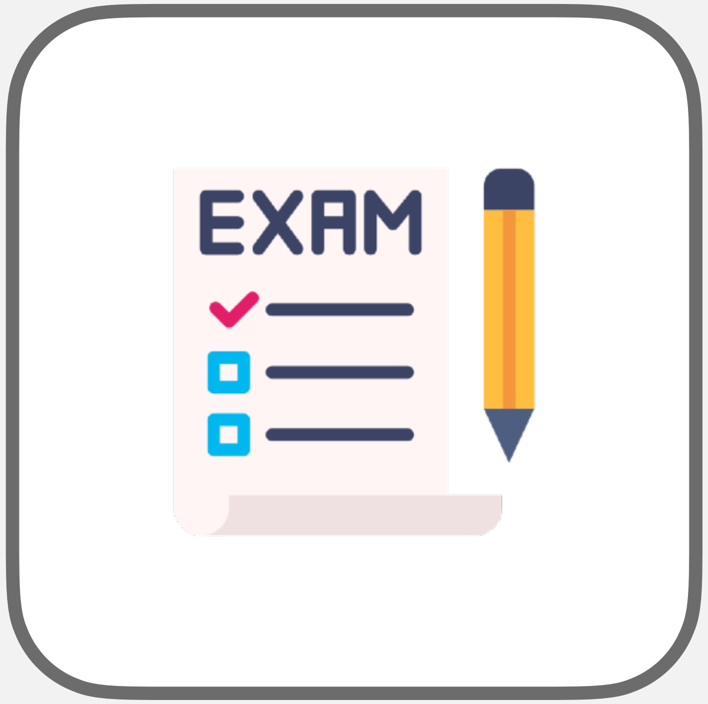
COP 3223H: Introduction to C Programming

Fall 2023

University of Central Florida

Dr. Kevin Moran

Week 5 - Class 2: Exam I Review



Administrivia

- *Small Programming Assignment 2* and *Large Programming Assignment 1* will come out today
    - I will be adjusting the timing of *Small Programming Assignment 3* - moving to later in the semester
- *Quiz 1* is due Today at 11:59 pm
- *Exam 1* is this Friday!
    - We will review the format and content extensively today.

Exam I Format

- 2 Parts, In-class exam, closed book, 100 points total
  - *Part 1:* Short Answer Questions
    - 7-8 questions
    - Either provide program output or answer with a code snippet or a few short sentences.
  - *Part 2:* Programming Questions
    - 4-5 questions
    - Either provide the output of a more complex program, or write several lines of code
  - Covers material from Weeks 1-5
  - You will have the ***entire*** class period to complete the exam
  - Please bring your UCF ID to the exam

Midterm Exam Review

Week 1 - Class 2: C Language Elements

Anatomy of a C Program

- Every C Program basically consists of the following parts:
    - Preprocessor Commands `#include <stdio.h>`
    - Functions `int main()`
    - Variables *We will cover next class!*
    - Statements & Expressions `printf("Hello World \n");`
    - Comments

      ```
      // main function -
      // where the execution of program begins
      ```

Semicolons

- In a C program, the semicolon is a statement terminator
- Each individual statement must be ended with a semicolon, as it indicates the end of a logical entity.
- However, whitespace does not matter (I will demonstrate).

Comments

- Comments allow programmers to make notes about their code, and this is generally considered to be good practice.

- Code is often reused, updated, refactored, etc. Therefore, it is important for the author of a certain piece of code to make sure the intent is clear!

- In other words, it helps you to document the reason code was written or document a solution to the problem that the code solves.

- It also allows future coders who work on a past project to see the program intent.

- Syntax:

```
// This comment has one line

 /* This comment has
 many lines!!!*/
```

- Compilers completely ignore comments

Identifiers

- A C identifier is a name used to identify a variable, function, or any other user-defined item.
- An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).
- C does not allow punctuation characters such as @, $, and % within identifiers.
- C is a case-sensitive programming language.
  - Thus, `Manpower` and `manpower` are two different identifiers in C.
- It's best to be consistent in your identifier scheme.
  - in this class, to keep things simple, we will use CamelCase for structs, and snake_case for everything else :-)

Anatomy of Hello World

```c
// Simple C program to display "Hello World"

// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    printf("Hello World \n");

    return 0;
}
```

Preprocessor Directive

- Provides information to the preprocessor
- A preprocessor modifies a c program prior to its compilation
- stdio.h is the standard input/output header file
    - It contains pre-defined functions that we can use!

Anatomy of Hello World

```
// Simple C program to display "Hello World"

// Header file for input output functions
#include <stdio.h>

// main function –
// where the execution of program begins
int main()
{
    // prints hello world
    printf("Hello World \n");

    return 0;
}
```

Main Function

- C programs always execute instructions starting at the main function from top to bottom.
- All c programs are required to have a main function - otherwise ***syntax error.***
- The main function end with `return 0;`
    - This terminates the function (and program) by sending the value 0 back to the operating system of the computer.
    - Other values are used to indicate errors and should not be used!

Anatomy of Hello World

```c
// Simple C program to display "Hello World"

// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins
int main()
{
    // prints hello world
    printf("Hello World \n");

    return 0;
}
```

`printf()` Function

- This is a pre-defined function from the `stdio.h` library

- The function displays information to the user (and can also be useful for debugging)

- It displays text on lines

    - You have to specify the newline character `\n` to create a new line.

Week 1 - Class 3: C Variables & Data Types

User-defined Identifiers

- We choose our own identifiers to name memory cells that will hold data and program result and to name operations that we define.
- Rules for User-Defined Identifiers
  - An identifier must consist only of letters, digits, and underscores.
  - An identifier cannot begin with a digit.
  - A C reserved word cannot be used as an identifier.
  - An identifier defined in a C standard library should not be redefined.
- We will use CamelCase for structs, and snake_case for variables/ functions.

Variables

- Variables are names associated with a memory cell whose value can change.
  - User-Defined Identifiers
- Variable Declarations are statements that communicate to the compiler the names of variables in the program and the kind of information stored in each variable.
  - Syntax
    - int *variable_list;*
    - double *variable_list;*
    - char *variable_list;*

	A	B	C
0	`int x = 0`		
1			`double num = 1.4`
2			
3	`char letter = ‘a’`		
4			

Sample Syntax for Declaring Variables

```
int val;

double x;

float y;

char letter;
```

Data Types

- A set of values and operations that can be performed on those values.
  - Types of Data that can be stored in C:
    1. `int` – integer numbers
    2. `double` – decimal numbers
    3. `float` – similar to double BUT different amount of allocation for memory storage (smaller allocation)
    4. `char` - a character from the keyboard

Type	Range in Typical Implementation
`int`	-2,147,483,647 ... 2,147,483,647
`double`	10^{-307}...10^{308} (15 significant digits)
`float`	10^{-37}...10^{38} (6 significant digits)

`double` and `float` Data Types

- Most beginners think that doubles and floats can be used interchangeably.
  - THIS IS FALSE!!!
- doubles have twice the precision of float type values.
- If they are used interchangeably, then you will likely encounter rounding errors.
- *When in doubt, always use double for extra precision!!!!! If any programming problem does not specify the data type for any real number, use double!!!*

char Data Type

- Data type char represents an individual character value: letter, digit, or a special symbol
  - Ex: ‘A’, ‘z’, ‘2’, ‘9’, ‘*’, ‘:’, ‘”’, ‘ ’
- Characters are represented uniquely in memory as an integer for the system to properly evaluate.
  - The value is known as ASCII Value
  - This can be utilized when comparing characters.

