A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure.

```
public delegate void MyDelegate(string msg); // declare a delegate

// set target method
MyDelegate del = new MyDelegate(MethodA);
// or
MyDelegate del = MethodA;
// or set lambda expression
MyDelegate del = (string msg) =>  Console.WriteLine(msg);

// target method
static void MethodA(string message)
{
    Console.WriteLine(message);
}
```

After setting a target method, a delegate can be invoked using the `Invoke()` method or using the `()` operator:

```
del.Invoke("Hello World!");
// or
del("Hello World!");
```

Example: Generic Delegate

```
public delegate T add<T>(T param1, T param2); // generic delegate

class Program
{
    static void Main(string[] args)
    {
        add<int> sum = Sum;
        Console.WriteLine(sum(10, 20));

        add<string> con = Concat;
        Console.WriteLine(conct("Hello ","World!!"));
    }

    public static int Sum(int val1, int val2)
    {
        return val1 + val2;
```

```
    }

    public static string Concat(string str1, string str2)
    {
        return str1 + str2;
    }
}
```

Multicast Delegate

The delegate can point to multiple methods. A delegate that points multiple methods is called a multicast delegate. The "+" or "+=" operator adds a function to the invocation list, and the "-" and "-=" operator removes it.

Example: Multicast Delegate

```
public delegate void MyDelegate(string msg); //declaring a delegate

class Program
{
    static void Main(string[] args)
    {
        MyDelegate del1 = ClassA.MethodA;
        MyDelegate del2 = ClassB.MethodB;

        MyDelegate del = del1 + del2; // combines del1 + del2
        del("Hello World");

                    MyDelegate del3 = (string msg) =>
Console.WriteLine("Called lambda expression: " + msg);
        del += del3; // combines del1 + del2 + del3
        del("Hello World");

        del = del - del2; // removes del2
        del("Hello World");

        del -= del1 // removes del1
        del("Hello World");
    }
}
```

```
class ClassA
{
    static void MethodA(string message)
    {
                Console.WriteLine("Called  ClassA.MethodA()  with
parameter: " + message);
    }
}

class ClassB
{
    static void MethodB(string message)
    {
                Console.WriteLine("Called  ClassB.MethodB()  with
parameter: " + message);
    }
}
```

Any lambda expression can be converted to a [delegate](#) type.

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
```

```
using System;

using System.Collections.Generic;

using System.Linq;

public static class demo

{

   public static void Main()

    {

        List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 }
;

        List<int> evenNumbers = list.FindAll(x => (x % 2) ==
0);
```

```
        foreach (var num in evenNumbers)
       {
           Console.Write("{0} ", num);
        }
        Console.WriteLine();
        Console.Read();


   }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}
class demo{
    static void Main()
    {
        List<Dog> dogs = new List<Dog>() {
            new Dog { Name = "Rex", Age = 4 },
            new Dog { Name = "Sean", Age = 0 },
            new Dog { Name = "Stacy", Age = 3 }
         };
         var names = dogs.Select(x => x.Name);
         foreach (var name in names)
         {
             Console.WriteLine(name);

         }
         Console.Read();
    }
}
```

Using Lambda Expressions with Anonymous Types

```
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. class Dog
5. {
6.    public string Name { get; set; }
7.    public int Age { get; set; }
8. }
9. class demo{
10.    static void Main()
11.    {
12.       List<Dog> dogs = new List<Dog>() {
13.          new Dog { Name = "Rex", Age = 4 },
14.          new Dog { Name = "Sean", Age = 0 },
15.          new Dog { Name = "Stacy", Age = 3 }
16.       };
17.       var newDogsList = dogs.Select(x => new { Age = x.Age,
   FirstLetter = x.Name[0] });
18.       foreach (var item in newDogsList)
19.       {
20.          Console.WriteLine(item);
21.       }
22.       Console.Read();
23.    }
24. }
```

Sorting using a lambda expression

The following is an examle of sorting with a lambda expression:

```
1. var sortedDogs = dogs.OrderByDescending(x => x.Age);
2. foreach (var dog in sortedDogs)
3. {
4.    Console.WriteLine(string.Format("Dog {0} is {1} years old.
   ", dog.Name, dog.Age));
5. }
```

LINQ Operators and Lambda Expressions
LINQ is a cool feature in C# 3.0. Most of the developers are struggling for the syntax and examples. Here I have collected various examples for each operator in LINQ and the equivalent Lambda Expressions.

Where

```
1. IEnumerable<Product> x = products.Where(p => p.UnitPrice >= 1
   0);
2.
3. IEnumerable<Product> x =
4. from p in products
5. where p.UnitPrice >= 10
6. select p;
```

Select

```
1. IEnumerable<string> productNames = products.Select(p => p.Nam
   e);
2. IEnumerable<string> productNames = from p in products select
   p.Name;
3.
4. var namesAndPrices =
5. products.
6. Where(p => p.UnitPrice >= 10).
7. Select(p => new { p.Name, p.UnitPrice }).
8. ToList();
9. IEnumerable<int> indices =
10. products.
11. Select((product, index) => new { product, index }).
12. Where(x => x.product.UnitPrice >= 10).
13. Select(x => x.index);
```

Function signature	C# type	Example
int → string	Func<int, string>	(int i) => i.ToString()
() → string	Func<string>	() => "hello"
int → ()	Action<int>	(int i) => WriteLine($"gimme {i}")
() → ()	Action	() => WriteLine("Hello World!")
(int, int) → int	Func<int, int, int>	(int a, int b) => a + b

Summary Note:

C# is an object-oriented language that emphasizes state changes through imperative programming. But that doesn’t mean that C# doesn’t support functional programming. On the contrary, the latest versions of the language show how much Microsoft is concerned with making C# geared toward functional programming.

LINQ and lambda expressions are the most relevant examples in C# in which we can use the functional approach because they were already developed with this in mind, but there are many other features in C# that fulfill this purpose.