4
Data Structures

In the last chapter, we discussed how to analyze time and memory complexity and how to measure performance. In this chapter, we are going to talk about how to choose and use data structures from the standard library. To understand why certain data structures work very well on the computers of today, we first need to cover some basics about computer memory. In this chapter, you will learn about:

- The properties of computer memory
- The standard library containers: sequence containers and associative containers
- The standard library container adaptors
- Parallel arrays

Before we start walking through the containers offered by the standard library and some other useful data structures, we will briefly discuss some properties of computer memory.

The properties of computer memory

C++ treats memory as a sequence of cells. The size of each cell is 1 byte, and each cell has an address. Accessing a byte in memory by its address is a constant-time operation, *O(1)*, in other words, it's independent of the total number of memory cells. On a 32-bit machine, you can theoretically address 2^{32} bytes,

that is, around 4 GB, which restricts the amount of memory a process is allowed to use at once. On a 64-bit machine, you can theoretically address 2^{64} bytes, which is so big that there is hardly any risk of running out of addresses.

The following figure shows a sequence of memory cells laid out in memory. Each cell contains 8 bits. The hexadecimal numbers are the addresses of the memory cells:

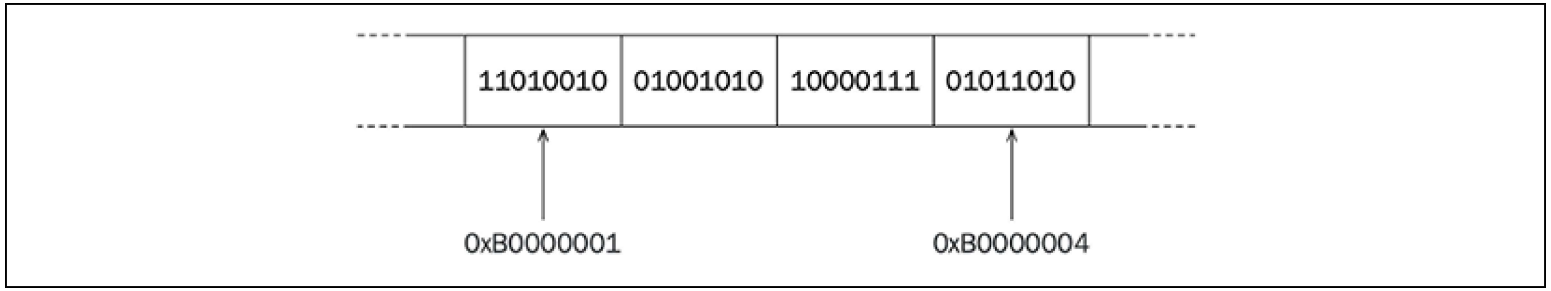

Figure 4.1: A sequence of memory cells

Since accessing a byte by its address is an *O(1)* operation, from a programmer's perspective, it's tempting to believe that each memory cell is equally quick to access. This approach to memory is simple and useful in many cases, but when choosing data structures for efficient use, you need to take into account the memory hierarchy that exists in modern computers. The importance of the memory hierarchy has increased, since the time it takes to read and write from the main memory has become more expensive when compared to the speed of today's processors. The following figure shows the architecture of a machine with one CPU and four cores:

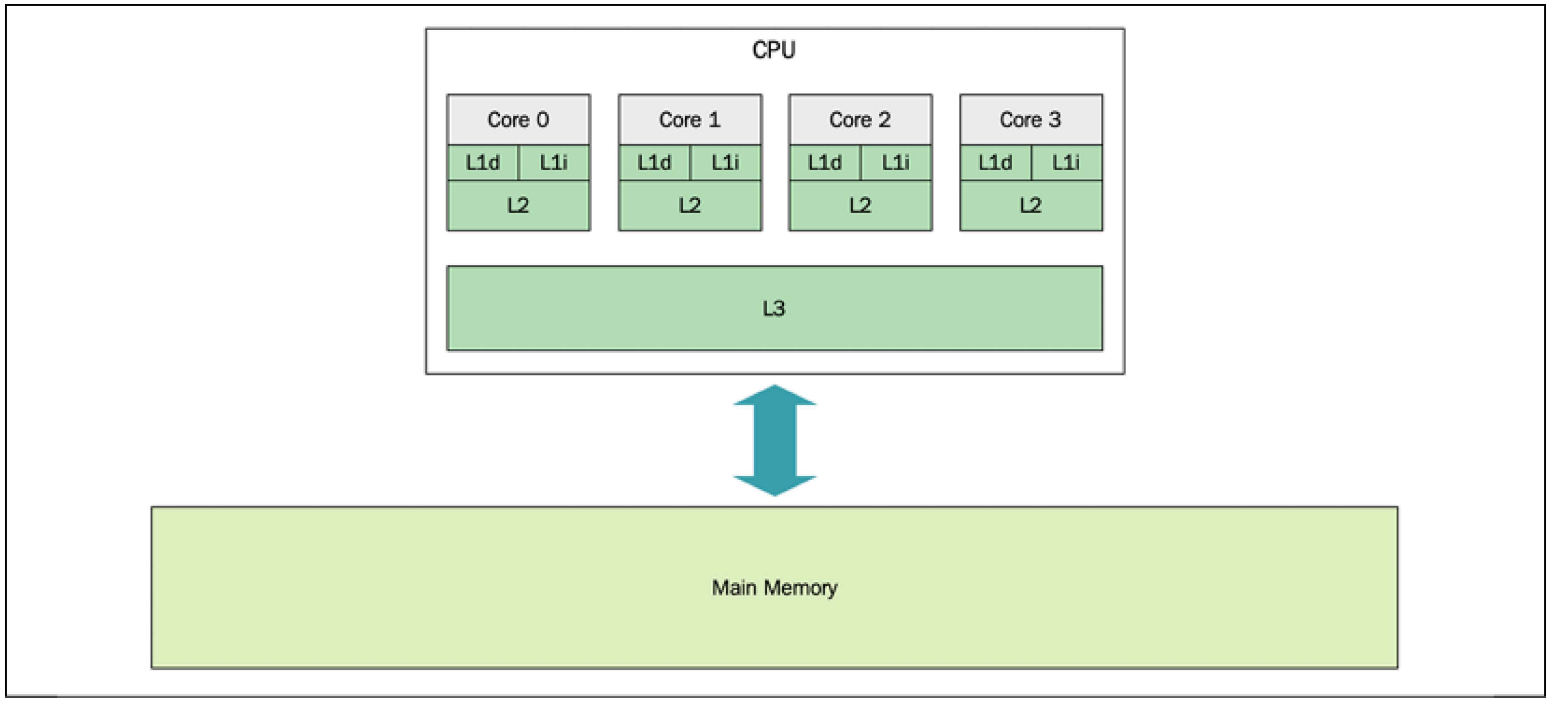

Figure 4.2: An example of a processor with four cores; the boxes labeled L1i, L1d, L2, and L3 are memory caches

I'm currently writing this chapter on a MacBook Pro from 2018, which is equipped with an Intel Quad-Core i7 CPU. On this processor, each core has its own L1 and L2 caches, whereas the L3 cache is shared among all four cores. Running the following command from a terminal:

```
sysctl -a hw
```

gives me, among other things, the following information:

```
hw.memsize: 17179869184
hw.cachelinesize: 64
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 262144
hw.l3cachesize: 8388608
```

The reported `hw.memsize` is the total amount of main memory, which is 16 GB in this case.

The `hw.cachelinesize`, which is reported to be 64 bytes, is the size of the cache lines, also known as blocks. When accessing a byte in memory, the machine is not only fetching the byte that is asked for; instead, the machine always fetches a cache line, which, in this case, is 64 bytes. The various caches between the CPU and main memory keep track of 64-byte blocks, instead of individual bytes.

The `hw.l1icachesize` is the size of the L1 instruction cache. This is a 32 KB cache dedicated to storing instructions that have been recently used by the CPU. The `hw.l1dcachesize` is also 32 KB and is dedicated to data, as opposed to instructions.

Lastly, we can read the size of the L2 cache and the L3 cache, which is 256 KB and 8 MB, respectively. An important observation is that the caches are tiny compared to the amount of main memory available.

Without presenting any detailed facts about the actual number of cycles required to access data from each layer in the cache hierarchy, a very rough guideline is that there are orders of magnitude differences of latency between two adjacent layers (for example, L1 and L2). The following table shows an extract from the latency numbers presented in an article by Peter Norvig called *Teach yourself programming in ten*

years, 2001 (**http://norvig.com/21-days.html**). The full table is usually referred to as *Latency numbers every programmer should know* and is credited to Jeff Dean:

L1 cache reference	0.5 ns
L2 cache reference	7 ns
Main memory reference	100 ns

Structuring the data in such a way that the caches can be fully utilized can have a dramatic effect on performance. Accessing data that has recently been used and, therefore, potentially already resides in the cache will make your program faster. This is known as **temporal locality**.

Also, accessing data located near some other data you are using will increase the likelihood that the data you need is already in a cache line that was fetched from the main memory earlier. This is known as **spatial locality**.

