Data Structures

CS 217

Structures

- A heterogeneous collection of variables

```
struct date {
   int day;           declares date;
   char month[4];     does not allocate space
   int year;
};
```

- Can be used to define variables

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

- Structure declaration + variable definition

```
struct date {. . .} birthday;
```

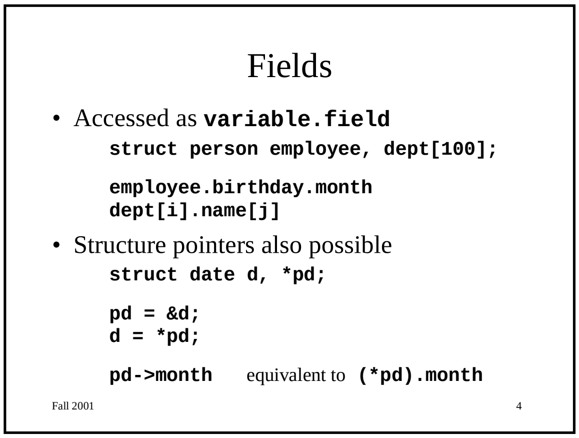
Structures (cont)

- Structures can be initialized

```
struct date today = {4, “Sep”, 2001};
```

- Structures can be nested

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



Fields

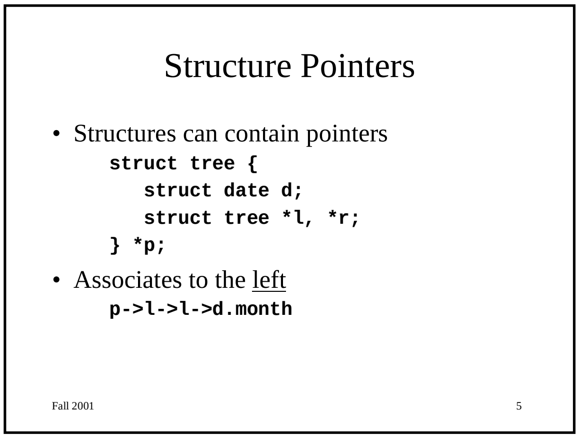
- Accessed as **`variable.field`**

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

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

- Structure pointers also possible

```
struct date d, *pd;

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

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



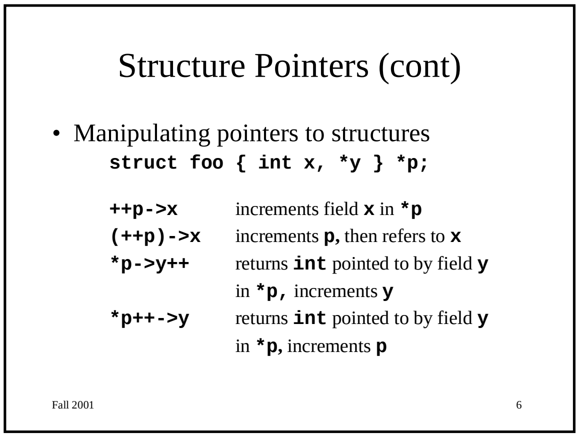
Structure Pointers

- Structures can contain pointers

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

- Associates to the left

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

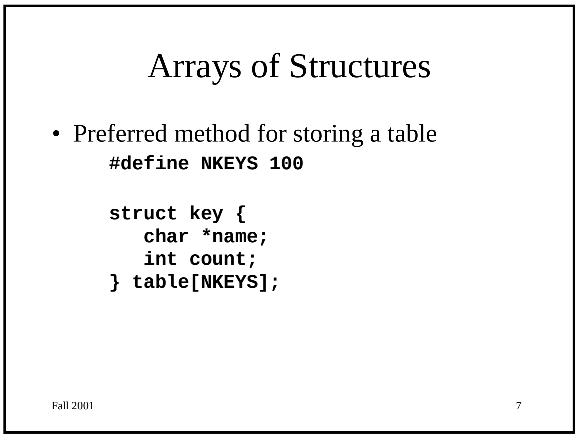


Structure Pointers (cont)

- Manipulating pointers to structures

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

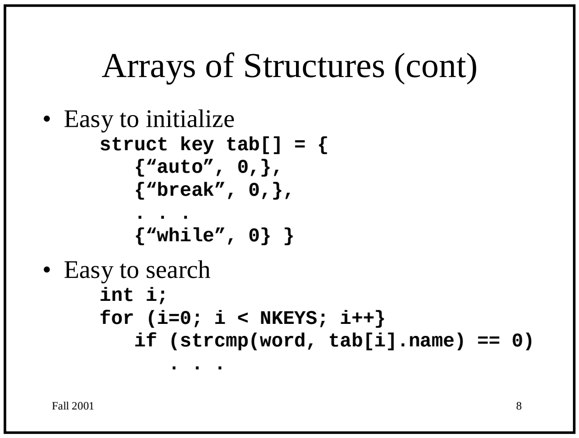
`++p->x`	increments field **`x`** in **`*p`**
`(++p)->x`	increments **`p`**, then refers to **`x`**
`*p->y++`	returns **`int`** pointed to by field **`y`** in **`*p`**, increments **`y`**
`*p++->y`	returns **`int`** pointed to by field **`y`** in **`*p`**, increments **`p`**



Arrays of Structures

- Preferred method for storing a table

