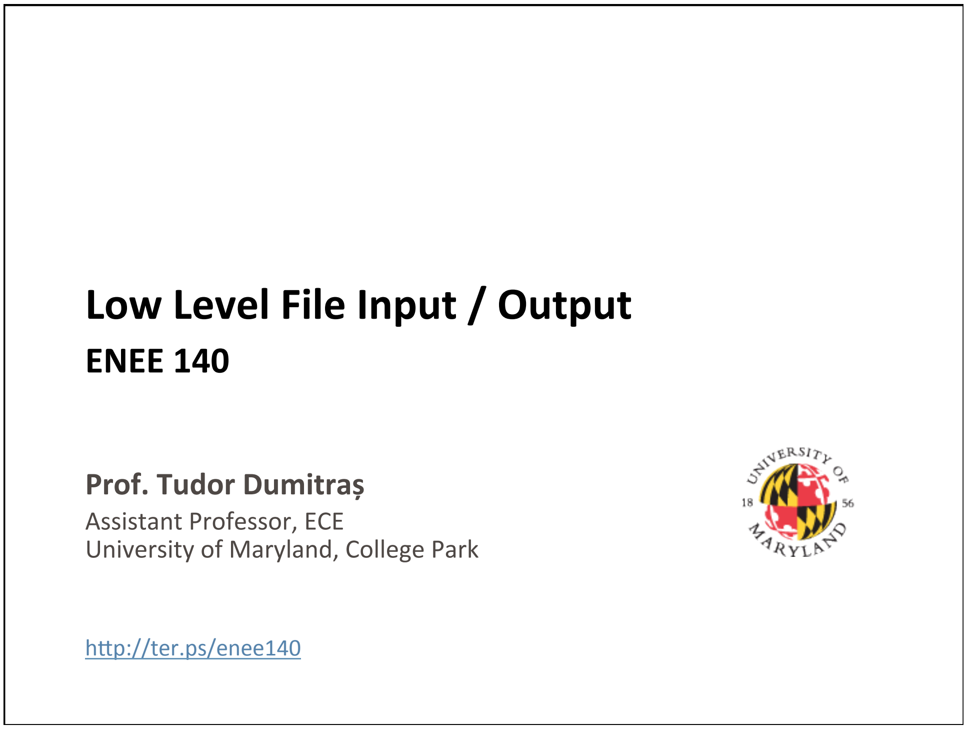
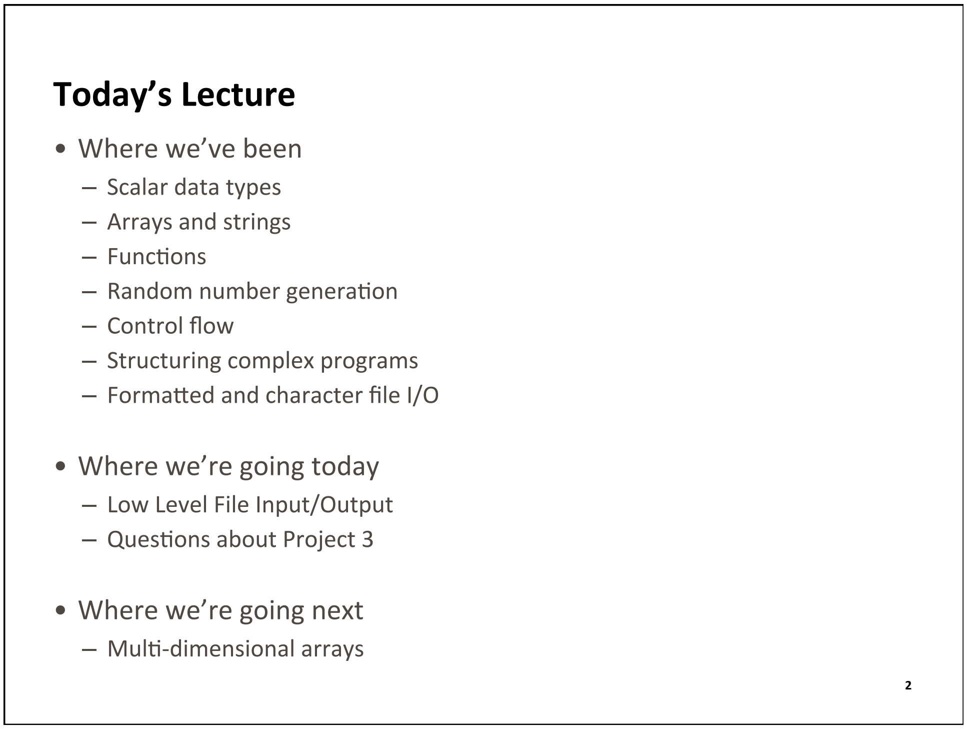


