Back to Basics:
Pointers

MIKE SHAH

Cppcon
The C++ Conference

2021
October 24-29

Please do not redistribute slides without prior permission.

Back to Basics: Pointers

Mike Shah, Ph.D.

@MichaelShah | mshah.io | www.youtube.com/c/MikeShah

3:15 pm MDT, Mon. October 25

60 minutes | Introductory Audience

Abstract

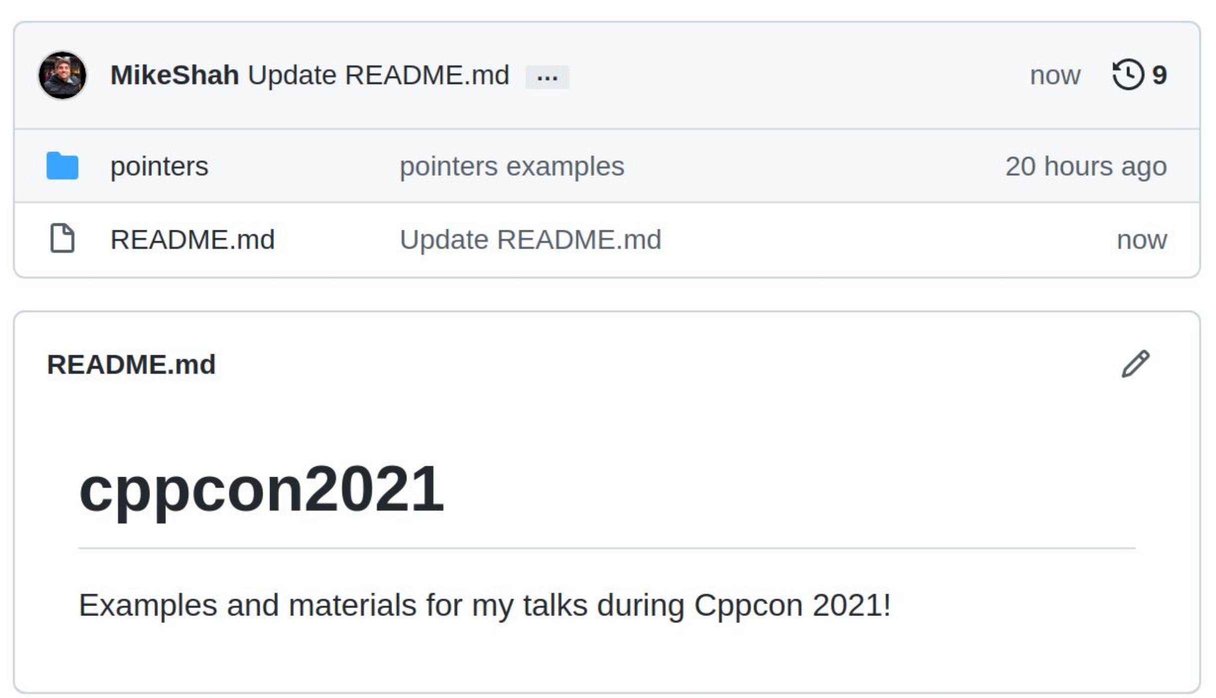
The abstract that you read and enticed you to join me is here!

Pointers are scary. Unfortunately that previous statement is what many beginners take away when first learning about pointers and the C++ language. In this talk, we will discuss the low level foundations of what a raw pointer is--a variable that stores an address. We will then see some examples of raw pointers for creating data structures, passing data into functions, dynamically allocated arrays, and function pointers. This portion will cover capabilities of raw pointers and syntax: * (asterisk), .(dot) , -> (arrow). By the end of the first portion of the talk, we will find pointers are not scary, but just another tool we can use in our programmers' toolbox.

After learning the foundations, we are then going to discuss some of the pitfalls of pointers (e.g. nullptr's, double frees, memory leaks). However, with modern C++, we can abstract away some of these problems using various "smart pointers" built into the standard library in <memory>. Attendees will leave understanding how we can use pointers in a safe manner through the standard library smart pointer abstractions.

Code for the talk

- Located here: https://github.com/MikeShah/cppcon2021



Who Am I?

by Mike Shah

- Assistant Teaching Professor at Northeastern University in Boston, Massachusetts.
    - I teach courses in computer systems, computer graphics, and game engine development.
    - My research in program analysis is related to performance building static/dynamic analysis and software visualization tools.
- I do consulting and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
    - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of computer graphics, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io

One of my fondest programming memories was...

... when I used a pointer correctly on the first try

- And maybe as a C or C++ programmer you have a similar memory or ‘eureka’ moment.
    - My sophomore year in college where I remember doing lots of small pointer examples similar to the right
    - This is what it took for me to understand pointers
        - It was supremely satisfying to see my code compile successfully
            - (Knowing that I did not ‘guess’ where the * and the & go.)

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

But truth be told, it was probably in graduate school....

... that I really understood the power of pointers

- It was at this point that I had more computer systems knowledge.
  - I had a mental model of a computer's memory
  - I was building data structures which were using pointers
  - And I could explain how to use pointers to other students.
- And post graduate school, I think about ‘ownership’, ‘lifetime’, ‘levels of indirection’ for performance and readability (Demeter's Law), and memory-safety.

So for this talk--I want to make:

1.) Pointers not be scary -- by showing their usage
   a.) (i.e., you don’t have to guess where an asterisk goes)
2.) Show how to use pointers and avoid potential pitfalls
   a.) Using lots of small examples, ranging from simple to more advanced usages (e.g., function pointers)
3.) And to appreciate that we have pointers in C++
   a.) Closing with modern C++ features (std::function and smart pointers)

... that I really understood the power of pointers

- It was at this point that I had more computer systems knowledge
  - I ha
  - I wa
  - And
- And pos ... of
  indirect ... nory-safety.

My disclaimer to C++ experts in attendance (or online) -- this is an introductory level talk focusing on fundamentals. I don’t want to lose future C++ programmers because they get scared of pointers.

However, it may be useful to see how I teach the topic, or otherwise how I introduce pointers in relation to modern C++ features towards the end.

So for this ta

1.) Pointers
  a.) (i.e., you don’t have to guess where an asterisk goes)
2.) Show how to use pointers and avoid potential pitfalls
  a.) Using lots of small examples, ranging from simple to more advanced usages (e.g., function pointers)
3.) And to appreciate that we have pointers in C++
  a.) Closing with modern C++ features (std::function and smart pointers)

Let’s start from the beginning*

What is a pointer?

*Although, not the exact beginning. My internet research shows the invention of pointers get credited to Harold Lawson in 1964 for the invention, though pointers may have been invented by Kateryna Yushchenko in the Address Programming Language around 1955.

What is a Pointer? (1/8)

- A **pointer** is a variable that stores the memory address of a specific object type
  - (Let's look at an example)

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

What is a Pointer? (2/8)

- A **pointer** is a variable that stores the memory address of a specific object type

- px is a pointer
- px's type is `int*`
    - px can stores addresses of integers.

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

What is a Pointer? (3/8)

- A **pointer** is a variable that stores the memory address of a specific object type

```
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout     "x is: " << x << std::endl;
    std::cout     "*px is: " << *px << std::endl;

    retur
}
```

- px is a pointer
- px’s type is `int*`
  - px can stores addresses of integers.

- I retrieve the address of a variable using the ampersand operator (&)
  - You could also use &(x) if you like
    - e.g., `int* px = &(x);`

What is a Pointer? (4/8)

- A **pointer** is a variable that stores the memory address of a specific object type

- px is a pointer
- px's type is `int*`
  - px can stores addresses of integers.

- The '=' (assignment operator) stores the address of x (remember, &x) inside of px.
  - We say 'px points to x'

```
// @fi      e.cpp
// g++ -      initialize.cpp -o prog
#include      am>

int main(){

    int x = 7;
    int* px = &x;

    std::cout    "x is: " << x << std::endl;
    std::cout    "*px is: " << *px << std::endl;

    return
}
```

- I retrieve the address of a variable using the ampersand operator (&)
  - You could also use &(x) if you like
    - e.g., `int* px = &(x);`

What is a Pointer? (5/8)

- A **pointer** is a variable that stores the memory address of a specific object type
- So if ‘px’ stores the address of x this allows us to:
  - access the value stored in ‘x’ through px indirectly.

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

What is a Pointer? (6/8)

- A **pointer** is a variable that stores the memory address of a specific object type
- So if ‘px’ stores the address of x this allows us to:
  - access the value stored in ‘x’ through px indirectly

- It’s clear x is storing an int with value 7

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

What is a Pointer? (7/8)

- A **pointer** is a variable that stores the memory address of a specific object type
- So if ‘px’ stores the address of x this allows us to:
  - access the value stored in ‘x’ through px indirectly

- It’s clear x is storing an int with value 7

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

- Here we see ‘px’ again
- And we see an ‘asterisk’ before px
  - When an asterisk is before the variable name (and the type is a pointer), it means to retrieve the value at the address we point to.
    - This is called **dereferencing**

What is a Pointer? (8/8)

```
mike:pointers$ g++ -std=c++17 initialize.cpp -o prog
mike:pointers$ ./prog
x is: 7
*px is: 7
```

- A **pointer** is a variable that stores the memory address of a specific object type
- So if ‘px’ stores the address of x this allows us to:
    - access the value stored in ‘x’ through px indirectly

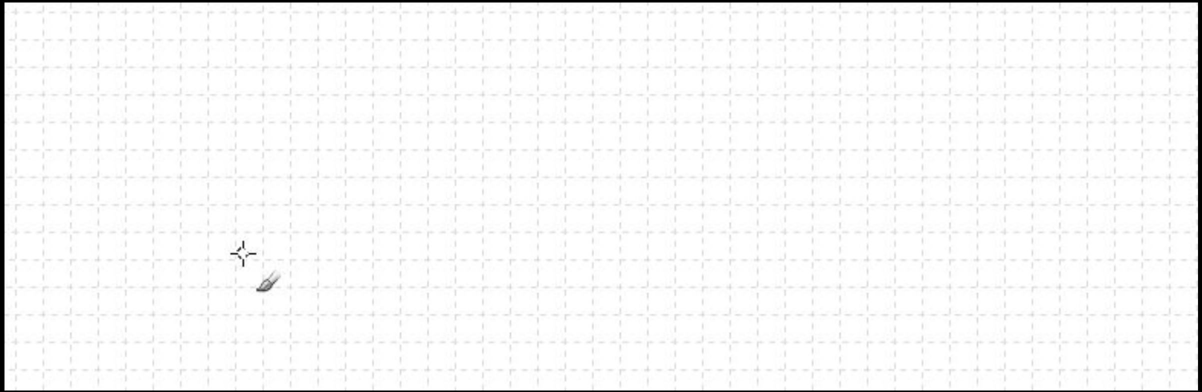
```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

- It’s clear x is storing an int with value 7

- Here we see ‘px’ again
- And we see an ‘asterisk’ before px
    - When an asterisk is before the variable name (and the type is a pointer), it means to retrieve the value at the address we point to.
        - This is called **dereferencing**

Visualizing Pointers and Memory

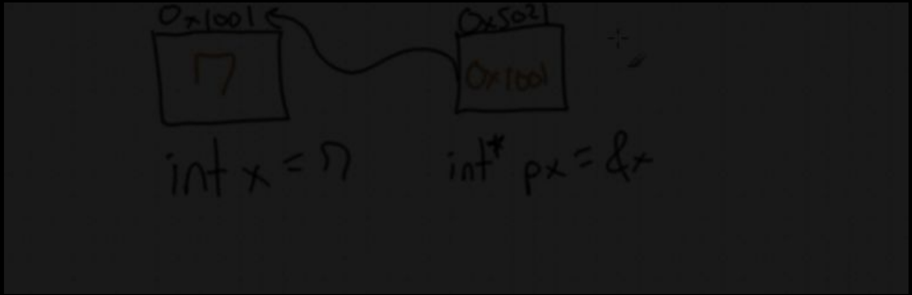
Let’s work on building our mental model when thinking about pointers

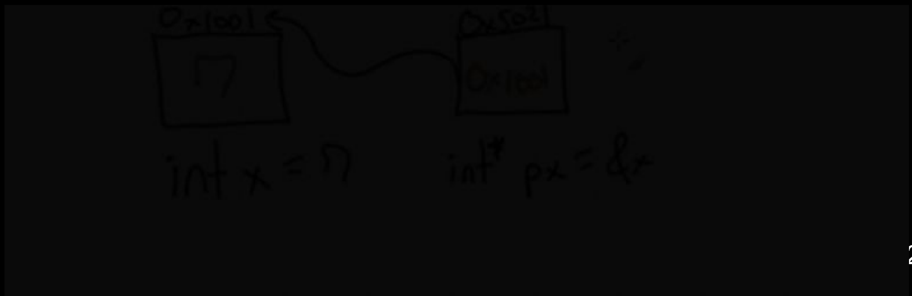


Visualizing our first program (1/5)

- When learning pointers, it's often useful to draw (on pen and paper) our memory
  - Let's represent our variables as boxes for now.

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```



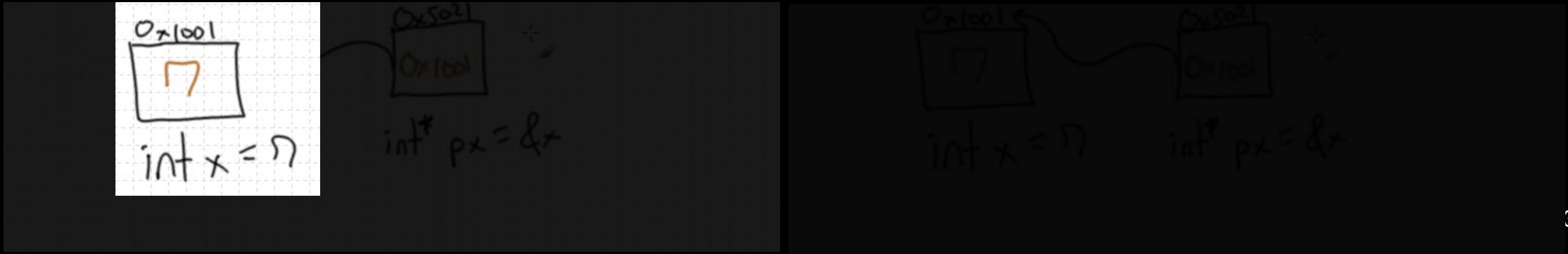


Visualizing our first program (2/5)

- When learning pointers, it's often useful to draw (on pen and paper) our memory
  - Let's represent our variables as boxes for now.

So every variable has some address (e.g., `0x1001`) and then at that address we can store some value (e.g., 7)

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

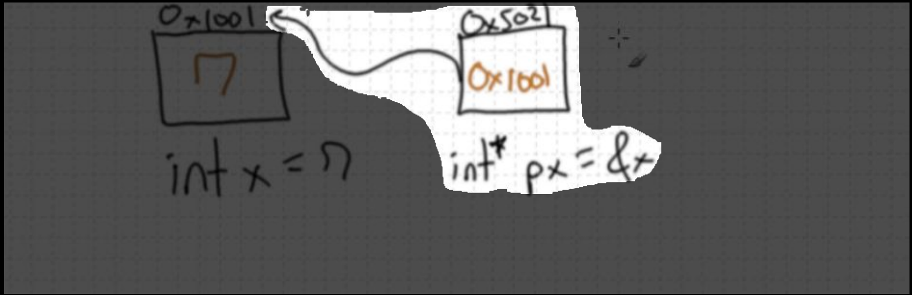


Visualizing our first program (3/5)

- When learning pointers, it’s often useful to draw (on pen and paper) our memory
  - Let’s represent our variables as boxes for now.

```
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::co   << "x is: " << x << std::endl;
    std::c    << "*px is: " << *px << std::endl;

    retu
}
```

So every variable has some address (e.g., `0x1001`) and then at that address we can store some value (e.g., 7)



`int*` (i.e., pointer to int) is no different than a variable and has an address (e.g., `0x5021`).

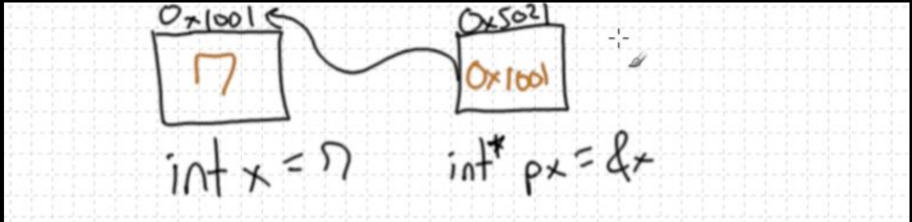
However, recall that pointers store an address as their value (so in this case, the address of x, which is at `0x1001`)

Visualizing our first program (4/5)

- When learning pointers, it's often useful to draw (on pen and paper) our memory
  - Let's represent our variables as boxes for now.

```
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::co      << "x is: " << x << std::endl;
    std::c       << "*px is: " << *px << std::endl;

    retu
}
```

So every variable has some address (e.g., `0x1001`) and then at that address we can store some value (e.g., 7)



So the major 'pro tip' is to try to draw your pointers

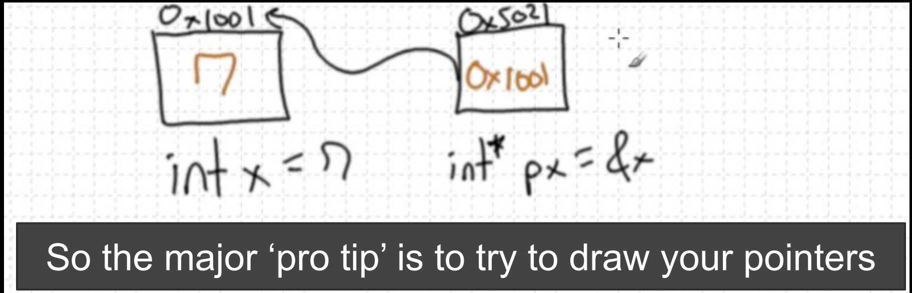
`int*` (i.e., pointer to int) is no different than a variable and has an address (e.g., `0x5021`).

However, recall that pointers store an address as their value (so in this case, the address of x, which is at `0x1001`)

Visualizing our first program (5/5)

- When learning pointers, it's often useful to draw (on pen and paper) our memory
  - Let's represent our variables as boxes for now.
  - Here's an animation of what is going on in memory when we assign a pointer to the address of a variable.

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```



Let’s visualize memory

So we can more concretely understand what a pointer is

Working with Pointers Means Thinking about Memory

- We have different layers and different types of memory locally available on our machines
  - Registers
  - Cache
  - DRAM (i.e., working memory)
  - Hard Drive(s)
  - (And even non-local memory like a Networked drive (or cloud storage))
- For this talk, let's assume all of our memory is in working memory (DRAM)
  - This means we have an array of randomly addressable memory where we can store data

Visualizing Memory - Linear array of addresses (1/11)

- Memory in our machines is represented as a linear array of addresses (And let's assume we have random access)
    - At each of these addresses we can store a value (i.e, some amount of bytes)
        - Depending on the data type, we use different amounts of memory.

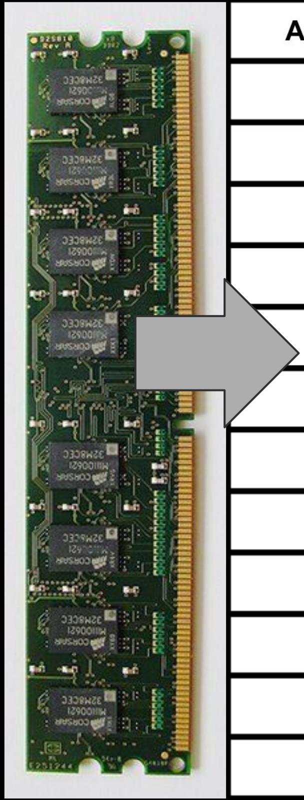
Visualizing Memory - Linear array of addresses (2/11)

- Memory in our machines is represented as a linear array of addresses (And let's assume we have random access)
  - At each of these addresses we can store a value (i.e, some amount of bytes)
    - Depending on the data type, we use different amounts of memory.

Address (in Hex)	Value
0x10000000	
0x10000001	
0x10000002	
0x10000003	
0x10000004	
0x10000005	
0x10000006	
0x10000007	
0x10000008	
0x10000009	
0x1000000A	
0x1000000B	

Visualizing Memory - Linear array of addresses (3/11)

- Memory in our machines is represented as a linear array of addresses (And let’s assume we have random access)
  - At each of these addresses we can store a value (i.e, some amount of bytes)
    - Depending on the data type, we use different amounts of memory.



Address (in Hex)	Value
0x10000000	
0x10000001	
0x10000002	
0x10000003	
0x10000004	
0x10000005	
0x10000006	
0x10000007	
0x10000008	
0x10000009	
0x1000000A	
0x1000000B	

Visualizing Memory - Linear array of addresses (4/11)

- Depending on the data type, we use different amounts of memory.

Address (in Hex)	Value
0x10000000	
0x10000001	
0x10000002	
0x10000003	
0x10000004	
0x10000005	
0x10000006	
0x10000007	
0x10000008	
0x10000009	
0x1000000A	
0x1000000B	

Visualizing Memory - Linear array of addresses (5/11)

- Depending on the data type, we use different amounts of memory.
    - `int x= 7;`

Address (in Hex)	Value
0x10000000	7
0x10000001	
0x10000002	
0x10000003	
0x10000004	
0x10000005	
0x10000006	
0x10000007	
0x10000008	
0x10000009	
0x1000000A	
0x1000000B	

Visualizing Memory - Linear array of addresses (6/11)

- Depending on the data type, we use different amounts of memory.
  - `int x= 7;`
  - Now, why did ‘x’ take 4 boxes?
    - On my architecture an integer takes up 4 bytes of memory

