Character strings in C

- Character strings (or just “strings”) are not separate data type.
- **String is array of chars, which ends with value ‘\0’.**
- Several elements of syntax and support in standard libraries make strings special entity of C language.
- Everything else is programmers responsibility.
- That is why we have to be extra careful, because working with strings can lead to a lot of problems.

Example:

```
char buffer[21];
```

If we treat this array as a string, we can place 20 characters in it.

Specific syntax

- There exists string literal.

```
“this is string literal”
```

  - String concatenation:

    ```
    “string” “ literal” “ with” “ separated” “ words”
    “string literal with separated words”
    ```

    It is useful in some cases, for example when continuing in the new line.

- Initialization

```
char string[] = {1, 2, 3, 4, 5};
char string[] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘\0’};
char string[] = “abcde”;
char* string = {1, 2, 3, 4, 5};
char* string = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘\0’};
char* string = “abcde”;
```

Where is a string literal stored?

```
char* p = “Hello!”;
p[3] = ‘t’;
printf(“Hello!”);
scanf(“%s”, str);
if (strcmp(“Hello!”, str) == 0)
{
    ...
}
```

- All three of `Hello`! string literals can end up to be one, i.e. reuse single memory space.
- That is why your code should not change string literals.

Type of string literal

- What is the type of a string literal?

Type of string literal

- What is the type of a string literal?

  `char*`

- Why it is not `const char*`, since you should not change it?
- Because `const` came in later...

- In C++ the type of string literal was changed to be `const char*`

Character literal vs string literal

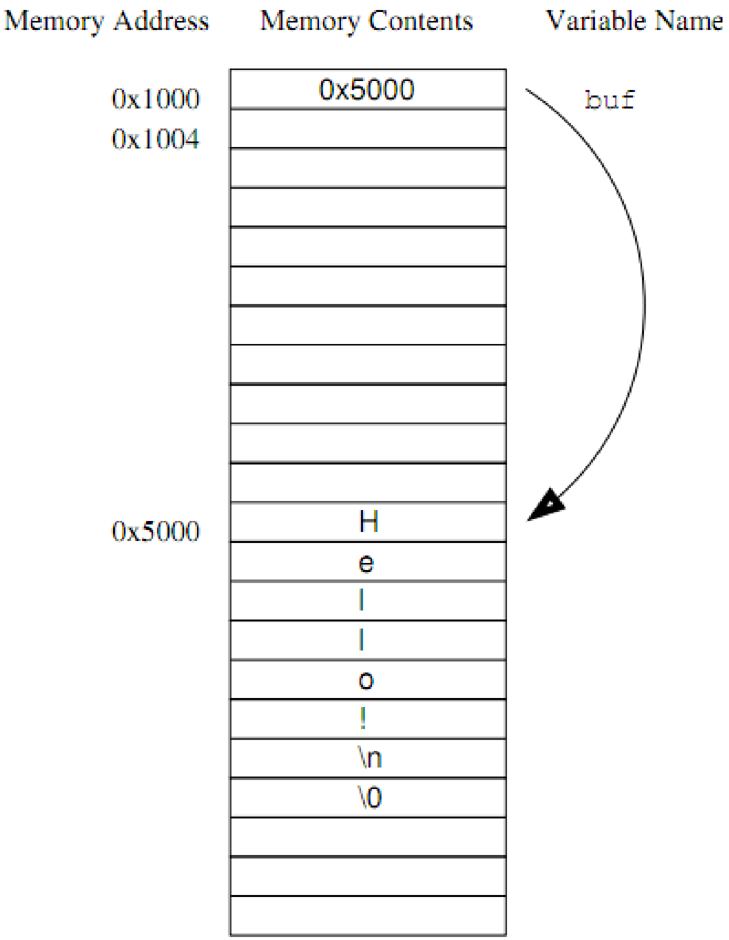
Character literal uses single quotes, not double quotes (’, not ”).

```
char buf[10];
buf[0] = ’A’; /* correct */
buf[0] = "A"; /* incorrect */
buf[1] = ’\0’;   /* NULL terminator */
```

Example

```
char* buf = "Hello!\n";
```

- Variable **buf** is pointer on memory where the string is located.
- Note null ('\0') character at the end – it is created automatically.



Memory Address	Memory Contents	Variable Name
0x1000	0x5000	buf
0x1004		
0x5000	H	
	e	
	l	
	l	
	o	
	!	
	\n	
	\0	

Library support

- \0 at the end is important for library functions because they expect it.

