User-defined types

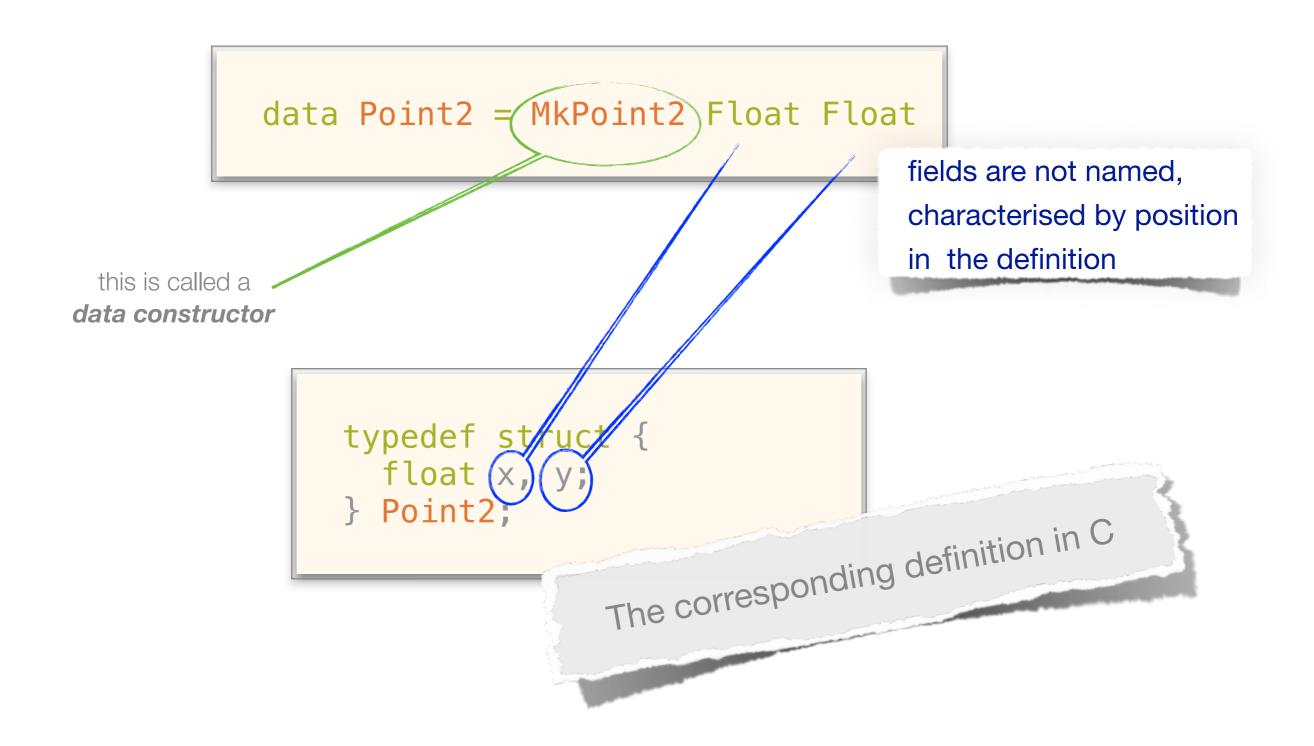
Type synonyms (typedefs in C)

```
type Point = (Float, Float)
type Path = [Point]
```

- Algebraic data types
 - Combination of structs and unions
 - together with pointers in C



Data types can be like structs in C (we call those data types product types)





Data Constructors

Data constructors are a (special kind of) functions:

```
data Point2 = MkPoint2 Float Float
MkPoint2 :: Float -> Float -> Point2
```

 Arguments to data constructors can always be recovered using pattern matching:



Data Constructors

We already know some other data constructors:

```
(,) :: a -> b -> (a,b)
fst (x, _) = x
```

```
[]:: [a]
(:) :: a -> [a] -> [a]
length [] = 0
length (_ : xs) = 1 + length xs
```



