Computing Science (CMPUT) 201

Practical Programming Methodology

Davood Rafiei

Department of Computing Science
University of Alberta
drafiei@cs.ualberta.ca

Fall 2012

Outline

Part II

First Examples

Vertical Strips

- A broad knowledge of Unix and C is achieved over years through experience
- Vertical strips are examples that cut deeply through many different layers of abstraction, but in a shallow way at many related subjects
- Fill out the details by consulting references and through experience
  - We depend on you to read, try, do, and ask your way to a mastery of the material.

A Vertical Strip Example: Running Programs

```
drafiei@ug20:~/201>wget https://webdocs.cs.ualberta.ca/~drafiei/201/Sketchpad.jar
...snip...
drafiei@ug20:~/201>wget https://webdocs.cs.ualberta.ca/~drafiei/201/sketchpad-sample-
...snip...
drafiei@ug20:~/201>ls -lt
total 100
-rw------- 1 drafiei prof    278 Sep  2 16:40 sketchpad-sample-input
-rw------- 1 drafiei prof  5565 Sep  2 16:25 Sketchpad.jar
drwxr-xr-x 3 drafiei prof   512 Aug 26 15:07 pub
drafiei@ug20:~/201>ls -l ~drafiei/201/pub
total 4
drwxr-xr-x 2 drafiei prof 512 Aug 27 17:37 code
drafiei@ug20:~/201>java -jar Sketchpad.jar <sketchpad-sample-input
```

1. The Unix shell (e.g., bash, tcsh) prompts the user for commands to execute with `drafiei@ug20`
2. `wget` and `ls` are all programs executed from the command line of the shell.
   Unix is often used via the command-line interface

`man` pages

- Manual (`man`) pages are the traditional, on-line documents for Unix.
  - `man wget`: details for a specific program
  - `man -k http`: does a (lame) search using http as keyword
  - `man man`: details how to use `man`
- Often, Unix commands have esoteric command-line options. For example, `ls -lt`.
  - `ls`: the “list directory contents” program
  - `-l`: long listing format
  - `-t`: sort by the timestamp

Compiling Programs

```
drafiei@ug20:~/201>cp ~drafiei/201/pub/Code/removeSpaceAndTab.c .
drafiei@ug20:~/201>ls -lt
total 8
-rw------- 1 drafiei prof 353 Aug 27 14:24 removeSpaceAndTab.c
drwxr-xr-x 3 drafiei prof 512 Aug 26 15:07 pub
drafiei@ug20:~/201>gcc -Wall -std=c99 removeSpaceAndTab.c -o removeSpaceAndTab
drafiei@ug20:~/201>ls -lt
total 16
-rwx------ 1 drafiei prof 7697 Aug 27 14:27 removeSpaceAndTab
-rw------- 1 drafiei prof  353 Aug 27 14:24 removeSpaceAndTab.c
drwxr-xr-x 3 drafiei prof  512 Aug 26 15:07 pub
```

1. Let's compile the C `removeSpaceAndTab.c` that we have written in class.
2. A program is compiled using a compiler, such as `gcc`
3. `-Wall -std=c99` are command-line options for `gcc`. They are also called compiler options. We will use `-std=c99` in our lectures and assignments. The option `-ansi` is another alternative.
4. `-o removeSpaceAndTab` specifies the name of the executable file. Default name is `a.out`

Running Find Double Word

```
drafiei@ug20:~/201>gcc -Wall -std=c99 finddw.c -o finddw
...snip (some compiler warnings)...
drafiei@ug20:~/201>cat finddw.test
Sometimes, it is is easy to
write the same word twice
twice and not make a spelling mistake.

drafiei@ug20:~/201>./finddw finddw.test
In file finddw.test, repeated word:  is is
Context:   easy to

Re-run program after fixing!!!
 In file finddw.test, repeated word:  twice twice
Context:   and not make a spelling mistake.

Re-run program after fixing!!!

drafiei@ug20:~/201>cat finddw.ok
This file is fine.
There is nothing to worry about.
drafiei@ug20:~/201>./finddw finddw.ok
drafiei@ug20:~/201>
```

- `cat` is a program to send the contents of a file to the terminal
- `finddw` knows which file to use from the command-line argument

Find Double Word

```
drafiei@ug20:~/201>cat finddw.c
#include <stdio.h>
#include <string.h>     /* For strncmp(), etc. */

#define MIN_BUF         256
#define MAX_BUF         2048

char Buffer[ MAX_BUF ];
char Word[ MIN_BUF ];
char NextWord[ MIN_BUF ];

void parseFile( FILE * fp, char * fname );
```

(code listing continues on later slides)

- Before the executable code, a best practice is to list:
  - header files (`#include`)
  - (macro) definitions (`#define`)
  - global variables
  - function prototypes

Find Double Word (2)

```
int main( int argc, char * argv[] ) {
        int     i;
        FILE    * fp;
        for (i = 1; i < argc; i++) {
                fp = fopen( argv[ i ], "r" );
                if ( fp == NULL ) {
                        printf( "Could not open file %s\n", argv[ i ] );
                        exit( -1 );

                }
                else {
                        parseFile( fp, argv[ i ] );
                        fclose( fp );
                }
        }
        return 0;
} /* main */
```

(code listing continues on later slides)

- The `main()` function is where the program starts executing. The Unix shell finds the executable and starts running it at function `main()`.
- `main()` is a function, not a method. But, it has similar syntax in its function definition compared to Java.

