Block 10

File handing.

The EXAMPLES IN BLOCK 10:

streamtxtwrt.cpp write a text file with an ofstream

streamtxtre.cp read a text file with an ifstream

fgetc01.c.txt C standard library text i/o: read character

fgets01.c.txt C standard library text i/o: read word

fputs.01.c.txt C standard library text i/o: write word

fscanf01.c.txt C standard library text i/o: versatile scanf()

I've reproduced the lecture from block08 here, because if we read it

two or three times, we'll have a better understanding of it, and it's

really four separate subjects:

System Level

C Standard Library Binary

C Standard Library Formatted

C++ Stream Objects

You'll be looking this stuff up in for the next ten years.

HOW we create and access files doesn't effect the nature of the files

themselves. There are two kinds of files: text and binary. (In fact,

all files are binary, "text" is how we refer to files which only contain

printible characters and space, no control codes.)

Files are streams of bytes.

They have no internal structure. That's all the job of the programmer.

Big systems like IBM mainframes store a lot of structure and organization

data on the disk volumes. This makes them slightly faster, but much

more complicated to program.

All C/C++ programs may be written as if they are running on Unix servers.

If the host operating system (Windows, say) doesn't support some of

the calls, they just get ignored. Unix stores a great deal of

information about files.

inode (integer number)

three separate dates

owner ID

group ID

number of links (names) of the file

size

three sets of permissions for owner, group, and public

13 separate attributes the chattr/lsattr commands manipulate

Windows stores name, number, one date, and four attributes.

Sometimes it stores one owner ID.

Files can be of the following kinds:

- regular file (data, executable, etc.)

d directory

l symbolic link (another name for a file)

p FIFO pipe (for interprocess communication)

c character special file (mouse, keyboard, searial port)

b block special file (disk, tape)

Right now, we're only interested in '-' files (regular

files sitting on the disk.)

Each file has three groups associated with it:

u user g group o other

They each have separate permissions:

U G O

SGT RWX RWX RWX

Don't worry about "SetUserID, SetGroupID, sTickeyBit for now.)

U,G and O all have Read, Write, and Execute bits.

They correspond to the values of their bits: 4,2,1.

So to set U'ser's Read and Write, Group's Read, and Others' Read,

we'd say:

chmod 644 filename

That would result in:

U G O

SGT RWX RWX RWX

110 100 000

Value: 6 4 0

The whole point of this, is, if you don't have the right

permissions, you can't read or write or erase a file.

Same goes for directories: if you don't have "write" rights

on a directory, you can't save your files there.

You have total rights on your home directories on Windows or

Unix, but you are limited in what you can do in the places

where the system files are kept. Just try to keep track of

where you are. If you don't know, run: pwd (print working

directory).

If you get lost, and want to go home (sniff) just run:

cd <enter>

Change Directory without a destination will take you home.

If you want to see what information the system stores about

a file, run stat on it.

stat caesar1.cpp

File: `/var2/home/hhaller/192/asst3/.st/caesar1.cpp'

Size: 910 Blocks: 8 IO Block: 4096 regular file

Device: 805h/2053d Inode: 37620216 Links: 1

Access: (0600/-rw-------) Uid: ( 2642/ hhaller) Gid: ( 2000/ staff)

Access: 2014-07-03 19:10:20.928016515 -0700

Modify: 2014-06-30 09:51:04.113510370 -0700

Change: 2014-07-02 23:48:51.114801247 -0700

Birth: -

See? Three different dates!

If you looked at the same file with the 'l' command:

-rw------- 1 hhaller staff 910 Jun 30 09:51 ../asst3/.st/caesar1.cpp

There are many command line tools for dealing with files:

ls type cat tac touch mv rm cp ln (ln -s) more less ws

Each of these has a man page.

---------------------------------------------------e

File I/O, and the File System.

SUMMARY

READ CHAPTER File Processing

There are:

C Standard Library Formatted (Text). Simple text-based.

C Standard Library Unformatted (Binary). Well-known, precise.

(You create and use FILE Pointers, to data structures.)

System Level. Assembly-language level, no real gain in use.

(You create and use integer file handles.)

C++ Streams ("easy", big, clumsy, labor-intensive, overly formalistic).

(You use objects, after learning all the methods they contain.)

MOST FILE I/O PROGRAMMING USES C METHODS, NOT C++.

MOST C FILE PROGRAMMING USES C STANDARD LIBRARY, NOT SYSTEM LEVEL.

