Preprocessor

Outline

Introduction

Preprocessing
- Occurs before a program is compiled
- Common Uses
  - ✓Inclusion of other files
  - ✓Definition of *symbolic constants* and *macros*
  - ✓*Conditional compilation* of program code
  - ✓*Conditional execution of preprocessor directives*

Format of preprocessor directives
- Lines begin with #
- Only white-space characters before directives on a line

The #include Preprocessor Directive

#include

- Copy of a specified file included in place of the directive
  - `#include <filename>`
    - ✓Searches standard library for file
    - ✓Use for standard library files
  - `#include "filename"`
    - ✓Use for user-defined files
    - ✓Searches current directory, then standard library

Used for

- Loading standard header files (`#include <stdio.h>`)
  - ✓Header file - has common declarations and definitionsfor UDT / Macros (structures, function prototypes)
- Programs with multiple source files to be compiled together

#define : Symbolic Constants

#define

- Creates symbolic constants and macros.

Symbolic constants

- On compilation, all occurrences of symbolic constant replaced with replacement text

Format

`#define identifier replacement-text`

- Example: `#define PI 3.14159`
- Everything to right of identifier replaces text
  `#define PI = 3.14159`
  - ✓replaces "**PI**" with " **= 3.14159**", probably results in an error
- Cannot redefine symbolic constants

#define : Macros

Macro

- Operation defined in `#define`
- Macro without arguments: treated like a symbolic constant
- Macro with arguments: arguments substituted for replacement text, macro expanded
- Performs a text substitution - no data type checking

Example:

```
#define CIRCLE_AREA( x ) ( (PI) * ( x ) * ( x ) )

area = CIRCLE_AREA( 4 );
```

is expanded to

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

#define: Macros

Use parenthesis, as otherwise

```
#define CIRCLE_AREA( x )  PI *  x  *  x
area = CIRCLE_AREA( c + 2 );
```

becomes

```
area = 3.14159 * c + 2 * c + 2;
```

✓Evaluates incorrectly

Macro's advantage

- **Avoid function call overhead**
- **As, macro inserts code directly.**

Macro's disadvantage

- **Argument may be evaluated more than once.**

While defining, don't give a space between Macro Name & parenthesis. Why!!!

#define: Macros

Multiple arguments

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

becomes

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```


#define: Macros

#undef

- Un-defines a symbolic constant or macro
- which can later be redefined

#define getchar() getc(stdin)

Conditional Compilation

Control preprocessor directives and compilation
Conditions refer to status / value of *#define symbols*

Structure similar to **`if`**

```
#if !defined( NULL )
   #define NULL 0
#endif
```

- Determines if symbolic constant `NULL` defined
  - ✓ If `NULL` is defined, `defined(NULL)` evaluates to `1`
  - ✓ If `NULL` not defined, defines `NULL` as `0`

Every `#if` ends with `#endif`
`#ifdef` short for `#if defined(name)`
`#ifndef` short for `#if !defined(name)`

Conditional Compilation - II

Other statements
`#elif` - equivalent of **`else if`** in an **`if`** structure
`#else` - equivalent of **`else`** in an **`if`** structure

"Comment out" code
- Cannot use `/* ... */`
- Use /* to prevent it from being compiled */

```
#if 0
   code commented out
#endif
```

- To enable code, change **`0`** to **`1`**

Conditional Compilation - III

Debugging

```
#define DEBUG 1
#ifdef DEBUG
   printf (" Variable x = %d \n", x) ;
#endif
```

Spot the problem

- Defining `DEBUG` enables code
- After code corrected, remove `#define` statement
- Debugging statements are now ignored

#error and #pragma

#error *tokens*

- Prints implementation-dependent message
- Tokens are groups of characters separated by spaces
  - ✓ `#error 1 - Out of range error` has 6 tokens
- Compilation may stop (depends on compiler)

#pragma *tokens*

- Actions depend on compiler
- May use compiler-specific options
- Unrecognized `#pragma`s are ignored
- E.g. `startup, exit`

