



COMP110: Principles of Computing
8: Data Structures I



Administration

Administration

- Research Journal Peer Review is this Thursday

Administration

- ▶ Research Journal Peer Review is this Thursday
- ▶ Final deadline is soon after

Administration

- ▶ Research Journal Peer Review is this Thursday
- ▶ Final deadline is soon after
- ▶ Worksheet 6 will be released this Friday evening

Administration

- ▶ Research Journal Peer Review is this Thursday
- ▶ Final deadline is soon after
- ▶ Worksheet 6 will be released this Friday evening
- ▶ Worksheet review session this week is a catch-up session

Arrays and lists



Memory allocation — recap

Memory allocation — recap

- ▶ Memory is allocated in **blocks**

Memory allocation — recap

- ▶ Memory is allocated in **blocks**
- ▶ The program specifies the size, in bytes, of the block it wants

Memory allocation — recap

- ▶ Memory is allocated in **blocks**
- ▶ The program specifies the size, in bytes, of the block it wants
- ▶ The OS allocates a **contiguous** block of that size

Memory allocation — recap

- ▶ Memory is allocated in **blocks**
- ▶ The program specifies the size, in bytes, of the block it wants
- ▶ The OS allocates a **contiguous** block of that size
- ▶ The program owns that block until it frees it

Memory allocation — recap

- ▶ Memory is allocated in **blocks**
- ▶ The program specifies the size, in bytes, of the block it wants
- ▶ The OS allocates a **contiguous** block of that size
- ▶ The program owns that block until it frees it
- ▶ Blocks can be allocated and deallocated at will, but can **never grow or shrink**

Collection types

Collection types

- ▶ Memory management is hard and programmers are lazy

Collection types

- ▶ Memory management is hard and programmers are lazy
- ▶ Collections are an **abstraction**

Collection types

- ▶ Memory management is hard and programmers are lazy
- ▶ Collections are an **abstraction**
 - ▶ Hide the details of memory allocation, and allow the programmer to write simpler code

Collection types

- ▶ Memory management is hard and programmers are lazy
- ▶ Collections are an **abstraction**
 - ▶ Hide the details of memory allocation, and allow the programmer to write simpler code
- ▶ Collections are an **encapsulation**

Collection types

- ▶ Memory management is hard and programmers are lazy
- ▶ Collections are an **abstraction**
 - ▶ Hide the details of memory allocation, and allow the programmer to write simpler code
- ▶ Collections are an **encapsulation**
 - ▶ Bundle together the data's representation in memory along with the algorithms for accessing it

Arrays

Arrays

- ▶ An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other

Arrays

- ▶ An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- ▶ Each array element has an **index**, starting from zero

Arrays

- ▶ An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- ▶ Each array element has an **index**, starting from zero
- ▶ Given the address of the 0th element, it is easy to find the i th element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

Arrays

- ▶ An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- ▶ Each array element has an **index**, starting from zero
- ▶ Given the address of the 0th element, it is easy to find the i th element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

- ▶ E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$

Arrays

- ▶ An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- ▶ Each array element has an **index**, starting from zero
- ▶ Given the address of the 0th element, it is easy to find the i th element:

$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

- ▶ E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$
- ▶ Accessing an array element is **constant time** $O(1)$

Lists

Lists

- ▶ An array is a block of memory, so its size is **fixed** once created

Lists

- ▶ An array is a block of memory, so its size is **fixed** once created
- ▶ A **list** is a variable size array