Low Level File Input / Output

ENEE 140

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

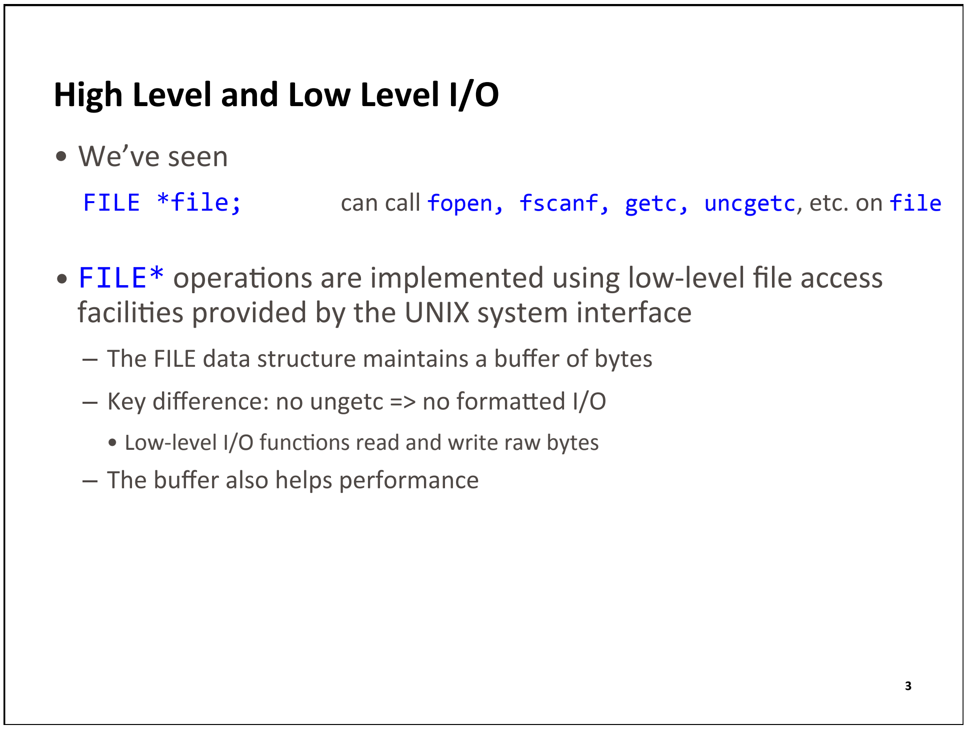
http://ter.ps/enee140



Today's Lecture

- Where we've been
  - Scalar data types
  - Arrays and strings
  - Functions
  - Random number generation
  - Control flow
  - Structuring complex programs
  - Formatted and character file I/O
- Where we're going today
  - Low Level File Input/Output
  - Questions about Project 3
- Where we're going next
  - Multi-dimensional arrays

