ENPM702

Introductory Robot Programming

L4: STL Containers

v2.1

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED
GRADUATE ENGINEERING

Table of Contents

Changelog

- **v2.1**: In some instances, changed “object” to “object variable”.
- **v2.0**: Added sections on arrays and vectors.
- **v1.0**: Original version.

Learning Objectives

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

- Declare, initialize, and manipulate `std::string` objects using common methods for access, modification, concatenation, and insertion operations.
- Describe the internal memory model of `std::string`, including Small String Optimization (SSO), the relationship between size and capacity, and dynamic memory allocation strategies.
- Manage `std::string` memory usage efficiently using `reserve()`, `resize()`, and `shrink_to_fit()` methods.
- Compare and contrast **C**-style arrays with `std::array`, understanding compilation requirements and safety considerations.
- Declare, initialize, and manipulate both single and multidimensional arrays using proper syntax and access patterns.
- Understand the dynamic nature of `std::vector`, including its memory management, capacity growth strategies, and performance characteristics.
- Perform efficient insertion and deletion operations on vectors using `push_back()`, `emplace_back()`, `insert()`, `erase()`, and related methods.
- Apply best practices for memory management across all STL container types to write efficient and maintainable code.

Overhead

In programming, overhead refers to the additional resources required to perform a task or manage a process beyond the minimal resources needed for the task itself. These resources can include time, memory, bandwidth, or computational power.

The C++ Standard Library

The C++ Standard Library is a collection of classes and functions, which are written in the core language[1]

[1] In the context of the statement, core language refers to the fundamental elements and constructs of the C++ programming language itself (basic features and syntax of the C++ language), excluding libraries and extensions. and part of the C++ ISO Standard itself.

C++ Standard Library

An Essential Toolkit for Programmers

The C++ Standard Library is a vast and powerful collection of pre-written code that provides generic classes and functions for common functionalities, allowing developers to focus on application-specific logic rather than reinventing fundamental components.

🏗️ Standard Template Library (STL)

📦 Containers

Objects that store data: vector, list, deque, set, map

⚡ Algorithms

Functions for operations: sort, find, reverse

🔗 Iterators

Pointer-like objects connecting algorithms and containers

Input/Output Streams

Handle I/O operations with cin, cout, cerr, and file streams (ifstream, ofstream) through the <iostream> header.

Strings

The std::string class provides robust character sequence manipulation with automatic memory management and rich functionality.

Numerics

Mathematical functions, complex numbers, and numerical algorithms through headers like <cmath> and <numeric>.

Language Support

Essential components for C++ language functionality: dynamic memory allocation, type identification, exception handling.

Concurrency

Modern threading tools through <thread>, <mutex>, and <future> for multi-core programming.

C Library Integration

Complete C Standard Library inclusion with 'c' prefixed headers like <cstdio> and <cstdlib>.

STL Containers

An STL container is a class template that manages a collection of objects of a specific type. STL containers handle the complexities of memory management and offer an efficient, standardized way to store and manipulate data.

Categories

- **Sequence Containers** organize a collection of elements in a linear order. The position of an element is determined by the order in which it was inserted, not by its value.
- **Associative Containers** are designed for fast retrieval of elements based on a key. By default, they keep their elements sorted according to a specified comparison function.
- **Unordered Associative Containers** also store elements based on a key but use a hash table for their internal structure. This allows for faster average-case insertion, deletion, and look-up (average constant time, O(1)) but does not keep the elements in any particular order.
- **Container Adapters** are not full-fledged containers themselves but are wrappers that provide a specific, restricted interface to an underlying sequence container.

Resources

- Containers

Why use STL containers?

- Containers are fast, and likely more efficient than anything you are trying to implement.
- Containers share common interfaces.
  - For instance, `std::string`, `std::array`, and `std::vector` have the following member functions in common to access their items: `at()`, `front()`, `back()`, and `[]`.
- Containers are well-documented and easily understood by other developers.

Common Containers

In this lecture we focus on `std::string`, `std::array`, and `std::vector`, which are the most common containers in C++

These three containers are **homogeneous aggregates** as all of their elements are of the same type.

Although this lecture only covers three containers, you should explore the other containers.

Strings

A string is a collection of sequential characters. Strings are used to represent text and are placed between double quotes.

```
std::cout << "hello, world\n";
```

C-style Strings (C-Strings)

C-style strings are not a formal type but a convention inherited from the C language. They are the most fundamental and low-level way to handle strings.

A C-string is a contiguous sequence of characters stored in an array, which is terminated by a special null character (`'\0'`). This null terminator is crucial, as it signals the end of the string.

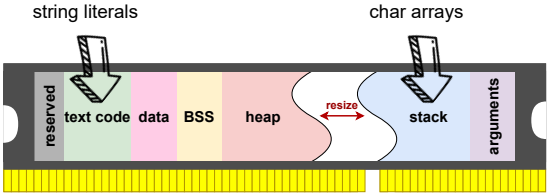
Representations

- **Character Array**: A mutable C-string declared as an array of characters.

```
char my_string[] = "hello"; // Compiler automatically adds '\0' at the end.
// You can stream the character array directly to std::cout
std::cout << my_string << '\n';
```

- **Pointer to a Character**: This is a pointer to the first character of a string. It is often used for string literals, which are immutable and stored in a read-only part of the program's memory.

```
const char* my_literal = "world"; // Points to the first character 'w'
// std::cout knows to print characters starting from the pointer
// until it finds the null-terminator ('\0').
std::cout << my_literal << '\n';
```




C-strings are relatively unsafe and can lead to many bugs.

C++ std::string

This is the standard, modern, and recommended string type in C++

`std::string` (from `<string>`)is a class that manages a dynamic sequence of characters. It encapsulates the complexity of memory management and provides a rich, user-friendly interface.

```
std::string identifier; // declaration
```

- Declared but uninitialized built-in data types (e.g., `int`, `double`) contain garbage values.
- Declared but uninitialized objects from the Standard Library (e.g., `std::vector`, `std::string`) are properly initialized to their default states.

`std::string` is technically not defined as a **container** by the C++ Standard, but in everyday practice, it behaves almost exactly like one.

A true STL container can be instantiated with any type (e.g., `int`, `double`, `Vehicle`). `std::string` is fundamentally designed to store character-like types. You cannot create a `std::string` of integers.

Argument	Why `std::string` fits...	Why it doesn't...
Genericity		It is not a template for any type `T`. It is specifically for character types.
Interface	Provides `begin()`, `end()`, `at()`, etc	It also has many string-specific functions like `c_str()`, `find()`, and `substr()`
STL Algorithms	Works perfectly with algorithms like `std::sort`, `std::for_each`	
Data Structure	Behaves as a contiguous, sequential collection of characters.	

Initialization

There are 9 different ways to initialize a `std::string`.

```cpp
std::string s0("Initial string");   // initialized from a C string

std::string s1;                     // empty string
std::string s2(s0);                 // initialized from another string
std::string s3(s0, 8, 3);           // "str"
std::string s4("Another character sequence", 12);  // "Another char"
std::string s5a(10, 'x');           // "xxxxxxxxxx"
std::string s5b(10, 42);            // "**********"
std::string s6(s0.begin(), s0.begin()+7);   // "Initial"
```

String Literals

`using namespace std::literals::string_literals` allows us to use the suffix `s` in a string literal. This suffix tells the compiler that the string literal is a `std::string` and not a C-string.

```
auto greeting1{"hello"};         // C string
std::cout << typeid(greeting1).name() << '\n';  // C string literal

auto greeting2{"hello"s};        // C++ string
std::cout << typeid(greeting2).name() << '\n';  // C++ string literal

std::cout << "hello" << '\n';  // C string literal
std::cout << "hello"s << '\n';  // C++ string literal
```

What is the point of using `s` in the last line?

