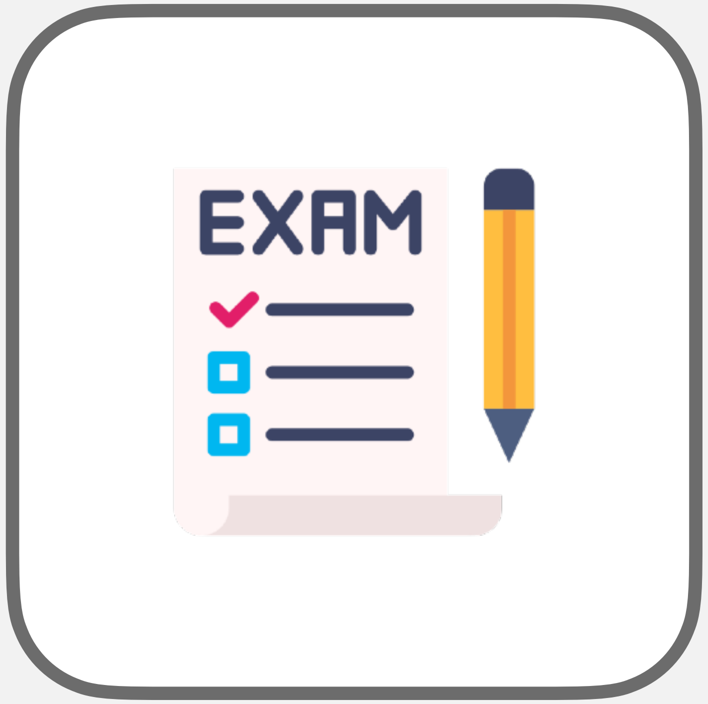
COP 3223H: Introduction to C Programming

Fall 2023

University of Central Florida

Dr. Kevin Moran

Week 10 - Class 1: Exam 2 Review



Administrivia

- *Small Programming Assignment 3* due Friday October 27th.
- *Quiz 1* is due Today at 11:59 pm
- *Exam 1* is Wednesday. October 25th!
    - We will review the format and content extensively today.

Today's Agenda

1. One more array topic
2. Exam Review

Exam 2 Format

- *2 Parts, In-class exam, closed book, 100 points total*
  - *Part 1:* Short Answer Questions
    - 4-5 questions
    - Either provide program output or answer with a code snippet or a few short sentences.
  - *Part 2:* Programming Questions
    - 4-5 questions with multiple parts
    - Either provide the output of a more complex program, or write several lines of code
  - Focused on material from Weeks 6-9, but this builds on concepts from Weeks 1-5.
  - You will have the ***entire*** class period to complete the exam
  - Please bring your UCF ID to the exam

Reading Input into Arrays

Reading Input into Arrays (Wrong!)

```
int num[2];
int num2[2];
int mynum[2];

printf("Enter: ");
scanf("%d", num);
printf("Enter: ");
scanf("%d", num2);
printf("Enter: ");
scanf("%d", mynum);

for(int i = 0; i < 2; i++){

printf("num[%d] = %d\n", i, num[i]);
printf("num2[%d] = %d\n", i, num2[i]);
printf("mynum[%d] = %d\n", i, mynum[i]);

}
```

This will result in garbage being saved to the array after each first slot.

Reading Input into Arrays

```c
#include<stdio.h>

void readInArray(int arr[], int size);

int main(void){

int arr[2];

readInArray(arr, 2);

}

void readInArray(int arr[], int size) {
    int i;
    printf("Enter your list of numbers: ");
    for (i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
        printf("%d\n", arr[i]);
    }
}
```

To read in values properly, create a for loop, and iterate through each element in the array.

Reading Strings into Arrays

```
char word[8];
printf("Enter: ");
scanf("%s", word);
printf("word = %s\n", word);
```

Note the behavior here is slightly different…
We can read multiple characters into the array at one.

This is special for strings.

However, there are still issues with this code.

For example, what if more than 8 chars are entered?

What if multiple words are entered?

Midterm Exam Review

Week 6 - Class I: Loops Part I

Different Kinds of Loops

Comparison of Different Loop Types

Type	When to Use	C Implementation
Counting Loop	When you know the number of iterations the loop will need.	while, for
Sentinel Controlled Loop	Input a list of data of any length ended by a special value.	while, for
Endfile-controlled Loop	Input any list of data of any length from a data file.	while, for
Input Validation Loop	Repeated interactive input of a data value until this value is within the desired range	do-while
General Conditional Loop	Repeated processing of data until a desired condition is met	while, for

Understanding the While Loop Flow

First condition is evaluated

Code inside the control structure is evaluated if the condition was *true*

```
while(condition)
{
     // instructions go here
}
```

Continue Statement

- There is a special keyword in C called continue that can cause an iteration to be skipped.
- *What will the code fragment display?*
- Why does this even exist?
    - In larger programs, there might be special iterations where a certain set values may be invalid to use.

```c
int num = 10;

while(num > 0){
    if(num ==5){
        num -= 1;
        continue;
    }

    printf("Continue: num=%d\n.", num);

    num -= 1;

}
```

Compound Assignment Operators

- You may have noticed instructions where variable have assignment statement that involves itself.
  - `var1 = var1 + 1;`
  - `var2 = var2 – 2;`
- C, this can be rewritten as a compound statement.
  - `+: +=` e.g., `var1 += 1;`
  - `-: -=` e.g., `var2 -= 2;`
  - `*: *=`
  - `/: /=`
  - `%: %=`

Examples of Compound Assignment Operators

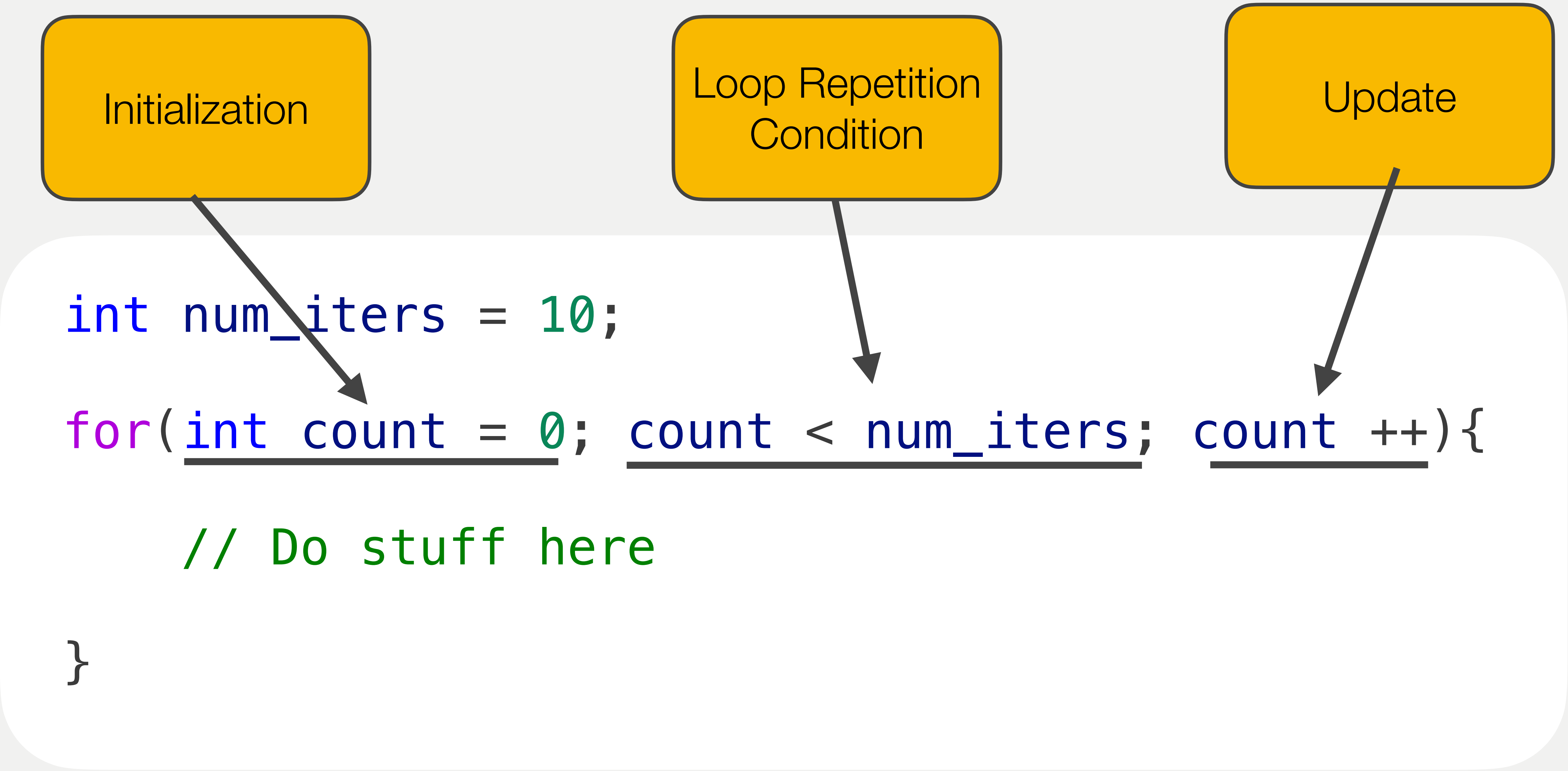
Compound Assignment Operators	
count_emp = count_emp + 1;	count_emp += 1;
time = time - 1;	time -= 1;
total_time = total_time + times;	total_time += times;
product = product * item;	product *= item;
n = n * (x + 1);	n *= (x + 1);

