Documentation for string2.h and string2.c

Steven Andrews, © 2003-2023

This library complements the standard string library with several useful functions.

Requires: `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<ctype.h>`

Example programs: `SpectFit.c`, `LibTest.c`, `Smoldyn`

History

1/99	Started writing
11/01	Transferred code from Utility.c library to newly created string2.c library.
11/01	Added array reading and `strnword` and transferred to Linux
1/24/02	Added `wordcount`
3/29/02	Added `StrrChrPQuote`
10/29/02	Added `StrChrPQuote`
1/19/03	Added `stringfind`
6/11/03	Added `strrpbrk`
1/16/04	Added `strreadns`
6/9/04	Renamed `isnumber` to `strisnumber` to avoid a name collision with some other command
2/13/06	Added `strchrreplace`
9/21/07	Added `strbslash2escseq`
10/28/07	Added `symbolcount`
11/16/07	Added `strcutwhite`
6/2/08	Added `strstrreplace`
10/28/09	Added `strstrbegin`
1/11/12	Added `strwildcardmatch`
1/30/12	Added `strparenmatch`
4/17/12	Added `STRCHARLONG`
1/16/14	Added `strchrindex`
10/6/15	Added `strreadnli`
1/25/16	Added `strwhichword`, `strnwordc`, `strmathsscanf`
2/5/16	Added `strmatchevalint`
2/29/16	Added `wordcountpbrk`, replaced `strexpandlogic`, edited wildcard functions
8/17/16	Added `strwordcpy`
10/28/16	Fixed a bug in `strmathsscanf`
2/16/17	Fixed a bug in `strmathsscanf`
2/17/17	Added `strisfunctionform`, `strevalfunction`, and `strloadmathfunctions`
5/24/17	Fixed minor bug in `strloadmathfunctions`
3/15/19	Added support for scientific notation in `strmatheval`
12/2/22	Added `str1wordcpy`
2/2/23	Fixed bug in `strmatheval` and added const qualifiers to math functions
8/15/23	Added `EmptyStringLong` function, increased `STRCHAR` to 512, and increased `STRCHARLONG` to 4096
3/28/24	Added `strunits` and `strfirstwordpbrk` functions. Edited `strmathsscanf` to accept and process units.

Definitions

```
#define STRCHAR 512
```

This is defined because it is often easiest for all strings to have the same length. That way concatenations and other manipulations are fairly easy. Of course, it doesn't have to be used.

```
#define STRCHARLONG 4096
```

A longer standard string length.

String classification

```
int strisnumber(const char *str);
```

Returns 1 if the string is a number and 0 if it isn't. Any type of number that is recognized by `strtod` (stdlib library) is recognized here as a number.

```
int okname(const char *name);
```

Returns 1 if the input string is valid as a variable name. The rule is that the first character must be a letter and subsequent characters may be letters, numbers, or an underscore. The length of the string is not considered.

```
int strhasname(const char *string,const char *name);
```

Tests to see if the name is within string as an isolated substring, meaning that the characters before and after the substring cannot be alphanumeric or an underscore. For example, the name "abc" is isolated within the string "53(abc+6)^xy" but not within "abcde".

```
int strbegin(const char *strshort,const char *strlong,int casesensitive);
```

Returns 1 if `strlong` begins with or is equal to `strshort` and returns 0 otherwise. Set `casesensitive` to 1 if the check should be case sensitive and 0 if not. A typical example is `strbegin(input,"yes",0);`, which returns 1 if `input` is "y", "Y", "yes", "YES", etc. and returns 0 for other things. This returns 0 if `strshort` is the empty string.

```
int symbolcount(const char *s,char c);
```

Counts and returns the number of times that character `c` occurs in string `s`.

```
int strisfunctionform(const char *str,const char **parenptr);
```

Determines if string `str` has the form of a function, meaning that it starts with a letter, then has alphanumeric characters or underscores, then has an open parenthesis, then has more stuff, and is ended by a close parenthesis. Example: "`myfunc_1(...)`". Returns 1 if so and 0 if not. Also returns a pointer to the open parenthesis in `parenptr` if that isn't `NULL`.