```
// @file sizeof.c
// Linked list
// g++ -Wall -    =c++17 sizeof.cpp -o prog
#incl

int m

    s                                    endl;

    return 0;
}
```

We can check using the sizeof operator to find the number of bytes to represent an object

```
mike:pointers$ g++ -Wall -Wextra -std=c++17 sizeof.cpp -o prog
mike:pointers$ ./prog
sizeof(int) 4
```

Address (in Hex)	Value
0x10000000	7
0x10000001	
0x10000002	
0x10000003	
0x10000004	
0x10000005	
0x10000006	
0x10000007	
0x10000008	
0x10000009	
0x1000000A	
0x1000000B	

Visualizing Memory - Linear array of addresses (7/11)

- Depending on the data type, we use different amounts of memory.
  - `int x= 7;`
  - Now, why did ‘x’ take 4 boxes?
    - On my architecture an integer takes up 4 bytes of memory

```cpp
// @file sizeof.cpp
// Linked list example
// g++ -Wall -Wextra -std=c++17 sizeof.cpp -o prog
#include <iostream>

int main(){

    std::cout << "sizeof(int) " << sizeof(int) << std::endl;

    return 0;
}
```

```
mike:pointers$ g++ -Wall -Wextra -std=c++17 sizeof.cpp -o prog
mike:pointers$ ./prog
sizeof(int) 4
```

Address (in Hex)	Value
0x10000000	7
0x10000001	
0x10000002	
0x10000003	
0x10000004	
0x10000005	
0x10000006	
0x10000007	
0x10000008	
0x10000009	
0x1000000A	
0x1000000B	

Visualizing Memory - Linear array of addresses (8/11)

- Okay, so what happens with our pointer?
  - `int x= 7;`
  - `int* px= &x;`

<table>
<tr><th>Address (in Hex)</th><th>Value</th></tr>
<tr><td>0x10000000</td><td rowspan="4">7</td></tr>
<tr><td>0x10000001</td></tr>
<tr><td>0x10000002</td></tr>
<tr><td>0x10000003</td></tr>
<tr><td>0x10000004</td><td></td></tr>
<tr><td>0x10000005</td><td></td></tr>
<tr><td>0x10000006</td><td></td></tr>
<tr><td>0x10000007</td><td></td></tr>
<tr><td>0x10000008</td><td></td></tr>
<tr><td>0x10000009</td><td></td></tr>
<tr><td>0x1000000A</td><td></td></tr>
<tr><td>0x1000000B</td><td></td></tr>
</table>

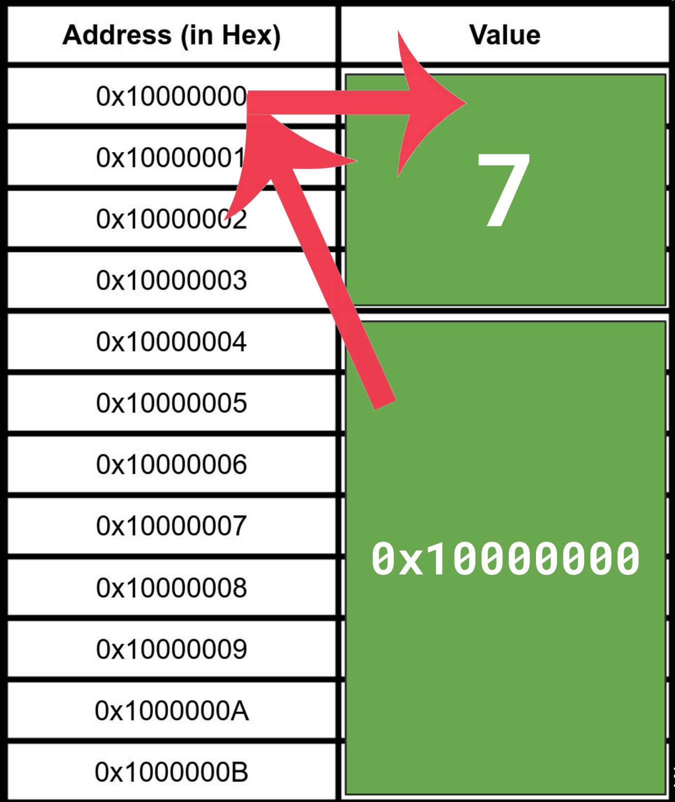
Visualizing Memory - Linear array of addresses (9/11)

- Okay, so what happens with our pointer?
  - `int x= 7;`
  - `int* px= &x;`

<table>
<tr><th>Address (in Hex)</th><th>Value</th></tr>
<tr><td>0x10000000</td><td rowspan="4">7</td></tr>
<tr><td>0x10000001</td></tr>
<tr><td>0x10000002</td></tr>
<tr><td>0x10000003</td></tr>
<tr><td>0x10000004</td><td rowspan="8">0x10000000</td></tr>
<tr><td>0x10000005</td></tr>
<tr><td>0x10000006</td></tr>
<tr><td>0x10000007</td></tr>
<tr><td>0x10000008</td></tr>
<tr><td>0x10000009</td></tr>
<tr><td>0x1000000A</td></tr>
<tr><td>0x1000000B</td></tr>
</table>

Visualizing Memory - Linear array of addresses (10/11)

- Okay, so what happens with our pointer?
  - `int x= 7;`
  - `int* px= &x;`
- Remember, px (*which is an integer pointer*), stores the address of the variable it points to
  - So we are indirectly accessing '7' at address `0x10000000` by doing `*px`.



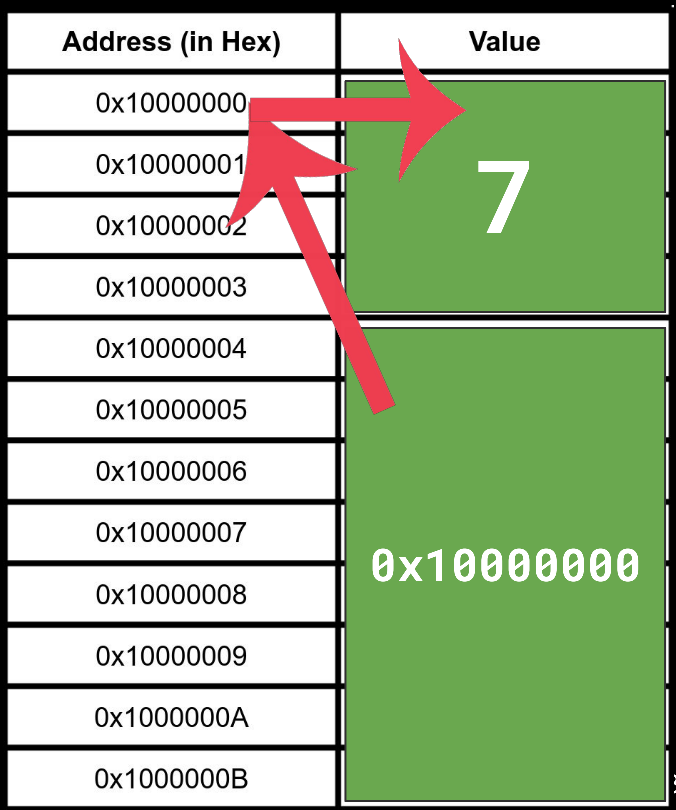
(Also note, a pointer takes 8 bytes on my 64-bit cpu -- try sizeof(int*))

Visualizing Memory - Linear array of addresses (11/11)

- Okay, so what happens with our pointer?
  - `int x= 7;`
  - `int* px= &x;`
- Remember, px (*which is an integer pointer*), stores the address of the variable it points to
  - So we are **indirectly accessing ‘7’** at address `0x10000000` by doing `*px`.

> Let’s take a moment to see how we access the value 7 from pointer px by **dereferencing** the pointer.

(Also note, a pointer takes 8 bytes on my 64-bit cpu -- try sizeof(int*))



Dereferencing a pointer

"Access the address stored in our pointer, and access the value *pointed to by our pointer*"

Dereferencing a pointer (Retrieving a value referred to by a pointer)

- We have previously observed dereferencing
  - We dereference by putting an asterisk(*) before our pointer variable (*px).
- In a plain sentence, dereferencing means:
  - ”If the type of my variable is a pointer, then if I dereference that variable, I will retrieve the value at the address at which I point to (i.e., retrieve the address stored in the pointer).”

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

Note on the asterisk (1/2)

- The asterisk (*) is used in two different contexts
  1. The first is part of the type when we create the pointer
     - Stylistically *I prefer* to put the asterisk next to the type name
     - Stylistically *I prefer* to prefix my pointers with 'p' or 'p_' to help me remember it is a pointer type

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

Note on the asterisk (2/2)

- The asterisk (*) is used in two different contexts
  1. The first is part of the type when we create the pointer
     - Stylistically *I prefer* to put the asterisk next to the type name
     - Stylistically *I prefer* to prefix my pointers with ‘p’ or ‘p_’ to help me remember it is a pointer type
  2. The second use of asterisk, is when dereferencing a pointer
     - This is what we just learned, when we want to retrieve a value.
     - The pointer already exists when we dereference.

```cpp
// @file initialize.cpp
// g++ -std=c++17 initialize.cpp -o prog
#include <iostream>

int main(){

    int x = 7;
    int* px = &x;

    std::cout << "x is: " << x << std::endl;
    std::cout << "*px is: " << *px << std::endl;

    return 0;
}
```

Pointers, &, and * (1/3)

- Here’s an example (to the right) to review what we’ve learned so far, specifically the C++ syntax

```cpp
// @file dereference.cpp
// g++ -std=c++17 dereference.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);
    // Print out for our understanding
    std::cout << "x value         : " <<  x  << std::endl;
    std::cout << "x address       : " << &x  << std::endl;
    std::cout << "px points to    : " << px  << std::endl;
    std::cout << "px dereferenced: " << *px << std::endl;

    return 0;
}
```

```
mike:pointers$ g++ -std=c++17 dereference.cpp -o prog
mike:pointers$ ./prog
x value         : 7
x address       : 0x7ffc2db2410c
px points to    : 0x7ffc2db2410c
px dereferenced: 7
```

Pointers, &, and * (2/3)

- Here's an example (to the right) to review what we've learned so far, specifically the C++ syntax

```cpp
// @file dereference.cpp
// g++ -std=c++17 dereference.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);
    // Print out for our understanding
    std::cout << "x value        : " <<  x  << std::endl;
    std::cout << "x address      : " << &x  << std::endl;
    std::cout << "px points to   : " << px  << std::endl;
    std::cout << "px dereferenced: " << *px << std::endl;

    return 0;
}
```

px points to x

```
mike:pointers$ g++ -std=c++17 dereference.cpp -o prog
mike:pointers$ ./prog
x value        : 7
x address      : 0x7ffc2db2410c
px points to   : 0x7ffc2db2410c
px dereferenced: 7
```

Pointers, &, and * (3/3)

- Here's an example (to the right) to review what we've learned so far, specifically the C++ syntax

```cpp
// @file dereference.cpp
// g++ -std=c++17 dereference.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);
    // Print out for our understanding
    std::cout << "x value         : " << x   << std::endl;
    std::cout << "x address       : " << &x  << std::endl;
    std::cout << "px points to    : " << px  << std::endl;
    std::cout << "px dereferenced: " << *px << std::endl;

    return 0;
}
```

px points to x

```
mike:pointers$ g++ -std=c++17 dereference.cpp -o prog
mike:pointers$ ./prog
x value        : 7
x address      : 0x7ffc2db2410c
px points to   : 0x7ffc2db2410c
px dereferenced: 7
```

dereferencing px retrieves us x's value

Dereferencing - Test Your Knowledge (1/3)

- What happens if I dereference px, and then change the value?
  - Make your predictions! (ans: next slide)

```cpp
// @file dereference2.cpp
// g++ -std=c++17 dereference2.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);
    // What happens if we dereference px
    // and change the value?
    *px = 42;
    std::cout << "x's value is: " << x << std::endl;

    return 0;
}
```

Dere

```
mike:pointers$ g++ -std=c++17 dereference2.cpp -o prog
mike:pointers$ ./prog
x's value is: 42
```

Here’s the result above

- What happens if I dereference px, and then change the value?
  - ans: The value changes!
  - The integer value in x changes because we store x’s address in px, and when we dereference, we follow the pointer to that value, and modify the value to 42

```cpp
// @file dereference2.cpp
// g++ -std=c++17 dereference2.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);
    // What happens if we dereference px
    // and change the value?
    *px = 42;
    std::cout << "x's value is: " << x << std::endl;

    return 0;
}
```

Dere

```
mike:pointers$ g++ -std=c++17 dereference2.cpp -o prog
mike:pointers$ ./prog
x's value is: 42
```

Here’s the result above

- Notice--dereferencing is happens first, before assignment
  - (I’ve wrapped the dereferencing portion in parenthesis, but this is not necessary as dereferencing has higher operator precedence than assignment)
  - https://en.cppreference.com/w/cpp/language/operator_precedence

```cpp
// @file dereference3.cpp
// g++ -std=c++17 dereference3.cpp -o prog
#include <iostream>

int main(){
	// Initialize integer
	int x= 7;
	// Pointer assigned to store address of x
	int* px= &(x);
	// What happens if we dereference px
	// and change the value?
	(*px) = 42;
	std::cout << "x's value is: " << x << std::endl;

	return 0;
}
```

(Now why does this work?) (1/2)

- So if you were comfortable with the last exercise--try to draw this one out
    - This time we have **p_px
    - This is a ‘pointer to an integer pointer’
        - Two levels of indirection, thus two *’s
            - Thus two *’s when we want to retrieve (dereference) a value that is two levels of indirection away.

```cpp
// @file pointerToPointer.cpp
// g++ -std=c++17 pointerToPointer.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);

    int** p_px = &px; // p_px is a pointer to an integer pointer

    // What happens if we dereference p_px
    // and change the value?
    **p_px = 77;

    // Follow two levels of indirection
    std::cout << "x's value is: " << x      << std::endl;
    std::cout << "*px       is: " << *px    << std::endl;
    std::cout << "**p_px    is: " << **p_px << std::endl << std::endl;
    // FYI (Here is one level of indirection)
    std::cout << "*p_px    is: " << *p_px << std::endl;
    std::cout << "&x       is: " << &x << std::endl;
    // FYI (Here is zero levels of indirection)
    std::cout << "p_px     is: " << p_px << std::endl;
    std::cout << "&px      is: " << &px << std::endl;

    return 0;
}
```

(Now why does this work?) (2/2)

```
x's value is: 77
*px          is: 77
**p_px       is: 77

*p_px      is: 0x7ffe67593cc4
&x         is: 0x7ffe67593cc4
p_px       is: 0x7ffe67593cc8
&px        is: 0x7ffe67593cc8
```

• Thus two *'s when we

And here's the result

indirection away.

```cpp
// @file pointerToPointer.cpp
// g++ -std=c++17 pointerToPointer.cpp -o prog
#include <iostream>

int main(){
    // Initialize integer
    int x= 7;
    // Pointer assigned to store address of x
    int* px= &(x);

    int** p_px = &px; // p_px is a pointer to an integer pointer

    // What happens if we dereference p_px
    // and change the value?
    **p_px = 77;

    // Follow two levels of indirection
    std::cout << "x's value is: " << x      << std::endl;
    std::cout << "*px         is: " << *px    << std::endl;
    std::cout << "**p_px      is: " << **p_px << std::endl << std::endl;
    // FYI (Here is one level of indirection)
    std::cout << "*p_px     is: " << *p_px << std::endl;
    std::cout << "&x        is: " << &x << std::endl;
    // FYI (Here is zero levels of indirection)
    std::cout << "p_px      is: " << p_px << std::endl;
    std::cout << "&px       is: " << &px << std::endl;

    return 0;
}
```

Why is dereferencing a big deal?

In this example I show how pointers allow us to share data, and from multiple locations (i.e., variables) we can retrieve the same value.

Pointers and sharing data (1/6)

I'm going to push the boundaries of what fits on a slide in a moment

- I'm allowed to do this in a technical presentation...if I explain the code :)
- (reminder full code examples are here: https://github.com/MikeShah/cppcon2021)

Pointers and sharing data (2/6)

- Three new data types, each holding 1 field
- Note:
  - `Company` and `Friends` have a 'pointer' in their field

```cpp
// @file sharing.cpp
// g++ -std=c++17 sharing.cpp -o prog
#include <iostream>
#include <string>
// Custom data structure
struct Person{
    std::string nickname;
    /* ... assume more attributes */
};
struct Company{
    Person* ceo; // The employees
};
struct Friends{
    Person* friend1; // Only 1 friend for simplicity...
};
```

1

2

3

```cpp
    // Let's confirm our pointers update
    std::cout << "MyFakeTwinBrother also is    : " << (*myFakeTwinBrother).nickname << std::endl;
    std::cout << "MyFakeTwinBrother is still   : " << myFakeTwinBrother->nickname << std::endl;
    // ^ Note the new syntax with the arrow, which derefences a field in a struct/class
    std::cout << "My employer should call me   : " << myEmployer.ceo->nickname << std::endl;
    std::cout << "My my friend should call me  : " << myFriends.friend1->nickname << std::endl;

    return 0;
}
```

Pointers and sharing data (3/6)

```cpp
    // Create 'me' with some attributes
    Person michael;
    michael.nickname = "Michael";

    Person* myFakeTwinBrother;
    Company myEmployer;
    Friends myFriends;
```

```cpp
// @file sharing.cpp
// g++ -std=c++17 sharing.cpp -o prog
#include <iostream>
#include <string>
// Custom data structure
struct Person{
    std::string nickname;
    /* ... assume more attributes */
};
struct Company{
    Person* ceo; // The employees
};
struct Friends{
    Person* friend1; // Only 1 friend for simplicity...
```

- First we create a few objects
    - The most important object is:
        - **`Person michael;`**
        - `michael` is a `Person` with a `nickname` member variable

Pointers and sharing data (4/6)

```cpp
// @file sharing.cpp
// g++ -std=c++17 sharing.cpp -o prog
#include <iostream>
#include <string>
// Custom data structure
struct Person{
    std::string nickname;
    /* ... assume more attributes */
};
struct Company{
    Person* ceo; // The employees
};
struct Friends{
    Person* friend1; // Only 1 friend for simplicity...
};
```

```cpp
int main(){
    // Create 'me' with some attributes
    Person michael;
    michael.nickname = "Michael";
    // ... Create other objects
    Person* myFakeTwinBrother;
    Company myEmployer;
    Friends myFriends;
    // For each of these other objects,
    // share some data
    myFakeTwinBrother = &michael;
    myEmployer.ceo = &michael;
    myFriends.friend1 = &michael;
    // Hmm, I've decided to change my ni
    michael.nickname = "Mike";
    // Let's confirm our pointers update

    return 0;
}
```

- Now, let’s initialize our pointers so that they point to the address of `michael` (i.e., `&michael`)
- If each pointer, points to the same thing, we are effectively sharing!

Pointers and sharing data (5/6)

