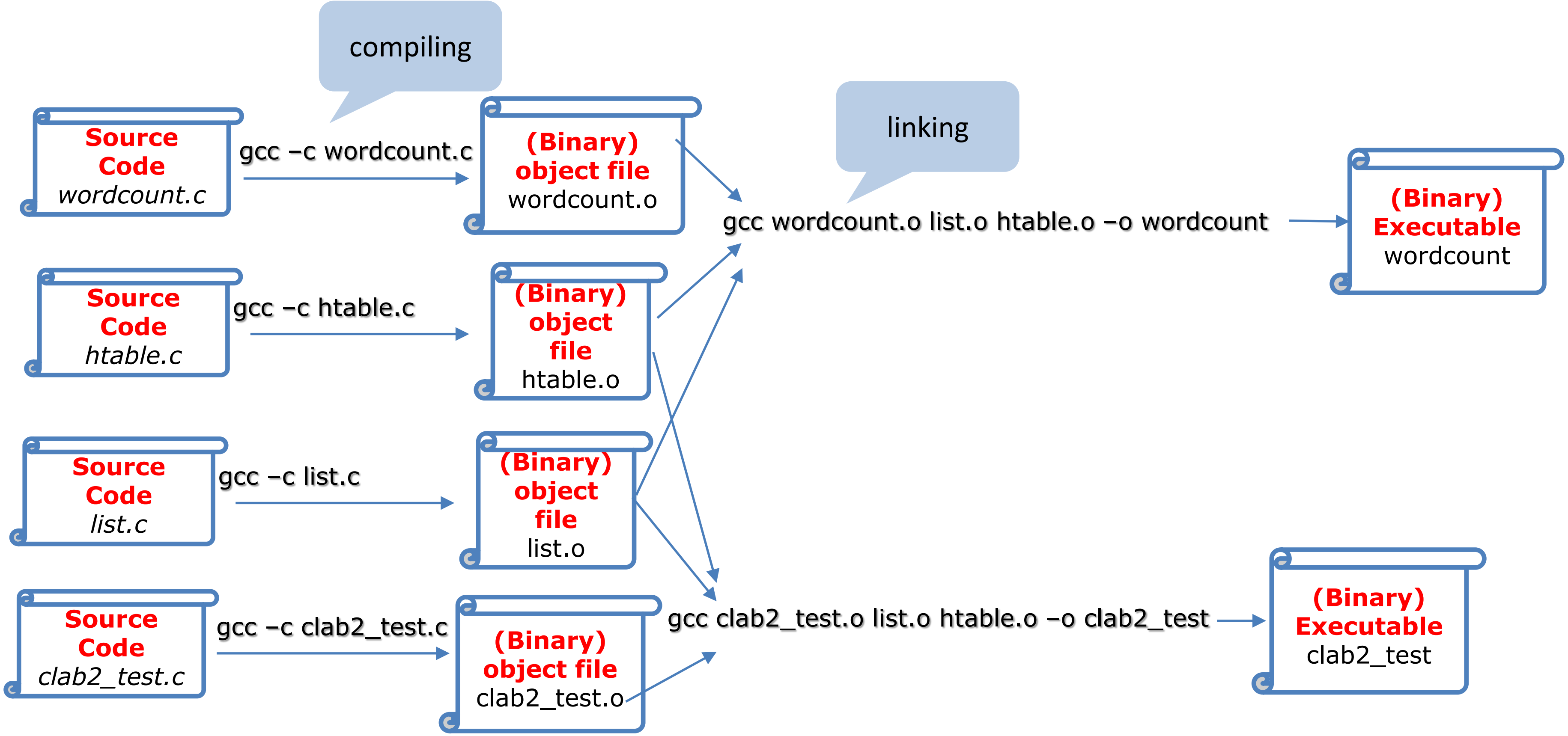
Large C Program organization, I/O

Jinyang Li

This lesson

- More on C project organization
  - C pre-processing
- Doing I/O

Lab2’s compilation sequence



compiling

Source Code
wordcount.c

```
gcc –c wordcount.c
```

(Binary) object file
wordcount.o

Source Code
htable.c

```
gcc –c htable.c
```

(Binary) object file
htable.o

Source Code
list.c

```
gcc –c list.c
```

(Binary) object file
list.o

Source Code
clab2_test.c

```
gcc –c clab2_test.c
```

(Binary) object file
clab2_test.o

linking

```
gcc wordcount.o list.o htable.o –o wordcount
```

(Binary) Executable
wordcount

```
gcc clab2_test.o list.o htable.o –o clab2_test
```

