INAP | CAMPUS VIRTUAL

Programación II

Ministerio de Modernización
Presidencia de la Nación

Módulo III

Programación II

Módulo III: Punteros

Módulo III:

- **¿Qué es un puntero?**
- **Zonas inseguras: Unsafe.**
- **Uso básico de punteros.**
- **Aritmetica de punteros.**
- **La palabra FIXED.**
- **Reservar espacio: stackalloc.**

Los punteros en C#

¿Qué es un puntero?

La palabra "puntero" se usa para referirse a una **dirección de memoria**. Lo que tiene de especial es que normalmente un puntero tendrá un tipo de datos asociado: por ejemplo, un "puntero a entero" será una dirección de memoria en la que habrá almacenado (o podremos almacenar) un número entero.

El hecho de poder acceder directamente al contenido de ciertas posiciones de memoria da una mayor versatilidad a un programa, porque permite hacer casi cualquier cosa, pero a cambio de un riesgo de errores mucho mayor.

En lenguajes como C, es imprescindible utilizar punteros para poder crear estructuras dinámicas, pero en C# podemos "esquivarlos", dado que tenemos varias estructuras dinámicas ya creadas como parte de las bibliotecas auxiliares que acompañan al lenguaje básico. Aun así, veremos algún ejemplo que nos muestre qué es un puntero y cómo se utiliza.

En primer lugar, comentemos la sintaxis básica que utilizaremos:

```
int numero;          /* "numero" es un número entero */
int* posicion;       /* "posicion" es un "puntero a entero" (dirección de
                        memoria en la que podremos guardar un entero) */
```

Es decir, escribiremos un asterisco entre el tipo de datos y el nombre de la variable. Ese asterisco puede ir junto a cualquiera de ambos, por lo que también es correcto escribir

```
int *posicion;
```

El valor que guarda "posicion" es una **dirección de memoria**. Generalmente no podremos hacer cosas como posicion=5; porque nada nos garantiza que la posición 5 de la memoria esté disponible para que nosotros la usemos. Será más habitual que tomemos una dirección de memoria que ya contiene otro dato, o bien que le pidamos al compilador que nos reserve un espacio de memoria (más adelante veremos cómo).

Si queremos que "posicion" contenga la dirección de memoria que el compilador había reservado para la variable "numero", lo haríamos usando el símbolo "&", así:

```
posicion = &numero;
```

Zonas "inseguras": unsafe

Como los punteros son "peligrosos" (es frecuente que den lugar a errores muy difíciles de encontrar), el compilador nos obligamos a que le digamos que sabemos que esa zona de programa no es segura, usando la palabra "unsafe":

```
unsafe static void pruebaPunteros() { ...
```

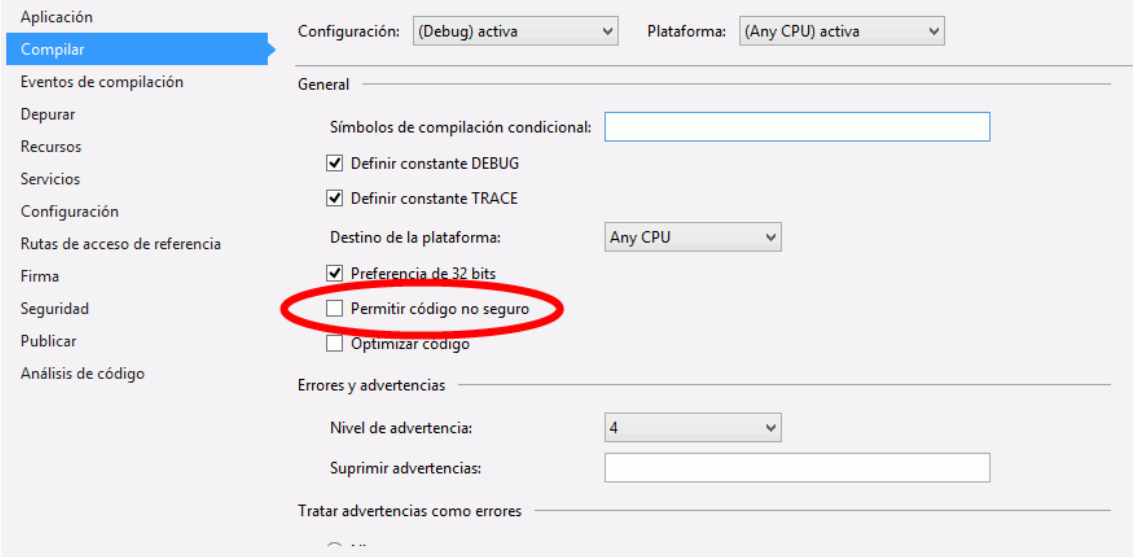
Es más, si intentamos compilar obtendremos un mensaje de error, que nos dice que si nuestro código no es seguro, deberemos compilarlo con la opción "unsafe":

```
mcs unsafe1.cs
unsafe1.cs(15,31): error CS0227: Unsafe code requires the `unsafe' command
line
option to be specified
Compilation failed: 1 error(s), 0 warnings
```

Por tanto, deberemos compilar con la opción /unsafe como forma de decir al compilador "sí, sé que este programa tiene zonas no seguras, pero aun así quiero compilarlo":

```
mcs unsafe1.cs /unsafe
```

En Visual Studio, tendremos que ir a las "propiedades del proyecto" (típicamente estará al final del menú "Proyecto") y decir que queremos "Permitir código no seguro":



