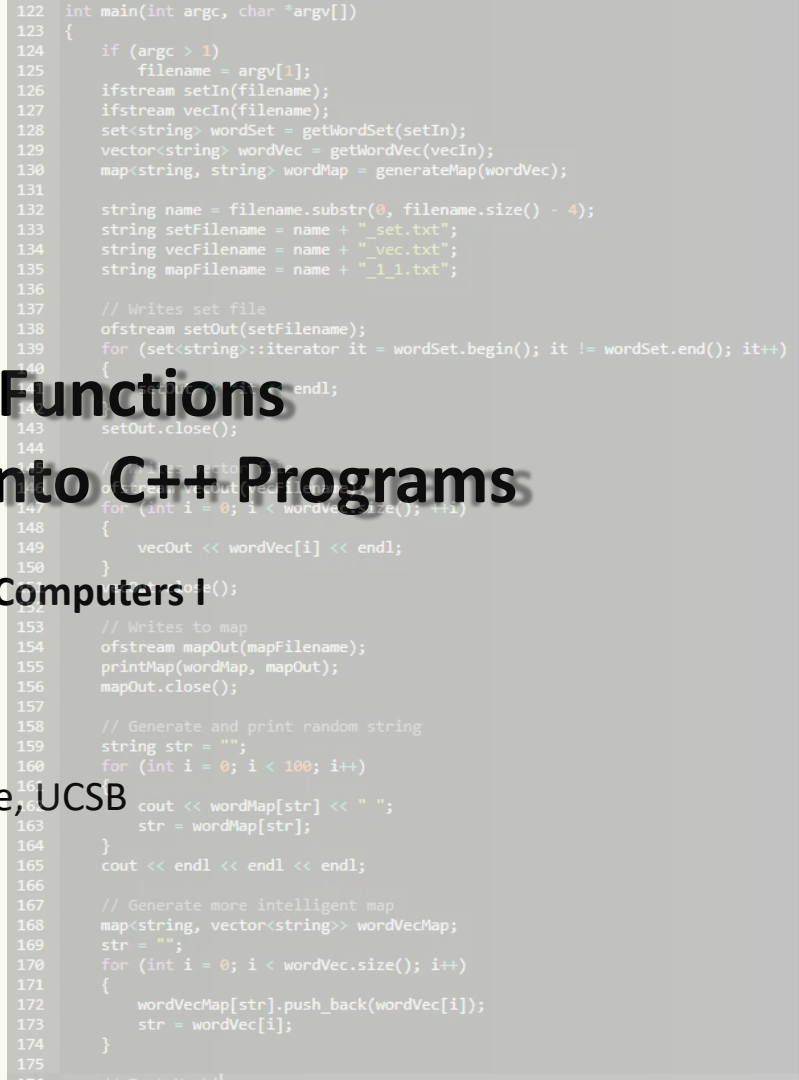
Exercises with Basic Functions
Command-Line Arguments into C++ Programs

CS 16: Solving Problems with Computers I
Lecture #6

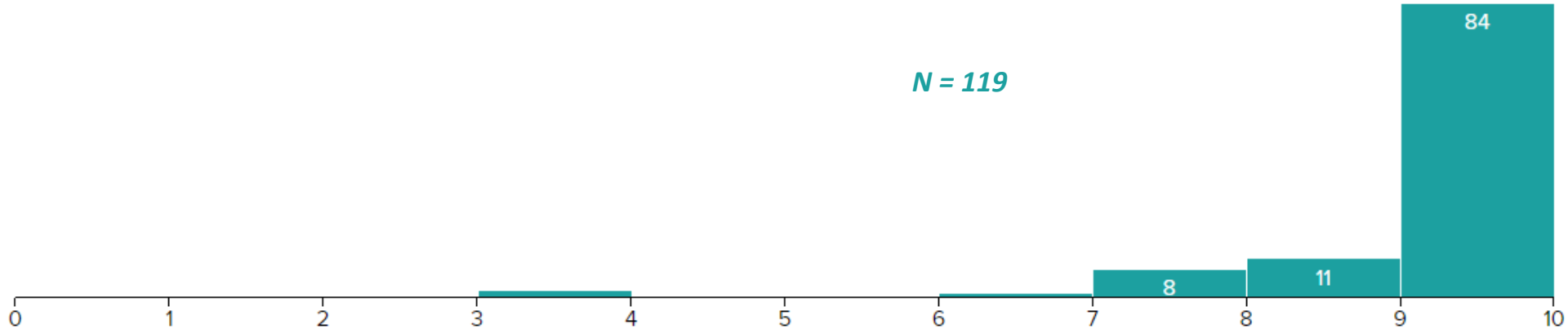
Ziad Matni
Dept. of Computer Science, UCSB



