Containers

Containers exist in programming, to make it easier to create and manage (potentially large) collections of objects. In general programming, a **container** is a data type that provides storage for a collection of unnamed objects (called **elements**). String is a type of container for a collection of characters. While the container object itself typically has a name (otherwise how would we use it?), the elements of a container are unnamed. This is so that we can put as many elements in our container as we desire, without having to give each element a unique name which is **why struct is not a container.** In most programming languages (including C++), containers are **homogenous**, meaning the elements of a container are required to have the same type.

Arrays

An **array** is a container data type that stores a sequence of values **contiguously** (meaning each element is placed in an adjacent memory location, with no gaps) which allows random access in opposition to sequential access which would recquire access through a particular order. Arrays allow fast, direct access to any element. They are conceptually simple and easy to use, making them the first choice when we need to create and work with a set of related values.

C++ contains three primary array types: (C-style) arrays, the `std::vector` container class, and the `std::array` container class.

Introduction to `std::vector`

In the book “From Mathematics to Generic Programming”, Alexander Stepanov wrote, “The name vector in STL was taken from the earlier programming languages Scheme and Common Lisp. Unfortunately, this was inconsistent with the much older meaning of the term in mathematics... this data structure should have been called array. Sadly, if you make a mistake and violate these principles, the result might stay around for a long time.”

So, basically, `std::vector` is misnamed, but it’s too late to change it now.

`std::vector` is defined in the <vector> header as a class template, with a template type parameter that defines the type of the elements. Thus, `std::vector<int>` declares a `std::vector` whose elements are of type `int`.

```
#include <vector>

int main()
{
    // Value initialization (uses default constructor)
    std::vector<int> empty{}; // vector containing 0 int elements

    return 0;
}
```

```
#include <vector>

int main()
{
    // List construction (uses list constructor)
    std::vector<int> primes{ 2, 3, 5, 7 };          // vector containing 4 int elements with values 2, 3, 5, and 7
    std::vector vowels { 'a', 'e', 'i', 'o', 'u' }; // vector containing 5 char elements with values 'a', 'e', 'i',
'o', and 'u'.  Uses CTAD (C++17) to deduce element type char (preferred).

    return 0;
}
```

When indexing an array, the index provided must select a valid element of the array. That is, for an array of length N, the subscript must be a value between 0 and N-1 (inclusive).

`operator[]` does not do any kind of **bounds checking**, meaning it does not check to see whether the index is within the bounds of 0 to N-1 (inclusive). Passing an invalid index to `operator[]` will return in undefined behavior.

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

int main()
{
    std::vector primes { 2, 3, 5, 7, 11 }; // hold the first 5 prime numbers (as int)

    std::cout << "The first prime number is: " << primes[0] << '\n';
    std::cout << "The second prime number is: " << primes[1] << '\n';
    std::cout << "The sum of the first 5 primes is: " << primes[0] + primes[1] + primes[2] + primes[3] + primes[4] <<
'\n';

    return 0;
}
```

Constructing a `std::vector` of a specific length

```
1 | std::vector<int> data { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; // vector containing 10 int values
```

This is obviously highly repetitive and prone to errors so the solution is

```
1 | std::vector<int> data( 10 ); // vector containing 10 int elements, value-initialized to 0
```

const and NO constexpr `std::vector`

```
#include <vector>

int main()
{
    const std::vector<int> prime { 2, 3, 5, 7, 11 }; // prime and its elements cannot be modified

    return 0;
}
```

The element type of a `std::vector` must not be defined as const (e.g. `std::vector<const int>` is disallowed).

One of the biggest downsides of `std::vector` is that it cannot be made `constexpr`. If you need a `constexpr` array, use `std::array`.

std::vector resizing

fixed-size arrays or **fixed-length arrays** like `std::array` and `C-style arrays` have fixed length of array that cannot be changed. `std::vector` is a dynamic array. A **dynamic array** (also called a **resizable array**) is an array whose size can be changed after instantiation.

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

int main()
{
    std::vector v{ 0, 1, 2 }; // create vector with 3 elements
    std::cout << "The length is: " << v.size() << '\n';

    v.resize(5);              // resize to 5 elements
    std::cout << "The length is: " << v.size() << '\n';

    for (auto i : v)
        std::cout << i << ' ';

    std::cout << '\n';

    return 0;
}
```

```
The length is: 3
The length is: 5
0 1 2 0 0
```

we can also **shrink** the array

length vs capacity

egg carton with 12 spots has capacity of 12 but if it only has 5 eggs then its length is 5
we can use arrayname.capacity() to find capacity & arraynamesize() to find length of array

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

void printCapLen(const std::vector<int>& v)
{
    std::cout << "Capacity: " << v.capacity() << " Length:" << v.size() << '\n';
}

int main()
{
    std::vector v{ 0, 1, 2 }; // length is initially 3

    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    v.resize(5); // resize to 5 elements

    printCapLen(v);

    for (auto i : v)
        std::cout << i << ' ';
    std::cout << '\n';

    return 0;
}
```

```
Capacity: 3  Length: 3
0 1 2
Capacity: 5  Length: 5
0 1 2 0 0
```

arrayname.shrink_to_fit() will bring the capacity down to the length of array

by separating length and capacity, this helps the vector efficiently store its values avoiding as many reallocations of memory as possible which are expensive to do!!

The container length sign problem

the data type used for subscripting an array should match the data type used for storing the length of the array to access the longest possible array. when the container classes in the C++ standard library was being designed (circa 1997), the designers had to choose whether to make the length (and array subscripts) signed(pos and neg) or unsigned(pos). They chose to make them unsigned. In retrospect, this is generally regarded as having been the wrong choice. We now understand that using unsigned values to try to enforce non-negativity doesn't work due to the implicit conversion rules (since a negative signed integer will just implicitly convert to a large unsigned integer, producing a garbage result), the extra bit of range typically isn't needed on 32-bit or 64-bit systems (since you probably aren't creating arrays with more than 2 billion elements)

std::size()

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

int main()
{
    std::vector prime { 2, 3, 5, 7, 11 };
    std::cout << "length: " << std::size(prime); // C++17, returns length as type `size_type` (alias for
`std::size_t`)

    return 0;
}
```

since size of a array and vector is stored in a unsigned datatype, and we want to operate the size with signed operations or even storing it in a signed datatype without any compile time conversion narrowing error to occur we can use.

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

int main()
{
    std::vector prime { 2, 3, 5, 7, 11 };
    int length { static_cast<int>(prime.size()) }; // static_cast return value to int
    std::cout << "length: " << length ;

    return 0;
}
```

std::ssize()

C++20 introduces the `std::ssize()` non-member function, which returns the length as a large *signed* integral type (usually `std::ptrdiff_t`, which is the type normally used as the signed counterpart to `std::size_t`):

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector prime{ 2, 3, 5, 7, 11 };
7      std::cout << "length: " << std::ssize(prime); // C++20, returns length as a large signed integral type
8
9      return 0;
10 }
```

This is the only function of the three which returns the length as a signed type.

