CSC 357 Lecture Notes Week 3

Leftovers from Week 2 Notes; Additional C Language and Library Features

I. C I/O (K&R Chapter 7).

A. Strictly speaking, I/O not part of C language.

B. Rather, it’s part of standard library.

C. You’ve seen and used `stdio.h`.

D. Further detail here.

II. Standard input and output (K&R Section 7.1).

A. Names of file streams are `stdin`, `stdout`.

B. Char-at-a-time functions `getchar` and `putchar`.

C. `printf` also goes to `stdout`.

Standard I/O, cont'd

D. Stdio streams *redirected* and *piped* with shell operators '<', '>', and ' | '.

III. Formatted output -- `printf` and `sprintf` (K&R Section 7.2).

A. You've used `printf` plenty already.

B. Read the `printf` man page.

C. Page 154 of K&R has a handy table of `%` formatting codes.

Formatted output, cont’d

D. `sprintf` let’s you do in-memory "printing".

1. Same as printf, but to a string buffer.

2. First arg is `char*` buffer; rest of args same as `printf`.

IV. Variable-length arg lists (K&R Section 7.3).

A. `printf` has variable number of args.

B. Define your own using macros in `<stdarg.h>`.

C. We’ll cover in upcoming lab.

V. Formatted input -- `scanf` (K&R Section 7.4).

A. `scanf` is input analog of `printf`.

B. First arg is a formatting string.

C. For `scanf`, '`%`' codes govern how inputs are interpreted and converted.

Formatted input, cont'd

D. The input variables are the `scanf` arguments following the formatting string.

E. We'll not use `scanf` much in 357, but will cover it a bit in an upcoming lab.

VI. File access (K&R Section 7.5).

A. The functions discussed thus far work with `stdin` and `stdout`.

B. To read from a stored file, you first use `fopen`:

```
    FILE* fopen(
        char* name, char* mode);
```

File access, cont'd

1. First arg is name of file.
2. Second arg is mode, as specified at shell level (see fopen man page).

File access, cont'd

C. `FILE` is a structure declared in `<stdio.h>`.

D. Character-level read/write with `getc` and `putc`.

E. Operate just as `getchar` and `putchar`:

```
#define getchar() getc(stdin)

#define putchar() putc(stdout)
```

File access, cont'd

F. File versions of `scanf` and `printf`:

```
int fscanf(
    FILE* fp, char* format, ...)

int fprintf(
    FILE* fp, char* format, ...)
```

See the man pages

File access, cont'd

G. `fclose` closes a file opened with `fopen`

- most OSs have a limit on the number of files that can be open at the same time
- always a good idea to use `fclose` whenever a file is no longer needed.

VII. Error handling (K&R Section 7.6).

A. `printf` goes to `stdout`.

B. C provides second output called `stderr`.

Error handling, cont’d

1. Sent to `stderr` using `fprintf`, as in

```
fprintf(stderr,
  "%s: No such file or directory",
  filename);
```

Error handling, cont’d

2. When `stdio` is redirected to a file with ’>’, `stderr` still appears on terminal.

3. To redirect both, use ’`>&`’.

Error handling, cont'd

C. C program signals an error in two ways -- `stderr` stream and `exit` system function.

1. Calling `exit` terminates program.

2. Integer argument is returned by entire program.

3. Conventionally, 0 means exit normally.

Error handling, cont’d

4. Non-zero used to signal specific errors.

5. UNIX system calls usually return -1 to signal an error, and set the external variable `errno`

VIII. Line input and output (K&R Section 7.7).

A. `fgets` reads a line of input from a file stream; its signature is

```
char* fgets(char* line,
            int maxline,
            FILE* fp)
```

If successful, returns line, null otherwise.

Line I/O, cont’d

B. The `fputs` function is the output analog

```
int fputs(char* line, FILE* fp)
```

If successful, returns number of chars output, EOF otherwise.

IX. Miscellaneous functions (K&R Section 7.8).

A. This section of K&R provides a brief overview of system functions we have been using in the assignments.

B. The man pages and Stevens book have more detailed information.

X. Makefiles.

A. See Gnu manual page, cited in lab writeup.

B. Hold commands to be conveniently executed.

1. Frequently, used for compilation.
2. However, any UNIX commands can be used.
3. Used to run tests, print, other tasks.

Makefiles, cont’d

C. Make also performs "smart recompilation".

D. During lecture/lab, we’ll dissect Makefiles for linked-list program.

Updates to OBJS-Style Makefile

```
CFLAGS = -Wall -ansi -g
CC = /opt/gnu/bin/gcc

OBJS = nwc.o hash.o getwd.o

nwc:     $(OBJS)
         $(CC) $(CFLAGS) $(OBJS) -o nwc

clean:
         rm *.o nwc
```

Notes on Program Organization

- Conventions say must use `.c`, `.h` pairs.

- E.g., `nwc.c`, `nwc.h`, `hash.c`, `hash.h`, ...

- The `main` function goes in "main" `.c` file, e.g., `nwc.c`.

Notes on Program 2 Testing

- Prog 2 testing dir had executable `nwc`.
- Replace it with your `nwc`.
- You can use `357/programs/2/nwc` to compare its output to yours.

XI. `static` storage class (K&R Section 4.6).

A. Vars can be declared `static`, as in

```
static int i;
```

B. External `static` vars are not visible in other C source files.

static, cont’d

C. `static` can be used for local function vars.

D. We’ll discuss further in upcoming lecture.

XII. Memory layout in a C program.

A. The memory used by a C program is organized conceptually into three storage areas:

1. *static pool*

2. *stack*

3. *heap*

Memory layout, cont’d

B. Lifetime of storage is:

1. Static-pool is lifetime of the entire program.

2. Function parameters and non-static local vars is activation lifetime of function.

3. Heap storage alive until `free`ed, or program ends.

Memory layout, cont’d

Consider following example.

```
#include <strings.h>
#include <stdlib.h>
#include <stdio.h>
```

NOTE: The code has one of my favorite C bugs; say something when you see it.

```
char s[20];

char* f(char* s1, char* s2, int* ip) {
    char* s3;

    s3 = strcat(s1, s2);

    strcpy(s, s3);

    *ip = strlen(s3);

    return s3;
}
```

```
void main() {

    char s1[20] = "abcdef";

    char* s2 = strcpy((char*)
        malloc(strlen("ghijklmn")),
        "ghijklmn");

    char* s3;

    int i;
    s3 = f(s1, s2, &i);

    free(s2);

    printf("...",
           s, s3, strlen(s), i);

    printf("...",
           sizeof(s), sizeof(s3));
}
```

XIII. Program modularization and information hiding in C.

A. C programs modularized using `.h`, `.c` files.

1. `.h` file contains type decls, function decls, global constants, global (module) vars.

2. `.c` file contains implementation of functions.

Modularization, cont'd

B. `static` declaration provides info hiding.

1. Global statics visible only to functions declared in the module.

2. Functions declared static are similarly limited in visibility.

XIV. `doxygen`

A. `doxygen` is a documentation-generation tool for C and C++ programs

B. Operates very much like `javadoc`.

C. See `person-record` example program (for Lab 3).

Example of Local Static

```
/* Simple string list iterator. */
char* next(char** str_list) {
    static int i = 0;

    if (str_list[i] != NULL) {
        return str_list[i++];
    }
    else {
        i = 0;
        return NULL;
    }
}
```

Local Static, cont’d

```
int main() {
  char* str_list[] =
    {"1", "2", "3", "4", "5", NULL};
  char* s;

  while ((s = next(str_list)) != NULL) {
      printf("%s ", s);
  }
  printf("0);
}
```