Not-So Secret Language Features

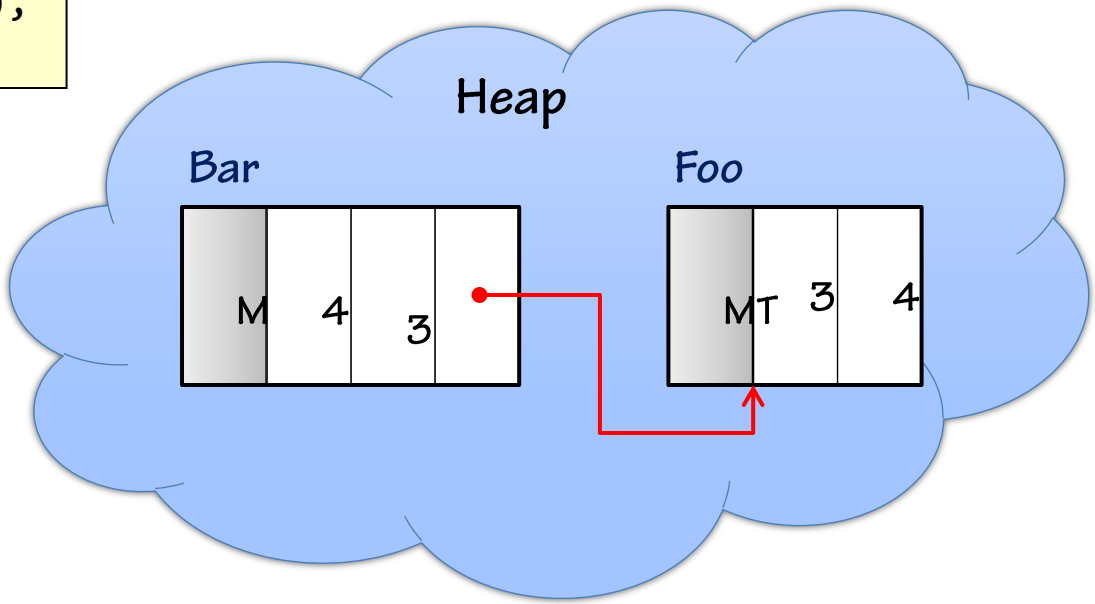
Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

Revisiting the CLR Type System

- **Each addressable location in memory has a type**
  - Reference types
    - Objects living on the heap
    - Payload prefixed with Method Table (MT)
    - Garbage Collector traverses heap using type layout info

