STANDARD C LANGUAGE

The following notations are used:
[ ]--enclosed item is optional; fn--function; b--block; rtn--return; ptd--pointed; ptr--pointer; expr--expression; TRUE--non-zero value; FALSE--zero value.

BASIC DATA TYPES

char	Single character (may signed or unsigned)
unsigned char	Non-negative character
short	Reduced precision integer
unsigned short	Non-negative reduced precision integer
int	Integer
unsigned int	Non-negative integer
long	Extended precision integer
unsigned long	Non-negative extended precision integer
float	Floating point
double	Extended precision floating point
long double	Extended precision floating point
void	No type; Used for denoting: 1) no return value from fn 2) no argument of fn 3) general pointer base

ARITHMETIC CONVERSION OF DATA TYPES

1. If either operand is long double the other is converted to long double.
2. If either operand is double, the other is converted to double.
3. If either operand is float, the other is converted to float.
4. All char and short operands are converted to int if it can represent the original value; otherwise it is converted to unsigned int.
5. If either operand is unsigned long the other is converted to unsigned long.
6. If the two operands are unsigned int and long and long represent all values of type unsigned int, the common type is long; otherwise it is unsigned long.
7. If either operand is long the other is converted to long.
8. If either operand is unsigned int the other is converted to unsigned int.
9. If this step is reached, both operands must be int.

STATEMENT SUMMARY

STATEMENT	DESCRIPTION
{ *local_var_decl* *statement* **... }**	Block. The *local_var_decl* (local variable declarations) is optional.
break;	Terminates execution of **for**, **while**, **do**, or **switch.**
continue;	Skips statement that follow in a **do**, **for**, or **while**; then continues executing the loop.

do
statement
while (*expr***);**
	Executes *statement* until *expr* is FALSE; *statement* is executed at least once.

*expr***;**	Evaluates *expr* ; discards result.
for (*e1***;***e2***;***e3***)** *statement*	Evaluates expr *e1* once; then repeatedly evaluates *e2*, *statement*, and *e3* (in that order) until *e2* is FALSE; eg: **for (i=1; i<=10,/; ++i) ...;** note that *statement* will not be executed if *e2* is FALSE on first evaluation; *e1* ,*e2* and *e3* are optional; *e2=1* assumed when omitted.

goto *label***;**	Branches to statement preceded by *label*, which must be in same function as the **goto.** eg.:
	int Fn(void) {
	... goto write; ...
	write: print("here am I"); ...}

if (*expr***)** *statement*	If *expr* is TRUE, then executes *statement;* otherwise skips it.

if (*expr***)**
statement1
else
statement2
	If *expr* is TRUE, then executes *statement1;* otherwise executes *statement2.*

;	Null statement.No effect.eg.: **while (t[i++]);**
return *expr***;**	Returns from function back to caller with value of *expr* ; *expr* is omitted in **void** functions.
switch (*expr***)** **{ case** *const1***:** *statement* **...** **break;** **case** *const2***:** *statement* **...** **break;** **...** **default:** *statement* **... }**	*expr* (must be an integer expression) is evaluated and then compared against integer constant exprs *const1*, *const2*, ... If a match is found, then the statements that follow the case (up to next **break,** if supplied) will be executed. If no match is found, then the statements in the **default** case (if supplied) will be executed.
while (*expr***)** *statement*	Executes *statement* as long as *expr* is TRUE; statement might not be executed if *expr* is FALSE the first time it's evaluated.

TYPE DEFINITION

typedef is to assign a new name to a data type. To use it make believe you're declaring a variable of that particular data type. Where you'd normally write the variable name, write the new data type name instead. In front of everything, place the keyword typedef. For example:

```
 /* define type COMPLEX */
   typedef struct
     {
       float real;
       float imaginary;
     } COMPLEX;
 /* declare variables with new type COMPLEX */
   COMPLEX c1, c2, sum;
```

CONSTANTS

char	**'**		**'a' '\n'**
char string	**"**		**"hello" ""**
float	…**f**,…**F**	*(1)*	**7.2f 2.e-15F -1E9f .5F**
double		*(1)*	**7.2 2.e-15 -1E9 .5**
long double	…**l**,…**L**	*(1)*	**7.2l 2.e-15l -1E9L .5L**
enumeration		*(2)*	**red january monday**
int			**17 -5 0,/**
long int	…**l**,…**L**	*(3)*	**251l 10,/0,/L**
unsigned int	…**u**,…**U**		**17u 5U 0,/u 65535u**
hex integer	**0,/ x,0,/ X**		**0,/xFF 0,/Xff 0,/xA0,/0,/ 0,/l**
octal int	**0,/**		**0,/777 0,/10,/0,/U 0,/ 573ul**

