



Arrays

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

array	1	2	3	4	5
	0	1	2	3	4

Data Structures – C++ Fundamentals

Agenda

What Are Arrays?

Properties

Benefits

Drawbacks

Arrays and Memory

Cache Locality

Memory Levels

Memory Lookup

Working with Arrays

Operations

Indexing

Syntax

Special Considerations

Printing

Functions Accepting Arrays

Undefined Behavior

Multi-dimensional Arrays

What Are Arrays?

Array Properties

What is an array?

An *array* is a fixed-length, contiguous, random-access sequence of homogeneous values

Array Properties

What is an array?

An *array* is a **fixed-length**, contiguous, random-access sequence of homogeneous values

Array Properties

What is an array?

An *array* is a **fixed-length**, contiguous, random-access sequence of homogeneous values

- The size of an array can't be changed after creation

Array Properties

What is an array?

An *array* is a **fixed-length**, contiguous, random-access sequence of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*

Array Properties

What is an array?

An *array* is a fixed-length, *contiguous*, random-access sequence of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*

Array Properties

What is an array?

An *array* is a fixed-length, *contiguous*, random-access sequence of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*
- Values are stored in adjacent memory slots

Array Properties

What is an array?

An *array* is a fixed-length, contiguous, **random-access** sequence of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*
- Values are stored in adjacent memory slots

Array Properties

What is an array?

An *array* is a fixed-length, contiguous, *random-access* sequence of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*
- Values are stored in adjacent memory slots
- Any value can be accessed at random

Array Properties

What is an array?

An *array* is a fixed-length, contiguous, random-access **sequence** of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*
- Values are stored in adjacent memory slots
- Any value can be accessed at random

Array Properties

What is an array?

An *array* is a fixed-length, contiguous, random-access **sequence** of homogeneous values

- The size of an array can't be changed after creation
- The size of an array must be known at *compile-time*
- Values are stored in adjacent memory slots
- Any value can be accessed at random
- Values are ordered (indexed for arrays)

Benefits of Arrays

Benefits of Arrays

- Fixed-size benefits

Benefits of Arrays

- Fixed-size benefits
 - Optimized

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality

1	2	3	4	5	

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality

1	2	3	4	5	

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality
 - Fast element lookup

1	2	3	4	5	

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality
 - Fast element lookup
- Sequence benefits

1	2	3	4	5	

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality
 - Fast element lookup
- Sequence benefits
 - Elements can be traversed in particular orders

1	2	3	4	5	

Benefits of Arrays

- Fixed-size benefits
 - Optimized
 - Stored on the stack
- Contiguous benefits
 - Great cache locality
 - Fast element lookup
- Sequence benefits
 - Elements can be traversed in particular orders
 - Can be subsequenced

1	2	3	4	5	

Drawbacks of Arrays

Drawbacks of Arrays

- Size must be known at compile-time

Drawbacks of Arrays

- Size must be known at compile-time
 - Can't create arrays from sizes computed at runtime

Drawbacks of Arrays

- Size must be known at compile-time
 - Can't create arrays from sizes computed at runtime
- Size is fixed

Drawbacks of Arrays

- Size must be known at compile-time
 - Can't create arrays from sizes computed at runtime
- Size is fixed
 - Must know exactly how many slots are needed, or must guess safely over the estimated number of slots (e.g., choosing a size of 50 if an actual size of around 20 is expected)

Arrays and Memory

Cache Locality

Cache Locality

What does cache locality refer to?

The tendency of an application to access the same set of memory locations repeatedly over a short period of time

Cache Locality

What does cache locality refer to?

The tendency of an application to access the same set of memory locations repeatedly over a short period of time

- Temporal locality

Cache Locality

What does cache locality refer to?

The tendency of an application to access the same set of memory locations repeatedly over a short period of time

- Temporal locality: recently accessed data is likely to be accessed again

Cache Locality

What does cache locality refer to?

The tendency of an application to access the same set of memory locations repeatedly over a short period of time

- Temporal locality: recently accessed data is likely to be accessed again
- Spatial locality

Cache Locality

What does cache locality refer to?

The tendency of an application to access the same set of memory locations repeatedly over a short period of time

- Temporal locality: recently accessed data is likely to be accessed again
- Spatial locality: data near recently accessed data is likely to be accessed next

How Does Computer Memory Work?

How Does Computer Memory Work?

- Where is computer data permanently stored?

How Does Computer Memory Work?

- Where is computer data permanently stored? Persistent storage!

How Does Computer Memory Work?

- Where is computer data permanently stored? Persistent storage!
 - Hard Disk Drive (HDD)



How Does Computer Memory Work?

- Where is computer data permanently stored? Persistent storage!
 - Hard Disk Drive (HDD)
 - Solid-state Drive (SSD)



How Does Computer Memory Work?

- Where is computer data permanently stored? Persistent storage!
 - Hard Disk Drive (HDD)
 - Solid-state Drive (SSD)
 - Non-volatile Memory Express (NVMe)



How Does Computer Memory Work?

- Where is computer data permanently stored? Persistent storage!
 - Hard Disk Drive (HDD)
 - Solid-state Drive (SSD)
 - Non-volatile Memory Express (NVMe)
- Retained after shutdown, but slow to access



How Does Computer Memory Work?

- What do CPUs use?

How Does Computer Memory Work?

- What do CPUs use? Volatile memory!

How Does Computer Memory Work?

- What do CPUs use? Volatile memory!



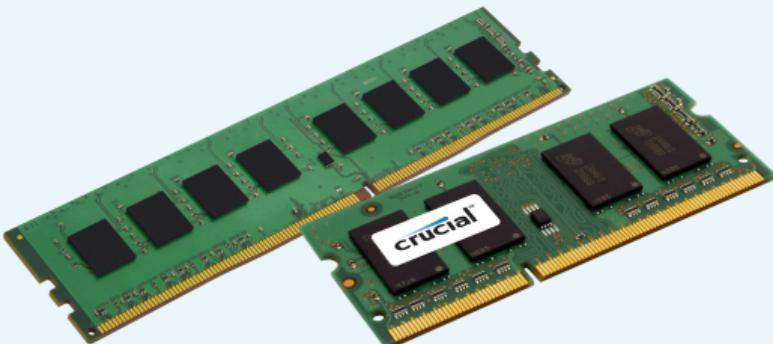
Static Random-access Memory
(SRAM)

How Does Computer Memory Work?

- What do CPUs use? Volatile memory!

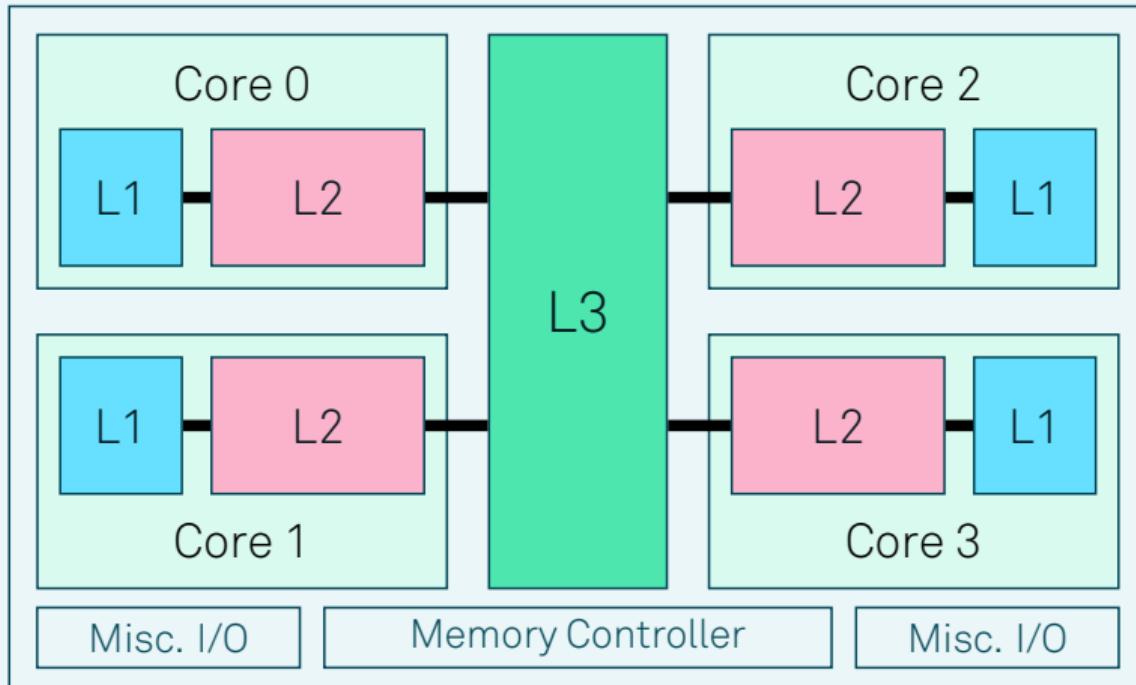


Static Random-access Memory
(SRAM)



Dynamic Random-access Memory
(DRAM)

