Pointers

COP 3223C – Introduction to Programming with C

Fall 2025

Yancy Vance Paredes, PhD

Recall

- Previously, we wrote functions that accepted arrays as arguments

- We observed that when an array is passed to a function, the function can modify the original array

- This is different from when a function receives an ordinary variable: any changes inside the function do not affect the original

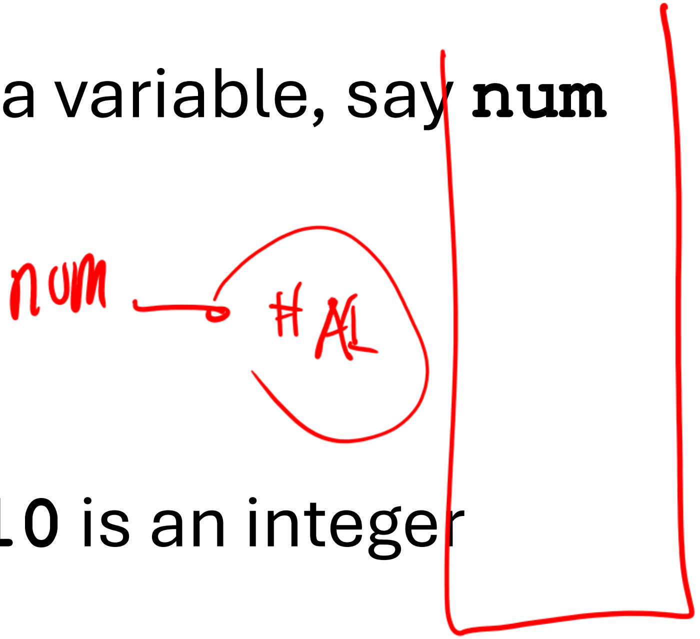
foo(x)

Discussion /1

int num ;

num = 10;

- Suppose I want to store the integer value **10** in my program
- Because I want to refer to it later, I can declare a variable, say **num**
- I then assign the value **10** to the variable **num**
- We declare **num** as an **int** because the value **10** is an integer



Discussion /2

&num ⇒ #A1 address

addr

- Every variable name is an alias for a specific memory location
- To find the exact location, we use the **&** (address-of operator)
- This gives us the address of the variable **num**, which in turn, is also another kind of value
- If the address is also a value, can we store it somewhere too?

Discussion /3

- Just like the value **`10`**, we can also store the address of a variable
- We will need to declare another variable that can hold an address
- However, what data type can store a memory address?

Pointer

A variable that stores the **memory address** of another variable

It’s value then is a location or a reference

Syntax:

```
int *ptr;          int num;

char *qtr;         char alpha;

double *r;         double amt;
```

Interlude

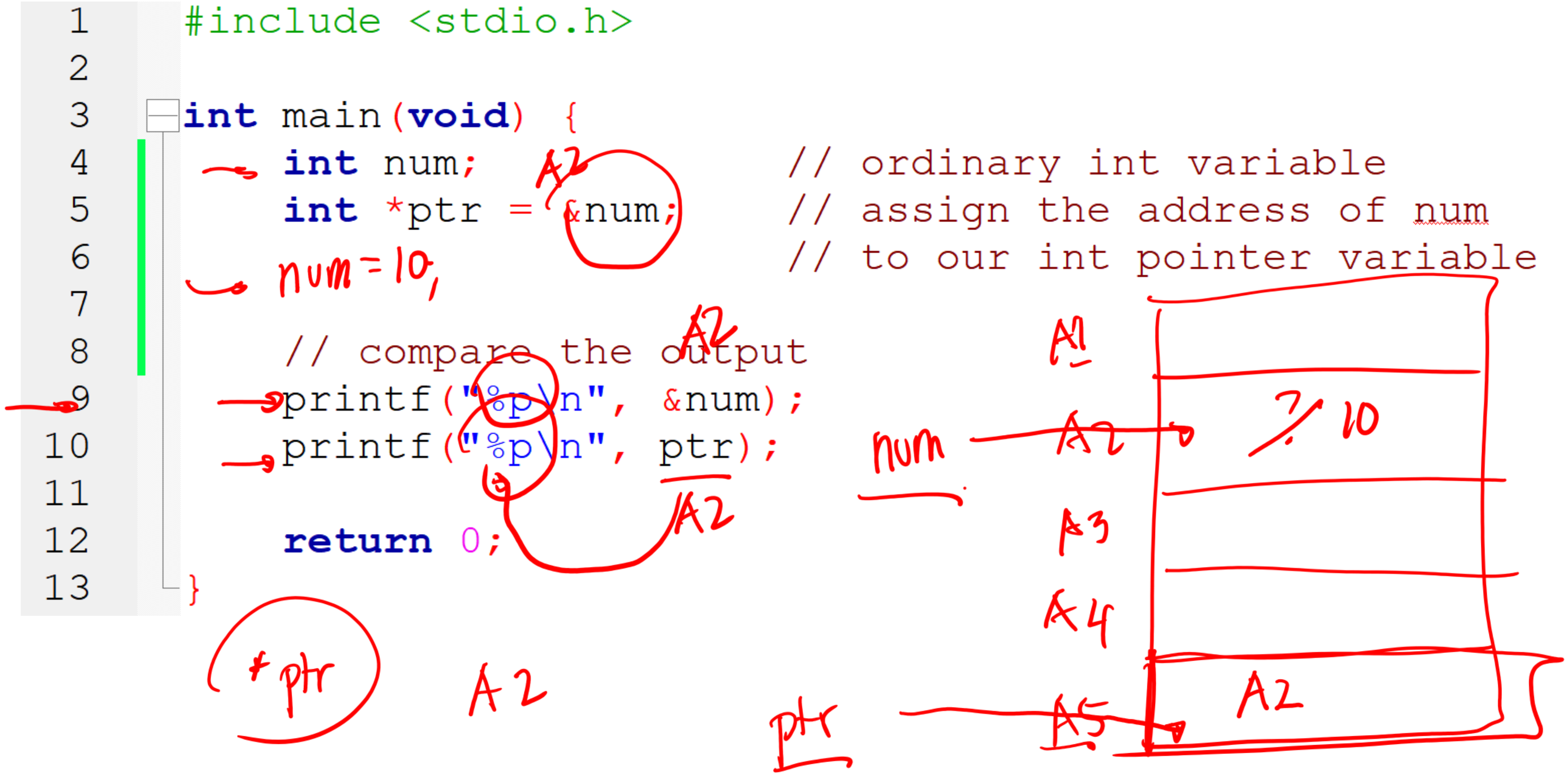
- Keep in mind that understanding pointers is a means to an end, not the end goal itself

- Once we grasp the core idea behind pointers, many advanced C concepts will start to make much more sense

