# Creating Functions

ITP 165 – Fall 2015

Week 6, Lecture 1

# A New Function

```
// Function: OutputNum
// Purpose: Outputs an integer to cout
// Parameters: Integer to output
// Returns: Nothing
void OutputNum(int number)
{
   std::cout << number << std::endl;
}
```

# What’s Different About this Declaration?

We now have a parameter!

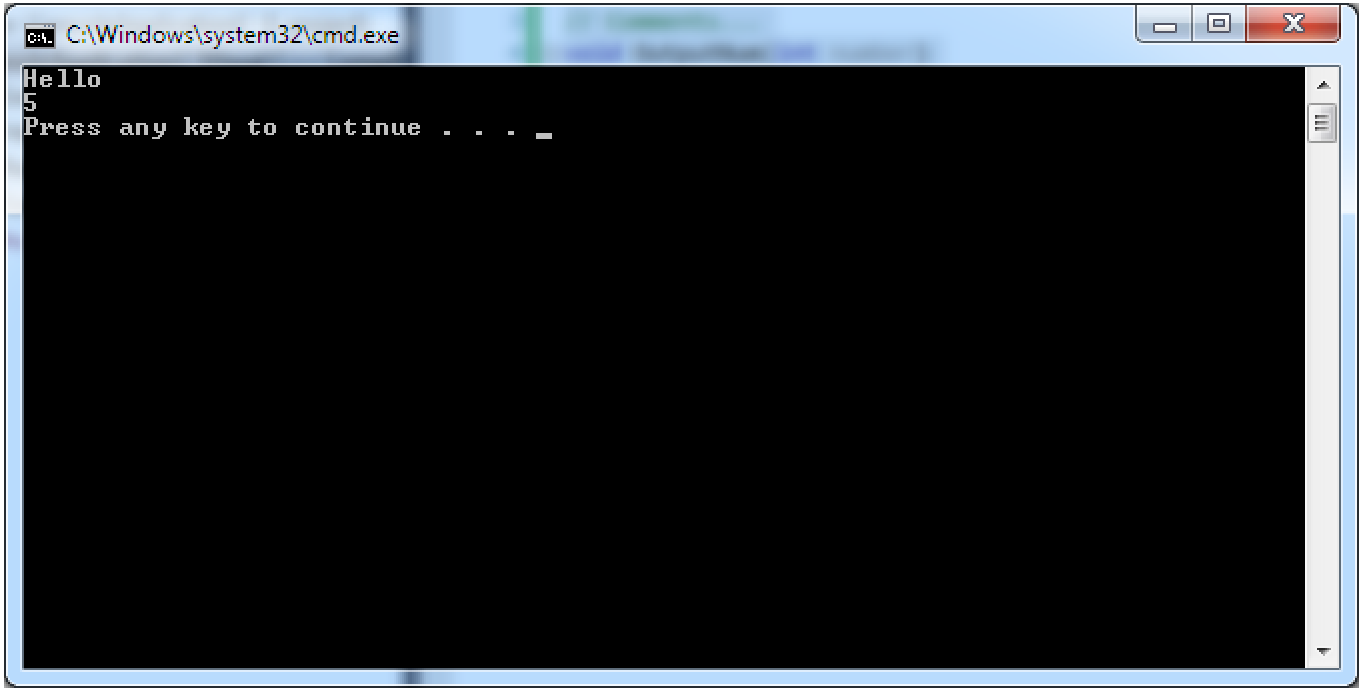
This parameter is of type int and is called number.

```
void OutputNum(int number)
{
    std::cout << number << std::endl;
}
```

# Using This New Function

```
#include <iostream>

// Comments...
void OutputNum(int number)
{
   std::cout << number << std::endl;
}

// Comments...
void SayHello()
{
   std::cout << "Hello" << std::endl;
}

int main()
{
   SayHello();
   OutputNum(5);

   return 0;
}
```

# Using This New Function, Cont’d



```
Hello
5
Press any key to continue . . .
```

# Functions and Modifying Parameters

- By default, parameters ***(other than arrays)*** are passed into functions by a copy

