

# Functions in C++

## Part 2

CS 16: Solving Problems with Computers I  
Lecture #5

Ziad Matni  
Dept. of Computer Science, UCSB

# Announcements

---

- NO more adds for this class
- If you want to switch labs, switch with SOMEONE
  - The TAs have to approve
- Your 1<sup>st</sup> Midterm Exam is NEXT THURSDAY (10/19)!!!
  - ***Omgomgomgomgomgomgomgomgomgomgomg***

# MIDTERM #1 IS COMING!

October 19<sub>th</sub>!

- Material: **Everything** we've done, incl. up to Tue. 10/17
  - Homework, Labs, Lectures, Textbook
- **Thursday, 10/19** in this classroom
- Starts at **2:00pm \*\*SHARP\*\*** (come early)
- Ends at **3:15pm \*\*SHARP\*\***
- **BRING YOUR STUDENT IDs WITH YOU!!!**
- Closed book: no calculators, no phones, no computers
- Only 1 sheet (single-sided) of written notes
  - Must be no bigger than 8.5" x 11"
  - **You have to turn it in with the exam**
- **You will write your answers on the exam sheet itself.**



# What's on the Midterm#1?

## *From the Lectures, including...*

---

- Intro to Computers, Programming, and C++
- Variables and Assignments
- Boolean Expressions (comparison of variables)
- Input and Output on Standard Devices (cout, cin)
- Data Types, Escape Sequences, Formatting Decimal
- Arithmetic Operations and their Priorities
- Boolean Logic Operators
- Flow of Control & Conditional Statements
- Loops: for, while, do-while
- Types of Errors in Programming
- Multiway Branching and the switch command
- Generating Random Numbers
- Functions in C++:
  - pre-defined, user-defined
  - void functions, the main() function
  - call-by-ref vs. call-by-value, overloading
- Command Line Inputs to C++ Programs
- Numerical Conversions

# Midterm Prep

---

1. Lecture slides
2. Homework problems
3. Lab programs
4. Book chapters 1 thru 5\*

\*check the lecture slides with it!!



# Lecture Outline

---

- **void** functions
- Call-by-**value** vs. Call-by-**reference** Functions
- Overloading Functions
- Command-line Arguments

# Let's Go Over Some of the Demos

---

...from the last lecture...



# void Functions

- Sometimes, we want sub-tasks to be implemented as functions.
  - Repetition involved
- A subtask might produce:
  - 1 or more values      --or--      no values at all!
- We just described how to implement functions that return 1 value
  - So what about the other cases?

We use a **void-function**



## Simple **void** Function Example

```
1 // void function example
2 #include <iostream>
3 using namespace std;
4
5 void printmessage ()
6 {
7     cout << "I'm a function!";
8 }
9
10 int main ()
11 {
12     printmessage ();
13 }
```

## void Function Definition

Example: A program does °F  $\leftrightarrow$  °C conversion and then wants to print out the results. It does this last thing with a void function.

```
void show_results(double f_degrees, double c_degrees)
{
    cout << f_degrees << " degrees Fahrenheit is equivalent to "
         << c_degrees << " degrees Celsius." << endl;

    return;
}
```

NOTE: The **return;** statement does **not** include a var. expression. The “return” is just there to let the compiler know: the function’s done. It’s also OPTIONAL TO USE in this case!

# Calling **void** Functions

- void-function calls are, essentially, *executable statements*
  - They do not need to be part of another statement
  - They end with a semi-colon

- Example:

```
void show_results(double f_degrees, double c_degrees)
{
    cout << f_degrees << " degrees Fahrenheit is equivalent to "
        << c_degrees << " degrees Celsius." << endl;
    return;
}
```

Call it with: `show_results(32.5, 0.3);`

NOT with: `cout << show_results(32.5, 0.3);`

Will not compile!!

