README

Athell Felix Ikechukwu O. Dec 31, 2023,
Dynamic arrays: std::vector

Introduction to containers and arrays

we be talking of vectors now.

The variable scalability challenge

Due to the fact that we would eventually interact with large quantity of data in some scenerio, it can prove cumbersome handling them (due to thier leviathan sized nature) without falling victim to error.
c++ has a solution to the problem: `containers`

Containers

think of containers as egg crates: they provide an easier, efficient and portable way of handling as many eggs as the crate is designed to hold.
containers in c++ exist with the same purpose as egg crates (in a way). they make it easier to manage (potentially) large collection of objects (called `elements`).
a `string` is a subtle example of containers in c++: they provide a collection of characters, which can be outputted as text.

The elements of a container are unnamed

while the container object itself has a name (else how would we refer to it?) its elements are unnamed: this is an `important` details of containers.
containers provide interfaces to enable access to the its elements.

The length of a container

teh number of elements in a container, in programming generally, is known as its `length`, or sometimes count.

```
std::string name { "Athell" };
std::cout << name << " has " << name.length() << " characters.\n";    // here the length member function is used to return the length of the container: std::string name.
```

Container operations

containiers should basically perform tasks:

1. create an container (empty or initialized)
2. access to elements
3. insert or removal of elements
4. get the number of elements in the container.
   any other opertaions, that should be beneficial to the logic of the container.

Element types

in most programming languages containers are `homogenous`, meaning the elements of the container are required to have the same type.
some containers use a preser type for its elements: like with `std::string` its elements are of type: `char`.

in c++ containers are usually made as class templates, to make it possible to have containers for any specified datatype and length: this makes it really convenient.

Introduction to arrays

an array is a container data-type that stored its elements `contiguously` (meaning everly element is placed in adjacent memory location with no gaps).
allowing fast and direct access to any element.

c++ generally contain 3 primary array types:

1. C-styled array : known to behave strangely and are dangerous, wth did you expect? its from fucking C.
2. std::vector container class
3. std::array container class

Introduction to std::vector and list constructors

here we be talking of `std::vectors`.

Introduction to std::vector

the `std::vector` is one of the containers in c++ library that implements an array. it is declared in the vector header as a class template, with a template parameter that defines the type of its elements. therefore a `std::vector<int>` defines a container with elements of type `int`.

```
#include <vector>

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

Initializing a std::vector with a list of values

most often we want to initialize our containerw with values: we can achieve this using list initialization and C++17 CTAD (Class Template Argument Deduction)

```
#include <vector>

int main()
{
  std::vector<int> arr_int { 1, 2, 3, 4 ,5 };         // explicit delcaraion of a container of int elements
  std::vetcor arr_char { 'a', 'b', 'c', 'f' };        // initializing and determining its type using CTAD.
}
```

List constructors and initializer lists

containers have a special constructor known as `list constructor`. it allows us to construct an instance of the container using an initializers list.
it does three things

1. ensures container has enough space to hold all the initialization values
2. sets the length of the container to the number of elements inputed
3. initializes the elemenst in sequntial order as listed in the list initializer.

Accessing array elements using the subscript operator (operator[])

just like in C: they are `zero based` tho.

an example:

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

int main()
{
  std::vector numbers { 1, 2, 3, 4, 5 };
  std::cout << "the first number is " << numbers[0] << '\n';
// `[x]` is the subscript operator with with x being the subscript: the
index to be accessed.
  std::cout << "the second number is " << number[1] << '\n';
}
```

Subscript out of bounds

when indexing an array: the index selected must be a valid element in the array, its subscript limit is usually `N-1` where N is the size length of the array.
the `operator[]` does not perform any sort of bound checking: meaning passing in an invalid index will lead to UB.

Arrays are contiguous in memory

one defining characteristics of arrays is that they are allocated contigously in memory: meaning they are located adjacent in memory with no gaps.
an example:

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

int main()
{
    std::vector primes { 2, 3, 5, 7, 11 };
    std::cout << "an int of of size " << sizeof(int) << '\n';
    std::cout << &(prime[0]) << '\n'
    std::cout << &(prime[1]) << '\n'
    std::cout << &(prime[2]) << '\n'
}
```

the results on my system:

```
an int of of size 4
0x55e84ecf3eb0                                                      # if you
look closely the addreess increment with 4 proving they are contigous.
```