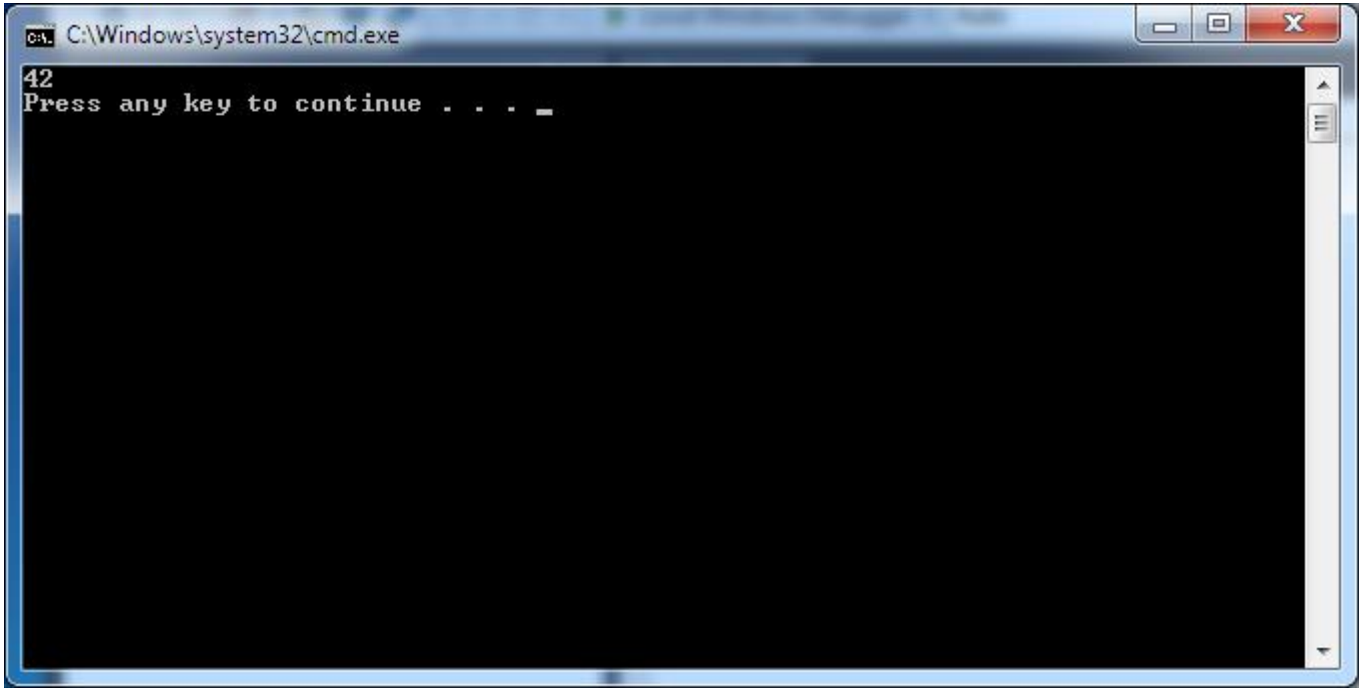
- This means that if a function modifies a parameter, when you return from the function, that new value will be lost

- This is also called ***pass by value***

# Functions and Modifying Parameters, Cont’d

```
#include <iostream>

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

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

# Functions and Modifying Parameters, Cont'd



```
42
Press any key to continue . . .
```

# Arrays and Functions

- If you pass in an array to a function, it ***does not*** make a copy