```
// @file sharing.cpp
// g++ -std=c++17 sharing.cpp -o prog
#include <iostream>
#include <string>
// Custom data structure
struct Person{
    std::string nickname;

};
str

};
str


```

```
int main(){
    // Create 'me' with some attributes
    Person michael;
    michael.nickname = "Michael";
    // ... Create other objects
    Person* myFakeTwinBrother;
    Company myEmployer;
    Friends myFriends;
    // For each of these other objects,
    // share some data
    myFakeTwinBrother = &michael;
    myEmployer.ceo = &michael;
    myFriends.friend1 = &mi
    // Hmm, I've decided to change my nickname.
    michael.nickname = "Mike";
    // Let's confirm our pointers update
    std::cout << "MyFakeTwinBrother also is    : " << (*myFakeTwinBrother).nickname << std::endl;
    std::cout << "MyFakeTwinBrother is still   : " << myFakeTwinBrother->nickname << std::endl;
    // ^ Note the new syntax with the arrow, which derefences a field in a struct/class
    std::cout << "My employer should call me   : " << myEmployer.ceo->nickname << std::endl;
    std::cout << "My my friend should call me : " << myFriends.friend1->nickname << std::endl;

    return 0;
}
```

- So now if I update `michael.nickname`
- Anything that also points to michael will be updated!
    - Nice-- 1 write/update (in a sense) results in 3 updated values!

Pointers and sharing data (6/6)

```cpp
// @file sharing.cpp
// g++ -std=c++17 sharing.cpp -o prog
#include <iostream>
#include <string>
// Custom data structure
struct Person{
    std::string nickname;

};
str

};
str

}
```

```
MyFakeTwinBrother also is   : Mike
MyFakeTwinBrother is still  : Mike
My employer should call me  : Mike
My my friend should call me : Mike
```

```cpp
int main(){
    // Create 'me' with some attributes
    Person michael;
    michael.nickname = "Michael";
    // ... Create other objects
    Person* myFakeTwinBrother;
    Company myEmployer;
    Friends myFriends;
    // For each of these other objects,
    // share some data
    myFakeTwinBrother = &michael;
    myEmployer.ceo = &michael;
    myFriends.friend1 = &mi
    // Hmm, I've decided to change my nickname.
    michael.nickname = "Mike";
    // Let's confirm our pointers update
    std::cout << "MyFakeTwinBrother also is   : " << (*myFakeTwinBrother).nickname << std::endl;
    std::cout << "MyFakeTwinBrother is still  : " << myFakeTwinBrother->nickname << std::endl;
    // ^ Note the new syntax with the arrow, which derefences a field in a struct/class
    std::cout << "My employer should call me  : " << myEmployer.ceo->nickname << std::endl;
    std::cout << "My my friend should call me : " << myFriends.friend1->nickname << std::endl;

    return 0;
}
```

Pointers and sharing data (All the code on one slide)

```cpp
// @file sharing.cpp
// g++ -std=c++17 sharing.cpp -o prog
#include <iostream>
#include <string>
// Custom data structure
struct Person{
    std::string nickname;
    /* ... assume more attributes */
};
struct Company{
    Person* ceo; // The employees
};
struct Friends{
    Person* friend1; // Only 1 friend for simplicity...
};
int main(){
    // Create 'me' with some attributes
    Person michael;
    michael.nickname = "Michael";
    // ... Create other objects
    Person* myFakeTwinBrother;
    Company myEmployer;
    Friends myFriends;
    // For each of these other objects,
    // share some data
    myFakeTwinBrother = &michael;
    myEmployer.ceo = &michael;
    myFriends.friend1 = &michael;
    // Hmm, I've decided to change my nickname.
    michael.nickname = "Mike";
    // Let's confirm our pointers update
    std::cout << "MyFakeTwinBrother also is    : " << (*myFakeTwinBrother).nickname << std::endl;
    std::cout << "MyFakeTwinBrother is still   : " << myFakeTwinBrother->nickname << std::endl;
    // ^ Note the new syntax with the arrow, which derefences a field in a struct/class
    std::cout << "My employer should call me   : " << myEmployer.ceo->nickname << std::endl;
    std::cout << "My my friend should call me  : " << myFriends.friend1->nickname << std::endl;

    return 0;
}
```

Subtle Syntax | The arrow operator (->)

- Some folks may have noticed that there is a new ‘->’ syntax that was introduced on the previous slide
  - Recall when accessing the field of a struct we use the ‘.’ operator.
  - If that field is a pointer, and we’d like to dereference that field and get the value, we can do that with ->
    - This is a shorthand for using the * (to dereference) and the . (dot operator to retrieve a field).
    - Note how the two are exactly the same, but I find the ‘->’ much easier to use.

```
std::cout << "MyFakeTwinBrother also is   : " << (*myFakeTwinBrother).nickname << std::endl;
std::cout << "MyFakeTwinBrother is still  : " << myFakeTwinBrother->nickname << std::endl;
// ^ Note the new syntax with the arrow, which derefences a field in a struct/class
std::cout << "My employer should call me  : " << myEmployer.ceo->nickname << std::endl;
std::cout << "My my friend should call me : " << myFriends.friend1->nickname << std::endl;
```

So we have the basic tools of pointers

And now that we understand one use case of pointers is about ‘sharing’ data.

Let’s see how pointers work when passed to functions

Passing Pointers into Functions (Pass by Pointer) (1/6)

- Let’s compare two functions
  1. One that takes an integer parameter
  2. One that takes an integer pointer parameter

```cpp
// @file passByPointer.cpp
// g++ -std=c++17 passByPointer.cpp -o prog
#include <iostream>

void passByValue(int x){
    x = 9999;
}

void passByPointer(int* intPointer){
    *intPointer = 9999;
}

int main(){
    int x = 5;
    int y = 6;
    passByValue(x);
    std::cout << "x is now: " << x << std::endl;
    passByPointer(&y);
    std::cout << "y is now: " << y << std::endl;

    return 0;
}
```

Passing Pointers into Functions (Pass by Pointer) (2/6)

- Let’s compare two functions
  1. One that takes an integer parameter
  2. One that takes an integer pointer parameter
- Now you’ll notice these functions are named in particular way
  - pass-by-value and pass-by-pointer

```cpp
// @file passByPointer.cpp
// g++ -std=c++17 passByPointer.cpp -o prog
#include <iostream>

void passByValue(int x){
    x = 9999;
}

void passByPointer(int* intPointer){
    *intPointer = 9999;
}

int main(){
    int x = 5;
    int y = 6;
    passByValue(x);
    std::cout << "x is now: " << x << std::endl;
    passByPointer(&y);
    std::cout << "y is now: " << y << std::endl;

    return 0;
}
```

Passing Pointers into Functions (Pass by Pointer) (3/6)

- pass-by-value
  1. **pass-by-value**: means whenever we pass in a value, a copy of that value is made.
     - This means, the address of ‘x’ at line 14 is going to be different than at line 6

```
1 // @file passByPointer.cpp
2 // g++ -std=c++17 passByPointer.cpp -o prog
3 #include <iostream>
4
5 void passByValue(int x){
6     x = 9999;
7 }
8
9 void passByPointer(int* intPointer){
10     *intPointer = 9999;
11 }
12
13 int main(){
14     int x = 5;
15     int y = 6;
16  1  passByValue(x);
17     std::cout << "x is now: " << x << std::endl;
18     passByPointer(&y);
19     std::cout << "y is now: " << y << std::endl;
20
21     return 0;
22 }
```

Passing Pointers into Functions (Pass by Pointer) (4/6)

- pass-by-value **and pass-by-pointer**
  1. **pass-by-value**: means whenever we pass in a value, a copy of that value is made.
     - This means, the address of ‘x’ at line 14 is going to be different than at line 6
  2. **pass-by-pointer**: Well...it’s actually still pass-by-value.
     - However! The value that a pointer holds is an actual address of ‘y’ (located at line 15).
     - So sometimes we call this pass-by-pointer
       - (Note: If we re-assign our pointer in the function, that however will not be maintained)

```
 1 // @file passByPointer.cpp
 2 // g++ -std=c++17 passByPointer.cpp -o prog
 3 #include <iostream>
 4
 5 void passByValue(int x){
 6     x = 9999;
 7 }
 8
 9 void passByPointer(int* intPointer){
10     *intPointer = 9999;
11 }
12
13 int main(){
14     int x = 5;
15     int y = 6;
16  1  passByValue(x);
17     std::cout << "x is now: " << x << std::endl;
18  2  passByPointer(&y);
19     std::cout << "y is now: " << y << std::endl;
20
21     return 0;
22 }
```

- **Question to audience**: What do you predict the output of x and y are at lines 17 and 10?

- pass-by-value and pass-by-pointer
  1. **pass-by-value**: means whenever we pass in a value, a copy of that value is made.
     - This means, the address of ‘x’ at line 14 is going to be different than at line 6 (
  2. **pass-by-pointer**: Well...it’s actually still pass-by-value.
     - However! The value that a pointer holds is an actual address of ‘y’ (located at line 15).
     - So sometimes we call this pass-by-pointer
       - (Note: If we re-assign our pointer in the function, that however will not be maintained)

```
1  // @file passByPointer.cpp
2  // g++ -std=c++17 passByPointer.cpp -o prog
3  #include <iostream>
4
5  void passByValue(int x){
6      x = 9999;
7  }
8
9  void passByPointer(int* intPointer){
10     *intPointer = 9999;
11 }
12
13 int main(){
14     int x = 5;
15     int y = 6;
16     passByValue(x);
17     std::cout << "x is now: " << x << std::endl;
18     passByPointer(&y);
19     std::cout << "y is now: " << y << std::endl;
20
21     return 0;
22 }
```

```
mike:pointers$ g++ -std=c++17 passByPointer.cpp -o prog
mike:pointers$ ./prog
x is now: 5
y is now: 9999
```

- pass-by-value and pass-by-pointer
  1. **pass-by-value**: means whenever we pass in a value, a copy of that value is made.
     - This means, the address of ‘x’ at line 14 is going to be different than at line 6 (
  2. **pass-by-pointer**: Well…it’s actually still pass-by-value.
     - However! The value that a pointer holds is an actual address of ‘y’ (located at line 15).
     - So sometimes we call this pass-by-pointer
       - (Note: If we re-assign our pointer in the function, that however will not be maintained)

```cpp
1 // @file passByPointer.cpp
2 // g++ -std=c++17 passByPointer.cpp -o prog
3 #include <iostream>
4
5 void passByValue(int x){
6     x = 9999;
7 }
8
9 void passByPointer(int* intPointer){
10    *intPointer = 9999;
11 }
12
13 int main(){
14    int x = 5;
15    int y = 6;
16    passByValue(x);
17    std::cout << "x is now: " << x << std::endl;
18    passByPointer(&y);
19    std::cout << "y is now: " << y << std::endl;
20
21    return 0;
22 }
```

Notes: pass-by-pointer (1/3)

- Again, pass-by-pointer is equivalent to pass-by-value, except you are able to mutate values through the address a pointer stores
  - (Teacher note: I like giving this a different name so it's clearer for students to communicate what they are doing)

```cpp
// @file passByPointerExcessively.cpp
// g++ -std=c++17 passByPointerExcessively.cpp -o prog
#include <iostream>

// Silly example that adds 3 input parameters and
// sets it as an output.
// The 3 inputs are consumed and set to 0.
void passByPointerExcessively(int* out, int* in1, int* in2, int* in3){
    *out = *in1 + *in2 + *in3;
    *in1 = 0;
    *in2 = 0;
    *in3 = 0;
}

int main(){
    int x = 5;    int y = 6;    int z = 7;
    int out;

    passByPointerExcessively(&out,&x,&y,&z);
    std::cout << "out is  : " << out << std::endl;
    std::cout << "x is now: " << x << std::endl;
    std::cout << "y is now: " << y << std::endl;
    std::cout << "z is now: " << z << std::endl;

    return 0;
}
```

Notes: pass-by-pointer (2/3)

- You’ll see pass-by-pointer this in C-style APIs (e.g., OpenGL)
  - Careful though, excessive use, or specifying ‘out’ parameters is an identified code smell from Jason Turners talk! [6 min mark]
  - Also be careful, if I reassign a pointer within a function, remember I’m only re-assigning the copy of the pointer (because, we are passing by value the pointer) variable.

```cpp
// @file passByPointerExcessively.cpp
// g++ -std=c++17 passByPointerExcessively.cpp -o prog
#include <iostream>

// Silly example that adds 3 input parameters and
// sets it as an output.
// The 3 inputs are consumed and set to 0.
void passByPointerExcessively(int* out, int* in1, int* in2, int* in3){
    *out = *in1 + *in2 + *in3;
    *in1 = 0;
    *in2 = 0;
    *in3 = 0;
}

int main(){
    int x = 5;    int y = 6;    int z = 7;
    int out;

    passByPointerExcessively(&out,&x,&y,&z);
    std::cout << "out is  : " << out << std::endl;
    std::cout << "x is now: " << x << std::endl;
    std::cout << "y is now: " << y << std::endl;
    std::cout << "z is now: " << z << std::endl;

    return 0;
}
```

Notes: pass-by-pointer (3/3)

- You’ll see pass-by-pointer this in C-style APIs (e.g., OpenGL)
    - Careful though, ex use, or specifying parameters is an i code smell from Ja Turners talk! [6 min mark]
    - Also be careful, if I reassign a pointer within a function, remember I’m only re-assigning the copy of the pointer (because, we are passing by value the pointer) variable.

```
// @file passByPointerExcessively.cpp
// g++ -std=c++17 passByPointerExcessively.cpp -o prog
#include <iostream>

// Silly example that adds 3 input parameters and
// sets it as an output
set to 0.
out, int* in1, int* in2, int* in3){

int main(){
    int x = 5;    int y = 6;    int z = 7;
    int out;

    passByPointerExcessively(&out,&x,&y,&z);
    std::cout << "out is  : " << out << std::endl;
    std::cout << "x is now: " << x << std::endl;
    std::cout << "y is now: " << y << std::endl;
    std::cout << "z is now: " << z << std::endl;

    return 0;
}
```

Note: My C++ folks, do not worry, I’ll mention pass-by-reference

Pointer Variables...check

Dereferencing...check

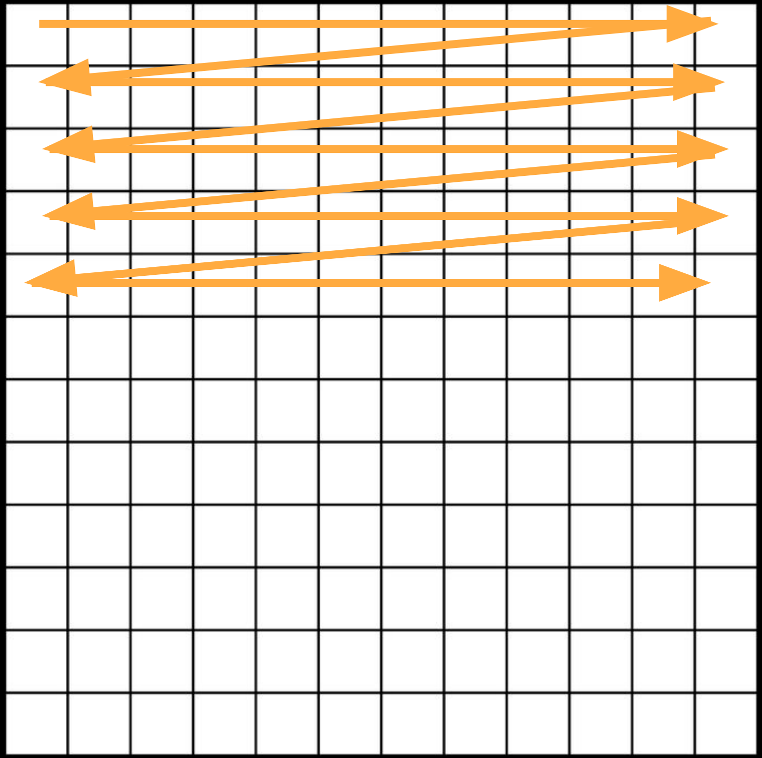
Pointers as parameters...check

Let’s now talk about pointers versus arrays (and dynamically allocated arrays)

Pointers and Arrays

Visualizing Memory - Linear array of addresses (1/4)

- Let's again visualize memory
- This time I'm going to show a grid
  - Still a linear array of addresses as indicated by the arrows.
  - I am using a grid so I can fit more memory on the screen.

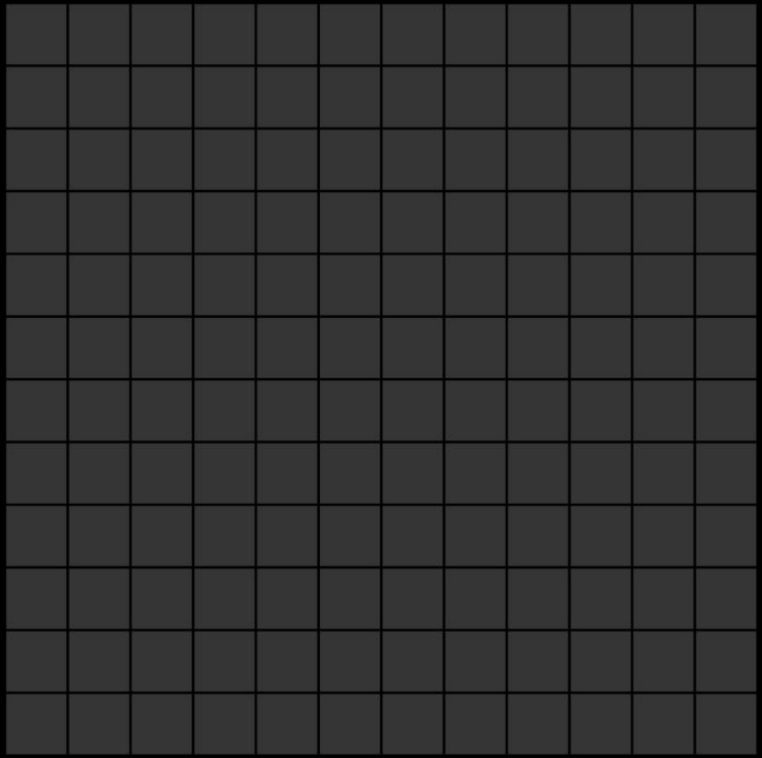


*Assume 1 byte per box and assume still a linear array of memory

Visualizing Memory - Linear array of addresses (2/4)

- Different data types (whether primitive types or user-defined types) will take different amounts of memory

```
// @file sizeof3.cpp
// g++ -Wall -Wextra -std=c++17 sizeof3.cpp -o prog
#include <iostream>
#include <cstdint>   // Fixed width integer types (c++11)

struct UserDefinedType{
    int x,y,z;  // 12 bytes
    char a,b,c; // 3 more bytes
    // +1 more bytes for padding
    // (Don't assume!)
};

int main(){

    std::cout << "sizeof(bool)           :" << sizeof(bool) << std::endl;
    std::cout << "sizeof(unsigned char)  :" << sizeof(unsigned char) << std::endl;
    std::cout << "sizeof(char)           :" << sizeof(char) << std::endl;
    std::cout << "sizeof(short)          :" << sizeof(short) << std::endl;
    std::cout << "sizeof(uint8_t)        :" << sizeof(uint8_t) << std::endl;
    std::cout << "sizeof(int)            :" << sizeof(int) << std::endl;
    std::cout << "sizeof(float)          :" << sizeof(float) << std::endl;
    std::cout << "sizeof(double)         :" << sizeof(double) << std::endl;
    std::cout << "sizeof(UserDefinedType):" << sizeof(UserDefinedType) << std::endl;

    return 0;
}
```

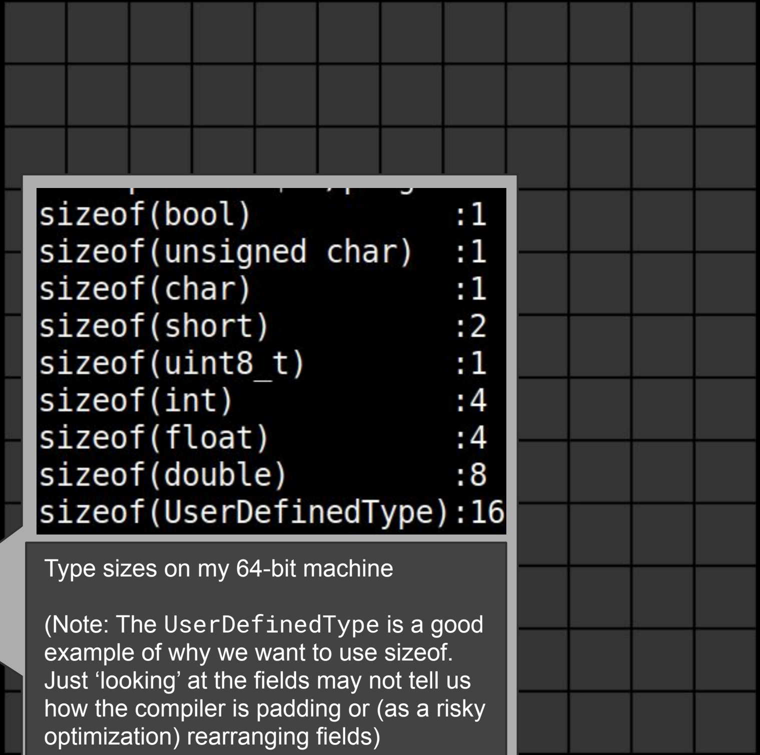


*Assume 1 byte per box and assume still a linear array of memory

Visualizing Memory - Linear array of addresses (3/4)

- Different data types (whether primitive types or user-defined types) will take different amounts of memory