```
0x55e84ecf3eb4
0x55e84ecf3eb8
```

arrays are one of the containers that allow `random access` meaing every element can be accessed directly with equal speed, regardless the length of the array.

Constructing a std::vector of a specific length

`std::vector` has an explicit ocnstructor `explicit std:vector<T>(int)` that takes in only one integer value to set the length std::vector when constructing.
an example:

```
std::vector<int> data (10);                          // we just declared
a vector of an immutable length: 10.
```

each of the elements are value-initialized and are therefore all initialized to zero.
this constructor must be called using `direct` intialization not `list`.

List constructors take precedence over other constructors

yup read this in the site. moral lesson is: list constuctors take precedene to other conctructors and thats why initializing with direct initialization will only set the
length of an array and zero initialize its members and using list will not (it just provide the elements).

note:

```
std::vector<int> num({ 10 });       // this is an alternative version of
copy list init.tion so this will not construct for specified length.
```

Declaring vectors as a member of a class type

due to the fac the direct initialization is not allowed for default memeber initializers: we cant do this to set the length of a vector:

```
struct Foo
{
    std::vector<int> vec ( 10 );                // error due to stated above
reason
};
```

but we can due this instead: creats an anonympus object and then use that to defaultly set the length of the vector via the copy constructor

```
struct foo
{
    std::vector<int> vec { std::vector<int>(10) }
}
```

Const and constexpr std::vector

they can be be made const and this makes their elements immutable: also they must be intialized:

```
const std::vector<int> vec {1, 2, 3, 4};
vec[0] = 9;                                          // error they are
immutable.
```

also normal std::vector elemenst are not allowed to be const: the must be mutable.
so doing something like this is prohibited.

```
std::vector<const int> vec { 1, 2, 3, 4 };          // illegal: error:
`<const int>` cant make the int const cause that will meke the elements
immutable which is not allowed
                                                     // vectors.
```

std::vector and the unsigned length and subscript problem

read this section and reread the paper bjourne stroustrop wrote regarding the mistake of using `unsigned integers` for subscripting due to it
bound wrapping into garbage values because it relies on modular arithmetics

Paper: why subscipts should be signed

A review: sign conversions are narrowing conversions, except when constexpr

`unsigned int` to `signed int` or the other way around are narrow conversions, which list initialization do not allow
but when made `constexpr` they allow them because the compiler would have to make sure the coversion is safer (at compiler time) otherwise halt the compilation

so narrow converting like declarations are allowed if the varaible to be narrowed is a `constexpr`.

```
#include <iostream>

void foo(unsigned int)
{}

int main()
{
    constexpr int s { 5 };                   // now constexpr
    [[maybe_unused]] unsigned int u { s }; // ok: x is constexpr and can be
converted safely, not a narrowing conversion
    foo(s);                                  // ok: x is constexpr and can be
converted safely, not a narrowing conversion

    return 0;
}
```

copy initialization do allow narrow conversions tho.
also: the non narrowing conversion of `constexpr int` to `constexpr std::size_t` is something we will use a lot. according to what i just read.

The length and indices of std::vector have type size_type

yup. and `std::size_t` is an alias for an `unsigned long` or `unsigned long long`.
each of the STL container classes define a nested typedef of `size_type`, which is the alias for they type used for the containers length or (indices if supproted).
almost always the `size_type` is usually an alias for `std::size_t`, but it can be overidden (rarely done).

when accessing the `size_t` member of a conatainer class er must scope the class with its full template declararion
an example:

```
std::vector<int>::size_type
```

Getting the length of a std::vector using the size() member function or std::size()

an example:

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

int main()
{
    std::vector<int> vec { 1, 2 , 3, 4, 5 };
    std::cout << "the size of vec is " << vec.size() << " used .size()\n";
    std::cout << "the size of vec is " << std::size(vec) << " used
std::size()\n";
}
```

in c++ most containers have thier length member functions dub as size, except in strings and string_view that have both length and size.

note:

- favor using `std::size()` that constainers `.size` this is becuase it works on a broader set of containers like non-decayed c-styled arrays.

if we wasnt to use the `std::size()` or `.size` to initialize some variable with the size of a container, we might sometimes run into signed/unsigned conversion warnings or errors.
we can prevent this by `static_cast`ing to the type same with the variable to be stored into.
an example:

```
#include <vector>

int main()
{
    std::vector<int> vec {1, 2, 3, 4, 5};
    int length { static_cast<int>( vec.size() ) };              // error
free!
}
```