```
// Function: ToUpper
// Purpose: Turns C-style string to uppercase
// Input: C-style string to modify
// Returns: Nothing
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;
      }
   }
}
```

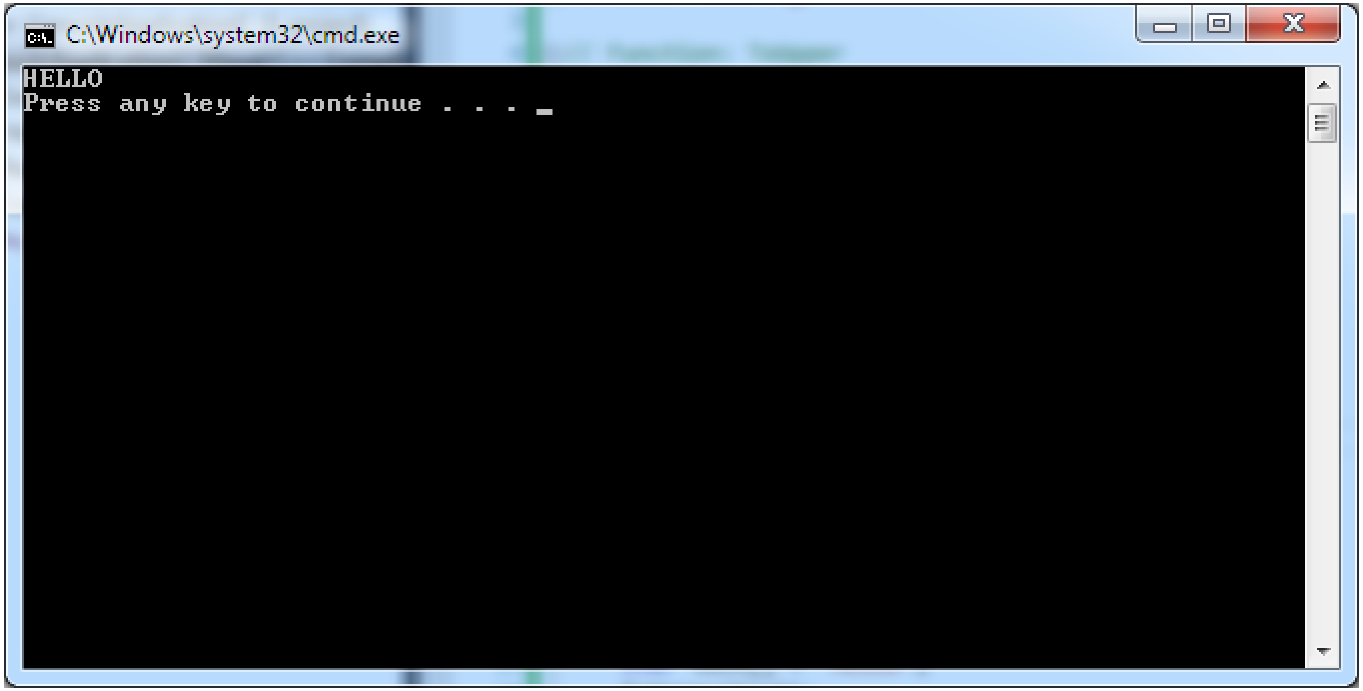
# Arrays and Functions, Cont’d

- So if I use `ToUpper` like this:

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

- It will work because `ToUpper` ***will*** modify the C-style string that we pass to it

# Arrays and Functions, Cont'd