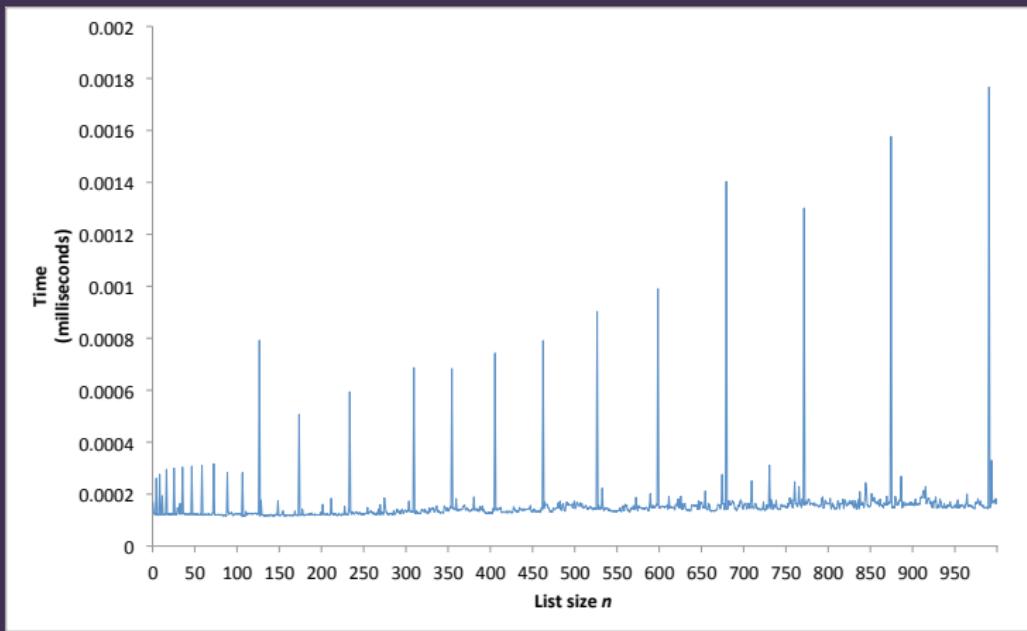
Lists

- ▶ An array is a block of memory, so its size is **fixed** once created
- ▶ A **list** is a variable size array
- ▶ When the list needs to change size, it **creates** a new array, **copies** the contents of the old array, and **deletes** the old array

Arrays and lists in C#

```
int[] myArray = new int[10];  
  
int[] myOtherArray = new int[] { 2, 3, 5, 7, 11 };  
  
List<int> myList = new List<int>();  
  
List<int> myOtherList = new List<int> { 2, 3, 5, 8, 13 };
```

Time taken to append an element to a list of size n



Operations on lists

Operations on lists

- ▶ **Appending** to a list is **amortised constant time**

Operations on lists

- ▶ **Appending** to a list is **amortised constant time**
 - ▶ Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size

Operations on lists

- ▶ **Appending** to a list is **amortised constant time**
 - ▶ Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- ▶ **Inserting** anywhere other than the end is **linear time**

Operations on lists

- ▶ **Appending** to a list is **amortised constant time**
 - ▶ Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- ▶ **Inserting** anywhere other than the end is **linear time**
 - ▶ Can't just insert new bytes into a memory block — need to move all subsequent list elements to make room

Operations on lists

- ▶ **Appending** to a list is **amortised constant time**
 - ▶ Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- ▶ **Inserting** anywhere other than the end is **linear time**
 - ▶ Can't just insert new bytes into a memory block — need to move all subsequent list elements to make room
- ▶ Similarly, **deleting** anything other than the last element is **linear time**

Stacks and queues



Stacks and queues

Stacks and queues



- ▶ A **stack** is a **last-in first-out (LIFO)** data structure

Stacks and queues



- ▶ A **stack** is a **last-in first-out (LIFO)** data structure
- ▶ Items can be **pushed** to the **top** of the stack

Stacks and queues



- ▶ A **stack** is a **last-in first-out (LIFO)** data structure
- ▶ Items can be **pushed** to the **top** of the stack
- ▶ Items can be **popped** from the **top** of the stack

Stacks and queues



- ▶ A **stack** is a **last-in first-out (LIFO)** data structure
- ▶ Items can be **pushed** to the **top** of the stack
- ▶ Items can be **popped** from the **top** of the stack
- ▶ A **queue** is a **first-in first-out (FIFO)** data structure



Stacks and queues



- ▶ A **stack** is a **last-in first-out (LIFO)** data structure
- ▶ Items can be **pushed** to the **top** of the stack
- ▶ Items can be **popped** from the **top** of the stack

- ▶ A **queue** is a **first-in first-out (FIFO)** data structure
- ▶ Items can be **enqueued** to the **back** of the queue

Stacks and queues



- ▶ A **stack** is a **last-in first-out (LIFO)** data structure
- ▶ Items can be **pushed** to the **top** of the stack
- ▶ Items can be **popped** from the **top** of the stack

- ▶ A **queue** is a **first-in first-out (FIFO)** data structure
- ▶ Items can be **enqueued** to the **back** of the queue
- ▶ Items can be **dequeued** from the **front** of the queue

Implementing stacks

Implementing stacks

- ▶ Stacks can be implemented efficiently as lists

Implementing stacks

- ▶ Stacks can be implemented efficiently as lists
- ▶ Top of stack = end of list