Character locating

```
int strchrindex(const char *cs,char c,int i);
```

This is similar to the standard library `strchr` function. This returns the index of the first occurrence of `c` in `cs`, starting from character number `i`. Enter `i` as 0 to start at the beginning of `cs`. It returns -1 if `c` is not found.

```
char *strrpbrk(char *cs,const char *ct);
```

Returns a pointer to the last occurrence in string `cs` of any character of string `ct`, or `NULL` if not present. It is identical to the standard library function `strpbrk`, except that it is for the last rather than first occurrence.

```
char *StrChrQuote(char *cs,char c);
```

`StrChrQuote` is just like the `strchr` function in the ANSII string.h library, in that it returns a pointer to the first occurrence of `c` in `cs`, or `NULL` if not present. However, it ignores any `c` characters after an odd number of `"` marks (i.e. within double quotes).

```
int StrChrPQuote(const char *cs,char c);
```

`StrChrPQuote` is similar to `StrChrQuote`. It looks for the first occurrence of `c` is `cs`, returning its index if found. It ignores any `c` characters in double quotes or inside parentheses. Any level of parenthesis nesting is permitted. If mismatched parentheses are encountered before a valid `c` is found, –2 is returned; if quotes are mismatched, –3 is returned; if no `c` was found, –1 is returned. It is impossible to search for a quote symbol, and the method of preceding a quote with a backslash to make it a symbol rather than a quote, is not supported.

```
int StrrChrPQuote(const char *cs,char c);
```

Like `StrChrPQuote`, except that it returns the last occurrence of `c`.

```
int strChrBrackets(const char *string,int n,char c,const char *delimit);
```

This is a fairly general version of the above specialized `strchr`-like functions. Looks for and returns the index of the first occurrence of `c` in string `string`, where `c` is outside of parentheses, brackets, quotes, etc. Choose which of these delimiters are wanted by listing the opening elements in `delimit`; the options are: `(` `[` `{` `"` and `'`. This ignores any delimiters that are not listed. For those that are listed, this checks to make sure that each opening item is matched with a closing item; the function returns -2 for mismatched parentheses, -3 for mismatched brackets, and -4 for mismatched braces. This does not check syntax between different types of delimiters; for example, the string “a(b[c)d]e” is valid here. Returns -1 if no `c` is found outside of listed delimiters. Enter `n` as the string length (which enables it to be set to a smaller value than the total string length) or as -1 if the total string length should be measured and used.

```
int strPbrkBrackets(const char *string,int n,const char *c,const char
    *delimit,int imin);
```

This is identical to strChrBrackets, but it looks for the first occurrence of any character within the string `c`. Also, `imin` is the first index where this will return a positive result.

```
int strparenmatch(const char *string,int index);
```

Finds the index of the matching parenthesis to the one that is indexed with `index`. This supports parenthesis, brackets, and braces, i.e. (), [], and {}. If `index` points to an opening object, then this looks forward for the matching closing object, ignoring nested ones. Similarly, if `index` points to a closing object, then this looks backwards in the string for the matching opening object, again ignoring nested ones. Returns the index of the match, or -1 if `index` doesn’t point to a supported object, or -2 if a match was not found.

```
int strsymbolmatch(char *str1,char *str2,const char *symbols);
```

Compares the sequence of single-character symbols, that are defined in `symbols`, in `str1` with those in `str2` and returns 1 if they are the same or 0 if they are not. Only symbol sequence is considered, not symbol positions.

For example, suppose symbols equals “{}&|”; then, “ab{c}d&e” matches to “{xy}&w” but not to “a&b{c}”, the reason being that the ampersand is in the wrong place relative to the braces in the latter situation. `str1` and `str2` are not modified here, but they are not allowed to be labeled with a `const` because the function uses pointers within them.

Word operations

`int wordcount(const char *s);`

Counts and returns the number of words in a string, where a word is defined as a contiguous collection of non-whitespace characters.

`int wordcountpbrk(const char *s,const char *symbols);`

Identical to `wordcount`, but any of the characters listed in `symbols` can serve as a word separation character. This function has not been tested.

`int strwhichword(const char *s,const char *end);`

Returns the word number within `s` where the pointer labeled `end` is in. This is essentially identical to `wordcount`, but it stops at `end`. For example, if the string is “this is a string” and `end` points to the ‘a’ character, then this returns 3, for the 3rd word. If `end` is not within the string (or is `NULL`), then this returns the total number of words in the string. If `end` points to a whitespace character, then this returns the number of preceding words. This treats multiple sequential whitespace characters as a single whitespace.

`char *strnword(char *s,int n);`

Returns a pointer to the `n`’th word in `s`, where a word is defined as any collection of non-whitespace characters. It returns `NULL` if there are less than `n` words in the string, and `s` if either `n` is 0 or if `n` is 1 and the first word starts at the left edge of `s`.

