More Functions; Pass by Reference

ITP 165 – Fall 2015

Week 6, Lecture 2

The “Current” Function

- Only one function can be the “current” function
- All other functions are “paused” until the current function finishes
- How is this tracked?

Papers

- Imagine you have 3 sheets of paper, each with the name of a function on it:

A Desk

- Now imagine you have a desk that you will stack the papers on

- (We will only have one stack of papers on the desk)

Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```

Desk

Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```

main

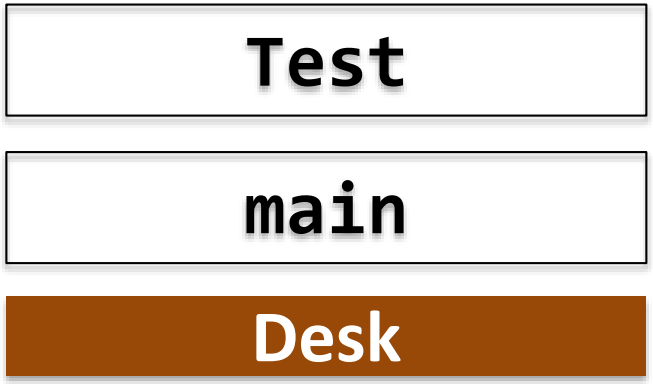
Desk

Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```

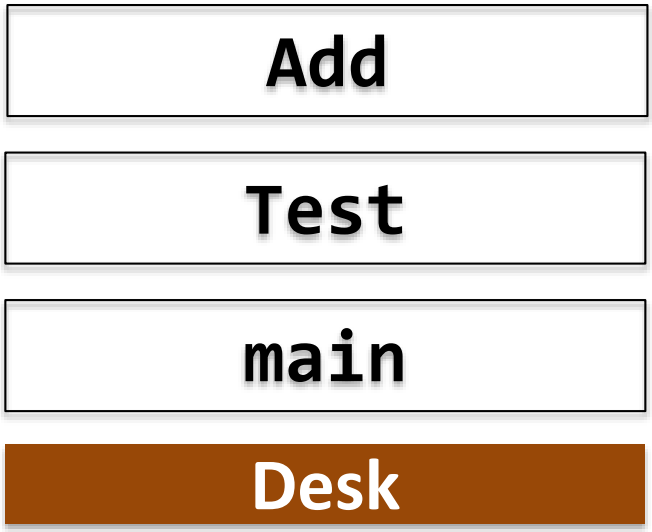


Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```



Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```

Test

main

Desk

Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```

main

Desk

Let's Stack Papers!

```
#include <iostream>
int Add(int x, int y) {
    return x + y;
}

void Test() {
    std::cout << Add(9, 1) << std::endl;
}

int main() {
    Test();
    return 0;
}
```

Desk

Papers Analogy, Summarized

1. The desk is empty right before the program starts
2. When a function starts, its paper gets put on top of the paper stack
3. The "current" function is the paper on top of the stack
4. When a function ends, its paper is removed, and the program continues on the paper below it
5. If there are no papers left on the desk, the program is over

- The papers and desk may seem silly, but conceptually it is *very similar* to how it actually works!

Function Parameter Refresher

- When we talked about function parameters, remember we talked about that by default:
  - Non-arrays are passed by value (aka copy)
  - Arrays are not passed by copy.
- Remember our example:

```
#include <iostream>

void Test(int number) {
   number = 90;
}

int main() {
   int number = 42;
   Test(number);
   std::cout << number << std::endl;
   return 0;
}
```

Problem: Swapping two values

- What if I want to make a function like this:

```
// Function: swap
// Purpose: Swaps the value of two integers.
// Parameters: The two integers to swap.
// Returns: Nothing
```

- This wouldn't work, because changes to x/y don't persist:

```
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
```

Solution: Pass by Reference

- If we pass a parameter ***by reference***, modifications to the parameter ***will*** persist.

