

e-lib Programmers' Reference Manual

Karel Kubat
e-tunity

2002 ff.

Contents

1	Introduction	6
1.1	For e-lib V3 users	6
1.2	Installation of e-lib	6
1.2.1	General preparations	6
1.2.2	Preparations for shared libraries	7
1.2.3	Compiling and installing	7
1.2.4	FastCGI	7
1.3	Using the functions, classes and libraries	7
1.3.1	Using the C functions	8
1.3.2	Using the C++ classes	8
1.3.3	Using getline	8
2	C Function Reference	9
2.1	base64_decode - decode a base64-format buffer	9
2.2	base64_encode - encode a string to base64-format	10
2.3	bth_init - initialize a BTHash structure	11
2.4	bth_insert - insert data into a BTHash storage given a unique identifier	11
2.5	bth_lookup - look up data in a BTHash storage given an identifier	12
2.6	btn_insert - insert a node into a binary tree	12
2.7	btn_lookup - look up a node in a btree	13
2.8	cgi_escape - escape a string to CGI format	14
2.9	cgi_getdata - convert CGI-sent data to variables and un-escaped values	14
2.10	cgi_ismultipart - determine whether a CGI buffer holds a multipart message	16
2.11	cgi_setpostfile - define the input stream for the CGI POST method	16
2.12	cgi_unescape - un-escape a CGI-value string	17
2.13	cgi_unescape_buf - un-escape a CGI value buffer	18
2.14	checksum - determine checksum of a file	18
2.15	checksum_adler32 - determine checksum of a buffer	19

2.16	<code>cmm_allocated</code> - determine how many bytes of memory are allocated	19
2.17	<code>cmm_blocksize</code> - determine the size of a CMM-allocated block	20
2.18	<code>cmm_cleanup</code> - deallocate blocks from a given level upwards	20
2.19	<code>cmm_free</code> - free <code>cmm_malloc()</code> -d memory	21
2.20	<code>cmm_malloc</code> - allocate memory, given some level	22
2.21	<code>cmm_realloc</code> - reallocate a <code>cmm_malloc()</code> -d block of memory	22
2.22	<code>cmm_setlevel</code> - set the level for cmm operations	23
2.23	<code>cmm_storeblock</code> - store an allocated memory block in the cmm administration	23
2.24	<code>copyfile</code> - copy source file onto destination file	24
2.25	<code>copyright</code> - show standard copyright notice	25
2.26	<code>copyright_contact</code> - redefine the "contact" string in a copyright message	26
2.27	<code>copyright_owner</code> - redefine the "owner" string in a copyright message	26
2.28	<code>copyright_setoutput</code> - define output file for the <code>copyright()</code> function	26
2.29	<code>daemonize</code> - fork a daemon process	27
2.30	<code>elib_require</code> - check e-lib version, stop if too low	29
2.31	<code>elib_version</code> - determine the version of this e-lib	30
2.32	<code>equal</code> - compare first parts of 2 strings without regard to casing	30
2.33	<code>error</code> - print error message including program name and exit	31
2.34	<code>fchecksum_adler32</code> - determine checksum of a file	31
2.35	<code>fgetline</code> - read a line from a file, allocate buffer and return it	31
2.36	<code>filetype</code> - determine the type of a file	32
2.37	<code>fin_push</code> - push a block of data back into a file, to be read by <code>fin_read()</code>	33
2.38	<code>fin_read</code> - read a block of data from a file, possibly a pushed-back block	34
2.39	<code>get_log_setup</code> - return currently active logging parameters	34
2.40	<code>hash</code> - compute hash-index for a string	35
2.41	<code>hashbuf</code> - compute hash-index for a buffer	35
2.42	<code>html_escape</code> - convert special chars in a buffer to HTML-escape sequences	36
2.43	<code>html_unescape</code> - convert escaped HTML to original characters	37
2.44	<code>image_size</code> - determine the size of an image	37
2.45	<code>lap_end</code> - stop a task stopwatch and log the results	38
2.46	<code>lap_setup</code> - define stopwatch laptime log file	39
2.47	<code>lap_start</code> - start the stopwatch mechanism	40
2.48	<code>log_dontflush</code> - suppress log file flushing when <code>logmsg()</code> is called	40
2.49	<code>logmsg</code> - log messages to a log file, defined by <code>log_setup()</code>	41
2.50	<code>log_rotate</code> - rotate a log file if it exceeds given limits	41
2.51	<code>log_setup</code> - define logging options	42
2.52	<code>log_timestamp</code> - enable or disable timestamping of log entries	43

2.53	log_wrap - enable or disable log line-wrapping	43
2.54	make_socket - create a server-side TCP/IP socket for a port	44
2.55	md5 - compute an MD5 digest	44
2.56	ml_decode - decode a *ML representation into a "normal" buffer	44
2.57	ml_encode - encode a buffer into *ML representation	45
2.58	mmsg_body - return n'th body of a parsed mail message	45
2.59	mmsg_destroy - deallocate a MailMsg handle	46
2.60	mmsg_fileparse - parse a mail message from file	46
2.61	mmsg_fparse - parse a mail message a file handle	47
2.62	mmsg_header - return header of a mail message	47
2.63	mmsg_headerval - retrieve value of a header variable	48
2.64	mmsg_new - initialize a MailMsg handle	48
2.65	mmsg_parse - parse a mail message	49
2.66	mmsg_partheaderval - retrieve header value of a mail message part	50
2.67	mmsg_parts - return number of parts in a mail message	50
2.68	mmsg_reset - reset a MailMsg handle	51
2.69	mmsg_setheader - set or modify a mail message header	51
2.70	movefile - move a source file to a destination file	52
2.71	msg - show message on stderr when in verbose mode	53
2.72	msg_and_log - log a message and optionally show verbose message	54
2.73	msg_on - increase program verbosity by 1	54
2.74	out_of_memory - the "out of memory" handler function	55
2.75	ps_freesnapshot - Deallocate memory occupied by a process snapshot	55
2.76	ps_makesnapshot - make a snapshot of the Unix process table	56
2.77	ps_sortsnapshot - sort a process list snapshot	57
2.78	random_ulong - generate random unsigned long number	57
2.79	rc_clean - reset resource-read values	58
2.80	rc_read - read resource file	58
2.81	rc_setting - return option/value setting, read via rc_read()	59
2.82	rc_setval - add value to resource-read set	60
2.83	rc_value - determine value of a resource file variable	60
2.84	rx_find - find a regular expression in a string	61
2.85	rx_ignorecase - ignore casing in regular expression matching	61
2.86	rx_match - match regular expression and a string	62
2.87	set_memerr - set the "out of memory" handler function	62
2.88	setprogname - set the program name for error() and msg()	63
2.89	sf_sleep - signal-free sleep implementation	63

2.90	sig_catch - flag-based signal handler	64
2.91	skipline - skip to the next line in a string buffer	65
2.92	stab_addsprintf - add a string to a string table using sprintf()-like capabilities	65
2.93	stab_addstr - add string to string table	66
2.94	stab_find - find string in a string table	67
2.95	stab_free - discard memory used by a Strtab, reset member fields	67
2.96	stab_ifind - find string in a string table, without regard to casing	67
2.97	stab_maxlen - determine max string length in a string table	68
2.98	stab_popstr - pop a string off a string table	68
2.99	str_add - add string to reallocatable string buffer	69
2.100	str_addchar - add a single character to a relocateble string buffer	69
2.101	str_equal - compare strings without regard to casing, returns matched characters	69
2.102	stricmp - compare two strings without regard to casing	70
2.103	stripstring - strip leading and trailing spaces in a buffer	70
2.104	strireplace - replace a substring in a string	71
2.105	stristr - search for substring in a string without regard to casing	71
2.106	strnicmp - compare two strings over a given length, ignoring casing	71
2.107	str_printf - fill a string using printf capabilities	72
2.108	str_short - return shortened duplicate of string, with ... appended	72
2.109	strupr - convert string to upper case	73
2.110	str_vprintf - fill a string using vprintf capabilities	73
2.111	suggest_ext - suggest an appropriate extension for a file name	73
2.112	supply_ext - supply an appropriate extension for a file name	74
2.113	tmp_clean - clean up files created by tmp_file()	75
2.114	tmp_file - construct name for a temporary file	75
2.115	tmp_fileprefix - define the file prefix for temporary files	76
2.116	tmp_fopen - open a tmp file for writing	76
2.117	xcalloc - allocate memory or issue an error	77
2.118	xmalloc - Allocate memory or issue an out-of-memory error	77
2.119	xrealloc - Reallocate memory or issue an error	78
2.120	xstrdup - Duplicate in memory a string or issue an error	78
2.121	zip_compress - compress a buffer	79
2.122	zip_decompress - decompress a buffer	79
2.123	zip_threshold - set the value for compression routines	80

3	C++ Class Reference	81
3.1	eArray - Storage in an array format	81
3.1.1	Synopsis	81
3.1.2	Description	82
3.1.3	Storing and retrieving	82
3.2	eHttpHdr - HTTP Request Header handling	82
3.2.1	Synopsis	82
3.2.2	Description	83
3.3	eHttpRequest - HTTP Request handling	84
3.3.1	Synopsis	84
3.3.2	Description	85
3.4	eObj - basic class type of the e++ library	86
3.4.1	Synopsis	86
3.4.2	Description	86
3.5	eProgram - basic program functions	86
3.5.1	Synopsis	86
3.5.2	Description	87
3.6	eStorable - base class for storable objects	88
3.6.1	Description	88
3.7	eStr - String class	89
3.7.1	Synopsis	89
3.7.2	Description	90
3.8	eStrArr - Storage of eStr in array format	91
3.8.1	Synopsis	91
3.8.2	Description	92
3.9	eSysinfo - System information	92
3.9.1	Synopsis	92
3.9.2	Description	92
4	Changelog	93

Chapter 1

Introduction

This is the Programmers' Reference Manual to *e-lib*, e-tunity's "miscellaneous tools library". *e-lib* contains C and C++ functions which we use in many of our programs. You'll probably need *e-lib* when building our software.

This chapter provides information on the installation and usage of e-lib. The following chapters are the core of the programmers' reference.

1.1 For e-lib V3 users

Important: If you're using *e-lib* version 3.XX, then this version should be considered as a major upgrade. *e-lib* 4 is a shared object archive `libe.so`, while older versions were statically linkable archives (`libe.a`).

This means that new versions of *e-lib* can now be installed without having to recompile the programs that use *e-lib*. We suggest that, during the first installation of *e-lib* V4, the file `/usr/e/lib/libe.a` is unlinked. This is the static library of version 3.

1.2 Installation of e-lib

The installation of *e-lib* is very straight-forward. If you're on a i386-based Linux platform, then the easiest way is to get the RPM named *e-lib-X.YY-R-i386.rpm* from <http://public.e-tunity.com>. Here, X.YY is a version ID, and R is a release ID. The RPM is installed by user *root* using `rpm -i rpmfile`.

Alternatively, you might want to build *e-lib* from source. In that case, proceed as follows:

1.2.1 General preparations

First of all, obtain the source archive from <http://public.e-tunity.com>. Unpack the archive in a suitable place.

Next, edit `etc/Makefile.def` and search for the symbol `PREFIX`. This is the global installation path. When not present, `$EBASE` or `/usr/local` are taken.

Make sure that you create the following directories under the prefix directory:

- `lib/` for libraries;
- `include/` for header files;
- `man/` for manual pages.

1.2.2 Preparations for shared libraries

If your operating system has a `gcc` compiler that supports `-shared` to build shared libraries, then read on. Otherwise skip to section 1.2.3.

Edit `/etc/ld.so.conf` and add the directory `/usr/e/lib` to the shared library search path. This step must be run by the user `root` and is only necessary during a 'fresh' install; i.e., on a system where `e-lib` isn't available yet. After this, the command `ldconfig` must be run by `root`.

Alternatively, the environment variable `LD_LIBRARY_PATH` may be set to point to the directory `/usr/e/lib`. We suggest however to change `/etc/ld.so.conf`, that way, program execution is not dependent on an environment setting. This may be relevant in, e.g., `cron` jobs, where the full login environment is not available.

1.2.3 Compiling and installing

`chdir` into the created directory `e-lib`, and `make install`. The installation will install all headers and shared or static libraries (depending on your system).

The prerequisites for a succesful full build of `e-lib` are a `C` and `C++` compiler.

In order to create the manpages and documentation, you will need the package `Yodl`. The installation is triggered by `make installdoc`.

1.2.4 FastCGI

FastCGI is an Apache module that allows CGI programs to run as server processes, with minimal code changes. CGI programs that require `e-lib` can be built as FastCGI server applications with the use of `e-lib` in FastCGI mode.

The construction of `e-lib` for FastCGI is not enabled by default. Contact `e-tunity` if you want to use `e-lib` in FastCGI programs; in that case, we'll happily let you know how to cook a FastCGI-`e-lib` version.

1.3 Using the functions, classes and libraries

Once installed, the following libraries are available:

- `libe.so` and `libefcgi.so`: these libraries contain the `C` functions, in plain mode and in FastCGI mode. The archives are 'shared libraries', which makes updates of `e-lib` functionality easier.
- `libepp.so`: this shared library holds the `C++` classes.
- `libgetline.a`: a separate small `C` library for light-weight command-line editing support.

1.3.1 Using the C functions

To use the C functions, the compile-time inclusion of `e-lib.h` is required. During linkage, the flags `-L/usr/e/lib -le` or `-L/usr/e/lib -lefcgi` are needed. (The flag `-lefcgi` of course selects the FastCGI version of `e-lib`.)

The thus linked program will connect to `libe.so` or `libefcgi.so` during the program execution. The shared object libraries are not directly linked to a program.

1.3.2 Using the C++ classes

To use the C++ classes, the class header must be included (e.g., `e-strarr.h`). During linkage, the flags `-L/usr/e/lib -lepp` are required.

1.3.3 Using `getline`

To use the functionality of `getline`, the header `getline.h` is included during compilation. During linkage, the flags `-L/usr/e/lib -lgetline` are needed. For more information on `getline`, please see the manpage (run `man getline` after installing `e-lib`).

Chapter 2

C Function Reference

2.1 base64_decode - decode a base64-format buffer

Prototype: unsigned char *base64_decode (char const *srcstr, int srclen, int *declenp)

Arguments: `srcstr` is the base64-encoded buffer to decode. This buffer may contain newlines and the padding character `=`.

`srclen` is the length of `srcstr`. This argument may be 0, in that case, `srcstr` must be 0-terminated and the length will be determined in `base64_decode()`

`declenp` is a pointer to an `int`. When non-0, the length of the decoded buffer will be stored in this value.

Returns: Pointer to allocated memory, where the base64-decoded version of the source buffer is stored. The returned buffer is a 0-terminated string.

Description: This function decodes a base64-encoded buffer to its original version. Note that the return value is allocated memory; use `cmm_free()` to de-allocate it.

See also `base64_encode()`.

Example: The following code snippet shows how to encode and decode a message:

```
unsigned char *encoded;
char *decoded;

encoded = base64_encode ("Hello World!", 0, 0);
decoded = base64_decode (encoded, 0, 0);
if (strcmp ("Hello World!", decoded))
    error ("Something went very wrong!\n");
```

The following code snippet decodes an external buffer `enobuf`, having length `enclen`. The result is stored in the integer `declen`.

```
extern unsigned char *enobuf;
extern int enclen;
```

```
int declen;
char *decbuf;

base64_decode (encbuf, enclen, &declen);
printf ("Encoded buffer having %d bytes "
        "is now decoded into %d bytes\n",
        enclen, declen);
```

2.2 base64_encode - encode a string to base64-format

Prototype: `char *base64_encode(const unsigned char *srcstr, int srclen, int *enclen)`

Arguments: `srcstr` : the source string to encode

`srclen` : the length of the source string. When this argument is 0, `base64_encode` will self-determine the length by assuming that `str` is a 0-terminated string.

`enclen` : A pointer to an `int`. When non-0, this value ' will be set to the length of the encoded buffer.

Returns: Pointer to allocated memory, where the base64-encoded version of the source string is located. The allocated buffer is a 0-terminated string.

Description: This function encodes a character buffer to base64 format. Note that the return value points to allocated memory, use `cmm_free()` to de-allocate it. The encoded buffer is formatted in such a way that the lines have a maximum of 76 characters. When necessary, the padding character `=` is used.

See also `base64_decode()`.

Example: The following code snippet shows the representation of the string `Hello World`.

```
char *ret = base64_encode ("Hello World!", 0, 0);

printf ("'Hello World' leads to: %s\n", ret);
cmm_free (ret);
```

The following code snippet encodes an arbitrary-length buffer `buf` having the length `buflen`. The buffer does not need to be 0-terminated. Also, the length of the output is stored.

```
extern char *buf;
extern int buflen;
int retlen;
char *ret;

ret = base64_encode (buf, buflen, &retlen);
printf ("Buffer having length %d expands to %d "
        "when encoded.\n", buflen, retlen);
```

2.3 `bth_init` - initialize a BTHash structure

Prototype: `BTHash *bth_init (int tabsize)`

Arguments: `tabsize` is the hash table size

Returns: pointer to an initialized BTHash structure, usable in other `bth_*()` functions

Description: This function initializes a BTHash structure (allocated in memory) and returns a pointer to it. The structure is then usable in other `bth_*()` functions.

The functions of the `bth_*()` family implement a hash algorithm for strings, where the hash table points to binary trees. BTHash-stored strings therefore have a storage that never runs out (except for out-of-memory errors). The relevant functions are:

- `bth_init(int tabsz)` - initializes;
- `bth_insert(char const *str, void *data, int dlen)` - inserts a string (key) and data;
- `bth_lookup(char const *str)` - looks up given a key

Example: For an example see the other `bth_*()` functions.

2.4 `bth_insert` - insert data into a BTHash storage given a unique identifier

Prototype: `BTNode *bth_insert (BTHash *h, char const *str, void *data, int dlen)`

Arguments: `h` is a pointer to an initialized BTHash structure, `str` identifies the information to insert, `data` is the information to insert, having length `dlen`

Returns: pointer to a binary tree node where the information is stored

Description: This function creates a new `BTNode`, duplicates the given information in the node, and stores a reference to the node in the hash-table specified by `h`.

If a node identified by the same `str` already exists, then its previous data contents are overwritten.

Note that the argument `h` must point to an initialized BTHash structure; use function `bth_init()` to initialize it.

Example: The following code snippet reads input from `stdin`. Each line is inserted in a BTHash structure, having the read buffer as key and an order counter as the data. Finally, the identifier `hello` is looked up. When found, its order number is shown.

```
int main (int argc, char **argv) {
    char buf[200], *cp;
    int count = 0, *data;
    BTNode *node;
    BTHash *h;
```

```
h = bth_init (50);
while (1) {
    fgets (buf, 199, stdin);
    if (feof (stdin))
        break;
    if ( (cp = strchr (buf, '\n')) )
        *cp = 0;

    ++count;
    node = bth_insert (h, buf, &count, sizeof(int));
}

if ( (data = bth_lookup (h, "hello")) )
    printf ("hello: %d\n", *data);

return (0);
}
```

2.5 bth_lookup - look up data in a BTHash storage given an identifier

Prototype: void *bth_lookup (BTHash *h, char const *str)

Arguments: str identifies the data to look up

Returns: pointer to the stored data, or 0 when the identifier str failed to match a node

Description: This function looks up a node in the binary tree / hash storage h. The return value 0 signals that no match was made.

A non-zero return value is a void*, pointing to the data area that was stored with the identifier str. **Note that** data doesn't point to the node itself, but to an amorph buffer. The caller is responsible for typecasting the return value into a meaningful pointer.

Example: For an example please see bth_insert().