First, because the `int` type may be smaller than the signed type returned by `std::ssize()`, if you are going to assign the length to an `int` variable, you should static_cast the result to `int` to make any such conversion explicit

runtime bounds checking

```
std::vector prime{ 2, 3, 5, 7, 11 };

std::cout << prime[3];  // print the value of element with index 3 (7)
std::cout << prime[9]; // invalid index (undefined behavior)
```

here prime[9] will result in undefined behaviour but the code will run, to prevent such a logical range error we can use **at()**

```
std::cout << prime.at(3); // print the value of element with index 3
std::cout << prime.at(9); // invalid index (throws exception)
```

`prime.at(9)`will fail at runtime.

Indexing `std::vector`

The subscripts used to index an array should be of datatype **std::size_t** or **constexpr int** to avoid errors related to using signed datatypes like int

passing std::vector

passing std::vector is **expensive** and hence it is preferred to pass it as const reference

```
void passByRef(const std::vector<int>& arr) // we must explicitly specify <int> here
{
    std::cout << arr[0] << '\n';
}
```

however such a function will require that the returned value be of type int and will give error otherwise. we can use template to avoid such a issue

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

template <typename T>
void passByRef(const std::vector<T>& arr)
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector primes{ 2, 3, 5, 7, 11 };
    passByRef(primes); // ok: compiler will instantiate passByRef(const std::vector<int>&)

    std::vector dbl{ 1.1, 2.2, 3.3 };
    passByRef(dbl);    // ok: compiler will instantiate passByRef(const std::vector<double>&)

    return 0;
}
```

move semantics

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

std::vector<int> generate() // return by value
{
    // We're intentionally using a named object here so mandatory copy elision doesn't apply
    std::vector arr1 { 1, 2, 3, 4, 5 }; // copies { 1, 2, 3, 4, 5 } into arr1
    return arr1;
}

int main()
{
    std::vector arr2 { generate() }; // the return value of generate() dies at the end of the expression

    // There is no way to use the return value of generate() here
    arr2[0] = 7; // we only have access to arr2

    std::cout << arr2[0] << '\n';

    return 0;
}
```

a temporary object is being used to initialise another vector with same values. this is quite sub optimal because we are creating a expensive copy of something that will be destroyed anyway. there is no need for two copies to exist at the same time. if there was a way for `arr2` to “steal” the temporary’s data instead of copying it? `arr2` would then be the new owner of the data, and no copy of the data would need to be made. As an added benefit, when the temporary was then destroyed at the end of the expression, it would no longer have any data to destroy, so we wouldn’t have to pay that cost either.

not many types support move semantics but, `std::vector` and `std::string` both do!

I will cover the invocation process of move semantics later however.

move capable datatype likes std::array, std::vector and std::string are efficiently moved and hence passing them by value to a function is not that expensive!

accessing each array element without explicit listing

Accessing each element of a container in some order is called **traversal**, or **traversing** the container. Traversal is often called **iteration**, or **iterating over** or **iterating through** the container.

```
int main()
{
    std::vector testScore { 84, 92, 76, 81, 56 };
    std::size_t length { testScore.size() };

    int average { 0 };
    for (std::size_t index{ 0 }; index < length; ++index) // index from 0 to length-1
        average += testScore[index];                      // add the value of element with index `index`
    average /= static_cast<int>(length);                  // calculate the average

    std::cout << "The class average is: " << average << '\n';

    return 0;
}
```

Off-by-one errors (where the loop body executes one too many or one too few times) are easy to make Typically, when traversing a container using an index, we will start the index at `0` and loop until `index < length`.

New programmers sometimes accidentally use `index <= length` as the loop condition. This will cause the loop to execute when `index == length`, which will result in an out of bounds subscript and undefined behavior.

array loop variables

The `size()` member function returns `size_type`, and `operator[]` uses `size_type` as an index, so using `size_type` as the type of your index is technically the most consistent and safe unsigned type to use (as it will work in all cases, even in the extremely rare case where `size_type` is something other than `size_t`.)

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

int main()
{
    std::vector arr { 1, 2, 3, 4, 5 };

    for (std::vector<int>::size_type index { 0 }; index < arr.size(); ++index)
        std::cout << arr[index] << ' ';

    return 0;
}
```

`size_type` is a nested type and to use it we have to explicitly prefix the name with the fully templated name of the container (meaning we have to type `std::vector<int>::size_type` rather than just `std::size_type`)

with function templates, this would look like

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

template <typename T>
void printArray(const std::vector<T>& arr)
{
    // typename keyword prefix required for dependent type
    for (typename std::vector<T>::size_type index { 0 }; index < arr.size(); ++index)
        std::cout << arr[index] << ' ';
}

int main()
{
    std::vector arr { 9, 7, 5, 3, 1 };

    printArray(arr);

    return 0;
}
```

`size_type` is almost always a typedef for `size_t`, many programmers just skip using `size_type` altogether and use the easier to remember and type `std::size_t` directly

```
for (std::size_t index { 0 }; index < arr.size(); ++index)
```

unsigned integral types like `std::size_t` for length and indices can cause errors like ex.

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

template <typename T>
void printReverse(const std::vector<T>& arr)
{
    for (std::size_t index{ arr.size() - 1 }; index >= 0; --index) // index is unsigned
    {
        std::cout << arr[index] << ' ';
    }

    std::cout << '\n';
}

int main()
{
    std::vector arr{ 4, 6, 7, 3, 8, 2, 1, 9 };

    printReverse(arr);

    return 0;
}
```

The problem here is that our loop executes as long as `index >= 0` (or in other words, as long as `index` is positive) but index will always be positive because it's a unsigned variable and will become a large value after going beyond zero

this will be solved by usage of a signed variable

If we are going to use signed loop variables, there are three issues we need to address:

- **what signed type?** (int for small or std::ptrdiff_t for very large arrays)
- **getting the length of the array as a signed value**(static casting size())
- **converting the signed loop variable to an unsigned index**(static casting the index for use in the forloop)

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

template <typename T>
void printReverse(const std::vector<T>& arr)
{
    for (int index{ static_cast<int>(arr.size()) - 1}; index >= 0; --index) // index is signed
    {
        std::cout << arr[static_cast<std::size_t>(index)] << ' ';
    }

    std::cout << '\n';
}

int main()
{
    std::vector arr{ 4, 6, 7, 3, 8, 2, 1, 9 };

    printReverse(arr);

    return 0;
}
```

All of the options presented above have their own downsides, so it's hard to recommend one approach over the other. However, there is a choice that is far more sane than the others: avoid indexing with integral values altogether with methods like **range based traversal** and **iterators**.

range based traversal

```
for (element_declaration : array_object)
    statement;
```

`element_declaration` should have the same type as the array elements, otherwise type conversion will occur.

