

Functions

How to reuse code

Massoud Hamad

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

int main() {
    int threeExpFour = 1;
    for (int i = 0; i < 4; i = i + 1) {
        threeExpFour = threeExpFour * 3;
    }
    cout << "3^4 is " << threeExpFour << endl;
    return 0;
}
```

Copy-paste coding

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

int main() {
    int threeExpFour = 1;
    for (int i = 0; i < 4; i = i + 1) {
        threeExpFour = threeExpFour * 3;
    }
    cout << "3^4 is " << threeExpFour << endl;
    int sixExpFive = 1;
    for (int i = 0; i < 5; i = i + 1) {
        sixExpFive = sixExpFive * 6;
    }
    cout << "6^5 is " << sixExpFive << endl;
    return 0;
}
```

Copy-paste coding (bad)

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

int main() {
    int threeExpFour = 1;
    for (int i = 0; i < 4; i = i + 1) {
        threeExpFour = threeExpFour * 3;
    }
    cout << "3^4 is " << threeExpFour << endl;
    int sixExpFive = 1;
    for (int i = 0; i < 5; i = i + 1) {
        sixExpFive = sixExpFive * 6;
    }
    cout << "6^5 is " << sixExpFive << endl;
    int twelveExpTen = 1;
    for (int i = 0; i < 10; i = i + 1) {
        twelveExpTen = twelveExpTen * 12;
    }
    cout << "12^10 is " << twelveExpTen << endl;
    return 0;
}
```

With a function

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
    int threeExpFour = raiseToPower(3, 4);
    cout << "3^4 is " << threeExpFour << endl;
    return 0;
}
```

With a function

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
    int threeExpFour = raiseToPower(3, 4);
    cout << "3^4 is " << threeExpFour << endl;
    int sixExpFive = raiseToPower(6, 5);
    cout << "6^5 is " << sixExpFive << endl;
    return 0;
}
```

With a function

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {
    int threeExpFour = raiseToPower(3, 4);
    cout << "3^4 is " << threeExpFour << endl;
    int sixExpFive = raiseToPower(6, 5);
    cout << "6^5 is " << sixExpFive << endl;
    int twelveExpTen = raiseToPower(12, 10);
    cout << "12^10 is " << twelveExpTen << endl;
    return 0;
}
```

Why define your own functions?

- Readability: `sqrt(5)` is clearer than copy-pasting in an algorithm to compute the square root
- Maintainability: To change the algorithm, just change the function (vs changing it everywhere you ever used it)
- Code reuse: Lets other people use algorithms you've implemented

Function Declaration Syntax

Function name

```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

Function Declaration Syntax

Return type

```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

Function Declaration Syntax

Argument 1

```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

- Argument order matters:
 - `raiseToPower(2,3)` is $2^3=8$
 - `raiseToPower(3,2)` is $3^2=9$

Function Declaration Syntax

Argument 2

```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

- Argument order matters:
 - `raiseToPower(2,3)` is $2^3=8$
 - `raiseToPower(3,2)` is $3^2=9$

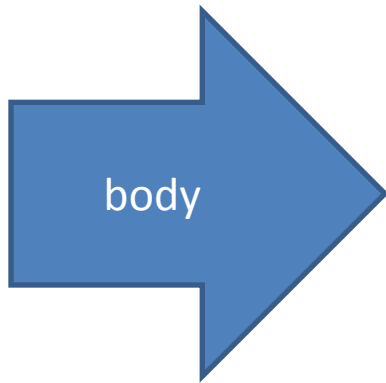
Function Declaration Syntax



signature

```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

Function Declaration Syntax



```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```

Function Declaration Syntax

```
int raiseToPower(int base, int exponent)
{
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```



Return statement



Function
declaration

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

```
int raiseToPower(int base, int exponent) {
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    return result;
}
```



Function invocation

```
int main() {
    int threeExpFour = raiseToPower(3, 4);
    cout << "3^4 is " << threeExpFour << endl;
    return 0;
}
```


Returning a value

- Up to one value may be returned; it must be the same type as the return type.

```
int foo()  
{  
    return "hello"; // error  
}
```

```
char* foo()  
{  
    return "hello"; // ok  
}
```

Returning a value

- Up to one value may be returned; it must be the same type as the return type.
- If no values are returned, give the function a **void** return type

```
void printNumber(int num) {  
    cout << "number is " << num << endl;  
}
```

```
int main() {  
    printNumber(4); // number is 4  
    return 0;  
}
```

Returning a value

- Up to one value may be returned; it must be the same type as the return type.
- If no values are returned, give the function a **void** return type
 - Note that you cannot declare a variable of type void

```
int main() {  
    void x; // ERROR  
    return 0;  
}
```

Returning a value

- Return statements don't necessarily need to be at the end.
- Function returns as soon as a return statement is executed.

```
void printNumberIfEven(int num) {  
    if (num % 2 == 1) {  
        cout << "odd number" << endl;  
        return;  
    }  
    cout << "even number; number is " << num << endl;  
}
```

```
int main() {  
    int x = 4;  
    printNumberIfEven(x);  
    // even number; number is 3  
    int y = 5;  
    printNumberIfEven(y);  
    // odd number  
}
```

Argument Type Matters

```
void printOnNewLine(int x)
{
    cout << x << endl;
}
```

- `printOnNewLine(3)` works
- `printOnNewLine("hello")` will not compile

Argument Type Matters

```
void printOnNewLine(char *x)
{
    cout << x << endl;
}
```

- `printOnNewLine(3)` will not compile
- `printOnNewLine("hello")` works

Argument Type Matters

```
void printOnNewLine(int x)
{
    cout << x << endl;
}
```

```
void printOnNewLine(char *x)
{
    cout << x << endl;
}
```

- `printOnNewLine(3)` works
- `printOnNewLine("hello")` also works

Function Overloading

```
void printOnNewLine(int x)
{
    cout << "Integer: " << x << endl;
}

void printOnNewLine(char *x)
{
    cout << "String: " << x << endl;
}
```

- Many functions with the same name, but different arguments
- The function called is the one whose arguments match the invocation

Function Overloading

```
void printOnNewLine(int x)
{
    cout << "Integer: " << x << endl;
}

void printOnNewLine(char *x)
{
    cout << "String: " << x << endl;
}
```

- `printOnNewLine(3)` prints “Integer: 3”
- `printOnNewLine(“hello”)` prints “String: hello”

Function Overloading

```
void printOnNewLine(int x)
{
    cout << "1 Integer: " << x << endl;
}

void printOnNewLine(int x, int y)
{
    cout << "2 Integers: " << x << " and " << y << endl;
}
```

- `printOnNewLine(3)` prints "1 Integer: 3"
- `printOnNewLine(2, 3)` prints "2 Integers: 2 and 3"

- Function declarations need to occur before invocations

```
int foo()  
{  
    return bar()*2; // ERROR - bar hasn't been declared yet  
}
```

```
int bar()  
{  
    return 3;  
}
```

- Function declarations need to occur before invocations
 - Solution 1: reorder function declarations

```
int bar()  
{  
    return 3;  
}
```

```
int foo()  
{  
    return bar()*2; // ok  
}
```

- Function declarations need to occur before invocations
 - Solution 1: reorder function declarations
 - Solution 2: use a function prototype; informs the compiler you'll implement it later

```
int bar();
```



function prototype

```
int foo()  
{  
    return bar()*2; // ok  
}
```

```
int bar()  
{  
    return 3;  
}
```

- Function prototypes should match the signature of the method, though argument names don't matter

```
int square(int);
```



```
int cube(int x)
{
    return x*square(x);
}
```

```
int square(int x)
{
    return x*x;
}
```

- Function prototypes should match the signature of the method, though argument names don't matter