Code Tracing /1

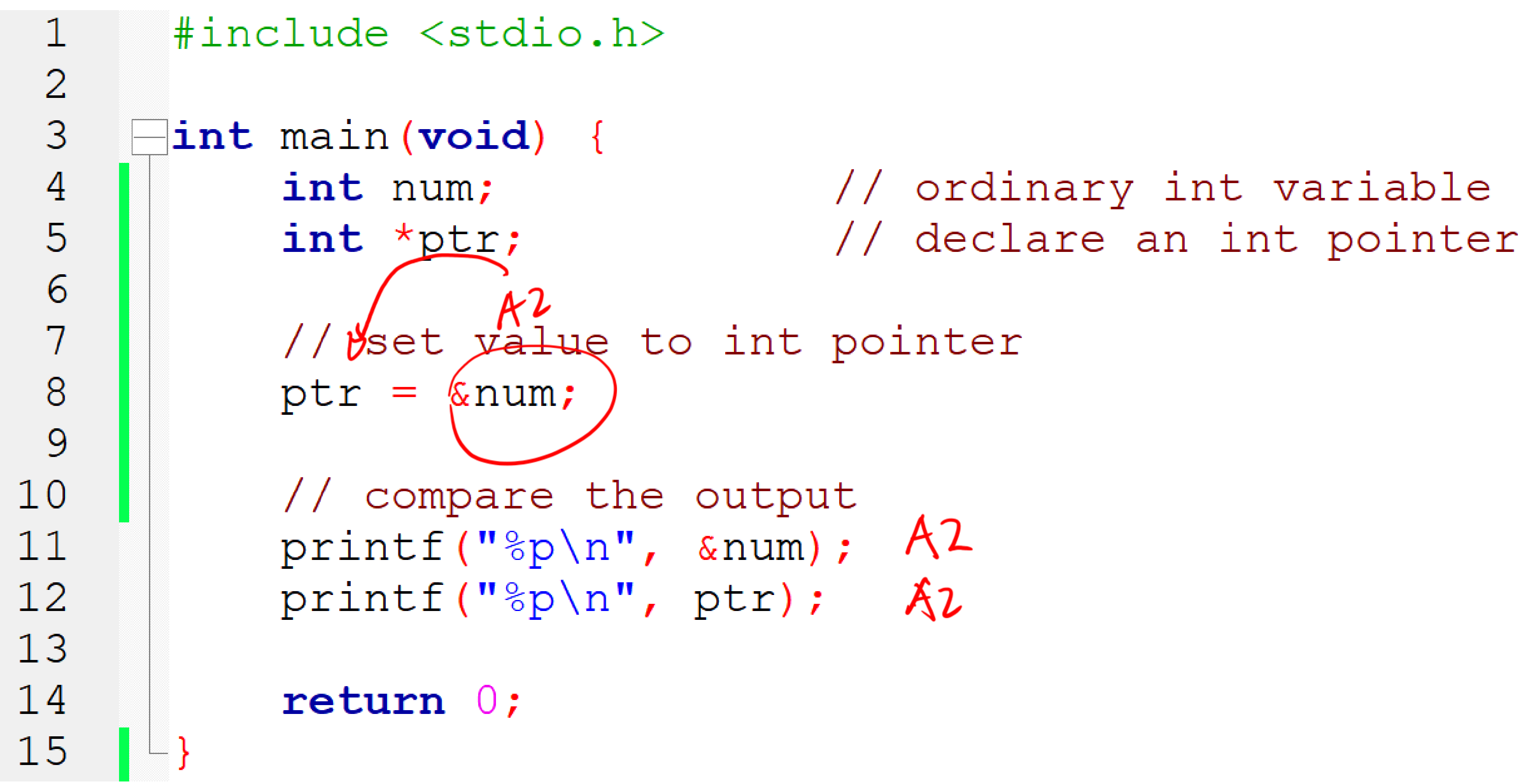
A2

A2



```
#include <stdio.h>

int main(void)  {
    int num;                // ordinary int variable
    int *ptr = &num;        // assign the address of num
                            // to our int pointer variable

    // compare the output
    printf("%p\n", &num);
    printf("%p\n", ptr);

    return 0;
}
```

Code Tracing /2



```c
#include <stdio.h>

int main(void) {
    int num;                    // ordinary int variable
    int *ptr;                   // declare an int pointer

    // set value to int pointer
    ptr = &num;

    // compare the output
    printf("%p\n", &num);
    printf("%p\n", ptr);

    return 0;
}
```

Notes /1

int *ptr;   ✓ int* ptr;   ✓int * ptr;

int *p, *q;

- You may encounter variations of how to declare a pointer

int *p, *q, r;

- What happens when we do multiple declarations on a single line?

- Remember: * binds to the variable name (not to the data type)

int *ptr, *ptr2, num, num2;

address address whole number whole number

pointers

ordinary int variable

int* a, b;

pointer

ordinary int variable

Notes /2

- The ***** (asterisk) we saw is **not** the multiplication operator

- It is important to understand the distinction between typing a pointer's variable name alone versus including the ***** before the variable name

- This is a confusing concept for beginners of C

Notes /3

It is important to put in mind that the following are different:

```
int num;
int *ptr;
```

dereference operator

✓ ptr vs *ptr ✓

Discussion

If we want to print the value of a pointer variable (i.e., address it currently holds), we use the format specifier **`%p`** for **`printf()`**

Accessing Values

If you have a memory address, it is possible to access its value

How?

*ptr

Dereference Operator /1

int *ptr;

- If you want to access the value of a particular memory location, use the * (dereference operator)

- Don’t confuse it with the multiplication operator

ptr #A1

- Yes, they both use the **asterisk**!

*ptr

- Also, don’t confuse it with the * during the declaration!

