CS 262 Lecture 6: Functions

GEORGE MASON UNIVERSITY

Overview of Lecture 6

Functions

- Function vs method
- How functions work in C
- Call by value

Pointers !!

- Building on what we saw with arrays last class
- A little more info on what exactly pointers are
- Lots of live demos

Functions again

- Call by reference

Function Basics

Functions are blocks of code that perform a specific task that can be executed when called

What Do Functions Do?

- Encapsulate logic
- Allow reuse of code
- Makes code more readable

How are Functions Used?

- A function is **called** with 0 or more **arguments**
- A function will **return** at most one piece of data, but can return none (void)

Functions Vs. Methods

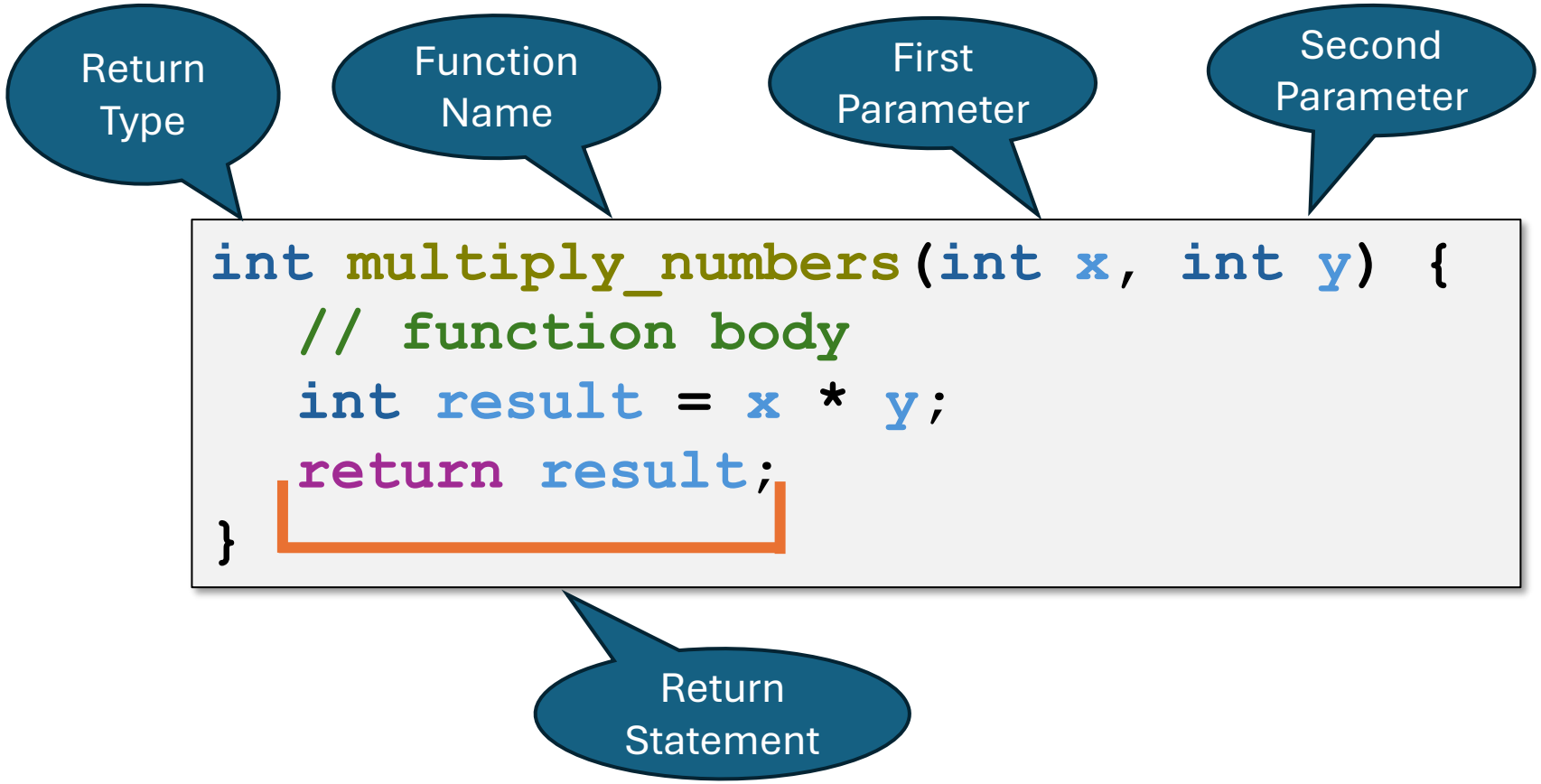
C does not have methods because doesn't have classes

The structure and purpose of methods and functions are very similar

The difference mainly comes down whether it belongs to an object or class (method), or exists independently within a program (function)

