Before Lecture

- Quick review over last lecture (pointers)
  - What a pointer is
    - Just a variable! Just like an int, char, float…just takes a different type of value
      - This box can only take oranges, that box can only take apples etc

Pointers, PT II

Lecture Overview

- Lecture
  - Pass by Value vs Pass by Reference
  - Why use pointers?
    - Swap Function
- Before We Code
  - Pointers to Pointers
  - Pointer Arithmetic
- Sample Program

LECTURE

Pass by Value vs Pass by Reference

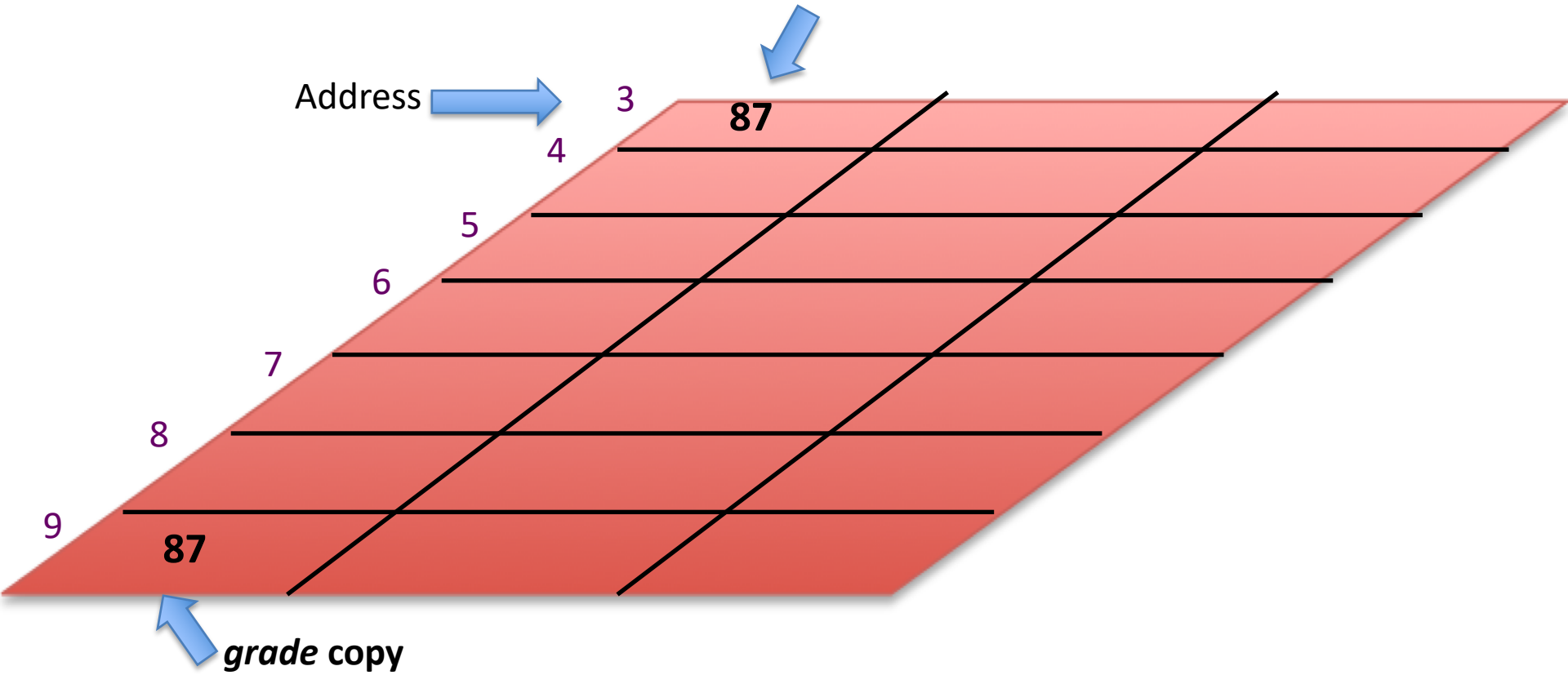
- One characteristic of a programming language is whether it is a language that **passes by value** or **passes by reference**
- What does this mean?
  - Simply, is a parameter used in a function a copy of a variable? Or is it the actual variable?

Pass by Value vs Pass by Reference

- C is considered a pass by value language
  - So all parameters passed into functions are using copies of variables, not the actual variables