Character	ASCII Code
’ ‘	32
*’	42
‘A’	65
‘B’	66
‘Z’	90
‘a’	97
‘b’	98
‘z’	122
‘0’	48
‘9’	57

Printing Variables

Format Specifier	Data Type	description	Syntax
%d	int	To print the integer value	printf("%d",<int_variable>);
%f	float	To print the floating number	printf("%f",<float_variable>);
%lf	double	To print the double precision floating number or long float	printf("%lf",<double_variable>);
%c	char	To print the character value	printf("%c",<char_variable>);

Week 2 - Class 1: Executable Statements

Assignment Statements

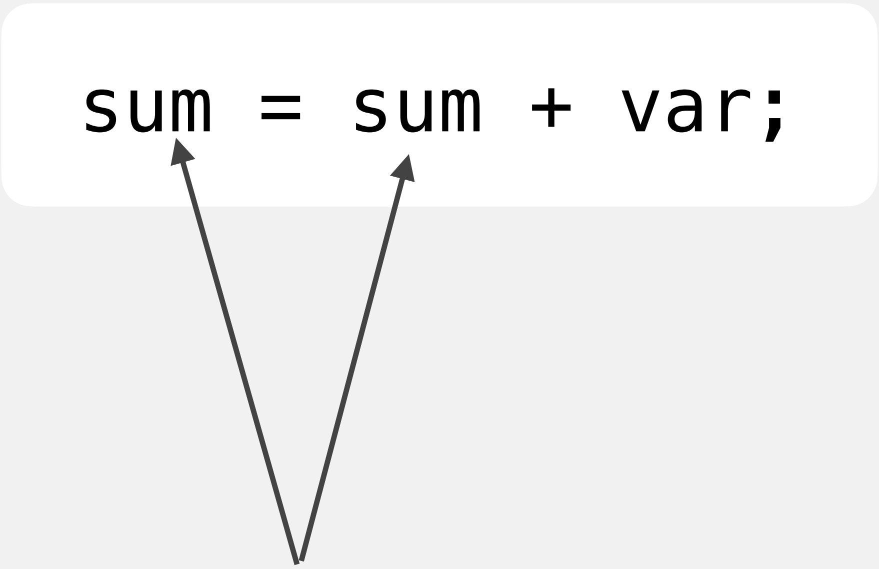
- Assignment statements stores a value or a computational result in a variable and is used to perform most arithmetic operations in a program.
- = is called the assignment operator

```
int var;
var = 32;
```

- Syntax:
  - `variable = expression;`

Compound Assignment Statements

- In C, you can create *compound assignment statements* in the form of:



```
sum = sum + var;
```

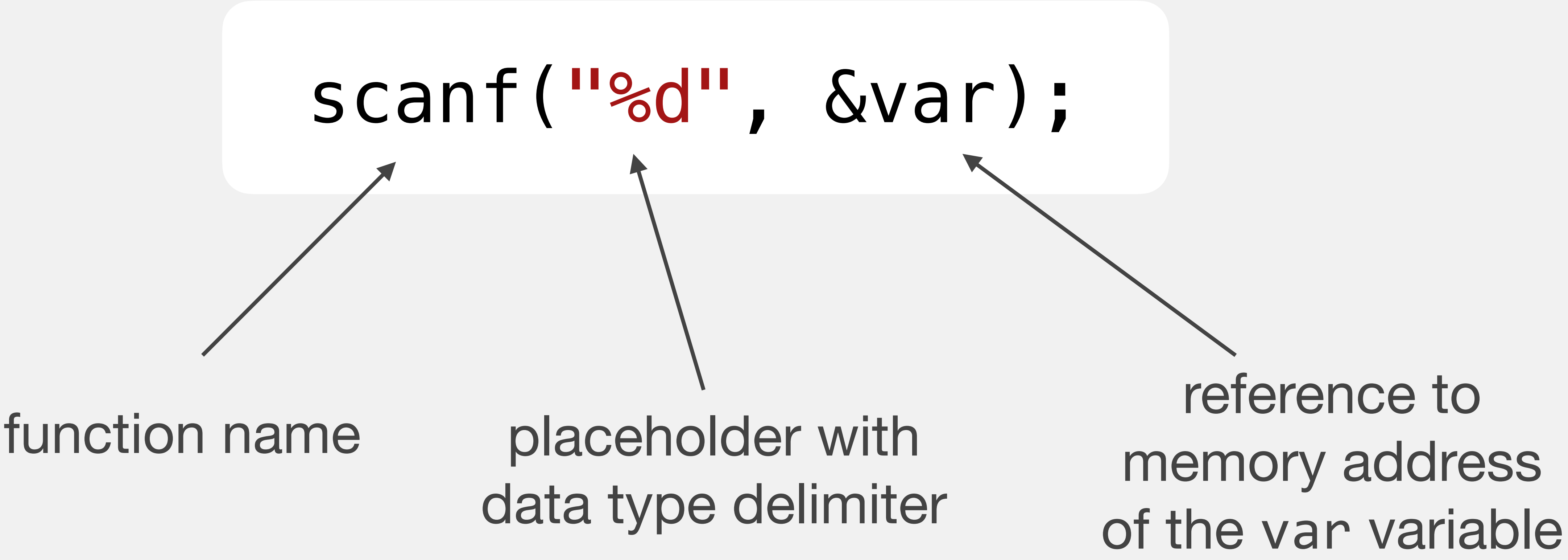
Yes! You are seeing double! Let’s take a look at what is happening in a statement like this!

