Structure Example (preview)

- This declaration introduces the type struct fraction (both words are required) as a new type.
- C uses the period (.) to access the fields in a record.
- You can copy two records of the same type using a single assignment statement, however == does not work on structs (see note link).

```
struct fraction {
   int numerator;
   int denominator;          // can't initialize
};

struct fraction f1, f2;       // declare two fractions
f1.numerator = 25;
f1.denominator = 10;
f2 = f1;                      // this copies over the whole struct
```

Structure Declarations

struct tag {member_list} variable_list;

```
struct S {
    int a;
    float b;
} x;
```

Declares x to be a structure having two members, a and b. In addition, the structure tag S is created for use in future declarations.

```
struct {
int a;
float b;
} z;
```

Omitting the tag field; cannot create any more variables with the same type as z

```
struct S {
int a;
float b;
};
```

Omitting the variable list defines the tag S for use in later declarations

```
struct S y;
```

Omitting the member list declares another structure variable y with the same type as x

```
struct S;
```

Incomplete declaration which informs the compiler that S is a structure tag to be defined later

Structure Declarations (cont)

- So tag, member_list and variable_list are all optional, but cannot all be omitted; at least two must appear for a complete declaration.

```
struct {
   int a;
   char b;
   float c;
} x;
```

Single variable x contains 3 members

Structs on the left are treated different by the compiler
DIFFERENT TYPES
i.e. z = &x is ILLEGAL

```
struct {
   int a;
   char b;
   float c;
} y[20], *z;
```

An array of 20 structures (y); and
A pointer to a structure of this type (z)

More Structure Declarations

- The TAG field
  - Allows a name to be given to the member list so that it can be referenced in subsequent declarations
  - Allows many declarations to use the same member list and thus create structures of the same type

```
struct  SIMPLE {
   int a;
   char b;
   float c;
} ;
```

Associates tag with member list; does not create any variables

So ➔ struct SIMPLE x;
struct SIMPLE y[20], *z;

Now x, y, and z are all the same kind of structure

Incomplete Declarations

- Structures that are mutually dependent
- As with self referential structures, at least one of the structures must refer to the other only through pointers
- So, which one gets declared first???

```
struct B;

struct A  {
      struct  B   *partner;
      /* etc */
} ;

struct B  {
      struct  A   *partner;
      /* etc */
} ;
```

- **Declares an identifier to be a structure tag**
- **Use this tag in declarations where the size of the structure is not needed (pointer!)**
- **Needed in the member list of A**

- **Doesn't have to be a pointer**

Initializing Structures

- Missing values cause the remaining members to get default initialization… whatever that might be!

```
typedef struct  {
    int       a;
    char      b;
    float     c;
}  Simple;

struct INIT_EX  {
    int       a;
    short     b[10];
    Simple  c;
}  x = {  10,
          { 1, 2, 3, 4, 5 } ,
          { 25, 'x', 1.9 }
} ;
```

What goes here (hint in blue below)?

```
struct INIT_EX   y = { 0 , {10,  20, 30, 40, 50,
                            60, 70, 80, 90, 100 },
                          { 1000, 'a', 3.14 }
                     } ;
```

Name all the variables and their initial values:

y.a = 0

y.b[0] = 10; y.b[1] = 20; y.b[2] = 30; etc

y.c.a = 1000;

y.c.b = 'a';

y.c.c = 3.14;

Structures as Function arguments

- Legal to pass a structure to a function similar to any other variable but often inefficient

```
/* electronic cash register individual
transaction receipt */
#define PRODUCT_SIZE   20;
typedef struct  {
    char     product[PRODUCT_SIZE];
    int      qty;
    float    unit_price;
    float    total_amount;
}  Transaction;
```

- **Function call:**
  - print_receipt(current_trans);
  - Copy by value copies 32 bytes to the stack which can then be discarded later
- **Instead…**
  - (Transaction *trans)
  - trans->product // fyi: (*trans).product
  - trans->qty
  - trans->unit_price
  - trans->total_amount
  - print_receipt(&current_trans);
  - void print_receipt(Transaction *trans)

```
void print_receipt (Transaction trans)  {
  printf("%s\n, trans.product);
  printf(%d @ %.2f total %.2f\n", trans.qty, trans.unit_price, trans.total_amount);
}
```

Struct storage issues

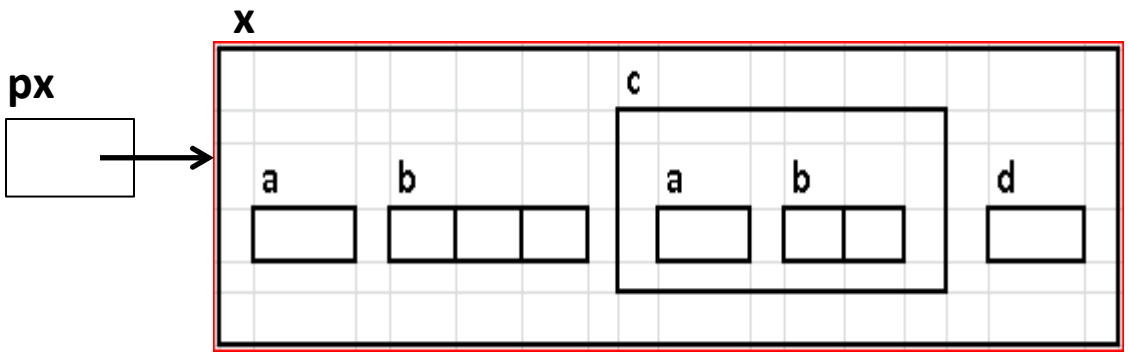
- A struct declaration consists of a list of fields, each of which can have any type. The total storage required for a struct object is the sum of the storage requirements of all the fields, plus any **internal padding**.

Structure memory (again)

- What does memory look like?

```
typedef struct  {
    int       a;
    short    b[2];
}  Ex2;

typedef struct EX  {
    int        a;
    char      b[3];
    Ex2       c;
    struct    EX  *d;
}  Ex;
```



Given the following declaration, fill in the above memory locations:

Ex x = { 10, "Hi", { 5 , { -1, 25 } } , 0 };

Ex *px = &x;