The # and ## Operators

`#`

- Replacement text token converted to string with quotes

```
#define HELLO( x ) printf (" Hello, " #x "\n");

HELLO(World)
```

becomes

```
printf (" Hello, " "World" "\n");
printf (" Hello, World\n");
```

Notice #

`##`

- Concatenates two tokens
  - ✓ Strings separated by whitespace are concatenated

```
#define TOKENCONCAT( x, y )  x ## y

TOKENCONCAT( O, K )
```

becomes

```
OK
```


Line Numbers

#line

- Renumbers subsequent code lines, starting with integer
  - ✓ `#line 100`
- File name can be included
- `#line 100 "file1.c"`
  - ✓ Next source code line is numbered `100`
  - ✓ For error purposes, file name is `"file1.c"`
  - ✓ Can make syntax errors more meaningful
  - ✓ Line numbers do not appear in source file

Predefined Symbolic Constants

Five predefined symbolic constants

- Cannot be used in `#define` or `#undef`

Symbolic constant	Description
`__LINE__`	The line number of the current source code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form `"Mmm dd yyyy"` such as `"Jan 19 2001"`).
`__TIME__`	The time the source file is compiled (a string literal of the form `"hh:mm:ss"`).

Assertions

`assert` macro

- Header `<assert.h>`
- Tests value of an expression
- If `0` (false) prints error message and calls `abort`

```
assert( x <= 10 );
```

If `NDEBUG` defined...

- Define before including assert.h
- All subsequent `assert` statements ignored
- `#define NDEBUG`

LIBRARY FUNCTIONS IN C LANGUAGE

❖Library functions is a set of ready-made software routines (functions) for programmers.

❖To use a library function, it is necessary to call the appropriate header file at the beginning of the program.

The C language is accompanied by a number of standard library functions which carry out various useful tasks. Some of them are discussed as follows:

SOME GENERAL FUNCTIONS <STDLIB.H>

bsearch

❖ PURPOSE :

Performs the binary search operation.

❖ SYNTAX :

```
void *bsearch(const void *key, const void *base,
                    size_t nelem, size_t width,
              int (*fcmp)(const void*, const void*));
```

❖ PARAMETER DESCRIPTION :

- ➔ key : Value to be searched.
- ➔ base : 0th element of the array.
- ➔ nelem : Number of elements in the array.
- ➔ width : Number of bytes in each entry.
- ➔ fcmp() : A user defined comparison routine that compares 2 elements and returns a value based on the comparison.

❖ RETURN VALUE :

The address of the first entry in the array that matches the key , on success.

0 , on failure (No match).

❖ E.g

```
int *itemptr;
itemptr = (int *) bsearch (&key, numarray,
                NELEMS(numarray), sizeof(int),
                numeric);
```


qsort

❖ PURPOSE :

Sorts an array using Quick sort algorithm.

❖ SYNTAX :

```
void qsort(void *base, size_t nelem, size_t width,
        int (*fcmp)(const void *, const void *));
```

❖ PARAMETER DESCRIPTION :

➔base : 0th element of the array.

➔ nelem : Size of the array.

➔ width : Number of bytes in each entry.

➔ fcmp() : A user defined comparison routine that compares 2 elements and returns a value based on the comparison.

❖ RETURN VALUE :

Does not return anything.

❖ E.g

```
qsort (numarray,NELEMS(numarray),
        sizeof(int), numeric);
```


min and max

❖ PURPOSE :

min : Finds the smaller of 2 values.

max : Finds the larger of 2 values.

❖ SYNTAX :

(type) min(a,b);

(type) max(a,b);

❖ RETURN VALUE :

min : Minimum of 2 numbers.

max : Maximum of 2 numbers.

❖ E.g

```
int z;
float z1;
z=min(2,7);
z1=max(4.6,1.0);
```


atoi and itoa

❖ PURPOSE :

atoi-Converts a string to an integer.

itoa- Converts an integer to string.

❖ SYNTAX :

```
int atoi(const char * string);
char * itoa(int value,char *string,int radix);
```


❖ RETURN VALUE :

For atoi

➔ converted integer value of input string on success.

➔ 0 on failure.

For itoa

➔ pointer to the target string.

➔ Null on error.