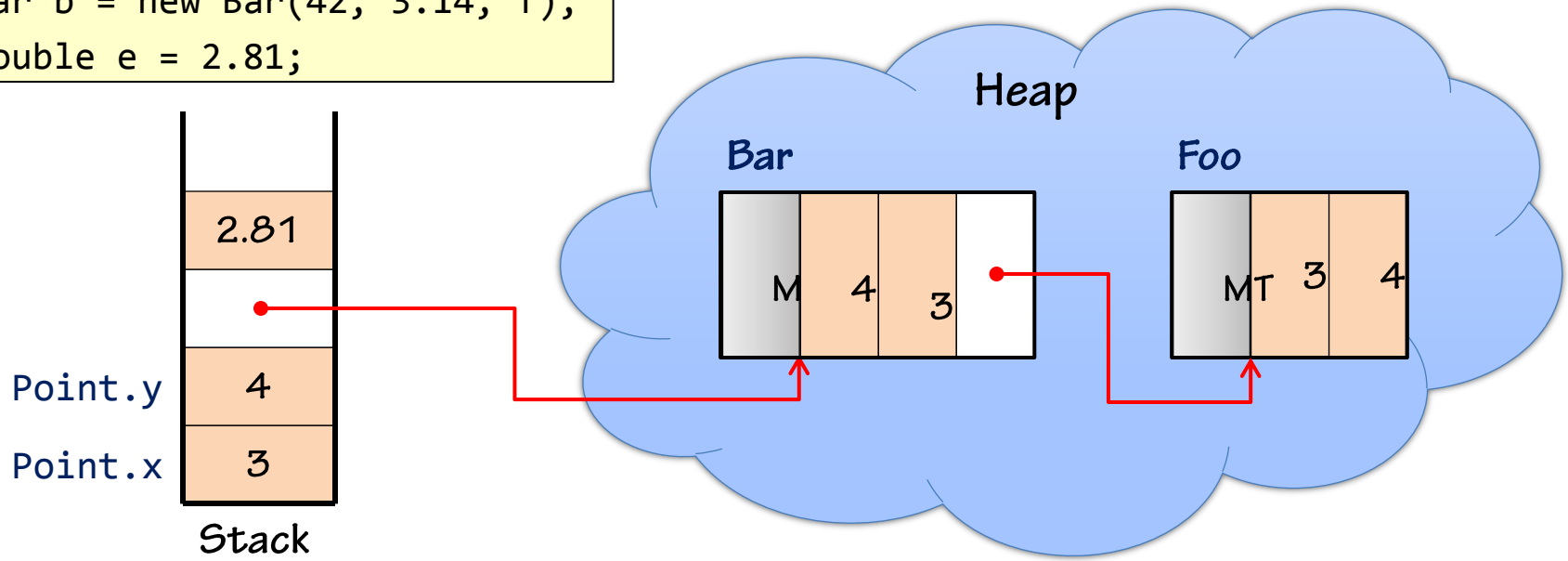
```
Foo f = new Foo(p);
Bar b = new Bar(42, 3.14, f);
```



Revisiting the CLR Type System

- **Each addressable location in memory has a type**
  - Value types
    - Can be interior to other types
    - Can live on the stack (parameters, locals, etc.)
    - Runtime knows layout of locations

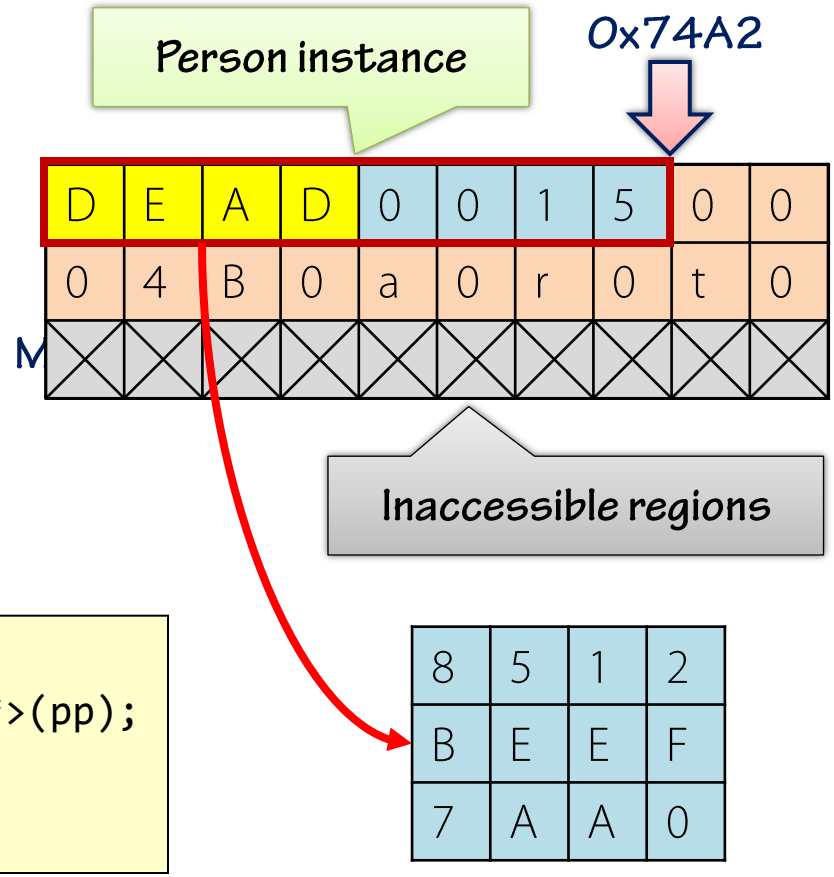
```
Point p = new Point(3, 4);
Foo f = new Foo(p);
Bar b = new Bar(42, 3.14, f);
double e = 2.81;
```



