COP 3223H: Introduction to C Programming

Fall 2023

University of Central Florida

Dr. Kevin Moran

Week 6 - Class 3: Pointers - Part I

Administrivia

- *Small Programming Assignment 2* and *Large Programming Assignment 1* are out!!
  - We will go over these in a minute.
- All assignments will be returned this week.
- Exams grades will be released today.

Today's Agenda

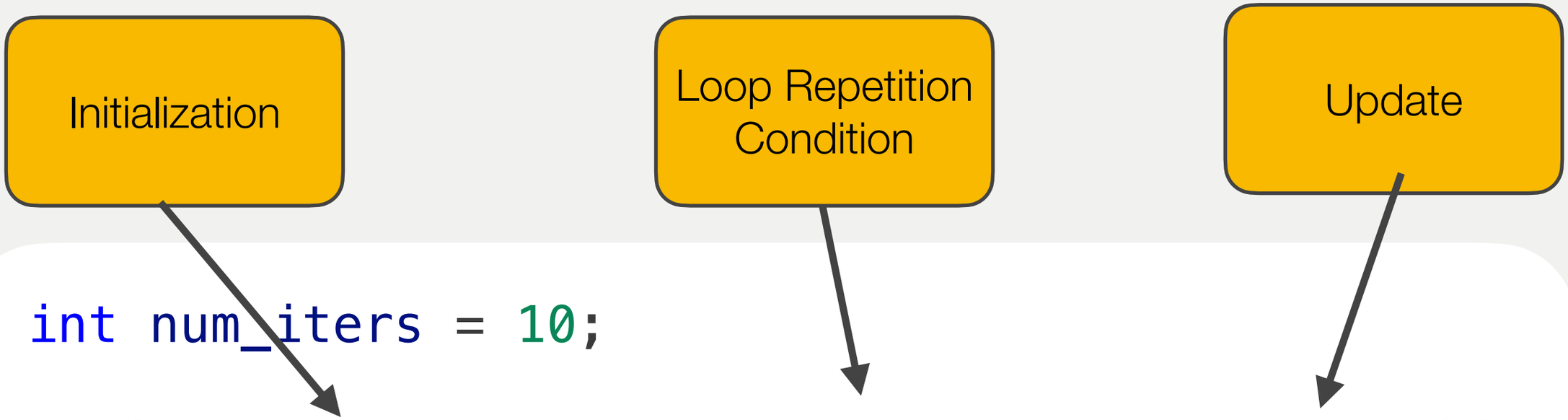
1. Quick Recap of past concepts
2. More on Loops!