`const char *strnwordc(const char *s,int n);`

Identical to `strnword`, except that strings are defined as `const char*` instead of `char*`.

`char *strnword1(char *s,int n);`

Similar to `strnword`, except that it counts words based on the first word starting at the beginning of the string and each subsequent word separated by a single space or tab from the preceding one (other whitespace characters are considered to be part of the word). Thus, a double space implies the existence of an empty word between the spaces. If there is no `n`’th word, either because it is empty or because the string has less than `n` words, the routine returns `NULL`.

`char *strnwordend(const char *s,int n);`

Like `strnword`, but returns a pointer to one character past the the end of the `n`’th word. If the last character of the string is the end of the `n`’th word, then this returns a pointer to the terminating ‘`\0`’. If the string has fewer than `n` words, then this returns `NULL`.

`char *strwordcpy(char *destination,const char *source,int n);`

Copies up to the end of the `nth` word of the source string to the destination string, terminating the destination string with ‘`\0`’. Returns the destination string. The destination needs to be long enough to hold the result.

```
char *str1wordcpy(char *destination,const char *source,int n);
```

Copies only the `nth` word of the source string to the destination string, leaving out any whitespace, and terminating the destination string with '`\0`'. Multiple spaces do not count as additional words. Returns the destination string. The destination needs to be long enough to hold the result.

```
int strfirstwordpbrk(char *destination,const char *source,const char *symbols);
```

Copies the first word of source into destination, where the word is defined to end one character before any of the characters listed in symbol, or one character before a whitespace character. Returns the index of the character after this word ends. Example: `strfirstwordpbrk(dest,"abc123","012")` copies "abc" to `dest` and returns value 3, which is the index of where the "1" is.

String arrays

```
int stringfind(const char **slist,int n,const char *s);
```

Locates string `s` in an array of strings called `slist`, which extends from index 0 to `n-1`. If an exact match for `s` is found, its index is returned; otherwise `-1` is returned. `n` may be 0 or negative.

```
int stringnfind(const char **slist,int n,const char *s,int nchar);
```

Identical to `stringfind`, but only uses the first nchar characters of `s`.

Reading sequential items from strings

```
int strreadni(char *s,int n,int *a,char **endp);
```

Reads up to the first `n` integers from the string `s`, delimited by white space; leading white space and multiple spaces between integers are ignored. Results are put in the integer vector `a`, which is assumed to be allocated to be sufficiently large. The function returns the number of integers parsed. Any unconverted suffix is pointed to by `*endp`, unless `endp` is `NULL`.

```
int strreadnli(char *s,int n,long int *a,char **endp);
```

Identical to `strreadni`, except that it reads long ints rather than integers.

```
int strreadnf(char *s,int n,float *a,char **endp);
```

Identical to `strreadni`, except that it reads floats rather than integers.

```
int strreadnd(char *s,int n,double *a,char **endp);
```

Identical to `strreadnf`, except that it reads doubles rather than floats.

```
int strreadns(char *s,int n,char **a,char **endp);
```

Identical to `strreadni`, except that it reads words rather than integers. It is assumed that each string in the list of strings `a` has already been allocated to be sufficiently large to hold the respective word, as well as a terminating `'\0'`. Any strings in `a` in addition to those that were parsed are not modified.

String copying with memory allocation

`char *EmptyString();`

Returns a blank string of `STRCHAR` characters, all initialized to `'\0'`.

`char *EmptyStringLong(int size);`

Returns a blank string of `size` characters, all initialized to `'\0'`.

`char *StringCopy(const char *s);`

Takes in a string and returns a copy of it. Exactly enough memory is allocated for the copy to contain the entire string; it returns `NULL` if memory allocation failed. This memory should be freed when it is no longer being used with the stdlib `free` function.

`unsigned char *PascalString(const char *s);`

Identical to `StringCopy`, except that it returns a pascal type string. The first character is the number of letters in the string. A previous implementation (pre-11/01) added a terminating `'\0'` as well, as with C type strings; this character is no longer added.

String modifying without memory allocation

`char *strPreCat(char *str,const char *cat,int start,int stop);`

Concatenates string `cat`, from the character at index `start` to the character at index `stop-1`, to the beginning of string `str`. No check is made for memory overflow.

`char *strPostCat(char *str,const char *cat,int start,int stop);`

Concatenates string `cat`, from the character at index `start` to the character at index `stop-1`, to the end of string `str`. No check is made for memory overflow.