NOTES:
1. Decimal point and/or scientific notation.
2. Identifiers previously declared for an enumerated type; value treated as int.
3. Or any int too large for normal int

TYPE QUALIFIERS

const	Constant object, cannot be altered by the program.
volatile	External hardware or software can alter the variable, no optimalization.

OPERATORS

OPERATOR	DESCRIPTION	EXAMPLE	ASSOCIATION
++	Postincrement	**ptr++**	
--	Postdecrement	**count--**	
[ ]	Array element ref	**values [10,/]**	⇒
( )	Function call	**sqrt (x)**	
.	Struct member ref	**child.name**	
->	Ptr to struct member	**child_ptr->name**	
sizeof	Size in bytes	**sizeof child**	
++	Preincrement	**++ptr**	
--	Predecrement	**--count**	
&	Address of	**&x**	
*****	Ptr indirection	***ptr**	⇐
+	Unary plus	**+a**	
-	Unary minus	**-a**	
~	Bitwise NOT	**~0,/77**	
!	Logical negation	**! ready**	
(*type***)**	Type conversion / casting	**(float) total/n**	
*****	Multiplication	**i * j**	
/	Division	**i / j**	⇒
%	Modulus	**i % j**	
+	Addition	**value + i**	⇒
-	Subtraction	**x - 10,/0,/**	
<<	Left shift	**byte << 4**	⇒
>>	Right shift	**i >> 2**	
<	Less than	**i < 10,/0,/**	
<=	Less than or equal to	**i <= j**	⇒
>	Greater than	**i > 0,/**	
>=	Greater than or eq to	**count >= 90,/**	
==	Equal to	**result == 0,/**	⇒
!=	Not equal to	**c != EOF**	
&	Bitwise AND	**word & 0,/77**	⇒
^	Bitwise XOR	**word1 ^ word2**	⇒
|	Bitwise OR	**word | bits**	⇒
&&	Logical AND	**j>0,/ && j<10,/**	➔
||	Logical OR	**i>80,/ || ready**	➔
? :	Conditional operator	**a>b ? a : b** If **a** greater than **b** then *expr*=**a** else **b**	⇐
= *= /= **%= += -=** **&= ^= |=** **<<= >>=**	Assignment operators	**count += 2** It is equal to **count=count+2**	⇐
,	Comma operator	**i=10,/ , j=0,/**	➔

NOTES:

Operators are listed in decreasing order of precedence.
Operators in the same box have the same precedence.
Associativity determines: ⇒ grouping; ➔ order of evaluation for operands with the same precedence:
(eg: **a = b = c;** is grouped right-to-left, as: **a = (b = c);**).

PREPROCESSOR STATEMENTS

STATEMENT	DESCRIPTION
#define *id text*	*text* is substituted for *id* wherever *id* later appears in the program; (eg: **#define BUFFERSIZE 512**) If construct **id(a1,a2,…)** is used, arguments **a1,a2,…** will be replaced where they appear in text by corresponding arguments of macro call (eg: **#define max(A,B) ((A)>(B)?(A):(B))**means, that **x=max(p+q,r+s)** macro will be substituted for **x=((p+q)>(r+s)?(p+q):(r+s))** in the program text)
#undef *id*	Remove definition of *id.*
#if *expr* **...** **#endif**	If constant expression *expr* is TRUE, statements up to **#endif** will be processed, otherwise they will not be
#if *expr* **...** **#else** **...** **#endif**	If constant expression *expr* is TRUE, statements up to **#else** will be processed, otherwise those between the **#else** and **#endif** will be processed
#ifdef *id* **...** **#endif**	If *id* is defined (with **#define** or on the command line) statements up to **#endif** will be processed; otherwise they will not be (optional **#else** like at **#if**)
#ifndef *id* **...** **#endif**	If *id* has not been defined, statements up to **#endif** will be processed; (optional **#else** like at **#if**).
#include "*file***"**	Inserts contents of *file* in program; look first in same directory as source program, then in standard places.
#include <*file***>**	Inserts contents of *file* in program; look only in standard places.
#line *n* **"***file***"**	Identifies subsequent lines of the program as coming from *file*, beginning at line *n; file* is optional.

