Data structures

So far we have mostly used built-in types

- integers ... -1, 0, 1, ...
- floating point numbers 3.14, ...
- characters 0..127

Pretty boring

We have seen one example of something bigger

- Arrays

Programming languages allow you to define more types:

- Enumerated types
- Making your own types
- Flatly structured data types
- Dynamic data types.

Enumerated types

An enumerated type represents some collection of constants:

```
enum suit { Club, Diamond, Heart, Spade } ;
```

Defines a type **`enum suit`** with 4 constants **`Club`**, **`Diamond`**, ...
It is an integer type, and the compiler allocates values starting at 0
It is just as if you had defined:

```
int Club = 0, Diamond = 1, Heart = 2, Spade = 3 ;
```

The difference is that **`Club`**, **`Diamond`**, ... are (compile-time) constants
You can use them as array sizes and in switch statements
You can specify any or all values explicitly in the enum, for example:

```
enum month { Jan = 1, Feb, Mar, ... } ;
```

Types are sets of values

An enumerated type is a subset of int, and you can define variables:

```
enum suit s;
s = Heart;
```

However, no proper type checking is done:

```
s = 42 ;    // The compiler allows this
```

So, many programmers just use anonymous enumerations to define integer and character constants

```
enum { Width = 500, Height = 500 } ;
enum { Newline = '\n' } ;
```

Conventionally, constants have initial capitals or all capitals

Type synonyms

`typedef` can be used to create synonyms of existing types

```
typedef int counter ;
typedef struct suit suit ;
typedef double image[1280][1024] ;
```

The format of a typedef is the same as a declaration of a variable, except it declares a new type name instead

The **`image`** type, for example, is a 1280 x 1024 array of doubles, which can be used just like any other type:

```
int display( image x ) { ... }
```

Typedefs can be combined with enums

```
typedef enum suit { Club, ... } suit ;
```

Structures

A structure *groups* values, forming aggregate data.
An array holds a variable number of fixed-type items
A structure holds a fixed number of variable-type items

A date is a day,
month and year

`8`	`May`	`1896`

Structures

A structure *groups* values, forming aggregate data.
An array holds a variable number of fixed-type items
A structure holds a fixed number of variable-type items

A date is a day,
month and year

Structures

A structure *groups* values, forming aggregate data.
An array holds a variable number of fixed-type items
A structure holds a fixed number of variable-type items

A date is a day,
month and year

A name consists of
first names and a surname

`8`	`May`	`1896`

`"Igor Fyodorovich"`

`"Stravinsky"`

Structures

A structure *groups* values, forming aggregate data.
An array holds a variable number of fixed-type items
A structure holds a fixed number of variable-type items

A date is a day,
month and year

A name consists of
first names and a surname

Structures

A structure *groups* values, forming aggregate data.
An array holds a variable number of fixed-type items
A structure holds a fixed number of variable-type items

A composer has a name and a birthday

A date is a day,
month and year

A name consists of
first names and a surname

8	May	1896

"Igor Fyodorovich"
"Stravinsky"