(Binary) Executable
clab2_test

Role of header files

```
…
typedef struct lnode{
   kv_t tuple;
   struct lnode *next;
}lnode_t;

void list_init(lnode_t **headdp);
bool list_insert_with_accum(…);
```

header file includes type definitions and exported function signatures

list.h

```
…
#include “list.h”

void simple_list_test()
{
   lnode_t *headp;
   list_init(&headp);
   panic_cond(headp==NULL, “….”);
}
```

If header file is not included, gcc would complain about unknown function “list_init”

clab2_test.c

Exporting global variables

```
typedef struct lnode{
   kv_t tuple;
   struct lnode *next;
}lnode_t;
extern int num_inserts;
void list_init(lnode_t **headdp);
bool list_insert_with_accum(…);
```

“Extern” declares variable but does not allocate space

list.h

```
int num_inserts;
bool list_insert_with_accum(…)
{
  num_inserts++;
}
```

Defines global variable and allocates space (upon program start)

list.c

```
#include “list.h”

void simple_list_test()
{
   lnode_t *headp;
   list_init(&headp);
   list_insert_with_accum(…);
   printf(“num_inserts=%d\n”, num_inserts);
}
```

Uses global variable exported in “list.h”

clab2_test.c

C does not have explicit namespace

- Scope of an (exported) global variable or function is across all files (that are linked together)
  - What if different files happen to use the same global variable name or function name?
- Restrict scope of a global variable / function to this file only
  - Use the “static” keyword

```
#include “list.h”
static int num_inserts;
static internal_func(…) {
    ...
}
```

list.c

No other files can use the num_inserts variable and internal_func function

“static” keyword has a diff meaning when prefixing local variables

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

```
void insert(…) {
    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 scope)

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 <f.h>
  - insert text of `f.h` in the current file
  - with `<f.h>` , preprocessor searches for `f.h` in system paths
  - with “`f.h`”, preprocessor searches for `f.h` in the local directory before searching in system paths

C processor supports 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);                a = 2*2;

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

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

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++);      ✗
```

what is NULL?

#define NULL ((void *)0)

Doing I/O in C

I/O in C

- I/O facilities are not part of core C language
  - provided by OS facilities (called syscalls)
  - For a list of syscalls provided, type \``man 2 syscalls`\`

- Two interfaces
  - (low level) UNIX(unbuffered) I/O:
    - A thin wrapper around OS I/O related syscalls.
  - (high level) Buffered I/O:
    - implemented by stdio library
    - uses low level interface internally
    - Buffers multiple I/Os together into a single low-level I/O call for better performance.

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

Type
man stdio

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 argume


   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 (1) {
     //read a line using 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.

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

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

        while (1) {
            char *buf = malloc(BUFSZ);
            if (!fgets(buf, BUFSZ, fp))
                break;
            count++;

        }

        return count;
}
```

It’s the responsibility of the caller (not fgets) to allocate buffer for reading a line.

🤓🤓Any problem??

Buffered I/O example

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

     while (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.

…

⚠ What if a line is longer than BUFSZ?

Buffered I/O example

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

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

   return count;
}
```

Replace with
`if buf[BUFSZ-2]!=‘\n’?`

Buffered I/O example

```
int countlines(FILE *fp)
{
     int count = 0;
     char buf[BUFSZ];
     while (fgets(buf, BUFSZ,fp)) {
       if (buf[strlen(buf)-1]!=‘\n’)
         continue;
       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

system call interface

- 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);
}
```

type “**man 2 open**”

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

Summary

- Review C project organization
  - Header files
  - C preprocessing
- I/O
  - Lower level I/O (open, read, write)
    - Unbuffered. Directly interface with OS (syscall)
  - Buffered I/O (fopen, fread, fwrite, fgets)
    - Built on top of low level I/O with a buffer.
    - Improves performance by buffering multiple I/Os into a single low-level I/O call.