Getting the length of a std::vector using std::ssize() C++20

it us a function that returns the length of a container as a large signed integeral type (usually `std::ptrdiff_t`), its usesd as a signed conterpart of `std::size_typr`
if we want to uses this as a method for assigning the length of a container to an `int` we have some options:

- first: using the `static_cast` to prevent narrow conversions:

```
std::vecttor num { 1, 2, 3, 4 };
int length { static_cast<int>( std::ssize(num) ) };
```

- or we can use the auto keyword to deduce the type of the length to be returned by the intgeral type:

```
std::vecttor num { 1, 2, 3, 4 };
auto length { std::ssize(num) };
```

Accessing array elements using operator[] does no bounds checking

yeah it doesnt if indexing for an out of bound index, UB can occur.

Accessing array elements using the at() member function does runtime bounds checking

yeah if performs bpund checking at runtime so if an out of bound index is requested using the `at()` member function it halts the program at runtime.
an exmaple:

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

int main()
{
    std::vector vec { 1,2 ,3, 4, 5 };
    std::cout << vec.at(4);
// this is fine. still in bound.
    std::cout << vec.at(8);
// terminates program at runtime: not in bounds.
}
```

when the `.at()` member function tries to acces an out of bound indec it throws an `std::out_of_range` error and terminates the program.
becuae of its runtime evaluation nature its slower than `operator[]` indexing but more safer.

Indexing std::vector with a constexpr signed int

we can do this tp enable the compiler convert the constexpr signed int to a std::size_t withput it being a narrow conversion.

an example:

```
std::vector<int> num { 1, 2, 3, 4 };
std::cout << num[2];

constexpr int index { 3 };
std::cout << num[ index ];      // not a narrow conversion.
```

dont get the use of this but okay.

Indexing std::vector with a non-constexpr value

the subscripts used to index an array can be non const.
an example:

```
std::vector num { 1, 2, 3, 4 };
std::size_t index { 2 };

std::cout << num[index];                    // operator[] requires an index
of type std::size_t so no narrow conversion required.
```

whn our subscripts are non-constexpr signed integer we run into some issues.
an example:

```
std::vector num { 1, 2, 4, 5 };
int index { 3 };
std::cout << num[index];                        // hers a problem occurs:
narrow conversion (implicilty converted signed int to std::size_t).
```

this might raise some warnings (not errors) if warning flags where set, this can be quite annoying.
we can prevent this by always:

1. `static_cast`ing our subcripts when not of the right type which is `std::size_t`
2. or just always using the `std::size_t` as the type of subsripts whenever.

Passing and returning std::vector, and an introduction to move semantics

the `std::vector` element type is part of the type information of the object. so when we declare `std:vectors` as parameter ina function.

we declare them with thier full template declaraion. amd becuase vectors can be quite expensive to copy we pass the as references.
an example:

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

voud passByRef(std::vector<int>& arr)                 // we must explicitly
specify the <int>.
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector vec { 1, 2, 3, 4, 5 };
    passByRef(vec);
    retunr 0;
}
```

Passing std::vector of different element types

vbebcuae the passByRef function only takes in vectors of element type `int` we cannot pass in another vectors of a different element type.

in c++17+ this can be bypassed tho using CTAD (class template argument deduction)
an exmaple:

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

voud passByRef(const std::vector& arr)                 // no need to specify
the <int> to allow CTAD work.
{
    std::cout << arr[0] << '\n';
}

int main()
{
    std::vector prime { 1, 2, 3, 4, 5 };
    passByRef(prime);                            //  CTAD uses its argument
to infer that te vector should be of type std::vector<int>
}
```

we can alos make the function a template function instead of reling on CTAD:
an exmaple:

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

template <typename T>
void passByRef(const std::vector<T>& arr)
{
    std::cout << array[0];
}
```

Passing a std::vector using a generic template or abbreviated function template

pass

Asserting on array length

when we try acces an index thats out of bounds of a vector it leads to UB.
so we can asser on the size first: if logically acceptable we can proceed.

Returning a std::vector

when we need to pass in a vector into function we pass it as a cont reference just not to make a copy.
but for some god dammed reason its okay to return it by value.