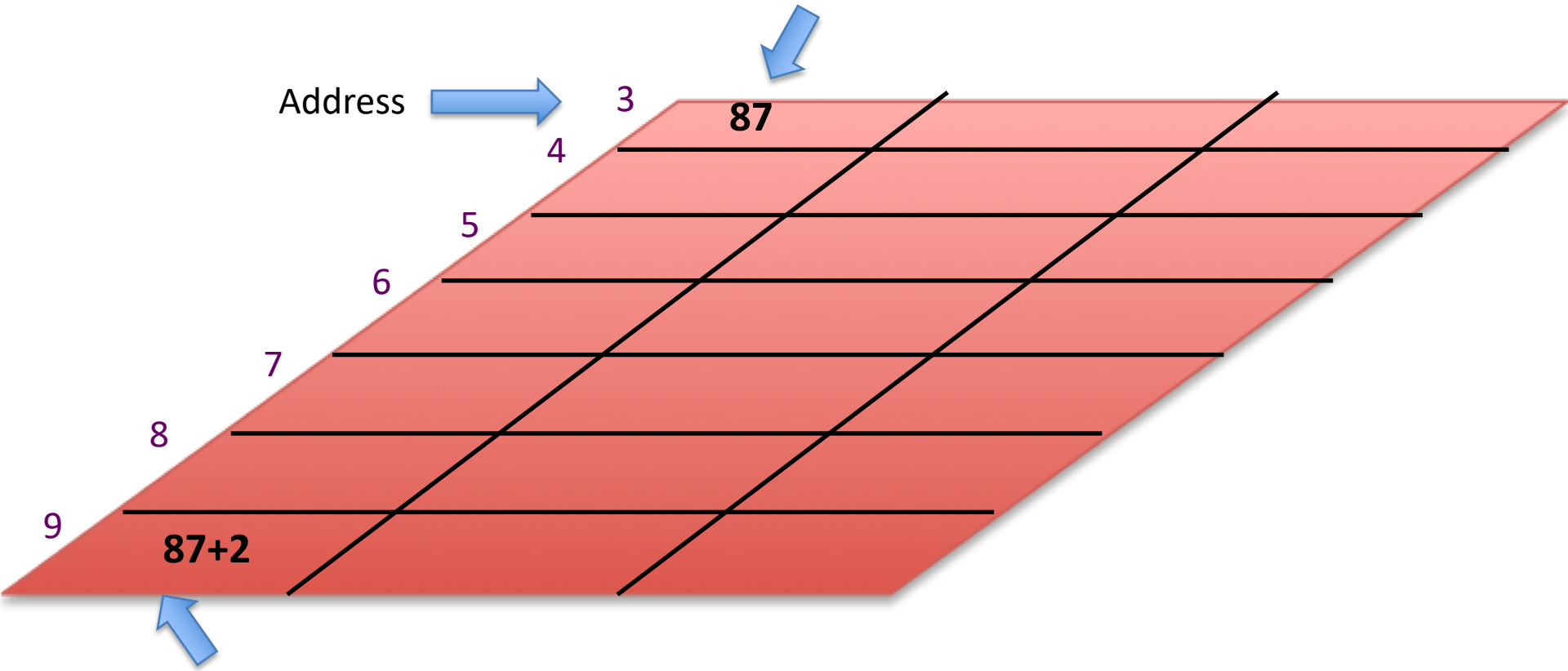
- Imagine you had a function like:
  void add_two(int grade);
  - You can think of it like this (next slide):

This is your original value of *grade* in your program:



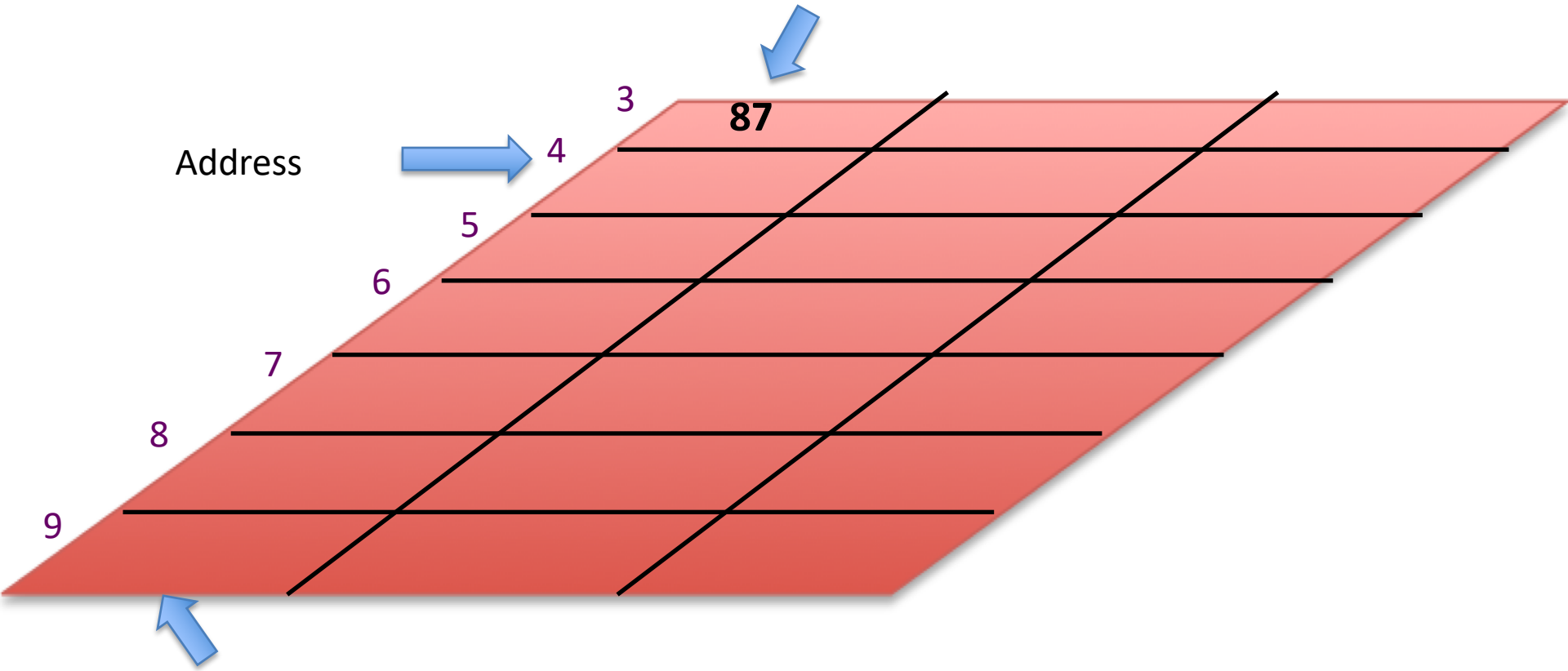
The function makes a copy of *grade (when you pass it as a parameter)* and does any changes to THAT COPY ALONE.

