Chapter 6: Strings

In C, a string is not an intrinsic type. A C-string is the convention to have a one-dimensional array of characters which is terminated by a null-character, by a `'\0'`.

This means that a C-string with a content of `"abc"` will have four characters `'a'`, `'b'`, `'c'` and `'\0'`.

See the basic introduction to strings example.

Section 6.1: Tokenisation: strtok(), strtok_r() and strtok_s()

The function `strtok` breaks a string into a smaller strings, or tokens, using a set of delimiters.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    int toknum = 0;
    char src[] = "Hello,, world!";
    const char delimiters[] = ", !";
    char *token = strtok(src, delimiters);
    while (token != NULL)
    {
        printf("%d: [%s]\n", ++toknum, token);
        token = strtok(NULL, delimiters);
    }
    /* source is now "Hello\0, world\0\0" */
}
```

Output:

```
1: [Hello]
2: [world]
```

The string of delimiters may contain one or more delimiters and different delimiter strings may be used with each call to `strtok`.

Calls to `strtok` to continue tokenizing the same source string should not pass the source string again, but instead pass `NULL` as the first argument. If the same source string *is* passed then the first token will instead be re-tokenized. That is, given the same delimiters, `strtok` would simply return the first token again.

Note that as `strtok` does not allocate new memory for the tokens, *it modifies the source string*. That is, in the above example, the string `src` will be manipulated to produce the tokens that are referenced by the pointer returned by the calls to `strtok`. This means that the source string cannot be `const` (so it can't be a string literal). It also means that the identity of the delimiting byte is lost (i.e. in the example the "," and "!" are effectively deleted from the source string and you cannot tell which delimiter character matched).

Note also that multiple consecutive delimiters in the source string are treated as one; in the example, the second comma is ignored.

`strtok` is neither thread safe nor re-entrant because it uses a static buffer while parsing. This means that if a function calls `strtok`, no function that it calls while it is using `strtok` can also use `strtok`, and it cannot be called by any function that is itself using `strtok`.

An example that demonstrates the problems caused by the fact that `strtok`is not re-entrant is as follows:

```
char src[] = "1.2,3.5,4.2";
char *first = strtok(src, ",");

do
{
    char *part;
    /* Nested calls to strtok do not work as desired */
    printf("[%s]\n", first);
    part = strtok(first, ".");
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok(NULL, ".");
    }
} while ((first = strtok(NULL, ",")) != NULL);
```

Output:

```
[1.2]
 [1]
 [2]
```

The expected operation is that the outer `do while` loop should create three tokens consisting of each decimal number string (`"1.2"`, `"3.5"`, `"4.2"`), for each of which the `strtok` calls for the inner loop should split it into separate digit strings (`"1"`, `"2"`, `"3"`, `"5"`, `"4"`, `"2"`).

However, because `strtok` is not re-entrant, this does not occur. Instead the first `strtok` correctly creates the "1.2\0" token, and the inner loop correctly creates the tokens `"1"` and `"2"`. But then the `strtok` in the outer loop is at the end of the string used by the inner loop, and returns NULL immediately. The second and third substrings of the `src` array are not analyzed at all.

`Version < C11`

The standard C libraries do not contain a thread-safe or re-entrant version but some others do, such as POSIX' `strtok_r`. Note that on MSVC the `strtok` equivalent, `strtok_s` is thread-safe.

`Version ≥ C11`

C11 has an optional part, Annex K, that offers a thread-safe and re-entrant version named `strtok_s`. You can test for the feature with `__STDC_LIB_EXT1__`. This optional part is not widely supported.

The `strtok_s` function differs from the POSIX `strtok_r` function by guarding against storing outside of the string being tokenized, and by checking runtime constraints. On correctly written programs, though, the `strtok_s` and `strtok_r` behave the same.

Using `strtok_s` with the example now yields the correct response, like so:

```
/* you have to announce that you want to use Annex K */
#define __STDC_WANT_LIB_EXT1__ 1
#include <string.h>

#ifndef __STDC_LIB_EXT1__
# error "we need strtok_s from Annex K"
#endif
```

```
char src[] = "1.2,3.5,4.2";
char *next = NULL;
char *first = strtok_s(src, ",", &next);

do
{
    char *part;
    char *posn;

    printf("[%s]\n", first);
    part = strtok_s(first, ".", &posn);
    while (part != NULL)
    {
        printf(" [%s]\n", part);
        part = strtok_s(NULL, ".", &posn);
    }
}
while ((first = strtok_s(NULL, ",", &next)) != NULL);
```

And the output will be:

```
[1.2]
[1]
[2]
[3.5]
[3]
[5]
[4.2]
[4]
[2]
```

Section 6.2: String literals

String literals represent null-terminated, static-duration arrays of `char`. Because they have static storage duration, a string literal or a pointer to the same underlying array can safely be used in several ways that a pointer to an automatic array cannot. For example, returning a string literal from a function has well-defined behavior:

```
const char *get_hello() {
    return "Hello, World!";  /* safe */
}
```

For historical reasons, the elements of the array corresponding to a string literal are not formally `const`. Nevertheless, any attempt to modify them has undefined behavior. Typically, a program that attempts to modify the array corresponding to a string literal will crash or otherwise malfunction.

```
char *foo = "hello";
foo[0] = 'y';  /* Undefined behavior - BAD! */
```

Where a pointer points to a string literal -- or where it sometimes may do -- it is advisable to declare that pointer's referent `const` to avoid engaging such undefined behavior accidentally.

```
const char *foo = "hello";
/* GOOD: can't modify the string pointed to by foo */
```

On the other hand, a pointer to or into the underlying array of a string literal is not itself inherently special; its value can freely be modified to point to something else:

```
char *foo = "hello";
foo = "World!"; /* OK - we're just changing what foo points to */
```

Furthermore, although initializers for `char` arrays can have the same form as string literals, use of such an initializer does not confer the characteristics of a string literal on the initialized array. The initializer simply designates the length and initial contents of the array. In particular, the elements are modifiable if not explicitly declared `const`:

```
char foo[] = "hello";
foo[0] = 'y';  /* OK! */
```

Section 6.3: Calculate the Length: strlen()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        puts("Argument missing.");
        return EXIT_FAILURE;
    }

    size_t len = strlen(argv[1]);
    printf("The length of the second argument is %zu.\\n", len);

    return EXIT_SUCCESS;
}
```

This program computes the length of its second input argument and stores the result in `len`. It then prints that length to the terminal. For example, when run with the parameters `program_name "Hello, world!"`, the program will output `The length of the second argument is 13.` because the string `Hello, world!` is 13 characters long.

`strlen` counts all the **bytes** from the beginning of the string up to, but not including, the terminating NUL character, `'\\0'`. As such, it can only be used when the string is *guaranteed* to be NUL-terminated.

Also keep in mind that if the string contains any Unicode characters, `strlen` will not tell you how many characters are in the string (since some characters may be multiple bytes long). In such cases, you need to count the characters (*i.e.*, code units) yourself. Consider the output of the following example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char asciiString[50] = "Hello world!";
    char utf8String[50] = "Γειά σου Κόσμε!"; /* "Hello World!" in Greek */

    printf("asciiString has %zu bytes in the array\\n", sizeof(asciiString));
    printf("utf8String has %zu bytes in the array\\n", sizeof(utf8String));
    printf("\\"%s\\" is %zu bytes\\n", asciiString, strlen(asciiString));
    printf("\\"%s\\" is %zu bytes\\n", utf8String, strlen(utf8String));
}
```

Output:

```
asciiString has 50 bytes in the array
utf8String has 50 bytes in the array
"Hello world!" is 12 bytes
"Γειά σου Κόσμε!" is 27 bytes
```

Section 6.4: Basic introduction to strings

In C, a **string** is a sequence of characters that is terminated by a null character ('\0').

We can create strings using **string literals**, which are sequences of characters surrounded by double quotation marks; for example, take the string literal `"hello world"`. String literals are automatically null-terminated.

We can create strings using several methods. For instance, we can declare a `char *` and initialize it to point to the first character of a string:

```
char * string = "hello world";
```

When initializing a `char *` to a string constant as above, the string itself is usually allocated in read-only data; `string` is a pointer to the first element of the array, which is the character `'h'`.

Since the string literal is allocated in read-only memory, it is non-modifiable1. Any attempt to modify it will lead to undefined behaviour, so it's better to add `const` to get a compile-time error like this

```
char const * string = "hello world";
```

It has similar effect2 as

```
char const string_arr[] = "hello world";
```

To create a modifiable string, you can declare a character array and initialize its contents using a string literal, like so:

```
char modifiable_string[] = "hello world";
```

This is equivalent to the following:

```
char modifiable_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};
```

Since the second version uses brace-enclosed initializer, the string is not automatically null-terminated unless a `'\0'` character is included explicitly in the character array usually as its last element.

1 Non-modifiable implies that the characters in the string literal can't be modified, but remember that the pointer `string` can be modified (can point somewhere else or can be incremented or decremented).

2 Both strings have similar effect in a sense that characters of both strings can't be modified. It should be noted that `string` is a pointer to `char` and it is a modifiable l-value so it can be incremented or point to some other location while the array `string_arr` is a non-modifiable l-value, it can't be modified.

Section 6.5: Copying strings

Pointer assignments do not copy strings

You can use the = operator to copy integers, but you cannot use the = operator to copy strings in C. Strings in C are represented as arrays of characters with a terminating null-character, so using the = operator will only save the address (pointer) of a string.