NOTES:

Preprocessor statements can be continued over multiple lines provided each line to be continued ends with a backslash character (****). Statements can also be nested.

STORAGE CLASSES

STORAGE CLASS	DECLARED	CAN BE REFERENCED	INIT WITH	NOTES
static	outside fn	anywhere in file	constant expr	1
	inside fn/b	inside fn/b	constant expr	1
extern	outside fn	anywhere in file	constant expr	2
	inside fn/b	inside fn/b	cannot be init	2
auto	inside fn/b	inside fn/b	any expr	3
register	inside fn/b	inside fn/b	any expr	3,4,6
(omitted)	outside fn	anywhere in file or other files with ext. declaration	constant expr	5
	inside fn/b	inside fn/b	any expr	3,6

NOTES:

1. Init at start of program execution; default is zero.
2. Variable must be defined in only one place w/o **extern**.
3. Variable is init each time fn/b is entered; no default value.
4. Register assignment not guaranteed; restricted (implementation dependent) types can be assigned to registers. **&** (addr. of) operator cannot be applied.
5. Variable can be declared in only one place; initialized at start of program execution; default is zero.
6. Defaults to auto.

EXPRESSIONS

An expression is one or more terms and zero or more operators. A term can be
- *name* (function or data object)
- *constant*
- **sizeof(***type***)**
- **(***expr***)**

An expression is a constant expression if each term is a constant.

ARRAYS

A single dimension array **aname** of **n** elements of a specified type ***type*** and with specified initial values (optional) is declared with :

```
  type aname[n] = { val1, val2, … };
```

If complete list of initial values is specified, **n** can be omitted.
Only static or global arrays can be initialized.
Char arrays can be init by a string of chars in double quotes.
Valid subscripts of the array range from **0,/** to **n-1**.
Multi dimensional arrays are declared with :

```
  type aname[n1][n2]… = { init_list };
```

Values listed in the initialization list are assigned in 'dimension order' (i.e. as if last dimension were increasing first). Nested pairs of braces can be used to change this order if desired.

EXAMPLES:

```
 /* array of char */
  static char hisname[] = {"John Smith"};
 /* array of char ptrs */
  static char *days[7] =
  {"Sun", "Mon", "Tue",
   "Wed", "Thu", "Fri", "Sat"};
 /* 3x2 array of ints */
  int matrix[3][2] = { {10,/,11},{-5,0,/},
{11,21} };
 /* array of struct complex */
  struct complex sensor_data[10,/0,/];
```

POINTERS

A variable can be declared to be a pointer to a specified type by a statement of the form:

```
   type *name;
```

EXAMPLES:

```
 /* numptr points to floating number */
  float *numptr;
 /* pointer to struct complex */
  struct complex *cp;
 /* if the real part of the complex struct
    pointed to by cp is 0,/.0,/ */
  if (cp->real == 0,/.0,/) {…}
 /* ptr to char; set equal to address of
    buf[25] (i.e. pointing to buf[25]) */
  char *sptr = &buf[25];
 /* store 'c' into loc ptd to by sptr */
  *sptr = 'c';
 /* set sptr pointing to next loc in buf */
  ++sptr;
 /* ptr to function returning int */
  int (*fptr) ();
```

FUNCTIONS

Functions follow this format :

```
  ret_type name (arg1_decl, arg2_decl, ... )
     {
        local_var_decl
        statement
        ...
        return value;
     }
```

Functions can be declared **extern** (default) or **static**.
static functions can be called only from the file in which they are defined.
ret_type is the return type for the function, and can be **void** if the function returns no value.

