

An SQL query using *minted*

```
1 CREATE OR REPLACE TRIGGER Tel_On_Off
2 AFTER INSERT ON STATE_CHANGE
3 FOR EACH ROW
4 WHEN (NEW.ChangeType='O' OR NEW.ChangeType='F')
5 DECLARE
6     CELLA NUMBER;
7     N_TEL_ATTIVI_MAX NUMBER;
8 BEGIN
9     --Trovo l'ID della cella in cui si trova il mio cellulare
10    SELECT CellID, MaxCalls INTO CELLA, N_TEL_ATTIVI_MAX
11    FROM CELL
12    WHERE x0<=:NEW.x AND x1>=:NEW.x AND
13           y0<=:NEW.y AND y1>=:NEW.y;
14
15    --Telefono acceso
16    IF(:NEW.ChangeType='O') THEN
17        --Inserisco il nuovo cellulare nella tabella
18        INSERT INTO TELEPHONE(PhoneNo, x, y, PhoneState)
19        VALUES(:NEW.PhoneNo, :NEW.x, :NEW.y, :NEW.ChangeType);
20        --Aggiorno la cella in cui si trova il cellulare
21        UPDATE CELL SET CurrentPhone#=CurrentPhone#+1
22        WHERE CellID=CELLA;
23    END IF;
24
25    --Telefono spento
26    IF(:NEW.ChangeType='F') THEN
27        --Rimozione telefono da tebella
28        DELETE FROM TELEPHONE WHERE PhoneNo=:NEW.PhoneNo;
29        --Aggiorno la cella in cui si trova il cellulare
30        UPDATE CELL SET CurrentPhone#=CurrentPhone#-1
31        WHERE CellID=CELLA;
32    END IF;
33 END;
```

An SQL query using *listings*

```
1 CREATE OR REPLACE TRIGGER Tel_On_Off
2 AFTER INSERT ON STATE_CHANGE
3 FOR EACH ROW
4 WHEN (NEW.ChangeType='O' OR NEW.ChangeType='F')
5 DECLARE
6     CELLA NUMBER;
7     N_TEL_ATTIVI_MAX NUMBER;
8 BEGIN
9     --Trovo l'ID della cella in cui si trova il mio
        cellulare
10    SELECT CellID, MaxCalls INTO CELLA, N_TEL_ATTIVI_MAX
11    FROM CELL
12    WHERE x0<=:NEW.x AND x1>=:NEW.x AND
13           y0<=:NEW.y AND y1>=:NEW.y;
14
15    --Telefono acceso
16    IF(:NEW.ChangeType='O') THEN
17        --Inserisco il nuovo cellulare nella tabella
18        INSERT INTO TELEPHONE(PhoneNo, x, y, PhoneState)
19        VALUES(:NEW.PhoneNo, :NEW.x, :NEW.y,
20               :NEW.ChangeType);
21        --Aggiorno la cella in cui si trova il cellulare
22        UPDATE CELL SET CurrentPhone#=CurrentPhone#+1
23        WHERE CellID=CELLA;
24    END IF;
25
26    --Telefono spento
27    IF(:NEW.ChangeType='F') THEN
28        --Rimozione telefono da tabella
29        DELETE FROM TELEPHONE WHERE PhoneNo=:NEW.PhoneNo;
30        --Aggiorno la cella in cui si trova il cellulare
31        UPDATE CELL SET CurrentPhone#=CurrentPhone#-1
32        WHERE CellID=CELLA;
33    END IF;
34 END;
```