Length of a String

`length()` or `size()` returns the length of a string. As per the documentation, these are just synonyms.

```cpp
std::string greeting{"hello"};
std::cout << greeting.length() << '\n';  // 5
// or
std::cout << greeting.size() << '\n';    // 5
```

The return type of `length()` and `size()` is `size_t`.

size_t

`size_t` is the idiomatic, portable, and correct unsigned integer type for representing **sizes**, **counts**, and **indices** of memory-based objects in **C** and **C++**

Characteristics

- **Unsigned**: `size_t` can never be negative. This makes sense because an object cannot have a negative size or a negative number of elements.
- **Platform-Dependent Size**: The most crucial feature of `size_t` is that its actual size (how many bytes it occupies) is platform-dependent.
  - On a 32-bit system, `size_t` is typically a 32-bit unsigned integer (`unsigned int`). Its maximum value is around 4 billion.
  - On a 64-bit system, `size_t` is typically a 64-bit unsigned integer (`unsigned long long`). Its maximum value is around 18 quintillion.

> This adaptability ensures that `size_t` is always large enough to represent the size of the largest possible object that can be created on the host system's architecture.

- **Location in Code**: To use `size_t`, you need to include one of several headers. The most common one is `<cstddef>`. However, it is also made available through other headers like `<vector>`, `<string>`, `<array>`, and more.

The Components of a `std::string` Object

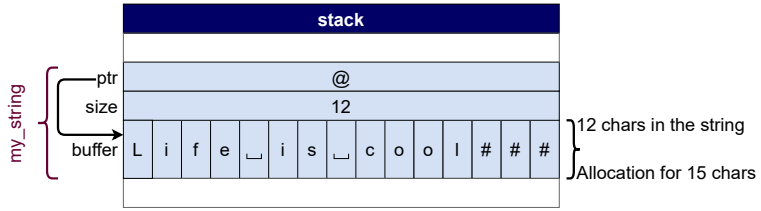
```cpp
std::string my_string{"short string"};
std::cout << sizeof(my_string) << '\n'; // 32
my_string = "This is now a longer string";
std::cout << sizeof(my_string) << '\n'; // 32
my_string = "This is now an even longer string";
std::cout << sizeof(my_string) << '\n'; // 32
```

`sizeof(std::string)` returns a constant value. On a typical 64-bit system, this is often 32 bytes, regardless of whether the string holds `"hello"` or the entire text of a novel.

This is because `sizeof` is a compile-time operator. It tells you the size of the `std::string` management object itself, not the size of the character data it might be managing on the side. The management object is a fixed-size structure that contains the necessary tools to handle the string data.

Example

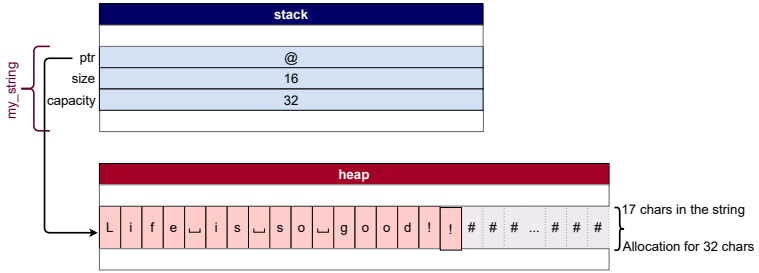
```
std::string my_string{"Life is cool"};
std::cout << sizeof(my_string) << '\n'; // 32
std::cout << my_string.length() << '\n'; // 12
std::cout << my_string.capacity() << '\n'; // 15
```



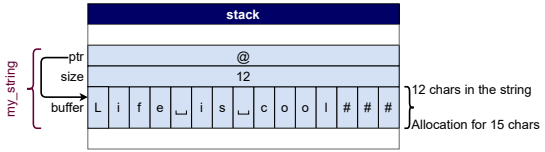
`capacity()` returns the total amount of memory currently allocated for the string, which can be used to store characters. This value is typically larger than or equal to the string's current length (`length()`), which is the actual number of characters in the string.

Example

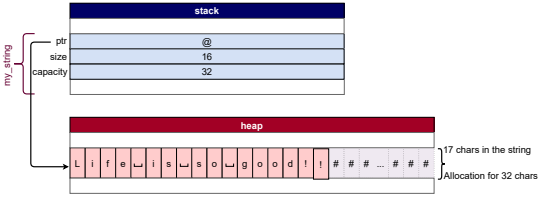
```cpp
std::string my_string{"Life is so good!"};
    std::cout << sizeof(my_string) << '\n';     // 32
    std::cout << my_string.size() << '\n';      // 16
    std::cout << my_string.capacity() << '\n'; // 16
    my_string.append("!");
    std::cout << my_string.size() << '\n';      // 17
    std::cout << my_string.capacity() << '\n'; // 32
```



What do you observe?



VS.



What do you observe?

- A `std::string` variable object has 3 components.

1. **A Pointer to the Character Data**: This is a pointer to a block of memory where the actual characters of your string are stored.
2. **Size**: This is an integer (of type `size_t`) that stores the number of characters currently in the string. This is what the `.size()` or `.length()` member functions return.
3. **Buffer** or **Capacity**.
   - **Buffer**: An array of characters that stores the string characters directly into the memory occupied by the `std::string` object.
   - **Capacity**: An integer (also `size_t`) that stores the total number of characters the currently allocated block of memory on the heap can hold before it needs to be reallocated.

Depending on the length of a string, a `std::string` object **either** contains **buffer** or **capacity**. This is possible because **buffer** and **capacity** are defined in the `union` data structure (see line 179 in basic_string.h).

`union`

- A `union` is a data structure that allows you to store multiple data types in the same memory location. The key characteristic of a `union` is that it can only hold a value for **one** of its member variables at a time, even though it can define multiple members.

  The size of a `union` is determined by the size of its largest member.

- The primary use of `union` is to save memory by allowing different types to be stored in the same location, especially in situations where it is guaranteed that only one type will be used at any given time.

Example

```cpp
union MyUnion{
    int int_value;
    char char_value;
};

int main(){
    // Declare a union variable
    MyUnion data;

    // Store an integer in the union
    data.int_value = 42;
    // Store a char in the union (overwrites the double)
    data.char_value = 'H';

    // Demonstrate that only the last assigned value is valid
    std::cout << "char: " << data.char_value << '\n';
    std::cout << "integer: " << data.int_value << '\n';
}
```

ToDo

Predict the output.

```cpp
union MyUnion{
    int int_value;
    char char_value;
};

int main(){
    // Declare a union variable
    MyUnion data;

    std::cout << "Size of: " << sizeof(data) << '\n';
}
```

What do you observe?

- Depending on the length of the string, it will either be stored on the stack or on the heap.

Small String Optimization (SSO)

Library implementers realized that many, many strings in typical programs are very short (e.g., less than 15 or 22 characters). It is wasteful to perform a heap allocation (which is a relatively slow operation) for every one of these tiny strings.

- **For long strings**: The memory is used to store the pointer, size, and capacity. The string's text lives on the heap.
- **For short strings (the SSO case)**: If the string is small enough, instead of allocating memory on the heap, the characters are stored directly inside the `std::string` object itself, in the space where the pointer and capacity would otherwise have been. A single bit or byte is used as a flag to know which mode the string is in.

What do you observe?

```cpp
std::string my_string(10, 'x');
std::cout << my_string.capacity() << '\n'; // 15
my_string.assign(16, 'x');
std::cout << my_string.capacity() << '\n'; // 30
my_string.assign(32, 'x');
std::cout << my_string.capacity() << '\n'; // 60
```

What do you observe?