EXAMPLE :

```
/* fn to find the length of a character string */
  int strlen (char *s)
     {
        int length = 0,/;
        while ( *s++ )
           ++length;
        return length;
     }
```

STRUCTURES

A structure **sname** of specified members is declared with a statement of the form:

```
  struct sname
     {
        member_declaration;
        ...
     } variable_list;
```

Each member declaration is a type followed by one or more member names.
An **n**-bit wide field **mname** is declared with a statement of the form:

```
   type mname:n;
```

If **mname** is omitted, **n** unnamed bits are reserved; if **n** is also zero, the next field is aligned on a word boundary. **variable_list** (optional) declares variables of that structure type.
If **sname** is supplied, variables can also later be declared using the format:

```
  struct sname variable_list;
```

EXAMPLE:

```
 /* declare complex struct */
  struct complex
     {
        float real;
        float imaginary;
     };
 /* define structures */
  struct complex c1 = { 5.0,/ , 0,/.0,/ };
  struct complex c2, csum;
  c2 = c1; /* assign c1 to c2 */
  csum.real = c1.real + c2.real;
```

UNIONS

A union **uname** of members occupying the same area of memory is declared with a statement of the form :

```
  union uname
     {
        member_declaration;
        …
     } variable_list;
```

Each member declaration is a type followed by one or more member names; **variable_list** (optional) declares variables of the particular union type. If **uname** is supplied, then variables can also later be declared using the format:

```
   union uname variable_list;
```

NOTE: unions cannot be initalized.

szandi@hit.bme.hu

ENUM DATA TYPES

An enumerated data type **ename** with values **enum1,enum2,…** is declared with a statement of the form :

```
  enum ename { enum1, enum2, … } variable_list;
```

The optional **variable_list** declares variables of the particular enum type. Each enumerated value is an identifier optionally followed by an equals sign and a constant expression. Sequential values starting at **0,/** are assigned to these values by the compiler, unless the **enum=value** construct is used.
If **ename** is supplied, then variables can also be declared later using the format:

```
   enum ename variable_list;
```

EXAMPLES:

```
 /* define boolean */
  enum boolean { false, true };
 /* declare variable and inicialize value */
  enum boolean done = false;
  if (done==true) {…} /* test value */
```

FORMATTED OUTPUT

printf is used to write data to standard output (normally, your terminal). To write to a file, use **fprintf**; to 'write' data into a character array, use **sprintf**. The general format of a printf call is :

```
   printf (format, arg1, arg2, … )
```

where **format** is a character string describing how **arg1,arg2,…** are to be printed. The general format of an item in the format string is :

```
   %[flags][size][.prec]type
```

flags:

-	left justify value (default is right justify)
+	precede value with a **+** or – sign
space	precede positiv value with a space
#	precede octal value with **0,/**, hex value with **0,/x**; force display of decimal point for float value, and leave trailing zeros for type **g** or **G**
0,/	display leading zeros

size : is a number specifying the minimum size of the field; * instead of number means next arg (must be type of int) to printf specifies the size

prec : is the minimum number of digits to display for **int**s; number of decimal places for **e** and **f**; max. number of significant digits for **g**; max. number of chars for **s**; * instead of number means next arg (int) to printf specifies the precision

type : specifies the type of value to be displayed per the following character codes:

arg	dec.	oct.	hex.	HEX.	±d.dd	±d.dd**e**±dd
short	**hd**					
unsigned short	**hu**	**ho**	**hx**	**hX**	*default precision is 6 decimal digits*	
int	**d**					
unsigned int	**u**	**o**	**x**	**X**		
long	**ld**					
unsigned long	**lu**	**lo**	**lx**	**lX**		
float, double					**f**	**e**
long double					**Lf**	**Le**

i	same as **d**
p	a pointer, void * (implementation-defined)
n	store how many characters have been displayed, arg is int *, no output
hn	store how many characters have been displayed, arg is short *, no output
ln	store how many characters have been displayed, arg is long *, no output
E	same as **e** except display **E** before exponent instead of **e**
g	a double in **f** or **e** format, whichever takes less space w/o losing precision
G	a double in **f** or **E** format, whichever takes less space w/o losing precision
c	a char
s	a null-terminated char string (null not required if precision is given)
%	% itself

NOTES:

characters in the format string not preceded by % are literally printed;
floating point formats display both floats and doubles;
integer formats can display chars, short ints or ints.