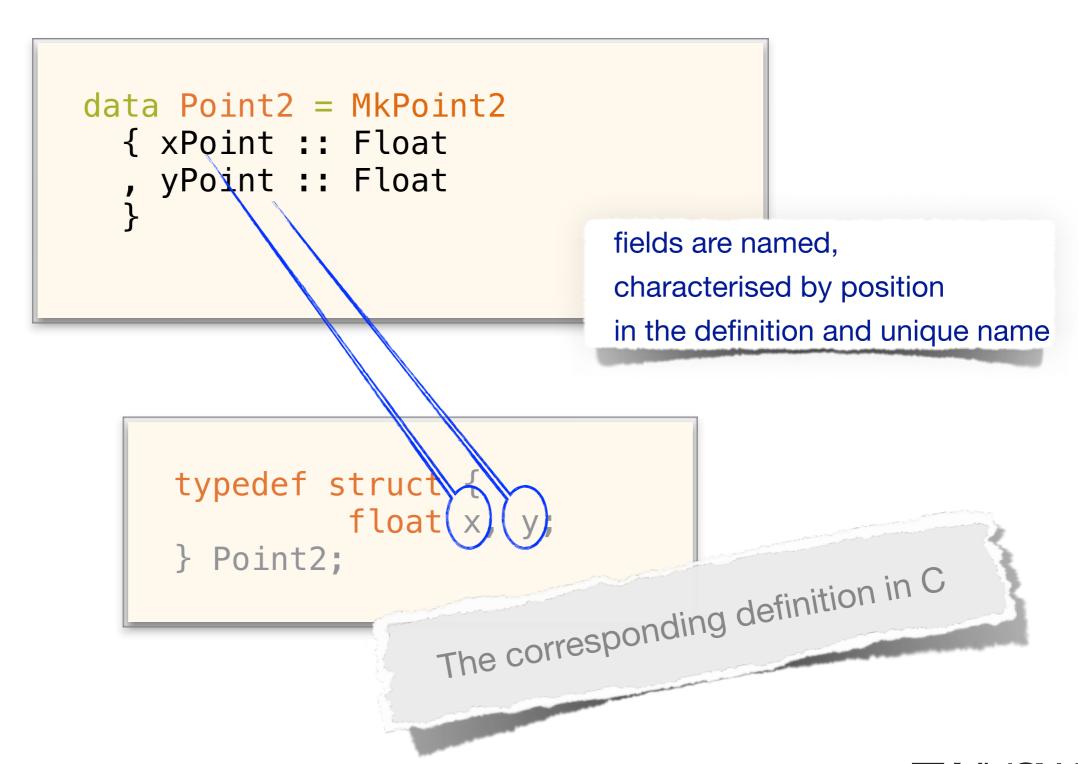
```
data Point2 = MkPoint2 Float Float
point :: Point2
point = MkPoint2 1.3 2.45
```

```
typedef struct {
   float x, y;
} Point2;

Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
```



• Data types can be like structs in C (we call those data types product types)





```
data Point2 = MkPoint2
    { xPoint :: Float
    , yPoint :: Float
    }

point :: Point2
point = MkPoint2 1.3 2.45
    - or
point = MkPoint2 {yPoint = 2.45, xPoint = 1.3}
```

```
typedef struct {
   unsigned int x, y;
} Point2;

Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
```



```
data Point2 = MkPoint2
   { xPoint :: Float
   , yPoint :: Float

    the above definition brings three functions

- into scope:
MkPoint2 :: Float -> Float -> Point2 - constructor
xPoint :: Point2 -> Float - access function for x
yPoint :: Point2 -> Float - access function for y

    using pattern matching to access components

distance :: Point2 -> Point2 -> Float
distance (MkPoint2 x1 y1) (MkPoint2 x2 y2) =
  sqrt ((x2 - x1)^2 + (y2 - y1)^2)

    using access functions

distance p1 p2 =
  sqrt ((xPoint p2 - xPoint p1)^2 +
        (yPoint p2 - yPoint p1)^2)
```



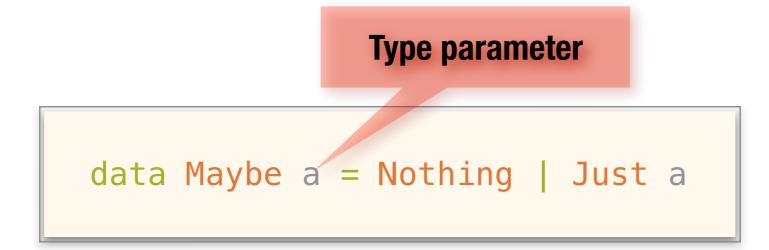
- Problem: define a type to model shapes. A shape can be a rectangle (position, width, height) or a circle (position, radius)
- Data types can be like unions in C (we call those data types sum types)

```
data Shape = Rectangle Point Float
| Circle Point Float
```

```
enum tag {RECTANGLE_SHAPE, CIRCLE_SHAPE};
struct mkRectangle {
 float height;
 float width;
                                      The definition in C
struct mkCircle {
 enum tag theTag;
 point pos;
 radius float:
typedef union {
 struct mkCircle aCircle;
 struct mkRectangle aRectangle;
} Shape;
```

Product-Sum Types

- We call Haskell's data types also product-sum types
- They can be recursive as well
- In contrast to data types in C, but much like generics in Java and C#, Haskell data types can be parameterised





Identifiers in Haskell

• Alphanumeric with underscores (_) and prime symbols (')

Case matters

Functions & variables	lower case	map, pi, (+), (++)
Data constructors	Upper case	True, Nothing, (:)
Type variables	lower case	a, b, c, eltType
Type constructors	Upper case	Int, Bool, IO



Next Thursday: guest lecture

- Patrick Flanagan (Jane St, Hongkong)
- Thu, 15 March