This is your original value of *grade* in your program:



The function makes a copy of *grade* and does any changes to THAT COPY ALONE (example: add 2 to the value. Two is being added to the copy, not the actual value of grade)

This is your original value of *grade* in your program (still the same):



When the function is finished, the copy is done. As you can see, no effect has been made on the original value of 87 in your program.

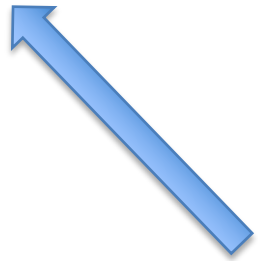
If you are dealing with one value, you could return that new value from the function. But what if you are dealing with two values?

Pass by Value vs Pass by Reference

How would we handle a problem like this?

Create a function that takes two students grades and swaps them.

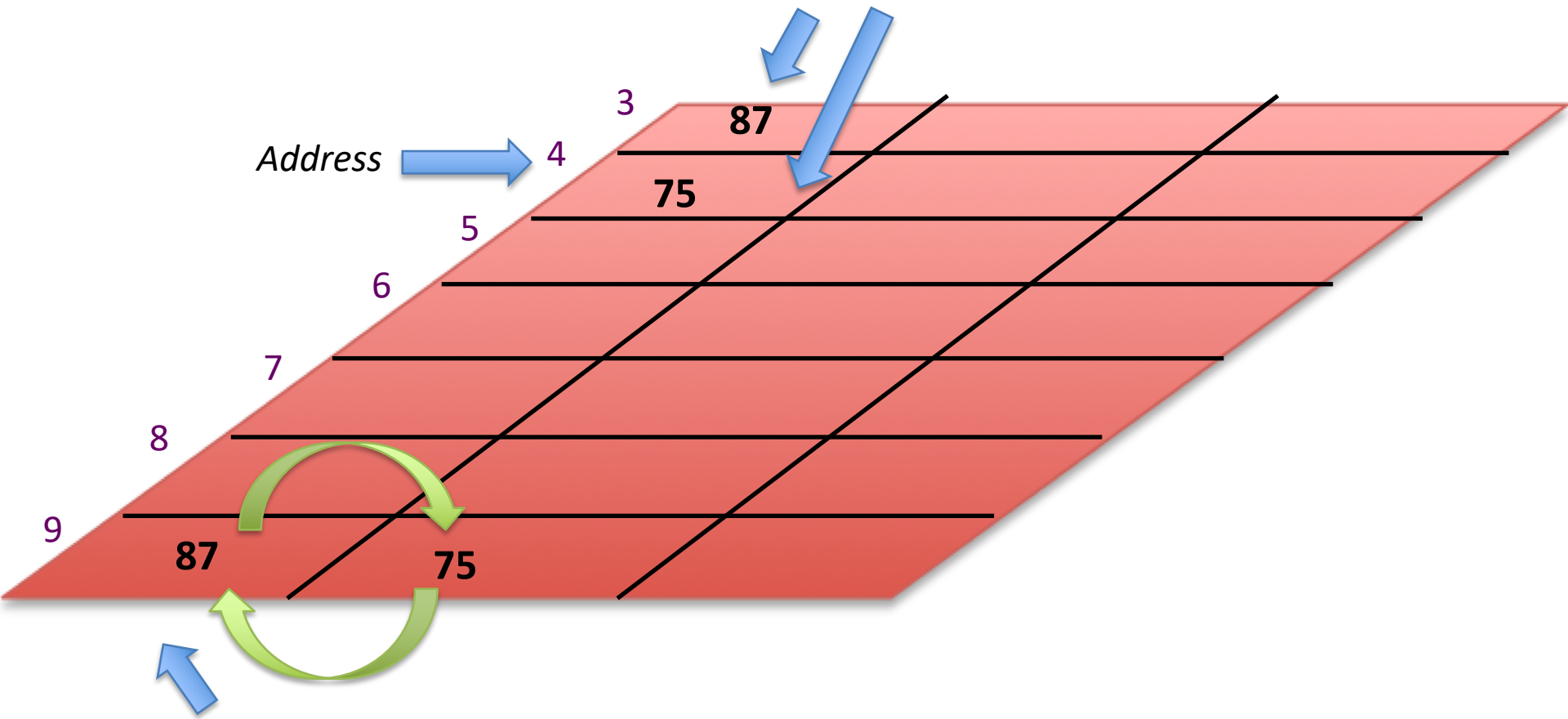
```
void swap_grades (int g1, int g2)
{
        int temp;

        temp=g1;
        g1=g2;
        g2=temp;

}
```



We are modifying copies of g1 and g2, not g1 and g2 themselves.