- Special format specifier in printf and scanf (and related functions).

```
char str[] = “Nesto”;
int i;
printf(“%s”, str); // what if there is no \0 at the end?
scanf(“%d%s”, &i, str); // what if more than 5 chars are
read?
```

- Library functions that work with string are in these headers
  - string.h
  - stdlib.h
  - stdio.h

Copying strings

```
char* buf1 = "Hello";
char* buf2 = "olleH";
buf2 = buf1;
buf2[2] = ‘M’;
printf("%s %s", buf1, buf2);
```

Runtime error!

```
char* buf1 = "Hello";
char buf2[100];
buf2 = buf1;

Compile error!
```

```
#include <string.h>
char* buf1 = "Hello";
char buf2[100];
strcpy(buf2, buf1);
buf2[2] = ‘M’;
printf("%s %s", buf1, buf2);
Output: Hello HeMlo
```

Copying strings

```
const char* buf1 = "Hello";
const char* buf2 = "olleH";
buf2 = buf1;
buf2[2] = ‘M’;
printf("%s %s", buf1, buf2);
Compiler error!
```

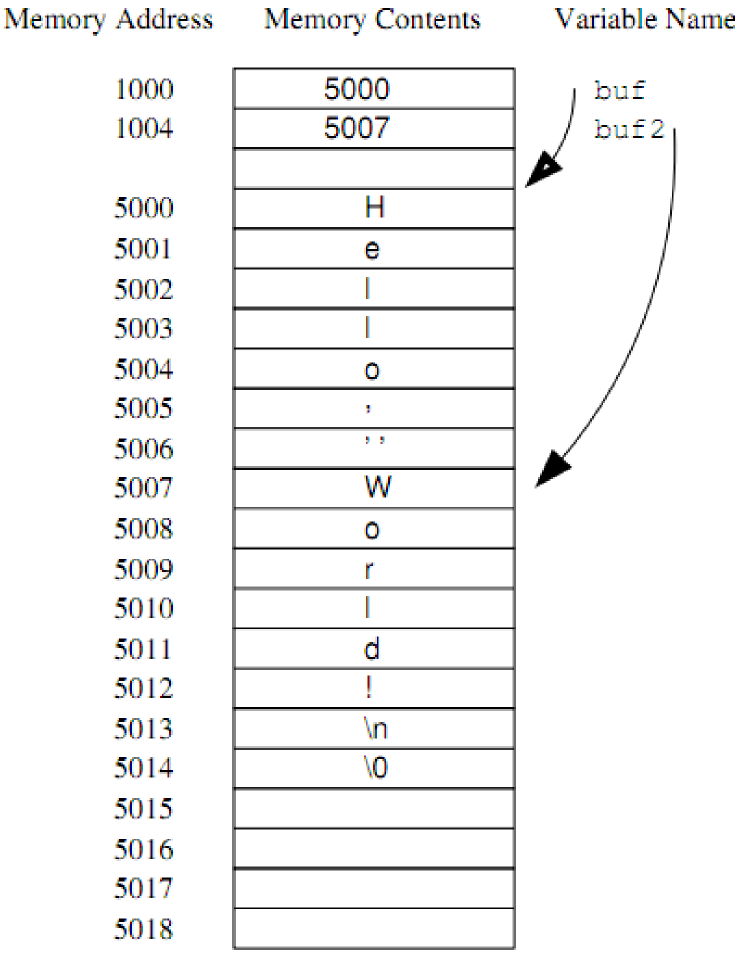
```
const char* buf1 = "Hello";
char buf2[100];
buf2 = buf1;

Compile error!
```

```
#include <string.h>
const char* buf1 = "Hello";
char buf2[100];
strcpy(buf2, buf1);
buf2[2] = ‘M’;
printf("%s %s", buf1, buf2);
Output: Hello HeMlo
```

Example

```
char buf[] = "Hello, World!\n";
char* buf2 = buf + 7;
printf("buf: %s\n", buf);
printf("buf2: %s\n", buf2);
buf2[0] = ’M’;
printf("buf: %s\n", buf);
```

What is the output?



Memory Address	Memory Contents	Variable Name
1000	5000	buf
1004	5007	buf2
5000	H	
5001	e	
5002	l	
5003	l	
5004	o	
5005	,	
5006	' '	
5007	W	
5008	o	
5009	r	
5010	l	
5011	d	
5012	!	
5013	\n	
5014	\0	
5015		
5016		
5017		
5018		

Overwriting buffer

String is not resized automatically. Buffer (piece of memory) allocated to it does not change.

