Advances in Programming Languages: Garbage collection and stack allocation

Stephen Gilmore
The University of Edinburgh

January 24, 2007

Garbage collectors and programming languages

Certain programming languages, such as C, do not provide a garbage collector. We must do all of the disposal of moribund addresses ourselves. Other programming languages, such as Java, provide a garbage collector. We cannot dispose of memory: the run-time is in charge. What of *mixed economy* languages such as C# which allow us to mix *managed* and *unmanaged* data?

C# asks us to separate methods into *safe* and *unsafe* categories. It might seem a bit odd to explicitly include unsafe features in a programming language (usually they make their way in by accident).

The reasoning for this with regard to the C# language is firstly that *it will sometimes be necessary to call unsafe code*, so this has to be achievable somehow. Secondly, *it should be more efficient to call unsafe C# code than to call out to (unsafe) C code.* Java native interface calls are known to have a high run-time performance overhead.

Unsafe methods

Methods in C# must be marked as *unsafe* if they need to perform C-style pointer operations. Statement blocks and even single statements can be marked as unsafe.

Inside an unsafe block we may take the address of a variable (using `&`), dereference pointers and perform pointer arithmetic. We can implement directly C-style idioms such as simulating call-by-reference by passing a C-style (thin) `*`-pointer to a function, sharing that variable between the caller and the callee.

During garbage collection, the language run-time will move objects around in memory to reduce the amount of wasted space between allocated objects (this is called *compaction*). If we hold the address of a managed object in a pointer the garbage collector could invalidate this address simply by moving the object which our pointer refers to. For this reason, pointers are used in *fixed* statements in C#. A fixed statement *pins* an object in memory so that the collector cannot move it around.

```
// File: programs/cs/Fixed.cs
// compile with: mcs --unsafe Fixed.cs
using System;

class IntegerRef {
  public int x;
}

class FixedTest {
```

```
  // unsafe method: takes pointer to int
  unsafe static void SquarePtrParam (int* p) {
    *p *= *p;
  }

  unsafe public static void Main() {
    IntegerRef i = new IntegerRef();
    i.x = 5;
    // pin object in place: take address
    fixed (int* p = &i.x) {
      SquarePtrParam (p);
    }
    // object is now unpinned
    Console.WriteLine ("i.x is {0}", i.x);
  }
}
```

To maintain run-time performance objects should be pinned only briefly.

Stack allocation

An alternative to pinning objects is to allocate them on the stack instead of in the heap. The run-time stack contains storage areas in each stack frame. The stack is not garbage-collected; all of the data in the run-time stack is assumed to be live. Stack allocated objects are freed when their owning method exits.

C# allows us to allocate objects on the stack in unsafe code, and to manipulate their addresses. Since these objects will not be moved by the garbage collector they do not need to be pinned.

```
// File: programs/cs/StackAlloc.cs
// compile with the ''unsafe'' flag
using System;
class StackAlloc {
  public static unsafe void Main() {
    int* fib = stackalloc int[100];
    int* p = fib;
    *p++ = *p++ = 1;
    for (int i=2; i<100; ++i, ++p)
      *p = p[-1] + p[-2];
    for (int i=0; i<10; i++)
      Console.WriteLine (fib[i]);
  }
}
```

The line

```
*p++ = *p++ = 1;
```

used in the previous program seems rather obscure. It is a common C programming idiom, but what is it doing? Firstly, it is a compound assignment, so it has the same meaning as the following.

```
*p++ = (*p++ = 1);
```

So, two variables are being assigned the value 1, but which variables?

Firstly, `*p++` means "increment `p`", not "increment what `p` points to". Secondly, the variable which is assigned to is the one which `p` points to *before* it is updated, not after. Thus the above line of code could have been written as follows.

```
*p = 1;
p++;
*p = 1;
p++;
```

There are constraints on the use of stack allocation. A stack allocation may only be used as the initialiser of a local variable of a method.

The previous program demonstrates the simplest case of stack allocation; the size of the array to be allocated is determined by a compile-time constant. The case when the size of the array to be allocated depends on a run-time value is more complex.