Find Double Word (3)

```
int main( int argc, char * argv[] ) {
        int     i;
        FILE    * fp;
        for (i = 1; i < argc; i++) {
                fp = fopen( argv[ i ], "r" );
                if ( fp == NULL ) {
                        printf( "Could not open file %s\n", argv[ i ] );
                        exit( -1 );
                }
                else {
                        parseFile( fp, argv[ i ] );
                        fclose( fp );
                }
        }
        return 0;
} /* main */
```

(code listing continues on later slides)

- Local variables (e.g., `int i`, `FILE * fp`) are defined before being used
- For loops (e.g., `for( ... )`) are similar to Java
- Control flow (e.g., `if() ... else`) is similar to Java

Find Double Word (4)

```
int main( int argc, char * argv[] ) {
        int     i;
        FILE    * fp;
        for ( i = 1; i < argc; i++ ) {
                fp = fopen( argv[ i ], "r" );
                if ( fp == NULL ) {
                        printf( "Could not open file %s\n", argv[ i ] );
                        exit( -1 );
                }
                else {
                        parseFile( fp, argv[ i ] );
                        fclose( fp );
                }
        }
        return 0;
} /* main */
```

(code listing continues on later slides)

- function calls (e.g., `fopen(), parseFile(), fclose()`) are similar to Java
- C input/output is done by the Standard I/O (`stdio`) library (e.g., `printf()`)

Find Double Word (5)

```
void parseFile( FILE * fp, char * fname ) {
        int     rval;

        /* Read first word */

        rval = fscanf( fp, "%s", Word );
        if ( rval != 1 ) {
                printf( "Failed to read first word\n" );
                exit( -1 );
        }
```

(code listing continues on later slides)

- comments should be used (e.g., `/* Read first word */`) to explain the semantics, not the syntax
- 90% of the time, clear code that follows standard conventions is preferred over clever code
- C file input is by the Standard I/O (`stdio`) library (e.g., `fscanf()`)

Find Double Word (6)

```
        while ( ! feof( fp ) ) {
                rval = fscanf( fp, "%s", NextWord );
                if ( rval != 1 )
                        continue;

                if ( strncmp( Word, NextWord, MIN_BUF ) == 0 ) {
                        printf( "In file %s, repeated word:  %s %s\n",
                                fname, Word, NextWord );

                        /* Heuristic as to when to print out context info */

                        /* First letter must be alphabets */
                        if ( isalpha( Word[ 0 ] ) ) {
                                fgets( Buffer, MAX_BUF, fp );
                                printf( "Context:  %s\n", Buffer );
                                printf( "Re-run program after fixing!!!\n " );
                        }
                }

                strncpy( Word, NextWord, MIN_BUF );
        } /* while */
} /* parseFile */
```

- While loops (e.g., `while( ... )`) are similar to Java
- C strings are handled using a family of functions (e.g., `strncmp()`, `strncpy()`)

Cautionary Tale: test programs, strncpy, man pages (1)

```
        /* Simple test program to understand strncpy() */
#include <stdio.h>
#define __USE_GNU       /* Not _GNU_SOURCE, as per "man strnlen" */
#include <string.h>     /* For memset(), etc. */

#define LEN 32
int main( int argc, char * argv[] )
{
        char * source = "hello";
        char dest[ LEN ];

        /* Setup */
        memset( dest, 'a', LEN ); /* In C, we can just "write" to mem */
        dest[ LEN - 1 ] = 0;
        printf( "s (%2d): %s||  d (%2d): %s|| d[ 30 ]  = '%c'\n",
                (int)strnlen( source, LEN ), source,
                (int)strlen( dest ), dest, dest[ 30 ]
        );

        /* Correct usage, careful of semantics */
        strncpy( dest, source, LEN );
        printf( "s (%2d): %s||  d (%2d): %s|| d[ 30 ]  = '%c'\n",
                (int)strlen( source ), source,
                (int)strlen( dest ), dest, dest[ 30 ]
        );

        /* Will cause segmentation fault */
        strncpy( dest, source, 100000 );
        return 0;
}
```

Cautionary Tale: test programs, strncpy, man pages (2)

```
drafiei@ug20:~/201>gcc -Wall -std=c99 -g teststrncpy.c
drafiei@ug20:~/201>./a.out
s ( 5): hello||  d (31): aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|| d[ 30 ]  = 'a'
s ( 5): hello||  d ( 5): hello|| d[ 30 ]  = ''
Segmentation fault
```

- The `strncpy()` copied the string `hello`, but it **also** overwrote the end of array `dest[]` too.
- For most C programmers, this is a surprise. But, the man page actually tells us about this behaviour.

