COMPUTER SCIENCE 1: STARTING COMPUTING CSCI 1300

Ioana Fleming

Lecture 6

University of Colorado
Boulder

Reminders

Submissions:

- Rec 2: Fri 9/14 at 11:55 pm
- H2: Sun 9/16 at 6:00 pm
- Quiz 1: Mon 9/10 at 11:59 pm
- Quiz 2: Mon 9/17 at 11:59 pm

Readings:

- 5.1 – 5.5 before Wed lecture
- 5.6 – 5.8 before Fri lecture
- Ch. 3 Decisions next week

1st Practicum: TBD

© samxmeg/iStockphoto.

Chapter Two: Fundamental Data Types

Increment and Decrement

- Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for these:

  The increment and decrement operators.

```
count++; // add 1 to count
count--; // subtract 1 from count
```

Example:

What is the value of variable `count` after the code below?

```
int count = 3;
count--;
count = count + 2;
count++;
```

Example

pre_post_increment.cpp

Formatted Output

- When you print an amount in dollars and cents, you want it to be *rounded* to two significant digits.

- A ***manipulator*** is something that is sent to **`cout`** to specify how values should be formatted.
- To use manipulators, you must include the **`iomanip`** header in your program:

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

Formatted Output for Dollars and Cents: `setprecision()`

Which do you think the user prefers to see on her gas bill?

```
Price per liter: $1.22
```

or

```
Price per liter: $1.21997
```

Formatted Output Examples: Table 7

Output Statement	Output	Comment
`cout << 12.345678;`	`12.3457`	By default, a number is printed with 6 significant digits.
`cout << fixed << setprecision(2) << 12.3;`	`12.30`	The `fixed` and `setprecision` manipulators control the number of digits after the decimal point.
`cout << ":" << setw(6) << 12;`	`:  12`	Four spaces are printed before the number, for a total width of 6 characters.
`cout << ":" << setw(2) << 123;`	`:123`	If the width not sufficient, it is ignored.
`cout << setw(6) << ":" << 12;`	`    :12`	The width only refers to the next item. Here, the : is preceded by five spaces.

Formatted Output, Dollars and Cents

You can combine manipulators and values to be displayed into a single statement:

```
price_per_liter = 1.21997;
cout << fixed << setprecision(2)
   << "Price per liter: $"
   << price_per_liter << endl;
```

This code produces this output:

```
Price per liter: $1.22
```

Formatted Output with `setw()` to Align Columns

Use the **`setw`** manipulator to set the *width* of the next output field.

width = the total number of characters, including digits, the decimal point, and spaces.

For aligned columns of certain widths, use the **`setw()`** manipulator.

For example, if you want a number to be printed, right justified, in a column that is eight characters wide, you use

```
<< setw(8)
```

before EVERY COLUMN's DATA.

Formatted Output - Example

This code:

```
price_per_ounce_1 = 10.2372;
price_per_ounce_2 = 117.2;
price_per_ounce_3 = 6.9923435;
cout << setprecision(2);
cout << setw(8) << price_per_ounce_1;
cout << setw(8) << price_per_ounce_2;
cout << setw(8) << price_per_ounce_3;
cout << "--------" << endl;
```

produces this output:

```
   10.24
  117.20
    6.99
--------
```

`setprecision` versus `setw`: Persistence

There is a notable difference between the **`setprecision`** and **`setw`** manipulators.

Once you set the precision, that precision is used for all floating-point numbers until the next time you set the precision.

But **`setw`** affects only the *next* value.

Subsequent values are formatted without added spaces.

ch02/volume2.cpp

A Complete Program for Volumes

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

int main()
{
   // Read price per pack

   cout << "Please enter the price for a six-pack: ";
   double pack_price;
   cin >> pack_price;

   // Read can volume

   cout << "Please enter the volume for each can (in ounces): ";
   double can_volume;
   cin >> can_volume;
```

A Complete Program for Volumes

```
   // Compute pack volume

   const double CANS_PER_PACK = 6;
   double pack_volume = can_volume * CANS_PER_PACK;

   // Compute and print price per ounce

   double price_per_ounce = pack_price / pack_volume;

   cout << fixed << setprecision(2);
   cout << "Price per ounce: " << price_per_ounce << endl;

   return 0;
}
```

Sample Program Run:

```
Please enter the price for a six-pack: 2.95
Please enter the volume for each can (in ounces): 12
Price per ounce: 0.04
```

Powers and Roots – #include <cmath>

- In C++, there are no symbols for powers and roots.
- To compute them, you must call *functions*. Don't forget to include the *cmath* library

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

Example of `pow()` function call

The power function has the base followed by a comma followed by the power to raise the base to:

```
pow(base, exponent)
```

Using the **`pow`** function:

```
double balance = b * pow(1 + r / 100, n);
```

Other Mathematical Functions (from <cmath>): Table 6

Table 6 Other Mathematical Functions

Function	Description
`sin(x)`	sine of x (x in radians)
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`log10(x)`	(decimal log) $\log_{10}(x)$, $x > 0$
`abs(x)`	absolute value $\lvert x \rvert$

Example:

```
double population = 73693997551.0;
double decimal_log = log10(population);
```


© attator/iStockphoto.

Chapter Five: Functions

by Cay Horstmann
University of Colorado Boulder

Chapter Goals

- To be able to implement functions
- To become familiar with the concept of parameter passing
- To appreciate the importance of function comments
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To recognize when to use value and reference parameters

Topic 1

1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. Stepwise refinement
8. Variable scope and globals
9. ~~Reference parameters~~
10. ~~Recursive functions~~

What Is a Function? Why Functions?

A function is a sequence of instructions with a name.

A function packages a computation into a form that can be easily understood and reused.

Calling a Function

```
int main()
{
   double z = pow(2, 3);
   ...
}
```

`main` *calls* the **`pow`** function, asking it to compute $\mathbf{2^3}$.

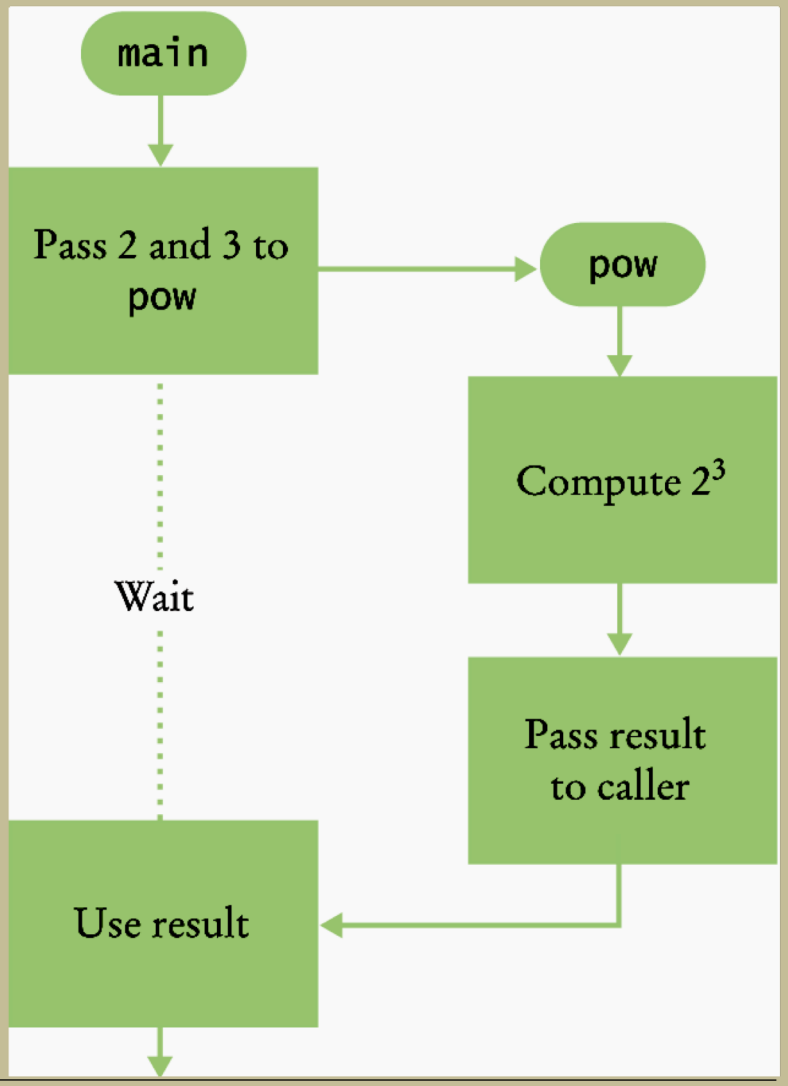
The **`main`** function is temporarily suspended.

The instructions of the **`pow`** function execute and compute the result.

The pow function *returns* its result back to **`main`**, and the **`main`** function resumes execution.

Flowchart: Calling a Function

Execution flow during a function call



Parameters