Printing Special Characters

Escape Sequence	Meaning
\a	Alert
\b	Backspace
\n	Newline
\t	Horizontal Tab
\v	Vertical Tab
\\	Backslash
\’	Single Quote
\”	Double Quote
\?	Question Mark
%%	Percent Symbol

Accepting User Input with `scanf()`

- Copies data into a variable stored in memory
- Collects user input through the keyboard and stores the value into the respective address of the variable in memory



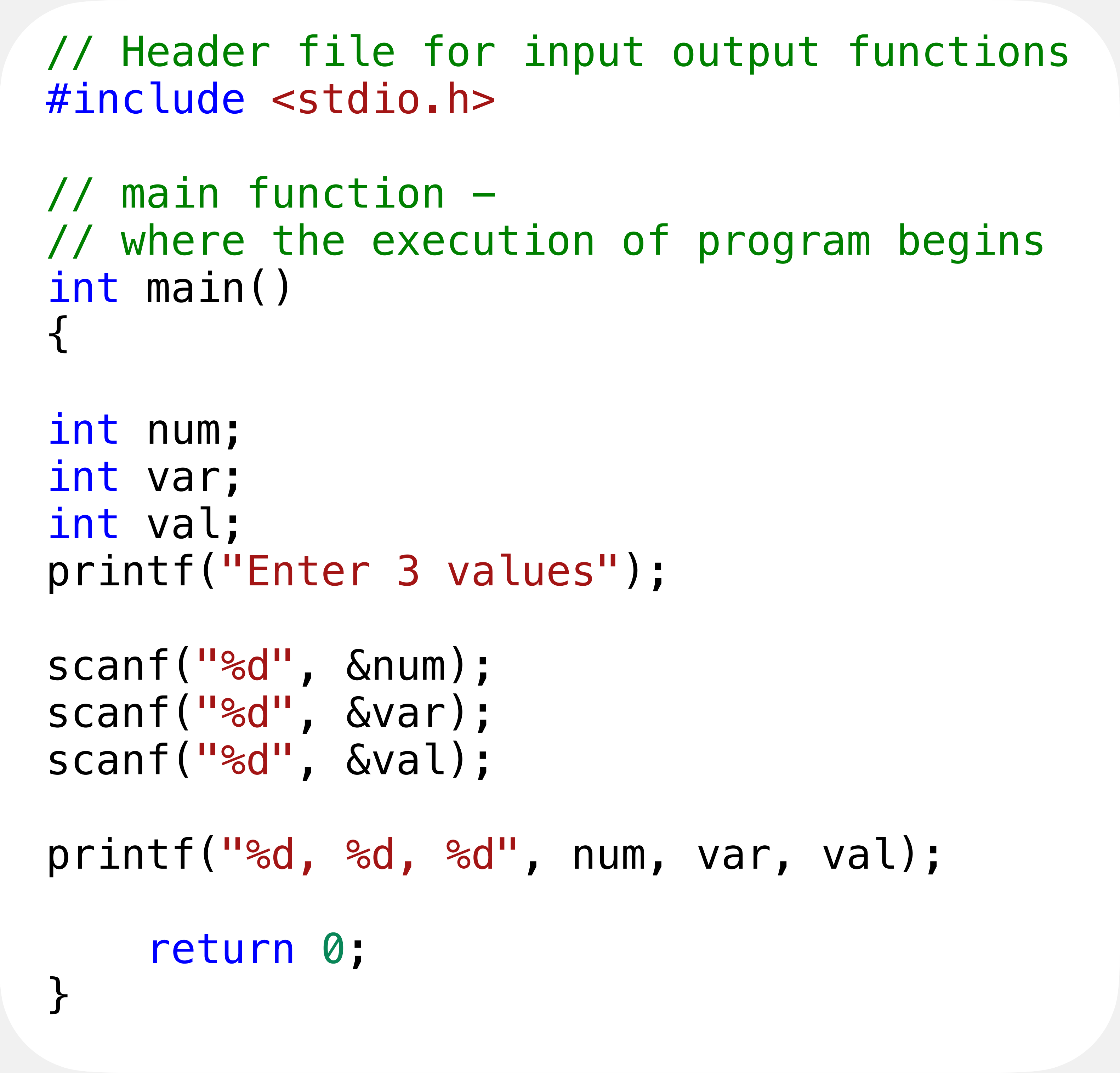
```
scanf("%d", &var);
```

function name

placeholder with data type delimiter

reference to memory address of the `var` variable

Example scanf()



```c
// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins
int main()
{

int num;
int var;
int val;
printf("Enter 3 values");

scanf("%d", &num);
scanf("%d", &var);
scanf("%d", &val);

printf("%d, %d, %d", num, var, val);

    return 0;
}
```

Example scanf()

```
// Header file for input output functions
#include <stdio.h>

// main function -
// where the execution of program begins
int main()
{

int num;
int var;
int val;
printf("Enter 3 values");

scanf(“%d%d%d”, &num, &var, &val);

printf("%d, %d, %d", num, var, val);

    return 0;
}
```

Return Statements

- Return terminates the function and transfers control from a function back to the activator of the function. For the main function, the control is transferred back to the operating system.

- A value is sent back to the operating system.
  - 0 means code executed successfully
  - 1 means code executed with run time error (code crash).

```
return 0; // function terminator
```

Constant Macro

- A name that is replaced by a particular constant value before program is sent to compiler
- Always seen at the top of a program file.
- Syntax:

```
#define MILES PER KM 0.62137
```

Week 2 - Class 3: Arithmetic Expressions & Library Functions

Modulus Operator

- You may not have heard about the modulus operator (remainder operator).
- The modulus operator returns the remainder value of a division result.
- Example: $\frac{4}{3}$ would result with the remainder 1
- The symbol denoted in C uses % to represent the modulus operator.
    - In mathematics (such as discrete mathematics) the notation *mod* also represents the modulus operator. In this course, we will only use the notation %.

```c
int result = 4 & 3;
printf ("4 & 3 = %d\n”, result);
```

Arithmetic Expressions

Arithmetic Operator	Meaning	Examples
+	addition	5 + 2 = 7 5.0 + 2.0 = 7.0
-	subtraction	5 - 2 = 3 5.0 - 2.0 = 3.0
*	multiplication	5 * 2 = 10 5.0 * 2.0 = 10.0
/	division	5.0 / 2.0 = 2.5 5 / 2 = 2
%	remainder	5 % 2 = 1

Casting

- Casting is converting an expression to a different type by writing the desired type in parentheses in front of the expression.

```
double n;
double x = 0.5;

n = (int)(9 * 0.5); //casting
```