2

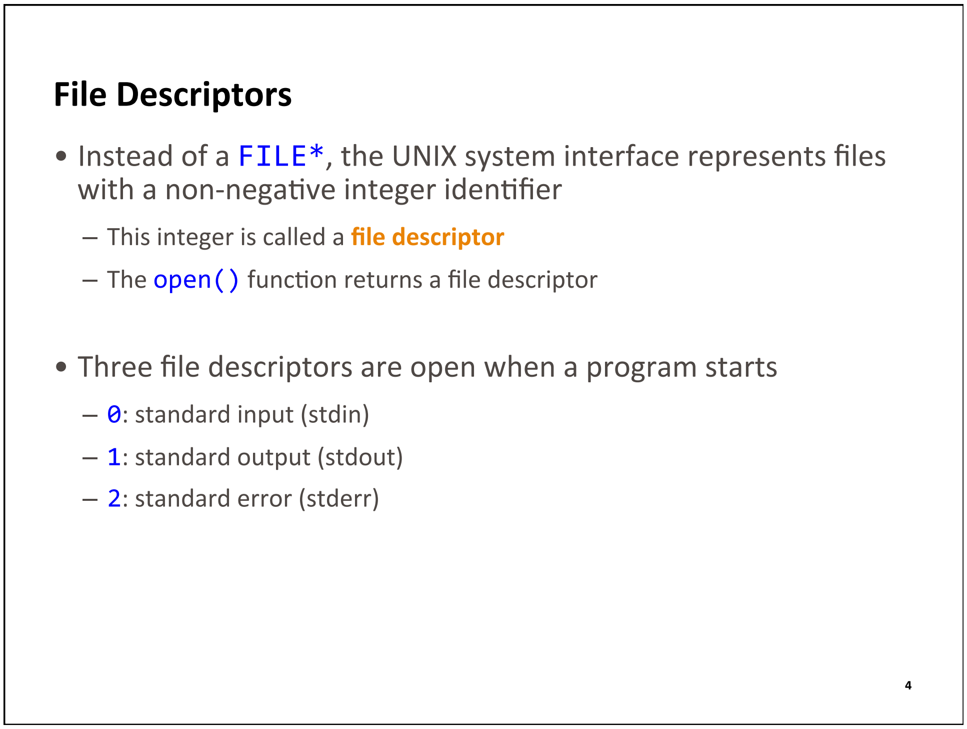


High Level and Low Level I/O

- We’ve seen

  `FILE *file;` can call `fopen, fscanf, getc, uncgetc`, etc. on `file`

- `FILE*` operations are implemented using low-level file access facilities provided by the UNIX system interface
  - The FILE data structure maintains a buffer of bytes
  - Key difference: no ungetc => no formatted I/O
    - Low-level I/O functions read and write raw bytes
  - The buffer also helps performance



File Descriptors

- Instead of a `FILE*`, the UNIX system interface represents files with a non-negative integer identifier
  - This integer is called a **file descriptor**
  - The `open()` function returns a file descriptor

- Three file descriptors are open when a program starts
  - `0`: standard input (stdin)
  - `1`: standard output (stdout)
  - `2`: standard error (stderr)

Low Level File I/O

- Functions for low-level file I/O manipulate file descriptors

```
#include <fcntl.h>
#include <unistd.h>

char buffer[N];                                data buffer
int fd1 = open(“file1.txt”, O_RDONLY);         open fd1 for reading
int n_read =                                   returns num. bytes read
     read(fd1, buffer, sizeof(buffer));        read up to N bytes into buffer

int fd2 = open(“file2.txt”, O_WRONLY);         open fd2 for writing
int n_written =                                returns num. bytes written
     write(fd2, buffer, sizeof(buffer));       write up to N bytes from buffer
```

Some Functions for Low-Level I/O

`int open(const char *pathname, int flags, mode_t mode);`

- Opens a file and returns a file descriptor
- `flags` must include one of `O_RDONLY`, `O_WRONLY`, or `O_RDWR`
- `flags` may also be bitwise-or'd with `O_APPEND` (write after end of file), `O_TRUNC` (if file exists, discard current data), `O_CREAT` (create the file if it doesn't exist), and a few others (full list in man page)
- `mode` must be provided with `O_CREAT` and specifies the file permissions (e.g. `0600` for giving RW permissions to the file owner)