Implementing stacks

- ▶ Stacks can be implemented efficiently as lists
- ▶ Top of stack = end of list
- ▶ To push an element, use `Add` — $O(1)$ complexity

Implementing stacks

- ▶ Stacks can be implemented efficiently as lists
- ▶ Top of stack = end of list
- ▶ To push an element, use `Add` — $O(1)$ complexity
- ▶ To pop an element we can do something like this:

```
x = myStack[myStack.Count - 1];  
myStack.RemoveAt(myStack.Count - 1);
```

Implementing stacks

- ▶ Stacks can be implemented efficiently as lists
- ▶ Top of stack = end of list
- ▶ To push an element, use `Add` — $O(1)$ complexity
- ▶ To pop an element we can do something like this:

```
x = myStack[myStack.Count - 1];  
myStack.RemoveAt(myStack.Count - 1);
```

- ▶ This is also $O(1)$

Implementing queues

Implementing queues

- ▶ Queues can be implemented as lists, but not efficiently

Implementing queues

- ▶ Queues can be implemented as lists, but not efficiently
- ▶ End of list = back of queue

Implementing queues

- ▶ Queues can be implemented as lists, but not efficiently
- ▶ End of list = back of queue
- ▶ Enqueue using `Add` — $O(1)$ complexity

Implementing queues

- ▶ Queues can be implemented as lists, but not efficiently
- ▶ End of list = back of queue
- ▶ Enqueue using `Add` — $O(1)$ complexity
- ▶ Dequeue by retrieving and removing from beginning of list:

```
x = myQueue[0];  
myQueue.RemoveAt(0);
```

Implementing queues

- ▶ Queues can be implemented as lists, but not efficiently
- ▶ End of list = back of queue
- ▶ Enqueue using `Add` — $O(1)$ complexity
- ▶ Dequeue by retrieving and removing from beginning of list:

```
x = myQueue[0];  
myQueue.RemoveAt(0);
```

- ▶ This is $O(n)$

Implementing queues

Implementing queues

- ▶ End of list = front of queue

Implementing queues

- ▶ End of list = front of queue
- ▶ Dequeue is like popping from end of list — $O(1)$ complexity

Implementing queues

- ▶ End of list = front of queue
- ▶ Dequeue is like popping from end of list — $O(1)$ complexity
- ▶ Enqueue using `Insert(0, x)` — $O(n)$ complexity

Using stacks and queues

Using stacks and queues

- ▶ C# has `Stack` and `Queue` classes which you should use instead of trying to use a list

Using stacks and queues

- ▶ C# has `Stack` and `Queue` classes which you should use instead of trying to use a list
- ▶ Python has `deque` (double-ended queue) which can work as either a stack or a list

Stacks and function calls

Stacks and function calls

- ▶ Stacks are used to implement **nested function calls**

Stacks and function calls

- ▶ Stacks are used to implement **nested function calls**
- ▶ Each invocation of a function has a **stack frame**

Stacks and function calls

- ▶ Stacks are used to implement **nested function calls**
- ▶ Each invocation of a function has a **stack frame**
- ▶ This specifies information like **local variable values** and **return address**

Stacks and function calls

- ▶ Stacks are used to implement **nested function calls**
- ▶ Each invocation of a function has a **stack frame**
- ▶ This specifies information like **local variable values** and **return address**
- ▶ Calling a function **pushes** a new frame onto the stack

Stacks and function calls

- ▶ Stacks are used to implement **nested function calls**
- ▶ Each invocation of a function has a **stack frame**
- ▶ This specifies information like **local variable values** and **return address**
- ▶ Calling a function **pushes** a new frame onto the stack
- ▶ Returning from a function **pops** the top frame off the stack

Stacks and function calls

- ▶ Stacks are used to implement **nested function calls**
- ▶ Each invocation of a function has a **stack frame**
- ▶ This specifies information like **local variable values** and **return address**
- ▶ Calling a function **pushes** a new frame onto the stack
- ▶ Returning from a function **pops** the top frame off the stack
- ▶ Hence the term **stack trace** when using the debugger or looking at error logs

Pass by reference



References

References

- ▶ Our picture of a variable: a labelled box containing a value

References

- ▶ Our picture of a variable: a labelled box containing a value
- ▶ For “plain old data” (e.g. numbers), this is accurate

References

- ▶ Our picture of a variable: a labelled box containing a value
- ▶ For “plain old data” (e.g. numbers), this is accurate
- ▶ For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)