MOST C INSTRUCTORS ONLY TEACH STUDENTS HOW TO READ AND WRITE SINGLE

FILES, AND DON'T COVER OPERATIONS ON THE FILE SYSTEM (DIRECTORIES).

WE WILL START THAT WAY, BUT I'LL ADD SOME EXAMPLES USING readdir()

and ftw() SO YOU HAVE THEM IN YOUR NOTES.

BASIC PRINCIPLE WHEN USING BIG, COMPLEX SYSTEMS OR THEORIES:

LEARN WHAT YOU NEED TO USE WHEN YOU NEED TO USE IT.

FOR NOW UNDERSTAND fopen() fclose() fseek() fprintf() fscanf()

fgets() fgetc() fputc() printf() ARE C STANDARD LIBRARY FORMATTED

AND fread() fwrite() ARE C STANDARD LIBRARY UNFORMATTED (BINARY).

END SUMMARY

After years of struggling to see it their way, I am cast

back upon my first impression: the file handling in C++ is a

bureaucratic kludge compounded by a bunch of centrally-heated,

shoe-wearing, city-slickin' bureaucratic white-paper publishing

theorists who get upset when people exercise their freedom

to do things the simplest and most reliable way.

Streams <iostream> are sold as being "type safe", which is fine

if the programmer can't remember the types s/he defined hir

variables to be, but generally, we WANT to "cheat" when doing

file I/O, and ignore type, because we want it to GET DONE,

not sit and grind the disk for minutes on end. Stream I/O

should be used ONLY when it's manifestly the EASIEST way.

Like for handling primitive data types without a lot of

file opening.

Otherwise, use C Standard Library techniques.

Since that comprises most of the code on the planet anyway,

it is a good thing to know well.

This study chapter is divided into two parts:

I. THE QUICK AND DIRTY HOW-TO.

and

II. THE THEORY, IN GREAT DETAIL AND COMPLETENESS.

My opinion: for this course, section I is what you

need, so work on that. After the semester is over,

you can read and re-read every aspect of file i/o,

and you will be a better programmer for doing so.

But for now, I'd focus on just what I needed.

-------------------

PART I. STREAMS QUICK AND DIRTY HOW-TO.

C++ Bare Minimum:

Read: Chapter 17 File I/O Streams.

There is a wealth of options in C++ for reading from

or writing to disk files.

I. std::fstream objects and methods

Generally, it's easy to "create" a stream to read a

given file. You do this by passing arguments to the

stream objects "constructor" like so:

#include iostream

#include fstream

std::ofstream ofile("struct.dat", std::ios::binary | std::ios::app);

(open it to be able to write binary data, and

to APPEND (not overwrite) the file if it's

already there.)

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/iostream\_one">iostream\_one, a review

of cin and cout</A>

Read: <A

HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/ifstream.ofstream.txt">ifstream.ofstream.txt,

how to use streams to read and write text files</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/ostream1.cpp.txt">ostream1.cpp, how to use

stream methods to open, write, read, close files</A>

All ofstream, ifstream, and fstream objects have the

.eof(), .get() and .put() methods we have been using

with cin and cout.

--

NOTE: And that is all the file processing you need

to pass this course.

OK? Calm down. Deep breathely.

-----------------------------

Now, for completeness in your notes, here is the

larger picture:

WHEN WRITING C/C++ PROGRAMS, YOU MAY SAFELY PRETEND

TO BE RUNNING UNDER UNIX, EVEN IF YOU'RE ON A WINDOWS

BOX, BECAUSE THE STANDARD LIBRARY IS WRITTEN THAT

WAY, AND THE ISO STANDARD REQUIRES THAT C/C++

METHODS BE VIABLE ON ALL OPERATING SYSTEMS.

So...even if you're running on Windows, your program

can act like it's on the Linux ext2 filesystem, or

ext3, or ReiserFS, or SYSVfs, or.... You get the idea.

C style: all file access is via a FILE \* pointer.

You create one with fopen().

FILE \*fp = fopen("my.txt", "r") ;

if (0 == fp)

exit(1) ;

char fname[25], lname[25] ;

fscanf(fp, "%s", fname) ;

fscanf(fp, "%s", lname) ;

// (also: fgets() , fgetc(), etc.)

fclose(fp) ;

Create pointer.

Check to see if it opened successfully.

Use it.