```
int main()
{
   double z = pow(2, 3);
   ...
}
```

When another function calls the **`pow`** function, it provides "inputs", such as the values 2 and 3 in the call **`pow(2, 3)`**.

In order to avoid confusion with inputs that are provided by a human user (**`cin >>`**), these values are called *parameter values*.

The "output" that the **`pow`** function computes is called the *return value* (not output using **<<**).

An Output Statement Does Not Return a Value

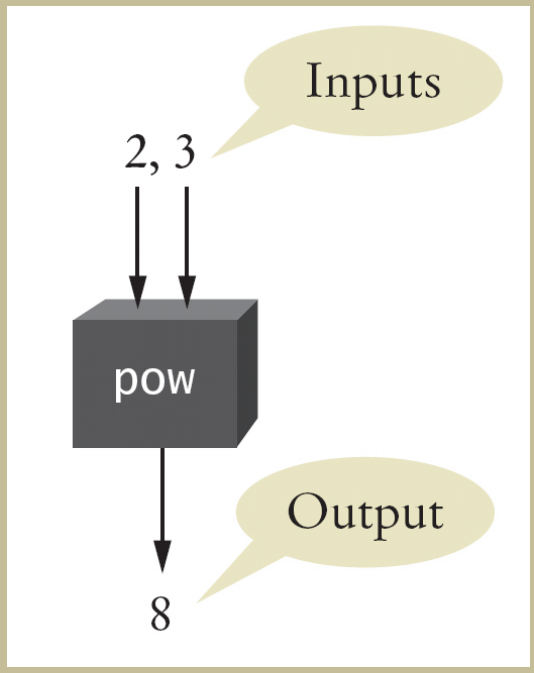
output ≠ return

- The **`return`** statement does not display output
  - Rather, it causes execution to resume in the calling program and ends the called function.
  - `return` may also pass a "value" back to the calling program

An output statement using << communicates *only* with the user running the program.

The Black Box Concept

- You can think of a function as a “black box” where you can’t see what’s inside but you know what it does.



- How did the **`pow`** function do its job?
- You don’t need to know.
- You only need to know its *specification*.

Topic 2

1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. Stepwise refinement
8. Variable scope and globals

Implementing Functions

Example: Calculate the volume of a cube

1. Pick a good, descriptive name for the function
2. Give a type and a name for each parameter.
3. There will be one parameter for each piece of information the function needs to do its job.
4. Specify the type of the return type:

```
double cube_volume(double side_length)
```

5. Then write the body of the function, as statements enclosed in braces

```
{  }
```

`cube_volume` Function

FYI: The comments at the top: description, parameters, return, algorithm

```
/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
{
  double volume = side_length * side_length * side_length;
  return volume;
}
```

How do you know your function works as intended?

Test your Functions

You should always test the function.

ch05/cube.cpp

You'll write a **`main`** function to do this.

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

/**
   Computes the volume of a cube.
   @param side_length the side length of the cube
   @return the volume
*/
double cube_volume(double side_length)
{
   double volume = side_length * side_length * side_length;
   return volume;
}
```

A Testbench Program (main)

```
int main()
{
   double result1 = cube_volume(2);
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume "
      << result1 << endl;
   cout << "A cube with side length 10 has volume "
      << result2 << endl;

   return 0;
}
```

Topic 3

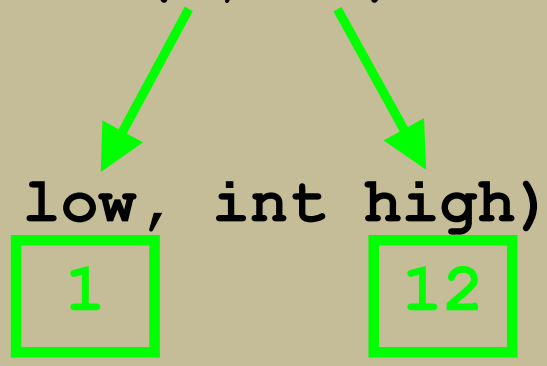
1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. Stepwise refinement
8. Variable scope and globals

Parameter Passing

When a function is called, a *parameter variable* is created for each value passed in.

