More String Standard Library Functions

The header `string.h` contains a number of useful utility functions. We met a few last week, but there are more:

String length:
`size_t strlen( const char *s );`

String copying: `char *strcpy( char *dest, const char *src ); // Requires dest to have min size strlen(src)+1`

Length-limited String copying: `char *strncpy( char *dest, const char *src, size_t n ); // Requires dest to have min size n, and strlen(src) < n`

String concatenation: `char *strcat( char *dest, const char *src ); // Requires dest to have min size strlen(dest)+strlen(src)+1`

Length-limited String concatenation: `char *strncat( char *dest, const char *src, size_t n ); // Requires dest to have size strlen(dest)+n+1`

This copies no more than the first n characters from `src` into `dest`, guaranteeing that `dest` is `nul`-terminated under all circumstances: If src contains `n` or more bytes, `strncat()` writes `n+1` bytes to `dest` (`n` from `src` plus the terminating `nul` byte)

String comparison: `int strcmp( const char *s1, const char *s2 );`

Lexicographically compares the strings s1 and s2. The return value is:

- `< 0` - if `str1` is less than `str2`
- `== 0` - if `str1` is equal to `str2`
- `> 0` - if `str1` is greater than `str2`

Length-limited string comparison: `int strncmp( const char *s1, const char *s2, size_t n );`

Which only compares up to the first `n` characters: the return value obeys the same lexicographic rules as `strcmp()`

`strcmp()` and `strncmp()` Example

```
#include <stdio.h>
#include <string.h>
int main( void ) {
  char a[] = "astring";
  char b[] = "astring";
  char c[] = "astr ing";
  if( strcmp(a, b) == 0 ) {
    printf("Strings a and b are the same\n");
  }
  if( strcmp(a, c) == 0 ) {
    printf("Strings a and c are the same\n");
  }
  if( strncmp(a, b, 5) == 0 ) {
    printf("Strings a#5 and b#5 are the same\n");
  }
  if( strncmp(a, c, 4) == 0 ) {
    printf("Strings a#4 and c#4 are the same\n");
  }
  return 0;
}
```

When run, this produces:

```
Strings a and b are the same
Strings a#5 and b#5 are the same
Strings a#4 and c#4 are the same
```

String comparison - ignoring case: `int strcasecmp( const char *s1, const char *s2 );`

Length-limited string comparison - ignoring case: `int strncasecmp( const char *s1, const char *s2, size_t n );`

Search for the first occurrence of a character within a string, searching forwards from the left hand end (the start): `char *strchr( const char *s, char ch );`

This either returns NULL - if `ch` is not present in `s` - or returns a pointer to the left-most occurrence of `ch` in `s`

Search for a character back from the right hand end of the string (the end): `char *strrchr( const char *s, char ch );`

This either returns NULL - if `ch` is not present in `s` - or returns a pointer to the right-most occurrence of `ch` in `s`

Search for a string in a string: `char * strstr( const char *haystack, const char *needle );`

This either returns NULL - if the needle (string) is not present in the haystack (a bigger string) - or returns a pointer to the left-most occurrence of the needle in the haystack

Search case insensitively for a string in a string: `char * strcasestr( const char *haystack, const char *needle );`

This ignores the case of the needle and the haystack

String tokenizing:

`char *strtok( char *str, const char *delim );`

The delimiter string is a sequence of possible delimiters, i.e. things you’re not interested in that separate the bits you want

See `man strtok` for details

String duplication: `char *strdup( const char *s );`

This is essentially:

```
char *dup = malloc( (1 + strlen(s)) * sizeof(char) );
if( dup == NULL ) return NULL;
strcpy( dup, s );
return dup;
```

You still need to check that the return value isn’t NULL

I often define a utility called `xstrdup()` which is basically `s = strdup()` plus `assert s != NULL`

The character classification macros in `ctype.h` (`isblank()`, `isdigit()`, `isalpha()`, `isalnum()`, `islower()`, `isupper()`) etc are extremely useful

On Unsafe String Functions

The C standard library contains many potentially unsafe string functions. Whenever you use them, check the documentation for the following:

- Whenever a buffer is being written to, check the requirements on the size of the destination buffer, and make sure you obey them
- If the function writes a string, check under what circumstances it terminates the string with a '`\0`'

For example - if the source string is long enough, `strncpy()` will not terminate the destination string with '`\0`', possibly causing later code to run off the end of the string. You should check (before using `strncpy()`) that `strlen(src) < n`