Attribute	Methods (OOP)	Functions
Belongs To	A class or object	Independent – belongs to a program
Call Syntax	object.method	function_name(arguments)
Encapsulation	Tied to the state of an object	Standalone – operates on given arguments

Anatomy Of A Function



Function Terminology

A Function **Prototype** is a declaration of a function that gives info about its name, return type, and parameters, but without the actual body or implementation of the function

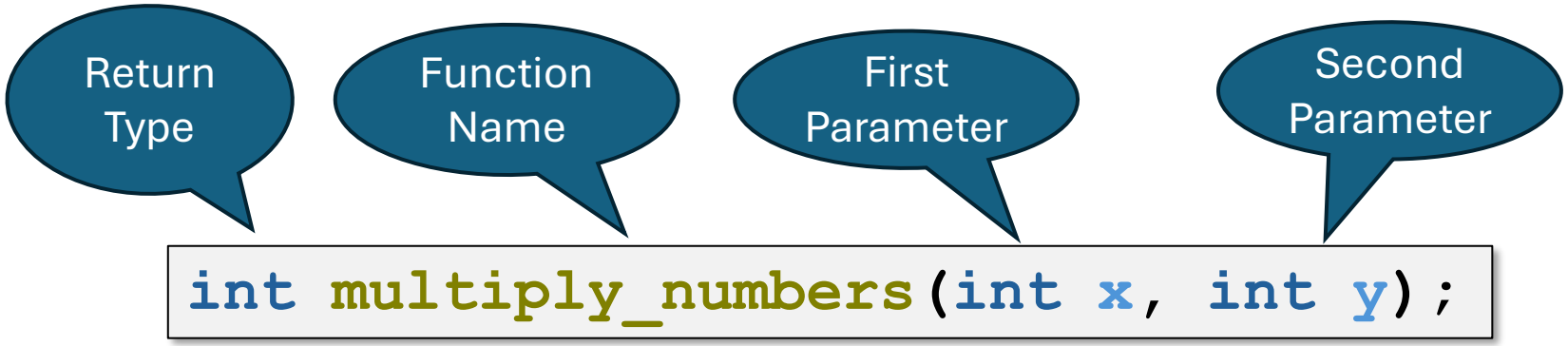
- It informs the compiler about how the function should be called
- Prototypes are placed at the beginning of the program or in a header (.h) file

Parameters are variables declared in the function definition that accept the arguments passed to the function when it is called

Function Rules - Prototype

Function Declarations (Prototypes)

This is just the function header:



It gives the system all the info needed (return type, number of arguments, argument type, function name) needed to know if it is being called correctly

Function Rules - Scope

Scope determines where a variable is accessible

Local scope: A variable declared within a function is only accessible within that function

Global scope: A variable declared outside any function is accessible everywhere in the program

Block scope: A variable declared in a block { } like in a loop is only accessible within that block

Variables declared in higher scope are accessible in lower scope

Function Rules – No Overloading

Function names all must be unique

Unlike Java, we can't do overloading

```
int hi() {
   printf("hello\n");
}

int hi(int var1) {
   printf("%d\n", var1);
}
```

```
int hi() {
   printf("hello\n");
}

int hello(int var1) {
   printf("%d\n", var1);
}
```

Functions - Call By Value

Call By Value

The parameters are received as copies of the actual arguments, **and the function only manipulates the copies of those values**

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```

A **copy** of num is made when the function is called

What do these 2 lines print out?