- When the size of a `std::string` exceeds its current capacity and a reallocation is required, many implementations of the C++ standard library choose to **double the capacity** (or **increase it by some other factor**) rather than increasing it just enough to accommodate the new size. This strategy is used for a few reasons.

- **Minimize Reallocation Overhead** – Reallocating memory is an expensive operation in terms of both time and system resources. By increasing the capacity by a factor (like doubling), the number of times reallocation is required during the string's lifecycle is reduced.
- **Reduce Memory Fragmentation (see Next Class)** – Frequently reallocating memory in small chunks can lead to memory fragmentation, which can degrade performance. By allocating larger chunks at once (like when doubling capacity), the likelihood of fragmentation is reduced.

Memory Reallocation: Real-life Analogy

Imagine you own a small bookshelf that holds exactly 10 books. You enjoy reading, so every now and then you buy more books. At some point, your bookshelf is full and you want to buy new books.

- **Scenario 1**: Minimal Expansion
  - You could go to the store and buy a new bookshelf that fits exactly one more book each time your shelf gets full. So, if you want to add 1 book, you get a bookshelf that holds exactly 11 books.
  - However, doing this repeatedly becomes inefficient because you constantly need to buy new bookshelves and move your books from the old shelf to the new one. It takes a lot of time and effort every time you want to add just one book.
- **Scenario 2**: Doubling Capacity
  - Instead of buying a shelf that holds only one more book each time, a smarter strategy would be to buy a much larger shelf (one that can hold double the number of books). So, if your current shelf holds 10 books, you buy a shelf that holds 20 books, even though you only need space for 11 books now.
  - This means that the next few times you buy books, you don't need to go through the hassle of buying a new shelf and moving everything over again. You have plenty of room for future purchases.

`std::string` supports the indexing operator `[]` and the method `at()` to access specific characters with indexing starting at 0. `front()` and `back()` return a reference to the first and last character, respectively.

```cpp
// initialization
std::string quote{"Just a flesh wound"};

// access characters
std::cout << quote.front() << '\n'; // return a ref to 1st character
quote.front() = 'j';                // modify the 1st character
std::cout << quote[0] << '\n';      // return the 1st character
std::cout << quote[5] << '\n';      // return the 6th character
std::cout << quote.at(5) << '\n';   // return the 6th character
std::cout << quote.back() << '\n';  // return a ref to the last character
quote.back() = 'D';                 // modify the last character
std::cout << quote.back() << '\n';
quote[6] = 'A';                     // modify the 7th character
quote.at(6) = 'a';                  // modify the 7th character
```

The Indexing Operator

The indexing operator [ ] is not safe to use.

```
std::string greeting{"Hi"};
std::cout << greeting[10] << '\n'; // UB
```

- The operator [ ] **does not perform bounds checking**. Accessing an out-of-bounds element results in **UB**, which could lead to anything from core dump/segmentation fault to silent data corruption.
  - **Core Dump/Segmentation Fault** is an error caused by accessing memory that "does not belong to you" (accessing a variable that has already been freed, writing to a read-only portion of the memory, accessing memory used by another application, etc).

The `at()` Method

The method `at()` performs bounds checking.

```
std::string greeting{"Hi"};
std::cout << greeting.at(10) << '\n'; // Error
```

- If you attempt to access an index that is out of range, `at()` throws an `std::out_of_range` exception. This can be helpful in catching bugs early in development.
  - The use of `at()` guarantees that an `std::out_of_range` exception is raised at runtime.

Which one to use? `[]` or `at()`?

- Because `at()` checks the bounds every time it is called, it can be slightly slower than `[]` when accessing elements. The difference might be negligible in many scenarios, but if performance is critical, this overhead could be a factor to consider.
- `[]` can be faster because it does not perform any checks. But with the benefit of speed comes the responsibility to ensure that you never access out-of-range elements.

Conclusions

Neither method is **universally better**. The preference for `at()` over `[]` (or vice versa) depends on the context in which they are used.

- `at()` offers safety against out-of-range accesses at the expense of a slight overhead.
- `[]` offers performance but leaves the responsibility of ensuring valid access to the programmer.

Example

Use a `for` loop to print each character of the variable `quote`. You can use `[]` or `at()`. The variable `i` in your `for` loop should not exceed the length (or size) of the string.

```
std::string quote{"Just a flesh wound!"};
/*
    write a for loop to print each character of the string
*/
```

Range-based `for` Loop

The range-based `for` loop (often called a range-for loop) was introduced in $\mathbf{C}_{\mathbf{11}}^{++}$ and provides a simpler and more readable way to iterate over all elements in a container.

```
for (declaration: range){
    statements
}
```

- `range` is the container you want to iterate over.
- `declaration` specifies the type and name of a variable that will represent the current element in `range`.
  - `declaration` will take the value of an element of the container in each iteration.

Example

```cpp
std::string quote{"Just a flesh wound!"};

for (char c : quote) {
    std::cout << c << ' ';
}
std::cout << '\n';
```

In each iteration, `c` is assigned a character from `quote`. In the first iteration, `c` is assigned to `'J'`, in the second iteration `c` is assigned to `'u'`, and so on.

Internally, the range-based `for` loop uses **iterators** (or pointer arithmetic for arrays) to traverse the elements in the container. The loop automatically fetches the `begin()` and `end()` iterators of the container and iterates over each element between them.

Performance Overhead

```cpp
std::string quote{"Just a flesh wound!"};

for (char c: quote){
    std::cout << c << '\n';
}
```

In each iteration of the loop, a new variable `char c` is created and initialized with the current character from the `quote` string.

- **Memory usage**: Each copy operation consumes additional memory. For simple data types like `char`, this overhead is minimal, but for complex objects, the memory usage can become significant.
- **Time complexity**: Copying objects takes time. For simple types like `char`, this time is negligible. However, for larger objects or those with complex copy operations, this can add up and slow down the program.


How do we improve the performance of this range-based for loop?

Iterators

An **iterator** is an object in C++ that facilitates the traversal of a container, allowing one to access and potentially modify the elements of the container.

An iterator conceptually behaves like a pointer, and for certain types of containers, iterators are implemented using pointers. However, not all iterators are pointers, and the iterator concept is more generalized than a raw pointer.

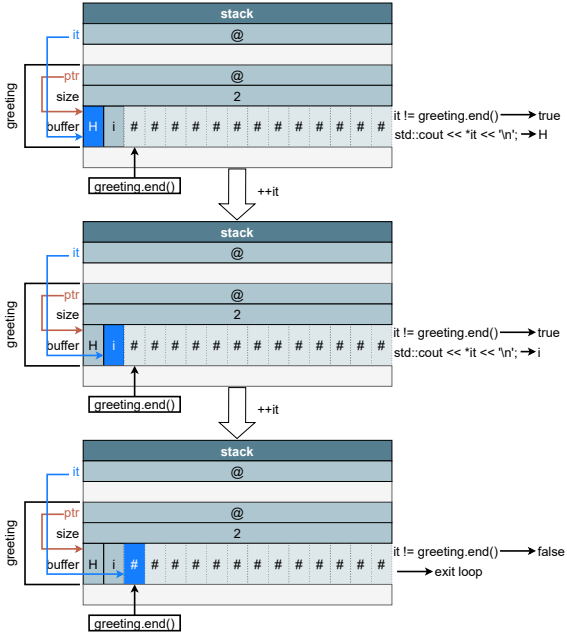
The purpose of iterators is to abstract the way we access and traverse different containers, allowing a unified interface regardless of the underlying data structure.