2.6 btn_insert - insert a node into a binary tree

Prototype: BTNode *btn_insert (BTNode *top, char const *str, void *data, int dlen)

Arguments: top is the tree root, str is the key for the node to insert, data points to a data buffer 'belonging' to the node, dlen is the length of the data buffer

Returns: (new) pointer to the tree root

Description: This function inserts a new node into the binary tree pointed to by top. The node is identified by its key str, and the data area identified by data and dlen is included in the node.

When a node with the identifier `str` already exists, then the node's data are overwritten with the specified data.

The internal node organization is very simple: each node has a `left` and `right` pointer, a `char *str` pointing to the key, and a `void *data`. Note that the data are referred to by a `void*`, it is the caller's responsibility to typecast to the right format during traversal. Note also that there's a function `btn_lookup()` to find nodes.

Example: The following example inserts 5 nodes into a tree, and traverses the tree:

```
void print (BTreeNode what) {
    if (what) {
        print (what->left);
        printf ("%s -> %s\n", what->str, (char *) what->data);
        print (what->right);
    }
}

int main () {
    BTreeNode *top;

    top = btn_insert ("Rik", "Kanaalstr. 122", 15);
    top = btn_insert ("Karel", "Marskramerstr. 3", 18);
    top = btn_insert ("Harry", "Hazenveld 12", 13);
    top = btn_insert ("Peter", "Rubensstr. 29", 14);

    print (top);
    return (0);
}
```

2.7 btn_lookup - look up a node in a btree

Prototype: `void *btn_lookup (BTreeNode *top, char const *str)`

Arguments: `top` is the tree root, `str` is the identifying key

Returns: pointer to the node's data area, or 0 if `str` wasn't found in the tree

Description: This function tries to find the node identified by `str` in the tree pointed to by `top`. When found, a pointer to the node's data area is returned. When not found, 0 is returned.

Example: The following code snippet inserts 5 nodes and tries to find one:

```
BTreeNode *top;
char *data;

top = btn_insert ("Rik", "Kanaalstr. 122", 15);
top = btn_insert ("Karel", "Marskramerstr. 3", 18);
top = btn_insert ("Harry", "Hazenveld 12", 13);
top = btn_insert ("Peter", "Rubensstr. 29", 14);
```

```
if ( (data = btn_lookup ("Harry")) )
    printf ("Address: %s\n", data);
else
    printf ("No address found for Harry.\n");
```

2.8 cgi_escape - escape a string to CGI format

Prototype: char *cgi_escape (char const *buf)

Arguments: buf is the buffer to "escape"

Returns: pointer to allocated memory, holding the escaped data

Description: Changes the contents of the buf argument. The parts that cannot pass through a CGI gateway are re-coded to CGI standards. The recoding includes:

- Non-printables and special characters are re-written as %XY, where XY are two hexadecimal cyphers.
- Space characters are changed into +.

Note that the function returns a pointer to allocated memory; the caller is responsible for freeing that memory after usage. The freeing **must** occur using `cmm_free()` or `cmm_cleanup()`, because internally in this function, the `cmm*` () approach is used.

Example: The following code will print something like Hello+World%0A:

```
char
    normalbuf [] = "Hello World\n",
    *cgibuf = cgi_escape (normalbuf);

printf ("CGI format: %s\n"
        "Normal      : %s\n", cgibuf, normalbuf);
cmm_free (cgibuf);
```

2.9 cgi_getdata - convert CGI-sent data to variables and un-escaped values

Prototype: int cgi_getdata (Strtab *vars, Strtab *vals)

Arguments: vars is a pointer to a string table where variable names will be stored, the values will be stored in vals. **Note:** both string tables must be initialized to {0, 0}.

Returns: CGI_OK (0) no errors, data converted

CGI_NOMETHOD no \$REQUEST_METHOD found in the environment

CGI_BADMETHOD \$REQUEST_METHOD is neither GET nor POST

CGI_NOQRYSTR in the `GET` method, no `$QUERY_STRING` found in the environment
CGI_NOCNTLEN in the `POST` method, no `$CONTENT_LENGTH` found in the environment
CGI_BADCNTLEN `$CONTENT_LENGTH` does not yield a number
CGI_BADMULTI the CGI data are a `multipart/form-data` result, but the decoding has failed

Description: Converts CGI-sent data into variable names and their values. This function is typically called by a CGI handler.

The variables are stored in the `Strtab` structure pointed to by the first argument, their values in the structure pointed to by the second argument. The conversion is handled in such a way that `vars->s[0]` is the name of the first variable, and `vals->s[0]` is its corresponding value. The values, as stored in `vals` are 'un-escaped', meaning that `+` characters are converted into spaces, `%XY` sequences are converted to normal characters, etc..

This function handles both the `GET` and `POST` methods. In the `GET` method, the raw CGI data are expected in the environment variable `$QUERY_STRING`. In the `POST` method, the raw data are expected on `stdin`, while `$CONTENT_LENGTH` is the length of the raw data buffer. The input stream for the `POST` method may be redefined using `cgi_setpostfile()` (e.g., for debugging).

Multipart form data are also supported. E.g., consider the following form:

```
<form name="myform" action="/cgi-bin/myaction"
  enctype="multipart/form-data">
  <input type="text" name="mytext">
  <input type="file" name="myfile">
  <input type="submit">
</form>
```

In this case, the call to `cgi_getdata()` in the "myaction" program will result in the following variable / value pairs:

Index	var	val
0	mytext	entered text
1	myfile	temporary filename

When handling uploaded files, the sent contents are placed in a temporary file with a name returned by `tmp_file()`. That name is stored as the value.

See also: `cgi_ismultipart()`, `cgi_unescape()`, `tmp_file()`, `tmp_clean()`, `cgi_setpostfile()`

Example: The following example defines two `Strtab` structures for the CGI data and calls `cgi_getdata()` to fill them. The CGI data are then printed in "human readable" format.

```
Strtab var = {0, 0},
        val = {0, 0};
int i;

if (cgi_getdata (&var, &val))
    error ("CGI conversion failed\n");
```

```
printf ("%d variable/value pairs:\n", var.n);
for (i = 0; i < var.n; i++)
    printf ("%s = %s\n", var.s[i], val.s[i]);
```

Now, assuming that there is a CGI variable called `myfile`, the following code snippet prints the filesize and removes the file:

```
struct stat statbuf;

if ( (i = stab_ifind (var, "myfile")) > -1 ) {
    printf ("myfile found at index %d\n", i);
    if (! stat (val.s[i], &statbuf))
        printf ("file size: %d\n", statbuf.st_size);
    tmp_clean ();
}
```

2.10 `cgi_ismultipart` - determine whether a CGI buffer holds a multipart message

Prototype: `char *cgi_ismultipart (char const *buf, int len)`

Arguments: `buf` is a bare (unencoded) CGI-received buffer, `len` is the buffer length

Returns: Pointer to the magic key, allocated in memory, or 0 when the message pointed to by `buf` is not multi-part

Description: This function is used internally in, e.g., `cgi_getdata()`. When the buffer is in the format:

```
-----xy123czz
Content-Disposition: ....
..
.. data
..
```

then the buffer is recognized as a multipart header. The magic key that splits the buffer into variable/value pairs is copied in memory (here: `xy123czz`) and a pointer is returned.

Example:

2.11 `cgi_setpostfile` - define the input stream for the CGI POST method

Prototype: `void cgi_setpostfile (FILE *inf)`

Arguments: `inf` is the input stream to use instead of `stdin`

Returns:

Description: This function can be used in debugging sessions to redefine the input for POST methods. In normal situations this stream is `stdin`.

Note: This function does not set the variable `METHOD`, nor does it set `CONTENT_LENGTH`. To cause a subsequent call to read the alternative stream, the `METHOD` must be `POST` and `CONTENT_LENGTH` must be the number of characters waiting in the alternative stream.

See also `cgi_getdata()`.

Example: Assuming that we have stored CGI data in `/tmp/tempdata`, the following code snippet causes `cgi_getdata()` to read from that input:

```
FILE *inf;
Strtab var = {0,0},
           val = {0,0};
int ret;

if (inf = fopen ("/tmp/tempdata", "r"))
    cgi_setpostfile (inf);
if ( (ret = cgi_getdata (&var, &val)) )
    error ("Error %d while reading CGI data\n", ret);
```

2.12 `cgi_unescape` - un-escape a CGI-value string

Prototype: `char *cgi_unescape (char *buf)`

Arguments: `buf` points to the ascii-Z string to change from CGI format to C-string format

Returns: Pointer to the changed buffer

Description: Changes the contents of the `buf` argument. The CGI-specific parts are re-coded into a "normal" string. The recoding includes:

- `+` characters are changed to spaces
- `%XY` sequences are changed to their corresponding values, `XY` being a hexadecimal specification `a-la %0A` or `%0a` for a newline, etc.

Note that the buffer argument is changed in-place; i.e, the function does not duplicate it. There is no re-allocation, the buffer may or may not be in allocated memory. To change buffers where the (decoded) contents may contain literal 0-characters, use `cgi_unescape_buf()`.

See also `cgi_escape()`, `cgi_unescape_buf()`.

Example: The following code snippet prints `"Hello World\n"`:

```
char buf [] = "Hello+World%0a";
printf (cgi_unescape (buf));
```

2.13 `cgi_unescape_buf` - un-escape a CGI value buffer

Prototype: `int cgi_unescape_buf (char *buf, int len)`

Arguments: `buf` is the buffer to change from CGI format to C-string format, `len` is its length

Returns: New buffer length, after decoding. The buffer is changed in-place.

Description: Changes the contents of the `buf` argument. The CGI-specific parts are re-coded into a "normal" string. The recoding includes the changing of `+` characters to spaces, and the changing of hexadecimal `%XY` sequences to their corresponding ascii characters.

Note that the buffer argument is changed in-place; i.e, the function does not duplicate it. There is no re-allocation, the buffer may or may not be in allocated memory. The new buffer length is returned; the caller may reallocate the used space to this size.

To change ascii-Z strings, you can also use `cgi_unescape()`.

Example: The following two code snippets are equivalent:

```
// (a)
char *buf = "Hello+World%0a";
printf (cgi_unescape (buf));

// (b)
char *buf = "Hello+World%0a";
printf (cgi_unescape_buf (buf, strlen (buf) + 1));
```

Note the "`strlen(buf)+1`"; the count of characters to decode must include the trailing `'\0'`.

2.14 `checksum` - determine checksum of a file

Prototype: `int checksum (char const *fname)`

Arguments: `fname` is the file to sum up

Returns: `-1` when the file could not be read
otherwise the checksum of the contents

Description: Determines a checksum of the contents of a file. The sum is computed by left-shifting by 1 bit the sum so far and then adding the next byte read from the file. This is repeated for the whole file.

Note that this is indeed a very basic and simple checksum algorithm. See `checksum_adler32()` and `fchecksum_adler32()` for a more elaborate and more widely adopted checksum approach.

Example: The following code snippet illustrates the usage.

```
int sum = checksum ("/etc/issue");

if (sum == -1)
    printf ("/etc/issue could not be read\n");
else
    printf ("checksum of /etc/issue: %d\n", sum);
```

2.15 checksum_adler32 - determine checksum of a buffer

Prototype: unsigned int checksum_adler32 (unsigned char *data, unsigned int len)

Arguments: data is a the buffer to investigate, len its length

Returns: Adler-32 checksum

Description: Determines the checksum of a buffer using the Adler-32 algorithm. See e.g. <http://en.wikipedia.org/wiki/32> for a description. See also `fchecksum_adler32()` for a function to determine the checksum of a file.

Example:

2.16 cmm_allocated - determine how many bytes of memory are allocated

Prototype: int cmm_allocated (int minlev, int maxlev)

Arguments: minlev is the minimum level to include in the count, use 0 for "all", maxlev is the maximum level, use -1 for "all"

Returns: number of allocated bytes within the given levels

Description: This function counts the size of the allocated memory in the given levels. See also the `cmm*()` functions, `xstrdup()`, `tt(str_add())`, etc.

Example: void *x;

```
// Allocate some memory in level 0
cmm_setlevel (0);
x = cmm_malloc (300);          // Allocate some memory
x = cmm_malloc (500);

// Allocate some more in level 1
cmm_setlevel (1);
x = cmm_malloc (200);

// Show what we have got so far
printf("Overall allocated: %d bytes\n")
```

```
"In level 0: %d bytes\n"  
"In level 1: %d bytes\n",  
cmm_allocated(0, -1),  
cmm_allocated(0, 0),  
cmm_allocated(1, 1));
```

2.17 `cmm_blocksize` - determine the size of a CMM-allocated block

Prototype: `int cmm_blocksize (void const *mem)`

Arguments: `mem` is a pointer to allocated memory

Returns: the size in bytes of the allocation block pointed to by `mem`, or 0 when the block was not allocated within CMM or when it was already freed

Description: This function queries the internal administration to see whether the memory block pointed to by `mem` was allocated. If so, the size of the reserved area is returned.

This function can be used to check whether a given pointer addresses CMM-allocated memory: if not, `cmm_blocksize()` will return 0.

Example:

```
void *mem;  
char *str;
```

```
mem = cmm_malloc (200);  
str = xstrdup ("Hello World\n");  
  
printf ("Memory reserved for 'mem': %d\n",  
        "Memory reserved for 'str': %d\n",  
        cmm_blocksize (mem), cmm_blocksize (str));
```

2.18 `cmm_cleanup` - deallocate blocks from a given level upwards

Prototype: `int cmm_cleanup (int minlev)`

Arguments: `minlev` is the minimum level to clean up

Returns: number of freed blocks

Description: Frees all memory that was allocated using `cmm_malloc()` (and hence using the associated functions, `xstrdup()` etc.) at the level `minlev` or above.

Note: to free one single block of memory, use `cmm_free()`.

See also `cmm_malloc()`, `cmm_setlevel()`, etc..

```
Example: void *x;
           int  n,
             oldlev;

           oldlev = cmm_setlevel (0xdeaf);    // Define a new level

           x = cmm_malloc (10);              // Allocate some memory
           x = cmm_malloc (20);

                                           // Cleanup and report.
           printf ("Within the level: %d blocks freed\n",
                  cmm_cleanup (0xdeaf));

           cmm_setlevel (oldlev);            // Reset the level
```

2.19 cmm_free - free cmm_malloc () -d memory

Prototype: int cmm_free (void *mem)

Arguments: mem is memory block to free

Returns: CMM_OK (0) when block was successfully freed

CMM_NOSUCHBLOCK when block mem was not previously allocated

Description: Deallocates a memory block that was previously reserved using cmm_malloc(), cmm_realloc(), xmalloc(), str_printf(), etc..

Note that to deallocate a series of blocks, belonging to a given "level", cmm_cleanup() may be used. If the memory block "mem" was NOT previously allocated with cmm_malloc(), then the memory is freed anyway – but in that case, the internal memory management has probably gone haywire. This is signalled to the caller with a nonzero return value. Ergo, you can safely use cmm_free() in your programs instead of free(), even for normal malloc()-d pointers.

To find out whether a block was allocated in cmm*(), you can use cmm_blocksize().

See also cmm_malloc(), cmm_setlevel(), cmm_cleanup(), cmm_blocksize() and other cmm*() functions.

```
Example: void *x;

           x = cmm_malloc (1000);           // Allocate memory

           if (cmm_free (x))                // Free or report an error
               fprintf (stderr,
                       "While trying to free block:\n"
                       "memory wasn't previously allocated\n");
```

2.20 `cmm_malloc` - allocate memory, given some level

Prototype: `void *cmm_malloc(int sz)`

Arguments: `sz` is the number of bytes to allocate

Returns: pointer to allocated memory

Description: Allocates the requested memory, or calls `out_of_memory()` when the pool is exhausted. The requested memory is internally stored with the current level, so that it may be subsequently freed using `cmm_free()` or `cmm_cleanup()`.

Note that the requested `sz` bytes must be a positive number. When requesting zero bytes, or when requesting a negative size, a program error occurs.

See also: `cmm_setlevel()`, `cmm_free()`, `out_of_memory()`, `cmm_storeblock()`, `cmm*()` functions.

Example:

2.21 `cmm_realloc` - reallocate a `cmm_malloc()`-d block of memory

Prototype: `void *cmm_realloc(void *mem, int newsz)`

Arguments: `mem` is a pointer to previously `cmm_malloc()`-d memory, or 0 in which case the memory will just be `cmm_malloc()`-d, `newsz` is the size, which may be 0 to free the block

Returns:

- Pointer to newly allocated memory, or
- 0 when the block was not previously `cmm_malloc()`-d, or
- 0 when `newsz == 0` and the block was already freed.

Description: This function reallocates a block of memory, or triggers the "out of memory" handler (default: `out_of_memory()`) when the pool is exhausted.

The first argument, `mem`, may be a 0-pointer, in which case the function behaves just like `cmm_malloc()`.

The second argument may be a size of 0 bytes, in which case the function behaves like `cmm_free()`.

The function `cmm_realloc()` tries to be "nice" when it encounters a reallocation request for a block that is not in the internal dministration yet. In such a case, `cmm_realloc()` will fullfil the request (i.e., will re-allocate) and will then insert a reference to the memory into the private administration. This practice is of course discouraged, blocks should be allocated right from the start using the `cmm*()` functions, or `xmalloc()`, `xstrdup()`, `xcalloc()` etc..

Note that a non-positive `newsz` count of bytes will lead to a program error. If that occurs, check your program for the code that causes this and fix it.

Example: void *x;

```
x = cmm_malloc (100);           // allocate some memory
x = cmm_realloc (x, 300);       // reallocate
printf ("%d bytes occupied\n", // report, this must
        cmm_allocated (0, -1)); // now show "300"
```

2.22 cmm_setlevel - set the level for cmm operations

Prototype: int cmm_setlevel (int newlev)

Arguments: newlev is the new cmm level

Returns: Previous level

Description: Sets the new cmm level, when newlev greater or equal to zero. Levels less than 0 should not be used, such levels are reserved for the e-lib internal allocation.

To retrieve the current level, simply call cmm_setlevel(-1) and inspect the return value.

The level is relevant regarding the functionality of cmm_cleanup(). You can define a "level" of allocation using cmm_setlevel(), and then call cmm_cleanup() to deallocate all blocks in that level.

See also cmm_malloc(), cmm_cleanup(), cmm_allocated(), cmm*() functions,

Example: void *x;
int oldlev;

```
oldlev = cmm_setlevel (100);    // Set the level.

x = cmm_malloc (1000);          // Allocate two blocks.
x = cmm_malloc (200);

                                // Show what we have got so far
printf ("Allocated at level 100: %d bytes\n",
        cmm_allocated (100, 100));

                                // Clean up level 100 and higher
printf ("For level 100 and up: %d blocks freed\n",
        cmm_cleanup (100));

cmm_setlevel (oldlev);          // Switch back to previous level
```

2.23 cmm_storeblock - store an allocated memory block in the cmm administration

Prototype: void *cmm_storeblock (void *mem, int sz)

Arguments: `mem` is the pointer to allocated memory to store in administration, `sz` is the size of the allocated memory_

Returns: Pointer to the stored memory (same as the `mem` argument), or 0 when the internal allocation (necessary for administrative purposes) has failed

Description: This function stores a reference to allocated memory in the `cmm` administration. The purpose would be to allow `cmm_cleanup()` to sweep the memory. The level is stored with a reference to the current level, see `cmm_setlevel()` for details.

This function is used internally. However, it can be used to store previously allocated user memory, e.g., memory allocated in a library function.

Example: In the following example a hypothetical library function is called (of which no source is available). Then `cmm_storeblock()` is called to put a reference to the block in the `cmm` administration.

```
char *mem;
mem = some_function_that_allocates_a_string ();
if (! cmm_storeblock (mem, strlen (mem)))
    out_of_memory ();

// Alternatively you could of course use:
char *mem, *tmp;
tmp = some_function_that_allocates_a_string ();
mem = xstrdup (mem);    // this automatically stores in cmm
free (tmp);
```

2.24 copyfile - copy source file onto destination file

Prototype: `int copyfile (char const *src, char const *dest, int *nread, int *nwritten)`

Arguments: Copies the contents of the source file onto the destination file.

