Appendix B The Standard Library

B.1 Introduction

C is supported by a collection of constants and functions to manage, for example, memory, strings and files. The constants and functions are grouped in library files. Commonly used constants and functions are described in this appendix.

B.2 The Standard Libraries

The standard libraries are listed below..

`assert.h`	diagnostic - check assertions at run-time
`ctype.h`	character processing
`errno.h`	error handling
`float.h`	floating point limits
`limits.h`	environmental and integer limits
`locale.h`	usage according to local customs
`math.h`	the mathematical functions
`setjmp.h`	non-local transfer of control
`signal.h`	signal or interrupt handling
`stdarg.h`	variable length argument list processing
`stddef.h`	standard definitions
`stdio.h`	standard file input and output
`stdlib.h`	char array conversions, memory management, sundry functions
`string.h`	char array manipulations
`time.h`	date and time management

]B.3 assert.h

There is only one function in this library.

```
void assert(int expression);
```

If *expression* is false (0), *assert* prints an error message and then calls *abort*. For example

```
/* program b1 - assert */
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>  /* for abort */
```

```
int main()
{
  int filesOpen = 11;
  assert(filesOpen <= 10);
}
```

results in

```
Assertion failed: filesOpen <= 10  file program.c
Abnormal program termination.
```

being displayed on the screen.

B.4 ctype.h

ctype.h contains the character-processing functions. The argument for each function is an *int* because the storage type must be capable of containing the end-of-file character, *EOF*. The following functions return true (none zero) if their argument meets the stated condition; otherwise, they return false.

function	**condition**
`int isalnum(int c);`	letter or a digit
`int isalpha(int c);`	either an upper-case or a lower-case letter
`int iscntrl(int c);`	control character e.g *NULL*, *TAB*, *CR*, *ESC*, *DEL*
`int isdigit(int c);`	base 10digit e.g ..., -2, -1, 1, 2, 3, ...
`int isgraph(int c);`	printable, but not space
`int islower(int c);`	lower case letter
`int isprint(int c);`	printable, including space
`int ispunct(int c);`	printable, excluding space, letter and digit
`int isspace(int c);`	space, formfeed, newline, return, tab
`int isupper(int c);`	upper case letter

The next two functions convert a character from one case to the other.

`int tolower(int c);`	returns the lower case version of *c* if *isupper(c*), else returns *c*
`int toupper(int c);`	returns upper case version of *c* if *islower(c)*, else returns *c*

B.5 errno.h

errno.h defines *errno* into which values are placed when a run-time error occurs. Functions such as *perror* and *strerror* can be used to translate these values into descriptive error messages - see program b3 in section B.10.1 for example.

B.6 float.h

Memory imposes limits on values of type *float*. The prefixes *DBL*, *FLT* and *LDBL* refer to *double*, *float* and *long double* respectively.

`DBL_DIG`	number of digits of precision, at least 10
`DBL_MAX`	largest number of type *double*, at least 10^{37}
`DBL_MIN`	smallest number of type *double*, at least 10^{-37}

FLT or *LDBL* can be used in place of *DBL*, with a corresponding change in the description.

B.7 limits.h

limits.h defines the limits for integer values (although they can vary from implementation to implemnetation).

`INT_MAX`	largest *int* value, at least 32,767
`INT_MIN`	smallest *int* value, at least -32,767
`LONG_MAX`	largest *long int*, at least 2147483647L
`LONG_MIN`	smallest *long int*, at least -2147483647L
`SHRT_MAX`	largest *short int* value, at most 32,767
`SHRT_MIN`	smallest *short int* value, at least -32767
`UINT_MAX`	largest *unsigned int*, at least 65535U
`ULONG_MAX`	largest *unsigned long int*, at least 4294967295UL

B.8 math.h

math.h contains the various maths functions.

Function	**Returns**	**Condition/comment**
`double exp(double n);`	e^n	
`double ldexp(double m, int n);`	$m \times 2^n$	
`double log(double n);`	$\log_e n$	$n > 0$
`double log10(double n);`	$\log_{10} n$	$n > 0$
`double acos(double n);`	inverse cosine	$-1.0 < n < 1.0$
`double asin(double n);`	inverse sine	$-1.0 < n < 1.0$
`double atan(double n);`	inverse tan	
`double cos(double n);`	cosine of *n*	
`double sin(double n);`	sine of *n*	
`double tan(double n);`	tangent of *n*	
`double ceil(double n);`	smallest integer not less than *n*.	*ceil(17.8) == 18.0* *ceil(-17.8) == -17.0*
`double fabs(double n);`	*n* if *n >= 0.0*, *-n* if *n < 0.0*.	*fabs(2.75) == 2.75* *fabs(-2.75) == 2.75*
`double floor(double n);`	largest integer not	*floor(17.8) == 17.0*