Introduction to move semantics

when inaitializing or assigning a value to an object copies value from another object we say its operating with `copy semantics`
sometimes when we intialize `some` objects with another objects of the same datatype instead of we accomplishing out goal with `copy semantics` the compiler performs `move semantics` instead.

this is when instead of copying the data from an object to intialize another object, we move the data of the initializing object to the one to be initialized (`move semantics`)
there are 2 criterias for this to occur:

1. the datatype must support move semantics

2. the object to be intialized must be initrialized with another object of the same datatype thats an `rvalue`.

not many datatypes support, `std::vector` and `std::string` support move semantics tho.

an example of move semantics in c++:

```
#include <vector>

std::vector generateVec()
{
    std::vector vec { 1, 2, 3, 4, 5 };
    return vec;
}

int main()
{
    std::vector vec { generateVec() };          // here move semantics occur
becuase firstly std::vector supports it and
                                                // secondly its being
intialized with an rvalue object returned by the generateVec() function.
}
```

We can return move-capable types like std::vector by value

Because of std::vector supports move semantics we can return by value without worrying about object duplication.
making retunr by value extreemely inexpensive for this types.

Arrays and loops

an example:

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

int main()
{
    std::vector scores {23, 455, 65, 76, 23};
    std::size_t length { std::size(scores) };

    int average { 0 };
    for (std::size_t index { 0 }; index < length; ++index)
```

```
        average += scores[index];
    average /= static_cast<int>( std::size(scores) );

    std::cout << "average score is " << average << '\n';
    return 0;
}
```

accessing the elements of a container in some order is known as `traversal` or `iterating through`.

Templates, arrays, and loops unlock scalability

an example:

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

template <typename T>
T average(const std::vector<T>& array)
{
    std::size_t len { array.size() };

    T average { 0 };
    for (std::size_t index { 0 }; index < len; ++index)
        average += array[index];
    average /= static_cast<int>(len);

    return average;
}

int main()
{
    std::vector class1 { 84, 92, 76, 81, 56 };
    std::cout << "The class 1 average is: " << calculateAverage(class1) <<
'\n';

    std::vector class2 { 93.2, 88.6, 64.2, 81.0 };
    std::cout << "The class 2 average is: " << calculateAverage(class2) <<
'\n';

    return 0;
}
```

Arrays and off-by-one errors

fuck this shi

Arrays, loops, and sign challenge solutions

we all know that container subscript, index and length type are `std::size_t` which is an unsigned integeral.
because of the unstable nature of unsigned types when interacting with negative values: this can pose some issues like so:

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

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

    for (std::size_t index { vec.size() -1 }; index >= 0; --index)
        std::cout << vec[index] << ' ';
}
```

firstly it iterates an prints the vector elements in reversal as expected then after (and at) the iteration where index becomes -1 undefined behaviour occurs.
this is due to the bound wrapping action unsigneds take when interacting with negative values.
because it bounds to some very large garbage value whenever
`index` should be negative the loop never ends (due to its `index >= 0` condition) and prints garbage values infinitely until the program crashes.

athough there are a some ways to fix this issue. using `signed` integral type can easily fix this but it itself has its own downsides.
an example:

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

int main()
{
    std::vector vec { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    for (int index {static_cast<int>( vec.size()-1 )}; index >= 0; --len)
        std::cout << vec[static_cast<std::size_t>(index)] << ' ';
}
```

in this snippet although its works the cluttering of code is crazy: which reduces readablity.
although the cluttering is not that severe in this example it becomes
really much in lerger code. so yeah `unsigned` compromises execution safety when interacting with negaative but `signed` creates a lot of clutter.
so what can we do?

Leave signed/unsigned conversion warnings off

makes debuginng easier and prevents being drowned with a buttload of warnings.

Using an unsigned loop variable

some developers prefer using this paradigm but we have to careful to prevent running into signed/unisgned mismatches when doing so.

1. using the `size_type` of conatiners (usually its just an alias of `std::size_t` which is just an unsigned long (long))
   an example:

   ```
   std::vector arr { 1, 2, 3, 4, 5 };

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

   however using `size_type` has some downsides: its nested so clutters code and when used in a function template, because it becomes a nested
   dependent type (due to being used in a template function that takes in template parameter to genereta the the type of vector container thus making
   it dependent) we have to prefix the `typename` keyword when declaring it. you have to use the prefix `typename` whenver declaring a dependent type
   an example:

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

   template <typename T>
   void printArray(std::vector<T>& arr)
   {
       for (typename std::vector<T>::size_type index { 0 }; index <
   arr.size(); ++index)
           std::cout << arr[index];
   }
   ```

```
int main()
{
    printArray( std::vector {1, 3, 4, 5, 6, 7} );
}
```

any name that depends on a type containing a template paramater is called a `dependent name`.
and they must be prefixed with the `typename` keyword.