- To pass a parameter by reference, in the function declaration add a & between the type and variable name:

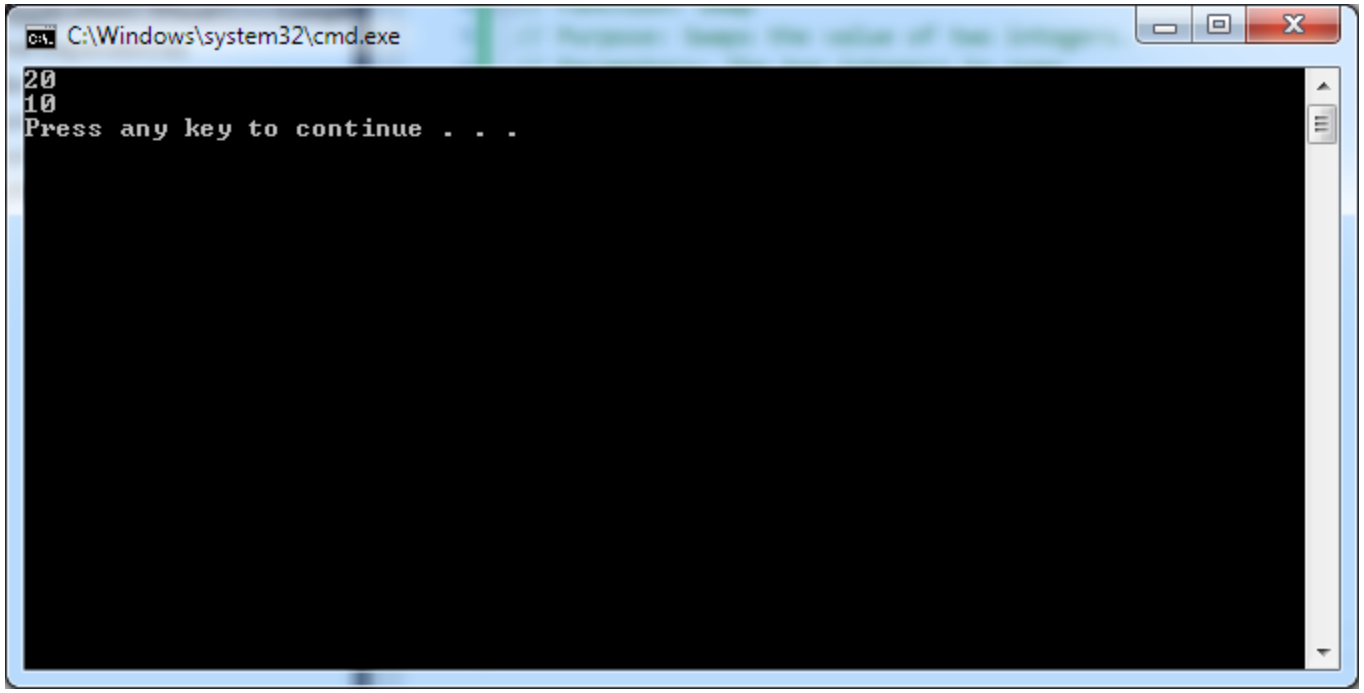
```
// Pass both x and y by reference
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
```

Swap in Action

```
int main() {
    int a = 10;
    int b = 20;
    swap(a, b);

    std::cout << a << std::endl;
    std::cout << b << std::endl;
    return 0;
}
```

