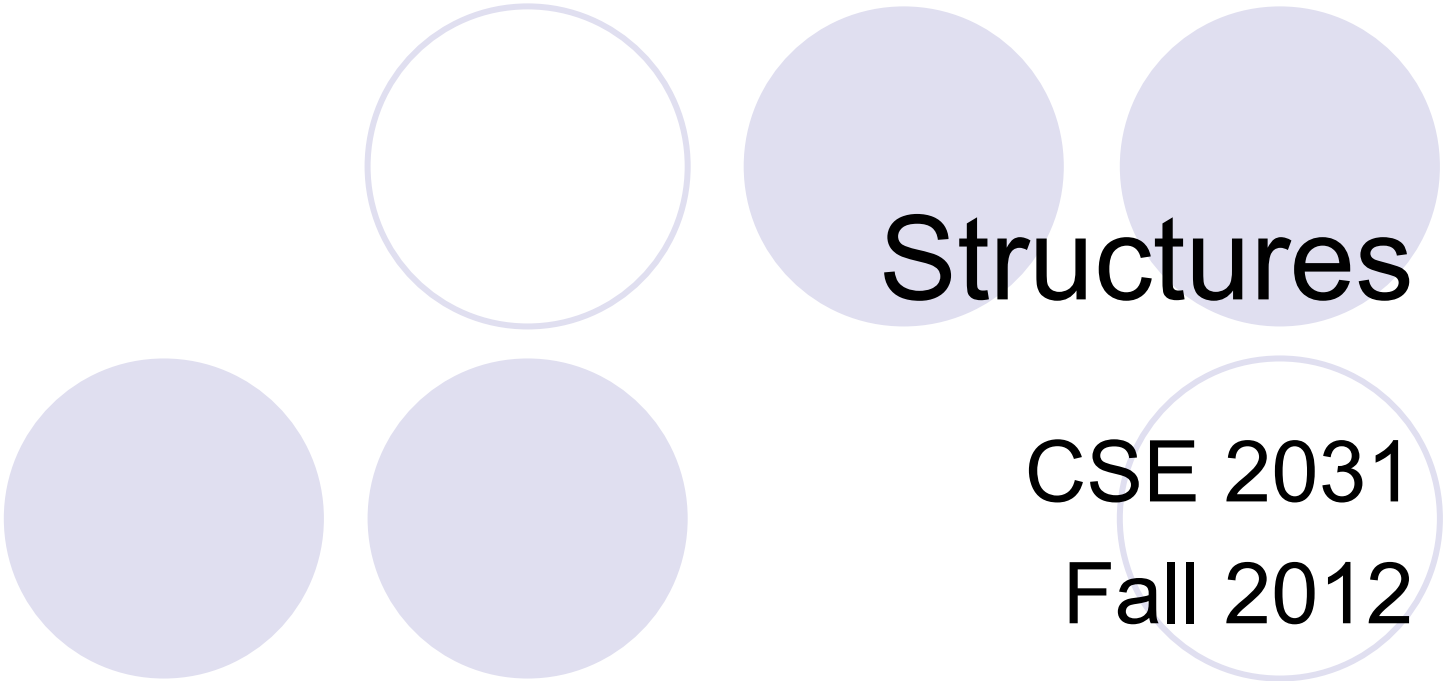


Structures

CSE 2031

Fall 2012

Basics of Structures (6.1)

```
struct point {
   int x;
   int y;
};
```

keyword **`struct`** introduces a structure declaration.

`point`: *structure tag*

`x, y`*: members*

The same member names may occur in different structures.

- Now **`struct point`** is a valid type.
- Defining `struct` variables:

```
struct point pt;
struct point
   maxpt = {320, 200};
```

- A **`struct`** declaration defines a type.

  `struct { ... } x, y, z;`

  or **`struct point x,y,z;`**

  is syntactically analogous to

  `int x, y, z;`

Using Structures

- Members are accessed using operator “.”