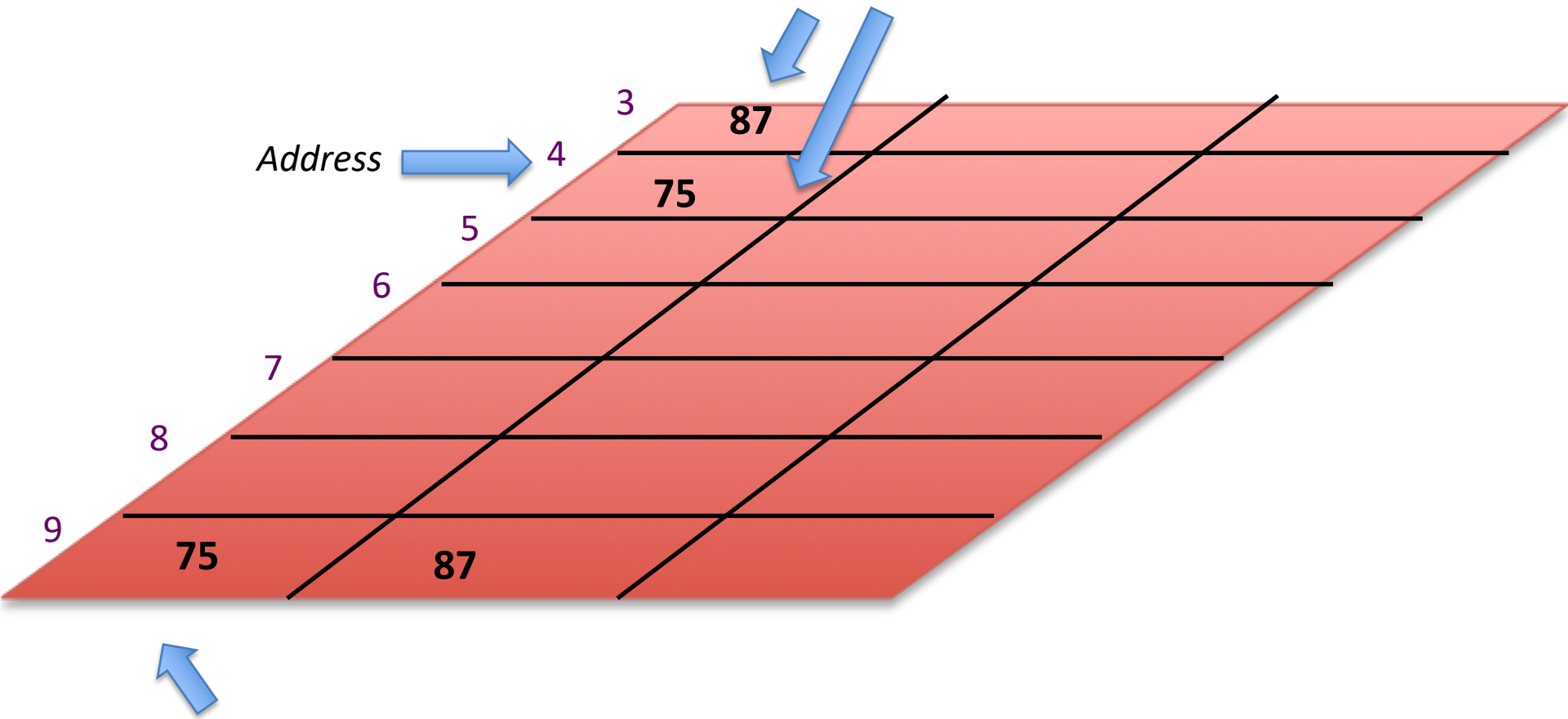
That means that after the function call is over, the old values of g1 and g2 will remain the same in the actual program

These are your original grades in the program:



When we swap the values, we are only swapping the copies-NOT the actual values themselves.

These are your original grades in the program:



These swapped values will only work for the duration of the FUNCTION. Once that function call is over, we will go back to our original values (that will NOT be swapped).

Note that we can't return both values.

