11 The Machine Model: Stack, Heap, and Garbage Collection

The machine model describes how data are stored and manipulated at run-time, during the execution of a C# program. It has two main components:

- The *stack*, which is divided into *frames*. A frame has room for the local variables and parameters of one invocation of a method. A new frame is created when the method is called, and thrown away when the method returns. Thus at any point in time, the frames on the stack correspond to those methods that have been called but have not yet returned.
- The *heap* stores objects, arrays and delegates created by the evaluation of `new`-expressions. These data are automatically removed from the heap by a *garbage collector* when they are no longer needed by the program. Then the space can be reused for other data. There is no way to explicitly remove anything from the heap, not even by calling a so-called destructor (which is not the same as in C++).

In fact, there is a separate stack for each thread (section 20), but only one heap shared among all threads.

The various values used in C# are stored as follows:

- All objects, arrays and delegates are stored in the heap. If an array has element type `t` which is a simple type, then each array element directly contains its value; if `t` is a reference type, then each array element contains either `null` or a reference to an object, array or delegate stored elsewhere in the heap; if `t` is a struct type, then each array element directly contains the struct value, that is, has room for its instance fields.
- Local variables and parameters are stored on the stack. A local variable of simple type directly contains its value. A local variable of reference type contains `null` or a reference to an object or array or delegate stored in the heap. A local variable of struct type directly contains the struct, that is, has room for its instance fields.

11.1 Class and Object versus Struct Type and Struct Value

At run-time, a class is represented by a chunk of storage, set aside when the class is loaded. The representation of a class contains the name of the class and all the static fields of the class; see the illustration in example 48. At run-time, an object (instance of a class) is allocated in the heap and has two parts: an identification of the *class* `C` of the object, which is the class `C` used when creating it; and room for all the instance fields of the object. The class identification is used when evaluating instance test expressions (section 12.11), cast expressions (section 12.18), and array element assignments (section 9.1).

Spring structs over

At run-time, a struct type is represented very much like a class: the representation of a struct type contains the name of the struct and all the static fields of the struct type. At run-time a struct value (instance of a struct type) is allocated on the stack or as part of an array or object in the heap. Like an object the struct value has room for the instance fields of the struct, but unlike an object it contains no indication of the struct type.

Point class on next pdf page

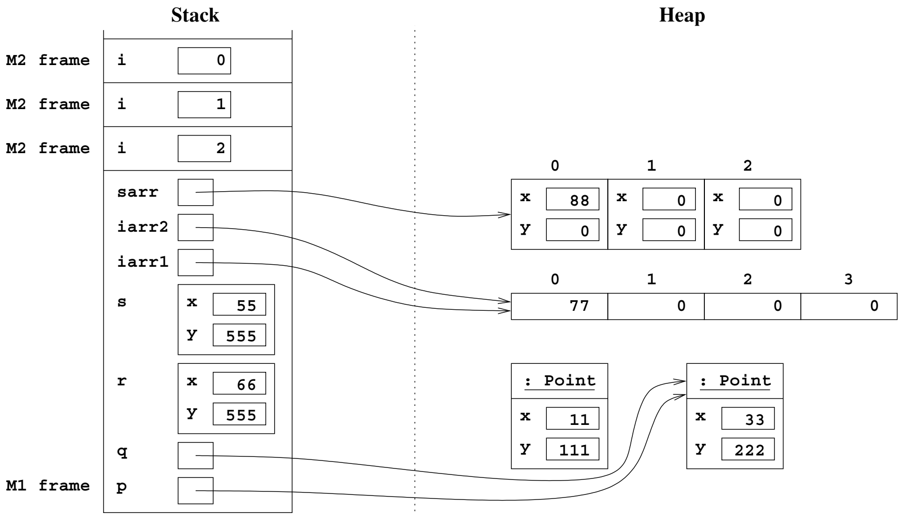
Example 64 Stack, Heap, Objects, Arrays, Structs, and Assignment
The assignment `p = q` of a variable of reference type (class Point, example 40) copies only the reference. Afterwards `p` and `q` point to the same object, so the assignment to `p.x` affects `q.x` also. Likewise for the array assignment `arr2 = arr1`. By contrast, an assignment `r = s` of a variable of struct type (SPoint, example 125) copies the entire struct from `s` to `r`; hence the later assignment to `r.x` does not affect `s.x`.

