Block 08

Two important things get introduced here: file i/o and classes. You'll do

a lot more file i/o in the early days than class manipulation, so let's start there.

Block 8: 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>

CLASSES

"DON'T PANIC" DISCLAIMER:

Classes are for big, complex progams. Like all "secret weapons", they

are an article of faith, and a lot of patriarchs preach dogma about how

they're going to make programming easier and more reliable. They don't.

Only good programmers produce good programs. Some tasks are appropriate

for classes, and we must understand and be able to use them.

Understand this: classes don't make things easier. If you need them,

they make the impossible possible, not easy.

They are necessary for programmng Windows or MACOS, because GUIs are

that complex. Some languages like Java can only be programmed using them.

You will be expected to know them.

In order to learn them, you will write essentially trivial programs

which could be shorter without them. That's ok, think of this as doing

ten thousand side snap kicks to the head before you're allowed to test

for your orange belt. It's just dues we pay.

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

Review:

Each program has a \*.cpp file containing main().

Each program #includes <solme files> necessary to compile it.

Some programs have other \*.cpp files.

Some programs have other \*.h files.

Example:

We want to write a tool that will reverse the case of alpha

characters, and count upper and lower letters, digits, punctuation,

and space.

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

\* FlipCase.cpp - impliment the methods of a FlipCase object

The simplest way to do an Object program: all in one \*.cpp file.

Textbooks perfer to declare in a \*.h,

impliment in a \*.cpp

and call from another \*.cpp.

We'll get to it.

First, let's understand the class' basics.

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

#include <iostream>

using namespace std ;

class FlipCase

{

int uppercount ;

int lowercount ;

int spacecount ;

int punctcount ;

int digitcount ;

public:

FlipCase() // constructor

{

this->uppercount = 0 ;

this->lowercount = 0 ;

this->spacecount = 0 ;

this->punctcount = 0 ;

this->digitcount = 0 ;

cout << "FlipCase Constructor running....\n" ;

}

~FlipCase() // destructor

{

cout << "lowercount : " << this->lowercount << endl ;

cout << "uppercount: " << this->uppercount << endl ;

cout << "spacecount: " << this->spacecount << endl ;

cout << "punctcount: " << this->punctcount << endl ;

cout << "digitcount: " << this->digitcount << endl ;

}

char flip(char c)

{

if (isupper(c))

{

this-> uppercount++ ;

c = tolower(c) ;

}

if (islower(c)) // test both, (it may not be alpha at all)

{

this->lowercount++ ;

c = toupper(c) ;

}

if (isspace(c)) // test both, (it may not be alpha at all)

{

this->spacecount++ ;

c = toupper(c) ;

}

if (ispunct(c)) // test both, (it may not be alpha at all)

{

this->punctcount++ ;

c = toupper(c) ;

}

if (isdigit(c)) // test both, (it may not be alpha at all)

{

this->digitcount++ ;

c = toupper(c) ;

}

return c ;

} // flip

} ; // class FlipCase ends

int main()

{

FlipCase f ;

int c = cin.get() ;

while (!cin.eof())

{

cout.put(f.flip(c)) ;

c = cin.get() ;

}

f.~FlipCase() ;

cout << "I just called the destructor!\n" ;

cout << "But it didn't destroy the object!\n" ;

}

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

\* FlipCase.cpp - impliment the methods of a FlipCase object

The simplest way to do an Object program: all in one \*.cpp file.

Textbooks perfer to declare in a \*.h,

impliment in a \*.cpp

and call from another \*.cpp.

We'll get to it.

First, let's understand the class' basics.