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

int main()
{
    std::vector fibonacci { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

    for (int num : fibonacci) // iterate over array fibonacci and copy each value into `num`
       std::cout << num << ' '; // print the current value of `num`

    std::cout << '\n';

    return 0;
}
```

`num` is assigned the value of the array element, this does mean the array element is copied (which can be expensive for some types).

we can also use **auto** to ensure `element_declaration` is the same type as array elements and cause no error in case we update the type of elements in array.

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

int main()
{
    std::vector fibonacci { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };

    for (auto num : fibonacci) // compiler will deduce type of num to be `int`
       std::cout << num << ' ';

    std::cout << '\n';

    return 0;
}
```

this also works with const references that are usually used with std::strings

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

int main()
{
    std::vector<std::string> words{ "peter", "likes", "frozen", "yogurt" };

    for (const auto& word : words) // word is now a const reference
        std::cout << word << ' ';

    std::cout << '\n';

    return 0;
}
```

Normally we'd use `auto` for cheap-to-copy types, `auto&` when we want to modify the elements, and `const auto&` for expensive-to-copy types. But with range-based for loops, many developers believe it is preferable to always use `const auto&` because it is more future-proof.

array indexing using enumerators

```cpp
#include <vector>

namespace Students
{
    enum Names
    {
        kenny, // 0
        kyle, // 1
        stan, // 2
        butters, // 3
        cartman, // 4
        max_students // 5
    };
}

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };

    testScores[Students::stan] = 76; // we are now updating the test score belonging to stan

    return 0;
}
```

enumerators are implicitly constexpr, as long as we stick to indexing with unscoped enumerators, we won't run into sign conversion issues but we may get sign conversion errors in cases where we define a non constexpr variable using enumerators and then index with that variable

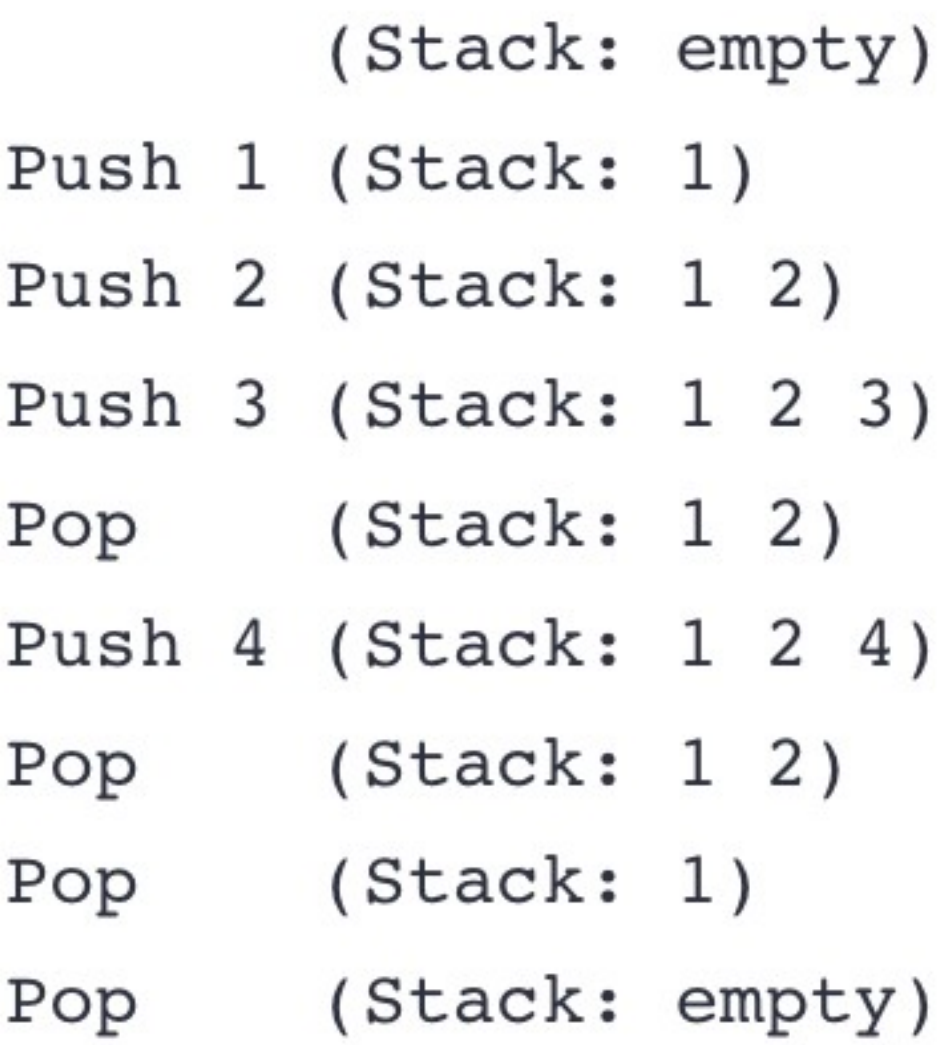
```cpp
int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };
    Students::Names name { Students::stan }; // non-constexpr

    testScores[name] = 76; // may trigger a sign conversion warning if Student::Names defaults to a signed underlying
type

    return 0;
}
```

this can be fixed by explicitly specifying the underlying type of enumeration as unsigned int

```cpp
#include <vector>

namespace Students
{
    enum Names : unsigned int // explicitly specifies the underlying type is unsigned int
    {
        kenny, // 0
        kyle, // 1
        stan, // 2
        butters, // 3
        cartman, // 4
        max_students // 5
    };
}

int main()
{
    std::vector testScores { 78, 94, 66, 77, 14 };
    Students::Names name { Students::stan }; // non-constexpr

    testScores[name] = 76; // not a sign conversion since name is unsigned

    return 0;
}
```

stack

a **stack** is a container data type where the insertion and removal of elements occurs in a LIFO manner. This is commonly implemented via two operations named **push**(put element at top of the stack) and **pop**(remove element from the top of the stack).



---

stack behavior with `std::vector`

Function Name	Stack Operation	Behavior	Notes
push_back()	Push	Put new element on top of stack	Adds the element to end of vector
pop_back()	Pop	Remove the top element from the stack	Returns void, removes element at end of vector
back()	Top or Peek	Get the top element on the stack	Does not remove item
emplace_back()	Push	Alternate form of push_back() that can be more efficient (see below)	Adds element to end of vector

emplace_back() is best when creating a temporary object to add or with explicit constructor (oop)

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

void printStack(const std::vector<int>& stack)
{
    if (stack.empty()) // if stack.size == 0
        std::cout << "Empty";

    for (auto element : stack)
        std::cout << element << ' ';

    // \t is a tab character, to help align the text
    std::cout << "\tCapacity: " << stack.capacity() << "  Length " << stack.size() << "\n";
}

int main()
{
    std::vector<int> stack{}; // empty stack

    printStack(stack);

    stack.push_back(1); // push_back() pushes an element on the stack
    printStack(stack);

    stack.push_back(2);
    printStack(stack);

    stack.push_back(3);
    printStack(stack);

    std::cout << "Top: " << stack.back() << '\n'; // back() returns the last element

    stack.pop_back(); // pop_back() pops an element off the stack
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    return 0;
}
```

```
Empty   Capacity: 0  Length: 0
1       Capacity: 1  Length: 1
1 2     Capacity: 2  Length: 2
1 2 3   Capacity: 4  Length: 3
Top:3
1 2     Capacity: 4  Length: 2
1       Capacity: 4  Length: 1
Empty   Capacity: 4  Length: 0
```

When pushing triggers a reallocation, `std::vector` will typically allocate some extra capacity to allow additional elements to be added without triggering another reallocation the next time an element is added.

reserve()

since reallocations are expensive, it is not feasible to change the capacity so many times when we are pushing a element onto a stack. we could just create a std::vector with allocated capacity but then it would have a series of zero initialised values in it, pushing any element on such a vector would only add the value after the set of zeroes. The `reserve()` member function can be used to reallocate a `std::vector` without changing the current length.