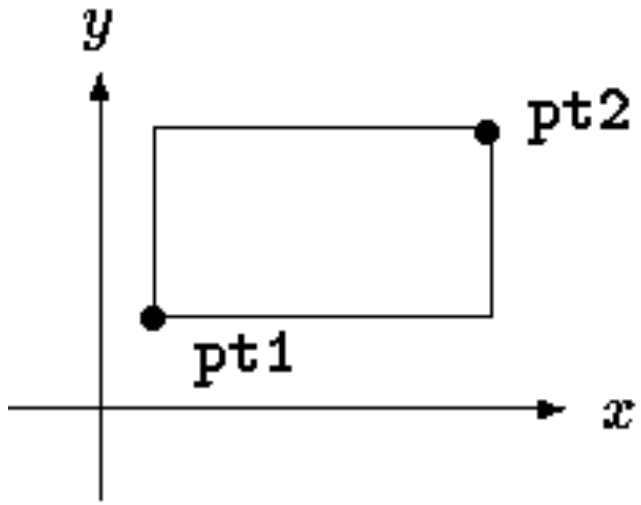
```
structure-name.member
printf(“%d,%d", pt.x, pt.y);
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x +
          (double)pt.y * pt.y);
```

- Structures cannot be assigned.

```
struct point pt1, pt2;
pt1.x = 0; pt1.y = 0;
pt2 = pt1;   /* WRONG !!! */
```

Nested Structures

```
struct rect {
  struct point pt1;
  struct point pt2;
};
struct rect screen;
screen.pt1.x = 1;
screen.pt1.y = 2;
screen.pt2.x = 8;
screen.pt2.y = 7;
```



Structures and Functions (6.2)

- Returning a structure from a function.

```
/* makepoint: make a point from x and y components */
struct point makepoint(int x, int y) {
   struct point temp;
   temp.x = x;
   temp.y = y;
   return temp;
}
struct rect screen;
struct point middle;
struct point makepoint(int, int);
screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
        (screen.pt1.y + screen.pt2.y)/2);
```

Structures and Functions (cont.)

- Passing structure arguments to functions: structure parameters are passed by values like int, char, float, etc. (a copy of the structure is sent to the function).

```
/* addpoints: add two points */
struct point addpoint(struct point p1, struct point p2)
{
   p1.x += p2.x;
   p1.y += p2.y;
   return p1;
}
```

- Note: the components in p1 are incremented rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others (no changes to original struct).

Pointers to Structures

- 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.

```
struct point *pp;
struct point origin;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

- Note: *pp.x means *(pp.x), which is illegal (why?)

Pointers to Structures: Example

```
/* addpoints: add two points */
struct point addpoint (struct point *p1, struct point *p2)
{
   struct point temp;
    temp.x = (*p1).x + (*p2).x;
    temp.y = (*p1).y + (*p2).y;
    return temp;
}

main() {
    struct point a, b, c;
    /* Input or initialize structures a and b */
    c = addpoint( &a, &b );
}
```

Pointers to Structures: Shorthand

- `(*pp).x` can be written as `pp->x`

```
printf("origin is (%d,%d)\n", pp->x, pp->y);

struct rect r, *rp = &r;
r.pt1.x = 1;
rp->pt1.x = 2;
(r.pt1).x = 3;
(rp->pt1).x = 4;
```

- Note: Both . and -> associate from left to right.

Arrays of Structures (6.3)

```
struct dimension {
   float width;
   float height;
};
struct dimension chairs[2];
struct dimension *tables;
tables = (struct dimension*) malloc
 (20 * sizeoff(struct dimension));
```

Initializing Structures

```
struct dimension sofa = {2.0, 3.0};

struct dimension chairs[] = {
                        {1.4, 2.0},
                        {0.3, 1.0},
                        {2.3, 2.0} };
```

Arrays of Structures: Example

```
struct key {
   char *word;
   int count;
};

struct key keytab[NKEYS];
struct key *p;
for (p = keytab;
     p < keytab + NKEYS; p++)
   printf("%4d %s\n",
       p->count, p->word);
```

```
struct key {
   char *word;
   int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
    /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};
```

Pointers to Structures (6.4)