```
int square(int x);
```



```
int cube(int x)
{
    return x*square(x);
}
```

```
int square(int x)
{
    return x*x;
}
```

- Function prototypes should match the signature of the method, though argument names don't matter

```
int square(int z);
```



```
int cube(int x)
{
    return x*square(x);
}
```

```
int square(int x)
{
    return x*x;
}
```


- Function prototypes are generally put into separate header files
 - Separates specification of the function from its implementation

```
// myLib.h - header  
// contains prototypes
```

```
int square(int);  
int cube (int);
```

```
// myLib.cpp - implementation  
#include "myLib.h"
```

```
int cube(int x)  
{  
    return x*square(x);  
}
```

```
int square(int x)  
{  
    return x*x;  
}
```

Recursion

- Functions can call themselves.
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ can be easily expressed via a recursive implementation

```
int fibonacci(int n) {  
    if (n == 0 || n == 1) {  
        return 1;  
    } else {  
        return fibonacci(n-2) + fibonacci(n-1);  
    }  
}
```

Recursion

- Functions can call themselves.
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ can be easily expressed via a recursive implementation



```
int fibonacci(int n) {  
    if (n == 0 || n == 1) {  
        return 1;  
    } else {  
        return fibonacci(n-2) + fibonacci(n-1);  
    }  
}
```

Recursion

- Functions can call themselves.
- $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ can be easily expressed via a recursive implementation



recursive step

```
int fibonacci(int n) {  
    if (n == 0 || n == 1) {  
        return 1;  
    } else {  
        return fibonacci(n-2) + fibonacci(n-1);  
    }  
}
```

Global Variables

- How many times is function `foo()` called? Use a global variable to determine this.
 - Can be accessed from any function

```
int numCalls = 0;
```



Global variable

```
void foo() {  
    ++numCalls;  
}
```

```
int main() {  
    foo(); foo(); foo();  
    cout << numCalls << endl; // 3  
}
```

Scope

```
int numCalls = 0;
```

```
int raiseToPower(int base, int exponent) {  
    numCalls = numCalls + 1;  
    int result = 1;  
    for (int i = 0; i < exponent; i = i + 1) {  
        result = result * base;  
    }  
    return result;  
}
```

```
int max(int num1, int num2) {  
    numCalls = numCalls + 1;  
    int result;  
    if (num1 > num2) {  
        result = num1;  
    }  
    else {  
        result = num2;  
    }  
    return result;  
}
```

- Scope: where a variable was declared, determines where it can be accessed from

Scope

```
int numCalls = 0;
```

```
int raiseToPower(int base, int exponent) {  
    numCalls = numCalls + 1;  
    int result = 1;  
    for (int i = 0; i < exponent; i = i + 1) {  
        result = result * base;  
    }  
    return result;  
}
```

- Scope: where a variable was declared, determines where it can be accessed from

```
int max(int num1, int num2) {  
    numCalls = numCalls + 1;  
    int result;  
    if (num1 > num2) {  
        result = num1;  
    }  
    else {  
        result = num2;  
    }  
    return result;  
}
```

- numCalls has global scope – can be accessed from any function

Scope

```
int numCalls = 0;
```

```
int raiseToPower(int base, int exponent) {  
    numCalls = numCalls + 1;  
    int result = 1;  
    for (int i = 0; i < exponent; i = i + 1) {  
        result = result * base;  
    }  
    return result;  
}
```

```
int max(int num1, int num2) {  
    numCalls = numCalls + 1;  
    int result;  
    if (num1 > num2) {  
        result = num1;  
    }  
    else {  
        result = num2;  
    }  
    return result;  
}
```

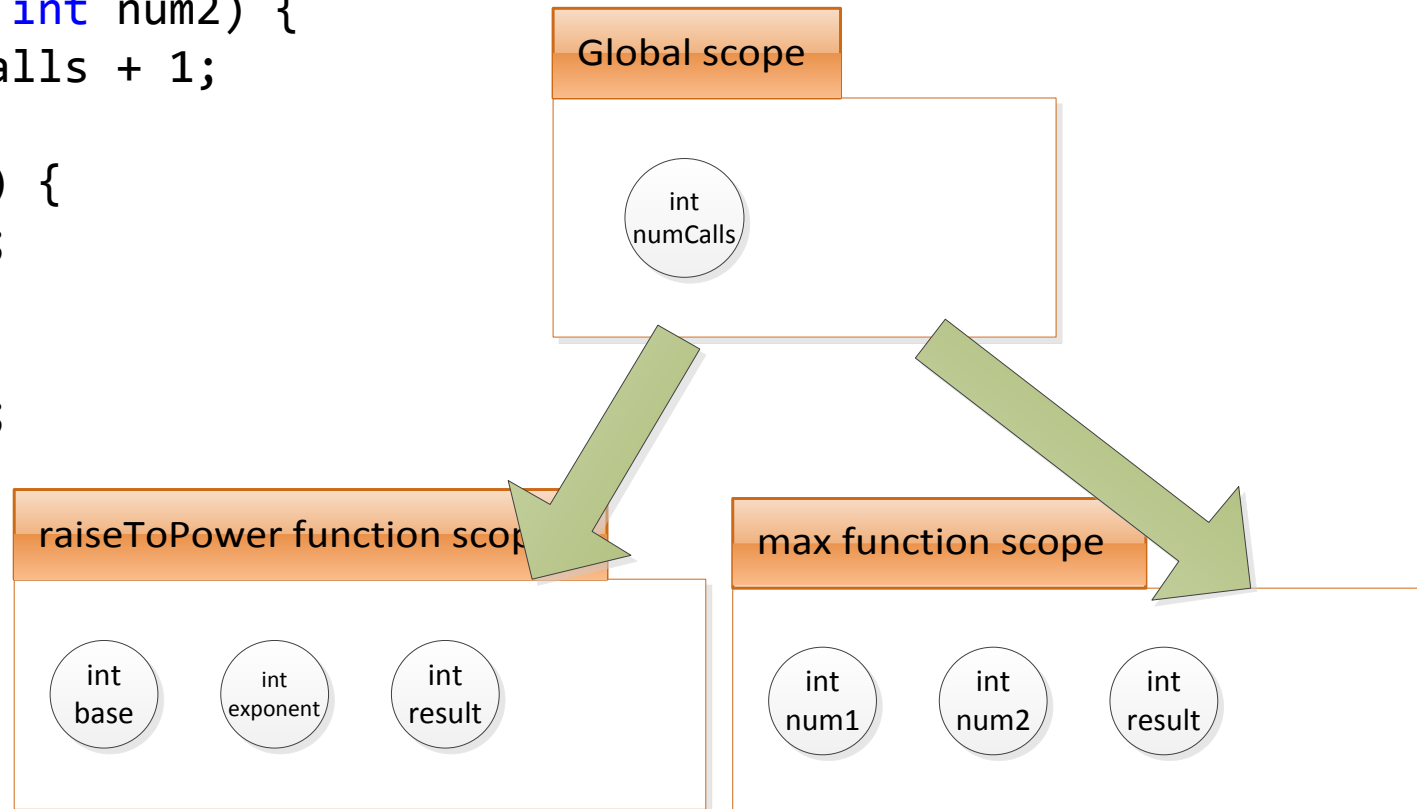
- Scope: where a variable was declared, determines where it can be accessed from
- numCalls has global scope – can be accessed from any function
- result has function scope – each function can have its own separate variable named result