The function increments the copy

The result is returned

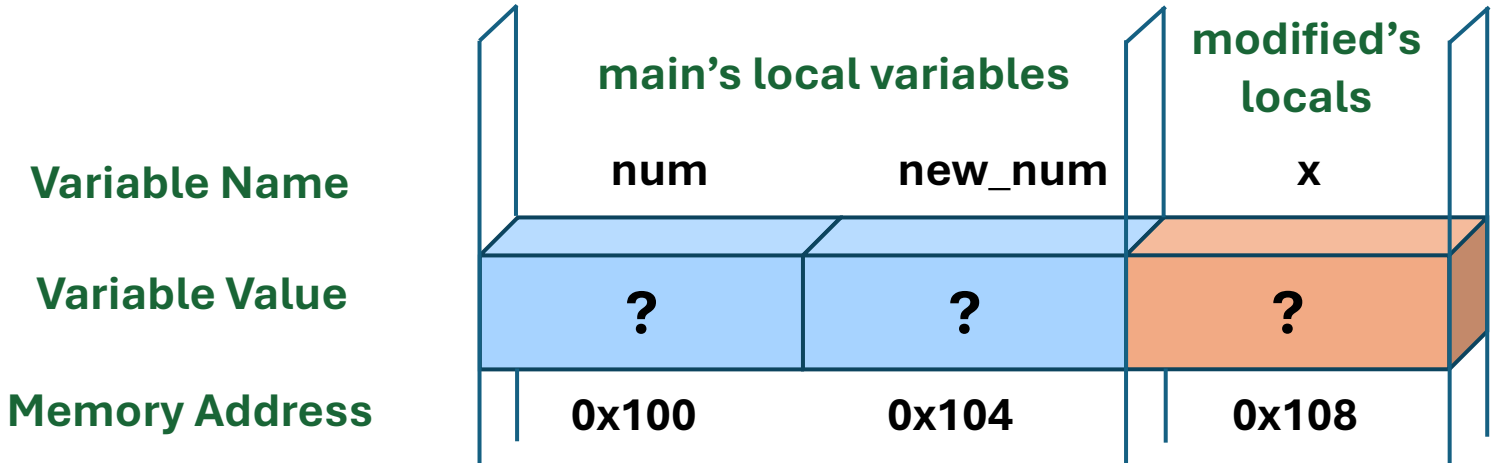
The function receives the copy

Call By Value – What’s In The Memory?

We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```

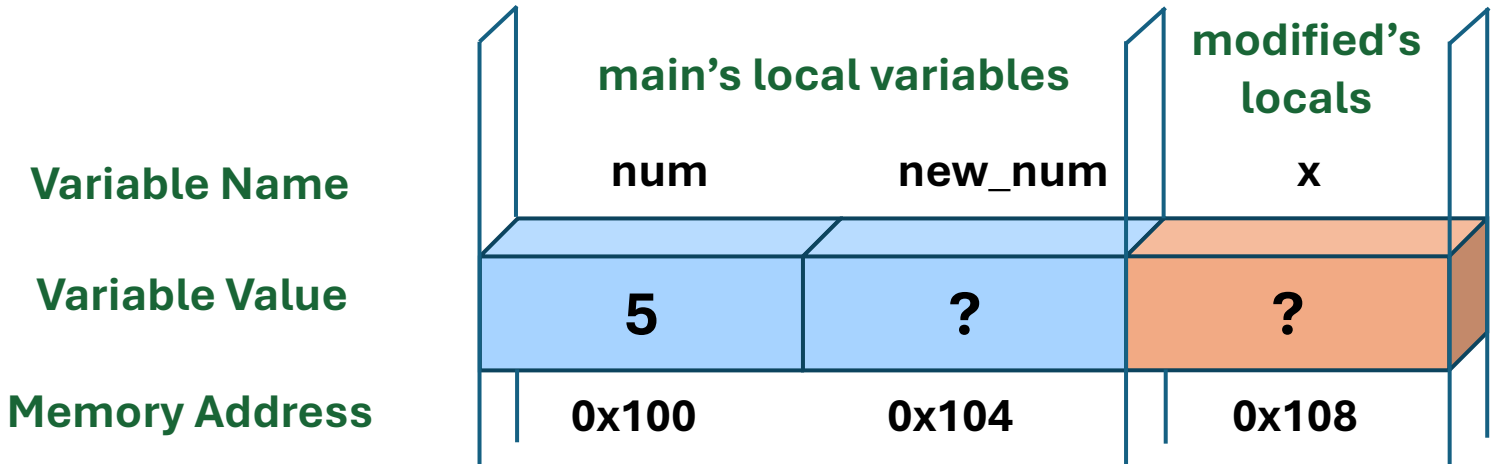


Call By Value – What’s In The Memory?

We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```



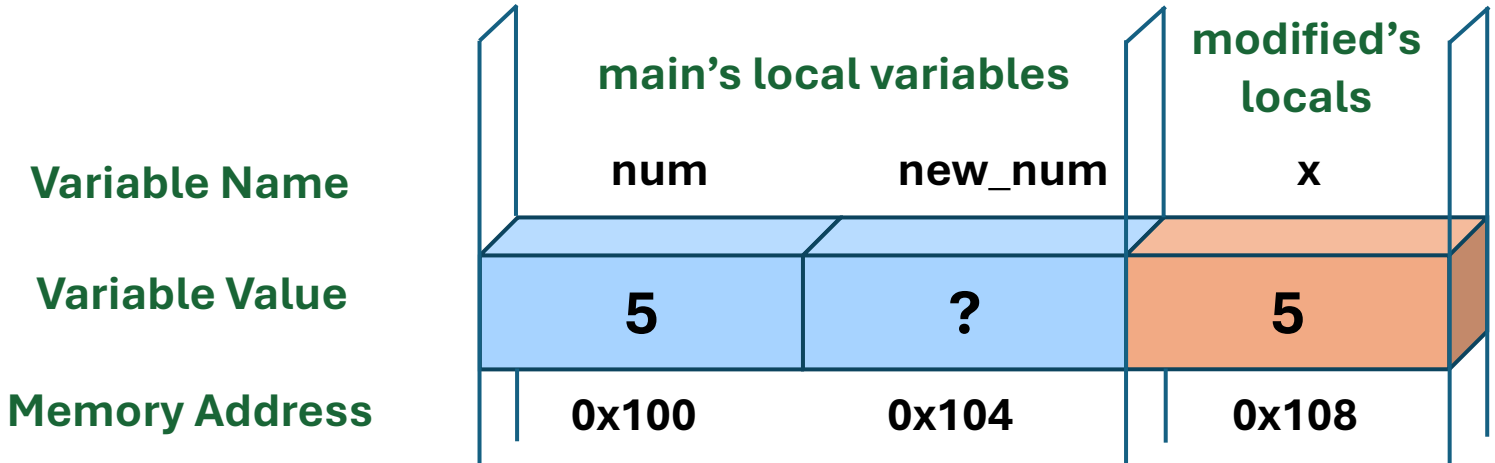
Call By Value – What’s In The Memory?

