Structs and Unions

In C, a `struct` type is a collection of data elements grouped together

The elements inside a `struct` are called members in C, although many (including me) call them fields

You can declare struct types like this, typically outside functions:

```
struct person {
        char forename[80];
        char surname[80];
        int age;
};
```

Structs do not have constructors, destructors or member functions

The name of our structure type is `struct XXX`

Once you have defined your structure type, you can define variables of that type, within a function, by:

```
int main( void ) {
        struct point p1;
        ...
```

You then access the fields of a struct using the field selector operator ".":

```
#include <stdio.h>
struct point { double x, y; };

int main( void ) {
        struct point p1;
        p1.x = 10;
        p1.y = 20;
        printf( "(%g, %g)\n", p1.x, p1.y );
```

```
        return 0;
}
```

You can also use initialiser syntax:

```
struct point {
        double x, y;
};
struct point p = {10, 20};
```

Or this initialiser syntax:

```
struct point {
        double x, y;
};

struct point p = {.y = 20, .x = 10};
```

Structs have value semantics

Structs can be assigned, and passed/returned from functions by value, as in:

```
#include <stdio.h>

struct point { double x, y; };

struct point flipx( struct point p ) {
        p.x = -p.x;
        return p;
}

int main( void ) {
        struct point p = { 5.0, 5.0 };
        printf( "(%g, %g)\n", p.x, p.y );
        struct point flipped = flipx( p );
        printf( "(%g, %g)\n", flipped.x, flipped.y );
}
```

Structs are the only multi-word values that are copied in C

Structs by reference - pointers to structs

We can also choose to pass structs by reference - as explicit pointers via `&p` - in order to change the fields of the structure:

```
#include <stdio.h>

struct point { double x, y; };

void flipx( struct point *pp ) {
        pp->x = -pp->x;
}

int main( void ) {
        struct point p = { 5.0, 5.0 };
        flipx( &p );
}
```

Rather than dereferencing pointers to structs and then using the field selection operator, as in `(*pp).x`, you can use the more convenient structure pointer dereference operator `pp->x`

You can declare a 1-D array of structs:

```
#include <stdio.h>

struct point { double x, y; };

void flip( struct point *p, int position ) {
        p[position].x = -p[position].x;
}

int main( void ) {
        struct point p[3];
        for( int i=0; i<3; i++ ) {
                p[i].x = 2.0 * i;
```

```
                p[i].y = 4.0 * i;
        }

        for( int i=0; i<3; i++ ) {
                printf( "(%g, %g)\n", p[i].x, p[i].y );
                flip( p, i );
                printf( "(%g, %g)\n", p[i].x, p[i].y );
        }
        return 0;
}
```

Just as we did with `enum`, we can use `typedef` to declare a new named type:

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

point p1;
point *p2;
```

An Example:

```c
#include <stdio.h>

typedef struct {
  double real, imag;
} complex;

void printComplex( complex *mc ) {
  printf( "%lg + %lgi\n", mc->real, mc->imag );
}

int main( void )
{
  complex c1 = {1.0, 0.5};
  printComplex( &c1 );      // pass pointer to struct
  c1.real = 10.0;           // piecewise assignment
  printComplex( &c1 );
  return 0;
}
```

- When we compile this and run it, we get:

```
./complex
1 + 0.5i
10 + 0.5i
```

Union Types

Unions are syntactically similar to structs (essentially: replace the keyword `struct` with the keyword `union`, the rest of the syntax is identical)

Unions are much rarer than structs

While a struct stores one value in every field, a union can only store a single optional value - which can be any of the fields. They're used in situations where the requirements to store the different fields are mutually exclusive

Thus, a union only holds enough storage space for the largest field

Consequently, assigning to one field of a union overwrites the value assigned to all the other fields. It is only safe to read from the last field of the union that was assigned to. It's the programmer's responsibility to make sure that they read from the right field. The C compiler doesn't keep track of which field (if any!) was last written to.

We said that a union only holds enough storage space for the largest field

For Example:

```
#include <stdio.h>

typedef union { char c; int i; } myunion;
```

```
sizeof(char)    = 1
sizeof(int)     = 4
sizeof(myunion) = 4
```

However, the C compiler is allowed to add padding, so `sizeof(myunion)` could be surprising in some circumstances

Union Memory Layout

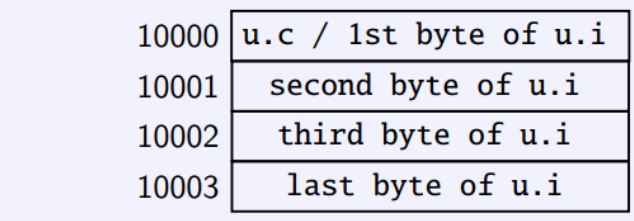
We said that assigning to a field of a union will overwrite values assigned to any other field

This is because the union and all its fields have the same starting address:

```
typedef union {
  char c;
  int i;
} myunion;

myunion u;
```

Memory allocated for u




```
&u = 10000
&u.i = 10000
&u.c = 10000
```

This may be directly exploited to provide another method of determining Endianism.

