Function Pointers & Delegates

**Definition:**

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

public delegate int PerformCalculation(int x, int y);

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

|  |
| --- |
| **Note** |
| In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate. |

This ability to refer to a method as a parameter makes delegates ideal for defining call-back methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

Delegates have the following properties:

* Delegates are like C++ function pointers but are type safe.
* Delegates allow methods to be passed as parameters.
* Delegates can be used to define call-back methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event.
* Methods do not have to match the delegate type exactly. For more information, see Using Variance in Delegates.
* C# version 2.0 introduced the concept of Anonymous Methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions. For more information about lambda expressions, see Anonymous Functions.