EXAMPLE:

```
  printf("%o + %#X is %+0,/*d",31,31,5,31+31);
    Produces:  37 + 0,/X1F is +0,/0,/62
  printf("%f %g %#.0,/f
%.2g",3.14,3.14,3.14,3.14);
    Produces:  3.140,/0,/0,/0,/ 3.14 3. 3.1
```

FORMATTED INPUT

scanf is used to read data from standard input. To read data from a particular file, use **fscanf**. To 'read' data from a character array, use **sscanf**. The general format of a **scanf** call is :

```
   scanf (format, arg1, arg2, … )
```

where **format** is a character string describing the data to be read and **arg1,arg2,…** point to where the read-in data are to be stored. The format of an item in the format string is :

```
   %[*][size]type
```

***** : specifies that the field is to be skipped and not assigned (i.e., no corresponding ptr is supplied in arg list)

size : a number giving the maximal size of the field

type : indicates the type of value being read :

arg is ptr to	dec.	oct.	hex.	HEX.	±d.dd *or* ±d.dd**e**±dd
short	**hd**				
unsigned short	**hu**	**ho**	**hx**	**hX**	
int	**d**				
unsigned int	**u**	**o**	**x**	**X**	
long	**ld**				
unsigned long	**lu**	**lo**	**lx**	**lX**	
float					**f,e,E,g,G**
double					**lf,le,lE,lg,lG**
long double					**Lf,Le,LE,Lg,LG**

i	same as **d**
p	pointer (same as in **printf**), arg type is void **
n	store number of chars have been matched, arg is int * , no input
hn	store number of chars have been matched, arg is short * , no input
ln	store number of chars have been matched, arg is long * , no input
c	single character, arg is char[ ]
s	string of chars terminated by a white-space character, arg is char[ ]
%	% itself
[…]	string of chars terminated by any char not enclosed between the **[** and **]**; if first char in brackets is **^**, then following chars are string terminators instead.

NOTES:

A scan function returns when:
– It reaches the terminating null in the format string.
– It cannot obtain additional input characters to scan.
– A conversion fails.

Any chars in format string not preceded by **%** will literally match chars on input (e.g. **scanf("value=%d",&ival);** will match chars **"value="** on input, followed by an integer which will be read and stored in **ival**.

Whitespace in format string matches the longest possible sequence of the zero or more whitespace characters on input.

EXAMPLE:

```
   sscanf("12Free of charge 21",
          "%X%c%*[^ab]%2s%d",&i,&c,text,&j);
```

will return **3** and **i=30,/3**, **c='r'**, **text="ar"**; **j** remains unchanged.

ESCAPE CHARACTERS

\b	Backspace (*BS*)	\\	Backslash (\\)
\f	Form feed (*FF*)	**\*nnn***	Octal character value (*n*: **0,/-7**)
\n	Newline (*NL*)	**\x*hh***	Hexadecimal character value (*h*: **0,/-9,a-f,A-F**)
\r	Carriage return (*CR*)		
\t	Horizontal tab (*HT*)	**\\"**	Double quote (**"**)
\v	Vertical tab (*VT*)	**\\'**	Single quote (**'**)
\a	Bell (*BEL*)	**\\?**	Question mark (**?**)

LIBRARY FUNCTIONS AND MACROS

Function argument types :

```
int c; /* char */            int n,n1,n2;
unsigned int u;              long l,l1,l2;
double d,d1,d2;              char *s,*s1,*s2;
FILE *f;                     size_t su,su1,su2;
time_t tl,tl1,tl2;           fpos_t fl;
void *v,*v1,*v2;             va_list ap;
```

char and **short** are converted to **int** when passed to functions;
float is converted to **double**.
/…/ return code on error (…) return code on success

Character classification	ctype.h
int isalnum(c)	TRUE if **c** is any alphanumeric char
int isalpha(c)	TRUE if **c** is any alphabetic char
int iscntrl(c)	TRUE if **c** is any control char
int isdigit(c)	TRUE if **c** is any decimal digit **0,/-9**
int isgraph(c)	TRUE if **c** is any printable char except *space*
int islower(c)	TRUE if **c** is any lowercase char
int isprint(c)	TRUE if **c** is any printable char including. *space*
int ispunct(c)	TRUE if **c** is neither a control nor alphanum. char
int isspace(c)	TRUE if **c** is one of the whitespace characters: space, *FF, NL, CR, HT, VT*
int isupper(c)	TRUE if **c** is any uppercase char
int isxdigit(c)	TRUE if **c** is any hexadecimal digit **0,/-9,A-F,a-f**
int tolower(c)	convert **c** to lowercase
int toupper(c)	convert **c** to uppercase

