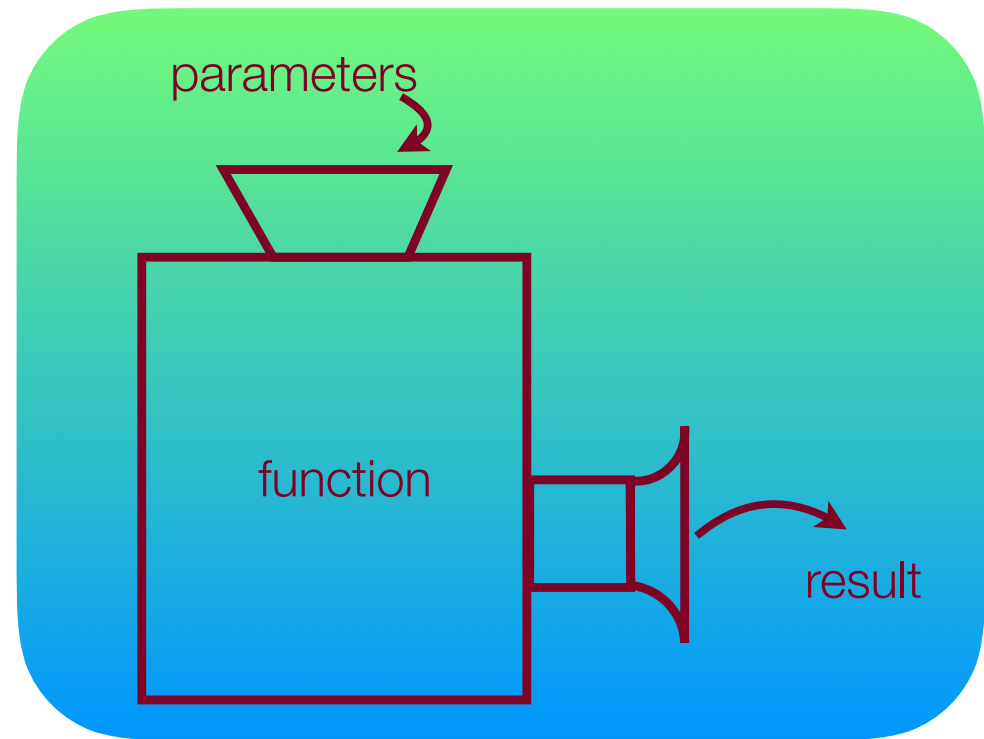# CS 106B
Lecture 2: C++ Functions

Wednesday, September 28, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapters 2-3

parameters

function

result

Today's Topics

- Logistics:
  - Signing up for section
  - CS 106L
  - Qt Creator installation help on Thursday
  - Stanford Local Programming Contest: Saturday
- Homework 1: Fauxtoshop!
  - Due Friday, October 7, at Noon
  - A note on the honor code and cheating
  - YEAH Hours tonight! (will be recorded!)
- Functions
  - Some review — functions are very similar to Java functions!
  - Value semantics
  - Reference semantics
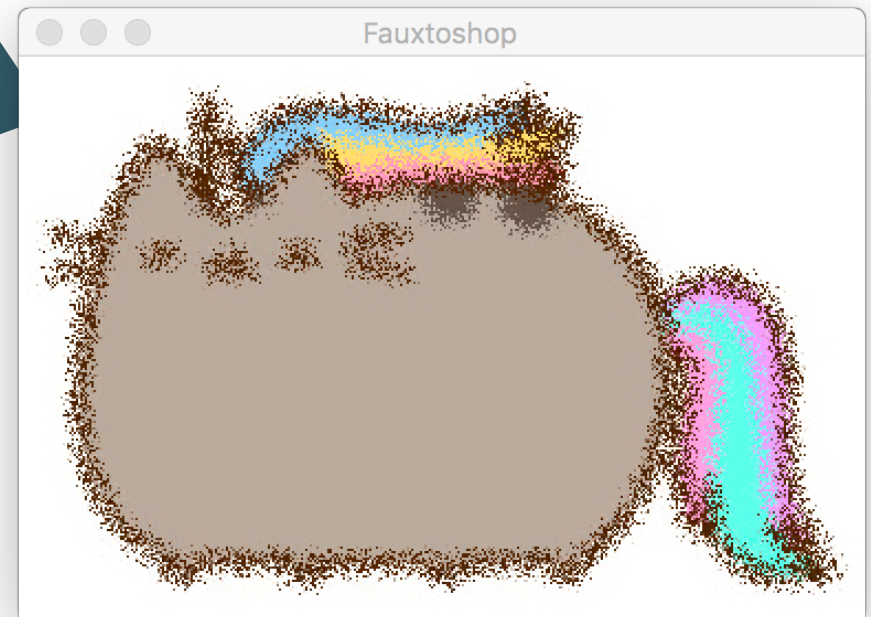- Reading Assignment: Chapters 2 and 3