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

#include <iostream>

using namespace std ;

class FlipCase

{

int uppercount ;

int lowercount ;

int spacecount ;

int punctcount ;

int digitcount ;

public:

FlipCase() // constructor

{

this->uppercount = 0 ;

this->lowercount = 0 ;

this->spacecount = 0 ;

this->punctcount = 0 ;

this->digitcount = 0 ;

cout << "FlipCase Constructor running....\n" ;

}

~FlipCase() // destructor

{

cout << "lowercount : " << this->lowercount << endl ;

cout << "uppercount: " << this->uppercount << endl ;

cout << "spacecount: " << this->spacecount << endl ;

cout << "punctcount: " << this->punctcount << endl ;

cout << "digitcount: " << this->digitcount << endl ;

}

char flip(char c)

{

if (isupper(c))

{

this-> uppercount++ ;

c = tolower(c) ;

}

if (islower(c)) // test both, (it may not be alpha at all)

{

this->lowercount++ ;

c = toupper(c) ;

}

if (isspace(c)) // test both, (it may not be alpha at all)

{

this->spacecount++ ;

c = toupper(c) ;

}

if (ispunct(c)) // test both, (it may not be alpha at all)

{

this->punctcount++ ;

c = toupper(c) ;

}

if (isdigit(c)) // test both, (it may not be alpha at all)

{

this->digitcount++ ;

c = toupper(c) ;

}

return c ;

} // flip

} ; // class FlipCase ends

int main()

{

FlipCase f ;

int c = cin.get() ;

while (!cin.eof())

{

cout.put(f.flip(c)) ;

c = cin.get() ;

}

f.~FlipCase() ;

cout << "I just called the destructor!\n" ;

cout << "But it didn't destroy the object!\n" ;

}

FileIO C STD LIB Random Access

The C Standard Library Unformatted (Binary) File Access methods

use

FILE \*

fopen()

fseek()

fread()

fwrite()

fclose()

Raw data is written and read between computer memory and disk.

It's fast, since there are no translations.

This illustrates random-access file i/o, which means, you

can go to any place in the file to get or put data.

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

rndtst.c - random access input and output from disk

This code illustrates that a file may be written as well as read

in ramdom order. The C techniques for laying down file records are

extremely flexible, and uniquely suited to database applications.

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

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <ctype.h>

#include <stdio.h>

main()

{

struct one

{

char line[20] ;

unsigned int num ;

} array[5] = { "this will never",0,

"come to pass",1,

"a back seat driver",2,

"out of gas",3,

"Burma-Shave!",4,} ;

FILE \*fp ;

unsigned int x, filepos[] = {3,1,4,2,0} ; /\* order to write items in \*/

if ((fp = fopen("test.dat","w")) == NULL)

{

puts("Error opening test.dat for write!") ;

exit(1) ;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*WRITE THE RECORDS IN THE ORDER OF filepos[]\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

for ( x = 0 ; x < 5 ; x++)

{

if (fseek(fp,(long) (sizeof(struct one) \* filepos[x]), 0) == -1L)

{

puts("Seek failed!") ;

exit(1) ;

}

fwrite(&array[x], sizeof(struct one), 1, fp) ;

} /\* for \*/

fclose(fp) ;

/\*\*\*\*\*\*\*\*\*DUMP THE DISKFILE IN THE ORDER IT APPEARS ON DISK\*\*\*\*\*\*\*\*\*\*\*\*\*/

if ((fp = fopen("test.dat","r")) == NULL)

{

puts("Error opening test.dat for read!") ;

exit(1) ;

}

puts("Order it occurs on disk file: ") ;

for ( x = 0 ; x < 5 ; x++)

{

if (fseek(fp,(long) (sizeof(struct one) \* x), 0) == -1L)

{

puts("Seek failed!") ; exit(1) ;

} /\* if \*/

fread(&array[x], sizeof(struct one), 1, fp) ;

printf("%u %s\n",array[x].num, array[x].line) ;

} /\* for \*/

/\*\*\*\*\*\*\*SEEK AND PRINT THE RECORDS IN THE ORDER THEY WERE WRITTEN\*\*\*\*\*\*\*\*/

puts("\n\nBut fetching it in random order: \n") ;

for ( x = 0 ; x < 5 ; x++)

{

if (fseek(fp,(long) (sizeof(struct one) \* filepos[x]), 0) == -1L)

{

puts("Seek failed!") ;

exit(1) ;

} /\* if \*/

fread(&array[x], sizeof(struct one), 1, fp) ;

printf("%u %s\n",array[x].num, array[x].line) ;

} /\* for \*/

getchar() ;

} /\* main \*/

/\* $ hexdump test.dat

HEXDUMP Written in Turbo C by T. E. Harrisburg.