Close it.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/cpp\_fileio">cpp\_fileio</A> Text: Chapter 17

C STANDARD LIBRARY FORMATTED: ("text")

Simple example: fgets and fprintf

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/fgets.cpp.txt">fgets.cpp</A>

Simple example: fgetc single-char i/o.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/fgetc.cpp.txt">fgetc.cpp</A>

C STANDARD LIBRARY UNFORMATTED: ("binary")

Binary file i/o:

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/structfile.cpp.txt">structfile.cpp</A>

SYSTEM LEVEL: (also "binary")

The C Standard Library commands

(fopen, fclose, fseek, fprintf, fscanf, fgets,

fputs, fgetc, fputc, ftell)

either go away of get shorter:

open, close, read, write, lseek, creat.

System level is unbuffered, so it's faster

to write to disk, but it makes your program sun

slower waiting for the disk operation to complete.

Only old programmers with something to prove

insist on using these tools nowadays.

Unless you absolutely have to use them (you're

maintaining old code) you can leave them alone.

System Level I/O

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/rdwt.c.txt">rdwt.c</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/rawio.c.txt">rawio.c</A>

The C++ analogues for C Std Lib I/O are "Stream I/O":

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/ofstream.members">ofstream.members</A> Text: Chapter 18

-------------------

C-Style:

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/diskfiles">Disk Files, permissions and modes.</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/c\_fileio">c\_fileio</A> Ref: Chapter 26

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/c\_fileio.extra">c\_fileio.extra</A> Ref: Chapter 26

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/frdfwt.c.txt">frdfwt.c</A> Ref: Chapter 26

C++Style:

Look at the tables, and see how some of the objects

work.

Look at my examples, compile and run them, then try

changing them and experimenting.

There are two types of files: text and binary.

Text files are a "translation" of what was stored

in memory, (text or numbers) into "human readable"

ASCII or UNICODE text which can be edited with Notepad

or a similar editor program.

Binary files are an untranslated "snapshot" of the contents

of memory.

Data Base Management Systems always save their

information in Binary files because to do so is much

faster, both to save and to read back in.

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/structstream.cpp.txt">structstream.cpp,

how to use stream methods to open, write, read,

close files</A>

As an example, if you wanted to write a program which opened

and read a file whose name you placed in the first command-line

argument, you would do something like:

int main (int argc, char \* argv[])

{

string s ;

std::istream infile(argv[1] ) ;

if ( !infile.good()) exit (-1) ;

do

{

infile >> s ;

// some processing here

} while ( ! infile.eof()) ;

infile.close() ;

Despite the fstream libraries, a lot of the file

access even in modern code uses the C Standard

Library functions.

Now, a word from our sponsor:

DON'T PANIC!!

Yes, file processing is huge. No, you don't

need to know it all right now. What I want you

to be able to do is to open, read from and

write to a text file without a great deal of

grief. Doing random-access file processing on

a binary data file is good to know, but it is

becoming less and less important in this day

of database management software.

OK?

NOW: THE WINDOWS SIDE:

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/windows\_filesystem">windows\_filesystem</A>

Read: <A HREF="http://209.129.16.61/~hhaller/data/cisc192/modules/doswindow.commands">doswindow.commands</A>

<A HREF="syllabus.html#Menu">Return to Menu</A>

/\*-----------------------------------------------------------------

\* fgetc01.c - C Std Lib I/O formatted (text)

\* print a file [file name on command line]

\* OR: file redirected with "<"

-----------------------------------------------------------------\*/

#include <stdio.h>

int main (int argc, char \* argv[])

{

FILE \* fp ;

char buf[1024] ;

int i , c ;

char s[81] ;

fp = fopen(argv[1], "r") ;

if (!fp) fp = stdin ;

while (!feof(fp))

{

c = fgetc(fp) ;

putchar(c) ;

}

fclose(fp) ;

} /\* main ends \*/

/\*-----------------------------------------------------------------

\* fputs01.c - C Std Lib I/O formatted (text)

\* create a file

-----------------------------------------------------------------\*/

#include <stdio.h>

int main (int argc, char \* argv[])

{

FILE \* fp ;

char buf[1024] ;

int i , c , array[500] ;

fp = fopen(argv[1], "w") ;

if (!fp)

{

fprintf(stderr,"Cannot open %s: program terminated.\n", argv[1]) ;

return -1 ;

}

for (i = 0 ; i < 500 ; ++i)

{

fprintf(fp, "array[%03d]: %03d\n", i, array[i]) ;

}

fclose(fp) ;

} /\* main ends \*/

/\* csdfmt1.c - C Std Lib I/O \*/

#include <stdio.h>

int main (int argc, char \* argv[])

{

FILE \* fp ;

char buf[1024] ;

int i , c ;

char s[81] ;

fp = fopen(argv[1], "r") ;

if (!fp) fp = stdin ;

while (!feof(fp))

{

fscanf(fp, "%s",buf) ;

fprintf(stdout, "%s ", buf) ;

}

fclose(fp) ;

} /\* main ends \*/

/\*-----------------------------------------------------------------

\* manyfiles.c - demonstrate an array of FILE \* pointers,

\* dynalloc of a \*[] data object.