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

void printStack(const std::vector<int>& stack)
{
    if (stack.empty()) // if stack.size == 0
        std::cout << "Empty";

    for (auto element : stack)
        std::cout << element << ' ';

    // \t is a tab character, to help align the text
    std::cout << "\tCapacity: " << stack.capacity() << "  Length " << stack.size() << "\n";
}

int main()
{
    std::vector<int> stack{};

    printStack(stack);

    stack.reserve(6); // reserve space for 6 elements (but do not change length)
    printStack(stack);

    stack.push_back(1);
    printStack(stack);

    stack.push_back(2);
    printStack(stack);

    stack.push_back(3);
    printStack(stack);

    std::cout << "Top: " << stack.back() << '\n';

    stack.pop_back();
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    stack.pop_back();
    printStack(stack);

    return 0;
}
```

```
Empty   Capacity: 0  Length: 0
Empty   Capacity: 6  Length: 0
1       Capacity: 6  Length: 1
1 2     Capacity: 6  Length: 2
1 2 3   Capacity: 6  Length: 3
Top: 3
1 2     Capacity: 6  Length: 2
1       Capacity: 6  Length: 1
Empty   Capacity: 6  Length: 0
```

No more reallocations occur because the std::vector is large enough to accommodate all of the elements we are pushing.

std::array

dynamic arrays are slightly less performant due to reallocation especially on sloppy code and do not support constexpr except in very limited context

declaration of std::arrays done under <array> header is different from vectors

```
#include <array>

int main()
{
    std::array<int, 7> a {}; // Using a literal constant

    constexpr int len { 8 };
    std::array<int, len> b {}; // Using a constexpr variable

    enum Colors
    {
         red,
         green,
         blue,
         max_colors
    };

    std::array<int, max_colors> c {}; // Using an enumerator

#define DAYS_PER_WEEK 7
    std::array<int, DAYS_PER_WEEK> d {}; // Using a macro (don't do this, use a constexpr variable instead)

    return 0;
}
```

the size of array must be define using constant expression so not random ints can be used.

Perhaps surprisingly, std::array is an aggregate. This means it has no constructors, and instead is initialized using aggregate initialization. Aggregate initialization allows us to directly initialize the members of aggregates. To do this, we provide an initializer list, which is a brace-enclosed list of comma-separated initialization values (oop concept)

initialisation

```
std::array<int, 5> a;   // Members default initialized (int elements are left uninitialized)
std::array<int, 5> b{}; // Members value initialized (int elements are zero initialized) (preferred)
```

```
std::array<int, 4> a { 1, 2, 3, 4, 5 }; // compile error: too many initializers
std::array<int, 4> b { 1, 2 };          // b[2] and b[3] are value initialized
```

const & constexpr arrays

```
const std::array<int, 5> prime { 2, 3, 5, 7, 11 };
```

the elements of a const std::array are not explicitly marked as const, they are still treated as const (because the whole array is const).

```
constexpr std::array<int, 5> prime { 2, 3, 5, 7, 11 };
```

Class template argument deduction (CTAD) for `std::array`

```
constexpr std::array a1 { 9, 7, 5, 3, 1 }; // The type is deduced to std::array<int, 5>
constexpr std::array a2 { 9.7, 7.31 };     // The type is deduced to std::array<double, 2>
```

partial omission of template arguments is unsupported in std::array

```
constexpr std::array<int> a2 { 9, 7, 5, 3, 1 };     // error: too few template arguments (length missing)
constexpr std::array<5> a2 { 9, 7, 5, 3, 1 };       // error: too few template arguments (type missing)
```

partial omission is supported in std::to_array (very expensive coz two arrays are made)

```
constexpr auto myArray1 { std::to_array<int, 5>({ 9, 7, 5, 3, 1 }) }; // Specify type and size
constexpr auto myArray2 { std::to_array<int>({ 9, 7, 5, 3, 1 }) };    // Specify type only, deduce size
constexpr auto myArray3 { std::to_array({ 9, 7, 5, 3, 1 }) };         // Deduce type and size
```

`operator[]`

```
constexpr std::array<int, 5> prime{ 2, 3, 5, 7, 11 };

std::cout << prime[3]; // print the value of element with index 3 (7)
std::cout << prime[9]; // invalid index (undefined behavior)
```

just like vectors, doesn't do any bounds checking and out of bounds index will bring undefine behaviour but not a error in code.

std::array length and indexing

`std::vector` and `std::array` have similar interfaces in regards to functions and theory

the length of a `std::array` is constexpr & all the functions will return the length of a `std::array` as a constexpr value (even when called on a non-constexpr `std::array` object)! Which allows us to use these functions in constant expressions and length returned can be converted implicitly to int without it being narrowing.

```
    std::array arr { 9, 7, 5, 3, 1 }; // note: not constexpr for this example
    constexpr int length{ std::size(arr) }; // ok: return value is constexpr std::size_t and can be converted to int,
not a narrowing conversion
```

`std::get()`

`operator[]` does no bounds checking by definition, and the `at()` member function only does runtime bounds checking. `std::get()` function template will do bounds checking even for constexpr indices because it takes index as a non-type template argument.

```
constexpr std::array prime{ 2, 3, 5, 7, 11 };

std::cout << std::get<3>(prime); // print the value of element with index 3
std::cout << std::get<9>(prime); // invalid index (compile error)
```

passing std::arrays in functions

passing std::arrays by value is very expensive and const references are usually preferred.

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

void passByRef(const std::array<int, 5>& arr) // we must explicitly specify <int, 5> here
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 }; // CTAD deduces type std::array<int, 5>
    passByRef(arr);

    return 0;
}
```

CTAD doesn't (currently) work with function parameters, so we cannot just specify `std::array` here and let the compiler deduce the template arguments.