We Are Here

We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```



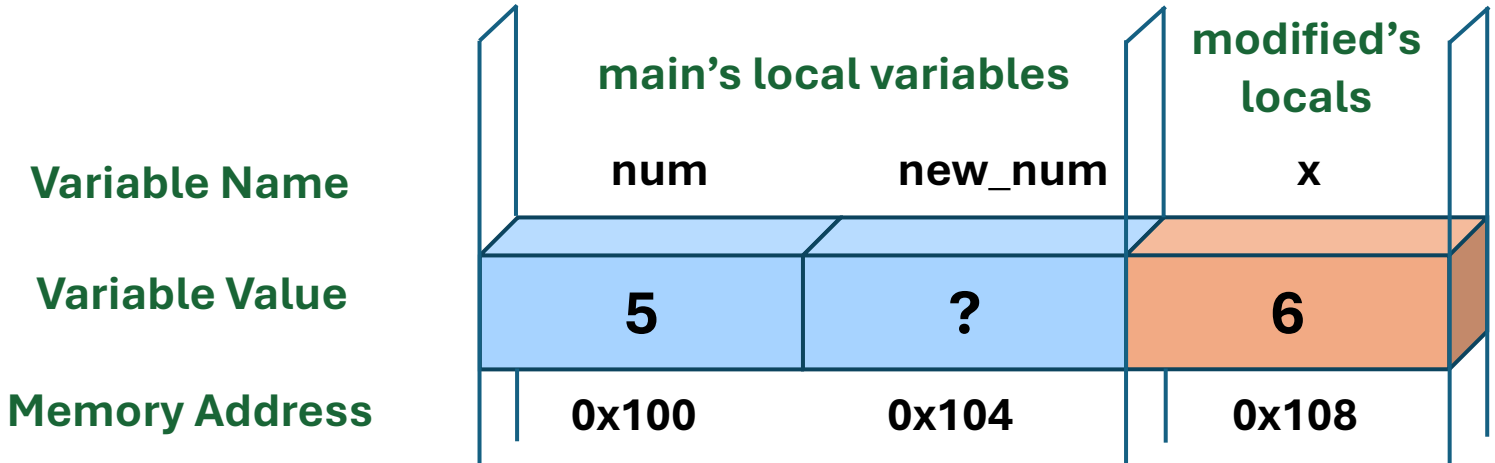
Call By Value – What's In The Memory?

We Are Here

We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```



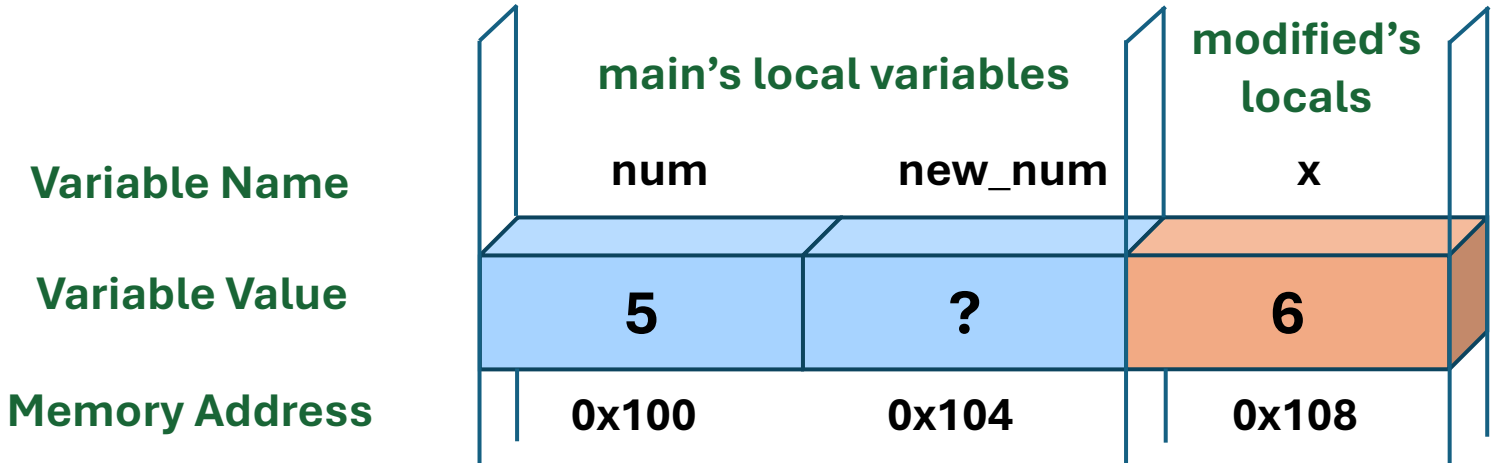
Call By Value – What’s In The Memory?