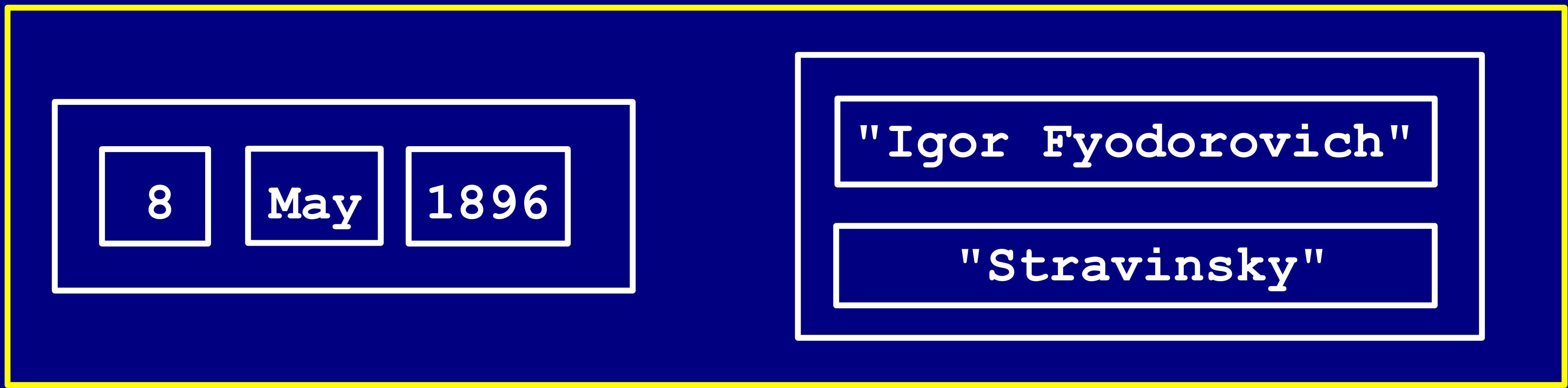
Structures

A structure *groups* values, forming aggregate data.
An array holds a variable number of fixed-type items
A structure holds a fixed number of variable-type items

A composer has a name and a birthday

A date is a day, month and year

A name consists of first names and a surname



The type of structured data

In maths, the type of a date would be

$$\mathbf{N} \times \text{Month} \times \mathbf{N}$$

The type of a name would be

$$\text{String} \times \text{String}$$

The type of a composer would be

$$\text{name} \times \text{date}$$

or

$$(\text{String} \times \text{String}) \times (\mathbf{N} \times \text{Month} \times \mathbf{N})$$

Structures in C

Use the **`struct`** keyword to define a structure type.

```
typedef enum { Jan=1, Feb, ..., Dec } Month ;
struct birthday {
   int day ;                /* order of fields */
   int year ;               /* doesn't matter */
   Month month ;            /* at this point */
} ;
typedef struct birthday Birthday ;
```

This defines

- Types called **`Month`** and **`Birthday`**
  - The type *Birthday* is a **`struct`** with three fields.
  - Two fields are of type **`int`**, one is of type **`Month`**.

Structures in C

Can use **`struct`** and **`typedef`** together.

```
typedef enum { Jan=1, Feb, ..., Dec } Month ;
typedef struct {          // can leave out the tag
   int day ;
   int year ;
   Month month ;
} Birthday ;
```

This defines

- Types called **`Month`** and **`Birthday`**
  - The type *Birthday* is a **`struct`** with three fields.
  - Two fields are of type **`int`**, one is of type **`Month`**.

Structures in C

Can use **`struct`** and **`typedef`** together.

```
typedef enum { Jan=1, Feb, ..., Dec } Month ;
typedef struct {
   int day, year ;    // Can combine

   Month month ;
} Birthday ;
```

This defines

- Types called **`Month`** and **`Birthday`**
  - The type *Birthday* is a **`struct`** with three fields.
  - Two fields are of type **`int`**, one is of type **`Month`**.

Full example

```
typedef enum { Jan=1, Feb, ..., Dec } Month ;
typedef struct {
  int day, year ; Month month ;
} Birthday ;



int main( void ) {
  Birthday stravinsky ;
  stravinsky.day = 8 ;
  stravinsky.month = May ;
  stravinsky.year = 1896 ;
  printf( "%d\n", stravinsky.month ) ;
}
```

The membership operator “.” is used to select a field

Full example

```
typedef enum { Jan=1, Feb, ..., Dec } Month ;
typedef struct {
  int day, year ; Month month ;
} Birthday ;
Month month_of_birth( Birthday b ) {
  return b.month ;
}
int main( void ) {
  Birthday stravinsky ;
  stravinsky.day = 8 ;
  stravinsky.month = May ;
  stravinsky.year = 1896 ;
  printf( "%d\n", month_of_birth ( stravinsky ) ) ;
}
```

You can pass a struct to a function (by value - it is copied)

Full example

```
typedef enum { Jan=1, Feb, ..., Dec } Month ;
typedef struct {
  int day, year ; Month month ;
} Birthday ;
Month month_of_birth( Birthday b ) {
  return b.month ;
}
int main( void ) {
  Birthday stravinsky = {8, 1896, May} ;



  printf( "%d\n", month_of_birth ( stravinsky ) ) ;
}
```

When you initialise a struct you can use { } with a following ;

So...