Uso básico de punteros

Veamos un ejemplo básico de cómo dar valor a un puntero y de cómo guardar un valor en él:

```
using System;

public class Ejemplo_11_09_03a
{
    private unsafe static void pruebaPunteros()
    {
        int* punteroAEntero;
        int x;

        // Damos un valor a x
        x = 2;
        // punteroAEntero será la dirección de memoria de x
        punteroAEntero = &x;
        // Los dos están en la misma dirección:
        Console.WriteLine("x vale {0}", x);
        Console.WriteLine("En punteroAEntero hay un {0}", *punteroAEntero);

        // Ahora cambiamos el valor guardado en punteroAEntero
        *punteroAEntero = 5;
        // Y x se modifica también:
        Console.WriteLine("x vale {0}", x);
        Console.WriteLine("En punteroAEntero hay un {0}", *punteroAEntero);
    }

    public static void Main()
    {
        pruebaPunteros();
    }
}
```

La salida de este programa es:

```
x vale 2
En punteroAEntero hay un 2
x vale 5
En punteroAEntero hay un 5
```

Es decir, cada cambio que hacemos en "x" se refleja en "punteroAEntero" y viceversa.

Aritmética de punteros

Si aumentamos o disminuimos el valor de un puntero, cambiará la posición que representa... pero no cambiará de uno en uno, sino que saltará a la siguiente posición capaz de almacenar un dato como el que corresponde a su tipo base. Por ejemplo, si un puntero a entero tiene como valor 40.000 y hacemos "puntero++", su dirección pasará a ser 40.004 (porque cada entero ocupa 4 bytes). Vamos a verlo con un ejemplo:

```
using System;

public class Ejemplo_11_09_06a
{
    public unsafe static void Main()
    {
        const int tamanyoArray = 5;
        int* datos = stackalloc int[tamanyoArray];
        int* posicion = datos;

        Console.WriteLine("Posicion actual: {0}", (int) posicion);

        // Ponemos un 0 en el primer dato
        *datos = 0;

        // Rellenamos los demás con 10,20,30...
        for (int i = 1; i < tamanyoArray; i++)
        {
            posicion++;
            Console.WriteLine("Posicion actual: {0}", (int) posicion);
            *posicion = i * 10;
        }

        // Finalmente mostramos el array
        Console.WriteLine("Contenido:");
        for (int i = 0; i < tamanyoArray; i++)
        {
            Console.WriteLine(datos[i]);
        }
    }
}
```

El resultado sería algo parecido (porque las direcciones de memoria que obtengamos no tienen por qué ser las mismas) a:

```
Posicion actual: 1242196
Posicion actual: 1242200
Posicion actual: 1242204
Posicion actual: 1242208
Posicion actual: 1242212
Contenido:
0
10
20
30
40
```

La palabra "fixed"

C# cuenta con un "recolector de basura", que se encarga de liberar el espacio ocupado por variables que ya no se usan. Esto puede suponer algún problema cuando usamos variables dinámicas, porque estemos accediendo a una posición de memoria que el entorno de ejecución haya previsto que ya no necesitaríamos… y haya borrado.

Por eso, en ciertas ocasiones el compilador puede avisarnos de que algunas asignaciones son peligrosas y obligar a que usemos la palabra "fixed" para indicar al compilador que esa zona "no debe limpiarse automáticamente".

Esta palabra se usa antes de la declaración y asignación de la variable, y la zona de programa que queremos "bloquear" se indica entre llaves:

```
using System;

public class Ejemplo_11_09_07a
{
    public unsafe static void Main()
    {
        int[] datos={10,20,30};

        Console.WriteLine("Leyendo el segundo dato...");
        fixed (int* posicionDato = &datos[1])
        {
            Console.WriteLine("En posicionDato hay {0}",

                *posicionDato);
        }

        Console.WriteLine("Leyendo el primer dato...");
        fixed (int* posicionDato = datos)
        {
            Console.WriteLine("Ahora en posicionDato hay {0}",
                *posicionDato);
        }
    }
}
```

Como se ve en el programa anterior, en una zona "fixed" no se puede modificar el valor de esa variables; si lo intentamos recibiremos un mensaje de error que nos avisa de que esa variable es de "sólo lectura" (read-only). Por eso, para cambiarla, tendremos que empezar otra nueva zona "fixed".

El resultado del ejemplo anterior sería:

```
Leyendo el segundo dato...
En posicionDato hay 20
Leyendo el primer dato...
Ahora en posicionDato hay 10
```

Reservar espacio: stackalloc

Podemos reservar espacio para una variable dinámica usando "stackalloc". Por ejemplo, una forma alternativa de crear un array de enteros sería ésta:

```
int* datos = stackalloc int[5];
```

Un ejemplo completo podría ser:

```
using System;

public class Ejemplo_11_09_05a
{
    public unsafe static void Main()
    {
        const int tamanyoArray = 5;
        int* datos = stackalloc int[tamanyoArray];

        // Rellenamos el array
        for (int i = 0; i < tamanyoArray; i++)
        {
            datos[i] = i*10;
        }

        // Mostramos el array
        for (int i = 0; i < tamanyoArray; i++)
        {
            Console.WriteLine(datos[i]);
        }
    }
}
```

Existen ciertas diferencias entre esta forma de trabajar y la que ya conocíamos: la memoria se reserva en la pila (stack), en vez de hacerlo en la zona de memoria "habitual", conocida como "heap" o montón, pero es un detalle sobre el que no vamos a profundizar.