The function fills its arguments `nread` and `nwritten` with respectively the byte count of the read information and of the written information.

The return value signals whether the copying went succesful, or whether one of the files could not be opened, or whether only some bytes could be copied. In the last case, the caller should remove the destination file, since the copying action has lead to a currrupt file.

Note that a function `movefile()` also exists, which (obviously) moves a source to a destination.

Returns: `src` is the name of file to copy from

`dest` is name of file to copy to

`nread` is total of bytes read, supply a 0-pointer if this value should not be updated

`nwrite` is the total of bytes written, supply 0 if this value should not be updated

Description: `CP_OK` (0) All OK, file copied.

`CP_NOINF` Could not open input file `src` for reading.

CP_NOOUTF Could not open output file `dest` for writing.

CP_MISMATCH `nread` does not equal `nwritten`, corrupted copy. Note: the caller should then remove `dest`, it is an invalid file.

CP_DST_IS_NO_FILE `dest` does not yield a regular file path. The caller should check that `dest` yields a regular file, and not e.g. a directory; or that all directory components in `dest` exist.

Example: // (1) Example WITH storage of read/written bytes:

```
int ret, totread, totwritten;
ret = copyfile ("/etc/profile", "/tmp/out", &totread, &totwritten);
if (ret) {
    fprintf (stderr, "Copy failure\n");
    if (ret == CP_MISMATCH)
        unlink ("/tmp/out");
} else
    printf ("Copying succesful, %d bytes\n", totread);

// (2) Example WITHOUT storage of read/written bytes:
if ( (ret = copyfile ("/etc/profile", "/tmp/out", 0, 0)) ) {
    if (ret == CP_MISMATCH)
        unlink ("/tmp/out");
    error ("Copy failure\n");
}
```

2.25 copyright - show standard copyright notice

Prototype: `void copyright (char const *purpose, char const *version)`

Arguments: `purpose` is a one-liner stating the program purpose, `version` is a version ID

Returns:

Description: Displays a standardized copyright notice. The linkage version that is displayed is the e-lib internal version. The output goes to `stderr`, unless overruled by `copyright_setoutput()`.

See also:

- `copyright_setoutput()` - to define a different output stream
- `copyright_owner()` - to define a different "owner"
- `copyright_contact()` - to define different contact information

Example: `copyright ("log file parser", "2.12");`

This leads to something similar to:

```
e-tunity log file parser V2.12 (linkage 1.09)
Copyright (c) e-tunity 2000 ff. All rights reserved.
Contact e-tunity (info@e-tunity.com) for more information.
```

2.26 `copyright_contact` - redefine the "contact" string in a copyright message

Prototype: `void copyright_contact (char const *new_contact)`

Arguments: `new_contact` is the new contact name to appear

Returns:

Description: Redefines the contact that is displayed by the `copyright ()` function to `new_contact`.

See also: `copyright ()`.

Example:

2.27 `copyright_owner` - redefine the "owner" string in a copyright message

Prototype: `void copyright_owner (char const *new_owner)`

Arguments: `new_owner` is the new "owner" name

Returns:

Description: Redefines the owner that is displayed by the `copyright ()` function to `new_owner`.

See also: `copyright ()`.

Example:

2.28 `copyright_setoutput` - define output file for the `copyright ()` function

Prototype: `void copyright_setoutput (FILE *outputfile)`

Arguments: `outputfile` is new output file for `copyright ()`, instead of `stderr`

Returns:

Description: The default output stream for the `copyright ()` function is `stderr`. This function redefines that stream.

See also: `copyright ()`.

Example:

2.29 daemonize - fork a daemon process

Prototype: `int daemonize ()`

Arguments:

Returns: Similar to `fork ()`:

- A positive value is returned to the parent. The value is the process ID of the forked process.
- Zero is returned to the forked process.
- Negative values signify errors. See below for a list.

Description: This function combines `fork ()` and a number of other calls to 'safely' start a detached daemon process. The parent process may wait for the daemon to terminate, but doesn't have to; the daemon is fully detached.

The actions are the following:

- A new process is forked.
- The new process closes the file descriptors 0, 1 and 2 (`stdin`, `stdout`, `stderr`) and re-opens them to point to `/dev/null`. That way, if the program calls a spurious `printf ()`, the information will be silently discarded to `/dev/null`. Similarly, a `gets ()` won't wait for input.
- The new process is flagged as session leader (see `setsid (2)`) so that when terminating, *zombies* are avoided.

Following these actions, the parent process can safely exit and leave the forked process running, which now will be detached and belonging to `init`.

Error conditions are returned as negative values both in the parent and child branch. The following symbolic names are used:

- `DM_P_FORK_ERR`: Failure to fork, reported in the parent branch.
- `DM_P_IPC_ERR`: Failure to set up inter process communication (IPC) with the child. Reported in the parent branch.
- `DM_P_STDIN_ERR`: The child failed to to reopen `stdin` on `/dev/null`. Reported in the parent branch.
- `DM_C_STDIN_ERR`: The child failed to to reopen `stdin` on `/dev/null`. Reported in the child branch.
- `DM_P_STDOUT_ERR`: The child failed to reopen `stdout`. Reported in the parent branch.
- `DM_C_STDOUT_ERR`: The child failed to reopen `stdout`. Reported in the child branch.
- `DM_P_STDERR_ERR`: The child failed to reopen `stderr`. Reported in the parent branch.
- `DM_C_STDERR_ERR`: The child failed to reopen `stderr`. Reported in the child branch.
- `DM_P_SESSION_ERR`: Child failed to become a session leader. This may actually not be a fatal error if the parent process already called `setsid ()` itself. See the manpage for `setsid (3)`. Reported in the parent branch.
- `DM_C_SESSION_ERR`: Child failed to become a session leader. Reported in the child branch.

A valid approach to using `daemonize()` and to handle errors is the following.

1. `daemonize()` is called, and its result is stored in a local variable.
2. When the result is a positive value, then the child successfully daemonized. The result is the process ID of the child.
3. When the result is zero, then you're in the child branch that has just successfully daemonized. Using `printf()` etc. will no longer work.
4. When the result is a negative value, then something's afoot:
 - `DM_P_*` values mean that you're in the parent branch. Such values can be reported using the standard functions like `(f)printf()` or `error()`.
 - `DM_C_*` values mean that you're in the child branch. Reporting through `(f)printf()` is no use. Such values can be reported using `logmsg()` or `syslog(3)`, but don't have to. Their counterparts are available as `DM_P_*` to the parent process.

Upon receiving a negative value, the program (both the parent branch and the child branch) should exit.

Example: The following program starts a little daemon that hangs around for 1 minute and then exits.

```
// The daemon function.
void act_daemon () {
    int i;

    // Let's log to /var/log/messages.
    log_setup ("/dev/syslog/user/notice", 0, 0);

    // Loop every approx. 5 seconds for one minute.
    for (i = 0; i < 60; i += 5) {
        logmsg ("Test daemon now active for %d secs", i);
        sleep (5);
    }
}

// The main function to fire up the daemon.
int main () {
    int pid;

    printf ("Starting daemon...\n");
    if ( (pid = daemonize()) > 0) {
        // Parent branch: no errors.
        printf ("Daemon started at process ID %d\n"
            "Watch /var/log/messages for the output.\n", pid);
    } else if (pid == 0) {
        // Child branch: no errors.
        act_daemon();
    } else {
        // Parent or child branch, error. It's enough when we report the
        // DM_P_* errors that are seen by the parent; the child process
```

```
    // can rely on the parent reporting this.
    switch (pid) {
    case DM_P_FORK_ERR:
        error ("Failed to fork\n");
    case DM_P_IPC_ERR:
        error ("Failed to set up IPC pipe\n");
    case DM_P_STDIN_ERR:
        error ("Child failed to close/reopen stdin\n");
    case DM_P_STDOUT_ERR:
        error ("Child failed to close/reopen stdout\n");
    case DM_P_STDERR_ERR:
        error ("Child failed to close/reopen stderr\n");
    case DM_P_SESSION_ERR:
        error ("Child failed to become session leader\n");
    default:
        error ("Other (unspecified) error, code %d\n", pid);
    }
}

// All done.
return (0);
}
```

2.30 `elib_require` - check `e-lib` version, stop if too low

Prototype: `void elib_require (double v, char const *msg)`

Arguments: `v` is the required version for this program, `msg` is an appropriate error message, or 0 to get a default

Returns:

Description: This function verifies that the version of the shared object of *e-lib* against which the current program is executed, is at least `v`. If not, an error is issued.

The argument `v` is the required version number. E.g., imagine that *e-lib* 5.15 implements some very handy function that your program uses. In that case it would be a good idea to verify that the system that your program runs on, has *e-lib* 5.15 or higher.

The argument `msg` is an error message, displayed when the *e-lib* version is lower than the required `v`. In that case, a fatal error will occur.

The `msg` argument may be set to 0, in which case a default error message is displayed. `msg` may alternatively be a string, optionally containing two format specifiers for `double` values. The first one will be expanded to the currently active *e-lib* version, the second one will be expanded to the required version. (Good format specifiers are, e.g., `%.2f`, which yields two decimals. This nicely conforms with the *e-lib* versioning.)

See also `elib_version()`.

Example: Below are examples of requiring minimal versions of *e-lib*:

```
// Example 1: we require version 5.15, and use
// the default error message
elib_require (5.15, 0);

// Example 2: we require version 5.15, and have
// our own error message. We show the active
// e-lib version and the required version.
char *msg = str_printf ("You seem to have e-lib V%g. However,\n"
                       "this program needs %g. Ask your admin\n"
                       "to upgrade e-lib! The most recent\n"
                       "version can be found on\n"
                       "http://public.e-tunity.com.\n",
                       5.15, elib_version());
elib_require (5.15, msg);
// Since we got this far, msg is no longer needed.
cmm_free (msg);
```

2.31 `elib_version` - determine the version of this e-lib

Prototype: `double elib_version (void)`

Arguments:

Returns: version of this e-lib

Description: This function returns the version ID of the used *e-lib*. Determining the version is relevant for programs that rely on a given version of the *e-lib* shared object library; running such programs with an *e-lib* library with a too low version number might cause problems.

See also `elib_require()`.

Example: The following code snippet shows the version ID.

```
printf ("Current e-lib version: %.2f\n", elib_version());
```

2.32 `equal` - compare first parts of 2 strings without regard to casing

Prototype: `int equal (char const *a, char const *b)`

Arguments: `a` and `b` are the strings to compare

Returns: 0 if the strings are **not** equal, !0 otherwise

Description: The strings are compared over their minimum length; i.e., over the length of `a` if `a` is smaller, or over the length of `b` if `b` is smaller. When the strings match over this length, !0 is returned.

See also `strnicmp()`.

Example:

2.33 error - print error message including program name and exit

Prototype: void error (char const *fmt, ...)

Arguments: `fmt` is a `printf`-like format string, other arguments may follow

Returns:

Description: Prints an error message to `stderr` in a formatted way. The program name, if set using `setprogname ()`, is included. When logging is enabled using `log_setup ()`, then the error message is also written to the log file before exiting.

See also `setprogname ()`, `log_setup ()`, `logmsg ()`.

Example: The following call:

```
error ("bad input: %s\n", buf);
```

will produce something similar to:

```
progname: bad input: whatever
progname: aborting...
```

2.34 fchecksum_adler32 - determine checksum of a file

Prototype: int fchecksum_adler32 (FILE *f, unsigned int *csum)

Arguments: `f` is the file to investigate

Returns: 0 upon success, -1 for `fstat ()` errors, -2 for `fread ()` errors

Description: Determines the checksum of a file (which must be open for reading) using the Adler-32 algorithm. The computed checksum is stored in the `unsigned int` pointed to by `csum`.

See e.g. <http://en.wikipedia.org/wiki/Adler-32> for a description. See also `checksum_adler32 ()` for a function to determine the checksum of a buffer.

Example:

2.35 fgetline - read a line from a file, allocate buffer and return it

Prototype: char *fgetline (FILE *f)

Arguments: `f` is the file to read from, it must be opened for reading

Returns: Pointer to allocated memory where a read line is stored, or 0 when EOF was encountered.

Description: A line is read from the stream *f* and stored in allocated memory. A pointer to that memory block is returned. The line length in the file virtually unlimited; `fgetline()` will continue reading until one whole line, terminated by `\n`, is read.

Note: The caller is responsible for freeing the memory pointed to by the return value. This must be done using `cmm_free()` or `cmm_cleanup()`.

To strip any terminating `\n` characters, `stripstr()` can be used.

Example: The following code snippet acts like the command *cat*, but each line is prefixed with a line number. The program can handle lines of unlimited length.

```
int main (int argc, char **argv) {
    FILE *f;
    char *buf;
    int lineno = 0;

    if (argc != 2)
        error ("Usage: %s inputfile\n", argv[0]);
    if (! (f = fopen (argv[1], "r")) )
        error ("Cannot read %s: %s\n", argv[1], strerror(errno));
    while ( (buf = fgetline (f)) ) {
        printf ("%8d %s", ++lineno, buf);
        cmm_free (buf);
    }
    fclose (f);
    return (0);
}
```

2.36 filetype - determine the type of a file

Prototype: `int filetype (char const *fname)`

Arguments: *fname* is the name of the file to inspect

Returns: The return type is an integer ≥ 0 stating the file type, or < 0 during errors. See below.

Description: This function determines the filetype of the file named in *fname*. The return values can be:

- `FT_UNKNOWN_TYPE` Unknown file type
- `FT_TIFF_BIG_END` Tiffs on big endians
- `FT_TIFF_LITTLE_END` Tiffs on little endians
- `FT_PNG` PNGs (PNG is Not Gif)
- `FT_GIF` GIFs, all versions
- `FT_JPG` JPEGs

FT_BMP Windows Bitmaps

FT_XPM XPMs, X Portable Bitmaps

FT_OFFDOC Microsoft Office Document (e.g., a Word or Excel document)

FT_RTF Rich Text Format document

FT_XML XML Document format

These values range from 0 to 10. The following values are reserved for errors:

FT_NOFILE -1: file cannot be accessed

FT_CANNOTREAD -2: file cannot be read

FT_READFAILED -3: file read does not yield a header

Note that currently a **FT_OFFDOC** return value cannot distinguish between the Office document types.

Example: The following code prints the file type for all program arguments.

```
static char *desc [] = {
    "unknown", "tiff (big endian)", "tiff (little endian)",
    "png", "gif", "jpeg", "bmp", "xpm", "officedoc",
    "rtf", "xml",
};

int main (int argc, char **argv) {
    int i, ret;

    for (i = 1; i < argc; i++) {
        printf ("%s: ", argv [i]);
        if ( (ret = filetype (argv [i])) < 0 )
            printf ("error determining type\n");
        else
            printf ("%s\n", desc [ret]);
    }
    return (0);
}
```

2.37 **fin_push** - push a block of data back into a file, to be read by **fin_read()**

Prototype: void fin_push (void const *block, int nbytes, FILE *inf)

Arguments: • block is a pointer to data to "push back" into the file

- nbytes is the block size in bytes
- inf is the file where fin_read() will read the next time

Returns:

Description: "Pushes" a block of data back into a file, so that upon the next read (which **must** use `fin_read()`) the block will be re-read.

Example: see `fin_read()`

2.38 `fin_read` - read a block of data from a file, possibly a pushed-back block

Prototype: `int fin_read (void *block, int nbytes, FILE *inf)`

Arguments:

- `block` is a pointer where the data will be stored
- `nbytes` is the number of bytes to read
- `inf` is the file (stream) to read from

Returns: Number of read bytes, which may be less than `nbytes` when an end-of-file condition occurs

Description: This function is used in conjunction with `fin_push()` to read data from a file, and to be able to "push back" data. Following a push-back operation by `fin_push()`, the next `fin_read()` call will return the pushed-back bytes.

When used *without* `fin_push()`, `fin_read()` is similar to `fread()`.

The pushing-back of bytes is a FILO-operation, first-in-last-out (or LIFO, if you prefer). That means that the bytes that are pushed back first, are read as last.

There is no limit on the number of bytes to push back (except for situations of memory exhaustion). The functions `fin_read()` and `fin_push()` are re-entrant; they can be used for many reading operations in the same program. (The internal administration is based on the input file handle).

Example: The following code snippet reads 100 bytes, pushes back 10, and reads 100 more. Ergo, the second reading operation truly reads 90 bytes from the input file and returns a block where the first 10 bytes are the pushed-back ones.

```
FILE *inf;
char block [100];

fin_read (block, 100, inf);
fin_push (block, 10, inf);
fin_read (block, 100, inf);
```

2.39 `get_log_setup` - return currently active logging parameters

Prototype: `void get_log_setup (char **logfp, int *maxszp, int *nhistp)`

Arguments: `logfp` is a pointer to a `char*`, where the current logfile name will be stored; `maxszp` is a pointer to an `int` where the current maximum logfile size will be stored, `nhistp` is a pointer to an `int` where the current number of history logs will be stored.

Returns:

Description: This function returns the values that have been defined by the last `log_setup()` call.

Each argument can be a NULL-pointer, in which case the appropriate value will not be stored. E.g., when only `logfp` is a valid pointer, but the other arguments are 0, then only the current logfile name is stored.

Note that the `char*` pointed to by `logfp` is set to point to allocated memory. The caller is responsible for `cmm_free()`-ing it.

Example: Using this function, it is possible to temporarily redirect logging to another stream. E.g., the snippet below sends some log statements to `syslog`, and then restores the previous logging.

```
char *logf;
int  maxsz, nhist;

// Temporarily redirect logging.
get_log_setup (&logf, &maxsz, &nhist);
log_setup ("/dev/syslog/user/notice", 0, 0);

// Log our stuff.
logmsg ("Hi there!\n");

// Restore logging to whatever it was.
log_setup (logf, maxsz, nhist);
cmm_free (logf);
```

2.40 hash - compute hash-index for a string

Prototype: `int hash (char const *key)`

Arguments: `key` is the string to compute a hash for

Returns: Hash value

Description: This function internally calls `hashbuf()` to compute a hash value, and returns it. See `hashbuf()` for more information.

This function is used by the `bth_*()` functions.

Example: See the `bth_*()` functions.

2.41 hashbuf - compute hash-index for a buffer

Prototype: `unsigned hashbuf (unsigned char *key, unsigned long int length, unsigned long int seed)`

Arguments: `key` is the buffer to compute the hash for, `length` is the buffer length, `seed` is an initial value

Returns: Hash value

Description: This function applies Bob Jenkins' (bob_jenkins@burtleburtle.net<bob_jenkins@burtleburtle.net>) hash algorithm to the buffer. For the full algorithm please see the sources.

Example:

2.42 `html_escape` - convert special chars in a buffer to HTML-escape sequences

Prototype: `char *html_escape (char const *buf)`

Arguments: `buf` is the buffer in which to escape special characters

Returns: Pointer to allocated memory holding the escaped version of `buf`

Description: This function is used when, e.g., pasting a chunk of text into an HTML page. The chunk of text may not include "special" characters, such as `<`, `>`, `&` and quotes. Function `html_escape()` converts these characters to HTML-escape-sequences, such as `<`, `>`, `&`, and so on (see also the ISO8859-1 specification.)

The character conversions are:

- `<` and `>` are converted to `<` and `>`;
- `&` is converted to `&`;
- Single quotes are converted to `'`;
- Double quotes are converted to `"`;
- High or low ASCII characters are converted to `&#xyz;` where `xyz` is a three-digit 10-based ASCII code.

Note that the returned value points to allocated memory. The caller is responsible for the deallocation of it. (As with all e-lib functions, use `cmm_free()` to deallocate memory..)

