Structs

- A collection of related data items

```
struct record {
   char name[MAXNAME];
   int count;
};
```

/* The semicolon is important! It terminates the declaration. */

```
struct record rec1;  /*allocates space for the record */
strncpy(rec1.name, ".exe", MAXNAME);
struct record *rec2 = malloc(sizeof(*rec2));
strncpy(rec2->name, ".gif", MAXNAME);
```

structs as arguments

```
/* Remember: pass-by-value */
void print_record(struct record r) {
   printf("Name = %s\n", r.name);
   printf("Count=%d\n", r.count);
}
print_record(rec1);
print_record(*rec2);
```

Passing pointer or struct?

```
/* Incorrect */
void incr_record(struct record r) {
   r.count++;
}
/* Correct */
void incr_record(struct record *r) {
   r->count++;
}
```

Concrete Example

```
int stat(const char *file_name,
              struct stat *buf);
```

```
struct stat {
    dev_t           st_dev;      /* device */
    ino_t           st_ino;      /* inode */
    mode_t          st_mode;     /* protection */
    nlink_t         st_nlink;    /* number of hard links */
    uid_t           st_uid;      /* user ID of owner */
    gid_t           st_gid;      /* group ID of owner */
    dev_t           st_rdev;     /* device type (if inode device) */
    off_t           st_size;     /* total size, in bytes */
    blksize_t       st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t        st_blocks;   /* number of blocks allocated */
    time_t          st_atime;    /* time of last access */
    time_t          st_mtime;    /* time of last modification */
    time_t          st_ctime;    /* time of last change */
};
```

stat

- By calling the `stat` function on a filename you want to fill in the fields of the `struct stat`.
- You must pass in a pointer, and there must be space allocated!!!

```
struct stat sbuf;
if(stat("myfile", &sbuf) == -1) {
   perror("stat");
   exit(1);
}
printf("Owner = %d", sbuf.st_uid);
```

File Interfaces in Unix

- Unix has two main mechanisms for managing file access.
- file pointers: standard I/O library (Ch. 11)
  - You deal with a pointer to a FILE structure that contains a file descriptor and a buffer.
  - Use for regular files (more abstract and portable)
- file descriptors: low-level (Ch. 2)
  - Each open file is identified by a small integer.
  - Use for pipes, sockets.

`stdin, stdout, stderr`

Ch 2.2

- 3 files are automatically opened for any executing program:

	`stdio` name	File descriptor
Standard input	`stdin`	0
Standard output	`stdout`	1
Standard error	`stderr`	2

- Reading from `stdin` by default comes from the keyboard
- Writing to `stdout` or `stderr` by default goes to the screen.

Buffering

- un-buffered – output appears immediately
  - `stderr` is not buffered
- line buffered – output appears when a full line has been written.
  - `stdout` is line buffered when going to the screen
- block buffered – output appears when a buffer is filled.
  - normally output to a file is block buffered
  - `stdout` is block buffered when redirected to a file.

File Operations

Ch 2.3

- For regular files use: fopen, fread, fwrite, fprintf, fgets, fscanf, fclose.

```
FILE *fopen(const char *filename, const char *mode);

char *fgets(char *s, int size, FILE *stream);
```

- reads the next line from a file pointer
  - It reads at most size -1 characters
  - Reading stops after a newline or EOF
  - Appends a ‘\0’ character at the end of the string.

Ch. 4.4

Files, Strings

- Problem: Given a name in the format “Last, First”, return a string in the format “First Last”

```
char *reverse_name(char *src) {
   char *dest;
   char *sptr = strchr(src, ',');
   ...
   return dest;
}
```

- We'll first do an example with pointers.

Using string functions

```
char *reverse_name(char *src) {
   int src_len = strlen(src), dest_len = 0;
   char *dest;
   char *sptr = strchr(src, ',');

   /* allocate space for return string */
   if ((dest = malloc(src_len+1)) == NULL) {
      return NULL;
   }
   /* Move past the comma and the spaces between
     the comma and the first name */
   sptr++;
   while(*sptr == ' ')
      sptr++;
```

```
    /* Copy the first name to dest */
    strncat(dest, sptr, strlen(src) + 1);

    /* Add a space to the destination string */
    dest_len = strlen(dest);
    dest[dest_len] = ' ';
    dest[dest_len + 1] = '\0';

    /* Copy the last name from src to dest */
    strncat(dest, src, src_len - dest_len - 1);
    dest[src_len-1] = '\0';
    return dest;
}
```

Spot the errors

Calling reverse_name

```
int main()
{
    char *sptr, name[MAX];

    while((sptr = fgets(name, MAX, stdin)) != NULL) {
        /* strip the newline */
        sptr = strchr(name, '\n');
        *sptr = '\0';
        printf("%s\n", reverse_name(name));
    }
    return 0;
}
```

Reading from standard input

Spot the errors

Reading from a file?