Quick Review

The For Statement

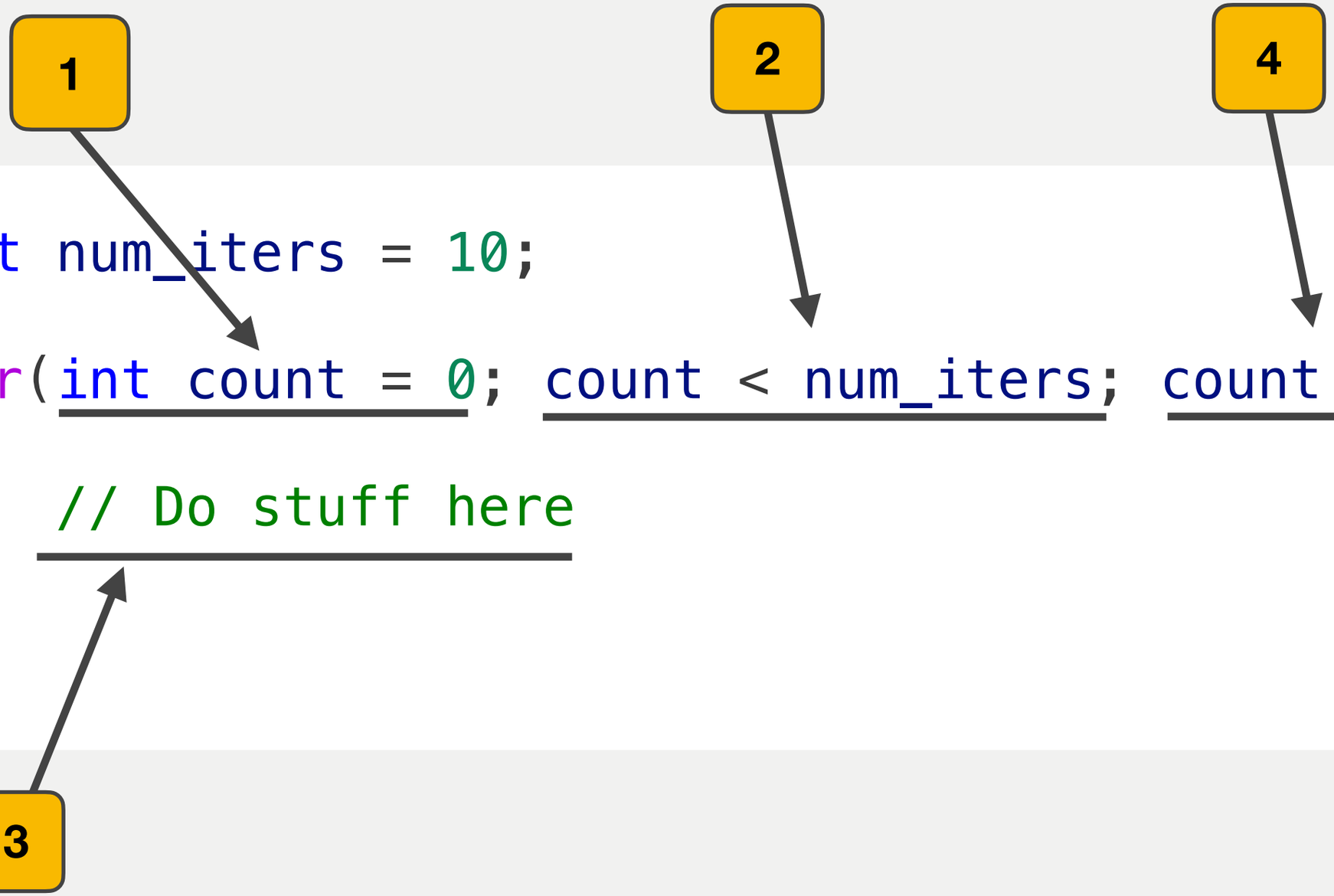
- While loops are very useful when programmers aren't sure how many time a set of instructions should be executed.
- For loops are another type of loops where we know exactly how many times a group set of instructions needs to be executed
- There are three components to the for loop:
  - Initialization of the loop control variable
  - Test of the loop repetition condition
  - Update to the loop control variable

The For Statement




```
int num_iters = 10;

for(int count = 0; count < num_iters; count ++){

    // Do stuff here

}
```

For Loop Control Flow



Nested Loops

- The past examples we have only observed one loop. However, it is possible to have loops within loops (nested loops)

- Nested loops have the following terminology:
  - Outer loop
  - Inner loop

```
for(int x = 0; x < 5; ++x){ // Outer Loop

    for(int y = 0; y < 2; ++y){ // Inner Loop

        printf("x = %d\n", x);
        printf("y = %d\n", y);
    }


}
```

Do-While Loops

- For loops allow programmers to execute instructions a set number times.
- While loops allow programmers to execute instructions multiple times until a condition is met.
- Do-while loops allow programmer to execute instructions multiple times until a condition is met, however the instructions will be executed once at least.