❖ E.g n=atoi("123.45"); (n being an integer)

itoa(12,str,10);

random

- PURPOSE :
  Generates an integer.
- SYNTAX :
  int random(int num);
- RETURN VALUE :
  An integer between 0 and (num-1).
- E.g
  z=random(100); (z being an integer)

randomize

- PURPOSE :
  Initializes a random number generator with a random value.
- SYNTAX :
  void randomize(void);
- RETURN VALUE :
  Does not return anything.
- E.g randomize();

SOME MATH FUNCTIONS <MATH.H>

ABSOLUTE VALUE FUNCTIONS

- PURPOSE :
- **abs** (a macro) gets the absolute value of an integer
- **fabs , fabsl** calculate the absolute value of a floating-point number
- **labs** calculates the absolute value of a long number

- SYNTAX:
  - int abs(int x);
  - double fabs(double x);
  - long double fabsl(long double x);
  - long int labs(long int x);

- RETURN VALUE :

ROUTINE	RETURN ABSOLUTE VALUE OF
abs	An integer
fabs,fabsl	A double
labs	A long

- E.g

z=abs(-5); (z being an integer)

EXPONENTIAL FUNCTION

exp

- PURPOSE :

  Calculates e raised to the power xth value.

- SYNTAX :

  double exp(double x);

- RETURN VALUE :

  e^x value , given x

- E.g result=exp(10.3); (result being a double)

LOGARITHMIC FUNCTION

log and log10

- PURPOSE :

  log : Calculates natural logarithm of double var.

  log10: Calculates base 10 logarithm of double var

- SYNTAX :

  double log(double x);

  double log10(double x);

- RETURN VALUE :

  log : Natural log of x.

  log10 : Log (base 10) of x.

- E.g

  result=log(10.3);

  result=log10(10.3); (result being a double)

SQUARE ROOT FUNCTION

sqrt

- PURPOSE :

  Calculates square root of a variable.

- SYNTAX :

  double sqrt(double x);

- RETURN VALUE :

  Square root of x, if x is real and positive.

- E.g result=sqrt(4);

TRIGONOMETRIC FUNCTIONS

Sin, cos and tan

- PURPOSE :

  Calculates the sine , cosine and tangent of input values.

- SYNTAX :

  double sin(double x);

  double cos(double x);

  double tan(double x);

- RETURN VALUE :
  - ➔ sin returns sine of the input value in the range –1 to 1.
  - ➔ cos returns cosine of the input value in the range –1 to 1
  - ➔ tan returns tangent of x (sin(x)/cos(x)) .
- E.g

  result=sin(0.5);

  result=cos(0.5);

  result=tan(0.5); (result being a double)

BOUNDARY FUNCTIONS

Ceil and floor

❖ PURPOSE :

ceil : Finds the smallest integer not < its argument.

floor : Finds the largest integer not > its argument.

❖ SYNTAX :

double ceil(double x);

double floor(double x);

❖ RETURN VALUE :

Both ceil and floor returns an integer found as double.

❖ E.g

```
double x=123.57;
double up, down;
up=ceil(x);
down=floor(x);
```


atof

❖ PURPOSE :

Converts a String to a floating point

❖ SYNTAX :

double atof(const char * string)

❖ RETURN VALUE :

➔ a converted double value of input string on success.

➔0 on failure.

❖E.g f=atof("123.73"); (f being a float)

SOME TIMING FUNCTIONS

<TIME.H>

time

- PURPOSE :
  Gets the time of the day in seconds.
- SYNTAX :
  time_t time(time_t *timer);

  time_t is a arithmetic type capable of representing time. However, how time is encoded within this type is unspecified.

- RETURN VALUE :
  The current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970.
  Stores the returned value in *timer variable.
- E.g

```
time_t  t;
t=time(NULL);   //(or time(&t))
printf("The number of seconds since January 1,
        1970 is %ld",t);
```

ctime

❖ PURPOSE :

Converts Date and Time to a string.

❖ SYNTAX :

char * ctime(const time_t * time);

It converts the time value * time into a 26-character long string.

The 26-character string is in the following form and is terminated with a \n and \0 character :

DDD MMM dd hh : mm : ss YYYY

where:

DDD = Day (Mon, Tue, Wed, etc.)
MMM = Month (Jan, Feb, Mar, etc.)
dd = Date (1, 2, ..., 31)
hh = Hour (1, 2, ..., 24)
mm = Minutes (1, ..., 59)
ss = Seconds (1, ..., 59)
YYYY = Year

❖ RETURN VALUE :

Returns a pointer to the character string containing the date and time.

