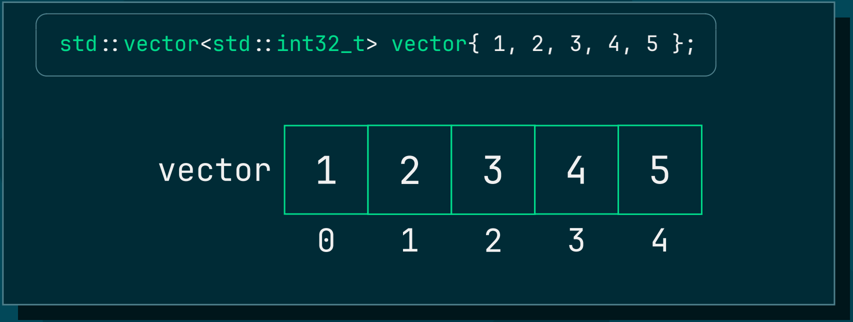


```
std::vector<std::int32_t> vector{ 1, 2, 3, 4, 5 };
```

Vectors

Data Structures – C++ Fundamentals

Agenda

- What Are Vectors?
  - Properties
  - Benefits
  - Drawbacks
- Dynamic-array Details
  - Abstract Data Types
  - How Do They Work?
  - Operations
  - Vector-specific Details
- Dynamic Allocation
  - The Stack
  - Return-value Optimization
  - The Heap
  - Vectors and the Heap
- Working with Vectors
  - Syntax
  - Undefined Behavior
  - Iterator Invalidation

What Are Vectors?

Vector Properties

What is a vector?

Vectors are C++'s implementation of a data abstraction known as a *dynamic array*

Vector Properties

What is a vector?

Vectors are C++'s implementation of a data abstraction known as a *dynamic array*

What is a dynamic array?

A *dynamic array* is a dynamically sizable array

Vector Properties

What is a vector?

Vectors are C++'s implementation of a data abstraction known as a *dynamic array*

What is a dynamic array?

A *dynamic array* is a dynamically sizable array

- Dynamic arrays share the same properties as arrays

Vector Properties

What is a vector?

Vectors are C++'s implementation of a data abstraction known as a *dynamic array*

What is a dynamic array?

A *dynamic array* is a dynamically sizable array

- Dynamic arrays share the same properties as arrays
- Dynamic arrays resize themselves as elements are added

Dynamic-array Properties

Dynamic-array Properties

- The sizes of dynamic arrays change as more elements are added

Dynamic-array Properties

- The sizes of dynamic arrays change as more elements are added
- The initial size of a dynamic array is known at compile-time, but dynamic-array sizes will change at *runtime*

Dynamic-array Properties

- The sizes of dynamic arrays change as more elements are added
- The initial size of a dynamic array is known at compile-time, but dynamic-array sizes will change at *runtime*
- Values are stored contiguously

Dynamic-array Properties

- The sizes of dynamic arrays change as more elements are added
- The initial size of a dynamic array is known at compile-time, but dynamic-array sizes will change at *runtime*
- Values are stored contiguously
- Values can be accessed at random

Dynamic-array Properties

- The sizes of dynamic arrays change as more elements are added
- The initial size of a dynamic array is known at compile-time, but dynamic-array sizes will change at *runtime*
- Values are stored contiguously
- Values can be accessed at random
- Values are ordered (indexed for dynamic arrays)

Benefits of Dynamic Arrays

Benefits of Dynamic Arrays

- Dynamic-size benefits

Benefits of Dynamic Arrays

- Dynamic-size benefits
  - Can store an unspecified number of elements

Benefits of Dynamic Arrays

- Dynamic-size benefits
    - Can store an unspecified number of elements
- Contiguous benefits

Benefits of Dynamic Arrays

- Dynamic-size benefits
    - Can store an unspecified number of elements
- Contiguous benefits
    - Great cache locality

Benefits of Dynamic Arrays

- Dynamic-size benefits
  - Can store an unspecified number of elements
- Contiguous benefits
  - Great cache locality
  - Fast element lookup

Benefits of Dynamic Arrays

- Dynamic-size benefits
    - Can store an unspecified number of elements
- Contiguous benefits
    - Great cache locality
    - Fast element lookup
- Sequence benefits

Benefits of Dynamic Arrays

- Dynamic-size benefits
    - Can store an unspecified number of elements
- Contiguous benefits
    - Great cache locality
    - Fast element lookup
- Sequence benefits
    - Elements can be traversed in particular orders

Benefits of Dynamic Arrays

- Dynamic-size benefits
  - Can store an unspecified number of elements
- Contiguous benefits
  - Great cache locality
  - Fast element lookup
- Sequence benefits
  - Elements can be traversed in particular orders
  - Can be subsequenced

Drawbacks of Dynamic Arrays

Drawbacks of Dynamic Arrays

- Dynamic size isn’t free

Drawbacks of Dynamic Arrays

- Dynamic size isn’t free
    - Dynamic arrays use more memory than the minimum amount of memory needed to store their elements

Drawbacks of Dynamic Arrays

- Dynamic size isn’t free
    - Dynamic arrays use more memory than the minimum amount of memory needed to store their elements
    - Dynamic-array data is stored on the heap

Dynamic-array Details

Abstract Data Types

An *abstract data type* (ADT) is a theoretical representation of a data type which outlines its properties and abilities

Abstract Data Types

An *abstract data type* (ADT) is a theoretical representation of a data type which outlines its properties and abilities

- They don't mention concrete implementation details

Abstract Data Types

An *abstract data type* (ADT) is a theoretical representation of a data type which outlines its properties and abilities

- They don’t mention concrete implementation details
- They allow for classifying a data type based on its properties and operations rather than on its implementation

Abstract Data Types

An *abstract data type* (ADT) is a theoretical representation of a data type which outlines its properties and abilities

- They don’t mention concrete implementation details
- They allow for classifying a data type based on its properties and operations rather than on its implementation
- Programmers can abstract away any implementation details that don’t matter by using ADTs

Data Abstractions

A *data abstraction* is a specification of operations on a set of values

Data Abstractions

A *data abstraction* is a specification of operations on a set of values

- Data abstractions are ADTs that are meant for representing data structures

Data Abstractions

A *data abstraction* is a specification of operations on a set of values

- Data abstractions are ADTs that are meant for representing data structures
- *Data structures* are concrete implementations of data abstractions using a particular representation of the data

Data Abstractions

A *data abstraction* is a specification of operations on a set of values

- Data abstractions are ADTs that are meant for representing data structures
- *Data structures* are concrete implementations of data abstractions using a particular representation of the data
- A dynamic array is a data abstraction since there are multiple data structures that could implement it

How Might We Implement a Dynamic Array?

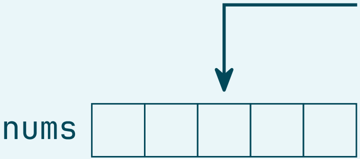
How Might We Implement a Dynamic Array?

- Let’s say we want to write a program which asks a user to enter an unlimited number of integers until they enter “stop”

How Might We Implement a Dynamic Array?

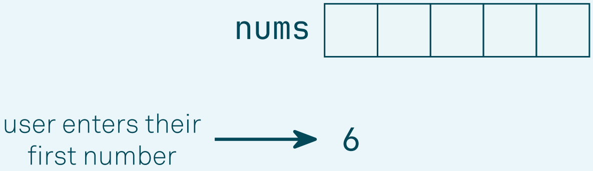
- Let’s say we want to write a program which asks a user to enter an unlimited number of integers until they enter “stop”
- How could we store the user’s numbers using only `std::array`?

How Might We Implement a Dynamic Array?



create initial array
with most probable size

How Might We Implement a Dynamic Array?

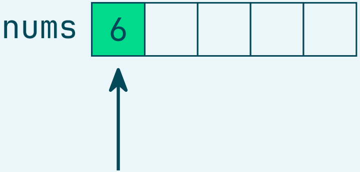


`nums`

user enters their first number

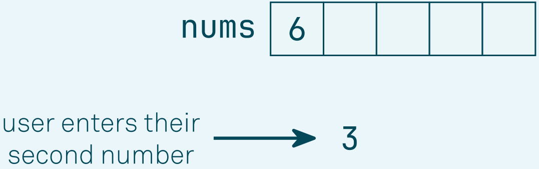
6

How Might We Implement a Dynamic Array?



number is stored
in first slot

How Might We Implement a Dynamic Array?



