Chapter 7
Vectors

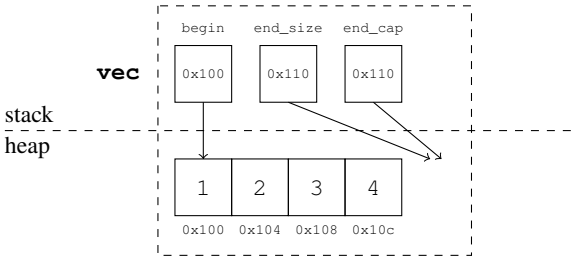
7.1 The STL Vector Container

If you wanted a dynamic array-based container that can grow with the size of your input, you do not have to implement it from scratch! This is thanks to the C++ **standard template library (STL)**, which provides efficient implementations of containers and algorithms for you. In this chapter, we will discuss the C++ `std::vector<>` container, which is an array-based container that manages its elements using a dynamically-sized heap-allocated C-style array. You can store elements in a vector much like how you can store elements in an array, but the vector can manage its own memory and will resize itself automatically if additional space is needed (contrary to an array, which is fixed in size).

To use a vector in your program, you must `#include <vector>` at the top of your code file. The following line declares a vector of integers with initial contents `{1, 2, 3, 4}`:

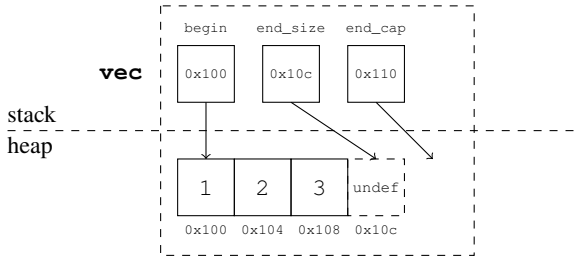
```
std::vector<int32_t> vec = {1, 2, 3, 4};
```

When the vector is initialized, it creates a C-style array on the heap (i.e., dynamic memory) with the contents `{1, 2, 3, 4}`. What exactly does a vector look like under the hood? This is implementation specific, but the GCC libstdc++ implementation of `std::vector<>` stores three pointers on the program stack: (1) a pointer that points to the beginning of the heap-allocated array, (2) a pointer that points one past the last *valid* element in the heap-allocated array, and (3) a pointer that points one past the last *allocated* element in the heap-allocated array. The last valid element and the last allocated element need *not* be the same.



Note that this is only one implementation of the `std::vector<>`, and the exact implementation details of a STL vector may be platform dependent as long as the public interface of the container conforms with the C++ standard. The underlying structure of a vector can differ across different standard library implementations (e.g., an implementation could keep track of one pointer and two integers for size and capacity instead of three pointers, and that would still be valid). The purpose of this chapter is to explore the general idea of how a vector works internally, and not to delve into the details of a single implementation, so you do not need to specifically memorize this vector implementation.

Regardless of how a vector is implemented, there are two important values that the vector must be able to determine at all times. The first is its **size**, which represents the number of valid elements that the underlying array *actually* holds. The second is its **capacity**, which represents the number of elements the underlying array *is able to* hold. These two values do not need to be the same. For instance, suppose we removed the last element, `4`, from the back of the previously defined vector:



Since the vector's underlying array has a fixed size, it can still hold at most 4 elements. However, only 3 elements in the vector are still valid after the last element was removed. Thus, the size of the vector is 3, while its capacity is 4.

The vector is able to calculate its size by finding the distance between the pointer to the first element and the pointer one past the last valid element (i.e., `end_size - begin` using the variable names above). It is also able to calculate its capacity by finding the distance between the first element and the pointer one past the last allocated element (i.e., `end_cap - begin`).[1]

The following methods can be used to initialize a vector:

```
template <typename T>
std::vector<T>();
```
Default constructor for vector that holds elements of type `T`; creates an empty vector with no elements; size and capacity are initially set to 0.

```
// initializes the vector v1 with zero size and zero capacity
std::vector<int32_t> v1;
```

```
template <typename T>
std::vector<T>(std::initializer_list<T> init);
```
Initializes the vector with the contents of the initializer list.

```
// initializes v2 and v3 to have contents {1, 2, 3}, with size and capacity 3
std::vector<int32_t> v2{1, 2, 3};
std::vector<int32_t> v3 = {1, 2, 3};
```

```
template <typename T>
std::vector<T>(const std::vector<T>& other);
```
Copy constructor, copies the contents of `other` into the constructed vector.

```
// initializes v4 with the contents of v3, or {1, 2, 3}
std::vector<int32_t> v4{v3};
```

```
template <typename T>
std::vector<T>(size_t sz);
```
Creates a vector of `sz` elements, where each element is value initialized; size and capacity both initially equal to `sz`.

```
// initializes v5 with size and capacity 5, each element initialized to 0
std::vector<int32_t> v5(5);
```

```
template <typename T>
std::vector<T>(size_t sz, T& val);
```
Creates a vector of `sz` elements, where each element is initialized to a value of `val`; size and capacity both equal to `sz`.

```
// initializes v6 with size and capacity 5, each element initialized to 1
std::vector<int32_t> v6(5, 1);
```

[1] Pointer arithmetic is performed relative to the base type of the pointer. Since integers take up 4 bytes each, incrementing `0x100` by 1 actually returns the address $\texttt{0x100} + 1 \times 4 = \texttt{0x104}$.

```
template <typename T, typename InputIterator>
std::vector<T>(InputIterator begin_iter, InputIterator end_iter);
```
Creates a vector with all elements in the iterator range `[begin_iter, end_iter)` — inclusive begin but exclusive end. Both `begin_iter` and `end_iter` are input iterators.

```
// initializes v7 with first three elements of v6
// .begin() returns an iterator to the first element of the vector
std::vector<int32_t> v7{v6.begin(), v6.begin() + 3};
```

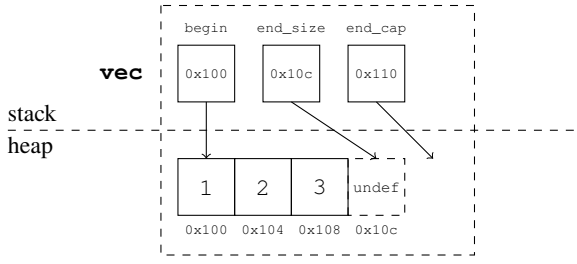
7.2 Inserting and Removing Elements

※ 7.2.1 *Push Back*

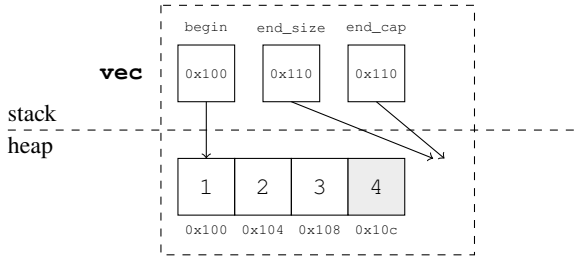
Two methods for inserting and removing elements from a vector are `std::vector::push_back()` and `std::vector::pop_back()`. When `.push_back(val)` is called, `val` is added to the back of the vector, and the size of the vector increases by one.

```
template <typename T>
void std::vector<T>::push_back(const T& val);
```
Appends a new value initialized to `val` to the back of the vector, after the current last element (if there is available capacity). If the new vector size surpasses the current capacity, the underlying array is reallocated before the new element is appended. After the object is successfully appended, the size of the vector increases by one.

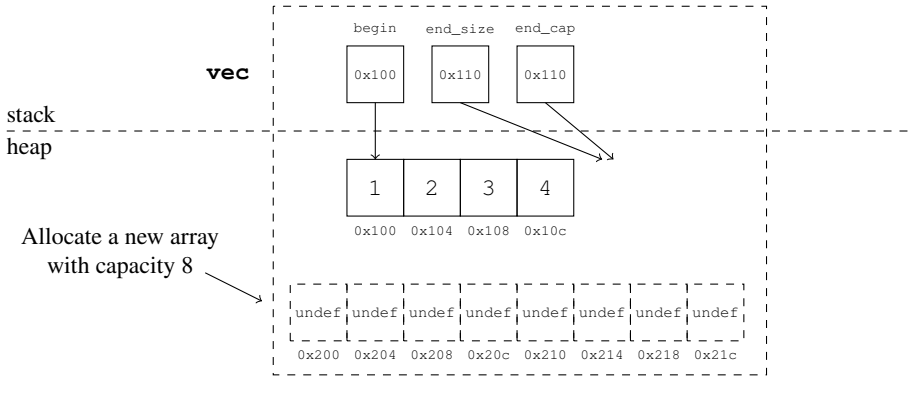
Let's consider the state of the previous vector, as reproduced below.