`char *strMidCat(char *str,int s1,int s2,const char *cat,int start,int stop);`

Concatenates string `cat`, from the character at index `start` to the character at index `stop-1`, into the middle of string `str`, starting at index `s1` and going to index `s2-1`. This replaces the characters from `s1` to `s2-1`. If `s1` and `s2` equal each other, then no characters in `str` are replaced and the inserted text will start at character `s1`. Enter `stop` as -1 to copy to the end of `cat`. Enter `cat` as `NULL` and `start` and `stop` to 0 to do no copying at all but just delete some characters from `str`.

`int strchrreplace(char *str,char charfrom,char charto);`

Searches string `str` and replaces all characters that are `charfrom` with `charto`. It returns the number of replacements that were made.

`int strstrreplace(char *str,const char *strfrom,const char *strto,int max);`

Searches string `str` and replaces all portions that match `strfrom` with `strto`. The number of replacements made is returned. Recursive substitutions are not performed. If `str` would exceed `max` characters because of replacements, the last characters are dropped and the negative of the number of replacements made is returned to indicate string overflow. `strto` may be `NULL`, in which case the `strfrom` strings are removed and nothing is put in their places.

`void strcutwhite(char *str,int end);`

Removes all white space (ctype `isspace` is 1) from an end of string `str`. If `end` is 1, this removes from the start of the string; if `end` is 2, this removes from the terminus; if `end` is 3, this removes from both ends. `str` must not be `NULL` and must have a length of at least 1.

`int strbslash2escseq(char *str);`

Replaces all backslash-letter sequences in string `str` with the proper escape sequences. For example, the two characters "\n" would be replaced with a single newline character. The escape sequences are:

\a	alert
\b	backspace
\t	tab
\n	newline
\v	vertical tab
\f	form feed
\r	carriage return
\\	backslash
\"	double quote

A backslash followed by any other character is left as a backslash. The function returns the number of replacements made.

Wildcards

`int strcharlistmatch(const char *pat,const char ch,int n);`

Determines if character `ch` matches a character or a character range listed in `pat`, starting at the first character of `pat` and continuing for `n` characters (i.e. up to and including the `n-1` character). Enter `n` as -1 for the entire `pat` string. The only patterns this recognizes are characters and hyphens. Any character that is not a hyphen is just another character here. For example: "abc" matches to any of 'a', 'b', and 'c'; "a-d" matches to any character within the range from 'a' to 'd', inclusive; "abf-jdq-s" matches to 'a', 'b', any character from 'f' to 'j', 'd', and any character from 'q' to 's'; "-k" matches to any character less than or equal to 'k'; "abe-" matches to 'a', 'b', and any character as large or larger than 'e'; "-" matches to all characters; "z-a" does not match to anything, including 'z' or 'a'. Behavior is not defined for two sequential hyphens.

This is a low level function, called only by `strwildcardmatch` and `strwildcardmatchandsub`. It handles the [ac-e] matching rule.

`int strwildcardmatch(const char *pat,const char *str);`

Determines if the string in `str` is a match for the string in `pat`, which might include wildcard characters and/or brackets, returning 1 if they do match and 0 if not. Wildcards are that '?' can represent any single character and '*' can represent any number of characters, including no characters. For example, m?s*sip* is a match for mississippi. Brackets are that any characters in brackets, or character ranges in brackets, can match to any single character. For example "[xya-c]" can match to 'x', 'y', or any character from 'a' to 'c', inclusive (see `strcharlistmatch` description). This function is case sensitive and it treats periods just like any other character. This function does not identify or return the represented text.

The chief challenge of this function is determining what text a star should represent. The answer is that it represents nothing at all at first. If that fails to produce a match, it is increased by one character. If that fails, it is increased by one more character, and so on. Eventually, if a match is possible, it will be found.

This function works by stepping through the two strings together, stopping early if the strings can’t match. If the strings are the same, or if `pat` has a ‘?’, then they match so far, and so stepping continues. If a star is found in `pat`, then `p1` and `p` are both put one character after the star and `s` is not incremented, to allow for 0-character star text. Then, just `s` walks forward until the two strings start matching again. Once they return to matching, both `s` and `p` walk forward; however, if the match then fails again, `p` is reset back to just after the star, where `p1` is still sitting, and `s` is put back to one place after where it started last time, marked by `s1`.

Brackets look complicated, but are best thought of as an extended version of the ‘?’ option. However, the ‘?’ option always matches, but the brackets don’t always match, so a failure for the brackets requires the code to go back into the if...else sequence. That’s not really possible, so it uses a goto statement instead.