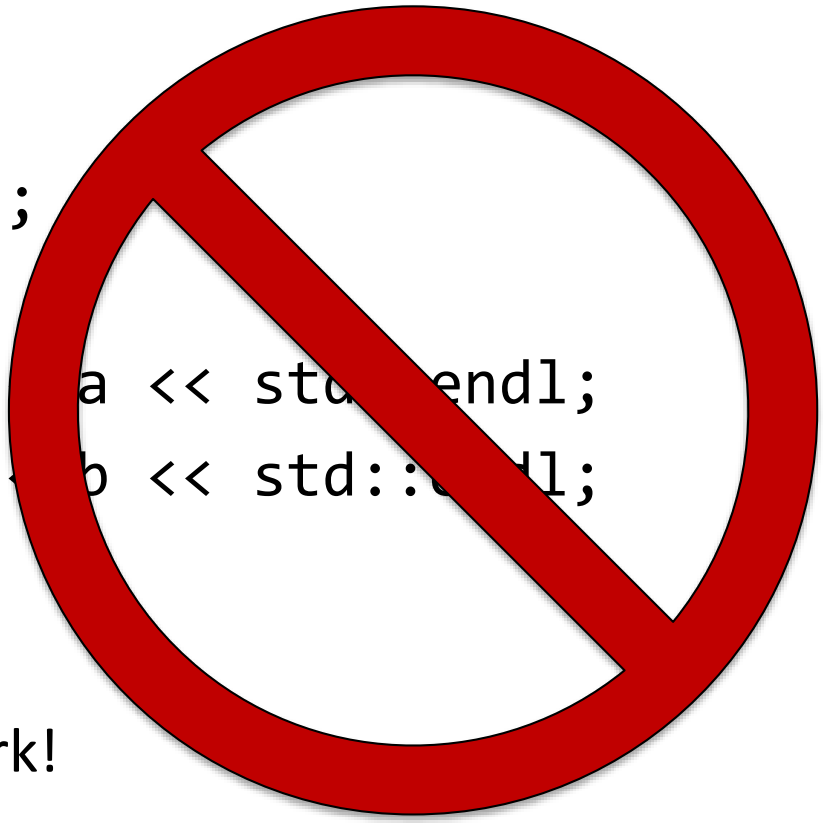
Swap in Action



Note: Pass by Reference

```
int main() {
    int a = 10;
    int b = 20;
    swap(a, 20);

    std::cout <<  a << std endl;
    std::cout <<  b << std::  l;
    return 0;
}
```



- This will not work!

Note: Pass by Reference

- When calling a function with ***pass-by-reference*** parameters, the parameters ***MUST BE*** variables

- If you pass in a literal number (e.g. 10), C++ will recognize it as a **constant**