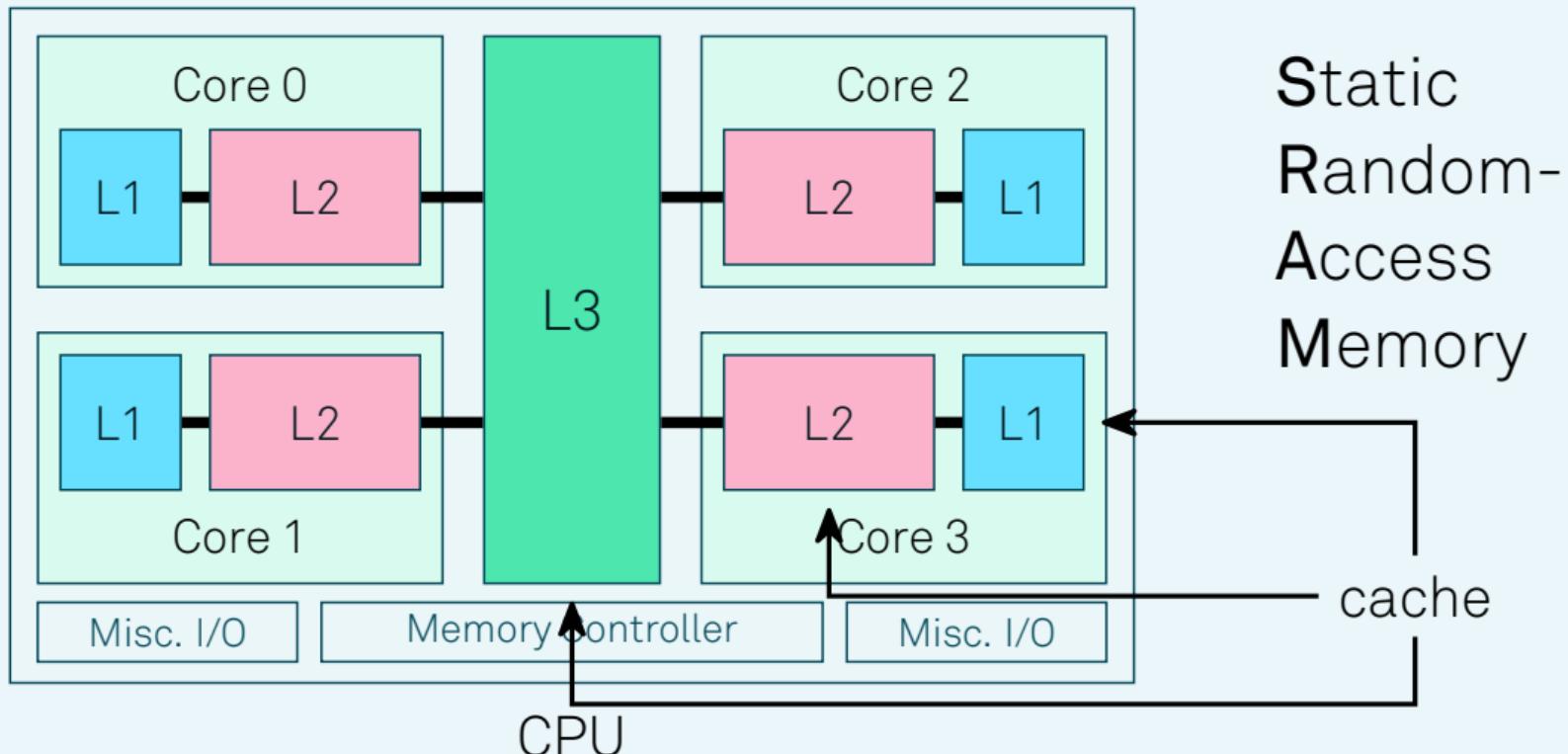
How Does Computer Memory Work?



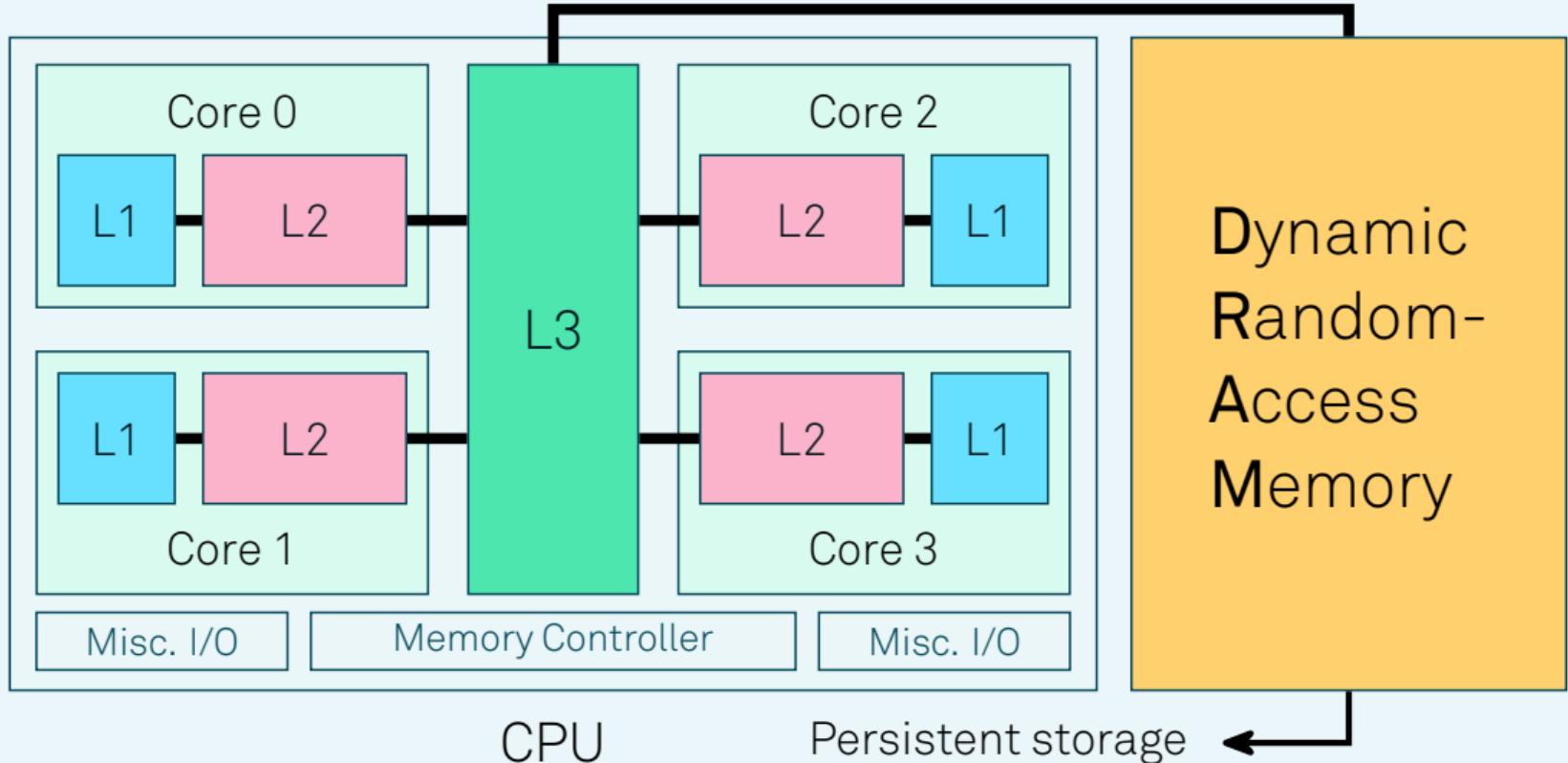
Static
Random-
Access
Memory

CPU

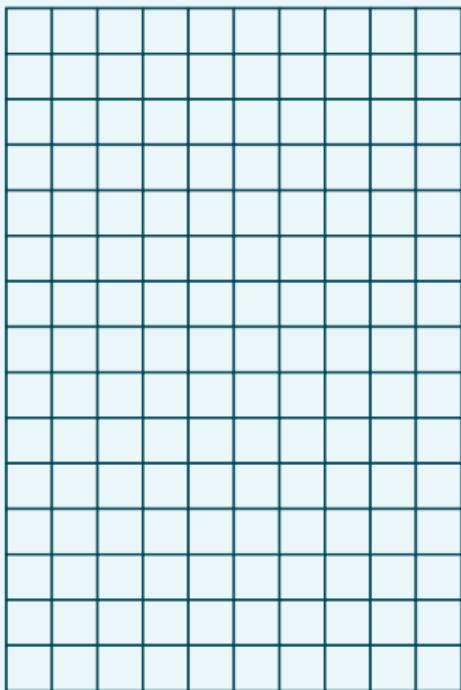
How Does Computer Memory Work?



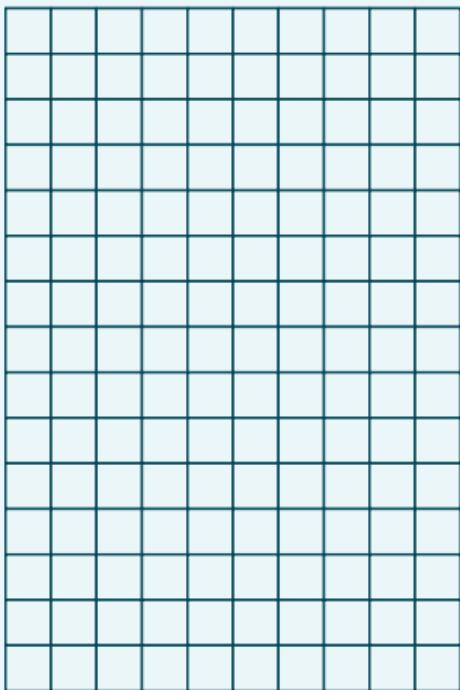
How Does Computer Memory Work?



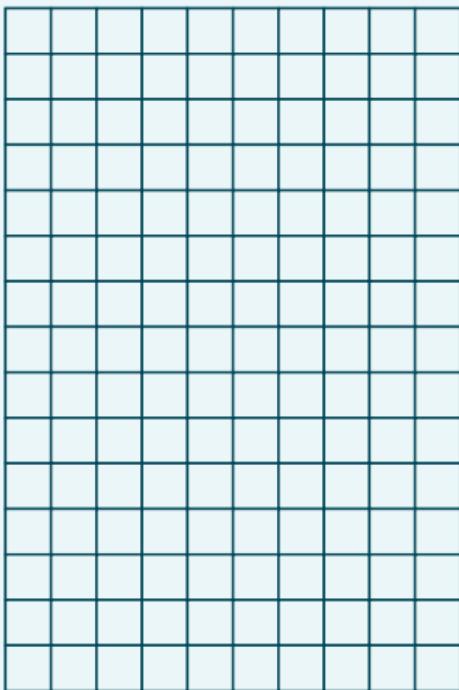
How Do Computers Look Up Data?