Administrative

- New lab (#3) and new homework (#3) are out!
- Homework 2 and Lab 2 were due yesterday
- Quiz 2

Quiz 2



- Mean: **9.19/10**
- Median: **10/10**
- Great job! ☺

Lecture Outline

- Practice with Basic Programming
- Command-line Arguments
- Intro to “Calling by Reference”

Assignments

- Do not use techniques I have not covered in class
- Do not copy answers from other people
- Do not copy answers from websites

- While the textbook is useful as additional material, **prioritize your attention to what I am covering in lectures**
  - This means, keep up with the lectures!

Quiz Question Review

- Which of these statements is **TRUE** about when I first declare a local int variable in C++ in this manner: `int varX;`

  – The variable **varX** will have a value of 0.

  – The variable **varX** will have a value of 1.

  – The variable **varX** will have an unspecified integer value.

  – The variable **varX** will not have any value at all.

  – None of the above are true statements.

Remember:
When you DECLARE a variable in C++, it is assigned a place in computer memory.

This location may already have a value in it - it is NOT wiped clean!

Reminder

- This is ok in C++:

```
var <= 42;
```

- This is NOT ok in C++:

```
42 >= var;
```

ANY QUESTIONS FROM THE LAST (pre-recorded) LECTURE?

Example of a Simple Function in C++

```
#include <iostream>
using namespace std;

int sum2nums(int num1, int num2); // returns the sum of 2 numbers

int main ( )
{
    int a(3), b(5);
    int sum = sum2nums(a, b);
    cout << sum << endl;
    return 0;
}

int sum2nums(int num1, int num2)
{
    return (num1 + num2);
}
```

Declaration

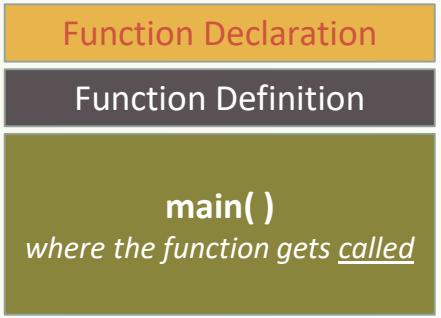
Call

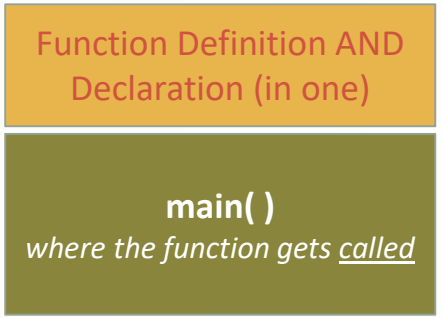
Definition

Block Placements for Functions

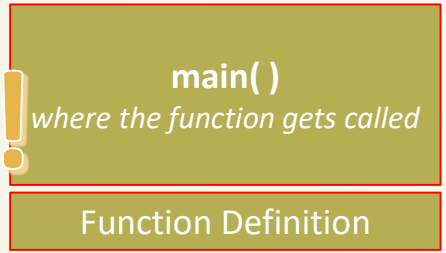
OK!

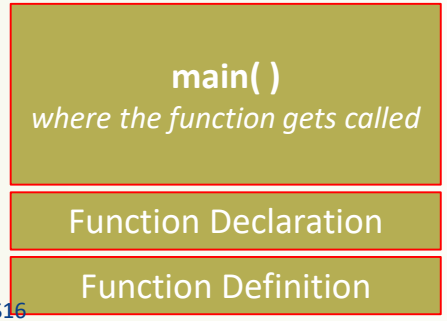
Most widely-used scheme, esp. with large programs





NOT OK!





Class Exercise 1

Demo!

- Let's write a program together that contains a function, called *FallTime*, that calculates the time it takes for a mass to be dropped from a variable height *h*, given the formula: $t = \sqrt{\frac{2d}{g}}$ `= sqrt(0.2038*d)`

Algorithm:

1. FallTime will take as argument, *d*. It will return the value of *t*.
2. main() will ask the user for h (in meters).
3. main() will call FallTime(h).
4. main() will print out the value of FallTime(h) (in seconds).