Pass by Value vs Pass by Reference

How would we handle a problem like this? **We can use pointers**

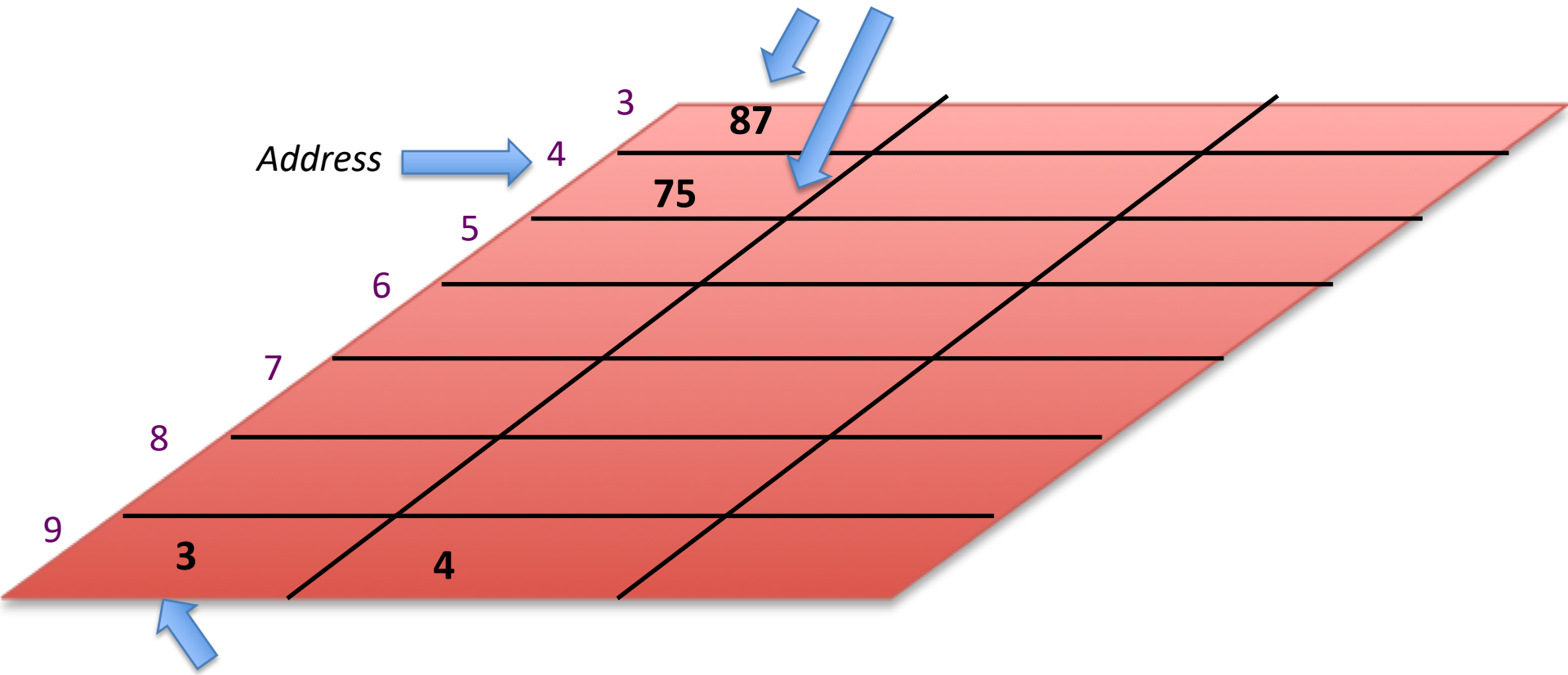
Create a function that takes two student grades and swaps them.

```
void swap_grades (int *g1, int *g2)
{
        int temp;

        temp=*g1;
        *g1=*g2;
        *g2=temp;

}
```

By using pointers, we can directly change the values in memory.

We are not just changing copies.

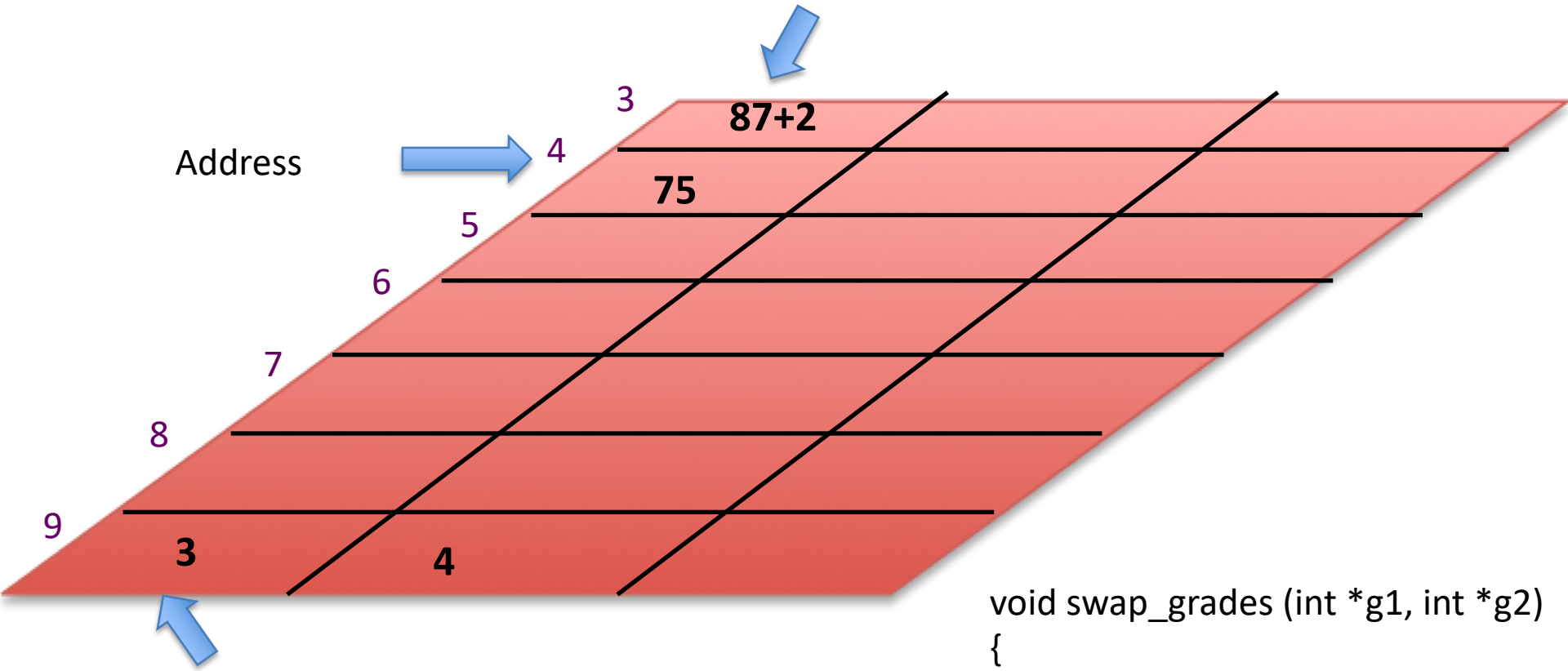
These are your original grades in the program:



We're passing in copies of the addresses of the original values

When we dereference the copies of the addresses, we end up accessing the original values.

These are your original grades in the program:

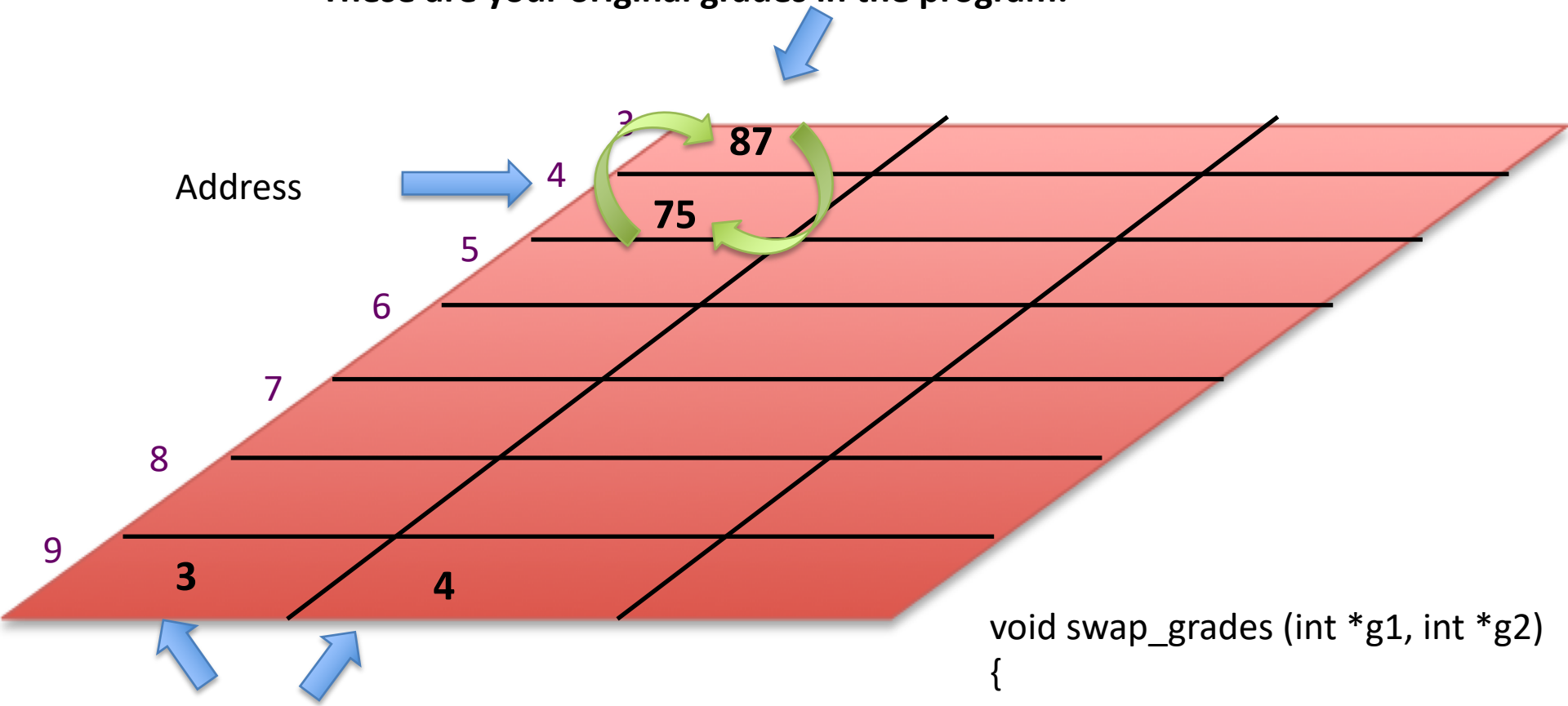


This is g1.
Dereferencing g1 means actually accessing what is at address 3 (the actual value of 87)

For example, if I said: *g1=*g1+2; I am changing the actual value.

```
void swap_grades (int *g1, int *g2)
{
        int temp;

        temp=*g1;
        *g1=*g2;
        *g2=temp;

        }
```

These are your original grades in the program:



This is g1.
Dereferencing g1 means actually accessing what is at address 3 (the actual value of 87)

So when I am swapping, I am now swapping the original values.

```
void swap_grades (int *g1, int *g2)
{
        int temp;

        temp=*g1;
        *g1=*g2;
        *g2=temp;

        }
```