	more than *n*.	*floor(-17.8) == -18.0*
`double fmod(double n,double d);`	remainder from *n / d*	*d > 0.0*, remainder same sign as *n*
`double pow(double n, double p);`	n^p	except if *n = 0* and *p <= 0* or if *n < 0* and *p* is not an integer
`double sqrt(double n);`	$\sqrt{n}$	*n >= 0.0*
`double modf(double n, double *ip);`	fraction part	splits *n* into integer and fraction. Integer part is in *ip*.

The use of *modf* is shown below in program b2.

```
/* program b2 - modf */
#include <stdio.h>
#include <math.h>

int main()
{
  double fraction, integer;
  double n = 3.1416;

  fraction = modf(n, &integer);
  printf("Integer part = %f, fraction part = %f",
              integer, fraction);
}
```

displays `Integer part = 3.000000, fraction part = 0.141600`

B.9 stddef.h

stddef contains the following definitions

`NULL`	null pointer
`size_t`	type returned by *sizeof* operator

B.10 stdio.h

A stream represents a file on disk or a device (e.g. printer, keyboard) from which data may be read or to which data may be written. *position* is the point in the stream where the next read or write operation is to take place. *stdio.h* provides functions which manipulate streams, useful constant definitions known as macros, and some type definitions.

`FILE`	the file type
`fpos_t`	holds current position within a file
`BUFSIZ`	default buffer size, typical value = 512
`EOF`	End Of File - returned by some *stdio* functions
`FOPEN_MAX`	maximum number of files that can be open at the same time
`SEEK_CUR`	seek from current position

`SEEK_END`	seek from end of file
`SEEK_SET`	seek from beginning of file
`stderr`	standard error stream
`stdin`	standard input stream
`stdout`	standard output stream

The three standard streams are automatically opened when a program runs.

B.10.1 The Error Functions

Some file processing functions set error and end-of-file indicators, and place a diagnostic integer value in *errno*.

`void clearerr(FILE *stream);`	clears end of file and *error* indicators for *stream*
`int feof(FILE *stream);`	returns non-zero if end of *stream* has been detected, otherwise returns zero.
`int ferror(FILE *stream);`	returns zero if no error has occurred on *stream*, otherwise returns non-zero
`void perror(const char *s);`	prints *s* and an error message corresponding to value contained in *errno*.

For example, program b3 shown below prints

```
junk: No such file or directory.
```

```
/* program b3 - perror */
#include <stdio.h>
int main()
{
  char filename[] = "junk";
  FILE *file = fopen(filename, "r");
     /* junk does not exist */
  perror(filename);
}
```

B.10.2 The File Operations

`int fclose(FILE *stream);`	flushes any unwritten data, discards any unread data held in input buffer; returns *EOF* if any errors occur, otherwise returns zero
`int fflush(FILE *stream);`	forces any data held in the buffer to be written into the output stream; returns *EOF* for a write error, otherwise returns zero
`FILE *fopen(const char *file, const char *mode);`	opens file and returns a stream if successful, otherwise returns *NULL*. mode strings include:

r	open text file for reading
w	create and open text file for writing
a	append ie create or open text file for writing at its end
r+	open text file for update ie reading and

writing
w+ create text file for update
rb open binary file for reading
wb create and open binary file for writing
ab append ie create / open binary file for writing at its end
r+b open binary file for update

If mode is *r+* or *r+b* (ie update) either *fflush* or a file-positioning function must be called between each read or write.

```
int remove(char *file);
```
erases the named *file*, returns non-zero if *file* cannot be removed

```
int rename(const char *oName,
           const char *nName);
```
changes *oName* to *nName*, returns non-zero if file cannot be renamed

B.10.3 The Direct File Input-Output Functions

```
size_t fread(void *buffer, size_t size, size_t nItems,
             FILE *stream);
```

- reads up to *nItems* items each of *size* bytes from *stream* into *buffer*
- file position is advanced by the number of bytes read
- returns the number of items actually read
- if the number of items actually read is not the requested number of items, *nItems*, *ferror* and *feof* indicates why.

```
size_t fwrite(const void *buffer, size_t size, size_t nItems,
                  FILE *stream);
```

- writes up to *nItems* each of *size* bytes from *buffer* into *stream*
- file position indicator is increased by amount appropriate to number of bytes written.
- returns number of items actually written
- if a write error occurs, items actually written will not be equal to *nItems*.