- Constants cannot be updated or modified

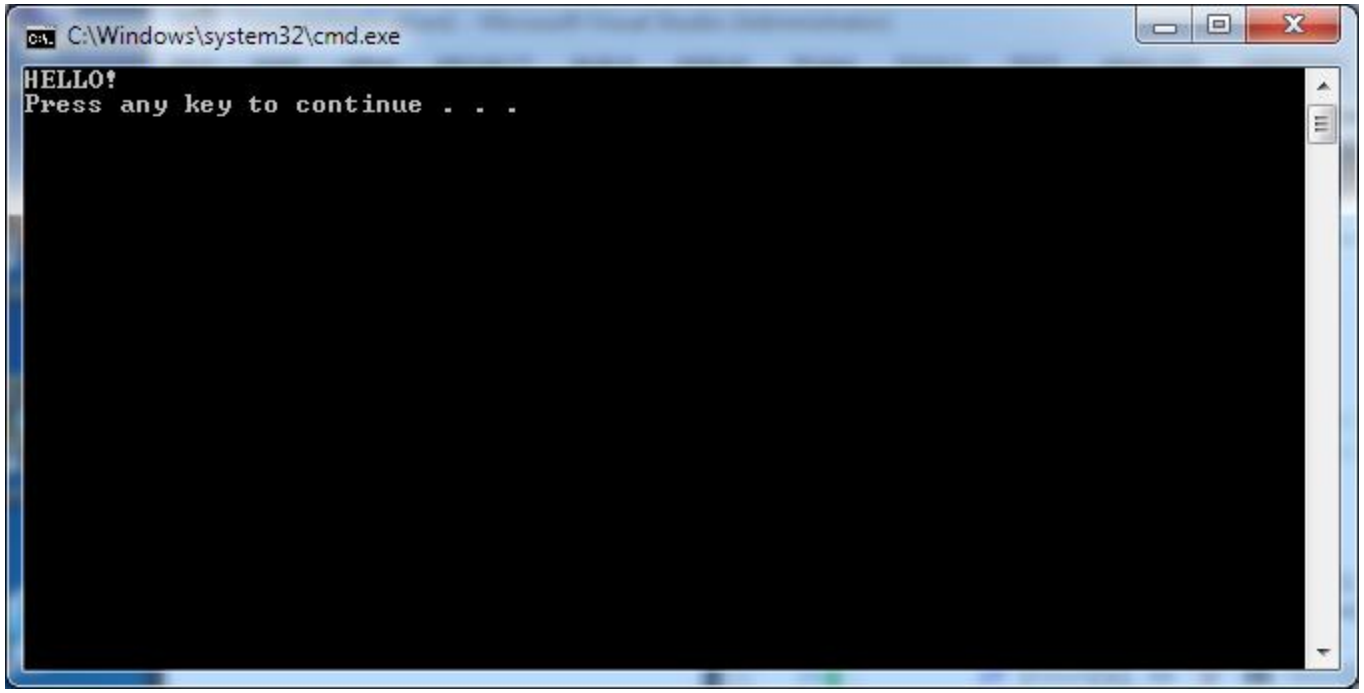
Another Example

- Remember that since arrays are not passed by copy, the following code works:

```
void ToUpper(char text[]) {
   int length = std::strlen(text);
   for (int i = 0; i < length; i++) {
      // It's a lowercase letter
      if (text[i] >= 'a' && text[i] <= 'z') {
         text[i] -= 32;
      }
   }
}


int main() {
   char test[] = "hello!";
   ToUpper(test);
   std::cout << test << std::endl;
   return 0;
}
```

Another Example, Cont'd



Another Example, Cont'd

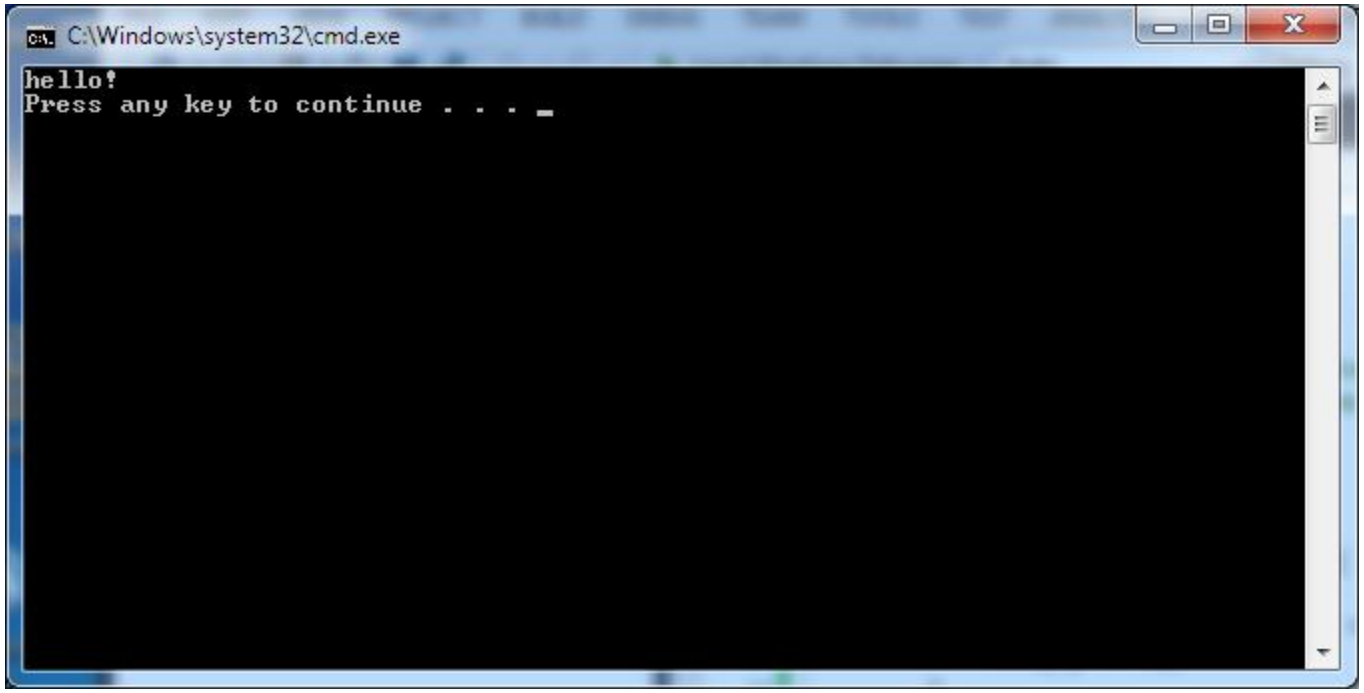
- If we just directly swap out the C-style string for a std::string, it won't work properly anymore ☹

```
void ToUpper(std::string text) {
   int length = text.length();
   for (int i = 0; i < length; i++) {
      // It's a lowercase letter
      if (text[i] >= 'a' && text[i] <= 'z') {
         text[i] -= 32;
      }
   }
}


int main() {
   std::string test = "hello!";
   ToUpper(test);
   std::cout << test << std::endl;
   return 0;
}
```