Various types of iterators are often provided via a container's interface.

- `begin()` returns the beginning position of the container.
- `end()` returns the **past-the-end** of the container.

Example

```
std::string greeting{"Hi"};
for (auto it{greeting.begin()};
     it ≠ greeting.end();
     ++it) {
    std::cout << *it << '\n';
}
```



`std::string` inputs with `std::cin` may lead to unexpected behavior.

```cpp
std::cout << "Enter your full name: ";    //e.g., John Doe
std::string name{};
std::cin >> name;                         // John

std::cout << "Enter your student id: ";   // e.g., 123223LUJ
std::string student_id{};
std::cin >> student_id;                   // Doe

std::cout << "Name: " << name << ", ID: " << student_id << '\n';
```

- `>>` only returns characters up to the first white space it encounters.
- Anything after the white space is left inside `std::cin`, waiting for the next extraction.
- The second `std::cin` will not wait for you to enter anything since it already has a string waiting to be extracted and it will use this one.

std::getline()

`std::getline()` reads characters from an input stream and places them into a string.
`std::getline()` takes two parameters: 1) `std::cin` and 2) a variable to store your string.

```
std::cout << "Enter your full name: ";
std::string name{};
std::getline(std::cin >> std::ws, name);

std::cout << "Enter your student id: ";
std::string student_id{};
std::getline(std::cin >> std::ws, student_id);

std::cout << "Name: " << name << ", ID: " << student_id << '\n';
```

The primary use of `std::ws` (from `<iomanip>`) is when switching between formatted and unformatted input.

- **Formatted input**, i.e., the usual `cin >> value`, skips leading white spaces and stops whenever the format is filled.
- **Unformatted input** , e.g., `std::getline()` does not skip leading white spaces. The use of `std::cin >> std::ws` skips the white spaces and carries on reading where the actual content is entered.

Concatenation and Appending

Concatenation and appending can be used to add characters and strings to another string.

- Using the `+=` operator.

```
std::string s1{"Hello"};
s1 += " World";  // append a string
s1 += '!';       // append a character
```

- Using the `+` operator.

```
std::string s2{"Hello"};
std::string s3{"World"};
std::string s4{s2 + " " + s3};
```

- Using the `append()` method.

```
std::string s5{"Hello"};
s5.append(" World!");
```

- Using the `push_back()` method to add a single character at the end of a string.

```
std::string s6{"Hello"};
s6.push_back('!');
```

Insert Characters or Strings

- Insert using position and string.

```
std::string s1{"Hello World"};
std::string s2{"Big "};
s1.insert(6, s2); // Inserts at position 6
```

- Insert a portion of another string.

```
std::string s3 = "Hello World";
std::string s4 = "Wonderful Big Day";
s3.insert(6, s4, 10, 4); // Inserts "Big" at the 6th position in s3
```

- Insert using iterators.

```
std::string s1 = "Hello World";
std::string s2 = "Beautiful ";
s1.insert(s1.begin() + 6, s2.begin(), s2.end());
```

Remove Characters

- Remove a specific portion of a string.

```
std::string s1{"Hello World"};
s1.erase(s1.size() - 6, 6);  // remove 6 characters starting from index 5
```

- Erase a range of characters using iterators.

```
std::string s2{"Hello World"};
s2.erase(s2.begin() + 6, s2.end());
```

- Erase a single character at a position.

```
std::string s3 = "Hello";
s3.erase(4);
```

- Remove all the characters.

```
std::string s4{"Hello World"};
s4.clear();
```

- Remove the last character.

```
std::string s5{"Hello"};
s5.pop_back();
```

Multiple containers from the C++ Standard Library provide methods to efficiently reallocate memory as needed.

`shrink_to_fit()`

`shrink_to_fit()` is a member function available in C++ containers (e.g., `std::vector`, `std::string`, `std::deque`) that requests the container to reduce its capacity to match its current size, potentially freeing unused memory.

Resources

- shrink_to_fit

Key Points

`shrink_to_fit()` is a non-binding request, not a command. The term non-binding means the request is advisory only. The implementation may choose to ignore it completely.

Why non-binding?

- Performance Considerations: Shrinking might involve expensive memory operations.
- Implementation Freedom: Different standard library implementations can optimize differently.
- Memory Fragmentation: The system might not be able to reclaim the memory effectively.
- Future Growth Anticipation: The implementation might predict the container will grow again soon.

Example

```
std::string s;
std::cout << "Size of std::string is " << sizeof s << " bytes\n"  // ?
          << "Default-constructed capacity is " << s.capacity()   // ?
          << " and size is " << s.size() << '\n';                 // ?

for (int i = 0; i < 42; i++)
    s.append(" 42 ");
std::cout << "Capacity after 42 appends is " << s.capacity()  // ?
          << " and size is " << s.size() << '\n';             // ?

s.clear();
std::cout << "Capacity after clear() is " << s.capacity()  // ?
          << " and size is " << s.size() << '\n';          // ?

s.shrink_to_fit();
std::cout << "Capacity after shrink_to_fit() is " << s.capacity()  // ?
          << " and size is " << s.size() << '\n';  // ?
```

✔ When to use `shrink_to_fit()`

- After significant data removal.
- When memory usage is critical.
- After bulk operations that created temporary capacity.
- Before long-term storage of containers.
- In memory-constrained environments.

✘ When not to use `shrink_to_fit()`

- In tight loops or frequent operations.
- When container will likely grow again soon.
- For small containers (overhead not worth it).
- In performance-critical paths.
- When the capacity difference is minimal.

`resize(<new_size>)`

`resize(<new_size>)` is a direct size management function that changes the number of elements in a container to exactly `new_size`, constructing new elements or destroying existing ones as needed.

Core Purpose

- Direct Size Control: Set exact number of elements immediately.
- Element Management: Automatically construct/destruct elements as needed.
- Initialization: New elements are value-initialized (often zero/default).
- Immediate Effect: Container size changes instantly, not just capacity.

`resize()` changes both size and potentially capacity, affecting actual elements.

Resources

- resize

`new_size > size`

Growing: New elements are constructed at the end.

- Existing elements remain unchanged.
- New elements added at the end.
- May trigger capacity reallocation.
- New elements are value-initialized or use provided value.

`new_size < size`

Shrinking: Excess elements are destroyed from the end.

- Elements destroyed from the end.
- Destructors called for removed elements.
- Capacity usually remains unchanged.
- Iterators to destroyed elements invalidated.

`new_size == size`

No Change: Container remains identical.

- No elements added or removed.
- No constructors or destructors called.
- Capacity unchanged.
- Very efficient - essentially a no-op.

✔ When to use `resize()`

- Need exact number of elements immediately.
- Initializing containers with known sizes.
- Creating buffers or matrices.
- Performance-critical bulk initialization.
- When default/specific value initialization is needed.
- Implementing size-based algorithms.

✘ When not to use `resize()`

- Just need more capacity (use reserve instead).
- Adding elements one by one (use push_back).
- Expensive-to-construct objects without good reason.
- When final size is very uncertain.
- Frequent size changes in performance-critical code.

`reserve(<new_cap>)`

`reserve(<new_cap>)` is a proactive memory management function that informs a container about planned size changes, allowing it to allocate storage appropriately to avoid frequent reallocations.

Core Purpose

- Performance Optimization: Avoid expensive reallocations during growth.
- Memory Efficiency: Reduce memory fragmentation.
- Predictable Behavior: Control when memory allocation happens.
- Iterator Safety: Prevent iterator invalidation during insertions.

`reserve()` only affects capacity, never the number of elements.