B.10.4 The File Positioning Functions

```
int fgetpos(FILE *stream,
            fpos_t *position);
```
copies file *position* for *stream* into **position*, returns non-zero on error

```
int fseek(FILE *stream,
  long offset, int beginning);
```
sets file position for *stream*; this is the point where the next read or write operation will occur. *offset* is the amount, in bytes, to move from beginning. *beginning* is one of
SEEK_CUR the current position
SEEK_END the end of the file
SEEK_SET the beginning of the file
For a text stream, *offset* must either be zero or a value returned by *ftell* (in which case

	beginning must be *SEEK_SET*). *fseek* returns a non-zero value if an error occurs.
`int fsetpos(FILE *stream, const fpos_t, *position);`	sets file position indicator at value returned by a previous call to *fgetpos*. Sets *errno* and returns a non-zero value if an error occurs.
`long ftell(FILE *pf);`	returns current position for *stream*, otherwise returns *-1L* and sets *errno*. (If file position cannot be encoded within *long*, use *fgetpos*)
`void rewind(FILE *stream);`	resets file position to beginning of *stream*

B.10.5 The formatted Output Functions

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

constructs a formatted text and writes it to an output stream; returns the number of characters written, or a value less than zero if an error occurred.

format points to a string which contains text and conversion specifications. Each successive conversion specification applies to the next argument in the argument list following format.

A conversion specification has the format

% flag minFieldWidth .precision modifier conversionCharacter

The *%* and *conversionCharacter* are compulsory; flag, *minFieldWidth*, *.precision* and *modifier* are optional.

Flags include

`-`	print converted argument left justified within its field (default is right justify)
`+`	prefix a signed number with a plus or minus sign
`<space>`	- sign printed if number is negative, otherwise space printed
`0`	for numeric conversions, pad with leading zeros to fill field

minFieldWidth - an integer which sets the minimum width of the field in which the converted argument value is to be printed. If the value is too large to fit in its specified field width, then it overflows the field to the right. If an asterisk is used (instead of an integer) then *printf* uses the following argument (which must be an *int*) as the field width.

.precision - an integer which sets

- the maximum number of characters to the printed from a string or
- the number of digits to be printed after a decimal point or
- the number of significant digits to be printed from an integer

If an asterisk is used, then *printf* uses the following argument (which must be an *int*) as the precision.

modifier specifies

`h`	*short int* or *unsigned short in*t when used before *d*, *i* or *u*
`l`	*long int* or *unsigned long int* when used before *d*, *i* or *u*
`L`	*long double* when used before *e* or *f*

The common *conversionCharacters* are shown below.

character	**argument type converted to**
`c`	*int* or *unsigned int* to character
`d`	*int* to signed decimal notation
`e`	*double* to exponential notation
`f`	*double* to decimal notation
`i`	same as *d*
`p`	*void* * print as a pointer in implementation specific format
`s`	*char* * characters from string until either *'\0'* reached or *precision* characters are printed
`u`	*int* to unsigned decimal notation
`%`	none - print % literaly

```
int printf(const char *format, argument list);
```

printf(...) is equivalent to *fprintf(stdout, ...);*

B.10.6 The Formatted Input Functions

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

reads characters from *stream* and uses *format* to assign values to successive arguments, each of which must be a pointer to an appropriate type; returns EOF if a read error has occurred or the end of file has been reached, otherwise returns the number of items converted.

format points to a string which contains conversion specifications for interpreting the input. Each successive conversion specification applies to the next input field and the result is (usually) placed in the next corresponding argument in the list following format. A format string may contain

- blanks or tabs - these are ignored
- conversion specifications

A conversion specification has the format

% suppressionCharacter maxFieldWidth modifier conversionCharacter

The *%* and *conversionCharacter* are compulsory; *suppressionCharacter*, *maxFieldWidth* and *modifier* are optional.

The suppression character is the asterisk, *. It mean "skip the next input field". An input

field is a string of non-white space characters and ends either at the next white-space character or when *maxFieldWidth* characters have been read. A white-space character is either a blank, a tab, a newline or a carriage-return. *fscanf* may (or may not) read through the newline character. If it does not then the programmer must dispose of the newline character in the input buffer before any subsequent call to *fscanf*, as shown below in program b4.

