CSCI 2132: Software Development

Strings in C

Norbert Zeh

Faculty of Computer Science
Dalhousie University

Winter 2019

Strings in C

- String = array of characters
- No explicit length information
- End marked with `NUL` (`'\0'`)

Example: `"hello, world\n"`

h	e	l	l	o	,		w	o	r	l	d	\n	\0
104	101	108	108	111	44	32	119	111	114	108	100	10	0

String Length vs Size

The NUL character occupies one byte.

⇒ Need one more byte than the length (number of characters) to store it.

Allocating more memory is okay!

```c
char str[14] = “hello, world\n”;
```

h	e	l	l	o	,		w	o	r	l	d	\n	\0

```c
char str[14] = “hello”;
```

h	e	l	l	o	\0	\0	\0	\0	\0	\0	\0	\0	\0

String Length vs Size

The `NUL` character occupies one byte.

⇒ Need one more byte than the length (number of characters) to store it.

Allocating more memory is okay!

```c
char str[14] = “hello, world\n”;
```

h	e	l	l	o	,		w	o	r	l	d	\n	\0

```c
char str[14] = “hello, world\n”; str[5] = 0;
```

h	e	l	l	o	\0		w	o	r	l	d	\n	\0

Working with Individual Characters

Since strings are character arrays, we can access individual characters (even modify them):

```
char text[32] = “hello, world!\n”;
for (int i = 0; text[i] != 0; ++i) {
  printf(“%c”, text[i]);
}
```

```
char text[32] = “hello, world!\n”;
text[5] = 0;
for (int i = 0; text[i] != 0; ++i) {
  printf(“%c”, text[i]);
}
```

Writing Strings

Using `printf`'s `%s` conversion specifier:

```
char text[] = “A string”;
printf(“%s\n”, text);
```

Using `puts` (“Put string”):

```
char text[] = “A string”;
puts(text);
```

Reading Strings

Using scanf's %s conversion

Why don't we use &text here?

```
char text[32];
scanf("%s", text);
```

- Skips whitespace and reads a non-whitespace word.
- Dangerous: `str` must have enough capacity.

Reading Strings

Using `scanf`’s `%s` conversion specifier:

```
char text[32];
scanf(“%s”, text);
```

- Skips whitespace and reads a non-whitespace word
- Dangerous: `str` must have enough capacity

Using `gets` (“Get string”):

```
char text[32];
gets(text);
```

- Reads up to the end of line.
- Again, the buffer must be big enough.

Reading Strings

Using `fgets` ("File get string"):

```
char text[32];
fgets(text, 32, stdin);
```

- Reads up to the end of line or up to a maximum number of characters.
- Prevents buffer overflow.
- Needs the file to be read as an argument.

Reading Strings

Using `getline`:

```c
char *text = NULL;
size_t capacity = 0;
int numchars;
numchars = getline(&text, &capacity, stdin);
free(text);
```

- Reads up to the end of line.
- Buffer must be allocated on the heap and may be reallocated if it is too small.
- Buffer must be freed after use.

Reading Strings

As individual characters (safe):

```
char text[32];
scanf(“%31c”, text);
text[31] = 0;
```

```
char text[32];
int i = 0;
do {
  scanf(“%c”, text + i++);
} while (i < 32 && text[i] != 0);
text[i] = 0;
```

Buffer Overflow

Reading input into a fixed-length buffer without length control (`scanf` or `gets`) may write past the end of the buffer!

May crash the program! This is the best-case scenario!

Worse: A malicious user can provide a long carefully crafted input to overwrite stack and cause the program to start executing injected machine code.

Major source of security issues (buffer overflow attack).

Useful String Functions

Find the length of a string: `strlen`

```
char text[32] = “Hello”;
printf(“%lu\n”, strlen(text));
```

Useful String Functions

Find the length of a string: `strlen`

```
char text[32] = “Hello”;
printf(“%lu\n”, strlen(text));
```

Compare two strings: `strcmp`

- Return values:
  - < 0: $x < y$
  - 0: $x == y$
  - > 0: $x > y$

```
char a[] = “Hello”, b[] = “Hearth”;
if (strcmp(a, b) > 0) {
  // This gets executed
}
```

Useful String Functions

Compare two prefixes of strings: `strncmp`

```
char a[] = “Hello”, b[] = “Hearth”;
if (strncmp(a, b, 2) == 0) {
  // This gets executed
}
```

```
char a[] = “Hello”, b[] = “Hell”;
if (strncmp(a, b, 5) > 0) {
  // This gets executed
}
```

Useful String Functions

Copy a string: `strcpy`

- The source and destination must not overlap!

```
char src[] = “Hello”, dst[32];
strcpy(dst, src);
```

Useful String Functions

Copy a string: `strcpy`

- The source and destination must not overlap!

```
char src[] = “Hello”, dst[32];
strcpy(dst, src);
```

Copy a string up to a maximum number of characters: `strncpy`

```
char src[] = “Hello”, dst[32];
strncpy(dst, src, 3);
```

Also, see `stpcpy` and `stpncpy`.

Useful String Functions

Append a string to another string: `strcat`

```
char src[] = “world”, dst[32] = “Hello, “;
strcat(dst, src);
```

Append a string up to a maximum number of characters: `strncat`

```
char src[] = “world”, dst[32] = “Hello, “;
strncat(dst, src, 24);
```

```
char src[] = “worldwide”, dst[32] = “Hello, “;
strncat(dst, src, 5);
```

Manipulating Raw Memory

Copy a block of memory: `memcpy`

```
// “Manual string copy”
char src[] = “Hello, world”, dst[32];
memcpy(dst, src, strlen(src) + 1);
```

```
int src[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int dst[10];
memcpy(dst, src, 10 * sizeof(int));
```

Copy a block of memory with possible overlap: `memmove`

```
char str[] = “Remove the the duplication”;
memmove(str + 7, str + 11, strlen(str + 11) + 1);
```