Cautionary Tale: test programs, strncpy, man pages (3)

```
drafiei@ug20:~>man strncpy
STRCPY(3)                 Linux Programmer's Manual                 STRCPY(3)

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);

       char *strncpy(char *dest, const char *src, size_t n);

DESCRIPTION
       The  strcpy()  function copies the string pointed to by src (including
       the terminating '\0' character) to the array pointed to by dest.   The
       strings may not overlap, and the destination string dest must be large
       enough to receive the copy.

       The strncpy() function is similar, except that not more than  n  bytes
       of  src  are  copied. Thus, if there is no null byte among the first n
       bytes of src, the result will not be null-terminated.

       In the case where the length of src  is  less  than  that  of  n,  the
       remainder of dest will be padded with null bytes.
```

- Note the comment about `the remainder of dest will be padded with null bytes`

Cautionary Tale: test programs, strncpy, man pages (4)

```
drafiei@ug20:~/201>gdb a.out
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...Using host libthread_db libr
ary "/lib/tls/libthread_db.so.1".

(gdb) run
Starting program: /var/user/a.out
s ( 5): hello||  d (31): aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa|| d[ 30 ]  = 'a'
s ( 5): hello||  d ( 5): hello|| d[ 30 ]  = ''

Program received signal SIGSEGV, Segmentation fault.
0xb7f60412 in strncpy () from /lib/tls/libc.so.6
(gdb)where
#0  0xb7f60412 in strncpy () from /lib/tls/libc.so.6
#1  0x0804854b in main (argc=0, argv=0x0) at testStrncpy.c:25
(gdb) list 25
20        printf("s (%2d): %s||  d (%2d): %s|| d[ 30 ]  = '%c'\n",
21                (int)strlen( source ), source,
22                (int)strlen( dest ), dest, dest[ 30 ]);
23
24        /* Will cause segmentation fault */
25        strncpy( dest, source, 100000 );
26        return 0;
27      }
28
(gdb) quit
```

Cautionary Tale: test programs, strncpy, man pages (5)

```
...snip...

#define LEN 32
int main( int argc, char * argv[] )
{
        char * source = "hello";
        char dest[ LEN ];

...snip...

        /* Correct usage, careful of semantics */
        strncpy( dest, source, LEN );
        printf( "s (%2d): %s||  d (%2d): %s|| d[ 30 ]  = '%c'\n",
                (int)strlen( source ), source,
                (int)strlen( dest ), dest, dest[ 30 ]
        );

        /* Will cause segmentation fault */
        strncpy( dest, source, 100000 );
```

- The first call-site `strncpy( dest, source, LEN );` is OK, since it remains within the 32 byte buffer.
- The second call-site `strncpy( dest, source, 100000 );`, causes a segmentation fault (usually, a memory error), since it overruns the buffer. This is a common mistake.

Makefiles

```
drafiei@ug20:~/201>cat Makefile
# A simple makefile

finddw: finddw.c
        gcc -Wall -std=c99 finddw.c -o finddw
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
```

1. By default, the program `make` looks for `Makefile` or `makefile` (which are text files).
2. Makefiles are examples of declarative programs: You tell the system what you want, more than how to achieve it. In truth, it is a hybrid of declarative and imperative.
3. First Common Pitfall: The white space in front of `gcc -Wall ...` is actually an invisible tab.

Makefiles (2)

```
drafiei@ug20:~/201>cat Makefile
# A simple makefile

finddw: finddw.c
        gcc -Wall -std=c99 finddw.c -o finddw
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
```

1. `finddw:  finddw.c:`
    1. Defines a makefile target (i.e., `finddw`) and its (list of) dependencies (i.e., `finddw.c`) (aka prerequisite-list).
    2. A makefile can have multiple targets.
    3. The dependencies can be files, or other targets.
2. TAB `gcc -Wall -std=c99 finddw.c -o finddw`
    1. Defines an action (aka construction-commands).
    2. Actions are anything that you can do from the Unix command line.

Make Execution Model

```
drafiei@ug20:~/201>cat Makefile
# A simple makefile

finddw: finddw.c
        gcc -Wall -ansi finddw.c -o finddw
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
```

1. In this example, `make` and `make finddw` have the same effect.
2. When you execute `make`
   1. Make looks for a makefile.
   2. Make looks for your target (e.g., `finddw`) or uses the first one by default.
   3. Make looks at the timestamps (cf. `ls -lt`) on the target and the dependencies.
   4. If the target is more recent than the dependencies, then make does nothing more.

Make Execution Model (2)

```
drafiei@ug20:~/201>cat Makefile
# A simple makefile

finddw: finddw.c
        gcc -Wall -std=c99 finddw.c -o finddw
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
```

1. If the target is less recent than the dependencies, or if the target does not exist, then make invokes the targets for the dependencies (if any).
2. Then, make invokes the actions for the current target.
3. Second Common Pitfall: It is easy to forget a dependency for a target, especially when refactoring code. Solution: Be careful. And, have a `make clean` target, as discussed below.