We Are Here

We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```

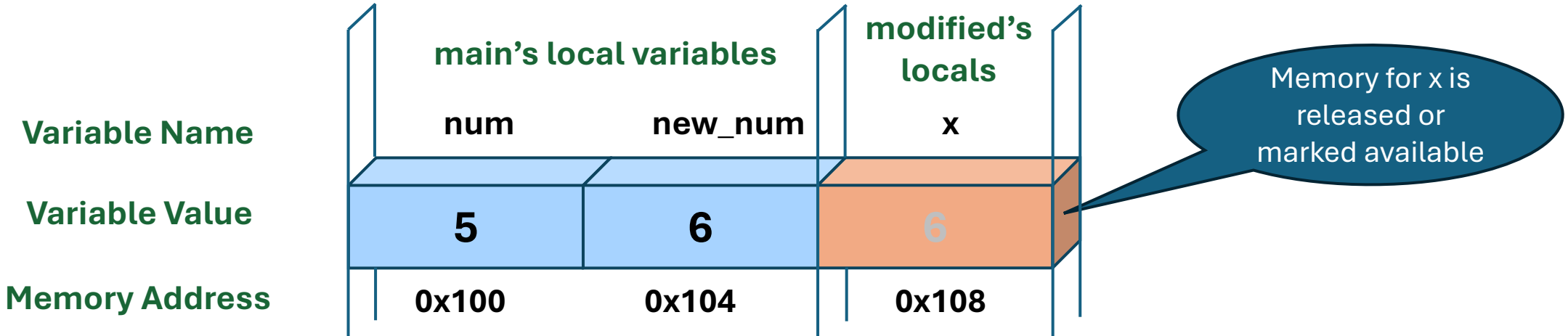


Call By Value – What's In The Memory?

We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```



Call By Value – What's In The Memory?

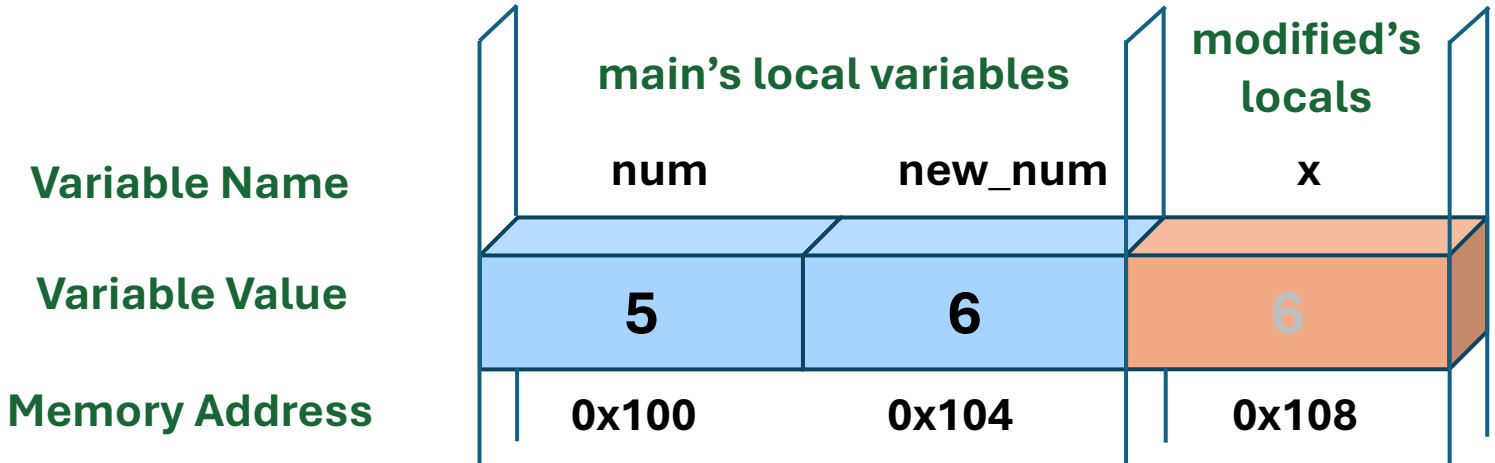
We Are Here

```
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```

Terminal