Class Exercise 2

- Let's write a program together that contains a function, called *WriteIt*, that takes a string called ***message*** and an integer called ***r***.

- The function prints out the string repeated ***r*** times with an exclamation mark and space between each repetition.

- The function does not return anything.

Function Calls

- When you call a function, your arguments are getting passed on as ***values*** into the function
  - At least, with what we've seen so far…
  - The call **`funcX(a, b)`** passes (into the function) the *values* of **a** and **b**
    - Seems logical enough…!?

- The vars representing **a** and **b** are treated as **local variables to the function**

Call-by-Value vs Call-by-Reference

- Our "usual" way of passing variables into functions is called "**Calling by Value**"

- You can ***also*** call a function with your arguments used as ***references*** to the actual variable location in memory
  - So, you're not passing the variable itself, but it's *location in memory!*
  - **What practical reason would we want to do that?**

ANS: Vars inside functions are local to the function!
What if we wanted them to change ***outside of it***?

Call-by-Reference Parameters

- "**Call-by-reference**" parameters allow us to change the variable used in the function call
- "**Call-by-value**" (i.e. the "usual" way) parameters do NOT change the variable used in the function call
- In the example shown here, the output would be:

```
x in fun1: 9
x in fun2: 9
a = 5; b = 9
```

*Why did **a** not change??*
*Why did **b** change??*

- We use the ampersand symbol (**&**) to distinguish a variable as being called-by-reference, in a **function definition**

```
int main()
{
…   …   …
    int a = 5, b = 5;
    fun1(a);
    fun2(b);
    cout << "a = " << a << "; ";
    cout << "b = " << b << endl;
…   …   …
}

void fun1(int x)  // call by value
{
    x += 4;
    cout << "x in fun1: " << x << endl;
}

void fun2(int &x)  // call by ref.
{
    x += 4;
    cout << "x in fun2: " << x << endl;
}
```

Demo!

Command Line Arguments with C++

- In C++ you can accept **command line arguments**
  - That is, *when you execute* your code, *you can pass input values* ***at the same time***

- These are arguments (inputs) that are passed *into* the program *from* the OS command line

- For example, from the Linux OS command line:

```
$ ./addThese 2 3
5
```

← You're passing 2 and 3 as inputs to the program

← and when it's executed, the program gives you its output (answer).

Command Line Arguments with C++

- To use command line arguments in your program,
  you must add **2 special arguments** to the **main( )** function
- Argument #1:

  The number of elements that you are passing in: **argc**

- Argument #2:

  The full list of all of the command line arguments as an array: ***argv[ ]**

  This is an array pointer … never mind the details, but more on those in a later class…

Command Line Arguments with C++

- The **main()** function header should be written as:

```
int main(int argc, char* argv[]) { … }
```

instead of

```
int main() { … }
```

- In the OS, to execute the program, the command line form should be:

```
$ program_name  argument1 argument2 … argumentn
```

example:

```
$ sum_of_squares 4 5 6
```

Demo!

```
int main ( int argc, char *argv[] )
{
   cout << "There are " << argc << " arguments here:" << endl;
   cout << "Let's print out all the arguments:" << endl;

   for (int i = 0; i < argc; i++)
      cout << "argv[" << i << "] is : " << argv[i] << endl;

   return 0;
}
```

argv[*n*] Is Always a Character Type!

- While **argc** is always an **int** (it's calculated by the compiler for you)…

  …all you get from the command-line is **character arrays**

  – This is a hold-out from the early days of C (i.e. pre-C++)

  – So, the data type of argument being passed is always an *array of characters* (a.k.a. a *C-string* – more on those later in the quarter…)

- To treat an argument as another type (like a number, for instance), you have to first ***convert it inside your program***

- **<cstdlib>** library has pre-defined functions to help!

What If I Want an Argument That's a Number?

argv[] to int

- Examples: **atoi( )** and **atof( )** ← *argv[] to double*
  Convert a **character array** into **int** and **double**, respectively.

*These functions are in **<cstdlib>***

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    int num1 = atoi(argv[1]);
    int num2 = atoi(argv[2]);
    int add = num1 + num2;
    int prod = num1 * num2;
    cout << add << " " << prod <<
endl;
    return 0;
}
```

*This is the only way that we can do **arithmetic** on the first 2 arguments*

YOUR TO-DOs

- ❑ Start **Lab3** today
- ❑ Do **Homework3**
- ❑ Do **Quiz3** this week (Fri.)


</LECTURE>