    [](const citizen& c)
    {
    // Returns true if age is less than 18
        return (c.age != 17);
    });

std::cout << "Citizens that will be eligible for voting next year: ";
for(const auto& c: citizens_copy)
    std::cout << c << " ";
std::cout << std::endl;
}
```

   As you can see, we are only keeping those citizens with an age of 17.

5. Run the exercise. You should get an output like this:

```
All the citizens: [Name: Raj, Age: 22] [Name: Rohit, Age: 25] [Name:
Rohan, Age: 17] [Name: Sachin, Age: 16]
Eligible citizens for voting: [Name: Raj, Age: 22] [Name: Rohit, Age: 25]
Citizens that will be eligible for voting next year: [Name: Rohan, Age:
17]
```

The **remove_if** function has a time complexity of O(*n*) since it simply traverses the list once while removing all the elements as required. If we want to remove the elements with specific values, we can use another version of **remove**, which simply takes one parameter of the object and removes all the objects from the list matching the given value. It also requires us to implement the **==** operator for the given type.

forward_list also provides a **sort** function to sort the data. All the array-related structures can be sorted by a generic function, **std::sort(first iterator, last iterator)**. However, it can't be used by linked list-based structures because we can't access any data randomly. This also makes the iterators provided by **forward_list** different from the ones for an array or a vector. We'll take a look at this in more detail in the next section. The **sort** function that is provided as part of **forward_list** has two overloads – **sort** based on the less than operator (**<**), and **sort** based on a comparator provided as a parameter. The default **sort** function uses **std::less<value_type>** for comparison. It simply returns **true** if the first parameter is less than the second one, and hence, requires us to define the less than operator (**<**) for custom-defined types.

In addition to this, if we want to compare it based on some other parameters, we can use the parametric overload, which takes a binary predicate. Both the overloads have a linearathmic time complexity – O(*n* × *log n*). The following example demonstrates both overloads of **sort**:

```
std::forward_list<int> list1 = {23, 0, 1, -3, 34, 32};


list1.sort();

// list becomes {-3, 0, 1, 23, 32, 34}


list1.sort(std::greater<int>());

// list becomes {34, 32, 23, 1, 0, -3}
```

Here, **greater<int>** is a predicate provided in the standard itself, which is a wrapper over the greater than operator (**>**) to sort the elements into descending order, as we can see from the values of the list..

Other functions provided in **`forward_list`** are **`reverse`** and **`unique`**. The **`reverse`** function simply reverses the order of the elements, in a time duration that is linear to the number of elements present in the list, that is, with a time complexity of O(*n*). The **`unique`** function keeps only the unique elements in the list and removes all the repetitive valued functions except the first one. Since it is dependent on the equality of the elements, it has two overloads – the first takes no parameters and uses the equality operator for the value type, while the second takes a binary predicate with two parameters of the value type. The **`unique`** function was built to be linear in time complexity. Hence, it doesn't compare each element with every other element. Instead, it only compares consecutive elements for equality and removes the latter one if it is the same as the former one based on the default or custom binary predicate. Hence, to remove all of the unique elements from the list using the **`unique`** function, we need to sort the elements before calling the function. With the help of a given predicate, **`unique`** will compare all the elements with their neighboring elements and remove the latter elements if the predicate returns **`true`**.

Let's now see how we can use the **`reverse`**, **`sort`**, and **`unique`** functions for lists:

```
std::forward_list<int> list1 = {2, 53, 1, 0, 4, 10};


list1.reverse();
// list becomes {2, 53, 1, 0, 4, 10}


list1 = {0, 1, 0, 1, -1, 10, 5, 10, 5, 0};


list1.sort();
// list becomes {-1, 0, 0, 0, 1, 1, 5, 5, 10, 10}


list1.unique();
// list becomes {-1, 0, 1, 5, 10}


list1 = {0, 1, 0, 1, -1, 10, 5, 10, 5, 0};


list1.sort();
// list becomes {-1, 0, 0, 0, 1, 1, 5, 5, 10, 10}
```

The following example will remove elements if they are not greater than the previously valid element by at least 2:

```
list1.unique([](int a, int b) { return (b - a) < 2; });
// list becomes {-1, 1, 5, 10}
```

> **Note**
>
> Before calling the **unique** function, the programmer must make sure that the data is already sorted. Hence, we are calling the **sort** function right before it. The **unique** function compares the element with the previous element that has already met the condition. Additionally, it always keeps the first element of the original list. Hence, there's always an element to compare with.

In the next section, we will take a look at how the **forward_list** iterator is different from the vector/array iterators.

Iterators

As you may have noticed in some of the examples for arrays and vectors, we add numbers to the iterators. Iterators are like pointers, but they also provide a common interface for STL containers. The operations on these iterators are strictly based on the type of iterators, which is dependent on the container. Iterators for vectors and arrays are the most flexible in terms of functionality. We can access any element from the container directly, based on its position, using **operator[]** because of the contiguous nature of the data. This iterator is also known as a random access iterator. However, for **forward_list**, there is no direct way to traverse back, or even go from one node to its preceding node, without starting from the beginning. Hence, the only arithmetic operator allowed for this is increment. This iterator is also known as a forward iterator.

There are other utility functions that we can use, such as **advance**, **next**, and **prev**, depending on the type of iterators. **next** and **prev** take an iterator and a distance value, and then return the iterator pointing to the element that is at the given distance from the given iterator. This works as expected provided that the given iterator supports the operation. For example, if we try to use the **prev** function with a **forward** iterator, it will throw a compilation error, since this iterator is a forward iterator and can only move forward. The time taken by these functions depends on the type of iterators used. All of them are constant time functions for random access iterators, since addition and subtraction are constant-time operations. For the rest of the iterators, all of them are linear to the distance that needs to be traversed forward or backward. We shall use these iterators in the following exercise.

Exercise 4: Exploring Different Types of Iterators

Let's say that we have a list of the winners of the Singapore F1 Grand Prix from the last few years. With the help of vector iterators, we'll discover how we can retrieve useful information from this data. After that, we'll try to do the same thing with **`forward_list`**, and see how it differs from vector iterators:

1. Let's first include the headers:

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

int main()
{
```

2. Let's write a vector with a list of winners:

```
std::vector<std::string> vec = {"Lewis Hamilton", "Lewis Hamilton", "Nico
Roseberg", "Sebastian Vettel", "Lewis Hamilton", "Sebastian Vettel",
"Sebastian Vettel", "Sebastian Vettel", "Fernando Alonso"};

auto it = vec.begin();       // Constant time
std::cout << "Latest winner is: " << *it << std::endl;

it += 8;                    // Constant time
std::cout << "Winner before 8 years was: " << *it << std::endl;

advance(it, -3);            // Constant time
std::cout << "Winner before 3 years of that was: " << *it << std::endl;
```

3. Let's try the same with the **`forward_list`** iterators and see how they differ from vector iterators:

```
std::forward_list<std::string> fwd(vec.begin(), vec.end());

auto it1 = fwd.begin();
std::cout << "Latest winner is: " << *it << std::endl;

advance(it1, 5);   // Time taken is proportional to the number of elements

std::cout << "Winner before 5 years was: " << *it << std::endl;
```

```
    // Going back will result in compile time error as forward_list only
    allows us to move towards the end.

    // advance(it1, -2);      // Compiler error
    }
```

4. Running this exercise should produce the following output:

```
Latest winner is : Lewis Hamilton
Winner before 8 years was : Fernando Alonso
Winner before 3 years of that was : Sebastian Vettel
Latest winner is : Sebastian Vettel
Winner before 5 years was : Sebastian Vettel
```

5. Now, let's see what happens if we add a number to this iterator by putting the following line inside the **`main`** function at the end:

```
it1 += 2;
```

   We'll get an error message similar to this:

```
no match for 'operator+=' (operand types are std::_Fwd_list_iterator<int>'
and 'int')
```

The various iterators we have explored in this exercise are quite useful for easily fetching any data from your dataset.

As we have seen, **`std::array`** is a thin wrapper over a C-style array, and **`std::forward_list`** is nothing but a thin wrapper over a singly linked list. It provides a simple and less error-prone interface without compromising on performance or memory.

Apart from that, since we can access any element immediately in the vector, the addition and subtraction operations on the vector iterator are $O(1)$. On the other hand, **`forward_list`** only supports access to an element by traversing to it. Hence, its iterators' addition operation is $O(n)$, where n is the number of steps we are advancing.

In the following exercise, we shall make a custom container that works in a similar way to **`std::forward_list`**, but with some improvements. We shall define many functions that are equivalent to **`forward_list`** functions. It should also help you understand how these functions work under the hood.

Exercise 5: Building a Basic Custom Container

In this exercise, we're going to implement an **std::forward_list** equivalent container with some improvements. We'll start with a basic implementation called **singly_ll**, and gradually keep on improving:

1. Let's add the required headers and then start with the basic implementation of **singly_ll** with a single node:

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

struct singly_ll_node
{
    int data;
    singly_ll_node* next;
};
```

2. Now, we'll implement the actual **singly_ll** class, which wraps the node around for better interfacing:

```
class singly_ll
{
public:
    using node = singly_ll_node;
    using node_ptr = node*;

private:
    node_ptr head;
```

3. Now, let's add **push_front** and **pop_front**, just like in **forward_list**:

```
public:
void push_front(int val)
{
    auto new_node = new node{val, NULL};
    if(head != NULL)
        new_node->next = head;
    head = new_node;
}

void pop_front()
{
    auto first = head;
    if(head)
```

```
        {
            head = head->next;
            delete first;
        }
        else
            throw "Empty ";
    }
```

4. Let's now implement a basic iterator for our **`singly_ll`** class, with constructors and accessors:

```
    struct singly_ll_iterator
    {
    private:
        node_ptr ptr;
    public:
        singly_ll_iterator(node_ptr p) : ptr(p)
        {
    }

    int& operator*()
    {
        return ptr->data;
    }

    node_ptr get()
    {
        return ptr;
    }
```

5. Let's add the **`operator++`** functions for pre- and post-increments:

```
    singly_ll_iterator& operator++()     // pre-increment
    {
            ptr = ptr->next;
            return *this;
    }

    singly_ll_iterator operator++(int)    // post-increment
    {
        singly_ll_iterator result = *this;
    ++(*this);
    return result;
    }
```

6. Let's add equality operations as **friend** functions:

```
    friend bool operator==(const singly_ll_iterator& left, const singly_
ll_iterator& right)
    {
        return left.ptr == right.ptr;
    }

    friend bool operator!=(const singly_ll_iterator& left, const singly_
ll_iterator& right)
    {
        return left.ptr != right.ptr;
    }
};
```

7. Let's jump back to our linked list class. Now that we've got our iterator class, let's implement the **begin** and **end** functions to ease the traversal. We'll also add **const** versions for both:

```
singly_ll_iterator begin()
{
    return singly_ll_iterator(head);
}

singly_ll_iterator end()
{
    return singly_ll_iterator(NULL);
}

singly_ll_iterator begin() const
{
    return singly_ll_iterator(head);
}

singly_ll_iterator end() const
{
    return singly_ll_iterator(NULL);
}
```

8. Let's implement a default constructor, a copy constructor for deep copying, and a constructor with **initializer_list**:

```
singly_ll() = default;

singly_ll(const singly_ll& other) : head(NULL)
{
    if(other.head)
        {
            head = new node;
            auto cur = head;
            auto it = other.begin();
            while(true)
            {
                cur->data = *it;

                auto tmp = it;
                ++tmp;
                if(tmp == other.end())
                    break;

                cur->next = new node;
                cur = cur->next;
                it = tmp;
            }
        }
}

singly_ll(const std::initializer_list<int>& ilist) : head(NULL)
{
    for(auto it = std::rbegin(ilist); it != std::rend(ilist); it++)
            push_front(*it);
}
};
```

9. Let's write a **main** function to use the preceding functions:

```
int main()
{
    singly_ll sll = {1, 2, 3};
    sll.push_front(0);

    std::cout << "First list: ";
    for(auto i: sll)
        std::cout << i << " ";
    std::cout << std::endl;

    auto sll2 = sll;
    sll2.push_front(-1);
    std::cout << "Second list after copying from first list and inserting
-1 in front: ";
    for(auto i: sll2)
        std::cout << i << ' ';  // Prints -1 0 1 2 3
    std::cout << std::endl;

    std::cout << "First list after copying - deep copy: ";
for(auto i: sll)
        std::cout << i << ' ';  // Prints 0 1 2 3
    std::cout << std::endl;
}
```

10. Running this exercise should produce the following output:

```
First list: 0 1 2 3
Second list after copying from first list and inserting -1 in front: -1 0 1
2 3
First list after copying - deep copy: 0 1 2 3
```

As we can see in the preceding example, we are able to initialize our list using **std::initializer_list**. We can call the **push**, **pop_front**, and **back** functions. As we can see, **sll2.pop_back** only removed the element from **sll2**, and not **sll**. **sll** is still intact with all five elements. Hence, we can perform a deep copy as well.

Activity 1: Implementing a Song Playlist

In this activity, we'll look at some applications for which a doubly-linked list is not enough or not convenient. We will build a tweaked version that fits the application. We often encounter cases where we have to customize default implementations, such as when looping songs in a music player or in games where multiple players take a turn one by one in a circle.

These applications have one common property – we traverse the elements of the sequence in a circular fashion. Thus, the node after the last node will be the first node while traversing the list. This is called a circular linked list.

We'll take the use case of a music player. It should have following functions supported:

1. Create a playlist using multiple songs.
2. Add songs to the playlist.
3. Remove a song from the playlist.
4. Play songs in a loop (for this activity, we will print all the songs once).

> **Note**
>
> You can refer to *Exercise 5*, *Building a Basic Custom Container* where we built a container from scratch supporting similar functions.

Here are the steps to solve the problem:

1. First, design a basic structure that supports circular data representation.
2. After that, implement the **`insert`** and **`erase`** functions in the structure to support various operations.
3. We have to write a custom iterator. This is a bit tricky. The important thing is to make sure that we are able to traverse the container using a range-based approach for a loop. Hence, **`begin()`** and **`end()`** should return different addresses, although the structure is circular.
4. After building the container, build a wrapper over it, which will store different songs in the playlist and perform relevant operations, such as **`next`**, **`previous`**, **`print all`**, **`insert`**, and **`remove`**.

> **Note**
>
> The solution to this activity can be found on page 476.

std::forward_list has several limitations. **std::list** presents a much more flexible implementation of lists and helps overcome some of the shortcomings of **forward_list**.

std::list

As seen in the previous section, **std::forward_list** is just a nice and thin wrapper over the basic linked list. It doesn't provide functions to insert elements at the end, traverse backward, or get the size of the list, among other useful operations. The functionality is limited to save memory and to retain fast performance. Apart from that, the iterators of **forward_list** can support very few operations. In most practical situations in any application, functions such as those for inserting something at the end and getting the size of the container are very useful and frequently used. Hence, **std::forward_list** is not always the desired container, where fast insertion is required. To overcome these limitations of **std::forward_list**, C++ provides **std::list**, which has several additional features owing to the fact that it is a bidirectional linked list, also known as a doubly-linked list. However, note that this comes at the cost of additional memory requirements.

The plain version of a doubly-linked list looks something like this:

```
struct doubly_linked_list
{
    int data;
    doubly_linked_list *next, *prev;
};
```

As you can see, it has one extra pointer to point to the previous element. Thus, it provides us with a way in which to traverse backward, and we can also store the size and the last element to support fast **push_back** and **size** operations. Also, just like **forward_list**, it can also support customer allocator as a template parameter.

Common Functions for std::list

Most of the functions for **std::list** are either the same or similar to the functions of **std::forward_list**, with a few tweaks. One of the tweaks is that function names ending with **_after** have their equivalents without **_after**. Therefore, **insert_after** and **emplace_after** become simply **insert** and **emplace**. This is because, with the **std::list** iterator, we can also traverse backward, and hence there's no need to provide the iterator of the previous element. Instead, we can provide the iterator of the exact element at which we want to perform the operation. Apart from that, **std::list** also provides fast operations for **push_back**, **emplace_back**, and **pop_back**. The following exercise demonstrates the use of insertion and deletion functions for **std::list**.

Exercise 6: Insertion and Deletion Functions for std::list

In this exercise, we shall create a simple list of integers using **`std::list`** and explore various ways in which we can insert and delete elements from it:

1. First of all, let's include the required headers:

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

int main()
{
```

2. Then, initialize a list with a few elements and experiment on it with various insertion functions:

```
std::list<int> list1 = {1, 2, 3, 4, 5};

list1.push_back(6);
// list becomes {1, 2, 3, 4, 5, 6}

list1.insert(next(list1.begin()), 0);
// list becomes {1, 0, 2, 3, 4, 5, 6}

list1.insert(list1.end(), 7);
// list becomes {1, 0, 2, 3, 4, 5, 6, 7}
```

As you can see, the **`push_back`** function inserts an element at the end. The **`insert`** function inserts **`0`** after the first element, which is indicated by **`next(list1.begin())`**. After that, we are inserting **`7`** after the last element, which is indicated by **`list1.end()`**.

3. Now, let's take a look at the remove function, **`pop_back`**, which was not present in **`forward_list`**:

```
list1.pop_back();
// list becomes {1, 0, 2, 3, 4, 5, 6}

std::cout << "List after insertion & deletion functions: ";
for(auto i: list1)
    std::cout << i << " ";
}
```

4. Running this exercise should give the following output:

   ```
   List after insertion & deletion functions: 1 0 2 3 4 5 6
   ```

   Here, we are removing the last element that we just inserted.

> **Note**
>
> Although **push_front**, **insert**, **pop_front**, and **erase** have the same time complexity as equivalent functions for **forward_list**, these are slightly more expensive for **std::list**. The reason for this is that there are two pointers in each node of a list instead of just one, as in the case of **forward_list**. So, we have to maintain the validity of the value of both the pointers. Hence, when we are repointing these variables, we need to make almost double the effort compared to singly linked lists.

Earlier, we saw an insertion for a singly-linked list. Let's now demonstrate what pointer manipulation looks like for a doubly-linked list in the following diagram:

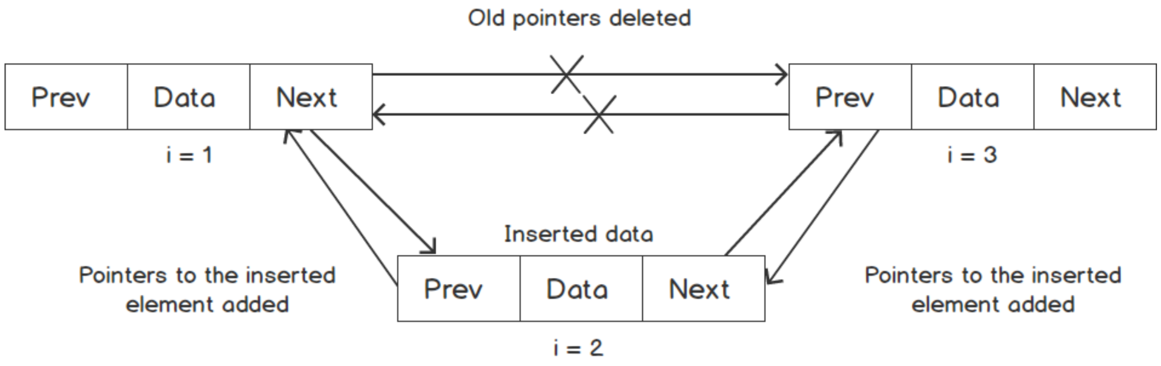


Figure 1.7: Inserting an element in a doubly linked list

As you can see, the number of operations is constant even in the case of **std::list**; however, compared to **forward_list**, we have to fix both the **prev** and **next** pointers in order to maintain a doubly-linked list, and this costs us almost double in terms of memory and performance compared to **forward_list**. A similar idea applies to other functions as well.

Other functions such as **remove**, **remove_if**, **sort**, **unique**, and **reverse** provide similar functionalities as compared to their equivalent functions for **std::forward_list**.

Bidirectional Iterators

In the *Iterators* section, we saw the difference between the flexibility of array-based random access iterators and **`forward_list`**-based forward iterators. The flexibility of **`std::list::iterator`** lies between both of them. It is more flexible compared to forward iterators, since it can allow us to traverse backward. Hence, **`std::list`** also supports functions for reverse traversal by exposing reverse iterators where the operations are inverted. Having said that, it is not as flexible as random access iterators. Although we can advance in either direction by any number of moves, since these moves have to be done by traversing the elements one by one instead of jumping directly to the desired element, the time complexity is still linear, and not a constant, as in the case of random access iterators. Since these iterators can move in either direction, they are known as bidirectional iterators.

Iterator Invalidation for Different Containers

So far, we've seen that iterators provide us with a uniform way of accessing, traversing, inserting, and deleting elements from any container. But there are some cases when iterators become invalid after modifying the container, because the iterators are implemented based on pointers, which are bound to memory addresses. So, if the memory address of any node or element changes because of modification in the container, it invalidates the iterator, and using it regardless can lead to undefined behavior.

For example, a very basic example would be **`vector::push_back`**, which simply adds a new element at the end. However, as we saw earlier, in some cases, it also requires the movement of all the elements to a new buffer. Hence, all iterators, pointers, and even the references to any of the existing elements will be invalidated. Similarly, if the **`vector::insert`** function leads to reallocation, all the elements will need to be moved. Hence, all the iterators, pointers, and references are invalidated. If not, the function will invalidate all the iterators pointing to the element that is on the right side of the insertion position, since these elements will be shifted during the process.

Unlike vectors, linked list-based iterators are safer for insertion and deletion operations because the elements will not be shifted or moved. Hence, none of the insertion functions for **`std::list`** or **`forward_list`** affect the validity of the iterators. An exception is that deletion-related operations invalidate iterators of the elements that are deleted, which is obvious and reasonable. It doesn't affect the validity of the iterators of the rest of the elements. The following example shows iterator invalidation for different iterators:

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


auto it4 = vec.begin() + 4;
```

```
// it4 now points to vec[4]


vec.insert(vec.begin() + 2, 0);
// vec becomes {1, 2, 0, 3, 4, 5}
```

it4 is invalid now, since it comes after the insertion position. Accessing it will lead to undefined behavior:

```
std::list<int> lst = {1, 2, 3, 4, 5};


auto l_it4 = next(lst.begin(), 4);