\* TEK.

-----------------------------------------------------------------\*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <ctype.h>

#include <stdio.h>

#define SIZE 10000

int main (int argc, char \*argv[], char \*\*env)

{

FILE \*\* fpa ;

int c ;

char s[81] ;

for (c = 0 ; c < SIZE ; c++)

{

fpa[c] = malloc (sizeof (FILE \*)) ;

if (fpa[c] == NULL)

{

perror("can't allocate!\n") ;

exit(-1) ;

}

sprintf(s, "./file.%03d", c) ;

fpa[c] = fopen(s, "w") ;

if (NULL == fpa[c])

{

perror("can't open!\n") ;

exit(-1) ;

}

printf("%05X\n", fpa[c]) ;

fprintf(fpa[c], "This is file %s\n", s) ;

}

for (c = 0 ; c < SIZE ; c++)

fclose(fpa[c]) ;

}

C was created to write Operating Systems. It is a portable form of Assembly Language.

Because Unix was the first Operating System not tailored for a single computer hardware

family, it was possible to port one system to every major brand and model of

computer. The students at UC Berkeley modified Unix to add robust networking, and

returned the new code to AT&T Bell Labs, Murray Hill, NJ, where Unix had

originally been created in 1969.

A powerful operating system with networking which could be ported to another

computer in less than three months changed the course of history. If there

were no Unix, there would be no internet.

So the C language was directly responsible for more social change than the private

passenger car.

In beginning programming, simple and useless applications are developed, because the

students have no grounding or understanding of any of the scientific, technical,

or social contexts in which computers operate. But in this course, we walk partway

along the path of the original architects of Unix.

Later we will consider the baroque and complex way that floating point data can

be stored on computers. For now, we'll just consider what we might have to

go through if we wanted to ask the user to type in a CString which contained

a floating point value, and as they typed, we built the number, while checking

for illogical and improper keystrokes, but accepting proper ones.

This is almost "system" programming, and it in fact uses low-level calls, because

we must craft our own form of input, since the C Standard Library doesn't contain

any way of getting keystrokes instantly from a terminal without echoing the

keystrokes. If you study this file until you understand it, you will have made

real progress in understanding real programming.

Here is "scanfloat": which controls the terminal while conditioning

input and building non-integral data.

Understand: when reading in a float, the following characters are valid:

"+-.0123456789\n".

BUT: they aren't all valid all the time. Understand the different

phases the data entry goes through, and which keys are assumed to have

been pressed even if they have not bee. Think on that.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* SCANFLOAT.C: Get a floating-point number character by character from

\* the keyboard. Process leading + or - and leading or embedded decimal

\* points. Reject inappropriate characters. Return a floating point

\* number to the calling program

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include "/var2/local/include/getchne.h"