```cpp
// @file sizeof3.cpp
// g++ -Wall -Wextra -std=c++17 sizeof3.cpp -o prog
#include <iostream>
#include <cstdint>   // Fixed width integer types (c++11)

struct UserDefinedType{
    int x,y,z;  // 12 bytes
    char a,b,c; // 3 more bytes
    // +1 more bytes for padding
    // (Don't assume!)
};

int main(){

    std::cout << "sizeof(bool)            :" << sizeof(bool) << std::endl;
    std::cout << "sizeof(unsigned char)   :" << sizeof(unsigned char) << std::endl;
    std::cout << "sizeof(char)            :" << sizeof(char) << std::endl;
    std::cout << "sizeof(short)           :" << sizeof(short) << std::endl;
    std::cout << "sizeof(uint8_t)         :" << sizeof(uint8_t) << std::endl;
    std::cout << "sizeof(int)             :" << sizeof(int) << std::endl;
    std::cout << "sizeof(float)           :" << sizeof(float) << std::endl;
    std::cout << "sizeof(double)          :" << sizeof(double) << std::endl;
    std::cout << "sizeof(UserDefinedType):" << sizeof(UserDefinedType) << std::endl;

    return 0;
}
```



```
sizeof(bool)           :1
sizeof(unsigned char)  :1
sizeof(char)           :1
sizeof(short)          :2
sizeof(uint8_t)        :1
sizeof(int)            :4
sizeof(float)          :4
sizeof(double)         :8
sizeof(UserDefinedType):16
```

Type sizes on my 64-bit machine

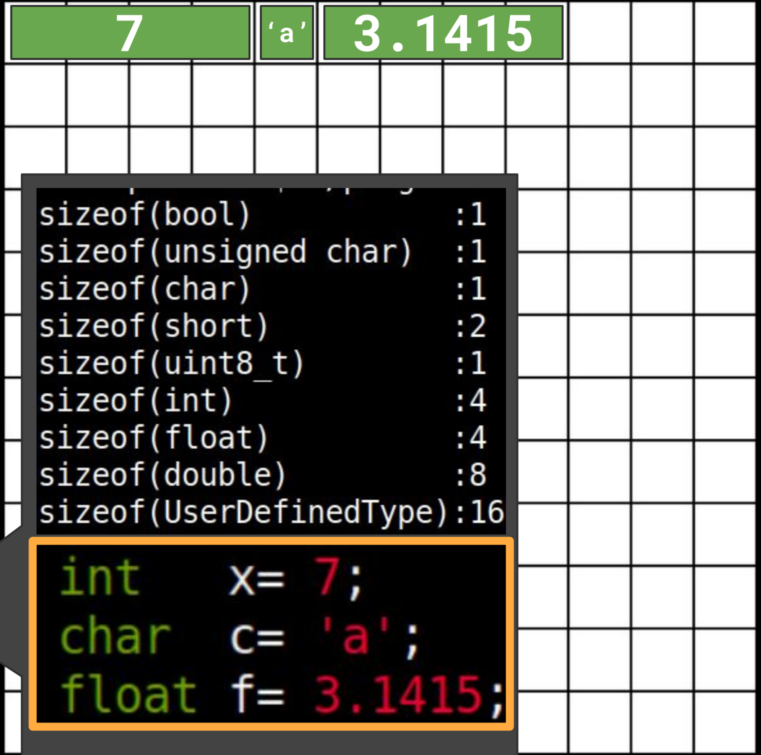
(Note: The `UserDefinedType` is a good example of why we want to use sizeof. Just 'looking' at the fields may not tell us how the compiler is padding or (as a risky optimization) rearranging fields)

*Assume 1 byte per box

Visualizing Memory - Linear array of addresses (4/4)

- Here are a few examples of how our memory may fill up
  - (All local variables for now, that are stack allocated)

```cpp
// @file sizeof3.cpp
// g++ -Wall -Wextra -std=c++17 sizeof3.cpp -o prog
#include <iostream>
#include <cstdint>   // Fixed width integer types (c++11)

struct UserDefinedType{
    int x,y,z;  // 12 bytes
    char a,b,c; // 3 more bytes
    // +1 more bytes for padding
    // (Don't assume!)
};

int main(){

    std::cout << "sizeof(bool)            :" << sizeof(bool) << std::endl;
    std::cout << "sizeof(unsigned char)   :" << sizeof(unsigned char) << std::endl;
    std::cout << "sizeof(char)            :" << sizeof(char) << std::endl;
    std::cout << "sizeof(short)           :" << sizeof(short) << std::endl;
    std::cout << "sizeof(uint8_t)         :" << sizeof(uint8_t) << std::endl;
    std::cout << "sizeof(int)             :" << sizeof(int) << std::endl;
    std::cout << "sizeof(float)           :" << sizeof(float) << std::endl;
    std::cout << "sizeof(double)          :" << sizeof(double) << std::endl;
    std::cout << "sizeof(UserDefinedType):" << sizeof(UserDefinedType) << std::endl;

    return 0;
}
```




```
sizeof(bool)           :1
sizeof(unsigned char)  :1
sizeof(char)           :1
sizeof(short)          :2
sizeof(uint8_t)        :1
sizeof(int)            :4
sizeof(float)          :4
sizeof(double)         :8
sizeof(UserDefinedType):16
```

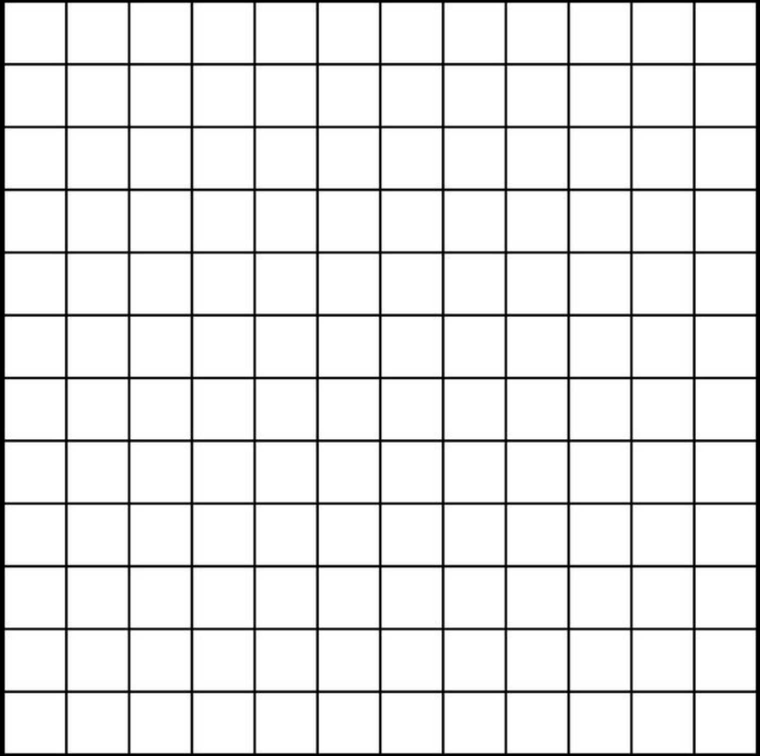
```cpp
int   x= 7;
char  c= 'a';
float f= 3.1415;
```

*Assume 1 byte per box

Visualizing Arrays (1/8)

- Okay, so what happens when we create an array of data?

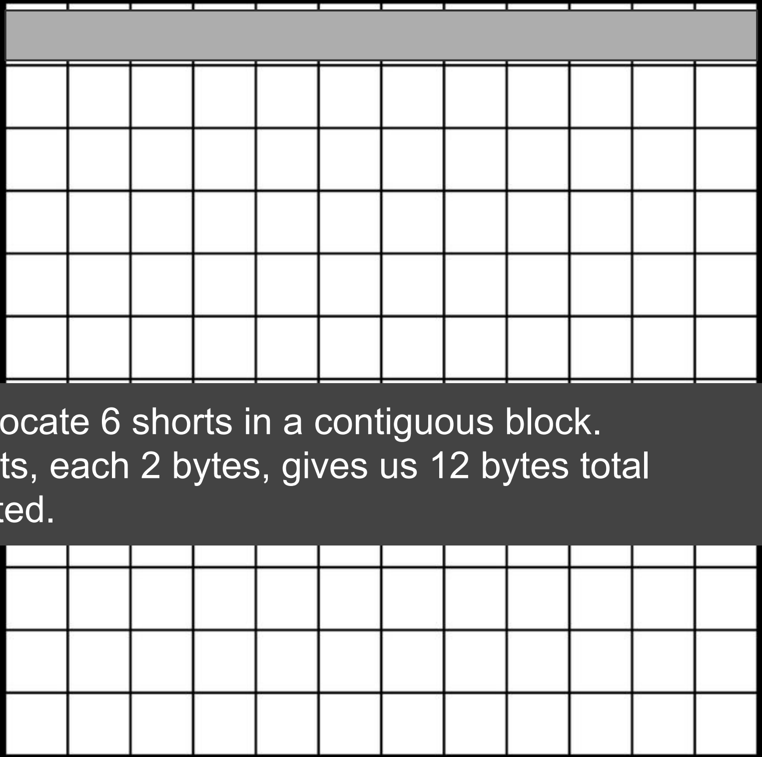
```
// @file array.cpp
// g++ -std=c++17 array.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Note: Here's a modern C++
    //       std::array container
    // std::array<short,6> test;
    // test.fill(1);
    return 0;
}
```



Visualizing Arrays (2/8)

- Okay, so what happens when we create an array of data?

```cpp
// @file array.cpp
// g++ -std=c++17 array.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Note: Here's a modern C++
    //       std::array container
    // std::array<short,6> test;
    // test.fill(1);
    return 0;
}
```



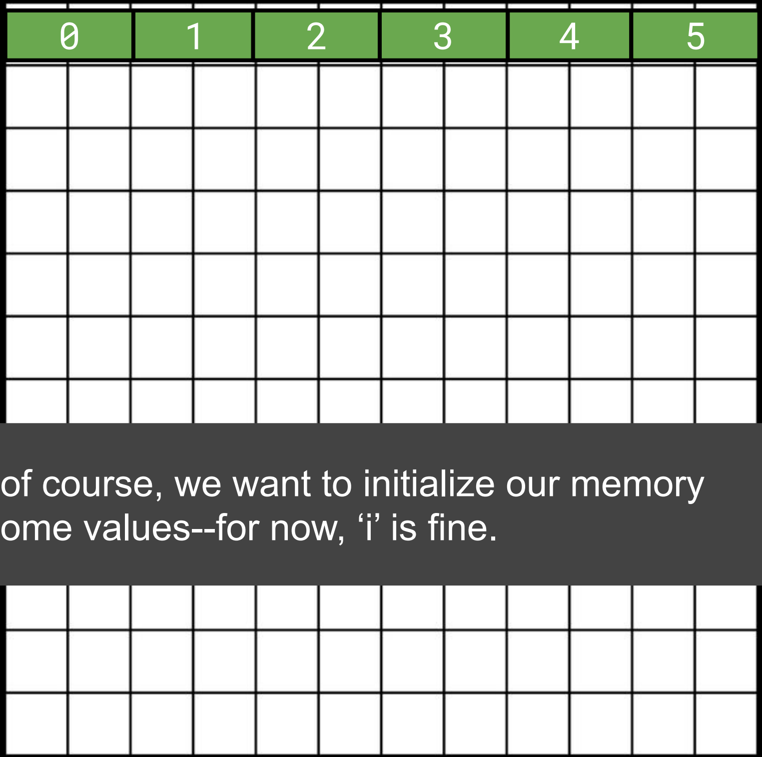
We allocate 6 shorts in a contiguous block.
6 shorts, each 2 bytes, gives us 12 bytes total allocated.

Visualizing Arrays (3/8)

- Okay, so what happens when we create an array of data?

```cpp
// @file array.cpp
// g++ -std=c++17 array.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Note: Here's a modern C++
    //        std::array container
    // std::array<short,6> test;
    // test.fill(1);
    return 0;
}
```

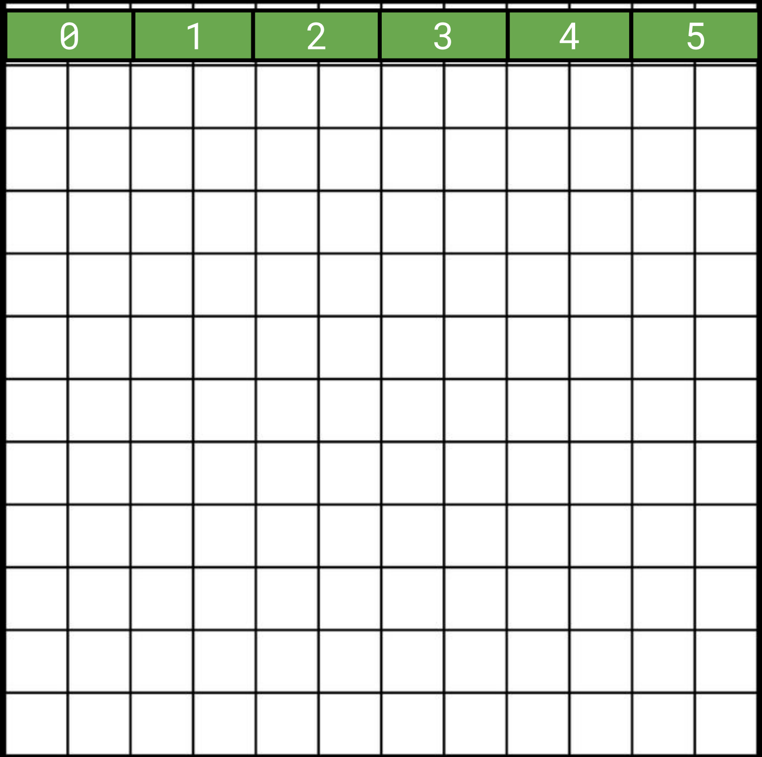
Then of course, we want to initialize our memory with some values--for now, ‘i’ is fine.



Visualizing Arrays (4/8)

- So if I create a `short* p_s`, based off what we learned, I should be able to point to each individual element.

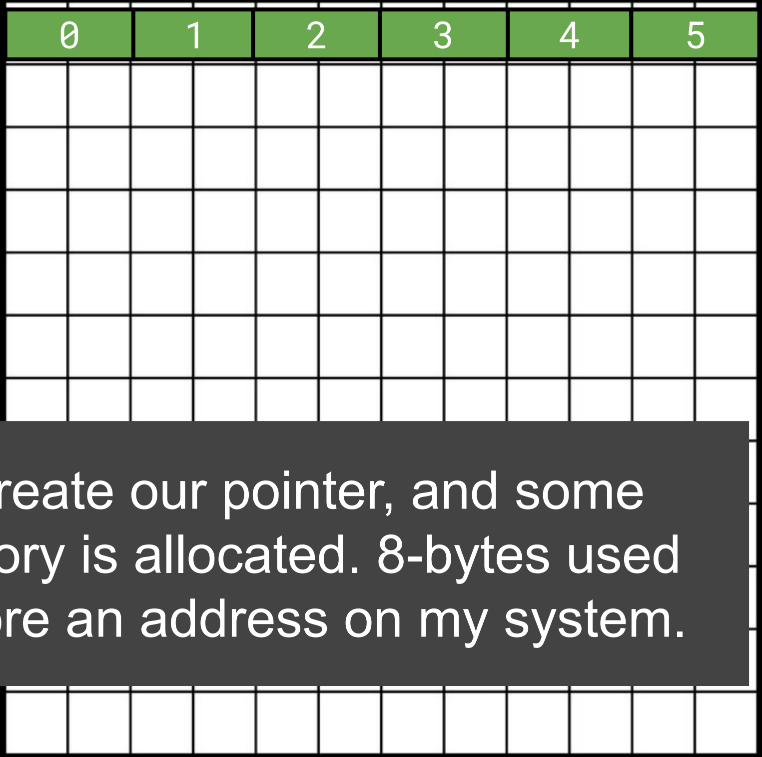
```cpp
// @file array2.cpp
// g++ -std=c++17 array2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to element in array
    short* p_s = &array[2];
    std::cout << "&array[2]:" << *p_s << std::endl;
    // Point to another element in array
    p_s = &array[3];
    std::cout << "&array[3]:" << *p_s << std::endl;

    return 0;
}
```



Visualizing Arrays (5/8)

- So if I create a `short* p_s`, based off what we learned, I should be able to point to each individual element.

```
// @file array2.cpp
// g++ -std=c++17 array2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to element in array
    short* p_s = &array[2];
    std::cout << "&array[2]:" << *p_s << std::endl;
    // Point to another element in array
    p_s = &array[3];
    std::cout << "&array[3]:" << *p_s << std::endl;

    return 0;
}
```

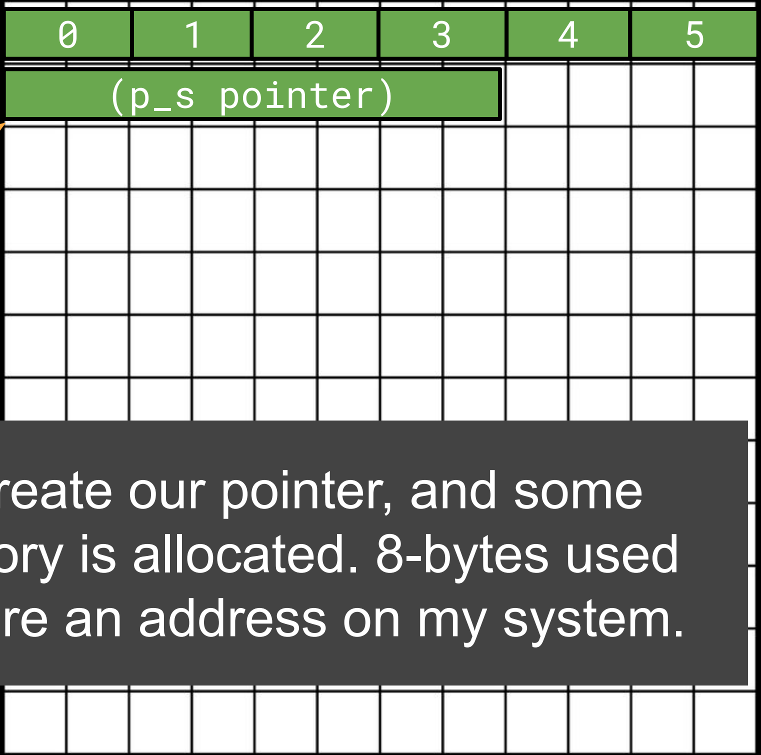


We create our pointer, and some memory is allocated. 8-bytes used to store an address on my system.

Visualizing Arrays (6/8)

- So if I create a `short* p_s`, based off what we learned, I should be able to point to each individual element.