```c
#include <stdio.h>

int main(void) {
    int a = 10, b;
    char c[] = "abc", *d;

    b = a; /* Integer is copied */
    a = 20; /* Modifying a leaves b unchanged - b is a 'deep copy' of a */
    printf("%d %d\n", a, b); /* "20 10" will be printed */

    d = c;
    /* Only copies the address of the string -
    there is still only one string stored in memory */

    c[1] = 'x';
    /* Modifies the original string - d[1] = 'x' will do exactly the same thing */

    printf("%s %s\n", c, d); /* "axc axc" will be printed */

    return 0;
}
```

The above example compiled because we used `char *d` rather than `char d[3]`. Using the latter would cause a compiler error. You cannot assign to arrays in C.

```c
#include <stdio.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    b = a; /* compile error */
    printf("%s\n", b);

    return 0;
}
```

Copying strings using standard functions
`strcpy()`

To actually copy strings, `strcpy()` function is available in `string.h`. Enough space must be allocated for the destination before copying.

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "abc";
    char b[8];

    strcpy(b, a); /* think "b special equals a" */
    printf("%s\n", b); /* "abc" will be printed */

    return 0;
}
```

Version ≥ C99

`snprintf()`

To avoid buffer overrun, `snprintf()` may be used. It is not the best solution performance-wise since it has to parse the template string, but it is the only buffer limit-safe function for copying strings readily-available in standard library, that can be used without any extra steps.

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char a[] = "012345678901234567890";
    char b[8];

#if 0
    strcpy(b, a); /* causes buffer overrun (undefined behavior), so do not execute this here! */
#endif

    snprintf(b, sizeof(b), "%s", a); /* does not cause buffer overrun */
    printf("%s\n", b); /* "0123456" will be printed */

    return 0;
}
```

`strncat()`

A second option, with better performance, is to use `strncat()` (a buffer overflow checking version of `strcat()`) - it takes a third argument that tells it the maximum number of bytes to copy:

```
char dest[32];

dest[0] = '\0';
strncat(dest, source, sizeof(dest) - 1);
    /* copies up to the first (sizeof(dest) - 1) elements of source into dest,
    then puts a \0 on the end of dest */
```

Note that this formulation use `sizeof(dest) - 1`; this is crucial because `strncat()` always adds a null byte (good), but doesn't count that in the size of the string (a cause of confusion and buffer overwrites).

Also note that the alternative — concatenating after a non-empty string — is even more fraught. Consider:

```
char dst[24] = "Clownfish: ";
char src[] = "Marvin and Nemo";
size_t len = strlen(dst);

strncat(dst, src, sizeof(dst) - len - 1);
printf("%zu: [%s]\n", strlen(dst), dst);
```

The output is:

```
23: [Clownfish: Marvin and N]
```

Note, though, that the size specified as the length was *not* the size of the destination array, but the amount of space left in it, not counting the terminal null byte. This can cause big overwriting problems. It is also a bit wasteful; to specify the length argument correctly, you know the length of the data in the destination, so you could instead specify the address of the null byte at the end of the existing content, saving `strncat()` from rescanning it:

```
    strcpy(dst, "Clownfish: ");
    assert(len < sizeof(dst) - 1);
    strncat(dst + len, src, sizeof(dst) - len - 1);
    printf("%zu: [%s]\n", strlen(dst), dst);
```

This produces the same output as before, but `strncat()` doesn't have to scan over the existing content of `dst` before it starts copying.

`strncpy()`

The last option is the `strncpy()` function. Although you might think it should come first, it is a rather deceptive function that has two main gotchas:

1. If copying via `strncpy()` hits the buffer limit, a terminating null-character won't be written.
2. `strncpy()` always completely fills the destination, with null bytes if necessary.

(Such quirky implementation is historical and was initially intended for handling UNIX file names)

The only correct way to use it is to manually ensure null-termination:

```
strncpy(b, a, sizeof(b)); /* the third parameter is destination buffer size */
b[sizeof(b)/sizeof(*b) - 1] = '\0'; /* terminate the string */
printf("%s\n", b); /* "0123456" will be printed */
```

Even then, if you have a big buffer it becomes very inefficient to use `strncpy()` because of additional null padding.

Section 6.6: Iterating Over the Characters in a String

If we know the length of the string, we can use a for loop to iterate over its characters:

```
char * string = "hello world"; /* This 11 chars long, excluding the 0-terminator. */
size_t i = 0;
for (; i < 11; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Alternatively, we can use the standard function `strlen()` to get the length of a string if we don't know what the string is:

```
size_t length = strlen(string);
size_t i = 0;
for (; i < length; i++) {
    printf("%c\n", string[i]);    /* Print each character of the string. */
}
```

Finally, we can take advantage of the fact that strings in C are guaranteed to be null-terminated (which we already did when passing it to `strlen()` in the previous example ;-)). We can iterate over the array regardless of its size and stop iterating once we reach a null-character:

```
size_t i = 0;
while (string[i] != '\0') {       /* Stop looping when we reach the null-character. */
    printf("%c\n", string[i]);    /* Print each character of the string. */
    i++;
}
```

Section 6.7: Creating Arrays of Strings

An array of strings can mean a couple of things:

1. An array whose elements are `char *`s
2. An array whose elements are arrays of `char`s

We can create an array of character pointers like so:

```
char * string_array[] = {
    "foo",
    "bar",
    "baz"
};
```

Remember: when we assign string literals to `char *`, the strings themselves are allocated in read-only memory. However, the array `string_array` is allocated in read/write memory. This means that we can modify the pointers in the array, but we cannot modify the strings they point to.

In C, the parameter to main `argv` (the array of command-line arguments passed when the program was run) is an array of `char *`: `char * argv[]`.

We can also create arrays of character arrays. Since strings are arrays of characters, an array of strings is simply an array whose elements are arrays of characters:

```
char modifiable_string_array_literals[][4] = {
    "foo",
    "bar",
    "baz"
};
```

This is equivalent to:

```
char modifiable_string_array[][4] = {
    {'f', 'o', 'o', '\0'},
    {'b', 'a', 'r', '\0'},
    {'b', 'a', 'z', '\0'}
};
```

Note that we specify `4` as the size of the second dimension of the array; each of the strings in our array is actually 4 bytes since we must include the null-terminating character.

Section 6.8: Convert Strings to Number: atoi(), atof() (dangerous, don't use them)

Warning: The functions `atoi`, `atol`, `atoll` and `atof` are inherently unsafe, because: *If the value of the result cannot be represented, the behavior is undefined.* (7.20.1p1)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int val;
    if (argc < 2)
    {
        printf("Usage: %s <integer>\n", argv[0]);
        return 0;
    }

    val = atoi(argv[1]);

    printf("String value = %s, Int value = %d\n", argv[1], val);

    return 0;
}
```

When the string to be converted is a valid decimal integer that is in range, the function works:

```
$ ./atoi 100
String value = 100, Int value = 100
$ ./atoi 200
String value = 200, Int value = 200
```

For strings that start with a number, followed by something else, only the initial number is parsed:

```
$ ./atoi 0x200
0
$ ./atoi 0123x300
123
```

In all other cases, the behavior is undefined:

```
$ ./atoi hello
Formatting the hard disk...
```

Because of the ambiguities above and this undefined behavior, the `atoi` family of functions should never be used.

- To convert to `long int`, use `strtol()` instead of `atol()`.
- To convert to `double`, use `strtod()` instead of `atof()`.

Version ≥ C99

- To convert to `long long int`, use `strtoll()` instead of `atoll()`.

Section 6.9: string formatted data read/write

Write formatted data to string

```
int sprintf ( char * str, const char * format, ... );
```

use `sprintf` function to write float data to string.

```
#include <stdio.h>
int main ()
{
  char buffer [50];
  double PI = 3.1415926;
  sprintf (buffer, "PI = %.7f", PI);
  printf ("%s\n",buffer);
  return 0;
}
```

Read formatted data from string

```
int sscanf ( const char * s, const char * format, ...);
```

use `sscanf`function to parse formatted data.

