# Functional Programming - Passing a function as a parameter (C#)

One of the core principles of Functional Programming (FP) is that functions are ‘first-class objects’ - meaning, amongst other things, that a function can be passed as a parameter into another function, or returned as the result of another function.

In this exercise we are going explore what passing a function as a parameter looks like in code and, more importantly, the benefits that that possibility offers.

Let us start by imagining that you have written a function that sorts a list of strings into alphabetical order. You might have implemented another of the standard algorithms, such as Insertion or Quick Sort - or perhaps you’ve even designed a brand new high-performance algorithm. The example code below implements the Merge Sort algorithm. (FList is a custom implementation of a ‘functional list’ - meaning that the list is immutable, and is made up of a Head and Tail – but that is not our concern here). It is actually two functions: SortAlphabetical, which splits a list, and then delegates to a second function to merge them. Both functions are recursive:

public static FList<string> SortAlphabetical(FList<string> list)

{

if (list.Count() < 2)

{

return list;

}

else

{

var half = list.Count() / 2;

return MergeAlphabetical(SortAlphabetical(list.Skip(half)),   
 SortAlphabetical(list.Take(half)));

}

}

public static FList<string> MergeAlphabetical(FList<string> a, FList<string> b)

{

if (a.IsEmpty)

{

return b;

}

else if (b.IsEmpty)

{

return a;

}

else if (string.Compare(a.Head, b.Head) < 0)

{

return FList.Cons(a.Head, MergeAlphabetical(a.Tail, b));

}

else

{

return FList.Cons(b.Head, MergeAlphabetical(a, b.Tail));

}

}

We can prove that this works either by calling the SortAlphabetical function from within a Console program, or a unit test, for example:

[TestMethod]

public void TestSortAlphabeticalHappyCase()

{

var list = FList.Cons("Flag", "Nest", "Cup", "Burg", "Yacht", "Next");

var sorted = MergeSort.SortAlphabetical(list);

var expected = FList.Cons("Burg", "Cup", "Flag", "Nest", "Next", "Yacht");

Assert.AreEqual(expected, sorted);

}

Now, suppose that we want to change the program to sort the names in reverse alphabetical order, this would require a change to just one line of code, where we compare two string values, from:

else if (string.Compare(a.Head, b.Head) < 0)

to

else if (string.Compare(a.Head, b.Head) > 0)

But what if, as is more likely, we want the option to sort *either* way? The simplest way to do this, in terms of programmer effort, would be to duplicate the original two methods, rename them to, say, SortReverse and MergeReverse, and change the line doing the comparison as shown above.

However, duplicating and then slightly modifying code is *always* a bad idea – the algorithm is essentially the same but now we have twice as much code to test and maintain. If, in future, we find a way to improve the algorithm we need to remember to alter it in both places.

A better option would be to pass in an additional Boolean parameter indicating whether we want the sorting to be alphabetical or reverse. For example:

public static FList<string> SortAlphabetical(FList<string> list, bool reverse = false)

{

if (list.Count() < 2)

{

return list;

}

else

{

var half = list.Count() / 2;

return MergeAlphabetical(SortAlphabetical(list.Skip(half), reverse),

SortAlphabetical(list.Take(half), reverse), reverse);

}

}

public static FList<string> MergeAlphabetical(

FList<string> a, FList<string> b, bool reverse)

{

if (a.IsEmpty)

{

return b;

}

else if (b.IsEmpty)

{

return a;

}

else if ((!reverse && string.Compare(a.Head, b.Head) < 0 )||

(reverse && string.Compare(a.Head, b.Head) > 0))

{

return FList.Cons(a.Head, MergeAlphabetical(a.Tail, b, reverse));

}

else

{

return FList.Cons(b.Head, MergeAlphabetical(a, b.Tail, reverse));

}

}

This is definitely a better solution: we’ve avoided duplicating all the common code. But now suppose another requirement comes along for a list of words to be sorted by word-length. We could expand the previous pattern and accommodate the new requirement, but the code will start to get ugly, and the risks increase that we will accidentally break one of the other forms of sorting in the process (though we could certainly mitigate that risk by having comprehensive automated tests). What we want is a single version of our sort function where we can somehow just change a small part of the code inside the function - in this case just the one line where we compare two strings. This is the cue for ‘passing a function as a parameter’. In the code below we’ve renamed the two functions to just Sort and Merge respectively - partly because this reflects their more generic capability and partly to allow us to keep the old and new versions alongside each other in the same file without a clash:

public static FList<string> Sort(

FList<string> list, Func<string, string, bool> greaterThan)

{

if (list.Count() < 2)

{

return list;

}

else

{

var half = list.Count() / 2;

return Merge(Sort(list.Skip(half), greaterThan),

Sort(list.Take(half), greaterThan), greaterThan);

}

}

public static FList<string> Merge(FList<string> a,

FList<string> b, Func<string, string, bool> greaterThan)

{

if (a.IsEmpty)

{

return b;

}

else if (b.IsEmpty)

{

return a;

}

else if (greaterThan(a.Head, b.Head))

{

return FList.Cons(a.Head, Merge(a.Tail, b, greaterThan));

}

else

{

return FList.Cons(b.Head, Merge(a, b.Tail, greaterThan));

}

}

As in our previous version, both functions take an additional parameter to specify *how* we want the list sorted, but this time it is not a simple Boolean, but rather a function, called greaterThan. The type of this parameter is defined as:

Func<string, string, bool> greaterThan

Which may be read as ‘a function that takes in two strings as parameters and returns a Boolean result’. Each of the following, separate, standalone, functions fits this specification:

public static bool alphabetical(string s1, string s2)

{

return string.Compare(s2, s1) > 0;

}

public static bool reverse(string s1, string s2)

{

return string.Compare(s2, s1) < 0;

}

public static bool length(string s1, string s2)

{

return s2.Length > s1.Length;

}

Notice that each of these three functions has a different name, but they all have the same type signature to fit the requirements of the greaterThan function needed as the second parameter for the new sort function.