lst.insert(next(lst.begin(), 2), 0);
// l_it4 remains valid
```

As we saw, **std::list** is much more flexible compared to **std::forward_list**. A lot of operations, such as **size**, **push_back**, and **pop_back**, are provided, which operate with a time complexity of O(1). Hence, **std::list** is used more frequently compared to **std::forward_list**. **forward_list** is a better alternative if we have very strict constraints of memory and performance, and if we are sure that we don't want to traverse backward. So, in most cases, **std::list** is a safer choice.

Activity 2: Simulating a Card Game

In this activity, we'll analyze a given situation and try to come up with the most suitable data structure to achieve the best performance.

We'll try to simulate a card game. There are 4 players in the game, and each starts with 13 random cards. Then, we'll try to pick one card from each player's hand randomly. That way, we'll have 4 cards for comparison. After that, we'll remove the matching cards from those 4 cards. The remaining cards, if any, will be drawn back by the players who put them out. If there are multiple matching pairs out of which only one can be removed, we can choose either one. If there are no matching pairs, players can shuffle their own set of cards.

Now, we need to continue this process over and over until at least one of them is out of cards. The first one to get rid of all their cards wins the game. Then, we shall print the winner at the end.

Perform the following steps to solve the activity:

1. First, determine which container would be the most suitable to store the cards of each player. We should have four containers that have a set of cards – one for each player.
2. Write a function to initialize and shuffle the cards.
3. Write a function to randomly deal all the cards among the four players.
4. Write a matching function. This function will pick a card from each player and compare it as required by the rules of the game. Then, it will remove the necessary cards. We have to choose the card wisely so that removing it would be faster. This parameter should also be considered while deciding on the container.
5. Now, let's write a function, to see whether we have a winner.
6. Finally, we'll write the core logic of the game. This will simply call the matching function until we have a winner based on the function written in the previous step.

> **Note**
>
> The solution to this activity can be found on page 482.

std::deque – Special Version of std::vector

So far, we have seen array-based and linked list-based containers. **`std::deque`** mixes both of them and combines each of their advantages to a certain extent. As we have seen, although vector is a variable-length array, some of its functions, such as **`push_front`** and **`pop_front`**, are very costly operations. **`std::deque`** can help us overcome that. Deque is short for double-ended queue.

The Structure of Deque

The C++ standard only defines the behavior of the containers and not the implementation. The containers we have seen so far are simple enough for us to predict their implementation. However, deque is slightly more complicated than that. Therefore, we'll first take a look at its requirements, and then we will try to dive into a little bit of implementation.

The C++ standard guarantees the following time complexities for different operations of deque:

- O(1) for **`push_front`**, **`pop_front`**, **`push_back`**, and **`pop_back`**
- O(1) for random access to all the elements
- Maximum of N/2 steps in the case of insertion or deletion in the middle, where N = the size of the deque

Looking at the requirements, we can say that the container should be able to grow in either direction very fast, and still be able to provide random access to all the elements. Thus, the structure has to be somewhat like a vector, but still expandable from the front as well as the back. The requirement for insertion and deletion gives a slight hint that we will be shifting the elements because we are only allowed to take up to N/2 steps. And that also validates our previous assumption regarding behavior that is similar to vector. Since the container can grow in either direction quickly, we don't necessarily have to shift the elements toward the right every time. Instead, we can shift the elements toward the nearest end. That will give us a time complexity of a maximum of N/2 steps, since the nearest end can't be more than N/2 nodes away from any insertion point inside the container.

Now, let's focus on random access and insertion at the front. The structure can't be stored in a single chunk of memory. Rather, we can have multiple chunks of memory of the same size. In this way, based on the index and size of the chunks (or the number of elements per chunk), we can decide which chunk's indexed element we want. That helps us to achieve random access in O(1) time only if we store pointers to all the memory chunks in a contiguous location. Hence, the structure can be assumed to be similar to a vector of arrays.

When we want to insert something at the front, and we don't have enough space in the first memory chunk, we have to allocate another chunk and insert its address in the vector of pointers at the front. That might require reallocation of the vector of pointers, but the actual data will not be moved. To optimize that reallocation, instead of starting from the first chunk, we can start the insertion from the middle chunk of the vector. In that way, we are safe up to a certain number of front insertions. We can follow the same while reallocating the vector of pointers.

> **Note**
>
> Since the deque is not as simple as the other containers discussed in this chapter, the actual implementation might differ or might have a lot more optimizations than we discussed, but the basic idea remains the same. And that is, we need multiple chunks of contiguous memory to implement such a container.

The functions and operations supported by deque are more of a combination of functions supported by vectors and lists; hence, we have **`push_front`**, **`push_back`**, **`insert`**, **`emplace_front`**, **`emplace_back`**, **`emplace`**, **`pop_front`**, **`pop_back`**, and **`erase`**, among others. We also have the vector's functions, such as **`shrink_to_fit`**, to optimize the capacity, but we don't have a function called **`capacity`** since this is highly dependent on the implementation, and is, therefore, not expected to be exposed. And, as you might expect, it provides random access iterators just like a vector.

Let's take a look at how we can use different insertion and deletion operations on deque:

```
std::deque<int> deq = {1, 2, 3, 4, 5};


deq.push_front(0);

// deque becomes {0, 1, 2, 3, 4, 5}


deq.push_back(6);

// deque becomes {0, 1, 2, 3, 4, 5, 6}


deq.insert(deq.begin() + 2, 10);

// deque becomes {0, 1, 10, 2, 3, 4, 5, 6}


deq.pop_back();

// deque becomes {0, 1, 10, 2, 3, 4, 5}


deq.pop_front();

// deque becomes {1, 10, 2, 3, 4, 5}


deq.erase(deq.begin() + 1);

// deque becomes {1, 2, 3, 4, 5}


deq.erase(deq.begin() + 3, deq.end());