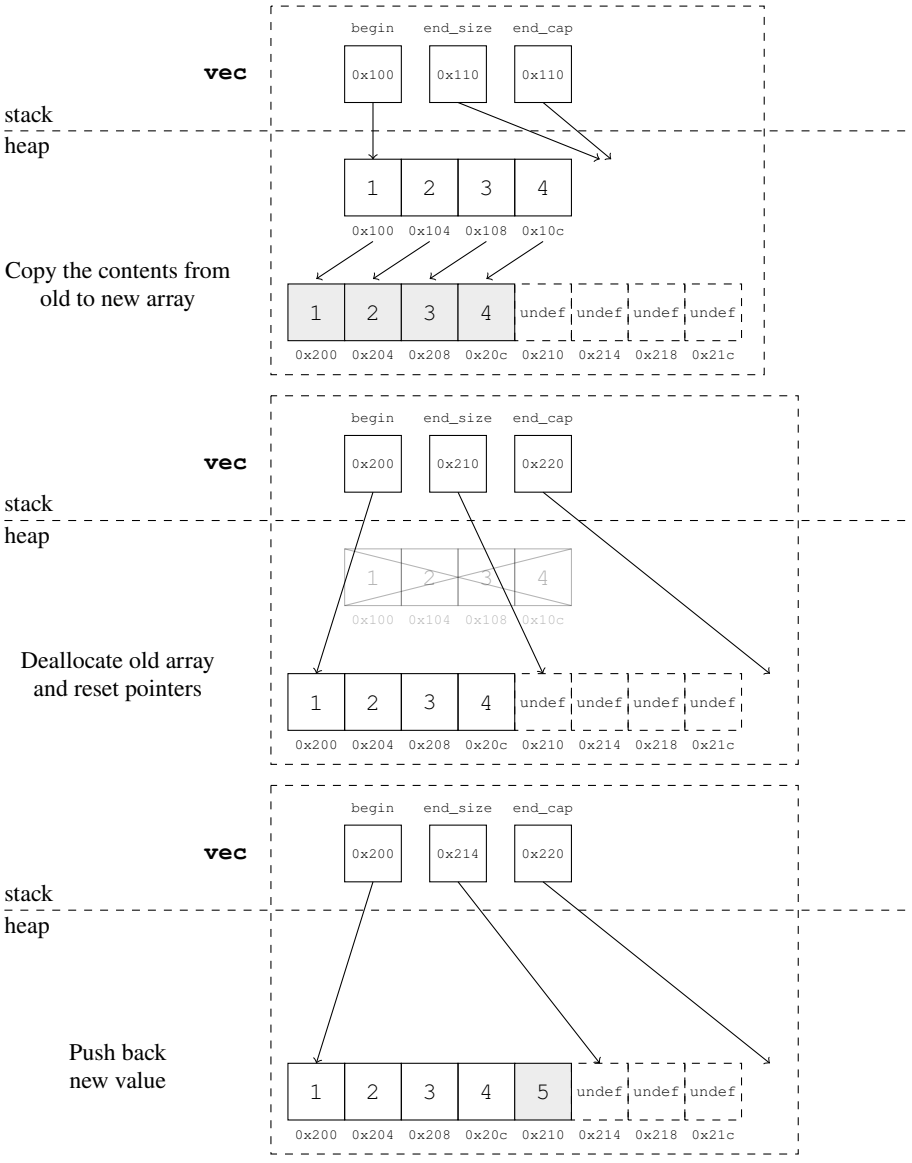


If we call `vec.push_back(4)`, we push the value 4 to the back of the vector's underlying array, after the current last element (which is 3).



What happens if we try to push back another element by calling `vec.push_back(5)`? If we add this element, the size of the vector would be 5. However, the underlying heap-allocated array that stores the vector's data has a fixed size of 4, so it cannot hold a fifth element! Therefore, we will need a new array that can support a size of 5. When this happens, the vector allocates a new array in memory that is double the capacity of the original array, copies over the data, frees up the memory of the old array, and resets its internal pointers to point to the new array.





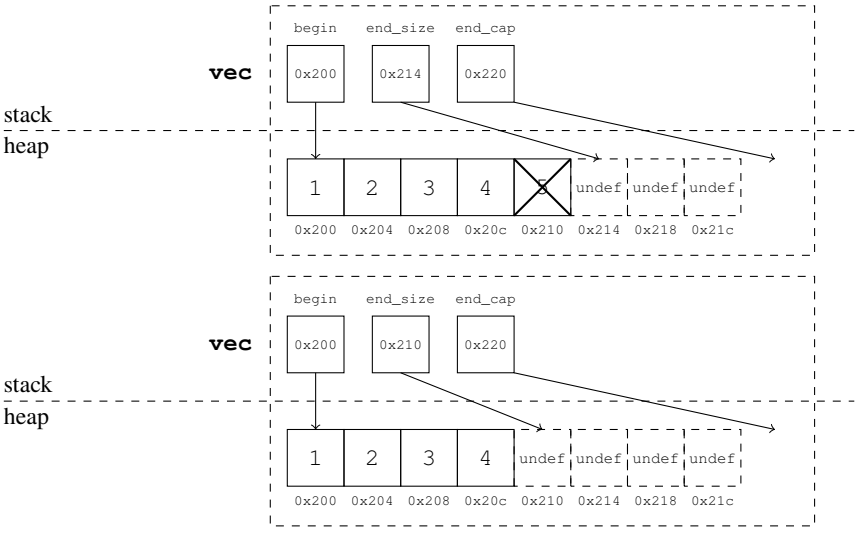
※ 7.2.2 *Pop Back*

To remove the last element in a vector, the `.pop_back()` method can be used.

```
template <typename T>
void std::vector<T>::pop_back();
```

Removes the last element in the vector, reducing its size by one.

If we call `vec.pop_back()` on our vector above, the 5 would be removed.



Removing an element using `.pop_back()` does not trigger a reallocation. Here, the size of the vector is 4 after the last element is removed, but the capacity is still 8. Reallocation can only happen if an insertion occurs when the vector is at capacity, or the programmer explicitly shrinks or expands the capacity of the underlying array. We will discuss several reasons why this is the case in the next section.

In addition, calling `.pop_back()` on an empty vector causes undefined behavior. The programmer is expected to know whether the vector is empty or not; this removes the need for `.pop_back()` to conduct an extra (and potentially unnecessary) "empty" check every time it is called (effectively sacrificing safety for performance).

A call to `.pop_back()` takes $\Theta(1)$ time. A call to `.push_back()` takes $\Theta(n)$ time if reallocation occurs and $\Theta(1)$ time otherwise. However, because the capacity of the underlying array is doubled upon reallocation, each $\Theta(n)$ push makes subsequent pushes less expensive; it turns out that this allows the time complexity of `.push_back()` to be treated as a $\Theta(1)$ operation if we average out all the work of reallocation across all the individual pushes. This idea is known as amortization, and we will cover it in more detail in chapter 12.

※ 7.2.3 *Emplace Back*

If you have a vector of large objects, `std::vector::emplace_back()` can be used as an alternative to `std::vector::push_back()`.

```
template <typename T, typename... Args>
T& std::vector<T>::emplace_back(Args&&... args);
```

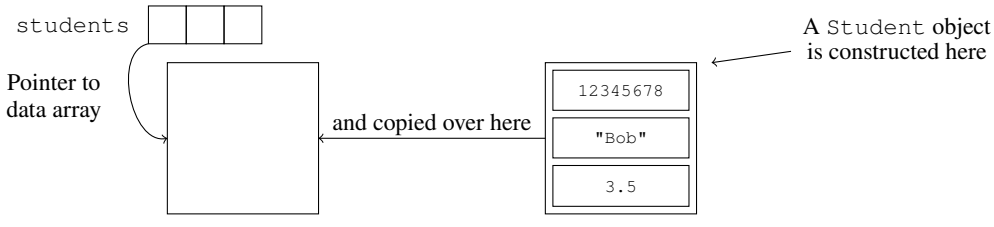
Appends a new element at the back of the vector, right after the current last element (if there is available capacity). If the new vector size surpasses the current capacity, the underlying array is reallocated before the new element is emplaced. The new element is constructed *in place* using arguments for its constructor (`args`), which are passed into the `.emplace_back()` function call. After the object is successfully emplaced, the size of the vector increases by one. Since C++17, a reference to the emplaced element is returned.

