# Algorithms, C, and Systems

**CS 5002**
**Spring 2019**
**Seattle**

Adrienne Slaughter, Ph.D.
ahslaughter@northeastern.edu

Joe Buck
j.buck@northeastern.edu

# Overview

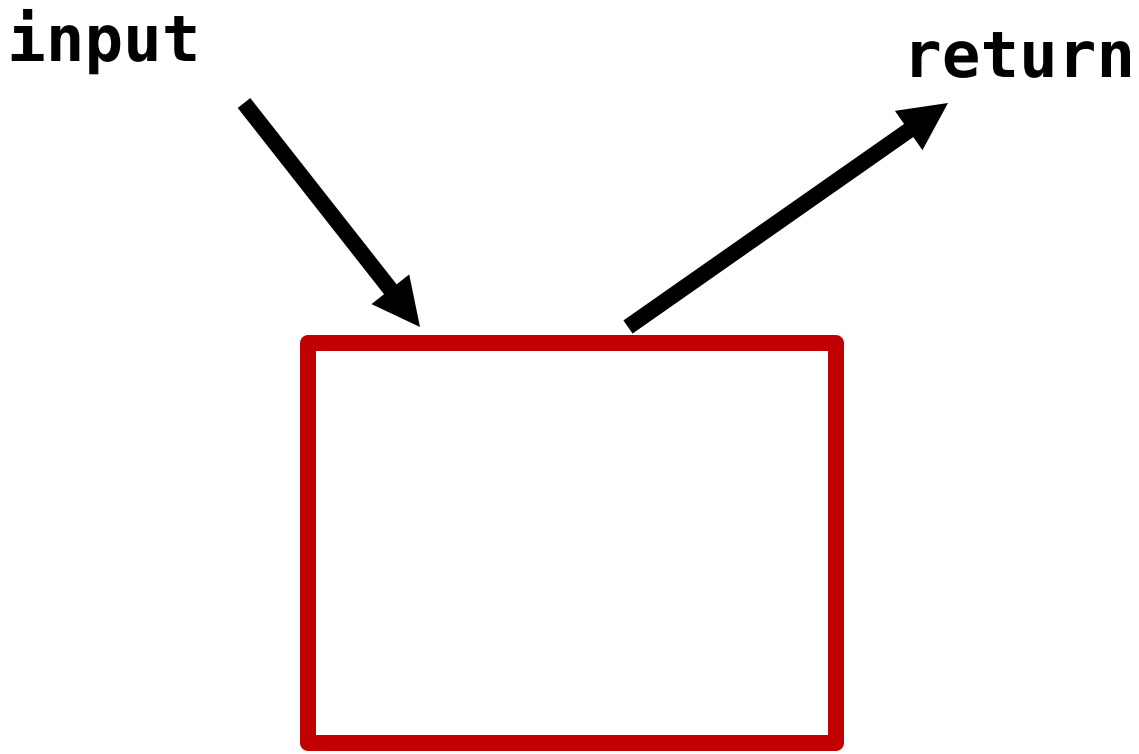
- Introduce function terminology
- Introduce function headers
- Review arrays

# NEXT: FUNCTIONS

# Why functions?

- Software reusability
- Divide and conquer
- Abstraction
- Modularization

# What is a function?



```
return-value-type function-name(parameter list) {
    statements;
}
```

```
return-value-type function-name(parameter list) {
    statements;
}
```

Function signature

```
return-value-type function-name(parameter list) {
    statements;
}
```

**Data type of the result returned.**
**`void` indicates there is no return value.**

```
int function-name(parameter list) {
    statements;
}
```

```
int function-name(parameter list) {
    statements;
}
```

Any valid identifier (name).

```
int square(parameter list) {
    statements;
}
```

```
int square(parameter list) {
    statements;
}
```

A list of the parameters:
`type1 name1, type2 name2, …`

```
int square(int n) {
    statements;
}
```

```
int square(int n) {
    statements;
}
```

The lines of code that do the work.

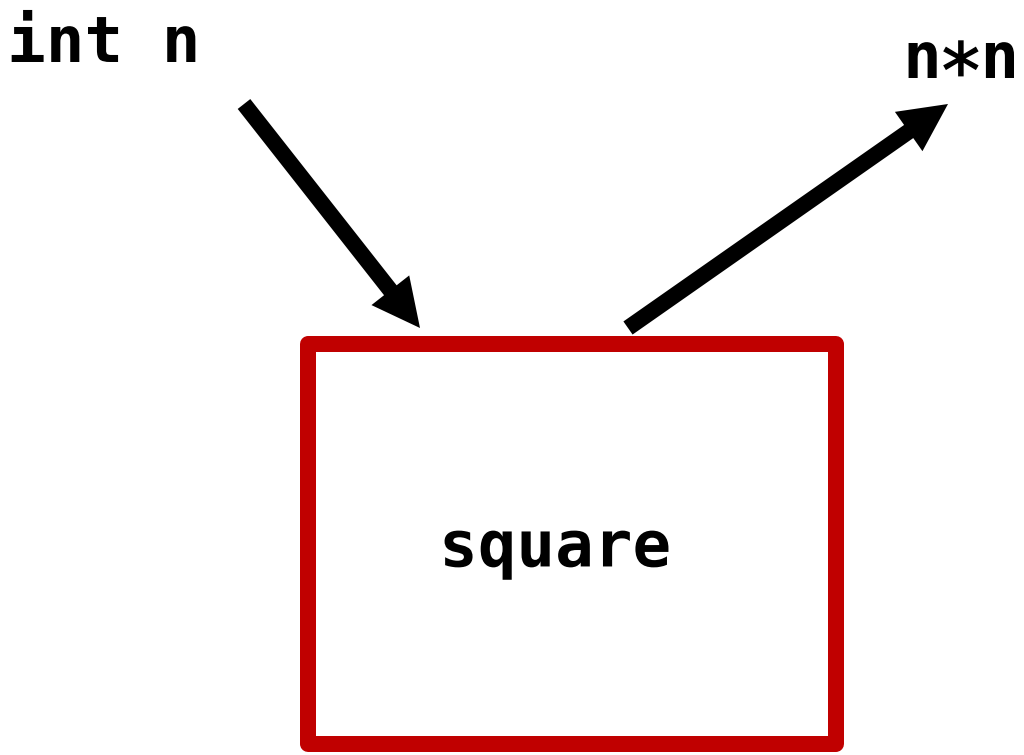
```
int square(int n) {
    return n * n;
}
```

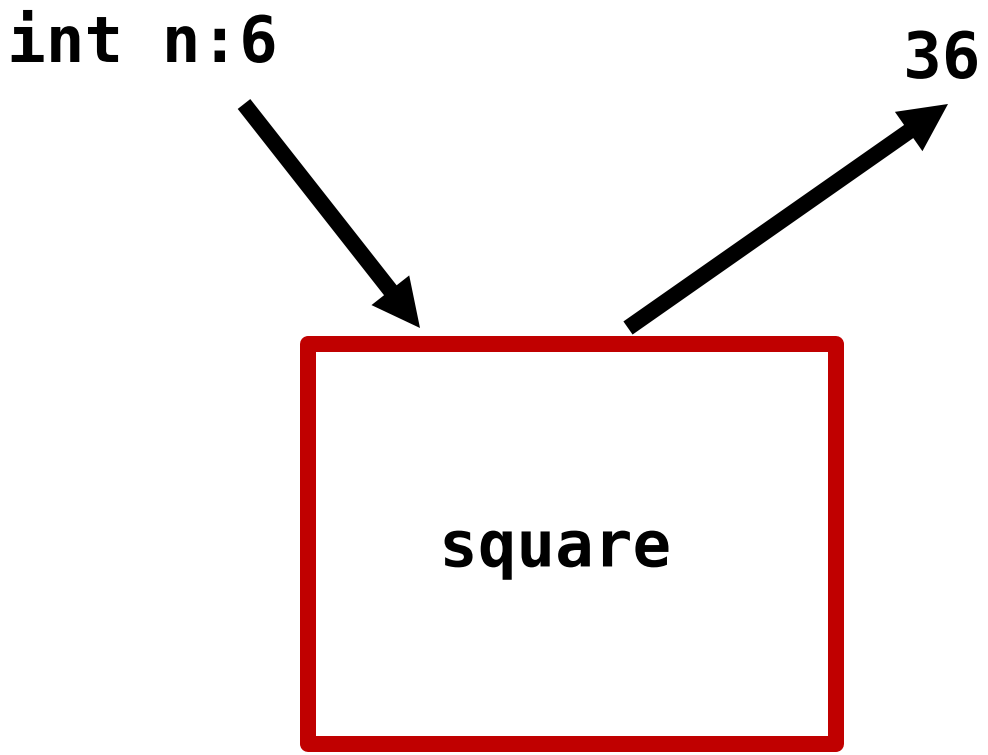
# Which of these is a valid return?

