Lecture 14 — More About Functions

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 21, 2016

Acknowledgments: W.D. Bishop

Part I

Re-Using Functions

Functions, Continued

In the last lecture, we discussed the “what” of functions, but not why.

One of the strategies for engineering problem-solving is the divide-and-conquer approach.

Break a large problem into smaller sub-problems. Solve the smaller problems, then recombine the answers to get the final answer.

A major advantage of functions is the ability to re-use some code we have already written.

Re-Use of Functions

```
int students = 385;
int[] assignments = new int[students];
int[] midterms = new int[students];
int[] finals = new int[students];

for ( int i = 0; i < assignments.Length; i++ )
{
    assignments[i] = 0;
}
for ( int j = 0; j < midterms.Length; j++ )
{
    midterms[j] = 0;
}
for ( int k = 0; k < finals.Length; k++ )
{
    finals[k] = 0;
}
```

Re-Use of Functions

First, define the function `fill_with_zeros()`:

```
void fill_with_zeros( int[] array, int length )
{
    for ( int i = 0; i < length; i++ )
    {
        array[i] = 0;
    }
}
```

Then, use it:

```
int students = 385;
int[students] assignments;
int[students] midterms;
int[students] finals;

fill_with_zeros( assignments, students );
fill_with_zeros( midterms, students );
fill_with_zeros( finals, students );
```

Advantages of Re-Using Functions

1. Decompose the problem into smaller subproblems
2. Reduce duplicate code in the program
3. Allow re-use of code elsewhere in the program
4. Hide the implementation details

Duplicate code is usually produced by copy-and-pasting existing code.

That means twice as many places to change to fix something.

Avoid duplicating code where possible.

Structuring Your Program

Functions also give us the ability to structure our program, rather than just have all parts of it jammed together in `main`.

It should be possible to summarize the purpose of a function in a sentence or two.

If a function is long or does multiple things, chances are that there are smaller functions within the code that can be separated out.

Writing code with good structure will make your program:

- Easier to understand
- Easier to update in the future
- Easier to fix if it’s not correct
- More re-usable

On Good Structure

Using good structure may not seem like it makes a difference in programming for ECE 150 since the amount of code written is small.

Get into proper habits now and it will save you headaches later on.

In later courses (operating systems, fourth year design project), you will write a lot more code and you'll need to manage it properly.

In industry, codebases can be hundreds of thousands of lines. Without structure, the program will quickly become a disaster.

Part II

Documenting Functions

Documentation

This is not intended as a long essay on documentation; just a strategy that will help you write better functions.

Using comments, above each function you write, take a moment to write the precondition and postcondition.

Precondition: anything that is assumed to be true when the function is called.

Postcondition: the effect of the function call (i.e., what happens if you call it when the precondition is true).

These two pieces of documentation can tell you how to use a function properly, without having to examine the implementation.

Using Pre- and Postconditions

```
// Precondition: there is at least one grade in the array
// Postcondition: the average of all the grades in the array is returned
static double calculate_average( double[] grades, int length )
{
    double sum = 0;
    for ( int i = 0; i < length; i++ )
    {
        sum += grades[i];
    }
    return ( sum / length );
}
```

Violating the Precondition

What happens if we violate the precondition of `calculate_average()` and `grades` is an array of capacity 0?

And who is at fault if the precondition is violated and an error occurs?

Think of pre- and postconditions as a contract.

If the caller of the function lives up to the precondition, the function is responsible for fulfilling the postcondition.

Part III

Variable Scope

Variable Scope

You may have noticed:

```
for ( int i = 0; i < 10; i++ )
{
    // Loop Body
}

cout << i << endl;
```

... produces a compile-time error at the `cout` statement, because the compiler does not know what `i` is.

Variable Scope

The scope of a variable is the context in which that variable is declared and available for use.

Here, variable `i` below is available only within the body of the loop.

```
for ( int i = 0; i < 10; i++ )
{
    cout << i << endl; // Valid
}

cout << i << endl; // Compile time error
```

Variable Scope in Functions

This same principle applies to functions.

  A variable defined in `main` is not defined in another function.