To illustrate how this works, consider the following `Student` object defined below:

```
1  struct Student {
2    int32_t id;
3    std::string name;
4    double gpa;
5    Student(int32_t id_in, std::string name_in, double gpa_in) :
6      id{id_in}, name{name_in}, gpa{gpa_in} {}
7  };
8
9  std::vector<Student> students;
```

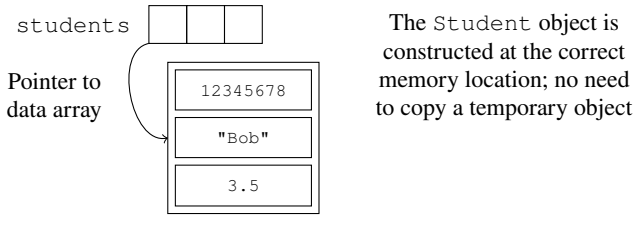
When you push back a `Student` object, you must first create a temporary instance of the `Student` you want to push back (as `.push_back()` can only accept an argument of the same type as the vector). Then, when `.push_back()` is invoked, the temporary `Student` is copied to the back of the vector.

```
1  Student s{12345678, "Bob", 3.5};
2  students.push_back(s);  // or on one line: students.push_back(Student{12345678, "Bob", 3.5});
```



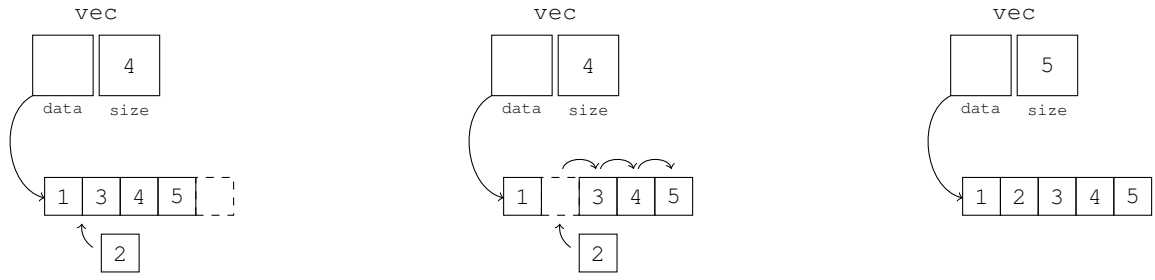
What makes `.emplace_back()` different is that it accepts a list of *arguments* rather than a `Student` object itself. It then forwards the arguments to a constructor that can construct the object in the destination memory address of the vector itself. Here, `.emplace_back()` takes the three arguments, sends it to the `Student` constructor, and builds the object in the right spot. There is no need to create a temporary variable and copy it over!

```
1  // pass into emplace_back() what you would pass into the object's constructor
2  students.emplace_back(12345678, "Bob", 3.5);
```



※ 7.2.4 *Inserting and Erasing From Any Position*

Appending an element to the back of a vector is easy, but inserting elements anywhere else becomes slightly trickier. Because vectors require their elements to be stored contiguously in memory, if you attempt to insert an element to the vector, you must shift all the elements after the insertion point rightward to make room for the new element. The same idea applies with deletion; when you delete an element, you must shift all elements after the deletion point leftward to ensure there are no gaps in your data. Recall the illustration from the previous chapter:



Because `.push_back()`, `.emplace_back()`, and `.pop_back()` all deal with adding or removing elements at the end of the vector, there is no need to shift any elements (since there is no data after the insertion or deletion point). Thus, these operations all take $\Theta(1)$ time, assuming no reallocation is done.

If you want to insert elements elsewhere in a vector, you should use the `.insert()` function. There are several ways this can be done:

```
template <typename T>
iterator std::vector<T>::insert(const_iterator position, const T& val);
```
Inserts `val` directly before the element pointed to by the iterator `position` and returns an iterator the newly inserted element. Reallocation is done if the new size after insertion exceeds capacity.

```
std::vector<int32_t> vec = {0, 1, 2, 4, 5};
// insert 3 at index 3
vec.insert(vec.begin() + 3, 3);
// vec now {0, 1, 2, 3, 4, 5}
```

```
template <typename T>
iterator std::vector<T>::insert(const_iterator position, size_t n, const T& val);
```
Inserts `n` copies of `val` directly before the element pointed to by the iterator `position` and returns an iterator the first new element added. Reallocation is done if the new size after insertion exceeds capacity.

```
std::vector<int32_t> vec = {0, 1, 2, 4, 5};
// insert 4 copies of 3 at index 3
vec.insert(vec.begin() + 3, 4, 3);
// vec now {0, 1, 2, 3, 3, 3, 3, 4, 5}
```

```
template <typename T, typename InputIterator>
iterator std::vector<T>::insert(const_iterator position, InputIterator first, InputIterator last);
```
Inserts all elements in the iterator range `[first, last)` directly before `position` and returns an iterator the first new element added. Reallocation is done if the new size after insertion exceeds capacity.

```
std::vector<int32_t> vec1 = {0, 1, 5};
std::vector<int32_t> vec2 = {2, 3, 4};
// insert vec2 at index 2 of vec 1
vec1.insert(vec1.begin() + 2, vec2.begin(), vec2.end());
// vec1 now {0, 1, 2, 3, 4, 5}
```

```
template <typename T>
iterator std::vector<T>::insert(const_iterator position, std::initializer_list<T> init);
```
Inserts the elements in the initializer list into the vector directly before the iterator `position` and returns an iterator to the first new element added. Reallocation is done if the new size after insertion exceeds capacity.

```
std::vector<int32_t> vec = {0, 1, 2};
// insert {3, 4, 5} at the back
vec.insert(vec.end(), {3, 4, 5});
// vec1 now {0, 1, 2, 3, 4, 5}
```

The time complexity of `.insert()` is linear on the number of elements inserted plus the number of the elements after the point of insertion (since each element after this point needs to be shifted over). Similar to `.emplace_back()`, which directly constructs an element at the back of a vector, it is also possible to directly construct an element in the middle of a vector using `.emplace()`.

```
template <typename T, typename... Args>
iterator std::vector<T>::emplace(const_iterator pos, Args&&... args);
```
Inserts a new element directly before the element at position `pos`. The new element is constructed in place using arguments for its constructor (`args`). An iterator pointing to the emplaced object is returned.

```
1  std::vector<Student> students;
2  students.emplace_back(12345677, "Alice", 3.4);
3  students.emplace_back(12345679, "Cathy", 3.6);
4  students.emplace(students.begin() + 1, 12345678, "Bob", 3.5);
5  // students now stores the students in this order: {Alice, Bob, Cathy}
```

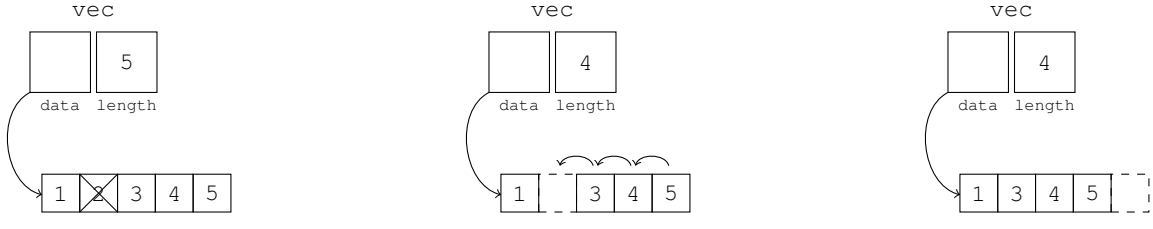
To remove elements from the vector at any position, you should use the `.erase()` member function:

```
template <typename T>
iterator std::vector<T>::erase(const_iterator position);
```
Erases the element pointed to by the iterator `position` and returns an iterator pointing to the new location of the element that followed the erased element.

```
template <typename T>
iterator std::vector<T>::erase(const_iterator first, const_iterator last);
```
Erases all elements in the iterator range `[first, last)` and returns an iterator pointing to the new location of the element that followed the last element erased.

```
1  std::vector<int32_t> vec = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3  // erase everything from index 6 to the end
4  vec.erase(vec.begin() + 6, vec.end());
5  // vec now {0, 1, 2, 3, 4, 5}
6
7  // erase element at index 3
8  vec.erase(vec.begin() + 3);
9  // vec now {0, 1, 2, 4, 5}
```

The time complexity of `.erase()` is linear on the number of elements erased plus the number of elements after the erasure point (since elements have to be shifted to ensure there are no gaps in the data).



Both `.insert()` and `.erase()` utilize iterators, which are generalized pointers that can be used to traverse the contents of a container. More detail on iterators will be provided in chapter 11, but the following methods can be used to retrieve iterators for a `std::vector<>` container:

Function	Behavior
`.begin()`	Returns a random access iterator to the first element in the vector
`.end()`	Returns a random access iterator to the position one past the last element in the vector
`.cbegin()`	Returns a *constant* random access iterator to the first element in the vector
`.cend()`	Returns a *constant* random access iterator to the position one past the last element in the vector
`.rbegin()`	Returns a *reverse* iterator to the last element in the vector
`.rend()`	Returns a *reverse* iterator to the position one before the first element in the vector
`.crbegin()`	Returns a constant reverse iterator to the last element in the vector
`.crend()`	Returns a constant reverse iterator to the position one before the first element in the vector

Listed below are some additional operations that are supported by a `std::vector<>`:

Function	Behavior
`operator[]`	Returns a reference to an element in the vector at a given index
`.front()`	Returns a reference to the first element in the vector (undefined behavior if empty)
`.back()`	Returns a reference to the last element in the vector (undefined behavior if empty)
`.empty()`	Returns whether the vector is empty (true or false)
`.clear()`	Erases all elements in the vector and leaves it with a size of 0 (but keeps capacity unchanged)

```
1  std::vector<int32_t> vec = {183, 203, 280, 281};
2
3  std::cout << vec.front() << '\n';                // prints 183
4  std::cout << vec.back() << '\n';                 // prints 281
5  std::cout << vec[1] << '\n';                     // prints 203
6
7  vec.clear();
8  std::cout << vec.empty() << '\n';                // prints 1 (true)
```

7.3 Resize and Reserve

※ 7.3.1 *The Costs of Reallocation*

A vector stores and manages its data in an array that is dynamically allocated on the heap. The size of a vector represents the number of valid elements that the vector *actually* holds, while the capacity of a vector represents the number of elements the vector's underlying array is *able* to hold. The capacity of the vector's underlying array may be larger than the size of the vector itself, since additional space may be allocated in advance whenever reallocation occurs. To get the size or capacity of the vector, you can use the following member functions:

Function	Behavior
`.size()`	Returns the size of the vector
`.capacity()`	Returns the capacity of the vector's underlying data array

The size and capacity of an empty vector both start at 0. Once the first element is pushed back, space is allocated for an underlying array and the capacity becomes 1. After another element is pushed back, the data is reallocated to another array and its capacity doubles to 2. The doubling process continues whenever reallocation occurs: adding the third element would double the capacity to 4, adding the fifth element would double the capacity to 8, and so on and so forth. This is the behavior that happens if every element were added using `.push_back()`.

