

## The C libraries

The library defined by the ANSI standard is declared in the following header files.

```
<assert.h>   <float.h>   <math.h>    <stdarg.h>  <stdlib.h>
<ctype.h>    <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>    <locale.h> <signal.h>  <stdio.h>   <time.h>
```

### <assert.h>

This contains one entry, a macro used to add diagnostics to programs. When called with an expression, if the result is zero a diagnostic line is printed to stdout. The diagnostic line displays the expression, the source filename and the line number in the source file. It then terminates the program.

### <ctype.h>

Declares functions for testing characters. Each function takes an `int` whose value must be `EOF` or the value of an unsigned `int`. It returns an `int`. Non-zero is true, zero is false.

<code>isalnum(c)</code>	<code>isalpha(c)</code> or <code>isdigit(c)</code> is true
<code>isalpha(c)</code>	<code>isupper(c)</code> or <code>islower(c)</code> is true
<code>iscntrl(c)</code>	control character
<code>isdigit(c)</code>	decimal digit
<code>isgraph(c)</code>	printing character except space
<code>islower(c)</code>	lower-case letter
<code>isprint(c)</code>	printing character including space
<code>ispunct(c)</code>	printing character except space, letter or digit
<code>isspace(c)</code>	space, formfeed, newline, carriage return, tab, vertical tab
<code>isupper(c)</code>	upper-case letter
<code>isxdigit(c)</code>	hexadecimal digit

There are also 2 case conversion functions, `int tolower(c)` and `int toupper(c)`.

### <errno.h>

These declarations allow explicit testing and setting of the status indicators used by many of the library functions when an error or end of file occurs. It includes the function `void perror(const char *s)` which will print an implementation defined message that relates to `errno`.

### <float.h> & <limits.h>

These define the implementation dependant constants that relate to floating point arithmetic. They are primarily the minimum magnitude of various floating point values such as the number of decimal digits of precision or the smallest representable number etc.

`<limits.h>` does the same sort of thing for the minimum and maximum values for `int` and `char` etc.

### <locale.h>

This is where all the regional differences, such as the date representation, are defined. It also includes the character used for a decimal point, the way currency is represented and so forth.

### <math.h>

The maths library contains all the expected trigonometric functions, which return values expressed in radians, as well as various other maths functions. All the functions return `double`. A partial list is given below. In the list `x` and `y` are both `double`.

<code>sin(x)</code>	<code>cos(x)</code>	<code>tan(x)</code>	<code>asin(x)</code>
<code>acos(x)</code>	<code>atan(x)</code>	<code>atan2(y,x)</code>	<code>sinh(x)</code>
<code>cosh(x)</code>	<code>tanh(x)</code>	<code>exp(x)</code>	<code>log(x)</code>
<code>log10(x)</code>	<code>pow(x,y)</code>	<code>sqrt(x)</code>	<code>fabs(x)</code>

### <setjmp.h>

The functions in this library allow you to avoid the usual function call and return sequence by letting you escape from deeply nested function calls. A bit like the `break` statement only worse.

### <signal.h>

This permits you to handle exceptions that might arise during program execution. It allows you to set up pointers to functions that will be used if certain signals occur. For example a function could be written to handle attempts to access outside memory limits (segmentation fault). If this signal was sent by the system it can be trapped and the appropriate user defined function called. If the function returns, execution of the program code will continue from the point at which the interrupt occurred.

```
void (*signal(int sig, void (*handler)(int)))(int);
```

This declaration says that `signal` is a function that returns a pointer to another function. This second function takes an `int` and returns `void`. The second **argument** to `signal` is also a pointer to a function that returns `void` and takes a single `int` argument. An example might help!†

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

FILE *temp_file;
void leave(int sig);

main()
{
    (void) signal(SIGINT,leave);

    temp_file=fopen("tmp","w");
    for(;;) {
        /*
         * Do things...
         */
        printf(
            (void) getchar());
    }
    /* can't get here */
    exit(EXIT_SUCCESS);
}
```

---

† This example is from *The C Book* by Banahan, Brady & Doran.

```

/*
 * on receipt of SIGINT, close tmp fi le
 * but beware - calling library functions from a
 * signal handler is not guaranteed to work in all
 * implementations....
 * this is not a strictly conforming program
 */

void
leave(int sig) {
    fprintf(temp_fi le, "\nInterrupted...\n");
    fclose(temp_fi le);
    exit(sig);
}

```

### <stdarg.h>

This provides the means to handle functions with variable length argument lists that you might write. ie. functions akin to `printf()`.

### <stddef.h>

This contains various definitions and macros that are used in other headers. It includes such things as a special type for the result of subtracting one pointer from another. It also includes a macro for determining the offset, in bytes, of a member of a struct. It results in the distance between the start of the struct and the member.

### <stdio.h>†

Everything, almost, to do with input and output is declared in this header. It contains almost a third of the standard library. It deals with *streams*. In other words a source, or destination, of data. This data may be a text stream or a binary stream. (unix tends to see no difference). Streams are connected to files or devices by *opening* and disconnected by *closing*. Some of the available file operations are

```

FILE *fopen(const char *filename, const char *mode);

FILE *freopen(const char *filename, const char *mode, FILE *stream);

/* a typical use is for redirecting stdin, stdout or
 * stderr as in
 *
 * FILE *erfile;
 * erfile = freopen("error_log", "a", stderr);
 *
 * which will send all error messages generated by the program
 * to the file error_log
 */

int fflush(FILE *stream) /*flush the output buffer*/

```

---

†This section discusses only a few of the commonly used functions declared in `stdio.h`.