Data conversion	stdlib.h
double atof(s)	ASCII to double conversion /HUGE_VAL,0,//
int atoi(s)	ASCII to int conversion
long atol(s)	ASCII to long conversion
double strtod(s1,*s2)	ASCII to double conversion; on return, ***s2** points to char in **s1** that terminated the scan/0,//
long strtol(s1,*s2,n)	ASCII to long conversion, base **n**; on return, ***s2** points to char in **s1** that terminated the scan /0,//
unsigned long strtoul(s1,*s2,n)	ASCII to unsigned long conversion (see **strtol**)

File handling and input/output	stdio.h
void clearerr(f)	reset error (incl. EOF) on file
int fclose(f)	close file /EOF/ (0,/)
int feof(f)	TRUE if end-of-file on **f**
int ferror(f)	TRUE if I/O error on **f**
int fflush(f)	write buffered output to **f** /EOF/ (0,/)
int fgetc(f)	read next char from **f** /EOF/
int fgetpos(f,*fl)	get the file position indicator to **fl**/TRUE/(0,/)
char *fgets(s,n,f)	read **n-1** chars from **f** unless newline or end-of-file reached; newline is stored in **s** if read /NULL/
FILE *fopen(s1,s2)	open file **s1**, mode **s2**: "w"=write, "r"=read, "a"=append, "b"=binary, "+"=update /NULL/
int fprintf(f,s,…)	write args to **f** using format **s** (see **printf**)
int fputc(c,f)	write **c** to **f** ; rtn **c** /EOF/
int fputs(s,f)	write **s** to **f** /EOF/ (≥0,/)
size_t fread (v,su1,su2,f)	read **su2** data items from **f** into **v**; **su1** is number bytes of each item /0,// (*bytes read*/su1)
FILE *freopen(s1,s2,f)	close **f** and open **s1** with mode **s2** (see **fopen**)
int fscanf(f,s,…)	read args from **f** using format **s** (see **scanf**)
int fseek(f,l,n)	position file pointer; if **n=SEEK_SET**, **l** is offset from beginning; if **n=SEEK_CUR**, from current pos.; if **n=SEEK_END**, from end of file /TRUE/ (0,/)
int fsetpos(f,*fl)	sets the file position to **fl** (0,/) /TRUE/
long ftell(f)	current offset from the beginning of the file /-1L/
size_t fwrite(v,su1, su2, f)	write **su2** data items to **f** from **v**; **su1** is number of bytes of each item /0,// (*bytes written*/su1)
int getc(f)	read next char from **f** /EOF/
int getchar()	read next char from **stdin** /EOF/
char *gets(s)	read chars into **s** from **stdin** until newline or eof reached; newline not stored /NULL/
void perror(s)	write **s** followed by descr. of last err. to **stderr**
int printf(s,…)	write args to **stdout** per format **s**; return number of characters written /<0,//
int putc(c,f)	write **c** to **f** ; rtn **c** /EOF/
int putchar(c)	call **fputc(c,stdout)**
int puts(s)	write **s** and newline to **stdout** /EOF/ (≥0,/)
int remove(s)	removes the file named **s** (0,/) /TRUE/
int rename(s1,s2)	rename the file named **s1** to file **s2** (0,/) /-1/
void rewind(f)	rewind **f**; calls **fseek(f,0,/L,SEEK_SET)**
int scanf(s,…)	read args from **stdin** per format **s**; return number of values read or EOF
void setbuf(f,s)	if **s<>NULL** calls **setvbuf(f,s,_IOFBF,BUFSIZ)** otherwise calls **setvbuf(f,NULL,_IONBF,BUFSIZ)**
int setvbuf(f,s,n,su)	sets buffering mode for **f**, the buffer is **s** with size **su**, **n** must be one of **_IOFBF** (full buffering), **_IOLBF** (line buffering), **_IONBF** (no buffering) (0,/) /TRUE/
int sprintf(s1,s2,…)	write args to buffer **s1** per format **s2** (see **printf**)
int sscanf(s1,s2,…)	read args from **s1** per format **s2**; (see **scanf**)
FILE *tmpfile()	create temporary file, open with "wb+" mode; return ptr to it /NULL/
char *tmpnam(s)	generate temporary file name; place result in **s** if **s<>NULL** (**L_tmpnam** size buffer); rtn ptr to name
int ungetc(c,f)	insert **c** back into file **f** (as **c** wasn't read) /EOF/
int vfprintf(f,s,ap)	see **vprintf** and **fprintf**
int vprintf(s,ap)	same as **printf** with variable argument list **ap**; **va_start** must be called before and **va_end** after the function
int vsprintf(s1,s2,ap)	see **vprintf** and **sprintf**