```
5
```



Call By Value – What’s In The Memory?

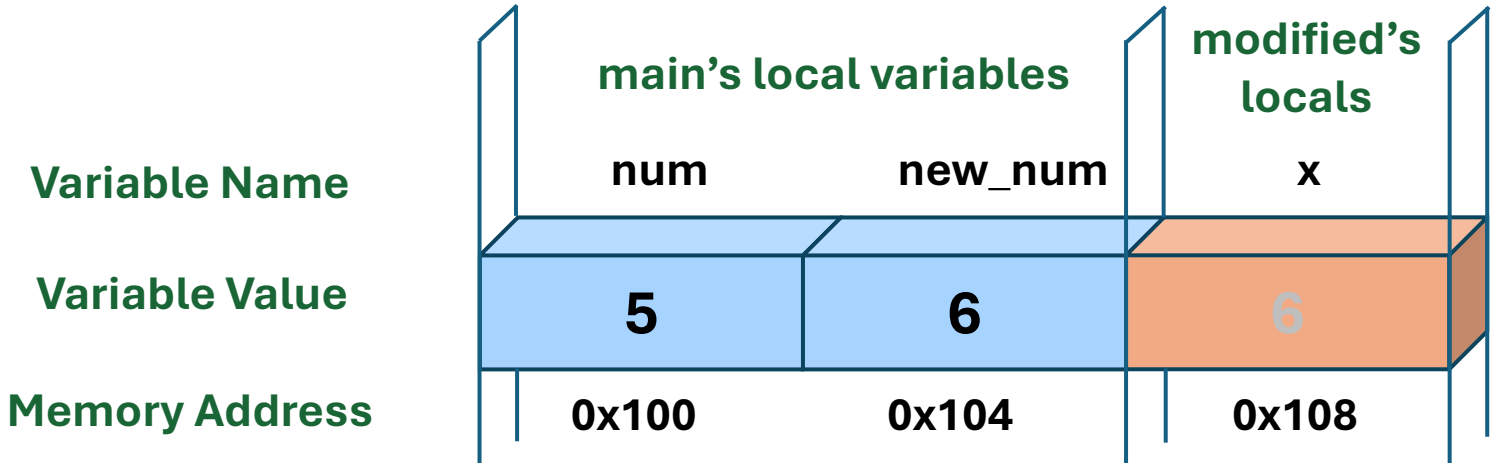
We Are Here

```c
int main() {
    int num = 5;
    int new_num = modify(num);
    printf("num = %d\n", num);
    printf("new_num = %d\n", new_num);
    return 0;
}