Page 0

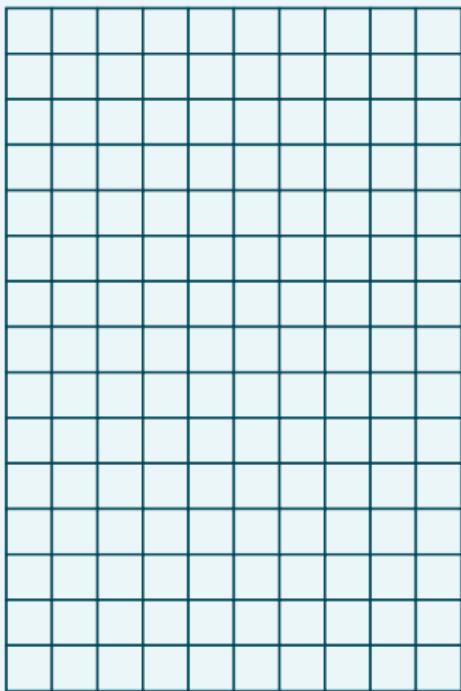


Page 1

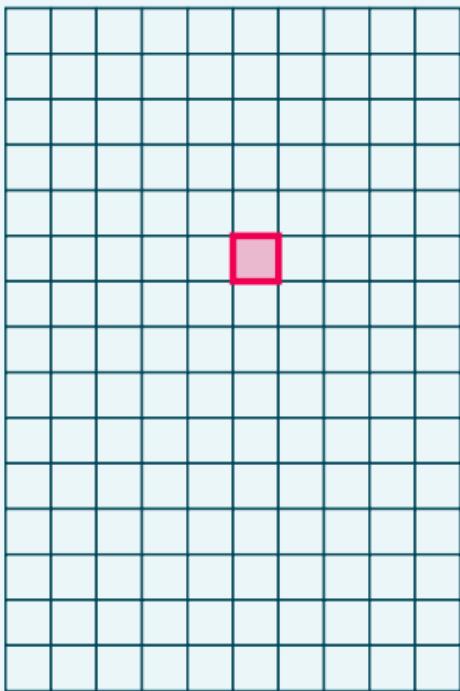


Page 2

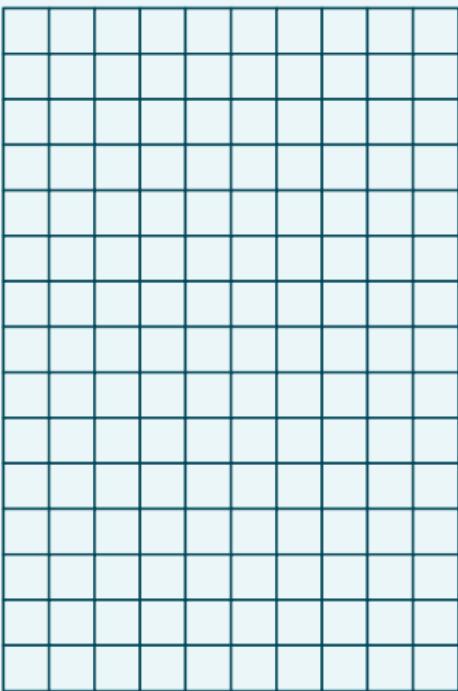
How Do Computers Look Up Data?



Page 0

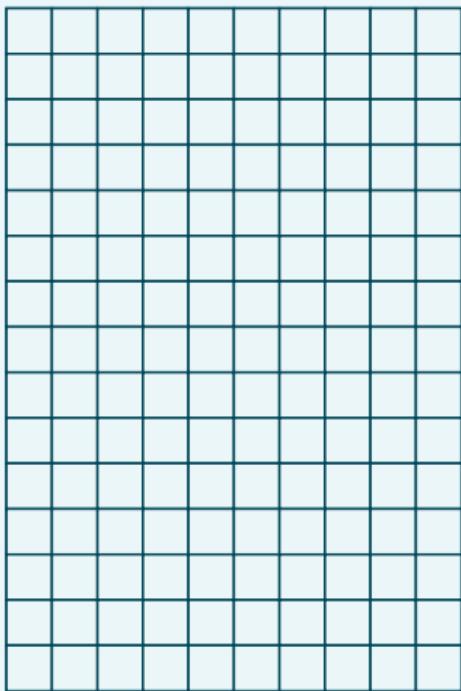


Page 1

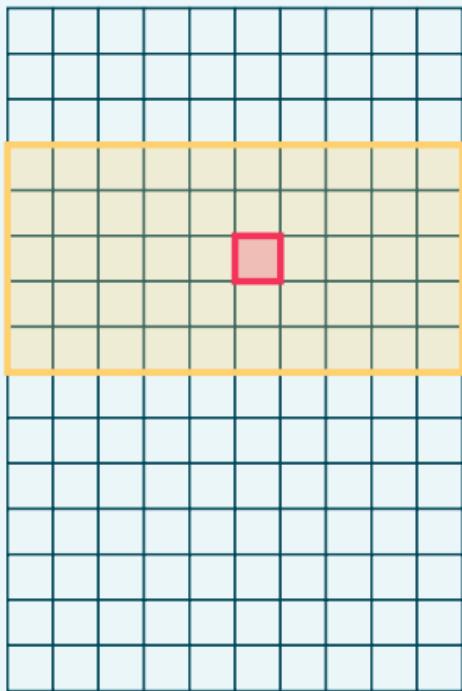


Page 2

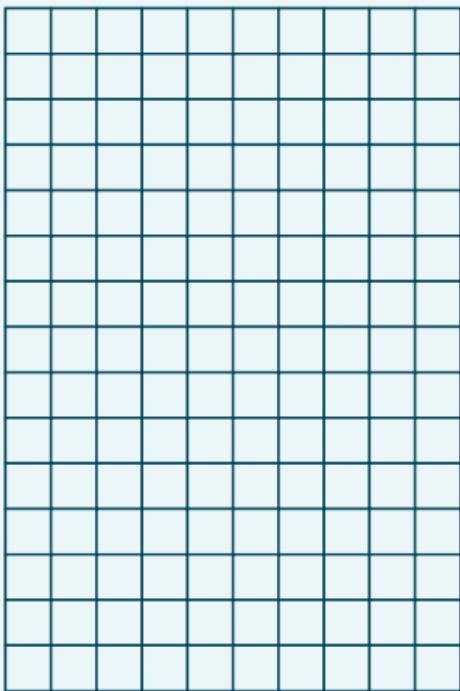
How Do Computers Look Up Data?



Page 0



Page 1



Page 2

Why do arrays have good
cache locality?

Working with Arrays

Array Operations

- What operations do arrays have?

Array Operations

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

Array Operations

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

Array Operations

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

Accessing Elements

- Array elements are *indexed*

Accessing Elements

- Array elements are *indexed*

What is an index?

Accessing Elements

- Array elements are *indexed*

What is an index?

An index is a way of enumerating the elements of a sequence so they can be accessed by number

Accessing Elements

- Array elements are *indexed*

What is an index?

An index is a way of enumerating the elements of a sequence so they can be accessed by number

- Indices start from zero and count up

Accessing Elements

- Array elements are *indexed*

What is an index?

An index is a way of enumerating the elements of a sequence so they can be accessed by number

- Indices start from zero and count up
- The last index value is the size minus one

Accessing Elements

- Array elements are *indexed*

What is an index?

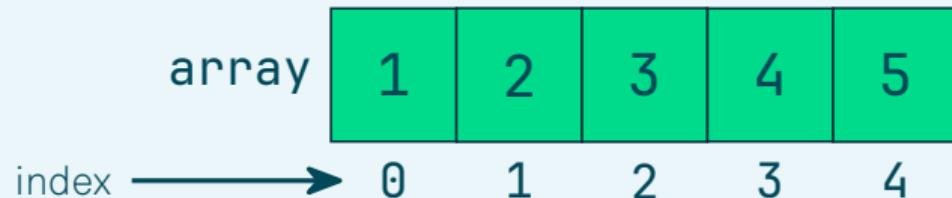
An index is a way of enumerating the elements of a sequence so they can be accessed by number

- Indices start from zero and count up
- The last index value is the size minus one
- Some languages allow negative index values (e.g., Python), but C++ is not one of them

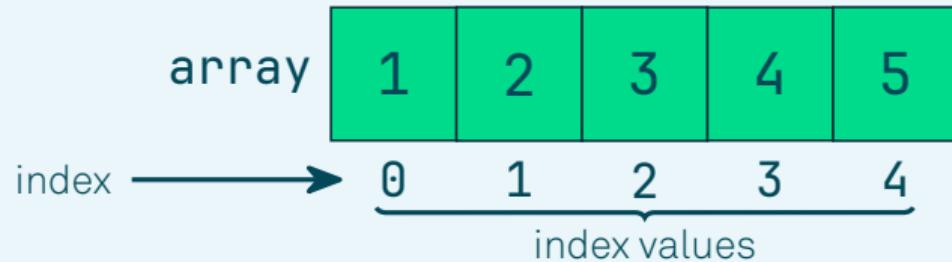
Accessing Elements

array	1	2	3	4	5
	0	1	2	3	4

Accessing Elements



Accessing Elements



Array Syntax: Creation

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

Array Syntax: Creation

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

Diagram illustrating the components of the C++ array syntax:

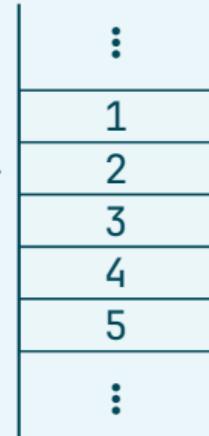
- C++ array type (from `<array>`)**: `std::array`
- element type**: `std::int32_t`
- variable name**: `array`
- size**: `5`
- elements**: `{ 1, 2, 3, 4, 5 }`