sometimes soem developers use alias to make loops easier to read.
an example:

```
using arrayi = std::vector<int>;
for (array1::size_type index { 0 }; index < arr.size(); ++index)
```

2. We can also tell the compiler to return the type of the container using the `decltype` keyword, whih returns the type of its parameter.
an example:

```
for (decltype(arr)::size_type index { 0 }; index < arr.size(); ++index)
```

however if arr is a reference the above doesnt work so we have to remove the reference:
and to do that we:

```
template <typename T>
void printArray(std::vector<int>& arr)
{
    for ( typename std::remove_reference_t<decltype(arr)>::size_type
index {0}; index < arr.size(); ++index )
        std::cout << arr[index];
}
```

undortunately this is neither concise or easy to remeber so most times programmers just use `std::size_t` becasue most of the times
containers `size_type` are usually aliases of `std::size_t`.

Using a signed loop variable

What signed type should we use?

1. with small arrays `int`

2. with very larger arrays, or if wantung to be a bit defensive we use the strangely named `std::ptrdiff_t`. its a typedef of the positive counterpart to `std::size_t`.
3. becuase it has a wierd name long name, another approach is using an alias for it. an example:

```
using Index = std::ptrdiff_t

for (Index index { 0 }; index < static_cast<Index>(arr.size()); ++index)
```

4. in cases where you can deuce the type of your loop variable from from the initializer, you can use the `auto` to have the compiler deduce the type.

```
for (auto index {static_cast<std::ptrdiff_t>( arr.size-1 )}; index >= 0;
--index)
```

in the c++23 the `Z` suffix was introduced as a literal definer for the signed counterpart of `std::size_t` (probably `std::ptrdiff_t`)

```
for (auto index {0Z}; index < static_cast<std::ptrdiff_t>(arr.size());
++index)
```

Getting the length of an array as a signed value

1. using `static_cast` to cast the return of std::size() or .size() to a signed. an example:

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

using Index = std::ptrdiff_t

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

    for (auto index { 0Z }; index < static_cast<Index>(arr.size()-1);
++index)
        std::cout << arr[static_cast<std::size_t>(index)] << ' ';
}
```

this works well but the downside is how much it clutters code. one way to elimated ths is removing the length variable out of the
loop block

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

using Index = std::ptrdiff_t

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

    auto len { static_cast<Index>(arr.size) };
    for (auto index { len-1 }; index >= 0; --index)
        std::cout << arr[static_cast<std::size_t>>(index)] << ' ';
}
```

more evidence that the designers of c++ want to push the paradigm of interacting with arrays and their indices using `signed` integrals
is the introduction of `std::ssize()` in C++20. the function retunrs the size of a container as a signed intgral (likely `std::ptrdiff_t`)

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

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

    for (auto index{ std::ssize(vec)-1 }; index >= 0; --index)
        std::cout << vec[static_cast<std::size_t>(index)] << ' ';
}
```

Converting the signed loop variable to an unsigned index

once we have a signed loop variable, we woul run into wanring for implicit signed to unsigned conversions. so we need ways to curb this.
solutions:

1. using `static_cast` to cast to unsigned where neccessay: works but clutters code.
2. use a conversion function witha short name.

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

using Index = std::ptrdiff_t;

constexpr std::size_t getU( Index index )
{
    return static_cast<std::size_t>(index)
}

int main()
{
    std::vector vec {1, 2, 4, 5, 6, 7};

    auto len { static_cast<Index>(vec.size()) };
    for(Index index { len-1 }; index >= 0; --index)
        std::cout << vec[getU(index)];
}
```

3. creats a custome view for arrays that do exactly what you want.
   our custom array wsuld be able to do this 2 to any container that supports indexing

   1. retrieve the element of an array.
   2. return the size of an array.

