Structs

- sometimes it is desirable to create your own data types which contain more than one other type
- for example, suppose we want to create a `coord` data type
- we would expect it to contain an `x` and `y` component
- in C# we can do this using a `struct`

struct coord

- once the `coord` struct is defined we can use it
  - it should make our programs easier to maintain and clearer

struct coord

-

```
using System;

public struct coord
{
  public int x;
  public int y;
}
```

struct coord

```
public class test
{
  public static void Main ()
  {
    coord monster;

    monster.x = 50;
    monster.y = 10;

    Console.WriteLine ("monster x pos = {0}, y pos = {1}"
                       monster.x, monster.y);
  }
}
```


Improving coord

- C# allows us to employ operator overloading and constructors with `structs`

Improving coord

```
using System;

public struct coord
{
  public int x;
  public int y;

  public override string ToString ()
  {
    return String.Format ("{0}, {1}", x, y);
  }
}
```


Improving coord

```
public class test
{
  public static void Main ()
  {
    coord monster;

    monster.x = 50;
    monster.y = 10;

    Console.WriteLine ("monster at {0}", monster);
  }
}
```

Output of the improved program

-
```
monster at 50, 10
```
- the function/method `ToString` will be called by the `Console.WriteLine` to resolve how a data type is converted into a string
- technique referred to as polymorphism
  - we associate a method with its data type
  - if nothing else this mechanism provides a way of debugging these data types
- in subsequent years you will meet this issue again

Pedalogical example: fract

- let us create a fraction data type `fract`
- `fract` will have a whole number, numerator and denominator
- $w + \frac{n}{d}$

Pedalogical example: fract

-
```
using System;

public struct fract
{
  public int w;  // the whole component
  public int n;  // the numerator
  public int d;  // the denominator
  private bool positive;  // is this a positive number?
  private bool is_simplified;  // have we simplified this
```


Pedalogical example: fract

- the constructor method
-
```
  public fract (int whole, int num, int dem)
  {
    w = whole;
    n = num;
    d = dem;
    positive = true;
    is_simplified = false;
  }
```

Pedalogical example: fract

```
  public override string ToString ()
  {
    if (w == 0 && n == 0)
      return "zero";

    if (positive)
      {
        if (w == 0)
          return String.Format ("{0}/{1}", n, d);
        else {
          if (n == 0)
            return String.Format ("{0}", w);
          return String.Format ("{0} and {1}/{2}", w, n,
        }
      }
```


Pedalogical example: fract

```
      else
        {
          if (w == 0)
            return String.Format ("-{0}/{1}", n, d);
          else {
            if (n == 0)
              return String.Format ("-{0}", w);
            return String.Format ("-({0} and {1}/{2})", w,
          }
        }
    }
}
```


Pedalogical example: fract

```
public class test
{
  public static void Main ()
  {
    fract a = new fract (1, 1, 2);

    Console.WriteLine ("a = {0}", a);
  }
}
```

- output when the program is run

```
a = 1 and 1/2
```


Pedalogical example: fract

- notice that we needed to use the keyword `new`

Improved struct fract to include simplification

- let us revisit the code and introduce the ability for the output routine to simplify the fraction if necessary

Improved struct fract to include simplification

-
```
using System;

public struct fract
{
  public int w;
  public int n;
  public int d;
  private bool positive;
  private bool is_simplified;

  public fract (int whole, int num, int dem)
  {
    w = whole;
    n = num;
    d = dem;
    positive = true;
    is_simplified = false;
  }
```


Improved struct fract to include simplification

-
```
/*
 *  gcd - Euclid's Greatest Common Denominator algorithm
 *        pre-condition :  x and y are both >0
 *        post-condition:  return the greatest denominator of
 *                         x, and, y.
 */

public int gcd (int x, int y)
{
  while (x != y)
    {
      if (x > y)
        x -= y;
      else
        y -= x;
    }
  return x;
}
```


Dryrun of Euclid's algorithm

- the result of this algorithm is to return the largest whole number which can be divided into both, x, and, y

-

Iteration number	x	y
0	20	15
1	5	15
2	5	10
3	5	5

Dryrun of Euclid's algorithm

Iteration number	x	y
0	9	12
1	9	3
2	6	3
3	3	3

Dryrun of Euclid's algorithm

Iteration number	x	y
0	72	48
1	24	48
2	24	24

Completing the code associated with the fract struct

```
/*
 *  simplify - pre-condition :  an initialised fract
 *             post-condition:  whole, num, denom are
 *                              converted into their
 *                              simplist form.
 */

public void simplify ()
{
  int t;

  if (is_simplified)   // if it has been simplified bef
    return;
  if (n == 0)   // if the numerator is 0 set the demoni
    d = 0;
```


Completing the code associated with the fract struct

```
  if ((n != 0) && (d != 0))  // if we have a fraction c
    {
      if (n > d)  // if the fraction is vulgar
      {
        t = n / d;  // t is the whole number in the vul
        w += t;  // add it to the whole number
        n = n % d;  // find remainder which becomes new
      }
```

Completing the code associated with the fract struct

```
      if ((n != 0) && (d != 0))
        {
          t = gcd (n, d);  // t is the largest number whi
          if (t > 1)  //  into n, d
            {
              n = n / t;  // reduce both numerator and de
              d = d / t;  // by t
            }
        }
      if (n == d)  // is the fraction a whole number?
        {
          n = 0;
          d = 0;
          w++;
        }
      }
    is_simplified = true;  // set flag saying it is simpl
  }
```


ToString method

```
  public override string ToString ()
  {
    if (w == 0 && n == 0)
      return "zero";

    simplify ();

    /* as before.  */
```


Test code

```
public class test
{
  public static void Main ()
  {
    fract a = new fract (0, 3, 2);

    Console.WriteLine ("a = {0}", a);
  }
}
```

- output when running code:

```
a = 1 and 1/2
```


Tutorial

- download the ⟨`http://floppsie.comp.glam.ac.uk/download/csharp/pacman.cs`⟩ code and ⟨`http://floppsie.comp.glam.ac.uk/download/csharp/level1.txt`⟩ file
  - `level1.txt` will need to be placed into your *projectname*`/bin/debug` directory
- build it and see the game start
- copy `level1.txt` to `level2.txt`
  - now edit `level1.txt` and make some changes

Tutorial

- notice that the move_ghost is a function which calls four other functions depending upon the name of the ghost
  - currently only `move_ghost_1` is implemented
  - see if you can create `move_ghost_2`, `move_ghost_3` and `move_ghost_4` and call them appropriately inside `move_ghost`
- `move_ghost_2` should prioritise y movement
- `move_ghost_3` should only change direction if necessary
- `move_ghost_4` should change direction if at all possible
  - hint code 1 ghost at a time