Call-by-Type Functions in C++
Command-Line Arguments in C++

CS 16: Solving Problems with Computers I
Lecture #5

Ziad Matni
Dept. of Computer Science, UCSB

Administrative

- **CHANGED T.A. OFFICE/OPEN LAB HOURS!**
  - Thursday, 10 AM – 12 PM Muqsit Nawaz
  - Friday, 11 AM – 1 PM Xiyou Zhou
- Linux Workshop **THIS** Week!
  - HFH Conference Room (HFH 1132)
  - Friday, April 20th, 1:00 – 2:30 PM
  - Material will be put up on the class website
- Your 1st Midterm Exam is NEXT TUESDAY (4/24)!!!
  - ***Omgomgomgomgomgomgomgomgomgomg***

MIDTERM IS COMING!

- **Tuesday, 4/24** in this classroom
- **Starts at 2:00 PM **SHARP****
  - Please start arriving 5-10 minutes before class
- **I may ask you to change seats**
- Please bring your UCSB IDs with you

- **Closed book**: no calculators, no phones, no computers
- ***Only*** allowed ONE 8.5”x11” sheet of notes − one sided only
  - You have to turn it in with your 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
- Command Line Inputs to C++ Programs
- Separate compilations and makefiles

Midterm Prep

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

   *check which lecture slides go with it!!

Lecture Outline

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

Class Exercise 1

- 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***. It then prints out the string repeated ***r*** times with an exclamation mark and space between each repetition. The function does not return anything.

Call-by-Value vs Call-by-Reference

- 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 on (into the function) the *values* of **a** and **b**
    - Seems logical enough…!?

- 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 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**" 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;
}
```

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 memory 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;
}
```

Call-By-Reference Details

```
void fun2(int &x)  // call by ref.
```

- 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!

- It has the effect of making the called-by-reference variable act like a global var.
  - If it changes inside the function, it changes outside the function too
  - But it's better than using a global variable! …(why?)

Class Exercise 3

- Let's write a program together that contains a function, called ***swap***, that takes a two integer variables as input arguments and causes their values to swap, like in this example:

```
int a = 3, b = 9;
cout << a << "; " << b << endl;
// This should print out "3; 9"
swap(a, b);
cout << a << "; " << b << endl;
// This should print out "9 ; 3"
```

Example: **swap_values**

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

We 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
  - You just ensured that a variable will remain local to the function

    (when your intention was NOT to do that!)

- **This is a hard error to debug/find**… because it looks right!
  - So, be careful…

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( )** Convert a **character array** into **int** and **double**, respectively.

argv[] to double

*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 << num3 << endl;
    return 0;
}
```

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

YOUR TO-DOs

- ❑ Do Lab3 tomorrow (due Monday)
- ❑ Do HW5 by next Thursday

- ❑ Visit Prof's and TAs' office hours if you need help!

- ❑ Eat your vegetables

</LECTURE>