// deque becomes {1, 2, 3}
```

Such a structure may be used in cases such as boarding queues for flights.

The only thing that differs among the containers is the performance and memory requirements. Deque will provide very good performance for both insertion and deletion at the front as well as the end. Insertion and deletion in the middle is also a bit faster than for a vector on average, although, asymptotically, it is the same as that of a vector.

Apart from that, deque also allows us to have customer allocators just like a vector. We can specify it as a second template parameter while initializing it. One thing to note here is that the allocator is part of the type and not part of the object. This means we can't compare two objects of two deques or two vectors where each has a different kind of allocator. Similarly, we can't have other operations, such as an assignment or copy constructor, with objects of different types of allocators.

As we saw, **`std::deque`** has a slightly more complex structure compared to other containers we examined before that. It is, in fact, the only container that provides efficient random access along with fast **`push_front`** and **`push_back`** functions. Deque is used as an underlying container for others, as we'll see in the upcoming section.

Container Adaptors

The containers that we've seen until now are built from scratch. In this section, we'll look at the containers that are built on top of other containers. There are multiple reasons to provide a wrapper over existing containers, such as providing more semantic meaning to the code, restricting someone from accidentally using unintended functions just because they are available, and to provide specific interfaces.

One such specific use case is the **stack** data structure. The stack follows the **LIFO** (**Last In First Out**) structure for accessing and processing data. In terms of functions, it can insert and delete only at one end of the container and can't update or even access any element except at the mutating end. This end is called the top of the stack. We can easily use any other container, such as a vector or deque too, since it can meet these requirements by default. However, there are some fundamental problems in doing that.

The following example shows two implementations of the stack:

```
std::deque<int> stk;


stk.push_back(1);  // Pushes 1 on the stack = {1}


stk.push_back(2);  // Pushes 2 on the stack = {1, 2}
stk.pop_back();    // Pops the top element off the stack = {1}

stk.push_front(0); // This operation should not be allowed for a stack

std::stack<int> stk;

stk.push(1);       // Pushes 1 on the stack = {1}

stk.push(2);       // Pushes 2 on the stack = {1, 2}

stk.pop();         // Pops the top element off the stack = {1}

stk.push_front(0); // Compilation error
```

As we can see in this example, the first block of the stack using deque provides a semantic meaning only by the name of the variable. The functions operating on the data still don't force the programmer to add code that shouldn't be allowed, such as **`push_front`**. Also, the **`push_back`** and **`pop_back`** functions expose unnecessary details, which should be known by default since it is a stack.

In comparison to this, if we look at the second version, it looks much more accurate in indicating what it does. And, most importantly, it doesn't allow anyone to do anything that was unintended, even accidentally.

The second version of the stack is nothing but a wrapper over the previous container, deque, by providing a nice and restricted interface to the user. This is called a container adaptor. There are three container adaptors provided by C++: **`std::stack`**, **`std::queue`**, and **`std::priority_queue`**. Let's now take a brief look at each of them.

std::stack

As explained earlier, adaptors simply reuse other containers, such as deque, vector, or any other container for that matter. **`std::stack`**, by default, adapts **`std::deque`** as its underlying container. It provides an interface that is only relevant to the stack – **`empty`**, **`size`**, **`top`**, **`push`**, **`pop`**, and **`emplace`**. Here, **`push`** simply calls the **`push_back`** function for the underlying container, and **`pop`** simply calls the **`pop_back`** function. **`top`** calls the **`back`** function from the underlying container to get the last element, which is the top of the stack. Thus, it restricts the user operations to LIFO since it only allows us to update values at one end of the underlying container.

Here, we are using deque as an underlying container, and not a vector. The reason behind it is that deque doesn't require you to shift all the elements during reallocation, unlike vector. Hence, it is more efficient to use deque compared to vector. However, if, for some scenario, any other container is more likely to give better performance, stack gives us the facility to provide a container as a template parameter. So, we can build a stack using a vector or list as well, as shown here:

```
std::stack<int, std::vector<int>> stk;
std::stack<int, std::list<int>> stk;
```

All the operations of a stack have a time complexity of O(1). There is usually no overhead of forwarding the call to the underlying container as everything can be inlined by the compiler with optimizations.

std::queue

Just like **std::stack**, we have another container adapter to deal with the frequent scenario of **FIFO** (**First In First Out**) in many applications, and this structure is provided by an adaptor called **std::queue**. It almost has the same set of functions as a stack, but the meaning and behavior are different in order to follow FIFO instead of LIFO. For **std::queue**, **push** means **push_back**, just like a stack, but **pop** is **pop_front**. Instead of **pop**, since queue should be exposing both the ends for reading, it has **front** and **back** functions.

Here's a small example of the usage of **std::queue**:

```
std::queue<int> q;

q.push(1);  // queue becomes {1}

q.push(2);  // queue becomes {1, 2}

q.push(3);  // queue becomes {1, 2, 3}

q.pop();    // queue becomes {2, 3}

q.push(4);  // queue becomes {2, 3, 4}
```

As shown in this example, first, we are inserting **1**, **2**, and **3** in that order. After that, we are popping one element off the queue. Since **1** was pushed first, it is removed from the queue first. Then, the next push inserts **4** at the back of the queue.

`std::queue` also uses **`std::deque`** as an underlying container for the same reason as stack, and it also has a time complexity of O(1) for all the methods shown here.

std::priority_queue

Priority queue provides a very useful structure called **heap** via its interface. A heap data structure is known for fast access to the minimum (or maximum) element from the container. Getting the min/max element is an operation with a time complexity of O(1). Insertion has O(*log n*) time complexity, while deletion can only be performed for the min/max element, which always stays on the top.

An important thing to note here is that we can only have either the min or max function made available quickly, and not both of them. This is decided by the comparator provided to the container. Unlike stack and queue, a priority queue is based on a vector by default, but we can change it if required. Also, by default, the comparator is **`std::less`**. Since this is a heap, the resultant container is a max heap. This means that the maximum element will be on top by default.

Here, since insertion needs to make sure that we can access the top element (min or max depending on the comparator) instantly, it is not simply forwarding the call to the underlying container. Instead, it is implementing the algorithm for heapifying the data by bubbling it up to the top as required using the comparator. This operation takes a time duration that is logarithmic in proportion to the size of the container, hence the time complexity of O(*log n*). The invariant also needs to be maintained while initializing it with multiple elements. Here, however, the **`priority_queue`** constructor does not simply call the insertion function for each element; instead, it applies different heapification algorithms to do it faster in O(*n*).

Iterators for Adaptors

All the adaptors that we have seen so far expose functionality only as required to fulfill its semantic meaning. Logically thinking, traversing through stack, queue, and priority queue doesn't make sense. At any point, we should only be able to see the front element. Hence, STL doesn't provide iterators for that.

Benchmarking

As we have seen that different containers have a variety of pros and cons, no one container is the perfect choice for every situation. Sometimes, multiple containers may give a similar performance on average for the given scenario. In such cases, benchmarking is our friend. This is a process of determining the better approach based on statistical data.

Consider a scenario where we want to store data in contiguous memory, access it, and operate on it using various functions. We can say that we should either use **`std::vector`** or **`std::deque`**. But we are not sure which among these will be the best. At first glance, both of them seem to give good performance for the situation. Among different operations, such as access, insertion, **`push_back`**, and modifying a specific element, some are in favor of **`std::vector`** and some of are in favor of **`std::deque`**. So, how should we proceed?

The idea is to create a small prototype of the actual model and implement it using both **`std::vector`** and **`std::deque`**. And then, measure the performance of both over the prototype. Based on the result of the performance testing, we can choose the one that gives better results overall.

The simplest way to do that is to measure the time required to perform different operations for both and compare them. However, the same operation may take different amounts of time during different runs, since there are other factors that come into the picture, such as OS scheduling, cache, and interrupts, among others. These parameters can cause our results to deviate quite heavily, because, to perform any operation once, is a matter of a few hundred nanoseconds. To overcome that, we can perform the operation multiple times (by that, we mean a few million times) until we get a considerable time difference between both the measurements.

There are some benchmarking tools that we can use, such as quick-bench.com, which provide us with an easy way to run benchmarks. You can try running the operations mentioned earlier on vector and deque to quickly compare the performance differences.

Activity 3: Simulating a Queue for a Shared Printer in an Office

In this activity, we'll simulate a queue for a shared printer in an office. In any corporate office, usually, the printer is shared across the whole floor in the printer room. All the computers in this room are connected to the same printer. But a printer can do only one printing job at any point in time, and it also takes some time to complete any job. In the meantime, some other user can send another print request. In such a case, a printer needs to store all the pending jobs somewhere so that it can take them up once its current task is done.

Perform the following steps to solve the activity:

1. Create a class called **`Job`** (comprising an ID for the job, the name of the user who submitted it, and the number of pages).
2. Create a class called **`Printer`**. This will provide an interface to add new jobs and process all the jobs added so far.

3. To implement the **`printer`** class, it will need to store all the pending jobs. We'll implement a very basic strategy – first come, first served. Whoever submits the job first will be the first to get the job done.
4. Finally, simulate a scenario where multiple people are adding jobs to the printer, and the printer is processing them one by one.

> **Note**
>
> The solution to this activity can be found on page 487.

Summary

In this chapter, we learned how we should go about designing an application based on its requirements by choosing the way we want to store the data. We explained different types of operations that we can perform on data, which can be used as parameters for comparison between multiple data structures, based on the frequency of those operations. We learned that container adaptors provide a very useful way to indicate our intentions in the code. We saw that using more restrictive containers provided as adaptors, instead of using primary containers providing more functionality, is more effective in terms of maintainability, and also reduces human errors. We explained various data structures – **`std::array`**, **`std::vector`**, **`std::list`**, and **`std::forward_list`**, which are very frequent in any application development process, in detail and their interfaces provided by C++ by default. This helps us to write efficient code without reinventing the whole cycle and making the process a lot faster.

In this chapter, all the structures we saw are linear in a logical manner, that is, we can either go forward or backward from any element. In the next chapter, we'll explore problems that can't be solved easily with these structures and implement new types of structures to solve those problems.