Make Execution Model (3)

```
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
drafiei@ug20:~/201>make
make: 'finddw' is up to date.
drafiei@ug20:~/201>touch finddw.c
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw

drafiei@ug20:~/201>which which
which: shell built-in command.
drafiei@ug20:~/201>which touch
/usr/bin/touch
```

1. `touch` is a Unix command that "updates" the timestamp of a file to, for example, simulate editing a file. The contents of the file does not change.
2. Notice how make's declarative execution model only looks at timestamps, and does not look at the contents of a file.
3. `which` is a Shell command to tell you where (or which) an executable is located

Make Execution Model (4)

```
drafiei@ug20:~/201>cat Makefile.v2
finddw: finddw.c
	gcc -Wall -std=c99 finddw.c -o finddw

clean:
	-rm *.o finddw

tar:
	tar cvf submit.tar finddw.c Makefile README
```

Makefiles are useful for more than just building programs. Anything that can be automated as a sequence of actions and/or targets and dependencies can be automated via a Makefile.

Target `clean` is a common target (and required in your assignments). Note that you should never have a file with the name `clean`.

Make Execution Model (5)

```
drafiei@ug20:~/201>cat Makefile.v2
finddw: finddw.c
        gcc -Wall -std=c99 finddw.c -o finddw

clean:
        -rm *.o finddw

tar:
        tar cvf submit.tar finddw.c Makefile README
```

The `-rm ...` removes files that can be (and should be) re-created from other files.

- For example, `.o` files are created from `.c` files (more later).
- For example, executable `finddw` is created from `.c` files.
- The `-` in front of `rm` says, “If the action should fail in any way, continue with the next action”

Make Execution Model (6)

```
drafiei@ug20:~/201>make -f Makefile.v2 clean
rm *.o finddw
rm: cannot remove `*.o': No such file or directory
make: [clean] Error 1 (ignored)
drafiei@ug20:~/201>make -f Makefile.v2
gcc -Wall -std=c99 finddw.c -o finddw
drafiei@ug20:~/201>touch README
drafiei@ug20:~/201>make -f Makefile.v2 tar
tar cvf submit.tar finddw.c Makefile README
finddw.c
Makefile
README
```

1. Target `tar` automates the error-prone command to create a tape archive (tar) file.
   - The `cvf` command-line options say: **c**reate a new tar file, be **v**erbose when doing so, and use the **f**ilename that follows <u>immediately</u> after the `cvf` (i.e., `submit.tar`).
   - After the `cvf submit.tar` is the list of files to put into the tar file.
2. Common Pitfall: It is a **big** mistake to do something like: `tar cvf finddw.c Makefile README.`

Make Execution Model (7)

```
drafiei@ug20:~/201>cat Makefile.v3
finddw: finddw.o
        gcc -Wall -std=c99 finddw.o -o finddw

finddw.o: finddw.c
        gcc -Wall -std=c99 -c finddw.c

clean:
        -rm *.o finddw

tar:
        tar cvf submit.tar finddw.c Makefile README
drafiei@ug20:~/201>make -f Makefile.v3 finddw.o
gcc -Wall -std=c99 -c finddw.c
drafiei@ug20:~/201>ls -lt *.o
-rw-------  1 drafiei prof 2020 Aug 30 09:53 finddw.o
drafiei@ug20:~/201>make -f Makefile.v3 finddw
gcc -Wall -std=c99 finddw.o -o finddw
drafiei@ug20:~/201>ls -lt finddw*
-rwx------ 1 drafiei prof 10808 Aug 30 09:55 finddw
-rw------- 1 drafiei prof  1992 Aug 30 09:53 finddw.o
-rw------- 1 drafiei prof    52 Aug 27 16:54 finddw.ok
-rw------- 1 drafiei prof    93 Aug 27 16:53 finddw.test
-rw------- 1 drafiei prof  1504 Aug 27 16:53 finddw_imp.c
-rw------- 1 drafiei prof  1345 Aug 27 16:53 finddw.c
drafiei@ug20:~/201>make -f Makefile.v3 clean
rm *.o finddw
drafiei@ug20:~/201>make -f Makefile.v3
gcc -Wall -std=c99 -c finddw.c
gcc -Wall -std=c99 finddw.o -o finddw
drafiei@ug20:~/201>
```

Make Execution Model (8)

```
drafiei@ug20:~/201>cat Makefile.v3
finddw: finddw.o
        gcc -Wall -std=c99 finddw.o -o finddw

finddw.o: finddw.c
        gcc -Wall -std=c99 -c finddw.c

clean:
        -rm *.o finddw

tar:
        tar cvf submit.tar finddw.c Makefile README
drafiei@ug20:~/201>make -f Makefile.v3 finddw.o
gcc -Wall -std=c99 -c finddw.c
drafiei@ug20:~/201>ls -lt *.o
-rw------- 1 drafiei prof 1992 Aug 30 09:53 finddw.o
drafiei@ug20:~/201>make -f Makefile.v3 finddw
gcc -Wall -std=c99 finddw.o -o finddw
...snip...
```

- The `gcc -Wall -std=c99 -c` illustrates separate compilation in C.

