



COMP110: Principles of Computing
9: Data Structures II

Worksheets Resume

Worksheet 6 is out now!
Formative deadline: next Friday

Basic data structures in C# (continued from last time)



Strings

```
string myString = "Hello, world!";
```

Strings

```
string myString = "Hello, world!";
```

- ▶ **string** can be thought of as a collection

Strings

```
string myString = "Hello, world!";
```

- ▶ `string` can be thought of as a collection
- ▶ In particular, it implements `IEnumerable<char>`

Strings

```
string myString = "Hello, world!";
```

- ▶ `string` can be thought of as a collection
- ▶ In particular, it implements `IEnumerable<char>`
- ▶ So for example we can iterate over the characters in a string:

```
foreach (char c in myString)
{
    Console.WriteLine(c);
}
```

Strings are immutable

Strings are immutable

- ▶ Strings are **immutable** in C#

Strings are immutable

- ▶ Strings are **immutable** in C#
- ▶ This means that the contents of a string **cannot be changed** once it is created

Strings are immutable

- ▶ Strings are **immutable** in C#
- ▶ This means that the contents of a string **cannot be changed** once it is created
- ▶ But wait... we change strings all the time, don't we?

```
string myString = "Hello ";
myString += "world";
```

Strings are immutable

- ▶ Strings are **immutable** in C#
- ▶ This means that the contents of a string **cannot be changed** once it is created
- ▶ But wait... we change strings all the time, don't we?

```
string myString = "Hello ";
myString += "world";
```

- ▶ This isn't changing the string, it's creating a new one and throwing the old one away!

Strings are immutable

- ▶ Strings are **immutable** in C#
- ▶ This means that the contents of a string **cannot be changed** once it is created
- ▶ But wait... we change strings all the time, don't we?

```
string myString = "Hello ";
myString += "world";
```

- ▶ This isn't changing the string, it's creating a new one and throwing the old one away!
- ▶ Hence building a long string by appending can be slow (appending strings is $O(n)$)

Strings are immutable

- ▶ Strings are **immutable** in C#
- ▶ This means that the contents of a string **cannot be changed** once it is created
- ▶ But wait... we change strings all the time, don't we?

```
string myString = "Hello ";
myString += "world";
```

- ▶ This isn't changing the string, it's creating a new one and throwing the old one away!
- ▶ Hence building a long string by appending can be slow (appending strings is $O(n)$)
- ▶ C# has a **mutable** string type: `StringBuilder`

Dictionaries

Dictionaries

- Dictionaries are **associative maps**

Dictionaries

- ▶ Dictionaries are **associative maps**
- ▶ A dictionary maps **keys** to **values**

Dictionaries

- ▶ Dictionaries are **associative maps**
- ▶ A dictionary maps **keys** to **values**
- ▶ Takes two generic parameters: the **key type** and the **value type**

Dictionaries

- ▶ Dictionaries are **associative maps**
- ▶ A dictionary maps **keys** to **values**
- ▶ Takes two generic parameters: the **key type** and the **value type**
- ▶ A dictionary is implemented as a **hash table**

Using dictionaries

```
var age = new Dictionary<string, int> {
    ["Alice"] = 23,
    ["Bob"] = 36,
    ["Charlie"] = 27
};
```

Using dictionaries

```
var age = new Dictionary<string, int> {  
    ["Alice"] = 23,  
    ["Bob"] = 36,  
    ["Charlie"] = 27  
};
```

Access values using []:

```
Console.WriteLine(age["Alice"]); // prints 23  
age["Bob"] = 40; // overwriting an existing item  
age["Denise"] = 21; // adding a new item  
age.Add("Emily", 29); // adding a new item -- will raise  
// an error if already present
```

Iterating over dictionaries

Iterating over dictionaries

- Dictionary<Key, Value> implements
IEnumerable<KeyValuePair<Key, Value>>

Iterating over dictionaries

- ▶ Dictionary<Key, Value> implements
IEnumerable<KeyValuePair<Key, Value>>
- ▶ KeyValuePair<Key, Value> stores Key and Value

