Gorgo Starting C++

April 11, 2026

Contents

8. Containers

In Chapter 2 you used C-style arrays declared with `[]` to store sequences of values, and in Chapters 5 and 6 you wrote loops and functions to work with them. But C-style arrays have serious problems: they do not know their own size, they silently decay to pointers when passed to functions (losing size information), they cannot be returned from functions, and they cannot grow or shrink at runtime. You end up passing a separate size parameter everywhere and hoping nothing goes out of bounds. The standard library provides **containers** — classes that manage collections of elements and solve all of these problems. They know their size, they can be passed to and returned from functions safely, and some can resize dynamically. In this chapter you will learn `std::array` for fixed-size collections, `std::vector` for dynamic collections, and how to iterate through both using range-based for loops and iterators.

std::array

`std::array` fixes the C-style array problems described above. It is a fixed-size container that wraps a C-style array and gives it a proper interface. To use it, include the `<array>` header.

Here are the key methods you will use most often:

```
T& operator[](size_t pos);          // access element, no bounds checking
// access element, throws if out of range
T& at(size_t pos);
constexpr size_t size() const;      // number of elements
```

These work the same way as the string methods you saw in Chapter 3, but now they operate on whatever type `T` the array holds.

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

int main()
{
    std::array<int, 5> scores = {90, 85, 92, 88, 76};

    std::cout << "First score: " << scores[0] << "\n";
    std::cout << "Number of scores: " << scores.size() << "\n";

    return 0;
}
```

```
First score: 90
Number of scores: 5
```

The declaration `std::array<int, 5>` creates an array that holds exactly 5 `int` values. The type and the size are both part of the type — a `std::array<int, 5>` is a different type from a `std::array<int, 10>`.

Accessing Elements

You can access elements in two ways:

```
std::array<std::string, 2> songs = {"Wannabe", "No Diggity"};

// Using [] --- no bounds checking, fast
std::cout << songs[0] << "\n";    // Wannabe

// Using .at() --- bounds checking, safer
std::cout << songs.at(1) << "\n"; // No Diggity
```

Tip: Use `.at()` while developing and debugging. It throws an exception if the index is out of range, which immediately tells you something is wrong. The `[]` operator does not check bounds — accessing an invalid index is undefined behavior.

Why std::array Over C-Style Arrays?

A `std::array` knows its own size. You can pass it to a function and the function knows how many elements it contains:

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

void print_scores(const std::array<int, 3>& scores)
{
    std::cout << "There are " << scores.size() << " scores\n";
    for (size_t i = 0; i < scores.size(); i++) {
        std::cout << "  " << scores[i] << "\n";
    }
}

int main()
{
    std::array<int, 3> my_scores = {95, 87, 91};
    print_scores(my_scores);
    return 0;
}
```

```
There are 3 scores
  95
  87
  91
```

With a C-style array you would have to pass the size as a separate parameter. With `std::array`, the size is built in.

Trap: The size of a `std::array` must be known at compile time. You cannot write `std::array<int, n>` where `n` is a variable. If you need a size determined at runtime, use `std::vector` instead.

std::vector

`std::vector` is the workhorse container of C++. It is a dynamic array that can grow and shrink as needed. To use it, include the `<vector>` header:

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

int main()
{
    std::vector<int> numbers = {10, 20, 30};

    std::cout << "Size: " << numbers.size() << "\n";
    std::cout << "First: " << numbers[0] << "\n";
```

```
    return 0;
}
```

```
Size: 3
First: 10
```

Creating Vectors

There are several ways to create a vector:

```
std::vector<int> empty;                    // empty vector
std::vector<int> zeros(5);                 // 5 elements, all 0
std::vector<int> fives(5, 42);             // 5 elements, all 42
// initializer list
std::vector<std::string> songs = {"Wannabe", "No Diggity"};
```

The **initializer list** syntax with {} is the most common way to create a vector with specific values.

Adding and Removing Elements

The power of `std::vector` is that it can grow. The two main methods for changing a vector's contents are:

```
void push_back(const T& value);  // add element to the end
void pop_back();                 // remove last element
```

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> playlist;

    playlist.push_back("Wannabe");
    playlist.push_back("No Diggity");

    std::cout << "Playlist has " << playlist.size() << " songs\n";

    playlist.pop_back();  // removes the last element

    std::cout << "After pop: " << playlist.size() << " songs\n";
    std::cout << "Remaining: " << playlist[0] << "\n";

    return 0;
}
```

```
Playlist has 2 songs
After pop: 1 song
Remaining: Wannabe
```

`push_back` adds an element to the end of the vector. `pop_back` removes the last element.