Grouped data:

- Need to define a type, with a name
  - List the types that we are going to store inside as fields
- Need some kind of constructor
  - An operation to create a structure of that type
- Need an operation to look inside
  - Use the ‘ . ’ operator in C

Another example

Let's define a type point, in the plane XY, consisting of two reals:

$$\text{Point} = R \times R$$

If (x,y) is a point, then the point rotated with an angle ϕ is given by

$$(x \cos\phi + y \sin\phi,\ y \cos\phi - x \sin\phi)$$

Define a function rotate:

$$\text{rotate}((x,y),\phi) = (x \cos\phi + y \sin\phi,\ y \cos\phi - x \sin\phi)$$

Rotate with C structures

```
typedef struct {
  double x, y ;
} Point ;
```

Rotate with C structures

```
typedef struct {
  double x, y ;
} Point ;

Point rotate( Point s, double phi ) {
  Point t ;
  double sin_phi = sin( phi ) ;
  double cos_phi = cos( phi ) ;
  t.x = s.x * cos_phi + s.y * sin_phi ;
  t.y = s.y * cos_phi - s.x * sin_phi ;
  return t ;
}
```

Union types

A union type (also known as a variant) allows you to store either X or Y in a data type. Use **`union`** in C.

```
typedef struct {
  double d, alpha ;
} Polar ;

typedef struct {
  double x, y ;
} Cartesian ;

typedef union {
  Polar p ;
  Cartesian c ;
} Point ;
```

Alternatives

```
typedef union {
  Polar p ;
  Cartesian c ;
} Point ;
```

Defines a data type **`Point`**, which consists of

- Either a cartesian part (with two floats),
- Or a polar description (with two floats).

(or, for example, a vehicle which is a car, truck, or caravan)

A member of a union is accessed with the membership operator ‘.’
(The members of a struct are accessed using ‘.’)
So:

- if **`s`** is of type Point, and currently contains Cartesian coords,
- then **`s.c.x`** refers to member **`x`** of member **`c`** of **`s`**

Alternatives

```
int main( void ) {
  Point s, t ;
  s.c.x = 2 ;
  s.c.y = 1 ;
  t.p.d = 2.236 ;
  t.p.alpha = 0.4636 ;
}
```

`s` specifies the point (2,1) using Cartesian Coordinates

- 2 along the X axis, 1 along the Y axis

`t` specifies the point (2,1) using Polar Coordinates

- An angle of 0.4636 radians, a distance of 2.236 ($\sqrt{5}$)

Difference between Struct and Union

```
typedef union {
  Polar p ;
  Cartesian c ;
  int i ;
} uniontype ;

uniontype u ;
```

```
typedef struct {
  Polar p ;
  Cartesian c ;
  int i ;
} structtype ;

structtype s ;
```

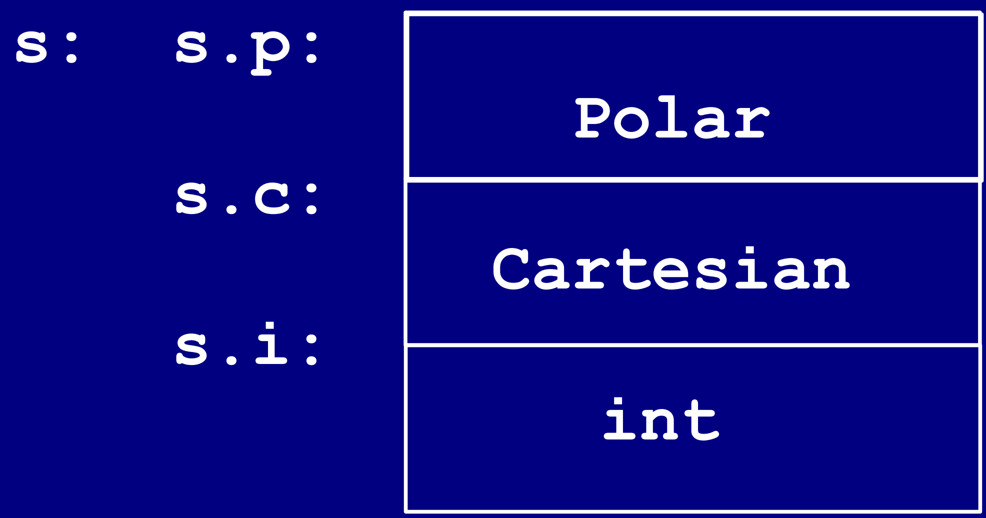
Difference between Struct and Union

```
typedef union {
  Polar p ;
  Cartesian c ;
  int i ;
} uniontype ;

uniontype u ;
```

```
typedef struct {
  Polar p ;
  Cartesian c ;
  int i ;
} structtype ;

structtype s ;
```