We now factor out the Fibonacci function into a separate method.

```
// File: programs/cs/DynamicStackAlloc.cs
class DynamicStackAlloc {
  unsafe static int Fib (int n) {
    int* fib = stackalloc int[n+2];
    int* p = fib;
    *p++ = *p++ = 1;
    for (int i=2; i<=n; ++i, ++p)
      *p = p[-1] + p[-2];
    return fib[n];
  }
  public static unsafe void Main() {
    for (int i=0; i<10; i++)
      System.Console.WriteLine (Fib(i));
  }}
```

I compiled this program with the Ximian Mono C# compiler, version 1.1.13.7 (using `mcs -warn:4 -unsafe`) and, under Windows, with VisualStudio 2005. Both execute successfully, producing the Fibonacci numbers.

```
[scap]stg: mono DynamicStackAlloc.exe
1
1
2
3
5
8
13
21
34
55
```

Heap allocation

In object-oriented languages most dynamic data structures are allocated on the (garbage collected) heap. Such data stuctures have addresses which can be taken in `fixed` statements in C#. Thus we can view an `int` array using an `int` pointer, simply by taking the address of the first element in the array. Use of pointers in C# needs to be marked as unsafe, as before.

```
// File: programs/cs/DynamicHeapAlloc.cs
using System;
class DynamicHeapAlloc {
   unsafe static int Fib (int n) {
      int[] fib = new int[n+1];
      // An array address can be used in the
      // initialiser in a C# fixed statement
      fixed (int* p = fib) { // or: p = &fib[0]
        p[0] = (p[1] = 1); // cannot assign to (fixed) p
        for (int i=2; i<=n; i++)
          p[i] = p[i-1] + p[i-2];
      }
      return fib[n];
   }

   // Main method same as before
   public static unsafe void Main() {
      for (int i=0; i<10; ++i)
         Console.WriteLine (Fib(i));
   }
}
```

This program compiles and runs as expected, producing the first ten Fibonacci numbers. There is no real reason to use unsafe code in the above example, so we can remove the use of the `int` pointer in the above example.

```
// File: programs/cs/SafeHeapAlloc.cs
class SafeHeapAlloc { // all unsafe code removed
   static int Fib (int n) {
      int[] fib = new int[n+1];
      fib[0] = (fib[1] = 1);
      for (int i=2; i<=n; i++)
        fib[i] = fib[i-1] + fib[i-2];
      return fib[n];
   }
   public static void Main() {
      for (int i=0; i<10; i++)
         System.Console.WriteLine (Fib(i));
   }
}
```

This program fails at run-time. Under Windows XP:

```
Unhandled Exception: System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
```

```
   at SafeHeapAlloc.Fib(Int32 n) in
     C:\SafeHeapAlloc.cs:line 6
   at SafeHeapAlloc.Main() in
     C:\SafeHeapAlloc.cs:line 14
```

Under Linux:

```
Unhandled Exception: System.IndexOutOfRangeException:
  Array index is out of range.
in <0x00030> SafeHeapAlloc:Fib (Int32 n)
in <0x00015> SafeHeapAlloc:Main ()
```

The reason why this happens is that we have re-introduced array bounds checking and trapped an error which had gone uncaught in the unsafe version which used an integer pointer in a `fixed` block.

On the first call to the `Fib` method, the parameter has the value zero, so inside the method we allocate an integer array of size one (`n+1`, `n` being zero). When we access the location `fib[1]` to assign it the value one, this is an out-of-bounds violation. When we reference this array via the pointer `p` we do not invoke bounds checking, so this violation is not detected at run-time. That is to say, we are really using C-like rules when we write unsafe C# code. We will receive fewer warnings from the compiler and run-time errors may or may not be trapped.

The version of this program which follows avoids the out-of-bounds violation by checking that the parameter `n` is at least two before allocating the array.

```
// File: programs/cs/SaferHeapAlloc.cs
class SaferHeapAlloc {
  // all unsafe code removed
  // bug for low values of n fixed
  static int Fib (int n) {
    if (n < 2) return 1;
    int[] fib = new int[n+1];
    // the access to fib[1] below is safe: the
    // array has at least three elements
    fib[0] = (fib[1] = 1);
    for (int i=2; i<=n; i++)
     fib[i] = fib[i-1] + fib[i-2];
    return fib[n];
  }

  public static void Main() {
    for (int i=0; i<10; i++)
      System.Console.WriteLine (Fib(i));
  }
}
```

This program compiles and runs as expected, producing the first ten Fibonacci numbers.

Unsafe code and addresses

Now that we know how to use addresses and pointers in C# we can go back to revisit some C idioms which we implemented previously and investigate their behaviour in C#.