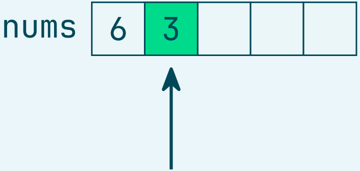
nums

6

user enters their second number

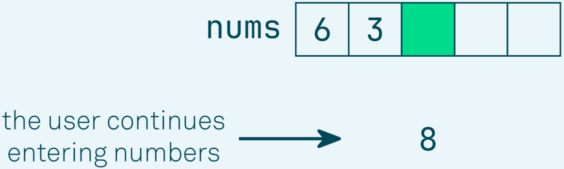
3

How Might We Implement a Dynamic Array?



number is stored
in second slot

How Might We Implement a Dynamic Array?



nums

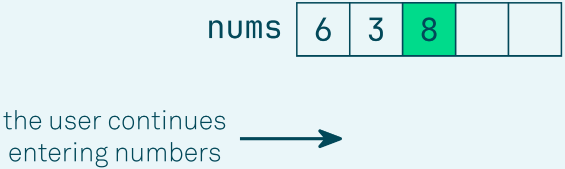
6

3

the user continues entering numbers

8

How Might We Implement a Dynamic Array?

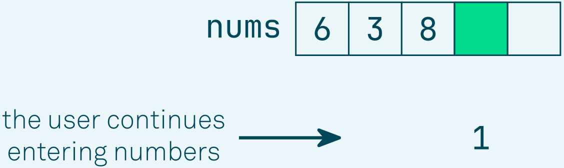


nums

6	3	8		

the user continues entering numbers →

How Might We Implement a Dynamic Array?



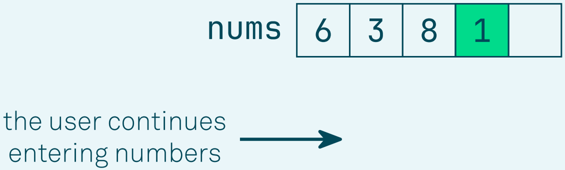
nums

6	3	8	🟩	

the user continues entering numbers →

1

How Might We Implement a Dynamic Array?

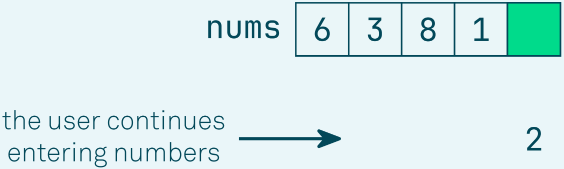


nums

6	3	8	1	

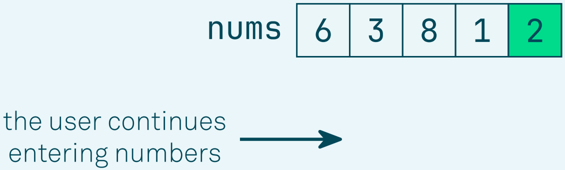
the user continues entering numbers

How Might We Implement a Dynamic Array?



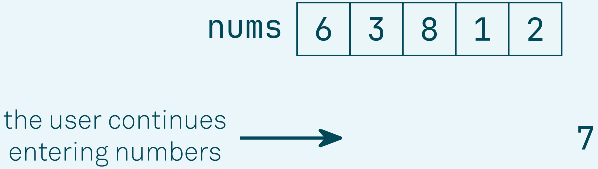
the user continues entering numbers

How Might We Implement a Dynamic Array?



the user continues entering numbers

How Might We Implement a Dynamic Array?

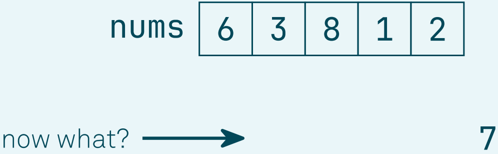


nums					
	6	3	8	1	2

the user continues entering numbers

7

How Might We Implement a Dynamic Array?

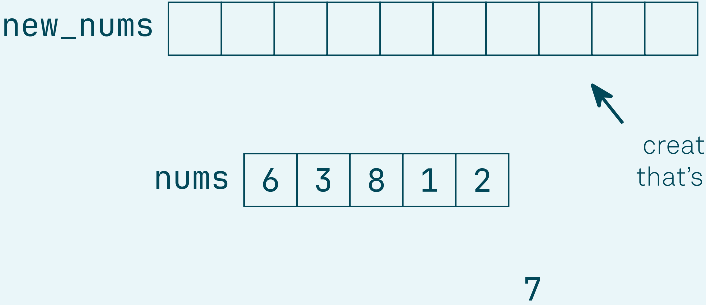


nums				
6	3	8	1	2

now what? →

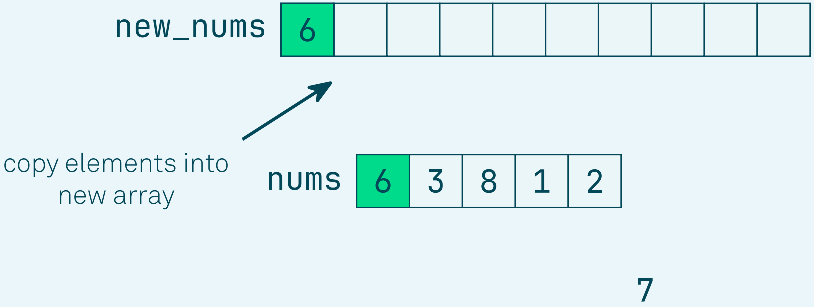
7

How Might We Implement a Dynamic Array?

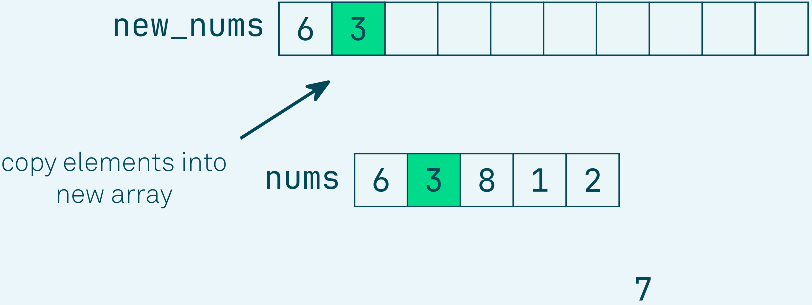


create a new array that's 1.5 to 2 times bigger

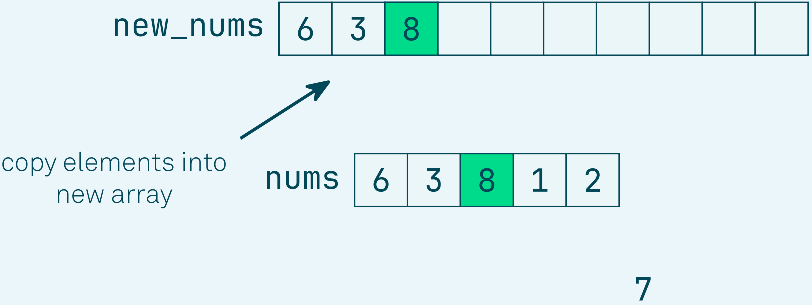
How Might We Implement a Dynamic Array?



How Might We Implement a Dynamic Array?



How Might We Implement a Dynamic Array?



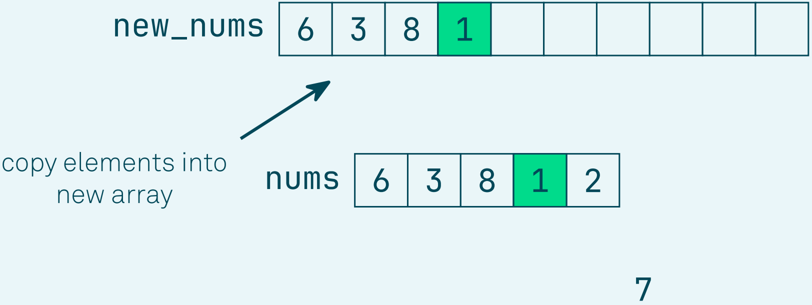
new_nums 6 3 8

copy elements into new array

nums 6 3 8 1 2

7

How Might We Implement a Dynamic Array?



new_nums

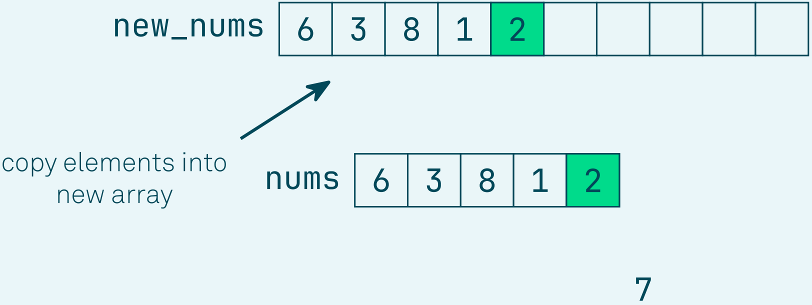
6	3	8	1						

copy elements into new array

nums

6	3	8	1	2

How Might We Implement a Dynamic Array?