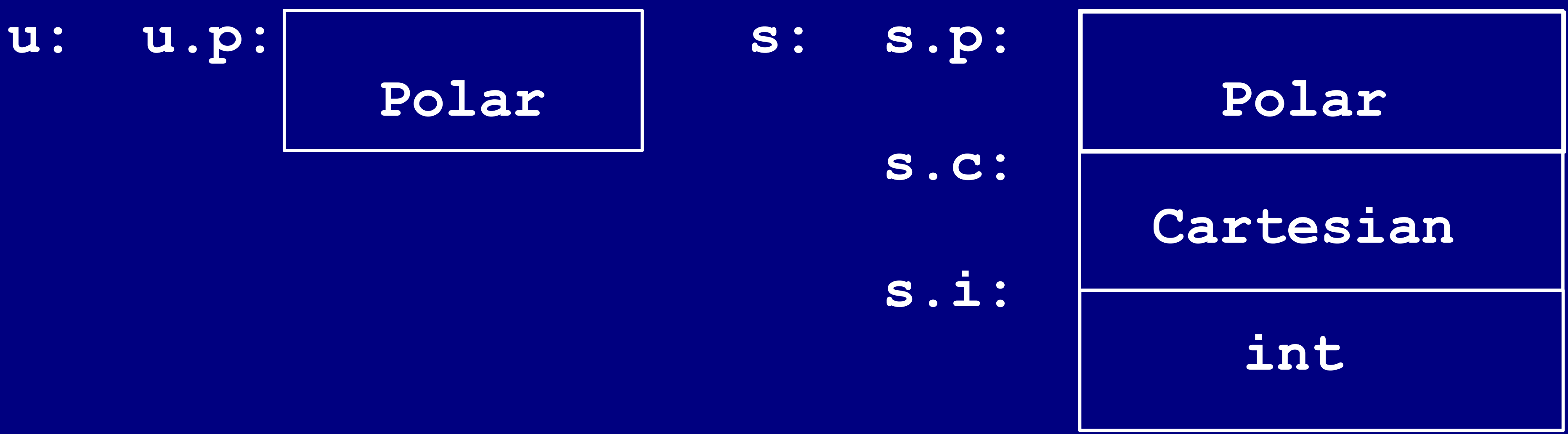
Difference between Struct and Union

```
typedef union {
   Polar p ;
   Cartesian c ;
   int i ;
} uniontype ;

uniontype u ;
```

```
typedef struct {
   Polar p ;
   Cartesian c ;
   int i ;
} structtype ;

structtype s ;
```



Difference between Struct and Union

```
typedef union {
   Polar p ;
   Cartesian c ;
   int i ;
} uniontype ;

uniontype u ;
```

```
typedef struct {
   Polar p ;
   Cartesian c ;
   int i ;
} structtype ;

structtype s ;
```