float scanfloat(void) { /\* fetch a float num char by char \*/

int dpoint = 0, /\* number decimal digits in num \*/

sign = 0; /\* pos or neg num \*/

float temp = 0.0; /\* running total of num \*/

char c = ' '; /\* input char \*/

while ((c = (char) getchne()) != '\r') /\* get char with no screen echo \*/

{

if (((c == '-') || (c == '+')) && /\* if - or + entered \*/

(sign == 0)) /\* and no sign entered yet \*/

{ /\* and numeric entry hasn't begun \*/

if (c == '-') /\* if negative sign entered \*/

sign = -1; /\* negative number flag \*/

else /\* otherwise \*/

sign = 1; /\* positive number flag \*/

(void) putchar(c); /\* echo character to screen \*/

continue; /\* get next character \*/

} /\* if (((c == '-')|| (c \*/

if ((c == '.') && (dpoint == 0)) { /\* if first decimal point entered \*/

dpoint = 1; /\* start counting decimal places \*/

(void) putchar(c); /\* echo character to screen \*/

continue; /\* get next character \*/

} /\* if ((c == '.') && (dpo \*/

if (isdigit(c)) { /\* if it's a number \*/

if (sign == 0) sign = 1; /\* assume positive number \*/

(void) putchar(c); /\* echo character to screen \*/

temp \*= 10.0; /\* shift number right one space \*/

temp += (float)(c - '0'); /\* add new digit \*/

if (dpoint != 0) dpoint++; /\* keep track of decimal places \*/

} /\* if (isdigit(c) \*/

else printf("\x7"); /\* bad character, ring bell \*/

} /\* while (( c = getch()) \*/

/\* we have now received a '\r', so correct the number for decimal & sign \*/

if (0 != dpoint) /\* if a decimal was entered... \*/

while (0 != --dpoint) /\* shift number to the right \*/

temp /= 10.0; /\* dpoint number of places \*/

if (sign == -1) /\* if a minus sign was entered \*/

temp =(float) 0.0 - temp; /\* convert to negative \*/

(void) putchar('\n'); /\* jump to new line \*/

return temp;

} /\* scanfloat() \*/

int main()

{

puts("Enter a floating point number: " ) ;

double d ;

d = scanfloat() ;

printf("that was: %8.3f\n", d) ;

}

/\*-----------------------------------------------------------------

\* streamtxtrd.cpp - read a text stream with ofstream

\* C++ streams are for untrained dweebs, usually

Try replacing c = i.get() with i >> c ;

What happens?

-----------------------------------------------------------------\*/

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <cstdlib>

#include <fstream>

using namespace std ;

int main (int argc, char \*argv[], char \*\*env)

{

char c ;

ifstream i;

i.open (argv[1], std::ifstream::in | std::ifstream::in);

if (i.bad()) return -1 ;

c = i.get() ;

while (!i.eof())

{

cout << c ;

c = i.get() ;

}

i.close() ;

return EXIT\_SUCCESS ;

} // main ends

/\*

open (filename, mode);

filename is a string name of file, and mode is an optional parameter with a combination of the following flags:

ios::in Open for input operations.

ios::binary Open in binary mode.

ios::ate Set the initial position at the end of the file.

If this flag is not set, the initial position is the beginning of the file.

ios::app All output operations are performed at the end of the file, appending the content to the current content of the file.

ios::trunc If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

\*/

/\*-----------------------------------------------------------------

\* streamtxtwrt.cpp - write a text stream with ofstream

\* C++ streams are for untrained dweebs, usually

-----------------------------------------------------------------\*/

#include <cstdio>

#include <cctype>

#include <iostream>

#include <iomanip>

#include <cstdlib>

#include <fstream>

using namespace std ;

int main (int argc, char \*argv[], char \*\*env)

{

ofstream o;

o.open (argv[1], std::ofstream::out | std::ofstream::app);

if (o.bad()) return -1 ;

for (int i = 0 ; i < 50000 ; ++i)

o << i << "\t" ;

o.close() ;

return EXIT\_SUCCESS ;

} // main ends

/\*

member constant stands for access

out \* output File open for writing: the internal stream buffer supports output operations.

binary binary Operations are performed in binary mode rather than text.

ate at end The output position starts at the end of the file.

app append All output operations happen at the end of the file, appending to its existing contents.

trunc truncate Any contents that existed in the file before it is open are discarded.

\*/

/\*-----------------------------------------------------------

\* varargs1.c -writing funcs with variable number of arguments.

\* Written Sun May 21 20:49:52 PDT 2000

\* by TEK.

-----------------------------------------------------------\*/

#include <stdio.h>

#include <stdlib.h>

#include <stdarg.h>

int sum(int , ...) ;

/\*----------

\* I'm gonna call sum() with 9 ints...

-----------\*/

void main()

{

printf("%d\n", sum(9, 1,2,3,4,5,6,7,8,9) ) ;

}

/\*-----------

\* I define the function with an elipsis: ("...").

-----------\*/

int sum(int x, ...)

{

va\_list ap ;

int c, sum = 0 , i = 0 ;

/\*-----------------

make ap point to the first UNNAMED argument by

providing the name of the LAST NAMED argument

-----------------\*/

va\_start( ap, x ) ;

for (i = 0 ; i < x ; i++ )

{

/\*

\* each call to va\_arg must specify WHERE to put the

\* result, and the TYPE of the thing to grab, so the

\* compiler can calculate how many bytes it has to

\* retrieve. SOME programs like this won't use a

\* for() loop, since all args won't be homogenious,

\* instead, the code would process each arg

\* explicitly....

\*/

c = (int) va\_arg (ap, int ) ;

printf("sum: %d c == %d\n", sum, c) ;

sum += c ; // the actual working func code...

}

va\_end (ap) ;

return sum ;

} /\* sum() ends \*/