```cpp
// @file array2.cpp
// g++ -std=c++17 array2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to element in array
    short* p_s = &array[2];
    std::cout << "&array[2]:" << *p_s << std::
    // Point to another element in array
    p_s = &array[3];
    std::cout << "&array[3]:" << *p_s << std::endl;

    return 0;
}
```

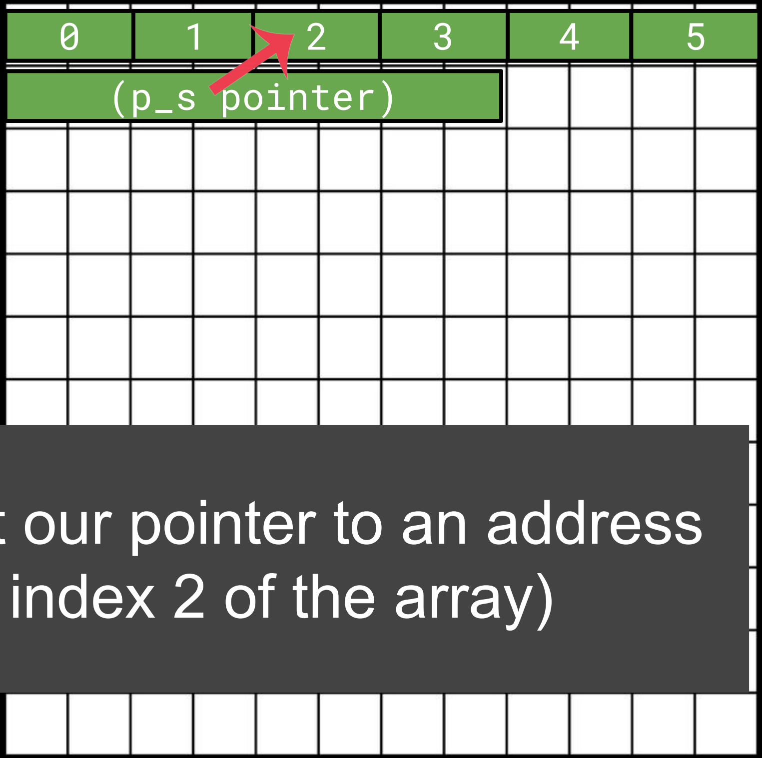


We create our pointer, and some memory is allocated. 8-bytes used to store an address on my system.

Visualizing Arrays (7/8)

- So if I create a `short* p_s`, based off what we learned, I should be able to point to each individual element.

```
// @file array2.cpp
// g++ -std=c++17 array2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to element in array
    short* p_s = &array[2];
    std::cout << "&array[2]:" << *p_s << std::
    // Point to another element in array
    p_s = &array[3];
    std::cout << "&array[3]:" << *p_s << std::endl;

    return 0;
}
```

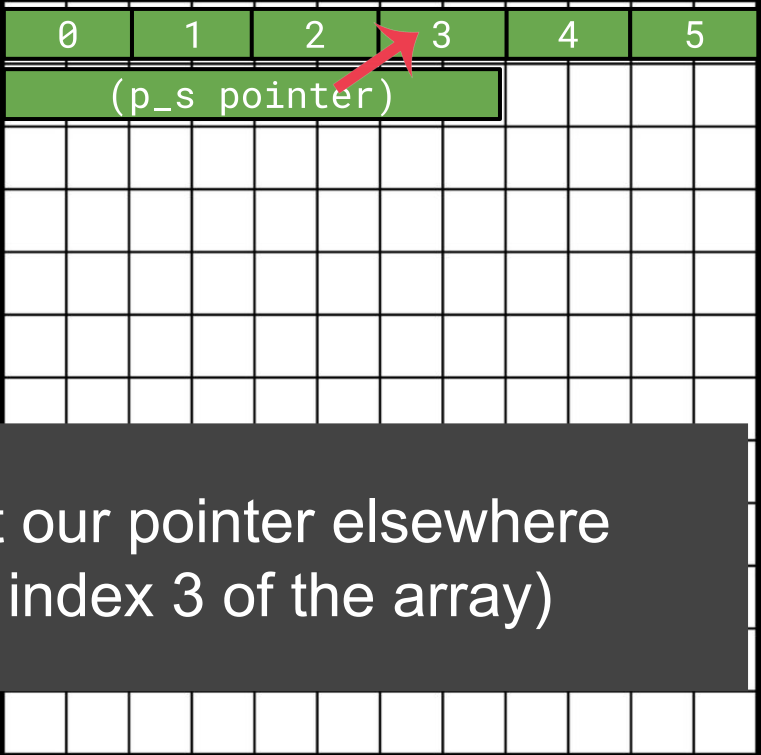


point our pointer to an address (i.e., index 2 of the array)

Visualizing Arrays (8/8)

- So if I create a `short* p_s`, based off what we learned, I should be able to point to each individual element.

```
// @file array2.cpp
// g++ -std=c++17 array2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to element in array
    short* p_s = &array[2];
    std::cout << "&array[2]:" << *p_s << s
    // Point to another element in array
    p_s = &array[3];
    std::cout << "&array[3]:" << *p_s << std::endl;

    return 0;
}
```

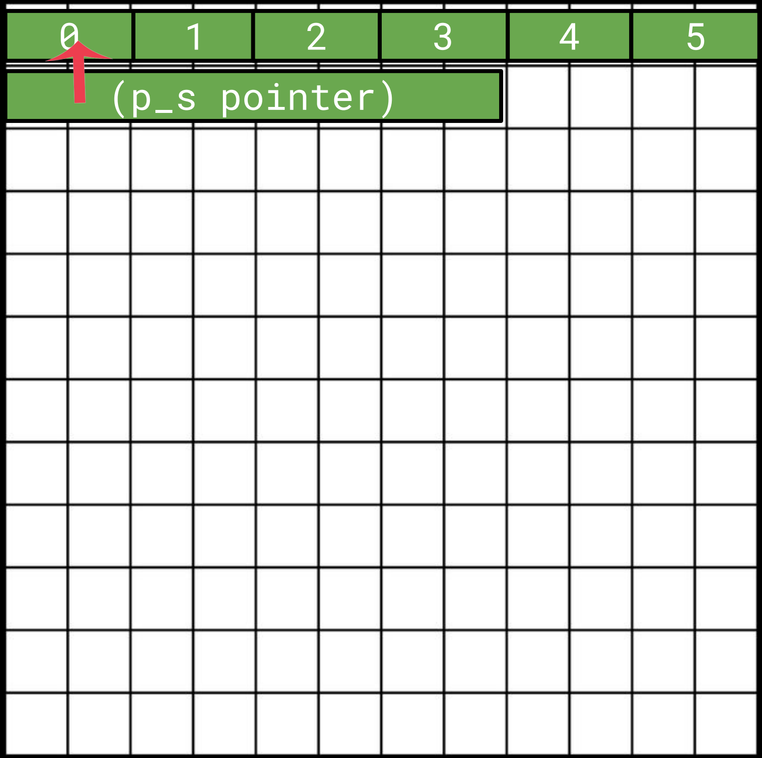


point our pointer elsewhere (i.e., index 3 of the array)

Pointer arithmetic (1/7)

- What happens if I try to ‘increment’ a pointer?
  - Well--we can do p_s++ or ++p_s

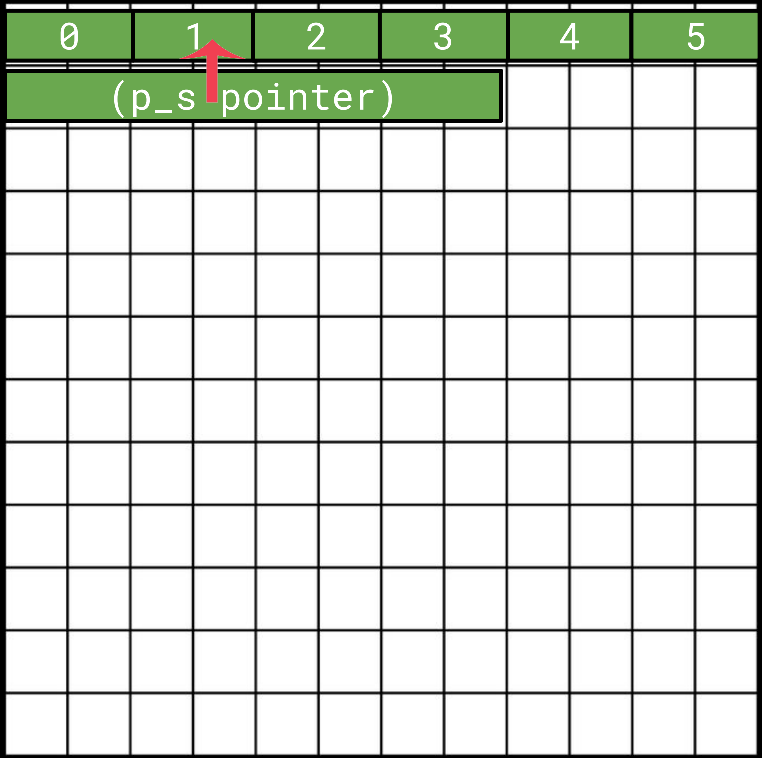
```cpp
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```



Pointer arithmetic (2/7)

- What happens if I try to ‘increment’ a pointer?
  - Well--we can do p_s++ or ++p_s

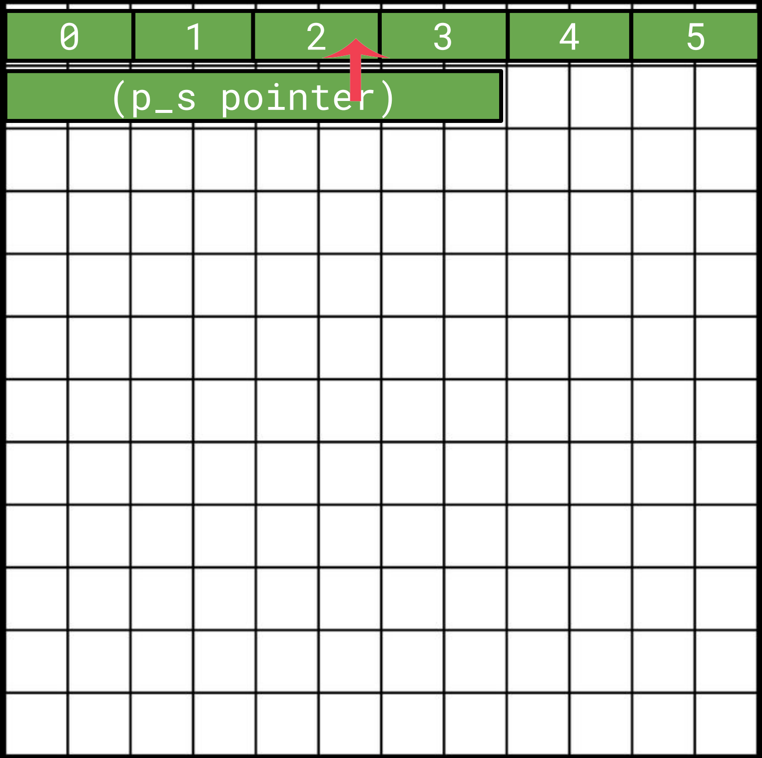
```cpp
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```



Pointer arithmetic (3/7)

- What happens if I try to ‘increment’ a pointer?
  - Well--we can do p_s++ or ++p_s

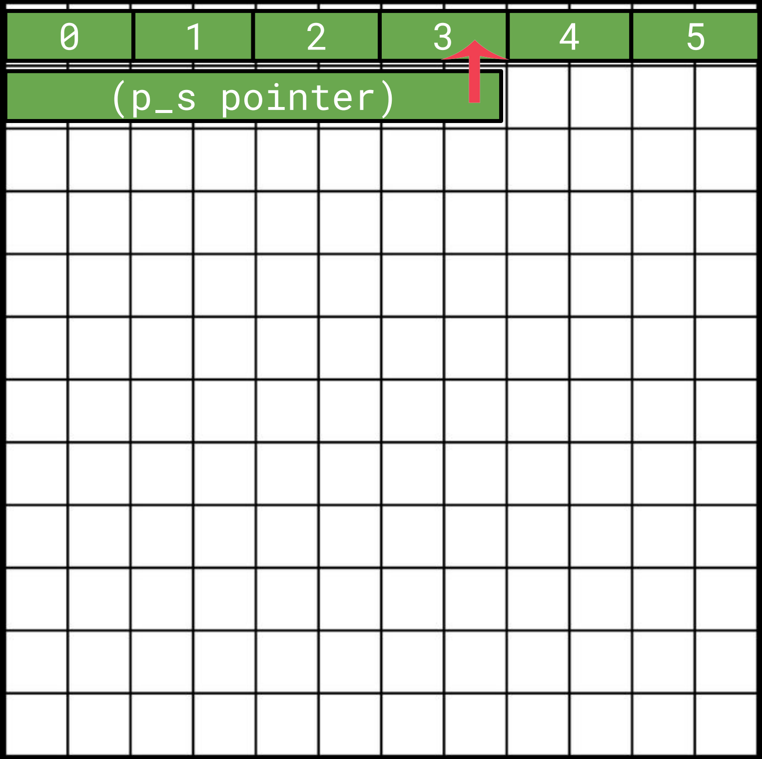
```
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```



Pointer arithmetic (4/7)

- What happens if I try to 'increment' a pointer?
  - Well--we can do p_s++ or ++p_s

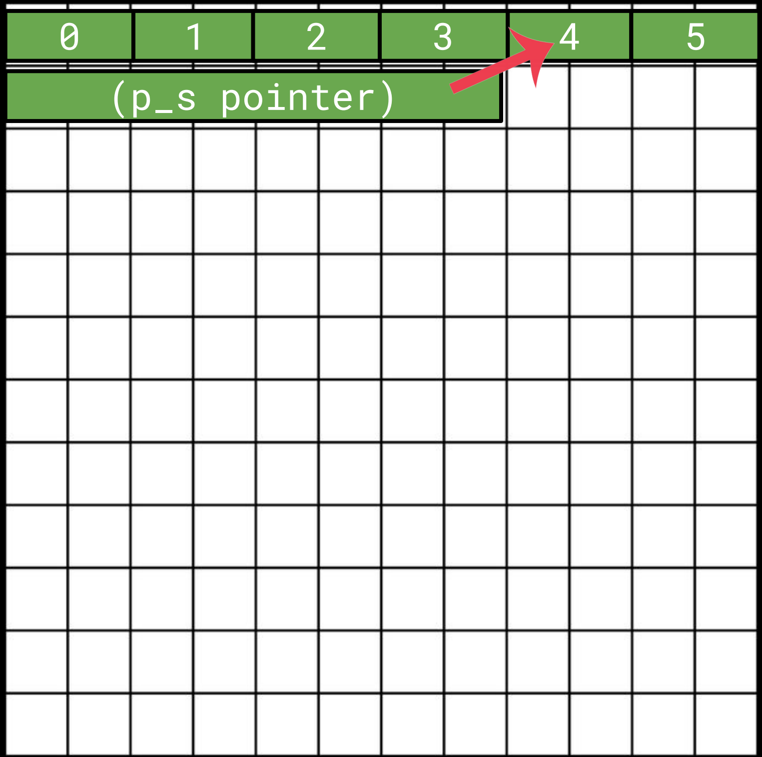
```cpp
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```



Pointer arithmetic (5/7)

- What happens if I try to 'increment' a pointer?
  - Well--we can do p_s++ or ++p_s

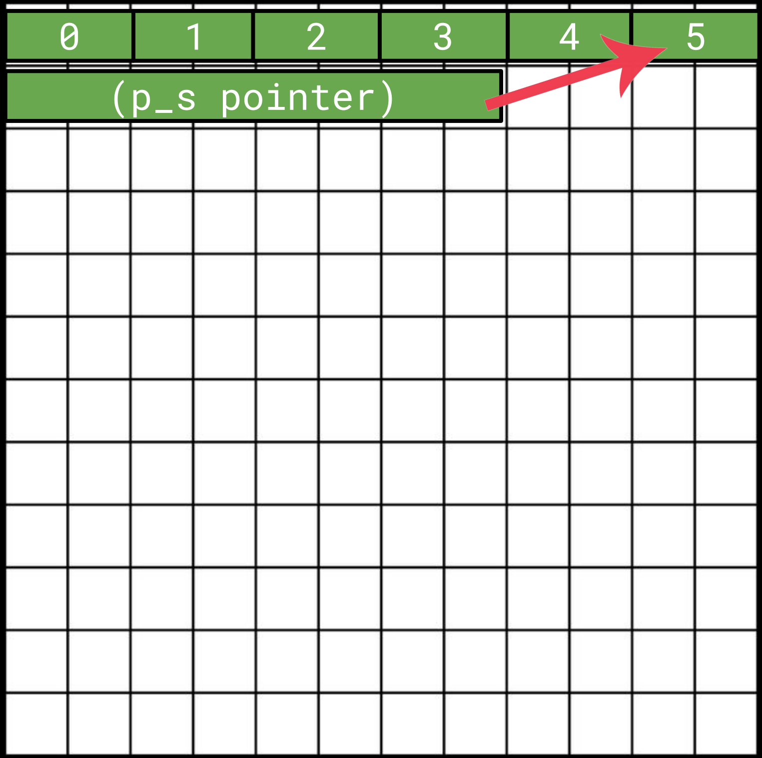
```
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```



Pointer arithmetic (6/7)

- What happens if I try to 'increment' a pointer?
  - Well--we can do p_s++ or ++p_s

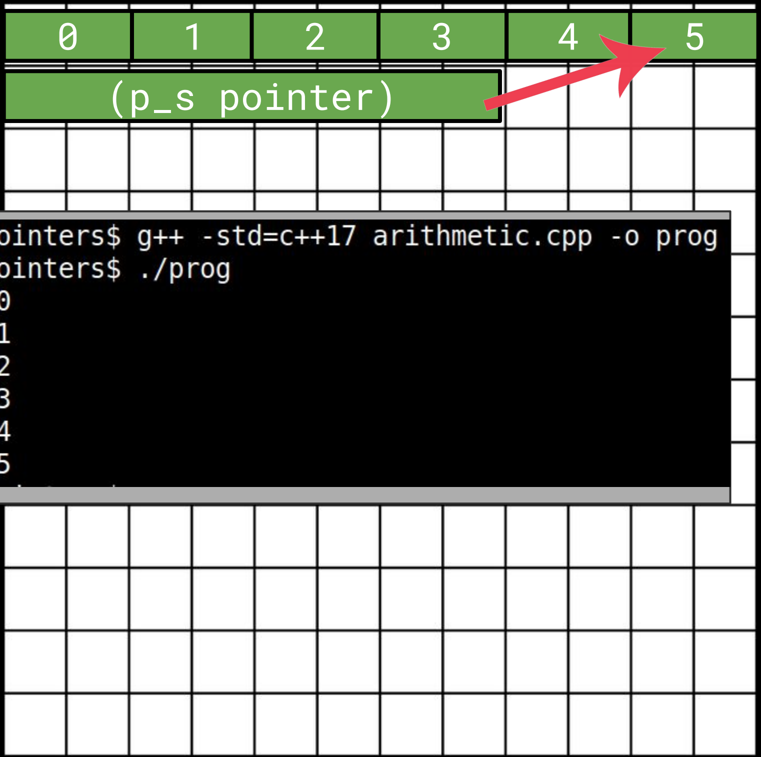
```
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```



Pointer arithmetic (7/7)

- Because our pointer type(p_s) is '2 bytes', ++ (post-increment) shifts our pointer 2 bytes when we add.