   array_view.h:

```cpp
#ifndef CUSTOM_ARRAY_VIEW
#define CUSTOM_ARRAY_VIEW

template <typename T>
class s_array_view
{
private:
    T& m_array;

public:
    using Index = std::ptrdiff_t;

    s_array_view(T& array) : m_array { array } {}

    constexpr auto& operator[](Index index) {
        return m_array[static_cast<typename T::size_type>(index)];
    }
    constexpr const auto& operator[](Index index) const {
        return m_array[static_cast<typename T::size_type>(index)];
    }
    constexpr auto ssize() {
        return static_cast<Index>(m_array.size());
    }
};
#endif
```

main.cpp:

```
#include <iostream>
#include <vector>
#include "array_view.h"

int main()
{
    std::vector array { 1, 2, 3, 4, 5, 6 };
    s_array_view sv_array { array };

    for (auto index { sv_array.ssize()-1 }; index >= 0; --index)
        std::cout << sv_array[index] << ' ';
}
```

The only sane choice: avoid indexing altogether!

all options presented here has their own downsides. so its difficult to recommend a best practice.
however theres a choice far more sane than the others: avoid indexing with integral values altogether.

this can be done using for-each loops or iterators. they will be discussed later.
and we should use them than interacting with intergral values for traversals whenever possible (*best practice*).

Range-based for loops (for-each)

altho forloops provide in interface for traversing through a container, they often lead to one-off errors, easy to mess up and are subjected to array indexing sign problems.
because traversing forward in an array is such a common thing to do c++ provides for loop called the `ranged based` for loop. which helps iteration and traversion of a container without explicit indexing.

ranged based forloops are simpler, safer and work with all common array types in c++ including std::vector, std::array and c-style arrays.

Range-based for loops

syntax:

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

when a range based loop is encountered, the loop is traversed and assigns the current iterated upon element to the element_declaration, then the statement is executed.
for best result element_declaration should have the same type as the elements in the array or else type conversion will occur.

an example:

```
#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)
        std::cout << num << ' ';
    std::cout << '\n';
}
```

becuase num is assigned the value of the elements of the array, through copy assignment this can be quite expensive for some datatypes.

Range-based for loops and type deduction using the auto keyword

because element_declaration should be of the same data-type as the elements in the array to be traversed. this is an ideal case to use the `auto` keyword
for type deduction of the element.
an example:

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

```cpp
int main()
{
    std::vector array { 1, 2, 3, 4, 5, 6 };

    for (auto num : array)
        std::cout << num;
}
```

this is a best practice.

Avoid element copies by using references

copying some elements in an array can be expensive so we can assign them to references thereby circumventing the problem of the making expensive copies.

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

int main()
{
    using namespace std::literals;
    std::vector words{ "peter"s, "likes"s, "frozen"s, "yogurt"s };

    for (const auto& word : words)
        std::cout << word << ' ';
    std::cout << '\n';
}
```

if the element_declaration is made non-const we will be able to modify the elements of the array(something not possible if our element declaration is a copy of the value).

Consider always using const auto& when you don’t want to work with copies of elements

normaly we use:

- `auto` : for cheap to copy elements
- `const auto&` : for elements that are expensive to copy and we require only read access to them.
- `auto&` : for elements that are expensive to copy but we desire both read and write access to them.

case where using auto can be acceptable:

```
using namespace std::literals;
std::vector words{ "peter"sv, "likes"sv, "frozen"sv, "yogurt"sv };
// sv is a literal suffix string_views.

for (auto word : word)  // the auto keyword is fine here only becuase the
type deduces to a string_view that acts as an inexpensive view for strings
(just like a reference but only for strings).
    std::cout << word;
```

for expensive to copy elements like strings: a string_view, `const auto&` or `auto&` are more memory conservative.

ranged based for loops dont work on decayed C-style arrays (whatever the fuckk that is).

Getting the index of the current element

ranged based forloops dont provide an interface to access the index of the elements it traverses over. but this issue can be resolved by creating you own personal counter.

Range-based for loops in reverse : `c++20`

range based for loops only traverse in a forwaed order, hover there are cases where we might want to traverses an array in a reverse order.
in c++20 we can use the `std::views::reverse` of the `<ranges>` library to create a reverse view of the elements to be traveresed.
an example:

```
#include <iostream>
#include <vector>
#include <ranges>   // c++20
#include <string_view>

int main()
{
    using namespace std::literals;
    std::vector words{ "Alex"sv, "Bobby"sv, "Chad"sv, "Dave"sv };

    for (const auto& word : std::views::reverse(words))
        std::cout << word;
}
```

Array indexing and length using enumerators

one of the biggest flaws of array indexing is that the subscript tells us nothing about the index or the element to be retreived.

using unscoped enumerators for indexing

subscipts of caintainers like `std::vector` and its likes have thier type as `conatainer::size_type` which is usually an alias for `std::size_t`.
and since unscoped enumarations will implicitly convert to `std::size_t` we can use the for indexing to help document the meaning of an index.