Operator Precedence

Operator	Precedence
`function calls`	Highest
`! + - & (unary)`	
`* / %`	
`+ -`	
`< <= >= >`	
`!= ==`	
`&&`	
`||`	
`(=, +=, -=, *= …)`	Lowest

Week 6 - Class II: Loops Part 2

The For Statement



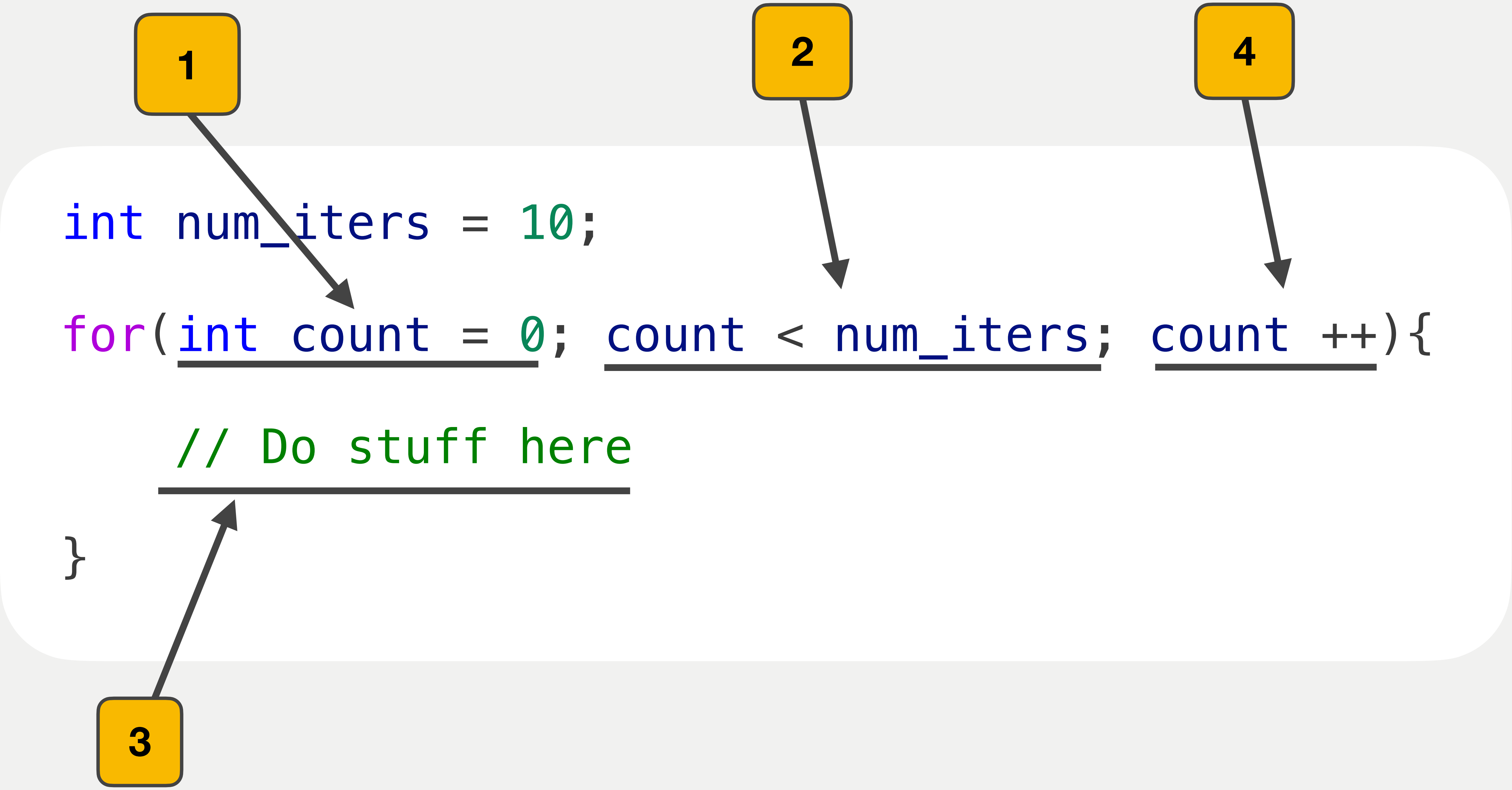
Initialization

Loop Repetition Condition

Update

```
int num_iters = 10;

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

    // Do stuff here

}
```

For Loop Control Flow



```
int num_iters = 10;

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

    // Do stuff here

}
```

Increment and Decrement Operators

- C provides an alternative when writing an increment and decrement by 1 statement.
- `counter = counter + 1`; can be rewritten as `counter++`;
- `counter = counter - 1`; can be rewritten as `counter--`;
- Pre increment/ Pre decrement (`--counter`;)
- Post increment/Post decrement (`counter++`)

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 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

Week 6 - Class III: Pointers Part I

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.

Here

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

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.