If you instead initialize the vector with a non-zero number of elements (either using a range constructor or initializer list), the initial size and capacity are set to the number of elements present at initialization. For instance, if you create the following:

```
std::vector<int32_t> vec = {1, 2, 3, 4, 5};
```

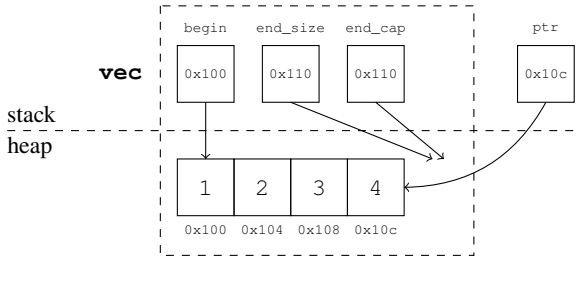
the vector would start off with a size and capacity of 5. Inserting another element into this vector (using `.push_back()`) would increase size to 6, but double capacity to 10.

It is important to understand the difference between size and capacity for many reasons. Even though a vector's size may be more important when implementing an algorithm or program, failure to consider its capacity could make your code less efficient. Several issues with capacity and reallocation are summarized below.

1. Reallocation invalidates all existing pointers and iterators to elements in the vector.

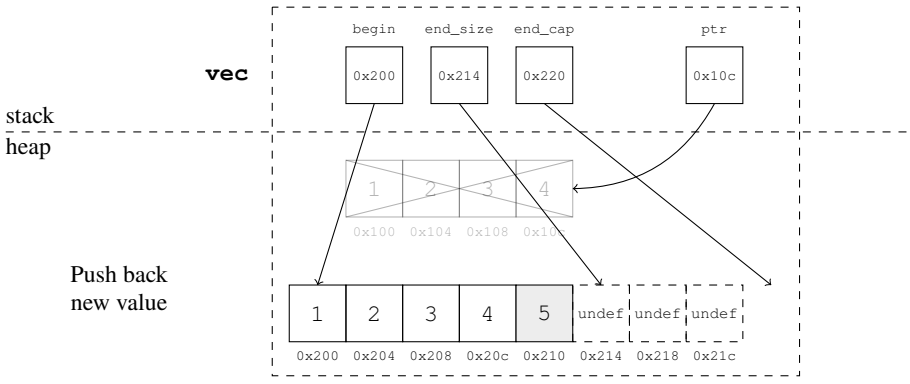
The data in a vector is stored in a fixed-size array on the heap. If the amount of data you have exceeds the capacity of the underlying array, the vector has to move the data to a larger array *elsewhere in memory*. Thus, the memory address that a pointer or iterator pointed to in the old array would no longer be valid. Consider the following code, which initializes a vector of size (and capacity) 4 and creates a pointer to the last element:

```
std::vector<int32_t> vec = {1, 2, 3, 4};
int* ptr = &vec[3];
```



Now, suppose we push back a fifth element to the back of the vector. Since the vector's underlying array only has a capacity of 4, a new larger array must be allocated to hold the fifth element. This triggers a reallocation.

```
vec.push_back(5);
```



Here, the value of `ptr` is the address location `0x10c`, since that was where the element `4` was originally located. However, because of reallocation, the `4` moved in memory — it is no longer at address `0x10c`! Thus, pointers or iterators that were declared before the reallocation are now invalid.[2]

2. Reallocation takes time.

It takes time to move data to a larger array, since all the elements have to be shifted over from the old array to the the new array. If you push back 1,000,000 elements into a vector that doubles capacity during every reallocation, the reallocation process happens 21 times!

3. If you know the intended size of your data, letting the vector set the capacity for you can waste memory.

If you do not tell the vector the size of your data, its default behavior is to double the capacity of the underlying array with every reallocation. This may end up being more than you need! For example, if you wanted to store 1,025 elements in a vector, and you only call `.push_back()` to insert these elements, the vector would double capacity from 1 to 2 to 4 to 8 to 16 to 32 to 64 to 128 to 256 to 512 to 1,024. When you insert the 1,025th element, the vector sees that the underlying array can only hold 1,024 elements, so it doubles capacity to 2,048. The vector doesn't know that element 1,025 was the last one! As a result, even though you only need space for 1,025 elements, you are actually reserving memory for an array of size 2,048 under the hood. All of this remaining memory is unused and wasted.

※ 7.3.2 *Reserving a Vector*

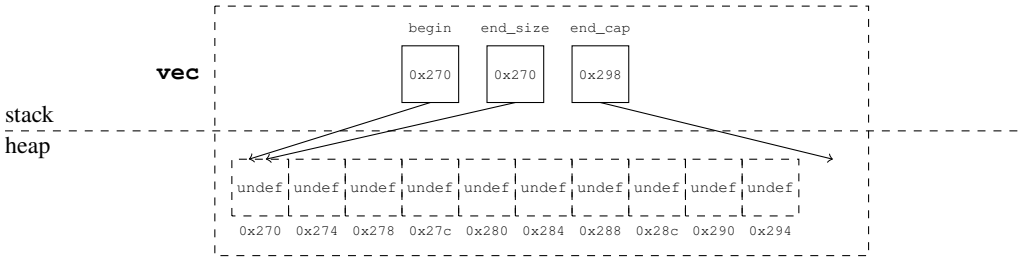
From the three issues above, we can see that reallocation should be avoided whenever possible. If you know the intended size of the vector beforehand, you can prevent these three issues from happening by presetting the capacity of the vector's underlying array upon initialization. For instance, if you know you only need space for 1,025 elements, you could tell the vector to allocate that exact amount of space at the beginning. This can be done using the `.reserve()` member function:

```
template <typename T>
void std::vector<T>::reserve(size_t n);
```

Requests that the vector capacity is at least `n`. If `n` is larger than the current vector capacity, this operation causes the vector to reallocate its data to an array with capacity `n`. Otherwise, no reallocation occurs. This function does *not* affect the size of the vector.

It is important to note that the `.reserve()` function does *not* actually change the size of the vector itself! You are *not* creating new elements when you reserve a vector; you are just allocating more space for future elements. For example, the following code reserves a vector of size 10:

```
std::vector<int32_t> vec;
vec.reserve(10);
```

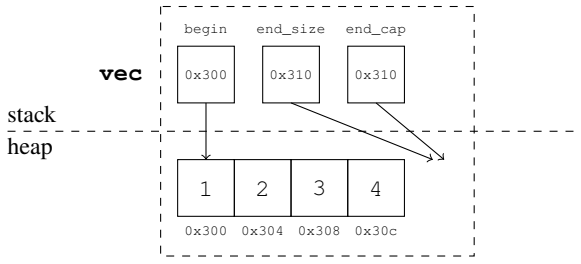


The size of the vector is still 0, since it holds no elements, but the capacity is 10 since the underlying array can hold at most 10 elements. Calling `.push_back()` would add the new element at the first open location (`0x270`).

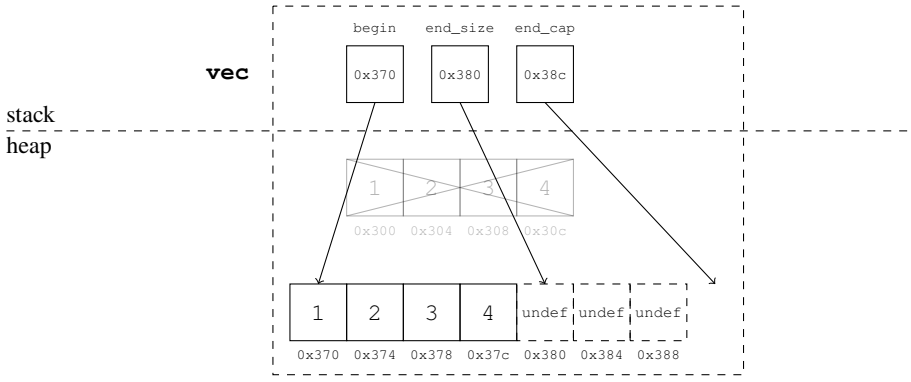
[2]Pointer and iterator invalidation is a pesky bug that is hard to detect. You must be careful when working with pointers or iterators if you modify the contents of the underlying container after creating them. Also, you do not need to know this, but the boost library provides a `boost::container::stable_vector<>` container that ensures that references and iterators to an element will always remain valid, as long as the element is not erased.