Trap: Calling `pop_back()` on an empty vector is undefined behavior. Always check that the vector is not empty first using `.empty()` or `.size()`.

Accessing Elements

Vectors provide multiple ways to access elements. Like `std::array`, vectors support `operator[]` and `.at()` for indexed access. They also provide methods for the first and last elements:

```
T& front();    // first element
T& back();     // last element
```

```
std::vector<std::string> bands = {"Spice Girls", "Blackstreet", "Oasis"};

std::cout << bands[0] << "\n";        // Spice Girls --- no bounds check
std::cout << bands.at(1) << "\n";     // Blackstreet --- bounds checked
std::cout << bands.front() << "\n";   // Spice Girls --- first element
std::cout << bands.back() << "\n";    // Oasis --- last element
```

Just like with `std::array`, `.at()` throws an exception on an invalid index while `[]` does not check.

Size, Capacity, and Empty

A vector tracks two things: its **size** (how many elements it holds) and its **capacity** (how much memory it has allocated). You already know `.size()` — the `.capacity()` method tells you how much room has been allocated:

```
// number of elements held without reallocating
size_t capacity() const;
```

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

int main()
{
    std::vector<int> v;

    std::cout << "Size: " << v.size()
              << ", Capacity: " << v.capacity() << "\n";

    for (int i = 0; i < 5; i++) {
        v.push_back(i * 10);
        std::cout << "Size: " << v.size()
                  << ", Capacity: " << v.capacity() << "\n";
    }

    return 0;
}
```

The output will look something like:

```
Size: 0, Capacity: 0
Size: 1, Capacity: 1
Size: 2, Capacity: 2
Size: 3, Capacity: 4
Size: 4, Capacity: 4
Size: 5, Capacity: 8
```

Notice that capacity grows in jumps. When the vector runs out of room, it allocates a larger block of memory (typically doubling) and copies everything over. This means `push_back` is usually fast, but occasionally it has to do extra work.

The `.empty()` method returns `true` if the vector has no elements:

```cpp
bool empty() const;  // true if size() == 0
```

```cpp
if (v.empty()) {
    std::cout << "Nothing here\n";
}
```

To remove all elements, use `.clear()`:

```cpp
void clear();  // remove all elements, size becomes 0
```

```cpp
v.clear();
std::cout << "Size after clear: " << v.size() << "\n";  // 0
```

Wut: After calling `.clear()`, the size is 0 but the capacity is unchanged. The vector still holds onto its allocated memory. This is intentional — if you are going to refill it, there is no point in freeing the memory just to reallocate it.

Inserting, Erasing, and Reserving

`push_back` and `pop_back` work at the end of the vector. Sometimes you need to insert or remove elements at other positions.

The `.insert()` method inserts an element before a given position. The `.erase()` method removes an element (or a range of elements) at a given position. Both take a position expressed as an **iterator** — you will learn more about iterators in the next section, but for now, `vec.begin() + n` means "the position at index n":

```cpp
iterator insert(const_iterator pos, const T& value);
iterator erase(const_iterator pos);
iterator erase(const_iterator first, const_iterator last);
```

```cpp
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> lista = {"Creep", "No Rain", "Linger"};

    // Insert "Possum Kingdom" at index 1 (before "No Rain")
    lista.insert(lista.begin() + 1, "Possum Kingdom");

    // Erase the element at index 0 ("Creep")
    lista.erase(lista.begin());

    for (const auto& s : lista) {
        std::cout << s << "\n";
    }

    return 0;
}
```

```
Possum Kingdom
No Rain
Linger
```

Trap: Inserting or erasing elements can **invalidate** existing iterators, pointers, and references to elements in the vector. After an insert or erase, do not use any iterators you obtained before the operation — get fresh ones.

Inserting and erasing at positions other than the end are slower than `push_back` and `pop_back` because elements must be shifted in memory. If you need frequent insertion and removal in the middle, other containers may be more efficient.

Two methods let you manage capacity explicitly:

```
void reserve(size_t new_cap);   // preallocate memory without adding elements
// request that capacity be reduced to match size
void shrink_to_fit();
```

`reserve` is useful when you know how many elements you will add. It avoids the repeated reallocations that happen when a vector grows one element at a time:

```
std::vector<int> v;
v.reserve(1000);              // allocate room for 1000 ints
// v.size() is still 0, but v.capacity() is at least 1000
```

`shrink_to_fit` is a non-binding request — the implementation is allowed to ignore it.

You can also construct a vector from a range of iterators, copying elements from another container or a subrange:

```
std::vector<int> all = {10, 20, 30, 40, 50};
std::vector<int> middle(all.begin() + 1, all.begin() + 4);
// middle is {20, 30, 40}
```

Iterating Through Containers

Now that you have containers with data in them, you need to loop through them. C++ gives you several ways to do this.

Range-Based For Loop

The simplest and most modern way to iterate is the **range-based for loop**:

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> songs = {"Wannabe", "No Diggity"};

    for (const auto& song : songs) {
        std::cout << song << "\n";
    }

    return 0;
}
```

```
Wannabe
No Diggity
```

The syntax `for (const auto& song : songs)` means "for each element in `songs`, call it `song`." The `const auto&` part means:

- `auto` — let the compiler figure out the type
- `&` — use a reference so the element is not copied
- `const` — promise not to modify the element

If you want to modify the elements, drop the `const`:

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

for (auto& v : values) {
    v *= 10;  // modify each element in place
}
// values is now {10, 20, 30, 40, 50}
```

Tip: Use `const auto&` when you only need to read elements. Use `auto&` when you need to modify them. Avoid plain `auto` (without `&`) for anything larger than a primitive type — it makes a copy of each element, which is wasteful.

Using Iterators

Under the hood, the range-based for loop uses **iterators**. An iterator is an object that points to an element in a container, similar to how a pointer points to a memory address.

Every container provides `.begin()` and `.end()`:

```
iterator begin();  // iterator to the first element
iterator end();    // iterator to one past the last element
```

- `.begin()` returns an iterator pointing to the first element
- `.end()` returns an iterator pointing to one past the last element

```
#include <iostream>
#include <string>
#include <vector>

int main()
{
    std::vector<std::string> canciones = {"Wannabe", "No Diggity"};

    for (auto it = canciones.begin(); it != canciones.end(); ++it) {
        std::cout << *it << "\n";
    }

    return 0;
}
```

```
Wannabe
No Diggity
```

The `*it` syntax **dereferences** the iterator to get the element it points to, just like dereferencing a pointer. The `++it` moves the iterator to the next element.

Wut: `.end()` does not point to the last element — it points to one *past* the last element. This is a deliberate design choice in C++ called a "half-open range." It means the valid range is `[begin, end)` in mathematical notation. This makes loops cleaner and avoids off-by-one errors.

You might wonder why you would use iterators directly when the range-based for loop exists. Some algorithms in the standard library require iterators, and iterators give you more control when you need to do things like erase elements while iterating or iterate backward.

auto with Iterators

Without `auto`, iterator types can be verbose:

```
// Without auto --- quite a mouthful
std::vector<std::string>::iterator it = canciones.begin();

// With auto --- much cleaner
auto it = canciones.begin();
```

This is one of the best uses of `auto`. The type is obvious from context, and `auto` saves you from writing out the full iterator type.

Try It: Container Starter

Here is a program that exercises vectors and iteration. Type it in, compile it, and experiment:

```
#include <array>
#include <iostream>
#include <string>
#include <vector>

int main()
{
    // std::array
    std::array<int, 4> fixed = {10, 20, 30, 40};
    std::cout << "Array size: " << fixed.size() << "\n";
    std::cout << "Element 2: " << fixed.at(2) << "\n\n";

    // std::vector
    std::vector<std::string> lista;
    lista.push_back("Wannabe");
    lista.push_back("No Diggity");

    std::cout << "Playlist:\n";
    for (const auto& song : lista) {
        std::cout << "  " << song << "\n";
    }
    std::cout << "\n";

    // Modify elements
    std::vector<int> nums = {1, 2, 3, 4, 5};
    for (auto& n : nums) {
        n *= 2;
    }
    std::cout << "Doubled: ";
    for (const auto& n : nums) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    // Size vs. capacity
```