On the Danger of Pointers

- **Safety matters**
  - Memory safety
    - Cannot point to invalid memory
    - E.g. no access violations
  - Type safety
    - Cannot write unexpected data
    - E.g. no cast from Person to Book
- **void* is unsafe**
  - Type information is lost
    - Reference is opaque to runtime
    - Recipient doesn't know what to find

```
string name = "Bart";
int age = 21;
Person p = new Person(name, age);
```




```
Person* pp = &p;
void*   up = static_cast<void*>(pp);
       *up = 0xDEAD;
string hmm = p.Name;
```

Typed References

- **System.IntPtr**
  - Representation of a raw pointer

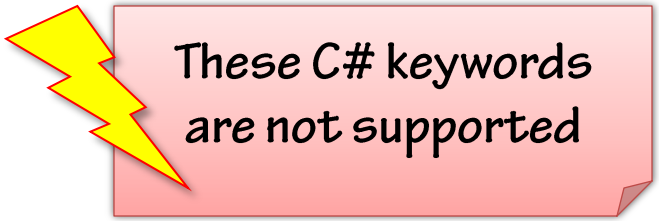
```
public struct IntPtr {
    unsafe void* m_value;
}
```

- **System.TypedReference**
  - Association of a type with a pointer
    - IntPtr to refer to the value
    - IntPtr to refer to a type descriptor

```
public struct TypedReference {
    IntPtr Type;
    IntPtr Value;
}
```

- **Type safety not violated**
  - Reference and type are treated as one
  - GC knows what to find where

- **Operations**
  - Convert between objects and typed references
  - Access the data without causing boxing
  - Some "secret" C# keywords

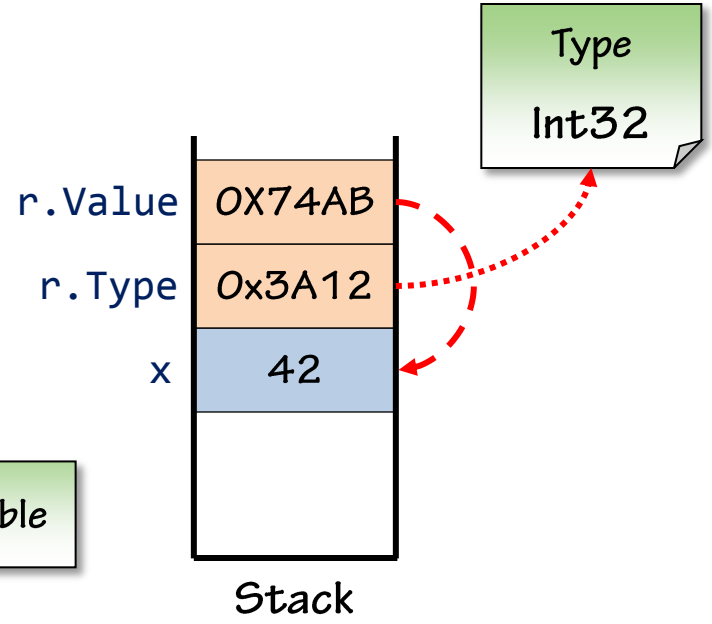


__makeref

- **Create a TypedReference**
  - Similar to "address of" operator & in C/C++

```
void Foo() {
    int x = 42;
    TypedReference r = __makeref(x);
}
```