Each parameter variable is *initialized* with the corresponding parameter value from the call.

```
int hours = read_value_between(1, 12);

. . .

int read_value_between(int low, int high)
```



Parameter Passing - example

Here is a call to the **`cube_volume`** function:

```
double result1 = cube_volume(2);
```

Here is the function definition:

```
double cube_volume(double side_length)
{
  double volume = side_length * side_length * side_length;
  return volume;
}
```

We'll keep up with their variables and parameters:

`result1`
`side_length`
`volume`

Parameter Passing

In the calling function (`main`), the variable **`result1`** is declared. When the **`cube_volume`** function is called, the parameter variable **`side_length`** is created & initialized with the value that was passed in the call (2).
After the return statement, the local variables `side_length` and `volume` disappear from memory.

Topic 4

1. Functions as black boxes
2. Implementing functions
3. Parameter passing
4. Return values
5. Functions without return values
6. Reusable functions
7. Stepwise refinement
8. Variable scope and globals

Return Values

The `return` statement ends the function execution. This behavior can be used to handle unusual cases.

What should we do if the side length is negative?

We choose to return a zero and not do any calculation:

```
double cube_volume(double side_length)
{
  if (side_length < 0)
      return 0;
  double volume = side_length * side_length * side_length;
  return volume;
}
```

Nothing is executed after a return statement !!! Execution returns to main

Return Values: Shortcut

The **`return`** statement can return the value of any expression.

Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

```
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

Common Error – Missing Return Value

Your function always needs to return something.

The code below: what is returned if the call passes in a negative value!

You need to ensure all paths of execution include a `return` statement.

```
double cube_volume(double side_length)
{
  if (side_length >= 0)
  {
     return side_length * side_length *
   side_length;
  }
}
```

Function Declarations (Prototype Statements)

- It is a compile-time error to call a function that the compiler does not know
  - just like using an undefined variable.
- So define all functions before they are first used
  - But sometimes that is not possible, such as when 2 functions call each other

Function Declarations (Prototype Statements)

- Therefore, some programmers prefer to include a definition, aka "prototype" for each function at the top of the program, and write the complete function after `main(){}`
- A prototype is just the function header line followed by a semicolon:

```
double cube_volume(double side_length);
```

- The variable names are optional, so you could also write it as:

```
double cube_volume(double);
```

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

// Declaration of cube_volume
double cube_volume(double side_length);

int main()
{
   double result1 = cube_volume(2); // Use of cube_volume
   double result2 = cube_volume(10);
   cout << "A cube with side length 2 has volume "<< result1<< endl;
   cout << "A cube with side length 10 has volume "<< result2<< endl;
   return 0;
}

// Definition of cube_volume
double cube_volume(double side_length)
{
   return side_length * side_length * side_length;
}
```

Steps to Implementing a Function

1. Describe what the function should do.
   - EG: Compute the volume of a pyramid whose base is a square.
2. Determine the function's "inputs".
   - EG: height, base side length
3. Determine the types of the parameters and return value.
   - EG: `double pyramid_volume(double height, double base_length)`
4. Write pseudocode for obtaining the desired result.

   volume = 1/3 x height x base length x base length
5. Implement the function body.