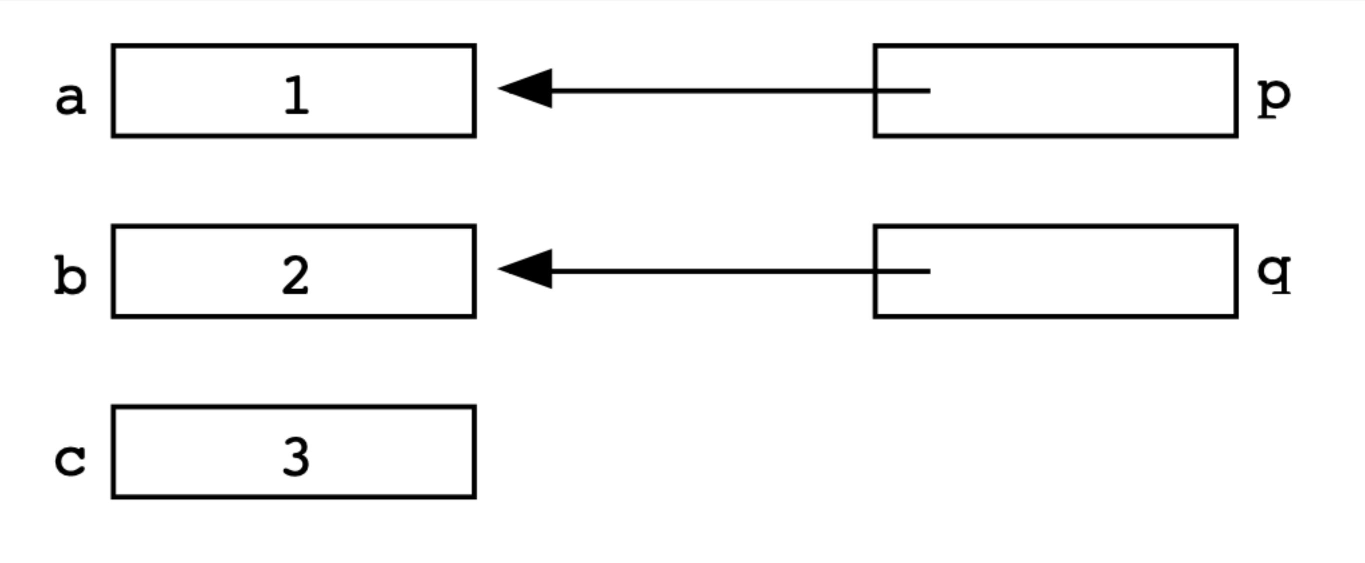
The Pointer Placeholder %p

- 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);
```

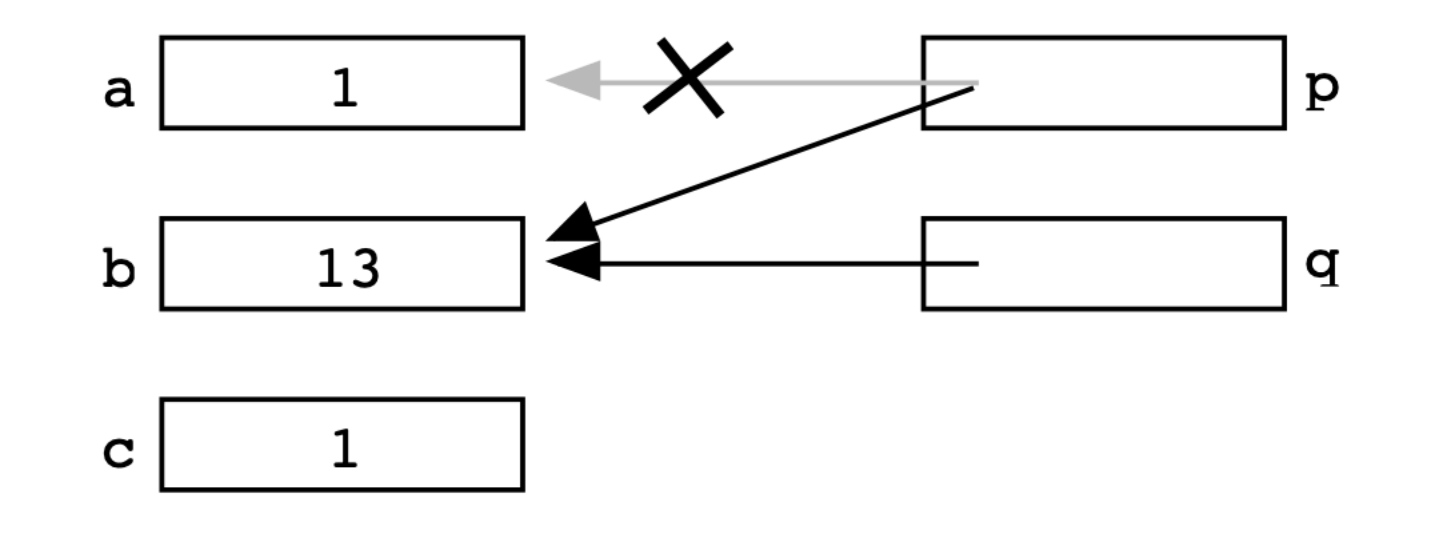
Pointer Example

```
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)
```



Week 7 - Class I: Pointers Part II

The NULL/NIL Value

- Pointers that we have seen hold an address.
- Can pointers hold a value that doesn't represent an address in memory?
  - The simple answer is YES!
- NULL (or NIL) is a special value that represents nothing.
- We will see more of the value NULL being utilized when discussing dynamic memory.

```
int *ptr = NULL;
```

Stack	Space
AA3	
AA2	
AA1	
AA0	ptr = NULL

Functions with Parameters

- In past sessions, we have seen that variables have been passed by value.
- With pointers, we can now past variables by reference.
- Instead of making a local copy for the function, we can pass the memory location and perform computation on the variable in its original location. This is known as pass-by-reference.

Pass By Value Example

Here

```
#include<stdio.h>

void myFunction (int numl, int num2, int num3);

int main()
{
int num1 = 3;
int num2 = 2;
int num3 = 1;
printf (“num1 = %d\n", num1);
printf ("num2 = %d\n", num2);
printf ("num3 = %d\n", num3);

myFunction (num1, num2, num3);

printf ("num1 = %d\n", num1);
printf (“num2 = %d\n", num2) ;
printf ("num3 = %d\n", num3);
return 0;
}


void myFunction (int num1, int num2, int num3)
{
num1 = 5;
num2 = 8;

printf ("num1 = %d\n", num1);
printf ("num2 = %d\n" , num2);
printf ("num3 = %d\n", num3);
}
```

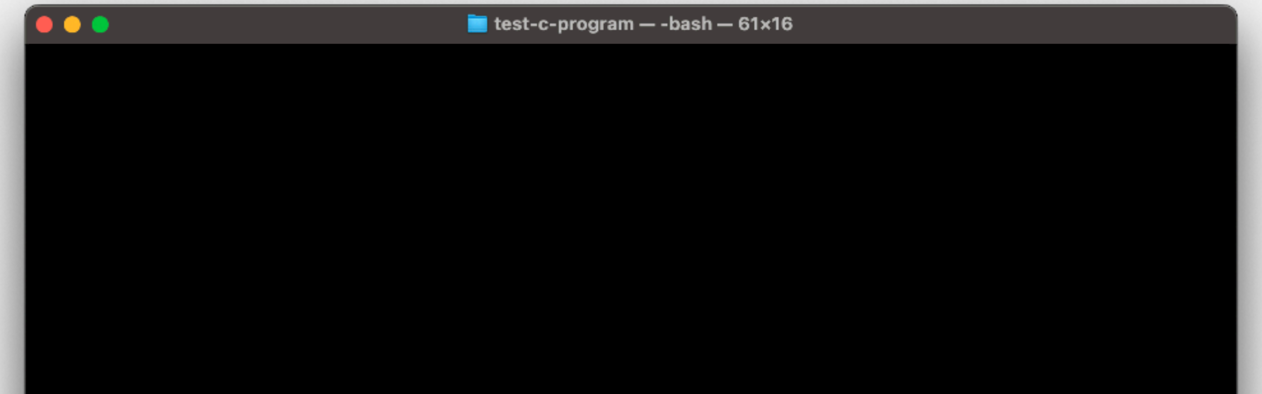
Stack Space	
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	
AA3	
AA2	
AA1	
AA0	

Pass By "Reference" Example

Here

```
#include <stdio.h>

void increaseValue(int *num);

int main(void){

   int num = 13;

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

    increaseValue(&num);

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

    return 0;

}

void incraseValue(int *num){
    *num = *num + 1;
}
```

Stack Space	
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	
AA3	
AA2	
AA1	
AA0	



Scope of Names

- Scope of a name refers to the region in a program where a particular meaning of a name is visible.

- Local and Global Variables

- When variables are being used, certain functions may not be able to access them due to where they were declared!

- Why can’t everything be global? Would that be easier?