Compiler Warnings

```
drafiei@ug20:~/201>head -11 finddw.c
#include <stdio.h>
#include <string.h>     /* For strncmp(), etc. */

#define MIN_BUF         256
#define MAX_BUF         2048

char Buffer[ MAX_BUF ];
char Word[ MIN_BUF ];
char NextWord[ MIN_BUF ];

void parseFile( FILE * fp, char * fname );
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
first.finddw.c: In function ‘main’:
first.finddw.c:20: warning: implicit declaration of function ‘exit’
first.finddw.c: In function ‘parseFile’:
first.finddw.c:49: warning: implicit declaration of function ‘isalpha’
```

1. As discussed earlier, the `finddw` program will generate compiler warnings.
2. Warnings are not the same, fatal things as errors. But, good programming style and methodology requires that warnings be fixed.

Compiler Warnings (2)

```
drafiei@ug20:~/201>head -2 finddw.c
#include <stdio.h>
#include <string.h>     /* For strncmp(), etc. */
drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
first.finddw.c: In function ‘main’:
first.finddw.c:20: warning: implicit declaration of function ‘exit’
first.finddw.c: In function ‘parseFile’:
first.finddw.c:49: warning: implicit declaration of function ‘isalpha’
drafiei@ug20:~/201>cat -n finddw.c | grep "^ *20"
    23                  exit( -1 );
drafiei@ug20:~/201>cat -n finddw.c | grep "^ *49"
    62                          if( isalpha( Word[ 0 ] ) )
```

1. The man page for `exit()` (i.e., `man 3 exit`) shows that I need `#include <stdlib.h>`.
   - At the end of `man exit`, there is a `SEE ALSO` section that pointed me to `exit(3)`, which is why we need to do `man 3 exit`.
2. The man page for `isalpha()` (i.e., `man isalpha`) shows that I need `#include <ctype.h>`.

Compiler Warnings (3)

```
drafiei@ug20:~/201>head -5 finddw_fixed.c
#include <stdio.h>
#include <string.h>      /* For strncmp(), etc. */
#include <stdlib.h>      /* For exit(), etc. */
#include <ctype.h>       /* For isalpha(), etc. */

drafiei@ug20:~/201>make
gcc -Wall -std=c99 finddw.c -o finddw
```

1. The warnings have now been fixed.
2. Remember, your main tools for this are:
   - Man pages
   - Google
   - Your textbooks
   - Your TAs, instructor, and fellow students.

The Unix Component Model

1. Re-using entire executables is a key element of the so-called Unix component model.
2. If an executable does (nearly) what you want it to do, then simply run it and control it from within another program via standard in (stdin) and standard out (stdout).
3. Your program is the parent process. The other executable (or component) is the child process.
4. The executables can be written in different languages (or not).
5. The parent and child processes run concurrently.

popen

```
drafiei@ug20:~/201>gcc -Wall -std=c99 testpopen.c
drafiei@ug20:~/201>./a.out
hello
drafiei@ug20:~/201>cat testpopen.c
#include <stdio.h>

extern  FILE *popen(const char *command, const char *type);
extern  int pclose(FILE *stream);

int main( int argc, char * argv[] ) {
        FILE * fp;

        fp = popen( "grep hello", "w" );
        if ( fp == NULL ) {
                printf( "Error\n" );
                return  -1;
        }

        fprintf( fp, "greetings\n" );
        fprintf( fp, "hello\n" );
        fprintf( fp, "good bye\n" );
        pclose( fp );

        return 0;
} /* main */
```

popen (2)

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

        fp = popen( "grep hello", "w" );
        if( fp == NULL )
        {
                printf( "Error\n" );
                return  -1;
        }
```

1. `popen` (aka “pipe open”) allows you run a helper, child process to perform a function. There is a corresponding `pclose` function.
2. Unix is a multiprocessing operating system: different processes (i.e., programs being executed) can run concurrently (i.e., at the same time).
3. Unix Component Model: Processes calling/controlling other processes is a powerful abstraction and modularization technique.

popen (3)

```
        fprintf( fp, "greetings\n" );
        fprintf( fp, "hello\n" );
        fprintf( fp, "good bye\n" );
        pclose( fp );

        return 0;
} /* main */
```

1. `popen` returns a `FILE *` (aka file pointer). This is the **same** file pointer seen in the finddw example for open files. You can `fprintf()`, `fscanf()`, etc.
2. In Unix, many different "objects", interfaces, and mechanisms (i.e., a specific abstraction by which to accomplish a task) look like files. Sometimes, the similarity is incomplete; you use `pclose()` instead of `fclose()` here.

popen (4)

Also

1. The same ability to combine or pipeline processes can be done from the command line of the shell. This means one can fearlessly try pipelines before incorporating them in one's program.

