Lecture 15 − Parameter Passing

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Part I

Optional Parameters

Last time, we examined function overloading as a way of having a default value for a function.

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

It is also possible to specify in a function's signature that one or more parameters are optional.

```
double compute( double a, double b,  double c = 1.0d )
{
    return a * b / c;
}
```

In the previous example, the parameter c is optional:
  If the caller provides a value, that value is used.
  Otherwise, the default value of 1 is used.

Thus, both of these function calls are valid:
```
compute(5, 9, 2.5);
compute(45, 12);
```

This is like having an overloaded function, except in this case the compiler generates it for you. If we wrote it explicitly, it would be:

```
double compute( double a, double b )
{
    return compute( a, b, 1.0d );
}
```

We may have multiple optional parameters in the same function.

```
double compute( double a, double b = 2d,  double c = 1.0d )
```

When default values are added, they always start at the last
parameter and move right to left.
  This avoids developer and compiler confusion.

The following are errors:
  compute(double a = 2d, double b, double c)
  compute(double a = 2d, double b, double c = 1.0d)

Part II

Pass-By-Value, Pass-By-Reference

Consider the function shown below:

```
static void changeValueTo99 ( int number )
{
    number = 99;
}
```

And suppose this code appears in `Main`:

```
int score = 0;
changeValueTo99( score );
Console.WriteLine( score );
```

What do you expect the output will be?

Why did `score` stay at 0 and not get changed to 99?

Answer: because the default behaviour is pass-by-value.

When `changeValueTo99()` is called, the parameter `number` is created and initialized with 0; the same value as `score` holds.

However, `number` is its own copy, so a change made to `number` is not reflected in `score`. Two separate boxes in memory.

By default, all simple types and `struct` are passed by value.
  A copy is made when the function is called.

Is there an alternative to pass-by-value? Yes: pass-by-reference.

First, we need to define a reference.

Recall our model for a variable x: it is a box labelled x.
    This box is located somewhere in memory.

The computer will keep track of the memory address of x.
    When the code uses x, the computer knows where it is.

A reference is, quite simply, a memory address.
    A reference to x is the location of where to find x in memory.

Suppose a function has a formal parameter y and we provide an actual parameter x, both of type int.

When we pass x by value to a function, y is created as a copy of x; variables x and y have different memory addresses.

If we pass x by reference, instead of making a copy, we instead tell the function, "here's where x is" and y gets the same location.

Because x and y refer to the same thing, a change made to y is automatically a change to x.

The keyword in C# for passing by reference is `ref`.

The `ref` keyword on a function's formal parameter causes the parameter to be passed by reference.

The `ref` keyword must appear in both the declaration of the function and the invocation of the function.

Consider the modified function shown below:

```
static void changeValueTo99 ( ref int number )
{
    number = 99;
}
```

And suppose this code appears in `Main`:

```
int score = 0;
changeValueTo99( ref score );
Console.WriteLine( score );
```

What do you expect the output will be now?

Why is score now 99 rather than 0?

When we passed score by reference, no copy of it was made.

Instead, number was initialized in such a way that it is the same box in memory as score.
    Thus a change to number is automatically a change to score.

Consider the function shown below:

```
static void setFirstEntryToOne ( int[] numbers )
{
    numbers[0] = 1;
}
```

And suppose this code appears in `Main`:

```
int[] scores = new int[10];
scores[0] = 0;
setFirstEntryToOne( scores );
Console.WriteLine( scores[0] );
```

What do you expect the output will be?

We did not make use of the `ref` keyword, so why did the value of `scores[0]` change to 1?

Unlike an `int`, `double`, or `struct`, in C# an array is a reference type.

`int[] scores` is a reference to a memory location for the array.
  It's directions telling us how to get to a memory location.
  Think of it like written directions on how to get to a restaurant.

When we access `scores[0]`, the computer looks up the location of the array, goes to the location, and then takes the first entry there.

scores is passed by value, but this just makes a copy of the reference.
    In other words, numbers contains a copy of the address information.
    Extending the analogy: write out another copy of the directions.

Thus, numbers[0] is the same location in memory as scores[0].
    Both sets of directions lead us to the same place.
    Once there, examine the first entry: it's the same memory address.

Consider the revised function shown below:

```
static void setFirstEntryToOne ( int[] numbers )
{
    numbers = new int[10];
    numbers[0] = 1;
}
```

And suppose this code appears in Main:

```
int[] scores = new int[10];
scores[0] = 0;
setFirstEntryToOne( scores );
Console.WriteLine( scores[0] );
```

What do you expect the output will be?

This time `scores[0]` is not changed. Why?

In `setFirstEntryToOne()` the `numbers` reference changed.
  We erased the directions and wrote new directions.

Thus when we follow the directions to `numbers` we don't end up at the same place as we would if we followed the directions to `scores`.

Because we made another copy of the directions, the original directions are not affected.

Thus `scores` still contains directions to the original memory location.

Now let's add `ref` to the array example:

```
static void setFirstEntryToOne ( ref int[] numbers )
{
    numbers = new int[10];
    numbers[0] = 1;
}
```

And suppose this code appears in `Main`:

```
int[] scores = new int[10];
scores[0] = 0;
setFirstEntryToOne( ref scores );
Console.WriteLine( scores[0] );
```

What do you expect the output will be?

Why did `scores[0]` change this time?

This time the reference `scores` was passed by reference.

Instead of two copies of the directions, there is only one copy.

Thus when the statement `numbers = new int[10];` changes the directions to `numbers`, the directions to `score` are changed too.

Passing parameters by reference can be useful when swapping things.
    Example: exchange two integer values (see the next slide).

Passing parameters by reference allows a software designer to build a function that returns several results or changes several variables.

In reality, this is rarely necessary since you can always create more functions to produce more results.

Passing by reference should be avoided unless absolutely necessary.

We still need to examine pass by reference vs. pass by value because of the behaviour of arrays when used as function parameters.
    Strings are immutable; to "change" one, we update the reference.

```
static void swap ( ref int r1, ref int r2 )
{
    int tmp = r1;
    r1 = r2;
    r2 = tmp;
}

static void Main( )
{
    int a = 5;
    int b = 12;

    Console.WriteLine( a );
    Console.WriteLine( b );

    swap( ref a, ref b );

    Console.WriteLine( a );
    Console.WriteLine( b );
}
```