```
#define NKEYS 100

struct key {
   char *name;
   int count;
} table[NKEYS];
```



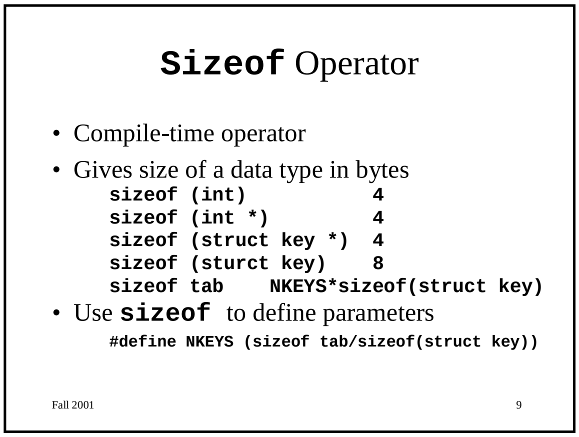
Arrays of Structures (cont)

- Easy to initialize

```
struct key tab[] = {
   {“auto”, 0,},
   {“break”, 0,},
   . . .
   {“while”, 0} }
```

- Easy to search

```
int i;
for (i=0; i < NKEYS; i++}
   if (strcmp(word, tab[i].name) == 0)
      . . .
```



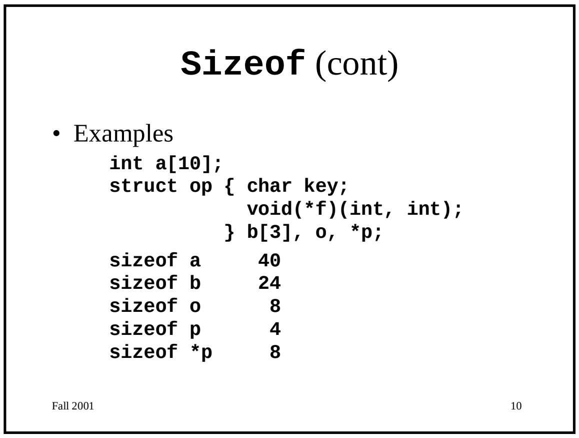
`Sizeof` Operator

- Compile-time operator
- Gives size of a data type in bytes

```
sizeof (int)           4
sizeof (int *)         4
sizeof (struct key *)  4
sizeof (sturct key)    8
sizeof tab    NKEYS*sizeof(struct key)
```

- Use `sizeof` to define parameters

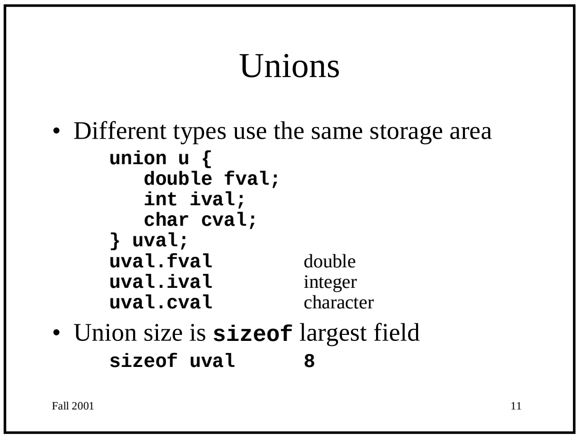
```
#define NKEYS (sizeof tab/sizeof(struct key))
```



`Sizeof` (cont)

- Examples

```
int a[10];
struct op { 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
```



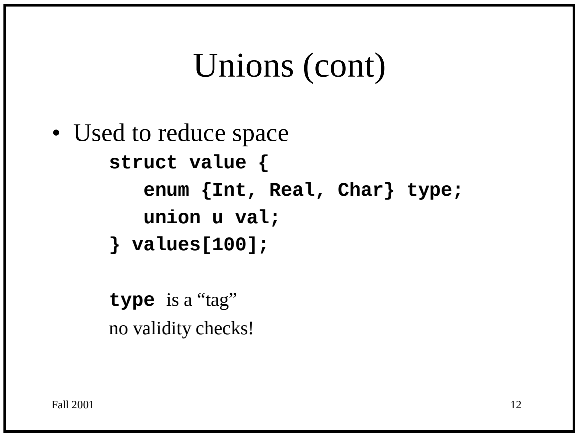
Unions

- Different types use the same storage area

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

- Union size is **`sizeof`** largest field