```
drafiei@ug20:~/201>cat input.file | grep hello
hello
drafiei@ug20:~/201>./a.out
hello
drafiei@ug20:~/201>cat -n finddw.c | grep "^ *63"
drafiei@ug20:~/201>grep "popen.*grep" testpopen2.c
        fp = popen( "cat -n finddw.c | grep \"^ *63\"", "r" );
```

2. Depending on what you need to have done, there is also the `system()` library function for running child processes.

```
        system( "cat -n finddw.c | grep \"^ *63\"" );
```

3. But, what are the disadvantages of concurrent processes and the “everything is a file” design philosophy?

Read a File Into Memory and Buffer Contents

```
#include <stdio.h>
#include <string.h>              /* For memset() */
#include <stdlib.h>              /* For exit() */

#define MAX_BUF         256      /* Max line size */
#define MAX_LINES       32       /* Max number of lines */

struct aLine            /* A structure for heterogeneous data */
{
        char line[ MAX_BUF ];
        int type;
};
```

1. A `struct` (or structure) is C's mechanism for heterogeneous, aggregate data structures.
2. The closest thing in Java is a class. Structures are like classes with only public data members.
    - Structures and classes are templates and must be instantiated (aka allocated).
    - But, structures do not have associated methods as part of the structure itself.

Structures

```
struct aLine             /* A structure for heterogeneous data */
{
        char line[ MAX_BUF ];
        int type;
}; /* This semicolon is necessary */
```

1. A `struct` ends with a semicolon.
2. One refers to the new `struct` “type” with `struct aLine` (see the upcoming code).
3. A structure can contain any built-in C type (e.g., `int`), or arrays, multi-dimensional arrays, or other structures.
4. Yes, you can have arrays of structures (like arrays of objects).

Read a File Into Memory (2)

```
void bufferFile( FILE * fp, struct aLine * theLines,
                int * currentLinePtr, int maxLines );
void classifyLine( struct aLine * theLines, int currentLine );
void printClass( struct aLine * theLines, int currentLine );

int main( int argc, char * argv[] ) {
        int     i;
        FILE    * fp;
        /* Here, I avoid global variables */
        struct aLine theLines[ MAX_LINES ];     /* An array of structs */
        int     currentLine;

        currentLine = 0;
        memset( theLines, 0, sizeof( theLines ) ); /* Initialize.  Defensive. */
        for ( i = 1; i < argc; i++ ) {
                fp = fopen( argv[ i ], "r" );
                if ( fp == NULL ) {
                        printf( "Could not open file %s\n", argv[ i ] );
                }
                else {
                        /* currentLine is passed-by-reference/pointer */
                        bufferFile( fp, theLines, &currentLine, MAX_LINES );
                        fclose( fp );
                }
        }
        return 0;
} /* main */
```

Read a File Into Memory (3)

```
int main( int argc, char * argv[] )
{
        int     i;
        FILE    * fp;
        /* Here, I avoid global variables */
        struct aLine theLines[ MAX_LINES ];     /* An array of structs */
        int     currentLine;

        currentLine = 0;
        memset( theLines, 0, sizeof( theLines ) ); /* Initialize.  Defensive. */
```

1. Local variables are not initialized automatically. There are no Java-like constructors.
2. Initializing strings is a good defensive programming methodology (simulating what Java does).
3. Once you understand the memory model of C and strings, `memset()` is a quick, easy, and effective way to initialize things.
4. The `sizeof()` operator is also useful for statically allocated data structures, as we see here.

Read a File Into Memory (4)

```
/* currentLine is passed-by-reference/pointer */
bufferFile( fp, theLines, &currentLine, MAX_LINES );
```

The default for parameter passing is pass-by-value (like Java). But, what value?

1. Arrays and strings (which are arrays) always pass the **value of the pointer**. They are implicitly pass-by-reference (aka pass-by-pointer). Pass-by-reference is an important way to get data **out** of a function (aka persistence).
2. For other (smaller) data types, the **value of the variable** is passed. In this function call example, only `MAX_LINES` is pass-by-value.
3. But, the programmer can explicitly choose and use pass-by-reference. The parameter `&currentLine` is also an example of pass-by-reference. The function itself has to be expecting pointers, as necessary.

Read a File Into Memory (5)

```
/* currentLine is passed-by-reference/pointer */
bufferFile( fp, theLines, &currentLine, MAX_LINES );
```

1. In Java, objects are always accessed via references and handles, and changes to an **object** inside a method affect the actual object. This is like pass-by-reference (of objects) in the C world.
2. Changes to the **reference** do not affect the original reference since the reference itself is pass-by-value.
3. So, the way C treats arrays (and strings) is analogous to the way Java treats object references. But, for other types, C gives you a choice.

Read a File Into Memory (6)