Other *ML conversion functions are `ml_encode()` and `ml_decode()`. These are similar to `html_unescape()` and `html_escape()`, but support a larger collection of character maps.

See also `html_unescape()`.

Example: The following code fragment prints something like `<"Hello World!">`

```
char *converted = html_escape ("<\"Hello World\">");
printf ("%s\n", converted);
```

The following code undoes this:

```
html_unescape (converted);
```

2.43 `html_unescape` - convert escaped HTML to original characters

Prototype: `char *html_unescape(char *buf)`

Arguments: `buf` is the buffer to process

Returns: pointer to the modified buffer, which is modified in-place

Description: This function is the inverse of `html_escape()`. Special sequences in the buffer, such as `<`; are converted to their original characters.

Note that the buffer which is passed as the argument, is modified in-place. The caller should make a local copy before calling `html_unescape()` if the original buffer contents are to be preserved.

Other *ML conversion functions are `ml_encode()` and `ml_decode()`. These are similar to `html_unescape()` and `html_escape()`, but support a larger collection of character maps.

See also `html_escape()`.

Example:

2.44 `image_size` - determine the size of an image

Prototype: `int image_size(char const *fname, ImageSize *isp)`

Arguments: `fname` is the name of afile holding the image file to inspect, `isp` is a pointer to an `ImageSize` structure that will be filled with size data

Returns: `IS_OK (0)` size determined and `ImageSize` structure filled

`IS_NOFILE` Mentioned file cannot be read

`IS_UNSUPP` Unsupported format, cannot determine the size

`IS_JPEGERR` Error while parsing a JPEG file

`IS_PNGERR` Error while parsing a PNG file

`IS_GIFERR` Error while parsing a GIF file

`IS_BMPERR` Error while parsing a BMP file

Description: This function inspects an image file and determines its size. Note that only a subset of images is supported, currently: JPEG, PNG, GIF, BMP. Furthermore note that not the full JFIF standard is supported, so that some JPEG files may fail.

For these images, the structure pointed to by `isp` is filled with the correct width and height data. The `ImageSize` structure just holds two `int` elements: a `w` (for width) and `h` (for height).

See also: `filetype()`

Example: The following code snippet inspects a file and shows the size. All image parsing error messages are bundled in this example.

```
ImageSize is = {0, 0};
int ret;

if (! (ret = image_size ("myfile.gif", &is)) )
    printf ("width: %d, height: %d\n", is.w, is.h);
else switch (ret) {
    case IS_NOFILE:
        printf ("no such file\n");
        break;
    case IS_UNSUPP:
        printf ("unsupported image format\n");
        break;
    default:
        printf ("file cannot be parsed\n");
        break;
}
```

2.45 lap_end - stop a task stopwatch and log the results

Prototype: void lap_end ()

Arguments:

Returns:

Description: A stopwatch that must have been started by lap_start () is stopped and the recorded laptimes are logged to a file, defined by lap_setup ().

Note that task and their stopwatches are stacked. This function stops the **last started** stopwatch and outputs the results.

The logging of laptimes is suppressed if the logfile name (as defined by lap_setup ()) is an invalid string (NULL) or if the corresponding file cannot be opened.

Lap times and logging can be suppressed for an entire program by either never calling lap_setup (), or temporarily by calling lap_setup(0). Temporary suppression of logging must however occur **prior** to a stopwatch start by lap_start (); and **not** between a lap_start () and a lap_end ().

The output that is generated is in the reversed order; last stopped stopwatches are written first to the log file. This is inherent to the stack principle of the lap_* () functions. Use, e.g., *tac logfile* to reverse the lines order.

The shown output represents, per generated line:

- the task name
- the elapsed time ticks since the stopwatch start
- the CPU time ticks spent in user time of the program itself
- the CPU time ticks spent in system time of the program itself
- the CPU time ticks of user time of child processes
- the CPU time ticks of system time of child processes

- the number of ticks in one second

Example: The following code snippet shows how the overall program performance is measured and how two tasks, one of which is nested, is measured.

```
void function () {
    int i;
    double d;

    lap_start ("myfunc");
    for (i = 0; i < 1000; i++)
        d = (10.0 * (double) i) / 3;
    system ("ls > /dev/null");
    lap_end ();
}

int main () {
    int i;

    lap_setup ("/tmp/lap.log");
    lap_start ("main");

    lap_start ("main:loop");
    for (i = 0; i < 10; i++)
        function ();
    lap_end ();

    system ("ls > /dev/null");
    lap_end();
    return (0);
}
```

After `tac /tmp/lap.log` an overview is shown of the following:

- one task "main" with the overall time spent;
- one task "main:loop" with the time spent inside the loop that calls `function()`;
- ten tasks "function"

2.46 lap_setup - define stopwatch laptime log file

Prototype: `void lap_setup (char const *fname)`

Arguments: `fname` is the filename to log laptimes to, or 0 if no logging required

Returns:

Description: This function defines the filename where laptimes will be logged. The actual starting of the stopwatch and the logging is done via `lap_start()` and `lap_end()`.

Note: This function does not make a duplicate of the `fname` argument. Be sure to keep this string in memory during the run of the program where the `lap*()` functions are used.

See also: `lap_start()`, `lap_end()`

Example:

2.47 `lap_start` - start the stopwatch mechanism

Prototype: `void lap_start(char const *task)`

Arguments: `task` is the name of the task whose stopwatch should be started

Returns:

Description: This function creates a stopwatch for the task and starts it. Note that the task stopwatches are "stacked"; the last task to start must be the first one to end. The tasks can then be ended via `lap_end()`. Note that the log file for stopwatch output must be defined via `lap_setup()`.

See also: `lap_setup()`, `lap_end()`

Example:

2.48 `log_dontflush` - suppress log file flushing when `logmsg()` is called

Prototype: `void log_dontflush()`

Arguments:

Returns:

Description: After this function has been called, the logs that are generated with `logmsg()` are not "flushed" after each write. This may speed up the application (especially on slow file systems), but will cause the logs not to be up-to-date.

Note that by calling `log_dontflush()` after (or before) a `log_setup()`, the rotating and sizing of logfiles gets disabled during the current program run.

Also note that when the the system log facility is used (by using `/dev/syslog/` as the log file name in `log_setup()`), logfile flushing does not apply.

Example: see `logmsg()`

2.49 logmsg - log messages to a log file, defined by log_setup()

Prototype: void logmsg (char const *fmt, ...)

Arguments: *fmt* is a format string a-la `printf()`, the remaining arguments are expanded according to the format

Returns:

Description: Writes a message to a log file and "rotates" logs if necessary.

- If the logfile points to the system log facility (`/dev/syslog/`), then the following bullets do not apply. I.e., timestamping is left to the system log handling and so on.
- The logged message is prefixed by a date/time string in the format: YYYY-MM-DD HH:MM:SS. The length of the prefix is 20 characters (including a trailing space). This timestamping behavior can be switched off by calling `log_timestamp(0)`.
- When *fmt* is a 0, nothing is logged but the file is rotated (if necessary).
- Each newly created log file receives permissions 0666, i.e., read/writable by all.
- The file name of the log file is set up via `log_setup()`, see the info there.
- By default, logs are flushed after each write. Not flushing logs will somewhat speed up the program. See `log_dontflush()`. However, this will prevent the logs from being rotated.
- By default, long log messages will not "wrap" to a next line, unless you call `log_wrap(1)` before issuing `logmsg()` calls.

See also `log_setup()`, `log_timestamp()`, `log_dontflush()`, `log_rotate()`.

Example:

```
log_setup ("/var/log/myprog.log", 10000, 3);
logmsg ("some message\n");
```

This defines `/var/log/myprog.log` as the log file, with a max size of 10000 bytes and 3 history files. Then a message is sent to the file.

2.50 log_rotate - rotate a log file if it exceeds given limits

Prototype: int log_rotate (char const *fname, int maxsz, int nhist)

Arguments: *fname* is the logfile name, *maxsz* is the limit, *nhist* is the number of old versions to keep

Returns: <0 upon error, 0 when the file doesn't need rotating, 1 otherwise

Description: This function rotates the file indicated by *fname* if its size exceeds *maxsz* bytes. When rotating, the file is renamed to *fname.1*. Any pre-existing *fname.1* is renamed to *.2* and so on; *nhist* history files are kept.

This function is used by, e.g., `logmsg()`. See the description there.

Example:

2.51 log_setup - define logging options

Prototype: void log_setup (char const *logfname, int maxsz, int nhist)

Arguments: **logfname** is the name of the log file to create or update

maxsz is max size of the log

nhist is the number of history files to keep

Returns:

Description: Defines the logging mechanism which is used with the `logmsg()` function. The `logfname` defines the log file to which messages are appended. The argument `maxsz` defines the logfile size. When the size of the file exceeds this, the old file is renamed to a `*.1` version and compressed into `*.1.gz`, and a new file is created. The `nhist` argument specifies the number of history files to keep.

E.g., when `nhist == 2`, then a bare log file will exist plus two history files, named `*.1.gz` and `*.2.gz`.

Newly created log files automatically receive the permissions 0666, i.e., readable and writable by all.

Notes:

- When present, the environment variables `ELOGNAME`, `ELOGSIZE` and `ELOGHIST` override the arguments `logfname`, `maxsz` and `nhist`. This is implemented in order to allow customization of the logging of already compiled programs.
- You **MUST** call `log_setup()` before calling `logmsg()`. Otherwise, the log file name will be empty and `logmsg()` will not produce output.
- The following names are special in the `logfname` argument:
 - `/dev/stdout` selects `stdout` for log file
 - `/dev/stderr` selects `stderr` for log file
 - `/dev/syslog/fac/lev` selects the system logger. In this case, logfile rotation does not apply. The filename parts `fac` and `lev` have special meanings; see below.

The 'special' filename `/dev/syslog/fac/lev` selects the system log facility for logging purposes. The name also selects the facility and level:

- The `fac` part of the filename may be `auth`, `authpriv`, `cron`, `daemon`, `ftp`, `kern`, `local0` through `local7`, `lpr`, `mail`, `news`, `syslog`, `user` and `uucp`. When the `fac` part matches none of the above, then `user` is taken.
- The `lev` part of the filename may be `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info` and `debug`. These are the 'emergency levels' in decreasing order of importance. When `lev` matches none of the above, then `notice` is taken.

See also `logmsg()` to log a message, `log_timestamp()` to enable or disable timestamping, and `msg_and_log()` to log a message and optionally produce a message on `stderr`.

Example: The following code snippet demonstrates C code:

```
log_setup ("/var/log/myprog.log", // filename
           10 * 1024,           // rotate when size > 10K
           4);                  // keep 4 history logs
logmsg ("something to log\n");
```

Once the program is compiled, logging can be changed by, e.g.:

```
$ ##### Example 1:
$ export ELOGNAME=/var/log/someother.log # log to some file
$ export ELOGSIZE=200000                 # 200K logs
$ export ELOGHIST=10                     # 5 history files
$ myprog                                  # run the program
$
$ ##### Example 2:
$ export ELOGNAME=                        # suppress logging
$ myprog                                  # run the program
```

2.52 log_timestamp - enable or disable timestamping of log entries

Prototype: void log_timestamp (int onoff)

Arguments: onoff is a switch; 0 turns timestamping off, !0 turns it on

Returns:

Description: This function turns the timestamping on or off. The next call to logmsg() will produce a log entry that is affected by the setting.

Note that timestamping is by default ON. Furthermore, when log_setup() is instructed to send its messages to the system log facility, then the timestamping cannot be influenced.

Example: log_setup ("/tmp/my.log", 50000, 5);
logmsg ("an entry with a timestamp\n");
log_timestamp (0);
logmsg ("an entry without\n");

2.53 log_wrap - enable or disable log line-wrapping

Prototype: void log_wrap (int onoff)

Arguments:

Returns: onoff is a switch, 0 will disable line wrapping

Description: Defines whether subsequent calls to logmsg() will wrap their lines, or not.

See also: logmsg(), log_*()

Example:

2.54 `make_socket` - create a server-side TCP/IP socket for a port

Prototype: `int make_socket (int port)`

Arguments: `port` is the port to which the socket will be bound

Returns:

- -1 when the socket could not be created (see `socket(2)`), you can analyze `errno` for details
- -2 when the socket couldn't be marked reusable, you can analyze `errno` for details
- -3 when binding to the socket failed (see `bind(2)`), you can analyze `errno` for details
- otherwise: a socket.

Description: This function creates a (typically server-side) socket for a given port and binds to it. The socket can later be listened to.

Example:

2.55 `md5` - compute an MD5 digest

Prototype: `void md5 (unsigned char const *buf, unsigned len, unsigned char digest[16])`

Arguments: `buf` is the buffer to digest,

`len` is its length,

`digest` is the target, where the 16-byte digest will be stored

Returns:

Description: This function computes the MD5 digest of its `buf` argument, over `len` bytes. The output is stored in the array `digest`, of which the first 16 bytes are used. The caller is responsible for making sure that `digest` is (at least) 16 bytes long. See also `sha1()`.

Example: `unsigned char digest[16];`

```
int i;
```

```
md5 ("Hello World", 11, digest);
```

```
for (i = 0; i < 16; i++)
```

```
    printf ("%2.2x", digest[i]);
```

```
putchar ('\n');
```

2.56 `ml_decode` - decode a *ML representation into a "normal" buffer

Prototype: `char *ml_decode (char const *buf)`

Arguments: `buf` is a buffer to use as source for decoding

Returns: allocated buffer holding decoded sequence

Description: Decodes *ML-specific sequences in the buffer into single characters. E.g., the sequence `<` in a HTML or XML string is decoded into `<`. In contrast to the function `html_unescape()`, a much "wider" range of character maps is supported.

The returned buffer is in allocated memory. The caller is responsible for freeing it.

See also `ml_encode()`, `html_escape()`, `html_unescape()`.

Example:

2.57 `ml_encode` - encode a buffer into *ML representation

Prototype: `char *ml_encode (char const *buf)`

Arguments: `buf` is buffer to use as source for encoding

Returns: allocated buffer holding encoded sequence

Description: Encodes an ascii-Z string into a buffer holding *ML-specific sequences. E.g., `<` is encoded into `<`.

The returned buffer is in allocated memory. The caller is responsible for freeing it.

See also `ml_decode()`.

Example:

2.58 `mmsg_body` - return n'th body of a parsed mail message

Prototype: `char *mmsg_body (MailMsg *mm, int partnr)`

Arguments: `mm` is a `MailMsg` handle, `partnr` indicates the ordinal number of the mail message body to retrieve

Returns: pointer to the mail message body, or 0 if no such part exists

Description: Returns the mail message body. When `partnr` is 0, the main body is returned. In multipart messages, this is the full text, including all parts. The first part of a multipart message starts at `partnr` 1. Note that `mmsg_parts()` indicates how many parts there are.

The return value points to a string holding the body. The caller may not deallocate or change this.

Example: The following code parses a mail message and shows each body when its content type is `text/plain`.

```
MailMsg *mm;                // MailMsg handle
extern char *buffer;        // some text buffer
int parts;                  // parts in message
char *ctype;                // content type
```

```
int i;                // loop var

mm = mmsg_new();
mmsg_parse (buffer);
parts = mmsg_parts (mm);

if (!parts)
    printf ("main body:\n%s\n", mmsg_body (mm, 0));
else for (i = 1; i <= parts; i++) {
    printf ("body of part %d:\n", i);
    ctype = mmsg_partheaderval (mm, i, "content-type");
    if (ctype && strstr (ctype, "text/plain"))
        printf ("%s\n", mmsg_body (mm, i));
    else
        printf ("cannot display\n");
}
```

2.59 mmsg_destroy - deallocate a MailMsg handle

Prototype: void mmsg_destroy (MailMsg *mm)

Arguments: mm is a MailMsg handle

Returns:

Description: All allocated data contained by mm are freed.

Example: MailMsg *mm;

```
mm = mmsg_new ();
mmsg_parse ("some mail message");
mmsg_destroy (mm);
```

2.60 mmsg_fileparse - parse a mail message from file

Prototype: int mmsg_fileparse (MailMsg *mm, char const *fname)

Arguments: mm is a MailMsg handle, fname is a filename, holding a mail message

Returns: number of scanned bytes when the mail file was parsed, or a -1 otherwise (in that case, investigate errno for reasons)

Description: This function is similar to mmsg_parse(), or mmsg_fparse(), except that the message to parse is expected in a file. See mmsg_parse() for further information.

Example: The following code parses a mail message contained in a file /tmp/msg.

```
MailMsg *mm;
int nbytes;

mm = mmsg_new();
if ( (nbytes = mmsg_fileparse (mm, "/tmp/msg")) < 0 )
    error ("cannot parse file msg: %s\n",
          strerror(errno));
else if (!nbytes)
    error ("empty mail message\n");

printf ("%d bytes parsed from message\n", nbytes);
```

2.61 mmsg_fparse - parse a mail message a file handle

Prototype: int mmsg_fparse (MailMsg *mm, FILE *f)

Arguments: mm is a MailMsg handle, f is a filehandle, opened for reading

Returns: number of scanned bytes when the mail file was parsed

Description: This function is similar to mmsg_parse(), except that the message to parse is read from a file handle. See mmsg_parse() and mmsg_fileparse() for further information.

Example:

2.62 mmsg_header - return header of a mail message

Prototype: char *mmsg_header (MailMsg *mm, int partnr)

Arguments: mm is a MailMsg handle, partnr indicates a part

Returns: pointer to the header, consisting of \n-separated lines, or 0 when partnr is out of range

Description: This function returns header information of a mail message. When partnr is 0, then the header of the main mail message is returned. For multipart messages, partnr can range from 1 up to and including mmsg_parts() to access the headers of parts.

Example: The following code shows all headers in a message, also when the message is in multipart format.

```
MailMsg *mm;
int i;

mm = mmsg_new();
if (mmsg_fileparse (mm, "/tmp/somefile") <= 0)
    error ("no valid mail in /tmp/somefile\n");
for (i = 0; i < mmsg_parts(mm); i++) {
```

```
printf ("Header of part %d:\n");
if (!i)
    printf ("Note: that's the main header\n");
printf ("%s", mmsg_header (mm, i));
}
```

2.63 mmsg_headerval - retrieve value of a header variable

Prototype: `char *mmsg_headerval (MailMsg *mm, char const *var)`

Arguments: `mm` is `MailMsg` handle, `var` is the variable to find in the main header

Returns: pointer to allocated memory, holding a copy of the value of the indicated header variable, or 0 when the variable was not present

Description: Returns the setting (value) of a given mail message variable. The variable is looked up in the main mail message header.

Example: The following code shows who sent a message and on which subject.

```
MailMsg *mm;
char *from, *subject;
extern char *buffer;           // some buffer holding a message

mm = mmsg_new ();
mmsg_parse (mm, buffer);

from = mmsg_headerval (mm, "from");
subject = mmsg_headerval (mm, "subject");

printf ("Mail sent by: %s\n"
        "Mail subject: %s\n",
        from ? from : "<<no sender address found>>\n",
        subject ? subject : "<<no subject specified>>\n");
mmsg_free (from);
mmsg_free (subject);
```

2.64 mmsg_new - initialize a MailMsg handle

Prototype: `MailMsg *mmsg_new()`

Arguments:

Returns: handle to an initialized `MailMsg` structure

Description: This function creates and initializes a `MailMsg` handle. It must be called before a mail message can be parsed using `mmsg_parse()`, `mmsg_fparse()` or `mmsg_fileparse()`.

Example: see `mmsg_parse()`, `mmsg_fparse()`.

2.65 mmsg_parse - parse a mail message

Prototype: void mmsg_parse (MailMsg *mm, char const *buffer)

Arguments: mm is a MailMsg handle, buffer is a mail message to parse

Returns:

Description: This function parses a mail message from a string buffer. Note that a similar function mmsg_fparse() exists to parse a mail message from a file handle, and a function mmsg_fileparse() exists to parse a mail message from a named file.