Do-While Loop Example

1

```
char letter_choice;

do{
printf("Enter a latter from A through E: ");
scanf(" %c", &letter_choice);

}while(letter_choice >= 'A' && letter_choice <= 'E');
```

2

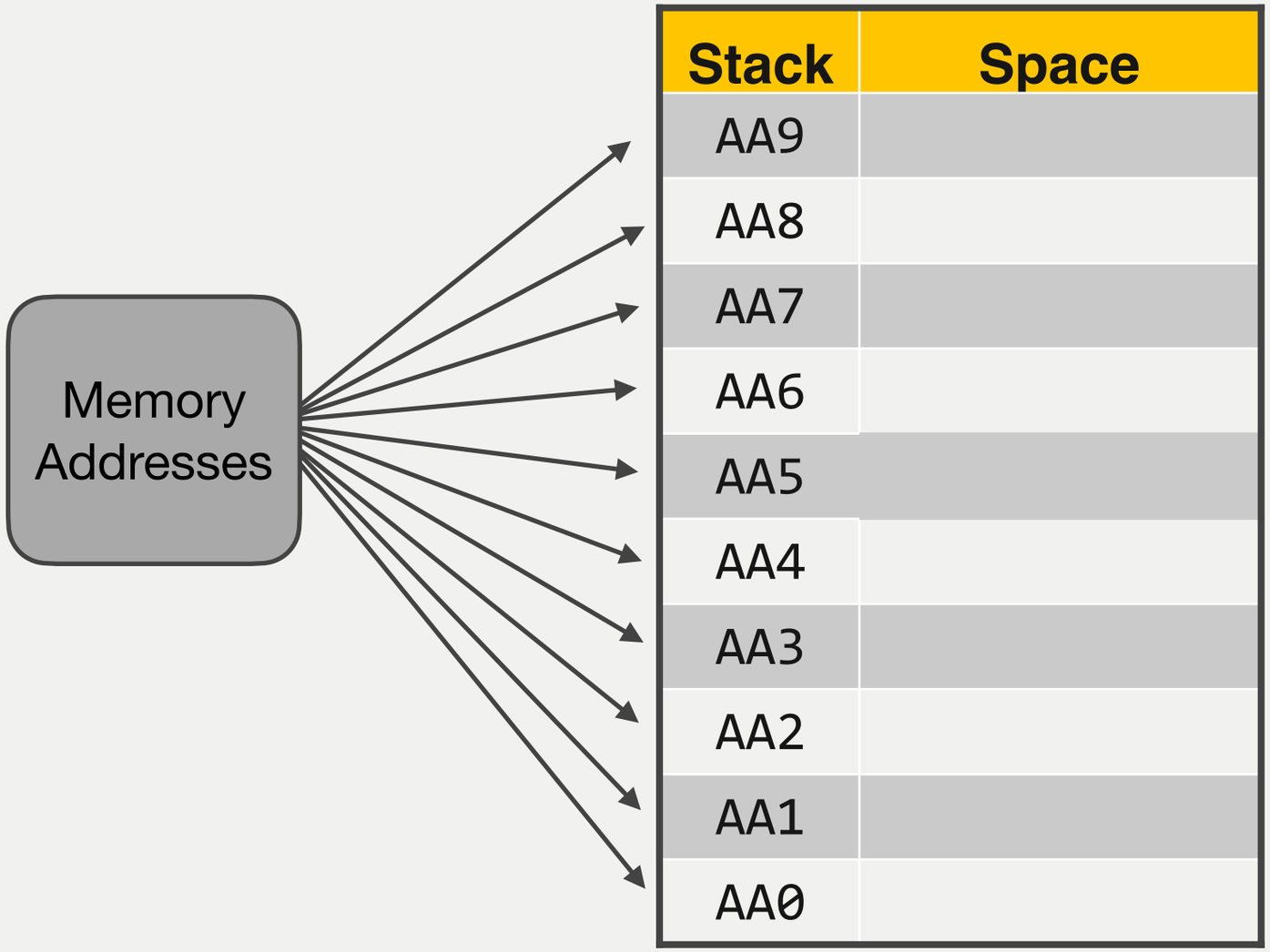
Introduction to Pointers in C

Quick Recap

- So far in this course, we have been doing call-by-value with variables.
- Call-by-value makes a temporary copy for the custom function in RAM and references that cell for all access and computational purposes.

Revisiting the Stack Space

- Remember in the beginning of the course we mentioned that every memory cell has a unique address.