References

- ▶ Our picture of a variable: a labelled box containing a value
- ▶ For “plain old data” (e.g. numbers), this is accurate
- ▶ For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)
- ▶ It is possible (indeed common) to have **multiple references** to the same underlying object

The wrong picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

The wrong picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value
x	
y	
z	

The wrong picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value	
x	a	30
	b	40
y		
z		

The wrong picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z		

The wrong picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }

    Thing x = new Thing(30, 40);
    Thing y = new Thing(50, 60);
    Thing z = y;
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z	a	50
	b	60

The right picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

The right picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value
x	
y	
z	

The right picture

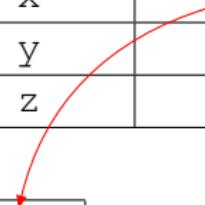
```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value
x	
y	
z	

a	30
b	40



The right picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value
x	
y	
z	

a	30
b	40

a	50
b	60

The right picture

```
class Thing
{
    public int a, b;

    public Thing(int a_, int b_)
    {
        a = a_; b = b_;
    }
}

Thing x = new Thing(30, 40);
Thing y = new Thing(50, 60);
Thing z = y;
```

Variable	Value
x	
y	
z	

a	30
b	40

a	50
b	60

Values and references

Socrative room code: FALCOMPED

```
int a = 10;
int b = a;
a = 20;
Console.WriteLine($"a: {a}");
Console.WriteLine($"b: {b}");
```

Values and references

Socrative room code: FALCOMPED

```
class Foo
{
    public int value;

    public Foo(int v)
    {
        value = v;
    }
}

Foo a = new Foo(10);
Foo b = a
a.value = 20
Console.WriteLine($"a: {a.value}");
Console.WriteLine($"b: {b.value}");
```

Values and references

Socrative room code: FALCOMPED

```
class Foo
{
    public int value;

    public Foo(int v)
    {
        value = v;
    }
}

Foo a = new Foo(10);
Foo b = new Foo(10);
a.value = 20
Console.WriteLine($"a: {a.value}");
Console.WriteLine($"b: {b.value}");
```

Pass by value

Socrative room code: FALCOMPED

In **function parameters**, “plain old data” is passed by **value**

Pass by value

Socrative room code: FALCOMPED

In **function parameters**, “plain old data” is passed by **value**

```
void doubleIt(int x)
{
    x = x * 2;
}

int a = 7;
doubleIt(a);
Console.WriteLine(a);
```

Pass by value

Socrative room code: FALCOMPED

In **function parameters**, “plain old data” is passed by **value**

```
void doubleIt(int x)
{
    x = x * 2;
}

int a = 7;
doubleIt(a);
Console.WriteLine(a);
```

What does it print?

Pass by reference

Socrative room code: FALCOMPED

However, objects (class instances) are passed by
reference

Pass by reference

Socrative room code: FALCOMPED

However, objects (class instances) are passed by **reference**