```
int fclose(FILE *stream);
```

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

```
/* creates a unique temporary file, opened for binary write  
* that will be automatically deleted when closed or on normal  
* program completion  
*/
```

Having opened a stream, or when using any of the automatically opened streams, the following functions provide formatted output

```
int fprintf(FILE *stream, const char *format, ...);
```

```
int printf(const char *format, ...);
```

```
/* this is equivalent to fprintf(stdout,...);  
*/
```

and formatted input

```
int fscanf(FILE *stream, const char *format, ...);
```

```
int scanf(const char *format, ...); /* eqv. fscanf(stdin,...) */
```

character input and output

```
int fgetc(FILE *stream)
```

returns the next char of stream or EOF

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

returns, at most, n-1 characters into array s. It includes the '\n'.

```
int fputc(int c, FILE *stream)
```

writes the character c on stream. It returns the character written or EOF.

```
int fputs(const char *s, FILE *stream)
```

writes the string s, adding '\n'. It returns non-negative or EOF.

```
int getc(FILE *stream)
```

This is equivalent to fgetc except that if its a macro, it may evaluate stream twice.

```
int getchar(void)
```

This is equivalent to getc(stdin)

```
char *gets(char *s)
```

reads the the next input line into array s. The newline is replaced with '\0'.

```
int putc(int c, FILE *stream)
```

is equivalent to fputc. The getc warning applies.

```
int putchar(int c)
```

is equivalent to putc(c, stdout)

```
int puts(const char *s)
```

writes the string s and a newline to stdout.

```
int ungetc(int c, FILE *stream)
```

pushes c back onto stream, where it can be re-read. can only push one character. can't push EOF.

stdio.h also includes functions for direct reading and writing of files. These have been discussed in other handouts. There are also a number of functions that provide for setting the position within a file. In brief they are

```
int fseek(FILE *stream, long offset, int origin)
    /* The permitted values of offset depend on whether it is */
    /* a text stream or a binary file. */

void rewind(FILE *stream)
    /* resets file pointer to start of file and clears errors */
    /* rewind(fp) ≡ fseek(fp, 0L, SEEK_SET); clearerr(fp) */

long ftell(FILE *stream)
    /* returns current file position for stream */

int fgetpos(FILE *stream, fpos_t *ptr)
    /* returns current file position in stream in *ptr */