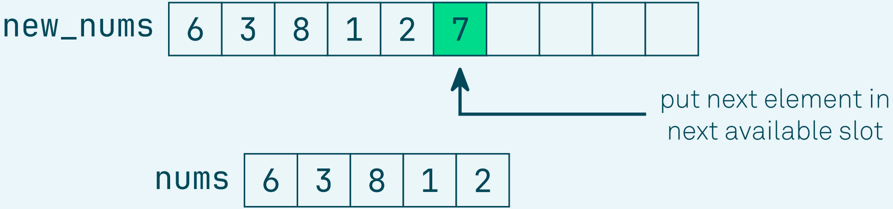
new_nums 6 3 8 1 2

copy elements into new array

nums 6 3 8 1 2

7

How Might We Implement a Dynamic Array?

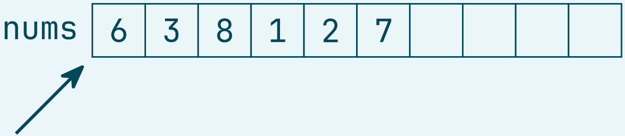


new_nums	6	3	8	1	2	7				

put next element in
next available slot

nums	6	3	8	1	2

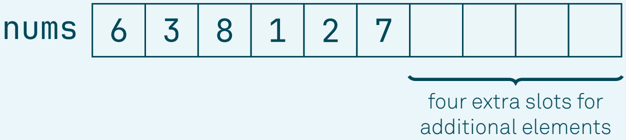
How Might We Implement a Dynamic Array?



nums	6	3	8	1	2	7				

replace old array
with new array

How Might We Implement a Dynamic Array?



nums

6	3	8	1	2	7				

four extra slots for additional elements

Dynamic-array Operations

Dynamic-array Operations

- What operations do dynamic arrays have?

Dynamic-array Operations

- What operations do dynamic arrays have?
  - Size (constant time)

Dynamic-array Operations

- What operations do dynamic arrays have?
    - Size (constant time)
    - Get (constant time)

Dynamic-array Operations

- What operations do dynamic arrays have?
  - Size (constant time)
  - Get (constant time)
  - Set (constant time)

Dynamic-array Operations

- What operations do dynamic arrays have?
    - Size (constant time)
    - Get (constant time)
    - Set (constant time)
    - Append (constant time)

Dynamic-array Operations

- What operations do dynamic arrays have?
    - Size (constant time)
    - Get (constant time)
    - Set (constant time)
    - Append (constant time)
    - Insert (linear time)

Dynamic-array Operations

- What operations do dynamic arrays have?
    - Size (constant time)
    - Get (constant time)
    - Set (constant time)
    - Append (constant time)
    - Insert (linear time)
    - Remove (linear time)

Dynamic-array Operations

- What operations do dynamic arrays have?
    - Size (constant time)
    - Get (constant time)
    - Set (constant time)
    - Append (constant time)
    - Insert (linear time)
    - Remove (linear time)
    - Pop (linear time)

Does `std::vector` Do Anything Differently?

Does `std::vector` Do Anything Differently?

- Vectors require *dynamic allocation* since the sizes of the backing arrays need to change if required

Does `std::vector` Do Anything Differently?

- Vectors require *dynamic allocation* since the sizes of the backing arrays need to change if required
- This means vectors allocate their data on the *heap*

Does `std::vector` Do Anything Differently?

- Vectors require *dynamic allocation* since the sizes of the backing arrays need to change if required
- This means vectors allocate their data on the *heap*
- Dynamic allocation is a bit expensive if it’s never needed

Does `std::vector` Do Anything Differently?

- Vectors require *dynamic allocation* since the sizes of the backing arrays need to change if required
- This means vectors allocate their data on the *heap*
- Dynamic allocation is a bit expensive if it’s never needed
- Vectors don’t allocate memory if they’re constructed empty

Does `std::vector` Do Anything Differently?

- Vectors require *dynamic allocation* since the sizes of the backing arrays need to change if required
- This means vectors allocate their data on the *heap*
- Dynamic allocation is a bit expensive if it’s never needed
- Vectors don’t allocate memory if they’re constructed empty
- First element added triggers allocation of an array with one slot. From there, array size increases by a factor of two

Dynamic Allocation

What Is the Stack?

What Is the Stack?

- The stack is working memory allocated for our program

What Is the Stack?

- The stack is working memory allocated for our program
- It's known as *automatic memory* since C++ handles it for us

What Is the Stack?

- The stack is working memory allocated for our program
- It’s known as *automatic memory* since C++ handles it for us
- When variables are declared, new slots are pushed onto the stack automatically

What Is the Stack?

- The stack is working memory allocated for our program
- It’s known as *automatic memory* since C++ handles it for us
- When variables are declared, new slots are pushed onto the stack automatically
- When variables go out of scope, the slots for those variables are popped off the stack automatically

What Is the Stack?

- The stack is working memory allocated for our program
- It’s known as *automatic memory* since C++ handles it for us
- When variables are declared, new slots are pushed onto the stack automatically
- When variables go out of scope, the slots for those variables are popped off the stack automatically
- When functions are called, stack memory is allocated to hold the arguments, local-variable data, and return-value data if function is non-void

What Is the Stack?

- The stack is working memory allocated for our program
- It’s known as *automatic memory* since C++ handles it for us
- When variables are declared, new slots are pushed onto the stack automatically
- When variables go out of scope, the slots for those variables are popped off the stack automatically
- When functions are called, stack memory is allocated to hold the arguments, local-variable data, and return-value data if function is non-void
- The memory allocated for a function called is known as a *stack frame*

Advantages of the Stack

Advantages of the Stack

- Quick allocations and deallocations

Advantages of the Stack

- Quick allocations and deallocations
- Automatically managed by C++

Advantages of the Stack

- Quick allocations and deallocations
- Automatically managed by C++
- It has a strict structure, so it’s easy to keep it organized

Any Disadvantages with the Stack?

Any Disadvantages with the Stack?

- Newer data must be popped off the stack before older data

Any Disadvantages with the Stack?

- Newer data must be popped off the stack before older data
- How do we keep data alive outside a function when it's created inside a function?

Any Disadvantages with the Stack?

- Newer data must be popped off the stack before older data
- How do we keep data alive outside a function when it’s created inside a function?
- The data is copied from the location referenced by the local variable to the location meant for the return value

Any Disadvantages with the Stack?

- Newer data must be popped off the stack before older data
- How do we keep data alive outside a function when it’s created inside a function?
- The data is copied from the location referenced by the local variable to the location meant for the return value
- What if the data is large? How expensive is this copy?

Any Disadvantages with the Stack?

- Newer data must be popped off the stack before older data
- How do we keep data alive outside a function when it’s created inside a function?
- The data is copied from the location referenced by the local variable to the location meant for the return value
- What if the data is large? How expensive is this copy?
- Are there ways to avoid these copies?