What value does n *hold?*

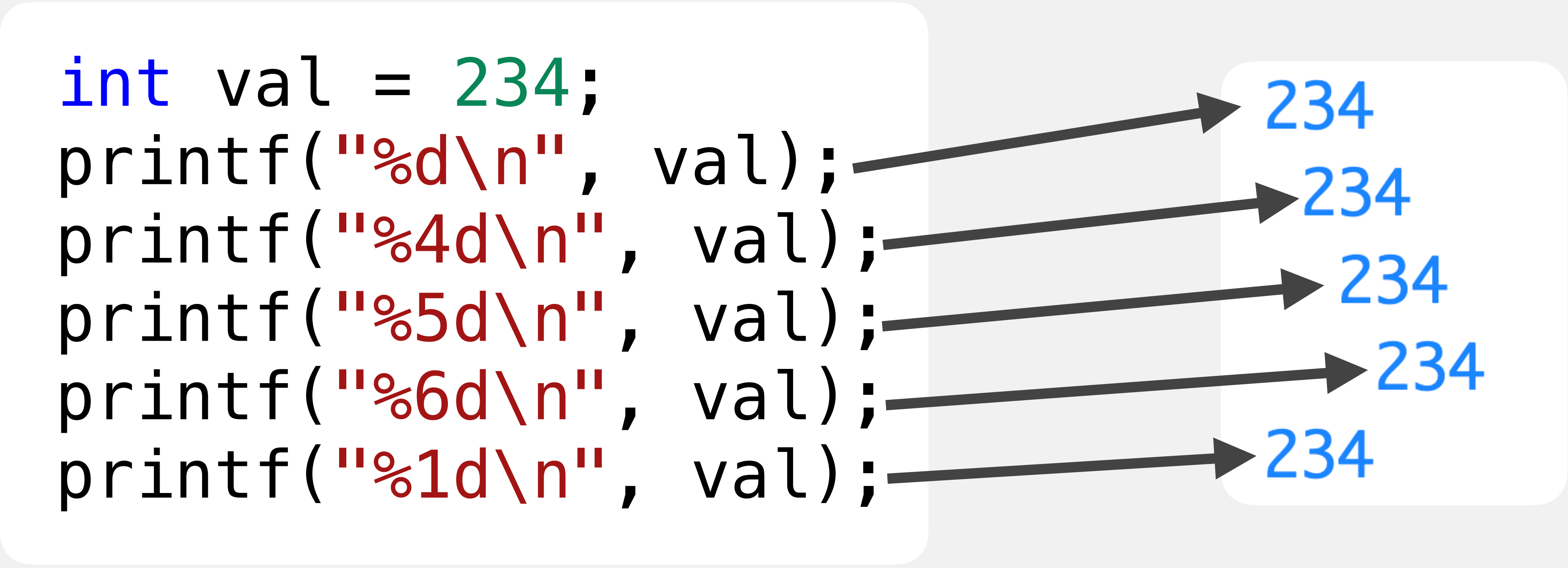
a) 4
b) 4.0
c) 4.5
d) 5

Writing Mathematical Formulas in C

Mathematical Formula	C Expression
$b^2 - 4ac$	`b * b - 4 * a * c;`
$a + b - c$	`a + b - c;`
$\frac{a+b}{c+d}$	`1 / (1 + x * x);`
$\frac{1}{1+x^2}$	`1 / (1 + x * x);`
$a * -(b+c)$	`a * - (b + c);`

Formatting Output

- C allows you to format output of numbers for consistency.
    - You can control the number of spaces
    - Text automatically aligns to the right



```c
int val = 234;
printf("%d\n", val);
printf("%4d\n", val);
printf("%5d\n", val);
printf("%6d\n", val);
printf("%1d\n", val);
```

```
234
 234
  234
   234
234
```

Code File

Output

The Math Library

- The C language has a math library with predefined functions that perform certain mathematical tasks.
- Task Examples: square root, Trigonometry, etc…
- `#include <math.h>` imports all reusable math functions

Math Library Functions

Function	Header File	Purpose	Argument(s)	Result
`abs(x)`	`<stdlib.h>`	Absolute Value	`int`	`int`
`ceil(x)`	`<math.h>`	Round Up	`double`	`double`
`cos(x)`	`<math.h>`	Cosine	`double (radians)`	`double`
`exp(x)`	`<math.h>`	Natural Exponent	`double`	`double`
`floor(x)`	`<math.h>`	Round Down	`double`	`double`
`log(x)`	`<math.h>`	Natural Logarithm	`double`	`double`
`log10(x)`	`<math.h>`	Base 10 Logarithm	`double`	`double`
`pow(x,y)`	`<math.h>`	x^y	`double`	`double`
`sin(x)`	`<math.h>`	Sine	`double`	`double`
`sqrt(x)`	`<math.h>`	Square Root	`double`	`double`
`tan(x)`	`<math.h>`	Tangent	`double`	`double`

Class Demo

- Write a program that computes the quadratic function.
  - This is defined as follows:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Revisiting the `scanf()` Function

- We know that functions return a value
- What does `scanf()` return?

```c
#include <stdio.h>

int main()
{
    int var1;
    double var2;
    int var3;

    printf("Enter 3 values:");
    int result = scanf("%d%lf%d", &var1, &var2, &var3);
    printf("result = %d\n", result);

    printf("Enter 2 values:");
    result = scanf("%d%d", &var1, &var3);
    printf("result = %d\n", result);

    return 0;
}
```

Week 3 - Class 1: User-defined Functions

User-defined Functions

- One way that programmers implement top-down design is defining their own functions (user defined functions)
- User defined functions are sets of instructions that are *defined* by the programmer
- Programmers will break down a larger problem into subproblems and will solve these subproblems in user-defined functions
- In order to invoke the function, you must *call* it