Another Example, Cont'd

- It doesn't work because std::string is passed by value (by default)



Another Example, Cont'd

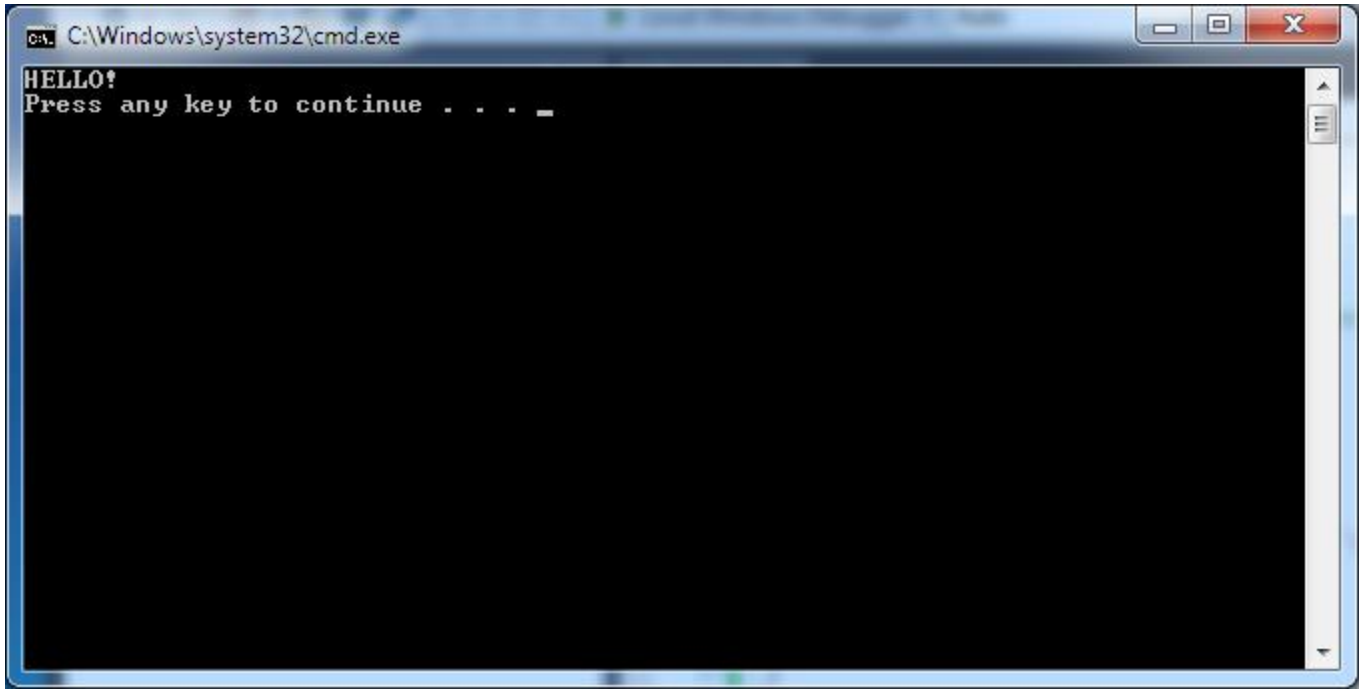
- To get ToUpper to work properly, we need to pass by reference

```
void ToUpper(std::string& text) {
   int length = text.length();
   for (int i = 0; i < length; i++) {
      // It's a lowercase letter
      if (text[i] >= 'a' && text[i] <= 'z') {
         text[i] -= 32;
      }
   }
}


int main() {
   std::string test = "hello!";
   ToUpper(test);
   std::cout << test << std::endl;
   return 0;
}
```