```
#include <stdio.h>

void increaseValue(int *num);
void calculate();

int var; // global variable BAD!!

int main(void){

   int num = 13;

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

return 0;

}

void calculate(){

    int num1;   // local variable
    int num2;   // local variable
    scanf("%d%d", &num1, &num2);

    int result = num1 + num2;

}
```

Week 8 - Class I: File I/O

Files

- In C we can access files (such as text files)
- This access allows for reading and writing.
  - Reading – Input
  - Writing – Output
- There is a special kind of variable in C that allows us access for text files.
- *File Pointers!*

```
FILE *inp; // pointer to input file
FILE *outp; // pointer to output file
```

File Pointer Access

- There are two basic types of access we will learn in this class
    - ***Reading*** – this allows the program to collect input from a text file. Think of it like scanf for collecting input from the keyboard
    - ***Writing*** – this allows the program to write output to a text file. Think of it like printf for displaying output to the monitor

Other Types of File I/O Access

- There are other modes for FILE I/O Access besides r and w mode.
  - *a – append mode*
    - Adds content to the next available space in the File
  - *r+ – both reading and writing*
    - Acts as both r and w mode. Assumes that File exists in memory
    - If file does not exist then it doesn’t work
  - *w+ – both reading and writing*
    - Acts as both and w mode. Doesn’t assume that File exist in memory
    - If it does exist already, content will be deleted by setting the length to zero bytes
    - If it doesn’t exist, it will create the File
  - *a+ – both reading and writing*
    - If file doesn’t exist, it will create it
    - When reading, pointer starts at the beginning of the file content
    - Writing to file will only be appended

Syntax for File Reading/Writing

```
// preparing files for input and output
inp = fopen("indata.txt", "r");
outp = fopen("outdata.txt", "w");
```

```
fscanf(inp, "%lf", &item); // reading file
fprintf(outp, "%f", item); // writing file
```

printf, scanf, fprintf, and fscanf

```
FILE *inp; // pointer to input file
FILE *outp; // pointer to output file

// preparing files for input and output
inp = fopen("indata.txt", "r");
outp = fopen("outdata.txt", "w");

scanf("%lf", &item); // reading input from command line
fscanf(inp, "%lf", &item); // reading input from file

printf("%f", item); // printing information to command line
fprintf(outp, "%f", item); // writing file

fclose(inp);
fclose(outp);
```

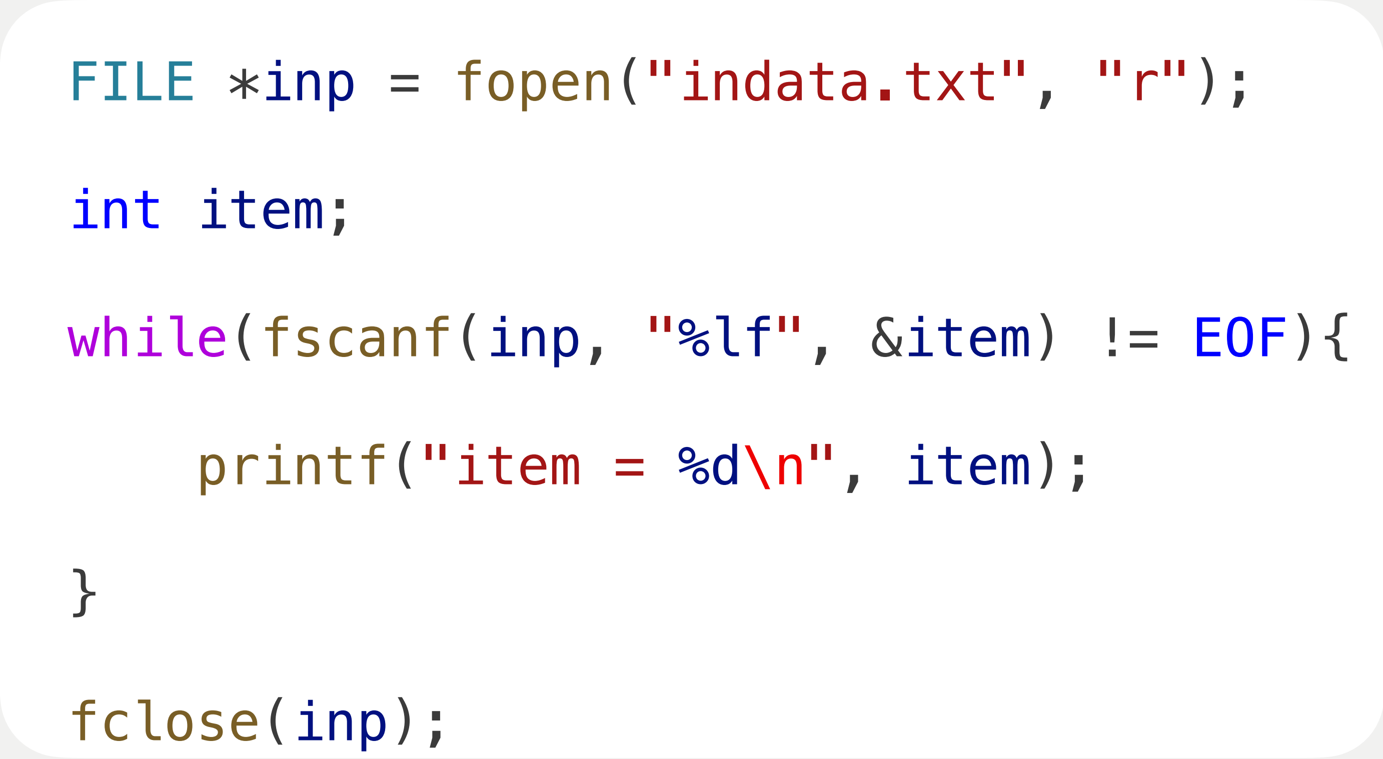
Notice the placeholder and variable address

Notice the placeholder and variable

The only addition is the file pointer!

EOF Macro Constant

- C has a special *predefined* macro constant called EOF in the stdio header file.
- EOF stands for "*E*nd *O*f *F*ile"
  - The value of EOF is –1. 0 is still used if it can read something potential, BUT wasn't processed successfully.
- EOF is widely used to assist with reading an ENTIRE file.