Pass by Value vs Pass by Reference

- By using pointers, we are passing in addresses of variables, not variables themselves
  - In this way, we can overcome the pass by value issue with swapping grades.
  - Note with arrays we are always passing in the addresses-that's why we don't have to return anything

Pass by Value vs Pass by Reference

- A general thing to think of with the variables:
  - Am I just using them to help do some computation (and I don't want to actually change them?
    - Pass by value concept is fine
  - Or am I trying to actually modify the variable itself?
    - We want to actually access the variable-we can pass in the address instead
      - The address will be a copy, but we can simply derive the actual value from that and work from there

Note: for most of the semester (and HWs) I will follow the above concept, but sometimes I am just trying to get you guys to practice with pointers.

```c
#include <stdio.h>

int main()
{
	int num=3;

	int *ptr_one=&num;
	int *ptr_two=ptr_one; /*ptr_one holds an address, so we can assign it to a pointer variable*/

	printf("Value in ptr_one: %p, ptr_one deref: %d\n", ptr_one, *ptr_one);
	printf("Value in ptr_two: %p, ptr_two deref: %d\n", ptr_two, *ptr_two);
}
```

Output:

Value in ptr_one: 0x7fff6e57dbb8, ptr_one deref: 3
Value in ptr_two: 0x7fff6e57dbb8, ptr_two deref: 3

Notice it doesn't matter where the address is coming from- when you dereference the address, you get the value you are looking for.

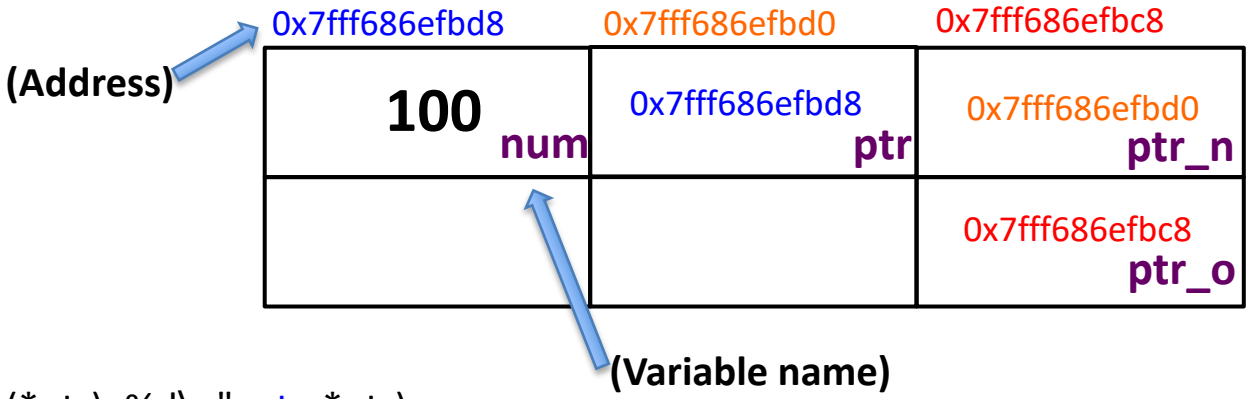
BEFORE WE CODE

Pointers to Pointers

- You can hold pointers to pointers
  - Remember a pointer is a variable that holds an address
    - Since the pointer is a variable itself, it has an address
  - We can hold the pointer's address in another pointer
- You can do this many different times (see next slide)

Pointers to Pointers

```
int num=100;
int * ptr=&num;
int **ptr_n=&ptr;
int ***ptr_o=&ptr_n;
```




```
printf("value in ptr: %p,   ptr deref: (*ptr): %d\n", ptr, *ptr);
printf("value in ptr_n: %p, ptr_n deref: (*ptr_n): %p\n", ptr_n, *ptr_n);
printf("value in ptr_o: %p, ptr_o deref: (*ptr_o): %p\n", ptr_o, *ptr_o);
```

Output:

value in ptr: 0x7fff686efbd8,   ptr deref: (*ptr): 100
value in ptr_n: 0x7fff686efbd0, ptr_n deref: (*ptr_n): 0x7fff686efbd8
value in ptr_o: 0x7fff686efbc8, ptr_o deref: (*ptr_o): 0x7fff686efbd0

```
int main (int argc, char **argv)
{
	int n=3;
	int* ptr=&n;

	int* ptr1=&ptr; //should be int** ptr1-you get a warning like below if not
}
```

```
(base) Computers-Air:C computer$ gcc practice.c
practice.c:13:8: warning: incompatible pointer types initializing 'int *' with an
      expression of type 'int **'; remove & [-Wincompatible-pointer-types]
<U+000B>        int* ptr1=&ptr;
                     ^    ~~~~
1 warning generated.
(base) Computers-Air:C computer$ ./a.out
0x7fff5b63e9d0, 0x7fff5b63e9d0
```

The warning above is showing us that the C language was built to distinguish between pointers at different ”layers” (* pointers, ** pointers, *** pointers etc)…yet notice I could still run the program and it works.

Think of it like the idea of managers at a national company: you can have a branch manager, a regional manager at state manager. They are all managers, but we to be more clear we can clarify and specify what type of manager it is.