The normal way of using mmsg_parse() and the other mmsg_*() functions is:

- A MailMsg handle is prepared using mmsg_new().
- mmsg_parse() is called to parse a mail message from a string buffer.
- Next, a plethora of functions is available to access the mail message:
 - mmsg_body(mm, 0) will always return the main mail message body;
 - mmsg_header(mm, 0) will return the main mail headers;
 - mmsg_headerval(mm, "from") will return the value of the main header stating the From: address;
 - mmsg_parts(mm) will return the number of parts when the mail message is a MIME/multipart message, or 0 when the mail message doesn't have parts;
 - mmsg_body(mm, 1) will return the first part of a multipart message, mmsg_header(mm, 1) will return the headers of that part;
 - mmsg_partheaderval(mm, 1, "content-type") will return the value of the Content-Type: header of the first part of the mail message;
 - mmsg_reset(mm) will deallocate parts of the MailMsg handle, so that mmsg_parse() can be called again;
 - mmsg_destroy(mm) will deallocate all parts of the MailMsg handle.

Example: The following sample creates a MailMsg handle and parses a mail message contained in a variable buffer. The number of parts is shown.

```
MailMsg *mm; // MailMsg handle
extern char *buffer; // buffer holding text

mm = mmsg_new();
mmsg_parse (buffer);
if (mmsg_parts(mm))
    printf ("mail message has %d parts\n",
           mmsg_parts());
else
    printf ("mail message has just one body\n");
```

2.66 `mmsg_partheaderval` - retrieve header value of a mail message part

Prototype: `char *mmsg_partheaderval (MailMsg *mm, int partnr, char const *var)`

Arguments: `mm` is a `MailMsg` handle, `partnr` is the mail message part to inspect, `var` is the variable whose value is retrieved

Returns: setting of the variable, or 0 if `partnr` is out of range, or if the variable was not found in the indicated header

Description: This function retrieves the setting of a mail message header variable. E.g., when `partnr` is 0 and `var` is `from`, then the `From:` address is returned. The argument `partnr` may be greater than 0 when multipart messages are expected.

Example: The following code determines whether a given part is base64-encoded. If so, the corresponding part is decoded and shown.

```
int i;
char *encoding, *decoded;
extern MailMsg *mm;

for (i = 1; i <= mmsg_parts(mm); i++) {
    encoding = mmsg_partheaderval (mm, i, "content-transfer-encoding");

    if (encoding && strstr (encoding, "base64")) {
        printf ("Message part %d is base64-encoded\n", i);
        decoded = base64_decode (mmsg_body (mm, i),
                                strlen(mmsg_body (mm, i)), 0);
        printf ("Decoded body part:\n%s\n", decoded);
        cmm_free (decoded);
    }

    cmm_free (encoding);
}
```

2.67 `mmsg_parts` - return number of parts in a mail message

Prototype: `int mmsg_parts (MailMsg *mm)`

Arguments: `mm` is a `MailMsg` handle

Returns: number of parts, 0 meaning only 1 main part

Description: This function returns the number of parts in a parsed mail message.

Example:

2.68 `mmsg_reset` - reset a `MailMsg` handle

Prototype: `MailMsg *mmsg_reset (MailMsg *mm)`

Arguments: `mm` is a `MailMsg` handle to reset

Returns: reset handle

Description: This function 'resets' the relevant parts of a `MailMsg` handle. The handle can be re-used to parse a new message.

Example: The following code re-uses `mm`:

```
MailMsg *mm;
extern char *buf1, *buf2;

mm = mmsg_new ();
mmsg_parse (mm, buf1);
.
.
.
mmsg_reset (mm);
mmsg_parse (mm, buf2);
.
.
.
mmsg_destroy (mm);
```

2.69 `mmsg_setheader` - set or modify a mail message header

Prototype: `void mmsg_setheader (MailMsg *mm, char const *var, char const *val)`

Arguments: `mm` is a `MailMsg` handle, `var` is the header variable to set, `val` is the value

Returns:

Description: This function sets a variable to a given value in the main header of a mail message, thereby overwriting any pre-existing matching header variables, or adding to the header.

Example: The following code snippet adds a custom header:

```
MailMsg *mm;
extern char *buffer;

mm = mmsg_new()
mmsg_parse (mm, buffer);
mmsg_setheader (mm, "X-Custom-Header",
                "new header variable");
printf ("Modified header:\n%s\n",
        mmsg_header (mm, 0));
```

```
// This will show
// X-Custom-Header: new header variable
// as one of the displayed header lines
```

2.70 movefile - move a source file to a destination file

Prototype: `int movefile (char const *src, char const *dst)`

Arguments: `src` is a source file, `dst` is the destination file name (not just a directory)

Returns: `CP_OK` (0) All OK, file copied.

`CP_NOINF` Could not open input file `src` for reading, `errno` holds the cause

`CP_NOOUTF` Could not open output file `dest` for writing, `errno` holds the cause

`CP_MISMATCH` `nread` does not equal `nwritten`, corrupted copy. Note: the caller should then remove `dest`, it is an invalid file.

`CP_DST_IS_NO_FILE` `dest` does not yield a regular file path. The caller should check that `dest` yields a regular file, and not e.g. a directory.

`MV_CANNOT_RM_SRC` The move was attempted using `copyfile()`, but the original `src` could not be removed afterwards.

Description: Moves file `src` to `dst`.

This function is similar to `copyfile()`, except that (obviously) `src` no longer exists after a successful operation. In contrast to the system function `rename()`, `movefile()` will also work across filesystems. (Internally, if a `rename` fails, `copyfile()` will be called to do the job.)

The return value signals the success or failure reason. Most of the return values are identical to `copyfile()`.

Example: The following code snippet attempts to move `/tmp/a` to `/usr/local/bin/a`. The error conditions are caught.

```
int res;
if (res = movefile ("/tmp/a", "/usr/local/bin/a")) {
    fprintf (stderr, "Failed to move file: ");
    switch (res) {
        case CP_NOINF:
            fprintf (stderr, "/tmp/a not found: %s\n",
                    strerror (errno));
            break;
        case CP_NOOUTF:
            fprintf (stderr, "cannot write /usr/local/bin/a: %s\n",
                    strerror (errno));
            break;
        case CP_MISMATCH:
            fprintf (stderr, "file corruption during operation\n");
            unlink ("/usr/local/bin/a");
            break;
    }
}
```

```
    case CP_DST_IS_NO_FILE:
        fprintf (stderr, "/usr/local/bin/a isn't a true file,\n"
                "maybe your system lacks /usr/local/bin?\n");
        break;
    case MV_CANNOT_RM_SRC:
        fprintf (stderr, "/tmp/a cannot be removed, but "
                "/usr/local/bin/a succesfully created\n");
        break;
}
}
```

2.71 msg - show message on stderr when in verbose mode

Prototype: void msg (int verb, char const *fmt, ...)

Arguments: **verb** is required verbosity to make the message appear

fmt is a format string a-la printf()

... are optional remaining arguments

Returns:

Description: The message appears on `stderr`, prefixed by a program name (if one is set using `setprogname()`), if the required verbosity as indicated by the `verb` argument, is matched by the number of previous calls to `msg_on()`.

For example, a `msg()` call with a required verbosity 1 will appear if `msg_on()` was called at least once.

See also `msg_and_log()` to log a message and optionally show it on `stderr`.

Example: The following code snippet sets the verbosity to level 1. Then some messages are shown.

```
msg_on ();
msg (0, "this will always appear\n");
msg (1, "this will appear with this verbosity level\n");
msg (2, "this will only appear with a higher level\n");
```

The following code snippet is more elaborate. As long as `-v` flags appear in the program options, the verbosity is increased. Further on through the code, `msg()` calls are placed that during a run will appear only if the "verbosity level" at invocation was sufficiently high.

```
int opt;

while ( (opt = getopt (argc, argv, "vcd:g:")) > 0 ) {
    switch (opt) {
        case 'v':
            msg_on ();
            break;
        .
    }
}
```

```
        . Other options, c, d and g would be handled
        . here
        .
    }
}
.
.
.
msg (1, "High-priority message\n");
msg (3, "This appears only when -vvv was given\n");
```

2.72 msg_and_log - log a message and optionally show verbose message

Prototype: void msg_and_log (int level, char const *fmt, ...)

Arguments: **level** is the verbosity level, as in msg ()

fmt is a format string, a-la printf ()

... are any remaining arguments

Returns:

Description: This function combines msg () and logmsg () into one function. Depending on the verbosity level, the resulting message is shown on stderr. The message is also sent to the logfile via logmsg ().

Before calling this function, setprogname () and logsetup () should be called.

Example: setprogname ("myprog"); // define program name
logsetup ("/var/log/myprog.log", // define logging name
50000, 5); // with maxsize and nhist

```
// Now the following two are equivalent:
// (a)
msg (2, "testing %d %d %d\n", 1, 2, 3);
logmsg ("testing %d %d %d\n", 1, 2, 3);
// (b)
msg_and_log (2, "testing %d %d %d\n", 1, 2, 3);
```

2.73 msg_on - increase program verbosity by 1

Prototype: void msg_on (void)

Arguments:

Returns:

Description: Increases the program verbosity by one. Typically this function is called while parsing the program arguments. Subsequent calls to `msg()` may or may not produce messages, depending on the verbosity level.

See also `msg()`.

Example:

```
int opt;
while ( (opt = getopt (argc, argv, "v")) > 0 ) {
    switch (opt) {
        case 'v':    msg_on ();
                    break;
        ...
    }
}
```

2.74 `out_of_memory` - the "out of memory" handler function

Prototype: `void out_of_memory ()`

Arguments:

Returns:

Description: Calls the "out of memory" handler. Normally, this handler will stop the program with an appropriate message using the `error()` function. You can redefine this handler using `set_memerr()`.

See also `xmalloc()`, `xrealloc()`, `xstrdup()` etc., `set_memerr()`, `error()`

Example:

2.75 `ps_freesnapshot` - Deallocate memory occupied by a process snapshot

Prototype: `void ps_freesnapshot (PsList *list)`

Arguments: pointer to snapshot memory

Returns:

Description: This function deallocates (frees) the memory occupied by a process snapshot, made by `ps_makesnapshot()`. See this function for more information.

Example:

2.76 ps_makesnapshot - make a snapshot of the Unix process table

Prototype: `PsList *ps_snapshot()`

Arguments:

Returns: Pointer to an allocated memory area, holding process information, or 0 upon failure

Description: This function makes a 'snapshot' of the Unix process table.

The return value is a pointer to a `PsList` structure. This structure holds:

- an entry count `nlist`,
- an allocated array of `PsEntry` structures, named `list`.

Each list entry is a structure of the following information:

- `char *name`: the program name, without path directory;
- `char **argv` and `int argc`: the arguments of the program, with `argv[0]` being the full program name, with directory;
- `char state`: the runstate, R for running, S for stopped, Z for zombie;
- `int pid` and `int ppid`: the PID and parent PID;
- `int uid` and `int gid`: the UID and group ID.

This function will work on systems where:

- `/proc` holds process information (e.g. Linux systems). This is by far the preferred method to make snapshots. If this directory is not found, then:
- the `ps` command can output all necessary information. In these cases, `ps_makesnapshot()` will run `ps` to obtain the necessary information.

Note that the allocated memory must be freed using `ps_freesnapshot()`.

Example: The following code snippet makes a snapshot and prints the process ID and name of each encountered process.

```
PsList *l = ps_makesnapshot();
int i;

for (i = 0; i < l->nlist; i++) {
    PsEntry e = l->list[i];
    printf ("At pid %d: %s\n", e.pid, e.name);
}

ps_freesnapshot(l);
```

2.77 ps_sortsnapshot - sort a process list snapshot

Prototype: void ps_sortsnapshot (PsList *list)

Arguments: pointer to snapshot, as returned by ps_makesnapshot ()

Returns:

Description: This function sorts a process list snapshot, as made by ps_makesnapshot (). The list is sorted by process ID. To sort otherwise (e.g. by command name or runstate), run your own qsort ().

Example:

```
PsList *list;
int i;

list = ps_makesnapshot();
ps_sortsnapshot(list);
for (i = 0; i < list->nlist; i++)
    printf ("%d ", list->list[i].pid);
ps_freesnapshot (list);
```

2.78 random_ulong - generate random unsigned long number

Prototype: double random_ulong (int ndigits, int firstzero)

Arguments: ndigits is the number of digits to fill in the result, firstzero specifies, when !0, that the leftmost digit may be a zero

Returns: random number, returned as a double to avoid precision loss

Description: This function computes a randomly chosen unsigned long number, consisting of a given number of digits. E.g., to generate a unique key with 10 digits, where the leftmost digit may not be a zero, the call would be random_ulong (10, 0). Such keys are typically used to identify unique records in a database, or the similar.

Note that the caller should call srand () before using this function to ensure more 'randomness'.

Example: The following code snippet prints 10 (pseudo)random keys having 30 digits each. The leftmost digit of each key may not be a zero.

```
int i;

srand (time(0));
for (i = 0; i < 10; i++)
    printf ("%0f\n", random_ulong (30, 0));
```

2.79 rc_clean - reset resource-read values

Prototype: int rc_clean ()

Arguments:

Returns: number of removed RC values

Description: This function 'resets' the internal storage that is filled by rc_read(). The number of entries **previously** in storage is returned; the new number of entries is of course zero.

See also rc_read(), rc_value().

Example:

2.80 rc_read - read resource file

Prototype: int rc_read (char const *fname)

Arguments: fname is the name of a resource file to scan

Returns: -1 when file could not be opened for reading, otherwise the number of read settings

Description: Reads lines from a resource file and adds setting/value pairs to an internal storage area. The parsing and processing follows these rules:

- The lines require the following global format:
variable value The value is assigned to the symbolic variable name.
- Lines are stripped off shell-type comment. I.e., lines starting with a # sign are ignored altogether, and lines with # in the middle are split, so that the #-part is ignored. E.g.,

```
# This is comment.  
variable value # This too.
```

- Lines ending with a backslash are combined with the next line, to provide line continuation. E.g., consider the following (the text *note the backslash* is only for the illustration):

```
variable This \ # <-- note the backslash!  
variable has a very long value.
```

This assigns the value *This variable has a very long value* to variable *variable*.

- Values can hold *meta-values* in the format $$(VAR)$, where VAR is a name of an (other) variable. E.g., after:

```
who      world  
msg      Hello $(who)
```

the variable *msg* has the value *Hello world*.

- A special directive in the form *INCLUDE filename* may occur. This directive instructs rc_read() to scan filename. Note that the *INCLUDE* must occur in upper case. The function rc_read() allows an *INCLUDE* nesting up to 100 deep.

After reading a resource, the functions `rc_value()` or `rc_setting()` can be used to access the data. Function `rc_clean()` can be used to reset (clean up) the storage of resource data. Function `rc_setval()` can be used to add a value to the pool of resource variables.

Note 1: The `rc_*` functions share one data area for the stored variables. Also, subsequent calls to `rc_read` etc. will overwrite previously stored settings. (In other words, this family of functions is not "reentrant".)

Note 2: The argument `fname` will be overruled by a filename stored in the environment variable `RC_FILE`. This behavior is implemented to support "closed" existing programs that rely on `rc_read()`. This means that if an environment variable `RC_FILE` exists, then that file will be read; regardless of the argument `fname`.

Example: Consider the following rc file (say `/tmp/my.rc`):

```
# this is comment
var_one      setting_one      # more comment
var_two      setting_two \    # even more comment
              continued here  # yet more comment
base         /etc
rc           $(base)/rc.d
initlev      $(rc)/rc3.d
```

`INCLUDE /etc/my-other.rc`

Then the following will apply:

- `rc_read ("/tmp/my.rc")` will return 5
- `rc_value ("var_two")` will return "setting_two continued here"
- `rc_value ("initlev")` will return `/etc/rc.d/rc3.d`
- The settings from `/etc/my-other.rc` will also be included in the set.

2.81 `rc_setting` - return option/value setting, read via `rc_read()`

Prototype: `int rc_setting (int index, char **var, char **val)`

Arguments: `index` is the slot of setting pair to find

`var` is a pointer to a `char*`, which `rc_setting()` will alter to point to allocated memory, filled with the `index`-th variable name

`val` is a pointer to a `char*`, will be set to point to the `index`-th value

Returns: 0 when the setting at the slot is returned via `var` and `val`, or !0 when there is no setting at the specified slot

Description: Following a successful `rc_read()` call, the scanned data can be either accessed using `rc_setting()` or `rc_value()`.

The `rc_setting()` function is used to loop through all the settings in a resource file. The first setting starts at slot 0, the last is at slot `max-1`. The `max` value is the return value from `rc_read()`.

Note that `rc_setting()` allocates memory and sets the pointers `var` and `val` to point to that memory. The caller is responsible for freeing this memory, which must be done using `cmm_free()` or `cmm_cleanup()`.

Example: The following code reads a resource file `"/tmp/my.rc"` and prints all found entries.

```
int max, i;
char *var, *val;

max = rc_read ("/tmp/my.rc");
for (i = 0; i < max; i++) {
    if (rc_setting (i, &var, &val))
        error ("no setting at slot %d, while there should be\n",
            i);
    printf ("name: %s, setting: %s\n", var, val);
    cmm_free (var);
    cmm_free (val);
}
```

2.82 rc_setval - add value to resource-read set

Prototype: `void rc_setval (char const *name, char const *setting)`

Arguments: `name` is a variable name, `setting` is its value

Returns:

Description: This function adds a variable and a value to the list of resource settings. These settings are normally read from a resource file, using `rc_read()`. By using `rc_setval()`, entries can be added that are not stated in the file.

The `setting` that is added is also taken into account during the expansion of variables. E.g., when the setting is `$(somevar)`, then `rc_value(name)` will return the expansion of the variable `somevar`. Inversely, if a pre-existing variable `a` with the value `$(somevar)` exists, then after the following call:

```
rc_setval ("somevar", "a new value");
```

the value of variable `a` will be `a new value`.

Example:

2.83 rc_value - determine value of a resource file variable

Prototype: `char const *rc_value (char const *name)`

Arguments: `name` is the name of variable to retrieve

Returns: value of variable associated with `name`, or 0 if setting is not known

Description: This function is used after a resource file has been scanned using `rc_read()`.

When the variable name to determine is a-priori known, `rc_value()` can be used to determine its setting.

NOTE THAT the list of variables is scanned without respect to casing.

When the variable name(s) to search are not known, `rc_setting()` can be used to loop through all variable/value sets. See also `rc_read()`, `rc_setting()` and `rc_setval()`.

Example:

2.84 `rx_find` - find a regular expression in a string

Prototype: `char *rx_find (char const *regex, char const *str, int *lenp)`

Arguments: `regex` is an extended regular expression to match string with, `str` is the string to search in, `lenp` points to an `int` where the matched length is stored (when `lenp` is not 0)

Returns: The return value points into `str` where `regex` was matched, or 0 when

- the regular expression wasn't matched in `str`, or when
- the regular expression was invalid

Description: This function attempts to find `regex` in `str`. A pointer to the first occurrence is returned. When `lenp` is not 0, then the number of matched chars is stored there.

To test whether an expression is valid, use `rx_match()`, whose return value may not be `RX_BADREGEX`. This function `rx_find()` doesn't provide that capability; the return value 0 is used for situations where the match cannot be made **and** for situations where `regex` is invalid.

This function heavily relies on the POSIX standard. Some "less than POSIXy" systems will lack the support for regular expressions.

Normally, matching will be with regard to casing. Use `rx_ignorecase(1)` to match while ignoring casing, or switch back to case-sensitivity with `rx_ignorecase(0)`.