- If we want to read from somewhere other than `stdin`, we need to open a file.
- How should we specify the filename?
  - `argv[0]` == name of program
  - `argv[1]` == first argument

```
int main(int argc, char **argv) {
   if(argc != 2)
      fprintf(stderr, "Usage: %s <filename>\n",
             argv[0]);
      exit(1);
   }
}
```

stdio

- To open a file:

```
FILE *fopen(const char *filename,
            const char *mode);
```

  – `filename` identifies the file to open.
  – `mode` tells how to open the file:
    - "r" for reading, "w" for writing, "a" for appending
  – returns a pointer to a `FILE` struct which is the handle to the file. This pointer will be used in subsequent operations.
  – To close a file: `void fclose(FILE *stream);`

Example

```
int main(int argc, char **argv)
{
   char *sptr, name[MAX];
   FILE *fp;

   if(argc != 2) {
     fprintf(stderr, "Usage: do_reverse2 <file>\n");
     exit(1);
   }
   if((fp = fopen(argv[1], "r")) == NULL) {
     perror(argv[1]);
     exit(1);
   }
```

Example (cont’d)

```
    while((sptr = fgets(name, MAX, fp)) != NULL) {
        /* strip the newline */
        sptr = strchr(name, '\n');
        *sptr = '\0';
        printf("%s\n", reverse_name(name));
    }
    return 0;
}
```

Error Handling

- Most system calls return -1 if an error occurs. (A few return NULL.)
- **errno** – global variable that holds the numeric code of the last system call.
- Every process has **errno** assigned to zero at process creation time.
- When a system call error occurs, **errno** is set.
- A successful system call never affects the current value of **errno**.
- An unsuccessful system call always overwrites the current value of **errno**.
- **Always check the return value of system calls!**

`perror()`

- Library routine:
- `void perror( char *str )`
- perror displays `str`, then a colon(:), then an English description of the last system call error as defined in errno.h.
- Protocol
  - check system calls for a return value of -1
  - call perror() for an error description.

Binary I/O

- Recall that `fgets` reads characters.
- By contrast, `fread` and `fwrite` operate on bytes.

```
size_t  fread(void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  - read nmemb * `size` bytes into memory at `ptr`

```
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

  - write nmemb * `size` bytes from `ptr` to the file pointer `stream`

Example

- It doesn’t matter what the bytes contain!

```
/* write an integer to the file */
int num = 21;
n = fwrite(&num, sizeof(num), 1, fp);

/* write a struct to the file */
struct rec {
  string name[20];
  int num;
} r;
r.num = 42;
strncpy(r.name, "koala", 20);
n = fwrite(&r, sizeof(r), 1, fp);
```

Example

- We need to know how to interpret the bytes from a file when reading.

```
/* write an integer to the file */
int num;
n = fread(&num, sizeof(num), 1, fp);

/* write a struct to the file */
struct rec r;
n = fread(&r, sizeof(r), 1, fp);

/* display the contents of the variables */
printf("%d %s %d\n", num, r.name, r.num);
```

stat()

- `int stat(const char *file_name,`
  `struct stat *buf);`
- need to allocate memory for the `stat` `struct` before passing it to `stat`
- `struct stat` contains many fields including `st_mode`
- Useful macros: `S_ISREG(modefield)`, `S_ISDIR(modefield)`

stat()

```
struct stat sbuf;
if(stat(pathname, &sbuf) == -1) {
  perror(“stat”);
}
if(S_ISREG(sbuf.st_mode)) {
  printf(“Regular file\n”);
}
```

- There are also defined variables for each of the permission sets. For example:

```
if(sbuf.st_mode & S_IRUSR) {
  printf(“Owner can read file\n”);
}
```

Directory Operations

- Recall that a directory is a special kind of file.
- We can read directory entries using similar functions.
- For directories use:

```
DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
```

- `readdir` works like `fread` on directory files. Each time `readdir` is called it returns a directory entry.

Ch. 4.4

Example

```
char *name, line[LINESIZE], *lp; int len;
DIR *dp;  struct dirent *entry;   FILE *fp;
name = argv[1];
len = strlen(name);
dp = opendir(".");

for (entry = readdir(dp); entry != NULL;
     entry = readdir(dp))
    if ((strncmp(name, entry->d_name, len)) == 0) {
      fp = fopen(entry->d_name, "r");
      lp = fgets(line, LINESIZE, fp);
      fprintf(stdout, "%s: %s", entry->d_name, lp);
    }
closedir(dp);
```

What is wrong with this code?

Example

```
char *name, line[LINESIZE], *lp; int len;
DIR *dp;  struct dirent *entry;   FILE *fp;
name = argv[1];
len = strlen(name);
dp = opendir(".");

while( (entry = readdir(dp)) != NULL )
    if ((strncmp(name, entry->d_name, len)) == 0) {
      fp = fopen(entry->d_name, "r");
      lp = fgets(line, LINESIZE, fp);
      fprintf(stdout, "%s: %s", entry->d_name, lp);
    }
closedir(dp);
```

while loop equivalent
to the for loop