The modifiers are

`h`	*short int* or *unsigned short int* when used before *d, i* or *u*
`l`	*long int* or *unsigned long int* or *double* when used before *d, i, u* or *f*
`L`	*long double* when used before *e* or *f*

The conversion characters include

character	**applies to**	**comment**
`c`	characters; *char **	Characters are placed in the array up to the given *maxFieldWidth*. The default *maxFieldWidth* is *1*. *'\0'* is not appended. Whie space is not skiped. Use *%1s* to read the next non-white space character.
`d`	decimal (base 10) integer; *int **	
`e`	floating point number; *float **	
`f`	floating point number; *float **	for *double* use *lf*
`i`	integer; *int **.	integer may be octal or hexadecimal
`p`	pointer value; *void **	
`s`	string of non-white space characters; *char **	receiving array must be sufficiently large and *'\0'* will be appended - if there is room in the array.
`u`	unsigned decimal integer; *unsigned int **	
`%`	literal %;	

```
/* program b4 - fscanf */
#include <stdio.h>
int main()
{
  int i;
  double f;
  char s[BUFSIZ];

  printf("integer? decimal? string? ");
  fscanf(stdin, "%d%lf%s%*[^\n]", &i, &f, s);
  printf("Integer is %d, decimal is %0.2f, string is %s\n", i,
                          f, s);
  printf("And again: integer? decimal? string? ");
  fscanf(stdin, "%d%lf%s%*[^\n]", &i, &f, s);
  printf("Integer is %d, decimal is %0.2f, string is %s\n", i,
                          f, s);
}
```

An example of a program run is

```
integer? decimal? string? 1 2.3 Catastrophe
1 2.3 Catastrophe
And again: integer? decimal? string? 9 9.8 Catastrophic
9 9.8 Catastrophic
```

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

int sscanf(char *s,
     const char *format, ...);
```

scanf(...) is equivalent to *fscanf(stdin, ...)*;
sscanf(s, ...) is equivalent to *scanf(...)*; except that the input is taken from the string *s*.

B.10.7 The Character Input-Output Functions

```
int fgetc(FILE *stream);
```
returns character read from *stream* and advances file position if no read error has occurred, otherwise returns *EOF* and sets *stream*'s error indicator

```
int fgets(char *string,
    int n, FILE *stream);
```
reads characters from *stream* into array *string* until either *n-1* characters have been read or a newline character has been read or the end-of-file has been reached. It retains the newline character (if there was one) and appends the null character. Returns *NULL* if end of file has been reached or if an error has occurred, otherwise it returns a string.

```
int fputc(int c,
    FILE *stream);
```
writes *c* to *stream* and advances file position. It returns *c* if write successful, otherwise sets error indicator for *stream* and returns *EOF*

```
int fputs(char *s,
    FILE *stream);
```
writes *s* to stream. Terminating null character is not written. *fputs* returns non-negative number if write was successful, otherwise returns *EOF*.

```
int getc(FILE *stream);
```
reads a character from *stream*. If no error, returns the character read otherwise sets error indicator and returns *EOF*.

```
int getchar(void);
```
reads a character from the standard input. If there is no read error, returns character read from standard input, otherwise sets error indicator and returns *EOF*.

```
char *gets(char *string);
```
reads characters from standard input into array *string* until either a newline character or *EOF* character is read; discards the newline or *EOF* character and appends the null character to *string*.

```
int putc(int character,
    FILE *stream);
```
writes character to *stream*. If no write error, returns *character* otherwise returns *EOF*.

```
int putchar(
    int character);
```
writes *character* into standard output stream. If character could not be written, error indicator is set and *EOF* is returned, otherwise returns the *character* written.