```
HELLO
Press any key to continue . . . _
```

# Functions and Scope

- Each function has a separate scope
- So something like this is okay:

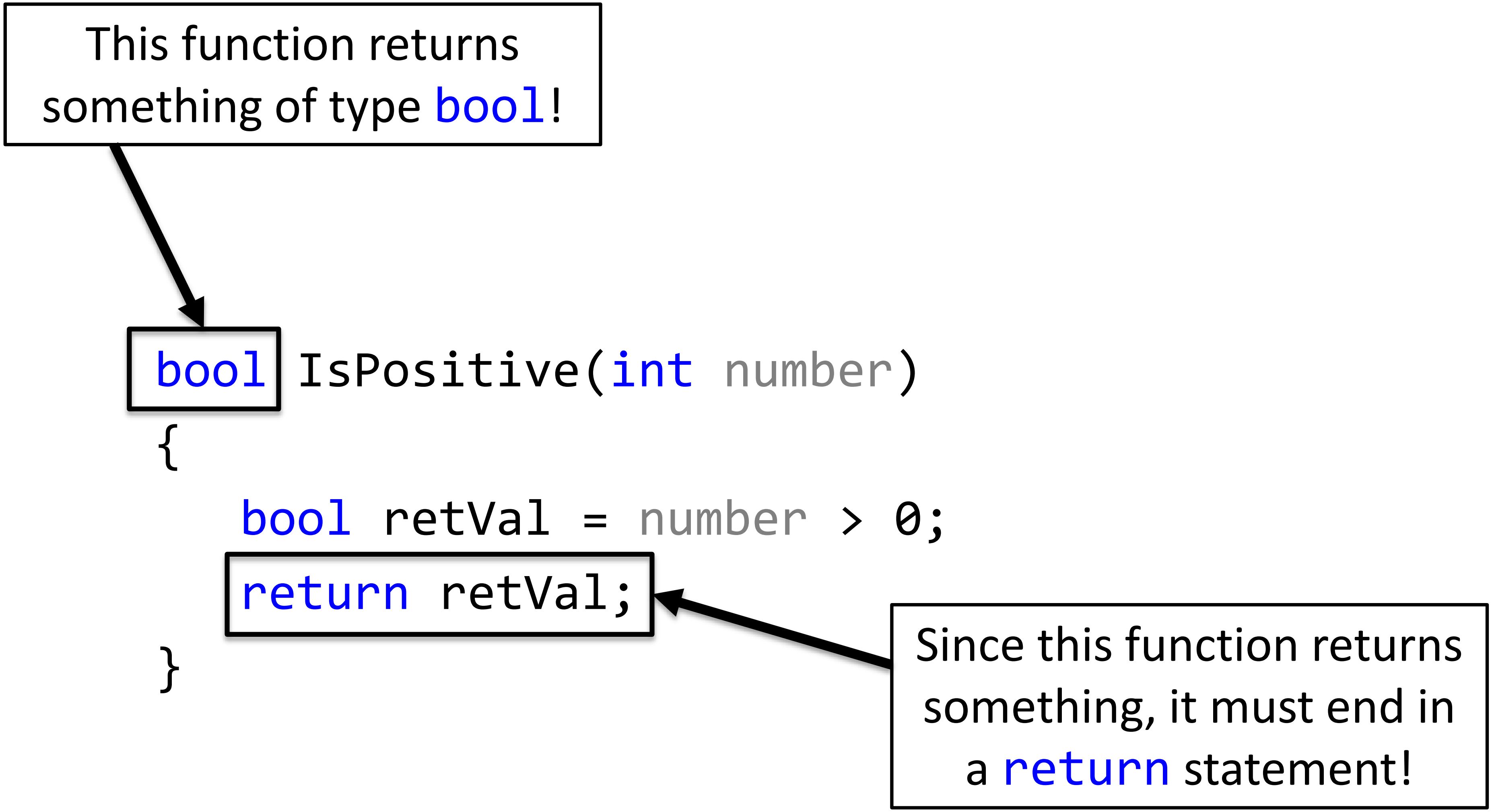
```
void Test()
{
   int x;
}

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

- Because the x in Test is different from the x in main

# A Function that Returns a Value…

```
// Function: IsPositive
// Purpose: Tests if a number is positive
// Parameters: Integer to test
// Returns: true if the number is positive
bool IsPositive(int number)
{
   bool retVal = number > 0;
   return retVal;
}
```

# What’s Different About this Declaration?



This function returns something of type `bool`!