Example:

2.85 `rx_ignorecase` - ignore casing in regular expression matching

Prototype: `void rx_ignorecase (int onoff)`

Arguments: `onoff` is not-0 to select case-insensitive matching (ignoring), or 0 to select case-sensitive matching (the default)

Returns:

Description: This function selects case-sensitivity during regular expressions matching. The choice affects how `rx_match()` and `rx_find()` match strings against regular expressions.

Example:

2.86 `rx_match` - match regular expression and a string

Prototype: `int rx_match (char const *regex, char const *str)`

Arguments: `regex` is an extended regular expression to match string with, `str` is the string to compare to `regex`

Returns: `RX_MATCH (0)` string matches `regex`,
`RX_NOMATCH (>0)` string does not match `regex`,
`RX_BADREGEX (<0)` bad regular expression

Description: Does a single match of a string against a regular expression. Notably, this function is NOT suitable for:

- rematching strings with one regular expression. If you want that, use the "standard" `regcomp()` and `regexec()`.
- determining WHAT PART of the string matches. The function just returns whether it matches or not.
- matching substrings and storing the result. The function just returns whether the string matches or not.

This function heavily relies on the POSIX standard. Some "less than POSIXy" systems will lack the support for regular expressions.

Normally, matching will be with regard to casing. Use `rx_ignorecase(1)` to match while ignoring casing, or switch back to case-sensitivity with `rx_ignorecase(0)`.

Example:

2.87 `set_memerr` - set the "out of memory" handler function

Prototype: `Errfunptr set_memerr (Errfunptr newhandler)`

Arguments: `newhandler` is a pointer to the new out of memory handler function

Returns: pointer to the previous out of memory handler

Description: Redefines the "out of memory" handler to call `newhandler()` during out-of-memory errors. A pointer to the previous out-of-memory handler is returned. The type `Errfunptr` is a pointer to a function that has a void argument list and that returns void.

Example: The following snippet defines and installs a handler that will close a file `f` and then exit. In `main()` the handler is installed, memory is allocated, and then the previous handler is re-installed.

```
void myhandler (void) {
    extern FILE *f;
    fclose (f);
    error ("out of memory\n");
}

int main () {
    Errfunptr prev;
    void *memory;

    .
    prev = set_memerr (myhandler);
    memory = xmalloc (10000);
    set_memerr (prev);
    .
    .
}
```

A more "elaborate" approach would be to let the out of memory handler (`myhandler()` in the above example) "chain" the call to any previous handler(s).

2.88 setprogname - set the program name for `error()` and `msg()`

Prototype: `void setprogname (char const *pname)`

Arguments: `pname` is the program name to use when displaying messages

Returns:

Description: This function defines the program name for display purposes.

Example: `setprogname ("myprog");`
`error ("Something bad has happened.\n");`

This leads to something similar to: *myprog: Something bad has happened.*

2.89 sf_sleep - signal-free sleep implementation

Prototype: `int sf_sleep (double sec)`

Arguments: `sec` is the sleeping time, as a double-precision value

Returns: 0 when the sleeping has failed (check `errno` in that case), !0 otherwise

Description: This function implements a signal-free sleeping. In contrast to `sleep(3)`, no signals occur, and hence, no signal catcher activity is disturbed.

The argument `sec` specifies the sleep time, up to microsecond precision.

Example:

2.90 sig_catch - flag-based signal handler

Prototype: `void sig_catch (int sig, int *flag)`

Arguments: `sig` is the signal to catch

`flag` is a pointer to an `int` value, which will be incremented when the signal is seen

Returns:

Description: This function implements a fairly simple signal handler, but is nevertheless useful in most situations. The function is called with two arguments: the signal to catch, and a pointer to an `int` value which is used as a flag when the signal is seen. The caller must at some given time investigate the flag integer to see whether the signal arrived or not.

Note that this function can be called more than once, to specify the catching of all signals that should be caught. **Note also** that the signals that are already caught by `sig_catch()` should not be specified as triggers via the standard `signal()` function; only one of the methods will work.

Furthermore, all restrictions of signal catching apply. E.g., the program behaviour is undefined when a `SIGSEGV` is caught, and `SIGKILL` signals cannot be caught. See the manpage on the `signal()` function for more information.

Example: The following code snippet shows the usage of this function. Two signals (1 and 15) are caught. The main loop investigates the flag values to see whether a signal did indeed arrive.

```
int main () {
    int sighthup_seen = 0,
        sigterm_seen = 0;

    sig_catch (SIGHUP, &sighthup_seen);
    sig_catch (SIGTERM, &sigterm_seen);

    while (1) {
        if (sighthup_seen) {
            printf ("Seen a SIGHUP\n");
            sighthup_seen = 0;
        }
        if (sigterm_seen) {
            printf ("Seen a SIGTERM\n");
            exit (0);
        }
        sleep (1);
    }
}
```

```
    }  
  
    return (0);  
}
```

2.91 skipline - skip to the next line in a string buffer

Prototype: `char *skipline (char const *buf)`

Arguments: `buf` is the buffer to inspect

Returns: Pointer to the next line in the buffer, or 0

Description: This function returns a pointer to the next line in the string pointed to by `buf`. I.e., all characters up to and including the next `\r\n` or just `\n` are skipped. A pointer to the position just beyond that `\r\n` or `\n` is returned.

Example:

```
char *buf = "hello\nworld";  
printf (skipline (buf));           // prints "world"  
  
int count = 0;  
char *cp;  
for (cp = buf, count = 0;  
     cp = skipline (cp);  
     cp)  
    ;  
printf ("There are %d newlines in the buffer\n", count);
```

2.92 stab_addsprintf - add a string to a string table using sprintf()-like capabilities

Prototype: `int stab_addsprintf (Strtab *tab, char const *fmt, ...)`

Arguments: `tab` is a pointer to a `Strtab` table to add a string to
`fmt` and `...` are the format string and remaining arguments

Returns: Number of strings in `tab` after addition of the new string

Description: This function manipulates `tab` similar to `stab_addstr()`, except for the fact that the string to add is specified using a format string and variable arguments.

See also `stab_addstr()`, `stab_free()`

Example: The following code snippet adds a formatted string to a table. The output will be something like:

```
Hello World  
3 + 4 = 7
```

```
Strtab tab = {0, 0};
int i, nstr;

stab_addstr (&tab, "Hello World");
nstr = stab_addsprintf (&tab, "%d + %d = %d", 3, 4, 3 + 4);
for (i = 0; i < nstr; i++)
    printf ("%s\n", tab.s[i]);
```

2.93 stab_addstr - add string to string table

Prototype: int stab_addstr (Strtab *tab, char const *newstr)

Arguments: **tab** is a pointer to a string table to which **newstr** will be added
newstr is the string to add

Returns: Number of strings in the table after **newstr** was added

Description: This function adds a string to the string table pointed to by **tab**.

String tables (type **Strtab**) are structures with two fields:

- a counter **n**, specifying the number of contained strings,
- **s**, an array of **char*-s**, pointing to the contained strings.

When using string tables, the following general rules apply:

- String tables must always be initialized to zero values, as in **Strtab mytab = {0,0}**. The functions **stab_*()** depend on a valid initialization.
- To determine how many strings are contained in a table, **mytab.n** may be inspected. Also, most **stab_*()** functions have a return value that serves the same purpose.
- To access strings in the table, **mytab.s[index]** is used, with **index** ranging from 0 to **mytab.n - 1**

See also **stab_free()** which de-allocates the **Strtab** components, and **stab_addsprintf()** to add strings in a **sprintf()**-like manner, or **stab_find()** and **stab_ifind()** to find string in a table.

Example: **Strtab stab = {0, 0};**

```
int i;

for (i = 0; i < 5; i++)
    stab_addstr (&stab, "Hello World\n");
printf ("%d strings in table\n", stab.n);
for (i = 0; i < stab.n; i++)
    printf ("String %d: %s\n", i, stab.s[i]);
stab_free (&stab);
```

2.94 `stab_find` - find string in a string table

Prototype: `int stab_find (Strtab const *tab, char const *string)`

Arguments: `tab` is table to search in

`string` is the string to search for

Returns: index in table, or -1 if the search string could not be found

Description: The table pointed to by `tab` is searched for `string`. See also `stab_ifind()`, which searches case-insensitively.

Example: `Strtab mytab;`

```
int i;
```

```
.
```

```
.
```

```
if ( ( i = stab_find (&mytab, "hello world") ) >= 0 )  
    printf ("string found at %d: %s\n",  
           i, mytab.s[i]);
```

2.95 `stab_free` - discard memory used by a `Strtab`, reset member fields

Prototype: `void stab_free (Strtab *stab)`

Arguments: `stab` is a pointer to the string table to reset

Returns:

Description: De-allocates the memory held by the `Strtab` structure pointed to by `stab`. The member fields are re-set to 0 (for subsequent usage).

See also `stab_addstr ()`.

Example:

2.96 `stab_ifind` - find string in a string table, without regard to casing

Prototype: `int stab_ifind (Strtab const *tab, char const *string)`

Arguments: `tab` is the table to search in

`string` is the string to search for

Returns: index in the table, or -1 if `string` could not be found

Description: This function searches `tab` for `string`, without regard to casing. See also `stab_find()`.

Example:

2.97 `stab_maxlen` - determine max string length in a string table

Prototype: `int stab_maxlen (Strtab const *tab)`

Arguments: `tab` is a pointer to the string table

Returns: Length of the longest string in the table, or 0 if there are no strings

Description: This function determines how long the longest string in `tab` is. It is mainly used before outputting strings, to determine the maximum span. (The implementation of this function is quite trivial.)

Example:

2.98 `stab_popstr` - pop a string off a string table

Prototype: `char *stab_popstr (Strtab *tab)`

Arguments: `tab` points to the string table to pop from

Returns: popped string, or 0 when there is nothing left to pop

Description: When a string table is used as a stack, strings can be added to it via `stab_addstr()` and `stab_addsprintf()`. These functions "push" strings onto the stack.

The function `stab_popstr()` removes the last added string, and returns it, thereby "popping". The return value points to allocated memory; the caller is responsible for freeing it. The freeing **must** be done using either `cmm_free()` or `cmm_cleanup()`.

A stack underflow is indicated by a 0-pointer. Note that the remaining stack size can always be retrieved via `tab->n`.

See also `stab_addstr()`, `stab_addsprintf()`.

Example: The following example pushes 10 strings onto the stack and pops them again.

```
Strtab stack = {0, 0};
int i;
char *cp;

// Push 10 strings onto the stack
for (i = 0; i < 10; i++)
    stab_addsprintf (&stack, "string number %d", i);

// Now pop, until the stack is empty. Also release the
// popped memory.
while ( (cp = stab_popstr (&stack)) ) {
    printf ("popped string: %s\n", cp);
    cmm_free (cp);
}
```

2.99 `str_add` - add string to reallocatable string buffer

Prototype: `char *str_add (char *buf, char const *newstr)`

Arguments: `buf` is a reallocatable buffer, or 0 if none exists yet
`newstr` is the string to add to `buf`

Returns: pointer to a new buffer

Description: This function dynamically reallocates the memory used by `buf` and adds `newstr` to it.

Note: The caller is responsible for freeing the memory afterwards, when `buf` is no longer needed. This **must** be done with either `cmm_free()` or with `cmm_cleanup()`.

Example: `char *buf = 0;`

```
buf = str_add (buf, "Hello");
buf = str_addchar (buf, ' ');
buf = str_add (buf, "world!");

printf ("Buffer is now: %s\n", buf);

cmm_free (buf);
```

2.100 `str_addchar` - add a single character to a relocateble string buffer

Prototype: `char *str_addchar (char *buf, int newchar)`

Arguments: `buf` points to pointer to relocatable memory, or 0 if none has been reserved yet
`newchar` is the character to add to the buffer

Returns: pointer to a new string

Description: This function adds a single character to `buf`. See also `str_add()`.

Example:

2.101 `str_equal` - compare strings without regard to casing, returns matched characters

Prototype: `int str_equal (char const *a, char const *b)`

Arguments: `a` and `b` are two strings to compare

Returns: Number of matched characters, which is `strlen(a)` and `strlen(b)` when the strings fully match (disregarding casing).

Description: The strings are compared over their respective lengths. The number of matching character is returned. See also `strnicmp()`.

Example: `int i;`

```
i = str_equal ("HELLO", "hello");      // i == 5
i = str_equal ("HELLO", "h");          // i == 1
i = str_equal ("HELLO", "world");     // i == 0
```

2.102 `stricmp` - compare two strings without regard to casing

Prototype: `int stricmp (char const *a, char const *b)`

Arguments: `a` and `b` are the strings to compare

Returns:

- `<0` if `a` is "less",
- `>0` if `a` is "more",
- `0` if `a` and `b` are equal

Description: This function is similar to `strcmp()`, except that the string comparisons are made without regard to casing. This function mainly exists because some Unices lack `stricmp()` or `strcasecmp()`.

See also `stristr()`, `strupr()`.

Example:

2.103 `stripstring` - strip leading and trailing spaces in a buffer

Prototype: `char *stripstring (char *buf)`

Arguments: `buf` is the buffer to strip

Returns: Pointer to stripped buffer. The buffer itself is not moved in the memory; its contents are.

Description: The leading and trailing spaces (including tabs and newlines) are removed from the buffer.

Example:

2.104 `strireplace` - replace a substring in a string

Prototype: `char *strireplace (char **src, char *sub1, char *sub2)`

Arguments: `src` is a pointer to the string to change,
`sub1` is the substring that is searched for,
`sub2` is the replacement.

Returns: `char*` pointer to changed string

Description: The string indicated by the first argument is changed. Its substring `sub1` is replaced by `sub2`. The searching for `sub1` within the string is case-insensitive.

The memory pointed to by `*src` must be dynamically allocated. This function will re-allocate and re-shift that memory. Also, a pointer to this memory block is returned.

Example: `char *x;`

```
x = xstrdup ("Goodbye World!\n");  
x = strireplace (&x, "goodbye", "Hello");
```

2.105 `stristr` - search for substring in a string without regard to casing

Prototype: `char *stristr (char const *buf, char const *search)`

Arguments: `buf` is the buffer to search in
`what` specifies what to search for

Returns: Pointer within `buf` where `search` occurs, or 0 when `search` was not found. A pointer to the first character of `buf` is also returned when `search` is 0.

Description: Searches in `buf` for the first occurrence of `search` and returns a pointer to it. The string search is case-insensitive. This function is therefore similar to `strstr()`, except for the insensitiveness.

Example:

2.106 `strnicmp` - compare two strings over a given length, ignoring casing

Prototype: `int strnicmp (char const *a, char const *b, int len)`

Arguments: `a` is the first string
`b` is the second
`len` is the length over which to compare

Returns:

- 0 when both strings are equal,
- >0 when a is "bigger",
- <0 when b is "bigger"

Description: Strings a and b are compared over len bytes, without regard to casing. This function is similar to strstr(), except that the comparisons are case-insensitive.

Example:

2.107 str_printf - fill a string using printf capabilities

Prototype: char *str_printf(char *fmt, ...)

Arguments: fmt and ... are the (standard) printf()-like arguments

Returns: pointer to an allocated buffer, filled according to the format string and the other arguments

Description: This function is comparable to sprintf(), except that the built string is dynamically allocated. The caller is responsible to returning this memory to the pool using cmm_free().

Example: char *x = str_printf("Hello %s...", "world");

2.108 str_short - return shortened duplicate of string, with ... appended

Prototype: char *str_short(char const *buf, int len)

Arguments: buf is the buffer to duplicate and shorten,
len is the requested duplicate length

Returns: pointer to a (shortened) duplicate

Description: Makes a duplicate of the buffer and shortens it to len. If the buffer is shortened, then ... is appended. Also, non-printable characters are replaced by a dot. This function can be used during e.g. logging.

Note: The caller is responsible for freeing the memory that is pointed to by the return value. This must be done using either cmm_free() or cmm_cleanup().

Example:

```
char verylongbuf [256];
char *cp;
.
.
cp = str_short (verylongbuf, 50);
logmsg ("buf = [%s]\n", cp);
cmm_free (cp);
```

2.109 `strupr` - convert string to upper case

Prototype: `char *strupr (char *buf)`

Arguments: pointer to `buf` where all characters are converted to upper case

Returns: pointer to the converted buffer

Description: This function converts all characters in `buf` to upper case. The conversion is done in-place; i.e., the caller should make a copy of `buf` if its contents should be preserved.

Example:

2.110 `str_vprintf` - fill a string using `vprintf` capabilities

Prototype: `char *str_vprintf (char const *fmt, va_list args)`

Arguments: `fmt` and `args` are `vprintf()`-like arguments

Returns: pointer to a formatted string

Description: This function is similar to `vsprintf()`, except that the string is created using allocated memory. There is therefore no risk of buffer overflow.

Example:

2.111 `suggest_ext` - suggest an appropriate extension for a file name

Prototype: `char *suggest_ext (char const *filename)`

Arguments: `filename` is the name of the file to inspect

Returns: Appropriate name for the mentioned file, including a fitting extension. The suggested name is in allocated memory.

Description: This function inspects the file stated by the argument `filename` and determines the type. For known types, an extension is suggested. Known types lead to the following extensions:

- `.tiff`,
- `.png`,
- `.gif`,
- `.jpg`,
- `.bmp`,
- `.xpm`,
- `.doc`,

- .rtf,
- .xml.

In order to succeed in determining an appropriate extension, the file must exist. Also note that the suggested name for MS-Office document types is always .doc. There is no support for different Office document types, e.g., .xls.

The return value is a suggested name. This function does not actually rename the file in question. The suggested name is allocated; the caller should free the memory when appropriate. The freeing must be done using `cmm_free()` or `cmm_cleanup()`.

See also `supply_ext()`, `filetype()`.

Example: Assuming that the file `/tmp/myfile` is a JPEG picture, this code snippet will print out `/tmp/myfile.jpg`:

```
char *newname;

newname = suggest_ext ("/tmp/myfile");
printf ("%s\n", newname);
cmm_free (newname);
```

2.112 `supply_ext` - supply an appropriate extension for a file name

Prototype: `char *supply_ext (char const *filename)`

Arguments: `filename` is the name of file to inspect

Returns: New name for the mentioned file, including a fitting extension. The suggested name is in allocated memory. The file is renamed to the new name.

Description: This function inspects the file stated by the argument `filename` and determines the type. When the file matches a known type, the file is renamed to the new name. For the supported known types see `suggest_ext()` or `filetype()`.

Note that in order to succeed in determining an appropriate extension, the file must exist and the process must be able to rename it. When these conditions are not met, then the returned "new name" is just a duplicate of the old name. In any case, the caller should free the memory occupied by the returned name when appropriate. This freeing **must** be done with either `cmm_free()` or `cmm_cleanup()`.

See also `filetype()`, `suggest_ext()`.

Example: Assuming that the file `/tmp/myfile` is a JPEG picture, this code snippet will print out `/tmp/myfile.jpg` and rename the file to the new extension:

```
char *newname;

newname = supply_ext ("/tmp/myfile");
printf ("%s\n", newname);
cmm_free (newname);
```

2.113 tmp_clean - clean up files created by tmp_file ()

Prototype: void tmp_clean (void)

Arguments:

Returns:

Description: All filenames created by tmp_file () or tmp_fopen () are removed. Note: for successful removal, the files must be closed. The caller is responsible for that. The removal of files is not checked for success or failure.

See also tmp_file (), tmp_fopen (), tmp_fileprefix ().

Example:

2.114 tmp_file - construct name for a temporary file

Prototype: char const *tmp_file (void)

Arguments:

Returns: Pointer to the temporary filename

Description: A temporary filename is constructed. Its copy is put in a local Strtab storage. That way, all filenames constructed by tmp_file () are known in e-lib. When tmp_cleanup () is called, all the files are removed.