function templates that uses template parameter declaration

```
template <typename T, std::size_t N> // note that this template parameter declaration matches the one for std::array
void passByRef(const std::array<T, N>& arr)
{
    static_assert(N != 0); // fail if this is a zero-length std::array

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

we can template only one parameter as well

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

template <std::size_t N> // note: only the length has been templated here
void passByRef(const std::array<int, N>& arr) // we've defined the element type as int
{
    static_assert(N != 0); // fail if this is a zero-length std::array

    std::cout << arr[0] << '\n';
}

int main()
{
    std::array arr{ 9, 7, 5, 3, 1 }; // use CTAD to infer std::array<int, 5>
    passByRef(arr);  // ok: compiler will instantiate passByRef(const std::array<int, 5>& arr)

    std::array arr2{ 1, 2, 3, 4, 5, 6 }; // use CTAD to infer std::array<int, 6>
    passByRef(arr2); // ok: compiler will instantiate passByRef(const std::array<int, 6>& arr)

    std::array arr3{ 1.2, 3.4, 5.6, 7.8, 9.9 }; // use CTAD to infer std::array<double, 5>
    passByRef(arr3); // error: compiler can't find matching function

    return 0;
}
```

replacement of std::size_t with auto will enable template parameter deduction to it (c++20)

```
template <typename T, auto N> // now using auto to deduce type of N
void passByRef(const std::array<T, N>& arr)
```

compile time capabilities with function templates

one advantage template parameters have over normal function parameters is that they are compile time constants which allows us to take advantage of constant expression dependent capabilities.

```
template <typename T, std::size_t N>
void printElement3(const std::array<T, N>& arr)
{
    std::cout << std::get<3>(arr) << '\n'; // checks that index 3 is valid at compile-time
}
```

```
template <typename T, std::size_t N>
void printElement3(const std::array<T, N>& arr)
{
    // precondition: array length must be greater than 3 so element 3 exists
    static_assert (N > 3);

    // we can assume the array length is greater than 3 beyond this point

    std::cout << arr[3] << '\n';
}
```

are two ways to ensure the function will result in an error at compile time, in case the array index that needs to be outputted is not valid with respect to the size of the array.

returning std::arrays

It is okay to return a `std:array` by value when

- The array isn't huge.
- The element type is cheap to copy (or move).
- The code isn't being used in a performance-sensitive context.

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

// return by value
template <typename T, std::size_t N>
std::array<T, N> inputArray() // return by value
{
    std::array<T, N> arr{};
    std::size_t index { 0 };
    while (index < N)
    {
        std::cout << "Enter value #" << index << ": ";
        std::cin >> arr[index];

        if (!std::cin) // handle bad input
        {
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            continue;
        }
        ++index;
    }

    return arr;
}
```

`std::vector` is move-capable and can be returned by value without making expensive copies. If you're returning a `std::array` by value, your `std::array` probably isn't constexpr, and you should consider using (and returning) `std::vector` instead.

std::array of struct members

initialisation of array-struct members

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    std::array<House, 3> houses{};

    houses[0] = { 13, 1, 7 };
    houses[1] = { 14, 2, 5 };
    houses[2] = { 15, 2, 4 };

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories * house.roomsPerStory)
                  << " rooms.\n";
    }

    return 0;
}
```

Initialisation in declaration of array structs

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    constexpr std::array houses { // use CTAD to deduce template arguments <House, 3>
            House{ 13, 1, 7 },
            House{ 14, 2, 5 },
            House{ 15, 2, 4 }
        };

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
            << " has " << (house.stories * house.roomsPerStory)
            << " rooms.\n";
    }

    return 0;
}
```

quality of life improvement initialisation during declaration

```
// Doesn't work
constexpr std::array<House, 3> houses { // initializer for houses
    { 13, 1, 7 }, // initializer for C-style array member with implementation_defined_name
    { 14, 2, 5 }, // ?
    { 15, 2, 4 }  // ?
};
```

this won't work because the std array. unless ordered otherwise will be taking the first element as a c-style array and then create an error when it sees two more initialisation values. the only method to declare array of struct without writing the struct name multiple times is using a extra flower bracket over the values and an extra comma at the end of the values

```
// This works as expected
constexpr std::array<House, 3> houses { // initializer for houses
    { // extra set of braces to initialize the C-style array member with implementation_defined_name
        { 13, 4, 30 }, // initializer for array element 0
        { 14, 3, 10 }, // initializer for array element 1
        { 15, 3, 40 }, // initializer for array element 2
    }
};
```

this would look like this

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

struct House
{
    int number{};
    int stories{};
    int roomsPerStory{};
};

int main()
{
    constexpr std::array<House, 3> houses {{ // note double braces
        { 13, 1, 7 },
        { 14, 2, 5 },
        { 15, 2, 4 }
    }};

    for (const auto& house : houses)
    {
        std::cout << "House number " << house.number
                  << " has " << (house.stories * house.roomsPerStory)
                  << " rooms.\n";
    }

    return 0;
}
```

here we don't have the final trailing comma in declaration but both version are syntactically correct and viable

we don't have to use double braces for initialising std::arrays with single scalar values and still initialise with all the values provided because of a concept supported in c++ called **brace elision** which omits us from making it necessary to add double braces but the code will run just fine even if we do add it.

navigating through arrays of struct via pointers (+ another initialisation method)

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

// Each student has an id and a name
struct Student
{
    int id{};
    std::string_view name{};
};

// Our array of 3 students (single braced since we mention Student with each initializer)
constexpr std::array students{ Student{0, "Alex"}, Student{ 1, "Joe" }, Student{ 2, "Bob" } };

const Student* findStudentById(int id)
{
    // Look through all the students
    for (auto& s : students)
    {
        // Return student with matching id
        if (s.id == id) return &s;
    }

    // No matching id found
    return nullptr;
}

int main()
{
    constexpr std::string_view nobody { "nobody" };

    const Student* s1 { findStudentById(1) };
    std::cout << "You found: " << (s1 ? s1->name : nobody) << '\n';

    const Student* s2 { findStudentById(3) };
    std::cout << "You found: " << (s2 ? s2->name : nobody) << '\n';

    return 0;
}
```

std::reference_wrapper //under #include <functional>

arrays can have elements of any object type which include fundamental datatypes like int or compound types like pointers to int but we cannot make a array of references as they are not objects.

```
    int x { 1 };
    int y { 2 };

    [[maybe_unused]] std::array<int&, 2> refarr { x, y }; // compile error: cannot define array of references

    int& ref1 { x };
    int& ref2 { y };
    [[maybe_unused]] std::array valarr { ref1, ref2 }; // ok: this is actually a std::array<int, 2>, not an array of
references
```

`std::reference_wrapper` helps bypass this mechanism giving us a array of references to multiple individual objects and the ability to change the value of array members directly changing the values of the individual objects assigned to it.