`int creat(const char *pathname, mode_t mode);`

- Equivalent to `open()` with `O_CREAT|O_WRONLY|O_TRUNC` for flags

`FILE *fdopen(int fd, const char *mode);`

- Associates a `FILE*` stream to an existing file descriptor

`int unlink(const char *pathname);`

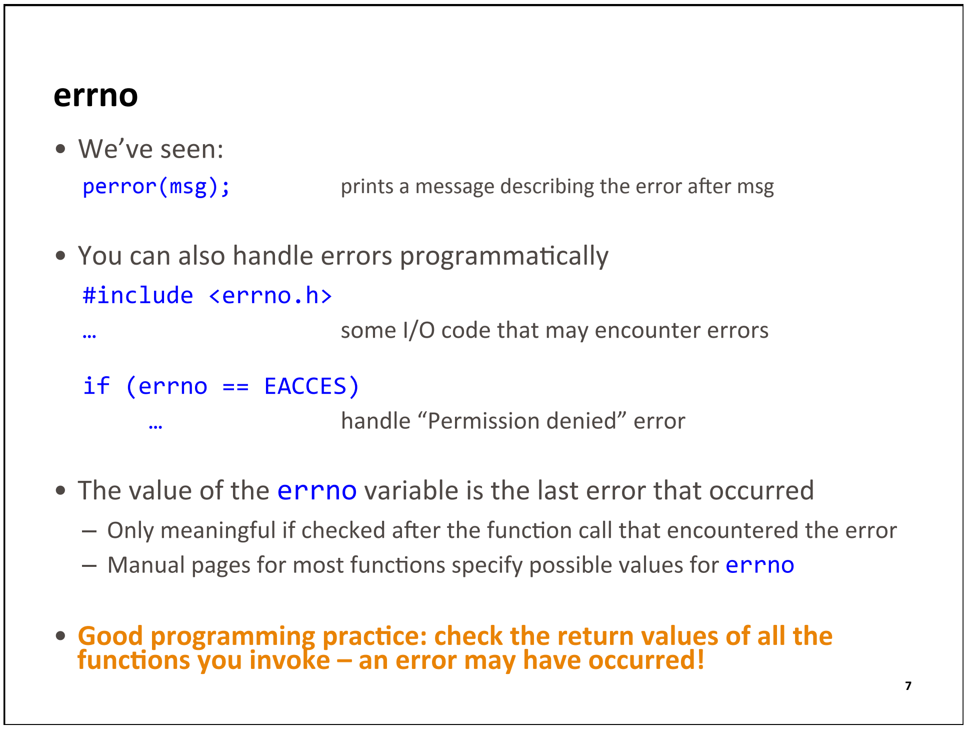
- Deletes a file from the filesystem

`off_t lseek(int fd, off_t offset, int whence);`

- Changes position in file

`int close(int fd);`

- Closes the file associated with fd



errno

- We've seen:

```
perror(msg);            prints a message describing the error after msg
```

- You can also handle errors programmatically

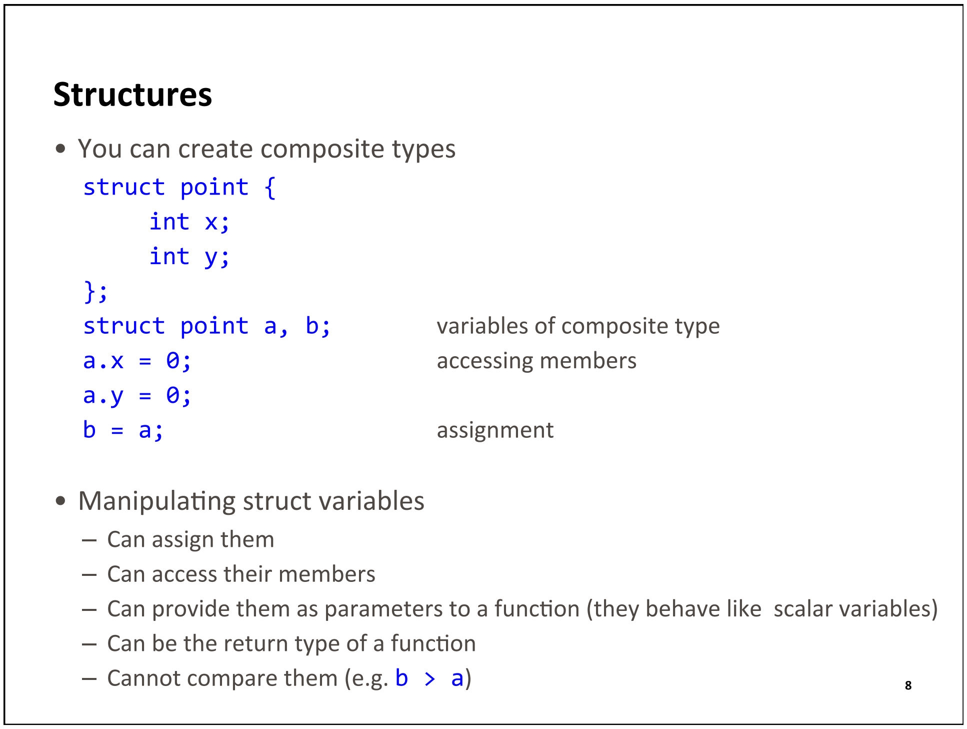
```
#include <errno.h>
…                       some I/O code that may encounter errors

if (errno == EACCES)
      …                 handle "Permission denied" error
```