```

int numCalls = 0;
int raiseToPower(int base, int exponent) {
    numCalls = numCalls + 1;
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    // A
    return result;
}

int max(int num1, int num2) {
    numCalls = numCalls + 1;
    int result;
    if (num1 > num2) {
        result = num1;
    }
    else {
        result = num2;
    }
    // B
    return result;
}

```

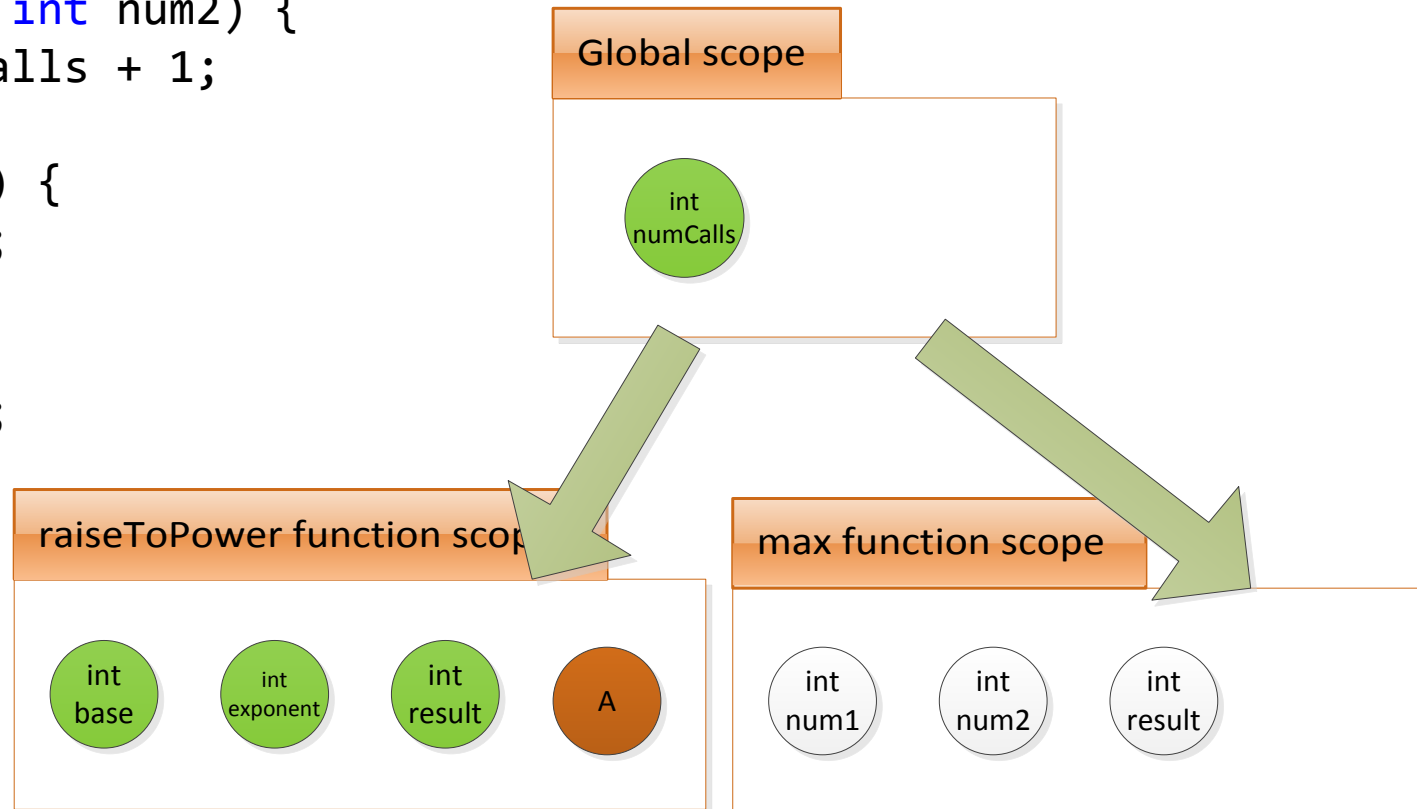


```

int numCalls = 0;
int raiseToPower(int base, int exponent) {
    numCalls = numCalls + 1;
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    // A
    return result;
}
int max(int num1, int num2) {
    numCalls = numCalls + 1;
    int result;
    if (num1 > num2) {
        result = num1;
    }
    else {
        result = num2;
    }
    // B
    return result;
}

```

- At **A**, variables marked in green are in scope

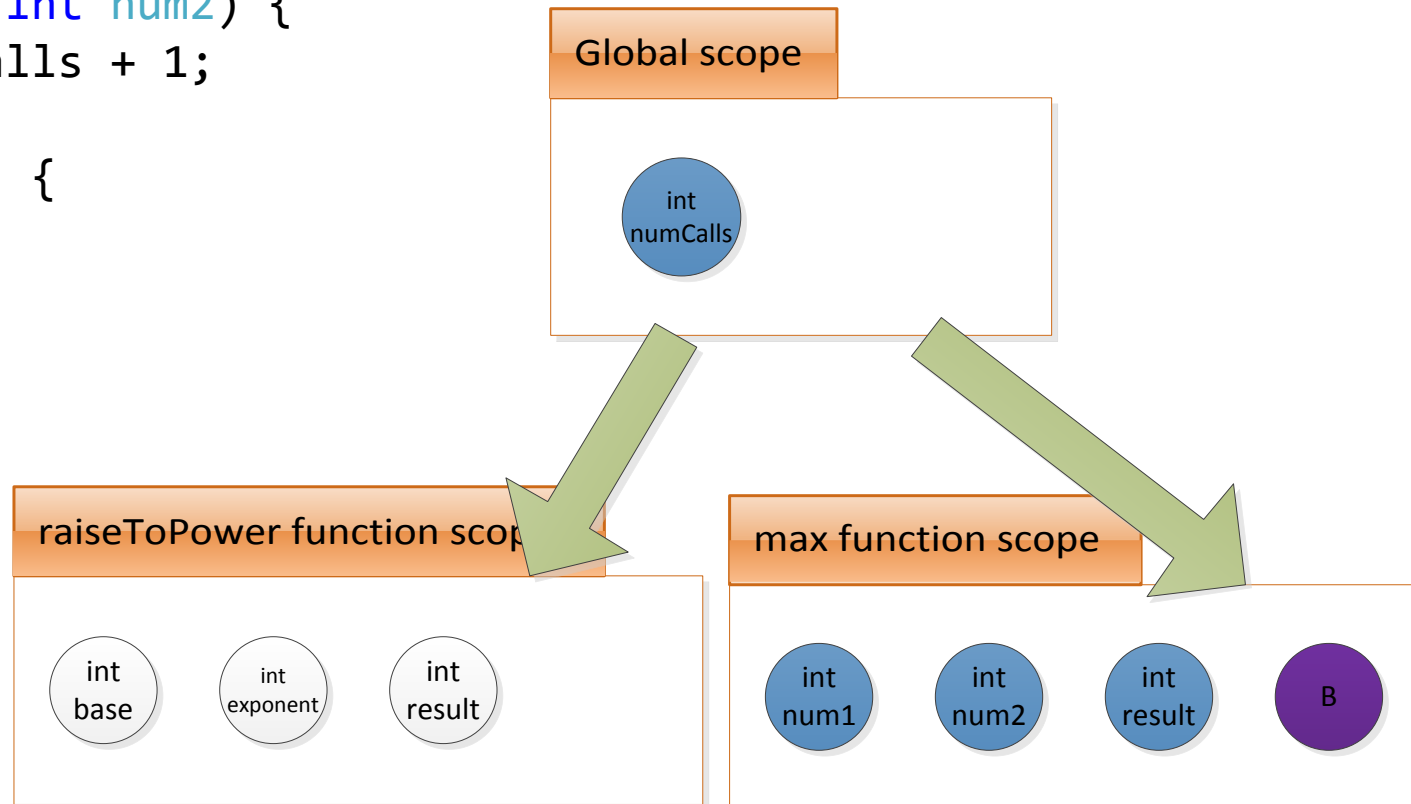