Resources

- reserve

`new_cap > capacity`

Guaranteed Behavior: New storage is allocated.

- Memory allocation occurs immediately.
- `capacity` becomes $\geq$ `new_cap`.
- All existing elements are copied/moved.
- Old storage is deallocated.

`size < new_cap < capacity`

Non-binding Request for `std::string`, does nothing for `std::vector`

`new_cap < size < capacity`

Non-binding Request for `std::string`, no-op for `std::vector`

- For `std::string`: This is a non-binding shrink-to-fit request - implementation may reduce capacity to fit the current size.
- For `std::vector`: Complete no-op - nothing happens.

✔ When to use `shrink_to_fit()`

- Known or estimated final size.
- Performance-critical insertion loops.
- Large data loading operations.
- Reducing memory fragmentation.
- Building containers from external sources.

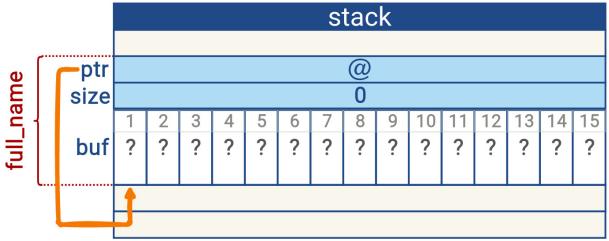
✘ When not to use `shrink_to_fit()`

- Unpredictable or highly variable sizes.
- Small containers (<100 elements).
- Memory-constrained environments.
- When exact size is unknown and might be much smaller.
- Temporary containers with short lifespans.

Demonstration

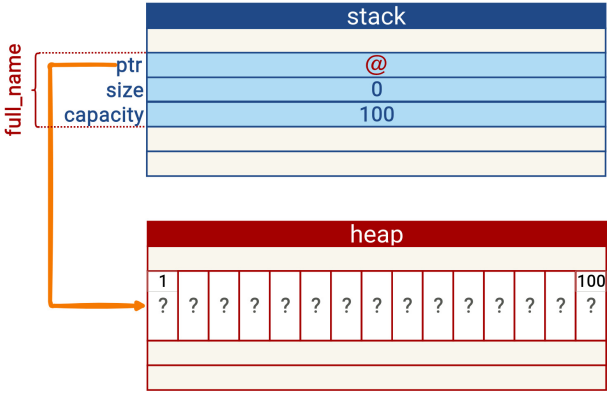
```cpp
std::string full_name;
full_name.reserve(100); // allocate memory for 100 characters

std::cout << "Enter your first name: "; // Albus
std::string first_name;
std::cin >> first_name;
full_name.append(first_name);

std::cout << "Enter your middle name: "; // Percival
std::string middle_name;
std::cin >> middle_name;
middle_name.resize(1);                    // keep only the first letter
full_name.append(" " + middle_name + "."); // add a dot

std::cout << "Enter your last name: "; // Dumbledore
std::string last_name;
std::cin >> last_name;
full_name.append(" " + last_name);

full_name.shrink_to_fit(); // capacity is now equal to size

std::cout << full_name << '\n'; // Albus P. Dumbledore
```

```
std::string full_name;
```



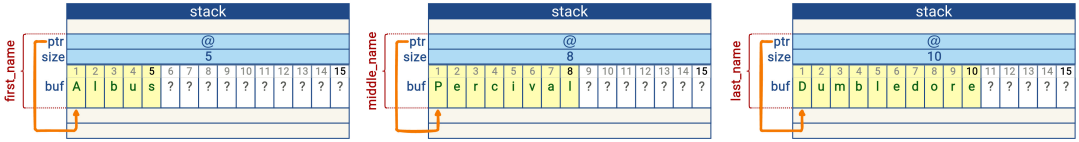
- `full_name` is empty string.

```
full_name.reserve(100);
```



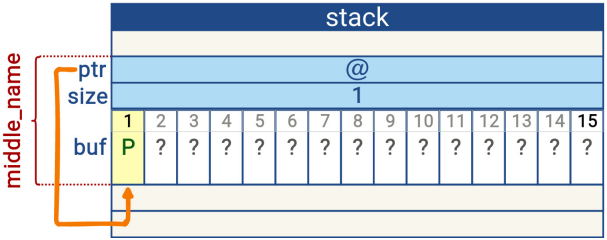
- Memory which can accommodate a string of 100 characters is allocated on the heap.

```cpp
std::cout << "Enter your first name: ";      // Albus
std::string first_name;
std::cin >> first_name;

std::cout << "Enter your middle name: ";     // Percival
std::string middle_name;
std::cin >> middle_name;

std::cout << "Enter your last name: ";       // Dumbledore
std::string last_name;
std::cin >> last_name;
```



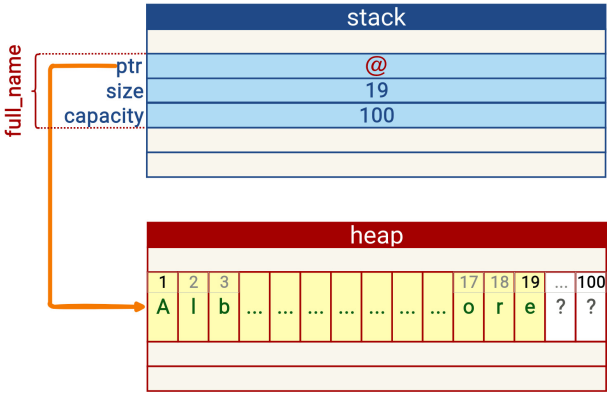
- The size of `first_name`, `middle_name`, and `last_name` can fit the buffer size. SSO is used.

```
middle_name.resize(1);
```



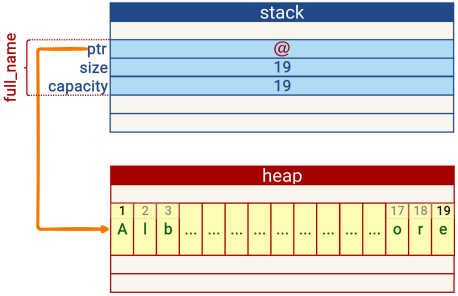
- The size of `middle_name` is resized to 1 character (the first character `'P'`). The other characters after the first character are lost.

```
full_name.append(first_name);
full_name.append(" " + middle_name + ".");
full_name.append(" " + last_name);
```



- The size of `full_name` is now 19 but its capacity is still 100. This extra memory space is not currently used.

```
full_name.shrink_to_fit();
```



The capacity was reduced to match the size of the string, thereby freeing unused memory.

Make sure to use `shrink_to_fit()` only when you are sure you are not going to modify `full_name` in the future. If you append even one character to `full_name`, new memory will be dynamically reallocated to store your original string + this extra character and the old memory will be deallocated. Deallocation and reallocation are expensive.

1. What is the type of the variable `name`?

```
auto name{"John"};
```

2. What header file must be included to use `std::string`?
3. What is the output?

```
std::string str = "Hello";
std::cout << str.at(7) << '\n';
```

4. True or False: A `std::string` variable object is always in the stack segment.
5. True or False: The content of a `std::string` object is always in the heap segment.
6. What does `shrink_to_fit()` do?
7. What does `reserve()` do?
8. What does `resize()` do?
9. How would you rewrite the following code to make `name` an `std::string`?

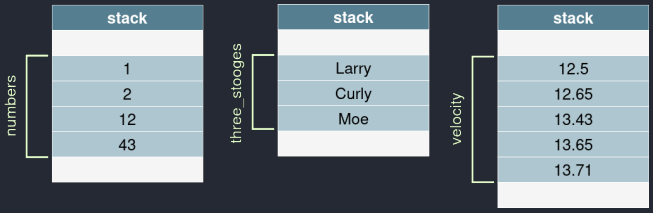
```
auto name{"John"};
```

10. What is the output?

```
std::string greeting{"Hello"};
auto it = greeting.end();
std::cout << *it << '\n';
```

Arrays

An array is a data structure that can store a **fixed-size** sequential collection of items of the same type.



