# Large C Program organization, I/O

Jinyang Li

# Organization of large C programs

- Breaking a large program into multiple files
  - *.h and *.c files
- C pre-processing

# Linked list: one big file

```
typedef struct {
     int val;
     struct node *next;
 }node;

node* insert(node *head, int val) {
    node *new_head = (node *)malloc(sizeof(node));
    new_head->next = head;
    new_head->val = val;
}

int main() {
    node *head = NULL;
    for (int i = 0; i < 3; i++)
        head = insert(head, i);
}
```

list.c

What if another program also wants to use this linked list implementation?

# linked list: multiple files

```
typedef struct {
     int val;
     struct node *next;
 }node;
node *insert(node *he
```

header file includes type definitions and exported function signatures

If not included, gcc does have info on the node type to compile list.c

list.h

```
#include "list.h"
node* insert(node *head, int val) {
    node *new_head = (node *)malloc(sizeof(node));
    new_head->next = head;
    new_head->val = val;
}
```

list.c

```
$ gcc -c list.c
```

generate object file list.o

```
$ gcc list.c
```

will not work since main() is not defined

# linked list: multiple files

```c
#include “list.h”
int main() {
    node *head = NULL;
    for (int i = 0; i < 3; i++)
        head = insert(head, i);
}
```

test1.c

```c
#include “list.h”
int main() {
    node *head;
    for (int i = 0; i < 3; i++)
        head=insert(head, i);
}
```

test2.c

```
$ gcc -c test1.c
$ gcc test1.o list.o
$ ./a.out
```

generate object file test1.o

link test1.o and list.o to form executable a.out

# Exporting global variables

```
typedef struct {
     int val;
     struct node *next;
 }node;
node *insert(node *head, int val);
```

list.h

```
#include “list.h”
int debug;
node* insert(node *head, int val) {
    ...
    if (debug > 0)
       printf(“inserted val %d\n”, val);
}
```

list.c

```
#inde “list.h”
int main() {
   debug = 1;
   ...
}
```

test1.c

# Exporting global variables

```
typedef struct {
     int val;
     struct node *next;
 }node;
extern int debug;
node *insert(node *head, int val);
```

Declares debug variable but does not allocate space

list.h

```
#include “list.h”
int debug;
node* insert(node *head, int val) {
    ...
    if (debug > 0)
       printf(“inserted val %d\n”, val);
}
```

list.c

```
#inde “list.h”
int main() {
   debug = 1;
   ...
}
```

test1.c

# C does not have explicit namespace

- Scope of a global variable / function by default is across all files (linked together)
- To restrict the scope of a global variable / function to this file only, prefix with “static” keyword

No other files can use the debug variable and insert function

```
#include “list.h”
static int debug;
static node* insert(node *head, int val) {
    ...
    if (debug > 0)
       printf(“inserted val %d\n”, val);
}
```

list.c

# static prefixing local variables means different things

- Normal local variables are de-allocated upon function exit
- Static local variables are not de-allocated
  - offers private, persistent storage across function invocation

```
node* insert(node *head, int val) {
    static int n_inserts = 0;
    ...
    n_inserts++;
    printf("number of inserts %d\n", n_inserts);
}
```

initialized once, never deallocated (like a global variable, except with local scape)

# C standard library

<assert.h> assert

<ctype.h> isdigit(c), isupper(c), isspace(c), tolower(c), toupper(c) ..

<math.h> log(f) log10(f) pow(f, f), sqrt(f), ...

<stdio.h> fopen, fclose, fread, fwrite, printf, ...

<stdlib.h> malloc, free, atoi, rand

<string.h> strlen, strcpy, strcat, strcmp

Section 3 of manpage is dedicated to C std library

To read manual, type

```
man 3 strlen
```

# The C pre-processor

- All the hashtag directives are processed by C pre-processor before compilation
- #include <stdio.h>
  - insert text of included file in the current file
  - with <...> , preprocessor searches system path for specified file
  - with "...", preprocessor searches local directory as well as system path

# C Macros

- #define name replacement_text

```
#define NITER 10000

int main()
   for (int i = 0; i < NITER; i++) {
      ....
   }
}
```

It's better to write:

```
static const int niter = 10000;
```

# C Macros

- Macro can have arguments
- Macro is NOT a function call

```
#define SQUARE(X) X*X

a = SQUARE(2);

b = SQUARE(i+1);

c = SQUARE(i++);
```

```
a = 2*2;
```

```
b = i+1*i+1;
```

# C Macros

- Macros can have arguments
- Macro is NOT a function call

```
#define SQUARE(X) (X)*(X)

a = SQUARE(2);          a = (2)*(2);

b = SQUARE(i+1);        b = (i+1)*(i+1);

c = SQUARE(i++);        c = (i++)*(i++);
```

Macro is hard to debug, avoid it if you can

# Doing I/O in C

# I/O in C

- I/O facilities are not part of core C language
  - provided by library using OS facilities.

- Two interfaces
  - (high level) Buffered I/O:
    - implemented by stdio library
    - uses low level interface internally
  - (low level) UNIX(Unbuffered) I/O:
    - an API provided by OS to invoke its I/O functionalities.

# Buffered I/O

- each I/O stream is represented by a file pointer of type FILE*
- Obtain the file pointer using fopen
  - file should be closed upon finish: fclose
- Access the file using file pointer with functions
  - fread, fwrite, fgetc, fgets

# Buffered I/O

- each I/O stream is represented by a file pointer of type FILE*
- Special streams: no need to explicitly open them
  - stdin
  - stdout
  - stderr

# Buffered I/O example

- Count # of lines in a file

```
// open file using (fopen)

// while not end of file stream
       read file line by line (fgets)
       increment counter

// close file (fclose)
// print out counter value
```

# Buffered I/O example

```
#include <stdio.h>

int main(int argc, char **argv)
{
    //open file based on argum

    int n = countlines(fp);

    //close file

    printf("# of lines %d\n", n);
}
```

**Type "man fopen"**

**FILE *fopen(const char *path, const char *mode);**

fopen opens the file whose name is the string pointed to by path and associates a stream with it.