If you reserve a vector with a capacity that is larger than the current capacity, reallocation occurs. For example, the following code creates a vector that starts at a capacity of 4, but is reserved to a vector of capacity 7.

```
std::vector<int32_t> vec = {1, 2, 3, 4};
```




```
vec.reserve(7);
```



Once again, the size of the vector hasn't changed, but the capacity went up to 7. If you `.push_back()` a new element, it will get added to the first open position (`0x380`).

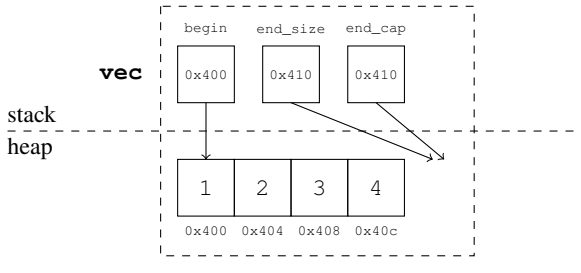
※ 7.3.3 *Resizing a Vector*

An alternative would be to use `.resize()`, which sets the *size* of the vector.

template <**typename** T> **void** std::vector<T>::resize(**size_t** n);
Resizes the vector so that it contains `n` elements. If `n` is smaller than the current size, the vector is forced down to a size of `n`, and any additional elements are destroyed (but capacity remains unchanged). If `n` is greater than the current size, new elements are value-initialized and added to the end of the array until the vector size becomes `n`. If `n` is greater than the current capacity, reallocation occurs, and the capacity also becomes `n` (see footnote for an exception to this rule).[3]
template <**typename** T> **void** std::vector<T>::resize(**size_t** n, T& val);
Same as above, but if `n` is greater than the current size, copies of `val` are added to the end of the vector until the vector size becomes `n` (instead of being value-initialized).

An example using `.resize()` is shown below. A call to `.resize()` actually instantiates data elements in the vector, instead of just allocating more space for the underlying array:

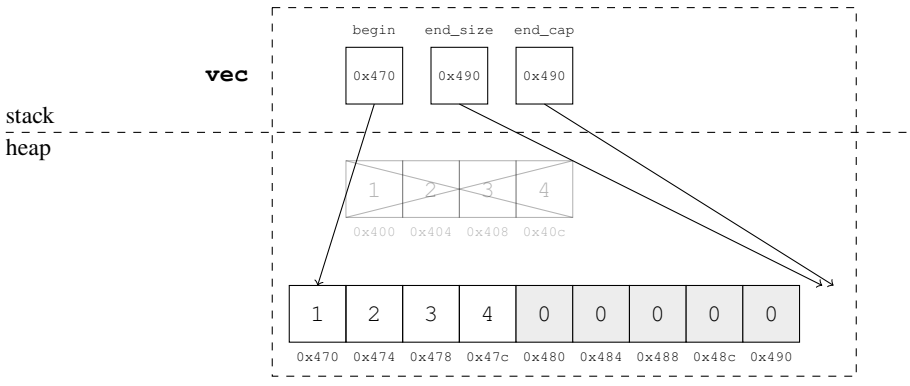
```
std::vector<int32_t> vec = {1, 2, 3, 4};
```



[3] This rule is not entirely true in all cases. For the GCC `std::vector<>` implementation, if `n` is larger than the current size, it turns out that capacity only becomes `n` if `n` > [2×(previous size)]. If `n` < [2×(previous size)], capacity is actually set to [2×(previous size)] instead of `n`. That being said, don't go and memorize these rules, since they are implementation-specific and not explicitly stated in the standard.

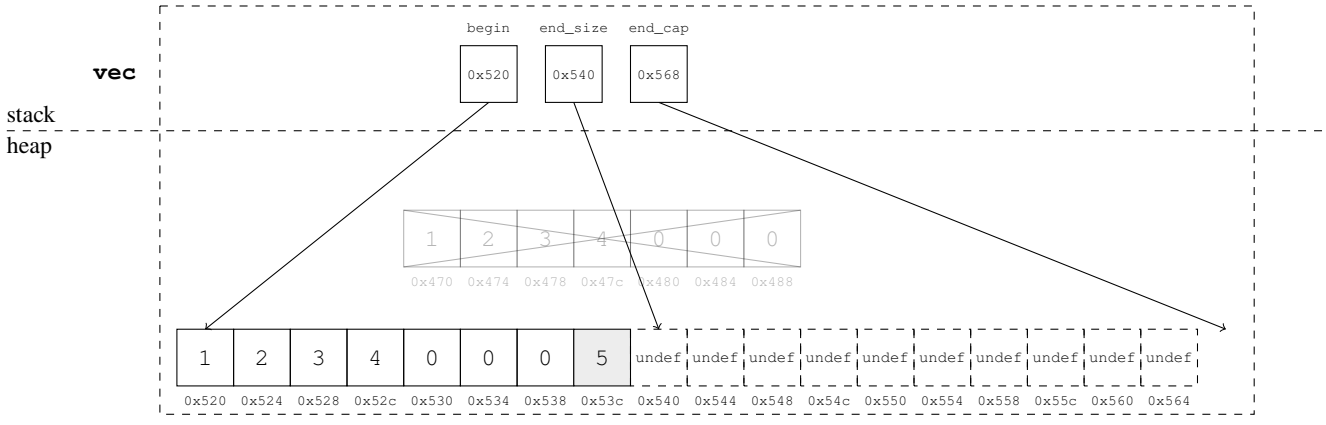
In the example below, the vector is resized to a size of 9 rather than reserved. As a result, the last five elements are value-initialized to 0 (the default behavior for integers), and both size and capacity become 9.

```
vec.resize(9);
```



If we attempt to push back a new element here (e.g., `vec.push_back(5)`), it will not be added to address location `0x480` since an element already exists there (created by `.resize()`). Instead, this element will be added after the ninth element. Since the capacity is only 9, a reallocation must be done to create room for an ninth element. The capacity thus doubles to 18, as shown below.

```
vec.push_back(5);
```



Both `.resize()` and `.reserve()` can be used if you know the size of your vector in advance. Not only can this save memory by ensuring you are only allocating memory you intend to use, it can also speed up your program by reducing the number of reallocations you have to make.

The process of adding new elements differs depending on which approach you use. If you use `.reserve()` to preset the capacity of the vector, you aren't creating any elements. As a result, you should use `.push_back()` to add an element to the back of the vector. On the other hand, if you use `.resize()` to preset the size of the vector, you are *creating* elements in the vector. In this case, using `.push_back()` may result in unintended behavior, since it would add an element *after* the values that were created (and may even trigger a reallocation, as shown above). To add values to a vector that has been resized upon initialization, you will have to modify the data values that were created using `operator[]`. An example is shown below:

```
1  std::vector<int32_t> v1;
2  v1.reserve(5);     // reserve capacity to 5
3  v1.push_back(1);   // add elements to back
4  v1.push_back(2);   // using .push_back()
5  v1.push_back(3);
6  v1.push_back(4);
7  v1.push_back(5);
```

```
1  std::vector<int32_t> v2;
2  v2.resize(5);      // resize size to 5
3  v2[0] = 1;         // add elements by
4  v2[1] = 2;         // modifying the values
5  v2[2] = 3;         // that were created
6  v2[3] = 4;
7  v2[4] = 5;
```

Remark: For a real life analogy of how `.resize()` and `.reserve()` work, imagine a portable refrigerator filled with water bottles for a sporting event. You have to bring the refrigerator to an event, but you are only allowed to bring a single refrigerator. You also need to ensure that you have enough water bottles on hand to satisfy all demand.

In this scenario, the refrigerator represents the underlying heap-allocated array that stores the vector's data. The water bottles represent the data values that are stored. And you, the person carrying the refrigerator around, represents the vector itself — you are responsible for managing the refrigerator and the water bottles inside it. The fact that you can only carry one refrigerator is analogous to the fact that a vector only manages one data array.

When you call `.resize()`, you are changing the number of valid elements in the vector. For example, if you call `vec.resize(50)`, you are setting the number of elements in the vector to 50. This is analogous to adding or removing water bottles from the refrigerator until only 50 water bottles are remaining.