Function Prototypes

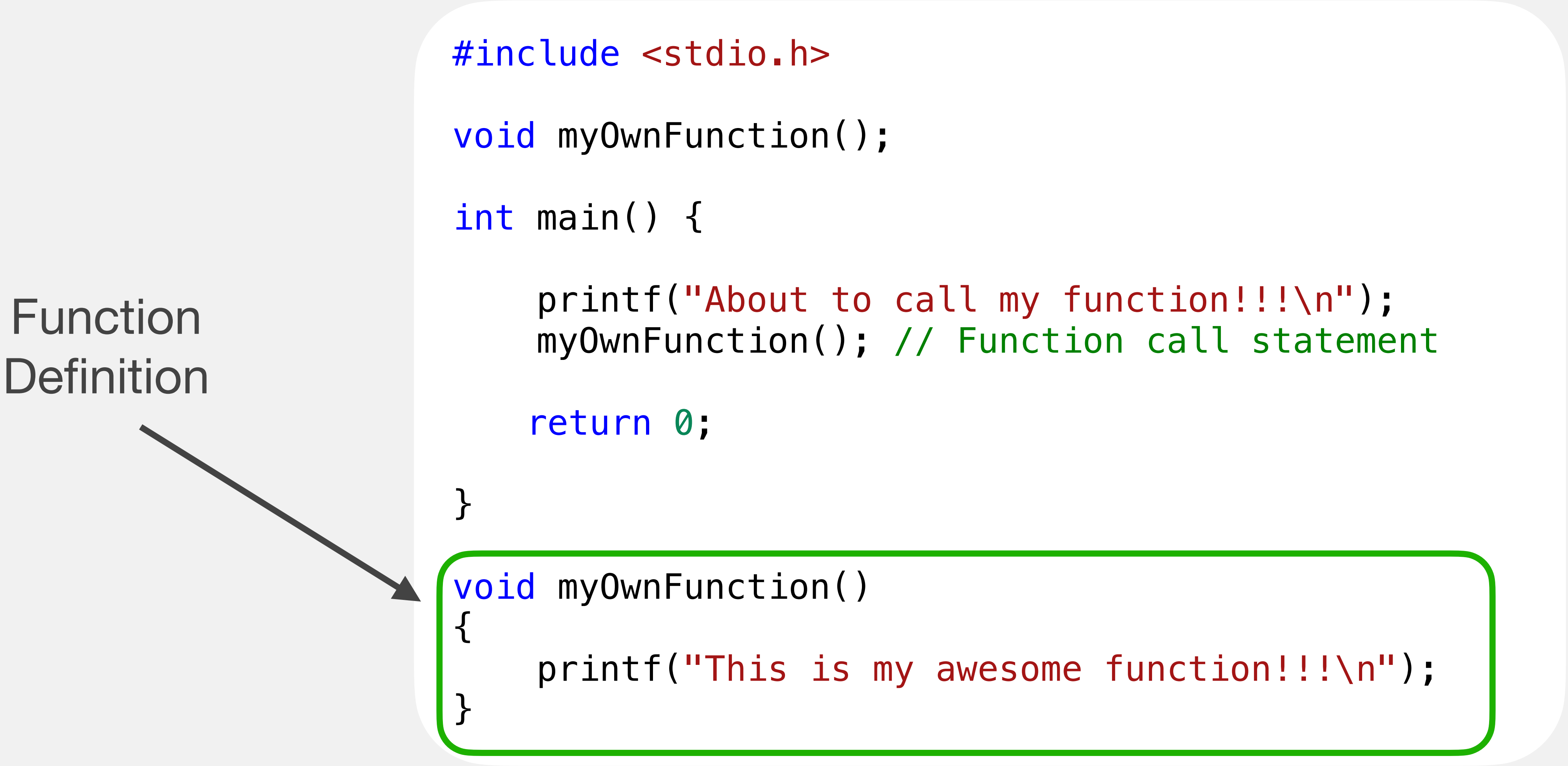
- Just like variables, functions must be declared as well.
- Prototypes allows the Operating System know how much memory space needs to be reserved based on the return type and arguments.

Function Prototype

```c
#include <stdio.h>

void myOwnFunction();

int main() {

    printf("About to call my function!!!\n");
    myOwnFunction(); // Function call statement

    return 0;

}

void myOwnFunction()
{
    printf("This is my awesome function!!!\n");
}
```

Function Definitions

- Just like declaring a variable, you must assign it a value.
- For function definitions, you must write out the set of instructions to perform the task that needs to be written out.



Function Definition

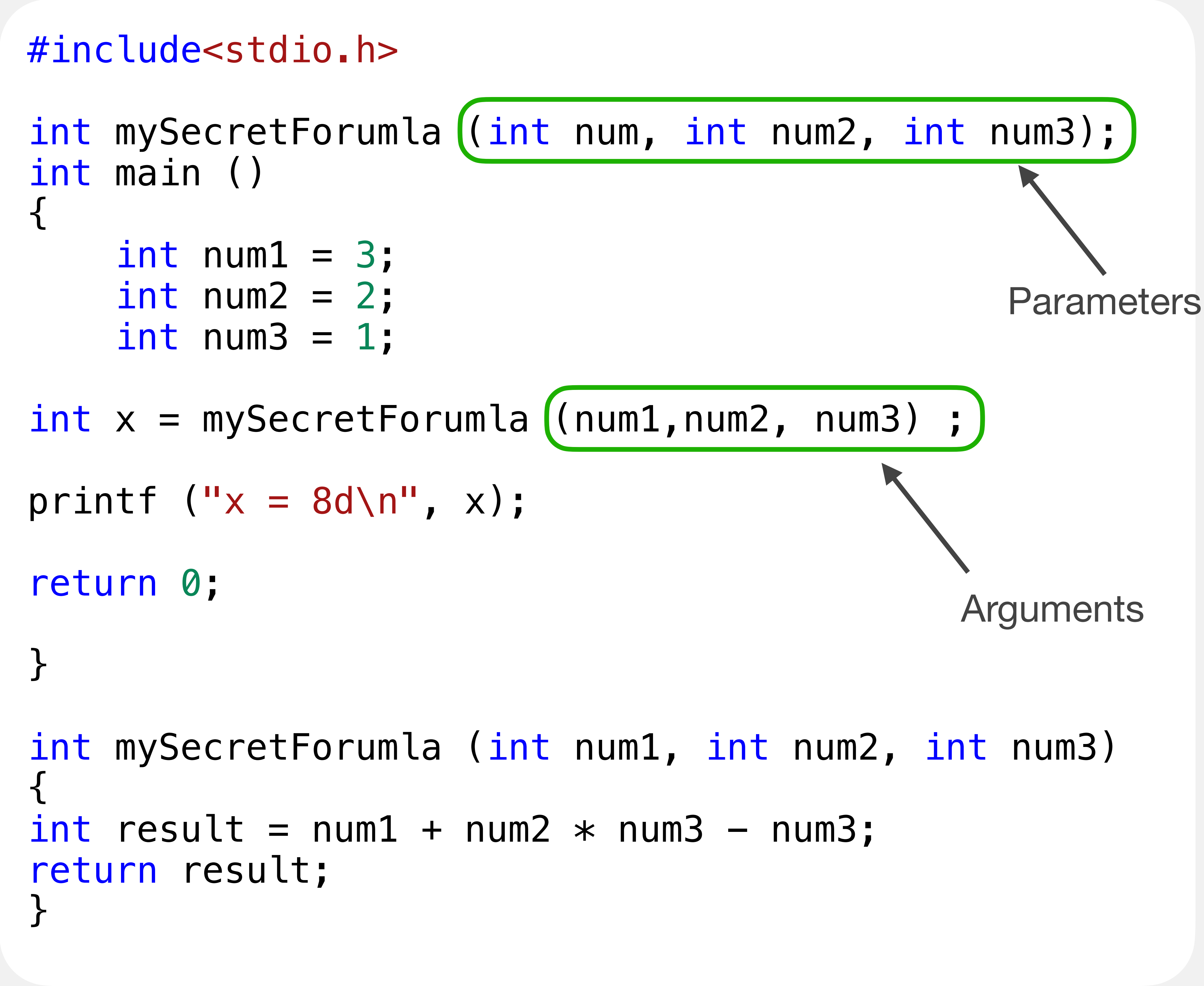
```c
#include <stdio.h>

void myOwnFunction();

int main() {

    printf("About to call my function!!!\n");
    myOwnFunction(); // Function call statement

   return 0;

}

void myOwnFunction()
{
    printf("This is my awesome function!!!\n");
}
```

Types of Functions

- There are ***two types*** of functions.
  - Functions that *return a value*.
  - Functions that *don't return a value*.
- These types of functions are defined through their prototypes.
  - Functions that don't return a value have the reserved word `void` in front of the name.
  - Functions that do return a value have the type of data (`int`, `double`, `char`) in front of the of the name.