Key Points

- Fixed Size – The size of an array is determined at compile-time and remains fixed throughout its lifetime.
- Contiguous Memory – Elements of an array are stored in contiguous memory locations.
- Homogeneous Elements – All elements in an array are of the same type.

Arrays


Array object variables and their contents are always stored in the stack segment.

ToDo

Inspect the outputs:

```cpp
std::array<int, 5> a1{1, 2, 3, 4, 5}; // [1, 2, 3, 4, 5]
std::array<int, 10> a2{1, 2, 3, 4, 5, 6, 7, 8, 9}; // [1, 2, 3, 4, 5, 6, 7, 8, 9]

std::cout << sizeof(a1) << '\n';
std::cout << sizeof(a2) << '\n';
```

C-style and C++-style Arrays

- **C**-style arrays (C_{array}) – This style of string has its roots in the **C** language but is still prevalent in many parts of **C++** for reasons such as compatibility with C libraries.
- **C++**-style arrays (C^{++}_{array}) – The **C++** style array refers to the `std::array` class provided by the **C++** Standard Library (from `<array>` header). The `std::array` class provides a more high-level, flexible, and safer way to work with arrays compared to C_{array}

You should not notice any difference in runtime performance between C_{array} and C^{++}_{array} arrays while you still get to enjoy the extra features of C^{++}_{array}.

C array

```
type name[size];
```

```
// array of 5 int
int numbers[5];
```

C++ array

```
std::array<type, size> name;
```

```
// array of 5 int
std::array<int, 5> numbers;
```

C array

```
type name[size]{<values>};
```

```
// C array of 2 int
int numbers[2]{1, 2};
```

```
type name[size] = <values>;
```

```
// C array of 2 int
int numbers[2] = {1, 2};
```

C++ array

```
std::array<type, size> name{<values>};
```

```
// C++ array of 2 int
std::array<int, 2> numbers{1, 2};
```

```
std::array<type, size> name = {<values>};
```

```
// C++ array of 2 int
std::array<int, 2> numbers = {1, 2};
```

C array

```cpp
// zero initialization
int a1[2]{}; // [0, 0]
// explicit initialization
int a2[2]{1, 2};
// partial initialization
int a3[2]{1}; // [1, 0]
//size inference
int a4[]{1, 2};

// fill with explicit initialization
// up to but not including
int a5[3]{}; // [0, 0, 0]
std::fill(a5, a5 + 2, 3); //[3, 3, 0]

// fill with partial initialization
// up to but not including
int a6[3]{}; // [0, 0, 0]
std::fill(a6, a6 + 2, 1); //[1, 1, 0]
```

C++ array

```cpp
// zero initialization
std::array<int, 2> a1{}; // [0, 0]
// explicit initialization
std::array<int, 2> a2 = {1, 2};
// partial initialization
std::array<int, 2> a3 = {1};  // [1, 0]
//no size inference


// fill with explicit initialization
// up to but not including
std::array<int, 2> a4;
std::fill(a4.begin(), a4.end(), 3);//[3, 3]

// fill with partial initialization
// up to but not including
std::array<int, 3> a5{};
std::fill(a5.begin(), a5.begin() + 2, 1);//[1, 1, 0]
```

C_{array} and Pointers

The name of an array acts as a pointer to the first element of the array. Pointers can be used to navigate through arrays.

```cpp
int my_array[5]{1, 2, 3, 4, 5};
std::cout << my_array<< '\n'; // @

int* ptr = my_array;
// first element using pointer
int first_number = *ptr;
std::cout << first_number << '\n'; // 1
// second element using pointer arithmetic
int second_number = *(ptr + 1);
std::cout << second_number << '\n'; // 2

for (size_t i{0}; i < std::size(my_array); ++i) {
  std::cout << my_array[i] << ' ';
}
std::cout << '\n';
```

Exercise #1

Print each item of `my_array` using `ptr`.

```cpp
int my_array[5]{1, 2, 3, 4, 5};
int* ptr = my_array;

for (size_t i{0}; i < std::size(my_array); ++i) {
  // write code here
}
```

Array Length

The length of the array **must be a compile-time constant** because the length of a fixed array **must be known at compile time**.

```cpp
constexpr size_t length1{3};
size_t length2{3};

// C++ style array
std::array<int, 3> cpp_array1{};        // ok: 3 is a compile-time constant
std::array<int, length1> cpp_array2{};  // ok: length1 is a compile-time constant
std::array<int, length2> cpp_array3{};  // Error: length2 is not a compile-time
↪ constant

// C-style array
int c_array1[3]{};        // ok: 3 is a compile-time constant
int c_array2[length1]{};  // ok: length1 is a compile-time constant
int c_array3[length2]{};  // Error: length2 is not a compile-time constant
```

```
size_t length2{3};
```

`length2` is a variable and not a compile-time constant. The compiler should raise an error for both lines 7 and 12.

```cpp
std::array<int, length2> cpp_array3{};
```

The implementation of C^{++}_{array} is very strict and prevents the program from compiling.

```cpp
// this should raise an error, but doesn't
int c_array3[length2]{};
```

The implementation of C_{array} will let you use a non compile-time constant, even though the standard specifies that it has to be a compile-time constant.

- The C++ standard defines rules about how programs should behave in specific circumstances and in most cases, compilers will follow these rules.
- However, many compilers implement their own changes to the language, often to enhance compatibility with other versions of the language or for historical reasons.
- These compiler-specific behaviors are called compiler extensions and are (unfortunately) often enabled by default.

-pedantic-errors

The **-pedantic-errors** option is a compiler flag used primarily with the GNU Compiler Collection (GCC) and some other compilers. It instructs the compiler to enforce strict compliance with the C++ (or C) standard and turns any violation of the standard into an error.

ToDo

- Edit CMakeLists.txt to add the **-pedantic-errors** option to your compilation.

```
# Compilation flags
add_compile_options(-Wall -pedantic-errors)
```

- Recompile.

C_{array}

Only `[]` is available for C_{array}

```cpp
constexpr size_t length{3};

int a1[length]{1, 2, 3};
std::cout << a1[0] << '\n'; // 1
a1[0] = 10; // [10, 2, 3]

// regular for loop
for (size_t i{0}; i < length; ++i)
    std::cout << a1[i] << ' ';
std::cout << '\n';

// range-based for loop
for (const auto &item : a1)
    std::cout << item << ' ';
std::cout << '\n';
```

C^{++}_{array}

Both `[]` and `at()` are available for C^{++}_{array}

```cpp
constexpr size_t length{3};

std::array<int, length> a2{10, 20, 30};
std::cout << a2.at(0) << '\n'; // 10
std::cout << a2.empty() << '\n'; // 0 (false)
std::cout << a2.size() << '\n';  // 3
std::cout << a2.front() << '\n'; // 10
std::cout << a2.back() << '\n';  // 30
a2.at(0) = 100; // [100, 20, 30]

// regular for loop
for (size_t i{0}; i < length; ++i)
    std::cout << a2.at(i) << ' ';
std::cout << '\n';

// range-based for loop
for (const auto &item : a2)
    std::cout << item << ' ';
std::cout << '\n';

a2.fill(0); // [0, 0, 0]
```

Multidimensional Array

A **multidimensional array** is an array with more than one level or dimension. For example, a 2D array, or two-dimensional array, is an array of array, meaning it is a matrix of rows and columns (think of a table). A 3D array adds another dimension, turning it into an array of array of array.