```
#include <vector>

namsepace Students
{
    enum Names
    {
        toby,
        dwight,
        jan,
        micheal,
        creed,
        kevin,
        max_employees
    };
}

int main()
{
    std::vector scores { 22, 333, 455, 666, 88, 999 };
    scores[Students::toby] = 100;
// here were updating the score of student toby.
}
```

becuase enumerations are implicitly constexpr, conversion of an enumerator to an unsigned integral type is not considered a narrow conversion, thus avoiding signed/unsigned indexing problems.

Using a non-constexpr unscoped enumeration for indexing

the underlying type of enumarators are implamantaion defined (so they could be either signed or unsigned). and because they are implicitly constexpr we wont run into
any warings when using them to index arrays directly (this is because constexprs do not narrow convert if they are used in a context that can work but might narrow convert for normal variables types: I suspect this is due to compiler optimizations for compile time variables like contexprs).

so we experience no narrow conversion warnings when used directly but we might experience if the enumartor is made non-constexpr:
an example:

```
#include <vector>

namsepace DunderMifflin
{
    enum Scranton
    {
        toby,
        dwight,
        jan,
        micheal,
        creed,
        kevin,
        max_employees
    };
}

int main()
{
    std::vector workScores {7, 9 ,9, 5, 6, 7};
    DunderMifflin::Scraton boss { DunderMifflin::micheal };          // non-
constexpr

    workScores[boss] = 10;                                           //
poosible narrow conversion warning becase boss is now a non constexpr
enumerator,
                                                                     // and
might default to a signed undertlying type.
}
```

we can fix this by specifying the underlying type of the enumarations.
an example:

```
#include <vector>

namsepace DunderMifflin
{
    enum Scranton : unsigned int    // specified.
    {
        toby,
        dwight,
        jan,
        micheal,
        creed,
        kevin,
        max_employees
    };
}

int main()
{
    std::vector workScores {7, 9 ,9, 5, 6, 7};
    DunderMifflin::Scraton boss { DunderMifflin::micheal };         // non-
constexpr but unsigned,

    workScores[boss] = 10;                                          // sign
conversion does not occur since unigned int can be converted to std::size_t
with no issues
                                                                    // i
think that integral promotion?
}
```

Using a count enumerator

if you notice we've been adding the enumaration `max_employees` as the last enumerator to all our enumarators. if all proior enumerators of the enumaration are using thier
default values this means that the `max_employees` enumerator should hold length value or count value of the valid enumerators.
we call this the `count enumerator` as it represnts the count of the prior enumarators:
an example:

```
#include <vector>

namsepace DunderMifflin
{
    enum Scranton : unsigned int    // specified.
    {
        toby,
        dwight,
        jan,
        micheal,
        creed,
        kevin,
        max_employees
    };
}

int main()
{
    std::vector<int> scores (DunderMifflin::max_employees);         //
create a vector that holds 5 elements.
    scores[DunderMifflin::kevin] = 78;                              //
updating the score that belongs to kevin.
    std::cout << DunderMifflin::max_employees << '\n';              // here
its used to prints the length of the scores or how many employees there are.
}
```

this technique is nice because if an enumerator should get larger we have the nice interface for adding new enumerators and the length is automatically updated
as long as any new enumerator is declared before the count enumerator making it always last to hold the count.

Asserting on array length with a count enumerator

this is useful for always making sure that the enumaration count enumerator is of the right value: which should be equal to the size of the vector it acts as an indexer for.
an example:

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

enum StudentNames
{
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5
};
```

```
int main()
{
    std::vector socres { 1, 2, 7, 4, 5, 6, 7, 8 };

    assert(std::size(scores) == max_students);
    // proceed to interact with array using its mapped unscoped enumerations
interface for indexing.
}
```

use `static_assert` if your array is constexpr, vectors do not support constexprs but `std::array` does.

Arrays and enum classes

because unsscped enumerators pollute the scope they are declared, we can opt to use the scoped enumarators: `enum classes`
but they do not support implicity conversions to their underlying types and becuase of this might run into issues when trying to
use thier enumerators as array indices.