❖ E.g

```
time_t t;
time(&t);
printf("Today's date and time: %s\n", ctime(&t));
```

Tues Nov 4 11:31:54 2008\n\0

stime

- PURPOSE :

  Sets the System Date and Time.

- SYNTAX :

  int stime(time_t *tp);

  tp points to the value of time.

- RETURN VALUE :

  0 , on success.

  -1, otherwise.

- E.g

```
time_t t;
t = time(NULL);
printf("Current date is %s", ctime(&t));
t -= 24L*60L*60L;  /* Back up to same time
                      previous day */
stime(&t);
printf("\nNew date is %s", ctime(&t));
```


difftime

- PURPOSE :

  Computes difference between 2 times.

- SYNTAX :

  double difftime(time_t time1, time_t time2);

- RETURN VALUE :

  Returns the elapsed time in seconds.

❖ E.g :

```
time_t first, second;
first = time(NULL);  /* Gets system
                        time */
delay(2000);        /* Waits 2 secs */
second = time(NULL); /* Gets system time
                        again */
printf("The difference is: %f seconds\n",
        difftime(second,first));
```


tm

A structure defining broken down time.

It is used by functions asctime and localtime.

The Structure Definition is :


```
struct tm {
int tm_sec;     /* Seconds */
int tm_min;     /* Minutes */
int tm_hour;    /* Hour (0--23) */
int tm_mday; /* Day of month (1--31) */
int tm_mon;     /* Month (0--11) */
int tm_year;    /* Year (calendar year minus 1900) */
int tm_wday; /* Weekday (0--6; Sunday = 0) */
int tm_yday;  /* Day of year (0--365) */
int tm_isdst;  /* 0 if daylight savings time is not in
                  effect */
        };
```

asctime

- PURPOSE :

  Converts Date and Time to ASCII.

- SYNTAX :

  char * asctime(const struct tm * tblock);

- RETURN VALUE :

  Returns a pointer to the character

  string containing the date and time.

- E.g

```
struct tm t;
char str[80];
t.tm_sec    = 5;  /* Seconds */
t.tm_min    = 30; /* Minutes */
t.tm_hour   = 11;  /* Hour */
t.tm_mday   = 4; /* Day of the Month  */
t.tm_mon    = 11; /* Month */
t.tm_year   = 108; /* Year - 1900 */
t.tm_wday   = 3;  /* Day of the week  */
t.tm_yday   = 0;  /* Does not show in asctime */
t.tm_isdst  = 0;
```


e..g continued…..

```
/* converts structure to null terminated string */
    strcpy(str, asctime(&t));
    printf("%s\n", str);
    Tues Nov 4 11:30:05 2008
```

localtime

❖ PURPOSE :

Converts Date and Time to structure.

❖ SYNTAX :

struct tm * localtime(const time_t * timer);

❖ RETURN VALUE :

A pointer to a tm structure containing the broken-down time.

File Handling

Files

Means of providing persistent Storage

Uses

- Save data between executions
- Temporary storage

Common set of properties

- ✓Location
- ✓Size
- ✓Date Created
- ✓Date Modified
- ✓Date Last Accessed
- ✓Name

Need For Files

All the programs we worked on so far

- Get their input from the keyboard.
- Write output to the screen.

Works well if

- Input data size is small / data is being entered for the first time.
- Data entered / Output generated is not needed after the program execution is over.

Requires test data / actual data to be entered every time the program is run.

Files can be are used in order to overcome these problems.

Files in C

- 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.
- Successfully opening a file returns a pointer to (i.e., the address of) a file structure.
- This pointer contains the information required to control the file at runtime.