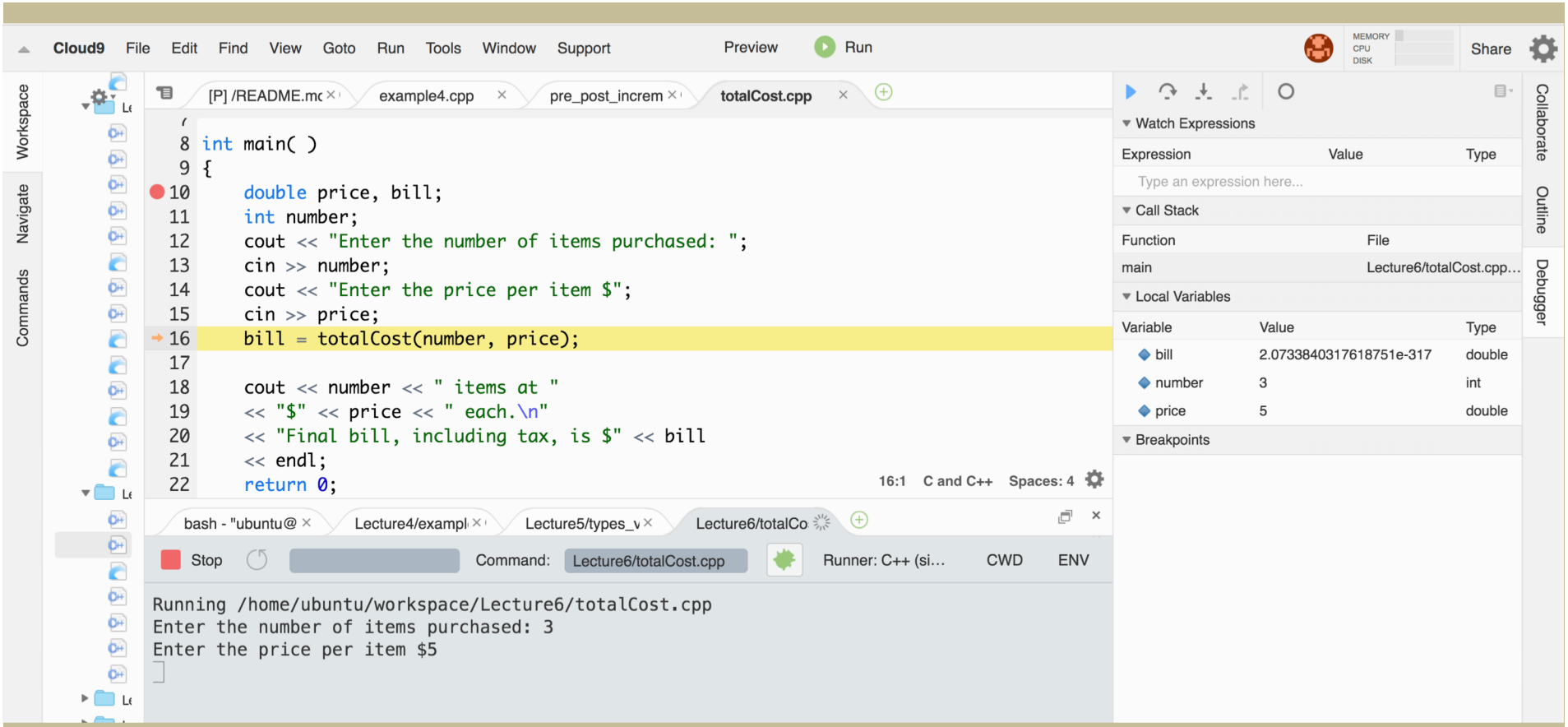
```
{
   double base_area = base_length * base_length;
   return height * base_area / 3;
}
```
6. Test your function
   - Write a `main()` to call it multiple times, including boundary cases

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

/**   Computes the volume of a pyramid whose base is a square.
   @param height the height of the pyramid
   @param base_length length of one side of the pyramid's base
   @return the volume of the pyramid
*/
double pyramid_volume(double height, double base_length)
{
   double base_area = base_length * base_length;
   return height * base_area / 3;
}

int main()
{
   cout << "Volume: " << pyramid_volume(9, 10) << endl;
   cout << "Expected: 300";
   cout << "Volume: " << pyramid_volume(0, 10) << endl;
   cout << "Expected: 0";
   return 0;
}
```

Using the IDE Debugger

- Your Cloud9 IDE includes a debugger that:
  - Allows execution of the program one statement at a time
  - Shows intermediate values of local function variables
  - Sets “breakpoints” to allow stopping the program at any line to examine variables

These features greatly speed your correcting your code.



- There is a breakpoint on line 10.
- Next line to be executed is shown by yellow arrow in the Breakpoint margin at left.
- The Debugger panel at right shows the Local Variables: notice *number* and *price* already have a value, but *bill* does not have a value yet

Using the IDE Debugger

Typical debug session:

1. Set a breakpoint early in the program, by clicking on a line in the source code
2. Start execution with the “Run” button
3. When the code stops at the breakpoint, examine variable values in the variables window
4. Step through the code one line at a time or one function at a time, continuing to compare variable values to what you expected
5. Determine the error in the code and correct it, then go to step 1.