- The value of the errno variable is the last error that occurred
  - Only meaningful if checked after the function call that encountered the error
  - Manual pages for most functions specify possible values for errno

- **Good programming practice: check the return values of all the functions you invoke – an error may have occurred!**

7



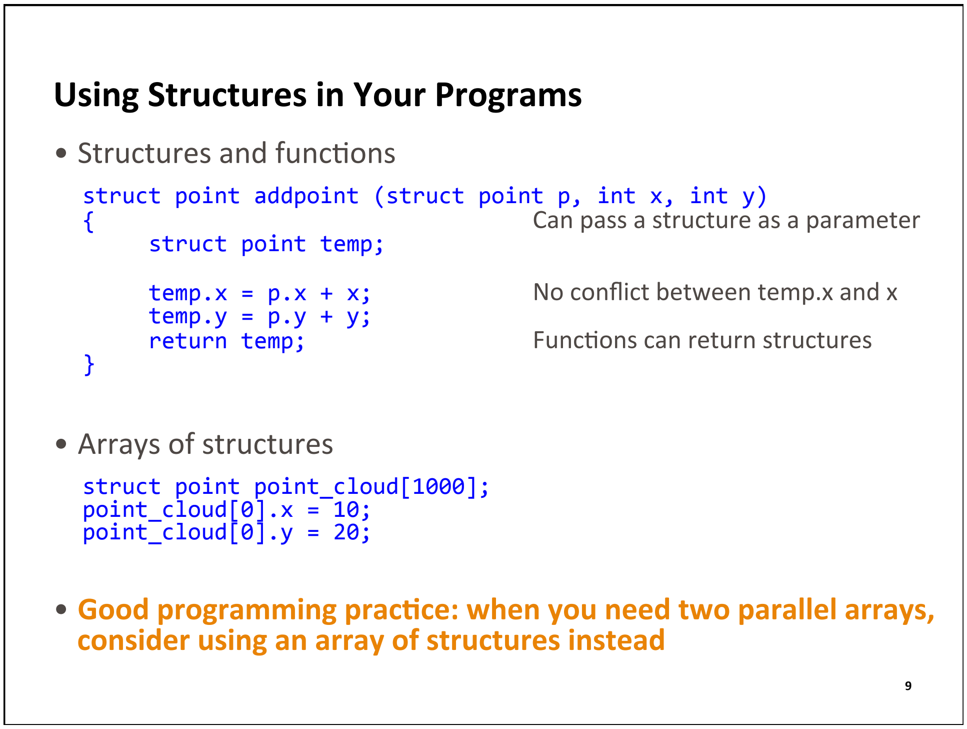
Structures

- You can create composite types

```
struct point {
      int x;
      int y;
};
struct point a, b;          variables of composite type
a.x = 0;                    accessing members
a.y = 0;
b = a;                      assignment
```

- Manipulating struct variables
  - Can assign them
  - Can access their members
  - Can provide them as parameters to a function (they behave like scalar variables)
  - Can be the return type of a function
  - Cannot compare them (e.g. b > a)

8



Using Structures in Your Programs

- Structures and functions

```
struct point addpoint (struct point p, int x, int y)
{
     struct point temp;

     temp.x = p.x + x;
     temp.y = p.y + y;
     return temp;
}
```