The temporary files consist of a prefix, followed by the process ID, a dot, and an order number. The prefix is normally /tmp/e. Therefore default temporary filenames are /tmp/e*processid.ordernumber*, e.g., /tmp/e002391.1. The directory part of the prefix can be redefined by calling tmp_fileprefix ().

Note that this function is neither reentrancy-safe or thread-safe. It's a pretty basic way of making temporary file names. It's **strongly suggested** that you take a look at tmp_fopen (), which is both reentrancy and thread-safe.

See also tmp_clean (), tmp_fileprefix ().

Example: The following code snippet shows the basic usage. Two temporary filenames are constructed, one under /tmp, the other under /usr/tmp. The last call to tmp_clean () removes all files having the temporary names.

```
char const *f1, *f2;

f1 = tmp_file ();
tmp_fileprefix ("/usr/tmp");
f2 = tmp_file ();

printf ("First one:      %s\n"
        "Second one:      %s\n", f1, f2);
tmp_clean ();

// This outputs something like:
```

```
//      /tmp/e123456.1
//      /usr/tmp/e123456.2
```

2.115 tmp_fileprefix - define the file prefix for temporary files

Prototype: void tmp_fileprefix (char const *prefix)

Arguments: prefix is the name prefix to use for temporary files

Returns:

Description: Defines the prefix part of temporary files, created by tmp_file() and tmp_fopen().

The argument prefix may in fact also contain a directory part, as in /tmp/, or /usr/tmp/my.

The default temporary file prefix is /tmp/e. Any calls to tmp_fileprefix() will overrule this default.

See also tmp_file(), tmp_clean().

Example: // from now on make temp files as /usr/tmp/myprog*
tmp_fileprefix ("/usr/tmp/myprog");

2.116 tmp_fopen - open a tmp file for writing

Prototype: FILE *tmp_fopen (char **filename)

Arguments: filename is set to point to the name of the opened file, if filename is not 0

Returns: opened stream, or 0 upon error

Description: This function attempts to open a temporary file for writing. The process is atomic and therefore reentrancy safe and thread-safe. (See e.g. a discussion of mktemp() and mkstemp() in the manpages for more info.)

The return value is the opened file, or 0 on error. In the latter case errno may be inspected to see what the error is.

The argument filename may be set by the caller to the address of a char*. In that case, tmp_fopen() will set this pointer to point to the name of the file in question. When filename is 0, then tmp_fopen() will not provide information about the temporary file, but will only add this name to the internal list of temporary files. After closing, all temporary files that were created using tmp_fopen() or tmp_file() may be removed using tmp_clean().

Example: The following code snippet opens a temporary file and writes a message into it. Then the file is closed and cleaned up. The temporary file is created in /usr/tmp/ with a prefix myfile. The file will therefore be created as something like /usr/tmp/myfileABCDEF.

```
char *n;
FILE *f;

tmp_fileprefix ("/usr/tmp/myfile");

if (! (f = tmp_fopen (&n)) )
    error ("Failed to open tmpfile %s: %s\n",
          n, strerror (errno));
printf ("Temp file %s is now open for writing.\n",
        n);

fprintf (f, "Hello World!\n");
fclose (f);
printf ("Temp file %s is now used and closed.\n",
        n);

tmp_clean();
printf ("Temp file %s is now removed.\n",
        n);
```

2.117 xcalloc - allocate memory or issue an error

Prototype: void *xcalloc (int n, int sz)

Arguments: n is the number of elements to allocate, sz is the size of each element

Returns: Pointer to allocated memory

Description: Allocates the memory or calls `out_of_memory()` to issue an error. The allocated memory is set to zero. Internally, the `cmm*` () set of functions is used so that a memory trail usage trail is available. Therefore, **note that** the thus allocated memory may only be freed using `cmm_free()` or `cmm_cleanup()`.

See also `set_memerr()`, `out_of_memory()`, `x...` ().

Example:

2.118 xmalloc - Allocate memory or issue an out-of-memory error

Prototype: void *xmalloc (int size)

Arguments: size is the number of bytes to allocate

Returns: Pointer to allocated memory

Description: Allocates the memory or calls `out_of_memory()` to issue an error. Internally, this function uses `cmm_malloc()` so that the memory usage trail is recorded. Therefore, **note that** `xmalloc(-d)` memory may only be freed using `cmm_free()` or `cmm_cleanup()`.

See also `set_memerr()`, `out_of_memory()`, `x*()`, `cmm*()`.

Example:

2.119 `xrealloc` - Reallocate memory or issue an error

Prototype: `void *xrealloc (void *oldmem, int size)`

Arguments: `oldmem` is a pointer to previously allocated memory, or 0 if no allocated memory yet present

`size` is the number of bytes to allocate

Returns: Pointer to reallocated memory

Description: Allocates the memory or calls `out_of_memory()` to issue an error. When `oldmem` is 0, the memory is just allocated. Internally, the function maps to the `cmm*()` functions so that a memory usage trail is available. Therefore **note that** you may only free `xrealloc(-d)` memory with `cmm_free()` or with `cmm_cleanup()`, and not with the standard `free()`.

See also `set_memerr()`, `out_of_memory()`, `x*()`, `cmm*()`.

Example:

2.120 `xstrdup` - Duplicate in memory a string or issue an error

Prototype: `char *xstrdup (char const *s)`

Arguments: `x` is the string to duplicate

Returns: pointer to duplicate

Description: This function is similar to `strdup()`, except that `out_of_memory()` is called when a string duplication fails. Furthermore, it uses internally the `cmm*()` set of functions, so that a memory usage trail is available. Therefore **note that** you may only free memory returned by `xstrdup()` with `cmm_free()` or `cmm_cleanup()`.

`xstrdup(0)` will return a 0-pointer.

See also `set_memerr()`, `out_of_memory()`, `x...()`.

Example:

2.121 zip_compress - compress a buffer

Prototype: `char *zip_compress (char *buf, unsigned long *len)`

Arguments: `buf` is the buffer to compress, in allocated memory
`len` is the length of the buffer

Returns: compressed buffer, in re-allocated memory

Description: The buffer pointed to by `buf` is compressed if the compression yields a gain greater to or equal to the defined threshold. The threshold is by default 95 and is controlled via `zip_threshold()`. The buffer is compressed in-place (i.e., its memory is reallocated) and the size `sz` is adjusted accordingly.

See also `zip_decompress()`, `zip_threshold()`.

Example: The following example fills a buffer with 10 strings, then compresses the buffer. After decompression, the contents are shown.

```
char *buf = 0;
int i;
long sz;

for (i = 0; i < 10; i++)
    buf = str_addprintf (buf, "This is the %d'th string.\n", i);

sz = strlen (buf);
printf ("Size before compression: %ld\n", sz);

buf = zip_compress (buf, &sz);
printf ("Size after compression: %ld\n", sz);

buf = zip_decompress (buf, &sz);
printf ("Size after decompression: %ld\n"
        "Buffer contents: %s\n", sz, buf);
```

2.122 zip_decompress - decompress a buffer

Prototype: `char *zip_decompress (char *buf, unsigned long *sz)`

Arguments: pointer to the decompressed buffer, in re-allocated memory

Returns: `buf` is the buffer to decompress, in allocated memory
`sz` is the size of the buffer

Description: The buffer pointed to by `buf` is decompressed (to a greater size) and the associated memory is re-allocated. The size, pointed to by `sz`, is adjusted.

See also `zip_compress()`, `zip_threshold()`.

Example: See `zip_compress()`.

2.123 zip_threshold - set the value for compression routines

Prototype: void zip_threshold (int diff)

Arguments: diff is the number of bytes to gain with compression

Returns:

Description: This function sets the threshold for zip compression. The threshold value must be reached as a gain when compressing, otherwise zip_compress() will leave its buffer in an uncompressed state.

The default threshold value is 95.

See also zip_compress(), zip_decompress().

Example: The following sample function compress() will compress a buffer only when the gain is more than 40%.

```
void *compress (char *buf,
                unsigned long *len) {
    // Determine the requested gain: 60% off
    // the original length. Then set the
    // requested gain.
    unsigned long req_gain = *len * 4 / 10;
    zip_threshold ( (int) req_gain );

    // Now zip it up.
    return (zip_compress (buf, len));
}
```

Chapter 3

C++ Class Reference

3.1 eArray - Storage in an array format

3.1.1 Synopsis

```
#include "e-array.h"

eArray mystore;           // storage definition
eStr  element;           // some eStorable element

mystore.add (element);   // add any eStorable element
cout << "There are "
     << mystore.n ()    // # of stored elements
     << " elements in storage.\n";

eStorable *el = mystore.get (0); // get a handle on the
                                // first one

mystore.push (element);    // add to end of storage
el = mystore.pop ();       // remove from end
mystore.unshift (element); // add to beginning of storage
el = mystore.shift ();     // remove from beginning

delete (mystore.shift());  // destructive removal
                                // from the start
delete (mystore.pop ());  // or from the end

mystore.empty ();         // throw everything away
```

3.1.2 Description

`eArray` is one of the storage types to handle `eStorable` objects. Note that `eArray` is storable itself, so that one may make arrays of arrays and so on.

It is the programmers responsibility to typecast objects that are put into storage and that are retrieved to the same type. E.g., a typical `eStorable` object is `eStr`. When retrieving elements via `eArray::get()`, the return pointer can be typecast to an `eStr` so that the members `eStr::s()` and so on are usable.

3.1.3 Storing and retrieving

There are two methods for storing, and to corresponding for retrieving. The matching pairs are

- `push()` and `pop()`, and
- `shift()` and `unshift()`.

The method `add()`, stated above, is an alias for `push()`. Using these methods a FIFO or LIFO storage can be created:

- Using `push()` and `pop()`, a FIFO storage (stack) is created. It is of course also possible to use `shift()` and `unshift()`, but this pair is more resource-intensive; it includes more memory handling, so `push()` / `pop()` is preferred.
- Using `push()` and `shift()`, a LIFO (queue) is created. Alternatively, one may use `unshift()` and `pop()`.

The two methods that retrieve an element and remove it from storage (`shift()` and `pop()`) leave the element's memory occupied. Therefore, to remove an element from storage and to also destroy its own occupied memory, one should use

```
delete (storage.pop());
```

or

```
delete (storage.shift());
```

3.2 eHttpHdr - HTTP Request Header handling

3.2.1 Synopsis

```
#include "e-httphdr.h"

eHttpHdr hdr;

int ret =  hdr.read (0, 20);           // scan a header, read
                                       // from file descriptor 0,
                                       // using a timeout of
                                       // 20 secs

if (ret > 0)
    prog->msg ("WARNING: Timeout\n");  // return value > 0:
                                       // timeout
else if (ret < 0)
    prog->error ("Error!\n");         // return value < 0:
                                       // error

hdr.write (1);                        // write it to stdout

eStr *v = hdr.request ();             // HTTP request:
                                       // GET, POST, etc..

cout << "Request: " << rq->s() << '\n';

if (v = hdr.lookup ("Content-Type"))  // Lookup header lines
    cout << "Content type line: "
         << v->s() << '\n';

char *cp;
if (cp = hdr.value ("Content-Type"))  // Lookup entry values
    cout << "Value of content type: "
         << cp << '\n';

hdr.setvalue ("Content-Type",        // Change entries
             "text/html");

cout << "The full header is:\n"      // ostream support
     << hdr;
```

3.2.2 Description

This class is derived from `eStrArr` and manages HTTP headers. That means that all the "usual" `eStrArr` functions are also present: `get()`, `n()`, `shift()`, `add()`, etc..

The extra functionality consists of:

Reading and writing a HTTP header The header is "filled" by calling `read(int fd, int timeout)`.

The data is read from the file descriptor `fd`, and the function returns `>0` when the timeout is exceeded. Normally, when a valid HTTP header waits on `fd`, no timeout should occur.

The return value from `read(fd, timeout)` can be:

- 2 signals a reading error,
- 1 signals an error in selecting the file descriptor for reading,
- 1 signals a timeout,
- 0 signals success.

Once a header is read, then the object in question contains the header lines, up until to and including one empty string (the header-end signal). These lines **do not** contain the CRLF sequence that arrived with these lines. Therefore, after reading a header say `hdr`:

`hdr.n()` will contain the number of header lines, including one (last) zero-length line;
`hdr.get(2)` will return the third line of the header, etc..

The function `write(int fd)` will write the header in HTTP format, i.e., each entry, followed by CRLF, to file descriptor `fd`.

Looking up HTTP header information The function `lookup(entry)` searches for a header line that starts with `entry`. The argument is either a `char const*`, or an `eStr &`.

The function `value(entry)` searches for a header line that starts with the given entry, but returns not the whole line, but only the "value" of the combination `entry: value`. The return value is a `char*`.

Changing values in the header The function `setvalue(entry, value)` searches for a header line that starts with `entry`, and changes its value.

3.3 eHttpRequest - HTTP Request handling

3.3.1 Synopsis

```
#include "e-httprequest.h"

eHttpRequest rq;

int ret = rq.read (0, 20);           // scan header + content,
                                   // read from fd 0 (stdin),
                                   // timeout is 20sec

switch (ret) {
    case 0:                          // 0: all OK
        cout << "Request read.\n";
        exit (1);
    case -1:                          // -1: select() failed
        cout << "Waiting failed.\n";
        exit (1);
    case -2:                          // -2: read() failed
        cout << "Read failed.\n";
        exit (1);
    case 1:                          // 1: timeout occurred
        cout << "WARNING - timeout\n";
```

```
        break;
    }

    rq.write (1); // write back to stdout

    // For other functionality: see eHttpHdr.
```

3.3.2 Description

This class handles HTTP headers (see `eHttpHdr`) and any following content; i.e., the whole HTTP request is handled.

The class is derived from `eHttpHdr`, so that the corresponding functions are available (e.g., `lookup()`, `value()`, `setvalue()`).

The read-write primitives basically scan the header, and optionally process any following information. The supported HTTP types are:

Content-Length: *decimal-number* When the header contains the specification of a content, over a given content length, then the content is scanned too. Example:

```
GET / HTTP/1.1
Content-Length: 100

[content block over 100 bytes expected here]
```

Transfer-Encoding: *chunked* When the header specifies that chunks follow the header, then the chunks are also scanned. Each chunk consists of a hexadecimal size specifier, followed by CRLF, and followed by the chunk. The chunk list ends when `chunksize 0` is seen. Example:

```
GET / HTTP/1.1
Transfer-Encoding: chunked

f
[chunk of 15 bytes]
a
[chunk of 10 bytes]
0
```

Connection: *closed* When the webserver indicates a "closed" connection, then all content will be read (until no more is available).

Default handling For all other situations, `eHttpRequest` will assume that there is no content. Only the header is scanned.

3.4 eObj - basic class type of the e++ library

3.4.1 Synopsis

```
#include "e-obj.h"
#include "streams"

eObj basicobj;
cout << "Version of e++ support library: "
      << basicobj.version () << '\n';
```

3.4.2 Description

This is the basic type from which all classes of the e++ library are derived. Basically, this type does version tracking.

3.5 eProgram - basic program functions

3.5.1 Synopsis

```
#include "e-program.h"

// Constructors
eProgram prog; // nameless construction
eProgram prog ("myprog"); // named construction
eProgram prog (int argc, char **argv, // construction with opts
               "vVa:b:c", // parsing, version ID and
               "1.01", // short description
               "Incredibly Smart Prog");

// General functions
prog.msg (1); // enable messages
prog.msg (0); // or disable them
prog.wrapmsg (1); // enable msg wrapping
// (the default)
prog.wrapmsg (0); // or disable it
prog.msg ("%s", "hello world"); // print a message to stdout
// when -v flag was seen or
// after msg(1) was issued
prog.error ("error code %d", 15); // print to stderr and stop

// Handling program options
prog.cmdline (argc, argv, "a:b:c"); // parse options (or use
// the constructor as
```

```
prog.cmdline ("a:b:c", // shown above)
              "/path/prog prog -a10 -b/tmp -c"); // re-parse commandline
                                                // at runtime
                                                // (note the path and
                                                // program name!)

char const *val;
if ( (val = prog.option ('a')) ) // retrieval of options
    cout << "Value of flag a is "
          << val << '\n';
if ( (prog.option ('c')) )
    cout << "Flag -c was set\n";

// Handling program arguments
for (int i = 0; i < prog.narg (); i++) // prog.narg() is the # of args
    cout << "Argument " << i << " is " // prog.arg(i) is the i-th argument
          << prog.arg (i) << '\n';
```

3.5.2 Description

This class provides basic program tasks: handling of the program name in messages, and parsing the command line.

Parsing program options The preferred constructor is the third alternative above, using `argc`, `argv`, an option specifier, a version ID and a short description. The option specifier is a string a-la `getopt(3)`: options are stated as characters, optionally followed by a `:` to indicate that this option requires an argument. The options can then be retrieved via the method `option()`.

The version ID and description are used internally, when a `-V` flag is seen (a-la all e-programs). If you do not want this behavior, then leave the version and description arguments empty. These two last arguments of the constructor are optional.

The `-v` flag causes `msg()` calls to become 'active', i.e., to show the output. It is the same as calling `msg(1)`.

Typically the following code sample will apply. A pointer to an `eProgram` object is used, the object is instanciated once `main()` starts. The object is global, so that throughout the program messages and errors can be sent. Note that the flags `-v` and `-V` are also stated.

```
eProgram *prog;

int main (int argc, char **argv) {
    prog = new eProgram (argc, argv, "vVa:b",
                       "1.00", "Some utility");
    .
    .
    char const *val = prog->option ('a');
    if (val) {
```

```
        cout << "Value of flag a is " << val << '\n';
        prog->msg ("Value of flag a is %s\n", val);
    }
    .
    .
    prog->error ("some failure, aborting\n");
}
```

Parsing program arguments Once the `argc/argv` construction is known to `eProgram`, two accessor functions return the arguments of the program:

- `int nargs ()`: returns the number of arguments
- `char const *arg (int i)`: returns the *i*-th argument

These functions will obviously only work after using the constructor that has the `argc` and `argv` as arguments, or after calling `cmdline()`.

Controlling the program verbosity The 'program verbosity' is controlled in two ways:

- When the command line parsing yields a `-v` flag, then the verbosity is turned 'on'. The default is off.
- When `msg(1)` is called, then the verbosity is turned on. As a side effect, this function returns the previous verbosity level, so that verbosity may be turned on temporarily.

Once the verbosity is turned 'on', then `msg(...)` calls can be used to generate messages. The messages appear in the format `programname: message`. The program name is set via the constructors. The messages appear on `stderr`.

By default messages will 'wrap' before exceeding column 79. To disable wrapping, use something like:

```
prog.wrapmsg (0);
prog.msg ("%s\n", verylongstring);
```

3.6 eStorable - base class for storable objects

3.6.1 Description

This class is a base class for generic storable objects. In order to become a storable class, a class must:

- Be derived from `eStorable`, as in:

```
class MyClass: public eStorable { .... } ;
```

- Define a function

```
eStorable *clone () const;
```

A typical cloning function is the following:

```
eStorable *MyClass::clone () const {  
    MyClass *ret = new MyClass (*this);  
    return ( (eStorable *) ret );  
}
```

This of course assumes that the right constructors are set up for MyClass (but that's always a good idea ;-).