When you call `.reserve()`, you are changing the capacity of the underlying data array, or how many elements the array is able to hold. If you reserve a vector to have a capacity of 50, the vector's underlying array will be allocated to have a capacity of 50. This is analogous to purchasing a refrigerator that can fit a total of 50 water bottles. A call to `.reserve()` does not change a vector's size, much like how

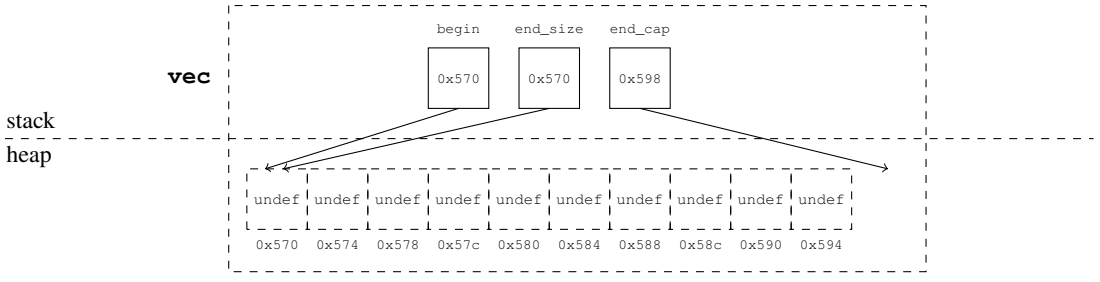
buying a bigger fridge does not change the number of water bottles you have.

If you have more water bottles than you have space in your refrigerator, you will need to buy a larger fridge to be able to hold all your water bottles. This is equivalent to allocating a new, larger array in memory. Then, you'll have to move all of your water bottles from the old fridge to the new fridge, and "discard" the old fridge (i.e., not carry it around anymore using our example). This is similar to the process of copying elements over and deallocating the old array.

Example 7.1 Consider the following code. What does line 3 do? Is the value of `b` on line 4 `true` or `false`?

```
1  std::vector<int32_t> vec;
2  vec.reserve(10);
3  vec[0] = 5;
4  bool b = vec.empty();
```

Line 3 actually results in undefined behavior. When `.reserve()` is called, it does not actually change the contents of the vector itself, just the capacity of the underlying data array. As a result, the vector is still empty, and the indexing on line 3 ends up accessing uninitialized memory (as there is no element at index 0 yet). Similarly, line 4 assigns `b` to `true`, since the vector is indeed empty (its size is 0).



Example 7.2 Consider the following code. What does line 4 print?

```
1  std::vector<int32_t> vec;
2  vec.resize(10);
3  vec.push_back(1);
4  std::cout << vec.size() << '\n';
```

When `.resize()` is called, it changes the size of the array. Thus, line 2 sets the size of the array to 10. Line 3 pushes back an element *after* the 10 elements that were created by `.resize()`, so the size becomes 11 (which is printed out on line 4).

※ 7.3.4 *Accessing a Vector's Underlying Data Array* (✻)

When you use `operator[]`, you end up retrieving an element from the vector by *reference* (the reasoning for this is explained in section 6.7). However, what if you wanted to return a *pointer* to the underlying array that stores the vector's data (e.g., `0x570` using the addresses in example 7.1)? If you are given a vector `vec`, one idea would be to retrieve the address by calling `&vec`. However, this does *not* give you a pointer to the data stored on the heap. Rather, this gives you a pointer to the vector object that lives on the *stack* (remember that the `std::vector<>` object itself, which is composed of three pointers, lives on the stack — this is not the same as the vector's *data*, which lives on the heap).

To retrieve a pointer to the vector's underlying data array, you will have to call the `.data()` member function instead. Given a vector `vec`, calling `vec.data()` would return a pointer to the the underlying data array of `vec`.

```
template <typename T>
T* std::vector<T>::data();
```

Returns a pointer to the vector's underlying data array.

Remark: An alternative would be to use `&vec[0]`, which also returns a pointer to the underlying data array. However, this variant may yield undefined behavior if the vector is empty. This is why `.data()` is preferred, since it is always safe to use regardless of whether or not the vector is empty.

7.4 Storing Multidimensional Data in Vectors

Similar to regular C-style arrays, you can use a vector to store multidimensional data. To create a vector that stores two-dimensional data, simply declare a vector that stores other vectors (i.e., a vector of vectors). Here, the outer vector represents one dimension (e.g., rows) while the inner vectors represent another dimension (e.g., columns).

```
std::vector<std::vector<int32_t>> vec_2d;
```

Remark: For these notes, we will refer to a vector of vectors as a "2-D vector" for succinctness (and a vector of vector of vectors as a "3-D vector", etc.). However, it should be emphasized that a multidimensional vector is merely a one-dimensional vector that stores other vectors. For example, even though we will refer to the above vector as a two-dimensional vector, it is actually represented as a one-dimensional vector of `std::vector<int32_t>` behind the scenes, rather than a two-dimensional vector of `int32_t`.

We can extend this definition to multiple dimensions. To declare a three-dimensional vector, simply create a vector object that stores 2-D vectors (where each layer represents a dimension).

```
std::vector<std::vector<std::vector<int32_t>>> vec_3d;
```

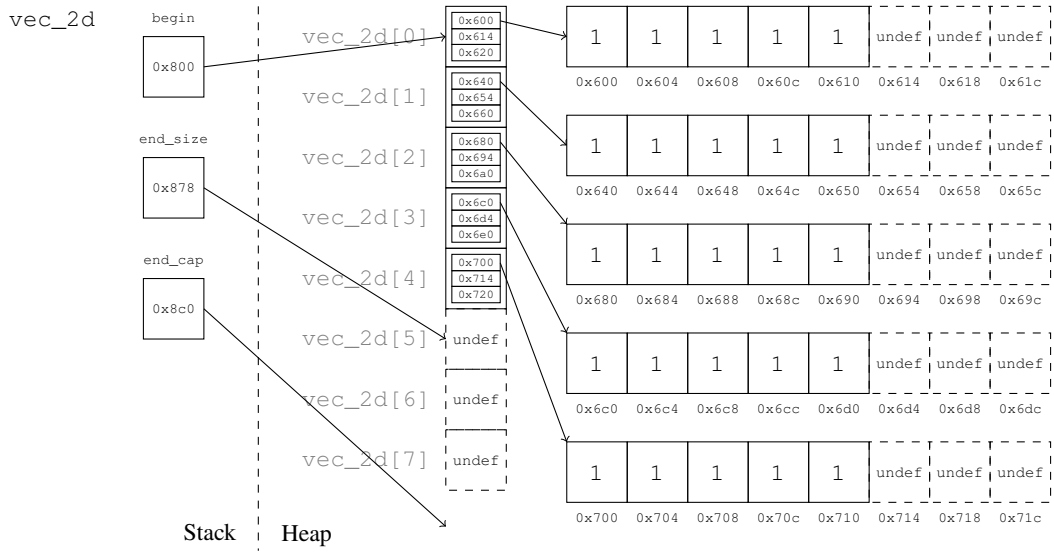
How do we fill the contents of a multidimensional vector? For instance, suppose we wanted to create a 5×5 vector with the following data:

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

A naïve approach would be to use `.push_back()` to insert every element into the 2-D vector. This approach is shown below:

```
1  std::vector<std::vector<int32_t>> vec_2d;
2  while (vec_2d.size() < 5) {
3    std::vector<int32_t> temp;
4    while (temp.size() < 5) {
5      temp.push_back(1);
6    } // while
7    vec_2d.push_back(temp);
8  } // while
```

However, there is an issue with this approach. If we were to look at the vector's underlying memory, we would see something like this (each vector object stores three pointers that each point to a position in its underlying data array):



This ended up happening because we did not specify the size of the vector at the beginning! Thus, the vector automatically doubled with each reallocation, and each dimension ended up having a capacity of 8 instead of 5. To resolve this issue, we must explicitly tell the vector that its size will be 5 using either `.resize()` or `.reserve()`.

```
1  vector<vector<int>> vec_2d;
2  vec_2d.reserve(5);
3  while (vec_2d.size() < 5) {
4    vector<int> temp(5, 1);      // resizes vector with 5 elements, each init to 1
5    vec_2d.push_back(temp);
6  } // while
```

However, this is still a better way to do this. It turns out that a multidimensional vector can be declared on a single line using the vector fill constructor. Recall that a vector declared using the following syntax will have a size of `sz`, where each element is initialized to `val`:

```
std::vector<T> vec(size_t sz, T& val);
```

This is exactly what we did on line 4 of the previous code. When we ran the line `vector<int> temp(5, 1)`, we created a vector of size 5, where each element was initialized to 1. The great thing about the fill constructor is that it also sets the *capacity* to the specified size! Because of this, the fill constructor will allow you to declare a vector whose underlying array has exactly the right capacity (e.g., 5 instead of 8).

The `val` in the above fill constructor syntax dictates what our vector is filled with. In the case of a 2-D vector, we want to create a vector of vectors. As a result, we can declare a 2-D vector by setting `val` equal to the constructor of our inner vector, as shown:

```
std::vector<std::vector<T>> vec_2d(m, std::vector<T>(n, data));
```

Here, `m` represents the size of the outer vector, `n` represents the size of the inner vector, and `data` represents the value each element is initialized to. In other words, we are essentially creating an outer vector of size `m`, where each element of this outer vector is initialized to a separate vector of size `n` whose elements are all initialized to `data`. For instance, we can create the previous 5×5 vector by running the following line of code.