C$_{array}$

```
type name [size]...[size]
```

```c
int a1[2][3];      // 2D array
int a2[2][3][4];   // 3D array
```

C++$_{array}$

```
std::array<std::array<...,size>,size> name;
```

```cpp
// 2D array
std::array<std::array<int, 3>, 2>  a1;
// 3D array
std::array<std::array<std::array<int, 2>, 3>, 2>  a2;
```

The total number of items than can be stored in a multidimensional array can be calculated by multiplying the size of each array size.

```cpp
int a1 [2][3]; // 6 items
std::array<std::array<std::array<int, 3>, 3>, 2> a2; // 18 items
```

The two following arrays are the same (2 layers of 3×4 matrix).

```cpp
int a1 [2][3][4];
std::array<std::array<std::array<int, 4>, 3>, 2> a2;
```

2D Arrays

A **2D array** is similar to a table which has x rows and y columns.

x \ y	[0]	[1]	[2]
[0]	numbers[0][0]	numbers[0][1]	numbers[0][2]
[1]	numbers[1][0]	numbers[1][1]	numbers[1][2]

Examples

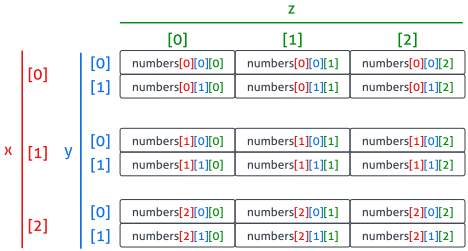
```
int numbers[2][3]{
    {1, 2, 3},  // row 0
    {4, 5, 6}   // row 1
};
std::array<std::array<int, 3>, 2>  numbers {{ // note the double braces
    {1, 2, 3},  // row 0
    {4, 5, 6}   // row 1
}};
```

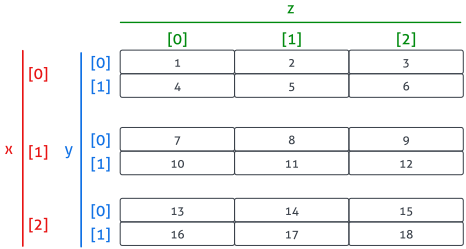
x \ y	[0]	[1]	[2]
[0]	1	2	3
[1]	4	5	6

3D Arrays

A 3D array is an array containing a 2D array (or x **layers of** y×z matrix).

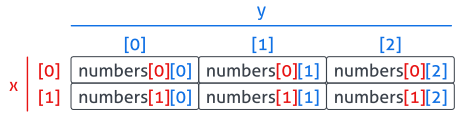
```
int  numbers [3][2][3]{
    {{1, 2, 3}, {4, 5, 6}},
    {{7, 8, 9}, {10, 11, 12}},
    {{13, 14, 15}, {16, 17, 18}}
};
std::array<std::array<std::array<int, 3>, 2>, 3>  numbers {{
    {{{1, 2, 3}, {4, 5, 6}}},
    {{{7, 8, 9}, {10, 11, 12}}},
    {{{13, 14, 15}, {16, 17, 18}}}
}};
```



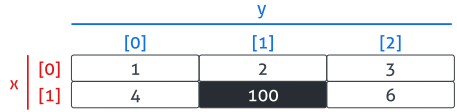


Access/Modification

```cpp
int  numbers [2][3]{
    {1, 2, 3},
    {4, 5, 6}
};
numbers[1][1] = 100;
std::cout << numbers[1][1] << '\n';
```




```cpp
std::array<std::array<int, 3>, 2>  numbers {{
    {1, 2, 3},
    {4, 5, 6}
}};

numbers.at(1).at(1) = 100;
std::cout << numbers.at(1).at(1) << '\n';
```



Quiz

1. True or False: An array can grow or shrink in size.
2. True or False: An `std::array` object variable is always in the stack segment.
3. True or False: The content of an `std::array` object is always in the heap segment.
4. How do you declare an `std::array number` of 5 integers?
5. What header file must be included to use `std::array`?
6. Predict the output.

```
std::array<double, 3> temperature {43.4, 45.3, 55.9};
std::cout << temperature[3] << '\n';
```

7. Predict the outputs.

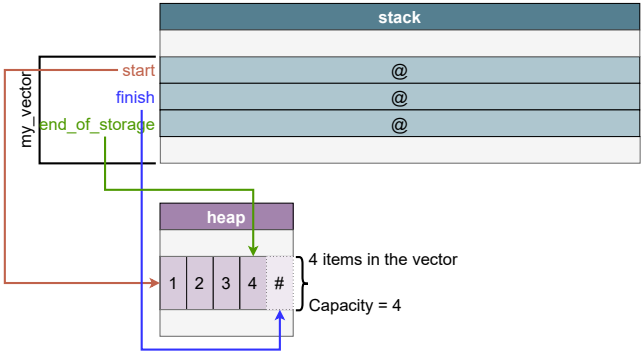
```
std::array<double, 3> temperature;

for (const auto& temp : temperature) {
    temp[0] = 45.0;
}

std::cout << temp[0] << '\n';
```

Vectors

A `std::vector` (from `<vector>`) is a dynamic array provided by the C++ Standard Library. Unlike C_{array} or C^{++}_{array}, which has a fixed size at compile time, the size of a `std::vector` can be changed at runtime, allowing for more flexibility.

The `std::vector` object variable is always in the stack segment and its contents are always in the heap segment.

```
std::vector<int> my_vector{1, 2, 3, 4};
```



Vector Structure

Internally, a `std::vector` object typically manages three main pointers.

- **start** – This pointer points to the beginning of the allocated memory block where the vector elements are stored. This is essentially the address of the first element of the vector.
- **finish** – This pointer points to **one past** the last element currently stored in the vector. It does not point to the end of the allocated memory but rather the end of the used portion of that memory.
- **end_of_storage** – This pointer points to the end of the allocated memory block, marking the boundary after which no more elements can be added without reallocating and resizing the internal storage. It is pointing to the last memory location of the current capacity.

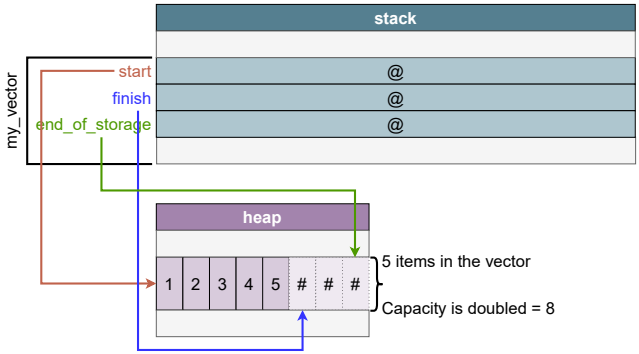
ToDo

Inspect the outputs:

```cpp
std::vector<int> vec1 = {1, 2, 3};
std::vector<int> vec2 = {1, 2, 3, 4, 5};

std::cout << sizeof(vec1) << '\n';
std::cout << sizeof(vec2) << '\n';
```

Similar to `std::string`, when the size of the `std::vector` exceeds the current capacity, the capacity is doubled.

```
std::vector<int> my_vector{1, 2, 3, 4};
my_vector.push_back(5);
```



Dynamic Reallocation

A `std::vector` manages its own dynamic structure internally. This structure is dynamically resized as needed when elements are added or removed.

- **Dynamic Allocation** – A `std::vector` allocates a contiguous block of memory to store its elements. This memory is managed dynamically and can grow or shrink as needed.
- **Automatic Resizing** – When elements are added and the current capacity is exceeded, the `std::vector` automatically allocates a new, larger block of memory, copies the existing elements to this new block, and then deallocates the old block.