3.7 eStr - String class

3.7.1 Synopsis

```
#include "e-str.h"  
#include "streams"  
  
// Constructors: empty string, initializers by string,  
// or character  
eStr s1,  
    s2 ("Hello World"),  
    s3 (s2),  
    s4 ('a');  
  
// Operators: assignment and concatenation with  
// strings or characters; *eStr also allowed  
s1 = "Hello there."  
s1 += " How are you";  
s1 += '?';  
s1 += s2;  
  
// Comparisons: < > <= >=  
if (s1 < s2)  
    cout << "s1 is smallest\n";  
  
// stream support  
cout << s << '\n';  
  
// Accessors: GET and SET, SET is also available as the  
// assignment operator
```

```
cout << s1.s () << '\n';
s1.s ("Hello again.");

// Other functions: length, (v)sprintf-like
cout << s1.len () << '\n';
s1.sprintf ("%d", 3 + 5);
s1.vsprintf (fmt, args);

// Comparisons: with or without regard to casing.
// With regard to casing is also available as < <= etc.
if (s1.compare (s2, 0) < 0)
    cout << "s1 is maller than s2 "
        << "with regard to case\n";
if (s1.compare (s2, 1))
    cout << "s1 differs from s2 "
        << "without regard to case\n";

// Fill string with 1 line from a file, remove leading
// and trailing whitespaces
FILE *f = fopen ("testfile", "r");
while (s4.fget (f)) {
    s4.strip ();
    cout << s4.s ();
}

// Conversions to lowercase, uppercase,
// or first character to upper case
s1.lowercase ();
s1.uppercase ();
s1.firstup ();

// Regular expression matching
if (s2.match ("[a-z]+")) {
    ...
}
```

3.7.2 Description

This class provides generic string handling. It is also an `eStorable` object, so that `eStr`-s can be added to e.g. `eArray`-s.

The methods are:

Constructors Construction is possible to create an empty string, or an initialized string. Initialization with one character is also allowed, this creates a one-character string.

Accessors The accessor `s ()` returns the contained string. To set the string, use the assignment operator discussed below, or provide a string-argument to `s ()`, as in `s ("Hello World")`. In both cases the contained `ascii-Z` string is returned.

Operators The plus-operator provides string concatenation (with an other `eStr`, or an `eStr*`, or a string, or a character). The assignment operator assigns the string. The usual comparison operators compare two strings (though without regard to casing).

Other functions `sprintf()` and `vsprintf()` The function `sprintf(formatspec, ...)` assigns the contained string according to the format specifier and remaining arguments. There is also `vsprintf(fmt, args)`.

`compare()` The member `compare()` compares the objects string with an other `eStr`, and returns -1, 0 or +1 according to the result. The second argument of `compare()` is an optional `int`, and when non-zero, indicates that comparison should be carried out without regard to casing.

`fget()` The member `fget(FILE*)` fills the contained string with the next line from the opened file. The line may be arbitrarily long.

`uppercase()` and family The functions `uppercase()`, `lowercase()` and `firstup()` modify the contained strings casing.

`match()` This member returns !0 when the contained string matches a regular expression. The return value is 0 indicates a non-match, or a bad regular expression.

`strip()` This function removes leading and trailing whitespace from the string.

3.8 eStrArr - Storage of eStr in array format

3.8.1 Synopsis

```
#include "e-strarr.h"

eStrArr mystore;
eStr    element,
        *ptr = &element;
int     max;

mystore.add (element);           // put an eStr in storage
mystore.add (ptr);              // or by pointer
mystore.add ("Hello World!\n"); // or a string

mystore.push (element);        // add to top of storage
mystore.unshift (element);    // add to bottom of storage
mystore.push ("Hello World");  // or use literal strings
mystore.push (ptr);            // or by ptr

max = mystore.n ();            // # of stored elements

eStr *p;
p = mystore.get (0);           // get handle on first element
p = mystore.pop ();            // retrieve and remove
                                //      from top
```

```
p = mystore.shift ();           // retrieve and remove
                                //      from bottom

mystore.sort ();               // sort alphabetically
mystore.sort (1);             // without respect to casing
mystore.sort (0, 1);          // with respect to casing,
                                // but reverse
mystore.sort (1, 1);          // without respect to casing,
                                // reverse

mystore.empty();              // reset storage
```

3.8.2 Description

`eStrArr` is based on `eArray` (see there) and just wraps `eArray`'s functions to accept and to return the right types. `eStrArr` doesn't provide any new functions, but makes storing and retrieving `eStr` objects a lot easier.

The method `sort()` takes two (default) 'boolean' arguments: whether to sort with or without respect to casing, and whether to sort in reverse order. The default value for both is 'false', i.e., without arguments, the sorting is

- with respect to casing,
- in normal order, not reverse.

3.9 eSysinfo - System information

3.9.1 Synopsis

```
#include "e-sysinfo.h"

eSysinfo info;                // determine system info
cout << info.desc ();         // show it
```

3.9.2 Description

The class `eSysinfo` collects system information and makes it accessible via the `desc()` method. The contained information is basic info, such as the host name, CPU, etc..

Chapter 4

Changelog

The following listing is a verbatim inclusion of the ChangeLog. It is included for reference purposes.

ChangeLog for the e-tunity utility library

- 4.38 [KK 2007-03-07] - Implemented ps_sortsnapshot()

- 4.37 [KK 2006-07-19] - Bugfix in ps_makesnapshot (related to Linux kernels 2.6+)
 - RPM creation & uploading disabled. This lib will have to be made from source.
 - Fix in rx_match(): when 2nd string argument is 0, RX_NOMATCH is returned.

- 4.36 [KK 2006-05-15] - Installation defaults to /usr/local/* when \$EBASE isn't set.
[KK 2006-06-02] - Small update regarding Cygwin portability.

- 4.35 [KK 2005-10-27] - A topline 'make install' now hits getline's 'make install' too.
 - A 'make install' in the C branch forces a 'make local' in a better way.

- 4.34 [KK 2005-09-28] - Added c-conf

- 4.33 [KK 2005-09-22] - Port to OSX, 'configure' utility added.
Dynamic build on Darwin implemented.

- 4.32 [RE 2005-07-14] - Added ­ to e-mlcode.txt
[KK 2005-07-11] - Added c/Makefile for e-lib onboarding in other projects. Removed fcgi based compile.
Revamped make process.

- [KK 2005-07-11] - Added c/Makefile for e-lib onboarding in other projects.
- [RE 2005-07-06] - Fixed compile errors in checksumadler32.c and fchecksumadler32.c
- 4.31 [KK 2005-06-21] - Added make_socket(), checksum_adler(), fchecksum_adler()
- 4.30 [KK 2005-05-10] - Rewired daemonize() to use better IPC. Improved docs.
- 4.29 [KK 2005-04-07] - Small documentation (manppage) fixes.
[KK 2005-05-06] daemonize() added.
- 4.28 [KK 2005-04-05] - getline() renamed to gl_getline(), to avoid conflicts with a new function of stdio.h
- 4.27 [RE 2005-02-04] - bugfix in logsetup.c
- 4.26 [KK 2005-01-06] - sha1() implemented
- 4.25 [KK 2004-12-20] - Library flag -lm added when testing C libs
- 4.24 [KK 2004-07-07] - tmp_fopen() and tmp_file() are more robust. When the compiletime symbol ETMP is missing, then "/tmp" gets substituted.
 - Minor documentation fixes.
 - "make installdoc" fixed; manpages for the separate C functions would sometimes not be installed.
- 4.23 [KK 2004-06-23] - random_ulong() implemented
- 4.22 [KK 2004-06-21] - rx_find() added
- 4.21 [KK 2004-06-21] - Renamed mm_*() to mmsg_*() to avoid name collisions with a memory allocation lib, that also holds mm_destroy().
- 4.20 [KK 2004-06-15] - Makefile.help added, RPM construction improved
 - mm_*() functions added for mail message parsing
 - Make target 'documentation' created.
 - Make target 'test' now uses local libraries. Previously, a 'make install' was necessary.
- 4.19 [KK 2004-05-25] - Added RPM build.

- Installed manpages are now compressed (also those of getline).
 - Dropped version ID in installed library name, RPM installs will handle that.
- 4.18 [KK 2004-04-27] - sf_sleep() added
- 4.17 [KK 2004-04-19] - Documentation updated (text on environment variables during installation added).
- Docs on some C functions (rc_read and so on) updated.
- 4.16 [KK 2003-12-16] - Fixed /dev/syslog/*/* logging via logmsg(). Services local0 thru local7 were handled incorrectly.
- 4.15 [KK 2003-11-18] - ' encoding added to html_(un)escape()
- 4.14 [KK 2003-11-14] - ml_encode() / ml_decode() have a wider range
- html_escape() / html_unescape() now support numeric codings as in {
- 4.13 [KK 2003-10-31] - get_log_setup() function implemented.
- 4.12 [KK 2003-10-09] - md5() function implemented.
- 4.11 [KK 2003-10-01] - /dev/syslog/FACILITY/LEVEL implemented for log_setup() and family.
- 4.10 [KK 2003-09-09] - Small code changes in C/C++ sources to make GCC 3.3 cleanly compile.
- C++ libs don't get built any more for FastCGI support. The C libs still do though, that's still being used.
 - logmsg() can now be called with 0 as the format string, in that case, only log rotations will occur.
- 4.09 [KK 2003-06-17] - Bugfix in fgetline(). The old function wouldn't properly handle extremely long lines that were not \n-terminated (e.g., a very long last line of a file).
- Bugfix in elib_require().
- 4.08 [KK 2003-06-16] - base64_encode() and base64_decode() rebuilt, implemented and documented
- Top level Makefile changed: "make install" will now just make the libs, a separate "make install_doc" is needed to install the manpages
- 4.07 [HK 2003-05-13] - encode_base64 () added

- decode_base64 () added
- 4.06 [KK 2003-04-07] - ps_makesnapshot() and ps_freesnapshot() added:
/proc-based process table analyzers
- 4.05 [KK 2003-01-30] - rc_setval() implemented.
- Small fix in logmsg().
- 4.04 [KK 2003-01-29] - Documentation update in rc_read().
- 4.03 [KK 2003-01-09] - Makefile slightly changed.
- rc_clean() implemented.
- btn_*() and bth_*() implemented. Finally, some generic binary tree / hash functions. Also hash() and hashbuf() are here.
- 4.02 [KK 2003-01-09] - Added sig_catch() to C library.
- 4.01 [KK 2003-01-02] - Added INCLUDE depth testing to rc_read(). This avoids endless nesting.
- 4.00 [KK 2002-12-04] - Conversion to shared object usage. The C functions are installed as libe.so (or libefcgi.so, for FastCGI), the C++ classes are installed as libepp.so (no FastCGI version available). The libs are installed with a version suffix, symlinks point to the right version.
- Source tree restructured.
- elib_version() and elib_require() implemented.
- Test runs implemented. LD_LIBRARY_PATH gets set during these runs, incase /etc/ld.so.conf isn't active yet.
- Docs updated.
- 3.22 [KK 2002-11-27] - cmm_blocksize() implemented.
- Default tmpfile prefix changed to /tmp/e; this was: /tmp/e- (note the hyphen).
- 3.21 [KK 2002-11-26] Makefile changed. The FastCGI versions are only made when FastCGI is available on the system.
- 3.20 [KK 2002-11-20] - copyfile(): return value CP_DST_IS_NO_FILE implemented
- movefile(): implemented
- 3.19 [KK 2002-11-19] - Zipping taken out of log_rotate(). We would probably prefer more allocation of disk space, than load on the CPU. (Previously the history files would be gzipped.)

- Some embedded documentation in C sources fixed.
- 3.18 [KK 2002-11-18] - Environment checks implemented, the current environment is checked during a make.
 - tmp_fileprefix(): a / is appended to the prefix if the argument name is a directory.
 - tmp_fopen() implemented
- 3.17 [KK 2002-11-11] - Bugfix in filetype(): fclose() was missing...
- 3.16 [KK 2002-10-29] - Separate function log_rotate() implemented. logmsg() now rotates via this one.
- 3.15 [KK 2002-10-28] - rc_read() expanded: INCLUDE directive added, environment variable RC_NAME implemented to overrule any filename to source
 - All-in-one manpage e-lib(3) added.
- 3.14 [RE 2002-10-23] e_zip_threshold renamed to zip_threshold.
- 3.13 [KK 2002-10-23] "make upload" now also sends the version ID and the ChangeLog.
- 3.12 [KK 2002-10-22] Updated docs for zip_threshold(). Makefile changed: "make upload" added, conforming to the public area of e-tunity.com.
- 3.11 [KK 2002-10-11] "make clean" now removes stale document results
- 3.10 [KK 2002-10-07] lap_start() and family now also log the duration (elapsed time), not only CPU ticks
- 3.09 [KK 2002-05-31] copyright() modified to show the right years
- 3.08 [KK 2002-05-27] Documentation: manpages are now named just like the contained C function or the contained class.
- 3.07 [RE 2002-05-21] Documentation changed to man format.
- 3.06 [KK 2002-05-16] More accessors for eStr made (using an eStr* argument).
- 3.05 [KK 2002-05-10] eStrArr changed again: (eStr const *) arguments also supported.
- 3.04 [KK 2002-05-08] eStrArr class changed: push(), add(), unshift() all support (char const *) arguments now
- 3.03 [KK 2002-05-01] Minor bugfix in the make process.

3.02 [KK 2002-04-25] The "make install_lib" doesn't invoke e-libman anymore to install PMMP's. This is now done during "make documentation".

3.01 [KK 2002-04-16] Added eStrArr::sort(), eStr::strip()
Bugfix in the wrapping during eProgram::msg().

3.00 [KK 2002-02-01] Merged e-lib (C-style) and epp-lib (C++ style) into new e-lib. Both are in two tastes: plain and FCGI.

Original C++-style epp-lib ChangeLog:

1.10 [KK 2002-01-23] eHttpRequest::empty() implemented

1.09 [KK 2002-01-21] eHttpRequest now also handles HTTP/1.0 type requests with "close" connection. This is used by e.g. htdig.

1.08 [KK 2002-01-17] Classes eHttpHdr and eHttpRequest added.
"make clean" cannot really clean up, but touches all cc and h files, so that a subsequent "make" is forced to recompile.

1.07 [KK 2002-01-15] eProgram::cmdline(char *switches, char *cmd) implemented

1.06 [KK 2002-01-14] eStrArr enhanced.

1.05 [KK 2002-01-02] ostream support for eStr added

1.04 [KK 2001-12-27] in eStrArr: proper includes added, const added for the "get" accessor, as in eArray.

1.03 [KK 2001-12-20] in eProgram: wrapmsg() implemented, and wrapping mechanism for msg() calls

1.02 [KK 2001-12-18] in eProgram::cmdline(): options that are specified more than once result in an error.

1.01 [KK 2001-12-17] eStrArr added

1.00 [KK 2001-12-01] First version.

Original C-style e-lib ChangeLog:

2.02 [KK 2002-01-23] Make target "install_lib" is new. The "install" also makes and installs the HTML docs.

- 2.01 [KK 2002-01-23] `html_unescape()` implemented
- 2.00 [KK 2002-01-22] Public release. Also:
 - HTML generation revisited
 - `tmp_file()`: compilation symbol `ETMP` gets `#defined` if it wasn't so already
- 1.44 - `cmm_setlevel(-1)` works again, that code somehow fell over. The internal `CMM_ELIB_LEVEL` is now `-2`.
- 1.43 - Check for positive size implemented in `cmm_malloc()` and `cmm_realloc()`.
 - Implemented: `log_wrap()`.
- 1.42 - C++ wrapping of all C-functions implemented.
 - e-mail adress of `copyright()` changed to `e-tunity.com`, years updated
- 1.41 - `copyfile()` changed: last 2 arguments can be `NULL` pointers, in which case the counts of read/written bytes are not updated.
 - Bugfix in `tmp_file()`: the temporary file prefix (directory) is now stored under `CMM` level `CMM_ELIB_LEVEL`. This makes `tmp_file()` c.s. re-entrant for programs that do garbage collection.
- 1.40 `rc_*()` functions now expand variables, in the format `$(var)`. This is done right after a file has been scanned in `rc_read()`.
- 1.39 "Make check" now disregards `*zip*.c` from checking. This is because the headers under `minilzo/` don't grok in `e-lint` (unsupported typedef syntax).
- 1.38 Message generation through `msg()` always results in flushing `stderr`.
- 1.37 `ml_encode()` and `ml_decode()` added
- 1.36 Bugfix in `zip_decompress()`. The compression/decompression would fail when the exact same buffer lengths would be reached.
- 1.35 `stricmp()` rewired. Previously the compared buffers would be allocated in memory, this is not necessary.
- 1.34 Added recognition for XML filetype (`.xml`)
- 1.33 Bugfix on 1.32.
- 1.32 Persistent `CMM` level for "e-lib internal usage" implemented. This level is used in: `log_setup()`, `set_progname()`, `tmp_fileprefix()`, `fin_push()` `fin_read()`, `lap_start()`, `rc_read()`, `tmp_file()`,

- 1.31 - Cosmetic change in `cmm_realloc()`.
 - Both normal and FCGI libs are created now.

- 1.30 Gigantic speedup in `cmm*()` achieved by using a hash table that accesses linked lists of memory blocks.

- 1.29 - FCGI build is now optional, via a macro in the Makefile.
 - Bugfixes in `cmm*()`.

- 1.28 - Global macros `C_DEBUGGING`, `C_OPTIMIZING` etc. used in the Makefile's `CFLAGS` settings for better portability
 - Installation of the docs to `$ELIBMANDIR` first removes old `*.pmp`'s. That ensures that old functions's docs are removed.
 - `dp_*()` functions removed, since the datapool for say Dantras is now in MySQL. Besides, these functions don't belong in e-lib anyway.
 - `cmm*()` functions implemented.

- 1.27- `ANSIDATE` and `ANSITIME` macros in the Makefile changed for better system independence.
 - MySQL functions removed from e-lib (`meta*()`).

- 1.26 In `image_size()`: the JPEG sizing now looks for the right frame marker `0xc0` (previously: `0xc*`); this means that we need the FIRST frame to determine the image size. This supports, e.g., HyperSnap jpegs.

- 1.25 `log_dontflush()` added

- 1.24 Double build implemented: once a single version, once a FCGI-enabled library construction.

- 1.23 Filetypes `.doc` (Office document) and `.rtf` (Rich Text format) added.

- 1.22 - `supply_ext()` added
 - `suggest_ext()` added

- 1.21 - `tmp_fileprefix()` added
 - script "makehtml" enhanced

- 1.20 - `copyright()` notice changed: build-information now includes username and hostname and build timestamp
 - `cgi_setpostfile()` added

- 1.19 - `skipline()` function added
 - multipart CGI processing added

- 1.18 `equal()` function renamed to `str_equal()`, and the semantics have somewhat changed. The function now returns the number of matched chars. To avoid problems, the old name no longer exists in the

library.

1.17 zip_*() functions and subdirectory minilzo/ added.

1.16 Makefile adapted to allow "make check".

1.15 Added: stab_popstr(): pops a string off a string table stack

1.14 Checks for `__E_LINT__` added in e-lib.h. When detected, `__GNUC__` gets undefined so that GNU extensions are suppressed in the included headers.

1.13 copyright_setoutput() added

1.12 stab_addsprintf() added

1.11 "Out of memory" handling added: xstrdup(), xmalloc(), xrealloc(), with the auxiliary functions out_of_memory() and set_memerr().

1.10 stab_maxlen() added

1.09 Make html calls "e-libman -1" via the script makehtml. The pages are now cleaner..

1.08 - Makefile enhanced.

- "make html" added as option for the make process; shell script "makehtml" will produce the e-libman-like pages in a subdir html.

1.07 - Installation of *.pmmp now creates a TimeStamp file.

- Function cgi_getdata() "un-escapes" both the CGI variable name and its value.

1.06 Textual information of some functions adapted (for e-libman).

1.05 Making order changed. The installation of pmmp files (poor-man's man pages) occurs now during the install phase. NOTE THAT e-libman is needed for that... so the "make install" in the e-libman directory must occur before the "make install" here. The top-level Makefile should take care of that.

1.04 msg_and_log() added

1.03 checksum() added

1.02 copyfile() added

1.01 log_setup() overrules its arguments via the environment ELOGFILE etc., see "e-libman log_setup"

1.00 First issued version number. Versioning is now implemented also in the e-lib and displayed via `copyright()`. Previous e-lib versions do not use an internal versioning.