A variable defined inside a function is called a local variable.

```
static void Main ()
{
    double val = 0;
    int[42] zeroFill;
    fill_with_zeros( zeroFill, 42 );
}

void fill_with_zeros( int[] array, int length )
{
    for ( int i = 0; i < length; i++ )
    {
        array[i] = 0;
    }
}
```

The variable `val` is not available in `fill_with_zeros`.

Variable Scope in Functions

Further observations on that same code:

```
int main ()
{
    double val = 0;
    int[42] zeroFill;
    fill_with_zeros(zeroFill, 42);
    return 0;
}

void fill_with_zeros(int[] array, int length)
{
    for (int i = 0; i < length; i++)
    {
        array[i] = 0;
    }
}
```

The array `zeroFill` is a parameter to `fill_with_zeros` so the will be accessible, but we must refer to it using the name `array`.

The name `zeroFill` is not in scope inside `fill_with_zeros`.

Knowing if a Variable is in Scope

The good news is that it is a compile-time error if we try to use a variable that is not in scope.

This means we know right away this needs to be fixed.

The use of { } braces defines the scope of a variable.

A variable is in scope within the braces that contain its definition.

It is possible to define a variable within a function, a loop, or even a selection statement.

Defining a Variable within an `if`

```
if ( x < 0 )
{
    bool invalid = ( x == -1 );
    // Block 1

} else if ( x > 100 ) {
    // Block 2
}
```

The `bool` variable is in scope only within the `if ( x < 0 )` block (1).

Manual Variable Scope

It is possible to manually define the scope of a variable using braces.

As before, the scope of the variable is defined by the { } braces it is in.

```
int x = 100;
{
    int y = 54;
    cout << y << endl;
}
cout << x << endl; // y unavailable in this context
```

But there is unlikely to be a good reason to do this.

Part IV

Function Overloading

Function Overloading

If you just copy and paste a function, you will see a compile-time error, because you have two functions with the same signature.

Yes, the signature is what matters, even if the implementations differ.

If two functions have the same signature, the compiler can’t tell which one you mean when you call that function.

Could two functions have the same name, but different signatures?
Yes – this is called function overloading.

Function Overloading

A function is overloaded if one function name has more than one implementation. Consider the two function signatures below:

```
double average( double n1, double n2 )
```

```
double average( double n1, double n2, double n3)
```

The compiler looks at the actual parameters when the function is called to figure out which implementation should execute.

Example: if `average( 7.5, 12.5 );` is called, the first of the two signatures above will be the function that is called.

Rules for Overloading

If two functions have the same name, they must have either:
  different numbers of formal parameters, or
  formal parameters of different types.

You cannot overload a function by giving two definitions that differ only in the return type. So this is an error:

```
double average( int n1, int n2 )
int average( int n1, int n2 )
```

Why not? Because the compiler won't know which `average` you mean.

Overloading and Type Promotion

Type promotion is still in effect when calling overloaded functions.

Picture these two functions:

```
int compute( int n1 )
int compute( double n1 )
```

If the call is `compute( 10 )`, it's possible both of these will match: 10 may be an `int` but it could be automatically promoted to `double`.

C++ will choose the "best" match: the first function is chosen because the compiler will try to find a match without conversion, if it can.

That is "better" than converting `int` to `double`.

This may result in some unexpected behaviour, but it is consistent.

Overloading and Type Promotion

Sometimes, however, C++ can't figure out which choice is "best".

Picture these two function signatures:

```
int compute( int n1, double n2 )
int compute( double n1, int n2 )
```

If the call is `compute( 10, 10 )` there are two ways to resolve this:
  promote the first parameter to `double`; or
  promote the second parameter to `double`

C++ can't figure out which of these is better, because they are equal. This will be a problem because the situation is ambiguous.

Proper Use of Overloading

You can make proper use of overloading as a way to have some “default” values in function calls.

Here are two function signatures:

```
double calculate( double x, double y )
double calculate( double x, double y, bool option )
```

The implementation of the first function might look like this:

```
double calculate( double x, double y )
{
    return calculate( x, y, true );
}
```

About Overloading

Overloading may be useful in some circumstances.

You can use overloading to have some “default” values, or in cases where you want to perform the same operation using different inputs.

Avoid overloading where there is the possibility for confusion, such as in the presence of type promotion.