Math math.h,stdlib.h(*)

int errno (errno.h) detects range error (ERANGE) and domain error (EDOM).

int abs(n)	*	absolute value of **n**
double acos(d)		arccosine of **d** /0,// [0,/,π]
double asin(d)		arcsine of **d** /0,// [-π/2,+π/2]
double atan(d)		arctangent of **d** [-π/2,+π/2]
double atan2(d1,d2)		arctangent of **d1/d2** [-π,+π]
double ceil(d)		smallest integer not less than **d**
double cos(d)		cosine of **d** (**d** in radians)
double cosh(d)		hiperbolic cosine of **d**
div_t div(n1,n2)	*	computes the quotient (**.quot**) and remainder (**.rem**) of division **n1/n2**
double exp(d)		***e*** to the **d**-th power /HUGE_VAL/
double fabs(d)		absolute value of **d**
double floor(d)		largest integer not greater than **d**
double fmod(d1,d2)		**d1** modulo **d2**
double frexp(d,*n)		returns *x* in interval [½,1), and **d=***x****2$^{\mathbf{n}}$**
long labs(l)	*	absolute value of **l**
double ldexp(d,n)		**d*2$^{\mathbf{n}}$**
ldiv_t ldiv(l1,l2)	*	computes the quotient (**.quot**) and remainder (**.rem**) of division **l1/l2**
double log(d)		natural log of **d** /0,//
double log10,/(d)		log base 10,/ of **d** /0,//
double modf(d1,*d2)		rtn *x* such that **d1=***x***+d2**, *x* in [0,1), **d2** integer
double pow(d1,d2)		**d1** to the **d2**-th power /0,/,HUGE_VAL/
int rand()	*	random number in range [0,/,RAND_MAX]
double sin(d)		sine of **d** (**d** in radians)
double sinh(d)		hyperbolic sine of **d**
double sqrt(d)		square root of **d** /0,//
void srand(u)	*	reset random number generator to **u**
double tan(d)		tangent of **d** (radians) /HUGE_VAL/
double tanh(d)		hyperbolic tangent of **d**

Memory allocation and manipulation		string.h,stdlib.h(*)
void *calloc(su1, su2)	*	allocate space for **su1** elements; each **su2** bytes large and set to **0,/** /NULL/
void free(v)	*	free block of space pointed to by **v**
void *malloc(su)	*	allocate **su** bytes and return ptr to it /NULL/
void *memchr(v,c,su)		return ptr in **v** of 1st incident of **c**, looking at **su** unsigned chars at most, or NULL if not found
int memcmp(v1,v2,su)		rtn <**0,/**, =**0,/**, >**0,/** if **v1** is lexicographically **<,=** or >**v2**, comparing up to **su** unsigned chars
void *memcpy(v1,v2,su)		copy **su** chars from **v2** to **v1** (**v1**, **v2** should not overlap); return **v1**
void *memmove (v1,v2,su)		copy **su** chars from **v2** to **v1** (**v1**, **v2** can overlap); return **v1**
void *memset(v,c,su)		set **su** unsigned chars ptd to by **v** to value **c**; return **v**
void *realloc(v,su)	*	change the size of block **v** to **su** and returns ptr to it /NULL/

Program contol		setjmp.h,stdlib.h(*)
void assert(*iexpr*)		if NDEBUG is not defined and ***iexpr*** is FALSE then write a diagnostic message to **stderr** and calls **abort()**; use assert.h header
void abort()	*	couse abnormal program termination
int atexit(void (*func)(void))	*	register **func** to be called by exit (0,/) /TRUE/
void exit(n)	*	terminate execution, returning exit status **n**
char *getenv(s)	*	rtn ptr to value of environment name **s** /NULL/
void longjmp (jmp_buf env,n)		restore environment from **env**; causes setjmp to return **n** if supplied or **1** if **n=0,/**
int setjmp(jmp_buf env)		save stack environment in **env**; (0,/) (see **longjmp**)
int system(s)	*	execute **s** as if it were typed at terminal; returns exit status /-1/