```
sizeof uval      8
```



Unions (cont)

- Used to reduce space

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

`type` is a “tag”

no validity checks!

Unions (cont)

- Check tag before accessing union fields

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

Bit Fields

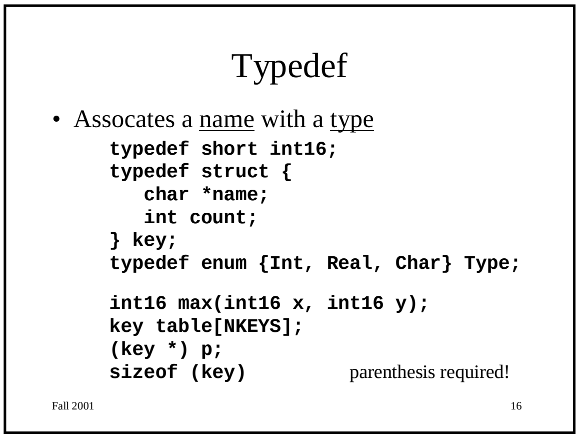
- Integers can be packed into bit fields

```
enum Type {Int=1, Real=2, Char=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;
   }
}
```

Bit Fields (cont)

- Both signed and unsigned integers
  - extracting sign extends the leftmost bit
- Unnamed fields help lay out the fields
  - used to access specific parts of the word

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

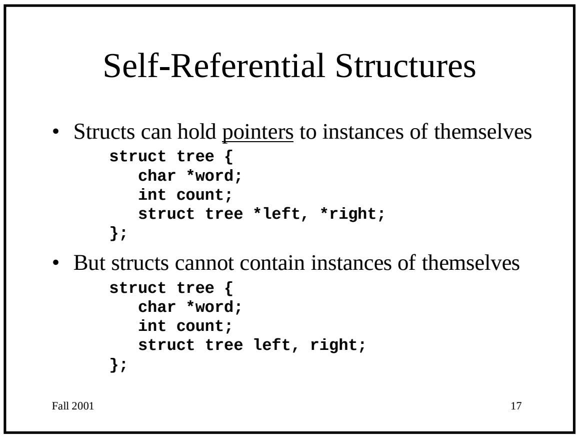


Typedef

- Assocates a name with a type

```
typedef short int16;
typedef struct {
   char *name;
   int count;
} key;
typedef enum {Int, Real, Char} Type;

int16 max(int16 x, int16 y);
key table[NKEYS];
(key *) p;
sizeof (key)
```

parenthesis required!



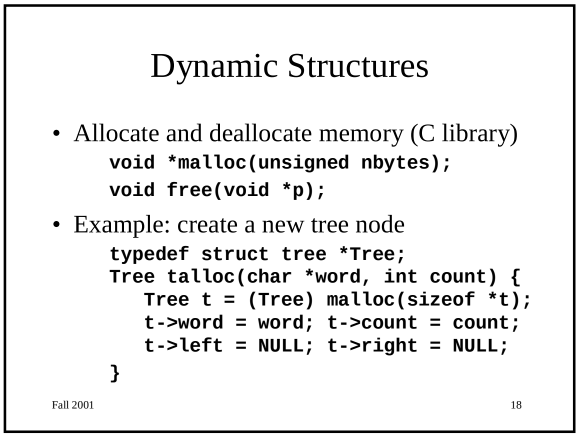
Self-Referential Structures

- Structs can hold pointers to instances of themselves

```
struct tree {
   char *word;
   int count;
   struct tree *left, *right;
};
```

- But structs cannot contain instances of themselves

```
struct tree {
   char *word;
   int count;
   struct tree left, right;
};
```



Dynamic Structures

- Allocate and deallocate memory (C library)

```
void *malloc(unsigned nbytes);
void free(void *p);
```

- Example: create a new tree node

```
typedef struct tree *Tree;
Tree talloc(char *word, int count) {
   Tree t = (Tree) malloc(sizeof *t);
   t->word = word; t->count = count;
   t->left = NULL; t->right = NULL;
}
```

Dynamic Structures (cont)

- Other allocation functions

  `void *calloc(unsigned n, unsigned nbytes)`

  allocates and clears **`n`** copies of **`nbytes`**

  `void *realloc(void *p, unsigned size)`

  expands/shrinks memory pointed at by **`p`** to **`size`** bytes; may relocate
- All allocation functions return **`NULL`** if there is no memory available

Example: Binary Tree

```
void insert(Tree *p, char *word) {
   Tree q = *p;
   if (q) {
      int cond = strcmp(word, q->word);
      if (cond < 0)
         insert(&q->left, word);
      else if (cond > 0)
         insert(&q->right, word);
      else
         q->count++;
   } else
      *p = talloc(strsave(word), 1);
}
```

Binary Tree (cont)

```
char *strsave(char *s) {
   char *new = malloc(strlen(s) + 1);
   assert(new);
   return strcpy(new, s);
}
```