```
else
{
        /* currentLine is passed-by-reference/pointer */
        bufferFile( fp, theLines, &currentLine, MAX_LINES );
        fclose( fp );
}
```

1. `fp` is a pointer, despite not having an `&`, because it is <u>already</u> of type `FILE *`. It is already a pointer.
2. `theLines` is a pointer, despite not having an `&`, because it is of type `struct aLine theLines[]`, an array. Thus, `theLines` is of type `struct aLine *`. But, `theLines[ 0 ]` is of type `struct aLine`.
3. `currentLine` is an integer, but we turn it into a pointer (just for this function call site) by using the `&` operator. We'll have lots more to say about C pointers later.
4. Can we use the `&` operator on `MAX_LINES` to get a pointer? No. Why not?

Read a File Into Memory (7)

```
void bufferFile( FILE * fp, struct aLine * theLines, int * currentLinePtr,
                 int maxLines ) {
        char *  rbuf;

        /* Line-oriented */
        while ( ( *currentLinePtr < maxLines ) && !feof( fp ) ) {
                rbuf = fgets( theLines[ *currentLinePtr ].line, MAX_BUF, fp );
                if ( rbuf == NULL )
                        break;
                printf( "Read (%2d):  %s",
                        *currentLinePtr, theLines[ *currentLinePtr ].line );
                classifyLine( theLines, *currentLinePtr );
                printClass( theLines, *currentLinePtr );
                fflush( stdout );

                (*currentLinePtr)++;
        } /* while */

        /* Warn user if we exceed maximum number of lines buffered */
        if ( *currentLinePtr >= maxLines ) {
                printf( "Warning:  Exceeded %d lines of buffering.\n",
                        *currentLinePtr);
        }
} /* bufferFile */
```

Read a File Into Memory (8)

```
void bufferFile( FILE * fp, struct aLine * theLines, int * currentLinePtr,
         int maxLines )
{
...snip...
} /* bufferFile */
```

1. If a parameter is supposed to be a pointer, it always has the form with `*` or `[]`.
   - `fp`, `theLines`, `currentLinePtr`
   - Note that `[]` is just another way to say `*`, which is why `char * argv[]` and `char ** argv` are the same!

Read a File Into Memory (9)

```
/* Line-oriented */
while ( ( *currentLinePtr < maxLines ) && !feof( fp ) ) {
        rbuf = fgets( theLines[ *currentLinePtr ].line, MAX_BUF, fp );
        if ( rbuf == NULL )
                break;
        printf( "Read (%2d):  %s",
                *currentLinePtr, theLines[ *currentLinePtr ].line );
        classifyLine( theLines, *currentLinePtr );
        printClass( theLines, *currentLinePtr );
        fflush( stdout );

        (*currentLinePtr)++;
} /* while */

/* Warn user if we exceed maximum number of lines buffered */
if ( *currentLinePtr >= maxLines ) {
        printf( "Warning:  Exceeded %d lines of buffering.\n",
                *currentLinePtr);
}
```

1. In Java, object references are <u>implicitly</u> dereferenced.
2. In C, pointers are <u>explicitly</u> dereferenced by an operator.
3. Pointers must be dereferenced using either `*`, `[]` (or `->`).

Read a File Into Memory (10)

```
while ( ( *currentLinePtr < maxLines ) && !feof( fp ) ) {
        rbuf = fgets( theLines[ *currentLinePtr ].line, MAX_BUF, fp );
        if ( rbuf == NULL )
                break;
        printf( "Read (%2d):  %s",
                *currentLinePtr, theLines[ *currentLinePtr ].line );
        classifyLine( theLines, *currentLinePtr );
        printClass( theLines, *currentLinePtr );
        fflush( stdout );

        (*currentLinePtr)++;
} /* while */
```

1. Pointers are a source of great programming power and great programming errors.
2. The compiler can do simple type checking.

```
drafiei@ug20:~/201>grep 'while.*currentLinePtr' bufferfile.c
        while( ( currentLinePtr < maxLines ) && !feof( fp ) )
drafiei@ug20:~/201>make
gcc -Wall -std=c99 bufferfile.c
bufferfile.c:47: warning: comparison between pointer and integer
bufferfile.c: In function `classifyLine':
bufferfile.c:70: warning: unused variable 'len'
```

3. This is only a warning. The executable is still created!

Read a File Into Memory (11)

```
/* In main() */
                        /* currentLine is passed-by-reference/pointer */
                         bufferFile( fp, theLines, &currentLine, MAX_LINES );
...snip...
        while( ( *currentLinePtr < maxLines ) && !feof( fp ) )
        {
                rbuf = fgets( theLines[ *currentLinePtr ].line, MAX_BUF, fp );
                if( rbuf == NULL )
                        break;
                printf( "Read (%2d):  %s",
                        *currentLinePtr, theLines[ *currentLinePtr ].line );
                classifyLine( theLines, *currentLinePtr );
                printClass( theLines, *currentLinePtr );
                fflush( stdout );

                (*currentLinePtr)++;
        } /* while */
```

1. `*currentLinePtr`: The value of `currentLine` in `main()`
2. `theLines[ *currentLinePtr ]`: The array `theLines` in `main()`, at index `currentLine` in `main()`

Read a File Into Memory (12)