Dereference Operator /2

- How to interpret it depends on context

- Typically, based on which side of the assignment statement*

Dereference Operator /3

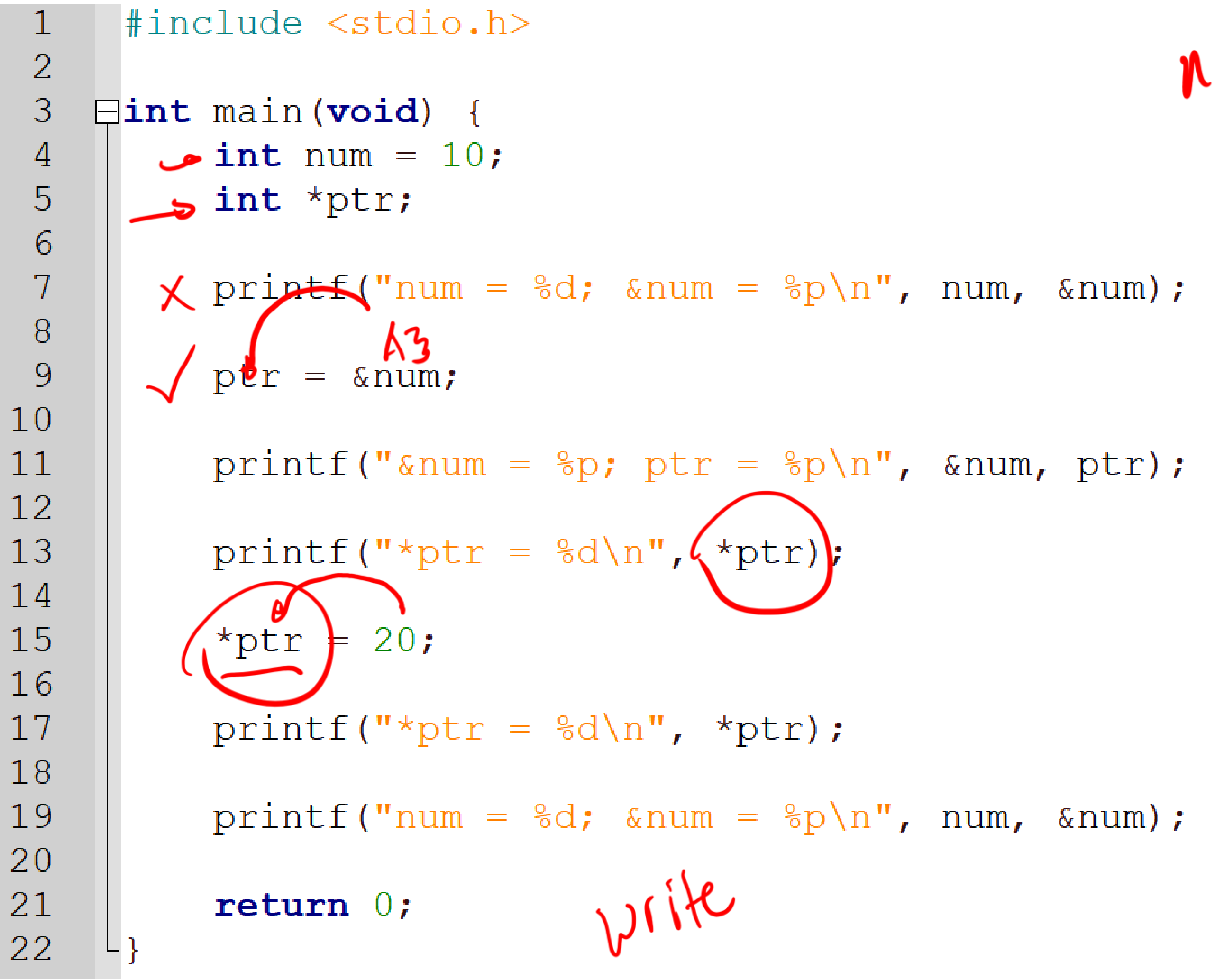
Left side of assignment statement

> *"I want to change the value of this memory location."*

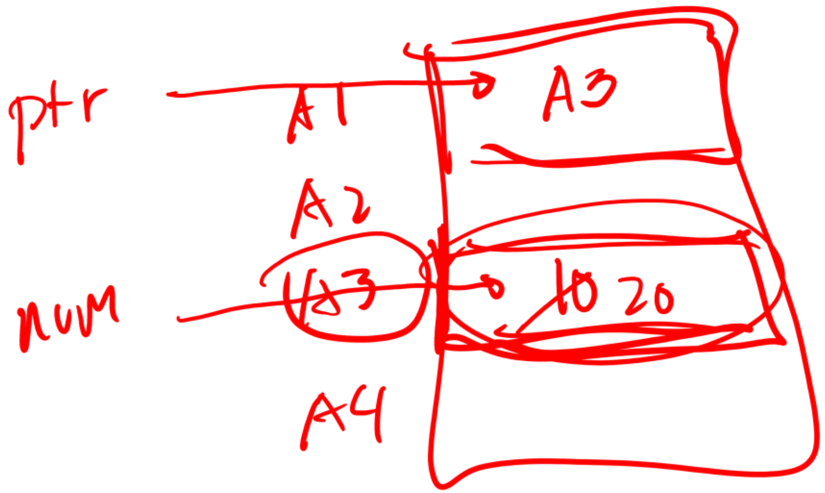
Right side of assignment statement (or by itself)

> *"I want to know what the value is at this memory location."*

Code Tracing



```c
#include <stdio.h>

int main(void) {
    int num = 10;
    int *ptr;

    printf("num = %d; &num = %p\n", num, &num);

    ptr = &num;

    printf("&num = %p; ptr = %p\n", &num, ptr);

    printf("*ptr = %d\n", *ptr);

    *ptr = 20;

    printf("*ptr = %d\n", *ptr);

    printf("num = %d; &num = %p\n", num, &num);

    return 0;
}
```



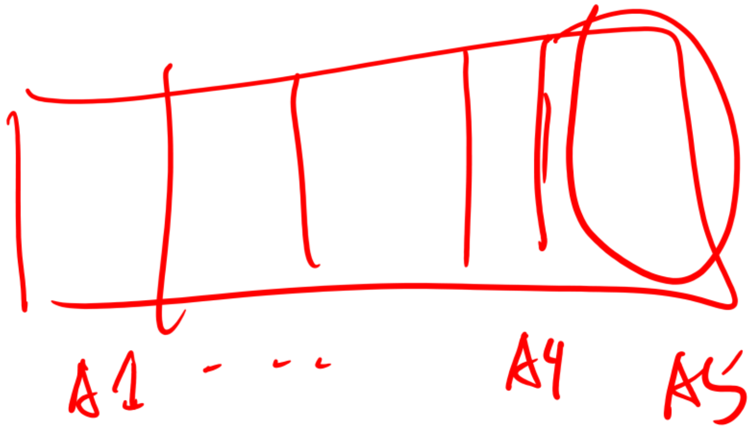
ptr
⇓
A3
⇓
10

*ptr
⇓
*(A3)
⇓
10

Discussion

C provides us the ability to **directly manipulate memory**

“With great power comes great responsibility”



int x;
x = 10;
&x
x

Scene from Spider-Man (2002): https://youtu.be/guuYU74wU70?si=J0mixcG7e0l0rgk4&t=71

Caution

- We want to be careful as we are dealing with memory addresses

- Possible to “access” an invalid memory location

- At times, we don’t have permission to access certain locations

Discussion /1

What happens if the data type of the variable is different? For example, **`char`** instead of **`int`**?

```
char alpha;

char *q;

q = &alpha;
```

```
double num;

double *r = &num;
```

Discussion /2

Can you use pointer as an argument to a **`scanf()`** function call?

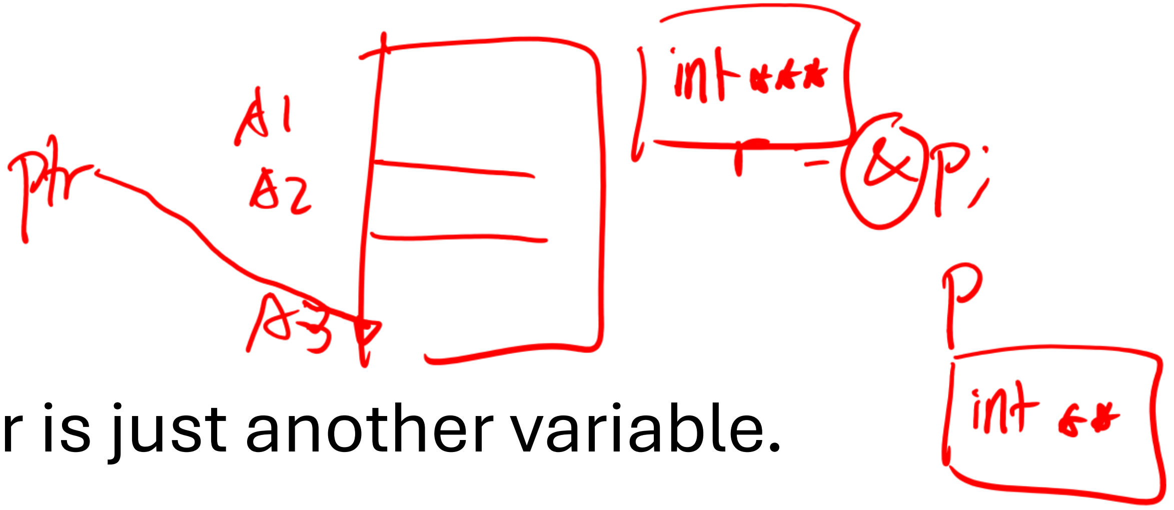
Notes

The **`scanf()`** function stores user input directly into a variable’s memory location

Because a pointer variable holds an address, we can use this as an argument to the function

We just have to ensure that the data type is correct

Discussion /1



- We already know that a pointer is just another variable.
- It serves as an alias to a specific memory location, storing an address rather than a value.
- Like any other variable, a pointer itself also resides in memory; therefore, it too has its own address.

int ***r;

r = &q;

int **q;

dt

int *p;

A3

dt

int x;