Searching and sorting	stdlib.h
void *bsearch(void *key, void *base, su1, su2, int (*cmp)(void *ck, void *ce))	binary search in array **base** (**su1** elements, each **su2** bytes large), using function **cmp** for comparison; **cmp** must return negativ if **ck<ce**, **0,/** if **ck=ce**, positiv if **ck>ce**
void qsort (void *base, su1, su2, int (*cmp)(void *ck, void *ce))	quick sort of array **base** (**su1** elements, each **su2** bytes large), using function **cmp** for comparison; (for **cmp** see **bsearch**)

String manipulation	string.h
char *strcat(s1,s2)	concatenate **s2** to end of **s1**; rtn **s1**
char *strchr(s,c)	return ptr to 1st occurence of **c** in **s** /NULL/
int strcmp(s1,s2)	compare **s1** and **s2**; returns **<0,/**, **0,/**, **>0,/** if **s1** lexicographically **<s2, =s2, >s2**
char *strcpy(s1,s2)	copy **s2** to **s1**; rtn **s1**
size_t strcspn(s1,s2)	search the first **s1[i]** that equals any element of **s2**; rtn **i**
char *strerror(n)	return a pointer to string that message cornresponds to errorcode **n**
size_t strlen(s)	length of **s** (not incl. NULL)
char *strncat (s1,s2,su)	concatenate at most **su** chars from **s2** to end of **s1**; rtn **s1**
int strncmp(s1,s2,su)	compare at most **su** chars from **s1** to **s2**; (see **strcmp**)
char *strncpy(s1,s2,su)	copy at most **su** chars from **s2** to **s1**; if **s2** is shorter than **su**, null bytes are appended; rtn **s1**
char *strpbrk(s1,s2)	searches the first **s1[i]** that equals any element of **s2**; return **&s1[i]**
char *strrchr(s,c)	return pointer to last occurence of **c** in **s** /NULL/
size_t strspn(s1,s2)	search the first **s1[i]** that equals none of the element of **s2**; rtn **i**
char *strstr(s1,s2)	search the first substring in **s1** that matches **s2**
char *strtok(s1,s2)	break **s1** into tokens delimited by **s2**; from the second call **s1**=NULL; **s2** may different from call to call; return the ptr to token or NULL

Time	time.h
char *asctime(*tm)	convert **tm struct** to string; rtn ptr to it
clock_t clock()	CPU time in 1.0,//CLOCKS_PER_SEC seconds since program startup /-1/
char *ctime(*tl)	convert time ptd to by **tl** to string; rtn ptr to it
double difftime(tl1,tl2)	difference **tl1-tl2** in seconds
struct tm *gmtime(*tl)	convert time pointed to by **tl** to Universal Time Coordinated (UTC) (formerly GMT)
struct tm *localtime(*tl)	convert time pointed to by **tl** to local time
time_t mktime (struct tm *tptr)	alters **tptr** to represent an equivalent encoded local time /-1/
size_t strftime(s1, su, s2, struct tm *tptr)	write **tptr** to buffer **s1** per format **s2**; buffer size is **su**; rtn number of characters stored /0,//
time_t time(*tl)	returns time & date in seconds; if **tl<>NULL**, time is stored in ***tl**; convert time returned with **ctime**, **localtime** or **gmtime** /-1/

Variable-type and number of arguments	stdarg.h
***type* va_arg(ap,*type*)**	get next argument; **ap** must be initialized by **va_start**; the argument type must be ***type***
void va_end(ap)	end variable argument list
void va_start(ap,*pN*)	start variable argument list; ***pN*** is the parameter just before the **(...)** in the function prototype

COMMAND LINE ARGUMENTS

Arguments typed in on the command line when a program is executed are passed to the program through **argc** and **argv**.
argc is a count of the number of arguments +1;
argv is an array of character pointers that point to each argument.
argv[0,/] points to the name of the program executed.
argv[argc] equal **NULL** pointer.
Use **sscanf** to convert arguments stored in **argv** to other data types. For example:

```
   check phone 35.79
```

starts execution of a program called **check**, with :

```
   argc = 3
   argv[0,/] = "check"    argv[2] = "35.79"
   argv[1] = "phone"      argv[3] = NULL
```

To convert number in **argv[2]**, use **sscanf**. For example :

```
   int main (int argc, char *argv[])
      {  float amount;
        … sscanf (argv[2],"%f",&amount); … }
```