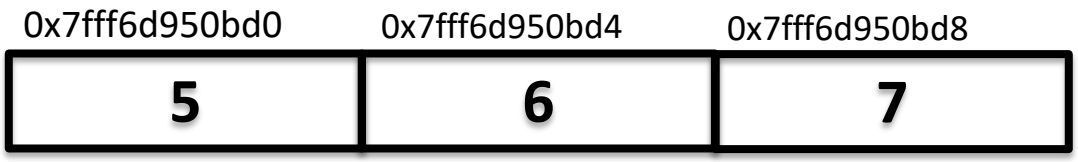
Pointer Arithmetic

- When you declare an array, you are setting aside contiguous space in memory
  - When we say the size of the array, we are telling the computer set aside a certain amount of space

```
int nums[] = {5, 6, 7};
```

We are saying to set aside 4*3 bytes (remember that an int is 4 bytes and there are three ints, so a total of 12 bytes)

Note that sizes of data types can vary by machine. We can use the sizeof operator to find out the size of an int on the machine running

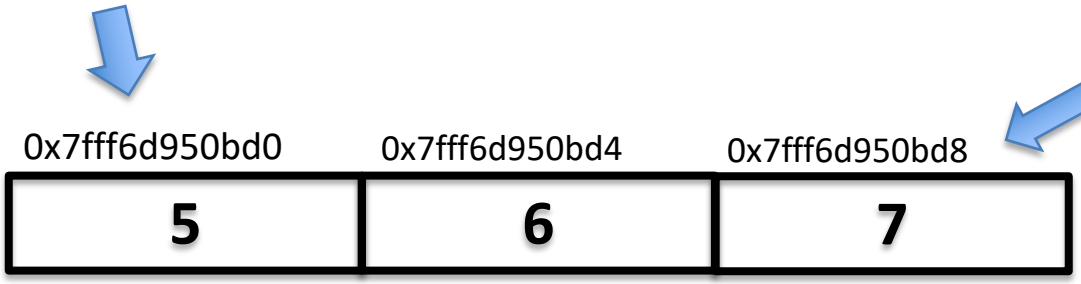


Pointer Arithmetic

- When you declare an array, you are setting aside contiguous space in memory
  - When we say this size of the array, we are telling the computer set aside a certain amount of bytes

```
int nums[] = {5, 6, 7};
```

Each of these is 4 bytes (remember, sizes are machine dependent-on mine it is 4 bytes)



Notice that the address increments by 4 (since the size of an int is 4 bytes-the next address starts 4 bytes later)

Pointer arithmetic with an int array:

Let's start with this sample program:

```
#include <stdio.h>

int main(void)
{
        int nums[] = {5, 6, 7};
         int i;
         int *nums_ptr = nums;

        for(i = 0; i < 3; i++)
        {
                printf("%p, %d\n", nums_ptr, *nums_ptr);
                nums_ptr++;
    }
}
```

Output:

0x7fff6d950bd0, 5

0x7fff6d950bd4, 6

0x7fff6d950bd8, 7

Pointer arithmetic with an int array:

```
#include <stdio.h>

int main(void)
{
        int nums[] = {5, 6, 7};
         int i;
         int *nums_ptr = nums;

        for(i = 0; i < 3; i++)
        {
                printf("%p, %d\n", nums_ptr, *nums_ptr);
                nums_ptr++;
    }
}
```

Here is the array nums:

Address of nums (first element): 0x7fff6b40ebd0 (so the address of 5 is ^^^)

This pointer is pointing at the array. It looks like this:

nums_ptr

This is nums_ptr. The value is the address of nums (the address of the first element of nums)

Pointer arithmetic with an int array:

```
#include <stdio.h>

int main(void)
{
        int nums[] = {5, 6, 7};
         int i;
         int *nums_ptr = nums;

        for(i = 0; i < 3; i++)
        {
                printf("%p, %d\n", nums_ptr, *nums_ptr);
                nums_ptr++;
    }
}
```

I am incrementing the pointer to the array. Everytime I increment, I am moving ahead 4 bytes (since an int is 4 bytes). This is called pointer arithmetic.

Output:
0x7fff6d950bd0, 5
0x7fff6d950bd4, 6
0x7fff6d950bd8, 7

Notice the addresses are sequential. Since an int is 4 bytes, the pointer simply increments by 4 bytes.

Pointer arithmetic with a char array:

```
#include <stdio.h>

int main(void)
{
   char c[] = {'a', 'b', 'c'};
   int i;
   int size=sizeof(c)/sizeof(c[0]);
   char *pc = c;

   for(i = 0; i < size; i++)
   {
            printf("%p, %c\n", pc, *pc);
            pc++;
   }
}
```

Note here that sizeof(c[0]) is 1 since it is a char and chars are 1 byte, so you don't really need it in this case but I'm just showing you how it works.

Output:
0x7fff60772bd9, a
0x7fff60772bda, b
0x7fff60772bdb, c

Notice the addresses are sequential. Since a char is 1 byte, the pointer simply increments by 1 bytes. (a is 10 in hex and b is 11)

I am incrementing the pointer to the array. Everytime I increment, I am moving ahead 1 byte (since a char is 1 byte).

SAMPLE PROGRAMS

Sample Program

- Pointer to pointer examples