Must be a variable



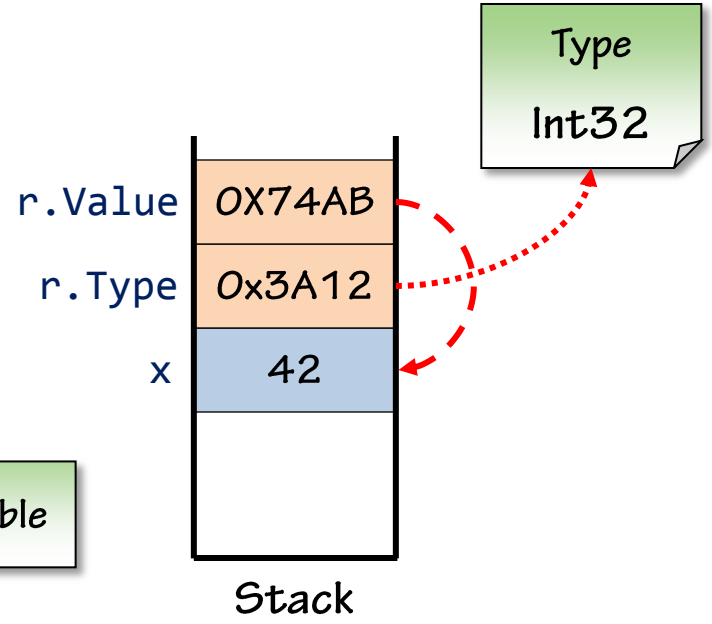
These C# keywords are not supported

__makeref

- **Create a TypedReference**
  - Similar to "address of" operator & in C/C++

```
void Foo() {
    int x = 42;
    TypedReference r = __makeref(x);
}
```

Must be a variable



- **Restrictions on TypedReference values**
  - Cannot be returned to callers

```
TypedReference Qux() {
    int x = 42;
    return __makeref(x);
}
```

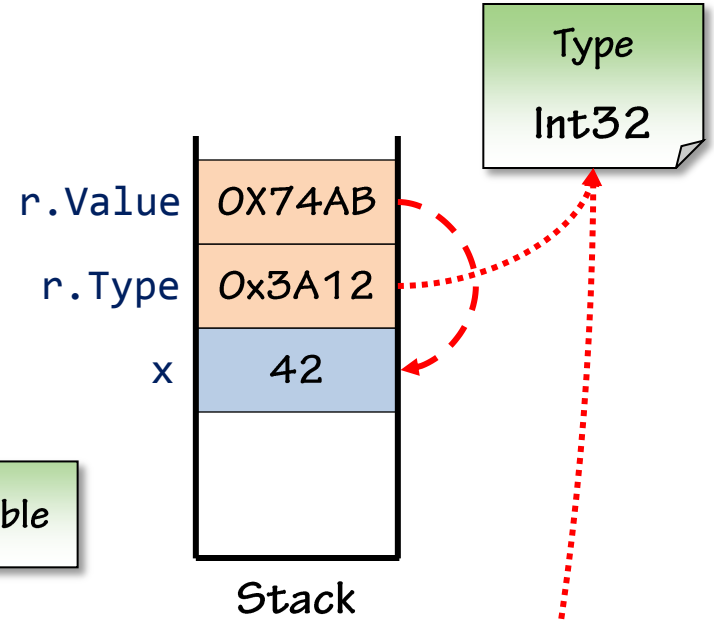
These C# keywords are not supported

__makeref

- **Create a TypedReference**
  - Similar to "address of" operator & in C/C++

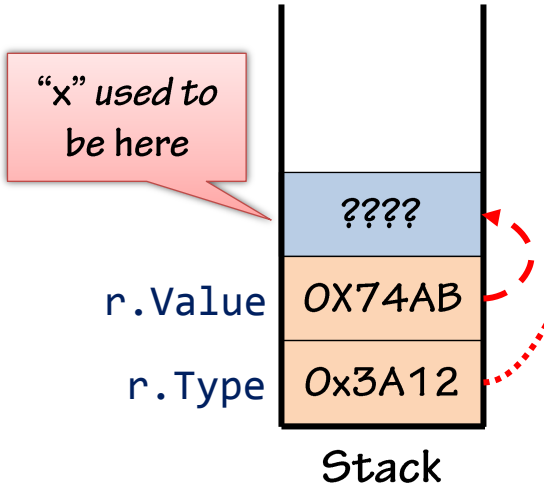
```
void Foo() {
    int x = 42;
    TypedReference r = __makeref(x);
}
```