```c
FILE *inp = fopen("indata.txt", "r");

int item;

while(fscanf(inp, "%lf", &item) != EOF){

    printf("item = %d\n", item);

}

fclose(inp);
```

One Last Thing…

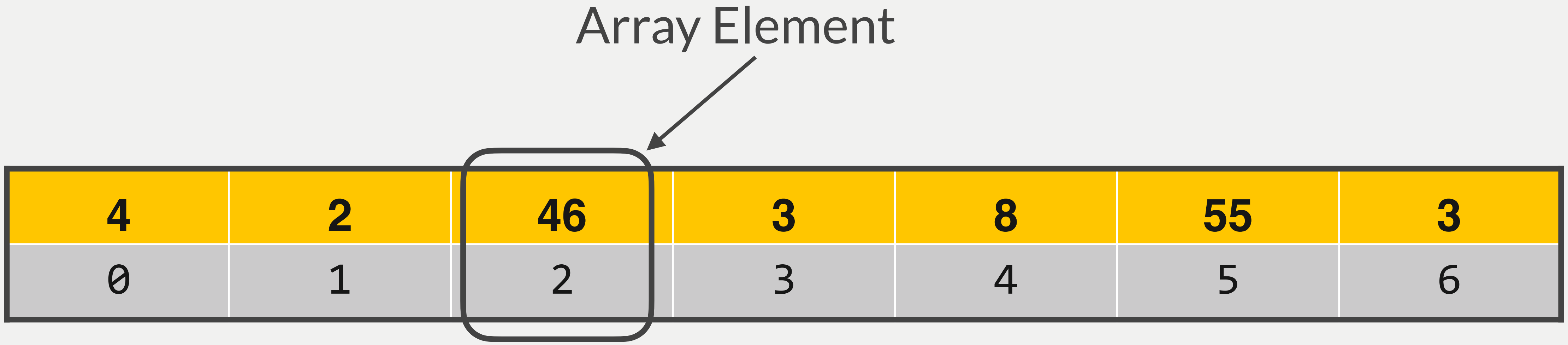
- After you done accessing the file for reading or writing you must CLOSE the file.
- If you forget to close the file, the program will still run BUT leaves files open with access.
- It’s a common mistake beginners make. Remember after opening to close the files.

```
fclose(inp);
fclose(outp);
```

Week 8 - Class II: Arrays Part I

Arrays

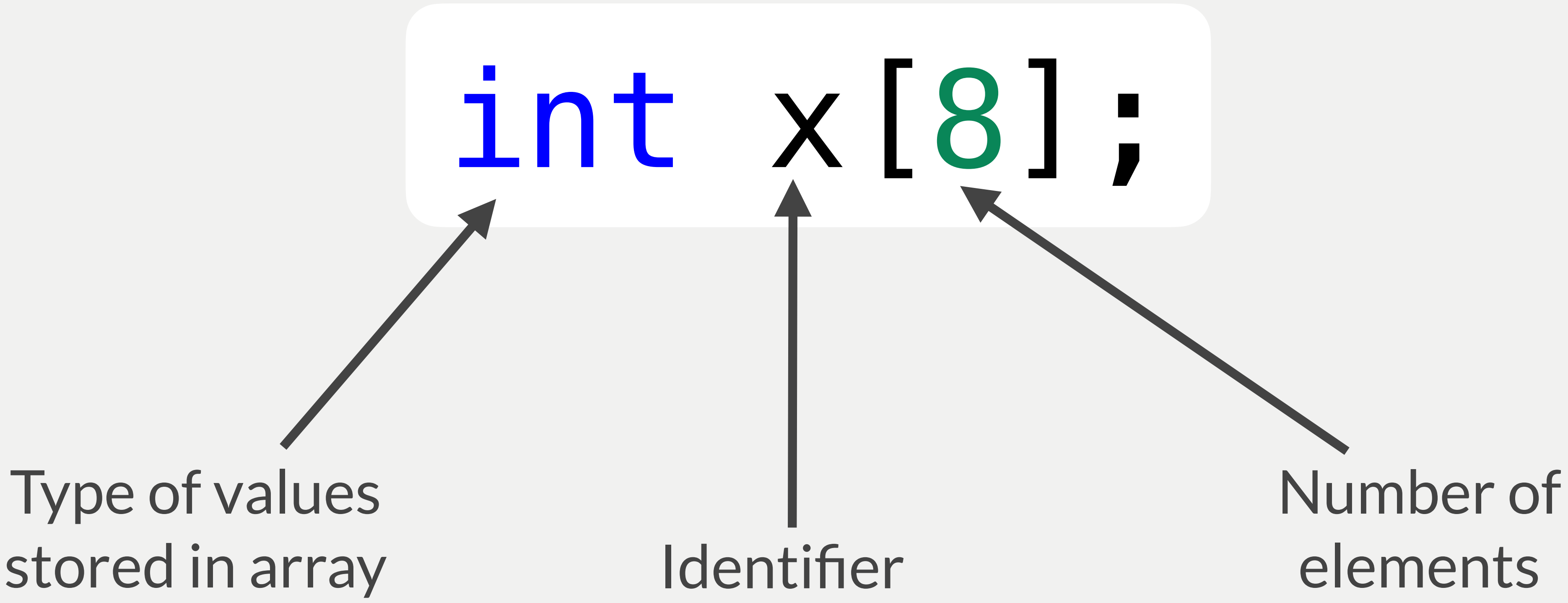
- An Array is a collection of data items of the same type.
- An array element is a data item that is part of an array.
- An array is a collection of two or more adjacent memory cells.



Array Element

4	**2**	**46**	**3**	**8**	**55**	**3**
0	1	2	3	4	5	6

Declaring an Array



```
int x[8];
```

Type of values
stored in array

Identifier

Number of
elements

Arrays and Stack Visualization

```
int x[8];
```

Here we have an array (called x) of 8 elements. That means there are 8 adjacent cells occupied.

Stack	Space
AA9	
AA8	
AA7	🟩
AA6	🟩
AA5	🟩
AA4	🟩
AA3	🟩
AA2	🟩
AA1	🟩
AA0	🟩

Week 8 - Class III: Arrays Part II

Accessing Values

- Now that we have observe the stack space visualization of arrays, we now have to understand how values are accessed.
- Subscripted variable are variables followed by a subscript in brackets, designating an array element.
- Array subscript is a value or expression enclosed in brackets after the array name, specifying which array element to access.

Array x

4	**2**	**46**	**3**	**8**	**55**	**3**
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]

Arrays and Stack Visualization

Here

```
int arr[5];

for(int x = 0; x < 5; x++){

    arr[x] = x * 3;

}
```

Stack	Space
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	arr[4] = ??
AA3	arr[3] = ??
AA2	arr[2] = ??
AA1	arr[1] = ??
AA0	arr[0] = ??

Useful Statements for Array Access

Statement	Explanation
printf(“%d,x[0]);	Displays the stored value at x[0]
x[3] = 1;	Stores the value 1 in x[3]
sum = x[0] + x[1];	Stores the sum of x[0] and x[1]
sum += x[2];	Adds x[2] to sum
x[3] +=13;	Adds 13 to x[3]
x[2] = x[0] + x[1]	Adds the values stored in x[0] and x[1].

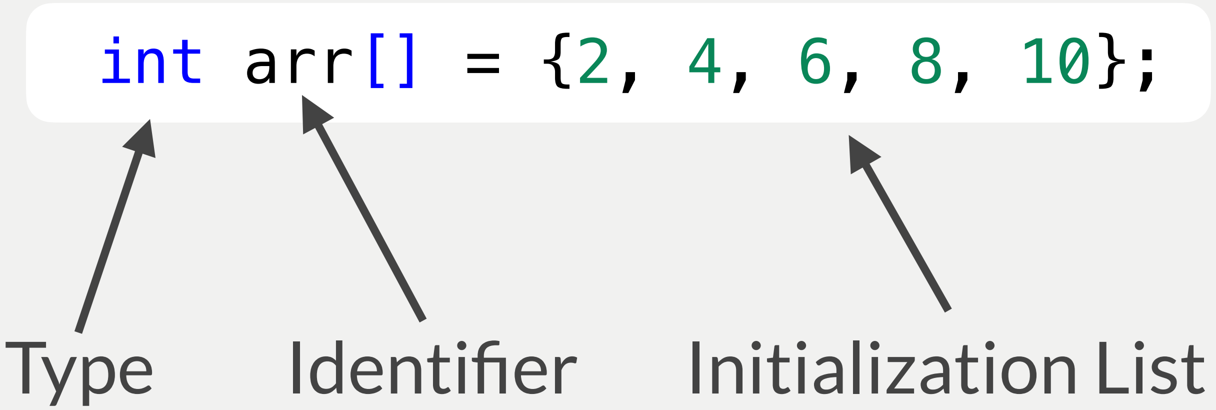
Array Initialization

- Like variables, arrays must be declared and initialize.
- In order to declare an array, programmers must specify the type of data it holds along with the predefined size.
- Programmers can also declare and initialize an array in one line of code (programmers don’t have to include the size if this method is done).
- When an array is declared, what values are automatically stored?