I/O Streams in C

File handling in C is provided by a number of functions in the standard library

All input/output is based on the concept of a stream

There are two types of streams

- Text stream
- Binary stream

I/O Streams in C

A **text stream**

- Is a sequence of characters composed of lines
- EOF character marks the end of stream

A **binary stream**

- Is a sequence of unprocessed bytes
- EOF, or any other character code, has no significance

Text Mode

- File is treated to be a collection of ASCII characters
- Numbers also treated as a set of characters
- "End-of-file" character marks end of the file
- In DOS Text files also contain the newline character (CR+LF) signifying the end of a line
  - No such translation in Linux

Binary Mode

- File is treated to be a collection of binary numbers
- No special EOF character
- No special treatment of *newline characters*
- Numbers stored as corresponding binary equivalents

Standard Streams in C

Name	Description	Remark
stdin	Standard Input Stream	Connected to the Keyboard
stdout	Standard Output Stream	Connected to the Screen
stderr	Standard Error Stream	Connected to the Screen Not redirectable

- Others: stdprn, stdaux
- The compiler produces code to open these files before the *main function* is invoked.
- The I/O macros and functions we have used so far read from stdin and write to stdout.
- These include getchar, putchar, scanf, and printf

File Processing

File Processing

The basic steps involved in processing data contained in files are:

- **Open / Create the file**
- **Process the data**
  - ✓Write / Read / Append
- **Close the Files**

A failure to close the file after processing is over may lead to data corruption.

Opening a File

- When a computer program opens a file, no data is read from it.
- The open request is given to the operating system.
- The operating system finds where the file is stored and issues a "file handle".
- When the program wants to read or write to the file at this point, it uses this file handle (the file descriptor) as a parameter to generate operating system requests.
- *When a file is opened, the logical name is translated into a physical location on a specific device (e.g. a disk drive)*

The FILE ADT

A `struct` type defined in stdio.h

Represents information about a file stream

Required to work on the file data

In general, the function `fopen()` prepares a file for use

Normally accessed via

- Pointers of `FILE *` type,
- Created using `fopen()`
- A function declared in the standard header: stdio.h

fopen()

`fopen(file_name, mode)`

opens the named file in a particular mode and returns a file pointer.

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

- **file_id** is the path, file name, and extension
- **mode** is how we intend to use it
- The **return value** is the file pointer: Use it in all subsequent references to the file.
- If `fopen` fails, NULL is returned.

```
FILE *payfile = fopen("payroll.txt", "r");
```

File Open Modes

Mode	Operations Allowed	Action
r	Read	Return NULL if file doesn't exist
w	Write	Create if file doesn't exist. Destroy file contents if it exists
a	Write at end	Create if file doesn't exist. Retain old contents
r+	Read Write	Return NULL if file doesn't exist
w+	Read Write	Create if file doesn't exist. Destroy file contents if it exists
a+	Read, Write-at-end	Create if file doesn't exist. Retain old contents

Mode is passed as a string constant / variable

In DOS Use t / b in order to specify text / binary mode

The exit() function

- Causes normal program termination.
- Sometimes error means we want an "emergency exit" from an application.
- In main we can use "return" to return to OS.
- In other functions we can use exit to do this.
- Exit is declared in the `stdlib.h` header

`exit(1);` in a function is exactly the same as
`return 1;` in the main routine

The `exit ()` Function

- This is used to leave the program at anytime, from anywhere, before the "normal" exit location.
- Syntax:
  - `exit (status);`
- Example:

```
if( (fp=fopen("a.txt","r")) == NULL){
   fprintf (stderr, "Cannot open a.txt!\n");
   exit(1);
}
```

Four Ways to Read and Write Files

- **Get and put a character**
- **Get and put a line**
- **Formatted file I/O**
- **Block read and write**

Read / Write a Character

```
#include <stdio.h>
int fgetc(FILE * fp);
int fputc(int c, FILE * fp);
```

- These two functions read or write a single byte from or to a file.
- `fgetc` returns the character that was read, converted to an integer.
- `fputc` returns the same value of parameter c if it succeeds, otherwise, return EOF.

fputc()

Writing single character

- Use the function fputc, found in stdio.h
  `int fputc (int c, FILE *fp)`
- Writes the next character to stream associated with fp
- Increments file offset automatically
- On success, returns integer value of characters written
- On failure, returns EOF
- E.g

```
char x = 'c' ;
fputc (x, fp);
```

fgetc()

Reads single character

- Use the function fgetc, found in stdio.h

```
int fgetc (FILE *fp)
```

- Returns the next character on the stream associated with fp
- Increments file offset automatically
- On success, returns the character read as an *int*
- returns EOF when end-of-file is encountered.
- E.g

```
char x = 'c' ;
fputc (x, fp);
```


Reading / Writing Strings

fgets and fputs

```
#include <stdio.h>
char *fgets(char *s, int n, FILE * fp);
int fputs(char *s, FILE * fp);
```