ToDo

Predict the outputs.

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

std::cout << "Initial size: " << vec.size() << '\n';
std::cout << "Initial capacity: " << vec.capacity() << '\n';

// Reserve space
vec.reserve(20);
std::cout << "Size: " << vec.size() << '\n';
std::cout << "Capacity: " << vec.capacity() << '\n';

// Add more elements to the vector
for (auto i{0}; i < 60; ++i)
    vec.push_back(i);

// Display updated size and capacity
std::cout << "Size: " << vec.size() << '\n';
std::cout << "Capacity: " << vec.capacity() << '\n';

// Shrink the vector to fit its size
vec.shrink_to_fit();
std::cout << "Size: " << vec.size() << '\n';
std::cout << "Capacity: " << vec.capacity() << '\n';
```

```
std::vector<type> name {values};
//or
std::vector<type> name = {values};
```

```
std::vector<int> v1{-1, 3, 5};    // initialization with explicit values
std::vector<int> v2{};            // zero initialization
std::cout << v2.size() << '\n';   // ???
std::cout << v2.at(0) << '\n';    // ???
v2 = v1;                          // copy items of v1 into v2
auto v3{v1};                      // initialization from an existing vector
```

Access and Modification

```cpp
std::vector<int> numbers{-1, 3, 5, -9};      // initialization with 4 items

std::cout << numbers.front() << '\n';        // -1
std::cout << numbers.back() << '\n';         // -9

// indexing operator (no bounds check)
numbers[0] = -2;                             // [-2, 3, 5, -9]
std::cout << numbers[0] << '\n';             // -2
// at() method
numbers.at(0) = 1;                           // [1, 3, 5, -9]
std::cout << numbers.at(3) << '\n';          // 4th item - bounds check

// regular for loop
for (size_t i{0}; i < numbers.size(); ++i)
    numbers[i] = 0;
std::cout << '\n';

// ranged-base for loop
for (auto &item : numbers)
    item = 1;
std::cout << '\n';

numbers.assign({1, 2, 3, 4});                // replace entire content with new values
```

Insertion: push_back()

- push_back() is used to add a new element to the end of the container.

```cpp
std::vector<int> my_vector;

// Add elements to the vector using push_back()
my_vector.push_back(10);
my_vector.push_back(20);
my_vector.push_back(30);

// Output the elements of the vector
std::cout << "Elements in my_vector: ";
for (size_t i{0}; i < my_vector.size(); ++i) {
    std::cout << my_vector.at(i) << " ";
}
std::cout << '\n';
```

Insertion: `emplace_back()`

- `emplace_back()` constructs a new element in place at the end of the container. This means it directly constructs the element at the end without creating a temporary object, which can improve performance by reducing the number of copy or move operations.

```cpp
// Declare a vector of pairs of integers
std::vector<std::pair<int, int>> pair_vector;

// Add elements to the vector using emplace_back()
pair_vector.emplace_back(1, 2);
pair_vector.emplace_back(3, 4);
pair_vector.emplace_back(5, 6);

// Output the elements of the vector
for (const auto& p : pair_vector) {
    std::cout << "First: " << p.first << ", Second: " << p.second << '\n';
}
```

Insertion: `insert()`

- `insert()` is used to insert elements at a specified position in the container.

```cpp
// Declare and initialize a vector of integers
std::vector<int> my_vector = {1, 2, 3, 5, 6};

// Insert a single element at the 4th position (index 3)
my_vector.insert(my_vector.begin() + 3, 4); // 1 2 3 4 5 6

// Insert multiple elements at the end
my_vector.insert(my_vector.end(), {7, 8, 9}); // 1 2 3 4 5 6 7 8 9

// Insert multiple copies of the same value
my_vector.insert(my_vector.begin(), 3, 0); // 0 0 0 1 2 3 4 5 6 7 8 9
```

Insertion: `emplace()`

- `emplace()` directly constructs the element at the specified position, avoiding unnecessary copy or move operations.

```cpp
std::vector<std::pair<int, std::string>> vec;

// Use emplace() to insert elements
vec.emplace(vec.begin(), 1, "one");
vec.emplace(vec.end(), 3, "three");
vec.emplace(vec.end(), 4, "four");
vec.emplace(vec.begin() + 1, 2, "two");
```

Deletion: pop_back()

`pop_back()` is used to remove the last element from the container. This function reduces the size of the container by one but does not return the removed element.

```cpp
// Declare and initialize a vector of integers
std::vector<int> vec = {1, 2, 3, 4, 5};


// Use pop_back() to remove the last element
vec.pop_back();

// Output the elements of the vector after pop_back()
std::cout << "Elements in vec after pop_back: ";
for (const int& value : vec) {
    std::cout << value << " ";
}
std::cout << '\n';
```

Deletion: `erase()`

`erase()` is used to remove elements from the container.

- Removing a single element. iterator erase(const_iterator pos);
  - `pos`: An iterator pointing to the element to be removed.
  - **Returns**: An iterator pointing to the element that follows the last element removed.
- Removing a range of elements. iterator erase(const_iterator first, const_iterator last);
  - `first`: An iterator pointing to the first element to be removed.
  - `last`: An iterator pointing to one past the last element to be removed.
  - **Returns**: An iterator pointing to the element that follows the last element removed.

Deletion: erase()

```cpp
// Declare and initialize a vector of integers
std::vector<int> vec = {1, 2, 3, 4, 5};

// Remove the element at the 2nd position (index 1)
vec.erase(vec.begin() + 1);

// Remove a range of elements (2nd and 3rd)
vec.erase(vec.begin() + 1, vec.begin() + 3);
```

Deletion: `clear()`

`clear()` is used to remove all elements from the container, effectively resetting its size to zero.

```cpp
// Declare and initialize a vector of integers
std::vector<int> my_vector = {1, 2, 3, 4, 5};

// Clear all elements from the vector
my_vector.clear();

// Output the size of the vector after clear
std::cout << "Size of my_vector after clear: " << my_vector.size() << '\n';

// Check if the vector is empty
if (my_vector.empty()) {
    std::cout << "my_vector is now empty.\n";
} else {
    std::cout << "my_vector is not empty.\n";
```

Quiz

1. What header file must be included to use `std::vector`?
2. What is the initial size (length) of a `std::vector` declared as `std::vector<int> vec;`?
3. What will the following code output?

```
std::vector<int> vec = {10, 20, 30};
vec.pop_back();
std::cout << vec.back() << '\n';
```

4. Which function is used to reduce the capacity of a `std::vector` to fit its size?
5. How do you reserve space for at least 100 elements in a `std::vector` named `vec`?
6. What will the following code output?

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

Next Class

- Quiz#3
- Lecture5: Functions.

Memory Fragmentation

Memory fragmentation refers to a condition in a computer system where the available memory is broken into small, non-contiguous blocks, making it difficult to allocate large chunks of memory, even if enough total free memory exists.

External Fragmentation

External fragmentation occurs when free memory is scattered in small, non-contiguous blocks across the memory space. Over time, as programs and processes dynamically allocate and deallocate memory, these gaps (or holes) are left in between allocated memory blocks. Even though the total amount of free memory might be large enough to satisfy a memory request, the free memory is not contiguous, so large memory allocations fail.

Internal Fragmentation

Internal fragmentation occurs when memory is allocated in blocks that are larger than necessary. When a program requests memory, the system might allocate a fixed-size block (often larger than the requested amount), leaving unused memory inside the block. This wasted space within the allocated block is called internal fragmentation.