Return-value Optimization (RVO)

Return-value Optimization (RVO)

- When possible, C++ can avoid any copying when returning by using the location where the value would end up after the function ends instead of a temporary location within the stack frame

Return-value Optimization (RVO)

- When possible, C++ can avoid any copying when returning by using the location where the value would end up after the function ends instead of a temporary location within the stack frame
- RVO only applies when return type is non-primitive

Return-value Optimization (RVO)

- When possible, C++ can avoid any copying when returning by using the location where the value would end up after the function ends instead of a temporary location within the stack frame
- RVO only applies when return type is non-primitive
- Two kinds of RVO:

Return-value Optimization (RVO)

- When possible, C++ can avoid any copying when returning by using the location where the value would end up after the function ends instead of a temporary location within the stack frame
- RVO only applies when return type is non-primitive
- Two kinds of RVO:
  - Unnamed return-value optimization (URVO)

Return-value Optimization (RVO)

- When possible, C++ can avoid any copying when returning by using the location where the value would end up after the function ends instead of a temporary location within the stack frame
- RVO only applies when return type is non-primitive
- Two kinds of RVO:
  - Unnamed return-value optimization (URVO)
  - Named return-value optimization (NRVO)

URVO Example

```cpp
#include <array>
#include <cstdint>
#include <cstdlib>

std::array<std::int32_t, 5> get_nums() noexcept {
   return std::array{ 1, 2, 3, 4, 5 };
}

int main() {
   const auto nums{get_nums()};
   return EXIT_SUCCESS;
}
```

URVO Example

```cpp
#include <array>
#include <cstdint>
#include <cstdlib>

std::array<std::int32_t, 5> get_nums() noexcept {
    return std::array{ 1, 2, 3, 4, 5 };
}

int main() {
    const auto nums{get_nums()};
    return EXIT_SUCCESS;
}
```

returned value is a *temporary*

URVO Example

```cpp
#include <array>
#include <cstdint>
#include <cstdlib>

std::array<std::int32_t, 5> get_nums() noexcept {
    return std::array{ 1, 2, 3, 4, 5 };
}

int main() {
    const auto nums{get_nums()};
    return EXIT_SUCCESS;
}
```

returned value is a *temporary*

array will be created and stored directly in `nums`

Things to Know about URVO

Things to Know about URVO

- As of C++17, URVO is mandatory for all compilers to perform

Things to Know about URVO

- As of C++17, URVO is mandatory for all compilers to perform
- URVO only applies if the return value is a temporary

NRVO Example

```cpp
#include <array>
#include <cstdint>
#include <cstdlib>

#include <UTL/random.hpp>

std::array<std::int32_t, 5> get_random_nums() {
   namespace rand = utl::random;
   std::array<std::int32_t, 5> nums{};
   for (std::int32_t &num : nums) {
      num = rand::uniform<std::int32_t>(-10, 10);
   }
   return nums;
}

int main() {
   const auto nums{get_random_nums()};
   return EXIT_SUCCESS;
}
```

NRVO Example

return value is created before return statement

```cpp
#include <array>
#include <cstdint>
#include <cstdlib>

#include <UTL/random.hpp>

std::array<std::int32_t, 5> get_random_nums() {
   namespace rand = utl::random;
   std::array<std::int32_t, 5> nums{};
   for (std::int32_t &num : nums) {
      num = rand::uniform<std::int32_t>(-10, 10);
   }
   return nums;
}

int main() {
   const auto nums{get_random_nums()};
   return EXIT_SUCCESS;
}
```

NRVO Example

return value is created before return statement

return value can be created directly in result variable

```cpp
#include <array>
#include <cstdint>
#include <cstdlib>

#include <UTL/random.hpp>

std::array<std::int32_t, 5> get_random_nums() {
   namespace rand = utl::random;
   std::array<std::int32_t, 5> nums{};
   for (std::int32_t &num : nums) {
      num = rand::uniform<std::int32_t>(-10, 10);
   }
   return nums;
}

int main() {
   const auto nums{get_random_nums()};
   return EXIT_SUCCESS;
}
```

Things to Know about NRVO

Things to Know about NRVO

- NRVO is a type of *copy elision*

Things to Know about NRVO

- NRVO is a type of *copy elision*

Copy elision

The creation of an object from an object of the same type can be omitted, even if the creation or destruction would have side effects

Things to Know about NRVO

- NRVO is a type of *copy elision*

Copy elision

The creation of an object from an object of the same type can be omitted, even if the creation or destruction would have side effects

- Copy elision is one of two allowed optimizations that can change observable side effects

Things to Know about NRVO

- NRVO is a type of *copy elision*

Copy elision

The creation of an object from an object of the same type can be omitted, even if the creation or destruction would have side effects

- Copy elision is one of two allowed optimizations that can change observable side effects
- Copy elision is *optional* for compilers to perform

Things RVO Can't Fix

Things RVO Can't Fix

- The stack isn’t that big, so storing large amounts of data is difficult

What Is the Heap?

What Is the Heap?

- The heap is large region of memory available for on-demand (dynamic) allocations

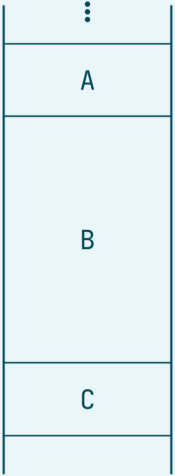
What Is the Heap?

- The heap is large region of memory available for on-demand (dynamic) allocations
- The structure of the stack makes it inconvenient for storing data which needs to persist for a while

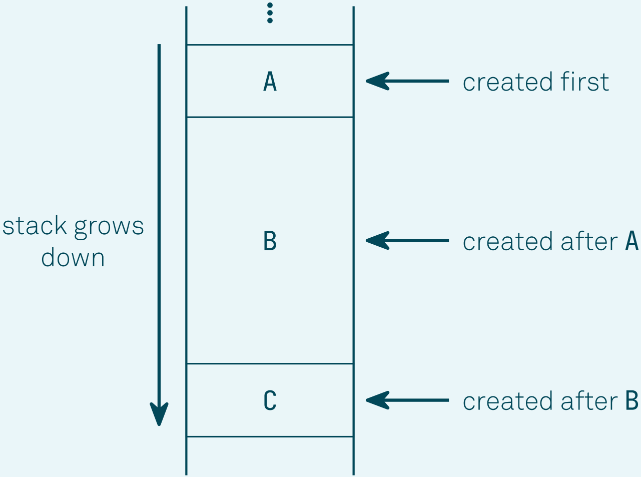
What Is the Heap?

- The heap is large region of memory available for on-demand (dynamic) allocations
- The structure of the stack makes it inconvenient for storing data which needs to persist for a while
- Newer memory must be freed before older memory can be freed

What Is the Heap?



What Is the Heap?



stack grows down

A

B

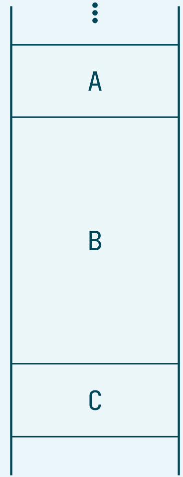
C

created first

created after A

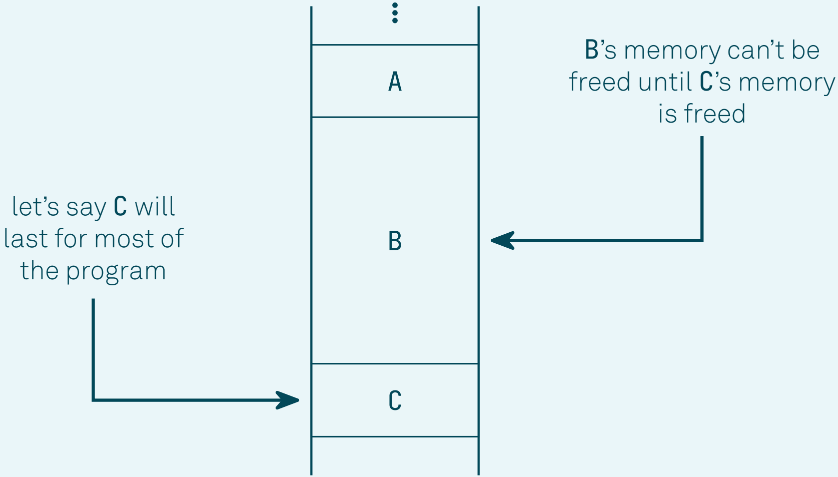
created after B

What Is the Heap?



let’s say C will last for most of the program

What Is the Heap?



B's memory can't be freed until C's memory is freed

let's say C will last for most of the program

What Is the Heap?

- The heap can be used to allocate large chunks of memory which need to last for a while

What Is the Heap?

- The heap can be used to allocate large chunks of memory which need to last for a while
- Each of these chunks can be freed at any time, regardless of when it was allocated

How Do Vectors Use Dynamic Allocation?

How Do Vectors Use Dynamic Allocation?

- Information like the size and the capacity is stored on the stack

How Do Vectors Use Dynamic Allocation?

- Information like the size and the capacity is stored on the stack
- Elements are stored on the heap

How Do Vectors Use Dynamic Allocation?

- Information like the size and the capacity is stored on the stack
- Elements are stored on the heap
- New elements can be added or removed without worrying about the stack

How Do Vectors Use Dynamic Allocation?

- Information like the size and the capacity is stored on the stack
- Elements are stored on the heap
- New elements can be added or removed without worrying about the stack
- Old backing arrays can have their memories freed as soon as they’re no longer needed

How Do Vectors Use Dynamic Allocation?