Array Syntax: Creation

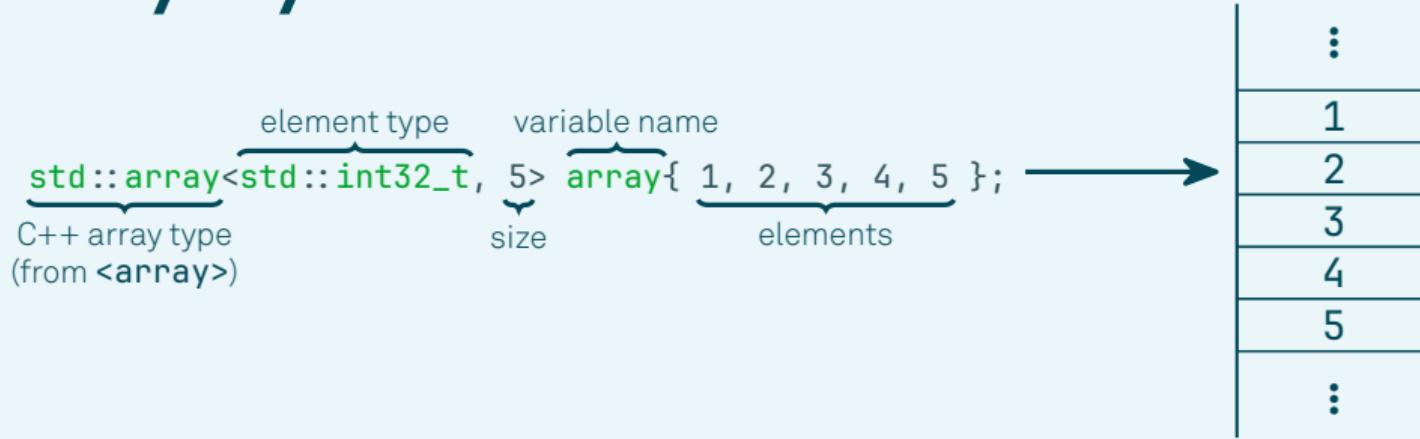
`std::array<std::int32_t, 5> array{ 1, 2, 3, 4, 5 };`

element type
variable name
size
elements

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

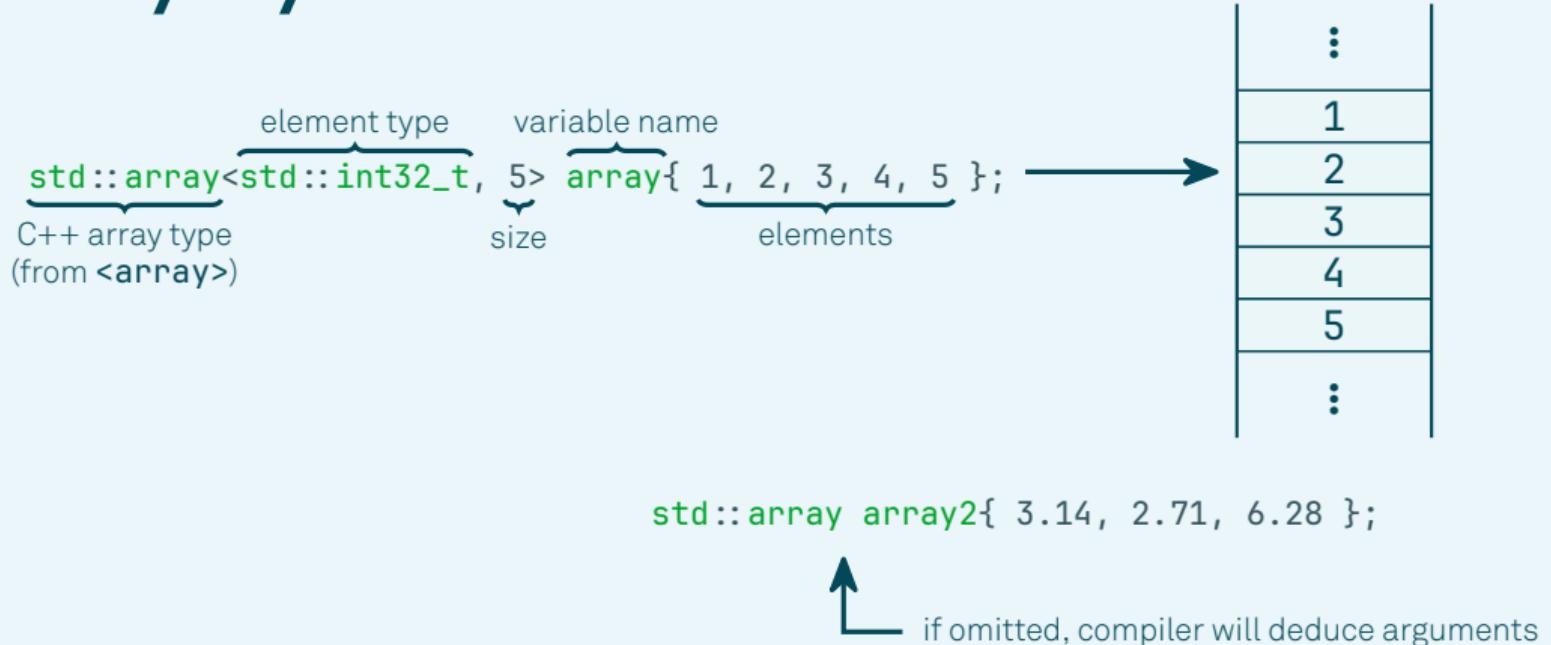


Array Syntax: Creation

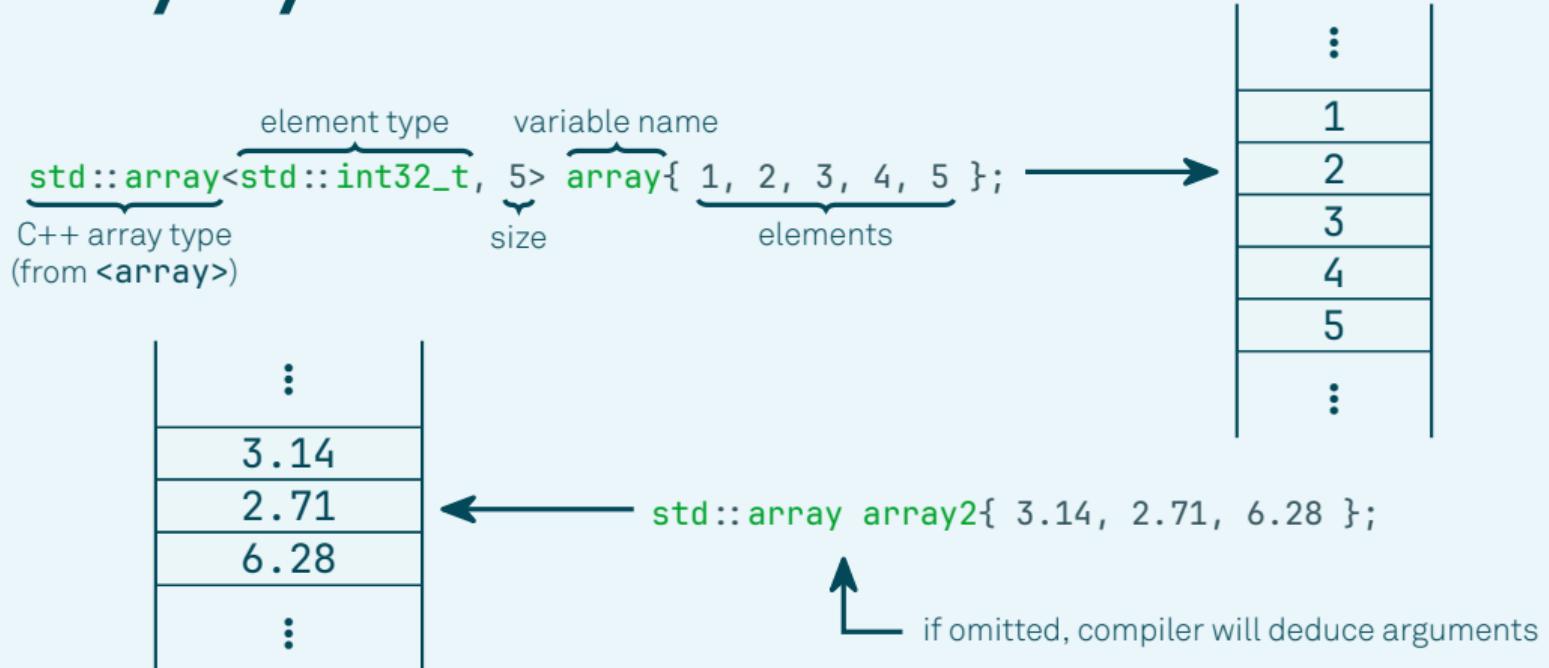


```
std::array array2{ 3.14, 2.71, 6.28 };
```

Array Syntax: Creation



Array Syntax: Creation



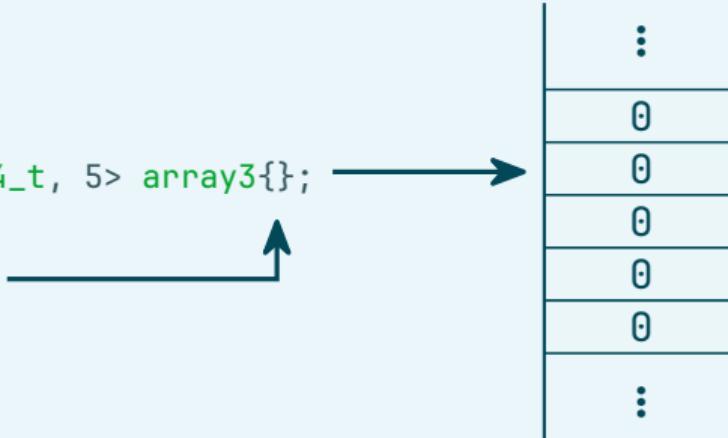
Array Syntax: Creation

```
std::array<std::uint64_t, 5> array3{};
```

Array Syntax: Creation

```
std::array<std::uint64_t, 5> array3{};
```

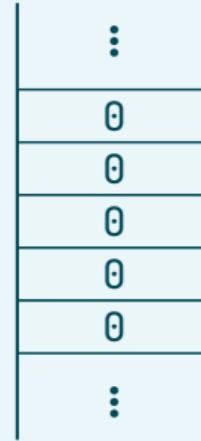
perform zero-initialization
array will be uninitialized without these



Array Syntax: Creation

```
std::array<std::uint64_t, 5> array3{};
```

perform zero-initialization
array will be uninitialized without these



```
std::array<double, 5> array4{ 3.14, 2.71, 6.28 };
```