```
int puts(char *string);
```
replaces the null character at the end of *string* with the newline character and then writes the result into the standard output stream. *puts* returns a non-

negative number if write was successful, otherwise it returns *EOF*.

B.11 stdlib.h

The types defined in *stdlib* are

`div_t`	type of object returned by *div*
`ldiv_t`	type of object returned by *ldiv*

The macros or constants are

`EXIT_FAILURE`	value to indicate failure to execute as expected
`EXIT_SUCCESS`	value to indicate function executed as expected
`RAN_MAX`	largest size of pseudo-random number

The functions include

`int abs(int n);`	return absolute value of *n*
`div_t div(int n, int d);`	divides *n* by *d*, stores quotient in *quot* and remainder in *rem* members of a structure of type *div_t*
`void exit(int status);`	immediately terminates program execution, flushing buffers and closing files.
`void free(void *p);`	de-allocate memory previously reserved with *malloc*. *NULL* should be assigned to a freed pointer.
`long labs(long n);`	returns absolute value *n*
`ldiv_t ldiv(ling n, long d);`	divides *n* by *d*, stores quotient in *quot* and remainder in *rem* members of a structure of type *ldiv_t*
`void *malloc(size_t size);`	returns a pointer to a free space in memory large enough to contain an item of the given *size*. The space is not initialised.
`int rand(void);`	returns a pseudo-random number in the range 0..RAND_MAX (typically more than 32766)
`void srand(unsigned int seed);`	uses *seed* to initialise the sequence of pseudo-random numbers generated by *rand*.

`double strtod(`
`  const char *string,`
`  char **tail);`
	converts *string* to a *double* number. An example of a call is
	double d;
	char string[BUFSIZ];
	gets(string);
	*d = strtod(string, (char **)NULL);*
	Returns zero if string is too small to be converted, returns *HUGE_VAL* if *string* is

`long strtol(`
`  const char *string,`
`  char **tail, int base);`
	too large to be converted..
	converts *string* to *long int*. Examples of calls are
	long m;
	int n;
	char string[BUFSIZ];
	gets(string);
	*m = strtol(string, (char**), 10);*
	*n = (int)strtol(string, (char**), 10);*

`unsigned long strtoul(const`
`  char *string,`
`  char **tail,`
`  int base);`
	converts *string* to *unsigned long int*. *LONG_MAX* or *LONG_MIN* is returned if *string* is too large or too small to be converted.

`int system(` `  const char *string);`	passes *string* to operating system for execution. *string* should be the name of an executable program. Any open files should be closed before *system* is called.

`void *bsearch(const void *key,`
`  const void *array,`
`  size_t n, size_t size,`
`  int (*compare)(const void`
`  *arg1, const void *arg2))`
	bsearch searches ordered *array[0]..array[n-1]* for an item that matches **key*. The function *compare* must return a negative value, zero or a positive value if its first argument is less than its second, equal, or greater than the second respectively. *bsearch* returns a pointer to a matching item - if one exists - otherwise it returns *NULL*.

`void qsort(void *array,`
`  size_t n, size_t size,`
`  int (*compare)(const void`
`  *, const void *));`
	sorts *array[0]..array[n-1]* into ascending order. The compare function is as described above for *bsearch*.

Program b5 contains an example of a call to *bsearch*.

```
/* program b5 - binary search */

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

int intCompare(const void *m, const void *n);

int main()
{
  int searchFor = 11;
  int *pResult;
  int numbers[5] = { 2, 4, 6, 8, 10 };

  pResult = (int *)bsearch((void *)&searchFor,
              (void *)numbers, sizeof(numbers) / sizeof(int),
                          sizeof(int), intCompare);
  if (pResult == NULL)
    printf("Not found\n");
  else
    printf("Item found is %d\n", *pResult);
}
```

```
int intCompare(const void *m, const void *n)
{
  int *p = (int *)m;
  int *q = (int *)n;

  if (*p < *q)
    return -1;
  else if (*p == *q)
    return 0;
  else
    return 1;
}
```

B.12 string.h

Functions include

```
int strcmp(const char *s,
     const char *t);
```

compares two strings for equality. Returns *-1* if **s* < **t*, *0* if **s* == **t* and *1* if **s* > **t*. The ordering is based on the underlying character set eg ASCII or EBCDIC.

```
int strncmp(const char *s,
     const char *t, int n);
```

compares at most the first *n* chars of **s* with **t* for equality. Returns *-1* if **s* < **t*, *0* if **s* == **t* and *1* if **s* > **t*.

```
char *strcat(char *s,
     const char *t);
```

adds **t* onto the end of **s*, returns *s*.

```
char *strncat(char *s,
     const char *t, int n);
```

adds at most *n* characters from **t* onto **s*, appends *'\0'*; returns *s*.

```
char *strcpy(char *s,
     const char *t);
```

copies **t* into **s*, returns *s*.

```
char *strncpy(char *s,
     const char *t, int n);
```

copies at most *n* characters from **t* into **s*. Appends *NULL* to **s* if fewer than *n* characters have been copied. WARNING: may not append *NULL* to **s* if *n* characters have been copied. Returns *s*.

```
char *strerror(int n);
```

return text of a pre-defined error message corresponding to *n*.

```
size_t strlen(
     char const *s);
```

returns length of **s*

```
char *strchr(const char *s,
     char c);
```

returns pointer to first occurrence of *c* in **s* - if it is there, otherwise returns *NULL*.