Have a look at `strlcpy()` and `strlcat()` for an example of safer functions. They both come from BSD, and are therefore non-portable. But you could easily write two similar functions yourself if you like

String Parsing Example

As an example of how one typically combines the above string handling functions, `ctype.h` classification macros, and low-level `char *` manipulation, let's develop a string parsing example:

```
// int data[maxnel];
// int nel = 0;
// char errmsg[BIGENOUGH];
// char *unused = parse_intlist( input, maxnel, errmsg, data, &nel );
//    Given a modifiable input string <input>, attempt to parse a []-enclosed,
//    comma-separated list of integers from the input, ignoring any amounts of
//    whitespace more or less anywhere.
//
//    If parsing fails for any reason - wrong input format, or even the
//    right format but there being more than maxnel integer values - write
//    an error message into errmsg[], and return NULL.
//
//    If parsing succeeds, set nel to the number of integers found (<= maxnel),
//    place those nel parsed integers into data[0..nel-1], and return the
//    pointer (into the input string) of the first unconsumed character
//    BEYOND any optional whitespace following the ']', If the input were
//    "[ 1001 , 13,15,17,19 ] leftover", then nel=5, data[0..4] =
//    1001,13,15,17,19, and the return value points at the 'l' of "leftover".
//
char *parse_intlist( char *input, int maxnel, char *errmsg, int *data, int *nel )
```

How would we build such a function?

It helps to understand the shape of correct input that we are going to try to parse, as a sequence of tiny character-sequence matches which form a state machine:

1. skip zero or more whitespace chars.
2. followed by the '[' - or error("'[' expected") if any other char
3. followed by skipping zero or more whitespace chars
4. followed by one or more digits, or ']' or error("digit or ']' expected") if any other char: if ']' goto step 7
5. followed by skipping zero or more whitespace chars
6. followed by the ']' or ',' or error("']' or ',' expected"): if ',' go back to step 3.
7. followed by skipping zero or more whitespace chars
8. if we get this far, we succeed and return a pointer to the first unconsumed character.

Steps (1), (3), (5) and (7) all skip optional whitespace, which assuming that s is a `char *` pointing at the unconsumed part of input, is `while( isspace(*s) ) s++;` is a simple char test, generating an error if it fails, otherwise an increment to move onto the next character:

```
if( *s != '[' ) {
        strcpy( errmsg, "missing '['" );
        return NULL;
}
s++;
```

Steps (3) through (6) implies a “while ! finished” or “do while ! finished” loop wrapped around the individual steps

At this point, I think the basic code can be written, in stages.

- Our first stage will only recognise matching []s with any amount of whitespace:

```
1    char *s = input;
2    while( isspace(*s) ) s++;                  // skipws
3    if( *s != '[' ) {                          // expect '['
4      strcpy( errmsg, "missing '['" ); return NULL;
5    }
6    s++;
7    while( isspace(*s) ) s++;                  // skipws
8    if( *s != ']' ) {                          // expect ']'
9      strcpy( errmsg, "']' expected" ); return NULL;
10   }
11   s++;
12   while( isspace(*s) ) s++;                  // skipws
13   return s;
```

- When we compile and run this against a test program runstage1 which invokes parse_intlist() for each command line argument, we get:

```
./runstage1 '[]' '[] leftover' '  [  ]  leftover'
pi([]): succeeded, unused <>

pi([] leftover): succeeded, unused <leftover>

pi(  [  ]  leftover): succeeded, unused <leftover>
```

```
./runstage1 '  ' '  [  '
pi(  ): failed, errmsg = missing '['

pi(  [  ): failed, errmsg = ']' expected
```

- Our second stage will only recognise a list of comma-separated single digits with whitespace inside [], without building anything. We replace lines 7-12 from the previous version with:

```
bool finished = false;
do {
  while( isspace(*s) ) s++;                    // skipws
                                               // expect digit or ']' or error
  printf( "debug: in loop looking for digit|']' at %s\n", s );
  if( *s == ']' ) {                            // ']':
    s++; finished = true;
  } else if( isdigit(*s) ) {                   // digit:
    printf( "debug: Found digit '%c'\n", *s );
    s++;
    while( isspace(*s) ) s++;                  // skipws
                                               // expect ',' or ']' or error
    printf( "debug: looking for ']'|',' at %s\n", s );
    if( *s == ']' ) {                          // ']':
      finished = true;
    } else if( *s != ',' ) {                   // ! ',':
      strcpy( errmsg, "',' or ']' expected" ); return NULL;
    }
    s++;
  } else {
    strcpy( errmsg, "digit or ']' expected" ); return NULL;
  }
} while( !finished );
```