Constantly wiping out the cache lines in inner loops might result in very bad performance. This is sometimes called **cache thrashing**. Let's look at an example:

```
constexpr auto kL1CacheCapacity = 32768; // The L1 Data cache size
constexpr auto kSize = kL1CacheCapacity / sizeof(int);
using MatrixType = std::array<std::array<int, kSize>, kSize>;
auto cache_thrashing(MatrixType& matrix) {
  auto counter = 0;
  for (auto i = 0; i < kSize; ++i) {
```

```
    for (auto j = 0; j < kSize; ++j) {
      matrix[i][j] = counter++;
    }
  }
}
```

This version takes about 40 ms to run on my computer. However, by only changing the line in the inner loop to the following, the time it takes to complete the function increases from 40 ms to over 800 ms:

```
matrix[j][i] = counter++;
```

In the first example, when using `matrix[i][j]`, most of the time we will access memory that is already in the L1 cache, whereas, in the modified version using `matrix[j][i]`, every access will generate an L1 cache miss. A few images might help you to understand what's going on. Instead of drawing the full 32768 x 32768 matrix, a tiny 3 x 3 matrix, as shown here, will serve as an example:

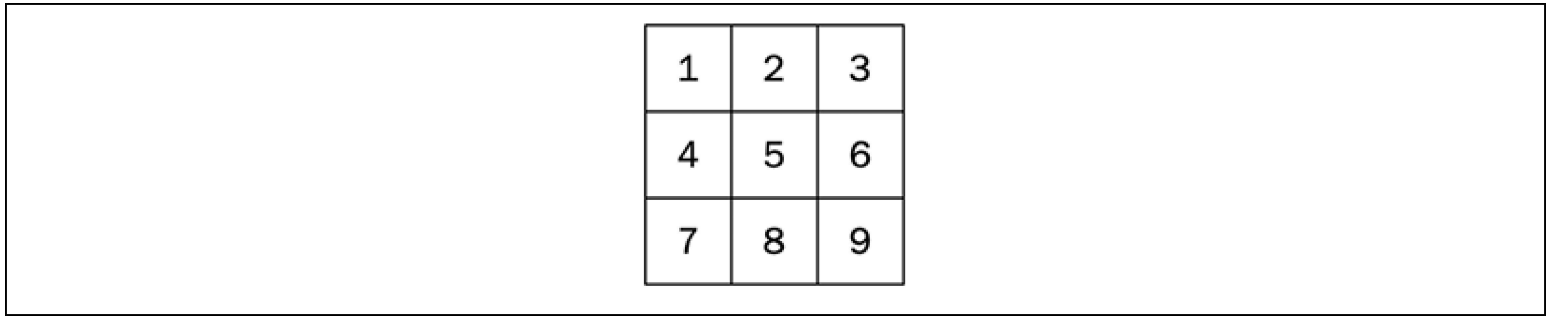

Figure 4.3: A 3x3 matrix

Even if this might be our mental image of how a matrix resides in memory, there is no such thing as 2-dimensional memory. Instead, when this matrix is laid out in a 1-dimensional memory space, it looks like this:

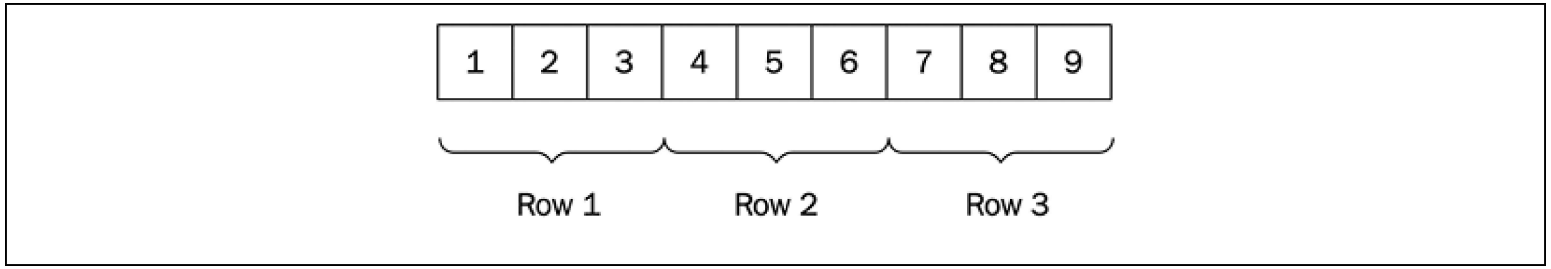

Figure 4.4: A 2-dimensional matrix in a 1-dimensional memory space

That is, it's a contiguous array of elements laid out row by row. In the fast version of our algorithm, the numbers are accessed sequentially in the same order in which they are contiguously laid out in memory, like this:

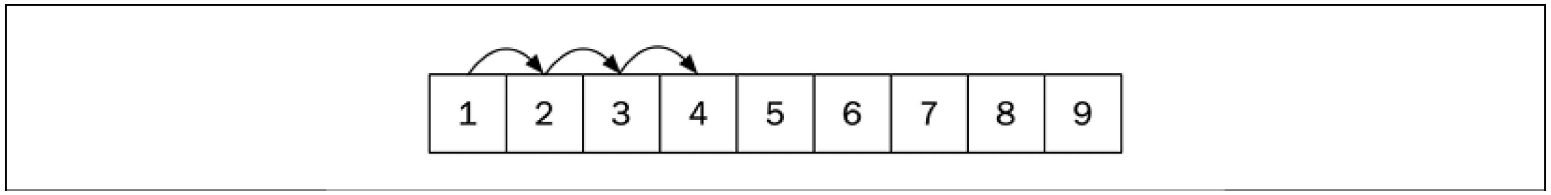

Figure 4.5: Fast sequential stride-1 accesses

Whereas in the slow version of the algorithm, the elements are accessed in a completely different pattern. Accessing the first four elements using the slow version would now look like this instead:

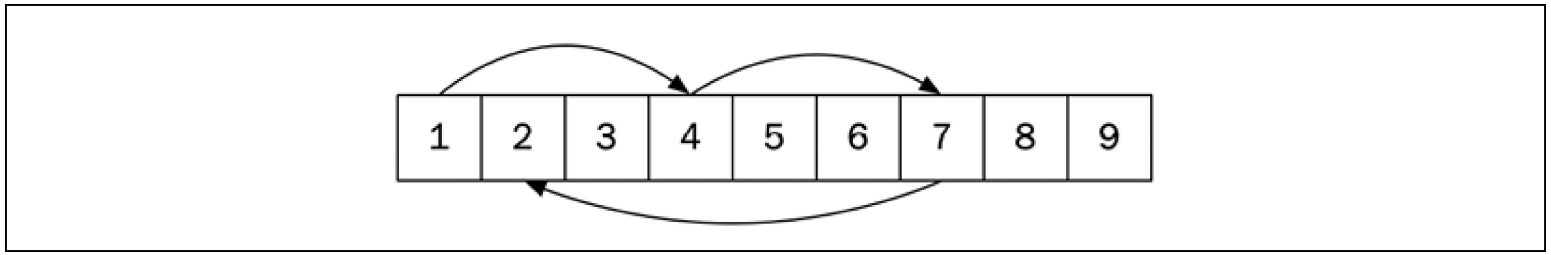

Figure 4.6: Slow access using a larger stride

Accessing data in this way is substantially slower due to poor spatial locality. Modern processors are usually also equipped with a **prefetcher**, which can automatically recognize memory access patterns and try to prefetch data from memory into the caches that are likely to be accessed in the near future. Prefetchers tend to perform best for smaller strides. You can read a lot more about this in the excellent book, *Computer Systems, A Programmer's Perspective*, by Randal E. Bryant and David R. O'Hallaron.

To summarize this section, even if memory accesses are constant-time operations, caching can have dramatic effects on the actual time it takes to access the memory. This is something to always bear in mind when using or implementing new data structures.

Next, I will introduce a set of data structures from the C++ standard library, called containers.

The standard library containers

The C++ standard library offers a set of very useful container types. A container is a data structure that contains a collection of elements. The container manages the memory of the elements it holds. This means that we don't have to explicitly create and delete the objects that we put in a container. We can

pass objects created on the stack to a container and the container will copy and store them on the free store.

Iterators are used to access elements in containers, and are, therefore, a fundamental concept for understanding algorithms and data structures from the standard library. The iterator concept is covered in *Chapter 5*, *Algorithms*. For this chapter, it's enough to know that an iterator can be thought of as a pointer to an element and that iterators have different operators defined depending on the container they belong to. For example, array-like data structures provide random access iterators to their elements. These iterators support arithmetic expressions using `+` and `-`, whereas an iterator to a linked list, for example, only supports `++` and `--` operators.

The containers are divided into three categories: sequence containers, associative containers, and container adaptors. This section will contain a brief introduction to the containers in each of the three categories and also address the most important things to consider when performance is an issue.

Sequence containers

Sequence containers keep the elements in the order we specify when adding the elements to the container. The sequence containers from the standard library are `std::array`, `std::vector`, `std::deque`, `std::list`, and `std::forward_list`. I will also address `std::basic_string` in this section, although it's not formally a generic sequence container because it only handles elements of character types.

We should know the answers to the following questions before choosing a sequence container:

1. What is the number of elements (order of magnitude)?
2. What are the usage patterns? How often are you going to add data? Read/traverse data? Delete data? Rearrange data?

3. Where in the sequence are you going to add data most often? At the end, at the beginning, or in the middle of the sequence?
4. Do you need to sort the elements? Or do you even care about the order?

Depending on the answers to these questions, we can determine which sequence containers are more or less suitable for our needs. But, to do that, we need a basic understanding of the interfaces and performance characteristics of each type of sequence container.

The sections that follow will briefly present the different sequence containers, starting with one of the most widely used containers overall.

Vectors and arrays

`std::vector` is probably the most commonly used container type, and for good reason. A vector is an array that grows dynamically when needed. The elements added to a vector are guaranteed to be laid out contiguously in memory, which means that you can access any element in the array by its index in constant time. It also means that it provides excellent performance when traversing the elements in the order they are laid out, thanks to the spatial locality mentioned earlier.

A vector has a **size** and a **capacity**. The size is the number of elements that are currently held in the container, and the capacity is the number of elements that the vector can hold until it needs to allocate more space:

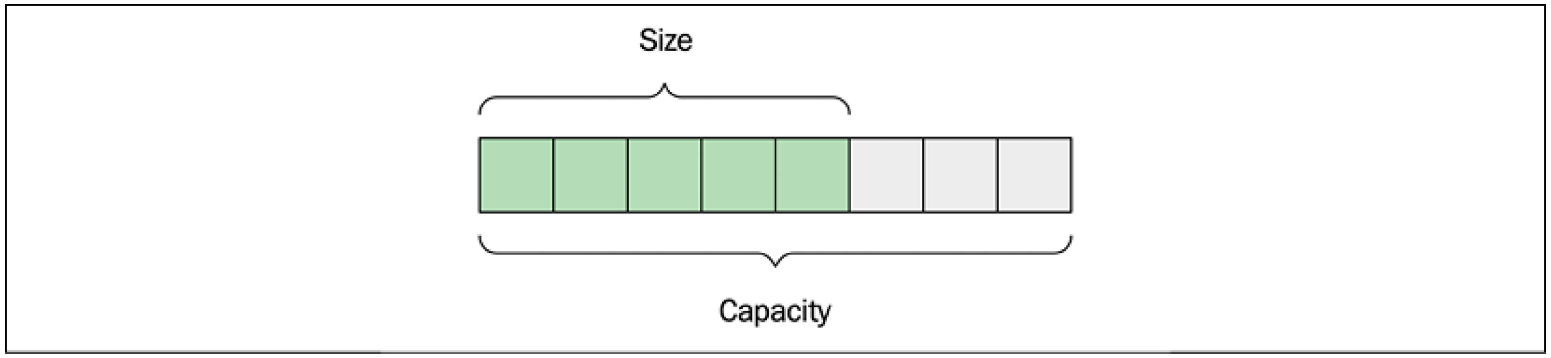

Figure 4.7: Size and capacity of a std::vector

Adding elements to the end of a vector using the `push_back()` function is fast, as long as the size is less than the capacity. When adding an element and there is no room for more, the vector will allocate a new internal buffer and then move all of the elements to the new space. The capacity will grow in such a way that resizing the buffer rarely happens, thus making `push_back()` an amortized constant-time operation, as we discussed in *Chapter 3*, *Analyzing and Measuring Performance*.