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


enum class StudentNames // now an enum class
{
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5
};

int main()
{
    std::vectors<int> students ( static_cast<int>
(StudentNames::max_students) );

    students[static_cast<size_t>(StudentNames::stan)] = 33;
    std::cout << static_cast<int>(StudentNames::max_student) << '\n';
}
```

although this works in eradictaing the pollution of the current scope problem, it also introdeuces some new ones:

1. a pain to write, clutters code alot due to the need to specify the namespace.
2. a lot of static_cast which leads to more clutters
3. due to all the wring might lead to errors.

we can fix this though by overloading the `operator+`:

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

enum class StudentNames // now an enum class
{
    kenny, // 0
    kyle, // 1
    stan, // 2
    butters, // 3
    cartman, // 4
    max_students // 5
};

constexpr auto operator+(StudentNames a) noexcept {
    return static_cast<std::underlying_type_t<StudentNames>>(a);
}

int main()
{
    std::vector<int> students (+StudentNames::max_students);

    students[static_cast<size_t>(StudentNames::stan)] = 33;
    std::cout << +StudentNames::max_students << '\n';
}
```

std::vector resizing and capacity

in this subtopic we are focusing on what makes `std::vector` significantly different than other array types: the ability to resize itself after being instantiated.

Fixed-size arrays vs dynamic arrays

most arrays have a limitation that is the size should be known at the point of instantiation and then cannot be changed. such arrays are called `fized-size arrays`.
both `std::array` and `C Style arrays` are fixed lengthed.

on the other hand std::vector is a `dynamic array`. a dynamic array is an a rray whose size can be changed after instantiation. the abilty to be resizable
is what makes `std::vector` special.

Resizing a std::vector at runtime

this is done by calling the `.resize()` with the size as the argument.
an example:

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

int main()
{
    std::vector vec { 1, 2, 4 };
    std::cout << "the length is: " << vec.size() << '\n';

    vec.resize(5);
    std::cout << "the length is: " << vec.size() << '\n';

    for (auto i : vec)
        std::cout << i << ' ';
    std::cout << '\n';
}
```

this prints

```
the length is: 3
the length is: 5
1 2 4 0 0
```

vectors can alos be reiszed to be smaller: it preserves the first occuring elements and discards the later elements to fit the new size.

The length vs capacity of a std::vector

the length of a vectors is the number of elements in a vector that is considered in use.
the *capacity* is the number of elements with allocated memory in the vector: it comprises of both in use and not in use elements.
we can get the capacity of a vector by calling the `.capacity()` function.

Reallocation of storage, and why it is expensive

when `std::vector` changes the amount of storage its managing this is known as **reallocation**.
what happens under the hood when reallocation occurs:

1. `std::vector` acquires new memory sufficient for the new capacity of elemnst to be catered for
2. the elements in the old memory gets copied (or moved, if possible) to the newly allocated storage and the old memory gets returned back to the system.
3. the capacity and length of the `std::vector` object are updated to conform to the new storage.

what happens when reallocation occurs is basically hidden replacement with the old std::vector object being replaced by a new one.
sounds expensive? thats because it is. we should avoid resizing where possible.

Why differentiate length and capacity?

basically at a vectors initialization capacity and length are the same.
when the vector gets expanded (via resizing and reallocation) the capacity and length, although incremented, resolve to be equal.
but something specaial happens when a vector is shruken.

refer to cpp file : test_capacity_length_vector.cpp
and run executanles file : test_capacity_length_vector
to notice the difference of them

something to note (personal findings):

1. once the capacity of a vector increases it doesnt reduce (implicitly i think)
2. the length can increase or reduce

tracking capacity seperately from length allows `std::vector` to avoid reallocations when length is changed.

Vector indexing is based on length, not capacity

that makes sense: becuase length leeps track of elements in use.
why would we want to index for an element that isnt in use?

A subscript is only valid if it is between 0 and the vector’s length (not its capacity)!

Shrinking a std::vector

resizing a vector to be larger will increase the vectors length, and will increase the capacity if required. however resizing a vector
to be smaller will only decrease its length, and not its capacity.

to fis this vectors have a member function `.shrink_to_fit()` that request that the vector shrinks its capacity to match its length
the requests is non-binding meaning the compiler is free to ignore it.
an example:

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

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

int main()
{
    std::vector num { 1, 3, 4, 5, 6 };
    printCapLen(num);

    num.resize(0);
    printCapLen(num);

    num.shrink_to_fit();
    printCapLen(num);
}
```

when `shrink_to_fit()` is called the compiler reallocated its capacity to 0: freeing 1000 allocated objects.

std::vector and stack behavior