Another Example, Cont'd

- That one little ampersand makes a big difference



Another Reason to Pass by Reference

- Suppose we have a std::string that has 100,000 characters in it
- If we pass by value, a copy of those 100,000 characters has to be made, which is potentially expensive
- On the other hand, if we pass by reference, no copy will be made!

Pass by Reference, Best Practices

- For ***basic types*** (such as `int`, `char`, `float`, `double`, and so on):
  - Usually, pass by value
  - …UNLESS you ***do*** want parameter modifications to persist (as is the case with swap)

- For any ***non-basic types*** (everything other than basic types including `std::string`, `std::ifstream`, `std::ofstream`, etc.):
  - Usually, pass by reference
  - …UNLESS you need to modify the parameter in the function and ***don't*** want modifications to persist

Best Practices, Example

- Not:

```
void printString(std::string text) {
    std::cout << text << std::endl;
}
```

- But instead:

```
// Non-basic type, so pass by reference!
void printString(std::string& text) {
    std::cout << text << std::endl;
}
```

Best Practices, Example

- Not:

```
int Add(int& x, int& y) {
    return x + y;
}
```

- But instead:

```
// No reason to pass by reference,
// since int is a basic type and
// I'm not changing x/y
int Add(int x, int y) {
return x + y;
}
```

Function Overloading

- Function ***overloading*** is when you declare two functions with the same name, but they take in different parameter types

- So for example, I could have two versions of "add":

```
// Add version for ints
int Add(int x, int y) {
    return x + y;
}

// Add version for doubles
double Add(double x, double y) {
    return x + y;
}
```

Function Overloading in Action

```
int main() {
    // This calls the int version
    int a = Add(5, 10);
    std::cout << a << std::endl;

    // This calls the double version
    double b = Add(6.5, 3.4);
    std::cout << b << std::endl;
    return 0;
}
```

Question

- Which version gets called in this case?

```
std::cout << Add(5, 10.5) << std::endl;
```

- Answer: It's a compile error, because it doesn't know which version to pick!
  - It could pick Add(int, int) and convert the 10.5 to an int
  - It could pick Add(double, double) and convert the 5 to a double
  - Neither version is clearly a better match – and C++ does not like ambiguity

Function Overloading, Cont'd

- You can't overload a function just by changing the return type. You ***have*** to also change parameter types

- So this would be an error:

```
int Add(int x, int y) {
    return x + y;
}

double Add(int x, int y) {
    return x + y;
}
```

- Error: cannot overload functions distinguished by return type alone

Function Overloading, Cont'd

- You are technically allowed to declare functions that are only overloaded by the fact that they pass by value vs. reference:

```
int Add(int x, int y) {
    return x + y;
}

int Add(int& x, int& y) {
    return x + y;
}
```

- However, this will cause the "by-value" version to be called in nearly every instance. So ***don't do this***!

Lab Practical #10