```

int numCalls = 0;
int raiseToPower(int base, int exponent) {
    numCalls = numCalls + 1;
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1) {
        result = result * base;
    }
    // A
    return result;
}
int max(int num1, int num2) {
    numCalls = numCalls + 1;
    int result;
    if (num1 > num2) {
        result = num1;
    }
    else {
        result = num2;
    }
    // B
    return result;
}

```

- At **B**, variables marked in blue are in scope

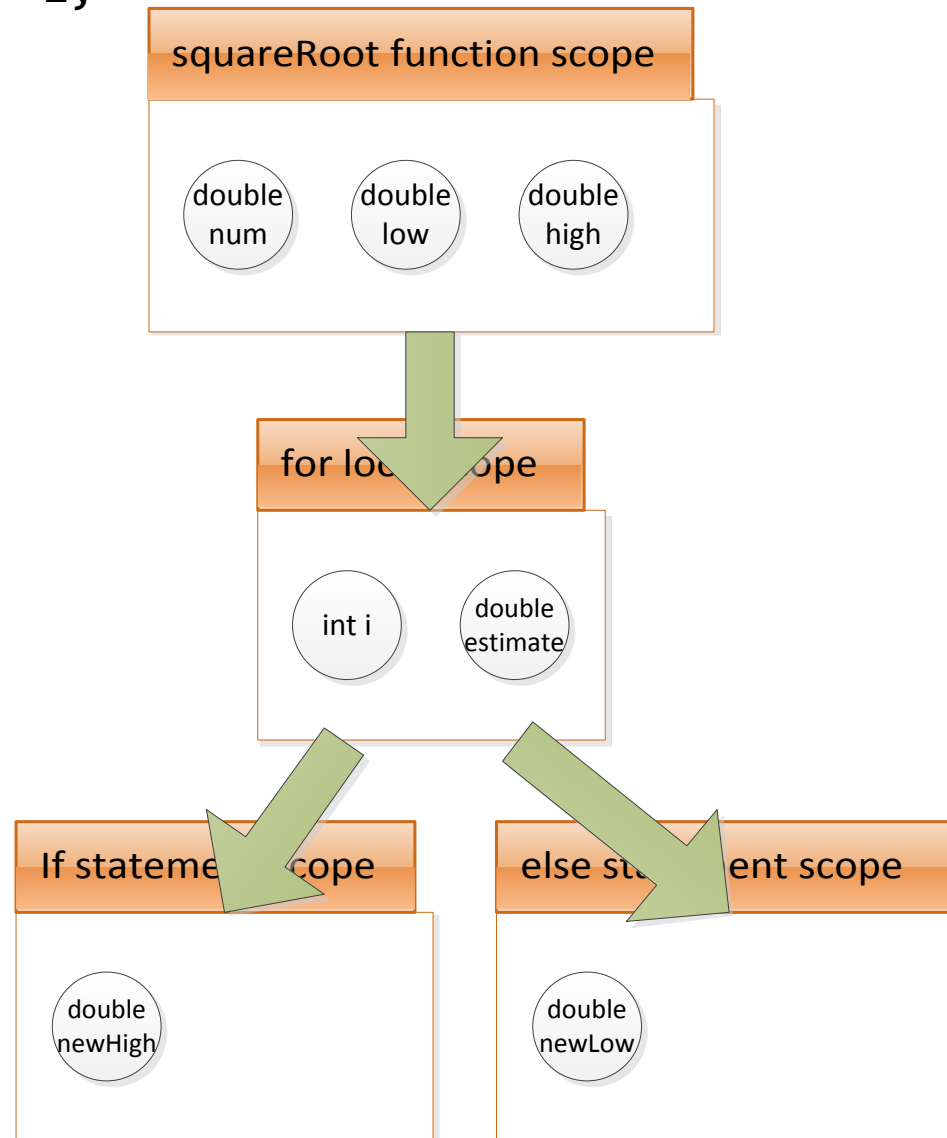


```

double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
    }
    return (high + low) / 2;
}

```

- Loops and if/else statements also have their own scopes
 - Loop counters are in the same scope as the body of the for loop

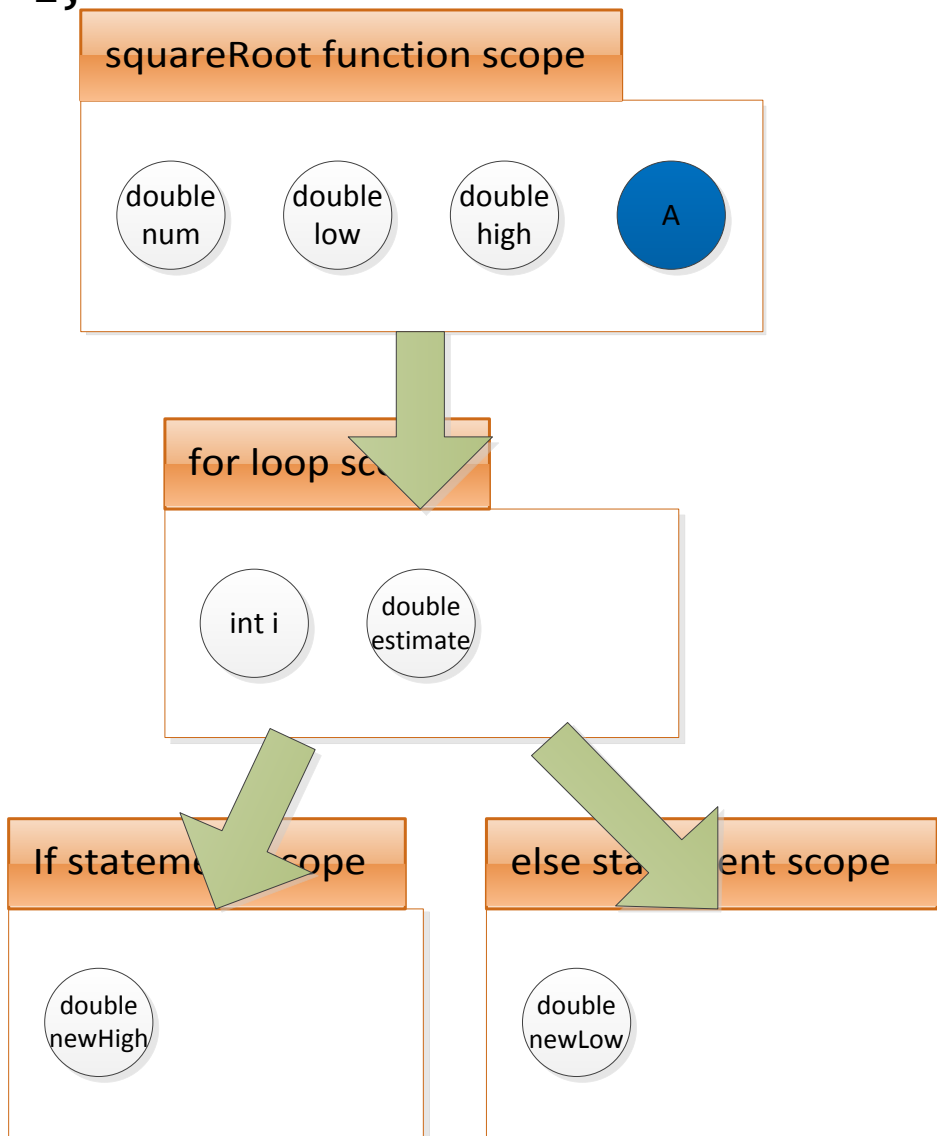