This function handles the *, ?, and [...] wildcards. It does not recognize {...}, |, or &, which are in the enhanced wildcard matching functions. This function is only called by `strEnhWildcardMatch` and `strEnhWildcardMatchAndSub`.

`int strwildcardmatchandsub(const char *pat,const char *str,char *dest,int starextra);`

Determines if the string in `str` is a match for the string in `pat`, which might include wildcard characters, just like the function `strwildcardmatch`. This function also finds out what text in `str` is being represented by wildcards and substitutes that represented text into the corresponding wildcard characters in `dest`. For example, suppose `pat` is “`m?s*sip*`”, `str` is “`mississippi`”, and `dest` is “`AB*CD*EF?GH`”. This will return 1 to indicate that `str` matches `pat` and it will return `dest` as “`ABsisCDpiEFiGH`”. Not all represented text is substituted into `dest` if `dest` has fewer wildcard characters than `pat`, and not all wildcard characters in `dest` are replaced if `dest` has more wildcard characters than `pat`. If `str` and `pat` don’t match, `dest` might still be modified; if this isn’t desired, then check for a match with `strwildcardmatch` first (or just copy over `dest` first).

Enter `starextra` as 0 for basic operation, looking for the first match possible. Larger numbers imply that the text represented by the first star should not be kept as small as possible. This allows all possible matches, and not just the first one, to be gotten from a pattern string that has 2 stars in it. It does not support more than 2 stars.

Function operation is best understood by realizing that a ‘?’ wildcard character is resolved as soon as it is found, thus enabling immediate replacement into `dest`. On the other hand, a ‘*’ character is resolved when

(1) another star is reached in `pat`, (2) it is the terminal character in `pat`, or (3) the end of `pat` is reached.

Returns 1 if `str` is a match to `pat` and 0 if not. Also returns -6 if more substitutions are required than the 16 spaces that are statically allocated, -7 if brackets in `pat` are mismatched, -8 if brackets in `dest` are mismatched, -9 if one or more wildcards in `dest` do not correspond to those in `pat`, or -10 if the $*n* format is used and done incorrectly, meaning that the *n* value is missing or not between 1 and the number of substitutions, inclusive.

This function handles the *, ?, and [...] wildcards and also the $*n* substitution. It does not recognize {...}, |, or &, which are in the enhanced wildcard matching functions. This function is only called by `strEnhWildcardMatchAndSub`.

`int permutelex(int *seq,int n);`

This is an internal function used in `strexpandlogic`. It was copied verbatim from my Zn.c library simply to reduce library dependencies.

This computes the next permutation of the items listed in `seq`, of which there are `n` items, according to lexicographical ordering, and puts the result back in `seq`. Multiple items of `seq` are allowed to equal each other; if this happens, then these items are not permuted (e.g. if the starting sequence is 1,2,2, then subsequent sequences are 2,1,2, and 2,2,1, which is the end). This returns 1 when the final sequence is reached, 2 when the sequence wraps around to the start, and 0 otherwise. If the final sequence is sent in as an input, then, this reverses the sequence so as to start over again. This algorithm is from the web and is supposedly from Dijkstra, 1997, p. 71.

`int allocresults(char ***resultsptr,int *maxrptr,int nchar);`

This is an internal function for use by `strexpandlogic`.

It allocates the results list and expands the list as necessary. For initial use, send in `resultsptr` as a pointer that points to a `NULL`, `maxrptr` as a pointer to points to an integer that will get overwritten, and `nchar` as the desired string length; `resultsptr` will be returned pointing to an array of strings and `maxrptr` will be returned pointing to the number of strings in the array. Afterwards, call this whenever the array should be expanded, using the same pointers for `results` and `maxrptr` and the same `nchar` value. To free the data structure, call this with `nchar` equal to -1.

Returns 0 for success or 1 for inability to allocate memory.

`int strexpandlogic(const char *pat,int start,int stop,char ***resultsptr);`

This is primarily an internal function, for use by `strEnhWildcardMatch` and `strEnhWildcardMatchAndSub`. However, it is exposed in the header file in case it is useful elsewhere.