Logistics

- Signing up for section: you must put your available times by Sunday October 2 at 5pm (opens Thursday at 5pm).
  - Go to cs198.stanford.edu to sign up.

- CS 106L: A great opportunity to dig a bit deeper into "real" C++, and to see interesting programming examples. Meets Tu/Th 1:30-2:50 in Hewlett 101.

- Qt Creator installation help: Thursday at 8pm, in Tressider (eating area). Please attempt to install Qt Creator before you arrive (see the course website for details).
  - Remember, Assignment 0 is due Friday at Noon

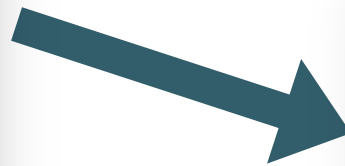- Stanford Local Programming Championship: Saturday: cs.stanford.edu/group/acm/SLPC/

Scatter!

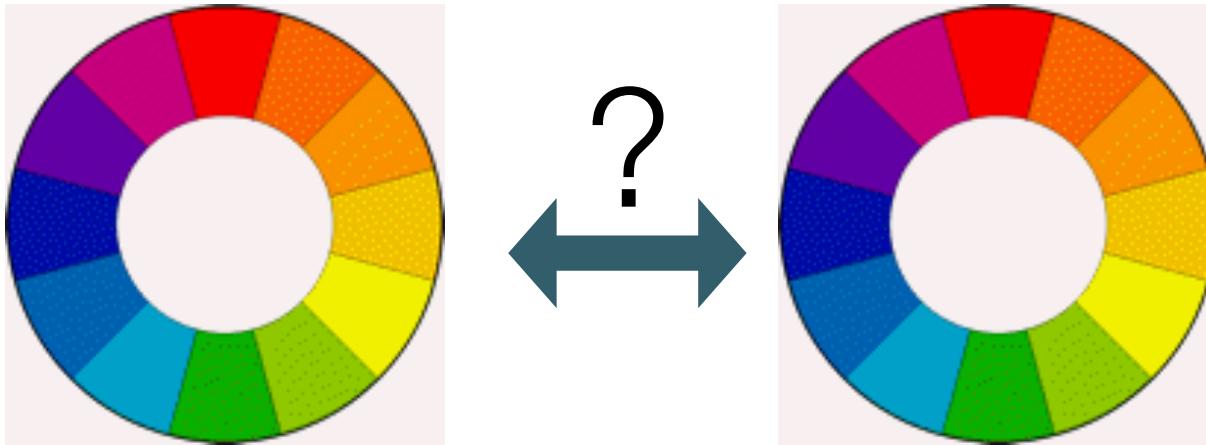# Assignment 1: Fauxtoshop!

Green Screen Merging!

Compare Images!

