Structures, Unions, and Typedefs

Cuauhtémoc Carbajal

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)

Reading Assignment

- Chapter 6 of Kernighan & Ritchie
  Chapter 10 of Deitel & Deitel

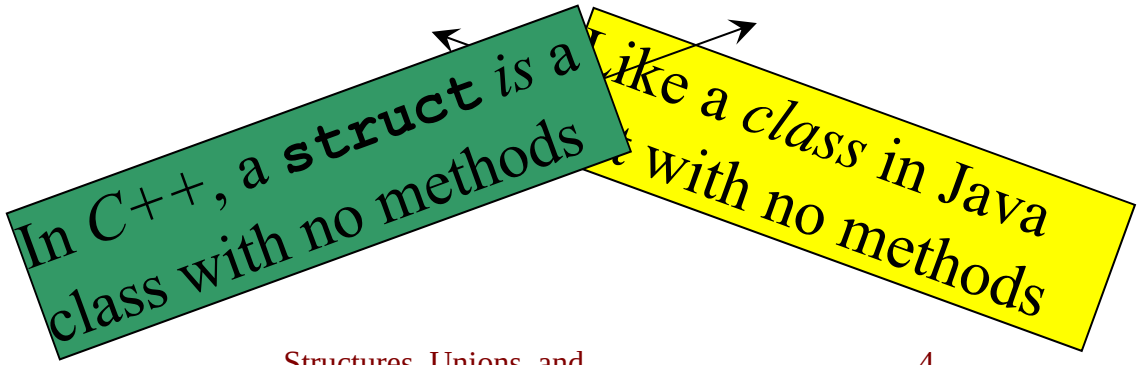
Structures and Unions

- Essential for building up “interesting” data structures — e.g.,
  - Data structures of multiple values of different kinds
  - Data structures of indeterminate size

Definition — *Structure*

- A collection of one or more variables, typically of different types, grouped together under a single name for convenient handling

- Known as **`struct`** in *C* and *C++*



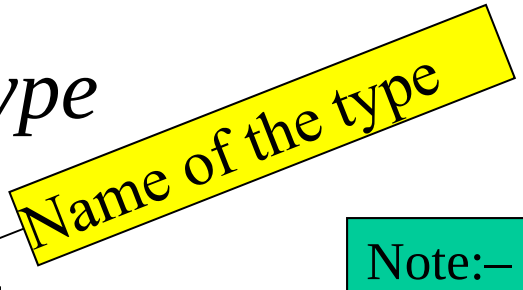
struct

- Defines a new *type*
  - I.e., a new kind of data type that compiler regards as a unit
- E.g.,

```
struct motor {
  float volts;     //voltage of the motor
  float amps;      //amperage of the motor
  int phases;      //# of phases of the motor
  float rpm;       //rotational speed of motor
};       //struct motor
```

`struct`

- Defines a new *type*
- E.g.,



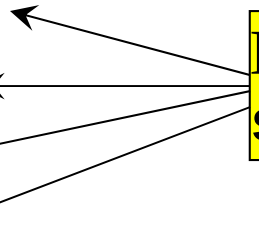

```
struct motor {
  float volts;
  float amps;
  int phases;
  float rpm;
};        //struct motor
```

Note:– name of type is optional if you are just declaring a single **`struct`** (middle p. 128 of K&R)

`struct`

- Defines a new *type*
- E.g.,

```
struct motor {
  float volts;
  float amps;
  int phases;
  float rpm;
};        //struct motor
```



Members of the **`struct`**

A *member* of a **`struct`** is analogous to a *field* of a class in Java

Declaring `struct` variables

```
struct motor p, q, r;
```

- Declares and sets aside storage for three variables – **`p`**, **`q`**, and **`r`** – each of type **`struct motor`**

```
struct motor M[25];
```

- Declares a 25-element array of **`struct motor`**; allocates 25 units of storage, each one big enough to hold the data of one **`motor`**

```
struct motor *m;
```

- Declares a pointer to an object of type **`struct motor`**

Accessing Members of a `struct`

- Let

```
struct motor p;
struct motor q[10];
```

- Then

`p.volts`	— is the voltage
`p.amps`	— is the amperage
`p.phases`	— is the number of phases
`p.rpm`	— is the rotational speed
`q[i].volts`	— is the voltage of the **`i`**th motor
`q[i].rpm`	— is the speed of the **`i`**th motor

Like Java!

Accessing Members of a **`struct`** (continued)

- Let

  `struct motor *p`

- Then

  Why the parentheses?