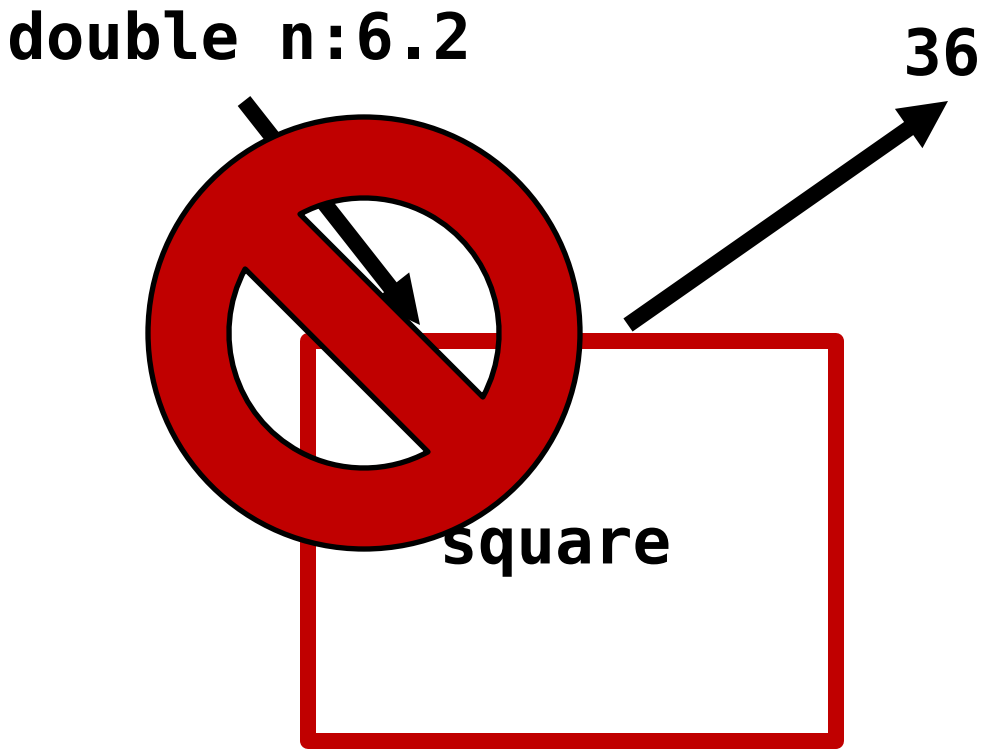
```
return (x);
return 5 + n;
return;
```

**They're all good!**
**There are 3 ways to return from a function:**

*** Return with a value**
*** Return with no value (just the "return" keyword)**
*** Implicit return (not using the return keyword; just finish execution).**







# [demo]

* Functions
* Function prototype
* Where to define functions
* Return types

**[demo]**

*** Write a function (makeCookies)**

*** Write the prototype**

*** Put the prototypes in a header file**

*** Compile with a header file**

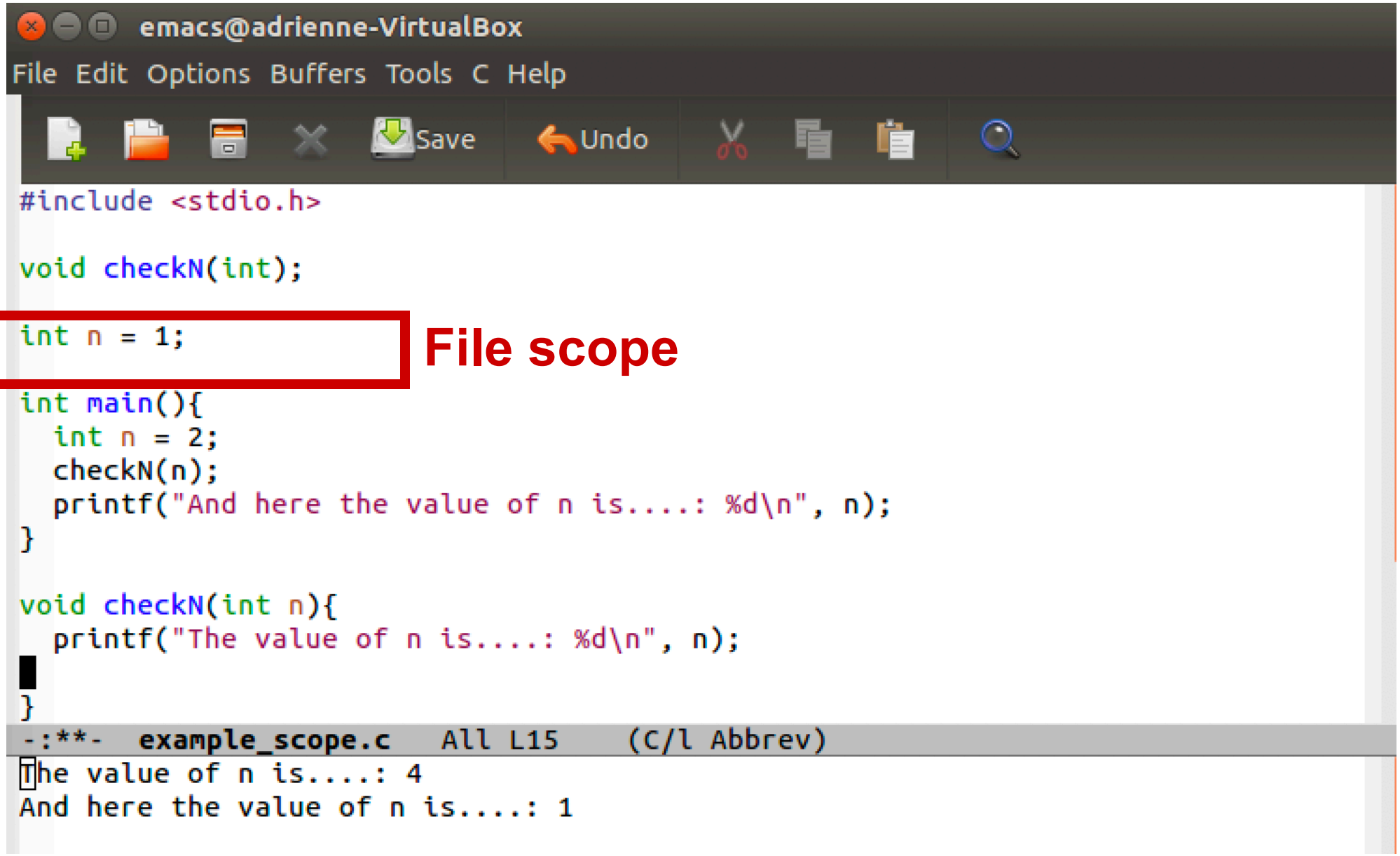
# Takeaways:

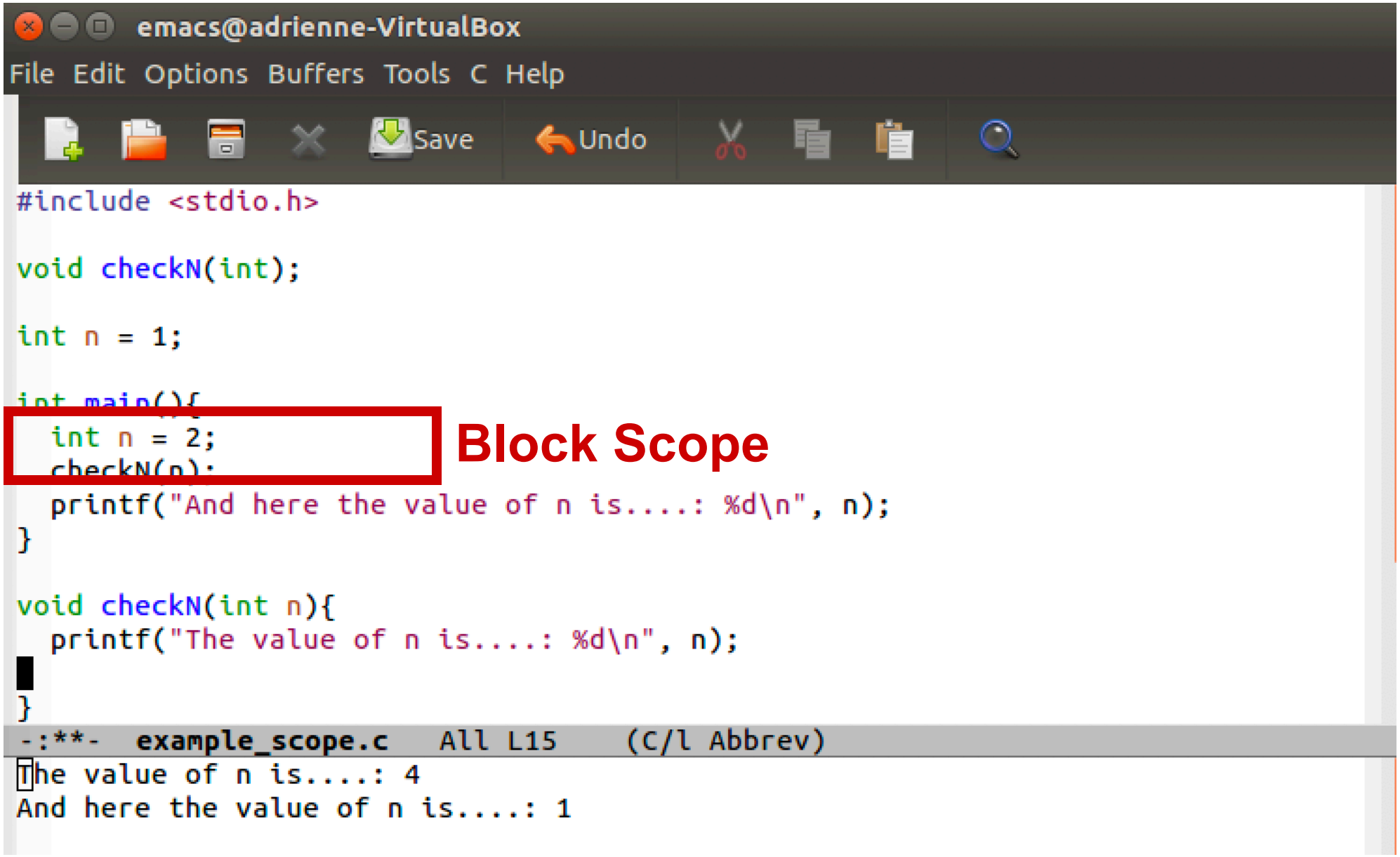
- C needs to know about functions you define before you use them.
- The **function prototype** tells C what to expect from your function.
- The **function definition** has a well-defined structure and implements the function.
- A **header file** contains function prototypes.
  - Named ".h"
- `#include "header_file.h"`
  - To include the header file.

