Structures

Structures are heterogeneous collections of variables.

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

- A structure declaration 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 };  // the old way
  ```

Structures can be nested:

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

- Structure fields are accessed by variable.

```
struct person employee, employees[100];
employee.birthday.month;
employees[i].name[j];
```

- A structure pointer points to instances of structures:
  ```
  struct date d, *pd;
  pd = &d;
  d = *pd;
  ```

- Structure assignment is legal:
  ```
  fpd = d;
  fpd = pd;
  ```

- Structure can contain pointers:
  ```
  struct binode { struct tree* l, *r; } *p;
  ```

- Structure field assignment is illegal:
  ```
  fpd = p;
  ```

Arrays of Structures

- Arrays of structures is the preferred method for storing a table:
  ```
  #define NKEYS 100
  struct key { char *keyword[NKEYS];
               char *keyword;
               int keycount[NKEYS];
               int keycount;
  } keytab[NKEYS];
  ```

- It is easy to initialize such tables:
  ```
  struct key keytable[] = {
      "auto", 0,
      "break", 0,
      ...
      "while", 0
  };
  ```

- It is easy to search them:
  ```
  int i;
  for (i = 0; i < NKEYS; i++)
      if (strcmp(word, keytable[i].keyword) == 0)
          ...
  ```

- Structures can contain pointers to structures:
  ```
  struct tree { p->l->l->l->d.month;
  } *p;
  ```

- Structure fields are accessed by variable.

Fields

- A field in a structure pointed by a pointer:
  ```
  struct person *p;
  p->birthday.month;
  ```

- Arrays of structures are the preferred method for storing a table:
  ```
  struct key keytable[] = {
      "auto", 0,
      "break", 0,
      ...
      "while", 0
  };
  ```

- It is easy to initialize such tables:
  ```
  struct key keytable[] = {
      "auto", 0,
      "break", 0,
      ...
      "while", 0
  };
  ```

- It is easy to search them:
  ```
  int i;
  for (i = 0; i < NKEYS; i++)
      if (strcmp(word, keytable[i].keyword) == 0)
          ...
  ```

- Structures can contain pointers to structures:
  ```
  struct tree { p->l->l->l->d.month;
  } *p;
  ```

- Structure fields are accessed by variable.
sizeof

- sizeof is a compile-time operator that gives the size of x in bytes. x can be (type) or expression.

```c
sizeof (int) 4
sizeof (int *) 4
sizeof (struct key *) 4
sizeof (struct key) 8
sizeof keytable NKEYS * sizeof (struct key) 32
```

- Use sizeof to define parameters
  ```c
  #define NKEYS (sizeof keytable / sizeof (struct key))
  ```

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

- Union size is equal to the sizeof the largest field:
  ```c
  sizeof uval 8
  ```

- No validity checks.

- Bit fields
  ```c
  enum Type { Integer=1, Real=2, Character=3 };
  struct value {
      int type : 3;
      unsigned printed : 1;
      union u val;
  } values[100];
  ```

- Extracting bit fields sign extend the leftmost bit of the field.

- Unions often appear in structures to reduce space.

```c
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 associates a name with a type why?