Fauxtoshop

- The program you write will utilize:
  - Functions
  - Constants
  - Loops
  - I/O (cout, getLine(), getInteger())
  - Reference semantics, Value semantics
  - Strings
  - Logic
  - Nicholas Cage
- We will discuss all of the above before the project is due
- **Get started early**! (Idea: finish Scatter by Friday!)
- Go to YEAH hours! Go to the LaIR!
- This is a *non*-pair programming program — you must work independently
- Due: 12pm (Noon) on Friday, October 7

Fauxtoshop

- A comment on the Honor Code (handout)

A C++ **function** is like a Java **method**. Similar declaration syntax but without the public or private keyword in front.

return type   *parameters*

```
type functionName(type name, type name, ..., type name) {
    statement;
    statement;
    ...
    statement;
    return expression; // if return type is not void
}
```

*arguments (called in the **same order** as the parameters)*

Calling a function:
```
functionName(value, value, ..., value);
```

Function Return Types

A C++ function must have a return type, which can be any time (including user-defined types, which we will cover later).

```cpp
double square(double x); // returns a double
Vector<int> matrixMath(int x, int y); // returns a Vector
                                      // probably not a good
                                      // idea! (covered later)
string lowercase(string s); // returns a string (maybe
                            // not a good idea...
void printResult(Vector<int> &v); // returns nothing!
```

A C++ function can only return a single type, and if you want to return multiple "things," you have to do it differently (unlike in languages such as Python). We will cover this later, as well.

Function Example, brought to you by

(bus drivers *hate* them!)

```cpp
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
        << (count-1) << " bottles of " << DRINK_TYPE
        << " on the wall." << endl << endl;
}

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
```

```
99 bottles of Coke on the wall.
99 bottles of Coke.
Take one down, pass it around, 99 bottles of Coke on the wall.

98 bottles of Coke on the wall.
98 bottles of Coke.
Take one down, pass it around, 98 bottles of Coke on the wall.

97 bottles of Coke on the wall.
97 bottles of Coke.
Take one down, pass it around, 97 bottles of Coke on the wall.

...

2 bottles of Coke on the wall.
2 bottles of Coke.
Take one down, pass it around, 2 bottles of Coke on the wall.

1 bottles of Coke on the wall.
1 bottles of Coke.
Take one down, pass it around, 1 bottles of Coke on the wall.

0 bottles of Coke on the wall.
0 bottles of Coke.
Take one down, pass it around, 0 bottles of Coke on the wall.
```

(bus drivers *hate* them!)

```cpp
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
```

How many functions does this program have?

Answer: 2. bottles() and main()

What does the `bottles()` function return?

Answer: nothing (void function)

Function Example, brought to you by

(bus drivers *hate* them!)

Why is it a good idea to make DRINK_TYPE a constant?

Answer: So we can change it to Pepsi if we are masochists. (actual answer: it allows us to make one change that affects many places in the code)

```cpp
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
        << (count-1) << " bottles of " << DRINK_TYPE
        << " on the wall." << endl << endl;
}

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
```

Function Example 2

```cpp
// Function example #2: returning values

#include <iostream>
#include "console.h"

using namespace std;

int larger(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Returns the larger of the two values.
int main() {
    int bigger1 = larger(17, 42); // call the function
    int bigger2 = larger(29, -3); // call the function again
    int biggest = larger(bigger1, bigger2);
    cout << "The biggest is " << biggest << "!!" << endl;
    return 0;
}
```