- When we compile and run this against a test program runstage2 we get:

```
./runstage2 ' [ ] leftover' '[  1 ]' ' [1,2 , 3 ] leftover'
debug: in loop looking for digit|']' at ] leftover
pi( [ ] leftover): succeeded, unused <leftover>

debug: in loop looking for digit|']' at 1 ]
debug: Found digit '1'
pi([  1 ]): succeeded, unused <>

debug: in loop looking for digit|']' at 1,2 , 3 ] leftover
debug: Found digit '1'
debug: in loop looking for digit|']' at 2 , 3 ] leftover
debug: Found digit '2'
debug: in loop looking for digit|']' at 3 ] leftover
debug: Found digit '3'
pi( [1,2 , 3 ] leftover): succeeded, unused <leftover>
```

```
./runstage2 '   ' '   [  ' ' [ 1  ' ' [ 1, leftover'
pi(   ): failed, errmsg = missing '['

debug: in loop looking for digit|']' at
pi(   [  ): failed, errmsg = digit or ']' expected

debug: in loop looking for digit|']' at 1
debug: Found digit '1'
pi( [ 1  ): failed, errmsg = ',' or ']' expected

debug: in loop looking for digit|']' at 1, leftover
debug: Found digit '1'
debug: in loop looking for digit|']' at leftover
pi( [ 1, leftover): failed, errmsg = digit or ']' expected
```

- Note that one case is slightly surprising:

```
./runstage2 ' [ 1, ] leftover'
debug: in loop looking for digit|']' at 1, ] leftover
debug: Found digit '1'
debug: in loop looking for digit|']' at ] leftover
pi( [ 1, ] leftover): succeeded, unused <leftover>
```

This succeeds despite the trailing comma.

Looking at our program code, and at the state machine pseudo-code, ",]" is allowed. We could add extra code to disallow this, but personally I'm happy to allow a trailing comma.

- Our final stage recognises our full list of comma-separated integers with whitespace inside [], and stores them in the result data[] array - and checks for array overflow. We replace lines 9-10 in the previous version with:

```
char *start = s;                    // start of digit sequence
while( isdigit(*s) ) s++;           // skipws
char ch = *s;                       // char beyond end of digits
*s = '\0';                          // temporarily: terminate
int x = atoi(start);
if( *nel >= maxnel ) {              // check for data[] overflow
  strcpy( errmsg, "too many ints" ); return NULL;
}
data[*nel] = x;                     // store x into data[n++]
(*nel)++;
*s = ch;                            // restore the original unused char
printf( "debug: found int %d at %s\n", x, start );
```

- Note our use of a clever trick: when we see the first digit, we store a pointer to it (start). Then having tracked past all the digits to the first non-digit character, we temporarily terminate the string at that point, *remembering the character we've overwritten*. This allows us to use atoi(start) to parse the integer. Then we restore the original character and carry on parsing. *This is why we insisted that input was modifiable.*

- I've placed all the code for this into strparse.c, and provided a unit test program teststrparse.c (there were actually teststage1.c and teststage2.c unit tests too).
- Compiling and running ./teststrparse we get debugging messages and test successes - there are no failures. The complete output is rather long, but here's a snippet:

```
T pi("[ 1001 , 13,15,17,19 ] x"): should succeed: OK
T pi("[ 1001 , 13,15,17,19 ] x"): nel == 5: OK
T pi("[ 1001 , 13,15,17,19 ] x"): sum == 1065: OK
T pi("[ 1001 , 13,15,17,19 ] x"): correct 1st unused == x, ascii code 120: OK
..
T pi("[ 1001 , 1 "): should fail: OK
T pi("[ 1001 , 1 "): errmsg == '',' or ']' expected': OK
T pi("[ 1001 , 1 hello"): should fail: OK
T pi("[ 1001 , 1 hello"): errmsg == '',' or ']' expected': OK
T pi("[ 1001 , 13,15,17,19,212"): should fail: OK
T pi("[ 1001 , 13,15,17,19,212"): errmsg == 'too many ints': OK
```

- Note that it is testing calls to parse_intlist() that should succeed - and checking that they do succeed, delivering the right results - and also testing calls that should fail - and checking that they do fail, and that they generate the correct error message.
- Note also that this sort of low level string manipulation, and similar examples making more use of strchr() and all it's friends, is *very typical in C programming*, and the sort of thing you should all be competent in.