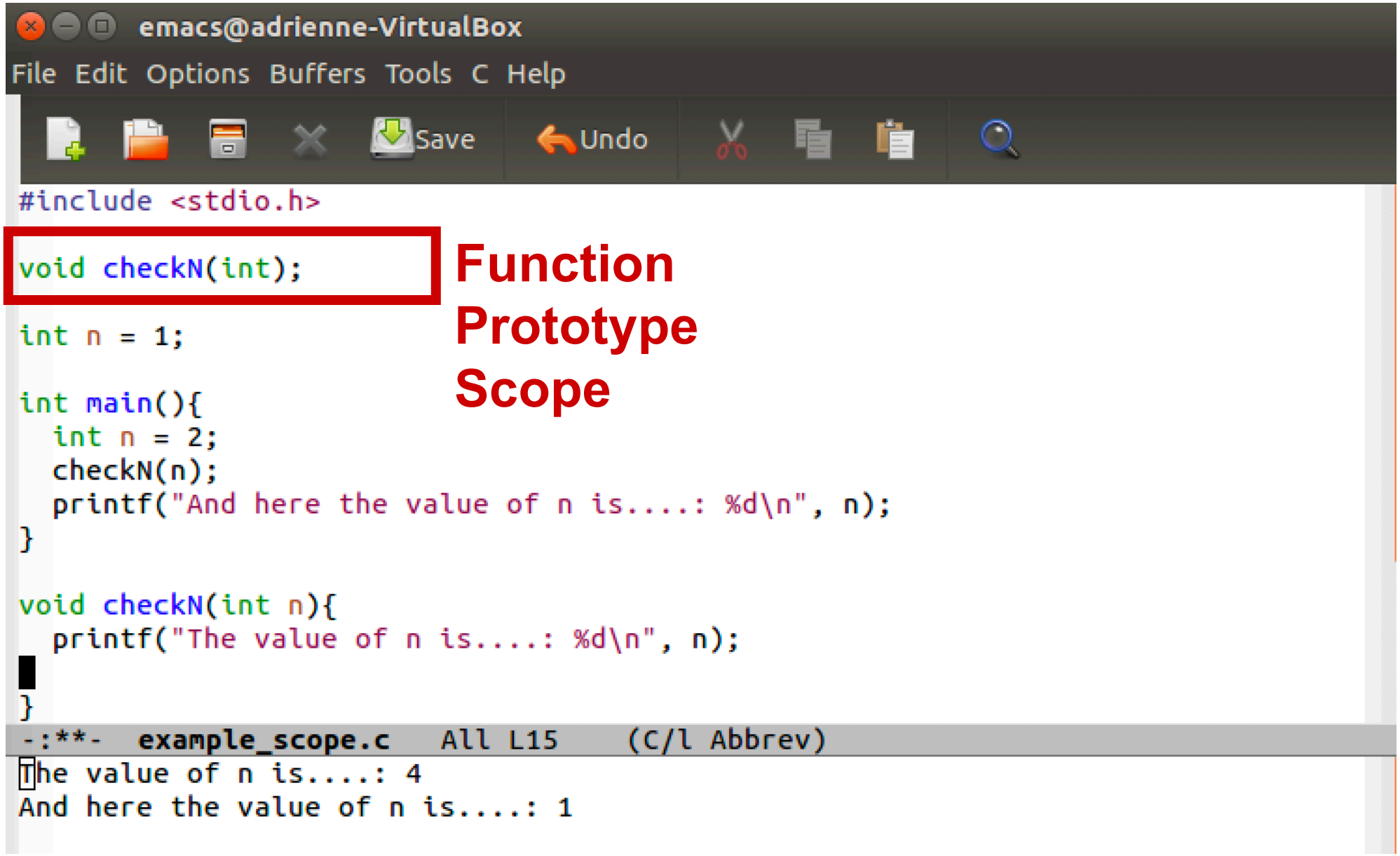
# [demo]
# * scope of a variable

# Takeaways

- There are 4 kinds of scope:
  - Function scope
  - File scope
  - Block scope
  - Function-prototype scope
- At this point, you'll care about **file** & **block** scope.


emacs@adrienne-VirtualBox
File Edit Options Buffers Tools C Help
Save
Undo
#include <stdio.h>
void checkN(int);
int n = 1;
File scope
int main(){
int n = 2;
checkN(n);
printf("And here the value of n is....: %d\n", n);
}
void checkN(int n){
printf("The value of n is....: %d\n", n);
}
-:**- example_scope.c All L15 (C/l Abbrev)
The value of n is....: 4
And here the value of n is....: 1


emacs@adrienne-VirtualBox
File Edit Options Buffers Tools C Help
Save
Undo
#include <stdio.h>
void checkN(int);
int n = 1;
int main(){
int n = 2;
checkN(n);
printf("And here the value of n is....: %d\n", n);
}
void checkN(int n){
printf("The value of n is....: %d\n", n);
}
-:**- example_scope.c All L15 (C/l Abbrev)
The value of n is....: 4
And here the value of n is....: 1
Block Scope


emacs@adrienne-VirtualBox
File Edit Options Buffers Tools C Help
Save
Undo
#include <stdio.h>
void checkN(int);
int n = 1;
int main(){
int n = 2;
checkN(n);
printf("And here the value of n is....: %d\n", n);
}
void checkN(int n){
printf("The value of n is....: %d\n", n);
}
-:**- example_scope.c All L15 (C/l Abbrev)
The value of n is....: 4
And here the value of n is....: 1
Function
Prototype
Scope

# REVIEW FROM LAST WEEK

**The elements of an array are related by the fact that they have the same _____ and _____.**

**The number used to refer to a particular element of an array is called its _____.**

**True or False:**
**An array can store many different kinds of values.**

**True or False:**

**An array index can be of data type `float`.**

**Naming an array, stating its type, and specifying the number of elements in the array is called _____ the array.**

**True or False:**

**An array declaration reserves space for the array.**

**True or False:**

**To indicate that 100 locations should be reserved for integer array p, the programmer writes the declaration:**

```
p[100];
```

# ARRAY DETAILS AND DEFINITIONS

# Variables

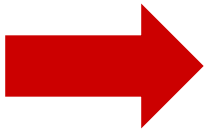
- name, type, value

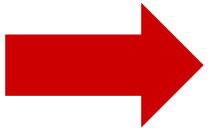
- int s = 1;