A vector template instance of type `std::vector<Person>` will store `Person` objects by value. When the vector needs to rearrange the `Person` objects (for example, as a result of an insert), the values will be copy constructed or moved. Objects will be moved if they have a `nothrow` move constructor. Otherwise, the objects will be copy constructed in order to guarantee strong exception safety:

```
Person(Person&& other) {       // Will be copied
  // ...
}
Person(Person&& other) noexcept { // Will be moved
```

```
    // ...
}
```

Internally, `std::vector` uses `std::move_if_noexcept` in order to determine whether the object should be copied or moved. The `<type_traits>` header can help you to verify at compile time that your classes are guaranteed to not throw an exception when moved:

```
static_assert(std::is_nothrow_move_constructible<Person>::value);
```

If you are adding newly created objects to the vector, you can take advantage of the `emplace_back()` function, which will create the object in place for you, instead of creating an object and then copying/moving it to the vector using the `push_back()` function:

```
persons.emplace_back("John", 65);
```

The capacity of the vector can change in the following ways:

- By adding an element to the vector when the `capacity == size`
- By calling `reserve()`
- By calling `shrink_to_fit()`

Other than that, the vector will not change the capacity, and hence will not allocate or deallocate dynamic memory. For example, the member function `clear()` empties a vector, but it does not change its capacity. These memory guarantees make the vector usable even in real-time contexts.

Since C++20, there are also two free functions that erase elements from a `std::vector`. Prior to C++20, we had to use the *erase-remove idiom*, which we will discuss in *Chapter 5*, *Algorithms*. However, now the recommended way to erase elements from a `std::vector` is by using `std::erase()` and `std::erase_if()`. Here is a short example of how to use these functions:

```
auto v = std::vector{-1, 5, 2, -3, 4, -5, 5};
std::erase(v, 5);                       // v: [-1,2,-3,4,-5]
std::erase_if(v, [](auto x) { return x < 0; }); // v: [2, 4]
```

As an alternative to the dynamically sized vector, the standard library also provides a fixed size version named `std::array` that manages its elements by using the stack as opposed to the free store. The size of the array is a template argument specified at compile time, which means that the size and type elements become a part of the concrete type:

```
auto a = std::array<int, 16>{};
auto b = std::array<int, 1024>{};
```

In this example, `a` and `b` are not the same type, which means that you have to specify the size when using the type as a function parameter:

```
auto f(const std::array<int, 1024>& input) {
  // ...
}

f(a);  // Does not compile, f requires an int array of size 1024
```

This might seem a bit tedious at first, but this is, in fact, the big advantage over the built-in array type (the C arrays), which loses the size information when passed to a function, since it automatically converts a pointer into the first element of the array:

```
// input looks like an array, but is in fact a pointer
auto f(const int input[]) {
  // ...
}

int a[16];
int b[1024];
f(a); // Compiles, but unsafe
```

An array losing its size information is usually referred to as **array decay**. You will see later on in this chapter how array decay can be avoided by using `std::span` when passing contiguous data to functions.

Deque

Sometimes, you'll find yourself in a situation where you need to frequently add elements to both the beginning and end of a sequence. If you are using a `std::vector` and need to speed up the inserts at the front, you can instead use `std::deque`, which is short for **double-ended queue**. `std::deque` is usually implemented as a collection of fixed-size arrays, which makes it possible to access elements by their index in constant time. However, as you can see in the following figure, all of the elements are not stored contiguously in memory, which is the case with `std::vector` and `std::array`.

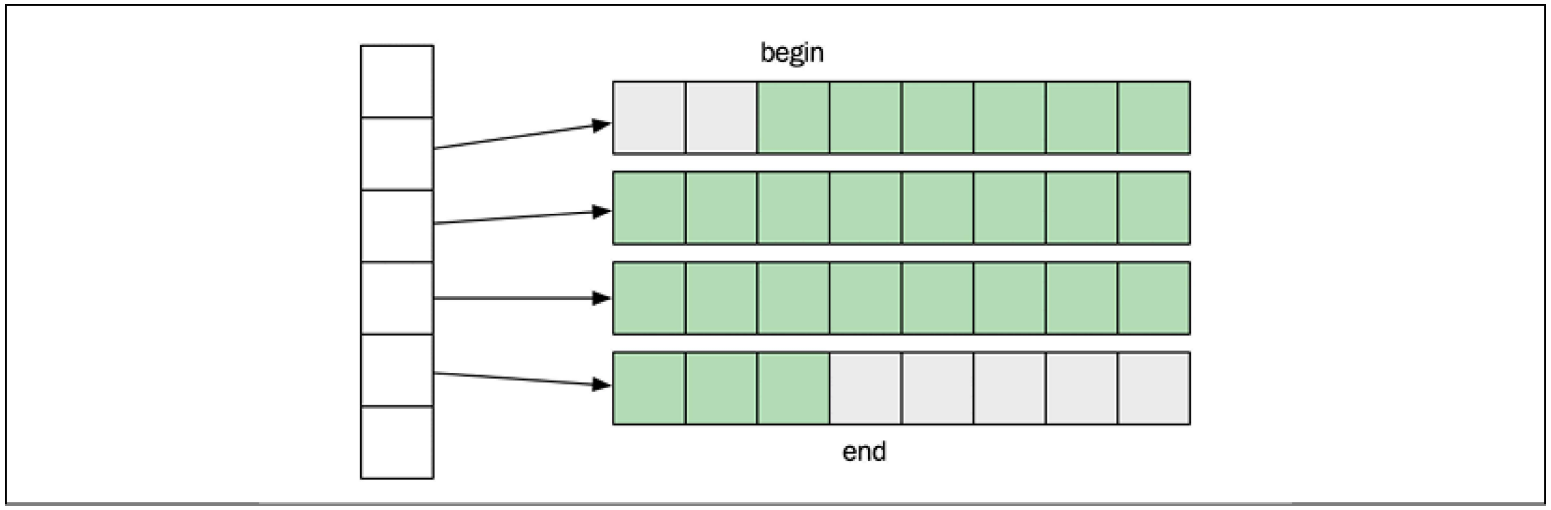

Figure 4.8: A possible layout of std::deque

List and forward_list

The `std::list` is a **doubly linked list**, meaning that each element has one link to the next element and one link to its previous element. This makes it possible to iterate over the list both backward and forward. There is also a **singly linked list** named `std::forward_list`. The reason you wouldn't always choose the doubly linked list over `std::forward_list` is because of the excessive memory that is occupied by the back pointers in the doubly linked list. So, if you don't need to traverse the list backward, use `std::forward_list`. Another interesting feature of the forward list is that it's optimized for very short lists. When the list is empty, it only occupies one word, which makes it a viable data structure for sparse data.

Note that even if the elements are ordered in a sequence, they are *not* laid out contiguously in memory like the vector and array are, which means that iterating a linked list will most likely generate a lot more cache misses compared to the vector.

To recap, the `std::list` is a doubly linked list with pointers to the next and previous elements:

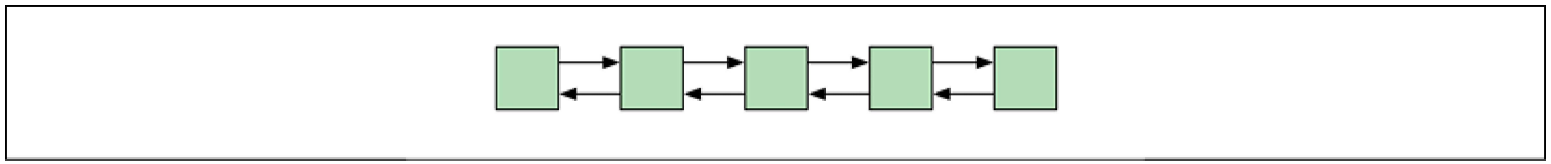

Figure 4.9: std::list is a doubly linked list

The `std::forward_list` is a singly linked list with pointers to the next element:

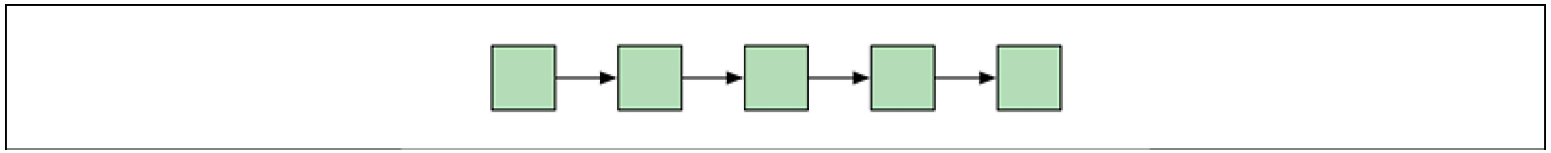

Figure 4.10: std::forward_list is a singly linked list

The `std::forward_list` is more memory efficient since it only has one pointer to the next element.

Lists are also the only containers that support **splicing**, which is a way to transfer elements between lists without copying or moving the elements. This means, for example, that it is possible to concatenate two lists into one in constant time, *O(1)*. Other containers require at least linear time for such operations.