  `(*p).volts` — is the voltage of the **`motor`** pointed to by **`p`**

  `(*p).phases` — is the number of phases of the **`motor`** pointed to by **`p`**

Accessing Members of a **`struct`** (continued)

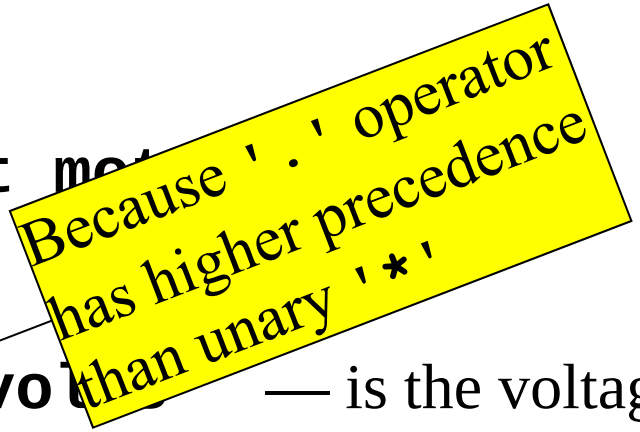
- Let

  `struct mo`

- Then

  `(*p).vo` — is the voltage of the **`motor`** pointed to by **`p`**

  `(*p).phases` — is the number of phases of the **`motor`** pointed to by **`p`**



Accessing Members of a `struct` (continued)

- Let

  ```
  struct motor *p;
  ```

- Then

  ```
  (*p).volts
  ```

  ```
  (*p).phases
  ```

Reason:– you really want the expression

```
m.volts * m.amps
```

to mean what you think it should mean!

Accessing Members of a **`struct`** (continued)

- The **`(*p).member`** notation is a nuisance
  - Clumsy to type; need to match **`( )`**
  - Too many keystrokes

- This construct is so widely used that a special notation was invented, i.e.,
  - **`p->member`**, where **`p`** is a pointer to the structure

- Ubiquitous in *C* and *C++*

Previous Example Becomes …

- Let

  ```
  struct motor *p;
  ```

- Then

  `p -> volts` — is the voltage of the **`motor`** pointed to by **`p`**

  `p -> phases` — is the number of phases of the **`motor`** pointed to by **`p`**

Operations on `struct`

- Copy/assign

```
struct motor p, q;
p = q;
```

- Get address

```
struct motor p;
struct motor *s
s = &p;
```

- Access members

```
p.volts;
s -> amps;
```


Operations on `struct` (continued)

- Remember:–
  - Passing an argument by value is an instance of *copying* or *assignment*
  - Passing a return value from a function to the caller is an instance of *copying* or *assignment*
- E.g,:–

```
struct motor f(struct motor g) {
  struct motor h = g;
  ...;
  return h;
}
```


Assigning to a `struct`

- K & R say (p. 131)
  - "If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure"
- *I disagree:–*
  - Copying is *very* fast on modern computers
  - Creating an object with `malloc()` and assigning a pointer is not as fast
  - Esp. if you want the object passed or returned by value
  - In real life situations, it is a judgment call

Initialization of a `struct`

- Let
```
struct motor {
        float volts;
        float amps;
        int phases;
        float rpm;
};          //struct motor
```
- Then

  `struct motor m = {208 ... 300};`
  initializes the `struct`

- See also p. 133 of K&R for initializing arrays of `structs`

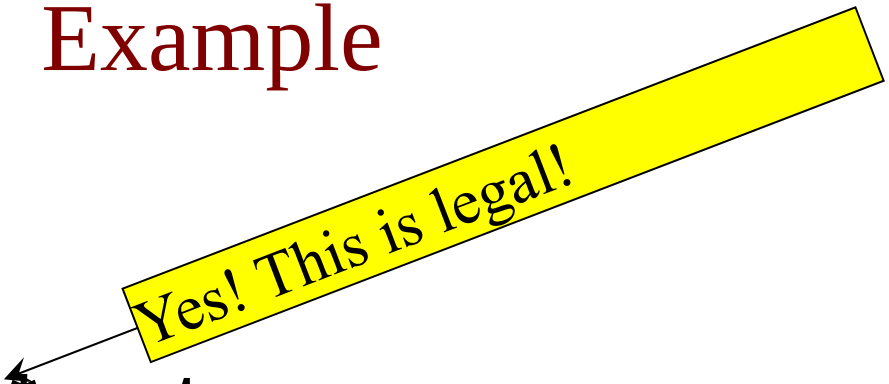
C99 introduces a new way of initializing a `struct` – truly ugly!

Why `structs`?

- Open-ended data structures
  - E.g., structures that may grow during processing
  - Avoids the need for **`realloc()`** and a lot of copying

- Self-referential data structures
  - Lists, trees, etc.

Example




```
struct item {
  char *s;
  struct item *next;
}
```

- I.e., an **`item`** can point to another **`item`**
- … which can point to another **`item`**
- … which can point to yet another **`item`**
- … etc.