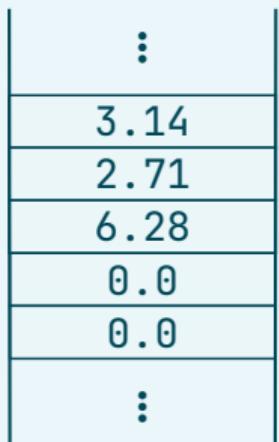
Array Syntax: Creation

```
std::array<std::uint64_t, 5> array3{};
```

perform zero-initialization
array will be uninitialized without these



```
std::array<double, 5> array4{ 3.14, 2.71, 6.28 };
```



Array Syntax: Creation

```
1 std::array<char, 3> chars{ 'a', 'b', 'c', 'd' };
```

Array Syntax: Creation

```
1 std::array<char, 3> chars{ 'a', 'b', 'c', 'd' };
```

```
/project/src/main.cpp:1:47: error: excess elements in struct initializer
```

```
1 |     std::array<char, 3> chars{ 'a', 'b', 'c', 'd' };  
|  
|  
^~~
```

```
1 error generated.
```

Array Syntax: Accessing

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

Array Syntax: Accessing

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

1

Array Syntax: Accessing

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

1

```
3 cout << array.back() << '\n';
```

Array Syntax: Accessing

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

1

```
3 cout << array.back() << '\n';
```

5

Array Syntax: Accessing

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

1

```
3 cout << array.back() << '\n';
```

5

```
4 cout << array.at(1) << '\n';
```

Array Syntax: Accessing

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

1

```
3 cout << array.back() << '\n';
```

5

```
4 cout << array.at(1) << '\n';
```

2

Array Syntax: Accessing

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

Array Syntax: Accessing

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

```
size: 5
```

Array Syntax: Accessing

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

size: 5

```
6 cout << array.at(array.size() >> 1) << endl;
```

Array Syntax: Accessing

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

```
size: 5
```

```
6 cout << array.at(array.size() >> 1) << endl;
```

```
3
```

```
Process finished with exit code 0
```

Array Syntax: Accessing

```
1 std::array<double, 10> numbers{};  
2 cout << numbers.at(17) << endl;
```

Array Syntax: Accessing

```
1 std::array<double, 10> numbers{};  
2 cout << numbers.at(17) << endl;
```

libc++abi: terminating due to uncaught exception of type std::out_of_range: array::at

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

Array Syntax: Modifying

```
1 std::array<std::string, 3> words{};
2 words.front() = "hello";
3 cout << words.front() << '\n';
```

Array Syntax: Modifying

```
1 std::array<std::string, 3> words{};
2 words.front() = "hello";
3 cout << words.front() << '\n';
```

"hello"

Array Syntax: Modifying

```
1 std::array<std::string, 3> words{};  
2 words.front() = "hello";  
3 cout << words.front() << '\n';
```

"hello"

```
4 words.at(1) = "world";  
5 cout << words.at(1) << endl;
```

Array Syntax: Modifying

```
1 std::array<std::string, 3> words{};  
2 words.front() = "hello";  
3 cout << words.front() << '\n';
```

"hello"

```
4 words.at(1) = "world";  
5 cout << words.at(1) << endl;
```

"world"

Process finished with exit code 0

Array Syntax: Modifying

```
1 std::array<bool, 10> flags{};  
2 flags.at(-5) = true;
```

Array Syntax: Modifying

```
1 std::array<bool, 10> flags{};  
2 flags.at(-5) = true;
```

libc++abi: terminating due to uncaught exception of type std::out_of_range: array::at

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

Array Syntax: Misc.

```
1 std::array<std::int32_t, 10> values{};  
2 values.fill(1);  
3 cout << "values: " << values << '\n';
```

Array Syntax: Misc.

```
1 std::array<std::int32_t, 10> values{};  
2 values.fill(1);  
3 cout << "values: " << values << '\n';
```

```
values: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Array Syntax: Misc.

```
1 std::array<std::int32_t, 10> values{};  
2 values.fill(1);  
3 cout << "values: " << values << '\n';
```

```
values: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
4 cout << boolalpha << "is values empty: " << values.empty() << '\n';
```

Array Syntax: Misc.

```
1 std::array<std::int32_t, 10> values{};  
2 values.fill(1);  
3 cout << "values: " << values << '\n';
```

```
values: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
4 cout << boolalpha << "is values empty: " << values.empty() << '\n';
```

```
is values empty: false
```

```
Process finished with exit code 0
```

Array Syntax: Iteration

```
1 constexpr std::array<std::int16_t, 5> nums{ 1, -2, 23, 4, 13 };
```

```
2 cout << "nums: ";
3 for (std::size_t i{0U}; i < nums.size(); ++i) {
4     cout << nums.at(i) << ' ';
5 }
6 cout << '\n';
```

Output:

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

Array Syntax: Iteration

```
1 constexpr std::array<std::int16_t, 5> nums{ 1, -2, 23, 4, 13 };
```

```
2 cout << "nums: ";
3 for (std::size_t i{0U}; i < nums.size(); ++i) {
4     cout << nums.at(i) << ' ';
5 }
6 cout << '\n';
```

Output:

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

```
7 cout << "nums: ";
8 for (std::size_t i{nums.size()}; i > 0U; --i) {
9     cout << nums.at(i - 1U) << ' ';
10 }
11 cout << '\n';
```

Output:

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

Array Syntax: Iteration

```
12 cout << "nums: ";
13 for (auto iter{nums.cbegin()}; iter != nums.cend(); ++iter) {
14     cout << *iter << ' ';
15 }
16 cout << '\n';
```

Output:

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

Array Syntax: Iteration

```
12 cout << "nums: ";
13 for (auto iter{nums.cbegin()}; iter != nums.cend(); ++iter) {
14     cout << *iter << ' ';
15 }
16 cout << '\n';
```

Output:

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

```
17 cout << "nums: ";
18 for (auto iter{nums.crbegin()}; iter != nums.crend(); ++iter) {
19     cout << *iter << ' ';
20 }
21 cout << '\n';
```

Output:

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

Array Syntax: Iteration

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

Output:

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

Special Considerations

Array Syntax: Printing

- By default, arrays are not printable with `std :: cout`

Array Syntax: Printing

- By default, arrays are not printable with `std::cout`

```
std::array<std::uint16_t, 3> scores{};  
cout << scores << endl;
```



C++ has no idea how to use the
insertion operator with `std::array`

Array Syntax: Printing

- By default, arrays are not printable with `std::cout`

```
std::array<std::uint16_t, 3> scores{};  
cout << scores << endl;
```



C++ has no idea how to use the
insertion operator with `std::array`

- What if we could tell C++ what to do?

Array Syntax: Printing

Operator overloading

Operator overloading is when we tell a programming language how to use a particular operator with certain types

Array Syntax: Printing

Operator overloading

Operator overloading is when we tell a programming language how to use a particular operator with certain types

- New types feel like existing types

Array Syntax: Printing

Operator overloading

Operator overloading is when we tell a programming language how to use a particular operator with certain types

- New types feel like existing types
- Can specify operator behavior externally

Array Syntax: Printing

Operator overloading

Operator overloading is when we tell a programming language how to use a particular operator with certain types

- New types feel like existing types
- Can specify operator behavior externally
- Can make code both more readable and less readable

Array Syntax: Printing

- How do we overload operators?

```
constexpr std::string name{"Stroustrup"};
cout << "Nice to meet you, " << name << "!\n";
```

Array Syntax: Printing

- How do we overload operators? Operators are actually functions!

```
constexpr std::string name{"Stroustrup"};
cout << "Nice to meet you, " << name << "!\n";
```

```
constexpr std::string name{"Stroustrup"};
operator<<(operator<<(operator<<(cout, "Nice to meet you, "), name), "!\n");
```

Array Syntax: Printing

- How do we overload operators? Operators are actually functions!

```
constexpr std::string name{"Stroustrup"};
cout << "Nice to meet you, " << name << "!\n";
```

```
constexpr std::string name{"Stroustrup"};
operator<<(operator<<(operator<<(operator<<(cout, "Nice to meet you, "), name), "!\n"));
```

Nice to meet you, Stroustrup!

Array Syntax: Printing

- To overload the insertion operator, we need to define a function with the following signature:

```
std::ostream &operator<<(std::ostream &ostream, const Type &object);
```

Array Syntax: Printing

- To overload the insertion operator, we need to define a function with the following signature:

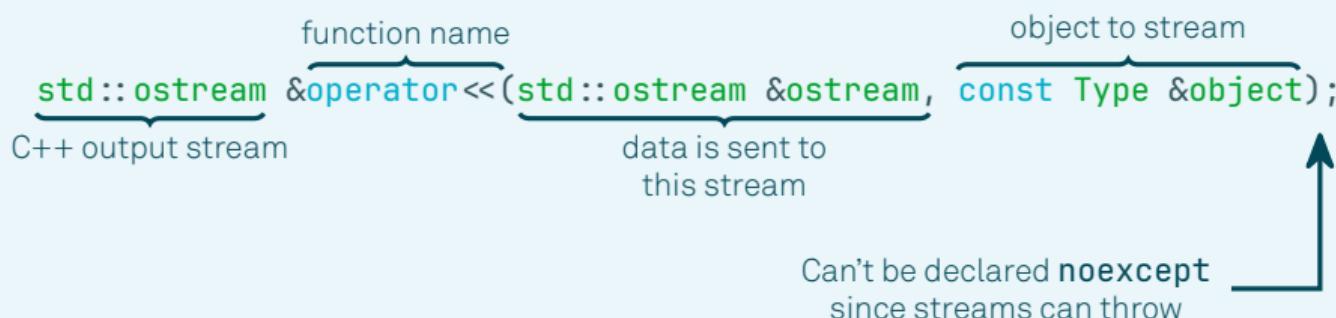
```
function name  
std::ostream &operator<<(std::ostream &ostream,  
                           const Type &object);  
C++ output stream          data is sent to  
                           this stream          object to stream
```

Array Syntax: Printing

- To overload the insertion operator, we need to define a function with the following signature:

```
function name  
std::ostream &operator<<(std::ostream &ostream,  
                           const Type &object);  
C++ output stream          data is sent to  
                           this stream          object to stream
```

Can't be declared **noexcept**
since streams can throw



Array Syntax: Printing

- To overload the insertion operator, we need to define a function with the following signature:

```
function name  
std::ostream &operator<<(std::ostream &ostream,  
                           const Type &object);  
  