u: u.c: Cartesian

s: s.p: Polar

s.c: Cartesian

s.i: int

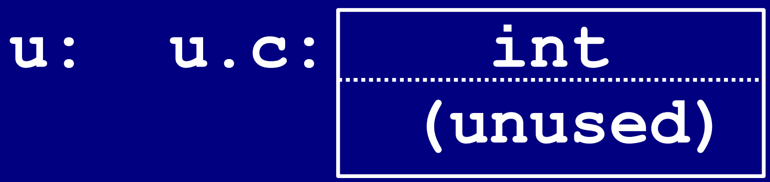
Difference between Struct and Union

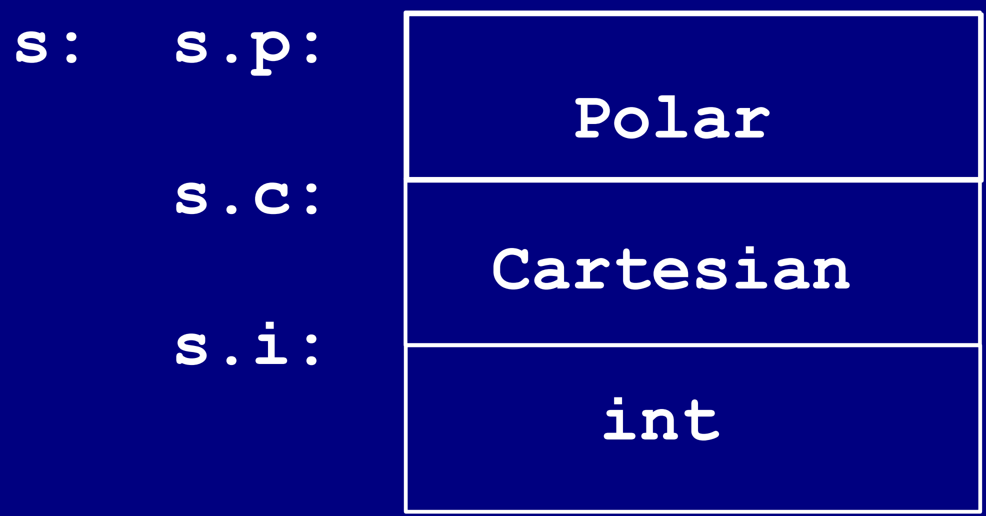
```
typedef union {
   Polar p ;
   Cartesian c ;
   int i ;
} uniontype ;

uniontype u ;
```

```
typedef struct {
   Polar p ;
   Cartesian c ;
   int i ;
} structtype ;

structtype s ;
```





What is currently in a union?

```
typedef struct {

  union {
    Polar p ;
    Cartesian c ;
  } pt ;
} Point ;
```

There is no way to tell - it is up to the programmer to keep track
One way is to:

- Maintain an extra field, a "tag"
- The tag specifies which type is currently stored in the inner union

What is currently in a union?

```
typedef enum { IsPolar, IsCartesian } Pointtag ;

typedef struct {
  Pointtag tag ;
  union {
    Polar p ;
    Cartesian c ;
  } pt ;
} Point ;
```

There is no way to tell - it is up to the programmer to keep track

Creating the union

```
int main( void ) {
  Point s, t ;
  s.tag = IsCartesian ;
  s.pt.c.x = 2 ;
  s.pt.c.y = 1 ;

  t.tag = IsPolar ;
  t.pt.p.d = 2.236 ;
  t.pt.p.alpha = 0.4636 ;
}
```

By inspecting the tag, the function **`rotate`** can now be defined properly

Using the union

```
Point rotate( Point s, double phi ) {
  Point t ;
  if( s.tag == IsCartesian ) {
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
  } else {
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
  }
  return t ;
}
```

Using the union

```
Point rotate( Point s, double phi ) {
  Point t ;
  if( s.tag == IsCartesian ) {
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
  } else {
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
  }
  return t ;
}
```

Using the union

```
Point rotate( Point s, double phi ) {
  Point t ;
  if( s.tag == IsCartesian ) {
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
  } else {
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
  }
  return t ;
}
```

Using the union

```
Point rotate( Point s, double phi ) {
  Point t ;
  if( s.tag == IsCartesian ) {
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
  } else {
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
  }
  return t ;
}
```

Using a switch statement instead

```
Point rotate( Point s, double phi ) {
  Point t ;
  switch ( s.tag ) {
  case IsCartesion:
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    break;
  case IsPolar:
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
    break;
  }
  return t ;
}
```

Using a switch statement instead

```
Point rotate( Point s, double phi ) {
  Point t ;
  switch ( s.tag ) {
  case IsCartesion:
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    break;
  case IsPolar:
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
    break;
  }
  return t ;
}
```

Using a switch statement instead

```
Point rotate( Point s, double phi ) {
  Point t ;
  switch ( s.tag ) {
  case IsCartesion:
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    break;
  case IsPolar:
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
    break;
  }
  return t ;
}
```

Using a switch statement instead

```
Point rotate( Point s, double phi ) {
  Point t ;
  switch ( s.tag ) {
  case IsCartesion:
    t.tag = IsCartesian ;
    t.pt.c.x=s.pt.c.x*sin(phi)+s.pt.c.y*cos(phi);
    t.pt.c.y=s.pt.c.y*sin(phi)-s.pt.c.x*cos(phi);
    break;
  case IsPolar:
    t.tag = IsPolar ;
    t.pt.p.alpha = phi + s.pt.p.alpha ;
    t.pt.p.d = s.pt.p.d ;
    break;
  }
  return t ;
}
```

Summarising Flat Data

Constructs:

```
typedef enum { value1, value2, ... } typename ;
typedef struct { type1 x1 ; type2 x2 ... } typename
typedef union { type1 x1 ; type2 x2 ... } typename
switch( value ) { case 1: ... ; break ; case 2: ...
```