```
  int x { 1 };
  int y { 2 };
  int z { 3 };

  std::array<std::reference_wrapper<int>, 3> arr { x, y, z };

  arr[1].get() = 5; // modify the object in array element 1

  std::cout << arr[1] << y << '\n'; // show that we modified arr[1] and y, prints 55

  return 0;
```

constexpr std::arrays w/ enumeration (understand this code)

```cpp
#include <array>
#include <iostream>
#include <string>
#include <string_view>

namespace Color
{
    enum Type
    {
        black,
        red,
        blue,
        max_colors
    };

    // use sv suffix so std::array will infer type as std::string_view
    using namespace std::string_view_literals; // for sv suffix
    constexpr std::array colorName { "black"sv, "red"sv, "blue"sv };

    // Make sure we've defined strings for all our colors
    static_assert(std::size(colorName) == max_colors);
};

constexpr std::string_view getColorName(Color::Type color)
{
    // We can index the array using the enumerator to get the name of the enumerator
    return Color::colorName[color];
}

// Teach operator<< how to print a Color
// std::ostream is the type of std::cout
// The return type and parameter type are references (to prevent copies from being made)!
std::ostream& operator<<(std::ostream& out, Color::Type color)
{
    return out << getColorName(color);
}

// Teach operator>> how to input a Color by name
// We pass color by non-const reference so we can have the function modify its value
std::istream& operator>> (std::istream& in, Color::Type& color)
{
    std::string input {};
    std::getline(in >> std::ws, input);

    // Iterate through the list of names to see if we can find a matching name
    for (std::size_t index=0; index < Color::colorName.size(); ++index)
    {
        if (input == Color::colorName[index])
        {
            // If we found a matching name, we can get the enumerator value based on its index
            color = static_cast<Color::Type>(index);
            return in;
        }
    }

    // We didn't find a match, so input must have been invalid
    // so we will set input stream to fail state
    in.setstate(std::ios_base::failbit);

    // On an extraction failure, operator>> zero-initializes fundamental types
    // Uncomment the following line to make this operator do the same thing
    // color = {};
    return in;
}

int main()
{
    auto shirt{ Color::blue };
    std::cout << "Your shirt is " << shirt << '\n';

    std::cout << "Enter a new color: ";
    std::cin >> shirt;
    if (!std::cin)
        std::cout << "Invalid\n";
    else
        std::cout << "Your shirt is now " << shirt << '\n';

    return 0;
}
```

c-style arrays

C-style arrays were inherited from the C language, and are built-in to the core language of C++ (unlike the rest of the array types, which are standard library container classes). This means we don't need to #include a header file to use them.

int arrayname[5] {}; //a c-style array with 5 zero initialised int elements
//std::array<int,5> arrayname{}; with #include <array> for comparison

we use square brackets ([]) to tell the compiler that a declared object is a C-style array. Inside the square brackets, we can optionally *(if we don't compiler will deduce the length of array as long as we have provided all initialisers)* provide the length of the array, which is an integral value of type `std::size_t` that tells the compiler how many elements are in the array.

c-style arrays can be indexed using any integral type or enumeration and **can have negative indexing** unlike standard library container classes like std::vectors and std::arrays.

```
    const int arr[] { 9, 8, 7, 6, 5 };

    int s { 2 };
    std::cout << arr[s] << '\n'; // okay to use signed index

    unsigned int u { 2 };
    std::cout << arr[u] << '\n'; // okay to use unsigned index
```

initialisation

initialise singular value at given index

```
int arr[5]; // define an array of 5 int values

arr[1] = 7; // use subscript operator to index array element 1
```

aggregated initialisation (**prefer list initialisation**)

```
int fibonnaci[6] = { 0, 1, 1, 2, 3, 5 }; // copy-list initialization using braced list
int prime[5] { 2, 3, 5, 7, 11 };         // list initialization using braced list (preferred)
```

initialise with empty braces for zero valued elements

```
int arr1[5];    // Members default initialized int elements are left uninitialized)
int arr2[5] {}; // Members value initialized (int elements are zero uninitialized) (preferred)
```

number of elements wrt to initialised array

```
int a[4] { 1, 2, 3, 4, 5 }; // compile error: too many initializers
int b[4] { 1, 2 };          // arr[2] and arr[3] are value initialized
```

downside of using a C-style array is that the element's type must be explicitly specified as CTAD wont not class templates and auto wont be able to deduce as well.

constexpr c-style arrays are possible

```
const int prime[5] { 2, 3, 5, 7, 11 }; // an array of const int
prime[0] = 17; // compile error: can't change const int
```

sizeof() c-style array

```
const int prime[] { 2, 3, 5, 7, 11 }; // the compiler will deduce prime to have length 5

std::cout << sizeof(prime); // prints 20 (assuming 4 byte ints)
```

std::size & std::ssize

```
const int prime[] { 2, 3, 5, 7, 11 };   // the compiler will deduce prime to have length 5

std::cout << std::size(prime) << '\n';  // C++17, returns unsigned integral value 5
std::cout << std::ssize(prime) << '\n'; // C++20, returns signed integral value 5
```

`std::size()` returns the array length as an unsigned integral value (of type `std::size_t`) and `std::ssize()` returns the array length as a signed integral value (of a large signed integral type, probably `std::ptrdiff_t`).

in much older codebases, the length of the array will be given by `sizeof(array) / sizeof(array[0])` this formula can fail quite easily when passing a decayed array. std::size and std::ssize are more reliable in such cases

assignment

individual assignment using indexes are fine but assigning multiple values to a array is not possible because such a process requires the left operand to be a modifiable lvalue which c-style arrays are not.

```
int arr[] { 1, 2, 3 }; // okay: initialization is fine
arr[0] = 4;            // assignment to individual elements is fine
arr = { 5, 6, 7 };     // compile error: array assignment not valid
```

its better to use std::vectors for multiple value assignment

since we can't directly copy arrays from one to another because of multiple assignment prohibition, we have to use std::copy under <algorithm>

```
int arr[] { 1, 2, 3 };
int src[] { 5, 6, 7 };

// Copy src into arr
std::copy(std::begin(src), std::end(src), std::begin(arr));
```

c-style array decay

whenever a c-style array is passed in a function, it is implicitly converted to a pointer to datatype of array, pointing to the first element in array which is called **array decay**. this means that passing a array to a function is array size independent and all elements are not copied into the function and the subscripting can also be applied on the pointer to first element because c-style arrays are stored in a adjacent series.

```
#include <iostream>

int main()
{
    const int arr[] { 9, 7, 5, 3, 1 };

    const int* ptr{ arr };  // arr decays into a pointer
    std::cout << ptr[2];    // subscript ptr to get element 2, prints 5

    return 0;
}
```

we can also pass arrays into pointer parameter functions and vice versa.

```
#include <iostream>

void printElementZero(const int* arr) // pass by const address
{
    std::cout << arr[0];
}

int main()
{
    const int prime[] { 2, 3, 5, 7, 11 };
    const int squares[] { 1, 4, 9, 25, 36, 49, 64, 81 };

    printElementZero(prime);   // prime decays to an const int* pointer
    printElementZero(squares); // squares decays to an const int* pointer

    return 0;
}
```

one downside is that sizeof() inside functions blocks will only return the size of first element while std::size and std::ssize will return compiler errors when provided a pointer.

another downside is no way of knowing what is the length of array inside a function to prevent undefined behaviour like out of bounds indexing. this can be prevented by providing the length of array as a separate parameter in the function but this could also lead to many problems like difference in valid elements and length.

c-style arrays are mostly avoided in modern programming except for purposes like storing constexpr global programming data because such arrays can be accessed directly from anywhere in the program, we do not need to pass the array, which avoids decay-related

issues and as parameters to functions or classes that want to handle non-constexpr c-style string arguments directly (rather than requiring a conversion to `std::string_view`).

pointer arithmetic

Given some pointer `ptr`, `ptr + 1` returns the address of the *next object* in memory (based on the type being pointed to). So if `ptr` is an `int*`, and an `int` is 4 bytes, `ptr + 1` will return the memory address that is 4 bytes after `ptr`, and `ptr + 2` will return the memory address that is 8 bytes after `ptr`. Although less common, pointer arithmetic also works with subtraction. `ptr - 1` returns the address of the *previous object* in memory (based on the type being pointed to).