This function expands regular expression patterns, in `pat`, that include AND and OR operators, as well as braces to express order of operations. The AND operator is a permutation operator. Examples: “A|B|C” expands to “A”, "B", and "C"; "A&B&C" expands to "ABC", "ACB", "BAC", "BCA", "CAB", and "CBA"; and "A&{B|C}" expands to "AB", "BA", "AC", and "CA", where the braces state that this OR symbol should take precedence over the AND symbol. For normal use, enter `start` as 0, `stop` as -1, and `resultsptr` as a pointer to an unallocated `char**`. The number of answer will be returned directly and the specific answers will be returned in an array of strings that is pointed to by `results`. This results array and all of the strings in it will need to be freed. Both the results array and the strings in it get allocated here to be the minimum necessary size.

Returns the number of strings in the results list on success (which can be 0, in which case `resultsptr` points to `NULL`), or an error code: -1 for inability to allocate memory, -2 for missing space operand, -3 for missing & operand, or -5 for mismatched braces. Note that empty | operands are allowed.

symbol	function	example
space	word separator	A|B+C expands to A+C, B+C
|	OR	A|B|C expands to A, B, C
&	AND	A&B expands to AB, BA
{}	sequence	A&{B|C} expands to AB, AC, BA, CA

The input string `pat` is unchanged by this function. It is only investigated over the range from index `start` to index `stop-1`. Each portion of this function allocates exactly enough memory for its own results. When recursive behavior happens, each portion of this function frees the memory that was sent to it. To process the OR operator, the function expands the two sides, combines the results, and returns them. The AND operator is similar, but now the sides are combined in both sequences. Others are similar as well.

`int strEnhWildcardMatch(const char *pat,const char *str);`

Identical to `strwildcardmatch`, except that this accepts enhanced wildcard strings in pat. For example, "a?&b*" matches to "albatross" and to "borax", but not to "walrus". This function uses internal memory, which should be freed when it is done being used. Enter `pat` as `NULL` to free this internal stored memory.

Returns 1 for a match, 0 for a non-match, -1 for inability to allocate memory, -2 for missing space operand, -3 for missing & operand, or -5 for mismatched braces.

This function expands `pat` to create a list of simplified patterns to work with, which it stores internally; it reuses this list until a new `pat` is entered. For this reason, it is most efficient to use this function with a single `pat` for as long as is useful.

`int strEnhWildcardMatchAndSub(const char *pat,const char *str,const char *destpat,char *dest);`

Identical to `strEnhWildcardMatch`, except that this also substitutes the represented text into a destination string and returns it in the string `dest`, which needs to be pre-allocated to size `STRCHAR`. Also, if a match can be made in many different ways, the way to find out is to call this function over and over again with exactly the same inputs, until the function returns 0.

Returns 1 for a match, 0 for a non-match or finished with all matches or missing input string, -1 for inability to allocate memory, -2 for missing + operand, -3 for missing & operand, -5 for mismatched braces, or -10 for a destination pattern that is incompatible with the matching pattern (i.e. there needs to be 1 destination option, or 1 pattern option, or the same number of destination options as pattern options).

This function creates lists of simplified patterns to work with, which it stores internally; it reuses these lists until a new `pat` is entered. For this reason, it is most efficient to use this function with a single `pat` for as long as is useful.

Math parsing

`double dblnan();`

Internal function. Returns `NAN` as a double.

`int strloadmathfunctions(void);`

Loads a list of math functions to the `strevalfunction` function. Note that the `fnptrdd` and `fnptrddd` variables are used here because they enable C++ compilers to determine which one of the overloaded functions should be chosen.

`double strevalfunction(char *expression,char *parameters,void *voidptr,void *funcptr,char **varnames,const double *varvalues,int nvar);`

Evaluates a function that returns a number. This function is primarily designed for internal use (call `strmatheval` instead) but needs to be called from externally as well to store some functions and to free everything at the end.

First, this function needs to be prepared with a list of functions, such as sin, cos, sqrt, etc. Taking sin(*x*) as an example, call this function with `expression` equal to “`sin`”, with `parameters` equal to “`dd`” (the first ‘d’ is because the sine function returns a double and the second ‘d’ is because the argument of the sine function is one double), and with `funcptr` equal to `(void*) &sin`. The other inputs are ignored. The current options for the parameters input are “`dd`”, “`ddd`”, and “dves”. The ‘v’ type of input is a void* input (used for the simulation structure) and the ‘e’ type of input is for return of an error string. The contents of this void* are not sent to `strevalfunction` during the evaluation phase, but during the preparation phase. This value is stored along with the function address and is sent along whenever the particular function is called.