Function Example 2

```cpp
// Function example #2: returning values

#include <iostream>
#include "console.h"

using namespace std;

int larger(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Returns the larger of the two values.
int main() {
    int bigger1 = larger(17, 42); // call the function
    int bigger2 = larger(29, -3); // call the function again
    int biggest = larger(bigger1, bigger2);
    cout << "The biggest is " << biggest << "!!" << endl;
    return 0;
}
```
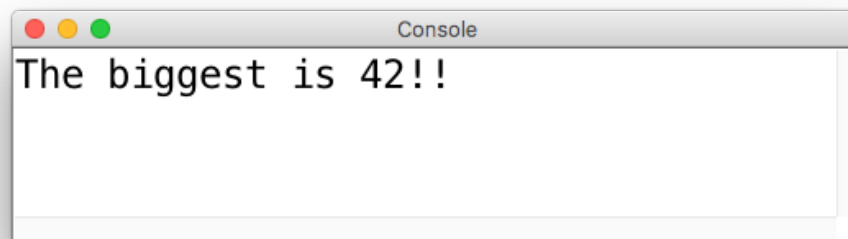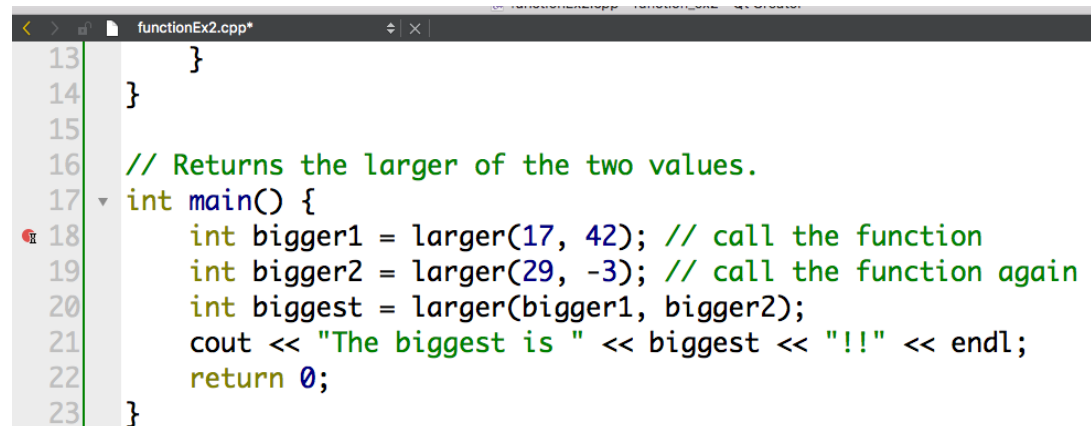
Output:

```
Console
The biggest is 42!!
```

Function Example 2: Debugging

- One of the most powerful features of an Integrated Development Environment (IDE) like Qt Creator is the built-in debugger.
- You can stop the program's execution at any point and look at exactly what is going on under the covers!
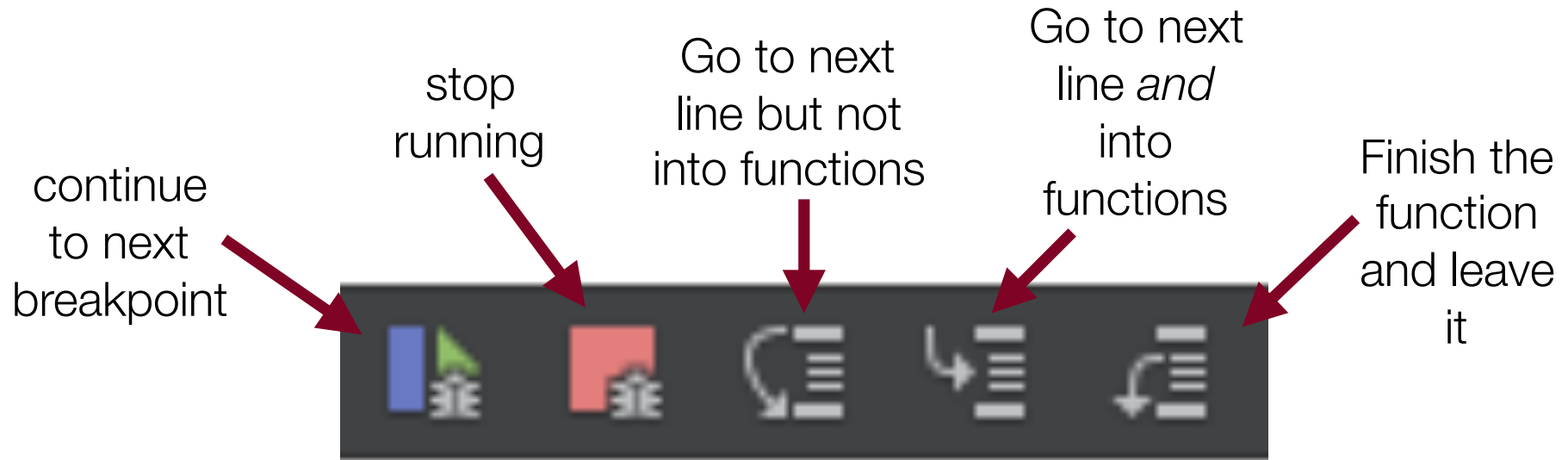- In your program, click to the left of a line of code (line 18 below, for example)

```cpp
13          }
14      }
15
16      // Returns the larger of the two values.
17  ▼ int main() {
18          int bigger1 = larger(17, 42); // call the function
19          int bigger2 = larger(29, -3); // call the function again
20          int biggest = larger(bigger1, bigger2);
21          cout << "The biggest is " << biggest << "!!" << endl;
22          return 0;
23      }
```