C++ output stream          data is sent to  
                           this stream          object to stream  
  
argument is returned to     Can't be declared noexcept  
allow for pipelining         since streams can throw
```

The diagram shows the signature of the insertion operator: `function name std::ostream &operator<<(std::ostream &ostream, const Type &object);`. Annotations with arrows point to specific parts of the code:

- An arrow from the left points to `std::ostream`, labeled "C++ output stream".
- An arrow from the top points to `&operator<<`, labeled "function name".
- An arrow from the middle points to `ostream` in the second parameter, labeled "data is sent to this stream".
- An arrow from the right points to `const Type &object`, labeled "object to stream".
- An arrow from the bottom left points to the closing parenthesis of the first parameter, labeled "argument is returned to allow for pipelining".
- An arrow from the bottom right points to the `&` before `object`, labeled "Can't be declared noexcept since streams can throw".

Array Syntax: Printing

- This means our insertion operator should look like the following:

```
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<std::int32_t, 5> &array);
```

Array Syntax: Printing

- This means our insertion operator should look like the following:

```
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<std::int32_t, 5> &array);
```

- Does this look good?

Array Syntax: Printing

- How do we write generic functions that accept different types of arrays?

Array Syntax: Printing

- How do we write generic functions that accept different types of arrays? Templating!

Array Syntax: Printing

- How do we write generic functions that accept different types of arrays? Templating!

What is templating?

Templating is a way of writing code that allows us to define a *family* of code entities

Array Syntax: Printing

```
std::int32_t add(const std::int32_t a, const std::int32_t b) noexcept {  
    return a + b;  
}
```

Array Syntax: Printing

```
std::int32_t add(const std::int32_t a, const std::int32_t b) noexcept {  
    return a + b;  
}
```

- What about this function is specific to **std::int32_t** values?

Array Syntax: Printing

```
std::int32_t add(const std::int32_t a, const std::int32_t b) noexcept {  
    return a + b;  
}
```

- What about this function is specific to `std::int32_t` values?
- How can we write this function so it works with any type that supports `operator+`?

Array Syntax: Printing

```
template <typename T>
T add(const T &a, const T &b) { return a + b; }
```

Array Syntax: Printing

declares a template
template arguments

template <typename T>

```
T add(const T &a, const T &b) { return a + b; }
```

Array Syntax: Printing

declares a template arguments

`template <typename T>`

`T add(const T &a, const T &b) { return a + b; }`

The diagram illustrates the relationship between the template declaration and the function parameters. It features two horizontal curly braces above the code. The first brace spans from the start of 'template' to the end of '<typename T>'. The second brace spans from the start of 'T' to the end of 'T'. Three dark blue arrows point from these two braces down to the word 'T' in the function parameters 'const T &a' and 'const T &b'. Below this, the text 'act as placeholders for some type T' is centered.

act as placeholders for some type T

Array Syntax: Printing

declares a template arguments

`template <typename T>`

`T add(const T &a, const T &b) { return a + b; }`

The diagram illustrates the relationship between the template declaration and the function parameters. It features two horizontal curly braces above the code. The first brace spans from the start of 'template' to the end of '<typename T>'. The second brace spans from the start of 'T' to the end of 'T'. Three dark blue arrows point from these brace spans down to the corresponding 'T' characters in the 'add' function's parameter list: one arrow from the first brace to the first 'T', another from the second brace to the second 'T', and a third from the first brace to the third 'T'.

act as placeholders for some type `T`

- Represents a family of functions where `T` could be any type.

Array Syntax: Printing

declares a template arguments

`template <typename T>`

`T add(const T &a, const T &b) { return a + b; }`

act as placeholders for some type `T`



- Represents a family of functions where `T` could be any type.
- Does this function seem like it would cause any issues?

Array Syntax: Printing

```
constexpr std::string string1{"Hello, "};  
constexpr std::string string2{"world!"};  
cout << "add(3, 5) = " << add(3, 5) << '\n';  
cout << "add(3.14, 2.71) = " << add(3.14, 2.71) << '\n';  
cout << "add(string1, string2) = " << add(string1, string2) << '\n';
```

```
add(3, 5) = 8  
add(3.14, 2.71) = 5.85  
add(string1, string2) = Hello, world!
```

Array Syntax: Printing

```
constexpr std::string string1{"Hello, "};  
constexpr std::string string2{"world!"};  
cout << "add(3, 5) = " << add(3, 5) << '\n';  
cout << "add(3.14, 2.71) = " << add(3.14, 2.71) << '\n';  
cout << "add(string1, string2) = " << add(string1, string2) << '\n';
```

```
add(3, 5) = 8  
add(3.14, 2.71) = 5.85  
add(string1, string2) = Hello, world!
```

- How do templates work?

Array Syntax: Printing

```
constexpr std::string string1{"Hello, "};  
constexpr std::string string2{"world!"};  
cout << "add(3, 5) = " << add(3, 5) << '\n';  
cout << "add(3.14, 2.71) = " << add(3.14, 2.71) << '\n';  
cout << "add(string1, string2) = " << add(string1, string2) << '\n';
```

at compile-time, compiler generates
a version for each function call

add(3, 5) = 8
add(3.14, 2.71) = 5.85
add(string1, string2) = Hello, world!

- How do templates work?

Array Syntax: Printing

- At compile time:

```
template <typename T>
T add(const T &a, const T &b) {
    return a + b;
}
```

```
template <>
int add(const int &a, const int &b) {
    return a + b;
}
```

```
template <>
double add(const double &a, const double &b) {
    return a + b;
}
```

```
template <>
std::string add(const std::string &a, const std::string &b) {
    return a + b;
}
```

Array Syntax: Printing

- Key aspects of templates

Array Syntax: Printing

- Key aspects of templates
 - Templates represent families of different code entities (for now, we only care about function templates)

Array Syntax: Printing

- Key aspects of templates
 - Templates represent families of different code entities (for now, we only care about function templates)
 - Templates are resolved at compile-time

Array Syntax: Printing

- Key aspects of templates
 - Templates represent families of different code entities (for now, we only care about function templates)
 - Templates are resolved at compile-time
 - After templates are resolved, it's as if we wrote separate versions for each code entity

Array Syntax: Printing

- Implications of templates

Array Syntax: Printing

- Implications of templates
 - What code will be generated if we don't use templated code?

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated
 - How might templates negatively affect compilation, if at all?

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated
 - Templated code takes longer to compile

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated
 - Templated code takes longer to compile
 - How might templates negatively affect application size, if at all?

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated
 - Templated code takes longer to compile
 - The more templated code is used, the more code is generated

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated
 - Templated code takes longer to compile
 - The more templated code is used, the more code is generated
 - How might templates negatively affect runtime, if at all?

Array Syntax: Printing

- Implications of templates
 - If a template is not used, no code is generated
 - Templated code takes longer to compile
 - The more templated code is used, the more code is generated
 - Templates don't negatively affect runtime since the magic happens at compile-time

Array Syntax: Printing

- How does this help with our insertion-operator overload?

Array Syntax: Printing

- How does this help with our insertion-operator overload?

```
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<std::int32_t, 5> &array);
```

Array Syntax: Printing

- How does this help with our insertion-operator overload?

```
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<std::int32_t, 5> &array);
```



```
template <typename T>  
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<T, 5> &array);
```

Array Syntax: Printing

- How does this help with our insertion-operator overload?

```
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<std::int32_t, 5> &array);
```

```
template <typename T>  
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<T, 5> &array);
```

```
template <typename T, std::size_t size>  
std::ostream &operator<<(std::ostream &ostream,  
                           const std::array<T, size> &array);
```

Array Syntax: Printing

- At compile time:

```
constexpr std::array nums{ 3.14, 2.71, 6.28 };
constexpr std::array<std::int32_t, 2> nums2{ 1, 2 };
cout << "nums: " << nums << '\n';
cout << "nums2: " << nums2 << '\n';
```

Array Syntax: Printing

- At compile time:

```
constexpr std::array nums{ 3.14, 2.71, 6.28 };
constexpr std::array<std::int32_t, 2> nums2{ 1, 2 };
cout << "nums: " << nums << '\n';
cout << "nums2: " << nums2 << '\n';
```



```
template <>
std::ostream &operator<<(std::ostream &os, const std::array<double, 3> &array);
```

Array Syntax: Printing

- At compile time:

```
constexpr std::array nums{ 3.14, 2.71, 6.28 };
constexpr std::array<std::int32_t, 2> nums2{ 1, 2 };
cout << "nums: " << nums << '\n';
cout << "nums2: " << nums2 << '\n';
```

template <>
std::ostream &operator<<(std::ostream &os, const std::array<double, 3> &array);

template <>
std::ostream &operator<<(std::ostream &os, const std::array<std::int32_t, 2> &array);

Undefined Behavior

- We can also use the subscript operator ([]) to access elements:

```
std::array nums{ 1.1, 2.2, 3.3 };
cout << "nums: " << nums << '\n';
nums[0] = 4.4;
cout << "first element: " << nums[0] << '\n';
cout << "nums: " << nums << '\n';
```

```
nums: [1.1, 2.2, 3.3]
first element: 4.4
nums: [4.4, 2.2, 3.3]
```

```
Process finished with exit code 0
```

Undefined Behavior

- What will happen if this code is run?

```
1 constexpr std::array nums{ 1.1, 2.2, 3.3, 4.4, 5.5 };
2 std::array nums2{ 1.23, 4.56, 2.2 };
3 cout << "nums[100]: " << nums[100] << '\n';
4 cout << "nums2[-20]: " << nums2[-20] << '\n';
```

Undefined Behavior

- What will happen if this code is run?

```
1 constexpr std::array nums{ 1.1, 2.2, 3.3, 4.4, 5.5 };
2 std::array nums2{ 1.23, 4.56, 2.2 };
3 cout << "nums[100]: " << nums[100] << '\n';
4 cout << "nums2[-20]: " << nums2[-20] << '\n';
```

nums[100]: 4.44388e+252
nums2[-20]: 2.15003e-314

Undefined Behavior

- What will happen if this code is run?