Must be a variable



- **Restrictions on TypedReference values**
  - Cannot be returned to callers
    - Would refer to memory up the stack
    - Violation of type and memory safety

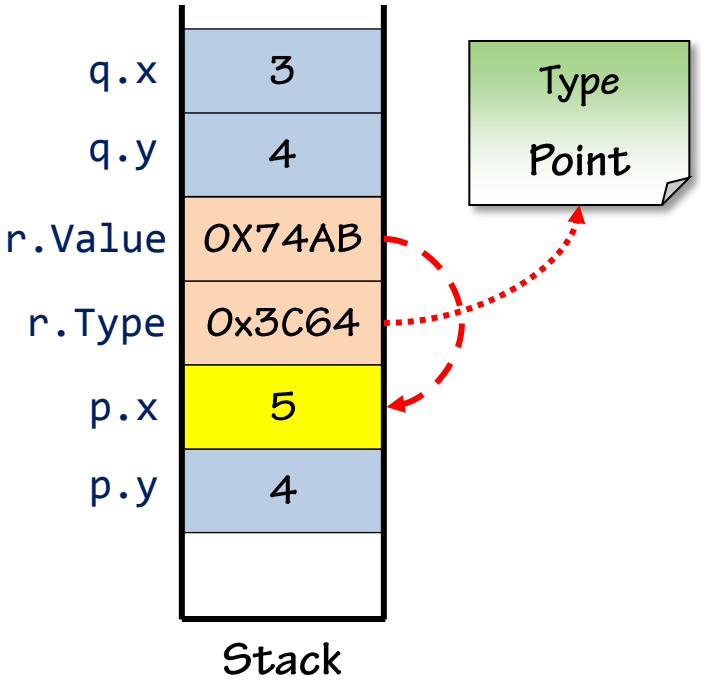
```
TypedReference Qux() {
    int x = 42;
    return __makeref(x);
}
```



__refvalue

- **Dereference a TypedReference**
  - Similar to * and -> in C/C++

```
void Foo() {
    Point p = new Point(3, 4);
    TypedReference r = __makeref(p);
    Point q = __refvalue(r, Point);
    __refvalue(r, Point).x = 5;
}
```



- **Copy-by-value semantics**
  - Target type specified in __refvalue
  - No boxing introduced

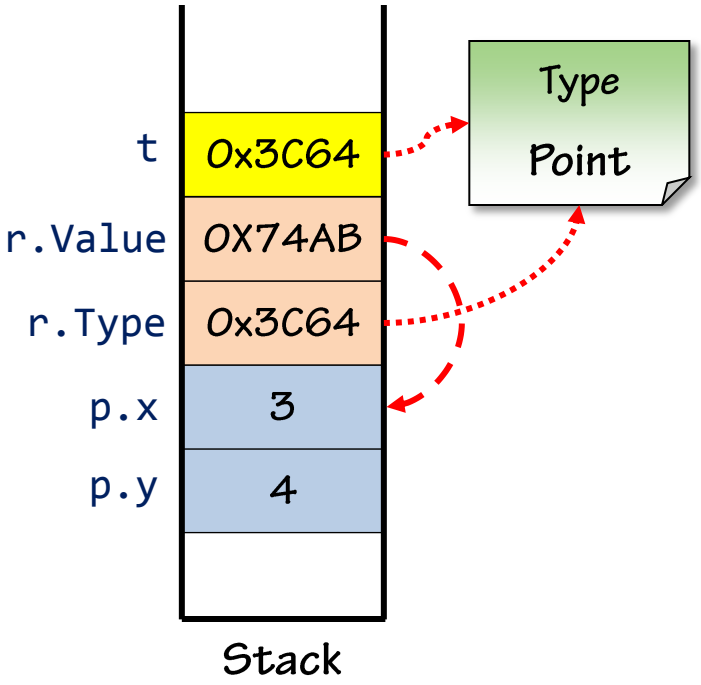
- **TypedReference.ToObject**
  - Returns boxed copy of the value