Can pass a structure as a parameter

No conflict between temp.x and x

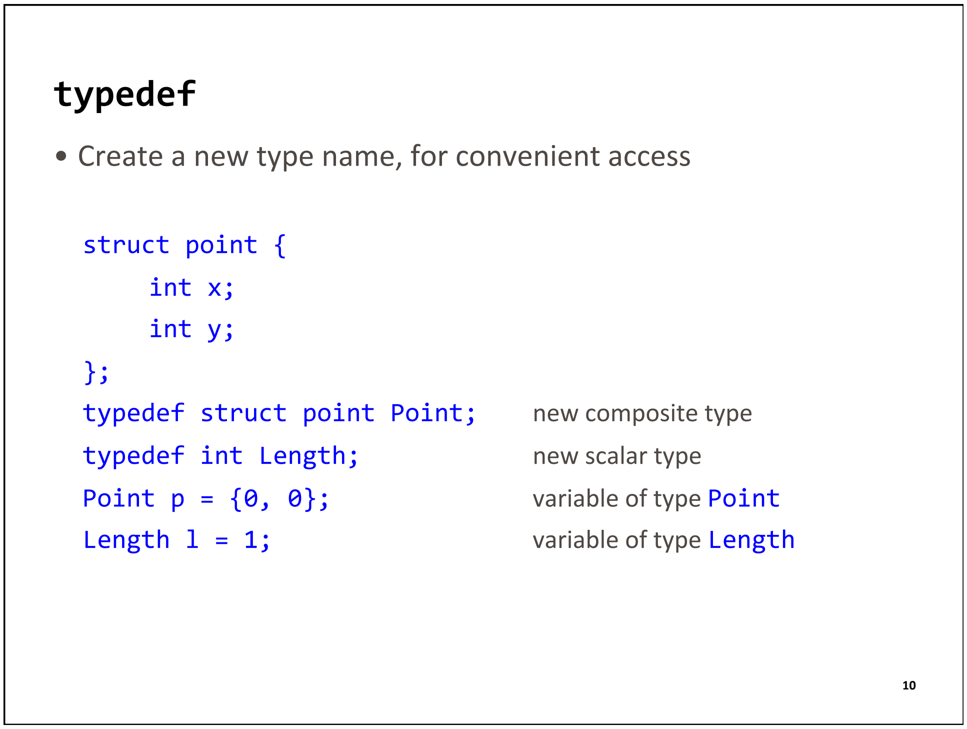
Functions can return structures

- Arrays of structures

```
struct point point_cloud[1000];
point_cloud[0].x = 10;
point_cloud[0].y = 20;
```

- **Good programming practice: when you need two parallel arrays, consider using an array of structures instead**