```
char *strstr(const char *s,
     const char *t);
```

returns pointer to first occurrence of **t* in **s* - if **t* is in **s* - otherwise returns *NULL*.

```
char *strtok(char *s,
     const char *t);
```

breaks string **s* into tokens, each delimited by a character from **t*. *strtok* is called repeatedly. On the first call, it searches **s* for the first character which is not in **t*. If one cannot be found, then *strtok* returns *NULL*. But if one can be found, then it searches **s* for a character which is in **t*; if one is found then it is overwritten by *'\0'*; a

pointer to the rest of **s* is stored and a pointer to the first token is returned. For each subsequent call to *strtok*, the first argument must be *NULL*. *strtok* then looks for subsequent tokens using the pointer to the part **s* not yet tokenised.

The program shown below uses *strtok* to split a full name into its constituent parts.

```
/* program b6 - strtok */

#include <stdio.h>
#include <string.h>
int main()
{
  char tokenSeparators[] = " ";  /* space */
  char name[] = "Ms Audrey Walker-Smith";
  char title[BUFSIZ], firstName[BUFSIZ], secondName[BUFSIZ];

  strcpy(title, strtok(name, tokenSeparators));
  strcpy(firstName, strtok(NULL, tokenSeparators));
  strcpy(secondName, strtok(NULL, tokenSeparators));

  printf("Title is %s, first name is %s, second name is %s\n",
         title, firstName, secondName);
}
```

When run, program b6 displays

```
Title is Ms, first name is Audrey, second name is Walker-Smith
```

B.13 time.h

Constants include

`CLK_TCK`	the number of "ticks" in a second

The types for representing time are

`clock_t`	system time
`time_t`	time
`tm`	calendar time

The components of a calendar time are held in *struct tm*. The components are

member	**description**	**range**	**comment**
`int tm_sec`	seconds after the minute	0..59	
`int tm_min`	minutes after the hour	0..59	
`int tm_hour`	hours since midnight	0..23	0 == midnight
`int tm_mday`	day of the month	1..31	
`int tm_mon`	months since January	0..11	0 == January
`int tm_year`	years since 1900	1900..	

`int tm_wday`	days since Sunday	0..6	0 == Sunday
`int tm_yday`	days since 1 January	0..365	
`int tm_isdt`	day-light saving time flag	0 off, > 0 on	

The functions are

`char *asctime(const struct tm *pt);`	converts calendar time into a string. An example of such as string is *Wed May 31 10:33:15 1995\n\0*
`clock_t clock(void);`	returns time since the start of function execution, or -1 if time not available. *clock()/CLK_TCK* is a time in seconds.
`char *ctime(const time_t *pt);`	converts calendar time to local time and returns it as a string format; equivalent to *asctime(localtime(pt))*
`double difftime(time_t t2, time_t t1);`	returns *t2 - t1* in seconds
`struct tm *gmtime( const time_t *pt);`	converts calendar time **pt* to Universal Coordinated Time (UCT); returns *NULL* if UCT not available.
`struct tm *localtime( const time_t *pt);`	converts calendar time **pt* to local time
`time_t mktime(struct tm *pt);`	converts locale time in structure **pt* into calendar time in the same format as used by time. Returns the calendar time, or *-1* if the time cannot be converted.
`size_t strftime(char *string, size_t maxChars, const char *format, const struct tm *pt);`	formats date and time from **pt* into *string* as defined by *format*. The string that *format* points to contains one or more conversion specifications.
`time_t time(time_t *pt);`	returns the current calendar time - if available - otherwise returns *-1*. If *pt* is not *NULL*, **pt* also contains the return value.

Each conversion specifier is prefixed with %. The conversion specifiers include

character	**convert to**
`a`	short weekday name e.g. Mon
`A`	long weekday name e.g. Monday
`b`	short month name e.g. Jan
`B`	long month name e.g. January
`c`	local date and time
`d`	day of the month 1..31
`H`	hour (24-hour clock) 0..23
`I`	hour (12-hour clock) 1..12
`j`	day of the year 1..366

m	month 1..12
M	minute 0..59
p	AM or PM
S	second 0..59
U	week of the year 0..53 (Sunday is 1st day of the week)
w	day of the week 0..6 (Sunday is 0)
W	week of the year 0.53 (Monday is 1st day of the week)
x	local date
X	local time
y	year without century 0..99
Y	year with century
Z	time zone name, if any
%	%