```
1 constexpr std::array nums{ 1.1, 2.2, 3.3, 4.4, 5.5 };
2 std::array nums2{ 1.23, 4.56, 2.2 };
3 cout << "nums[100]: " << nums[100] << '\n';
4 cout << "nums2[-20]: " << nums2[-20] << '\n';
```

nums[100]: 4.44388e+252
nums2[-20]: 2.15003e-314

```
5 nums2[5] = -33.3;
6 cout << "nums: " << nums << '\n';
```

Undefined Behavior

- What will happen if this code is run?

```
1 constexpr std::array nums{ 1.1, 2.2, 3.3, 4.4, 5.5 };
2 std::array nums2{ 1.23, 4.56, 2.2 };
3 cout << "nums[100]: " << nums[100] << '\n';
4 cout << "nums2[-20]: " << nums2[-20] << '\n';
```

```
nums[100]: 4.44388e+252
nums2[-20]: 2.15003e-314
```

```
5 nums2[5] = -33.3;
6 cout << "nums: " << nums << '\n';
```

```
nums: [1.1, 2.2, -33.3, 4.4, 5.5]
```

```
Process finished with exit code 0
```

Undefined Behavior

- How far can this be pushed?

Undefined Behavior

- How far can this be pushed?

```
7 cout << "nums[1'000]: " << nums[1'000] << endl;
```

Undefined Behavior

- How far can this be pushed?

```
7 cout << "nums[1'000]: " << nums[1'000] << endl;
```

```
Process finished with exit code 139 (interrupted by signal 11:SIGSEGV)
```

Undefined Behavior

- How far can this be pushed?

```
7 cout << "nums[1'000]: " << nums[1'000] << endl;
```

```
Process finished with exit code 139 (interrupted by signal 11:SIGSEGV)
```

- Why allow this?

Multi-dimensional Arrays

Two-dimensional Arrays

- Arrays of arrays

Two-dimensional Arrays

- Arrays of arrays

```
1 std::array<std::array<std::int32_t, 3>, 3> grid{{  
2     { 1, 2, 3 },  
3     { 4, 5, 6 },  
4     { 7, 8, 9 },  
5 };
```

Two-dimensional Arrays

- Arrays of arrays

```
1 std::array<std::array<std::int32_t, 3>, 3> grid{{  
2     { 1, 2, 3 },  
3     { 4, 5, 6 },  
4     { 7, 8, 9 },  
5 }};
```



note the double curly brackets
needed for multi-dimensional arrays

Two-dimensional Arrays

- Arrays of arrays

```
1 std::array<std::array<std::int32_t, 3>, 3> grid{{  
2     { 1, 2, 3 },  
3     { 4, 5, 6 },  
4     { 7, 8, 9 },  
5 }};
```

grid	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

note the double curly brackets
needed for multi-dimensional arrays

Two-dimensional Arrays