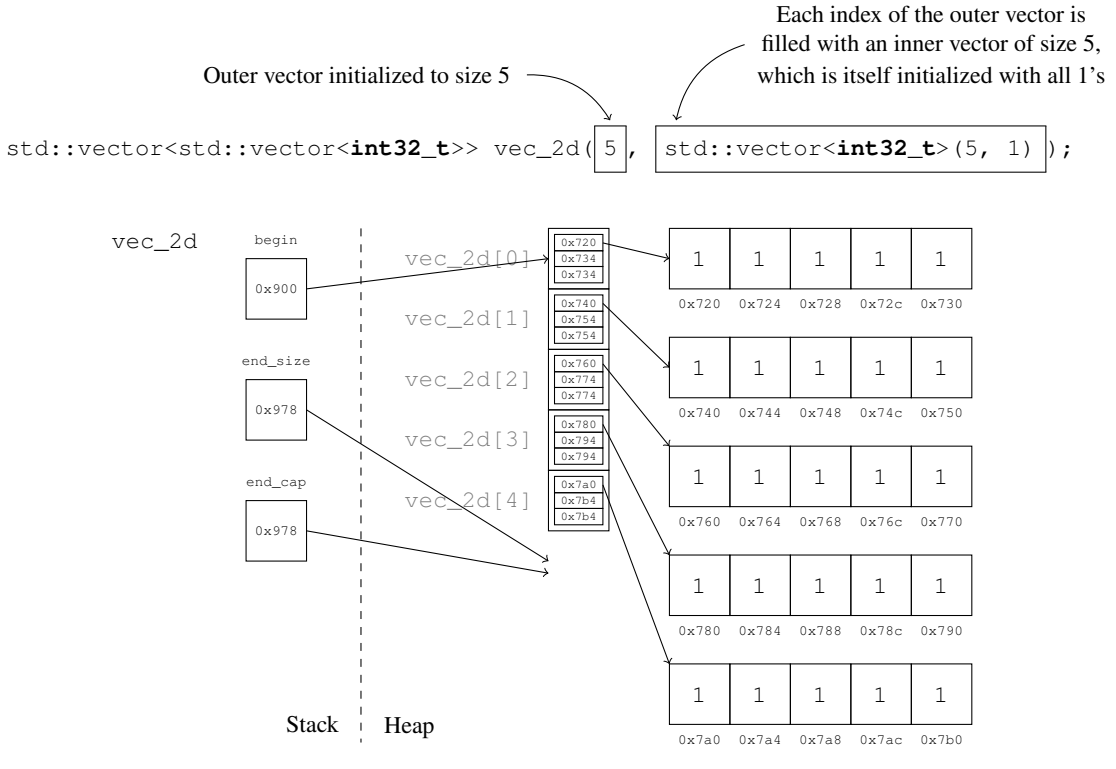
```
std::vector<std::vector<int32_t>> vec_2d(5, std::vector<int32_t>(5, 1));
```

This single line of code does the exact same thing as the previous `while` loop. This is the preferred way of declaring a 2-D vector, since it is clean, efficient, and ensures that memory is not wasted due to excess capacity. The result of constructing the 2-D vector with the correct size is shown at the top of the next page; unlike the vector in the previous diagram, no memory is wasted.

To initialize vectors of multiple dimensions, simply increase the number of nested vectors. For example, if you want to initialize a 3-D vector with dimensions $2 \times 5 \times 10$ with all elements initialized to 1, you can construct it using the following line of code:

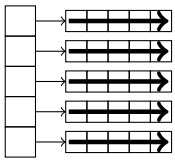
```
std::vector<std::vector<std::vector<int32_t>>> vec_3d(2, std::vector<std::vector<int32_t>>(5, std::vector<int32_t>(10, 1)));
```

To access an element in a multidimensional vector, you can use `operator[]` multiple times (e.g., `vec[2][3]` for the element at row index 2, column index 3 of a 2-D vector). This works because `vec[2]` is also a vector object itself, and thus can be indexed just like `vec`.

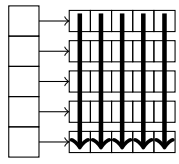


7.5 Vector Performance and Memory Overhead

With a 2-D vector, a common convention is to treat the row dimension as the outer vector and the column dimension as the inner vectors (i.e., each index of the outer vector stores a *row* of data). However, the order in which you define your dimensions could potentially affect the performance of data access. Recall that it is faster to sequentially access memory addresses that are closer in memory than ones that are farther apart. Thus, the dimension that you plan on iterating through first should be given the outermost vector in your vector declaration, if possible.

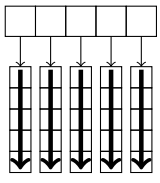


Loop rows, then columns
Faster since memory accesses are contiguous

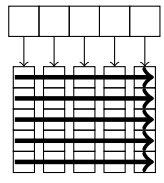


Loop columns, then rows
Slower since memory accesses are not contiguous

Because the row dimension is typically set as the outer dimension, it is faster to loop through a 2-D vector with rows in an outer loop and columns in an inner loop (under the assumption that rows are the outer dimension). However, different types of problems may require different orders of iteration. For instance, if a problem requires you to visit a 2-D vector column-wise from left to right, it may be preferable to store columns as the outer vector and rows as the inner vector. This allows you to take advantage of faster sequential access for memory in close proximity. If you do this, you will have to make sure to flip the indices when using `operator[]` (i.e., `vec_2d[row][col]` becomes `vec_2d[col][row]`).



Storing columns as outer dimension
speeds up column-wise data access



However, this approach would
slow down row-wise data access

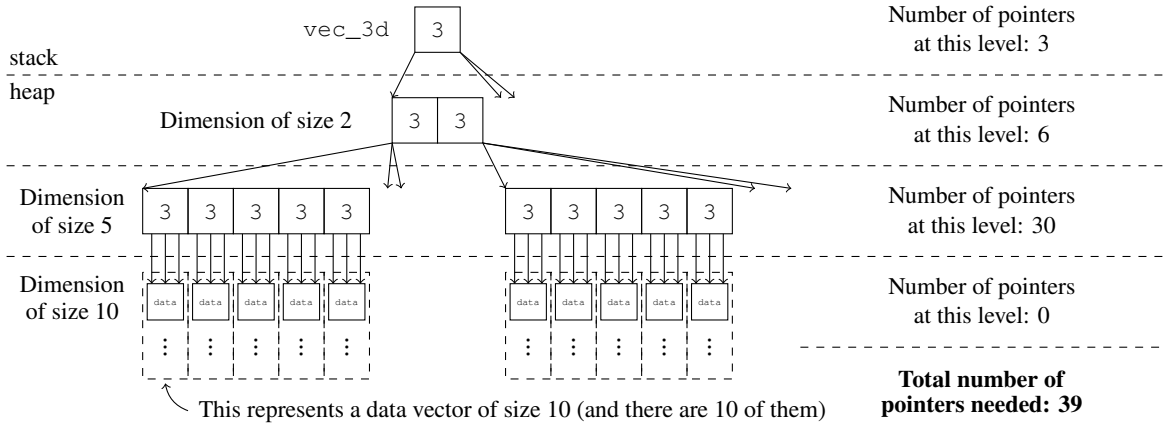
It turns out that the order of dimensions not only affects performance; it may also affect memory usage! This may seem a bit counterintuitive; for instance, if your data has dimensions of $2 \times 5 \times 10$, you would need to store 100 data values in your 3-D vector regardless of whether the outermost dimension is 2 or 5 or 10. Since the number of values you need to store is constant, wouldn't the memory usage be constant as well, regardless of how you declare your vectors?

It turns out that this is not exactly true. Even though the underlying data itself will always take up 400 bytes of memory (assuming the vector is properly resized or reserved), each vector object also stores 3 pointers of bookkeeping data on the stack! The ordering of the dimensions actually changes the total number of pointers needed for our multidimensional vector. Since each pointer takes up 8 bytes of memory on a 64-bit machine, differences in memory usage can quickly accumulate among different dimension orderings.

Let's look at our $2 \times 5 \times 10$ vector again. On the previous page, we declared this vector with 2 as the size of the outermost dimension, 5 as the size of the middle dimension, and 10 as the size of the innermost dimension:

```
std::vector<std::vector<std::vector<int32_t>>> vec_3d(2, std::vector<std::vector<int32_t>>(5, std::vector<int32_t>(10, 1)));
```

How much bookkeeping memory does this use? There is only 1 outer vector, which uses 3 pointers. This outermost vector stores 2 vectors of size 5 (since 2 is the outermost dimension), which use up 2×3, or 6 pointers. These 2 vectors of size 5 each store a vector of size 10 at each index. Thus, there are 2×5 vectors of size 10, which use up a total of $2 \times 5 \times 3$, or 30 pointers. The total number of pointers needed to declare this $2 \times 5 \times 10$ vector is therefore $3 + 6 + 30$, or 39. The below diagram illustrates the number of pointers needed at each level.