00000000: 42 75 72 6D 61 2D 53 68 - 61 76 65 21 00 00 00 00 - Burma-Shave!....

00000010: 00 00 00 00 04 00 00 00 - 63 6F 6D 65 20 74 6F 20 - ........come.to.

00000020: 70 61 73 73 00 00 00 00 - 00 00 00 00 01 00 00 00 - pass............

00000030: 6F 75 74 20 6F 66 20 67 - 61 73 00 00 00 00 00 00 - out.of.gas......

00000040: 00 00 00 00 03 00 00 00 - 74 68 69 73 20 77 69 6C - ........this.wil

00000050: 6C 20 6E 65 76 65 72 00 - 00 00 00 00 00 00 00 00 - l.never.........

00000060: 61 20 62 61 63 6B 20 73 - 65 61 74 20 64 72 69 76 - a.back.seat.driv

00000070: 65 72 00 00 02 00 00 00 - - er......

\*/

================================================================

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

mostly system calls executed in kernel space, so slow.

(You create and use integer file handles.)

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

\* rdwt.c - demonstrate system-level file i/o in

writing and reading an array of ints.

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

/\*

O\_xxxx (#defines)

Bit definitions for access argument used by the library file-open functions.

O\_APPEND Append to end of file

O\_BINARY No translation NOT in UNIX!

O\_CREAT Create and open file

O\_EXCL Exclusive open

O\_RDONLY Read only

O\_RDWR Read/write

O\_TEXT CR-LF translation

O\_TRUNC Open with truncation

O\_WRONLY Write only

Defined in:

fcntl.h

S\_Ixxxx (#defines)

Definitions used for file status and directory functions.

S\_IFMT File type mask

S\_IFDIR Directory

S\_IFIFO FIFO special

S\_IFCHR Character special

S\_IFBLK Block special

S\_IFREG Regular file

S\_IREAD Owner can read

S\_IWRITE Owner can write

S\_IEXEC Owner can execute

Defined in:

#include <sys/stat.h>

\*/

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <string.h>

int main(void)

{

int handle, array[250], x ;

int bytes ;

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

\* Create a file named "TEST.$$$" in the current directory and

\* write a string to it. If "TEST.$$$" already exists, it

\* will be overwritten.

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

for(x=0; x<250; x++)

array[x] = x ;

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

\* int open(const char \*path, int access [ , unsigned mode ] );

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

handle = open( "TEST.$$$", // filename

O\_WRONLY | O\_CREAT | O\_TRUNC , // write, (create or re-create)

S\_IREAD | S\_IWRITE ) ; // permissions 0600

if(handle == -1)

{

printf("Error opening file.\n");

exit(1);

}

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

\* int write(int handle, void \*buf, unsigned len);

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

bytes = write(handle, array, sizeof(int) \* 250) ;

if(bytes != sizeof(array)){

printf("Error writing to the file.\n");

exit(1);

}

printf("Wrote %d bytes to the file.\n", bytes);

close(handle);

/\*

\* zero out array

\*/

for(x = 0; x<250; x++)

array[x] = 0 ;

/\*

\* open file for reading

\*/

/\* handle = open("TEST.$$$", O\_RDONLY | O\_BINARY, S\_IWRITE | S\_IREAD) ; \*/

handle = open("TEST.$$$", O\_RDONLY ) ;

if(handle == -1)

{

printf("Error Opening File\n");

exit(1);

}

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

\* int read(int handle, void \*buf, unsigned len);

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

bytes = read(handle, array, 250 \* sizeof(int)) ;

if(bytes ==-1)

{

printf("Read Failed.\n");

exit(1);

}

else

printf("Read: %d bytes read.\n", bytes);

getchar() ;

for(x=0; x<250; x++)

printf("array[%d] = %d\n", x, array[x]) ;

close(handle) ;

return 0;

} /\* main() ends \*/