```cpp
// @file arithmetic.cpp
// g++ -std=c++17 arithmetic.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    short* p_s = &array[0];
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```




```
mike:pointers$ g++ -std=c++17 arithmetic.cpp -o prog
mike:pointers$ ./prog
*p_s= 0
*p_s= 1
*p_s= 2
*p_s= 3
*p_s= 4
*p_s= 5
```

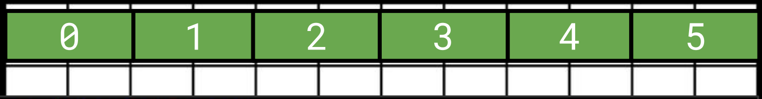
Array offset and dereference (1/3)

- So if we think about our previous example:
  - The number of times we increment p_s, was the offset into the array
  - We can access a value by offsetting to a position, and then dereferencing that address!
    - (See example on the right)


```cpp
// @file arithmetic2.cpp
// g++ -std=c++17 arithmetic2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Array offset shorthand
    std::cout << "array[0]:" << *(array+0) << std::endl;
    std::cout << "array[1]:" << *(array+1) << std::endl;
    std::cout << "array[2]:" << *(array+2) << std::endl;
    std::cout << "array[3]:" << *(array+3) << std::endl;
    std::cout << "array[4]:" << *(array+4) << std::endl;
    std::cout << "array[5]:" << *(array+5) << std::endl;

    return 0;
}
```

Array offset and dereference (2/3)

- So if we think about our previous example:
    - The number of times we increment p_s, was the offset into the array
    - We can access a value by offsetting to a position, and then dereferencing that address!
        - (See example on the right)




```cpp
// @file arithmetic2.cpp
// g++ -std=c++17 arithmetic2.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Array offset shorthand
    std::cout << "array[0]:" << *(array+0) << std::endl;
    std::cout << "array[1]:" << *(array+1) << std::endl;
    std::cout << "array[2]:" << *(array+2) << std::endl;
    std::cout << "array[3]:" << *(array+3) << std::endl;
    std::cout << "array[4]:" << *(array+4) << std::endl;
    std::cout << "array[5]:" << *(array+5) << std::endl;

    return 0;
}
```

Array offset and dereference (3/3)

```
mike:pointers$ g++ -std=c++17 arithmetic2.cpp -o prog
mike:pointers$ ./prog
array[0]:0
array[1]:1
array[2]:2
array[3]:3
array[4]:4
array[5]:5
```

- So remember--an array is just a contiguous chunk of memory.
- Arrays are a homogenous data structure, meaning all the data stored is the same type:
  - We can thus use a pointer arithmetic to navigate pointers through an array (Using `++`, `+1`, `+2`, `--`, `-2`, etc.)

Array Decay to Pointer (1/2)

- Now, while traversing our array using pointer arithmetic was neat--there was something subtle
  - When we are doing the 'traversal' (p_s++) we are losing information about the array--and instead incrementing along a pointer
  - We actually have a pointer type, not an array.
    - Notice the difference on the right
      - array vs &array[0]

```cpp
// @file decay.cpp
// g++ -std=c++17 decay.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    // Note: 'array' versus '&array[0]
    //       is slightly different.
    std::cout << "sizeof(array)     : " << sizeof(array) << std::endl;
    std::cout << "sizeof(&array[0]): " << sizeof(&array[0]) << std::endl;

    short* p_s = array; // Can just point to the pointer, intead of &array[0]
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```

```
mike:pointers$ g++ -std=c++17 decay.cpp -o prog
mike:pointers$ ./prog
sizeof(array)     : 12
sizeof(&array[0]): 8
*p_s= 0
*p_s= 1
*p_s= 2
*p_s= 3
*p_s= 4
*p_s= 5
```

Array Decay to Pointer (2/2)

- Now, while traversing our array using pointer arithmetic

If this is confusing, you can try:

```
array = p_s;       // not allowed
p_s = &array[3];  // allowed, we
                  // retrieve the
                  // address of
                  // element 3
```

pointer

- We actually have a pointer type, not an array.
  - Notice the difference on the right
    - array vs &array[0]

```cpp
// @file decay.cpp
// g++ -std=c++17 decay.cpp -o prog
#include <iostream>
#include <array>

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }
    // Pointer to start of array
    // Note: 'array' versus '&array[0]
    //        is slightly different.
    std::cout << "sizeof(array)     : " << sizeof(array) << std::endl;
    std::cout << "sizeof(&array[0]): " << sizeof(&array[0]) << std::endl;

    short* p_s = array; // Can just point to the pointer, intead of &array[0]
    for(int i=0; i < 6; i++){
        std::cout << "*p_s= " << *p_s << std::endl;
        p_s++;
    }

    return 0;
}
```

```
mike:pointers$ g++ -std=c++17 decay.cpp -o prog
mike:pointers$ ./prog
sizeof(array)     : 12
sizeof(&array[0]): 8
*p_s= 0
*p_s= 1
*p_s= 2
*p_s= 3
*p_s= 4
*p_s= 5
```

Pointers as parameters

arrays decay to pointers in function parameters

(Think for a moment what information we lose)

Arrays decay to pointers as function parameter (1/2)

- In the example of the right, I again show this, when attempting to pass an 'array' as a function parameter, it's thus treated as a pointer.
  - The dimensions of our array would need to be sent in as a parameter
  - Personally, I would prefer using as a parameter:
    - `std::vector<short>`

```cpp
// @file decay2.cpp
// g++ -std=c++17 decay2.cpp -o prog
#include <iostream>
#include <array>

void arrayDecay(short* arr){
    std::cout << "sizeof(arr)  : " << sizeof(arr) << std::endl;
}

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }

    std::cout << "sizeof(array): " << sizeof(array) << std::endl;
    arrayDecay(array);

    return 0;
}
                                                        1,19
```

```
mike:pointers$ g++ -std=c++17 decay2.cpp -o prog
mike:pointers$ ./prog
sizeof(array): 12
sizeof(arr)  : 8
```

Arrays decay to pointers as function parameter (2/2)

- Here's the fix
  - Just pass in the size of your collection as a second parameter
  - Then utilize your array as needed.

```cpp
// @file decay2_fixed.cpp
// g++ -std=c++17 decay2_fixed.cpp -o prog
#include <iostream>
#include <array>

void arrayAsPointerWithSize(short* arr, size_t collectionSize){
    std::cout << "sizeof(arr)  : " << sizeof(arr) << std::endl;
    for(int i=0; i < collectionSize; i++){
        std::cout << arr[i] << std::endl;
    }
}

int main(){

    short array[6];
    for(int i=0; i < 6; i++){
        array[i] = i;
    }

    std::cout << "sizeof(array): " << sizeof(array) << std::endl;
    arrayAsPointerWithSize(array,6);

    return 0;
}
```

(Just for fun--passing std::array with template parameter) (1/2)

- This is just for fun
  - We could use a template parameter to store the size
  - (For those who love templates)

```cpp
// @file decay3.cpp
// clang++-10 -std=c++20 decay3.cpp -o prog
#include <iostream>
#include <array>

// Kind of a wild example...
// We may want to use a std::vector
template<int T>
void printArrayWithoutSizeParameter(const std::array<short,T>& arr){
    std::cout << "sizeof(arr)  : " << sizeof(arr) << std::endl;
}

int main(){

    std::array<short,11> array;
    array.fill({{0,1,2,3,4,5,6,7,8,9,10}});
    // array.size() is constexpr
    printArrayWithoutSizeParameter<array.size()>(array);

    // Creating another array....
    std::array<short,10> array2;
    printArrayWithoutSizeParameter<10>(array2);

    return 0;
}

```

```
mike:pointers$ ./prog
sizeof(arr)  : 22
sizeof(arr)  : 20
```

(Just for fun--passing std::array with template parameter) (2/2)

- Very quickly we'll start generating lots of code for each uniquely sized array!
  - See with output with:
    - `clang++-10 -std=c++20 -Xclang -ast-print -fsyntax-only`

```
// @file decay3.cpp
// clang++-10 -std=c++20 decay3.cpp -o prog
#include <iostream>
#include <array>

// Kind of a wild example...
// We may want to use a std::vector
template<int T>
void printArrayWithoutSizeParameter(const std::array<short,T>& arr){
    std::cout << "sizeof(arr)  : " << sizeof(arr) << std::endl;
}

int main(){

    std::array<short,11> array;
    array.fill({{0,1,2,3,4,5,6,7,8,9,10}});
    // array.size() is constexpr
```

```
template <int T> void printArrayWithoutSizeParameter(const std::array<short, T> &arr) {
        std::cout << "sizeof(arr)  : " << sizeof (arr) << std::endl;
}
template<> void printArrayWithoutSizeParameter<11>(const std::array<short, 11> &arr) {
    std::cout << "sizeof(arr)  : " << sizeof (arr) << std::endl;
}
template<> void printArrayWithoutSizeParameter<10>(const std::array<short, 10> &arr) {
    std::cout << "sizeof(arr)  : " << sizeof (arr) << std::endl;
}
```

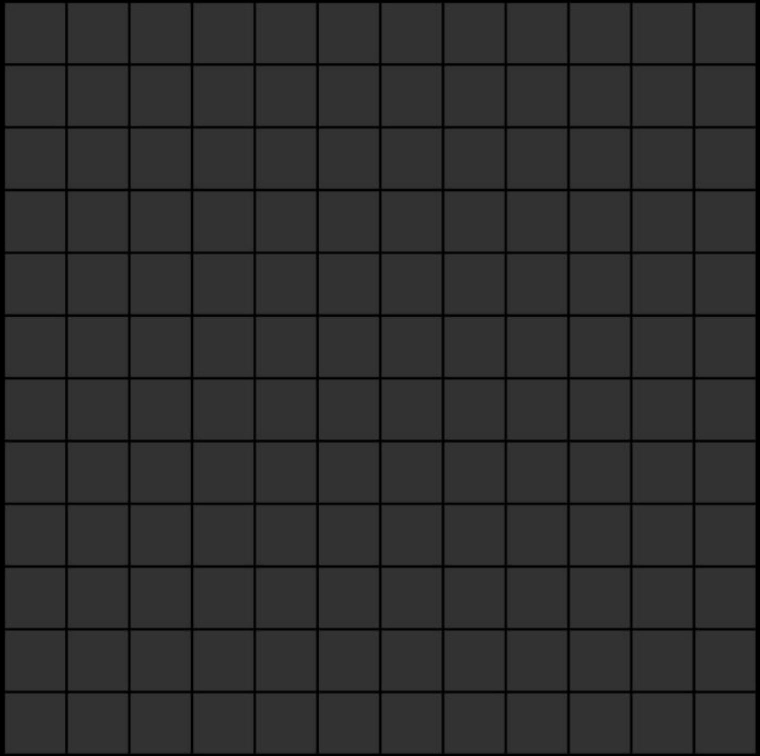
Dynamically allocated arrays

We need pointers to point to a chunk of memory that our allocator gives us

(Thus **pointers are necessary** for dynamic memory allocation)

Dynamically Allocated Arrays (i.e., using `new`) (1/5)

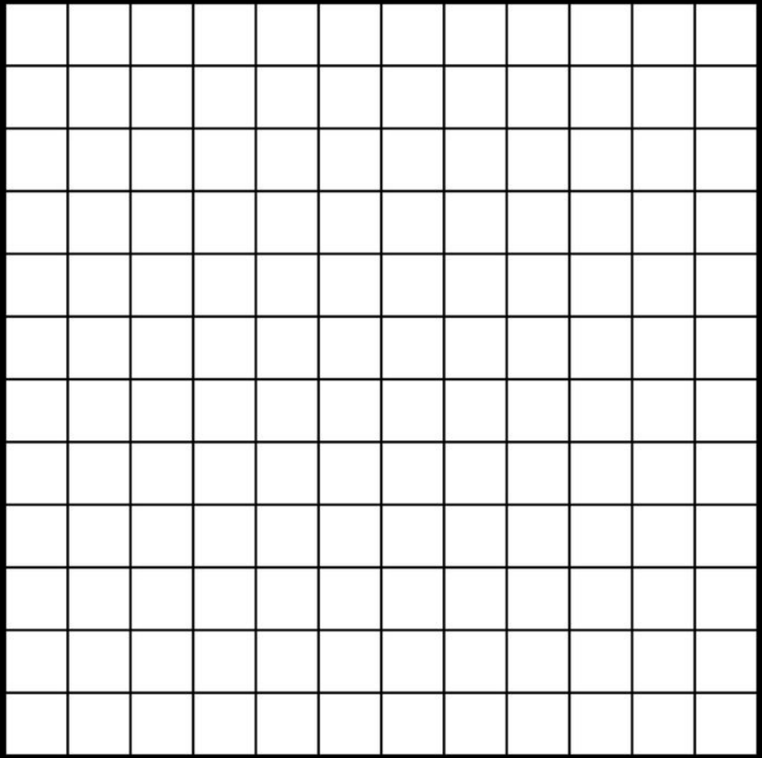
- Recall: indexing into arrays works by dereferencing at a specific offset
    - The element we access is the data type size multiplied by the index (i.e., how far we want to shift our pointer to access a specific piece of memory)
- Let’s now see how dynamically allocated arrays work
    - i.e., We want to see what happens when an allocator (e.g., `new`) returns a pointer



Dynamically Allocated Arrays (i.e., using new) (2/5)

- Let’s look at an example

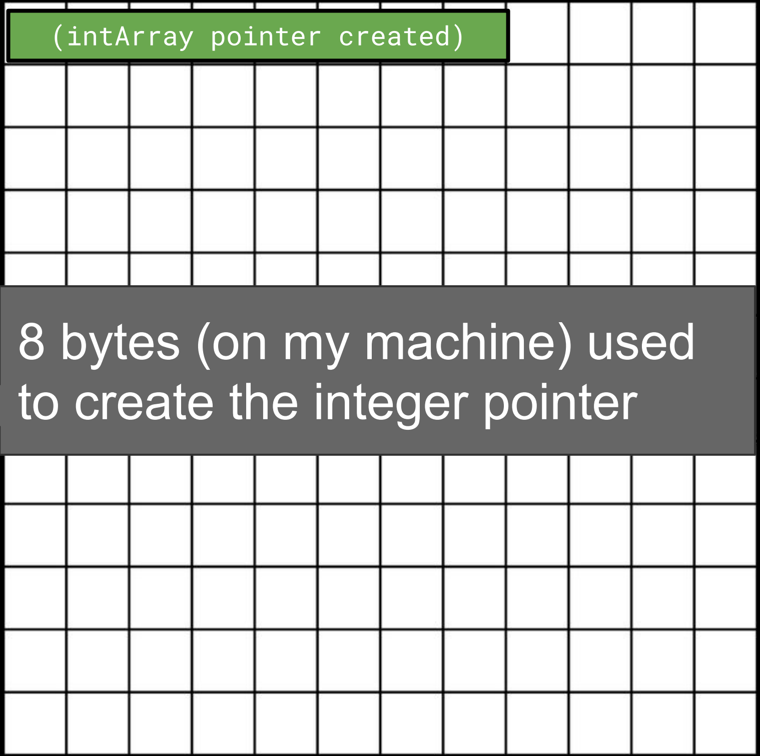
```
// @file new.cpp
// g++ -std=c++17 new.cpp -o prog
#include <iostream>

int main(){
    // Request enough bytes for: sizeof(int)*3
    // intArray points to the start of that
    // chunk of memory i.e.,
    // intArray = &(block of memory)
    int* intArray = new int[3];

    // Delete the entire array
    // Note: We use brackets to delete the entire
    //       allocated block.
    //       Using only 'delete' removes the first
    //       element.
    delete[] intArray;

    return 0;
}
```



Dynamically Allocated Arrays (i.e., using new) (3/5)

- Let's look at an example

```
// @file new.cpp
// g++ -std=c++17 new.cpp -o prog
#include <iostream>

int main(){
    // Request enough bytes for: sizeof(int)*3
    // intArray points to the start of that
    // chunk of memory i.e.,
    // intArray = &(block of memory)
    int* intArray = new int[3];

    // Delete the entire array
    // Note: We use brackets to delete the entire
    //       allocated block.
    //       Using only 'delete' removes the first
    //       element.
    delete[] intArray;

    return 0;
}
```

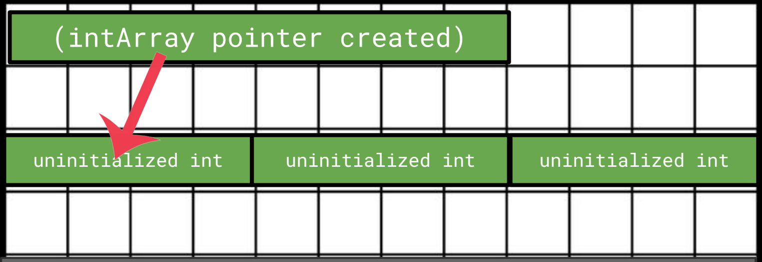


8 bytes (on my machine) used to create the integer pointer

Dynamically Allocated Arrays (i.e., using new) (4/5)

- Let's look at an example

```cpp
// @file new.cpp
// g++ -std=c++17 new.cpp -o prog
#include <iostream>

int main(){
    // Request enough bytes for: sizeof(int)*3
    // intArray points to the start of that
    // chunk of memory i.e.,
    // intArray = &(block of memory)
    int* intArray = new int[3];

    // Delete the entire array
    // Note: We use brackets to delete the entire
    //       allocated block.
    //       Using only 'delete' removes the first
    //       element.
    delete[] intArray;

    return 0;
}
```



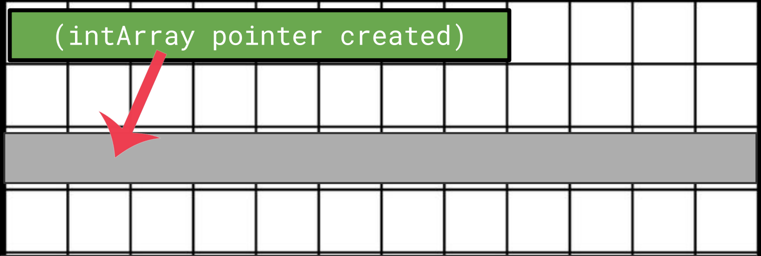
An allocator provides us 12 bytes, and our integer pointer points to the first integer (zero offset, or 0-index into array)

(Note: Those our 12 bytes are allocated *somewhere* in our memory-- specifically in our 'heap' memory)

Dynamically Allocated Arrays (i.e., using new) (5/5)

- Let's look at an example

```
// @file new.cpp
// g++ -std=c++17 new.cpp -o prog
#include <iostream>

int main(){
    // Request enough bytes for: sizeof(int)*3
    // intArray points to the start of that
    // chunk of memory i.e.,
    // intArray = &(block of memory)
    int* intArray = new int[3];

    // Delete the entire array
    // Note: We use brackets to delete the enti
    //        allocated block.
    //        Using only 'delete' re           first
    //        element.
    delete[] intArray;

    return 0;
}
```

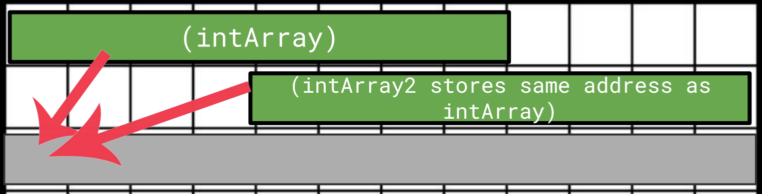


Whole chunk of original allocation is freed.

Note: pointer still pointing to something (maybe garbage, maybe not?)...careful if you dereference!

Dynamically Allocated Arrays - Round 2 (1/2)

```cpp
// @file new2.cpp
// g++ -std=c++17 new2.cpp -o prog
#include <iostream>

int main(){
    int* intArray = new int[3];
    for(int i=0; i < 3; i++){
        intArray[i] = i;
    }
    // Let's share our data by pointing
    // to it through another pointer
    int* intArray2 = intArray;

    std::cout << "intArray2[1]= " << intArray2[1] << std::endl;
    delete[] intArray;

    // Uh-oh, what if we try to work with intArray2
    std::cout << "intArray2[1]= " << intArray2[1] << std::endl;

    return 0;
}
```



Now, in this example we have two pointers pointing to the same data

Then we delete[]

Then, we try to use memory after it has been freed! (next slide for results)

Dynamically Allocated Arrays - Round 2 (2/2)

```
mike:pointers$ g++ -std=c++17 new2.cpp -o prog
mike:pointers$ ./prog
intArray2[1]= 1
intArray2[1]= 0
```

It’s a bit tricky, but we have to think about which pointer owns the memory, and when it is safe to clear the memory

```
    // Uh-oh, what if we try to work with intArray2
    std::cout << "intArray2[1]= " << intArray2[1] << std::endl;

    return 0;
}
```

`nullptr`

Does a pointer have to point to anything? (Think about our last example)

i.e., What happens if we dereference nothing?

(Some allocators when memory is freed will set memory to `nullptr`--that would be a problem as seen in our previous example!)

What if a pointer, points to...nothing? (1/2)

- We should always initialize our variables
  - In C++ 11 and beyond we can initialize a pointer to [‘nullptr’]() (This is a [prvalue]())
  - But if we try to retrieve a value by dereferencing a nullptr, we get a segmentation fault.
    - There’s nothing in-effect at that address where we can retrieve a value from--program terminates

```cpp
// @file nullptr.cpp
// g++ -std=c++17 nullptr.cpp -o prog
#include <iostream>

int main(){
    // Initialize px
    int* px= nullptr; // 'nullptr' is the modern C++ way
                      // Note: We could also assign to
                      //       NULL or 0, but that is more
                      //       of a C-style.
    std::cout << "What happens here? " << *px << std::endl;


    return 0;
}
```

```
mike:pointers$ g++ -std=c++17 nullptr.cpp -o prog
mike:pointers$ ./prog
Segmentation fault (core dumped)
```

What if a pointer, points to...nothing? (2/2)

- So the tip is:
  - Check for `nullptr` if you are going to attempt to dereference a pointer that may be null.
- Note:
  - Modern C++ programmers prefer `nullptr` as opposed to the macro NULL in C (which is essentially just `0`).
    - `nullptr` provides additional type safety

```cpp
// @file nullptr2.cpp
// g++ -std=c++17 nullptr2.cpp -o prog
#include <iostream>

int main(){
    // Initialize px
    int* px= nullptr; // 'nullptr' is the modern C++ way
                      // Note: We could also assign to
                      //       NULL or 0, but that is more
                      //       of a C-style.
    // Check for nullptr
    if(nullptr != px){
        std::cout << "What happens here? " << *px << std::endl;
    }

    return 0;
}
```

Pitfalls of pointers

We have seen one so far, and "with great power comes great responsibility"

```cpp
// @file nullptr.cpp
// g++ -std=c++17 nullptr.cpp -o prog
#include <iostream>

int main(){
    // Initialize px
    int* px= nullptr; // 'nullptr' is the modern C++ way
                      // Note: We could also assign to
                      //       NULL or 0, but that is more
                      //       of a C-style.
    std::cout << "What happens here? " << *px << std::endl;


    return 0;
}
```

```
mike:pointers$ g++ -std=c++17 nullptr.cpp -o prog
mike:pointers$ ./prog
Segmentation fault (core dumped)
```

Dereferencing a `nullptr` will cause a segmentation fault

Common Pitfalls of pointers

- Because pointers allow sharing, we need to think about ownership
  - When I talk about ownership, that means 'who or which object' is responsible for deleting dynamically allocated memory
    - (Note: We have some rules for this: Back to Basics: RAII and the Rule of Zero - Arthur O'Dwyer - CppCon 2019)
- So--in one slide each I want to show you the common pitfalls of pointers
  - (Note: We've already seen dereferencing a `nullptr`)

Memory Leaks (1/2)

- A memory leaks is when we forget to reclaim our memory
    - To the right is an example of never reclaiming (with delete or delete[]) our memory.

