An SQL query using *minted*

```sql
CREATE OR REPLACE TRIGGER Tel_On_Off
AFTER INSERT ON STATE_CHANGE
FOR EACH ROW
WHEN (NEW.ChangeType='O' OR NEW.ChangeType='F')
DECLARE
  CELLA NUMBER;
  N_TEL_ATTIVI_MAX NUMBER;
BEGIN
  --Trovo l'ID della cella in cui si trova il mio cellulare
  SELECT CellID, MaxCalls INTO CELLA, N_TEL_ATTIVI_MAX
  FROM CELL
  WHERE x0<=:NEW.x AND x1>=:NEW.x AND
        y0<=:NEW.y AND y1>=:NEW.y;

  --Telefono acceso
  IF(:NEW.ChangeType='O') THEN
    --Inserisco il nuovo cellulare nella tabella
    INSERT INTO TELEPHONE(PhoneNo, x, y, PhoneState)
    VALUES(:NEW.PhoneNo, :NEW.x, :NEW.y, :NEW.ChangeType);
    --Aggiorno la cella in cui si trova il cellulare
    UPDATE CELL SET CurrentPhone#=CurrentPhone#+1
    WHERE CellID=CELLA;
  END IF;

  --Telefono spento
  IF(:NEW.ChangeType='F') THEN
    --Rimozione telefono da tebella
    DELETE FROM TELEPHONE WHERE PhoneNo=:NEW.PhoneNo;
    --Aggiorno la cella in cui si trova il cellulare
    UPDATE CELL SET CurrentPhone#=CurrentPhone#-1
    WHERE CellID=CELLA;
  END IF;
END;
```

An SQL query using *listings*

```
CREATE OR REPLACE TRIGGER Tel_On_Off
AFTER INSERT ON STATE_CHANGE
FOR EACH ROW
WHEN (NEW.ChangeType='O' OR NEW.ChangeType='F')
DECLARE
  CELLA NUMBER;
  N_TEL_ATTIVI_MAX NUMBER;
BEGIN
  --Trovo l'ID della cella in cui si trova il mio
     cellulare
  SELECT CellID, MaxCalls INTO CELLA, N_TEL_ATTIVI_MAX
  FROM CELL
  WHERE x0<=:NEW.x AND x1>=:NEW.x AND
        y0<=:NEW.y AND y1>=:NEW.y;

  --Telefono acceso
  IF(:NEW.ChangeType='O') THEN
    --Inserisco il nuovo cellulare nella tabella
    INSERT INTO TELEPHONE(PhoneNo, x, y, PhoneState)
    VALUES(:NEW.PhoneNo, :NEW.x, :NEW.y,
       :NEW.ChangeType);
    --Aggiorno la cella in cui si trova il cellulare
    UPDATE CELL SET CurrentPhone#=CurrentPhone#+1
    WHERE CellID=CELLA;
  END IF;

  --Telefono spento
  IF(:NEW.ChangeType='F') THEN
    --Rimozione telefono da tebella
    DELETE FROM TELEPHONE WHERE PhoneNo=:NEW.PhoneNo;
    --Aggiorno la cella in cui si trova il cellulare
    UPDATE CELL SET CurrentPhone#=CurrentPhone#-1
    WHERE CellID=CELLA;
  END IF;
END;
```