```

double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
    }
    // A
    return estimate; // ERROR
}

```

- Cannot access variables that are out of scope

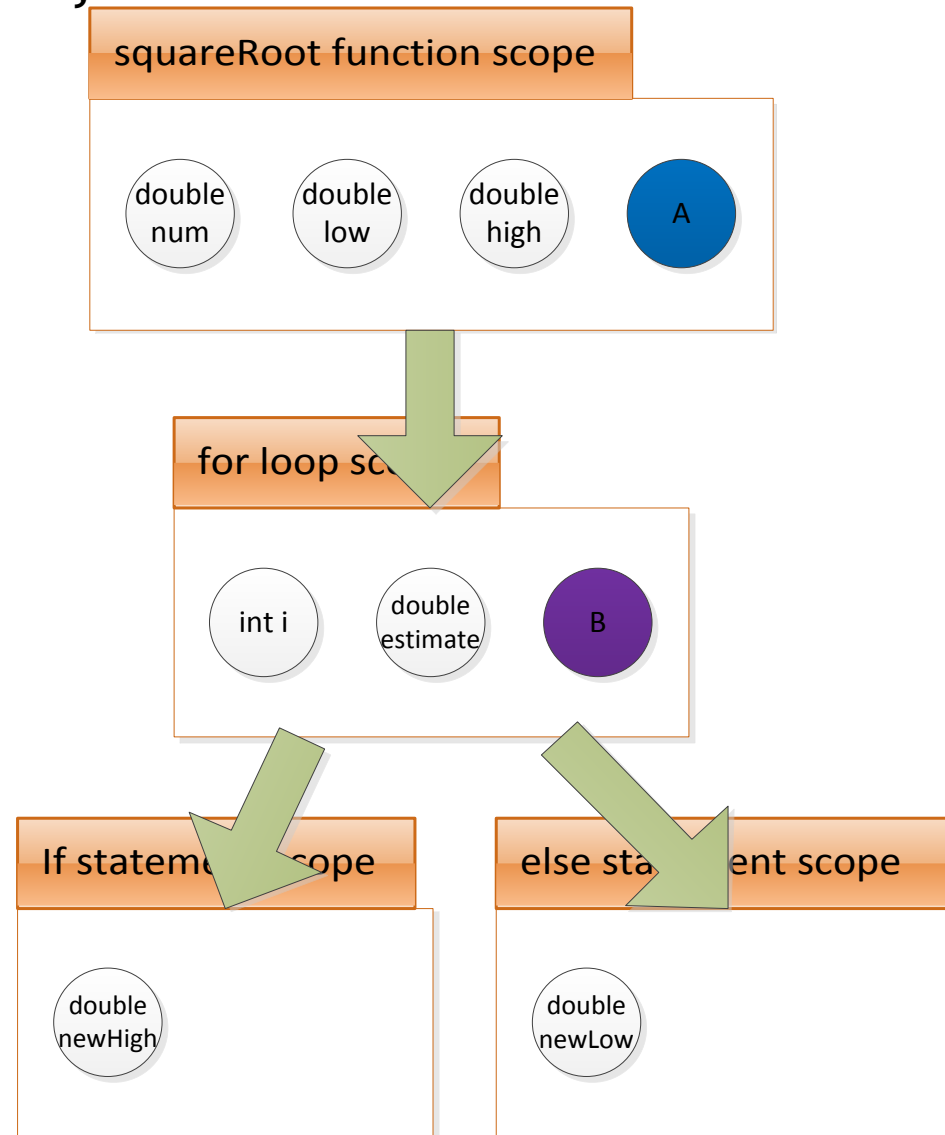


```

double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    for (int i = 0; i < 30; i = i + 1) {
        double estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
        if (i == 29)
            return estimate; // B
    }
    return -1; // A
}

```

- Cannot access variables that are out of scope
- Solution 1: move the code

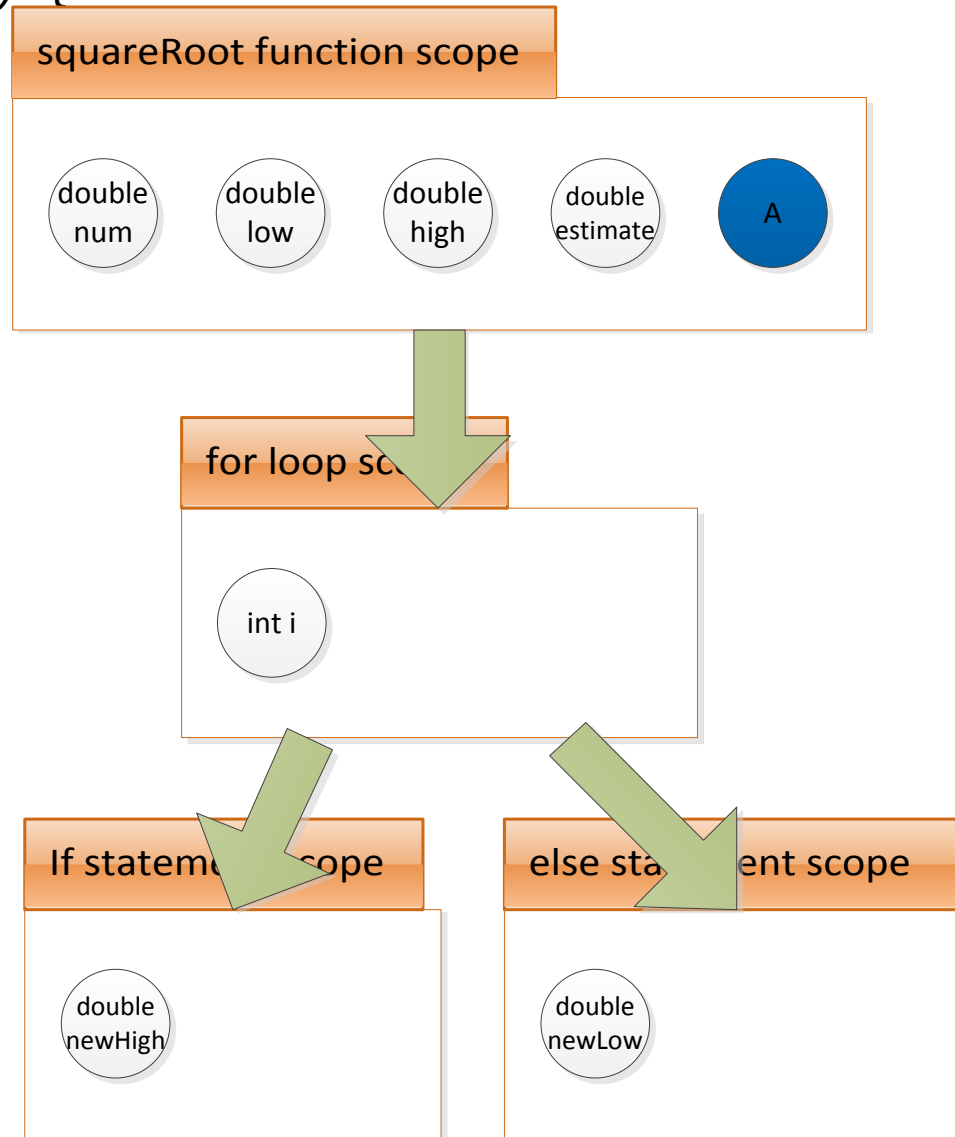