- Information like the size and the capacity is stored on the stack
- Elements are stored on the heap
- New elements can be added or removed without worrying about the stack
- Old backing arrays can have their memories freed as soon as they’re no longer needed
- Only the existing elements will need to get copied when a new backing array is created

Working with Vectors

Vector Syntax: Creation

Vector Syntax: Creation

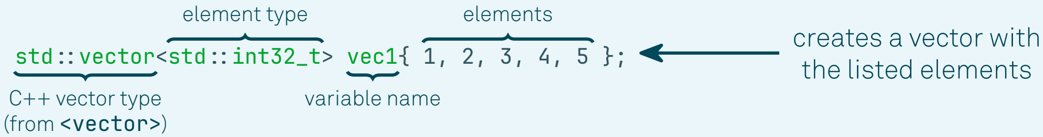
```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

Vector Syntax: Creation

```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

creates a vector with the listed elements

Vector Syntax: Creation



```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

element type

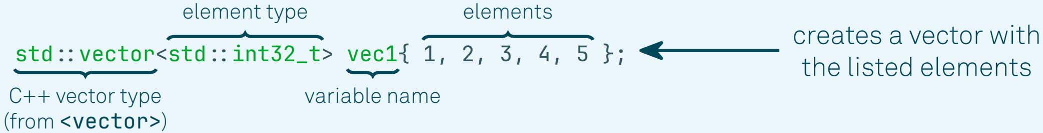
elements

creates a vector with the listed elements

C++ vector type (from `<vector>`)

variable name

Vector Syntax: Creation



element type

elements

```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

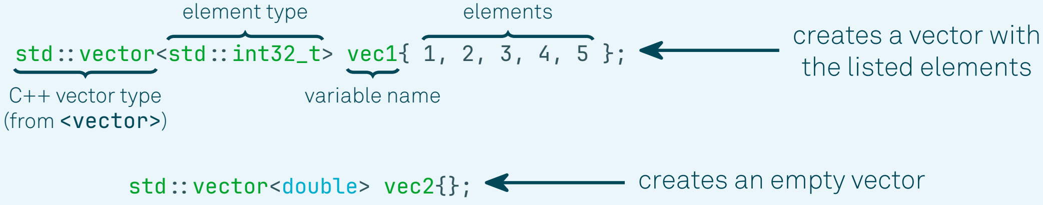
C++ vector type (from **`<vector>`**)

variable name

creates a vector with the listed elements

```
std::vector<double> vec2{};
```

Vector Syntax: Creation



element type

elements

```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

creates a vector with the listed elements

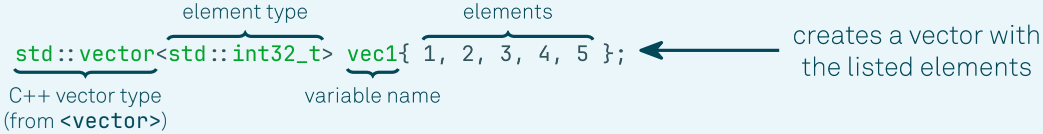
C++ vector type (from `<vector>`)

variable name

```
std::vector<double> vec2{};
```

creates an empty vector

Vector Syntax: Creation



```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

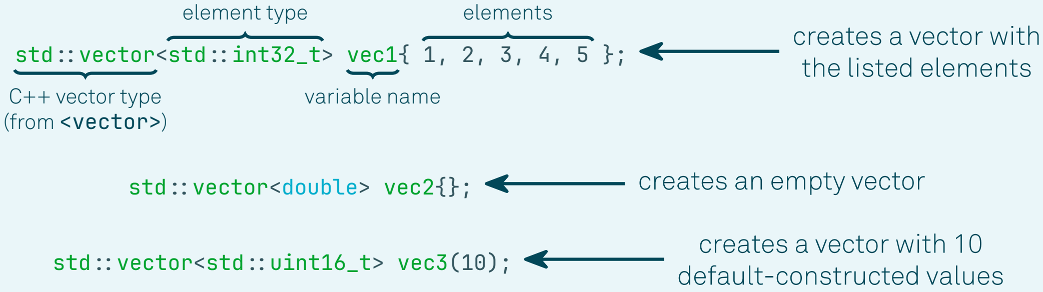
creates a vector with the listed elements

```
std::vector<double> vec2{};
```

creates an empty vector

```
std::vector<std::uint16_t> vec3(10);
```

Vector Syntax: Creation



```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

creates a vector with the listed elements

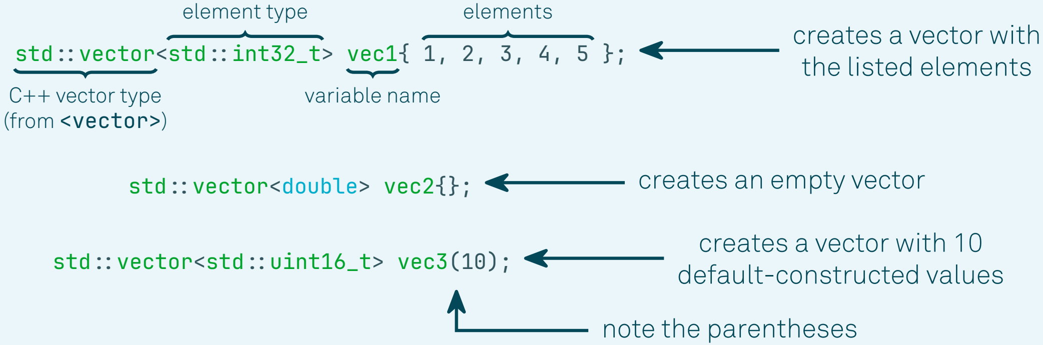
```
std::vector<double> vec2{};
```

creates an empty vector

```
std::vector<std::uint16_t> vec3(10);
```

creates a vector with 10 default-constructed values

Vector Syntax: Creation



element type

elements

```cpp
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

creates a vector with the listed elements

C++ vector type (from `<vector>`)

variable name

```cpp
std::vector<double> vec2{};
```

creates an empty vector

```cpp
std::vector<std::uint16_t> vec3(10);
```

creates a vector with 10 default-constructed values

note the parentheses

Vector Syntax: Creation

element type

elements

```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

C++ vector type (from `<vector>`)

variable name

creates a vector with the listed elements

```
std::vector<double> vec2{};
```

creates an empty vector

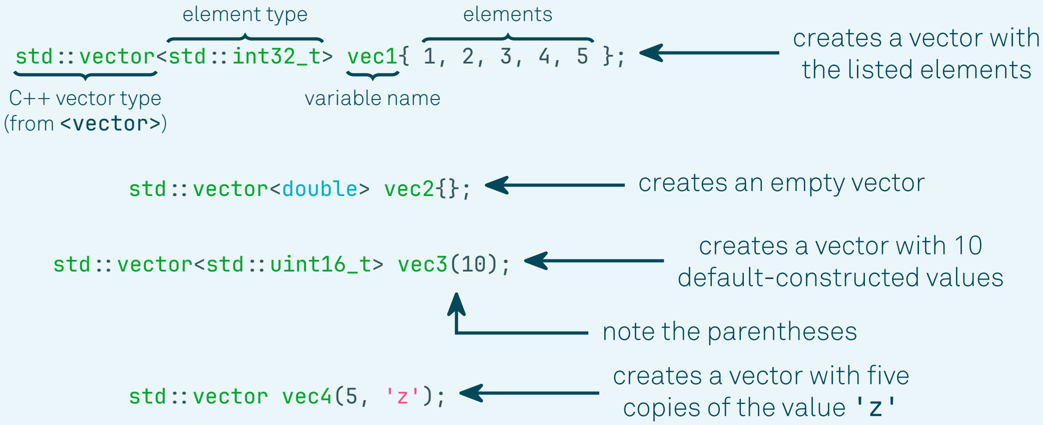
```
std::vector<std::uint16_t> vec3(10);
```

creates a vector with 10 default-constructed values

note the parentheses

```
std::vector vec4(5, 'z');
```

Vector Syntax: Creation



element type

elements