- These two functions read or write a string from or to a file.
- **fgets** reads an entire line into **s**, up to **n-1** characters in length (pass the size of the character array **s** in as **n** to be safe!)
- **fgets** returns the pointer s on success, or NULL if an error or end-of-file is reached.
- **fputs** returns the number of characters written if successful; otherwise, return EOF.

fgets()

```
char* fgets(char *s, int n, FILE *stream)
```

- Reads at most the next *n-1* characters into the array **s**, stopping if a newline is encountered
- Automatically appends a NULL character at the end of the string
- The string **s** must be large enough to hold **n** characters
- Reads white spaces and treats them as part of the string
- **Returns s**, or **NULL** if eof or error occurs.
- Moves the file pointer to the begining of the next line
- e.g. `fgets(sname, sizeof(sname), file_ptr);`
- File must be open to use fgets

fputs()

fputs

```
#include <stdio.h>
int fputs(const char *s, FILE *fp);
```

- Writes a NULL terminated string **s** to **fp**
- Does not append new line character
- Terminating NULL is <u>not copied</u>
- **On success**, returns a non-negative number, indicating the number of characters written
- **On error**, returns EOF
- File must be open to use `fputs`

Reading loops

To Read all the lines of text contained in a file

- Use a `while` loop and `fgets`.

```
FILE *fptr;
char tline[MAXLEN];
fptr= fopen ("testfile.txt","r");

while (fgets (tline, MAXLEN, fptr) != NULL) {
    /* Process the line */
}
fclose (fptr);
```

Formatted I/O

fprintf() and fscanf()

The functions `fprintf()` and `fscanf()` are file versions of the functions `printf()` and `scanf()` respectively.

They have the same parameter rules and format specifications as the functions `printf` and `scanf`

```
fprintf( FILE *fp, const char *format, …);
/*fills fp using format and other params */

fscanf( FILE *fp, const char *format, …);
/*retrieves data from fp, based on format */
```


fprintf()

```
int fprintf( FILE *fp, const char *format, …);
```

- Use function fprintf for multiple variable writes
- On success returns number of bytes written
- On failure returns EOF
- Format specifications same as printf()
- Found in the header file stdio.h
- File must be open to use fprintf

Example

```
fprintf(fp, "Name:%s, Pay:%.2f\n", name, pay);
```

fscanf()

```
int fscanf( FILE *fp, const char *format, …);
```

- Reads multiple variables based on format string
- Moves the file pointer to the next data element to be read
- **On success**: returns the number of input fields assigned successfully
- **On failure:** returns EOF
- Format specifications same as scanf ()
- Found in the header file stdio.h
- File must be open to use fscanf

Example

```
fscanf(fp, “%s %f”, name, pay);
```


I/O on Standard Streams

Recall that: every C program gets three streams at start-up: `stdin`, `stdout`, and `stderr`

They can be used wherever a `FILE *` can be used.

Examples:

- `fprintf(stdout, "Hello there!\n");`
  - ✓ This is the same as printf("Hello there!\n");
- `fscanf(stdin, "%d", &int_var);`
  - ✓ This is the same as scanf("%d", &int_var);
- `fprintf(stderr, “Error occurred!\n");`
  - ✓ This is useful to report errors to standard error - it flushes output as well, so this is really good for debugging!

Block Read / Write

fread and fwrite

Reading and Writing arrays of bytes

- Use the functions fread and fwrite
- Found in the header file stdio.h

Both of them takes four parameters

- a pointer to the array's first element (void*)
- the size (in bytes) of an element
- the number of elements in the array
- a file pointer

Generic Form of fwrite / fread

```
int fread / fwrite
  (void *buf, int size, int count, FILE *fp);
```

Examples

- Writing an array of integers ***arr*** to the file associated with ***fp***

```
fwrite( arr, sizeof(int), 100, fp);
```

- Reading from a file associated with ***fp*** into an array ***arr***

```
fread( arr, sizeof(int), 100, fp);
```

- fwrite (fread) returns the number of **items** actually written (read).