```
class Foo
{
    public int value;
    public Foo(int v) { value = v; }
}

void doubleIt(Foo x)
{
    x.value = x.value * 2;
}

Foo a = new Foo(7);
doubleIt(a);
Console.WriteLine(a.value);
```

Pass by reference

Socrative room code: FALCOMPED

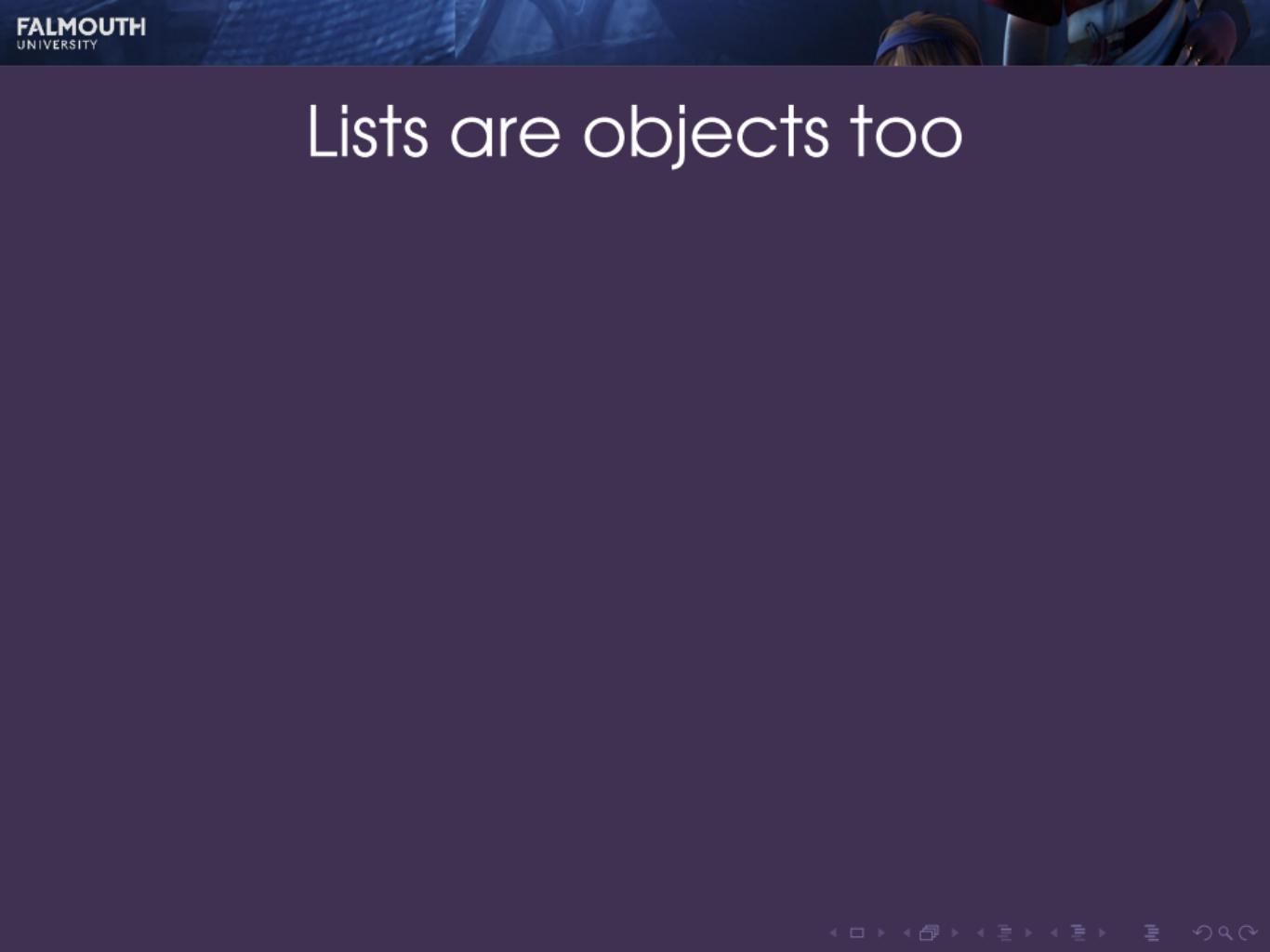
However, objects (class instances) are passed by **reference**

```
class Foo
{
    public int value;
    public Foo(int v) { value = v; }
}

void doubleIt(Foo x)
{
    x.value = x.value * 2;
}

Foo a = new Foo(7);
doubleIt(a);
Console.WriteLine(a.value);
```

What does it print?



Lists are objects too

Lists are objects too

```
List<string> a = new List<string>{ "Hello" };
List<string> b = a;
b.Add("world");
foreach (string word in a)
{
    Console.WriteLine(word);
}
// Output:
//   Hello
//   world
```

Lists are objects too

```
List<string> a = new List<string>{ "Hello" };
List<string> b = a;
b.Add("world");
foreach (string word in a)
{
    Console.WriteLine(word);
}
// Output:
//   Hello
//   world
```

... which means you should be careful when passing lists into functions, because the function might actually change the list!

Pass by value again

In C#, struct instances are passed by **value**

Pass by value again

In C#, struct instances are passed by **value**

```
struct Foo
{
    public int value;
    public Foo(int v) { value = v; }
}

void doubleIt(Foo x)
{
    x.value = x.value * 2;
}

Foo a = new Foo(7);
doubleIt(a);
Console.WriteLine(a.value);
```

Pass by value again

In C#, struct instances are passed by **value**