A1

Discussion /2

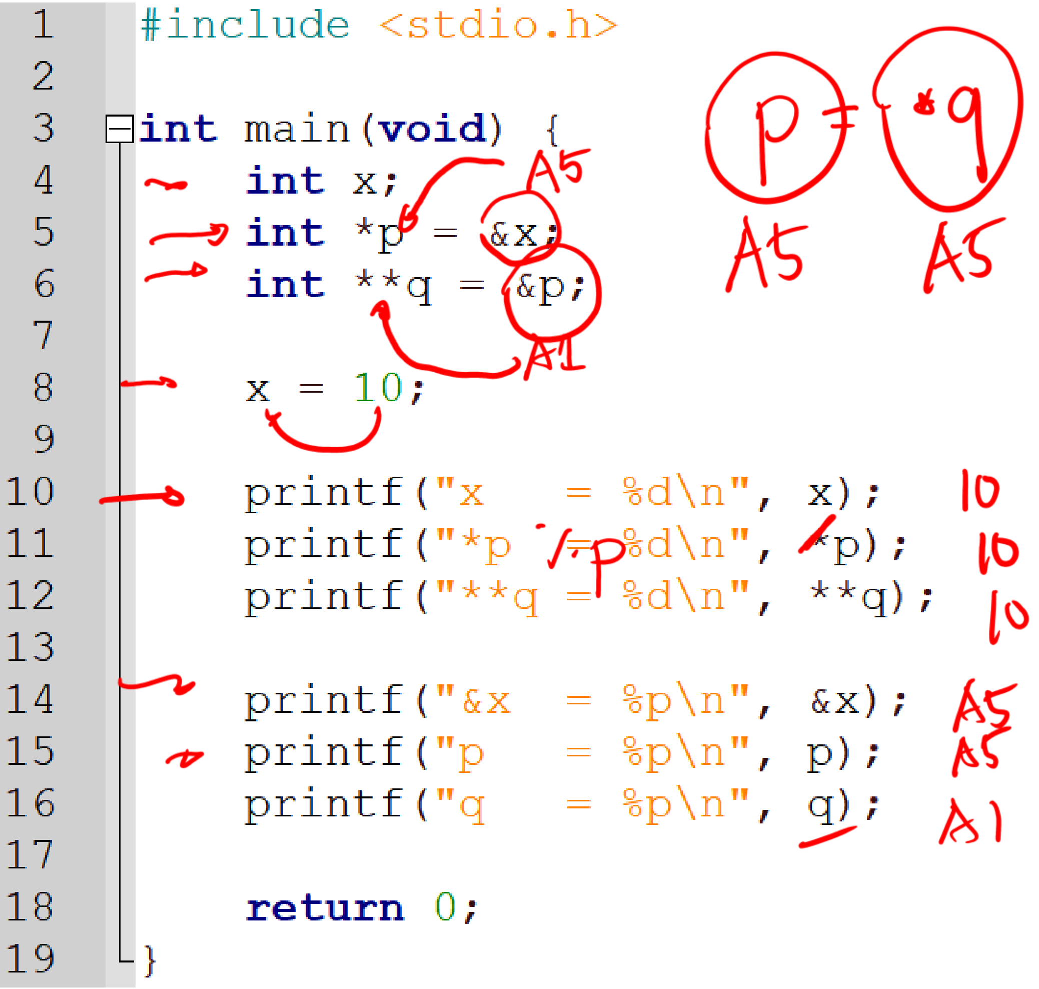
- We can access this address using the **&** operator.

- Can we have another pointer that stores this address?

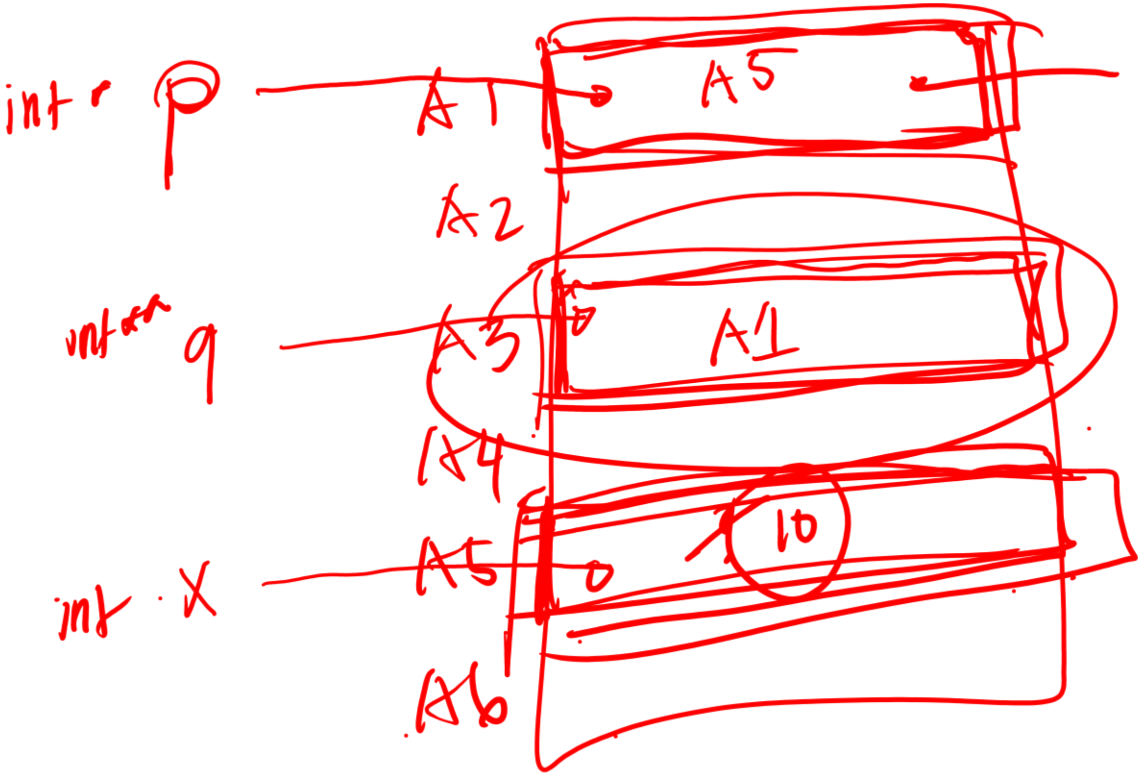
Pointer to a Pointer

A pointer that holds the memory address of another pointer variable

Code Tracing



```c
#include <stdio.h>

int main(void) {
    int x;
    int *p = &x;
    int **q = &p;

    x = 10;

    printf("x   = %d\n", x);
    printf("*p  = %d\n", *p);
    printf("**q = %d\n", **q);

    printf("&x  = %p\n", &x);
    printf("p   = %p\n", p);
    printf("q   = %p\n", q);

    return 0;
}
```



*(A5)
⇓
10

q
⇓
*(A1)
⇓
*(A5)
⇓
10

Notes /1

If we have a pointer to a pointer, such as **`int **q`**, and we want to access the value ultimately pointed to by that chain, we need to dereference twice

In the previous example, **`*q`** evaluates to the value stored in **`q`**, which is the address of **`p`** or in other words, **`*q`** gives you **`p`**

Notes /2

`**q` or **`*(*q)`** dereferences again accesses the value pointed to by **`p`**, which is the value of **`x`**

Thus, **`**q`** gives you the value of **`x`**

Common Runtime Errors /1

int num;

Wrong combination during declaration and initialization of pointer

✓ int *ptr = &num;

✗ int *ptr = num; ←

Forgetting to use the dereference operator

//set variable

✓ *ptr = 20;

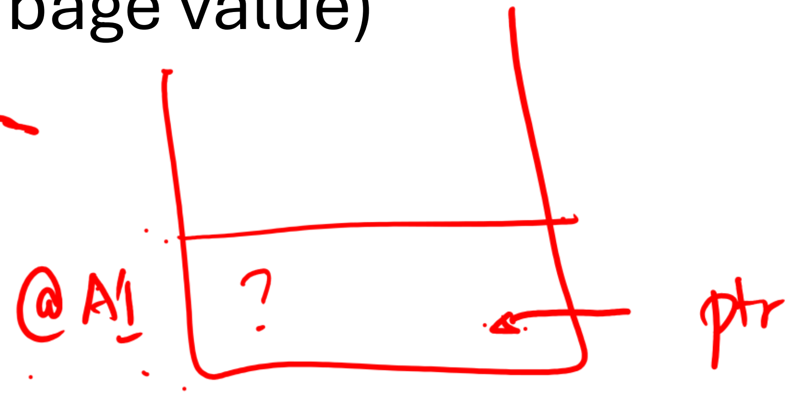
✗ ptr = 20; x ←

Common Runtime Errors /2

```
int *ptr = 1000;
printf("%d", *ptr);
```

Accessing an incorrect address (e.g., garbage value)

```
→ int *ptr;   1000
→ printf("%d", *ptr);
```



Again, think about the **`scanf()`** where you forgot to use the **`&`**

```
int num = 10;
scanf("%d", &num);
    scanf("%d", num);
                ↳ @10
```

The **`NULL`** Value

- If we want to assign a default and safe value to a pointer
- In reality, this is just a macro constant of **`0`**
- Good practice to **initialize** pointer to **`NULL`** when not used
- At times, it is good to set pointers to **`NULL`** when a task is done