# Data Types

- char
- int
- float
- double
- void

# int types





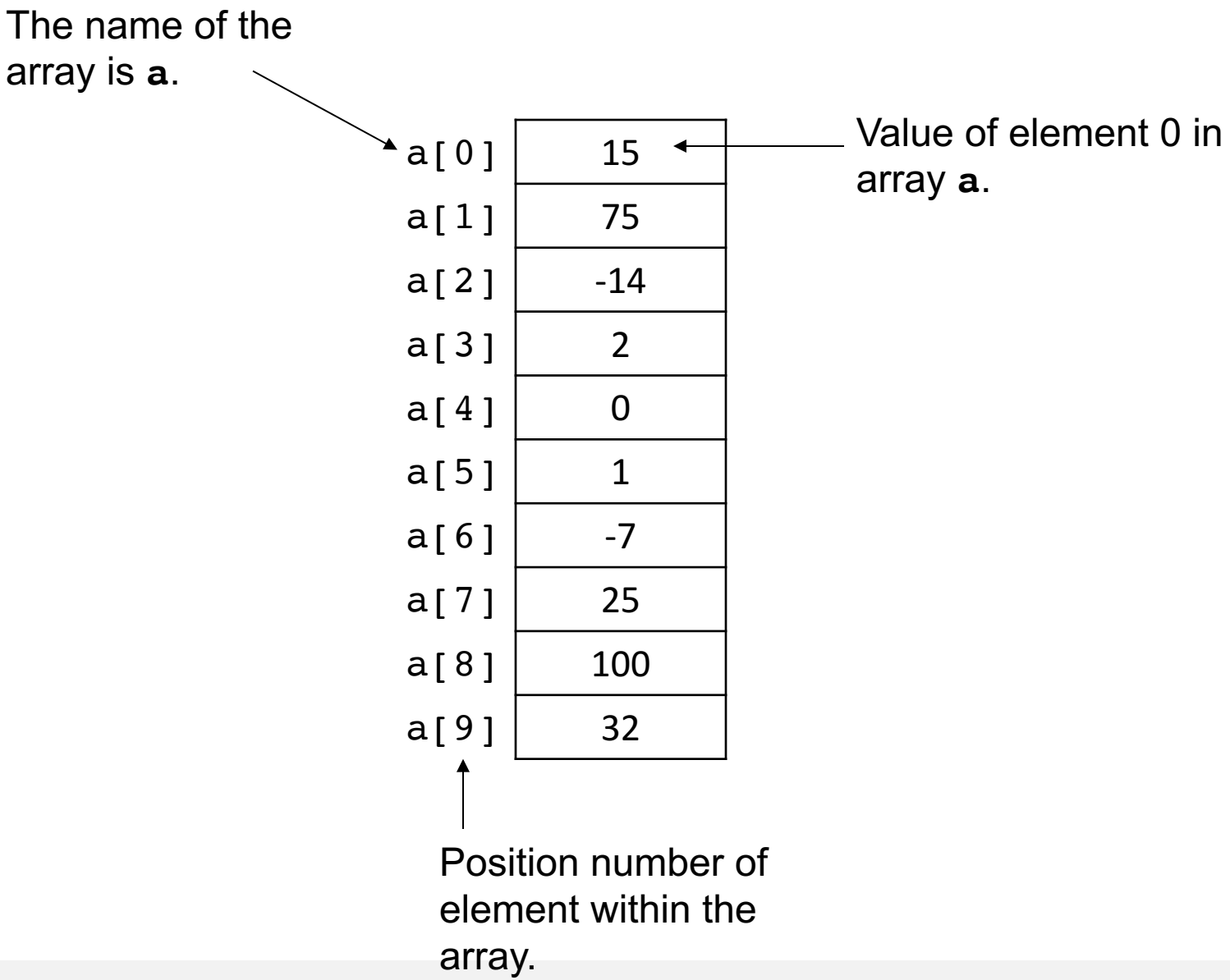
| Type | Storage size | Value range |
|---|---|---|
| char | 1 byte | -128 to 127 or 0 to 255 |
| unsigned char | 1 byte | 0 to 255 |
| signed char | 1 byte | -128 to 127 |
| int | 2 or 4 bytes | -32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| unsigned int | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,647 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

# Floating point numbers

| Type | Storage size | Value range | Precision |
|---|---|---|---|
| float | 4 byte | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double | 8 byte | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 byte | 3.4E-4932 to 1.1E+4932 | 19 decimal places |

# Arrays

- Group of memory locations related by the fact that they have the same name and same type.
- Index
  - 0-based
    - First element is the 0-th element
  - integer; can be an expression
  - might also be referred to as *subscript*
- Elements



# Declaring an array

- Arrays occupy space in memory, so we have to declare the:
  - name
  - type
  - how many (space)

- `int c[13];`
- `double radii[165];`

# Check your knowledge: Which one is an array?

```
1.int num_loops;
2.double tricia[21];
3.char initial;
4.int n;
```

# Initializing

- Just like with variables, we need to initialize the values.
  - Use a loop:
    - ```
      int n[3];
      for (int i=0; i<3; i++){
          n[i] = 0;
      }
      ```
  - Use a list of values:
    - ```
      int n[3] = {52, 13, -5};
      ```
  - Use a single value:
    - ```
      int n[3] = {0};
      ```

The array named `n` has all the same values, 0.

The array named `n` has all different values.

The array named `n` has all the same values, 0.
      ```

# <demo/coding>

# Problem: Create a histogram of values.

- Assume we took a poll from you guys: how much time did you spend on Assignment 2?
  - Some assumptions:
    - Max number of hours is 10
    - 23 responses total.
    - Everyone answered with a whole number

Take 2 minutes and think about how you'd approach this.

Turn to the person next to you and talk about it.

- Goal: Create a histogram of the number of each response.
  - The histogram array will contain the count of each unique value.
- Hint: we can solve this with arrays!

| | |
|---|---|
| response[0] | 2 |
| response[1] | 1 |
| response[2] | 0 |
| response[3] | 5 |
| response[4] | 0 |
| response[5] | 1 |
| response[6] | 10 |
| response[7] | 9 |
| response[8] | 8 |
| response[9] | 3 |

| | |
|---|---|
| count[0] | 2 |
| count[1] | 2 |
| count[2] | 1 |
| count[3] | 1 |
| count[4] | 0 |
| count[5] | 1 |
| count[6] | 0 |
| count[7] | 0 |
| count[8] | 1 |
| count[9] | 1 |
| count[10] | 1 |



Note: the sum of the values in this array is equal to the number of entries in the response array.

# <demo/coding>

# Example 2: Print a histogram

- Using the histogram we've already computed, print out the values like this:
- 
```
Element    Value    Histo
0          3        ***
1          5        *****
2          1        *
3          2        **
4          8        *********
```

# Passing arrays to functions

- To pass an array to a function, specify the name of the array without any brackets.
- ```
  int temps[24];
  …
  printTempVals(temps, 24);
  ```
- We usually pass the size of the array along with the array.
  ```

# Writing functions with arrays as params

```
void printArray(int *a, int len);
void printArray(int a[], int len);
```

# Example: PrintArray

- `void printArray(int *a, int len);`

Work on this.

Work with your neighbor.

# Multi-Dimensional Arrays

- We can have multiple subscripts/dimensions
  - Usually used to represent tables of data
  - Need to specify both row and column
  - You can use more than 2, but we're just going to work with 2 today
  - When passing to a function, *have* to specify all dimensions past the first.
- Example:

```
int b[2][2] = {{1, 2}, {3, 4}};
```

# Example: PrintDimDiags

```
printDiags(int*[3] box){


}
```

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

Work on this.

Work with your neighbor.

# Chars & Strings

- At this point, you have a pretty good understanding of what chars and strings are.
- Now, we're going to talk about things you can do with chars and strings, using various libraries

# Overview

- Character handling/manipulation
- String conversion
- String input/output
- String handling/manipulation
- String comparison
- String search functions
- Memory/string functions

# Difference between chars and strings

- ‘a’, ‘b’
  - Chars have single quotes
  - Represented by an **`int`** value
  - Includes letters and special chars
    - Special chars: things like ‘\n’ (newline), ‘\0’ (null), ’\t’ (tab)
- “a”, “b”
  - Strings have double quotes
  - Represented by an array of chars ending with a NULL character (‘\0’)

# Strings

- Strings are arrays of chars terminated by a \0.
- When you initialize a string, you get back a pointer to a char array
- Unlike regular arrays, you can use the existence of the \0 to find the end of the array
- An array of strings turns out to be an array of pointers to arrays of chars.
- Because an array is just a pointer to the first element of the array, an array of strings is a pointer to a pointer.

# Declaring Strings