```cpp
// @file leak.cpp
// g++ -std=c++17 leak.cpp -o prog
#include <iostream>
#include <array>


int main(){

    // Not the worse thing, but bad...
    int* memory = new int [1000];

    while(1){
        // Very bad...lots of allocations
        int* lotsOfAllocation = new int [1];
    }

    return 0;
}

// Eventually the operating system cleans up
// the memory after execution....I hope :)
```

Memory Leaks (2/2)

- You can use tools like:
  - [address sanitizer](#) or [valgrind](#) (pronounced val-grinn, not val-grind) to help you detect bugs
- For my advanced members in the audience, consider memory tagging strategies (i.e., override new for your objects)
  - (This applies to all pointer/memory bugs)

```cpp
// @file leak.cpp
// g++ -std=c++17 leak.cpp -o prog
#include <iostream>
#include <array>


int main(){

    // Not the worse thing, but bad...
    int* memory = new int [1000];

    while(1){
        // Very bad...lots of allocations
        int* lotsOfAllocation = new int [1];
```

```
mike:pointers$ clang++-10 -g -fsanitize=address leak.cpp -o prog; ASAN_OPTIONS=detect_leaks=1 ./prog

=================================================================
==20586==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4000 byte(s) in 1 object(s) allocated from:
    #0 0x4c357d in operator new[](unsigned long) (/home/mike/cppcon2021/pointers/prog+0x4c357d)
    #1 0x4c5d68 in main /home/mike/cppcon2021/pointers/leak.cpp:9:19
    #2 0x7f794216ab96 in __libc_start_main /build/glibc-2ORdQG/glibc-2.27/csu/../csu/libc-start.c:310

SUMMARY: AddressSanitizer: 4000 byte(s) leaked in 1 allocation(s).
```

Dangling pointers (1/2)

- Dangling pointers arise when we point to the address of a value that may not exist
  - Most of our compilers are good at giving warnings these days (see to the right)
- So we try to avoid pointing to data that does not have the same lifetime as our pointer
  - Otherwise, we need to update our pointer to valid data or nullptr

```
// @file dangling.cpp
// g++ -std=c++17 dangling.cpp -o prog
#include <iostream>

char* dangerouslyReturnLocalValue(){
        char c = 'c';
        return &c;
}

int main(){

    char* danglingPointer1 = dangerouslyReturnLocalValue();

    std::cout << "*danglingPointer1 is: " << *danglingPointer1 << std::endl;

    return 0;
}

"dangling.cpp" 19L, 334C written
```

```
mike:pointers$ g++ -std=c++17 dangling.cpp -o prog
dangling.cpp: In function ‘char* dangerouslyReturnLocalValue()’:
dangling.cpp:7:14: warning: address of local variable ‘c’ returned [-Wreturn-local-addr]
         char c = 'c';
              ^
mike:pointers$ ./prog
Segmentation fault (core dumped)
```

Dangling pointers (2/2)

- Again, use address sanitizers, memory tools, and your interactive debuggers (e.g., GDB) to help detect these errors.
    - (See some magic debug values like 0xDEADBEEF to help catch dangling pointers https://en.wikipedia.org/wiki/Magic_number_(programming)#Magic_debug_values)

Double Frees

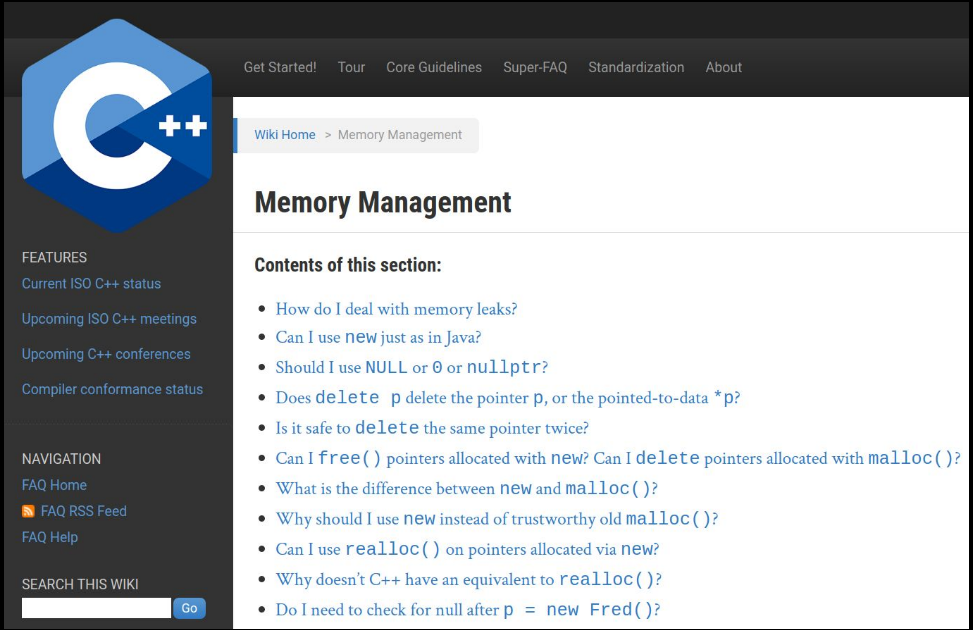
- A double free occurs when we are sharing data between 2 or more pointers
- We *are trying* to be good and free our memory
    - The problem is we end up freeing the same memory twice.
- Note:
    - My runtime protects me, so I don’t see a crash--at least on a toy example.
    - That does not mean it is not there though!
        - (What happens if I change allocators, platforms, hardware, etc.?)

```cpp
// @file double.cpp
// g++ -std=c++17 double.cpp -o prog
#include <iostream>


int main(){

    float* f1 = new float[100];
    float* f2 = f1;

    delete[] f2;
    f2 = nullptr;
    delete[] f1;
    // Be good and set f1 to nullptr
    f1 = nullptr;
    // Did I delete f2? I'll try again
    delete[] f2;

    return 0;
}

~
"double.cpp" 21L, 330C written
```

```
mike:pointers$ g++ -std=c++17 double.cpp -o prog
mike:pointers$ ./prog
```

And more....

- For memory and pointer related best practices on common pitfalls see the ISOCPP guide here:
  - https://isocpp.org/wiki/faq/freestore-mgmt
- And since we're on pointers...some tips on *when* to return a pointer from a function
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-return-ptr
  - (Careful!)



Bug Mitigate with a Wrapper Class

- We can build our own pointer class
  - (Example sketch to the right)
  - The idea is to build abstraction around a ‘raw/naked/plain pointer’
- Luckily, the standard library (C++11 and beyond) provides ‘smart pointers’ for us
  - Very briefly (one slide each) I will cover the three types of smart pointers
  - There will be at least one talk on smart pointers after this talk by Inbal Levi
    - (and there are a few other cppcon talks on smart pointers I will link at the end)

```cpp
// @file mikepointer.cpp
// g++ -std=c++17 mikepointer.cpp -o prog
#include <iostream>

// WARNING: NOT PRODUCTION CODE
//          Just a sketch for a talk :)
//          Please use built-in smart pointers
template <class T>
class MikeSafePointer{
public:
    MikeSafePointer(){
        rawPointer = new T;
        use_count = 1; // Consider an 'addUse'
    }                  // and 'Release' function
                       // to manage this value.
    // Destructor checks if it's safe to destroy
    ~MikeSafePointer(){
        if(1==use_count){
            delete rawPointer;
            rawPointer = nullptr;
        }else{
            // Maybe log a warning if in DEBUG mode
        }
    }

private:
    // Hide the raw pointer (even better use pIMPL*)
    T* rawPointer;
    int use_count; // Increment every time something is assigned
                   // to this pointer.
                   // i.e., this could be the 'ref count'
};

int main(){

    MikeSafePointer<int> mike_int_pointer;

    return 0;
}
```

Getting Smart (with smart pointers)

Using Modern C++ to write safer code

What is a smart pointer?

- A container in C++, that wraps a 'pointer'
  - It's a 'proxy' in the sense that we can use it in place of other pointers.
- We construct the pointer using one of the following types
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`

Dynamic memory management

Smart pointers

Smart pointers enable automatic, exception-safe, object lifetime management.

Defined in header <memory>

Pointer categories

unique_ptr (C++11)	smart pointer with unique object ownership semantics (class template)
shared_ptr (C++11)	smart pointer with shared object ownership semantics (class template)
weak_ptr (C++11)	weak reference to an object managed by std::shared_ptr (class template)

https://en.cppreference.com/w/cpp/memory

What problem does a smart pointer solve?

- We don’t have to call ‘`delete`’ explicitly anymore!
- We can even avoid calling ‘new’
  - (e.g. if we use `make_shared` or `make_unique`)
- We ultimately are enforcing constraints with each of the three types of smart pointers!

What smart pointers are doing?

- Behind the scenes, smart pointers have different ‘constraints’ and are otherwise doing bookkeeping for you.
    - May be reference counting
    - May be enforcing uniqueness
    - May be handling exceptions during the pointer creation(e.g. `make_shared` or `make_unique`)

`unique_ptr` example

- Scoped pointer
  - When it goes out of scope, it will automatically be deleted.
- We cannot copy them
  - This avoids the ‘double free’ issue
  - Can be your ‘default’ if you want to be very careful with your pointers, and do not intend on sharing data.
- We also cannot assign unique_ptr to something else, it has to be unique.
- We prefer the std::make_unique call generally (see comments)

```cpp
// @file unique.cpp
#include <iostream> // I/O stream library
#include <memory>   // Access smart pointers

class Object{
public:
    Object() { std::cout << "Constructor\n"; }
    ~Object() { std::cout << "Destructor\n"; }
};


// Entry point to program 'main' unique
int main(int argc, char* argv[]){

    {
        std::unique_ptr<Object> myObjectPtr(new Object);
        // NOTE: Below is 'illegal' because we cannot
        //       assign to a unique_ptr--awesome!
        //std::unique_ptr<Object> myObjectPtr2 = myObjectPtr;
    }

    // An alternative way to create the pointer
    // Much more explict, avoids the call to 'new'
    // Also does some error handling.
    {
        std::unique_ptr<Object> myObjectPtr = std::make_unique<Object>();
    }



    return 0;
}
```

shared_ptr example

- Allows a pointer to have multiple things pointing to it.
  - As long as other pointers are pointing to that memory, the memory will not be deleted.
- Internally ‘reference counting’ or otherwise keeping track of how many things point to it is taking place
  - If nothing is pointing to it, then the pointer can safely be deleted
    - (This is a similar idea to how garbage collection works in Java)

```cpp
// @file shared.cpp
#include <iostream> // I/O stream library
#include <memory>   // Access smart pointers

class Object{
public:
    Object() { std::cout << "Constructor\n"; }
    ~Object() { std::cout << "Destructor\n"; }
};


// Entry point to program 'main' shared
int main(int argc, char* argv[]){


    // This is how we 'always' want to create shared_ptr using
    // 'make_shared'
    // Again, we avoid using 'new'
    {
        std::shared_ptr<Object> mySharedObjectPtr;
        {
            // Make a second pointer
            std::shared_ptr<Object> mySharedObjectPtr2 = std::make_shared<Object>();
            // Assign our shared pointer to another shared pointer
            mySharedObjectPtr = mySharedObjectPtr2;
        }
    // At this point, mySharedObjectptr will 'die' but only because all of its
    // references have gone out of scope.
    }
    return 0;
}
```

`weak_ptr` example

- Very similar to a shared pointer, but it does not increase the 'reference count'
- In this way, you can have 'invalid' pointers
  - Sometimes you do not care however, and maybe you just want a lightweight way to point to some references.
  - e.g. You have a GameObject that was blown up mid-way in the game while other objects were communicating with it. You should check for nullptr, but it 'may' be okay if these objects still point to something deleted.

```cpp
// @file weak.cpp
#include <iostream> // I/O stream library
#include <memory>   // Access smart pointers

class Object{
public:
    Object() { std::cout << "Constructor\n"; }
    ~Object() { std::cout << "Destructor\n"; }
};


// Entry point to program 'main' weak
int main(int argc, char* argv[]){

    {
        // weak_ptr is almost like a 'temporary' pointer that we just
        // want to be able to point to something if it exists.
        std::weak_ptr<Object> myWeakPtr;
        {
            // Make a second pointer
            std::shared_ptr<Object> mySharedObjectPtr2 = std::make_shared<Object>();
            // Assign our weak pointer to a shared pointer, but
            // we do not increase the reference count on sharedObjectPtr2.
            myWeakPtr = mySharedObjectPtr2;
        }
    }

    return 0;
}
```

Another `weak_ptr` example

- Adapted from https://en.cppreference.com/w/cpp/memory/weak_ptr with some annotation as to what is going on
  - The motivation for weak_ptr is to ‘point’ to something that *may* exist, but if it does not, you are okay.
    - So the weak_ptr does not own the data in anyway, can only point to it if it exists.

```cpp
1 // @file weak_pointers.cpp
2 #include <iostream>
3 #include <memory>
4
5 // Here we're creating a global weak pointer(gw)
6 std::weak_ptr<int> gw;
7
8 void f(){
9
10     // Pointing a new shared pointer, to whatever
11     // our weak_ptr (gw) holds.
12     auto spt = gw.lock();
13
14     // If the shared pointer we created, which now
15     // also points to the same thingg as our weak_ptr,
16     // then we can proceed.
17     // However, if the thing our weak_ptr, pointed to
18     // no longer exists (i.e "sp" below"), thten
19     if (spt != nullptr) { // Has to be copied into a shared_ptr before usage
20         std::cout << *spt << "\n";
21     }
22     else {
23         std::cout << "gw is expired\n";
24     }
25 }
26
27 int main(){
28     { // Create a scope within main
29         int x = 5;
30         int* p = &x;
31         std::shared_ptr<int> sp = std::make_shared<int>(42);
32         gw = sp; // Assigning weak_ptr to the shared_ptr
33         f();
34     } // Exit the scope at line 23, thus deleting all local variables
35       // between lines 23 and 29
36
37     f();
38 }
```

~~std::auto_ptr - deprecated~~

- You may also see this type of pointer on occasion, but it has been deprecated in c++ 17
  - Thus, don’t use it.

Pointers and Functions

Functions themselves have an address in memory, so we have function pointers

Functions have an address

- Of course they do--functions must exist somewhere!
    - Below is a snippet showing where two functions exist in memory
    - (use the nm tool to find symbols after compiling a debug version of your code):
        - `nm -g -C ./prog`

```cpp
// @file functionPointer.cpp
// g++ -std=c++17 functionPointer.cpp -o prog
#include <iostream>

int add(int x,int y){
    return x+y;
}
int multiply(int x, int y){
    return x*y;
}
```

```
mike:pointers$ nm -g -C ./prog
0000000000201010 B __bss_start
                 U __cxa_atexit@@GLIBC_2.2.5
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000201000 D __data_start
0000000000201000 W data_start
0000000000201008 D __dso_handle
0000000000201010 D _edata
0000000000201138 B _end
0000000000000ac4 T _fini
                 w __gmon_start__
0000000000000778 T _init
0000000000000ad0 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000000ac0 T __libc_csu_fini
0000000000000a50 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000000931 T main
0000000000000800 T _start
0000000000201010 D __TMC_END__
000000000000090a T add(int, int)
000000000000091e T multiply(int, int)
```

Creating and Using Function Pointers (1/2)

- So to the right I have an example pointer to a function
    - Note the syntax on line 14 for creating.
        - We work inside to out
            - First naming the pointer
            - Then I have a list of parameters
            - Then a return type
    - On line 16 I assign the function pointer

```cpp
1 // @file functionPointer.cpp
2 // g++ -std=c++17 functionPointer.cpp -o prog
3 #include <iostream>
4
5 int add(int x,int y){
6     return x+y;
7 }
8 int multiply(int x, int y){
9     return x*y;
10 }
11
12 int main(){
13     // Create a pointer to the function
14     int (*pfn_arithmetic)(int,int);
15     // Point to the add function
16     pfn_arithmetic = add;
17     std::cout << "pfn_arithmetic(2,7) = " << pfn_arithmetic(2,7) << std::endl;
18     // Point to the multiply function
19     pfn_arithmetic = multiply;
20     std::cout << "pfn_arithmetic(2,7) = " << pfn_arithmetic(2,7) << std::endl;
21
22     return 0;
23 }
```

```
mike:pointers$ g++ -std=c++17 functionPointer.cpp -o prog
mike:pointers$ ./prog
pfn_arithmetic(2,7) = 9
pfn_arithmetic(2,7) = 14
```

Creating and Using Function Pointers (2/2)

- So to the right I have an example pointer to a function
  - Note the syntax on line 14 for creating.
    - We work inside to out
      - First naming the pointer
      - Then I have a list of parameters
      - Then a return type
  - On line 16 I assign the function pointer
    - Line 17 and 20 calls are made

```cpp
1 // @file functionPointer.cpp
2 // g++ -std=c++17 functionPointer.cpp -o prog
3 #include <iostream>
4
5 int add(int x,int y){
6     return x+y;
7 }
8 int multiply(int x, int y){
9     return x*y;
10 }
11
12 int main(){
13     // Create a pointer to the function
14     int (*pfn_arithmetic)(int,int);
15     // Point to the add function
16     pfn_arithmetic = add;
17     std::cout << "pfn_arithmetic(2,7) = " << pfn_arithmetic(2,7) << std::endl;
18     // Point to the multiply function
19     pfn_arithmetic = multiply;
20     std::cout << "pfn_arithmetic(2,7) = " << pfn_arithmetic(2,7) << std::endl;
21
22     return 0;
23 }
```