The argument mode points to a string beginning with one of the following sequences

**r** Open file for reading.
**r+** Open for reading and writing.
w Truncate file to zero length or create file for writing.
....

# Buffered I/O example

```
int main(int argc, char **argv)
{
    //open file based on argument
    FILE *fp = fopen(argv[1], “r”);

    int n = countlines(fp);

    //close file
    fclose(fp);

    printf(“# of lines %d\n”, n);
}
```

# Buffered I/O example

```
int countlines(FILE *fp)
{
      int count = 0;

      while (!feof(fp)) {
          fgets(...)
          count++;
      }

      return count;
}
```

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

**fgets**() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s.

Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

# Buffered I/O example

```
#define BUFSZ 1000
int countlines(FILE *fp)
{
      int count = 0;
      char buf[BUFSZ];

      while (!feof(fp)) {
          fgets(buf, BUFSZ, fp);
          count++;
      }

      return count;
}
```

# Buffered I/O example

```
int countlines(FILE *fp)
{
     int count = 0;
     char *buf;

     while (!feof(fp)) {
          buf = (char *)malloc(BUFSZ);
          fgets(buf, 1000, fp);
          count++;
     }

     return count;
}
```

# Buffered I/O example

```
int countlines(FILE *fp)
{
      int count = 0;
      char buf[BUFSZ];

      while (!feof(fp)) {
          fgets(buf, BUFSZ, fp);
          count++;
      }

      return count;
}
```

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

**fgets**() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s.

Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

**fgets**() returns s on success, and NULL on error or when end of file occurs while no characters have been read.

# Buffered I/O example

```
int countlines(FILE *fp)
{
      int count = 0;
      char buf[BUFSZ];

      while (!feof(fp)) {
          if (!fgets(buf, BUFSZ, fp))
              break;
          count++;
      }
      return count;
}
```

# Buffered I/O example

```
int countlines(FILE *fp)
{
      int count = 0;
      char buf[BUFSZ];

      while (!feof(fp)) {
          fgets(buf, BUFSZ, fp);
          count++;
      }

      return count;
}
```

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

**fgets**() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s.

Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

**fgets**() returns s on success, and NULL on error or when end of file occurs while no characters have been read.

# Buffered I/O example

```
int countlines(FILE *fp)
{
      int count = 0;
      char buf[BUFSZ];

      while (!feof(fp)) {
          if (!fgets(buf, BUFSZ, fp))
              break;
          if (buf[strlen(buf)-1]==‘\n’) {
             count++;
          }
      }

      return count;
}
```

# Buffered I/O example

```
int countlines(FILE *fp)
{
      int count = 0;
      char buf[1000];
      while (!feof(fp)) {
          if(!fgets(buf, 1000, fp))
              break;
          if(buf[strlen(buf)-1]==‘\n’){
             count++;
          }
      }
      return count;
}
```

buffer allocated by caller

buffer allocated by callee

```
BufferedReader br = new BufferedReader(new FileReader(file)));
String line;
int count = 0;
while ((line = br.readLine()) != null) {
    count++;
}
```

# (Low-level) UNIX I/O

- Used by stdio library to implement buffer I/O
- A thin wrapper to interface with OS kernel


- Each I/O stream is represented by an integer (called file descriptor).
- Special file descriptors:
  - 0: standard input
  - 1: standard output
  - 2: standard error

# UNIX I/O example: Count lines

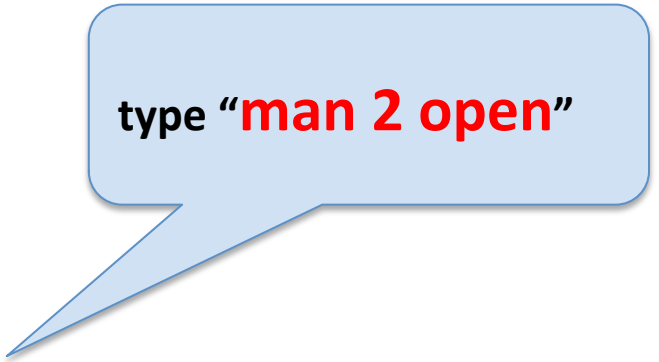
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
    //open file based on argument
    int fd = open(argv[1], O_RDONLY);

    int n = countlines(fd);

    //close file
    close(fd);

    printf("# of lines %d\n", n);
}
```



# UNIX I/O example: count lines

```
#include <unistd.h>
int countlines(int fd)
{
      int count = 0;
      char buf[BUFSZ];
      ssize_t n;

      while ((n = read(fd, buf, BUFSZ)) > 0
          for (ssize_t i = 0; i < n; i++) {
              if (buf[i] == '\n') {
                  count++;
               }
          }
      }

      return count;
}
```

typedef long ssize_t

**Type "man 2 read"**

**ssize_t read(int** fd**, void** *buf**, size_t** count**);**

**read**() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

On success, the number of bytes read is returned (zero indicates end of file), On error, -1 is returned...

# What is FILE*

```
typedef struct {
   int cnt;  // characters left in buffer
   char *ptr;  // next character in the buffer
   char *base; // location of buffer
   int mode;  // mode of file access
   int fileno;  // file descriptor

} FILE;
```

Can you implement fopen, fclose, fgets using open, close, and read?
see page 176-177 of K&R