Week 3 - Class 11: User-defined Functions II

Functions with Arguments/Parameters



```
#include<stdio.h>

int mySecretForumla (int num, int num2, int num3);
int main ()
{
    int num1 = 3;
    int num2 = 2;
    int num3 = 1;

int x = mySecretForumla (num1,num2, num3) ;

printf ("x = 8d\n", x);

return 0;

}

int mySecretForumla (int num1, int num2, int num3)
{
int result = num1 + num2 * num3 - num3;
return result;
}
```

Parameters

Arguments

Pass By Value

- Whenever a function with arguments is called, they must share the values properly.

- One way of doing this is pass by value.

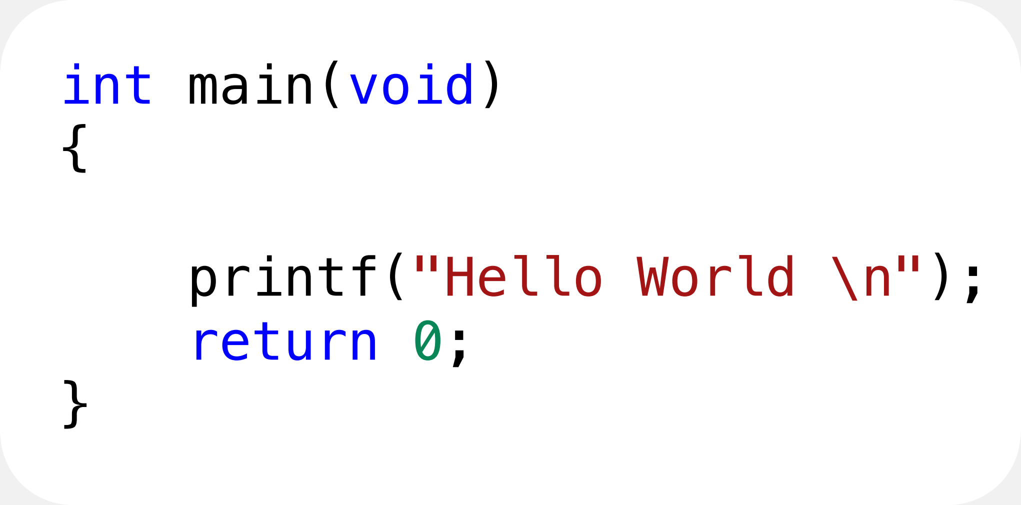
- Pass by value is when a value stored in memory (stack space) is ***copied*** and sent over to the proper parameter of the respective function (which is also stored in a different location of the stack space).

- The following set of slides shows a demonstration.

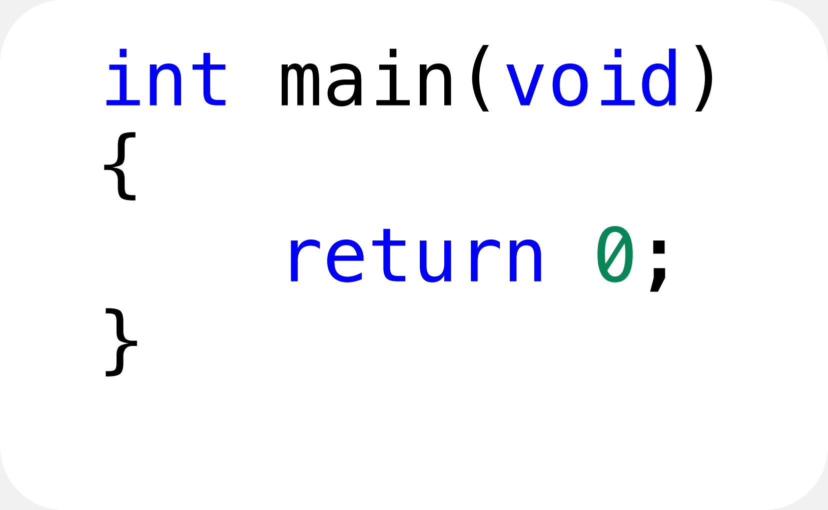
Week 4 - Class 1: Control Structures & Conditionals

Control Structures

- Control structures are a combination of individual instructions into a single logical unit with one entry point and one exit point
- Compound Statement is a group of statements bracketed `{ and }` that are executed sequentially.



```
int main(void)
{

    printf("Hello World \n");
    return 0;
}
```



```
int main(void)
{
    return 0;
}
```

Variable Scope

- Now that we have learned control structures, it is time to discuss variable scope.
- Scope is the level of access a variable has in a program run
- There are two types of scopes with variables.
  - Global Scope (*Bad!!!!!*)
  - Local Score (*Good!!!*)
- Global means all components (functions have access to the value and can manipulate it)
  - Why is that bad?
  - Never use Global Variables in this course unless Dr. Moran says it is ok
- Local means only the component within the control structure has access the value and can perform certain operations on it.
  - *Good Practice!!!*

Relational & Equality Operators

- When evaluating expressions, we make comparisons.
- There are 6 relational/equality operators.
    - Less than (<)
    - Greater than (>)
    - Less than or equal to (<=)
- Greater than or equal to (>=)
    - Equal to (==)
    - Not Equal to (!=)
- Important! = and == are two different operators!!
    - = is the assignment operator
    - == is the equality operator

Relational & Equality Operators in C

Operator	Meaning	Type
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
(==)	equal to	equality
!=	not equal to	equality

Logical Operators

- An expression that uses one or more of the three logical operators
  - && (and)
  - || (or)
  - ! (not)
  - && and || operators allows us to combine a set of conditions
- Examples:
  - `in_range = (num >= -10 && num <= 10)`
  - `is_letter = (letter == ‘a’ || letter == ‘b’)`
- ! operator complements (opposite result) the condition
- Examples:
  - `num1 == num2`
  - `!(num1 == num2)`

Logical Tables

The && Operator

Operand 1	Operand 2	Operand 1 && Operand 2
nonzero (T)	nonzero (T)	1 (T)
nonzero (T)	0 (F)	0 (F)
0 (F)	nonzero (T)	0 (F)
0 (F)	0 (F)	0 (F)

The || Operator

Operand 1	Operand 2	Operand 1 && Operand 2
nonzero (T)	nonzero (T)	1 (T)
nonzero (T)	0 (F)	1 (T)
0 (F)	nonzero (T)	1 (T)
0 (F)	0 (F)	0 (F)

The ! Operator

Operand 1	! Operand 1
nonzero (T)	0 (F)
0 (F)	1 (T)