The basic_string

The last template class that we will cover in this section is the `std::basic_string`. The `std::string` is a `typedef` for `std::basic_string<char>`. Historically, `std::basic_string` was not guaranteed to be laid out

contiguously in memory. This changed with C++17, which makes it possible to pass the string to APIs that require an array of characters. For example, the following code reads an entire file into a string:

```
auto in = std::ifstream{"file.txt", std::ios::binary | std::ios::ate};
if (in.is_open()) {
  auto size = in.tellg();
  auto content = std::string(size, '\0');
  in.seekg(0);
  in.read(&content[0], size);
  // "content" now contains the entire file
}
```

By opening the file using `std::ios::ate`, the position indicator is set to the end of the stream so that we can use `tellg()` to retrieve the size of the file. After that, we set the input position to the beginning of the stream and start reading.

Most implementations of `std::basic_string` utilize something called **small object optimization**, which means that they do not allocate any dynamic memory if the size of the string is small. We will discuss small object optimization later in the book. For now, let's move on to discuss associative containers.

Associative containers

The associative containers place their elements based on the element itself. For example, it's not possible to add an element at the back or front in an associative container as we do with `std::vector::push_back()` or `std::list::push_front()`. Instead, the elements are added in a way that makes it possible to find the ele-

ment without the need to scan the entire container. Therefore, the associative containers have some requirements for the objects we want to store in a container. We will look at these requirements later.

There are two main categories of associative containers:

- **Ordered associative containers**: These containers are based on trees; the containers use a tree for storing their elements. They require that the elements are ordered by the less than operator (`<`). The functions for adding, deleting, and finding elements are all O(log n) in the tree-based containers. The containers are named `std::set` , `std::map` , `std::multiset` , and `std::multimap` .
- **Unordered associative containers**: These containers are based on hash tables; the containers use a hash table for storing their elements. They require that the elements are compared with the equality operator (`==`) and that there is a way to compute a hash value based on an element. More on that later. The functions for adding, deleting, and finding elements are all *O(1)* in the hash table-based containers. The containers are named `std::unordered_set` , `std::unordered_map` , `std::unordered_multiset` , and `std::unordered_multimap` .

Since C++20, all associative containers are equipped with a function named `contains()` , which should be used when you want to know whether a container contains some specific elements. In earlier versions of C++, it was necessary to use `count()` or `find()` to find out whether a container contained an element.

Always use the specialized functions, such as `contains()` and `empty()` , instead of using `count() > 0` or `size() == 0` . The specialized functions are guaranteed to be the most efficient ones.

Ordered sets and maps

The ordered associative containers guarantee that insert, delete, and search can be done in logarithmic time, *O(log n)*. How that is achieved is up to the implementation of the standard library. However, the implementations we know about do use some kind of self-balancing binary search tree. The fact that the tree stays approximately balanced is necessary for controlling the height of the tree, and, hence, also the worst-case running time when accessing elements. There is no need for the tree to pre-allocate memory, so, typically, a tree will allocate memory on the free store each time an element is inserted and also free up memory whenever elements are erased. Take a look at the following diagram, which shows that the height of a balanced tree is *O(log n)*:

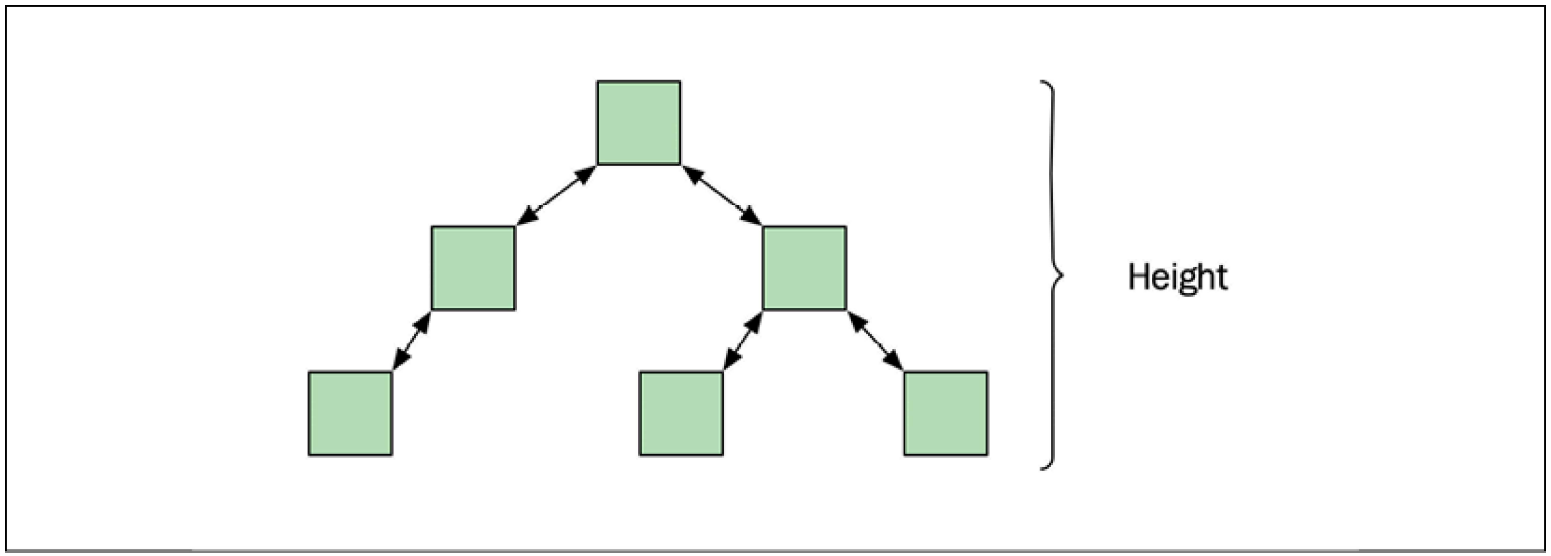

Figure 4.11: The height of the tree is O(log n) if it's balanced

Unordered sets and maps

The unordered versions of sets and maps offer a hash-based alternative to the tree-based versions. This data structure is, in general, referred to as hash tables. In theory, hash tables offer amortized constant-

time insert, add, and delete operations, which can be compared to the tree-based versions that operate in *O(log n)*. However, in practice, the difference might not be so obvious, especially if you are not storing a very large number of elements in your container.

Let's see how a hash table can offer *O(1)* operations. A hash table keeps its elements in some sort of array of buckets. When adding an element to the hash table, an integer is computed for the element using a hash function. The integer is usually called the **hash** of the element. The hash value is then limited to the size of the array (by using the modulo operation, for example) so that the new limited value can be used as an index in the array. Once the index is computed, the hash table can store the element in the array at that index. The lookup of an element works in a similar manner by first computing a hash value for the element we are looking for and then accessing the array.

Apart from computing the hash value, this technique seems straightforward. This is just half of the story, though. What if two different elements generate the same index, either because they produced the same hash value, or because two different hash values are being limited to the same index? When two non-equal elements end up at the same index, we call that a **hash collision**. This is not just an edge case: this will happen a lot, even if we are using a good hash function, and especially if the array is small when compared to the number of elements we are adding. There are various ways of dealing with hash collisions. Here, we will focus on the one that is being used in the standard library, which is called **separate chaining**.

Separate chaining solves the problem of two unequal elements ending up at the same index. Instead of just storing the elements directly in the array, the array is a sequence of **buckets**. Each bucket can contain multiple elements, that is, all of the elements that are hashed to the same index. So, each bucket is also some sort of container. The exact data structure used for the buckets is not defined, and it can vary for

different implementations. However, we can think of it as a linked list and assume that finding an element in a specific bucket is slow, since it needs to scan the elements in the buckets linearly.

The following figure shows a hash table with eight buckets. The elements have landed in three separate buckets. The bucket with index **2** contains four elements, the bucket with index **4** contains two elements, and the bucket with index **5** contains only one element. The other buckets are empty:

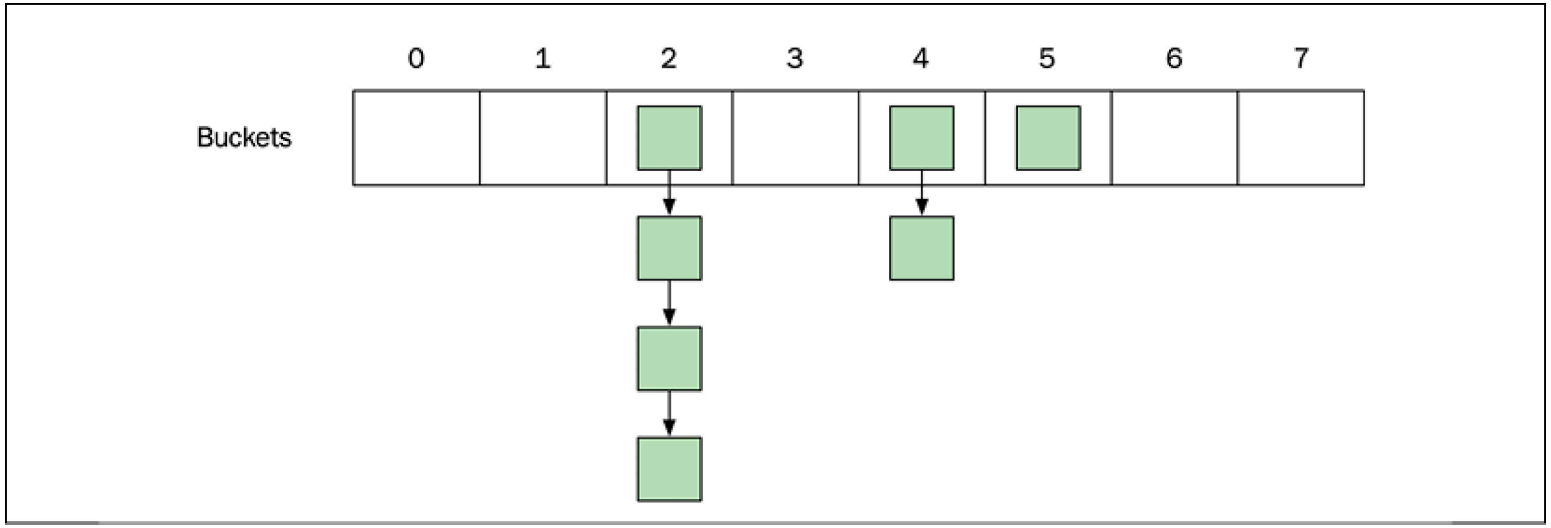

Figure 4.12: Each bucket contains 0 or more elements

Hash and equals

The hash value, which can be computed in constant time with respect to the size of the container, determines which bucket an element will be placed in. Since it's possible that more than one object will generate the same hash value, and therefore end up in the same bucket, each key also needs to provide an equals function, which is used to compare the key we are looking for with all of the keys in the bucket.