Iterating over dictionaries

- ▶ Dictionary<Key, Value> implements
IEnumerable<KeyValuePair<Key, Value>>
- ▶ KeyValuePair<Key, Value> stores Key and Value

```
foreach (var keyValue in age)
{
    Console.WriteLine("{0} is {1} years old",
                      kv.Key, kv.Value);
}
```

Iterating over dictionaries

- ▶ Dictionary<Key, Value> implements
IEnumerable<KeyValuePair<Key, Value>>
- ▶ KeyValuePair<Key, Value> stores Key and Value

```
foreach (var keyValue in age)
{
    Console.WriteLine("{0} is {1} years old",
                      kv.Key, kv.Value);
}
```

- ▶ (C# tip: the var keyword lets the compiler automatically determine the appropriate type to use for a variable)

Iterating over dictionaries

- ▶ Dictionary<Key, Value> implements
IEnumerable<KeyValuePair<Key, Value>>
- ▶ KeyValuePair<Key, Value> stores Key and Value

```
foreach (var keyValue in age)
{
    Console.WriteLine("{0} is {1} years old",
                      kv.Key, kv.Value);
}
```

- ▶ (C# tip: the var keyword lets the compiler automatically determine the appropriate type to use for a variable)
- ▶ Dictionaries are **unordered** — avoid assuming that **foreach** will see the elements in any particular order!

Hash sets

Hash sets

- ▶ Sets are **unordered** collections of **unique** elements

Hash sets

- ▶ Sets are **unordered** collections of **unique** elements
 - ▶ Sets **cannot** contain **duplicate** elements

Hash sets

- ▶ Sets are **unordered** collections of **unique** elements
 - ▶ Sets **cannot** contain **duplicate** elements
 - ▶ Attempting to `Add` an element already present in the set does nothing

Hash sets

- ▶ Sets are **unordered** collections of **unique** elements
 - ▶ Sets **cannot** contain **duplicate** elements
 - ▶ Attempting to `Add` an element already present in the set does nothing
- ▶ HashSets are like `Dictionary`s without the values, just the keys

Hash sets

- ▶ Sets are **unordered** collections of **unique** elements
 - ▶ Sets **cannot** contain **duplicate** elements
 - ▶ Attempting to `Add` an element already present in the set does nothing
- ▶ HashSets are like Dictionaries without the values, just the keys
- ▶ As discussed in Week 5, certain operations are much more efficient (constant time) on hash sets than on lists

Using sets

```
var numbers = new HashSet<int>{1, 4, 9, 16, 25};
```

Using sets

```
var numbers = new HashSet<int>{1, 4, 9, 16, 25};
```

Add and remove members with `Add` and `Remove` methods

```
numbers.Add(36);  
numbers.Remove(4);
```

Using sets

```
var numbers = new HashSet<int>{1, 4, 9, 16, 25};
```

Add and remove members with `Add` and `Remove` methods

```
numbers.Add(36);  
numbers.Remove(4);
```

Test membership with `Contains`

```
if (numbers.Contains(9))  
    Console.WriteLine("Set contains 9");
```

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

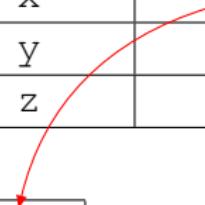
```
class Thing
{
    public int a, b;

    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}");
```

References and values — an analogy

References and values — an analogy

- ▶ Suppose you are sending someone a document

References and values — an analogy

- ▶ Suppose you are sending someone a document
- ▶ Pass **by value** is like sending a .docx file — the recipient gets their own copy

References and values — an analogy

- ▶ Suppose you are sending someone a document
- ▶ Pass **by value** is like sending a .docx file — the recipient gets their own copy
- ▶ Pass **by reference** is like sending a Google Docs link — there is one copy, which the recipient can potentially edit

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)

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)
- ▶ Implementation-wise, a pointer is literally just a memory address (i.e. a number)

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

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
- ▶ More on this next week when we look at **recursion**

Workshop



Workshop

Begin working on Worksheet 6
I am available for the rest of the session if you run into any
problems!
(Or if you have any general questions etc.)