```
int x {};
const int* ptr{ &x }; // assume 4 byte ints

std::cout << ptr << '\n';

++ptr; // ptr = ptr + 1
std::cout << ptr << '\n';

--ptr; // ptr = ptr - 1
std::cout << ptr << '\n';
```

traversing an array without index

```
#include <iostream>

int main()
{
    constexpr int arr[]{ 9, 7, 5, 3, 1 };

    const int* begin{ arr };                // begin points to start element
    const int* end{ arr + std::size(arr) }; // end points to one-past-the-end element

    for (; begin != end; ++begin)           // iterate from begin up to (but excluding) end
    {
        std::cout << *begin << ' ';     // dereference our loop variable to get the current element
    }

    return 0;
}
```

Note that `end` is set to one-past-the-end of the array. Having `end` hold this address is fine (so long as we don't dereference `end`, as there isn't a valid element at that address).

c-style strings

c-style strings are just arrays with type char and all the array theory works on them especially rules like prohibition of multiple value assignment. <cstring> header provides variety of c inherited string manipulating functions

- strlen() -- returns the length of a C-style string
- strcpy(), strncpy(), strcpy_s() -- overwrites one C-style string with another
- strcat(), strncat() -- Appends one C-style string to the end of another
- strcmp(), strncmp() -- Compares two C-style strings (returns `0` if equal)

Except for `strlen()`, we generally recommend avoiding these.

std::couting c-style arrays will usually print the pointer to first element but this is not the case for c-style strings unless we std::cout the static_cast to type void*.

avoid using c-style string in favour of std::strings.

multi dimensional c-style array

an array can be of any object type including other arrays.

```
int a[3][5]; // a 3-element array of 5-element arrays of int
```

```
// col 0    col 1    col 2    col 3    col 4
// a[0][0]  a[0][1]  a[0][2]  a[0][3]  a[0][4]  row 0
// a[1][0]  a[1][1]  a[1][2]  a[1][3]  a[1][4]  row 1
// a[2][0]  a[2][1]  a[2][2]  a[2][3]  a[2][4]  row 2
```

initialisation with nested braces

```
int array[3][5]
{
  { 1, 2, 3, 4, 5 },     // row 0
  { 6, 7, 8, 9, 10 },    // row 1
  { 11, 12, 13, 14, 15 } // row 2
};
```

can be zero initialised as well !!

```
int array[3][5] {};
```

changing elements of 2d array

```
a[2][3] = 7; // a[row][col], where row = 2 and col = 3
```

c++ supports arrays with **more than two dimensions** as well

```
int threedee[4][4][4]; // a 4x4x4 array (an array of 4 arrays of 4 arrays of 4 ints)
```

For example, the terrain in Minecraft is divided into 16x16x16 blocks (called chunk sections).

c++ uses **row major order** to store the elements of array in memory

```
[0][0] [0][1] [0][2] [0][3] [0][4] [1][0] [1][1] [1][2] [1][3] [1][4] [2][0] [2][1] [2][2] [2][3] [2][4]
```

for loop for traversal of 2d array

```cpp
#include <iostream>

int main()
{
    int arr[3][4] {
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }};

    // double for-loop with indices
    for (std::size_t row{0}; row < std::size(arr); ++row) // std::size(arr) returns the number of rows
    {
        for (std::size_t col{0}; col < std::size(arr[0]); ++col) // std::size(arr[0]) returns the number of columns
            std::cout << arr[row][col] << ' ';

        std::cout << '\n';
    }

    // double range-based for-loop
    for (const auto& arow: arr)   // get each array row
    {
        for (const auto& e: arow) // get each element of the row
            std::cout << e << ' ';

        std::cout << '\n';
    }

    return 0;
}
```

multi dimensional std::arrays

The canonical way to create a two-dimensional array of `std::array` is to create a `std::array` where the template type argument is another `std::array`.

```cpp
std::array<std::array<int, 4>, 3> arr {{  // note double braces
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
    { 9, 10, 11, 12 }}};
```

indexing works same as c-style array

```cpp
std::cout << arr[1][2];
```

passing a 2d array in a function template

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

template <typename T, std::size_t Row, std::size_t Col>
void printArray(const std::array<std::array<T, Col>, Row> &arr)
{
    for (const auto& arow: arr)   // get each array row
    {
        for (const auto& e: arow) // get each element of the row
            std::cout << e << ' ';

        std::cout << '\n';
    }
}

int main()
{
    std::array<std::array<int, 4>, 3>  arr {{
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }}};

    printArray(arr);

    return 0;
}
```

Getting the dimensional length of a 2d array using alias templates

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

// An alias template for a two-dimensional std::array
template <typename T, std::size_t Row, std::size_t Col>
using Array2d = std::array<std::array<T, Col>, Row>;

// Fetch the number of rows from the Row non-type template parameter
template <typename T, std::size_t Row, std::size_t Col>
constexpr int rowLength(const Array2d<T, Row, Col>&) // you can return std::size_t if you prefer
{
    return Row;
}

// Fetch the number of cols from the Col non-type template parameter
template <typename T, std::size_t Row, std::size_t Col>
constexpr int colLength(const Array2d<T, Row, Col>&) // you can return std::size_t if you prefer
{
    return Col;
}

int main()
{
    // Define a two-dimensional array of int with 3 rows and 4 columns
    Array2d<int, 3, 4> arr {{
        { 1, 2, 3, 4 },
        { 5, 6, 7, 8 },
        { 9, 10, 11, 12 }}};

    std::cout << "Rows: " << rowLength(arr) << '\n'; // get length of first dimension (rows)
    std::cout << "Cols: " << colLength(arr) << '\n'; // get length of second dimension (cols)

    return 0;
}
```

std::sort

```cpp
#include <algorithm> // for std::sort
#include <iostream>
#include <iterator> // for std::size

int main()
{
    int array[]{ 30, 50, 20, 10, 40 };

    std::sort(std::begin(array), std::end(array));

    for (int i{ 0 }; i < static_cast<int>(std::size(array)); ++i)
        std::cout << array[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

the C++ standard library includes a sorting function named `std::sort`. `std::sort` lives in the <algorithm> header, and can be invoked on an array

dynamic memory allocation

C++ supports three basic types of memory allocation

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program. The size of the variable / array must be known at compile time.
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary. The size of the variable / array must be known at compile time.
- **Dynamic memory allocation**

most normal variables (including fixed arrays) are allocated in a portion of memory called the **stack**. The amount of stack memory for a program is generally quite small. If you exceed this number, stack overflow will result, and the operating system will probably close

down the program. **Dynamic memory allocation** is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory -- instead, it is allocated from a much larger pool of memory managed by the operating system called the **heap**. On modern machines, the heap can be gigabytes in size.

