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
9. Reference parameters
10. Recursive functions

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;
}
```

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 calculates the cube only for a “reasonable” positive input, but consider what is returned if the call passes in a negative value!

You need to ensure all paths of execution include a `return` statement. So the code below needs an `else` with its own `return` after the `if`, to return perhaps a flag of -1.

```
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
- 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);`

Function Declarations – Complete Program

```
#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

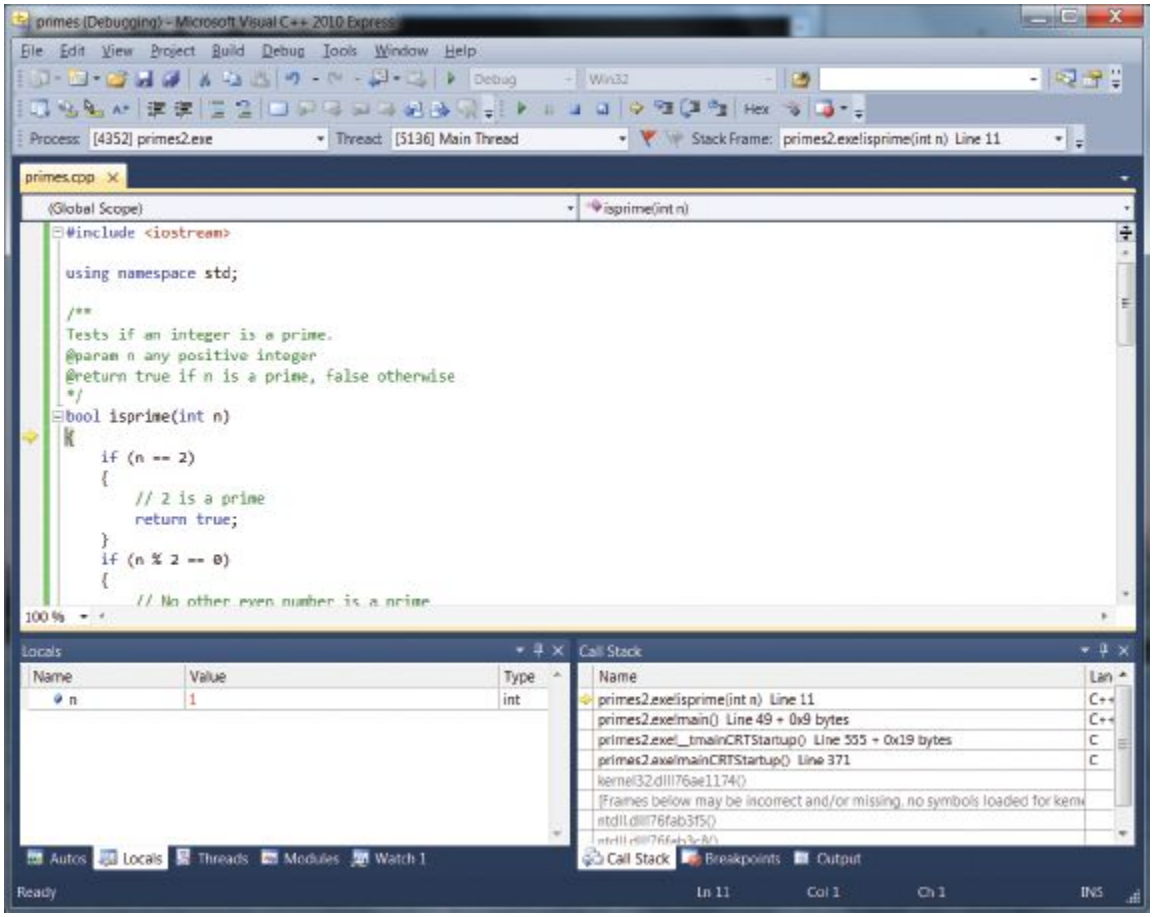
Complete Code for the Pyramid Function, with Testbench

```
#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 to Debug Functions

- Your 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.

Microsoft Visual Studio IDE / Debugger shown above, with next line to be executed shown by yellow arrow in the Breakpoint margin at left.

Using the IDE Debugger: Typical Session

- 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.

Topic 5

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

Functions Without Return Values

Consider the task of writing a string with the following format around it.

Any string could be used.

For example, the string `"Hello"` would produce:

```
-------
!Hello!
-------
```

Functions Without Return Values – The `void` Type

This kind of function is called a ***`void`*** *function*.

```
void box_string(string str)
```

Use a return type of **`void`** to indicate that a function does not return a value.

`void` functions are used to
simply do a sequence of instructions
– They do not return a value to the caller.

`void` Type Example

```
void box_string(string str)
{
   int n = str.length();
   for (int i = 0; i < n + 2; i++){ cout << "-"; }
   cout << endl;
   cout << "!" << str << "!" << endl;
   for (int i = 0; i < n + 2; i++) { cout << "-"; }
   cout << endl;
}
```

Note that this function doesn't compute any value.

It performs some actions and then returns to the caller – without returning a value.

(The return occurs at the end of the block.)

Calling `void` Functions

Because there is no return value, you cannot use **`box_string`** in an expression.

You can make this call kind of call:

```
box_string("Hello");
```

but not this kind:

```
result = box_string("Hello");
  // Error: box_string doesn't
  //        return a result.
```

Early Return from a `void` Function

If you want to return from a **`void`** function before reaching the end, you use a **`return`** statement without a value. For example:

```
void box_string(string str)
{
   int n = str.length();
    if (n == 0)
   {
      return;
   }                    // Return immediately
   . . .  // None of the statements after this
          // in the box_string function
        // will be executed
```

Topic 6

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

Designing Functions – Turn Repeated Code into Functions

When you found you have written nearly identical code multiple times,

you should write a function to replace the redundant code.

Repeated Code Example

Consider how similar the following statements are:

```
int hours;
do
{
   cout << "Enter a value between 0 and 23:";
   cin >> hours;
} while (hours < 0 || hours > 23);

int minutes;
do
{
   cout << "Enter a value between 0 and 59: ";
   cin >> minutes;
} while (minutes < 0 || minutes > 59);
```

Turn Repeated Code into Functions

Move the *common behavior* into *one* function.

```
int read_int_up_to(int high)
{
   int input;
   do
   {
      cout << "Enter a value between "
         << "0 and " << high << ": ";
      cin >> input;
   } while (input < 0 || input > high);
   return input;
}
```

Function Calls That Replace Repeated Code

Then we can use this function as many times as we need:

```
int hours = read_int_up_to(23);
int minutes = read_int_up_to(59);
```

Note how the code has become much easier to understand.

And we are not rewriting code

– code reuse!