241 lecture note:

For week6

* **Basic data types:** – int : integer ✓ – char : character ✓ – float : floating point number ✓ – double : double-precision floating point number ✓

• **Derived data types:** – Arrays ✓ – Strings ✓ – Structures and Unions • User defined data types – New “types” including enumeration types

* **Derived types**

– Arrays – all elements must be of the same data type

– Strings – array of characters with null \0 character at end

• **What if you need a collection / group of information consisting of different data types?**

– E.g. student record that comprises name (last, first, middle and preferred), student ID, course, type, etc.

– Use a **composite structure** or record that is made up different basic/derived data types;

– Use a **composite union** if different types do not exist at the same time;

– Use **enumeration enum** to define list of constants

**Enumeration**

Enumeration is a user-defined data type. It is defined using the keyword enum and the syntax is:

enum tag\_name {name\_0, …, name\_n} ;

The tag\_name is not used directly. The names in the braces are symbolic constants that take on integer values from zero through n. As an example, the statement:

enum colors { red, yellow, green } ;

creates three constants. red is assigned the value 0, yellow is assigned 1 and green is assigned 2.

/\* This program uses enumerated data types to access the elements of an array \*/

**#include int main( ) {**

**int August[5][7] = {**

**{0,0,1,2,3,4,5},**

**{6,7,8,9,10,11,12},**

**{13,14,15,16,17,18,19},**

**{20,21,22,23,24,25,26},**

**{27,28,29,30,31,0,0}**

**};**

**enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};**

**enum week {week\_one, week\_two, week\_three, week\_four, week\_five};**

**printf ("Monday the third week of August "**

**"is August %d\n", August[week\_three][Mon]);**

**}**

For week7

Command line arguments are parameters supplied to a program, when the program is invoked.

main can take 2 arguments, conventionally called argc and argv.

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

argc

– Number of arguments (including program name)

argv

– Array of char\*s (that is, an array of ‘c’ strings)

– argv[0] à program name

– argv[1] à first argument

– …

– argv[argc-1] à last argument

$ ./main\_arg NWEN241 is about Systems Programming using C 8 arguments

0: ./main\_arg

1: NWEN241

2: is

3: about

4: Systems

5: Programming

6: using

7: C

$

Total of 8 arguments including program name itself. Arguments are read in as strings.

In general, I/O is the process of copying data between main memory and external devices

In C, everything is a file; --🡪each file is simply a sequential stream of bytes;

C imposes no structure on a file.

BUT, Defined in stdio.h is the struct FILE that comprises a file descriptor and a file control block

A file must first be opened properly before it can be accessed for reading or writing.

When a file is opened, a stream is associated with the file. Pointer to (i.e. address of) the “file” is returned

Input / Output & stdio.h

Every UNIX/Linux process begins with three open files corresponding to the standard input, output and error streams, macros defined in stdio.h:

****

Also defined in stdio.h are three variable types (including FILE), several macros (including above) and various functions for performing input / output

e.g. printf(), scanf(), getchar() , gets(), putchar(), puts(), etc.

File operations

1. Creating a new file
2. Opening an existing file
3. Writing data to a file
4. Reading data from a file
5. Closing a file
6. Random access operations

Declaring FILE pointer and Opening file

A file must be “opened” before it can be used.

FILE \*fp; // pointer to data type FILE

fp = fopen (filename, mode);

fopen 🡪returns a pointer (fp) to the file;

* used in all subsequent file operations.

mode 🡪 “r” – open the file for reading only

“w” – open the file for writing only

“a” – open the file for appending data to it

Did the fopen(…) command succeed?

If the file was not able to be opened, then the value returned by the fopen routine is NULL.

For example, if the file mydata does not exist, then:

**FILE \*fptr ;**

**fptr = fopen ("mydata", "r") ;**

**if (fptr == NULL) {**

**printf ("File open failed.\n");**

**}**

Closing a file

After completing all operations on a file, it must be closed to ensure that all file data stored in memory buffers are written to the file.

**General format:** fclose (file\_pointer);

FILE \*fp; // pointer to data type FILE

:::

fp = fopen (filename, mode);

:::

fclose (fp); // close the file

Read/Write Operations on Files

Simplest file input-output (I/O) function: **getc & putc**

**char ch;**

**FILE \*fp;**

**:::**

**ch = getc(fp);**

**getc** will return an end-of-file marker EOF, when the

end of the file has been reached.