```
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

creates a vector with the listed elements

C++ vector type (from **<vector>**)

variable name

```
std::vector<double> vec2{};
```

creates an empty vector

```
std::vector<std::uint16_t> vec3(10);
```

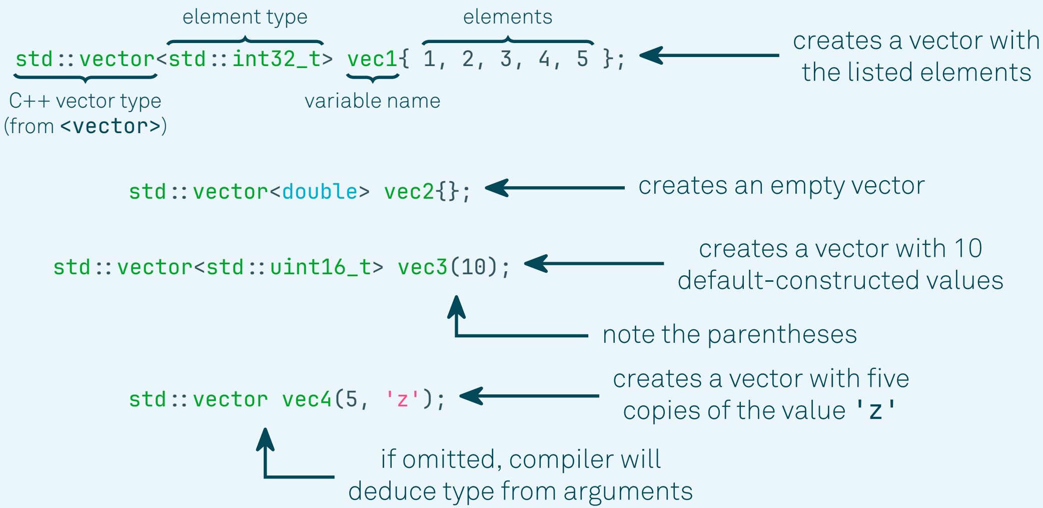
creates a vector with 10 default-constructed values

note the parentheses

```
std::vector vec4(5, 'z');
```

creates a vector with five copies of the value **'z'**

Vector Syntax: Creation



```cpp
std::vector<std::int32_t> vec1{ 1, 2, 3, 4, 5 };
```

element type

elements

C++ vector type (from `<vector>`)

variable name

creates a vector with the listed elements

```cpp
std::vector<double> vec2{};
```

creates an empty vector

```cpp
std::vector<std::uint16_t> vec3(10);
```

creates a vector with 10 default-constructed values

note the parentheses

```cpp
std::vector vec4(5, 'z');
```

creates a vector with five copies of the value `'z'`

if omitted, compiler will deduce type from arguments

Vector Syntax: Accessing

```cpp
std::vector<std::int32_t> vec{ 1, 2, 3, 4, 5 };
cout << vec.front() << '\n';
```

Vector Syntax: Accessing

```cpp
std::vector<std::int32_t> vec{ 1, 2, 3, 4, 5 };
cout << vec.front() << '\n';
```

```
1
```

Vector Syntax: Accessing

```
std::vector<std::int32_t> vec{ 1, 2, 3, 4, 5 };
cout << vec.front() << '\n';
```

```
1
```

```
cout << vec.back() << '\n';
```

Vector Syntax: Accessing

```cpp
std::vector<std::int32_t> vec{ 1, 2, 3, 4, 5 };
cout << vec.front() << '\n';
```

```
1
```

```cpp
cout << vec.back() << '\n';
```

```
5
```

Vector Syntax: Accessing

```
std::vector<std::int32_t> vec{ 1, 2, 3, 4, 5 };
cout << vec.front() << '\n';
```

```
1
```

```
cout << vec.back() << '\n';
```

```
5
```

```
cout << vec.at(2) << '\n';
```

Vector Syntax: Accessing

```
std::vector<std::int32_t> vec{ 1, 2, 3, 4, 5 };
cout << vec.front() << '\n';
```

```
1
```

```
cout << vec.back() << '\n';
```

```
5
```

```
cout << vec.at(2) << '\n';
```

```
3
```

Vector Syntax: Accessing

```cpp
cout << "size: " << vec.size() << '\n';
```

Vector Syntax: Accessing

```
cout << "size: " << vec.size() << '\n';
```

```
size: 5
```

Vector Syntax: Accessing

```
cout << "size: " << vec.size() << '\n';
```

```
size: 5
```

```
cout << "capacity: " << vec.capacity() << '\n';
```

Vector Syntax: Accessing

```
cout << "size: " << vec.size() << '\n';
```

```
size: 5
```

```
cout << "capacity: " << vec.capacity() << '\n';
```

```
capacity: 5
```

Vector Syntax: Accessing

```cpp
cout << "size: " << vec.size() << '\n';
```

```
size: 5
```

```cpp
cout << "capacity: " << vec.capacity() << '\n';
```

```
capacity: 5
```

```cpp
cout << vec.at(17) << '\n';
```

Vector Syntax: Accessing

```
cout << "size: " << vec.size() << '\n';
```

```
size: 5
```

```
cout << "capacity: " << vec.capacity() << '\n';
```

```
capacity: 5
```

```
cout << vec.at(17) << '\n';
```

```
libc++abi: terminating due to uncaught exception of type std::out_of_range: vector

Process finished with exit code 134 (interrupted by signal 6:SIGABRT)
```

Vector Syntax: Modifying

```cpp
std::vector words{ "hello", "world", "C++" };
words.front() = "goodbye";
cout << words.front() << '\n';
```

Vector Syntax: Modifying

```cpp
std::vector words{ "hello", "world", "C++" };
words.front() = "goodbye";
cout << words.front() << '\n';
```

```
goodbye
```

Vector Syntax: Modifying

```
std::vector words{ "hello", "world", "C++" };
words.front() = "goodbye";
cout << words.front() << '\n';
```

```
goodbye
```

```
words.at(2) = "Java";
cout << words.at(2) << '\n';
```

Vector Syntax: Modifying

```
std::vector words{ "hello", "world", "C++" };
words.front() = "goodbye";
cout << words.front() << '\n';
```

```
goodbye
```

```
words.at(2) = "Java";
cout << words.at(2) << '\n';
```

```
Java
```

Vector Syntax: Modifying

```
std::vector words{ "hello", "world", "C++" };
words.front() = "goodbye";
cout << words.front() << '\n';
```

```
goodbye
```

```
words.at(2) = "Java";
cout << words.at(2) << '\n';
```

```
Java
```

```
words.at(-1) = "word";
```

Vector Syntax: Modifying

```
std::vector words{ "hello", "world", "C++" };
words.front() = "goodbye";
cout << words.front() << '\n';
```

```
goodbye
```

```
words.at(2) = "Java";
cout << words.at(2) << '\n';
```

```
Java
```

```
words.at(-1) = "word";
```

```
libc++abi: terminating due to uncaught exception of type std::out_of_range: vector

Process finished with exit code 134 (interrupted by signal 6:SIGABRT)
```

Vector Syntax: Adding

```cpp
std::vector<std::uint16_t> nums{};
for (std::uint16_t i{1U}; i <= 5U; ++i) {
   nums.push_back(i);
}
// Assumes insertion operator has been overloaded for vectors.
cout << nums << '\n';
```

Vector Syntax: Adding

```cpp
std::vector<std::uint16_t> nums{};
for (std::uint16_t i{1U}; i <= 5U; ++i) {
   nums.push_back(i);
}
// Assumes insertion operator has been overloaded for vectors.
cout << nums << '\n';
```

```
[1, 2, 3, 4, 5]
```

Vector Syntax: Adding

```
std::vector<std::uint16_t> nums{};
for (std::uint16_t i{1U}; i <= 5U; ++i) {
   nums.push_back(i);
}
// Assumes insertion operator has been overloaded for vectors.
cout << nums << '\n';
```

```
[1, 2, 3, 4, 5]
```

```
auto middle_iter{std::next(nums.begin(), nums.size() >> 1)};
cout << "middle element: " << *middle_iter << '\n';
middle_iter = nums.insert(middle_iter, 0U);
cout << "new middle element: " << *middle_iter << '\n';
cout << nums << '\n';
```

Vector Syntax: Adding

```cpp
std::vector<std::uint16_t> nums{};
for (std::uint16_t i{1U}; i <= 5U; ++i) {
   nums.push_back(i);
}
// Assumes insertion operator has been overloaded for vectors.
cout << nums << '\n';
```

```
[1, 2, 3, 4, 5]
```

```cpp
auto middle_iter{std::next(nums.begin(), nums.size() >> 1)};
cout << "middle element: " << *middle_iter << '\n';
middle_iter = nums.insert(middle_iter, 0U);
cout << "new middle element: " << *middle_iter << '\n';
cout << nums << '\n';
```

```
middle element: 3
new middle element: 0
[1, 2, 0, 3, 4, 5]
```

Vector Syntax: Adding

```
nums.insert(nums.begin(), { 7U, 8U, 9U });
cout << nums << '\n';
```

Vector Syntax: Adding

```cpp
nums.insert(nums.begin(), { 7U, 8U, 9U });
cout << nums << '\n';
```

```
[7, 8, 9, 1, 2, 0, 3, 4, 5]
```

Vector Syntax: Adding

```cpp
nums.insert(nums.begin(), { 7U, 8U, 9U });
cout << nums << '\n';
```

```
[7, 8, 9, 1, 2, 0, 3, 4, 5]
```

```cpp
nums.insert(nums.end(), 2U, 6U);
cout << nums << endl;
```

Vector Syntax: Adding

```cpp
nums.insert(nums.begin(), { 7U, 8U, 9U });
cout << nums << '\n';
```

```
[7, 8, 9, 1, 2, 0, 3, 4, 5]
```

```cpp
nums.insert(nums.end(), 2U, 6U);
cout << nums << endl;
```

```
[7, 8, 9, 1, 2, 0, 3, 4, 5, 6, 6]

Process finished with exit code 0
```

Vector Syntax: Emplacing

Vector Syntax: Emplacing

- What if we wanted to add an object to a vector?