```

double squareRoot(double num) {
    double low = 1.0;
    double high = num;
    double estimate;
    for (int i = 0; i < 30; i = i + 1) {
        estimate = (high + low) / 2;
        if (estimate*estimate > num) {
            double newHigh = estimate;
            high = newHigh;
        } else {
            double newLow = estimate;
            low = newLow;
        }
    }
    return estimate; // A
}

```

- Cannot access variables that are out of scope
- Solution 2: declare the variable in a higher scope



Pass by value vs by reference

- So far we've been passing everything by value – makes a copy of the variable; changes to the variable within the function don't occur outside the function

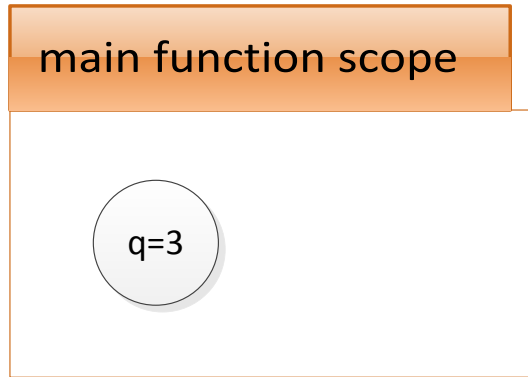
```
// pass-by-value
void increment(int a) {
    a = a + 1;
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3;
    increment(q); // does nothing
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main **3**

Pass by value vs by reference



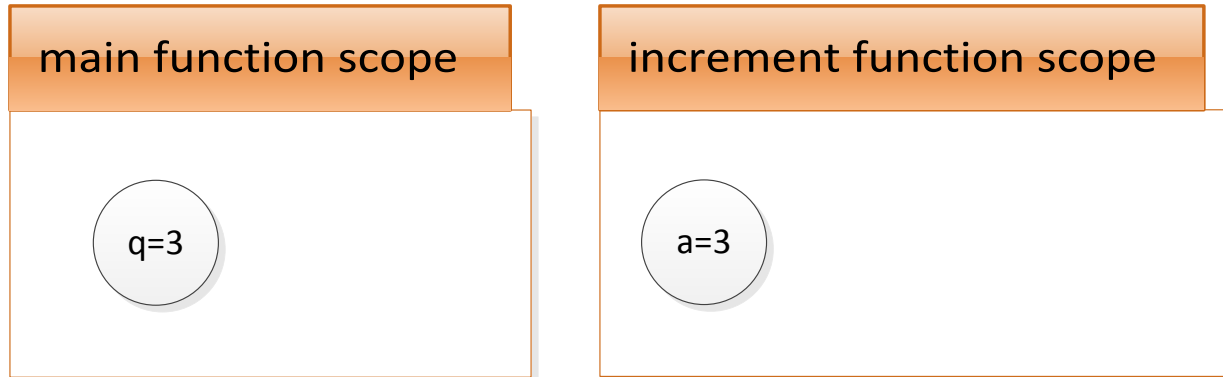
```
// pass-by-value
void increment(int a) {
    a = a + 1;
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3; // HERE
    increment(q); // does nothing
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main **3**

Pass by value vs by reference



// pass-by-value

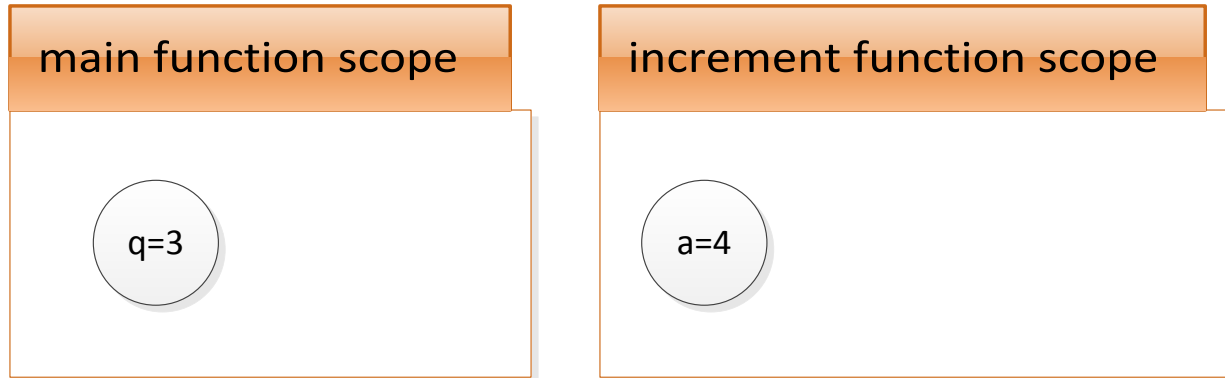
```
void increment(int a) { // HERE
    a = a + 1;
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3;
    increment(q); // does nothing
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main **3**

Pass by value vs by reference



```
// pass-by-value
void increment(int a) {
    a = a + 1; // HERE
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3;
    increment(q); // does nothing
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main **3**

Pass by value vs by reference

- If you want to modify the original variable as opposed to making a copy, pass the variable by reference (**int &a** instead of **int a**)

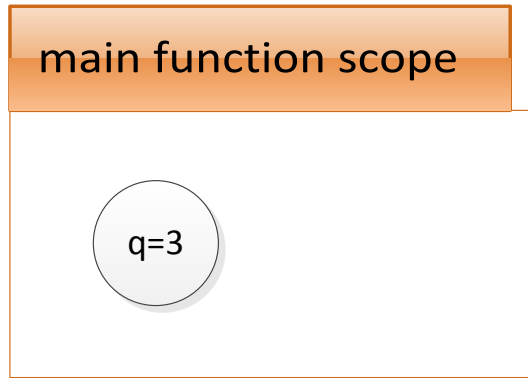
```
// pass-by-value
void increment(int &a) {
    a = a + 1;
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3;
    increment(q); // works
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main 4

Pass by value vs by reference



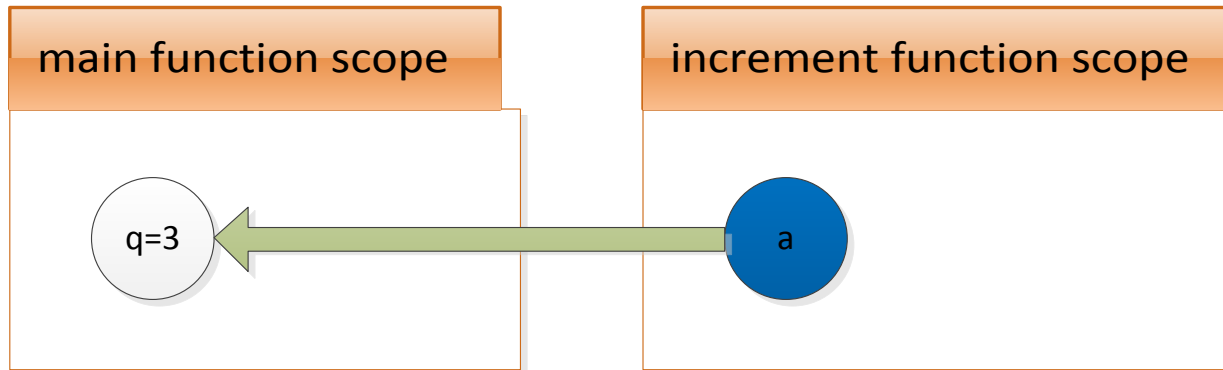
```
// pass-by-value
void increment(int &a) {
    a = a + 1;
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3; // HERE
    increment(q); // works
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main 4

Pass by value vs by reference



// pass-by-value

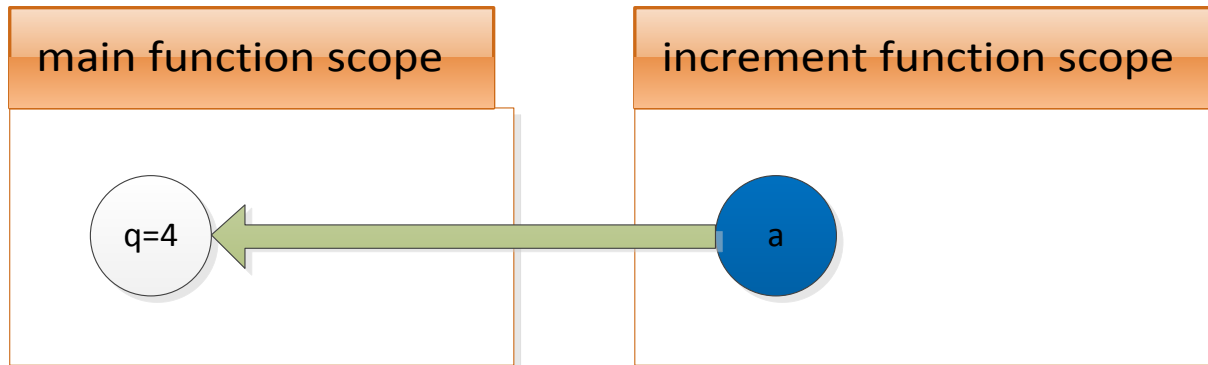
```
void increment(int &a) { // HERE
    a = a + 1;
    cout << "a in increment " << a << endl;
}
```