int modify(int x) {
    x++;
    return x;
}
```

Terminal

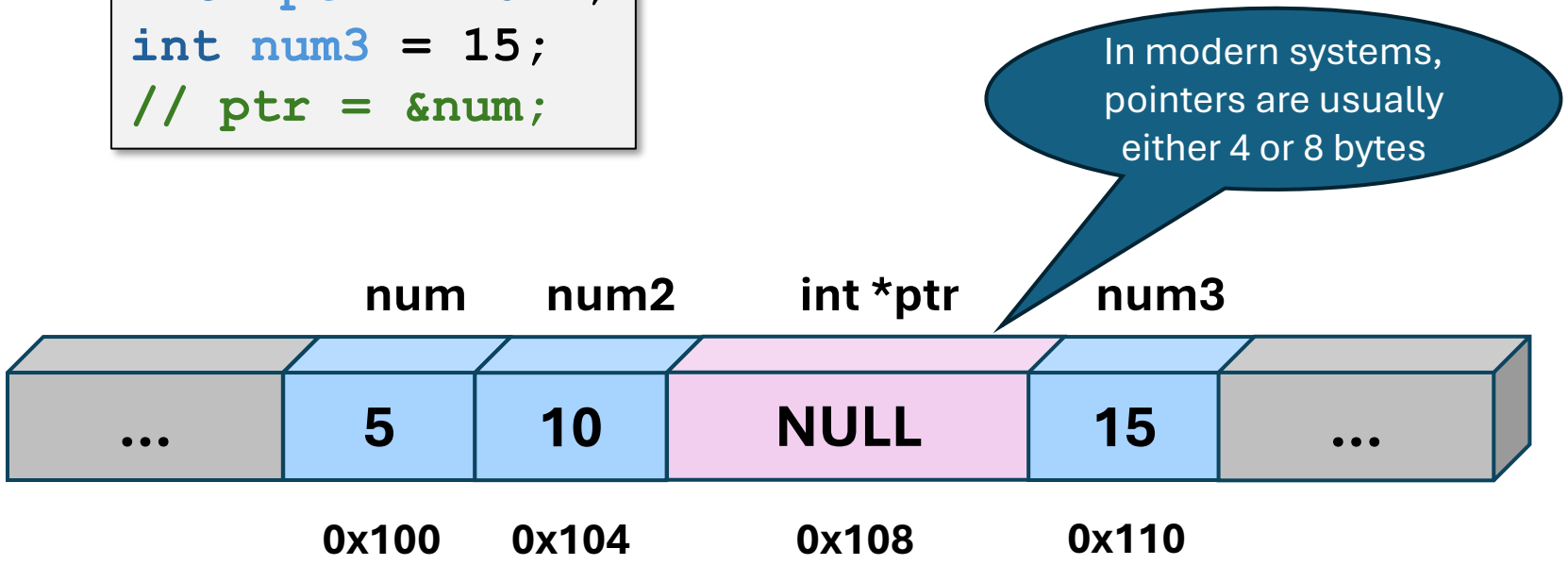
```
6
```



Pointer Basics

A **pointer** is a variable containing the address of another variable

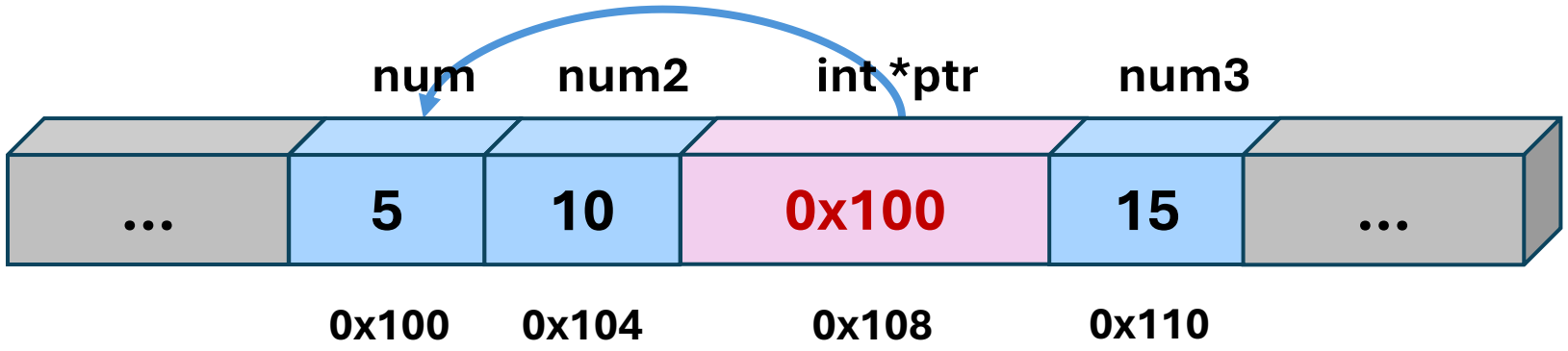
```
int num = 5;
int num2 = 10;
int *ptr = NULL;
int num3 = 15;
// ptr = &num;
```



Pointer Basics

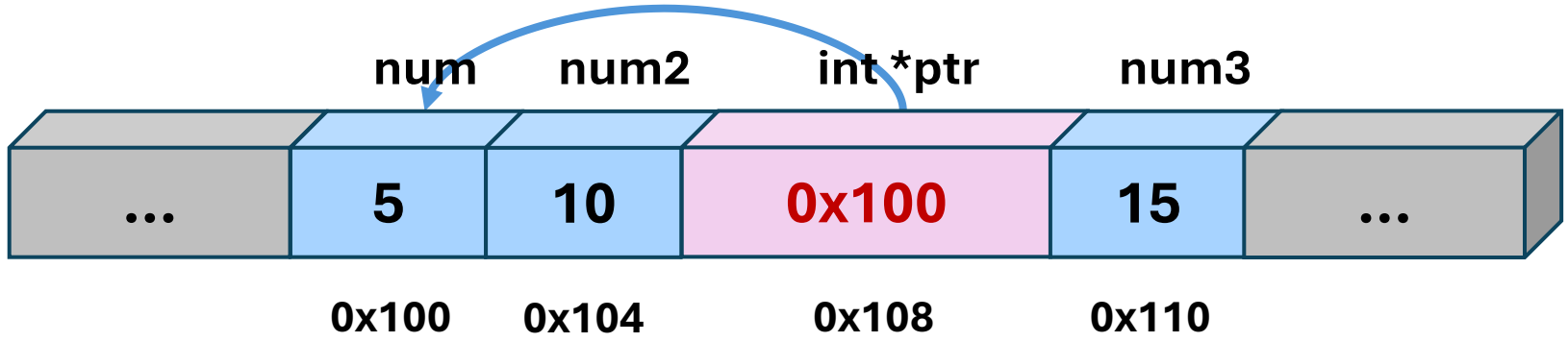
A **pointer** is a variable containing the address of another variable