Vector Syntax: Emplacing

- What if we wanted to add an object to a vector?

```
std::vector<std::bitset<8>> bitsets{};
bitsets.push_back(std::bitset<8>{0b11001100});
cout << bitsets << '\n';
```

Vector Syntax: Emplacing

- What if we wanted to add an object to a vector?

```cpp
std::vector<std::bitset<8>> bitsets{};
bitsets.push_back(std::bitset<8>{0b11001100});
cout << bitsets << '\n';
```

```
[11001100]
```

Vector Syntax: Emplacing

- What if we wanted to add an object to a vector?

```
std::vector<std::bitset<8>> bitsets{};
bitsets.push_back(std::bitset<8>{0b11001100});
cout << bitsets << '\n';
```

```
[11001100]
```

- This works, but it’s annoying to write the type every time

Vector Syntax: Emplacing

- What if we wanted to add an object to a vector?

```
std::vector<std::bitset<8>> bitsets{};
bitsets.push_back(std::bitset<8>{0b11001100});
cout << bitsets << '\n';
```

```
[11001100]
```

- This works, but it’s annoying to write the type every time
- This solution is also not as efficient as it could be

Vector Syntax: Emplacing

- This is where *emplacing* comes in!

Vector Syntax: Emplacing

- This is where *emplacing* comes in!
- Emplacing lets us construct objects in place

Vector Syntax: Emplacing

- This is where *emplacing* comes in!
- Emplacing lets us construct objects in place

```
const auto &r_last_element{bitsets.emplace_back(0b10101010)};
cout << "last element: " << r_last_element << '\n';
cout << bitsets << '\n';
```

Vector Syntax: Emplacing

- This is where *emplacing* comes in!
- Emplacing lets us construct objects in place

```cpp
const auto &r_last_element{bitsets.emplace_back(0b10101010)};
cout << "last element: " << r_last_element << '\n';
cout << bitsets << '\n';
```

```
last element: 10101010
[11001100, 10101010]
```

Vector Syntax: Emplacing

- This is where *emplacing* comes in!
- Emplacing lets us construct objects in place

```cpp
const auto &r_last_element{bitsets.emplace_back(0b10101010)};
cout << "last element: " << r_last_element << '\n';
cout << bitsets << '\n';
```

```
last element: 10101010
[11001100, 10101010]
```

```cpp
const auto start_iter{bitsets.emplace(bitsets.begin(), 0b11110000)};
cout << "first element: " << *start_iter << '\n';
cout << bitsets << '\n';
```

Vector Syntax: Emplacing

- This is where *emplacing* comes in!
- Emplacing lets us construct objects in place

```
const auto &r_last_element{bitsets.emplace_back(0b10101010)};
cout << "last element: " << r_last_element << '\n';
cout << bitsets << '\n';
```

```
last element: 10101010
[11001100, 10101010]
```

```
const auto start_iter{bitsets.emplace(bitsets.begin(), 0b11110000)};
cout << "first element: " << *start_iter << '\n';
cout << bitsets << '\n';
```

```
first element: 11110000
[11110000, 11001100, 10101010]
```

Vector Syntax: Removing

```cpp
std::vector letters{ 'a', 'b', 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
cout << letters << '\n';
letters.pop_back();
cout << letters << '\n';
```

Vector Syntax: Removing

```
std::vector letters{ 'a', 'b', 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
cout << letters << '\n';
letters.pop_back();
cout << letters << '\n';
```

```
[a, b, a, b, c, d, e, f, g]
[a, b, a, b, c, d, e, f]
```

Vector Syntax: Removing

```cpp
std::vector letters{ 'a', 'b', 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
cout << letters << '\n';
letters.pop_back();
cout << letters << '\n';
```

```
[a, b, a, b, c, d, e, f, g]
[a, b, a, b, c, d, e, f]
```

```cpp
auto middle_iter{std::next(letters.begin(), letters.size() >> 1)};
cout << *middle_iter << '\n';
middle_iter = letters.erase(middle_iter);
cout << *middle_iter << '\n';
cout << letters << '\n';
```

Vector Syntax: Removing

```
std::vector letters{ 'a', 'b', 'a', 'b', 'c', 'd', 'e', 'f', 'g' };
cout << letters << '\n';
letters.pop_back();
cout << letters << '\n';
```

```
[a, b, a, b, c, d, e, f, g]
[a, b, a, b, c, d, e, f]
```

```
auto middle_iter{std::next(letters.begin(), letters.size() >> 1)};
cout << *middle_iter << '\n';
middle_iter = letters.erase(middle_iter);
cout << *middle_iter << '\n';
cout << letters << '\n';
```

```
c
d
[a, b, a, b, d, e, f]
```

Vector Syntax: Removing

```cpp
const auto start_iter{std::prev(letters.end(), 3)};
auto end_iter{std::prev(letters.end())};
cout << *start_iter << ' ' << *end_iter << '\n';
end_iter = letters.erase(start_iter, end_iter);
cout << *end_iter << '\n'; // `start_iter` is no longer valid
cout << letters << '\n';
```

Vector Syntax: Removing

```cpp
const auto start_iter{std::prev(letters.end(), 3)};
auto end_iter{std::prev(letters.end())};
cout << *start_iter << ' ' << *end_iter << '\n';
end_iter = letters.erase(start_iter, end_iter);
cout << *end_iter << '\n'; // `start_iter` is no longer valid
cout << letters << '\n';
```

```
d f
f
[a, b, a, b, f]
```

Vector Syntax: Removing

```cpp
const auto start_iter{std::prev(letters.end(), 3)};
auto end_iter{std::prev(letters.end())};
cout << *start_iter << ' ' << *end_iter << '\n';
end_iter = letters.erase(start_iter, end_iter);
cout << *end_iter << '\n'; // `start_iter` is no longer valid
cout << letters << '\n';
```

```
d f
f
[a, b, a, b, f]
```

```cpp
std::size_t removed_count{std::erase(letters, 'a')};
cout << "removed " << removed_count << '\n';
cout << letters << '\n';
```

Vector Syntax: Removing

```cpp
const auto start_iter{std::prev(letters.end(), 3)};
auto end_iter{std::prev(letters.end())};
cout << *start_iter << ' ' << *end_iter << '\n';
end_iter = letters.erase(start_iter, end_iter);
cout << *end_iter << '\n'; // `start_iter` is no longer valid
cout << letters << '\n';
```

```
d f
f
[a, b, a, b, f]
```

```cpp
std::size_t removed_count{std::erase(letters, 'a')};
cout << "removed " << removed_count << '\n';
cout << letters << '\n';
```

```
removed 2
[b, b, f]
```

Vector Syntax: Removing

```cpp
std::vector<std::int16_t> nums{ 1, 2, 3, 4, 5, 6, 7, 8 };
cout << nums << '\n';
removed_count = std::erase_if(nums, [](const std::int16_t num) noexcept {
   return 3 <= num && num <= 6;
});
cout << "removed " << removed_count << '\n';
cout << nums << endl;
```

Vector Syntax: Removing

```cpp
std::vector<std::int16_t> nums{ 1, 2, 3, 4, 5, 6, 7, 8 };
cout << nums << '\n';
removed_count = std::erase_if(nums, [](const std::int16_t num) noexcept {
    return 3 <= num && num <= 6;
});
cout << "removed " << removed_count << '\n';
cout << nums << endl;
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
removed 4
[1, 2, 7, 8]

Process finished with exit code 0
```

Vector Syntax: Misc.

```cpp
std::vector nums{ 1.2, 3.4, 4.7 };
cout << nums << " : size: " << nums.size() << ", cap: ";
cout << nums.capacity() << ", empty: " << nums.empty() << '\n';
nums.clear();
cout << nums << " : size: " << nums.size() << ", cap: ";
cout << nums.capacity() << ", empty: " << nums.empty() << endl;
```