int fsetpos(FILE *stream, const fpos_t *ptr)
    /* sets file position to value in ptr returned by fgetpos */
```

### <stdlib.h>

Contains miscellaneous utility functions for number conversion, storage allocation and similar stuff. A fairly complete list is...

double atof(const char *s)	converts s into double
int atoi(const char *s)	converts s to int
long atol(const char *s)	converts s to long
double strtod(const char *s, char **endp)	converts the prefix of s to double, ignoring leading whitespace. A pointer to any unconverted suffix is stored in *endp
long strtol(const char *s, char **endp, int base)	as strtod but assumes s is expressed in base, if base is between 2 - 36. If base is zero it assumes the base is 8,10 or 16 and identifies it from the prefix.
int rand(void)	pseudo-random number between 0-RAND_MAX
void srand(unsigned int seed)	seed a new pseudo-random number
void *calloc(size_t nobj, size_t size)	returns pointer to space for array[nobj]
void *malloc(size_t size)	returns pointer to space for sizeof(obj)
void *realloc(void *p, size_t size)	changes the size of space pointed to by p to size
void free(void *p)	deallocate space, previously allocated dynamically, pointed to by p
void abort(void)	cause abnormal program termination ≡raise(SIGABRT)
void exit(int status)	cause normal program termination
int atexit(void (*fcn)(void))	registers the function fcn to be called on normal termination
int system(const char *s)	passes string s to environment for execution
char *getenv(const char *name)	returns the environment string associated with s
int abs(int n)	returns the absolute value

<code>long labs(long n)</code>	returns the absolute value
<code>div_t div(int num, int denom)</code>	returns the quotient and remainder in <code>int</code> members of a structure of type <code>div_t</code> , members are <code>quot</code> and <code>rem</code>
<code>ldiv_t ldiv(long num long denom)</code>	as above but values are long

There are other things of interest in this header, such as functions to search and sort arrays. A perusal of the header file can be useful.

### <string.h>

This is where all the useful string handling routines are kept. They were kept out of the language itself in order to keep the language small. After all most embedded systems have little or no need for string handling. Some of the commonly used functions are shown below. There are others which return pointers to the first occurrence of one string in another string. Again, browsing through `string.h` can be enlightening.

The arguments used below are defined as follows; `s` is of type `char *`, `cs` and `ct` are of type `const char *`; `n` is of type `size_t`, `c` is an `int` converted to `char`. As with most C functions, pointers will be returned with a `NULL` value on error. If there are less than `n` characters to be copied, the destination string will be padded with `'\0'` characters.

<code>char *strcpy(s, ct)</code>	copies <code>ct</code> to <code>s</code> , returns <code>s</code>
<code>char *strncpy(s, ct, n)</code>	copy up to <code>n</code> characters from <code>ct</code> to <code>s</code> , returns <code>s</code>
<code>char *strcat(s, ct)</code>	append <code>ct</code> to <code>s</code> , return <code>s</code>
<code>char *strncat(s, ct, n)</code>	append up to <code>n</code> characters to <code>s</code> , return <code>s</code>
<code>int strcmp(cs, ct)</code>	compare <code>cs</code> to <code>ct</code> , returns <code>&lt;0</code> if <code>cs&lt;ct</code> , <code>0</code> if <code>cs==ct</code> and <code>&gt;0</code> if <code>cs&gt;ct</code>
<code>int strncmp(cs, ct, n)</code>	compare, at most, <code>n</code> characters
<code>char * strchr(cs, c)</code>	return pointer to first occurrence of <code>c</code> in <code>cs</code>
<code>char * strrchr(cs, c)</code>	return pointer to last occurrence of <code>c</code> in <code>cs</code>
<code>size_t strlen(cs)</code>	return length of <code>cs</code>
<code>char * strerror(n)</code>	returns pointer to implementation defined string, referenced by error <code>n</code>

`string.h` also contains a number of functions for byte-wise copying and comparing.

### <time.h>

These functions provide for manipulating date and time values. Calls include `ctime()`, `gmtime()`, `localtime()`. See `man ctime` for details. Many of them rely on `struct tm` which contains the following components.

<code>int tm_sec;</code>	seconds after the minute, 0 - 61
<code>int tm_min;</code>	minutes after the hour, 0 - 59
<code>int tm_hour;</code>	hours since midnight, 0 - 23
<code>int tm_mday;</code>	day of the month, 1 - 31
<code>int tm_mon;</code>	months since January, 0 - 11
<code>int tm_year;</code>	years since 1900
<code>int tm_wday;</code>	days since Sunday, 0 - 6
<code>int tm_yday;</code>	days since January 1, 0 - 365
<code>int tm_isdst</code>	Daylight Saving Time flag, <code>&gt;0 = true</code> , <code>0 = false</code> , <code>&lt;0 = n/a</code>

The available functions provide for converting the contents of `struct tm` into an `ascii` string; the time, in `clock_t` ticks, that a program has been running and others.

### Other Libraries :-

The above provides a fairly complete list of the contents of the standard C library. However there are typically many other C libraries present on a system. There will be libraries to provide graphics functions, libraries for GUIs and windowing front ends, support libraries needed by particular applications and libraries providing operating system support. There may be libraries to support cross-compilation and embedded system development. There may also be two versions of the libraries, one for dynamic linking and one for static linking.

## System Calls

`man 2 syscalls` will provide a list of documented system calls. There should be an individual entry for each of the calls. Make sure that you refer to the volume that you wish to search. For example `man kill` will get you the user command, `man 2 kill` will provide the system call `kill`. If you don't know which volume you need then do `man -a kill` for example, which will produce all the manual pages called 'kill'. You can then page through until you get to the bit you need.

Not all system and library calls are documented. There is a link on my webpage to the current undocumented calls.

A few useful system and library calls that you might need in the shell assignment.

<code>getcwd()</code>	get current working directory
<code>chdir()</code>	change directory
<code>getenv()</code>	get the value of an environment variable
<code>fork()</code>	start a new process
<code>execve()</code>	execute a program (one of a family of calls)
<code>kill()</code>	send a signal to a named process
<code>opendir()</code>	open the named directory
<code>scandir()</code>	get the next entry from the named directory
<code>stat()</code>	get info about a file.
<code>lstat()</code>	get info about a link, not the file the link points to.
<code>getpwuid()</code>	get info about a user from the uid
<code>getgrgid()</code>	ditto for group id.

In all cases you are advised to read the fine manual pages and to discuss questions with your lab tutor. The manual page tells you which header files you need to include and the type of the parameters involved. It will also tell the value returned on error. Finally you do not need to copy the example in the man page verbatim. eg : the man page shows

```
int stat(const char *file_name, struct stat *buf);
```

so you would write ...

```
...
int stat_test = 1; /* stat returns 0 on success, -1 on error*/
char *file_name; /* will be set by call to readdir()*/
struct stat buf /* space to store the file info */
...

stat_test = stat( file_name, &buf);
if ( stat_test < 0 )
    perror( ...

...

```