Reading / Writing Composite Variables

```
struct student stud;
/*  Populate stud  */
fwrite( &stud, sizeof(stud), 1, fp);
```

```
struct student studList[MAX];
/*  Populate stud list, say count entries */
fwrite( studList, sizeof(stud)*count, 1, fp);
```

Closing & Flushing Files

Closing a File

When a program has no further use for a file, it should close the file

Flushes the buffers associated with the file

```
int fclose(FILE * fp;
```

Return Value:

- **On Success**: 0
- **On Failure**: EOF

Files are automatically closed at normal program termination, including `exit()`

- They are not closed when the program crashes

Flushing Files

- Buffer can be cleared without closing, if required
  - `int fflush (FILE * fp) ;`
- Essentially this is a force to disk.
- Very useful when debugging.
- Without `fclose` or `fflush`, updates to a file may not be written to the file on disk.
- Operating systems like Unix usually use ***write caching*** disk access.

Sequential & Random Access

Sequential / Random Access

There are two ways of accessing a file, either

- Sequential access
  - ✓Accessing one data element after another in a file
  - ✓If the data is to be process in the order of storage
  - ✓Generally used if all / most of the data records in a given range are to be processed
- Random/direct access
  - ✓Accessing characters at any position in a file
  - ✓Can randomly decide where to start reading/writing a file
  - ✓Use `fseek` and `ftell`

Sequential Access

- The OS maintains data to indicate the position of your next reading or writing.
- When you read/write, the position moves forward.
- You can “rewind” and start reading from the beginning of the file again:
  - `void rewind (FILE * fp) ;`

Random Access: fseek

Repositions the file pointer on a stream:

```
int fseek (FILE* fp, long offset, int
  origin);
```

- offset is the number of bytes to move the position indicator
- origin says where to move from

Three options/constants are defined for origin

- **SEEK_SET**
  - ✓move the indicator offset bytes from the beginning
- **SEEK_CUR**
  - ✓move the indicator offset bytes from its current position
- **SEEK_END**
  - ✓move the indicator offset bytes from the end

Random Access: ftell

```
long ftell (FILE * fp) ;
```

- Returns the current value of the file position indicator, i.e. the number of bytes from the start of the file (starts at zero)
- To determine where the position indicator is use:

```
long pos= ftell (fp) ;
```

  - ✓Returns a long giving the current position in bytes.
  - ✓The first byte of the file is byte 0.
  - ✓If an error occurs, ftell () returns -1.

Detecting End-Of-File

Text Mode

```
while (   (c = fgetc (fp) ) != EOF )
```

- Reads characters until it encounters the EOF
- The problem is that in binary files the byte of data read may actually be indistinguishable from EOF.

Binary Mode

```
int feof (FILE * fp) ;
```

- The feof function realizes the end of file ***only after a reading failed*** (fread, fscanf, fgetc …)

```
fseek(fp,0,SEEK_END);
printf("%d\n", feof(fp)); /* zero */
fgetc(fp);      /* fgetc returns -1 */
printf("%d\n",feof(fp));  /* nonzero */
```


File Management Functions

Erasing a file:

```
int remove (const char * filename);
```

Removes the file specified by ***filename***.

Return Value:

- **On Success**: 0
- **On Failure**: -1

Example

```
remove ("test.txt");
```


Renaming Files

```
int rename
  (const char * oldname,
  const char * newname);
```

Changes the name of a file from ***oldname*** to ***newname***.

Return Value:

- **On Success**: 0
- **On Failure**: -1

Example

```
rename ("test.txt", "paper.txt");
```


Temporary Files

Generic Form:

```
char  *tmpnam (char *s) ;
```

- Included in stdio.h.
- Creates a valid **filename** that does not conflict with any other existing files.

Note this does not create the file

- Just the NAME!
- You then go and open it and presumably write to it.
- The file created will continue to exist after the program executes unless it is deleted explicitly.

Temporary Files

```
char  *tempnam (char *dir, char*prefix) ;
```

- Included in stdio.h.
- Creates a valid filename in the specified directory
- No files are created
- Caller should call free on the allocated *name string* after use
- Returns NULL on failure

Temporary Files

Files that only exist during the execution of the program.

Generic Form:

```
FILE  *tmpfile (void) ;
```

- Included in stdio.h.
- Creates a temporary file in w+b mode
- returns FILE * on success
- returns NULL on failure
- The file is automatically removed when it is closed or when program terminates.