Vector Syntax: Misc.

```cpp
std::vector nums{ 1.2, 3.4, 4.7 };
cout << nums << " : size: " << nums.size() << ", cap: ";
cout << nums.capacity() << ", empty: " << nums.empty() << '\n';
nums.clear();
cout << nums << " : size: " << nums.size() << ", cap: ";
cout << nums.capacity() << ", empty: " << nums.empty() << endl;
```

```
[1.2, 3.4, 4.7] : size: 3, cap: 3, empty: false
[] : size: 0, cap: 3, empty: true

Process finished with exit code 0
```

Vector Syntax: Resizing

```cpp
std::vector<std::int32_t> nums{};
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
nums.reserve(10U); // can only increase capacity
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
for (std::int32_t i{0}; i < 10; ++i) { nums.push_back(i); }
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
```

Vector Syntax: Resizing

```cpp
std::vector<std::int32_t> nums{};
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
nums.reserve(10U); // can only increase capacity
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
for (std::int32_t i{0}; i < 10; ++i) { nums.push_back(i); }
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
```

```
[] : cap: 0, size: 0
[] : cap: 10, size: 0
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] : cap: 10, size: 10
```

Vector Syntax: Resizing

```cpp
nums.push_back(10);
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
nums.shrink_to_fit(); // may or may not shrink capacity to size
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << endl;
```

Vector Syntax: Resizing

```
nums.push_back(10);
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
nums.shrink_to_fit(); // may or may not shrink capacity to size
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << endl;
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : cap: 20, size: 11
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : cap: 11, size: 11

Process finished with exit code 0
```

Vector Syntax: Resizing

```cpp
nums.push_back(10);
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << '\n';
nums.shrink_to_fit(); // may or may not shrink capacity to size
cout << nums << " : cap: " << nums.capacity() << ", size: " << nums.size() << endl;
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : cap: 20, size: 11
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] : cap: 11, size: 11

Process finished with exit code 0
```

- Don’t overuse these methods!

Vector Syntax: Resizing

```cpp
std::vector nums{ 1.1, 2.2, 3.3, 4.4, 5.5 };
cout << nums << '\n';
nums.resize(10U);
cout << nums << '\n';
nums.resize(3U);
cout << nums << '\n';
```

Vector Syntax: Resizing

```cpp
std::vector nums{ 1.1, 2.2, 3.3, 4.4, 5.5 };
cout << nums << '\n';
nums.resize(10U);
cout << nums << '\n';
nums.resize(3U);
cout << nums << '\n';
```

```
[1.1, 2.2, 3.3, 4.4, 5.5]
[1.1, 2.2, 3.3, 4.4, 5.5, 0, 0, 0, 0, 0]
[1.1, 2.2, 3.3]
```

Vector Syntax: Resizing

```
nums.resize(6U, 3.14);
cout << nums << '\n';
nums.resize(4U, -2.71);
cout << nums << endl;
```

Vector Syntax: Resizing

```cpp
nums.resize(6U, 3.14);
cout << nums << '\n';
nums.resize(4U, -2.71);
cout << nums << endl;
```

```
[1.1, 2.2, 3.3, 3.14, 3.14, 3.14]
[1.1, 2.2, 3.3, 3.14]

Process finished with exit code 0
```

Vector Syntax: Iteration

```
const std::vector<std::int16_t> nums{ 1, -2, 23, 4, 13 };
```

```
cout << "nums: ";
for (std::size_t i{0U}; i < nums.size(); ++i) {
   cout << nums.at(i) << ' ';
}
cout << '\n';
```

Output:

```
nums: 1 -2 23 4 13
```

Vector Syntax: Iteration

```
const std::vector<std::int16_t> nums{ 1, -2, 23, 4, 13 };
```

```
cout << "nums: ";
for (std::size_t i{0U}; i < nums.size(); ++i) {
   cout << nums.at(i) << ' ';
}
cout << '\n';
```

Output:

```
nums: 1 -2 23 4 13
```

```
cout << "nums: ";
for (std::size_t i{nums.size()}; i > 0U; --i) {
   cout << nums.at(i - 1U) << ' ';
}
cout << '\n';
```

Output:

```
nums: 13 4 23 -2 1
```

Vector Syntax: Iteration

```cpp
cout << "nums: ";
for (auto iter{nums.cbegin()}; iter != nums.cend(); ++iter) {
    cout << *iter << ' ';
}
cout << '\n';
```

Output:

```
nums: 1 -2 23 4 13
```

Vector Syntax: Iteration

```cpp
cout << "nums: ";
for (auto iter{nums.cbegin()}; iter != nums.cend(); ++iter) {
   cout << *iter << ' ';
}
cout << '\n';
```

Output:

```
nums: 1 -2 23 4 13
```

```cpp
cout << "nums: ";
for (auto iter{nums.crbegin()}; iter != nums.crend(); ++iter) {
   cout << *iter << ' ';
}
cout << '\n';
```

Output:

```
nums: 13 4 23 -2 1
```

Vector Syntax: Iteration

```cpp
cout << "nums: ";
for (const std::int16_t num : nums) { cout << num << ' '; }
cout << '\n';
```

Output:

```
nums: 1 -2 23 4 13
```

Undefined Behavior

```
std::vector<std::int16_t> nums{ 1, -2, 23, 4, 13 };
cout << nums[2] << '\n'; // UB if index out of range
cout << nums.front() << '\n'; // UB if `nums` is empty
cout << nums.back() << '\n'; // UB if `nums` is empty
nums.pop_back(); // UB if `nums` is empty
```

Undefined Behavior

```cpp
std::vector<std::int16_t> nums{ 1, -2, 23, 4, 13 };
cout << nums[2] << '\n'; // UB if index out of range
cout << nums.front() << '\n'; // UB if `nums` is empty
cout << nums.back() << '\n'; // UB if `nums` is empty
nums.pop_back(); // UB if `nums` is empty
```

- Additionally, we’re met with UB if certain requirements aren’t met for type T

Iterator Invalidation

Iterator Invalidation

Iterator invalidation is when an existing iterator is rendered unusable by an operation performed on the underlying object

Iterator Invalidation

Iterator invalidation is when an existing iterator is rendered unusable by an operation performed on the underlying object

- The data they point to is no longer in the underlying object

Iterator Invalidation

Iterator invalidation is when an existing iterator is rendered unusable by an operation performed on the underlying object

- The data they point to is no longer in the underlying object
- The memory an iterator points to has been deallocated or repurposed

Iterator Invalidation

- Any time a vector resizes, all iterators are invalidated

Iterator Invalidation

- Any time a vector resizes, all iterators are invalidated
- Any time an element is removed, iterators pointing to that element or any element after it in the sequence are invalidated

Iterator Invalidation

- Any time a vector resizes, all iterators are invalidated
- Any time an element is removed, iterators pointing to that element or any element after it in the sequence are invalidated
- These both include the end iterator

Iterator Invalidation

```cpp
std::vector<std::int32_t> nums{ 1, 2, 3 };
const auto end_iter{nums.end()};
const auto middle_iter{std::next(nums.begin(), nums.size() >> 1)};
nums.erase(middle_iter); // middle_iter and all after invalidated
cout << *middle_iter << '\n'; // 🤖
for (auto iter{nums.begin()}; iter != end_iter; ++iter) { // 🤖
   cout << *iter << ' ';
}
cout << '\n';
```

Iterator Invalidation

```cpp
std::vector<std::int32_t> nums{ 1, 2, 3 };
const auto end_iter{nums.end()};
const auto middle_iter{std::next(nums.begin(), nums.size() >> 1)};
nums.erase(middle_iter); // middle_iter and all after invalidated
cout << *middle_iter << '\n'; // 🤖
for (auto iter{nums.begin()}; iter != end_iter; ++iter) { // 🤖
   cout << *iter << ' ';
}
cout << '\n';
```

```
3
1 3 3
```

Iterator Invalidation

```cpp
std::vector words{ "hello"s, "world"s };
const auto begin_iter{words.begin()};
words.push_back("word"s); // all iterators invalidated due to resize
cout << *begin_iter << endl; // 🤖
```

Iterator Invalidation

```cpp
std::vector words{ "hello"s, "world"s };
const auto begin_iter{words.begin()};
words.push_back("word"s); // all iterators invalidated due to resize
cout << *begin_iter << endl; // 🤖
```

```
���t�

Process finished with exit code 0
```