- When you run the program in Debug mode (the green triangle with the bug on it), the program will stop at that point. Let's see this in action!

Function Example 2: Debugging

- Notes from live debugging:
  - You can see variable values as the program executes
  - You use the following buttons to continue the program:

continue to next breakpoint

stop running

Go to next line but not into functions

Go to next line *and* into functions

Finish the function and leave it

- Debugging effectively takes a little time to learn, but is super effective if you have hard to find bugs.

Declaration Order

```cpp
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}


// Function Definition and Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
         << (count-1) << " bottles of " << DRINK_TYPE
         << " on the wall." << endl << endl;
}
```

- Believe it or not, this program does not compile!
- In C++, functions *must* be declared somewhere before they are used.
- But, we like to put our main() function first, because it is better style.

Declaration Order

```cpp
#include <iostream>
#include "console.h"

using namespace std;

const string DRINK_TYPE = "Coke";

// Function Definition
void bottles(int count);

int main() {
    for (int i=99; i > 0; i--) {
        bottles(i);
    }
    return 0;
}
// Function Code
void bottles(int count) {
    cout << count << " bottles of " << DRINK_TYPE << " on the wall." << endl;
    cout << count << " bottles of " << DRINK_TYPE << "." << endl;
    cout << "Take one down, pass it around, "
        << (count-1) << " bottles of " << DRINK_TYPE
        << " on the wall." << endl << endl;
}
```

- Believe it or not, this program does not compile!
- In C++, functions *must* be declared somewhere before they are used.
- But, we like to put our main() function first, because it is better style.
- What we can do is define the function (called a "*function prototype*") without its body, and that tells the compiler about the function "signature" and the compiler is happy.

C++ Pre-written Functions

- You have written a lot of functions before. What if we wanted to find the square root of a number?
- We could manually write out a function (remember Newton's Method??) -- see the textbook!
- But, this would be counterproductive, and many math functions have already been coded (and coded well!)
- The <cmath> library already has lots and lots of math functions that you can use (you can go look up the code! Actually...it's complicated -- see https://goo.gl/Y9Y55w if you are brave. It is most likely true that the square root function is *built into your computer's processor, so there isn't any readable code*)
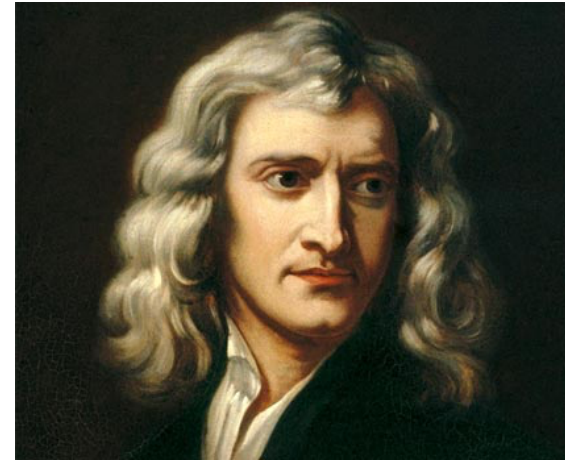
`#include <cmath>`