*This distinction is important and a typical rookie mistake to make!!!*

## void Functions: To Return or Not Return?


- In void functions, we need “return” to signal the end of the function
  - Is it strictly necessary for that? *No, it's optional*
- Can we use “return” to signal an “interrupt” to the function...
  - ...and end prematurely? *Yes!*
- Example: What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?
  - See example on next page of a void function that avoids division by zero with a return statement

## Use of *return* in a *void* Function

### Function Declaration

```
void ice_cream_division(int number, double total_weight);  
//Outputs instructions for dividing total_weight ounces of  
//ice cream among number customers.  
//If number is 0, nothing is done.
```

### Function Definition

```
//Definition uses iostream:  
void ice_cream_division(int number, double total_weight)  
{  
    using namespace std;  
    double portion;  
  
    if (number == 0)  return;  
    portion = total_weight/number;  
    cout.setf(ios::fixed);  
    cout.setf(ios::showpoint);  
    cout.precision(2);  
    cout << "Each one receives "  
        << portion << " ounces of ice cream." << endl;  
}
```

# The **main** Function in C++

- The **main** function in a program **is used like a void function**
  - So why do we have to end the program with a return statement?
  - And why isn't it DEFINED as a void function?
- The **main** function is defined to return a value of type **int**,  
therefore a return is needed
  - It's a matter of what is "legal" and "not legal" in C++
  - **void main ( )** is not legal in C++ !! (this ain't Java)
  - Most compilers **will not accept a void main** (*none of the ones we're using, anyway...*)
  - Solution? **Stick to what's legal**: it's ALWAYS: **int main ( )**



## The **main** Function in C++

---

- The C++ standard also says the **return 0** *can* be omitted, but many compilers still require it
- No compiler will complain if you *have* the **return 0** in **main( )**
- Solution?  
*Always* include **return 0** in the main function to be safe
  - Because you don't control everyone's compiler choices!

## Class Exercise

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:

$$t = \sqrt{\frac{2d}{g}} = \text{sqrt}(0.2038 \ d)$$

### Algorithm:

1. *FallTime* will take as argument,  $h$ . 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).

# Call-by-Value vs Call-by-Reference

- When you call a function, your arguments are getting passed on as *values*
  - At least, with what we've seen so far...
  - The call **func(a, b)** passes on (into the function) the *values* of **a** and **b**
- 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*!
  - Why would we want to do that?

**ANS:** Vars inside functions are local.  
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**” parameters do NOT change the variable used in the function call
- In the example shown here, the output would be:

```
9
9
a, b = 5, 9
```

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

```
...
//inside main...
...
int a = 5, b = 5;
fun1(a);
fun2(b);
cout << "a, b = "
      << a << ", " << b << endl;
...

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

void fun2 (int &x) {
    x += 4;
    cout << x << endl;
}          // call-by-reference
```

# Call-By-Reference Details

---

- The ***memory location*** of the argument variable is given to the formal parameter
  - Not the argument variable itself!
- Whatever is done to a formal parameter *inside* the function, is actually done to the value *at the memory location* of the argument variable
  - A subtle, but important, difference!



# Call-by-Reference Behavior

- Assume **int** variables **first** and **second** are assigned memory addresses **1036** and **1040** (*this is usually done by the compiler. Also, these are made-up mem addresses*)

- Now a function call executes: `get_numbers(first, second);`

- The function is defined as:

```
void get_numbers(int &first, int &second) {  
    cout << "Enter two integers: ";  
    cin >> first >> second; }
```

- The function may as well say:

```
void get_numbers(the int var at mem location 1036, the int var at mem location 1040) {  
    cout << "Enter two integers: "  
    cin >> the variable at memory location 1036;  
    >> the variable at memory location 1040; }
```



## Example: swap\_values

```
void swap(int &variable1, int &variable2)
{
    int temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

- If called with `swap(first_num, second_num);`
  - The values of `first_num` and `second_num` are swapped
  - Can ONLY do this if the function is call-by-reference

# Mixed Parameter Lists

- Call-by-value and call-by-reference parameters  
can be **mixed in the same function**
- Example:  
`void good_stuff(int &par1, int par2, double &par3);`
  - `par1` and `par3` are call-by-reference formal parameters
    - Changes in `par1` and `par3` *change* the argument variable
  - `par2` is a call-by-value formal parameter
    - Changes in `par2` *do not change* the argument variable

## Caution! Inadvertent Local Variables

---

- Forgetting the ampersand (&) creates a call-by-value parameter
  - The value of the variable will not be changed
  - You just ensured that a variable will remain local to the function  
*(when your intention was NOT to do that!)*
- Hard error to debug/find... because it looks right!

# YOUR TO-DOs

---

- ☐ Finish reading up Chapter 5
- ☐ Turn in Lab2 by TOMORROW AT NOON (Fri, 10/13)
- ☐ Start on HW3
- ☐ Visit Prof's and TAs' office hours if you need help!
- ☐ Did you drink enough water today?

**</LECTURE>**