```
int main() {
    int q = 3;
    increment(q); // works
    cout << "q in main " << q << endl;
}
```

Output

a in increment 4
q in main 4

Pass by value vs by reference



// pass-by-value

```
void increment(int &a) {  
    a = a + 1; // HERE  
    cout << "a in increment " << a << endl;  
}
```

```
int main() {  
    int q = 3;  
    increment(q); // works  
    cout << "q in main " << q << endl;  
}
```

Output

a in increment 4
q in main 4

Implementing Swap

```
void swap(int &a, int &b) {  
    int t = a;  
    a = b;  
    b = t;  
}  
  
int main() {  
    int q = 3;  
    int r = 5;  
    swap(q, r);  
    cout << "q " << q << endl; // q 5  
    cout << "r " << r << endl; // r 3  
}
```


Implementing Swap

```
void swap(int &a, int &b) {  
    int t = a;  
    a = b;  
    b = t;  
}
```

```
int main() {  
    int q = 3;  
    int r = 5; // HERE  
    swap(q, r);  
    cout << "q " << q << endl; // q 5  
    cout << "r " << r << endl; // r 3  
}
```

main function scope

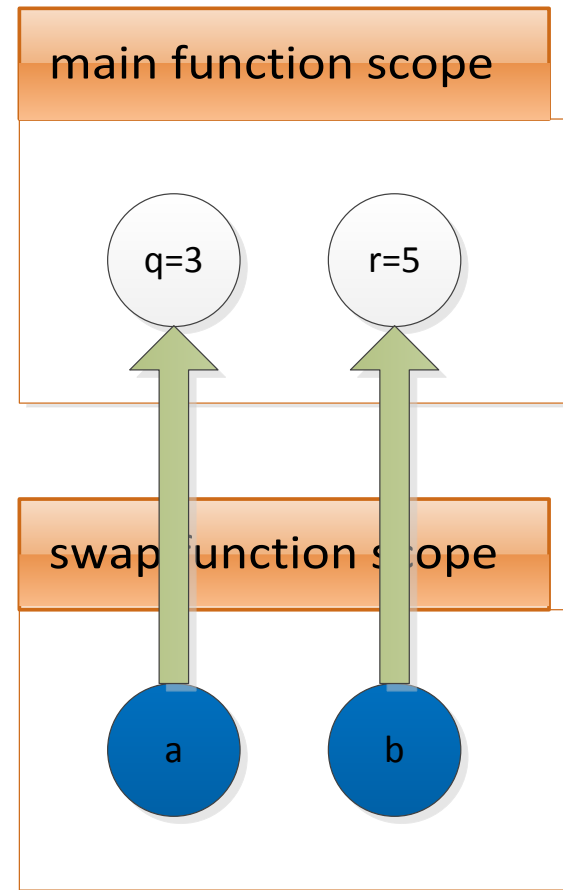
q=3

r=5

Implementing Swap

```
void swap(int &a, int &b) { // HERE
    int t = a;
    a = b;
    b = t;
}
```

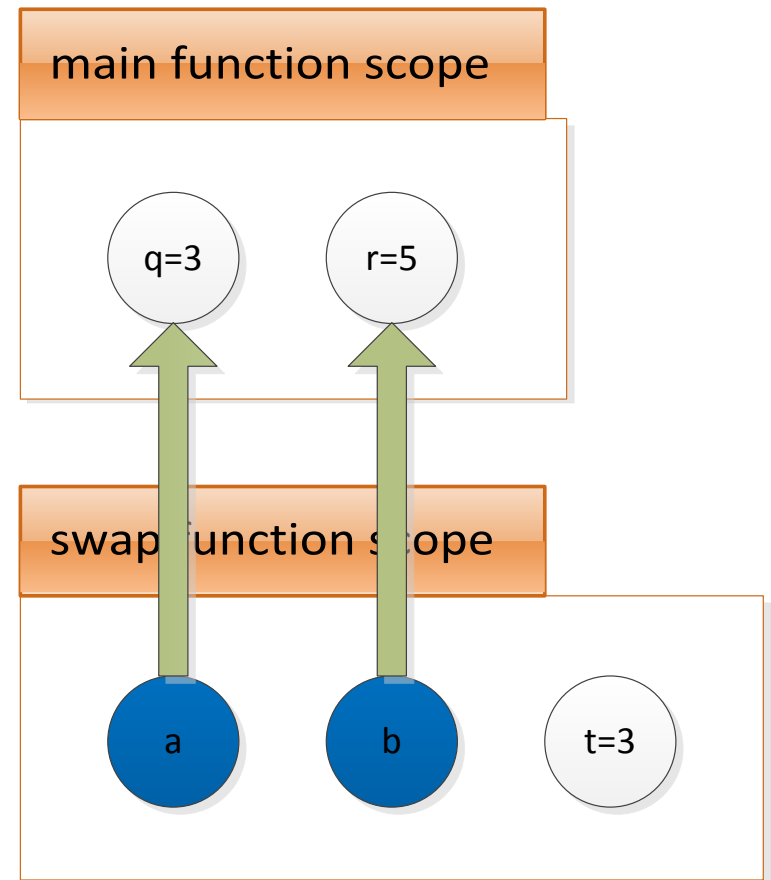
```
int main() {
    int q = 3;
    int r = 5;
    swap(q, r);
    cout << "q " << q << endl; // q 5
    cout << "r " << r << endl; // r 3
}
```



Implementing Swap