```
#include <stdio.h>
int main ()
{
  char sentence []="date : 06-06-2012";
  char str [50];
  int year;
  int month;
  int day;
  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);
  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);
  return 0;
}
```

Section 6.10: Find first/last occurrence of a specific character: strchr(), strrchr()

The `strchr` and `strrchr` functions find a character in a string, that is in a NUL-terminated character array. `strchr` return a pointer to the first occurrence and `strrchr` to the last one.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char toSearchFor = 'A';

    /* Exit if no second argument is found. */
    if (argc != 2)
    {
        printf("Argument missing.\n");
        return EXIT_FAILURE;
    }

    {
        char *firstOcc = strchr(argv[1], toSearchFor);
        if (firstOcc != NULL)
        {
            printf("First position of %c in %s is %td.\n",
              toSearchFor, argv[1], firstOcc-argv[1]); /* A pointer difference's result
                                     is a signed integer and uses the length modifier 't'. */
        }
        else
        {
            printf("%c is not in %s.\n", toSearchFor, argv[1]);
        }
    }

    {
        char *lastOcc = strrchr(argv[1], toSearchFor);
        if (lastOcc != NULL)
        {
            printf("Last position of %c in %s is %td.\n",
                toSearchFor, argv[1], lastOcc-argv[1]);
        }
    }

    return EXIT_SUCCESS;
}
```

Outputs (after having generate an executable named `pos`):

```
$ ./pos AAAAAAA
First position of A in AAAAAAA is 0.
Last position of A in AAAAAAA is 6.
$ ./pos BAbbbbbAccccAAAAzzz
First position of A in BAbbbbbAccccAAAAzzz is 1.
Last position of A in BAbbbbbAccccAAAAzzz is 15.
$  ./pos qwerty
A is not in qwerty.
```

One common use for `strrchr` is to extract a file name from a path. For example to extract `myfile.txt` from `C:\Users\eak\myfile.txt`:

```
char *getFileName(const char *path)
{
    char *pend;

    if ((pend = strrchr(path, '\')) != NULL)
        return pend + 1;

    return NULL;
}
```

Section 6.11: Copy and Concatenation: strcpy(), strcat()

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  /* Always ensure that your string is large enough to contain the characters
   * and a terminating NUL character ('\0')!
   */
  char mystring[10];

  /* Copy "foo" into `mystring`, until a NUL character is encountered. */
  strcpy(mystring, "foo");
  printf("%s\n", mystring);

  /* At this point, we used 4 chars of `mystring`, the 3 characters of "foo",
   * and the NUL terminating byte.
   */

  /* Append "bar" to `mystring`. */
  strcat(mystring, "bar");
  printf("%s\n", mystring);

  /* We now use 7 characters of `mystring`: "foo" requires 3, "bar" requires 3
   * and there is a terminating NUL character ('\0') at the end.
   */
  /* Copy "bar" into `mystring`, overwriting the former contents. */
  strcpy(mystring, "bar");
  printf("%s\n", mystring);

  return 0;
}
```

Outputs:

```
foo
foobar
bar
```

If you append to or from or copy from an existing string, ensure it is NUL-terminated!

String literals (e.g. `"foo"`) will always be NUL-terminated by the compiler.

Section 6.12: Comparsion: strcmp(), strncmp(), strcasecmp(), strncasecmp()

The `strcase*`-functions are not Standard C, but a POSIX extension.

The `strcmp` function lexicographically compare two null-terminated character arrays. The functions return a negative value if the first argument appears before the second in lexicographical order, zero if they compare equal, or positive if the first argument appears after the second in lexicographical order.

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcmp(lhs, rhs); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "BBB");
    compare("BBB", "CCCCC");
    compare("BBB", "AAAAAA");
    return 0;
}
```

Outputs:

```
BBB equals BBB
BBB comes before CCCCC
BBB comes after AAAAAA
```

As `strcmp`, `strcasecmp` function also compares lexicographically its arguments after translating each character to its lowercase correspondent:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs)
{
    int result = strcasecmp(lhs, rhs); // compute case-insensitive comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "bBB");
    compare("BBB", "ccCCC");
    compare("BBB", "aaaaaa");
    return 0;
}
```

Outputs:

```
BBB equals bBB
BBB comes before ccCCC
BBB comes after aaaaaa
```

`strncmp` and `strncasecmp` compare at most n characters:

```
#include <stdio.h>
#include <string.h>

void compare(char const *lhs, char const *rhs, int n)
{
    int result = strncmp(lhs, rhs, n); // compute comparison once
    if (result < 0) {
        printf("%s comes before %s\n", lhs, rhs);
    } else if (result == 0) {
        printf("%s equals %s\n", lhs, rhs);
    } else { // last case: result > 0
        printf("%s comes after %s\n", lhs, rhs);
    }
}

int main(void)
{
    compare("BBB", "Bb", 1);
    compare("BBB", "Bb", 2);
    compare("BBB", "Bb", 3);
    return 0;
}
```

Outputs:

```
BBB equals Bb
BBB comes before Bb
BBB comes before Bb
```

Section 6.13: Safely convert Strings to Number: strtoX functions

Version ≥ C99

Since C99 the C library has a set of safe conversion functions that interpret a string as a number. Their names are of the form `strtoX`, where `X` is one of `l`, `ul`, `d`, etc to determine the target type of the conversion

```
double strtod(char const* p, char** endptr);
long double strtold(char const* p, char** endptr);
```

They provide checking that a conversion had an over- or underflow:

```
double ret = strtod(argv[1], 0); /* attempt conversion */

/* check the conversion result. */
if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return;  /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

If the string in fact contains no number at all, this usage of `strtod` returns `0.0`.

If this is not satisfactory, the additional parameter `endptr` can be used. It is a pointer to pointer that will be pointed to the end of the detected number in the string. If it is set to `0`, as above, or `NULL`, it is simply ignored.

This `endptr` parameter provides indicates if there has been a successful conversion and if so, where the number ended:

```
char *check = 0;
double ret = strtod(argv[1], &check); /* attempt conversion */

/* check the conversion result. */
if (argv[1] == check)
    return; /* No number was detected in string */
else if ((ret == HUGE_VAL || ret == -HUGE_VAL) && errno == ERANGE)
    return; /* numeric overflow in in string */
else if (ret == HUGE_VAL && errno == ERANGE)
    return; /* numeric underflow in in string */

/* At this point we know that everything went fine so ret may be used */
```

There are analogous functions to convert to the wider integer types:

```
long strtol(char const* p, char** endptr, int nbase);
long long strtoll(char const* p, char** endptr, int nbase);
unsigned long strtoul(char const* p, char** endptr, int nbase);
unsigned long long strtoull(char const* p, char** endptr, int nbase);
```

These functions have a third parameter `nbase` that holds the number base in which the number is written.

```
long a = strtol("101",   0, 2 ); /* a = 5L */
long b = strtol("101",   0, 8 ); /* b = 65L */
long c = strtol("101",   0, 10); /* c = 101L */
long d = strtol("101",   0, 16); /* d = 257L */
long e = strtol("101",   0, 0 ); /* e = 101L */
long f = strtol("0101",  0, 0 ); /* f = 65L */
long g = strtol("0x101", 0, 0 ); /* g = 257L */
```

The special value `0` for `nbase` means the string is interpreted in the same way as number literals are interpreted in a C program: a prefix of `0x` corresponds to a hexadecimal representation, otherwise a leading `0` is octal and all other numbers are seen as decimal.

Thus the most practical way to interpret a command-line argument as a number would be

```
int main(int argc, char* argv[] {
    if (argc < 1)
        return EXIT_FAILURE; /* No number given. */

    /* use strtoull because size_t may be wide */
    size_t mySize = strtoull(argv[1], 0, 0);

    /* then check conversion results. */

     ...

    return EXIT_SUCCESS;
}
```

This means that the program can be called with a parameter in octal, decimal or hexadecimal.

Section 6.14: strspn and strcspn

Given a string, `strspn` calculates the length of the initial substring (span) consisting solely of a specific list of characters. `strcspn` is similar, except it calculates the length of the initial substring consisting of any characters except those listed:

```
/*
  Provided a string of "tokens" delimited by "separators", print the tokens along
  with the token separators that get skipped.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    const char sepchars[] = ",.;!?";
    char foo[] = ";ball call,.fall gall hall!?.,";
    char *s;
    int n;

    for (s = foo; *s != 0; /*empty*/) {
        /* Get the number of token separator characters. */
        n = (int)strspn(s, sepchars);

        if (n > 0)
            printf("skipping separators: << %.*s >> (length=%d)\n", n, s, n);

        /* Actually skip the separators now. */
        s += n;

        /* Get the number of token (non-separator) characters. */
        n = (int)strcspn(s, sepchars);
        if (n > 0)
            printf("token found: << %.*s >> (length=%d)\n", n, s, n);

        /* Skip the token now. */
        s += n;
    }

    printf("== token list exhausted ==\n");

    return 0;
}
```

Analogous functions using wide-character strings are `wcsspn` and `wcscspn`; they're used the same way.