A C# chunk of code using *minted*

```
using System;
using System.Runtime.InteropServices;

namespace Binarysharp.MemoryManagement.Memory
{
    /// <summary>
    /// Class representing a block of memory allocated in the
     ↪  local process.
    /// </summary>
    public class LocalUnmanagedMemory : IDisposable
    {
        #region Properties
        /// <summary>
        /// The address where the data is allocated.
        /// </summary>
        public IntPtr Address { get; private set; }
        /// <summary>
        /// The size of the allocated memory.
        /// </summary>
        public int Size { get; private set; }
        #endregion

        #region Constructor/Destructor
        /// <summary>
        /// Initializes a new instance of the <see
         ↪  cref="LocalUnmanagedMemory"/> class, allocating a
         ↪  block of memory in the local process.
        /// </summary>
        /// <param name="size">The size to allocate.</param>
        public LocalUnmanagedMemory(int size)
        {
            // Allocate the memory
            Size = size;
            Address = Marshal.AllocHGlobal(Size);
        }
        /// <summary>
        /// Frees resources and perform other cleanup operations
         ↪  before it is reclaimed by garbage collection.
        /// </summary>
        ~LocalUnmanagedMemory()
        {
            Dispose();
        }
        #endregion

        #region Methods
        #region Dispose (implementation of IDisposable)
        /// <summary>
        /// Releases the memory held by the <see cref="LocalUnmanagedMemory"/> object.
        /// </summary>
        public virtual void Dispose()
        {
            // Free the allocated memory
            Marshal.FreeHGlobal(Address);
            // Remove the pointer
            Address = IntPtr.Zero;
            // Avoid the finalizer
            GC.SuppressFinalize(this);
        }
        #endregion
        #region Read
        /// <summary>
        /// Reads data from the unmanaged block of memory.
        /// </summary>
        /// <typeparam name="T">The type of data to return.</typeparam>
        /// <returns>The return value is the block of memory casted in the specified type.</returns>
        public T Read<T>()
        {
            // Marshal data from the block of memory to a new allocated managed object
            return (T)Marshal.PtrToStructure(Address, typeof(T));
        }
        /// <summary>
        /// Reads an array of bytes from the unmanaged block of memory.
        /// </summary>
        /// <returns>The return value is the block of memory.</returns>
        public byte[] Read()
        {
            // Allocate an array to store data
            var bytes = new byte[Size];
            // Copy the block of memory to the array
            Marshal.Copy(Address, bytes, 0, Size);
            // Return the array
            return bytes;
        }
        #endregion
    }
}
```

A C# chunk of code using *listings*

```csharp
using System;
using System.Runtime.InteropServices;

namespace Binarysharp.MemoryManagement.Memory
{
    /// <summary>
    /// Class representing a block of memory allocated in the local process.
    /// </summary>
    public class LocalUnmanagedMemory : IDisposable
    {
        #region Properties
        /// <summary>
        /// The address where the data is allocated.
        /// </summary>
        public IntPtr Address { get; private set; }
        /// <summary>
        /// The size of the allocated memory.
        /// </summary>
        public int Size { get; private set; }
        #endregion

        #region Constructor/Destructor
        /// <summary>
        /// Initializes a new instance of the <see cref="LocalUnmanagedMemory"/> class, allocating a block of memory in the local process.
        /// </summary>
        /// <param name="size">The size to allocate.</param>
        public LocalUnmanagedMemory(int size)
        {
            // Allocate the memory
            Size = size;
            Address = Marshal.AllocHGlobal(Size);
        }
        /// <summary>
        /// Frees resources and perform other cleanup operations before it is reclaimed by garbage collection.
        /// </summary>
        ~LocalUnmanagedMemory()
        {
            Dispose();
        }
        #endregion

        #region Methods
        #region Dispose (implementation of IDisposable)
        /// <summary>
        /// Releases the memory held by the <see cref="LocalUnmanagedMemory"/> object.
        /// </summary>
        public virtual void Dispose()
        {
            // Free the allocated memory
            Marshal.FreeHGlobal(Address);
            // Remove the pointer
            Address = IntPtr.Zero;
            // Avoid the finalizer
            GC.SuppressFinalize(this);
        }
        #endregion
        #region Read
        /// <summary>
        /// Reads data from the unmanaged block of memory.
        /// </summary>
        /// <typeparam name="T">The type of data to return.</typeparam>
        /// <returns>The return value is the block of memory casted in the specified type.</returns>
        public T Read<T>()
        {
            // Marshal data from the block of memory to a new allocated managed object
            return (T)Marshal.PtrToStructure(Address, typeof(T));
        }
        /// <summary>
        /// Reads an array of bytes from the unmanaged block of memory.
        /// </summary>
        /// <returns>The return value is the block of memory.</returns>
        public byte[] Read()
        {
            // Allocate an array to store data
            var bytes = new byte[Size];
            // Copy the block of memory to the array
            Marshal.Copy(Address, bytes, 0, Size);
            // Return the array
            return bytes;
        }
```

```
            #endregion
        }
}
```