Now we are going to understand how we can store memory addresses in variables to reference them in new places!

What are Pointers?

- Pointers are variables that store the address of a memory cell that contains a certain data type.
- * indicates that variable holds a memory location of certain type
- & is the address

```
int m = 25; // stored in address AA0

    int *itemp = &m;
```

Stack	Space
AA3	
AA2	
AA1	itemp = AA0
AA0	m = 25

Examples of Pointers

```
int *ptr;          // Points to a memory cell holding an int value
double *ptr2;      // Points to a memory cell holding a double value
char *ptr3;        // Points to a memory cell holding a double value
float *ptr4;       // Points to a memory cell holding a float value
```

Why Use Pointers?

- To pass arguments by reference (e.g., easily share information between functions)
- For accessing array elements
- To return multiple values
- Dynamic memory allocation
- To implement data structures
- To do system-level programming where memory addresses are useful

With Great Power…

- If pointers are pointed to some incorrect location then it may end up reading a wrong value.
- Erroneous input always leads to an erroneous output
- Segmentation fault can occur due to uninitialized pointer.
- Pointers are slower than normal variable
- It requires one additional dereferences step
- If we forgot to deallocate a memory then it will lead to a memory leak.

Indirect Referencing

- Indirect reference is accessing the contents of a memory cell through a pointer variable that stores its address.
- This is known as the dereference operator.

Here

```
int m = 25; // stored in address AA0

    int *itemp = &m;

    *itemp = 14;
```

Stack	Space
AA3	
AA2	
AA1	
AA0	m = 25

Indirect Referencing

- Indirect reference is accessing the contents of a memory cell through a pointer variable that stores its address.
- This is known as the dereference operator.

```
int m = 25; // stored in address AA0
int *itemp = &m;
*itemp = 14;
```

Here

Stack	Space
AA3	
AA2	
AA1	itemp = AA0
AA0	m = 25

Indirect Referencing

- Indirect reference is accessing the contents of a memory cell through a pointer variable that stores its address.
- This is known as the dereference operator.

```
int m = 25; // stored in address AA0

    int *itemp = &m;

*itemp = 14;
```

Here →

Stack	Space
AA3	
AA2	
AA1	itemp = AA0
AA0	m = 14

The Dreference Operator *

- We have seen so far in this course that everything is stored somewhere in memory.
- Each memory has its own unique address.
- The pointer variable holds the specific address.
- The dereference operator acts like a “magic key” that allows access to the value stored.
- * is known as deference in C.

The Address Operator &

- We have been using & in our programs ever since scanf was introduced.
- & means address of
- Holds a value in hexadecimal that represents the location in memory.
  - This done with the placeholder **%p**.
  - Hexadecimal is a base 16 number. This means there are 16 unique digits.
- Think about it. Every time we used `scanf("%d", &num)` we were telling the compiler to store the value at the *Memory Address* of the variable named num.