```
int arr[5]; // What is stored inside memory after declaration
```

Array Initialization List

- Like variables, arrays must be declared and initialize.
- In order to declare an array, programmers must specify the type of data it holds along with the predefined size.
- Programmers can also declare and initialize an array in one line of code (programmers don’t have to include the size if this method is done).
- When an array is declared, what values are automatically stored?



```
int arr[] = {2, 4, 6, 8, 10};
```

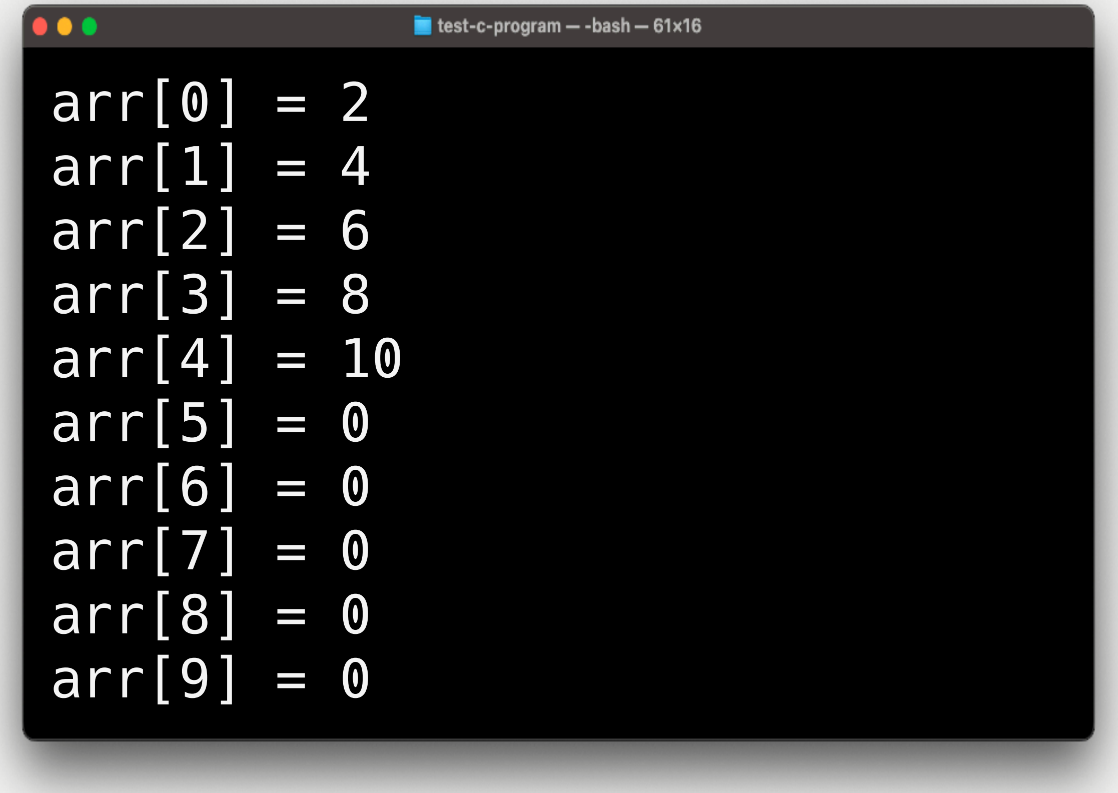
Array Initialization List

```
int arr[] = {2, 4, 6, 8, 10};
```

Stack	Space
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	🟩
AA3	🟩
AA2	🟩
AA1	🟩
AA0	🟩

Array Initialization List

```
int arr[10] = {2, 4, 6, 8, 10};

for(int x = 0; x < 10; x++){

    printf("arr[%d] = %d\n", x, arr[x]);

}
```



```
arr[0] = 2
arr[1] = 4
arr[2] = 6
arr[3] = 8
arr[4] = 10
arr[5] = 0
arr[6] = 0
arr[7] = 0
arr[8] = 0
arr[9] = 0
```

Default Values for Different Data Types

- `int` - 0
- `double` 0.0
- `float` - 0.0
- `char` - ‘\0’ Null Character
- `pointer` - Null

Variable Length Arrays ☹️

- The arrays we are dealing with use static memory (stack space).
- Static means no flexibility in changing the size of memory required.
- Adding this flexibility results in dynamic memory
- We will study this at the end of the semester.
- Never use variables when declaring an array as you can have potential danger in what the value a variable can hold.
- VLAs pose danger if we accidentally change a value to a size that can’t be properly handled in memory.

Variable Length Arrays ☹️

```c
int size;

printf("Enter the number of elements: ");

scanf("%d", &size);

int arr[size]; // GROSS!
```

NEVER DO THIS!

Week 9 - Class I: D Arrays Part III

ArraySubscripts

- Subscript are used to access and manipulate array elements.
- It’s very important to know how to manipulate array elements.

Statement	Explanation
`x[i-1] = x[i];`	Assign the value stored at index i to index i-1
`x[i] = x[i+1];`	Assignment the value stored at index i + 1 to index i
`x[i] -1 = x[i]`	Illegal!

Array Subscript Example

Here

```
for (int x = 0; x < 5; x++){
    arr[x] = arr[x + 1];
}
```

Stack Space	
AA9	arr[9] = 10
AA8	arr[8] = 9
AA7	arr[7] = 8
AA6	arr[6] = 7
AA5	arr[5] = 6
AA4	arr[4] = 5
AA3	arr[3] = 4
AA2	arr[2] = 3
AA1	arr[1] = 2
AA0	arr[0] = 1

`sizeof()` Operator

- In C, there's an operator that programmers can use to determine the exact size of the array.
- `sizeof()` is an operator that is used to determine the size of a variable allocated for memory.
    - Integer: 4 bytes
    - Double: 8 bytes
    - Character: 1 byte
    - Float (in Eustis): 4 bytes
    - Pointer: 8 bytes
- This operator can be used to determine the number of elements in a predefined array.