Function	Description (returns)
abs(value)	absolute value
ceil(value)	rounds up
floor(value)	rounds down
log10(value)	logarithm, base 10
max(value1, value2)	larger of two values
min(value1, value2)	smaller of two values
pow(base, exp)	*base* to the *exp* power
round(value)	nearest whole number
sqrt(value)	square root
sin(value) cos(value) tan(value)	sine/cosine/tangent of an angle in radians
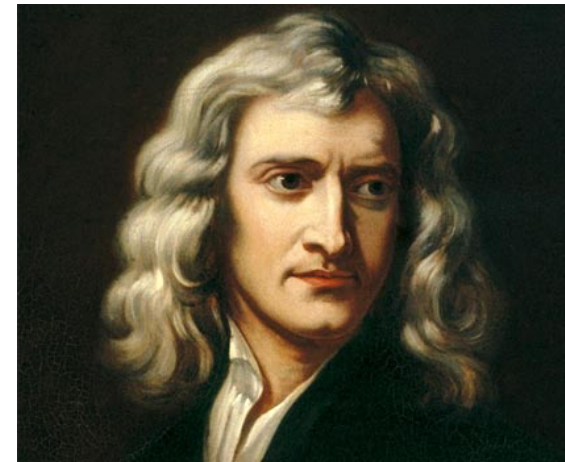
- unlike in Java, you don't write Math. in front of the function name
- see Stanford "gmath.h" library for additional math functionality

Value semantics

- **value semantics:** In Java and C++, when variables (`int, double`) are passed as parameters, their values are copied.
  - Modifying the parameter will not affect the variable passed in.

```cpp
void grow(int age) {
    age = age + 1;
    cout << "grow age is " << age << endl;
}

int main() {
    int age = 20;
    cout << "main age is " << age << endl;
    grow(age);
    cout << "main age is " << age << endl;
    return 0;
}
```

```
Output:
main age is 20
grow age is 21
main age is 20
```

- **reference semantics:** In C++, if you declare a parameter with an **&** after its type, instead of passing a copy of its value, it will link the caller and callee functions to the same variable in memory.
  - Modifying the parameter *will* affect the variable passed in.

```cpp
void grow(int &age) {
    age = age + 1;
    cout << "grow age is " << age << endl;
}

int main() {
    int age = 20;
    cout << "main age is " << age << endl;
    grow(age);
    cout << "main age is " << age << endl;
    return 0;
}
```

```
Output:
main age is 20
grow age is 21
main age is 21
```

- **Notes about references:**
  - References are super important when dealing with objects that have a lot of elements (Vectors, for instance). Because the reference does not copy the structure, it is fast. You don't want to transfer millions of elements between two functions if you can help it!
  - The reference syntax can be confusing, as the "**&**" (ampersand) character is also used to specify the address of a variable or object. The **&** is only used as a reference parameter in the function declaration, not when you call the function:

```
                  yes!
void grow(int &age) {
    age = age + 1;
    cout << "grow age is "
        << age << endl;
}
```

```
int main() {
    grow(age);
              yes!
    grow(&age);
           no!
    return 0;
}
```

Reference pros/cons

- benefits of reference parameters:
  - a useful way to be able to 'return' more than one value
  - often used with objects, to avoid making bulky copies when passing

- downsides of reference parameters:
- hard to tell from call whether it is ref; can't tell if it will be changed

```
foo(a, b, c); // will foo change a, b, or c? :-/
```
- (very) slightly slower than value parameters
- can't pass a literal value to a ref parameter

```
grow(39); // error
```

Reference Example

- Without references, you can't write a swap function to swap two integers. This is true about Java. What happens with the following function?

```
/*
 * Attempts to place a's value into b and vice versa.
 */
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Answer: the original variables are unchanged, because they are passed as copies (values)!

- *With* references, you *can* write a swap function to swap two integers, because you can access the original variables:

```
/*
 * Places a's value into b and vice versa.
 */
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Answer: the original variables are unchanged, because they are passed as copies (values)!