```
while( ( *currentLinePtr < maxLines ) && !feof( fp ) )
{
        rbuf = fgets( theLines[ *currentLinePtr ].line, MAX_BUF, fp );
        if( rbuf == NULL )
                break;
        printf( "Read (%2d):  %s",
                *currentLinePtr, theLines[ *currentLinePtr ].line );
        classifyLine( theLines, *currentLinePtr );
        printClass( theLines, *currentLinePtr );
        fflush( stdout );

        (*currentLinePtr)++;
} /* while */
```

1. Dot operator: `theLines[ *currentLinePtr ].line`: A field `line` in array `theLines` in `main()`, at index `currentLine` in `main()`.
2. `fflush( stdout );` forces the output of `printf()` to standard out.
3. `(*currentLinePtr)++;` increments the value of `currentLine` in `main()`. The dereferencing happens before the increment.

Read a File Into Memory (13)

```
#define KEY_MATRIX     "Matrix"
void classifyLine( struct aLine * theLines, int currentLine )
{
        int     rval, len;
        char    firstWord[ MAX_BUF ];

        theLines[ currentLine ].type = -1;              /* Default */

        /* Check for comments */
        if( theLines[ currentLine ].line[ 0 ]  == '#' )
        {
                theLines[ currentLine ].type = 0;
                return;
        }

        /* Check for matrix definitions */
        memset( firstWord, 0, MAX_BUF );        /* Initialize.  Defensive. */
        rval = sscanf( theLines[ currentLine ].line, "%s", firstWord );
        if( rval != 1 )
        {
#if 0
                printf( "Error in sscanf\n" );
#endif
                return;
        }
        if( strncmp( firstWord, KEY_MATRIX, MAX_BUF ) == 0 )
        {
                theLines[ currentLine].type = 1;
        }
} /* classifyLine */
```

Read a File Into Memory (14)

```
void classifyLine( struct aLine * theLines, int currentLine )
{
        int     rval;
        char    firstWord[ MAX_BUF ];

        theLines[ currentLine ].type = -1;              /* Default */

        /* Check for comments */
        if( theLines[ currentLine ].line[ 0 ]  == '#' )
        {
                theLines[ currentLine ].type = 0;
                return;
        }
```

1. Having a default value of -1 for `type` is a good, defensive programming (best practice).
2. `theLines[ currentLine ].line[0] == '#'`
   - Refers to the first character of field `line` of the structure at index `currentLine`, in the array `theLines` in `main()`
   - `'#'` (single quotes) is a character constant, not a string.
   - == can be used to compare characters, but not strings
3. `return` returns in the middle of the function

Read a File Into Memory (15)

```
#define KEY_MATRIX     "Matrix"

        /* Check for matrix definitions */
        memset( firstWord, 0, MAX_BUF );        /* Initialize.  Defensive. */
        rval = sscanf( theLines[ currentLine ].line, "%s", firstWord );
        if( rval != 1 )
        {
#if 0
                printf( "Error in sscanf\n" );
#endif
                return;
        }
        if( strncmp( firstWord, KEY_MATRIX, MAX_BUF ) == 0 )
        {
                theLines[ currentLine].type = 1;
        }
```

1. Sometimes zero'ing a string (i.e., `firstWord`) is a good, defensive programming.
2. `sscanf()` allows parsing from a string buffer.
   - `scanf()` gets its input from standard in
   - `fscanf()` gets its input from a file, via a file pointer
3. We use `sscanf()` to automatically handle white space issues

Read a File Into Memory (16)

```
#define KEY_MATRIX     "Matrix"

        /* Check for matrix definitions */
        memset( firstWord, 0, MAX_BUF );        /* Initialize.  Defensive. */
        rval = sscanf( theLines[ currentLine ].line, "%s", firstWord );
        if( rval != 1 )
        {
#if 0
                printf( "Error in sscanf\n" );
#endif
                return;
        }
        if( strncmp( firstWord, KEY_MATRIX, MAX_BUF ) == 0 )
        {
                theLines[ currentLine].type = 1;
        }
```

1. `#if 0` and `#endif` are examples of conditional compilation. Here, it means do not include the `printf()` line. Use this technique for your own debugging output; see assignment description.
2. Note the use of macro `KEY_MATRIX` to be more "general".

Read a File Into Memory (17)

```
void printClass( struct aLine * theLines, int currentLine )
{
        switch( theLines[ currentLine ].type )
        {
            case -1:
                printf( "\tUnknown\n" );
                break;
            case 0:
                printf( "\tComment\n" );
                break;
            case 1:
                printf( "\tMatrix Definition\n" );
                break;
            default:
                printf( "\tUndefined\n" );
                break;
        }
} /* printClass */
```

Best practices:

1. Always have a `default:` label.
2. Make sure you have a `break:` at the end of each `case`, unless you really, really mean it. And, then add a comment.