```c
int size = sizeof(arr)/sizeof(arr[0]);
```

Using Array Elements as Function Arguments

- We understand how arrays are declared, initialize, and accessed.
- How can arrays be used with other functions?
- Like variables, programmers can pass arrays to other functions.
- Something interesting about arrays are that they are memory addresses.
- What kind of pass-by does that handle?

Using Array Elements as Function Arguments

- Function prototype shows we are passing an array
- What does C pass arrays by reference?
- It is *Far* more efficient to always pass a pointer than to pass a copy of the entire array!

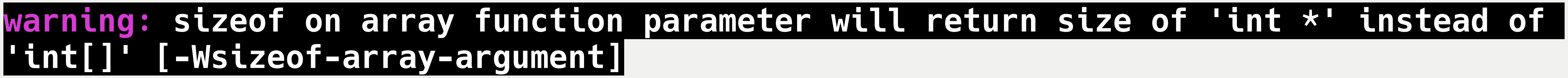
```c
#include<stdio.h>
# define SIZE 10

void fillArray(int list[], int val);

int main(void){

int list[SIZE];

fillArray(list, SIZE);

for(int i = 0; i < SIZE; i++){
    printf("arr[%d] = %d\n", i, list[i]);
}

return 0;
}

void fillArray(int list[], int val){

    for(int i = 0; i < sizeof(list)/sizeof(list[0]); i++){
        list[i] = val;
    }

}
```

Using Array Elements as Function Arguments

- In this code, you might notice that `sizeof()` operators are being used to calculate the # of elements.

- However, there is an issue with this code and we will get a compiler warning!

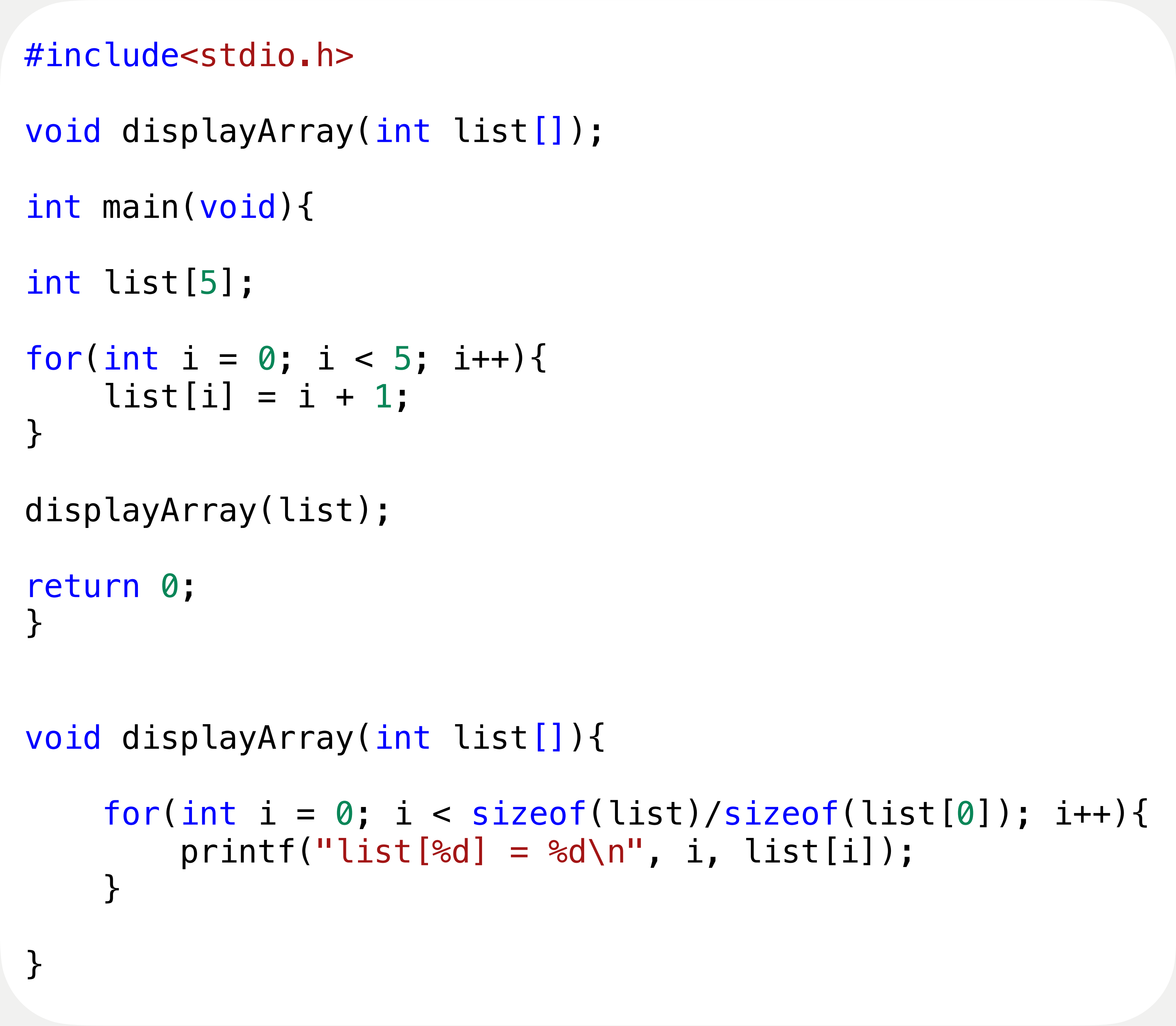
```c
#include<stdio.h>

void displayArray(int list[]);

int main(void){

int list[5];

for(int i = 0; i < 5; i++){
    list[i] = i + 1;
}

displayArray(list);

return 0;
}


void displayArray(int list[]){

    for(int i = 0; i < sizeof(list)/sizeof(list[0]); i++){
        printf("list[%d] = %d\n", i, list[i]);
    }

}
```



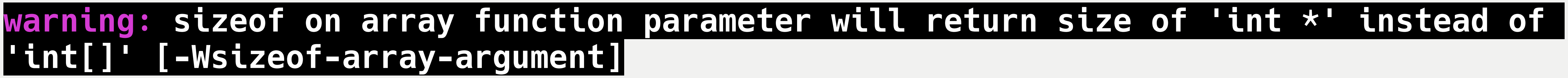
```
warning: sizeof on array function parameter will return size of 'int *' instead of
'int[]' [-Wsizeof-array-argument]
```

Using Array Elements as Function Arguments

- In this code, you might notice that `sizeof()` operators are being used to calculate the # of elements.
- However, there is an issue with this code and we will get a compiler warning!
- Remember a pointer is 8 bytes, and an integer is 4 bytes.



```c
#include<stdio.h>

void displayArray(int list[]);

int main(void){

int list[5];

for(int i = 0; i < 5; i++){
    list[i] = i + 1;
}

displayArray(list);

return 0;
}


void displayArray(int list[]){

    for(int i = 0; i < sizeof(list)/sizeof(list[0]); i++){
        printf("list[%d] = %d\n", i, list[i]);
    }

}
```

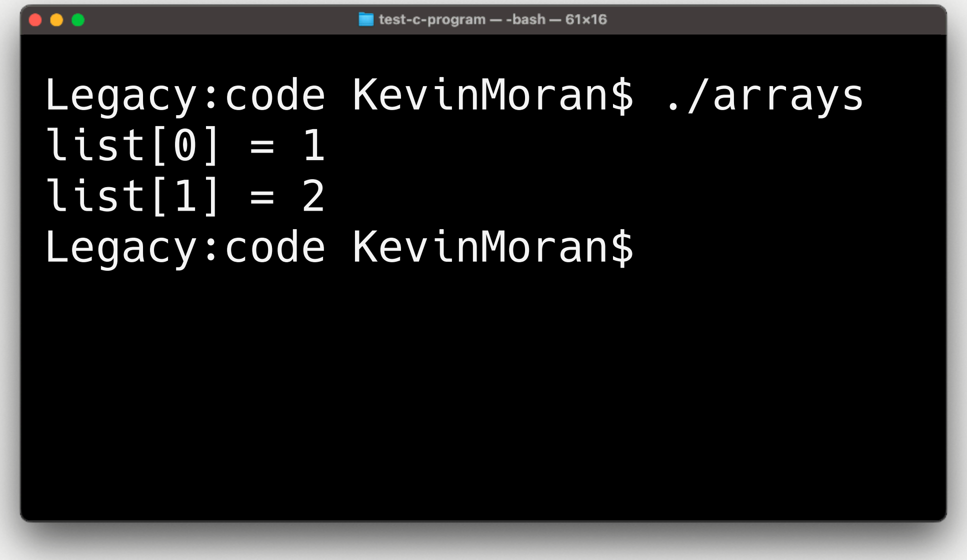


```
warning: sizeof on array function parameter will return size of 'int *' instead of
'int[]' [-Wsizeof-array-argument]
```

Using Array Elements as Function Arguments

```c
#include<stdio.h>

void displayArray(int list[]);

int main(void){

int list[5];

for(int i = 0; i < 5; i++){
    list[i] = i + 1;
}

displayArray(list);

return 0;
}


void displayArray(int list[]){

    for(int i = 0; i < sizeof(list)/sizeof(list[0]); i++){
        printf("list[%d] = %d\n", i, list[i]);
    }

}
```

- What happens if we run this code?