```
    std::vector<int> v;
    for (int i = 0; i < 8; i++) {
        v.push_back(i);
        std::cout << "size=" << v.size()
                  << " cap=" << v.capacity() << "\n";
    }

    return 0;
}
```

Key Points

- `std::array<T, N>` is a fixed-size container that knows its size and works with standard library algorithms. The size `N` must be a compile-time constant.
- `std::vector<T>` is a dynamic container that grows and shrinks as needed using `push_back()` and `pop_back()`.
- Use `.at()` for bounds-checked access and `[]` for unchecked access.
- Use `.front()` and `.back()` to access the first and last elements.
- A vector's **capacity** is the amount of memory it has allocated; its **size** is how many elements it actually holds.
- `insert()` and `erase()` modify a vector at any position but invalidate existing iterators. `reserve()` preallocates memory; `shrink_to_fit()` releases unused memory.
- The range-based for loop (`for (const auto& x : container)`) is the simplest way to iterate.
- Iterators provide lower-level access to container elements using `.begin()` and `.end()`.
- Use `auto` to avoid writing verbose iterator types.

Exercises

1. **Think about it:** Why does `std::array` require the size as part of its type (e.g., `std::array<int, 5>`) while `std::vector` does not? What trade-off does this create?

2. **What does this print?**

   ```
   std::vector<int> v = {10, 20, 30};
   v.push_back(40);
   v.pop_back();
   v.pop_back();
   std::cout << v.size() << " " << v.back() << "\n";
   ```

3. **Calculation:** If a `std::vector<int>` has a capacity of 8 and a size of 5, how many more elements can you `push_back` before it needs to reallocate memory?

4. **Where is the bug?**

   ```
   std::vector<int> scores;
   scores.push_back(95);
   scores.push_back(87);
   scores.push_back(91);

   for (int i = 0; i <= scores.size(); i++) {
       std::cout << scores[i] << "\n";
   }
   ```

5. **What does this print?**

   ```
   std::array<int, 4> a = {5, 10, 15, 20};
   for (auto it = a.begin(); it != a.end(); ++it) {
   ```

```
    std::cout << *it << " ";
}
std::cout << "\n";
```

6. **Think about it:** The range-based for loop `for (auto x : vec)` (without `&`) works, but why is it generally a bad idea for vectors of strings? When would it be acceptable?

7. **Where is the bug?**

```
std::vector<std::string> playlist = {"Wannabe", "No Diggity"};
std::cout << playlist.at(2) << "\n";
```

8. **Calculation:** A `std::vector<double>` contains 3 elements and has a capacity of 4. You call `push_back` 5 times. After all 5 calls, what is the size? Assuming the capacity doubles when exceeded, what is the capacity?

9. **What does this print?**

```
std::vector<int> v = {1, 2, 3};
v.clear();
std::cout << v.size() << " " << v.empty() << "\n";
```

10. **Write a program** that asks the user to enter numbers one at a time (enter -1 to stop), stores them in a `std::vector<int>`, and then prints them in reverse order using iterators or indexing.

11. **What does this print?**

```
std::vector<int> v = {10, 20, 30, 40, 50};
v.insert(v.begin() + 2, 25);
v.erase(v.begin());
for (const auto& n : v) {
    std::cout << n << " ";
}
std::cout << "\n";
```

12. **Calculation:** What is the size and capacity after calling `reserve(100)` on an empty `std::vector<int>`, then calling `push_back` 3 times?

13. **Where is the bug?**

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

int main()
{
    std::vector<int> v = {1, 2, 3, 4, 5};
    for (auto it = v.begin(); it != v.end(); ++it) {
        if (*it == 3) {
            v.push_back(99);
        }
    }
    for (int n : v) std::cout << n << " ";
    std::cout << "\n";
    return 0;
}
```

    The program may crash, may print garbage, or may even appear to work depending on the compiler. What is going on, and how would you fix it?

14. **Think about it:** A function takes three coordinates as a `std::array<double, 3>`:

```
double length(const std::array<double, 3> &v);
```

Why does the size `3` appear in the parameter type, and what would happen if a caller passed a `std::array<double, 4>` instead? Compare this to a function that takes `const std::vector<double> &v` — which one is more flexible, and which one gives stronger compile-time guarantees?

15. **Calculation:** Start with an empty `std::vector<int>` and call `reserve(8)`. Then call `push_back` 12 times. What are `size()` and `capacity()` after each of those 12 calls? (Assume the implementation doubles the capacity when it has to grow.) On which `push_back` calls (if any) are existing iterators into the vector invalidated?