```
struct Foo
{
    public int value;
    public Foo(int v) { value = v; }
}

void doubleIt(Foo x)
{
    x.value = x.value * 2;
}

Foo a = new Foo(7);
doubleIt(a);
Console.WriteLine(a.value);
```

This prints 7

By reference or value?

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:
 - ▶ Basic data types (`int`, `bool`, `float` etc)

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:
 - ▶ Basic data types (`int`, `bool`, `float` etc)
 - ▶ Instances of `structs`

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:
 - ▶ Basic data types (`int`, `bool`, `float` etc)
 - ▶ Instances of `structs`
- ▶ These function arguments are passed **by reference**:

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:
 - ▶ Basic data types (`int`, `bool`, `float` etc)
 - ▶ Instances of `structs`
- ▶ These function arguments are passed **by reference**:
 - ▶ Instances of `classes` — this includes classes built into .NET or Unity etc

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:
 - ▶ Basic data types (`int`, `bool`, `float` etc)
 - ▶ Instances of `structs`
- ▶ These function arguments are passed **by reference**:
 - ▶ Instances of `classes` — this includes classes built into .NET or Unity etc
 - ▶ Arguments with the `ref` keyword attached

By reference or value?

- ▶ In C#, these function arguments are passed **by value**:
 - ▶ Basic data types (`int`, `bool`, `float` etc)
 - ▶ Instances of `structs`
- ▶ These function arguments are passed **by reference**:
 - ▶ Instances of `classes` — this includes classes built into .NET or Unity etc
 - ▶ Arguments with the `ref` keyword attached
- ▶ Passing by value implies copying — not a problem for small data values but beware of passing large structs around

References and pointers

References and pointers

- ▶ Some languages (e.g. C, C++) use **pointers**

References and pointers

- ▶ Some languages (e.g. C, C++) use **pointers**
- ▶ Pointers are a type of reference, and have the same semantics

References and pointers

- ▶ Some languages (e.g. C, C++) use **pointers**
- ▶ Pointers are a type of reference, and have the same semantics
- ▶ References in other languages (e.g. C#, Python) are implemented using pointers

References and pointers

- ▶ Some languages (e.g. C, C++) use **pointers**
- ▶ Pointers are a type of reference, and have the same semantics
- ▶ References in other languages (e.g. C#, Python) are implemented using pointers
- ▶ C++ also has something called references, which are similar but different (pointers can be **retargeted** whilst references cannot)

Pointers

Pointers

- ▶ Recall that memory is a series of 1-byte locations, each with a numeric **address**

Pointers

- ▶ Recall that memory is a series of 1-byte locations, each with a numeric **address**
- ▶ A **pointer** to something is simply the **address** at which it starts

Pointers

- ▶ Recall that memory is a series of 1-byte locations, each with a numeric **address**
- ▶ A **pointer** to something is simply the **address** at which it starts
- ▶ When allocating a block of memory, the OS returns a pointer to the start of the block

Pointers

- ▶ Recall that memory is a series of 1-byte locations, each with a numeric **address**
- ▶ A **pointer** to something is simply the **address** at which it starts
- ▶ When allocating a block of memory, the OS returns a pointer to the start of the block
- ▶ When the memory is freed, any pointers into it are said to be **dangling**

Pointers

- ▶ Recall that memory is a series of 1-byte locations, each with a numeric **address**
- ▶ A **pointer** to something is simply the **address** at which it starts
- ▶ When allocating a block of memory, the OS returns a pointer to the start of the block
- ▶ When the memory is freed, any pointers into it are said to be **dangling**
- ▶ If the memory is subsequently reused for something else, those pointers could end up pointing to random data

Pointers

- ▶ Recall that memory is a series of 1-byte locations, each with a numeric **address**
- ▶ A **pointer** to something is simply the **address** at which it starts
- ▶ When allocating a block of memory, the OS returns a pointer to the start of the block
- ▶ When the memory is freed, any pointers into it are said to be **dangling**
- ▶ If the memory is subsequently reused for something else, those pointers could end up pointing to random data
- ▶ Again this is not really possible in Python/C#, but a common source of bugs in C/C++

Workshop time



Workshop

- ▶ Continue working on your **research journal** to prepare for the peer review on Thursday
- ▶ Today, pay particular attention to your **bibliography**
- ▶ Common pitfalls to watch out for:
 - ▶ Do entries have all required fields?
 - ▶ Are proper nouns etc in titles correctly capitalised?
 - ▶ Are names with accented characters properly formatted?
 - ▶ Are there duplicate entries?
- ▶ Feel free to use your breakout groups (from previous weeks) to check each other's bibliographies over and help each other fix any issues
- ▶ Post in chat here if you have questions or problems!