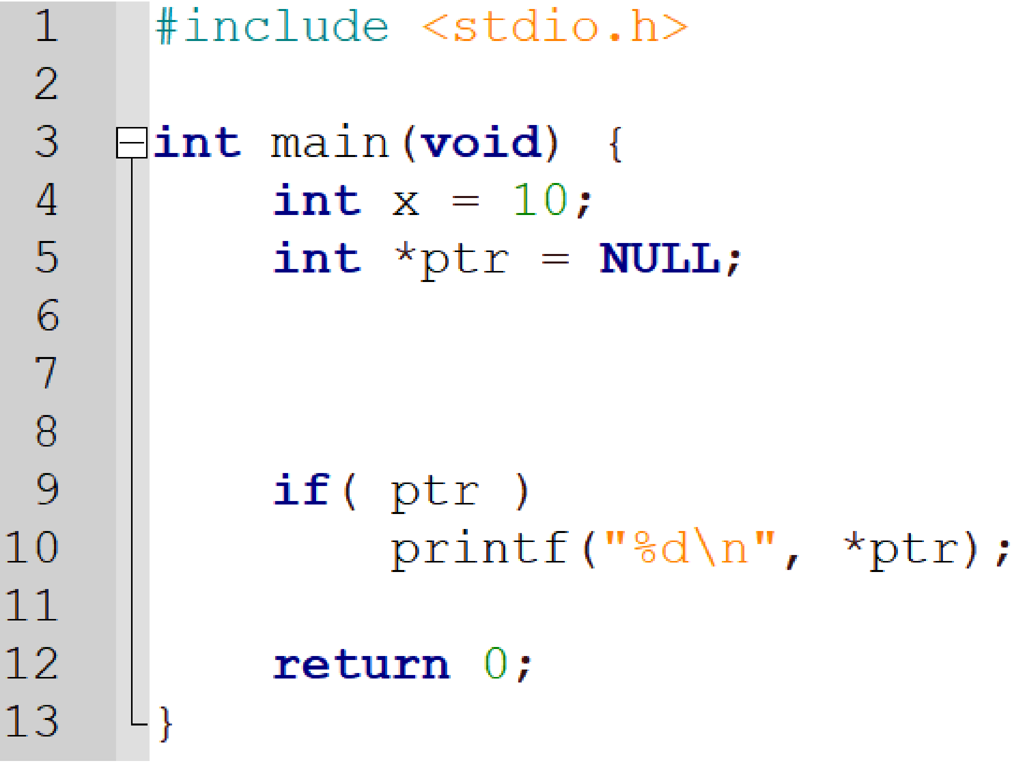
Practice

- Declare and initialize a pointer to NULL

- See what the output is when we print the pointer

- Can we dereference it?

Discussion



```c
#include <stdio.h>

int main(void) {
    int x = 10;
    int *ptr = NULL;



    if( ptr )
        printf("%d\n", *ptr);

    return 0;
}
```

Recall: User-Defined Functions

- We learned how to write our own functions
- Some functions accept arguments while some do not
- We have written functions that accept regular variables
- Can a function have a pointer as one of its parameters?

```
int x;
foo(x)
```

```
int arr[5];
bar(arr);
```

Practice /1

Write a **`void`** function called **`increment(*ptr)`** that accepts an address of an **`int`** variable (pointer). This function increases the value stored at that memory location by **`1`**.

Practice /2

Write a **`void`** function named **`swap()`** that takes as parameters the memory addresses of two **`int`** variables and exchanges the values stored at those addresses.

Recall

Previously, we implemented the **selection sort** algorithm

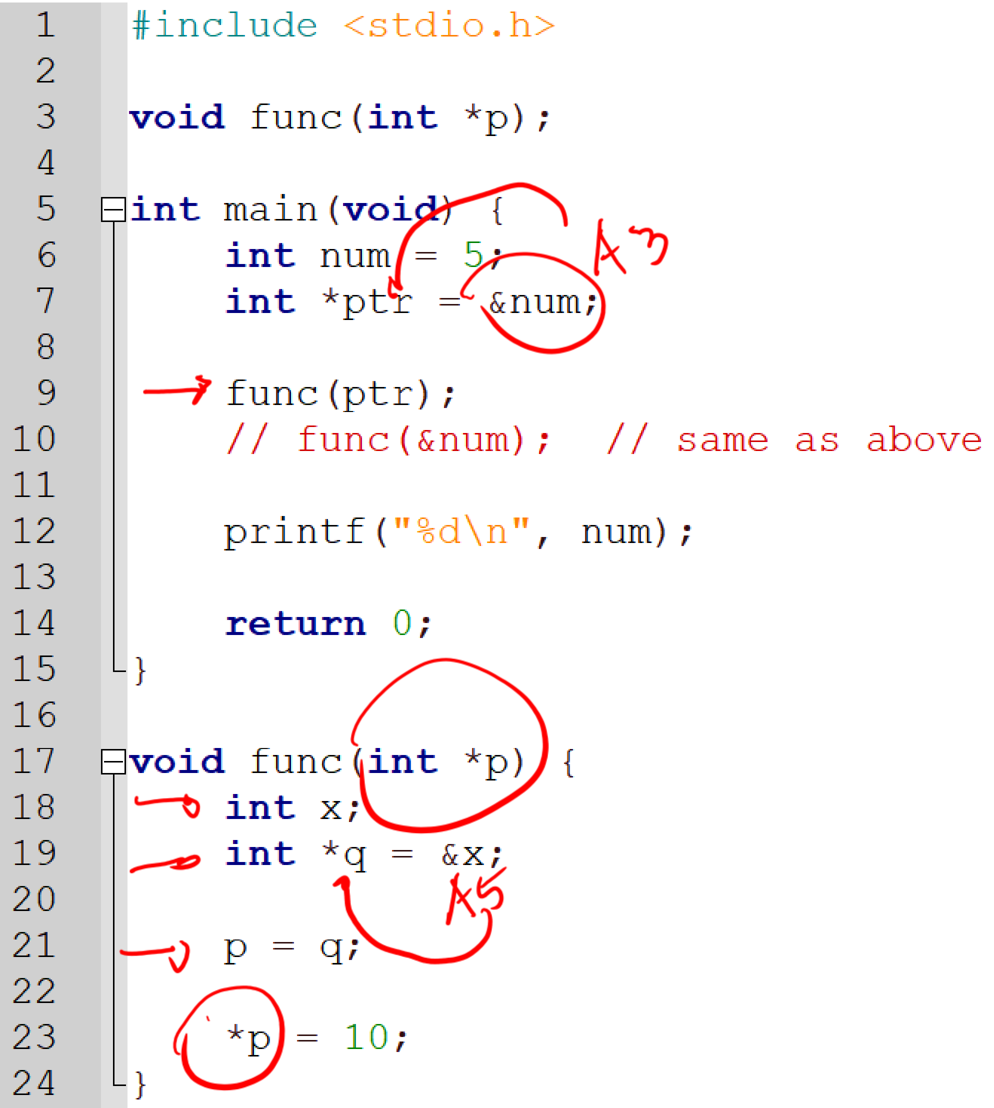
One key operation is **swapping** the values stored at two locations

Now that we have written a reusable **`swap()`** function, can we use it to simplify our selection sort implementation?