```c
#include <stdio.h>
#include <stdint.h>

typedef union { char c; uint32_t i; } myunion;

int main( void ) {
  myunion u;
  u.i = 0x12345678;
  printf( "%s endian\n", u.c == 0x12 ? "big" : "little" );
  return 0;
}
```

Here we are exploiting the fact that `u.i` and `u.c` are stored in memory at the same location, so if we write to `u.i` then read from `u.c` we must either get the least significant byte of `u.i` or the most significant byte - depending on the platform's endianism

Overwriting Fields

We said that it is only safe to read from the last field of the union that was assigned to

An Example:

```c
#include <stdio.h>

typedef union { char c; int i; } myunion;

void print_myunion( myunion u ) { printf( "u.i: %d (hex %x), u.c: %c\n", u.i, u.i, u.c ); }

int main( void ) {
  myunion u;
  u.i = 0;
  printf("Setting u.c to 'q'...    "); u.c = 'q';    print_myunion(u);
  printf("Setting u.i to 0x1234... "); u.i = 0x1234; print_myunion(u);
  printf("Setting u.c to 'q'...    "); u.c = 'q';    print_myunion(u);
  return 0;
}
```

Running this we get:

```
Setting u.c to 'q'...    u.i: 113 (hex 71), u.c: q
Setting u.i to 0x1234... u.i: 4660 (hex 1234), u.c: 4
Setting u.c to 'q'...    u.i: 4721 (hex 1271), u.c: q
```

See how the value of `u.i` changes when we update `u.c`: One byte gets overwritten, the other 3 are unchanged

How do we safely use unions, given that the programmer has to remember which (if any) union field is in use?

The typical way of using unions is to build something called a variant record in some languages. Suppose we want to represent a Haskell-style Inductive Type to represent some geometrical shapes:
`data shape = point( x, y ) or circle( x, y, radius ) or rectangle( x, y, x2, y2 )`

We observe that all 3 shapes contain an `(x, y)` coordinate, then some variable data. So we start with a coordinate:
`typedef struct { double x,y; } coordinate;`

Next, we define an enumerated type to say which shape we're dealing with:
`typedef enum { shape_is_point, shape_is_circle, shape_is_rectangle } shapetag;`

Next, we define a type to store the variable part of our data: nothing if it's a point, a radius if it's a circle, or a second coordinate if it's a rectangle:

```
typedef union {
        double radius; // for a circle
        coordinate c; // for a rectangle, 2nd coord (x2,y2) }
shapevariant;
```

Pulling this all together, all shapes will need a coordinate and a tag (this is the data common to all shapes), and then the variable data:

```
typedef struct {
        coordinate c;
        shapetag tag;
        shapevariant v;
} shape;
```

Moving onto functions, we'll need a coordinate constructor:

```
// coordinate c = make_coordinate( x, y );
// Construct and return a coord (x,y)

coordinate make_coordinate( double x, double y ) {
        coordinate c;
        c.x = x;
        c.y = y;
        return c;
}
```

Then we'll need one constructor for each shape

- First, the `point()` constructor:

```
// shape s = make_shape_point( x, y );
//   Construct and return a point (x,y) shape
shape make_shape_point( double x, double y ) {
  shape s;
  s.c   = make_coordinate( x, y );
  s.tag = shape_is_point;
  return s;
}
```

- Second, the `circle()` constructor:

```
// shape s = make_shape_circle( x, y, r );
//   Construct and return a circle shape
//   with centre (x,y) of radius r
shape make_shape_circle( double x, double y, double radius ) {
  shape s;
  s.c   = make_coordinate( x, y );
  s.tag = shape_is_circle;
  s.v.radius = radius;
  return s;
}
```

- Finally, the `rectangle()` constructor:

```
// shape s = make_shape_rectangle( x, y, x2, y2 );
//    Construct and return a rectangle with corners (x,y) and (x2, y2)
shape make_shape_rectangle( double x, double y, double x2, double y2 ) {
  shape s;
  s.c   = make_coordinate( x, y );
  s.tag = shape_is_rectangle;
  s.v.c = make_coordinate( x2, y2 );
  return s;
}
```

- I believe that every data type needs a print me out operator, so let's add:

```
void print_shape( shape s ) {
  double x = s.c.x; double y = s.c.y;
  switch( s.tag ) {
  case shape_is_point:
    printf( "point(%g,%g)", x, y ); break;
  case shape_is_circle:
    double r = s.v.radius;
    printf( "circle(%g,%g,%g)", x, y, r ); break;
  case shape_is_rectangle:
    double x2 = s.v.c.x; double y2 = s.v.c.y;
    printf( "rectangle(%g,%g,%g,%g)", x, y, x2, y2 ); break;
  }
}
```

- Now, we can write a `main()` function that exercises our shapes:

```
int main( void ) {
  shape shapes[10]; int nshapes = 0;
  shapes[nshapes++] = make_shape_point( 10,20 );
  shapes[nshapes++] = make_shape_circle( 1,2,10 );
  shapes[nshapes++] = make_shape_rectangle( 0,0, 10,10 );
  for( int i=0; i<nshapes; i++ ) {
    printf( "shapes[%d] = ", i ); print_shape( shapes[i] ); putchar( '\n' );
  }
  return 0;
}
```

- We collect all the code from the last 4 slides together, plus a few necessary `#include` statements - giving example unions/shapes.c.
- Having compiled this, and gone all this way, of course we want to see it work:

```
cb shapes
./shapes
shapes[0] = point(10,20)
shapes[1] = circle(1,2,10)
shapes[2] = rectangle(0,0,10,10)
```

- Shiny!
- Exercises: Try adding an extra shape (for example, a straight line), an extra common attribute (eg a colour), an extra function (eg an area calculating function).
- By the way, if this seems very repetitive - and therefore easy to generate automatically - see my tool datadec in my third C Tools lecture, which does more or less exactly this - but with arbitrarily complicated (and self-referential) Inductive Data Types, using pointers to structures throughout.