Later, to use this function (should only be used internally), send in `expression` with “`sin(2)`”, with `parameters` pointing to a string with the parameters in it, and with `funcptr` equal to `NULL`. The `parameters` string should not include the beginning and ending parentheses but should include commas to separate multiple parameters; it is not changed by this function. Also, send in the variable stuff so that this function can call `strmatheval` with any arguments that need evaluating.

Finally, call `strevalfunction` with `expression` equal to `NULL` to free the memory that it allocated.

```
double strmatheval(const char *expression,char **varnames,const double
    *varvalues,int nvar);
```

Evaluates the mathematical expression in the string `expression`, returning the answer as a double. This expression may include variables that are listed in the `varnames` list of strings and that have values in `varvalues` list, of which there are `nvar` total variables. Variable values and names are not modified here. Also, new variables cannot be created here. At present, this function supports addition (+), subtraction (-), multiplication (*), division (/), modulo division (%), powers (^), unary signs (+ and -), and all levels of parentheses, brackets, and braces. Also, if they have been loaded in with the `strloadmathfunctions` function, it supports all of the standard C math library functions, such as sin, cos, asin, tanh, exp, sqrt, fabs, floor, atan2, pow, rand, etc. On failure, this function returns `NAN`, sets the library-wide global variable `MathParseError` to 1, and sets an error message that can be retrieved using the `strmatherror` function. The expression variable is assumed to have less than `STRCHAR` characters and is not changed here.

```
int strmathevalint(const char *expression,char **varnames,const double
    *varvalues,int nvar);
```

Identical to `strmatheval`, except that it returns an integer instead of a double. This function performs all calculations as doubles (using `strmatheval`) and then rounds the result to the nearest integer.

```
int strmatherror(char *string,int clear);
```

Returns math parsing errors that can arise in `strmatheval`. This returns the error number directly (either 1 for error or 0 for no error) and a description of the error in `string`, which needs to be pre-allocated to at least `STRCHAR` characters. Set `clear` to 1 for any error to be cleared after this function is called and to 0 for any error to be left.

```
int strmathsscanf(char *str,char *format,char **varnames,const double
    *varvalues,int nvar,...);
```

A replacement for the normal `sscanf` function which evaluates math expressions. As with the normal `sscanf` function, send in `str` with a string to be scanned, `format` with a formatting string, and the destinations for the scanned information in the variable arguments, shown with the final ellipsis. All strings must have fewer than `STRCHAR` (256) characters. Also, if variables should be used in the math evaluation, send in those variable names in `varnames`, their values in `varvalues`, and the number of variables in `nvar`. To indicate that a math expression should be evaluated to a number, give its format specifier as “`%mlg`” for doubles and as “`%mi`” for integers. As with the sscanf function, this returns the number of successfully parsed arguments. If the math expression evaluates to an error, this will parse the up to the error and will then return a value that is fewer than the number of arguments. In this case, the error message can be retrieved from `strmatherror`.

If the format specifier ends with a pipe character and then additional characters, those are interpreted as unit specifiers. For example, the format string “%mlg|L” inputs a double that has units of length. The string that’s read can have units but does not need to; if not, then it is assumed to have units of the current input units. Either way, the value is converted to the value that’s in the units of the working units. It’s also possible for the unit specifier to be blank to show that the value is explicitly unitless (or should not have units, as is the case for variables).

This function transcribes the `str` and `format` strings to new ones, called `newstr` and `newformat`, evaluating any math functions in the process. Then, it reads the new strings using the `vsscanf` function.

`double strunits(const char *unitstring,const char *dimstring,double value,char *outstring,const char* function);`

This function manages units for floating point numbers in configuration files. It has its own internal storage, stored with static variables, and a long list of preprogrammed units; additional units cannot be added without editing the code. The basic use is that the `function` input determines what it does on each particular call. It is called initially with `function` equal to “initialize” to prepare its lists of units. Then, it is called with `function` equal to “push” to select a particular set of units that the configuration file will be using, which can be done repeatedly with multiple nested configuration files. At the end of each file, call with `function` equal to “pop” to return to the prior set of units. The first units that are entered are assumed to be the units that numbers should be converted to, called the working units, but if that’s not right, then enter `function` with “working” to specify the preferred output units. Enter `function` with “convert” to actually convert numbers between different units. Enter `function` with “getunits” to retrieve the current destination units. At the end, call with `function` set to “free” to free internal memory. This function sets errors in the `MathParseError` global variable and creates error strings in the `StrErrorString` global variable, both of which can be accessed through the `strmatherror` function.