```
// creates an array of chars ['b', 'l', 'u', 'e', '\0']
char color[] = "blue";
char *colorPtr = "blue";  // creates a variable of type char pointer
char color[] = {'b', 'l', 'u', 'e', '\0'}
```

**Note:**
Declaring a string this way can make it read-only. That means if you try to modify it later, you can't do it by modifying the characters in the underlying array.
(see the demo code)

**Declaring an array to eventually hold a string:**
** Make sure it's long enough to contain all chars AND the null character!!**

# Character functions <ctype.h>

- `isdigit(int c)`: Returns ***a true value*** if `c` is a digit, and 0 (false) otherwise.
- `isalpha(int c)`: true if `c` is a letter; 0 otherwise.
- `isalnum(int c)`: true if `c` is a digit or letter (alphanumeric); 0 otherwise.
- `isxdigit(int c)` : true if `c` is a hexadecimal digit; 0 otherwise.
- `islower(int c)` : true if `c` is an lower case letter; 0 otherwise.
- `isupper(int c)` : true if `c` is an upper case letter; 0 otherwise.

# Character functions <ctype.h>

- `tolower(int c)` : Returns c as a lower case letter.
- `toupper(int c)` : Returns c as an upper case letter.
- `isspace(int c)` : true if c is an upper case letter; 0 otherwise.
- `iscntrl(int c)` : true if c is a control character; 0 otherwise.
- `ispunct(int c)` : true if c is a printing character other than space, digit or letter; 0 otherwise.
- `isprint(int c)` : true if c is a printing character including space; 0 otherwise.
- `isgraph(int c)` : true if c is a printing character other than space; 0 otherwise.

# Character functions, list **<ctype.h>**

- isdigit(int)
- isalpha(int)
- isalnum(int)
- isxdigit(int)
- islower(int)
- isupper(int)
- tolower(int)
- toupper(int)
- isspace(int)
- iscntrl(int)
- ispunct(int)
- isprint(int)
- isgraph(int)

# String Conversion Functions

- Purpose: convert strings from chars to different data types (e.g., numbers– ints, double, etc.)
- As long as the input string can be converted to the specified type, that number will be returned.
  - If the input string isn't a number, the behavior is "undefined"
- The "a-to-XX" functions convert the given string to a number:
  - "3.14159" → `3.14159`
- The "str-to-XX" functions convert the start of a string to a number:
  - "51.2% of the people…" → `51.2`

# String Conversion Functions <stdlib.h>

- `double atof(const char *nPtr)`
- `int atoi(const char *nPtr)`
- `long atol(const char *nPtr)`
- `double strtod(const char *nPtr, char **endPtr)`
- `long strtol(const char *nPtr, char **endPtr, int base)`
- `unsigned long strtoul(const char *nPtr, char **endPtr, int base)`

The use of '`const`' indicates that the argument value will not be changed.

# Standard Input/Output Library Functions **<stdio.h>**

- Functions to get strings/chars from the ***standard input*** (e.g., the keyboard)
- Functions to put/write strings/chars to the ***standard output*** (e.g., the terminal/screen)

# <stdio.h>

- `int getchar(void)`
- `char *gets(char *s)`
- `int putchar(int c)`
- `int puts(const char *s)`
- `int sprintf(char *s, const char *format, …)`
- `int sscanf(char *s, const char *format, …)`

# String Manipulation

- Functions to manipulate strings
- This is either copying strings or concatenating strings
- strcpy copies a string into a specified array
- strcat appends one string onto another string, overwriting the \0 that terminates the first string.
- NOTE: These functions usually copy into an existing array. This can cause problems if the original array isn't long enough!

# String manipulation functions <string.h>

The type **size_t** is system dependent, and is either an **unsigned long** or **unsigned int**.

- `char *strcpy(char *s1, const char *s2):` Copy s2 into the array of s1, returns the value of s1.
- `char *strncpy(char *s1, const char *s2, size_t n):` Copy at most n characters of s2 into s1; returns the value of s1.
- `char *strcat(char *s1, const char *s2):` Append string s2 to the array s1 (overwriting the original \0 of s1); return the val of s1.
- `char *strncat(char *s1, const char *s2, size_t n):` Append at most n characters of s2 to s1; return the val of s1.

# String Comparison

- Given two strings, returns:
  - 0 if the strings are equal
  - a negative value if the first string is “less than” the second string
    - s1 = “apple”, s2 = “banana”
    - 1: apple, 2: banana
  - a positive value if the first string is “greater than” the second string
    - s1 = “banana”, s2 = “apple”
    - 1: apple, 2: banana
- 2 versions: one that compares entire strings, one that compares the first n characters of the strings.

# String Comparison **<string.h>**

- int strcmp(const char *s1, const char *s2)
- int strncmp(const char *s1, const char *s2, size_t n)

# String Search functions

- Goal:
  - Find a string within another string (“cat” in “caterpillar”)
  - Find substrings that do/do not contain specific characters
  - Breaking a string into tokens (“This is a sentence” → [“This”, “is”, “a”, ”sentence”]

# String search functions <string.h>

- `char *strchr(const char *s, int c)`: Finds first occurrence of a char c in s, returns a pointer to c in s. If not found, returns NULL.
- `size_t strcspn(const char *s1, const char *s2)`:Determines and returns the length of the initial segment of string s1 consisting of characters not contained in s2.
- `size_t strspn(const char *s1, const char *s2)` :Determines and returns the length of the initial segment of string s1 consisting only of characters contained in s2.
- `s1 = “Pi value is 3.14…”, s2 = “1234567890”`

# String search functions <string.h>

- `char *strpbrk(const char *s1, const char *s2)` Locates first occurrence of any character from s2 in s1.
- `char *strrchr(const char *s, int c):` Locates last occurrence of c in string s.
- `char *strstr(const char *s1, const char *s2):` Locates first occurrence of s2 in s1.
- `char *strtok(char *s1, const char *s2):` Break a string s1 into a series of 'tokens'

# String search functions, list <string.h>

- `char *strchr(const char *s, int c)`
- `size_t strcspn(const char *s1, const char *s2)`
- `size_t strspn(const char *s1, const char *s2)`
- `char *strpbrk(const char *s1, const char *s2)`
- `char *strrchr(const char *s, int c)`
- `char *strstr(const char *s1, const char *s2)`
- `char *strtok(char *s1, const char *s2)`

# Memory/string functions

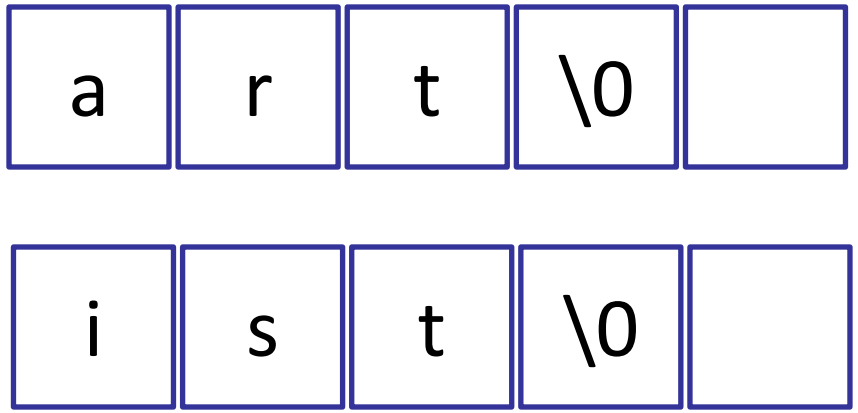
- Functions in the string handling library (string.h)
- They do memory manipulation, comparing and searching, but treats blocks of memory as character arrays.

# Primary takeaway about Chars and Strings…

- Chars and strings can be manipulated
- In C, the important aspects of chars are:
  - Checking to see if a char is upper/lower/digit/printable/special/etc.
  - Manipulating a char to upper/lower
- The important aspects of strings are:
  - copying
  - concatenating (combining 2 strings together)
  - determining substrings
  - reading in/out from std
  - converting from string to a number/type

# Primary takeaway about Chars and Strings…, cont.

- Important libraries:
  - string.h
  - ctype.h
  - stdlib.h
  - stdio.h



# [in-class spur-of-the-moment pointer discussion]

- Variable: type, name, value
- int bar = 0;
- int *barPtr = &bar;

- char *foo = “blue”;
- char *foo = {‘b’, ‘l’,’u’,’e’,’\0’};

- char **str = &foo;

- char * foo = {‘b’, ‘l’,’u’,’e’,’\0’};
- char ** str = &foo;

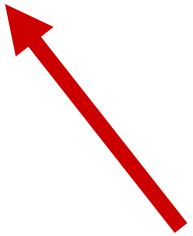
- Pointer == to char (array) == string
- String:: array of chars

# INTRODUCING DERIVED DATA TYPES

# Structs

- ***derived*** data type
- Collections of related variables
- Doesn't have to include only the same type of variables

```
struct card{
   char *face;
   char *suit;
};
```



**members:**
either basic types, or aggregates (arrays, other structs)

# Valid Operations

- Assignment structure variables to structure variables of the same type
  - struct card foo1;
  - struct card foo2 = foo1;
- Getting the address of a structure variable
  - struct card *cPtr = &foo1;
- Accessing the members of the structure variables
  - foo1.face
- Using sizeof to determine size of a structure variable
  - sizeof(foo1);

# Initializing structs

```
struct card q_spade = {"Queen", "Spades"};
```

# Accessing struct members

- dot operator (‘.’)
  - accesses a member of the structure by name
  - `struct card a = {“Queen”, “Spades”};`
  - `printf(“%s”, a.suit);`
- arrow operator (->)
  - accesses a member of the structure from a pointer to the struct
  - struct card *aPtr = &a;
  - `printf(“%s”, a->suit);`

# typedef

- aliases for previously defined data types
- Example:
  - `typedef struct card Card;`
- A shortcut:
  - ```
    typedef struct{
        char *face;
        char *suit;
    } Card;
    ```
- Why? because it's easier to type "Card" than "struct card"
  - Doesn't create a new type; just a new name.
    ```

# Enumerations

- User-defined type
- keyword: **`enum`**
- A set of integer constants that represent identifiers.
- Example:
  - `enum suits {SPADES=1, HEARTS, CLUBS, DIAMONDS}`
  - Automatically, the values are 0-3. SPADES = 0; CLUBS = 2.
  - Can specify starting #

# MISCELLANEOUS

# Pattern 1: Declare variable, function to initialize

# Pattern 2:

- Write a function to declare and initialize a variable,
- Return a pointer to it
- Free the pointer later

# Memory: Storing Variables

- **Global:** available to the whole program
- **Local (stack):** available "in scope"– in a function
- **Heap:** available to the program

# Malloc:

- Allocates memory “on the heap”.
- Returns a value of type void*
- Cast it into the pointer type you care about
- Don’t forget to free() the pointer when you’re done with it

# Review

- Memory
  - addresses
  - stack
- Variables
  - Name
  - Type
  - Value
  - Address
- Arrays
  - Name
  - Type
  - Size
- Declaring
- Initializing
- Multidimensional arrays
- Functions with arrays
- lvalues, rvalues
- byte-addressable structures

# A FEW NOTES ON EMACS

# A new chapter/lecture

- `cd projects`
- `mkdir lect2`
- `cd lect2`
- `emacs p1.c &`

# Starting a new problem/flashcard

- `Ctrl-x Ctrl-f` (new file)
- `p3.c` (filename)

- `Ctrl-x b` (*change buffer*)
- `Ctrl-k` (*a few times, to get the first few lines*)
- `Ctrl-x b` (*go back to my new buffer*)
- `Ctrl-y` (*yank the info into the new buffer*)