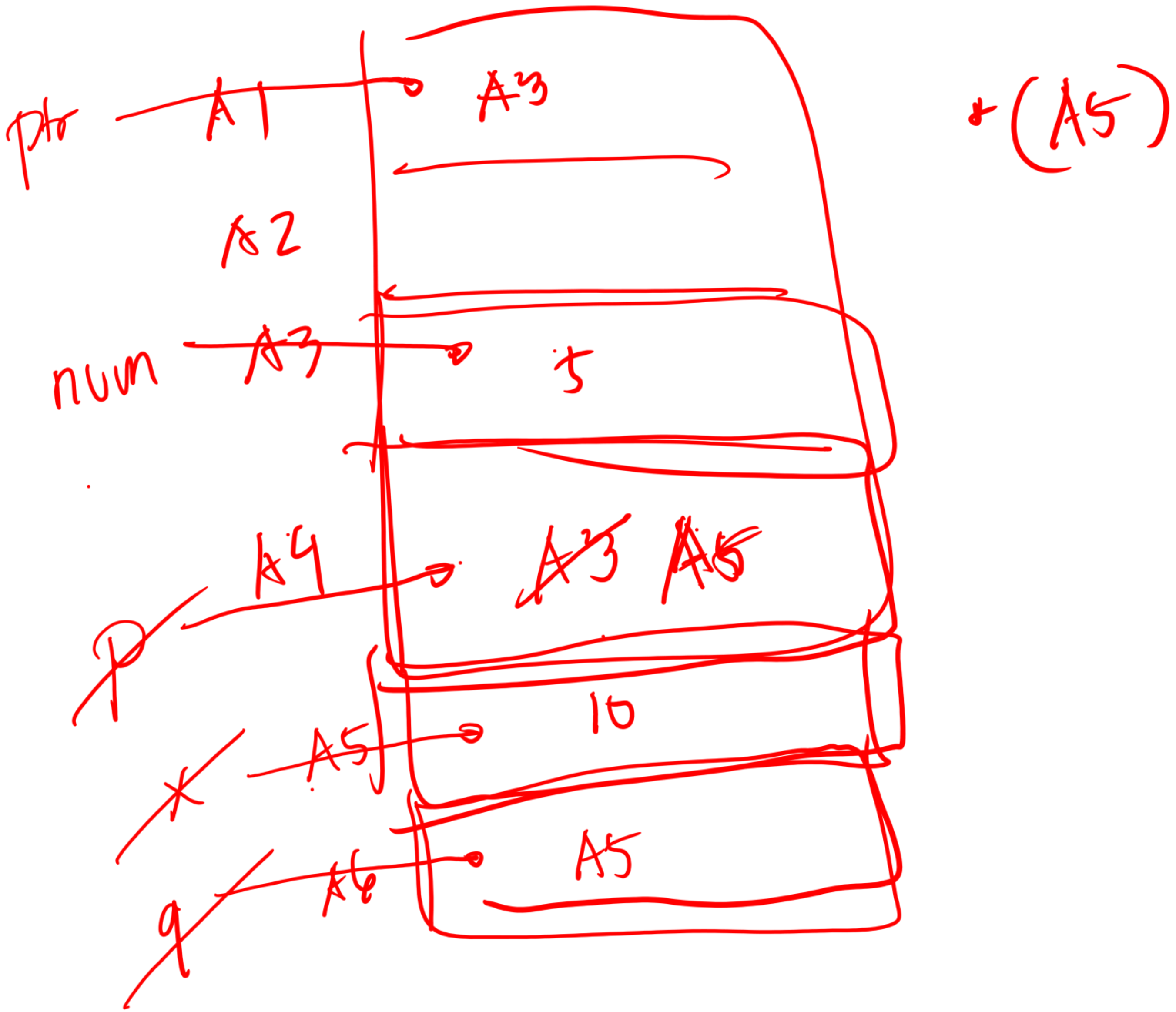
Discussion

What happens if we perform an assignment operation without the dereference operator?

Code Tracing



```c
#include <stdio.h>

void func(int *p);

int main(void) {
    int num = 5;
    int *ptr = &num;

    func(ptr);
    // func(&num);  // same as above

    printf("%d\n", num);

    return 0;
}

void func(int *p) {
    int x;
    int *q = &x;

    p = q;

    *p = 10;
}
```



Notes

- The behavior observed in the previous example is **expected** because a pointer simply holds an address
- When a pointer is assigned a new address, it no longer refers to the original variable it once pointed to; therefore, the link is lost
- This behavior is often most noticeable in functions, where assigning a new address to a parameter pointer does not affect the caller’s original variable

Your Turn!

Write a **`void`** function named **`capitalize()`** that takes a pointer to a **`char`** as its parameter. The function should modify the character it points to so that it becomes an uppercase letter. You may assume the input will always be an English alphabet letter.

Discussion /1

- Let’s take a moment to think about what happened in our program

- We saw that a function was able to *change* the value of a variable that wasn’t even inside its own **scope**

- Simply because we **passed** the variable’s **address** to that function

Discussion /2

- When a function is given the address of a variable, it can directly modify the value stored in that variable

- This technique is often referred to as **pass-by-reference**, because the argument passed to the function is an **address** (or reference) rather than the actual value

Recall: Arrays /1

- We previously learned about arrays in C

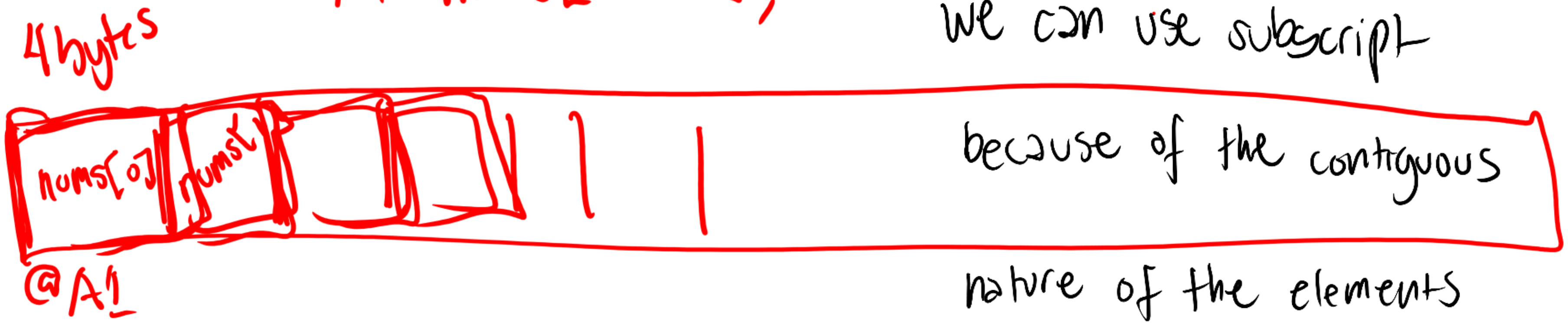
- When declaring an array, we must specify its size

- It is a contiguous block in the memory

int nums[100];

nums[0] or nums[1]

we can use subscript because of the contiguous nature of the elements



Recall: Arrays /2

- We access each element by specifying the index in the `[ ]`

- Also, we learned that each element has a specific address

- Further, we found that the name of the array *decays* to the address of the first element

Notes

- When an array is passed, we are passing its **address**

- This is a **crucial concept** that you may want to always remember!

Discussion /1

- An array’s name decays to the address of its first element.
- If we want to store that address, we can use a pointer.
- Since array elements are stored contiguously, we can compute the address of the next element from the current one.

Discussion /2

- In our example, we saw that the next element was 4 bytes away (i.e., **`int`** data type on my machine).
- Wouldn’t it be nice if C could handle that size calculation automatically?
- That way, we could just move to the next element instead of worrying about bytes.

Pointer Arithmetic

- In C, we can perform arithmetic operations on pointers to move between elements in memory.
- When we add or subtract an integer to a pointer, the pointer moves by that many elements, not bytes.
- The compiler automatically scales the movement by the size of the data type the pointer refers to.

Practice

Write a program that prints all the addresses of all the elements of an array.

Compare it with actual addresses of the same array.

Revisiting: Functions and Arrays /1

When an array is passed, behind the scenes, no new array is created

Rather, it is only the address that is passed, therefore there is only one array that exists in the memory

Revisiting: Functions and Arrays /2

Knowing that it is an address is passed to the function, we can rewrite the function’s prototype

Revisiting: Functions and Arrays /3

How do we access the individual elements if we are receiving a pointer?

Practice

Write a function named **`print_array_v2()`** that accepts a pointer to an array of integers (i.e., the address of its first element) and the number of elements. It then prints the elements, one per line.