Operator Precedence in C

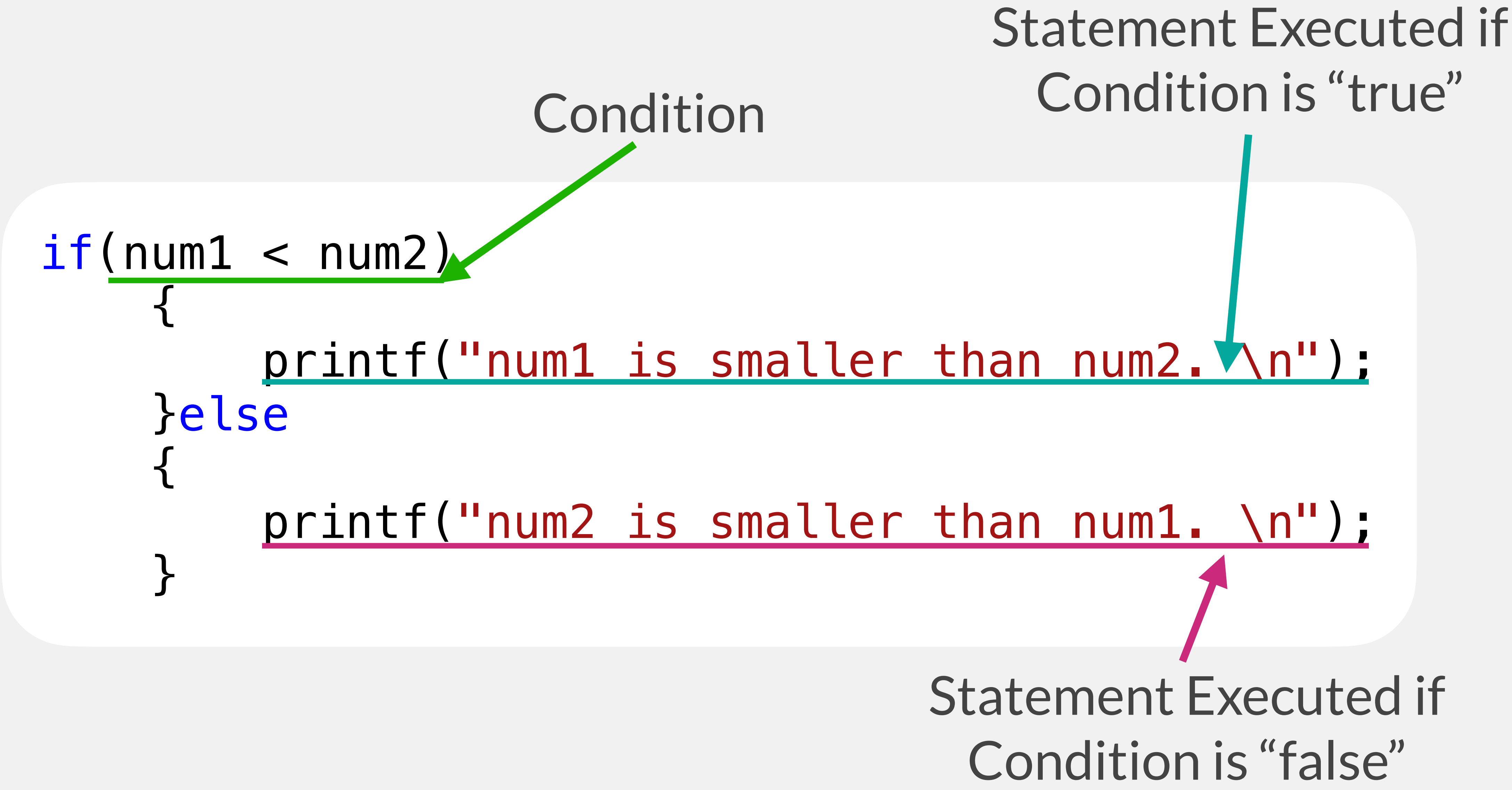
Operator	Precedence
function calls	Highest
! + - & (unary)	
* / %	
+ -	
< <= >= >	
!= ==	
&&	
||	
(=)	Lowest

Week 4 - Class 11: If Statements

The `If` Statement

- Conditions are setup in the if statement.

- Syntax example



```
if(num1 < num2)
    {
        printf("num1 is smaller than num2. \n");
    }else
    {
        printf("num2 is smaller than num1. \n");
    }
```

Condition

Statement Executed if Condition is “true”

Statement Executed if Condition is “false”

`If` Statement with One Alternative

- Conditions are setup in the `if` statement.
- Syntax example

```c
if(num1 != num2)
        printf("num1 does not equal num2. \n");
```

Q&A: What happens if the condition is false?

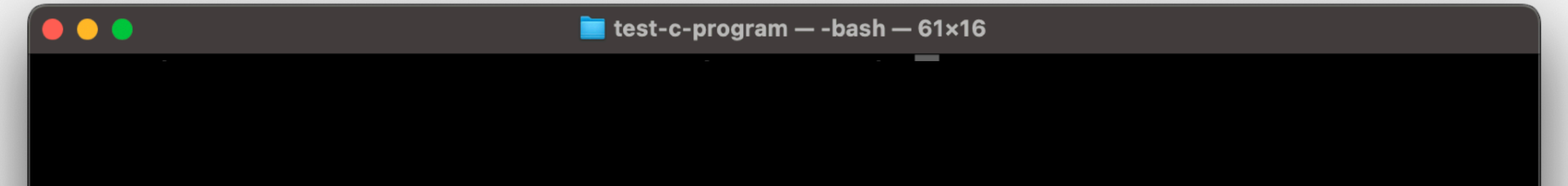
a) Program crashes at runtime
b) Program does not execute the printf statement
c) Program won't compile
d) None of the above