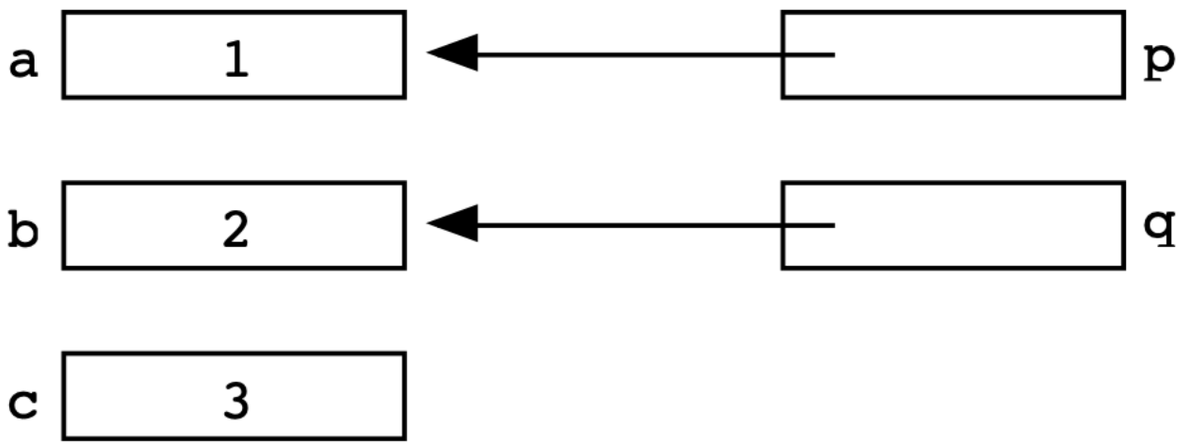
The Pointer Placeholder %p

- There exists a special placeholder that can display the memory address of a reference.

```c
int m = 25; // stored in address AA0

int *itemp = &m;

printf("The address of m is %p\n", &m);
printf("The address of itemp is %p\n", &itemp);
printf("itemp holds the value %p\n", itemp);
```

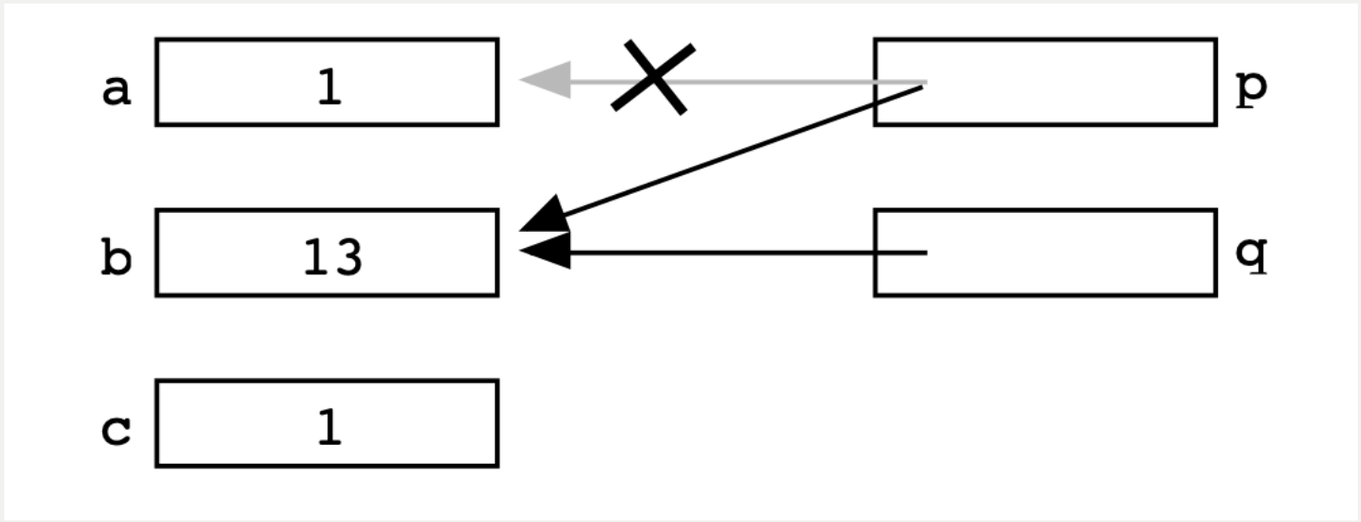
Pointer Example

```
int a = 1;
    int b = 2;
    int c = 3;
    int *p;
    int *q;

    p = &a; // set p to refer to a
    q = &b; // set q to refer to b
```



Pointer Example

```
int a = 1;
 int b = 2;
 int c = 3;
 int *p;
 int *q;

 p = &a; // set p to refer to a
 q = &b; // set q to refer to b

 c = *p; // retrieve p's pointee value (1) and put it in c
p = q;  // change p to share with q (p's pointee is now b)
*p = 13;  // dereference p to set its pointee (b) to 13 (*q is now 13)
```



Acknowledgements

Slides adapted from Dr. Andrew Steinberg's COP 3223H course