Discussion

For a simple 1D pointer parameter:

```
arr[i] == *(arr + i)
```

Refer to the visualization on the next slide

	`*(arr + 0)`	`*(arr + 1)`	`*(arr + 2)`	`*(arr + 3)`	`*(arr + 4)`	access the value of an element
dereference the sample addresses	`*(0xAA + 0)`	`*(0xAA + 1)`	`*(0xAA + 2)`	`*(0xAA + 3)`	`*(0xAA + 0)`	
	`*(0xAA)`	`*(0xAE)`	`*(0xB2)`	`*(0xB6)`	`*(0xBA)`	
	`arr[0]`	`arr[1]`	`arr[2]`	`arr[3]`	`arr[4]`	
int arr[5]	10	20	30	40	50	
	`&arr[0]`	`&arr[1]`	`&arr[2]`	`&arr[3]`	`&arr[4]`	addresses of each elements
sample addresses in hexadecimal	`0xAA`	`0xAE`	`0xB2`	`0xB6`	`0xBA`	
	`0xAA + 0`	`0xAA + 1`	`0xAA + 2`	`0xAA + 3`	`0xAA + 4`	
	`arr + 0`	`arr + 1`	`arr + 2`	`arr + 3`	`arr + 4`	

Notes /1

- When an array variable name is passed to a function, it **decays to a pointer** to its first element

- Therefore, if we expect a **formal parameter** to receive an array, the formal parameter can be declared as a pointer

- However, you must be **cautious** when tracing the code as this function is assuming that an array is passed when it is called

Notes /2

- Notice the difference between passing the address of an ordinary variable vs the address of the array?

- How about the formal parameter, can it distinguish what is being passed?

Notes /3

- Arrays and pointers are closely related, but they are not the same
- Pointer arithmetic works because both arrays and pointers deal with contiguous memory

Notes /4

- The pointer itself is not an array it simply holds the address of the array’s first element

- This idea will become even more important later when we discuss **dynamic memory allocation**

Discussion

What if we combine the dereference operator and the increment or decrement operator?

Code Tracing /1

```
int arr[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int *p = arr;

printf("%d\n", *p++);
printf("%d\n", *p);
```

Code Tracing /2

```
int arr[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int *p = arr;

printf("%d\n", (*p)++);
printf("%d\n", *p);
```

Code Tracing /3

```
int arr[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int *p = arr;

printf("%d\n", *++p);
printf("%d\n", *p);
```

Code Tracing /4

```
int arr[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int *p = arr;

printf("%d\n", ++*p);
printf("%d\n", *p);
```

Notes

Be mindful of the order of precedence especially when dealing with pointer arithmetic

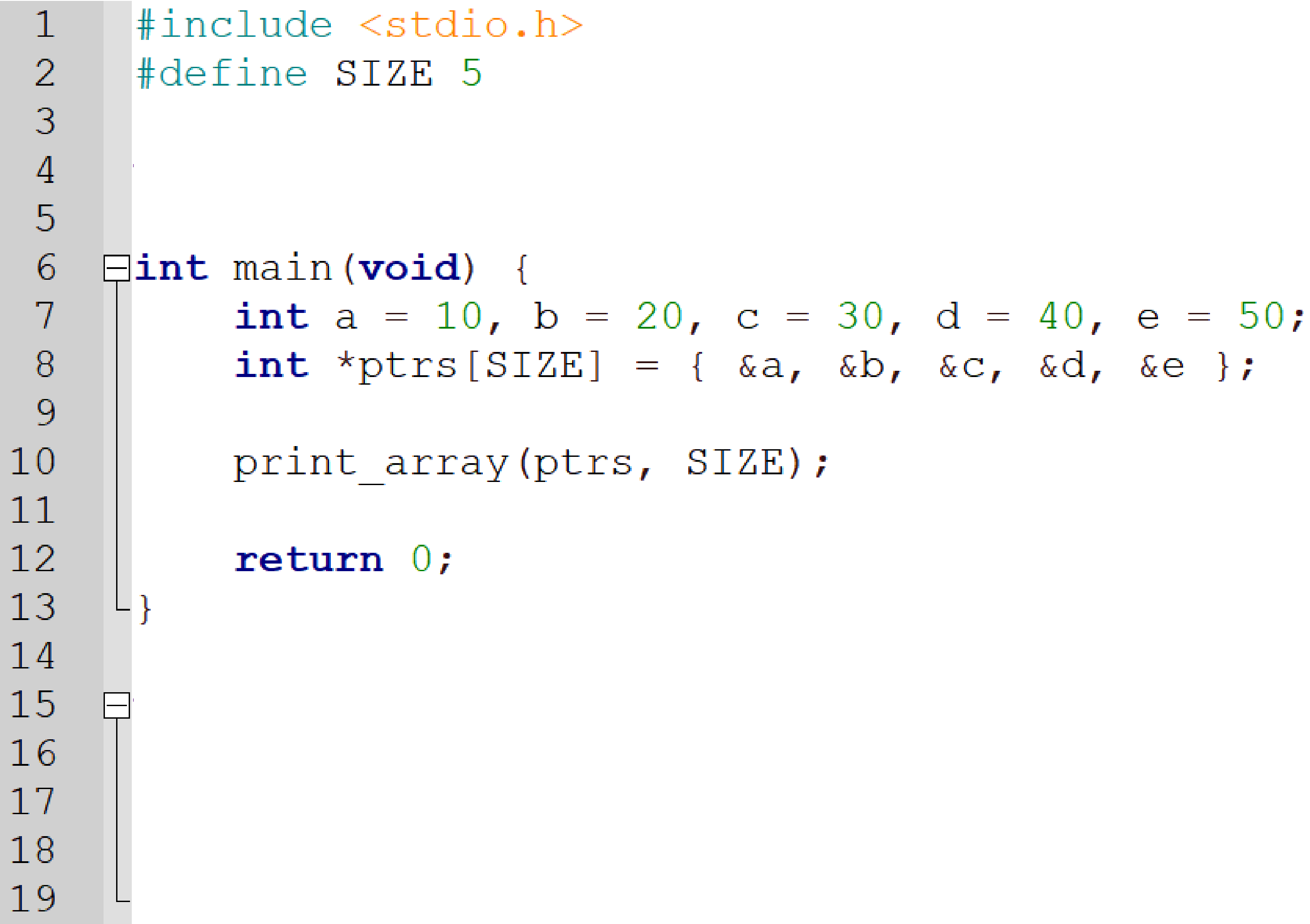
Your Turn!

With your current understanding of pointers, try experimenting with pointers that reference **`char`** variables. For example:

```
char letters[5] = { 'a', 'b', 'c', 'd', 'e' };
char *p;
```