**putc** is used to write a character to a file.

**char ch;**

**FILE \*fp;**

**:::**

**putc(c, fp);**

**main() {**

**FILE \*ifp, \*ofp;**

**char c;**

**ifp = fopen ("ifile.dat" , " r ");**

**ofp = fopen ("ofile.dat" , " w ");**

**while ((c = getc (ifp)) != EOF)**

**putc (toupper(c), ofp);**

**fclose (ifp);**

**fclose (ofp);**

**}**

1. fgetc() vs getc()

• **fgetc** is a subroutine that performs the same function as the **getc** macro; **fgetc** is NOT a macro.

• **fgetc** subroutine runs more slowly than **getc** but takes less disk space.

• Benefit: fgetc(\*p++) works but getc(\*p++) fails

1. fputc() vs putc()

• fputc is a subroutine while putc is a macro;

• same considerations for fputc as fgetc.

fscanf()

Same as scanf except need to **file pointer** as an argument.

Example:

int a, b;

FILE \*fptr1;

fptr1 = fopen ("datafile", "r");

fscanf( fptr1, "%d%d", &a, &b);

fscanf would read values from the file "pointed"

to by **fptr1** and **assign those values** to **a** and **b**.

End of File using EOF

The end-of-file indicator **EOF** informs the program when there are no more data (no more bytes) to be processed.

Check the value returned by the **fscanf** function:

**int istatus, var;**

**istatus = fscanf (fptr1, "%d", &var) ;**

**if ( istatus == EOF ) {**

**printf ("End-of-file encountered.\n") ;**

**}**

End of File using feof()

Use the **feof** function which returns a true or false condition:

**fscanf (fptr1, "%d", &var) ;**

**if ( feof (fptr1) ) {**

**printf ("End-of-file encountered.\n");**

**}**

**fprinf()**

Same as **printf** except need to file pointer as an argument.

**int a=5, b=20;**

**FILE \*fptr2;**

**fptr1 = fopen ("results", "w");**

**fprintf (fptr2, "%d %d\n", a, b);**

**fprintf** functions would write the values stored in **a and b** to the file "pointed" to by fptr2.

**Example using fscanf() & fprintf()**

#include

int main ( ) {

FILE \*outfile, \*infile ;

int b = 5, f ;

float a = 13.72, c = 6.68, e, g ;

outfile = fopen ("testdata", "w") ;

fprintf (outfile, "%6.2f%2d%5.2f", a, b, c) ;

fclose (outfile) ;

infile = fopen ("testdata", "r") ;

fscanf (infile,"%f %d %f", &e, &f, &g) ;

printf ("%6.2f,%2d,%5.2f\n", e, f, g) ;

fclose (outfile) ;

}

**Handling binary files**

Same as dealing with text files except in the opening step.

Need to open the file as a binary file using the binary mode identifier,

e.g.

**– "rb" r for read and b for binary**

**– "wb" w for write and b for binary**

**– ”ab" a for append and b for binary**

Example:

**FILE \*ptr;**

**ptr = fopen ("file1.exe","rb");**

**Reading binary files**

**fread** reads a block of binary data, up to **nmemb** elements of size,

**size** from **stream**, storing them at the address specified by **ptr.**

**size\_t fread ( void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**

**fread** returns the actual number of elements read.

Example:

**unsigned char buffer[10];**

**FILE \*ptr;**

**ptr = fopen("file1.exe","rb");**

**fread (buffer, sizeof(buffer), 1, ptr);**

Writing binary files

**fwrite** writes a block of binary data comprising **nmemb** elements of size,

**size** from **ptr** to **stream**.

**size\_t fwrite (const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**

**fwrite** returns the number of elements written.

Example:

**unsigned char buffer[10];**

**FILE \*write\_ptr;**

**write\_ptr = fopen("file2.exe","wb");**

**fwrite (buffer,sizeof(buffer),1,write\_ptr);**

Random Access (1)

Most often used with binary files using **fseek, ftell and rewind**.

**fseek** allows repositioning within a file.

**int fseek(FILE \*stream, long int offset, int startpoint);**

New position in the file is determined by:

**offset** – byte count (possibly -ve) relative to the position specified

by **startpoint** where

**startpoint = {SEEK\_SET, SEEK\_CUR, SEEK\_END}**

**| | |**

**Beginning of file Current file position End of file**

**Random Access (2)**

**ftell** returns the current file position:

**long int ftell(FILE \*stream);**

This may be saved and later passed to **fseek:**

**long int file\_pos;**

**file\_pos = ftell(fp);**

**…**

**fseek(fp, file\_pos, SEEK\_SET);**

/\* return to previous position \*/

**rewind(fp)** is equivalent to:

**fseek(fp, 0, SEEK\_SET).**