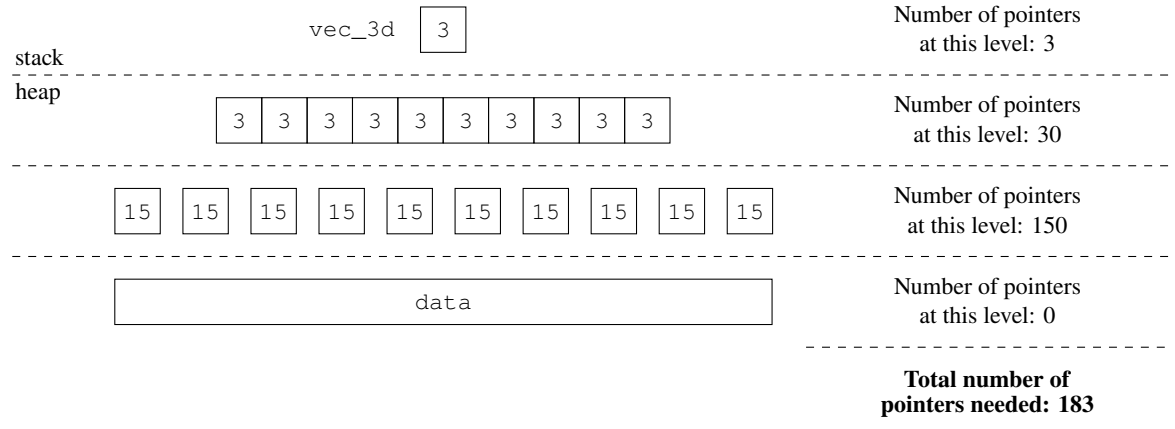
In this diagram, each boxed 3 represents the three pointers of a vector object (i.e., `begin`, `end_size`, and `end_cap`).

The total number of bookkeeping pointers needed for the 3-D vector can be obtained by summing up the number of pointers at each level. The vectors at the lowest layer of the diagram have size 10 (they were not fully drawn out for space reasons), but note that there are no pointers at this level. This is because the vectors at the bottom layer are filled with integers, and not other vector objects.

Let's see what happens when we flip the dimensions around and declare a $10 \times 5 \times 2$ vector instead.

```
std::vector<std::vector<std::vector<int32_t>>> vec_3d(10, std::vector<std::vector<int32_t>>(5, std::vector<int32_t>(2, 1)));
```

Now, our outermost vector stores 10 vectors of size 5, rather than 2 vectors of size 5 as before. This ends up using 10×3, or 30 pointers. These 10 vectors of size 5 each store a vector of size 2 at each index, which end up using $10 \times 5 \times 3$, or 150 pointers. Thus, the total number of pointers needed for this 3-D vector is $3 + 30 + 150$, or 183. This is over 4x the number of pointers we needed before we flipped the dimensions!



In the above diagram, each boxed `15` represents a vector of 5 vector objects (each vector uses 3 pointers, so a vector of 5 vectors uses 15 pointers in total). The arrows were omitted due to space constraints, but each of the 10 vectors in the first heap layer points to a vector of size 5, and each vector within the vectors of size 5 points to a data vector of size 2. Like before, the data layer stores integers and not pointers, so there are 0 pointers at this final level.

In general, if you have a three-dimensional vector with dimensions a, b, and c that is declared as follows, where a is the outermost dimension and c is the innermost dimension:

```
std::vector<std::vector<std::vector<T>>> vec_3d(a, std::vector<std::vector<T>>(b, std::vector<T>(c, VAL)));
```

the number of bookkeeping pointers needed is $3 + 3a + 3ab$. Therefore, when defining a multidimensional vector, the dimensions should be ordered from smallest to largest (i.e., $a < b < c$) to minimize additional memory overhead. This is true for dimensions above 3 as well (which can be proven using a similar process).

7.6 Summary of Vector Complexities

In this section, we will summarize the time complexities of several vector operations.

Operation	Average-Case Time	Worst-Case Time
Finding an element	$\Theta(n)$	$\Theta(n)$

The worst-case of finding an element in a vector of size n occurs when the element is the last one in the vector, or if the element cannot be found at all. This requires the search to look at all n elements in the vector. In the average case, the element is somewhere in the middle of the vector, which would require the algorithm to look at approximately $n/2$ elements. This is still $\Theta(n)$ since we can drop the coefficient term of $1/2$.

In the STL, this can be done using `std::find()` in the `<algorithm>` library, which will be covered in chapter 11.

Operation	Average-Case Time	Worst-Case Time
Accessing first element	$\Theta(1)$	$\Theta(1)$

The vector keeps track of a pointer to the first element in its underlying array. Thus, the first element can always be accessed in constant time. In the STL, this can be done using `std::vector::front()`.

Operation	Average-Case Time	Worst-Case Time
Accessing last element	$\Theta(1)$	$\Theta(1)$

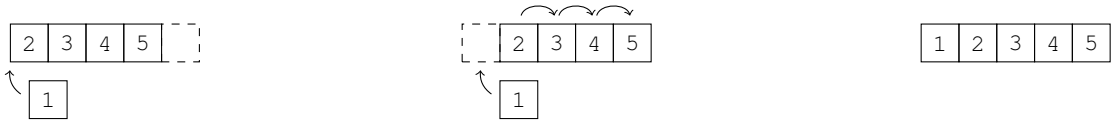
The vector keeps track of a pointer one past the last element in its underlying array. Thus, the last element can always be accessed in constant time. In the STL, this can be done using `std::vector::back()`.

Operation	Average-Case Time	Worst-Case Time
Accessing arbitrary element	$\Theta(1)$	$\Theta(1)$

Since elements in a vector are contiguous in memory, pointer arithmetic can be used to access an element at any index of the vector. Since arithmetic takes constant time, the complexity of accessing an arbitrary element in a vector also takes constant time. In the STL, this can be done using `std::vector::operator[]` or `std::vector::at()` (the latter throws an exception if the given index is out of bounds).

Operation	Average-Case Time	Worst-Case Time
Inserting element at front	$\Theta(n)$	$\Theta(n)$

Since elements in a vector are contiguous in memory, you must shift all elements after the insertion point before you can insert an element. If you attempt to insert an element at the front of a vector of size n, you must shift all n elements after it to the right, which takes $\Theta(n)$ time.



In the STL, this can be done using `std::vector::insert()` or `std::vector::emplace()` with the begin iterator.

Operation	Average-Case Time	Worst-Case Time
Inserting element at back	$\Theta(1)$	$\Theta(n)$

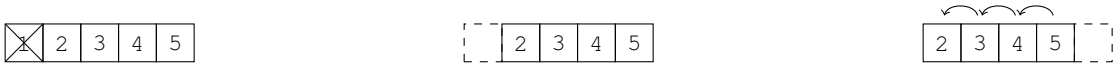
On average, you can insert an element at the back of a vector in constant time, since no elements need to be shifted. However, the worst-case time complexity is $\Theta(n)$ because there's a chance that you will run out of capacity; if this happens, the vector will need to reallocate a new array and copy the existing n elements over. In the STL, this can be done using `std::vector::push_back()` or `std::vector::emplace_back()`.

Operation	Average-Case Time	Worst-Case Time
Inserting element at arbitrary index	$\Theta(n)$	$\Theta(n)$

On average, an insertion will be done in the middle of the vector. Because of this, roughly half of the elements in the vector (i.e., $n/2$ elements) must be shifted to make room for the new element while maintaining the contiguity of elements. This is still considered as $\Theta(n)$ since the coefficient term of ½ is dropped. The worst-case happens if the index is at the front, which requires all n elements to be shifted. In the STL, this can be done using `std::vector::insert()` or `std::vector::emplace()` with an iterator pointing to the position of insertion.

Operation	Average-Case Time	Worst-Case Time
Erasing element at front	$\Theta(n)$	$\Theta(n)$

Much like insertion, erasing an element will require all elements after the erasure point to the shifted to maintain the contiguity of elements. If you attempt to erase an element at the front of a vector of size n, you must shift all n elements after it to the left, which takes $\Theta(n)$ time. This is needed to ensure that the data starts at index 0 and that there are no gaps in the data.



In the STL, this can be done using `std::vector::erase()` with the begin iterator.

Operation	Average-Case Time	Worst-Case Time
Erasing element at back	$\Theta(1)$	$\Theta(1)$

If you erase an element at the back of the vector, you do not need to shift anything — the remaining elements are still in their correct positions. The worst-case of an erase at the back is also $\Theta(1)$. This is because, unlike insertion, erasing an element will never trigger a reallocation on its own (so the vector will never allocate a new array and copy elements over). In the STL, this can be done using `std::vector::pop_back()`.

Operation	Average-Case Time	Worst-Case Time
Erasing element at arbitrary index	$\Theta(n)$	$\Theta(n)$

On average, an erasure will be done in the middle of the vector. This requires roughly half of the elements in the vector to be shifted to remove the gap in the data, which is $\Theta(n)$ after dropping the coefficient of ½. The worst case happens if the index is at the front, which requires all n elements to be shifted. In the STL, this is done using `std::vector::erase()` with an iterator (or iterator range) to the value(s) to erase.

Operation	Average-Case Time	Worst-Case Time
Checking if vector is empty	$\Theta(1)$	$\Theta(1)$

This takes constant time by using the vector's pointers to check if the size is 0. In the STL, this is done using `std::vector::empty()`.