If two keys are equal, they are required to generate the same hash value. However, it's perfectly legal for two objects to return the same hash value while not being equal to each other.

A good hash function is quick to compute and will also distribute the keys evenly among the buckets in order to minimize the number of elements in each bucket.

The following is an example of a *very bad*, but valid, hash function:

```
auto my_hash = [](const Person& person) {
  return 47; // Bad, don't do this!
};
```

It is valid because it will return the same hash value for two objects that are equal. The hash function is also very quick. However, since all elements will produce the same hash value, all keys will end up in the same bucket, which means finding an element will be *O(n)* instead of *O(1)*, which we are aiming for.

A good hash function, on the other hand, ensures that the elements are distributed evenly among the buckets to minimize hash collisions. The C++ standard actually has a note about this, which states that it should be very rare for a hash function to produce the same hash value for two different objects. Fortunately, the standard library already provides us with good hash functions for basic types. In many cases, we can reuse these functions when writing our own hash functions for user-defined types.

Suppose we want to use a `Person` class as a key in an `unorordered_set`. The `Person` class has two data members: `age`, which is an `int`, and `name`, which is a `std::string`. We start by writing the equal predicate:

```
auto person_eq = [](const Person& lhs, const Person& rhs) {
  return lhs.name() == rhs.name() && lhs.age() == rhs.age();
};
```

For two `Person` objects to be equal, they need to have the same name and the same age. We can now define the hash predicate by combining the hash values of all of the data members that are included in the equals predicate. Unfortunately, there is no function in the C++ standard yet to combine hash values, but there is a good one available in Boost, which we will use here:

```
#include <boost/functional/hash.hpp>
auto person_hash = [](const Person& person) {
  auto seed = size_t{0};
  boost::hash_combine(seed, person.name());
  boost::hash_combine(seed, person.age());
  return seed;
};
```

If, for some reason, you cannot use Boost, `boost::hash_combine()` is really just a one-liner that can be copied from the documentation found at **https://www.boost.org/doc/libs/1_55_0/doc/html/hash/reference.html#boost.hash_combine**.

With the equality and hash functions defined, we can finally create our `unordered_set`:

```
using Set = std::unordered_set<Person, decltype(person_hash),                    decltype(person_eq)>;
auto persons = Set{100, person_hash, person_eq};
```

A good rule of thumb is to always use all of the data members that are being used in the equal function when producing the hash value. That way, we adhere to the contract between equals and hash, and, at the same time, this enables us to provide an effective hash value. For example, it would be correct but inefficient to only use the name when computing the hash value, since that would mean that all `Person` objects with the same name would end up in the same bucket. Even worse, though, would be to include data members in the hash function that are not being used in the equals function. This would most likely result in a disaster where you cannot find objects in your `unordered_set` that, in fact, compare equally.

Hash policy

Apart from creating hash values that distribute the keys evenly among the buckets, we can reduce the number of collisions by having many buckets. The average number of elements per bucket is called the **load factor**. In the preceding example, we created an `unordered_set` with 100 buckets. If we add 50 `Person` objects to the set, `load_factor()` would return 0.5. The `max_load_factor` is an upper limit of the load factor, and when that value is reached, the set will need to increase the number of buckets, and, as a consequence, also rehash all the elements that are currently in the set. It's also possible to trigger a rehash manually with the `rehash()` and `reserve()` member functions.

Let's move on to look at the third category: container adaptors.

Container adaptors

There are three container adaptors in the standard library: `std::stack`, `std::queue`, and `std::priority_queue`. Container adaptors are quite different from sequence containers and associative containers because they represent **abstract data types** that can be implemented by the underlying sequence container. For example, the stack, which is a **last in, first out** (**LIFO**) data structure supporting push and pop on the top of the stack, can be implemented by using a `vector`, `list`, `deque`, or any other

custom sequence container that supports `back()`, `push_back()`, and `pop_back()`. The same goes for `queue`, which is a **first in, first out** (**FIFO**) data structure, and `priortiy_queue`.

In this section, we will focus on `std::priority_queue`, which is a pretty useful data structure that is easy to forget.

Priority queues

A **priority queue** offers a constant-time lookup of the element with the highest priority. The priority is defined using the less than operator of the elements. Insert and delete both run in logarithmic time. A priority queue is a partially ordered data structure, and it might not be obvious when to use one instead of a completely sorted data structure, for example, a tree or a sorted vector. However, in some cases, a priority queue can offer you the functionality you need, and for a lower cost than a completely sorted container.

The standard library already provides a partial sort algorithm, so we don't need to write our own. But let's look at how we can implement a partial sort algorithm using a priority queue. Suppose that we are writing a program for searching documents given a query. The matching documents (search hits) should be ordered by rank, and we are only interested in finding the first 10 search hits with the highest rank.

A document is represented by the following class:

```
class Document {
public:
  Document(std::string title) : title_{std::move(title)} {}
private:
  std::string title_;
```

```
  // ...
};
```

When searching, an algorithm selects the documents that match the query and computes a rank of the search hits. Each matching document is represented by a `Hit`:

```
struct Hit {
  float rank_{};
  std::shared_ptr<Document> document_;
};
```

Finally, we need to sort the hits and return the top *m* documents. What are the options for sorting the hits? If the hits are contained in a container that provides random access iterators, we could use `std::sort()` and only return the *m* first elements. Or, if the total number of hits is much larger than the *m* documents we are to return, we could use `std::partial_sort()`, which would be more efficient than `std::sort()`.

But what if we don't have random access iterators? Maybe the matching algorithm only provides forward iterators to the hits. In that case, we could use a priority queue and still come up with an efficient solution. Our sort interface would look like this:

```
template<typename It>
auto sort_hits(It begin, It end, size_t m) -> std::vector<Hit> {
```

We could call this function with any iterator that has the increment operator defined. Next, we create a `std::priority_queue` backed by a `std::vector`, using a custom compare function for keeping the *lowest* ranking hits at the top of the queue:

```
auto cmp = [](const Hit& a, const Hit& b) {
 return a.rank_ > b.rank_; // Note, we are using greater than
};
auto queue = std::priority_queue<Hit, std::vector<Hit>,                    decltype(cmp)>{cmp};
```

We will only insert, at most, *m* elements in the priority queue. The priority queue will contain the highest-ranking hits seen so far. Among the elements that are currently in the priority queue, the hit with the lowest rank will be the topmost element:

```
for (auto it = begin; it != end; ++it) {
 if (queue.size() < m) {
  queue.push(*it);
 }
 else if (it->rank_ > queue.top().rank_) {
  queue.pop();
  queue.push(*it);
 }
}
```

Now, we have collected the highest-ranking hits in the priority queue, so the only thing left to do is to put them in a vector in reverse order and return the *m*-sorted hits:

```
  auto result = std::vector<Hit>{};
  while (!queue.empty()) {
    result.push_back(queue.top());
    queue.pop();
  }
  std::reverse(result.begin(), result.end());
  return result;
} // end of sort_hits()
```

What is the complexity of this algorithm? If we denote the number of hits with n and the number of returned hits with *m*, we can see that the memory consumption is *O(m)*, whereas the time complexity is *O(n * log m)*, since we are iterating over *n* elements. Additionally, in each iteration, we might have to do a push and/or pop, which both run in *O(log m)* time.

We will now leave the standard library containers and focus on a couple of new useful class templates that are closely related to standard containers.

Using views

In this section, we will discuss some relatively new class templates in the C++ standard library: `std::string_view` from C++17 and `std::span`, which was introduced in C++20.

These class templates are not containers but lightweight views (or slices) of a sequence of contiguous elements. Views are small objects that are meant to be copied by value. They don't allocate memory, nor do they provide any guarantees regarding the lifetime of the memory they point to. In other words, they are non-owning reference types, which differ significantly from the containers described previously in

this chapter. At the same time, they are closely related to `std::string`, `std::array`, and `std::vector`, which we will look at soon. I will start by describing `std::string_view`.

Avoiding copies with string_view

A `std::string_view` contains a pointer to the beginning of an immutable string buffer and a size. Since a string is a contiguous sequence of characters, the pointer and the size fully define a valid substring range. Typically, a `std::string_view` points to some memory that is owned by a `std::string`. But it could also point to a string literal with static storage duration or something like a memory-mapped file. The following diagram shows a `std::string_view` pointing at memory owned by a `std::string`:

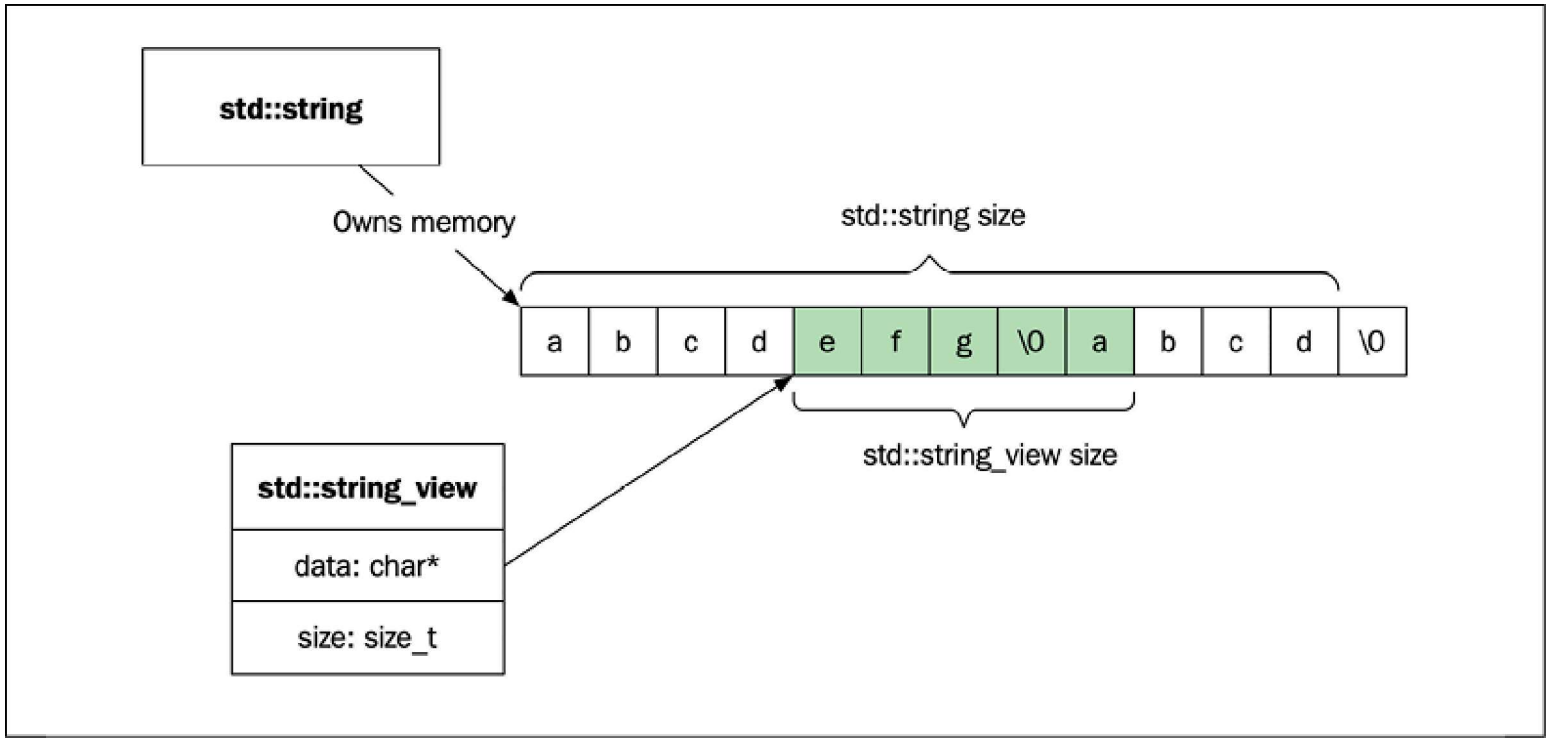

Figure 4.13: A std::string_view object pointing at memory owned by an instance of std::string

The character sequence defined by a `std::string_view` is not required to be terminated by a null character, but it is perfectly valid to have a sequence of characters that contains null characters. The `std::string`, on the other hand, needs to be able to return a null-terminated string from `c_str()`, which means that it always stores an extra null character at the end of the sequence.

The fact that `string_view` does not need a null terminator means that it can handle substrings much more efficiently than a C-style string or a `std::string` because it does not have to create new strings just to add the null terminator. The complexity of `substr()` using a `std::string_view` is constant, which should be compared to the `substr()` version of `std::string`, which runs in linear time.

There is also a performance win when passing strings to functions. Consider the following code:

```
auto some_func(const std::string& s) {
  // process s ...
}
some_func("A string literal"); // Creates a std::string
```

When passing a string literal to `some_func()`, the compiler needs to construct a new `std::string` object to match the type of the argument. However, if we let `some_func()` accept a `std::string_view`, there is no longer any need to construct a `std::string`:

```
auto some_func(std::string_view s) { // Pass by value
  // process s ...
```

```
}
some_func("A string literal");
```

A `std::string_view` instance can be constructed efficiently both from a `std::string` and a string literal and is, therefore, a well-suited type for function parameters.

Eliminating array decay with std::span

When discussing `std::vector` and `std::array` earlier in this chapter, I mentioned that array decay (losing the size information of an array) happens with built-in arrays when they are passed to a function:

```
// buffer looks like an array, but is in fact a pointer
auto f1(float buffer[]) {
 const auto n = std::size(buffer);  // Does not compile!
 for (auto i = 0u; i < n; ++i) {    // Size is lost!
  // ...
 }
}
```

We could get around this problem by adding a size parameter:

```
auto f2(float buffer[], size_t n) {
 for (auto i = 0u; i < n; ++i) {
  // ...
 }
}
```

Although this technically works, passing the correct data to this function is both error-prone and tedious, and if `f2()` passes the buffer to other functions, it needs to remember to pass the correctly sized variable `n`. This is what the call site of `f2()` might look like:

```
float a[256];
f2(a, 256);
f2(a, sizeof(a)/sizeof(a[0])); // A common tedious pattern
f2(a, std::size(a));
```

Array decay is the source of many bound-related bugs, and in situations where built-in arrays are used (for one reason or another), `std::span` offers a safer way to pass arrays to functions. Since the span holds both the pointer to the memory and the size together in one object, we can use it as the single type when passing sequences of elements to functions:

```
auto f3(std::span<float> buffer) {  // Pass by value
  for (auto&& b : buffer) {       // Range-based for-loop
    // ...
  }
}
float a[256];
f3(a);        // OK! Array is passed as a span with size
auto v = std::vector{1.f, 2.f, 3.f, 4.f};
f3(v);        // OK!
```

A span is also more convenient to use over a built-in array since it acts more like a regular container with support for iterators.

There are many similarities between `std::string_view` and `std::span` when it comes to the data members (pointer and size) and the member functions. But there are also some notable differences: the memory pointed to by `std::span` is mutable, whereas the `std::string_view` always points to constant memory. `std::string_view` also contains string-specific functions such as `hash()` and `substr()`, which are naturally not part of `std::span`. Lastly, there is no `compare()` function in `std::span`, so it's not possible to directly use the comparison operators on `std::span` objects.

It's now time to highlight a few general points related to performance when using data structures from the standard library.

Some performance considerations

We have now covered the three major container categories: sequence containers, associative containers, and container adaptors. This section will provide you with some general performance advice to consider when working with containers.

Balancing between complexity guarantees and overhead

Knowing the time and memory complexity of data structures is important when choosing between containers. But it's equally important to remember that each container is afflicted with an overhead cost, which has a bigger impact on the performance for smaller datasets. The complexity guarantees only become interesting for sufficiently large datasets. It's up to you, though, to decide what sufficiently large means in your use cases. Here, again, you need to measure your program while executing it to gain insights.

In addition, the fact that computers are equipped with memory caches makes the use of data structures that are friendly to the cache more likely to perform better. This usually speaks in favor of the `std::vector`, which has a low memory overhead and stores its elements contiguously in memory, making access and traversal faster.

The following diagram shows the actual running time of two algorithms. One runs in linear time, *O(n)*, and the other runs in logarithmic time, *O(log n)*, but with a larger overhead. The logarithmic algorithm is slower than the linear time algorithm when the input size is below the marked threshold:

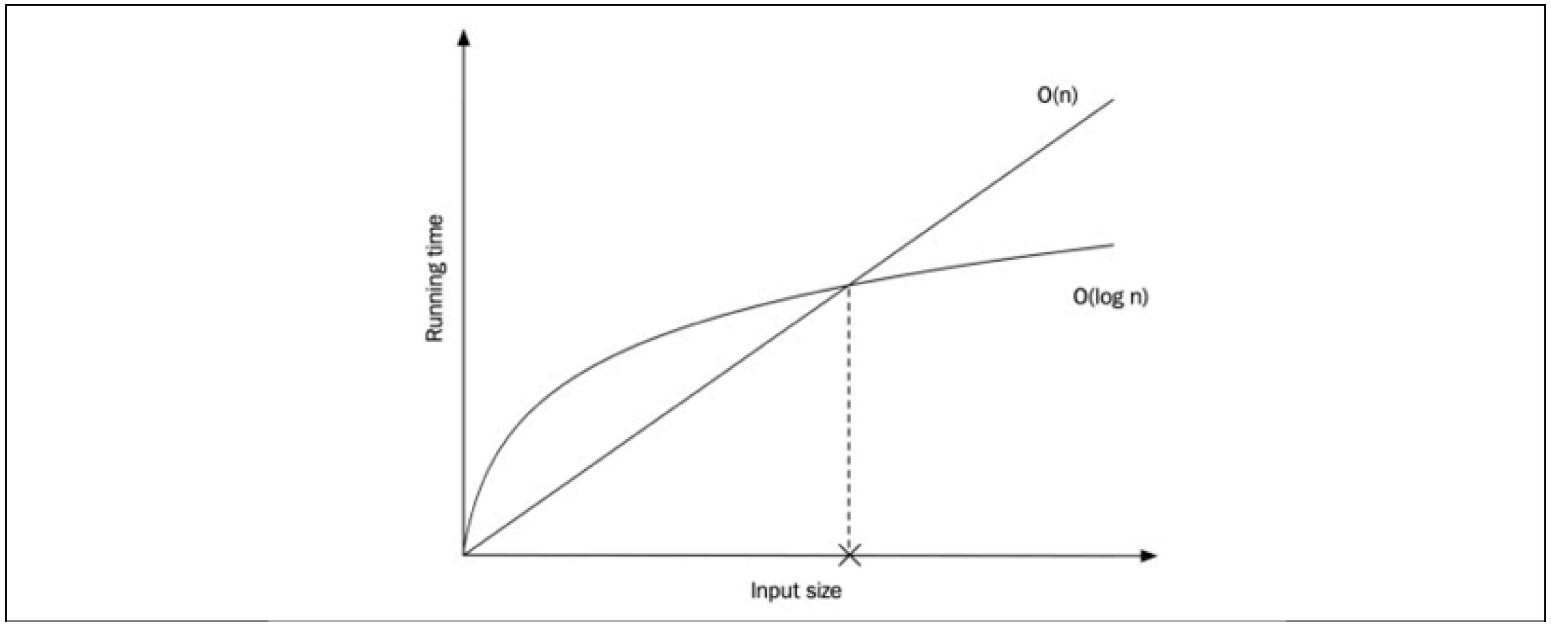

Figure 4.14: For small sizes of n, the linear algorithm, O(n), is faster than the algorithm running in O(log n)

Our next point to keep in mind is more concrete and highlights the importance of using the most suitable API functions.

Knowing and using the appropriate API functions

In C++, there is usually more than one way to do something. The language and the library continue to evolve, but very few features are being deprecated. When new functions are added to the standard library, we should learn when to use them and reflect on what patterns we might have used to compensate for a previously missing function.