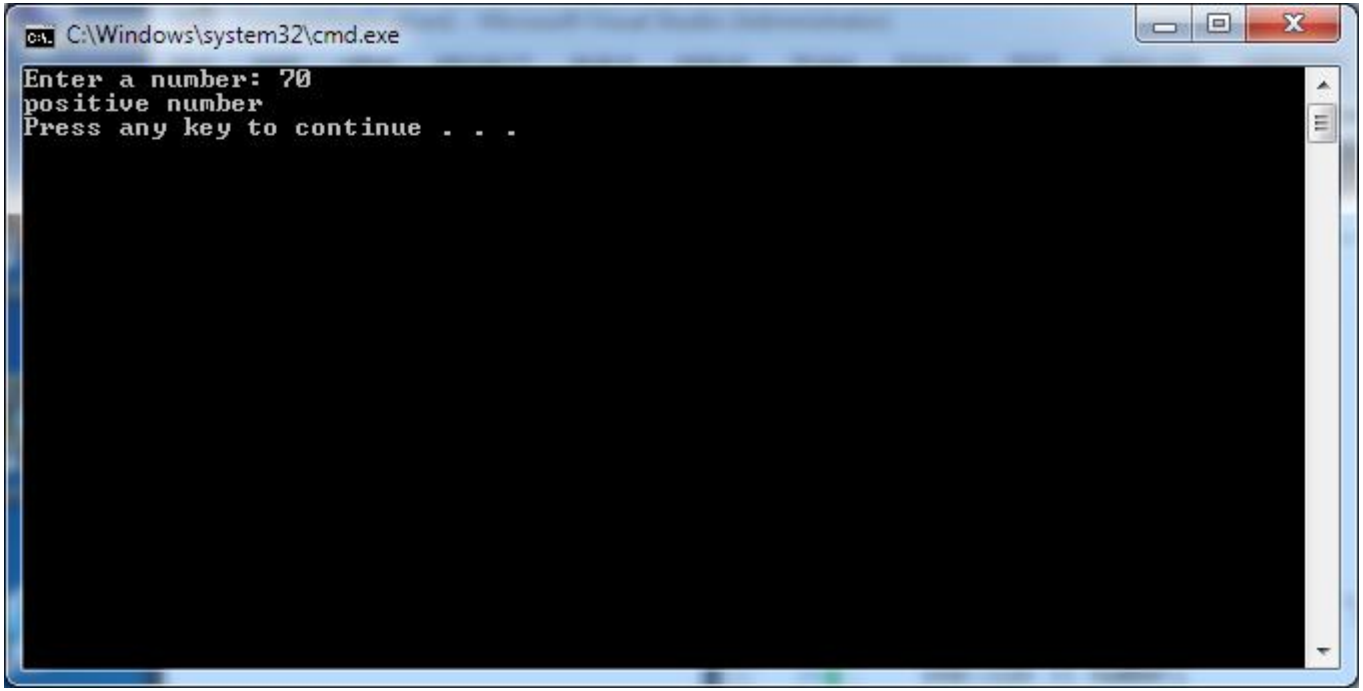
```
bool IsPositive(int number)
{
    bool retVal = number > 0;
    return retVal;
}
```

Since this function returns something, it must end in a `return` statement!

# A Function that Returns a Value, Cont’d

```
#include <iostream>

// Comments...
bool IsPositive(int number) {
   bool retVal = number > 0;
   return retVal;
}

int main() {
   std::cout << "Enter a number: ";
   int number = 0;
   std::cin >> number;

   if (IsPositive(number)) {
      std::cout << "positive number" << std::endl;
   } else {
      std::cout << "negative or zero" << std::endl;
   }

   return 0;
}
```

# A Function that Returns a Value, Cont’d



```
Enter a number: 70
positive number
Press any key to continue . . .
```

# A Function that Takes Multiple Parameters

```
// Function: Add
// Purpose: Adds two integers
// Parameters: Two integers to add
// Returns: The result of adding the two integers
int Add(int x, int y)
{
    return x + y;
}
```