`convert. unitstring=` unit or `NULL, dimstring=` dimensions or `NULL, value=` value to convert, `outstring=NULL`. Converts from input units to working units. Either `unitstring` or `dimstring` or both must be entered for unit conversion to occur; if neither is entered, then no unit conversion happens. If `unitstring` is given, this assumes that value is in those units. If `dimstring` is also given, this checks that the `unitstring` and `dimstring` are consistent with each other. If `unitstring` is not given, this uses the units that are at the top of the stack, which it chooses from using the dimensions in `dimstring`. Example: `unitstring` = “um”, `dimstring` = “L”, value=0.01; suppose the working unit for length is “nm”, then this returns 10. The `unitstring` and `dimstring` inputs can include compound units, such as “um/ms”, “um3/s”, etc.

`getunits. unitstring=NULL, dimstring=` dimensions or `NULL, value=0, outstring=` allocated string of length `STRCHAR`. Enter `dimstring` with some combination of dimensions, and this will return in `outstring` the same combination but with the working units; e.g. if `dimstring` is “L2/T” then `outstring` will equal “um2/s” if those are the working units. Or enter `dimstring` as NULL and this will write all working units to the `outstring`, such as “um s”. If there are no `workUnits`, this returns a value of 1 and an empty string in `outstring`; if there are `workUnits`, the returns a value of 0 and the result in `outstring`.

`initialize. unitstring=NULL, dimstring=NULL, value=0, outstring=NULL.` Initializes all internal data structures. This can be called multiple times, but there’s no reason to do so. Returns 0 for success and 1 for inability to allocate memory.

`free. unitstring=NULL, dimstring=NULL, value=0, outstring=NULL`. Frees all internal data structures and resets them to an uninitialized state. It's possible to start over again with `initialize` if desired. Returns 0.

`debug. unitstring=NULL, dimstring=NULL, value=0, outstring=NULL`. Prints the contents of all internal data structures to stdout. Returns 0.

`push. unitstring=` space-separated unit list, `dimstring=` filename, `value=0`, `outstring=NULL`. Pushes a list of units onto the unit stack. For example, enter `unitstring` as "um ms" for microns and milliseconds. Missing values are left as undefined, so there is no unit conversion for them; thus, the entire unit list needs to be pushed simultaneously. The filename is needed so that popping works correctly. Returns 0 for success or 1 for parsing errors. This function should be called any time the user defines a new set of units, which is typically at the top of a file, but could be within a file.

`pop. unitstring=NULL, dimstring=` filename or `NULL`, `value=0, outstring=NULL`. This pops lists of units off the unit stack. It goes to the lowest-most instance of filename in the nameStack if `dimstring` is defined and down one level if `dimstring` is `NULL`. This should be called with `dimstring` equal to `NULL` if the user says to end the current units, and with `dimstring` equal to the filename at the end of any file (whether units were defined there or not). Returns 0.

`working. unitstring=` space-separated unit list, `dimstring=NULL, value=0`, `outstring=NULL`. Set this to the units that all numbers should be converted to. If this is not entered, then the default working units are the first ones that are entered with the push function. Returns 0 for success, 1 for parsing error.

`parse. unitstring=` unit, `dimstring=NULL, value=` value to convert, `outstring=NULL`. This is an internal function, which is called by the convert function. It can be called from externally, but that's not its intent. This parses the unit string and turns it into a list of powers of base units, which it stores in the static variable `dimUnit`. This also converts the value based on the unit conversion that it determines.

The constants that are defined at the top of this function are hopefully self-explanatory, but are briefly described here. `numTypes` is the number of unit types, where types are things like length, time, mass, etc. `maxUnits` is the allocated space for the units of each type. `maxUnitChars` is the allocated space for each unit name. `maxStack` is the allocated space for the unit stack, meaning the number of nested unit definitions that are allowed. `workUnits` gives the unit index for the working units of each unit type. `dimUnit` gives the powers of the units, such as 2 for m^2, and is only used for recursive function calls. `haveWorkUnits` is set to 0 initially and then updated to 1 if working units have been defined; they are normally set to the first set of units that are entered, but can also be set to something else using the "working" function option.

The internal storage for this function starts with `numUnits`, which is the number of units that are defined internally for each unit type. `unitNames` is one list for each unit type, and in each list are the names of the units. Likewise, `unitRatios` is one list for each unit type, and in each list are conversion factors. `unitStack` is the stack of temporary units. Each time new units are used, they are pushed onto the stack. `nameStack` is the corresponding stack of filenames. Finally, `numStack` is the number of elements in the two stacks.