Here, we will focus on two small, but important, functions that can be found in the standard library: `contains()` and `empty()`. Use `contains()` when checking whether an element exists in an associated container. Use `empty()` if you want to know whether a container has any elements or is empty. Apart from expressing the intent more clearly, it also has performance benefits. Checking the size of a linked list is an *O(n)* operation, whereas calling `empty()` on a list runs in constant time, *O(1)*.

Before C++20 and the introduction of the `contains()` function, we had to take a detour every time we wanted to check for the existence of some value in an associative container. You have most likely stumbled upon code that uses various ways to look for the existence of an element. Suppose we have a bag-of-words implemented using a `std::multiset`:

```
auto bag = std::multiset<std::string>{}; // Our bag-of-words
// Fill bag with words ...
```

If we want to know whether some specific word is in our bag-of-words, there are numerous ways to go forward. One alternative would be to use `count()`, like this:

```
auto word = std::string{"bayes"}; // Our word we want to find
if (bag.count(word) > 0) {
  // ...
}
```

It seems reasonable, but it may have a slight overhead since it counts *all* elements that match our word. Another alternative is to use `find()`, but it has the same overhead since it returns all the matching words, not just the first occurrence:

```
if (bag.find(word) != bag.end()) {
  // ...
}
```

Before C++20, the recommended way was to use `lower_bound()`, since it only returns the first matching element, like this:

```
if (bag.lower_bound(word) != bag.end()) {
  // ...
}
```

Now, with C++20 and the introduction of `contains()`, we can express our intent more clearly and also be sure that the library will provide us with the most efficient implementation when we only want to check for existence of an element:

```
if (bag.contains(word)) { // Efficient and with clear intent
  // ...
}
```

The general rule is that if there is a specific member function or a free function designed for a specific container, then use it if it matches your needs. It will be efficient, and it will express the intent more clearly. Don't use detours like the ones shown earlier just because you haven't learned the full API or because you have old habits of doing things in a certain way.

It should also be said that the zero-overhead principle applies particularly well to functions like these, so don't spend time trying to outsmart the library implementors by handcrafting your own functions.

We will now go ahead and look at a lengthier example of how we can reorder data in different ways to optimize runtime performance for a specific use case.

Parallel arrays

We will finish this chapter by talking about iterating over elements and exploring ways to improve performance when iterating over array-like data structures. I have already mentioned two important factors for performance when accessing data: spatial locality and temporal locality. When iterating over elements stored contiguously in memory, we will increase the probability that the data we need is already cached if we manage to keep our objects small, thanks to spatial locality. Obviously, this will have a great impact on performance.

Recall the cache-thrashing example, shown at the beginning of this chapter, where we iterated over a matrix. It demonstrated that we sometimes need to think about the way we access data, even if we have a fairly compact representation of the data.

Next, we will compare how long it takes to iterate over objects of different sizes. We will start by defining two structs, `SmallObject` and `BigObject`:

```
struct SmallObject {
  std::array<char, 4> data_{};
  int score_{std::rand()};
};

struct BigObject {
 std::array<char, 256> data_{};
 int score_{std::rand()};
};
```

`SmallObject` and `BigObject` are identical, except for the size of the initial data array. Both structs contain an `int` named `score_`, which we initialize to a random value just for testing purposes. We can let the compiler tell us the size of the objects by using the `sizeof` operator:

```
std::cout << sizeof(SmallObject); // Possible output is 8
std::cout << sizeof(BigObject);   // Possible output is 260
```

We need plenty of objects in order to evaluate the performance. Create one million objects of each kind:

```
auto small_objects = std::vector<SmallObject>(1'000'000);
auto big_objects = std::vector<BigObject>(1'000'000);
```

Now for the iteration. Let's say that we want to sum the scores of all the objects. Our preference would be to use `std::accumulate()`, which we will cover later in the book, but, for now, a simple `for`-loop will do. We write this function as a template so that we don't have to manually write one version for each type of object. The function iterates over the objects and sums all the scores:

```
template <class T>
auto sum_scores(const std::vector<T>& objects) {
 ScopedTimer t{"sum_scores"};    // See chapter 3

 auto sum = 0;
 for (const auto& obj : objects) {
  sum += obj.score_;
 }
 return sum;
}
```

Now, we are ready to see how long it takes to sum the scores in the small objects compared to the big objects:

```
auto sum = 0;
sum += sum_scores(small_objects);
sum += sum_scores(big_objects);
```

To achieve reliable results, we need to repeat the test a couple of times. On my computer, it takes about 1 ms to compute the sum of the small objects and 10 ms to compute the sum of the big objects. This example is similar to the cache thrashing example at the beginning of the chapter, and one reason for the big difference is, again, because of the way the computer uses the cache hierarchy to fetch data from the main memory.

How can we utilize the fact that it's faster to iterate over collections of smaller objects than bigger objects when working with more realistic scenarios than the preceding example?

Obviously, we can do our best to keep the size of our classes small, but it's often easier said than done. Also, if we are working with an old code base that has been growing for some time, the chances are high that we will stumble across some really large classes with too many data members and too many responsibilities.

We will now look at a class that represents a user in an online game system and see how we can split it into smaller parts. The class has the following data members:

```
struct User {
  std::string name_;
  std::string username_;
  std::string password_;
  std::string security_question_;
  std::string security_answer_;
  short level_{};
  bool is_playing_{};
};
```

A user has a name that is frequently used and some information for authentication that are rarely used. The class also keeps track of which level the player is currently playing at. Finally, the `User` struct also knows whether the user is currently playing by storing the `is_playing_` boolean.

The `sizeof` operator reports that the `User` class is 128 bytes when compiling for a 64-bit architecture. An approximate layout of the data members can be seen in the following figure:

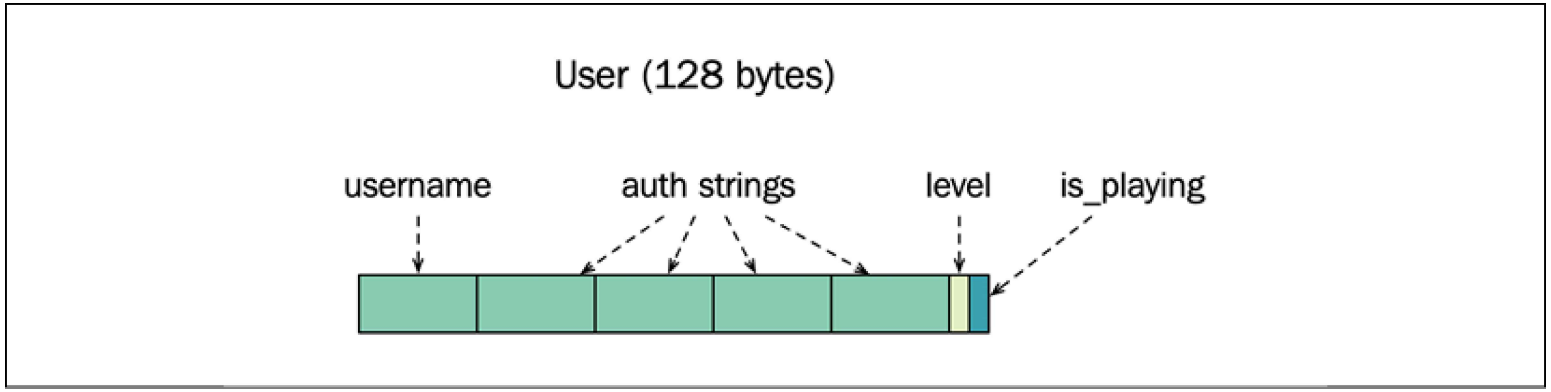

Figure 4.15: Memory layout of the User class

All users are kept in a `std::vector`, and there are two global functions that are called very often and need to run quickly: `num_users_at_level()` and `num_playing_users()`. Both functions iterate over all users, and therefore we need to make iterations over the user vector quick.

The first function returns the number of users who have reached a certain level:

```
auto num_users_at_level(const std::vector<User>& users, short level) {
  ScopedTimer t{"num_users_at_level (using 128 bytes User)"};
```

```
  auto num_users = 0;
  for (const auto& user : users)
    if (user.level_ == level)
      ++num_users;
  return num_users;
}
```

The second function computes how many users are currently playing:

```
auto num_playing_users(const std::vector<User>& users) {
  ScopedTimer t{"num_playing_users (using 128 bytes User)"};

  return std::count_if(users.begin(), users.end(),
    [](const auto& user) {
      return user.is_playing_;
    });
}
```

Here, we use the algorithm `std::count_if()` instead of a handwritten loop, as we did in `num_users_at_level()`. `std::count_if()` will call the predicate we provide for each user in the user vector and return the number of times the predicate returns `true`. This is basically what we are doing in the first function as well, so we could also have used `std::count_if()` in the first case. Both functions run in linear time.

Calling the two functions with a vector of one million users results in the following output:

```
11 ms num_users_at_level (using 128 bytes User)
10 ms num_playing_users (using 128 bytes User)
```

We hypothesize that by making the `User` class smaller, it would be faster to iterate over the vector. As mentioned, the password and security data fields are rarely used and could be grouped in a separate struct. That would give us the following classes:

```
struct AuthInfo {
  std::string username_;
  std::string password_;
  std::string security_question_;
  std::string security_answer_;
};

struct User {
  std::string name_;
  std::unique_ptr<AuthInfo> auth_info_;
  short level_{};
  bool is_playing_{};
};
```

This change decreases the size of the `User` class from 128 bytes to 40 bytes. Instead of storing four strings in the `User` class, we use a pointer to refer to the new `AuthInfo` object. The following figure shows you how we have split up the `User` class into two smaller classes:

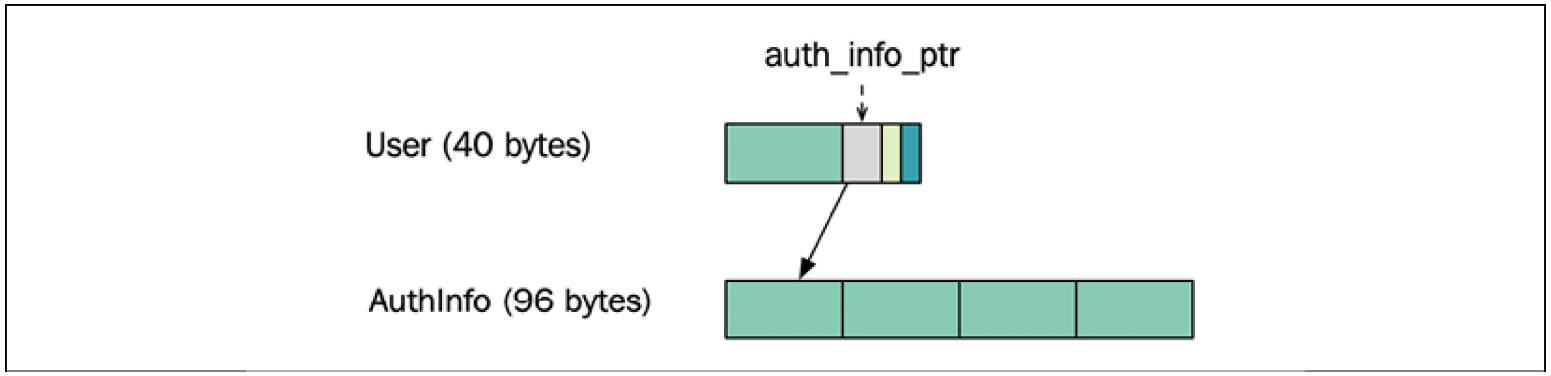

Figure 4.16: Memory layout when authentication information is kept in a separate class

This change makes sense from a design perspective too. Keeping the authentication data in a separate class increases the cohesion of the `User` class. The `User` class contains a pointer to the authentication information. The total amount of memory that the user data occupies has not decreased, of course, but the important thing right now is to shrink the `User` class in order to speed up the functions that iterate over all users.

From an optimization point of view, we have to measure this again to verify that our hypothesis regarding smaller data is valid. It turns out that both functions run more than twice as fast with the smaller `User` class. The output when running the modified version is:

```
4 ms num_users_at_level with User
3 ms num_playing_users with User
```

Next, we are going to try a more aggressive way of shrinking the amount of data we need to iterate through by using **parallel arrays**. First, a warning: this is an optimization that, in many cases, has too

many drawbacks to be a viable alternative. Don't take this as a general technique and apply it without thinking twice. We will come back to the pros and cons of parallel arrays after looking at a few examples.

By using parallel arrays, we simply split the large structures into smaller types, similar to what we did with the authentication information for our `User` class. But instead of using pointers to relate objects, we store the smaller structures in separate arrays of equal size. The smaller objects in the different arrays that share the same index form the complete original object.

An example will clarify this technique. The `User` class we have worked with consists of 40 bytes. It now only contains a username string, a pointer to the authentication information, an integer for the current level, and the `is_playing_` boolean. By making the user objects smaller, we saw that the performance improved when iterating over the objects. The memory layout of an array of user objects would look something like the one shown in the following figure. We will ignore memory alignment and padding for now, but will get back to these topics in *Chapter 7*, *Memory Management*:

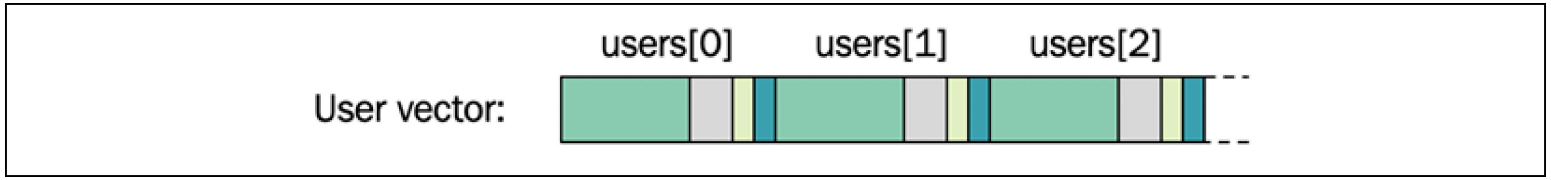

Figure 4.17: User objects stored contiguously in a vector

Instead of having one vector with user objects, we can store all the `short` levels and `is_playing_` flags in separate vectors. The current level for the user at index 0 in the user array is also stored at index 0 in the level array. In that way, we can avoid having pointers to the levels, and instead just use the index for connecting the data fields. We could do the same thing with the boolean `is_playing_` field and end up with

three parallel arrays instead of just one. The memory layout of the three vectors would look something like this:

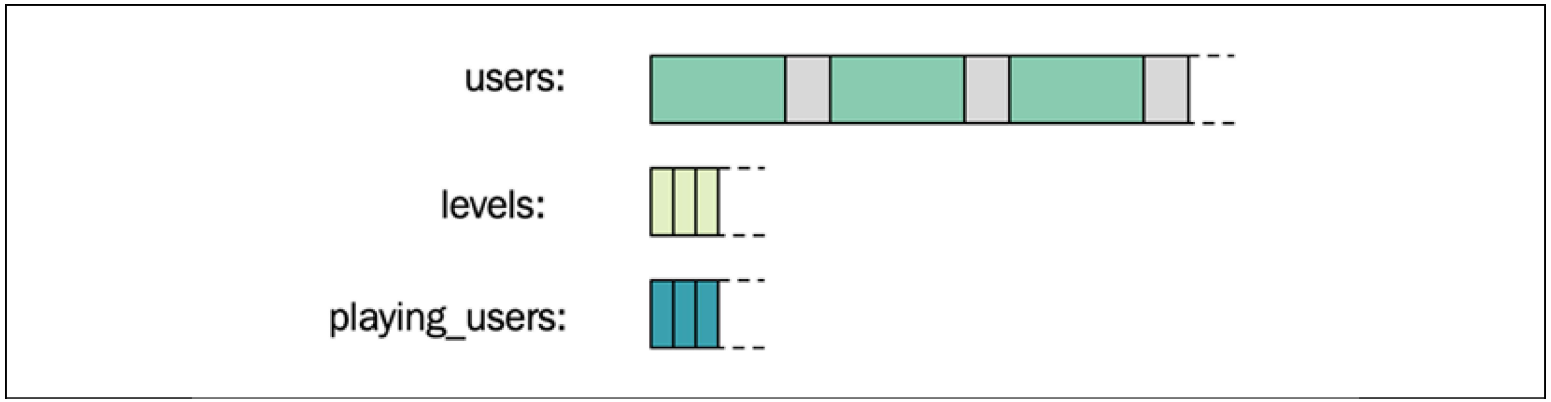

Figure 4.18: Memory layout when using three parallel arrays

We are using three parallel arrays to make iterations over one particular field quickly. The `num_users_at_level()` function can now compute the number of users at a specific level by only using the level array. The implementation is now simply a wrapper around `std::count()`:

```
auto num_users_at_level(const std::vector<int>& users, short level) {
  ScopedTimer t{"num_users_at_level using int vector"};
  return std::count(users.begin(), users.end(), level);
}
```

Likewise, the `num_playing_users()` function only needs to iterate over the vector of booleans to determine the number of playing users. Again, we use `std::count()`:

```
auto num_playing_users(const std::vector<bool>& users) {
  ScopedTimer t{"num_playing_users using vector<bool>"};
  return std::count(users.begin(), users.end(), true);
}
```

With parallel arrays, we don't have to use the user array at all. The amount of memory occupied by the extracted arrays is substantially smaller than the user array, so let's check whether we have improved on performance when running the functions on one million users again:

```
auto users = std::vector<User>(1'000'000);
auto levels = std::vector<short>(1'000'000);
auto playing_users = std::vector<bool>(1'000'000);

// Initialize data
// ...

auto num_at_level_5 = num_users_at_level(levels, 5);
auto num_playing = num_playing_users(playing_users);
```

Counting the number of users at a certain level only takes about 0.7 ms when using the array of integers. To recap, the initial version using the `User` class with a size of 128 bytes took around 11 ms. The smaller `User` class executed in 4 ms, and now, by only using the `levels` array, we are down to 0.7 ms. This is quite a dramatic change.

For the second function, `num_playing_users()`, the change is even bigger—it only takes around 0.03 ms to count how many users are currently playing. The reason why it can be so fast is thanks to a data struc-

ture called **bit arrays**. It turns out that `std::vector<bool>` is not at all a standard vector of C++ `bool` objects. Instead, internally, it's a bit array. Operations such as `count()` and `find()` can be optimized very efficiently in a bit array since it can process 64 bits at a time (on a 64-bit machine), or possibly even more by using SIMD registers. The future of `std::vector<bool>` is unclear, and it might be deprecated soon in favor of the fixed-size `std::bitset` and a new dynamically sized bitset. There is already a version in Boost named `boost::dynamic_bitset`.

This is all fantastic, but I warned you about some drawbacks. First of all, extracting the fields from the classes where they actually belong will have a big impact on the structure of the code. In some cases, it makes perfect sense to split large classes into smaller parts, but in other cases, it totally breaks encapsulation and exposes data that could have been hidden behind interfaces with higher abstraction.

It's also cumbersome to ensure that the arrays are in sync, such that we always need to ensure that fields that comprise one object are stored at the same index in all arrays. Implicit relationships like this can be hard to maintain and are error-prone.

The last drawback is actually related to performance. In the preceding example, you saw that for algorithms that iterate over one field at a time, there was a big performance gain. However, if we have an algorithm that would need to access multiple fields that have been extracted into different arrays, it would be substantially slower than iterating over one array with bigger objects.

So, as is always the case when working with performance, there is nothing that comes without a cost, and the cost for exposing data and splitting one simple array into multiple arrays may or may not be too high. It all depends on the scenario you are facing and what performance gain you encounter after measuring. Don't consider parallel arrays before you actually face a real performance issue. Always opt for sound design principles first, and favor explicit ways of expressing relationships between objects rather than implicit ones.

Summary

In this chapter, the container types from the standard library were introduced. You learned that the way we structure data has a big impact on how efficiently we can perform certain operations on a collection of objects. The asymptotic complexity specifications of the standard library containers are key factors to consider when choosing among the different data structures.

In addition, you learned how the cache hierarchy in modern processors impacts the way we need to organize data for efficient access to memory. The importance of utilizing the cache levels efficiently cannot be stressed enough. This is one of the reasons why containers that keep their elements contiguously in memory have become the most used, such as `std::vector` and `std::string`.

In the next chapter, we will look at how we can use iterators and algorithms to operate on containers efficiently.