# What’s Different About this Declaration?

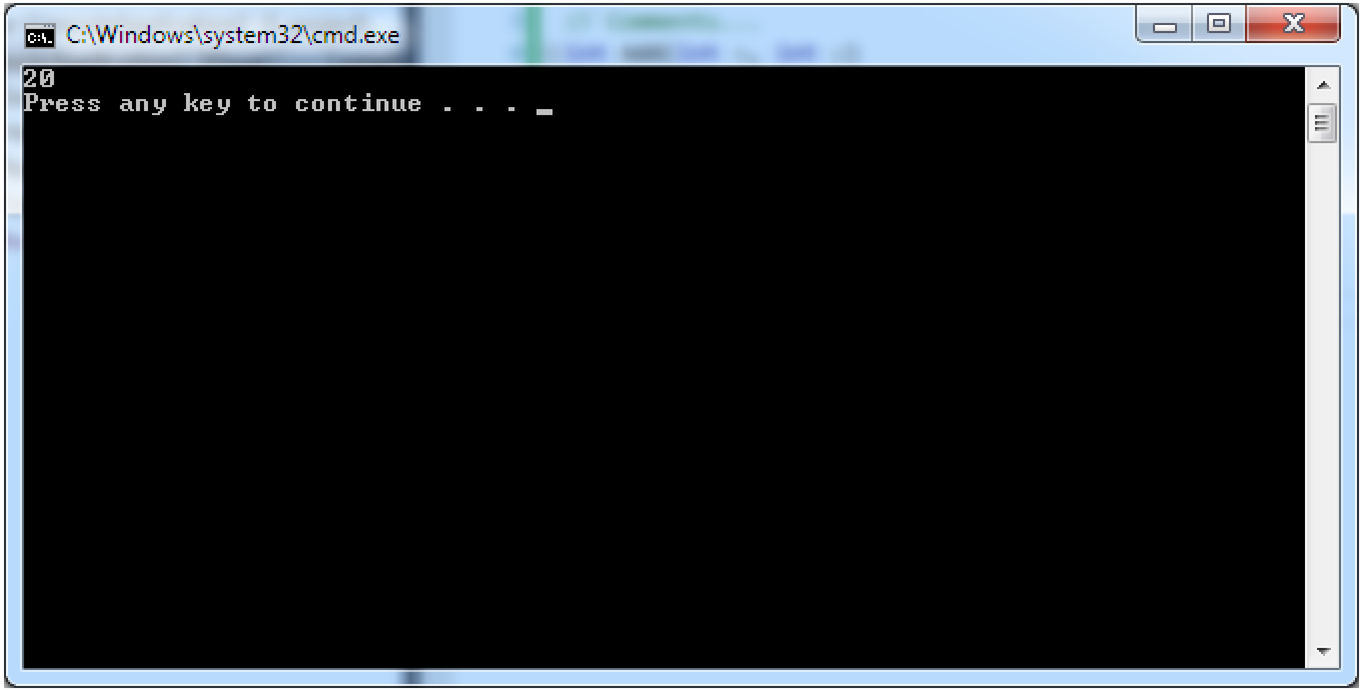
We have two parameters, so they are separated by commas.

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

# Add in Action

```
#include <iostream>

// Comments...
int Add(int x, int y)
{
   return x + y;
}

int main()
{
   std::cout << Add(17, 3) << std::endl;
   return 0;
}
```

# Add in Action, Cont’d



```
20
Press any key to continue . . .
```

# An Example w/ Two Functions

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

int Sub(int x, int y) {
    return x - y;
}

int main() {
    std::cout << Add( Sub(19, 2) , 3) << std::endl;
    return 0;
}
```

# Let’s Look at this line…

```
std::cout << Add( Sub(19, 2) , 3) << std::endl;
```

- So we have a call to Add, but the first parameter is the result of a call to Sub

- This means that first it will call Sub with 19 and 2, which will return 17

- Then it will call Add with 17 and 3

# A Function that Calls Another Function

```cpp
#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;
}
```

# A Function that Calls Another Function, Cont’d

```
#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;
}
```

Program still starts at main

# A Function that Calls Another Function, Cont’d

```
#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;
}
```

Call to Test, so “pause” main

# A Function that Calls Another Function, Cont’d

```
#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;
}
```

Start running code in Test

# A Function that Calls Another Function, Cont’d

```
#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;
}
```

Call to Add, so “pause” Test

# A Function that Calls Another Function, Cont’d

```
#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;
}
```

Start running code in Add

# A Function that Calls Another Function, Cont'd

```
#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;
}
```

Add ends, so return the value

# A Function that Calls Another Function, Cont'd

```
#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;
}
```

Resume Test where it was paused

# A Function that Calls Another Function, Cont'd

```cpp
#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 ends

# A Function that Calls Another Function, Cont’d

```
#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;
}
```

Resume main where it was paused

# Lab Practical #9