```
Legacy:code KevinMoran$ ./arrays
list[0] = 1
list[1] = 2
Legacy:code KevinMoran$
```

- What is going on??

Week 9 - Class II: 2-D Arrays Part I

2D Arrays

- We have seen that arrays can be useful, but what if we need to store multidimensional data?

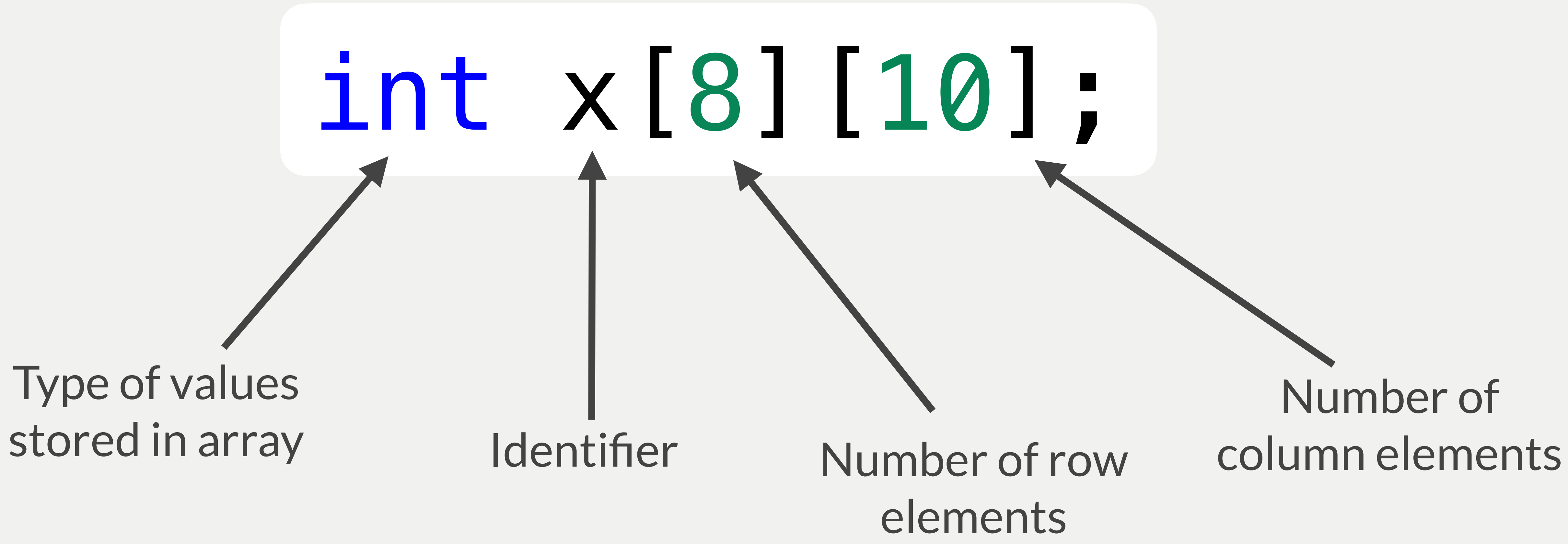
- 2D-Arrays to the rescue!

- 2D Arrays allow us to store information in a matrix-like format, as shown below.

	0	1	2	3
0	a	s	d	f
1	n	k	i	v
2	h	j	k	l
3	f	e	o	p

Example of a 2-D Array of Characters

Declaring a 2D Array



```
int x[8][10];
```

Type of values stored in array

Identifier

Number of row elements

Number of column elements

Accessing Array Elements

```
int arr[3][3]  = { {24, 15, 34}, {26, 134, 194}, {67, 23, 345} };
```

	0	1	2
0	24	15	34
1	26	134	194
2	67	23	345

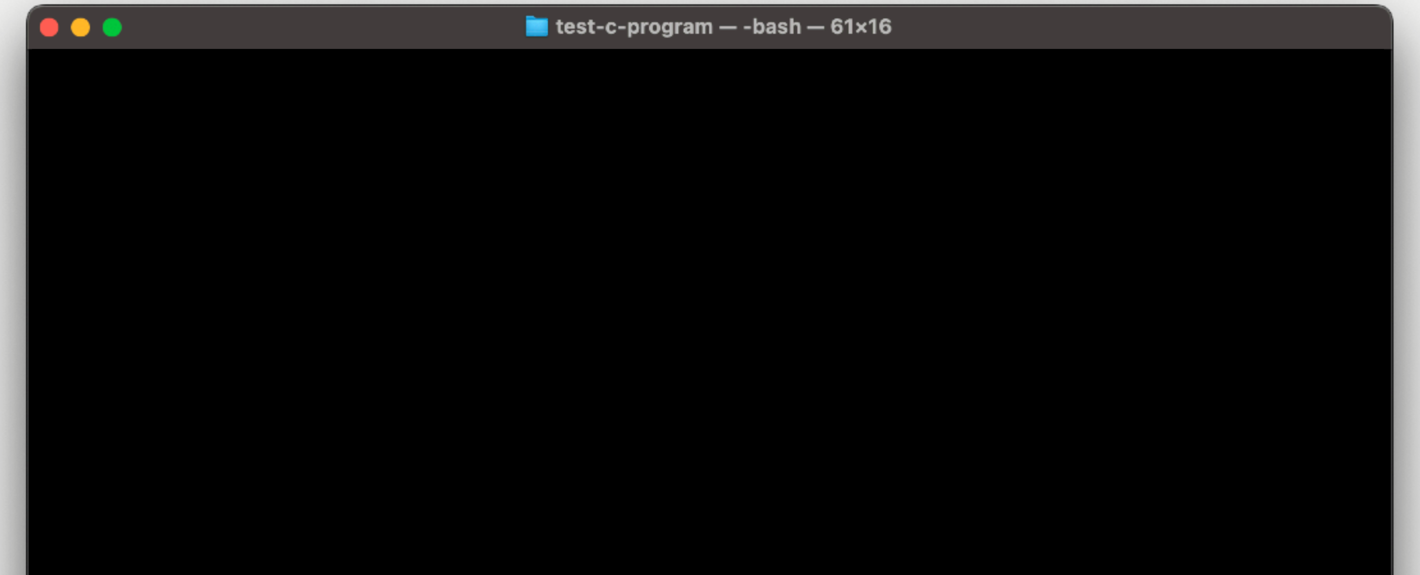
```
int test_val = arr[1][0];

printf("First element in second row is: %d\n", test_val);
```

Week 9 - Class III: 2-D Arrays Part II

2D-Array Stack Visualization

Here

```
int arr[3][3]  = { {24, 15, 34},
                   {26, 134, 194},
                   {67, 23, 345} };

for(int i =0; i < 3; i++){
    for(int j = 0; j < 3; j ++){
        printf("arr[%d][%d] value is: %d\n",
               i,j,arr[i][j]);
    }
}
```



Stack Space	
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	
AA3	
AA2	
AA1	
AA0	

Acknowledgements

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