Structures

- Structures are ***heterogenous collections*** of **variables.**

```
struct date {
    int day;
    char month[4];
    int year;
};
```

declares the structure **`date`**, but does ***not*** allocate space

- **`struct date`** can be used like **`int`** and **`char`**, e.g. to declare variables

```
struct date birthday, *graduation;
```

- Structure declarations can be ***combined*** with variable definitions

```
struct date { ... } birthday, *graduation;
```

- ***external*** and ***static*** local structures can be ***initialized*** at compile time:

```
struct date independence = { 4, "Jul", 1776 };
```

- Structures can be ***nested***

```
struct person {
    char name[30];
    long ssn;
    struct date birthday;
} p;
```


Fields

- Structure fields are accessed by ***variable*.*field***

```
struct person employee, employees[100];

employee.birthday.month
employees[i].name[j]
```

- ***structure pointers*** point to instances of structures

```
struct date d, *pd;

pd = &d;
d = *pd;
```

structure assignment is legal!

- "**->**" references a field in a structure pointed to by a pointer

`pd->month` equivalent to **`(*pd).month`**

- Structures can contain pointers; **`->`** associates to the ***left***

```
struct tree {
    struct date d;
    struct tree *l, *r;
} *p;
```

```
p->l->l->l->d.month;
```


Pointers to Structures

- Manipulating pointers to structures:

`struct foo { int x, *y; } *p;`

`++p->x`	increments field **`x`** in **`*p`**
`(++p)->x`	increments **`p`**, then refers to field **`x`**
`*p->y++`	return **`int`** pointed to by field **`y`** in **`*p`**, increments **`y`**
`*p++->y`	return **`int`** pointed to by field **`y`** in **`*p`**, increment **`p`**

- An ***array of structures*** is the preferred method for storing a table

```
#define NKEYS 100

struct key {
    char *keyword;
    int keycount;
} keytab[NKEYS];
```

"the old way:"

```
char *keyword[NKEYS];
int keycount[NKEYS];
```


Arrays of Structures

- Easy to initialize such tables:

```
struct key keytable[] = {
    { "auto", 0, },
    { "break", 0, },
    ...
    { "while", 0 }
}
```

- Easy to search them:

```
int i;

for (i = 0; i < NKEYS; i++)
    if (strcmp(word, keytable[i].keyword) == 0)
        ...
```

Sizeof

- `sizeof` *x* is a ***compile-time operator*** that gives the size of *x* in bytes

  x can be **(*type*)** or ***expression***

```
sizeof (int)           4
sizeof (int *)         4
sizeof (struct key *)  4
sizeof (struct key)    8

sizeof keytable        NKEYS*sizeof (struct key)
```

- Use `sizeof` to define parameters

```
#define NKEYS (sizeof keytable/sizeof (struct key))
```

- Examples

```
int a[10];
struct operator { char key; void(*f)(int, int); } b[3], o, *p;

sizeof a    40
sizeof b    24
sizeof o    8
sizeof p    4
sizeof *p   8
```


Unions

- Unions provide a way to use ***different types*** for data in a ***single storage*** area

```
union u {
    double fval;
    int ival;
    char cval;
} uval;
```

`uval.fval`	double
`uval.ival`	integer
`uval.cval`	character

- Union size is equal to the `sizeof` the largest field

  `sizeof uval` 8

- **No validity checks**

Unions, cont'd

- Unions often appear in structures to reduce space

```
struct value {
    enum { Integer, Real, Character } type;
    union u val;
} values[100];
```

`type` — a "type tag" — keeps track of the type stored in `val`

- Check type tag before accessing union fields:

```
void print(int i) {
    switch (values[i].type) {
    case Integer:  printf("%d", values[i].val.ival); break;
    case Real:     printf("%g", values[i].val.fval); break;
    case Character:printf("%c", values[i].val.cval); break;
    default:       assert(0);
    }
}
```


Bit Fields

- Signed and unsigned integers can be ***packed*** into ***bit fields***

```
enum Type { Integer=1, Real=2, Character=3 };

struct value {
    int type : 3;
    unsigned printed : 1;
    union u val;
} values[100];

void print(int i) {
    if (!values[i].printed) {
        switch (values[i].type) {
        ...
        }
        values[i].printed = 1
    }
}
```

- Extracting `int` bit fields ***sign extends*** the leftmost bit of the field
- ***Unnamed*** fields help lay out fields to access specific parts of a word

```
struct instruction { unsigned op:2; :5; unsigned op2:3;
                     int immed:22; };
```

Typedef

- `typedef` ***associates*** a ***name*** with a ***type*****, why?**
- Standard declaration; the “variable” is a new type

```
typedef short int16;

typedef struct {
    char *keyword;
    int keycount;
} key;

typedef enum { Integer, Real, Character } Type;


int16 max(int16 x, int16 y);

key keytable[NKEYS];

(key *)p

sizeof (key)
```

parentheses are required!