```
struct key keytab[NKEYS];
struct key *p;
for (p = keytab; p < keytab + NKEYS; p++)
    printf("%4d %s\n", p->count, p->word);
```

- p++ increments p by the correct amount (i.e., structure size) to get the next element of the array of structures.

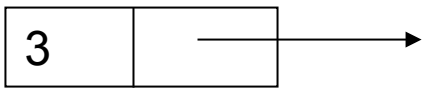
```
struct {
    char c;  /* one byte */
    int i;   /* four bytes */
};
```

- What is the total structure size?
- Use the **`sizeof`** operator to get the correct structure size.

Self-referential Structures (6.5)

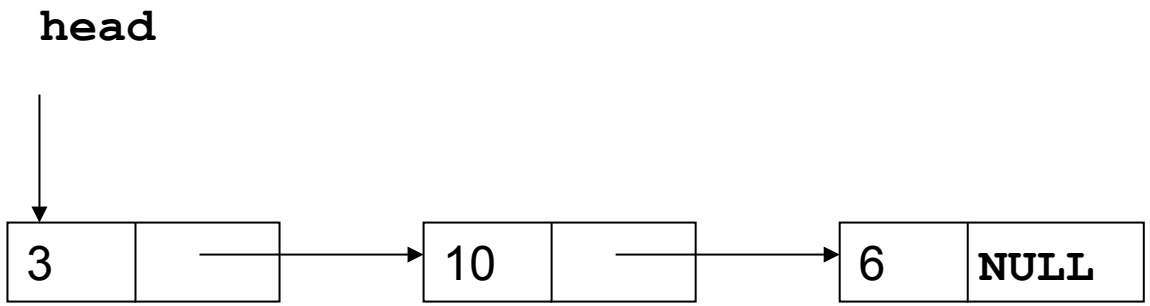
Example: (singly) linked list

```
struct list  {
     int data;
     struct list *next;
 };
```



Linked List

- Pointer **`head`** points to the first element
- Last element pointer is **`NULL`**
- Example (next slide): build a linked list with **`data`** being non-negative integers, then search for a number.
  - Insertion at the end (rear) of the list.
- We also learn how to dynamically allocate a structure.



Linked List Implementation

```
#include <stdio.h>
#include <stdlib.h>
main() {
    struct list {
            int data;
            struct list *next;
    } *head, *p, *last;
    int i;

    /* Create a dummy node, which
       simplifies insertion and deletion */
    head = (struct list *) malloc
            ( sizeof( struct list ) );
    head—>data = -1;
    head—>next = NULL;
    last = head;
    scanf( "%d", &i );  /* input 1st element */
    while( i >= 0 ) {
            p = (struct list  *)
                    malloc( sizeof( struct list ) );
            p—>data = i;
            p—>next = NULL;
            last—>next = p;
            last = p;
            scanf( "%d", &i );
    } /* while */

    printf("Enter the number to search for  ");
    scanf( "%d", &i );
    for( p = head;  p != NULL; p = p—>next )
            if( p—>data == i )
                printf( "FOUND %d \n", i );
} /* main */
```

`typedef` (6.7)

- For creating new data type names

```
typedef int Length;
Length len, maxlen;
Length *lengths[];