SPoint - last pdf page

When a method `M2` calls itself recursively, the stack contains a separate copy of local variables and parameters (here just `i`) for each unfinished call of the method:

```
public static void M1() {
  Point p = new Point(11, 111), q = new Point(22, 222);
  p = q;
  p.x = 33;
  SPoint r = new SPoint(44, 444), s = new SPoint(55, 555);
  r = s;
  r.x = 66;
  int[] iarr1 = new int[4];
  int[] iarr2 = iarr1;
  iarr1[0] = 77;
  SPoint[] sarr = new SPoint[3];
  sarr[0].x = 88;
  M2(2);
}
public static void M2(int i) {
  if (i > 0)
    M2(i-1);
}
```

After executing the body of `M1` and three calls of `M2`, just before `M2(0)` returns, the stack contains three frames for `M2`. The left-most Point object in the heap is not referred to by a variable any longer and will be removed by the garbage collector:



Example 40 Class Declaration
The `Point` class is declared to have two instance fields `x` and `y`, one constructor, and two instance methods. It is used in examples 21 and 64.

```
public class Point {
  protected internal int x, y;
  public Point(int x, int y) { this.x = x; this.y = y; }
  public void Move(int dx, int dy) { x += dx; y += dy; }
  public override String ToString() { return "(" + x + ", " + y + ")"; }
}
```

Example 41 Class with Static and Instance Members
The APoint class declares a static field `allpoints` and two instance fields `x` and `y`. Thus each APoint object has its own `x` and `y` fields, but all objects share the same `allpoints` field in the APoint class.

The constructor inserts the new object (`this`) in the ArrayList object `allpoints`. The instance method `GetIndex` returns the point's index in the arraylist. The static method `GetSize` returns the number of APoints created so far. Instead of the methods `GetIndex` and `GetSize` it would be natural to use an instance property and a static property; see section 10.13 and example 59. The static method `GetPoint` returns the `i`'th APoint in the arraylist. The class is used in example 81.

Beware that every APoint created will be reachable from `allpoints`. This prevents automatic recycling (garbage collection) of APoint objects at run-time; objects should not be kept live in this way unless there is a very good reason to do so.

```
public class APoint {
  private static ArrayList allpoints = new ArrayList();
  private int x, y;
  public APoint(int x, int y) {
    allpoints.Add(this); this.x = x; this.y = y;
  }
  public void Move(int dx, int dy) {
    x += dx; y += dy;
  }
  public override String ToString() {
    return "(" + x + ", " + y + ")";
  }
  public int GetIndex() {
    return allpoints.IndexOf(this);
  }
  public static int GetSize() {
    return allpoints.Count;
  }
  public static APoint GetPoint(int i) {
    return (APoint)allpoints[i];
  }
}
```

Example 125 A Struct Type for Points
The SPoint struct type declaration is very similar to the Point class declaration in example 40. The difference is that assigning an SPoint value to a variable copies the SPoint value; see examples 64 and 86. (Thus struct types are most suitable for data structures whose fields are read-only as in example 128.)

```
public struct SPoint {
  internal int x, y;
  public SPoint(int x, int y) { this.x = x; this.y = y; }
  public SPoint Move(int dx, int dy) { x += dx; y += dy; return this; }
  public override String ToString() { return "(" + x + ", " + y + ")"; }
}
```

Example 126 Differences between Struct Values and Objects
An array of structs directly contains the struct values (not references to them), and element assignment copies the struct value. Assigning a struct value to a variable of type Object creates a boxed copy of the struct value on the heap, and assigns the variable a reference to that copy; see section 14.1.

```
SPoint p = new SPoint(11, 22);              // Create a struct value in p
SPoint[] arr = { p, p };                    // Two more copies of p
arr[0].x = 33;
Console.WriteLine(arr[0] + " " + arr[1]);   // Prints (33, 22) (11, 22)
Object o = p;                               // Another copy of p, in heap
p.x = 44;
Console.WriteLine(p + " " + o);             // Prints (44, 22) (11, 22)
Console.WriteLine(o is SPoint);             // Prints True
Console.WriteLine(o is int);                // Prints False
```

Example 127 The `this` Reference in a Struct Type
In a struct type declaration, `this` is assignable like a `ref` parameter, so method `Move` inside struct type SPoint from example 125 can equivalently be written as shown in the first line below. Although this does not create a "new" struct value, it may be less efficient than direct field assignment.

Even when a struct method returns `this`, the method call chain `p.Move(5,5).Move(6,6)` may not have the expected effect. The expression `p.Move(5,5)` is a value, so the second call to `Move` works on a copy of struct `p`, not the struct held in variable `p`. Also, a `readonly` field of struct type is a value, so `q.Move(5,5)` works on a copy of `q` and therefore does not modify the field `q`; see section 14.3.

```
public SPoint Move(int dx, int dy) { return this = new SPoint(x+dx, y+dy); }
...
static readonly SPoint q = new SPoint(33, 44);
public static void Main(String[] args) {
  SPoint p = new SPoint(11, 22);        // Now p = (11, 22)
  p.Move(9,8);                          // Now p = (20, 30)
  p.Move(5,5).Move(6,6);                // Now p = (25, 35) not (31, 41)
  q.Move(5,5);                          // Now q = (33, 44) not (38, 49)
}
```