9



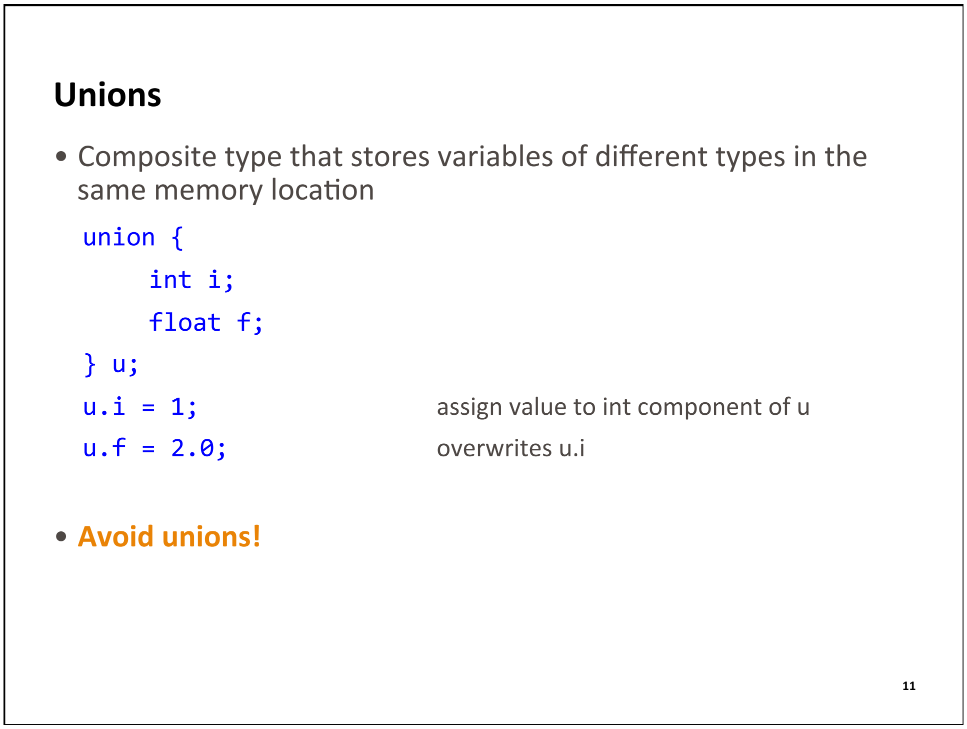
typedef

- Create a new type name, for convenient access

```
struct point {
     int x;
     int y;
};
typedef struct point Point;
typedef int Length;
Point p = {0, 0};
Length l = 1;
```

`typedef struct point Point;`	new composite type
`typedef int Length;`	new scalar type
`Point p = {0, 0};`	variable of type `Point`
`Length l = 1;`	variable of type `Length`

10

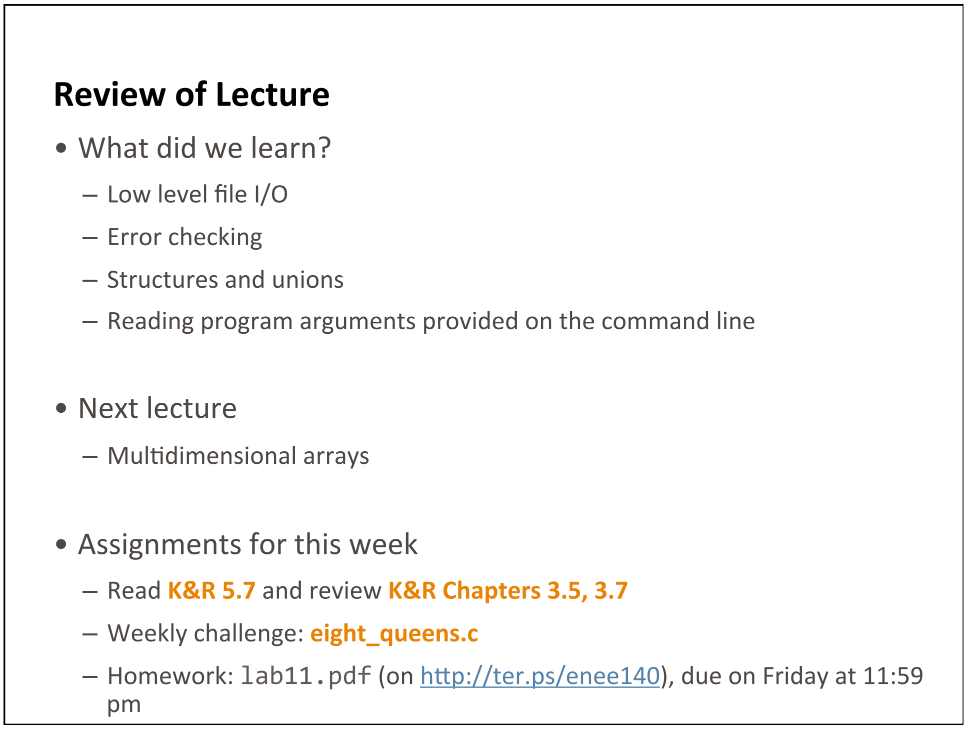


Unions

- Composite type that stores variables of different types in the same memory location

```
union {
     int i;
     float f;
} u;
u.i = 1;                assign value to int component of u
u.f = 2.0;              overwrites u.i
```

- **Avoid unions!**



Review of Lecture

- What did we learn?
  - Low level file I/O
  - Error checking
  - Structures and unions
  - Reading program arguments provided on the command line

- Next lecture
  - Multidimensional arrays

- Assignments for this week
  - Read **K&R 5.7** and review **K&R Chapters 3.5, 3.7**
  - Weekly challenge: **eight_queens.c**
  - Homework: `lab11.pdf` (on http://ter.ps/enee140), due on Friday at 11:59 pm