So we can now test the sort function using any of those three, or any other function that has the same type signature:

[TestMethod]

public void TestSortWithAlphabeticalFunction()

{

var list = FList.Cons("Flag", "Nest", "Cup", "Burg", " Yacht", "Next");

var sorted = MergeSort.Sort(list, alphabetical);

var expected = FList.Cons("Burg", "Cup", "Flag", "Nest", "Next", " Yacht");

Assert.AreEqual(expected, sorted);

}

[TestMethod]

public void TestSortWithReverseFunction()

{

var list = FList.Cons("Flag", "Nest", "Cup", "Burg", "Yacht", "Next");

var sorted = MergeSort.Sort(list, reverse);

var expected = FList.Cons("Yacht ", "Next", "Nest", "Flag", "Cup","Burg");

Assert.AreEqual(expected, sorted);

}

[TestMethod]

public void TestSortByLengthDecreasing()

{

var list = FList.Cons("Flag", "Nest", "Cup", "Burg", " Yacht ", "Next");

var sorted = MergeSort.Sort(list, length);

var expected = FList.Cons( "Cup", "Flag", "Nest", "Burg","Next", " Yacht ");

Assert.AreEqual(expected, sorted);

}

We don’t even need to code the implementation of ‘greaterThan’ as a separate standalone function: we can define it a ‘lambda’ - which is just like a function declared in-line. The following example of using a lambda, produces the same result as using the pre-defined length function (above):

[TestMethod]

public void TestSortByLengthDecreasingUsingLambda()

{

var list = FList.Cons("Flag", "Nest", "Cup", "Burg", " Yacht ", "Next");

var sorted = MergeSort.Sort(list, (s1, s2) => s2.Length > s1.Length);

var expected = FList.Cons( "Cup", "Flag", "Nest", "Burg", "Next", " Yacht ");

Assert.AreEqual(expected, sorted);

}

In the above code the lambda (s1, s2) => s2.Length > s1.Length may be read as ‘*Given* two strings, s1 and s2, *return* the result of s2.Length > s1.Length. The ‘=>’ symbol is sometimes pronounced as ‘goes to’ or even ‘fat arrow’.

Typically, you would only define an explicit standalone function for performing the string comparison, if you wanted to be able to use that same comparison more than once. (Using Lambda’s has some other advantages, too, but these are outside the scope of this lesson).

It is also now possible for us to generalise our mergesort function further, so that it can not just sort strings, but any type of object. To do this we use the ‘generics’ syntax, where we specify the type of object being sorted as ‘T’, and where we must now provide a ‘greaterThan’ function that takes in two objects of type T and returns a Boolean:

public static FList<T> Sort<T>(FList<T> list, Func<T, T, bool> greaterThan)

{

if (list.Count() < 2)

{

return list;

}

else

{

var half = list.Count() / 2;

return Merge(Sort(list.Skip(half), greaterThan),

Sort(list.Take(half), greaterThan), greaterThan);

}

}

public static FList<T> Merge<T>(FList<T> a, FList<T> b, Func<T, T, bool> greaterThan)

{

if (a.IsEmpty)

{

return b;

}

else if (b.IsEmpty)

{

return a;

}

else if (greaterThan(a.Head, b.Head))

{

return FList.Cons(a.Head, Merge(a.Tail, b, greaterThan));

}

else

{

return FList.Cons(b.Head, Merge(a, b.Tail, greaterThan));

}

}

The following code shows the same function now being used to sort a list of integers, first in increasing, then in decreasing order:

[TestMethod]

public void TestSortIntegers()

{

var list = FList.Cons(4, 7, 12, 3, 88, 9, 2, 7);

var sorted = MergeSort.Sort(list, greaterThan);

var expected = FList.Cons(2,3,4,7,7,9,12,88);

Assert.AreEqual(expected, sorted);

}

[TestMethod]

public void TestSortIntegersInReverse()

{

var list = FList.Cons(4, 7, 12, 3, 88, 9, 2, 7);

var sorted = MergeSort.Sort(list, reverse);

var expected = FList.Cons(88,12,9,7,7,4,3,2);

Assert.AreEqual(expected, sorted);

}

Making use of these two implementations of the greaterThan function *specifically* for comparing two integers:

public static bool greaterThan(int i1, int i2)

{

return i2 > i1;

}

public static bool reverse(int i1, int i2)

{

return i1 > i2;

}

The MergeSort example that we’ve worked through here is a simple, but nonetheless realistic case for passing a function as a parameter. In fact, there are standard libraries that do exactly this. For example, the following code, which uses a regular list rather than our functional list, calls the standard Microsoft Linq function OrderBy, passing in a lambda to determine what to order by:

var list = new List<string>() { "Flag", "Nest", "Cup", "Burg", "Yacht", "Next" };

var sorted = list.OrderBy(s => s.Length);