Compound If Statement Example #1

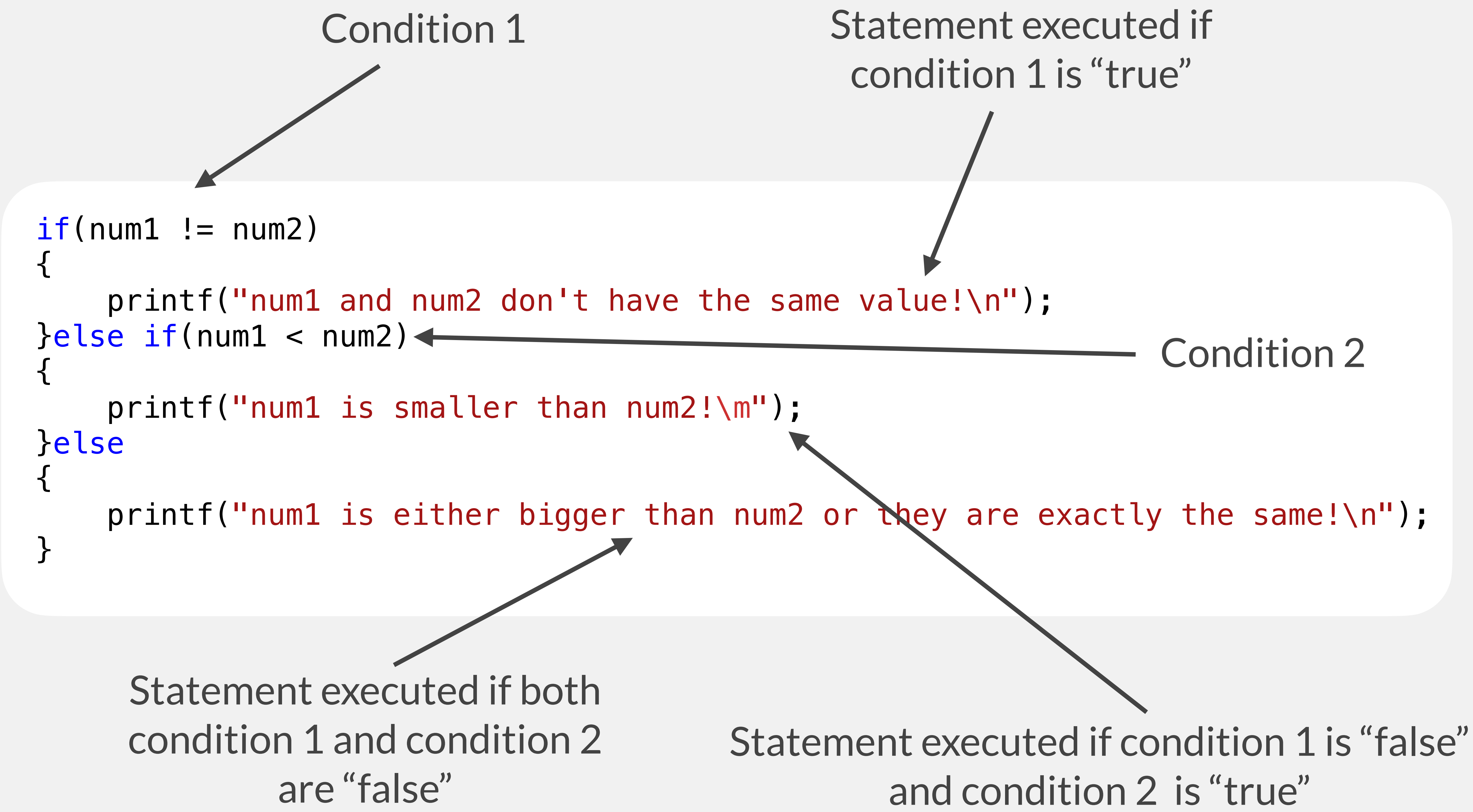
Here

```
#include <stdio.h>

int main(void)
{
    int num1;
    int num2;

    scanf("%d%d", &num1, &num2);

    if(num1 != num2)
    {
        printf("num1 is smaller than num2. \n");
        printf("Still in the true block. \n");
    }else
    {
        printf("num2 is smaller than num1. \n");
        printf("Still in the false block. \n");
    }

    printf("I will always be displayed! \n");

    return 0;

}
```

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



Multiple Alternative `if-else` Statement



Condition 1

Statement executed if condition 1 is “true”

```
if(num1 != num2)
{
    printf("num1 and num2 don't have the same value!\n");
}else if(num1 < num2)
{
    printf("num1 is smaller than num2!\m");
}else
{
    printf("num1 is either bigger than num2 or they are exactly the same!\n");
}
```

Condition 2

Statement executed if both condition 1 and condition 2 are “false”

Statement executed if condition 1 is “false” and condition 2 is “true”

Nested `if` Statements

- After testing and determining the outcome, it is possible to dive into another condition.
- This is known as creating nested statements.
- Think about nesting dolls!
    - Inside a nest doll is another doll. Inside a nest if statement is another if statement.

```
if (num1 != 0)
    if(num1 !=1)
        if(num1!=2)
            if(num1!=3)
                printf("num is neither 0, 1, 2, or 3 ...");
```

`switch` Statement

- Some of the `if else` statements can deal with checking for an exact match.
- What would happen if there are lots of multiple-alternative `if-else` statements that dealt with only equality checks
- Switch Statement allows programmers to write a cleaner version of `if-else` that only deals with `==` operator.

Q&A: Switch statements use relational operators for comparison?

a) True

b) False

switch Statement Syntax

```
switch(ticket) ←  variable being evaluated for equality
{
    case 1: ←  ticket == 1
        printf("Proceed to entrance 1.\n");
        break;

    case 2: ←  ticket ==2
        printf("Proceed to entrance 2.\n");
        break;

    case 3: ←  ticket ==3
        printf("Proceed to entrance 3.\n");
        break;

    default: ←  ticket !=1 && ticket !=2 && ticket !=3
        printf("Sorry, your ticket does not match!");

}
```

Week 5- Class 1: Grouping Expressions

Operator Precedence in C

- Precedence determines how operators in C are grouped together.

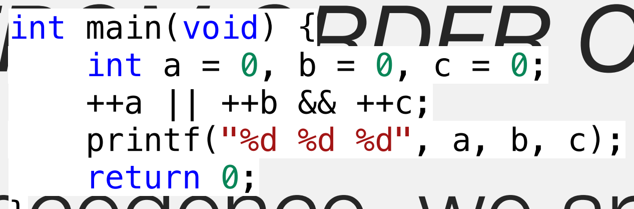
- When we were writing mathematical expressions in C , we learned that "( )" was how we grouped certain operands together for an operator to perform some sort of action.

- Example:

$$\frac{a+b}{c+d} \rightarrow (a+b)/(c+d)$$

Logical Operator Precedence

- !, &&, || are the 3 logical operators in C we utilize
- A common misconception when we talk about precedence with logical operators is who gets to be executed first.
- *VERY DIFFERENT FROM ORDER OF OPERATIONS!!!*



```c
int main(void) {
    int a = 0, b = 0, c = 0;
    ++a || ++b && ++c;
    printf("%d %d %d", a, b, c);
    return 0;
}
```

- When we discuss precedence, we are discussing how logical operators group expressions together and what is being evaluated.

Logical Operator Precedence

```c
int main(void) {
    int a = 0, b = 0, c = 0;
    ++a || ++b && ++c;
    printf("%d %d %d", a, b, c);
    return 0;
}
```

What is the output?

Operator Precedence in C

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

Some Examples

- Assume A, B, C, and D are relation expressions (e.g., `x > y`)
  - `A && B` → `(A && B)`
  - `A && B || C` → `((A&&B) || C)`
  - `A || B && C || D` → `((A || (B&&C)) || D)`
  - `!A` → `!(A)`

Acknowledgements

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