Documentation for string2.h and string2.c

Steven Andrews, © 2003-2012

Header

```
/* Steven Andrews, 11/01.
See documentation called string2_doc.doc
Copyright 2003-2009 by Steven Andrews.  This work is distributed under the
terms
of the Gnu Lesser General Public License (LGPL). */

#ifndef __string2_h
#define __string2_h

#define STRCHAR 256
#define STRCHARLONG 4000

int strisnumber(char *str);
int okname(char *name);
char *strrpbrk(char *cs,char *ct);
char *StringCopy(char *s);
unsigned char *PascalString(char *s);
char *EmptyString();
char *StrChrQuote(char *cs,char c);
int StrChrPQuote(char *cs,char c);
int StrrChrPQuote(char *cs,char c);
int strreadni(char *s,int n,int *a,char **endp);
int strreadnf(char *s,int n,float *a,char **endp);
int strreadnd(char *s,int n,double *a,char **endp);
int strreadns(char *s,int n,char **a,char **endp);
char *strnword(char *s,int n);
char *strnword1(char *s,int n);
int wordcount(char *s);
int symbolcount(char *s,char c);
int stringfind(char **slist,int n,char *s);
int strchrreplace(char *str,char charfrom,char charto);
int strstrreplace(char *str,char *strfrom,char *strto,int max);
int strbegin(char *cs,char *ct);
int strbslash2escseq(char *str);
void strcutwhite(char *str,int end);
int strwildcardmatch(char *pat,char *str);

#endif
```

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

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

Started writing 1/99; moderate testing. Works with Metrowerks C. Transferred code from Utility.c library to newly created string2.c library 11/01. Added array reading and `strnword` and transferred to Linux 11/01. Added `wordcount` 1/24/02. Added `StrrChrPQuote` 3/29/02. Added `StrChrPQuote` 10/29/02. Added `stringfind`

1/19/03. Added `strrpbrk` 6/11/03. Added `strreadns` 1/16/04. Renamed `isnumber` to `strisnumber` 6/9/04 to avoid a name collision with some other command. Added `strchrreplace` 2/13/06. Added `strbslash2escseq` 9/21/07. Added `symbolcount` 10/28/07. Added `strcutwhite` 11/16/07. Added `strstrreplace` 6/2/08. Added `strstrbegin` 10/28/09. Added `strwildcardmatch` 1/11/12. Added `strparenmatch` 1/30/12. Added `STRCHARLONG` 4/17/12.

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

Definitions

`#define STRCHAR 256`

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 4000`

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 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`.

Character locating

`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 the the standard library function `strpbrk`, except that it is for the last rather than first occurence.

`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 occurance 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 occurance of c is cs, returning its index if found. It ignores any c characters in double quotes or inside parentheses. Any level of paranthesis 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 occurance 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 occurance 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 mis-matched parentheses, -3 for mis-matched brackets, and -4 for mis-matched 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 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.

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.

`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.

`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 existance 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`.

String arrays

`int stringfind(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.

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 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 initiallized 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`.

```
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.

<u>Wildcards and enhanced wildcards</u>

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

Determines if the string in `str` is a match for the string in `pat`, which might include wildcard characters. 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. This function is case sensitive and it treats periods just like any other character. This function does not identify or return the represented text.

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

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.

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.

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

This is an internal function. 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 ***results,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 `results` 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; `results` 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.

```
int strexpandlogic(const char *pat,int start,int stop,char ***results,int
    top,int *nrptr,int *maxrptr);
```

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 precendence of the AND symbol. For normal use, enter `start` as 0, `stop` as -1, `results` as a pointer to a `char**` which is set to `NULL`, `top` to 0, `nrptr` as a pointer to a value that equals 0, and `maxrptr` as a pointer to a value that equals 0. The answers will be returned in an array of strings that is pointed to by `results` and that will have `*nrptr` values in it. The total number of strings created will also be returned by the function. The other inputs are primarily for recursion purposes. Enter pat as NULL to have the memory pointed to by `results` freed.

For more detail on the inputs, the input string `pat` is unchanged by this function. It is only investigated over the range from index `start` to index `stop-1`. The results array of strings is used here but allocated and freed with the library internal function `allocresults`. When this function is called, it appends the results list starting at index `top`. The results list is filled to level `*nrptr` and has `*maxrptr` total allocated spaces.

Processing the OR operator is fairly simple. Each operand that are separated by OR operators is expanded separately and the results are appended to the results list.

Processing the AND operator is much more complicated. Initially, AND processing is very similar to the OR segment; the function expands each operand separately and appends the results to the results list. At this point, `results` from 0 to `top` is old stuff that prior recursion levels created, and `results` from `top` to `temptop` are the expanded operands. Because each operand might expand to multiple results (e.g. for “A&{B|C}”), the `ampindx1` array lists the starting index for each operand and `ampindx2` lists one plus the ending index for each operand. Next, the function permutes the expanded operands, and puts the solutions in results from `temptop` to `ptop`. Within each permutation, the function goes through all possibilities by scanning the `indx` variables through the possible `ampindx` options. Finally, it condenses the results list to remove the individual operands.

```
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”. Returns 1 for a match, 0 for a non-match, -1 for failure to allocate memory, -2 for a missing OR operand, -3 for a missing AND operand, -4 for more than 8 sequential AND terms (which leads to an excessive number of permutations), and -5 for mis-matched braces. Enter `pat` and/or `str` as `NULL` to free internal stored memory.

```
int strEnhWildcardMatchAndSub(const char *pat,const char *str,const char
    *destpat,char *dest);
```
Not tested at all, and incomplete.
Identical to `strEnhWildcardMatch`, except that this also susbtitutes the represented text into a destination string and returns it as `dest`.