# Compiling and Running your code

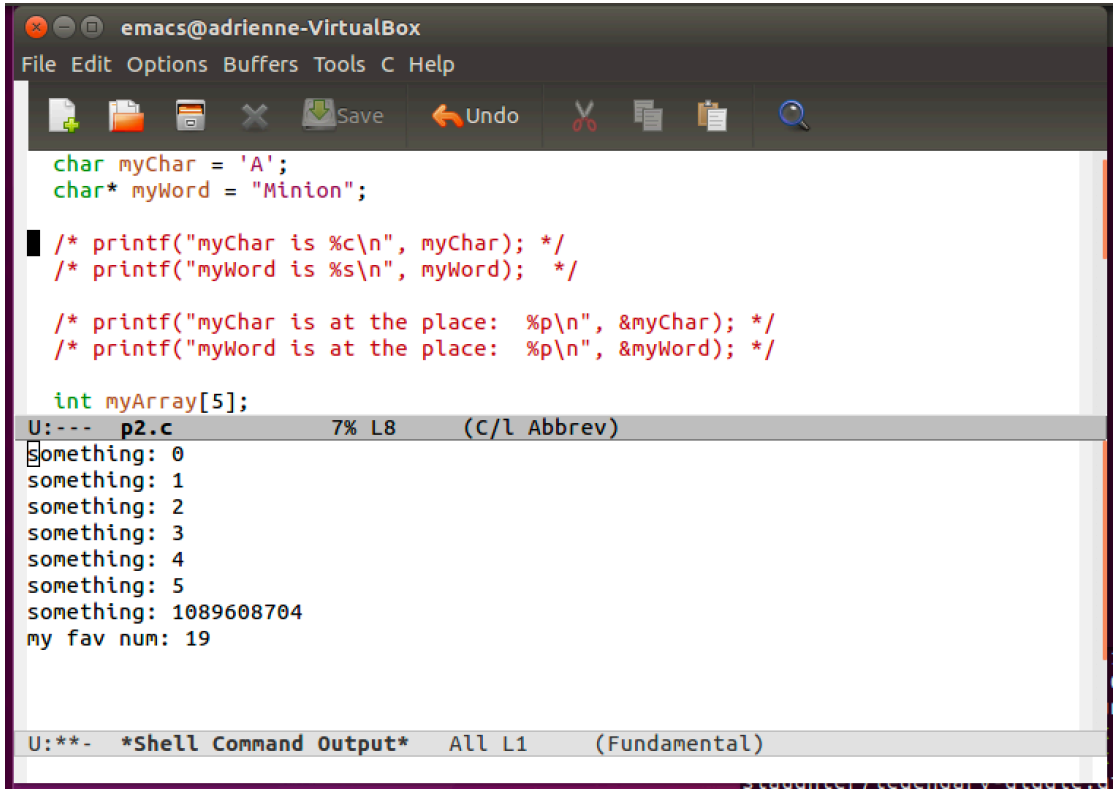
- `M-x shell-command <enter>`
- `gcc p2.c <enter>`
- `M-x shell-command <enter>`
- `./a.out`
- `M-! (M + Shift + 1) <enter>`
- `./a.out`

# Making your code look good: Indenting a region

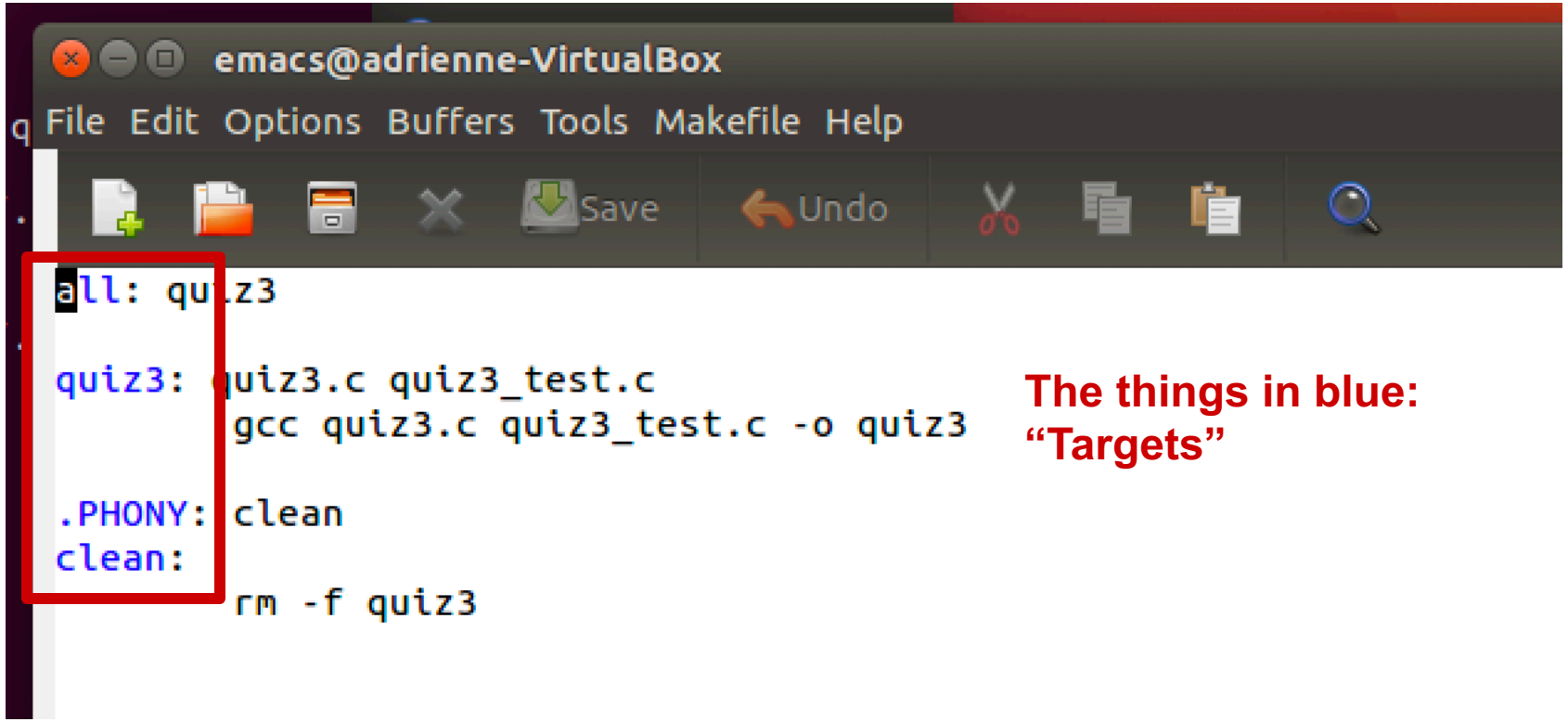
- `Ctrl-<space>`
- `Ctrl-P/Ctrl-N, etc (selecting your region)`
- `M-x indent-region`

# Windows

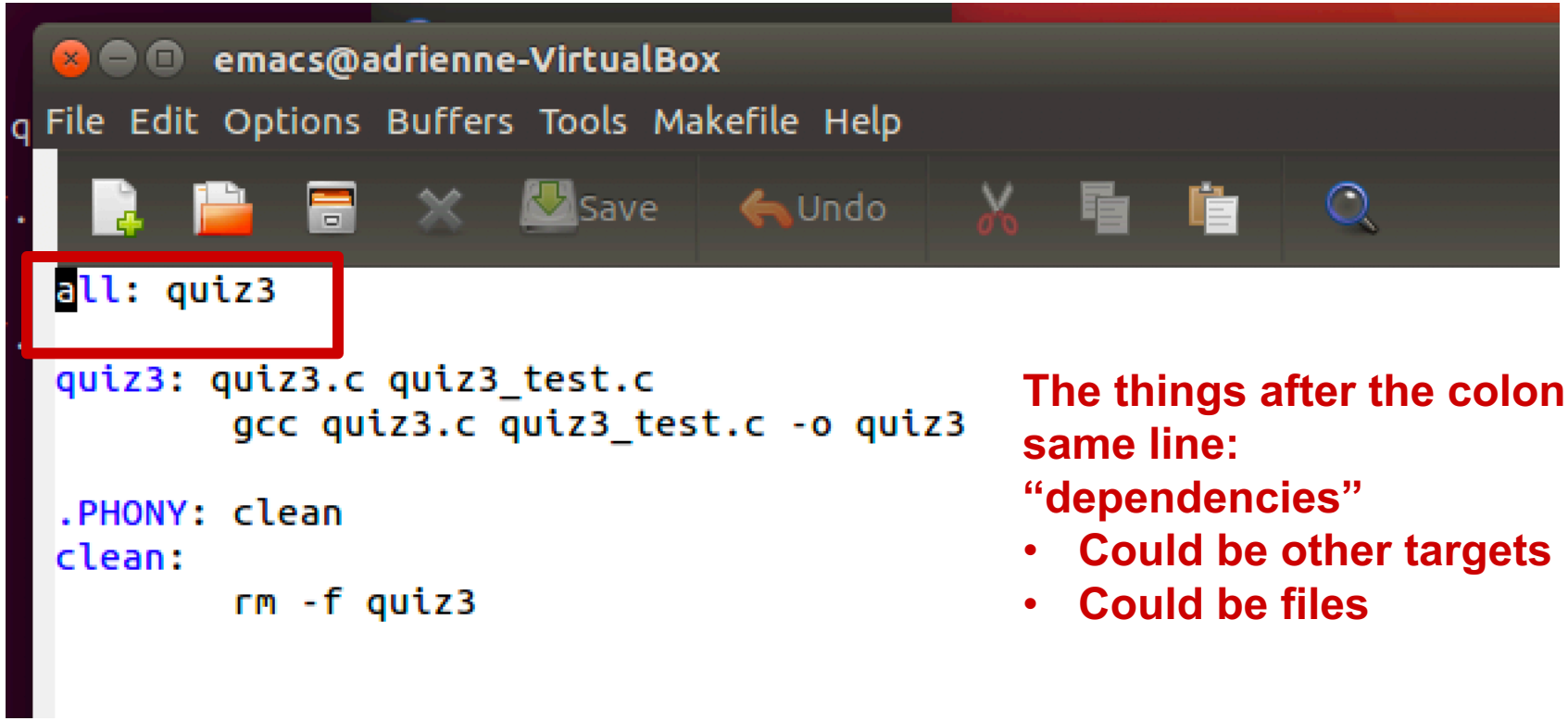
- Sometimes you get something like this. It's 2 buffers in 2 windows.
- To get rid of the "other" window:
- **C-x 1**