Example:

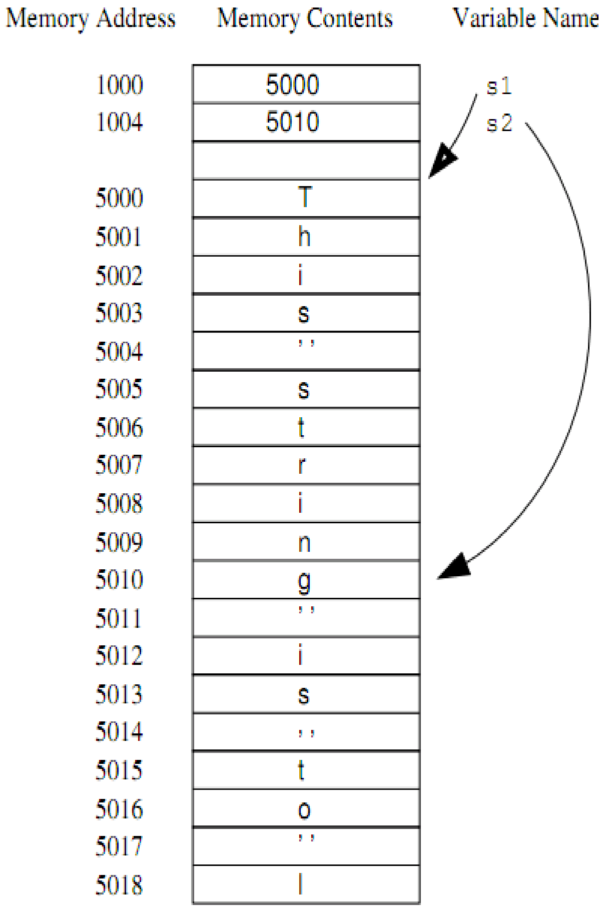
```
char s1[] = “1. string”;
char s2[] = “2. string”;
strcpy(s1, "This string is too long!\n");
```

We copy string of 25 chars to memory which can accept only 9 chars!

It is very probable that we have overwritten s2!

Further more, since we started writing even after s2, we written over some other stuff too.

Compiler will not detect this, and very often it won’t even be detected in runtime (except that program will behave strangely)!



Memory Address	Memory Contents	Variable Name
1000	5000	s1
1004	5010	s2
5000	T	
5001	h	
5002	i	
5003	s	
5004	' '	
5005	s	
5006	t	
5007	r	
5008	i	
5009	n	
5010	g	
5011	' '	
5012	i	
5013	s	
5014	' '	
5015	t	
5016	o	
5017	' '	
5018	l	

String literals

Example 1

```
char* str;
str = "hello";
printf("%s\n", str);
```

Example 2

```
char str[100];
strcpy(str, "hello");
printf("%s\n", str);
```

Same output, but the behavior is very different.

String literals

Example 1

```
char* str;
str = "hello";
printf("%s\n", str);
strcpy(str, "hello");
```

Example 2

```
char str[100];
strcpy(str, "hello");
printf("%s\n", str);
str = "hello";
```

Example 1 causes writing to protected area.
Example 2 will not even compile.

String literals

Example 1

```
const char* str =
  "hello";
printf("%s\n", str);
strcpy(str, "hello");
```

Example 2

```
char str[100];
strcpy(str, "hello");
printf("%s\n", str);
str = "hello";
```

With **const** we ensure that example 1 will be cause an error in compile time.

One more example

```
int main()
{
    char* str;
    str = (char*)malloc(100);
    str = "hello";
    free(str);
    return 0;
}
```

```
int main()
{
    char* str;
    str = (char*)malloc(100);
    strcpy(str, "hello");
    free(str);
    return 0;
}
```

Left example compiles correctly, but report error in runtime. Why?

Strings as function parameters

Just as regular arrays, strings can be passed only “by reference”.

```
void Print1(char* str)
{
    printf("%s", str);
}

void Print2(char* ary, int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%c", ary[i]);
}
```

<string.h> some more important functions

```
char* strcpy(char* s1, const char* s2);
char* strncpy(char* s1, const char* s2, size_t n);

char* strcat(char* s1, const char* s2);
char* strncat(char* s1, const char* s2, size_t n);

int strcmp(const char* s1, const char* s2);
int strncmp(const char* s1, const char* s2, size_t n);

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

Look in C standard for description of these functions.

<string.h>
more functions

```
void* memcpy(void* s1, const void* s2, size_t n);
void* memmove(void* s1, const void* s2, size_t n);

int memcmp(const void* s1, const void* s2, size_t n);

void* memset(void* str, int c, size_t n);
```

Conversion functions

From character string to numbers. <stdlib.h>

```
int atoi(const char* nptr);
long atol(const char* nptr);
long long atoll(const char* nptr);
double atof(const char* nptr);
```

Vice versa? <stdio.h>

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

sprintf(s, “%d”, 5); // s = “5”
```

<ctype.h>

```
int isalnum(int ch);

int isalpha(int ch);

int islower(int ch);

int isupper(int ch);
```