```
mike:pointers$ g++ -std=c++17 functionPointer.cpp -o prog
mike:pointers$ ./prog
pfn_arithmetic(2,7) = 9
pfn_arithmetic(2,7) = 14
```

Modern C++ `std::function` [reference]

- `std::function` allows you to store a callable object.
  - A function pointer for example would be something that is callable
  - Syntax is almost the same

```cpp
// @file stdfunction.cpp
// g++ -std=c++17 stdfunction.cpp -o prog
#include <iostream>
#include <functional> // std::function

int add(int x,int y){
    return x+y;
}
int multiply(int x, int y){
    return x*y;
}

int main(){
    // Use the modern std::function
    std::function<int(int,int)> f_arithmetic = add;
    // Point to the add function
    f_arithmetic = add;
    std::cout << "pfn_arithmetic(2,7) = " << f_arithmetic(2,7) << std::endl;
    // Point to the multiply function
    f_arithmetic = multiply;
    std::cout << "pfn_arithmetic(2,7) = " << f_arithmetic(2,7) << std::endl;

    return 0;
}
```

```
mike:pointers$ g++ -std=c++17 stdfunction.cpp -o prog
mike:pointers$ ./prog
f_arithmetic(2,7) = 9
f_arithmetic(2,7) = 14
```

Other odds and ends...for a full day course in the future :)

- (I’m probably running out of time at this point!)
  - void*
  - Casting pointers
  - Using uintptr_t
  - ptrdiff_t
  - const and pointers
  - Some examples of multi-dimensional arrays
  - Hiding behind a pointer (pIMPL idiom)
- How is a reference different? (It is essentially a const behavior)
  - It is really just a **`int* const`** pointer (pay attention to const after int*)
    - https://godbolt.org/z/7W9coGbYd
    - (i.e., we cannot change what we point to when passing by reference)
  - reference is an ‘alias’ (or another name) for which to refer to a symbol
  - reference always points to same object, so much harder to create a nullptr (still could get a dangling reference however)

Data structures

Singly Linked List
(If time allows)

TT the clock from Diddy Kong Racing N64

So Because Pointers point to other pointers

- We can build some cool data 'linked' data structures
  - My audience here attending Cppcon I am sure has done this...
  - For those watching in the future though--think about how you could implement a 'list' data structure.
    - How would you add nodes?
    - How would you delete nodes? The entire list?
  - (Let's take a look--if we have time)
    - https://github.com/MikeShah/cppcon2021/blob/main/pointers/ll.cpp
    - (Note: This is how I cheat if my talk is going too long or too short ;)

```cpp
// @file ll.cpp
// Linked list example
// g++ -Wall -Wextra -std=c++17 ll.cpp -o prog
#include <iostream>

struct Node{
    Node(){
        data =0;
        next = nullptr;
    }
    int data;
    Node* next;
};

class SinglyLinkedList{
    public:
        SinglyLinkedList() {
            m_head = nullptr;
        }

        void PrependNode(int data){
            if(m_head==nullptr){
                m_head = new Node;
                m_head->data = data;
                m_head->next = nullptr;
            }else{
                Node* newNode = new Node;
                newNode->data = data;
                newNode->next = m_head;
                m_head = newNode;
            }
        }
        void PrintList(){
            Node* iter = m_head;
            while(iter!=nullptr){
                std::cout << iter->data << "\n";
                iter=iter->next;
            }
            std::cout << std::endl;
        }

        ~SinglyLinkedList(){
            Node* iter = m_head;
            Node* next = m_head->next;
            while(iter!=nullptr){
                delete iter;
                iter = next;
                if(iter!=nullptr){
                    next = iter->next;
                }
            }
        }
        // Copy constructor/Copy assignment operator
        // removed for brevity...
    private:
        Node* m_head;
};

int main(){

    SinglyLinkedList myList;

    myList.PrependNode(1);
    myList.PrependNode(2);
    myList.PrependNode(3);
    myList.PrependNode(4);

    myList.PrintList();
    // Free nodes...or handle in destructor

    return 0;
}
```

Conclusion

Wrapping up what we’ve learned

Conclusion -- C++ Programmers

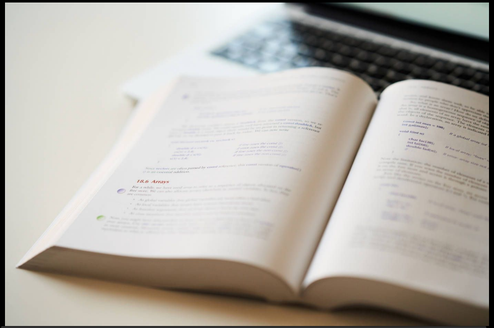
- You still need to know about raw pointers
- Whether you are an expert or a beginner
  - If you’re a beginner
    - Now you know a little bit more about the foundations
    - Now you’ll understand what smart pointers are doing behind the scenes for you
    - Now you should try to build some data structures for practice, or perhaps some more advanced ones for optimization
  - If you’re an expert
    - Consider you may need to interface with C-APIs, embedded systems, or simply using a legacy code base.
    - You’ll have to design your functions using pointers for example
  - For expert C++ programmers teaching C++
    - It’s always worth teaching the foundations (in which order and where in the curriculum differs however)

Some Analogies on Pointers for Educators

(Whether teacher/professor or if you're trying to explain to your team members about pointers)

A (common) analogy of what a pointer is

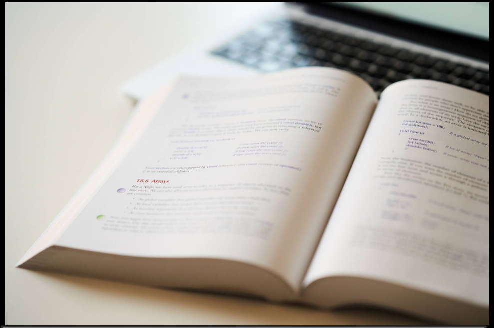
- A **pointer** is a variable that stores the memory address of a specific object type
  - Okay--not so bad.
    - So a ‘pointer’ is a data type
    - And it can store objects of a specific data type
      - ^So what exactly does this mean, and how could we do this efficiently?
- If an object is a page in a book
- then a ‘pointer’ would be the index in the back of the book that points you to a specific page.



object: A page in a book

A (common) analogy of what a pointer is

- A **pointer** is a variable that stores the memory address of a specific object type
    - Okay--not so bad.
        - So a ‘pointer’ is a data type
        - And it can store objects of a specific data type
            - ^So what exactly does this mean, and how could we do this efficiently?
- If an object is a page in a book
- then a ‘pointer’ would be the index in the back of the book that points you to a specific page.



object: A page in a book

—Index—

–A–

about the author 128, 132, 412
account info 295
active table of contents 34, 120-124, 238-239, 285-286, 354, 366, 370
ACX 465-467
Adobe 506
advertising 434, 439-449
age 312
aggregator 17-18, 322
alignment 68, 101-103, 105-106, 229-230, 261-262, 353-354, 380, 389
Alt codes 39
Amazon Associates 415
Amazon Follow 430, 437, 480
Amazon Giveaway 436-439
Amazon Marketing Services (AMS) 439-449
Android 167-169, 171, 371-375
apostrophe 40, 42-44
app 141-142
Apple 169, 342, 372, 506
automatic renewal 327-329, 341, 343
Automatically Update 73-75, 94, 144
AZK 371

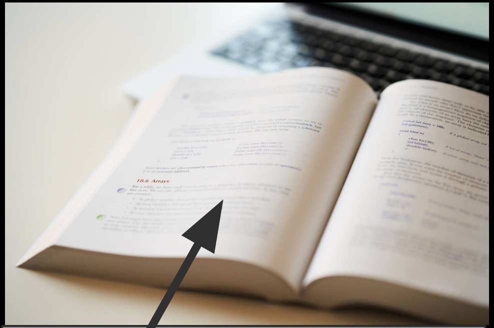
–B–

back matter 124-129
background 47, 93, 181, 184, 192-193, 246, 252-253, 355, 370, 385, 390
bank information 295
Barnes & Noble 506
biography 128, 132, 410
black 47, 93, 184, 192, 252-253, 355, 370, 385, 390
Blackberry 372-373
blank line 27-28, 110, 112-114, 276-277, 284-285, 385
blank page 354, 385-386
block indent 50, 52, 67, 82, 106-107, 234-235
blog 411, 429, 479
Blogger 429
bloggers 327, 430
blurb 300-306, 364, 406, 411-412, 417, 477
blurry 162-164, 172, 175, 193, 246, 387, 389
body text 66, 68, 79-82, 92-94, 115, 233-235

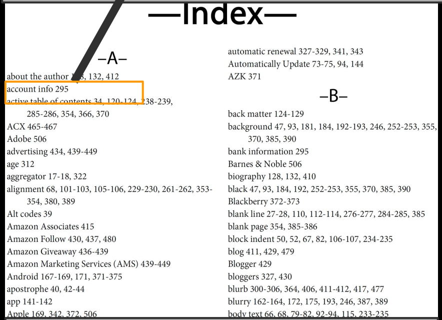
index entry, points to a specific page that exists

A (common) analogy of what a pointer is

- A **pointer** is a variable that stores the memory address of a specific object type
  - Okay--not so bad.
    - So a 'pointer' is a data type
    - And it can store objects of a specific data type
      - ^So what exactly does this mean, and how could we do this efficiently?
- If an object is a page in a book
- then a 'pointer' would be the index in the back of the book that points you to a specific page.



object: A page in a book



—Index—

–A–

about the author 1?8, 132, 412
account info 295
active table of contents 34, 120-124, 238-239, 285-286, 354, 366, 370
ACX 465-467
Adobe 506
advertising 434, 439-449
age 312
aggregator 17-18, 322
alignment 68, 101-103, 105-106, 229-230, 261-262, 353-354, 380, 389
Alt codes 39
Amazon Associates 415
Amazon Follow 430, 437, 480
Amazon Giveaway 436-439
Amazon Marketing Services (AMS) 439-449
Android 167-169, 171, 371-375
apostrophe 40, 42-44
app 141-142
Apple 169, 342, 372, 506
automatic renewal 327-329, 341, 343
Automatically Update 73-75, 94, 144
AZK 371

–B–

back matter 124-129
background 47, 93, 181, 184, 192-193, 246, 252-253, 355, 370, 385, 390
bank information 295
Barnes & Noble 506
biography 128, 132, 410
black 47, 93, 184, 192, 252-253, 355, 370, 385, 390
Blackberry 372-373
blank line 27-28, 110, 112-114, 276-277, 284-285, 385
blank page 354, 385-386
block indent 50, 52, 67, 82, 106-107, 234-235
blog 411, 429, 479
Blogger 429
bloggers 327, 430
blurb 300-306, 364, 406, 411-412, 417, 477
blurry 162-164, 172, 175, 193, 246, 387, 389
body text 66, 68, 79-82, 92-94, 115, 233-235

index entry, points to a specific page that exists

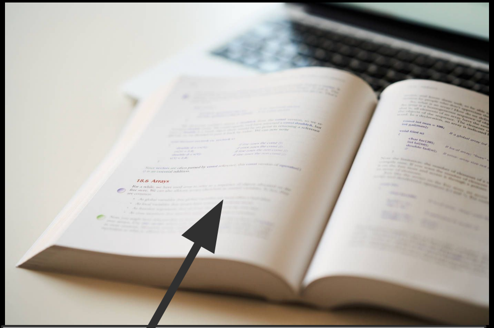
A (common) analogy of what a pointer is

- A **pointer** is a variable that stores the memory address of a specific object type
  - Okay--not so bac
    - So a ‘pointe
    - And it can s data type
      - ^So w mean this ef
- If an object is a pa
- then a ‘pointer’ would be the index in the back of the book that points you to a specific page.

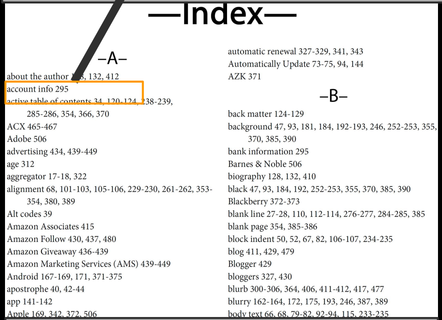
So our index in a book stores a location (i.e., the page number)

In C++, a pointer is thus storing a memory location

Let’s review and visualize memory to get a concrete understanding



object: A page in a book



—Index—

–A–

about the author , 132, 412
account info 295
active table of contents 34, 120-124, 138-239, 285-286, 354, 366, 370
ACX 465-467
Adobe 506
advertising 434, 439-449
age 312
aggregator 17-18, 322
alignment 68, 101-103, 105-106, 229-230, 261-262, 353-354, 380, 389
Alt codes 39
Amazon Associates 415
Amazon Follow 430, 437, 480
Amazon Giveaway 436-439
Amazon Marketing Services (AMS) 439-449
Android 167-169, 171, 371-375
apostrophe 40, 42-44
app 141-142
Apple 169, 342, 372, 506
automatic renewal 327-329, 341, 343
Automatically Update 73-75, 94, 144
AZK 371

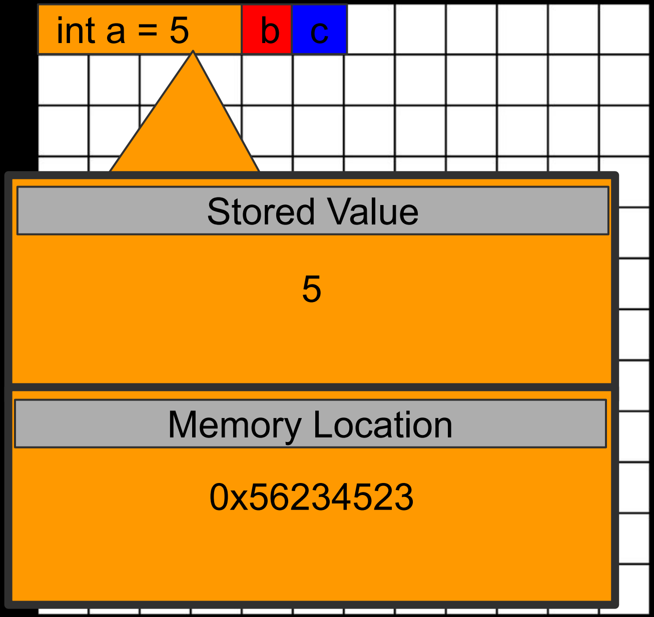
–B–

back matter 124-129
background 47, 93, 181, 184, 192-193, 246, 252-253, 355, 370, 385, 390
bank information 295
Barnes & Noble 506
biography 128, 132, 410
black 47, 93, 184, 192, 252-253, 355, 370, 385, 390
Blackberry 372-373
blank line 27-28, 110, 112-114, 276-277, 284-285, 385
blank page 354, 385-386
block indent 50, 52, 67, 82, 106-107, 234-235
blog 411, 429, 479
Blogger 429
bloggers 327, 430
blurb 300-306, 364, 406, 411-412, 417, 477
blurry 162-164, 172, 175, 193, 246, 387, 389
body text 66, 68, 79-82, 92-94, 115, 233-235

index entry, points to a specific page that exists

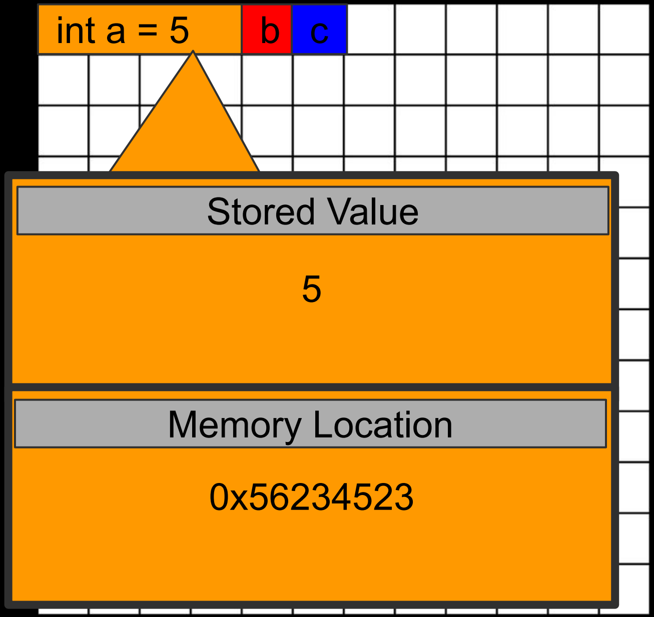
Zooming into our memory (each individual rectangle)

- Each piece of memory has a value, and the address (in hexadecimal) where it lives.



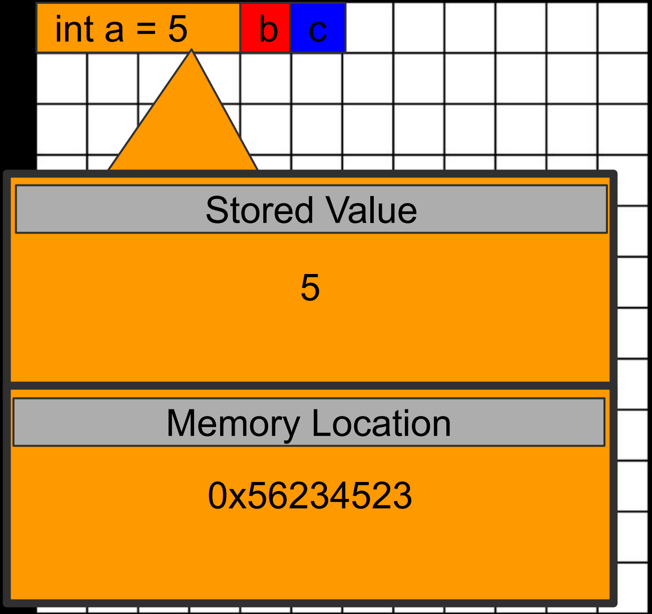
Returning the address of Memory Location (1/3)

- We can retrieve that address using the ‘&’ operator.
- Ampersand (&) you can think of as ‘address of’
  - (i.e. “hey, tell me where in memory ‘a’ lives)



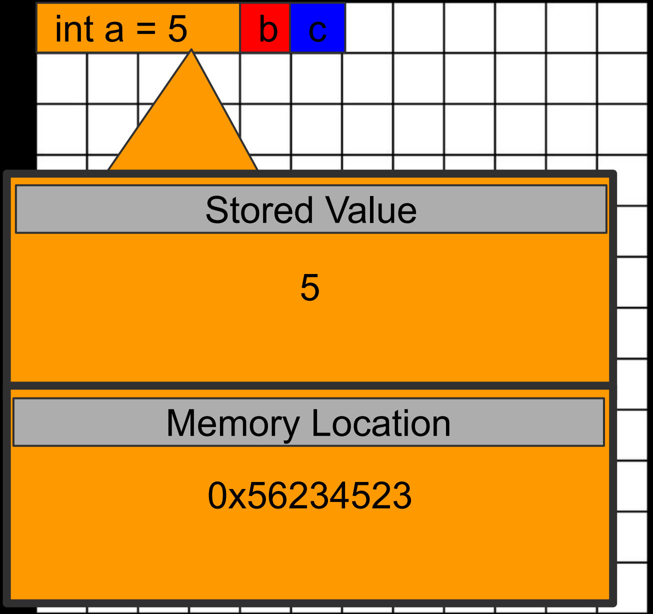
Returning the address of Memory Location (2/3)

- We can retrieve that address using the ‘&’ operator.
- Ampersand (&) you can think of as ‘address of’
- ‘Address of’ gives you the exact location in memory, just like a mailbox.



Returning the address of Memory Location (3/3)

- We can retrieve that address using the ‘&’ operator.
- Ampersand (&) you can think of as ‘address of’
- ‘Address of’ gives you the exact location in memory, just like a mailbox.



The actual content (or value) is stored in the mailbox

Further resources and training materials

- Pointers
  - Back to Basics: Pointers and Memory by Ben Saks (CPPCON 2020)
- Smart Pointers
  - Back to Basics: Smart Pointers by Arthur O’Dwyer (CppCon 2019)
  - Back to Basics: Smart Pointers by Rainer Grimm (CppCon 2020)
  - Back to Basics: Smart Pointers and RAII by Inbal Levi (CPPCON 2021 on Thursday)

Back to Basics: Pointers

Mike Shah, Ph.D.

@MichaelShah | mshah.io | www.youtube.com/c/MikeShah

Thank you Cppcon attendees, reviewers, chairs!

Thank you!