# MAKEFILE

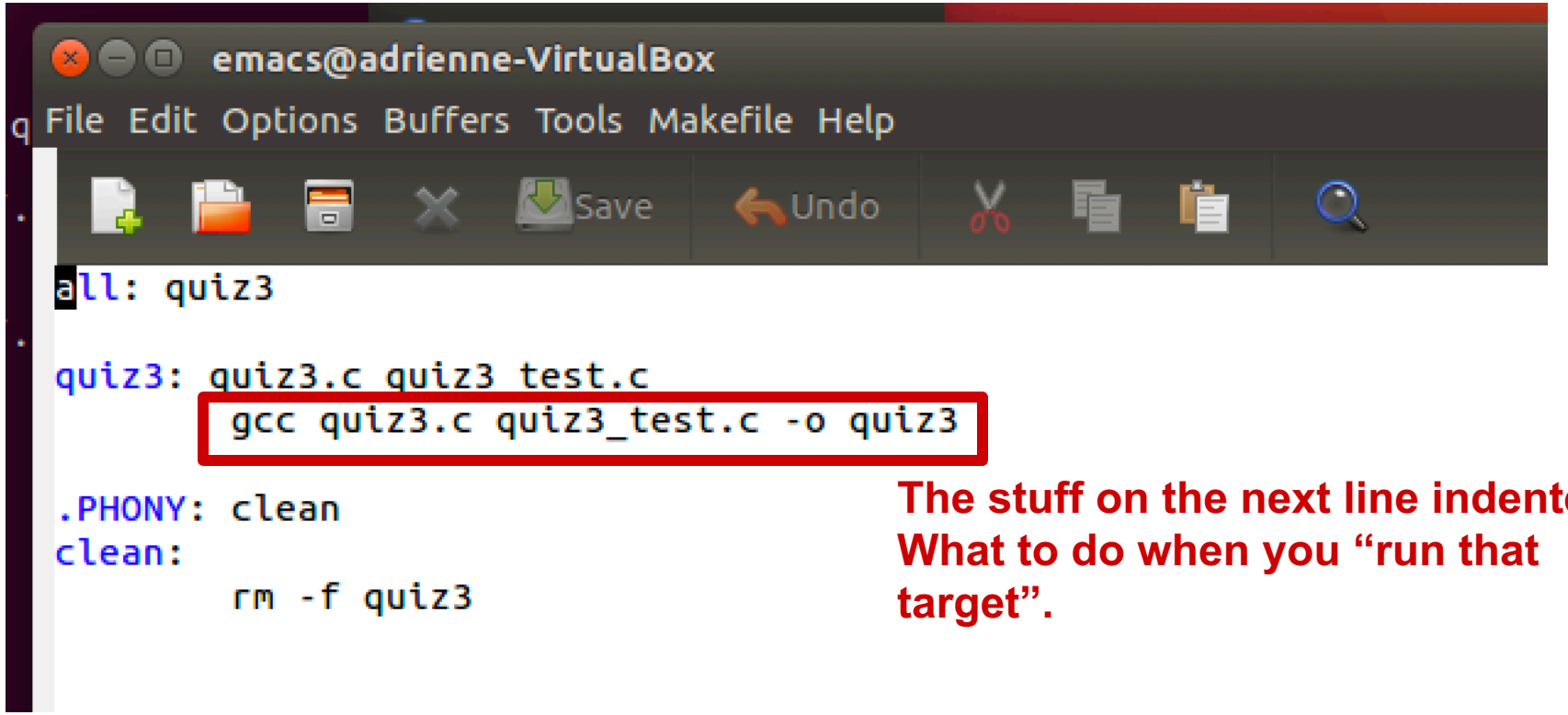



**The things in blue: "Targets"**

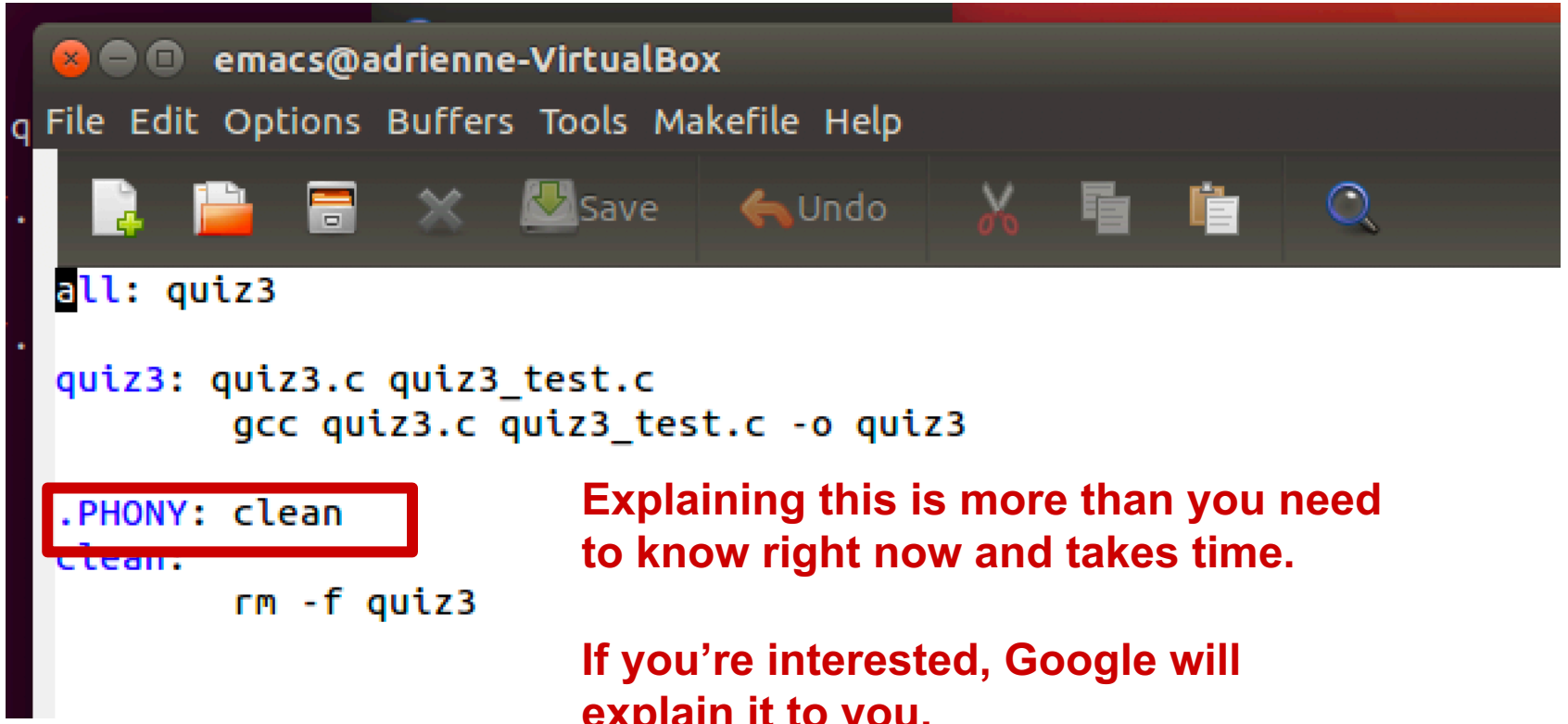



**The things after the colon on the same line:**
**"dependencies"**

- **Could be other targets**
- **Could be files**

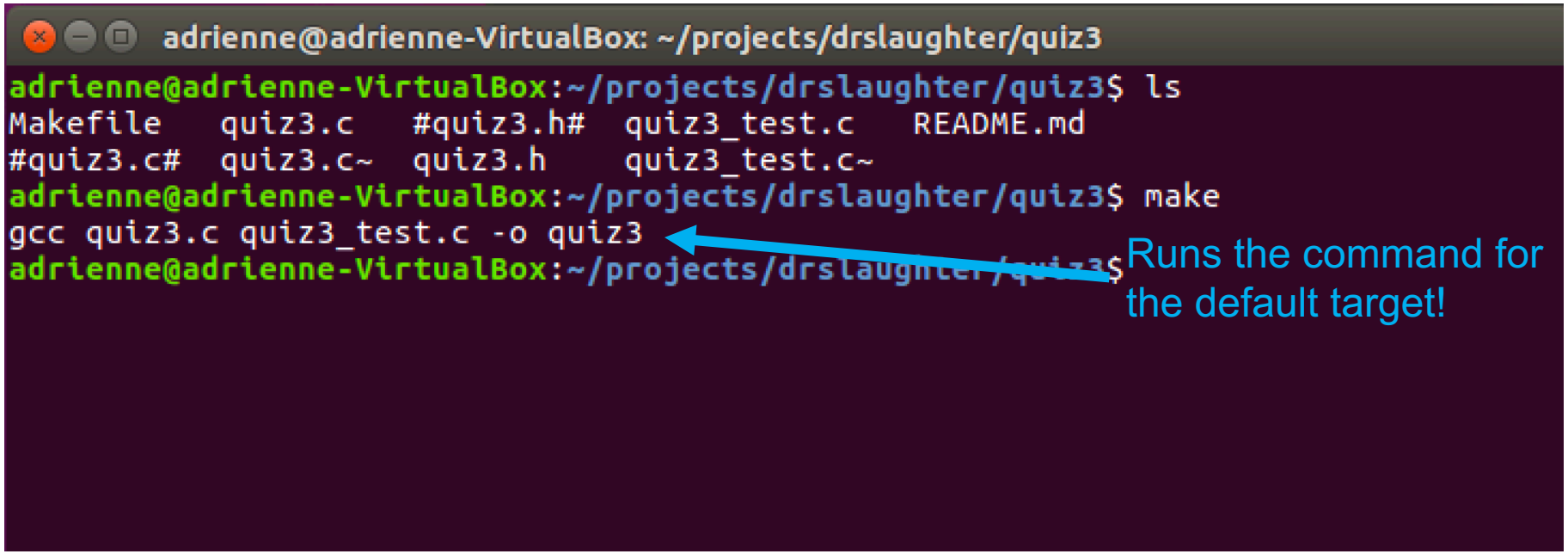


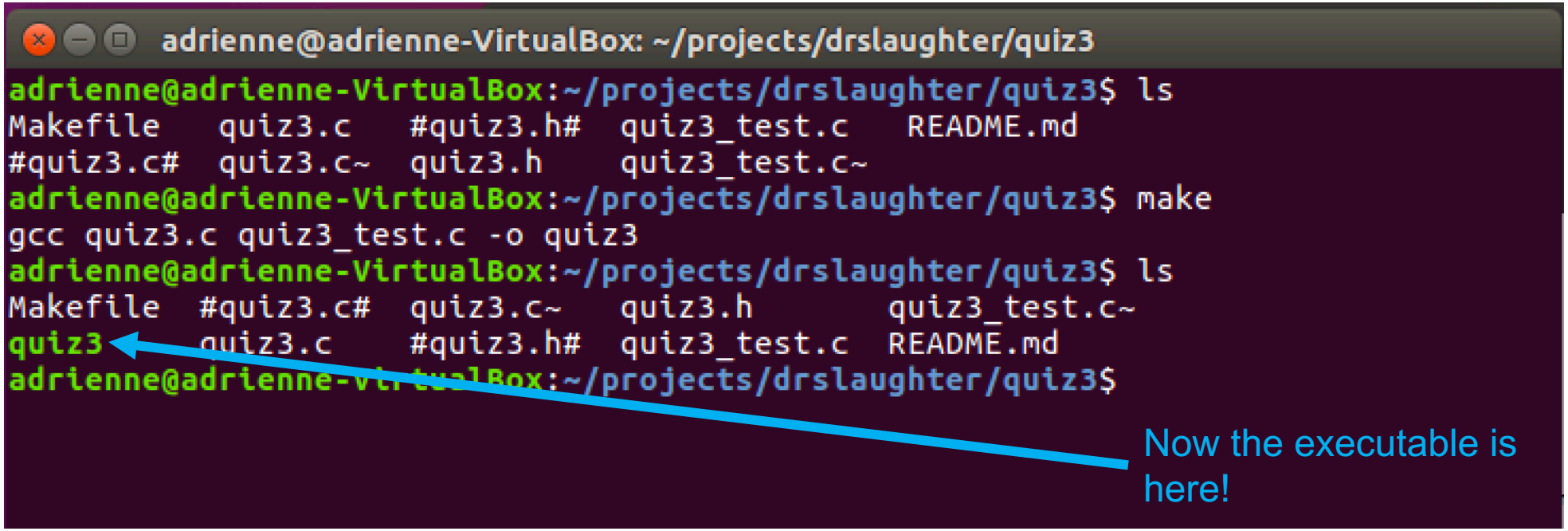

**The stuff on the next line indented: What to do when you "run that target".**

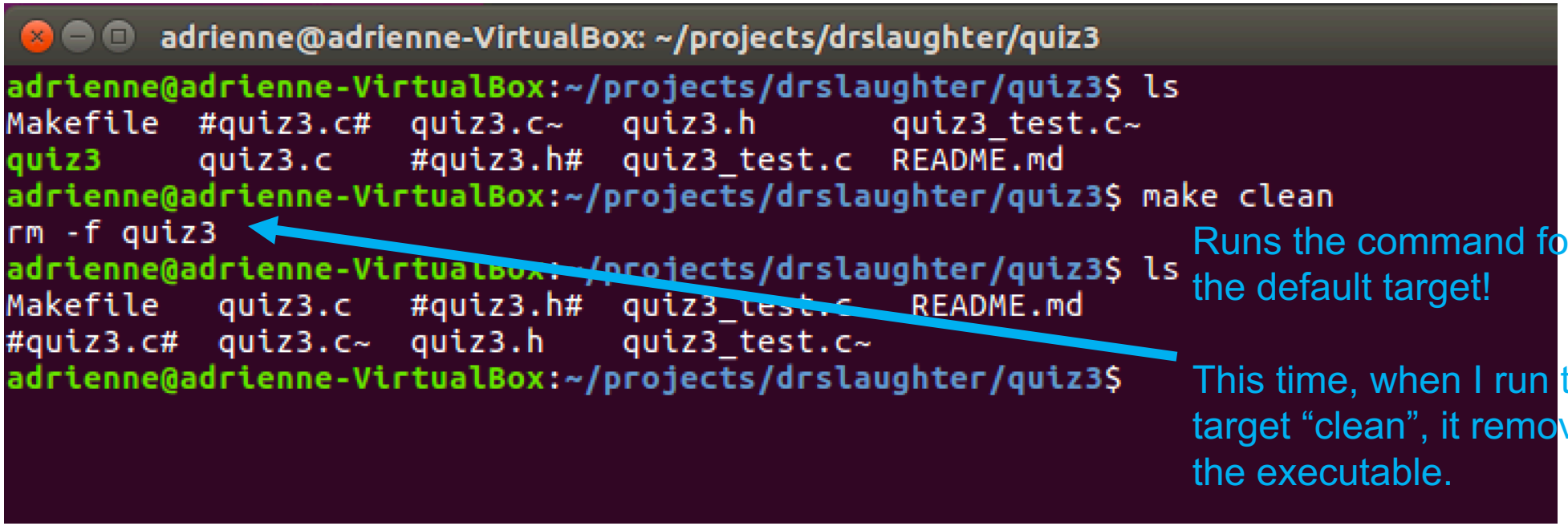



**Explaining this is more than you need to know right now and takes time.**

**If you're interested, Google will explain it to you.**


adrienne@adrienne-VirtualBox: ~/projects/drslaughter/quiz3
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ ls
Makefile quiz3.c #quiz3.h# quiz3_test.c README.md
#quiz3.c# quiz3.c~ quiz3.h quiz3_test.c~
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ make
gcc quiz3.c quiz3_test.c -o quiz3
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$
Runs the command for the default target!


adrienne@adrienne-VirtualBox: ~/projects/drslaughter/quiz3
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ ls
Makefile quiz3.c #quiz3.h# quiz3_test.c README.md
#quiz3.c# quiz3.c~ quiz3.h quiz3_test.c~
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ make
gcc quiz3.c quiz3_test.c -o quiz3
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ ls
Makefile #quiz3.c# quiz3.c~ quiz3.h quiz3_test.c~
quiz3 quiz3.c #quiz3.h# quiz3_test.c README.md
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$
Now the executable is here!


adrienne@adrienne-VirtualBox: ~/projects/drslaughter/quiz3
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ ls
Makefile #quiz3.c# quiz3.c~ quiz3.h quiz3_test.c~
quiz3 quiz3.c #quiz3.h# quiz3_test.c README.md
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ make clean
rm -f quiz3
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$ ls
Makefile quiz3.c #quiz3.h# README.md
#quiz3.c# quiz3.c~ quiz3.h quiz3_test.c~
adrienne@adrienne-VirtualBox:~/projects/drslaughter/quiz3$
Runs the command for the default target!
This time, when I run the target “clean”, it removes the executable.

# Why make?

- Short answer: because it will eventually make your life easier.
- Long answer: it will *so totally* make your life easier in the future.
- More seriously: We'll revisit this at the end of the semester.