A C# chunk of code using *minted*

```
1 using System;
2 using System.Runtime.InteropServices;
3
4 namespace Binarysharp.MemoryManagement.Memory
5 {
6     /// <summary>
7     /// Class representing a block of memory allocated in the
8     ↪ local process.
9     /// </summary>
10    public class LocalUnmanagedMemory : IDisposable
11    {
12        #region Properties
13        /// <summary>
14        /// The address where the data is allocated.
15        /// </summary>
16        public IntPtr Address { get; private set; }
17        /// <summary>
18        /// The size of the allocated memory.
19        /// </summary>
20        public int Size { get; private set; }
21        #endregion
22
23        #region Constructor/Destructor
24        /// <summary>
25        /// Initializes a new instance of the <see
26        ↪ cref="LocalUnmanagedMemory"/> class, allocating a
27        ↪ block of memory in the local process.
28        /// </summary>
29        /// <param name="size">The size to allocate.</param>
30        public LocalUnmanagedMemory(int size)
31        {
32            // Allocate the memory
33            Size = size;
34            Address = Marshal.AllocHGlobal(Size);
35        }
36        /// <summary>
37        /// Frees resources and perform other cleanup operations
38        ↪ before it is reclaimed by garbage collection.
39        /// </summary>
40        ~LocalUnmanagedMemory()
41        {
42            Dispose();
43        }
44        #endregion
45
46        #region Methods
47        #region Dispose (implementation of IDisposable)
48        /// <summary>
```

```

45     /// Releases the memory held by the <see
    ↪ cref="LocalUnmanagedMemory"/> object.
46     /// </summary>
47     public virtual void Dispose()
48     {
49         // Free the allocated memory
50         Marshal.FreeHGlobal(Address);
51         // Remove the pointer
52         Address = IntPtr.Zero;
53         // Avoid the finalizer
54         GC.SuppressFinalize(this);
55     }
56     #endregion
57     #region Read
58     /// <summary>
59     /// Reads data from the unmanaged block of memory.
60     /// </summary>
61     /// <typeparam name="T">The type of data to
    ↪ return.</typeparam>
62     /// <returns>The return value is the block of memory
    ↪ casted in the specified type.</returns>
63     public T Read<T>()
64     {
65         // Marshal data from the block of memory to a new
    ↪ allocated managed object
66         return (T)Marshal.PtrToStructure(Address, typeof(T));
67     }
68     /// <summary>
69     /// Reads an array of bytes from the unmanaged block of
    ↪ memory.
70     /// </summary>
71     /// <returns>The return value is the block of
    ↪ memory.</returns>
72     public byte[] Read()
73     {
74         // Allocate an array to store data
75         var bytes = new byte[Size];
76         // Copy the block of memory to the array
77         Marshal.Copy(Address, bytes, 0, Size);
78         // Return the array
79         return bytes;
80     }
81     #endregion
82 }
83 }

```

A C# chunk of code using *listings*

```
1 using System;
2 using System.Runtime.InteropServices;
3
4 namespace Binarysharp.MemoryManagement.Memory
5 {
6     /// <summary>
7     /// Class representing a block of memory
8     /// allocated in the local process.
9     /// </summary>
10    public class LocalUnmanagedMemory : IDisposable
11    {
12        #region Properties
13        /// <summary>
14        /// The address where the data is allocated.
15        /// </summary>
16        public IntPtr Address { get; private set; }
17        /// <summary>
18        /// The size of the allocated memory.
19        /// </summary>
20        public int Size { get; private set; }
21        #endregion
22
23        #region Constructor/Destructor
24        /// <summary>
25        /// Initializes a new instance of the <see
26        cref="LocalUnmanagedMemory"/> class,
27        allocating a block of memory in the local
28        process.
29        /// </summary>
30        /// <param name="size">The size to
31        allocate.</param>
32        public LocalUnmanagedMemory(int size)
33        {
34            // Allocate the memory
35            Size = size;
36            Address = Marshal.AllocHGlobal(Size);
37        }
38        /// <summary>
39        /// Frees resources and perform other cleanup
40        operations before it is reclaimed by
41        garbage collection.
42        /// </summary>
43        ~LocalUnmanagedMemory()
44        {
45            Dispose();
46        }
47        #endregion
48    }
49 }
```

```

41
42 #region Methods
43 #region Dispose (implementation of
    IDisposable)
44 /// <summary>
45 /// Releases the memory held by the <see
    cref="LocalUnmanagedMemory"/> object.
46 /// </summary>
47 public virtual void Dispose()
48 {
49     // Free the allocated memory
50     Marshal.FreeHGlobal(Address);
51     // Remove the pointer
52     Address = IntPtr.Zero;
53     // Avoid the finalizer
54     GC.SuppressFinalize(this);
55 }
56 #endregion
57 #region Read
58 /// <summary>
59 /// Reads data from the unmanaged block of
    memory.
60 /// </summary>
61 /// <typeparam name="T">The type of data to
    return.</typeparam>
62 /// <returns>The return value is the block of
    memory casted in the specified
    type.</returns>
63 public T Read<T>()
64 {
65     // Marshal data from the block of memory
        to a new allocated managed object
66     return (T)Marshal.PtrToStructure(Address,
        typeof(T));
67 }
68 /// <summary>
69 /// Reads an array of bytes from the
    unmanaged block of memory.
70 /// </summary>
71 /// <returns>The return value is the block of
    memory.</returns>
72 public byte[] Read()
73 {
74     // Allocate an array to store data
75     var bytes = new byte[Size];
76     // Copy the block of memory to the array
77     Marshal.Copy(Address, bytes, 0, Size);
78     // Return the array
79     return bytes;
80 }

```

```
81         #endregion
82     }
83 }
```