One concern that we had in C was that taking the address of a local variable could expose *stale addresses* which could be overwritten at any later time. When we used the `&` operator in C to take the address of a local variable GCC warned us about this.

```
// File: programs/cs/StaleAddresses.cs
using System;
class StaleAddresses {
  public static unsafe int* makeCounter() {
    int counter = 0;
    return &counter; // return address of local variable
  }
  public static unsafe void Main() {
    int* p;
    p = makeCounter();
    Console.WriteLine("*p is {0}", *p);
    Environment.ExitCode = 0;
  }
}
```

When compiled with `mcs -warn:4 -unsafe` (or the same settings for VisualStudio 2005) the C# compiler *does not* warn us that the address of a local variable is returned from a function. This is a fault which even the C compiler warns about. Under Linux the results are unpredictable.

```
[scap]stg: mono StaleAddresses.exe
*p is -1080615424
[scap]stg: mono StaleAddresses.exe
*p is -1081838688
[scap]stg: mono StaleAddresses.exe
*p is -1080574240
```

Under Windows XP we have a different result.

```
C:> bin\Debug\StaleAddresses.exe
*p is 0
```

We introduce an intermediate function as before.

```
// File: programs/cs/StaleAddresses2.cs
using System;
class StaleAddresses2 {

  public static unsafe int* makeCounter() {
    int counter = 0;
    return &counter; // return address of local variable
  }

  // no use of pointers or addresses in this function
  public static void dummy() {
    int x = 13; // simple integer variable
```

```
  }

  public static unsafe void Main() {
    int* p;
    p = makeCounter();
    Console.WriteLine("*p is {0}", *p);
    dummy();
    Console.WriteLine("*p is {0}", *p);
    Environment.ExitCode = 0;
  }
}
```

For this code both the Mono and VisualStudio compiler warn us that variables are assigned but their values are never used (which is a good warning) but neither compiler warns us that the address of a local variable is returned from a function.

Again, the Linux and Windows versions differ. On Linux:

```
[scap]stg: mono StaleAddresses2.exe
*p is -1075549920
*p is -1075549920
[scap]stg: mono StaleAddresses2.exe
*p is -1079301072
*p is -1079301072
[scap]stg: mono StaleAddresses2.exe
*p is -1077084272
*p is -1077084272
```

On Windows:

```
C:> bin\Debug\StaleAddresses2.exe
*p is 0
*p is 1242780
C:> bin\Debug\StaleAddresses2.exe
*p is 0
*p is 1242780
```

Unsafe contexts

The unsafe statements which we have seen in C# are part of a larger category in the language specification known as *unsafe contexts*.

As an example of an unsafe context which is not a statement, the declaration of a class may use the `unsafe` modifier along with other modifiers. In this case the entire textual extent of the class declaration is considered to be an unsafe context. Thus, for example, a heap-allocated object of a C# class can have an `int *` field.

```
// File: programs/cs/StaleAddresses3.cs
using System;
class StaleAddresses3 {

  // An unsafe context
  public unsafe class Counter {
```

```
    // An integer pointer
    public int* p;

    public Counter(int* p) {
      this.p = p;
    }
  }

  public static unsafe Counter makeCounter() {
    int counter = 0;
    // return address of local variable packaged
    // in an instance of the Counter class
    return new Counter(&counter);
  }

  // no use of pointers or addresses in this function
  public static void dummy() {
    // simple integer variable
    int x = 13;
  }

  public static unsafe void Main() {
    Counter c;
    int x = 15;
    int* p = &x;
    c = new Counter(p);
    Console.WriteLine("*p is {0}", *c.p);
    c = makeCounter();
    Console.WriteLine("*p is {0}", *c.p);
    dummy();
    Console.WriteLine("*p is {0}", *c.p);
    Environment.ExitCode = 0;
  }
}
```

As might be expected, the behaviour of this program is unsure. On Linux:

```
[scap]stg: mono StaleAddresses3.exe
*p is 15
*p is 5429332
*p is 5429332
[scap]stg: mono StaleAddresses3.exe
*p is 15
*p is 5785684
*p is 5785684
```

On Windows:

```
C:> bin\Debug\StaleAddresses3.exe
*p is 15
*p is 0
```

```
*p is 2045189024
C:> bin\Debug\StaleAddresses3.exe
*p is 15
*p is 0
*p is 2045189024
```

Summary

- C#, unlike other languages, asks programmers to classify their methods as safe or unsafe.
- Safe code manipulates managed objects, includes bounds checks and interacts well with the garbage collector.
- Unsafe code manipulates unmanaged objects, interacts poorly with the garbage collector, misses problems which the C compiler warns about, re-introduces problems known from C such as undetected out-of-bounds violations and produces non-portable programs with unpredictable results.
- Using unsafe code as little as possible seems to be a good approach.