Thereby forming a *list* of **`items`**

A note about **`structs`** and pointers

- The following is legal:–

```
/* in a .c or .h file */
struct item;
struct item *p, *q;

...    /* In another file */
struct item {
  int member1;
  float member2;
  struct item *member3;
};
```

Called an *opaque type!*
Program can use *pointers* to items but cannot see *into* items.
Cannot define any items, cannot malloc any items, etc.

Implementer of item can change the definition without forcing users of pointers to change their code!

Another note about **`structs`**

- The following is *not* legal:–

```
struct motor {
        float volts;
        float amps;
        float rpm;
        unsigned int phases;
 };  //struct motor
```

```
motor m;
motor *p;
```

You must write

```
struct motor m;
struct motor *p;
```


Typedef

- Definition:– a **`typedef`** is a way of *renaming* a type
  - See §6.7
- E.g.,

```
typedef struct motor M

Motor m, n;
Motor *p, r[25];
Motor function(const Motor m; …);
```

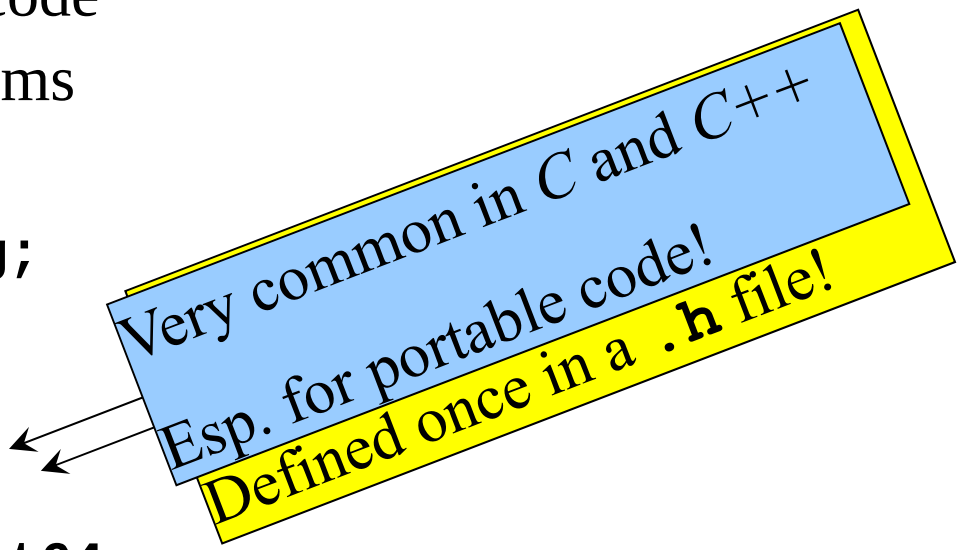
E.g., **`typedef`**, lets you leave out the word "**`struct`**"

`typedef` (continued)

- **`typedef`** may be used to rename *any* type
  - Convenience in naming
  - Clarifies purpose of the type
  - Cleaner, more readable code
  - Portability across platforms
- E.g.,
  - **`typedef char *String;`**
- E.g.,
  - **`typedef int size_t;`**
  - **`typedef long int32;`**
  - **`typedef long long int64;`**

`typedef` (continued)

- **`typedef`** may be used to rename *any* type
  - Convenience in naming
  - Clarifies purpose of the type
  - Cleaner, more readable code
  - Portability across platforms
- E.g.,
  - **`typedef char *String;`**
- E.g.,
  - **`typedef int size_t;`**
  - **`typedef long int32;`**
  - **`typedef long long int64;`**



Revisit note about **`structs`** and pointers

- The following *is* legal:–

```
/* in a .c or .h file */
typedef struct _item Item;
Item *p, *q;

...    /* In another file */
struct _item {
  char *info;
  Item *nextItem;
};
```


Questions about **`structs`** and pointers?

Unions

- A **`union`** is like a **`struct`**, but only one of its members is stored, not all
  - I.e., a single variable may hold different types at different times
  - Storage is enough to hold largest member
  - Members are overlaid on top of each other
- E.g.,

```
union {
  int ival;
  float fval;
  char *sval;
} u;
```


Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a **`union`** at any given time!
- E.g., (p. 148)

```
struct taggedItem {
  enum {iType, fType, cType} tag;
  union {
   int ival;
   float fval;
   char *sval;
  } u;
};
```


Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a **`union`** at any given time!
- E.g., (p. 148)

```
struct taggedItem {
  enum {iType, fTyp
  union {
    int ival;
    float fval;
    char *sval;
  } u;
};
```

Members of **`struct`** are:–

```
enum tag;
union u;
```

Value of **`tag`** says which member of **`u`** to use

Unions (continued)

- **`unions`** are used much less frequently than **`structs`** — mostly
  - in the inner details of operating system
  - in device drivers
  - in embedded systems where you have to access registers defined by the hardware

Questions?