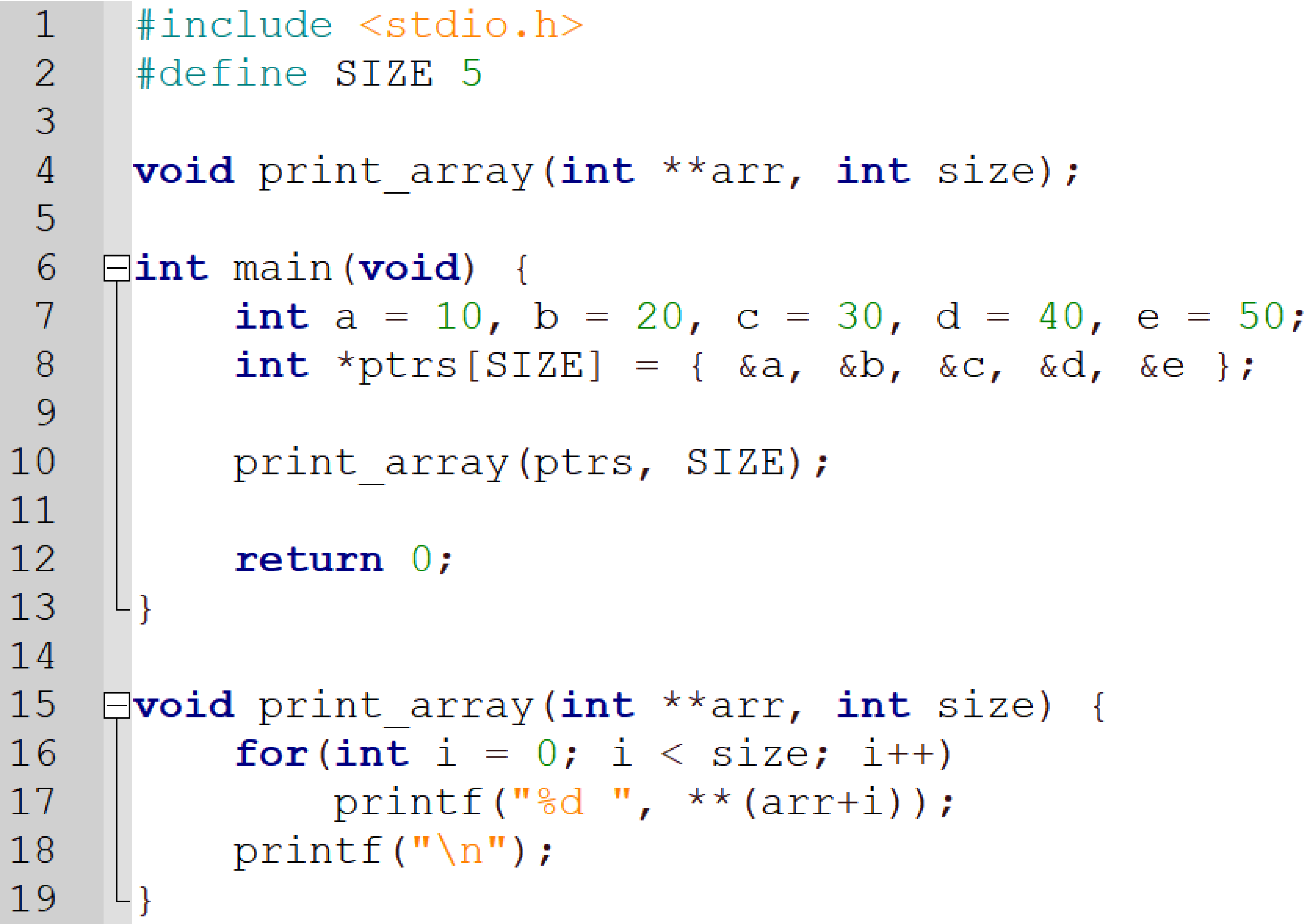
Challenge /1

Define a function named **`print_array()`** that receives an array of integer pointers and its size, then prints the values stored in the pointed variables. The expected output is: **`10 20 30 40 50`**



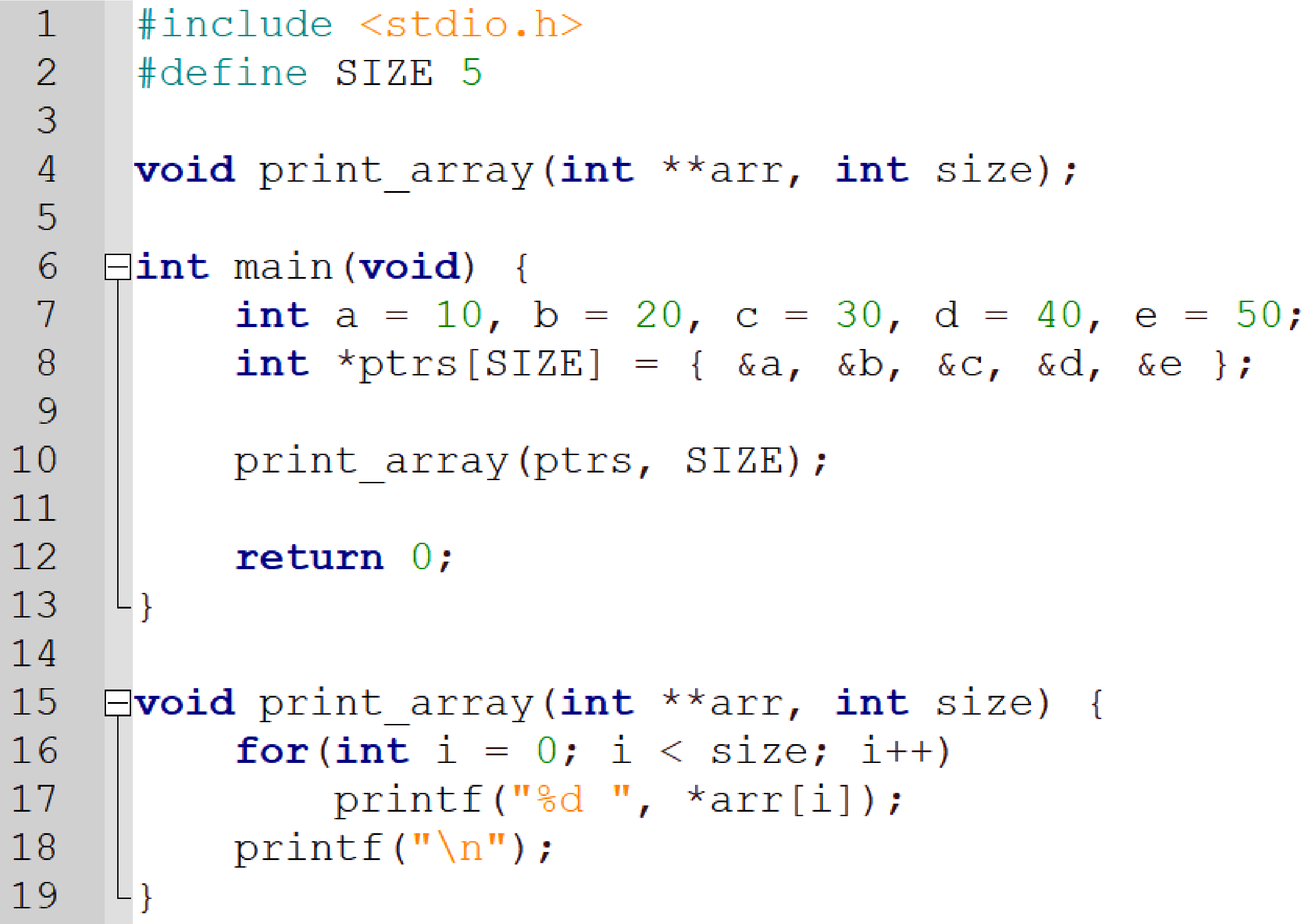
```c
#include <stdio.h>
#define SIZE 5



int main(void) {
    int a = 10, b = 20, c = 30, d = 40, e = 50;
    int *ptrs[SIZE] = { &a, &b, &c, &d, &e };

    print_array(ptrs, SIZE);

    return 0;
}
```

Challenge /2



```c
#include <stdio.h>
#define SIZE 5

void print_array(int **arr, int size);

int main(void) {
    int a = 10, b = 20, c = 30, d = 40, e = 50;
    int *ptrs[SIZE] = { &a, &b, &c, &d, &e };

    print_array(ptrs, SIZE);

    return 0;
}

void print_array(int **arr, int size) {
    for(int i = 0; i < size; i++)
        printf("%d ", **(arr+i));
    printf("\n");
}
```

Challenge /3



```c
#include <stdio.h>
#define SIZE 5

void print_array(int **arr, int size);

int main(void) {
    int a = 10, b = 20, c = 30, d = 40, e = 50;
    int *ptrs[SIZE] = { &a, &b, &c, &d, &e };

    print_array(ptrs, SIZE);

    return 0;
}

void print_array(int **arr, int size) {
    for(int i = 0; i < size; i++)
        printf("%d ", *arr[i]);
    printf("\n");
}
```

Final Notes

- Remember, the concept of pointers is a means to an end

- It helped us understand how data is stored, accessed, and manipulated in the memory

- So, keep in mind the strategies we learned as we will revisit and apply them later

Questions?