dynamically allocating single variables

```
new int; // dynamically allocate an integer (and discard the result)
```

without a pointer to hold the address of the memory that was just allocated, we'd have no way to access the memory that was just allocated for us

```
int* ptr{ new int }; // dynamically allocate an integer and assign the address to ptr so we can access it later
```

accessing heap-allocated objects is generally slower than accessing stack-allocated objects. we can access and assign values to the new int by deferencing the pointer assigned to it.

```
1 *ptr = 7; // assign value of 7 to allocated memory
```

Your computer has memory (probably lots of it) that is available for applications to use. When you run an application, your operating system loads the application into some of that memory. When you dynamically allocate memory, you're asking the operating system to reserve some of that memory for your program's use. If it can fulfil this request, it will return the address of that memory to your application.

```
int* ptr1{ new int (5) }; // use direct initialization
int* ptr2{ new int { 6 } }; // use uniform initialization
```

deleting a single variable

when we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse.

```
// assume ptr has previously been allocated with operator new
delete ptr; // return the memory pointed to by ptr to the operating system
ptr = nullptr; // set ptr to be a null pointer
```

The delete operator does not *actually* delete anything. It simply returns the memory being pointed to back to the operating system. The operating system is then free to reassign that memory to another application (or to this application again later). Although it looks like we're deleting a *variable*, this is not the case! The pointer variable still has the same scope as before, and can be assigned a new value just like any other variable. deleting a pointer that is not pointing to dynamically allocated memory or deferencing a deleted pointer that is now dangling will cause bad things to happen. That is why it is best practice to set the dangling/deleted pointer to null after it is deleted if it is still in scope for usage after being deleted.

deleting null pointers does not do anything because the delete operation by default checks if the pointer is not null before doing the operation. this means that we don't have to check if the pointer is null before deleting it.

Deleting a null pointer has no effect. Thus, there is no need for the following:

```
1 if (ptr) // if ptr is not a null pointer
2     delete ptr; // delete it
3 // otherwise do nothing
```

if dynamic allocation fails

rarely the operating system may not have the memory to grant the request & the program will simply terminate with an unhandled exception error. to ensure we know, why & when the error is happening we can use std::nothrow to set the pointer null if allocation fails. deferencing the null pointer will cause an error but we can check in code itself & know what error is happening by checking for null pointer.

```
int* value { new (std::nothrow) int{} }; // ask for an integer's worth of memory
if (!value) // handle case where new returned null
{
    // Do error handling here
    std::cerr << "Could not allocate memory\n";
}
```

Memory leaks

Memory leaks happen when your program loses the address of some bit of dynamically allocated memory before giving it back to the operating system. When this happens, your program can't delete the dynamically allocated memory, because it no longer knows where it is. The operating system also can't use this memory, because that memory is considered to be still in use by your program.

```
int value = 5;
int* ptr{ new int{} }; // allocate memory
ptr = &value; // old address lost, memory leak results
```

Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash. Only after your program terminates is the operating system able to clean up and "reclaim" all leaked memory.

```
1 void doSomething()
2 {
3     int* ptr{ new int{} };
4 }
```

here we forget to delete the new integer & the memory stays allocated without being able to be used until the end of the whole program

dynamically allocating arrays

In addition to dynamically allocating single values, we can also dynamically allocate arrays of variables. Unlike a fixed array, where the array size must be fixed at compile time, dynamically allocating an array allows us to choose an array length at runtime (meaning our length does not need to be constexpr).

its most common to dynamically allocate c-style arrays because for std::arrays its almost always better to use std::vectors in its place. The length of dynamically allocated arrays has type `std::size_t`. If you are using a non-constexpr int, you'll need to `static_cast` to `std::size_t` since that is considered a narrowing conversion and your compiler will warn otherwise.

```
#include <cstddef>
#include <iostream>

int main()
{
    std::cout << "Enter a positive integer: ";
    std::size_t length{};
    std::cin >> length;

    int* array{ new int[length]{} }; // use array new.  Note that length does not need to be constant!

    std::cout << "I just allocated an array of integers of length " << length << '\n';

    array[0] = 5; // set element 0 to value 5

    delete[] array; // use array delete to deallocate array

    // we don't need to set array to nullptr/0 here because it's going out of scope immediately after this anyway

    return 0;
}
```

Because we are allocating an array, C++ knows that it should use the array version of new instead of the scalar version of new. Essentially, the new[] operator is called, even though the [] isn't placed next to the new keyword.

One of the most common mistakes that new programmers make when dealing with dynamic memory allocation is to use delete instead of delete[] when deleting a dynamically allocated array. Using the scalar version of delete on an array will result in undefined behavior, such as data corruption, memory leaks, crashes, or other problems.

Most we can initialise the array one by one via index or with initialiser lists

```
int* array{ new int[5]{ 9, 7, 5, 3, 1 } }; // initialize a dynamic array since C++11
// To prevent writing the type twice, we can use auto. This is often done for types with long names.
auto* array{ new int[5]{ 9, 7, 5, 3, 1 } };
```

Dynamically allocating an array allows you to set the array length at the time of allocation. However, C++ does not provide a built-in way to resize an array that has already been allocated. It is possible to work around this limitation by dynamically allocating a new array, copying the elements over, and deleting the old array. However, this is error prone, especially when the element type is a class (which have special rules governing how they are created).

Consequently, we recommend avoiding doing this yourself. Use `std::vector` instead.

iterators & in-built useful functions and timing the code

https://www.learncpp.com/cpp-tutorial/introduction-to-iterators/
https://www.learncpp.com/cpp-tutorial/introduction-to-standard-library-algorithms/
https://www.learncpp.com/cpp-tutorial/timing-your-code/

errors and exception handling

https://www.learncpp.com/cpp-tutorial/introduction-to-testing-your-code/
https://www.learncpp.com/cpp-tutorial/code-coverage/
https://www.learncpp.com/cpp-tutorial/common-semantic-errors-in-c/
https://www.learncpp.com/cpp-tutorial/detecting-and-handling-errors/
https://www.learncpp.com/cpp-tutorial/stdcin-and-handling-invalid-input/
https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/
https://www.learncpp.com/cpp-tutorial/the-need-for-exceptions/
https://www.learncpp.com/cpp-tutorial/basic-exception-handling/
https://www.learncpp.com/cpp-tutorial/exceptions-functions-and-stack-unwinding/
https://www.learncpp.com/cpp-tutorial/uncaught-exceptions-catch-all-handlers/
https://www.learncpp.com/cpp-tutorial/exceptions-classes-and-inheritance/
https://www.learncpp.com/cpp-tutorial/rethrowing-exceptions/
https://www.learncpp.com/cpp-tutorial/function-try-blocks/
https://www.learncpp.com/cpp-tutorial/exception-dangers-and-downsides/
https://www.learncpp.com/cpp-tutorial/exception-specifications-and-noexcept/
https://www.learncpp.com/cpp-tutorial/stdmove_if_noexcept/