These C# keywords are not supported

__reftype

- **Gets the System.Type of a TypedReference**
  - Akin to an indirect GetType call

```
void Foo() {
    Point p = new Point(3, 4);
    TypedReference r = __makeref(p);
    Type t = __reftype(r);
}
```



- **Where are these used?**
  - System.TypedReference
    - GetHashCode, etc.
  - Interlocked.CompareExchange<T>
    - JIT intrinsic under the hood

- **Performance**
  - Not so great…

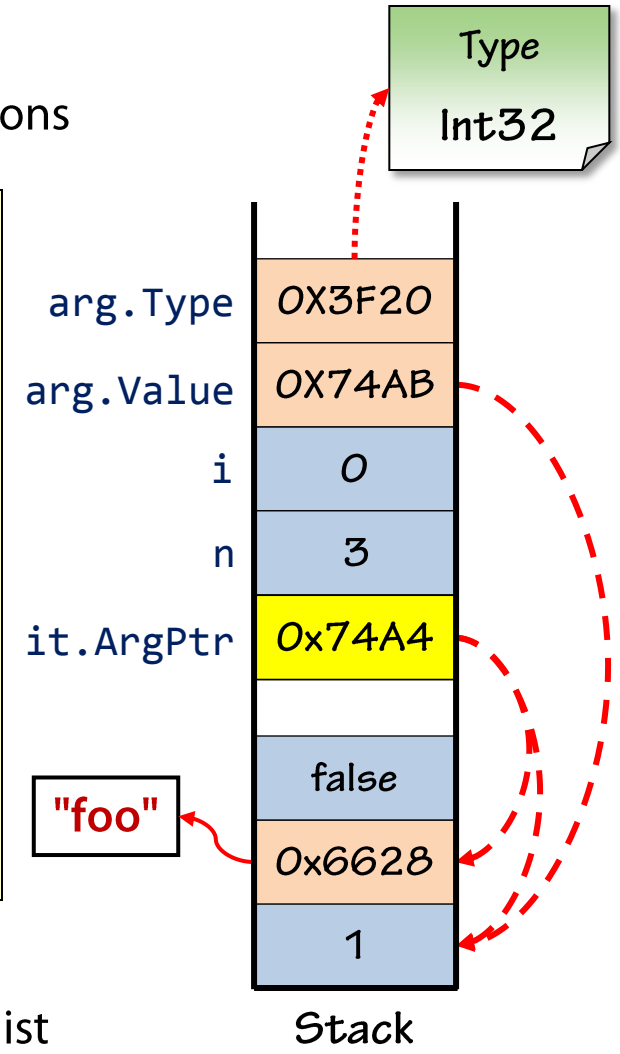
These C# keywords are not supported

__arglist

- **Supporting variable length argument lists**
  - Similar to params arrays, but without heap allocations

```
void Foo(__arglist) {
  var it = new ArgIterator(__arglist);
  var n = it.GetRemainingCount();
  for (var i = 0; i < n; i++) {
    TypedReference arg = it.GetNextArg();
    …
  }
}

void Bar() {
  Foo(__arglist(1, "foo", false));
}
```



- **Used for interop with Managed C++**
  - E.g. Console.WriteLine has an overload with __arglist

Summary

- **Static typing runs deeps in the CLR**
  - Memory and type safety guarantees
  - Types known for each location on
    - Heap
    - Stack
- **Typed references**
  - “Pointers bundled with types”
  - Used in a few places in the BCL
  - Not officially supported
- **Variable length argument lists**
  - Interop with C++