```
void swap(int &a, int &b) {  
    int t = a; // HERE  
    a = b;  
    b = t;  
}
```

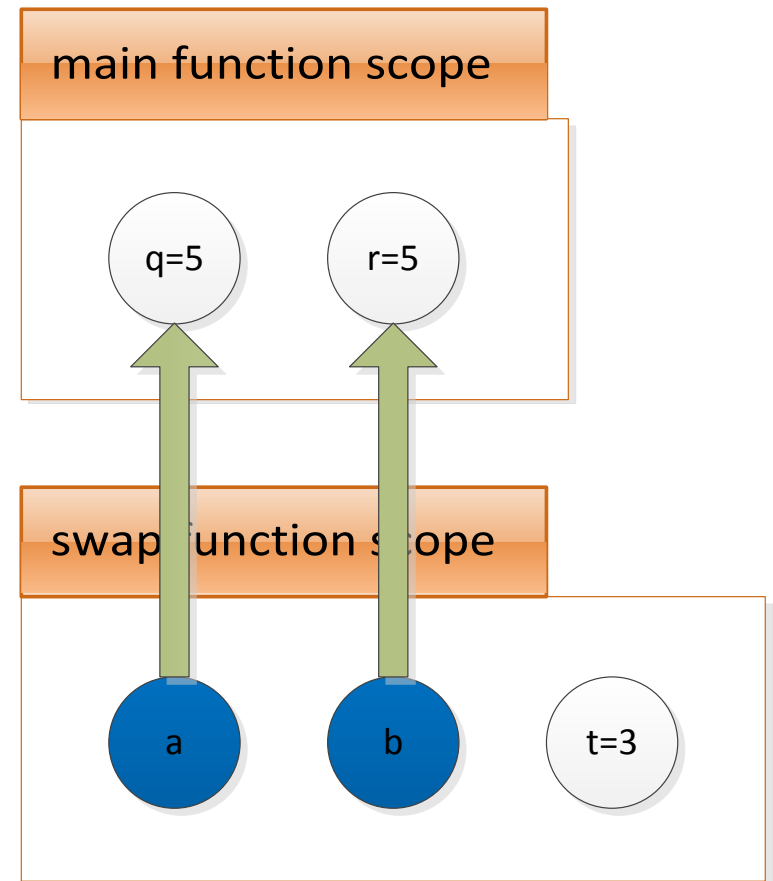
```
int main() {  
    int q = 3;  
    int r = 5;  
    swap(q, r);  
    cout << "q " << q << endl; // q 5  
    cout << "r " << r << endl; // r 3  
}
```



Implementing Swap

```
void swap(int &a, int &b) {  
    int t = a;  
    a = b; // HERE  
    b = t;  
}
```

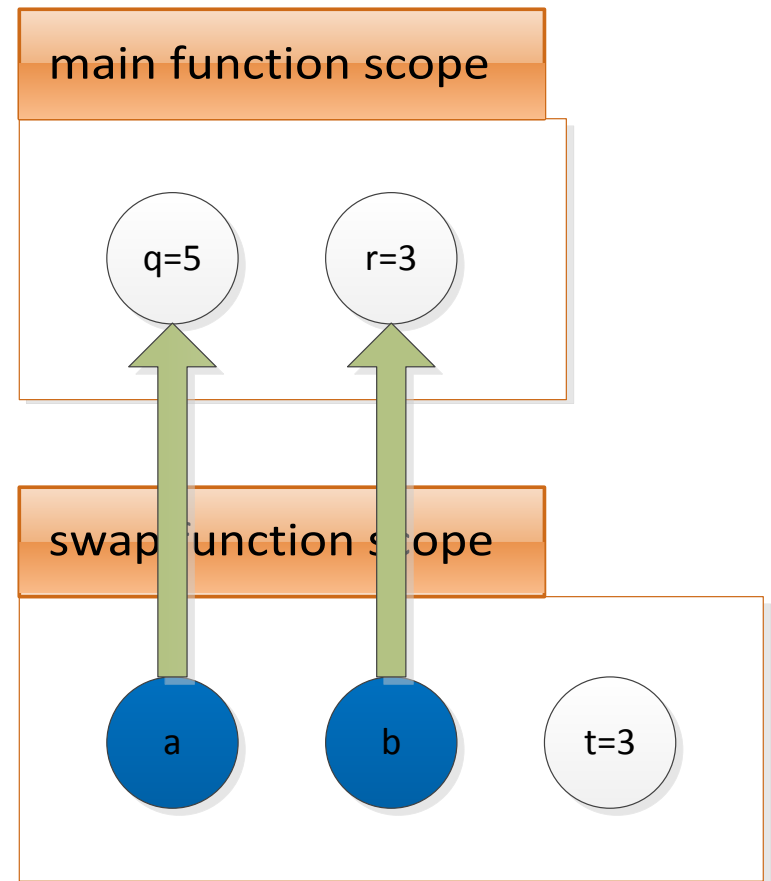
```
int main() {  
    int q = 3;  
    int r = 5;  
    swap(q, r);  
    cout << "q " << q << endl; // q 5  
    cout << "r " << r << endl; // r 3  
}
```



Implementing Swap

```
void swap(int &a, int &b) {  
    int t = a;  
    a = b;  
    b = t; // HERE  
}
```

```
int main() {  
    int q = 3;  
    int r = 5;  
    swap(q, r);  
    cout << "q " << q << endl; // q 5  
    cout << "r " << r << endl; // r 3  
}
```



Returning multiple values

- The return statement only allows you to return 1 value. Passing output variables by reference overcomes this limitation.

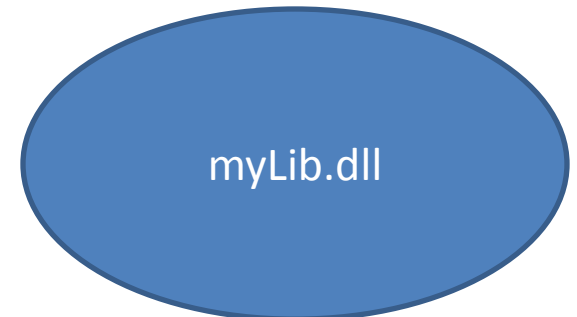
```
int divide(int numerator, int denominator, int &remainder) {  
    remainder = numerator % denominator;  
    return numerator / denominator;  
}
```

```
int main() {  
    int num = 14;  
    int den = 4;  
    int rem;  
    int result = divide(num, den, rem);  
    cout << result << "*" << den << "+" << rem << "=" << num << endl;  
    // 3*4+2=12  
}
```

Libraries

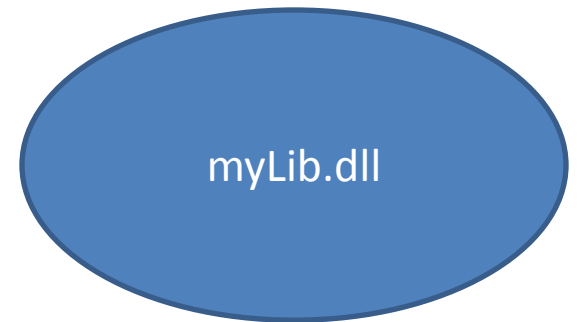
- Libraries are generally distributed as the header file containing the prototypes, and a binary .dll/.so file containing the (compiled) implementation
 - Don't need to share your .cpp code

```
// myLib.h - header  
// contains prototypes  
double squareRoot(double num);
```



- Library user only needs to know the function prototypes (in the header file), not the implementation source code (in the .cpp file)
 - The **Linker** (part of the compiler) takes care of locating the implementation of functions in the .dll file at compile time

```
// myLib.h - header  
// contains prototypes  
double squareRoot(double num);
```



```
// libraryUser.cpp - some other guy's code  
#include "myLib.h"  
  
double fourthRoot(double num) {  
    return squareRoot(squareRoot(num));  
}
```


Final Notes

- You don't actually need to implement `raiseToPower` and `squareRoot` yourself; `cmath` (part of the standard library) contains functions **`pow`** and **`sqrt`**

```
#include <cmath>

double fourthRoot(double num) {
    return sqrt(sqrt(num));
}
```

MIT OpenCourseWare

<http://ocw.mit.edu>

6.096 Introduction to C++

January (IAP) 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.