typedef char *String;
String p, lineptr[MAXLINES];
p = (String) malloc(100);
int strcmp(String, String);
```

`typedef` with `struct`

- We can define a new type and use it later

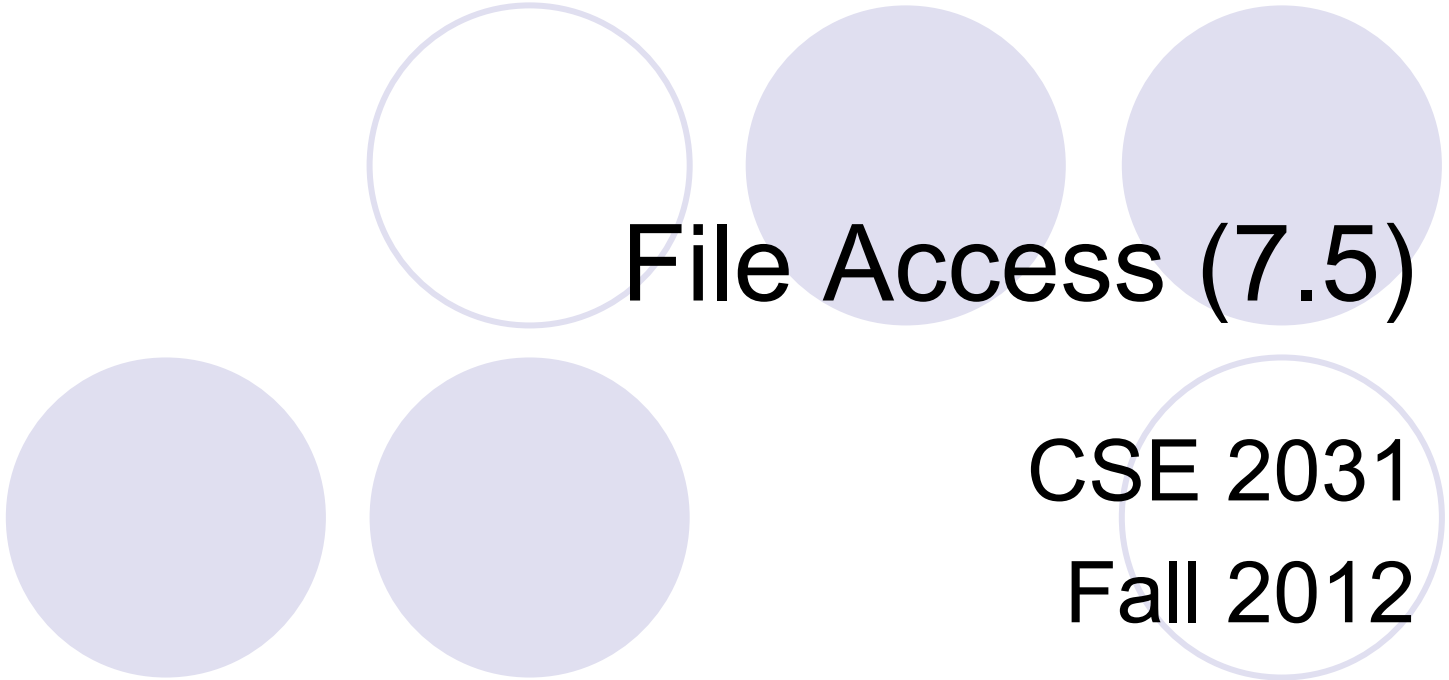
```
typedef struct {
        int x,y;
        float z;
    } mynewtype;
 mynewtype a, b, c, x;
```

- Now, `mynewtype` is a type in C just like `int` or `float`.

Self-referential Structures: More Examples

- Binary trees (6.5)
- Hash tables (6.6)

To be covered later if time permits.



File Access (7.5)

CSE 2031

Fall 2012

Declaring and Opening Files

```
FILE *fp; /* file pointer */
FILE *fopen(char *name, char *mode);
```

Example:

```
FILE *ifp, *ofp;
char iname[50], oname[50];
scanf( “%s %s”, iname, oname );
ifp = fopen( iname, "r" );
if ( ifp == NULL ) { ... }
ofp = fopen( oname, "w" );
if ( ofp == NULL ) { ... }
```

Modes

```
fp = fopen( name, "r" );
```

- Returns NULL if file does not exist, or has no read permission.

```
fp = fopen( name, "w" );
```

- If file does not exist, one will be created for writing.
- If file already exists, the content will be erased when the file is opened. So be careful!
- Returns NULL if file has no write permission.

Modes (cont.)

```
fp = fopen( name, "a" );    /* append */
```

- If file does not exist, one will be created for writing.
- If file already exists, the content will be preserved.
- Returns NULL if file has no write permission.

- May combine multiple modes.

```
fp = fopen( name, "rw" );
```

File may be read first, but the old content will be erased as soon as something is written to the file.

```
fp = fopen( name, "ra" );
fp = fopen( name, "aw" );   /* same as "a" */
```

Reading and Writing Files

```
int getc( FILE *fp )
int putc( int c, FILE *fp )
int fscanf( FILE *fp, char *format, ... )
int fprintf( FILE *fp, char *format, ... )

int c;
while ( (c = getc( ifp )) != EOF )
   putc( c, ofp );

char ch;
while ( fscanf(  ifp, "%c", &ch ) != EOF )
   fprintf(  ofp, "%c", ch );
```

Closing Files

```
int fclose( FILE *fp )
```

```
fclose( ifp );
fclose( ofp );
```

- Most operating systems have some limit on the number of files that a program may have open simultaneously ⇒ free the file pointers when they are no longer needed.
- **`fclose`** is called automatically for each open file when a program terminates normally.
- For output files: **`fclose`** flushes the buffer in which **`putc`** is collecting output.

Reminder: I/O Redirection

- In many cases, I/O redirection is simpler than using file pointers.

```
a.out < input_file > output_file
```

```
a.out < input_file >> output_file
```

Reminders

- Midterm (next week)
- Lab test 1 (Oct. 26 and 29)
- Next lecture: Unix