What is the output of this code? Talk to your neighbor!

```cpp
void mystery(int& b, int c, int& a) {
    a++;
    b--;
    c += a;
}

int main() {
    int a = 5;
    int b = 2;
    int c = 8;
    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

```
// A. 5 2 8
// B. 5 3 7
// C. 6 1 8
// D. 61 13
// E. other
```

Note: please don't obfuscate your code like this! :(
See the International Obfuscated C Contest for much, much worse examples

What is the output of this code?

```
void mystery(int& b, int c, int& a) {
    a++;
    b--;
    c += a;
}

int main() {
    int a = 5;
    int b = 2;
    int c = 8;
    mystery(c, a, b);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

```
// A. 5 2 8
// B. 5 3 7
// C. 6 1 8
// D. 61 13
// E. other
```

Note: please don't obfuscate your code like this! :(

See the International Obfuscated C Contest for much, much worse examples

- A quadratic equation for variable x is one of the form:

    $ax^2 + bx + c = 0$, for some numbers `a, b,` and `c`.

- The two roots of a quadratic equation can be found using the quadratic formula at right.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Example: The roots of `x²-3x-4=0` are `x=4` and `x=-1`

- How would we write a function named quadratic to solve quadratic equations?
  - What parameters should it accept?
    - Which parameters should be passed by value, and which by reference?
  - What, if anything, should it return?

- We have choices!

```
/*
 * Solves a quadratic equation ax^2 + bx + c = 0,
 * storing the results in output parameters root1 and root2.
 * Assumes that the given equation has two real roots.
 */
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```

$$\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- How are we "returning" the results?   Answer: by reference
- What other choices could we have made? Talk to your neighbor!

- Possible choices:
  - We could have returned a boolean if the roots were imaginary
  - We could have added extra parameters to support some form of imaginary numbers
  - We could have called an error function inside this function (but that is not always a good idea -- functions like this should generally have an interface through the parameters and/or return value, and should gracefully fail)
  - We could have re-written the function as two functions that return either the positive or negative root, without using references.
  - We could have returned a Vector<double> object (tricky syntax!)

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Recap

- Fauxtoshop is out — start early!
- There are plenty of opportunities to get help before the deadline next Friday!
- Functions are to C++ as methods are to Java (and very, very similar)
- The Qt Creator debugger can show you real-time details of what your program is doing, and it will come in super handy when you are trying to find tricky bugs in your code.
- You must declare function prototypes before using them in C++!
- There are lots of pre-written functions (e.g., <cmath> and the Stanford Library functions) that have been written already. Use them!
- Value semantics: pass by "value" means that you get a copy of a variable, *not the original!*
- Reference semantics: using the **&** in a parameter definition will give the function access to *the original variable*. This can be tricky until you get used to it.

References and Advanced Reading

- **References (in general, not the C++ references!):**
  - Textbook Chapters 2 and 3
  - <cmath> functions: http://en.cppreference.com/w/cpp/header/cmath
  - Obfuscated C contest: http://www.ioccc.org
  - Code from class: see class website (https://cs106b.stanford.edu)

- **Advanced Reading:**
  - Wikipedia article on C++ References: https://en.wikipedia.org/wiki/Reference_(C%2B%2B)
  - More information on C++ references: http://www.learncpp.com/cpp-tutorial/611-references/
  - C++ Newton's Method question on StackOverflow: http://codereview.stackexchange.com/questions/43456/square-root-approximation-with-newtons-method
  - If you are super-brave, look at the square root C++ function in the C library: http://osxr.org:8080/glibc/source/sysdeps/ieee754/dbl-64/e_sqrt.c?v=glibc-2.14#0048