```
6 cout << "grid: " << grid << '\n';
7 cout << "center value: " << grid.at(1).at(1) << '\n';
8 grid.at(1) = { 40, 50, 60 };
9 cout << "grid: " << grid << endl;
```

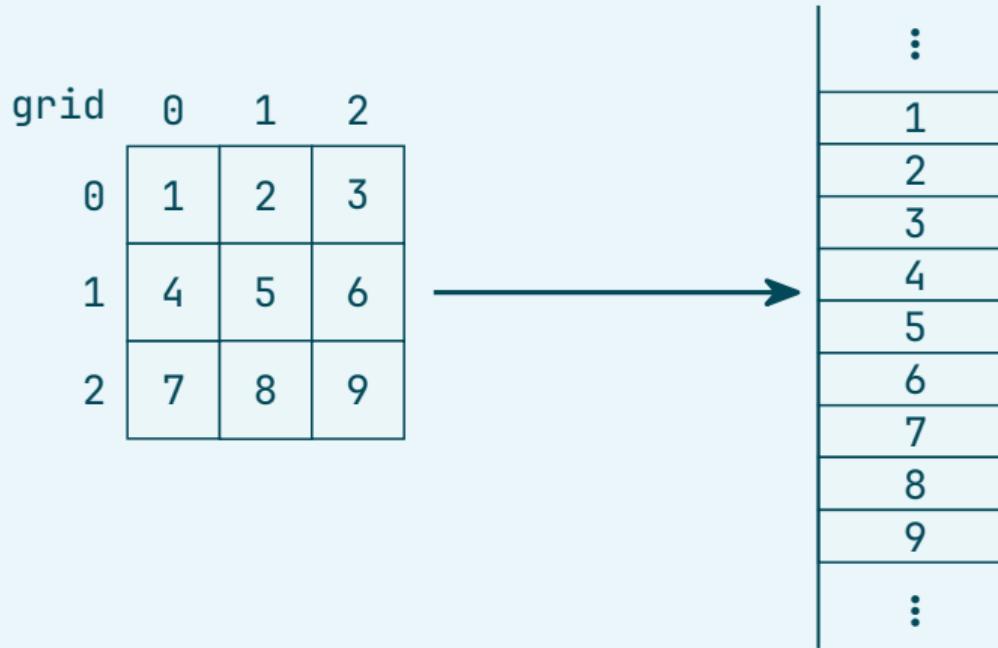
```
grid: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
center value: 5
grid: [[1, 2, 3], [40, 50, 60], [7, 8, 9]]
```

Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?

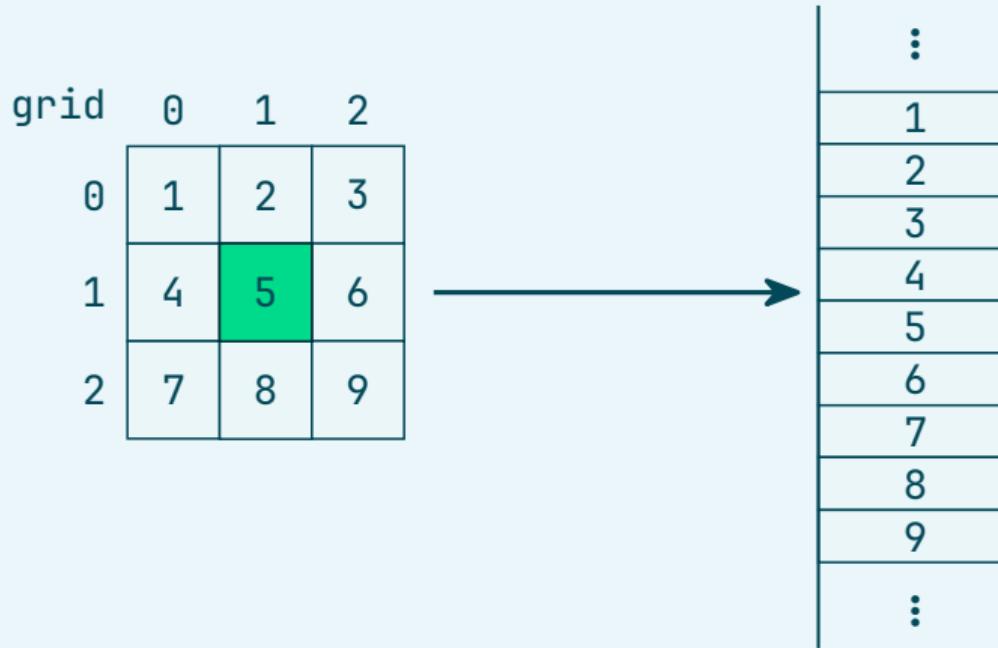
Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



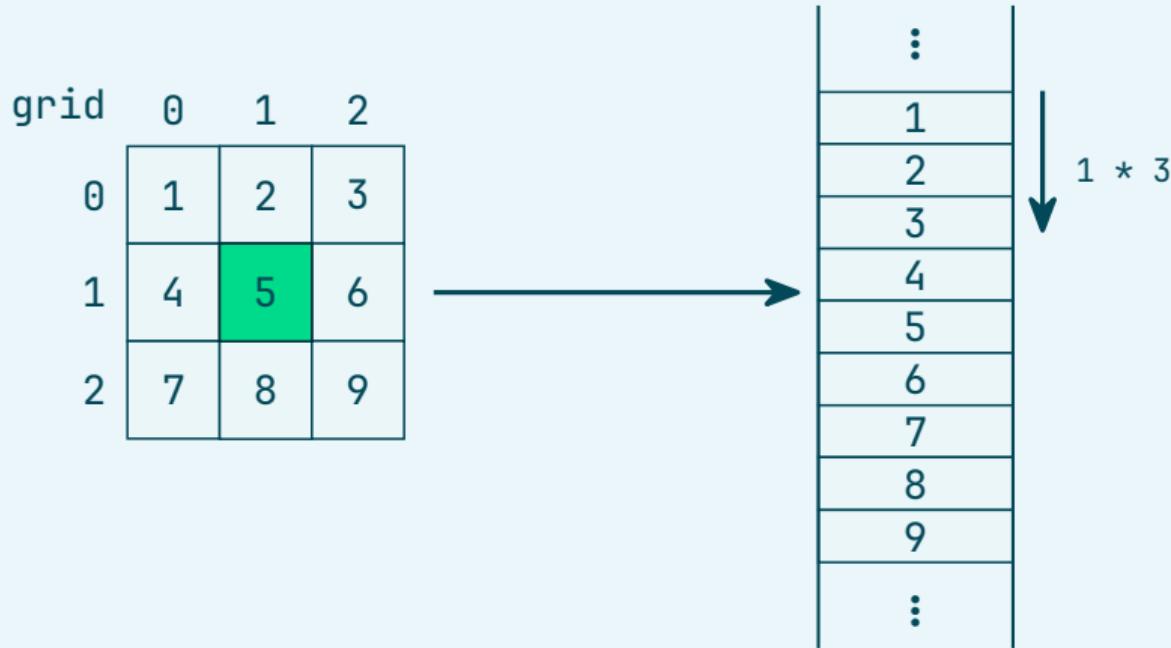
Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



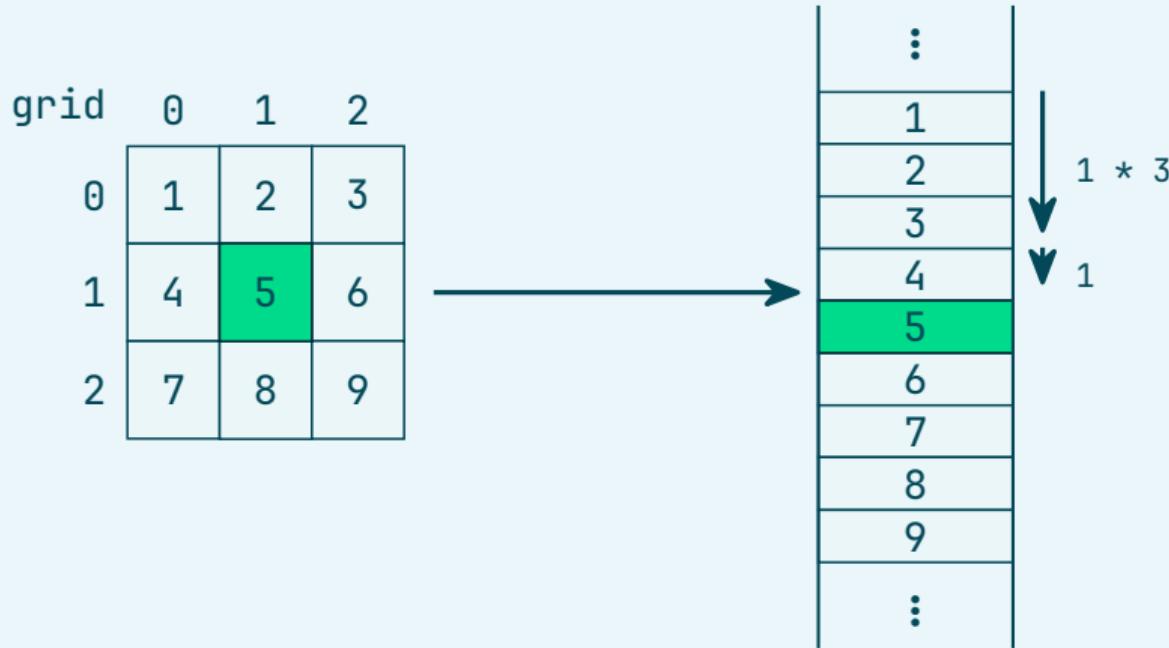
Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



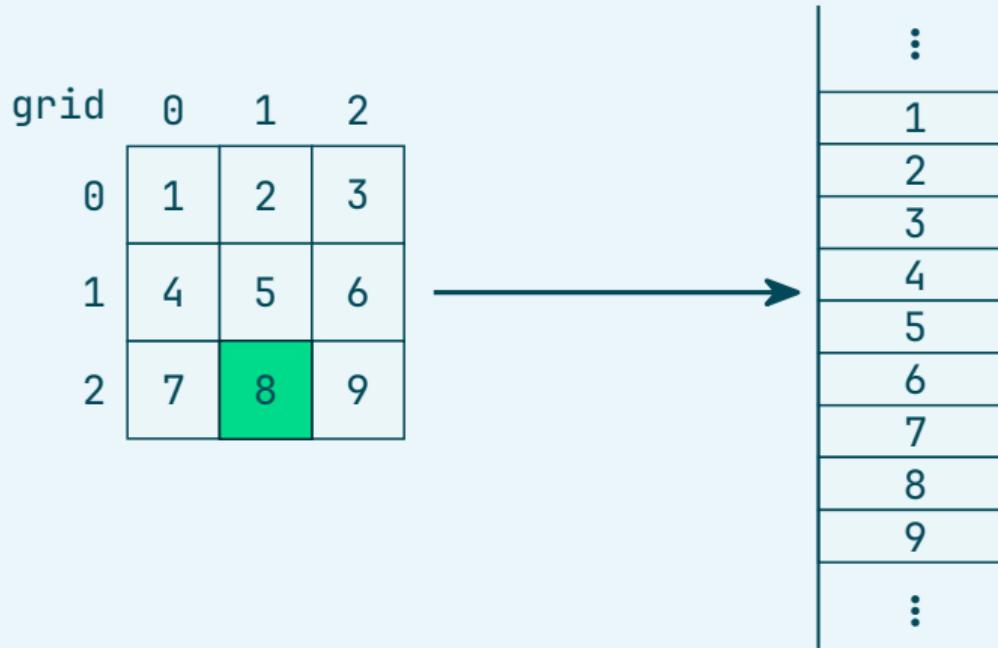
Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



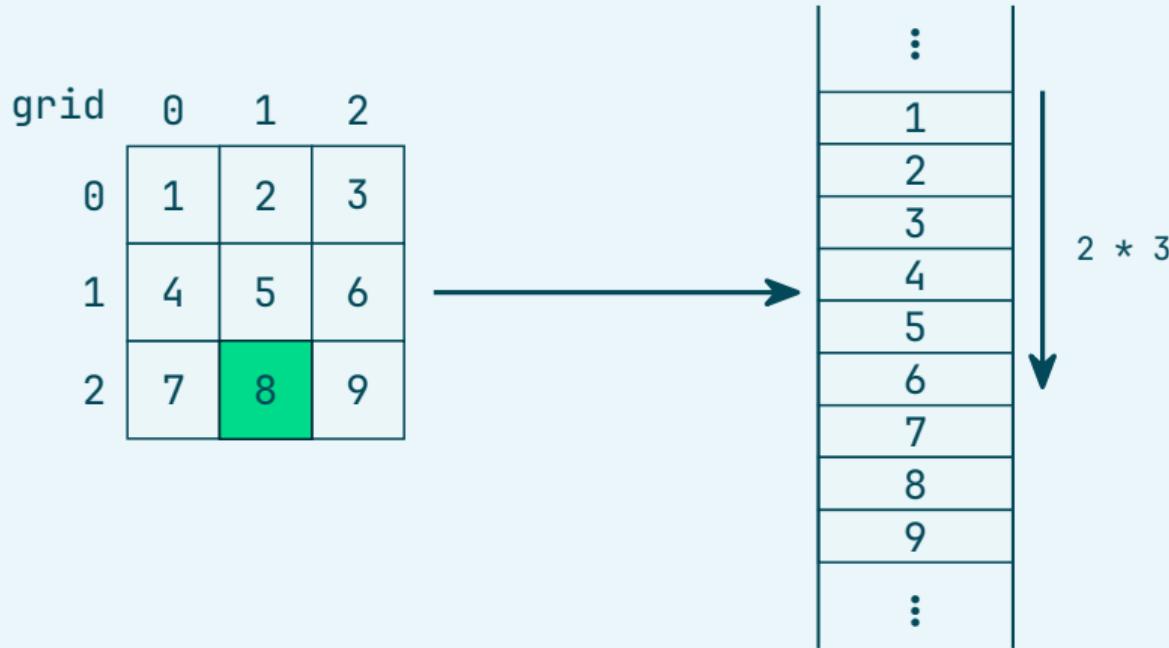
Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



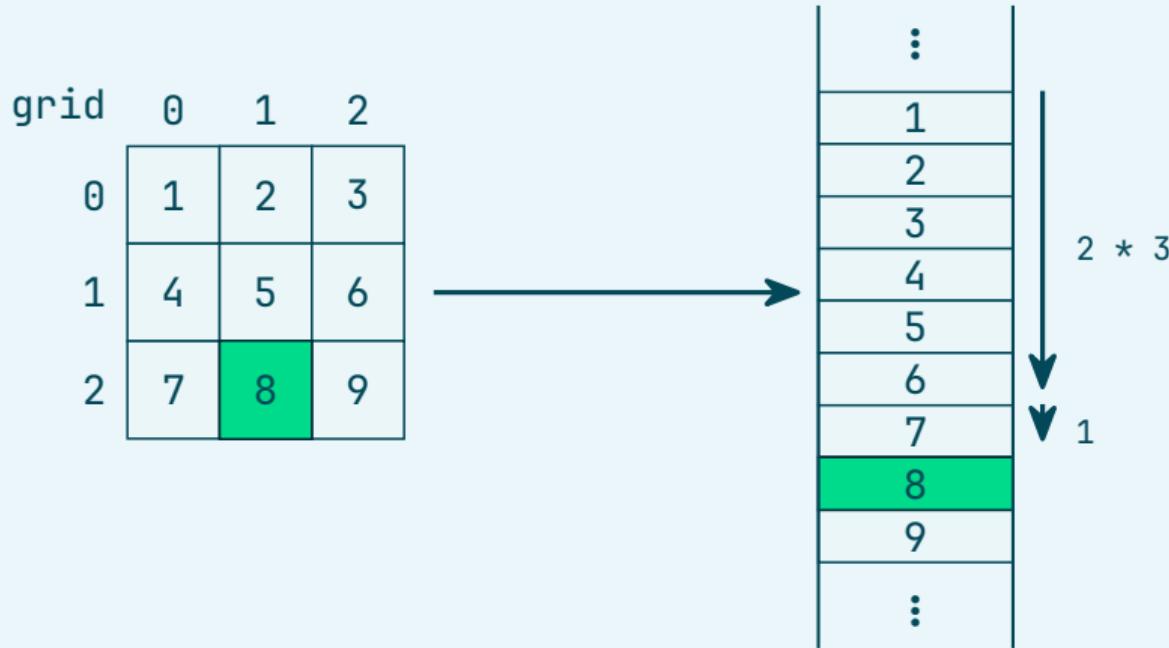
Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



Two-dimensional Arrays

- What do two-dimensional arrays look like in memory?



Multi-dimensional Arrays

- These concepts extend to more than two dimensions

Multi-dimensional Arrays

- These concepts extend to more than two dimensions
 - Three-dimensional array: rectangular prism

Multi-dimensional Arrays

- These concepts extend to more than two dimensions
 - Three-dimensional array: rectangular prism
 - Four-dimensional array: hyper rectangular prism

Multi-dimensional Arrays

- These concepts extend to more than two dimensions
 - Three-dimensional array: rectangular prism
 - Four-dimensional array: hyper rectangular prism
 - n -dimensional array: n -dimensional shape

Multi-dimensional Arrays

- These concepts extend to more than two dimensions
 - Three-dimensional array: rectangular prism
 - Four-dimensional array: hyper rectangular prism
 - n -dimensional array: n -dimensional shape
- Most uses of multi-dimensional arrays in systems programming stop at three-dimensional arrays

Any Questions?