```
int num = 5;
int num2 = 10;
int *ptr = NULL;
int num3 = 15;
ptr = &num;
```

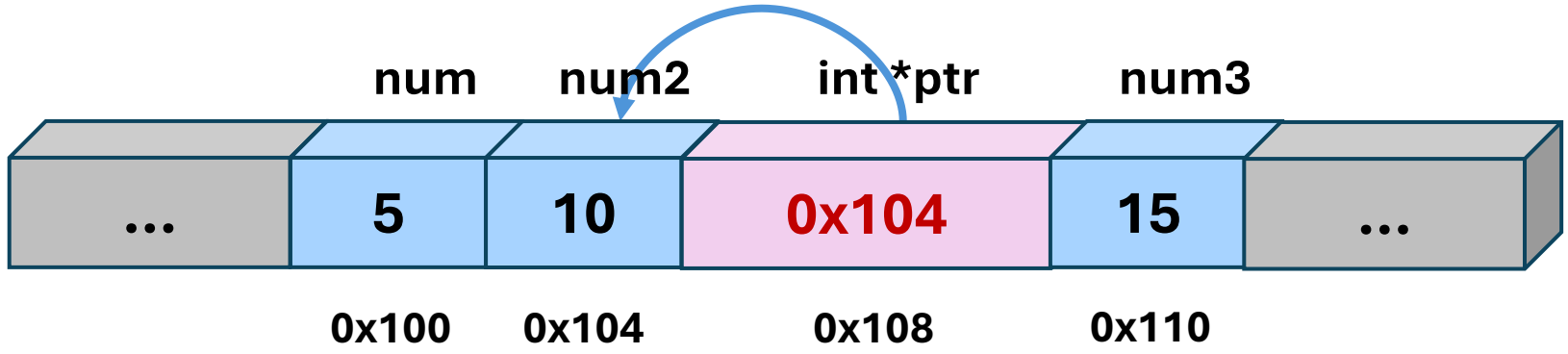


Pointer Basics

```
ptr = &num;
```




```
ptr++;
```



Functions - Call By Reference

Call By Reference

Rather than passing copies of the variables to a function, call by reference passes the **memory address** of a variable

Guess what – we’ve already done this with sscanf

```
sscanf(buffer, "%d", &val);
```

`&val` is the memory address of the variable val

Functions - Call By Reference

Call By Reference

The parameters are received as references to the actual arguments, allowing the function to directly modify the original values

```
int main() {
    int num = 5;
    modify(&num);
    printf("%d\n", num);
    return 0;
}
void modify(int *num) {
    (*num)++;
}
```

`&num` is the memory address of num

`(*num)++` increments the value stored at the address by 1

`*num` means "the value at the address stored in num"

Functions – Working With Arrays

```
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    increment_array(arr, size);
    return 0;
}
```

What's going on here?

```
void increment_array(int arr[], int size) {
    arr[0]++;
}
```

Pseudorandom Number Generation

Random Numbers

Computers aren't capable of generating **truly** random numbers

Instead, they generate what we call 'pseudo-random' numbers

pseudo-random number generators do an excellent job simulating true randomness

returns the next 'random' number between 1 and some value that varies based on the system

```
int rand(void)
```

Pseudorandom Number Generation

Generating Random Numbers

```
int main() {
    for(int i=0; i<10; i++) {
        int r = rand();
        printf("%d\n", r);
    }
}
```

prints 10 random (pseudo-random) integers

What if I want random numbers within a range?

Pseudorandom Number Generation

We can use modulus to generate random numbers within a specified range

Prints 10 ‘random’ numbers between 0 and 99

```
int main() {
    for(int i=0; i<10; i++) {
        int r = (rand() % 100);
        printf("%d\n", r);
    }
}
```

Prints 10 ‘random’ numbers between 1 and 100

```
int main() {
    for(int i=0; i<10; i++) {
        int r = (rand() % 100) + 1;
        printf("%d\n", r);
    }
}
```

Pseudorandom Number Generation

The Seed State

This is the number that initializes the PRNG algorithm

This number is only specified once at the start, using `srand()`

```
int main() {
    int seed = 25;
    srand(seed);
    for(int i=0; i<10; i++) {
        int r = (rand() % 100);
        printf("%d\n", r);
    }
}
```

What do you think happens if I put `srand(seed)` in the loop?

Pseudorandom Number Generation

The Seed State

In practice, the seed state is often chosen to be the current time

```
int main() {
    srand(time(NULL)); // seed with current time
    printf("roll = %d\n", roll_dice());

}

int roll_dice() {
    return (rand() % 6) + 1;

}
```