Strings – Dynamic memory management

Basics of Programming 1

G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

18 October, 2023

Content

1 Strings
- Strings

2 Dynamic memory management
- Allocating and releasing memory
- String example
- Smart array example

Chapter 1

Strings

Strings

- In C, text is stored in character arrays with termination sign, called as strings.
- The termination sign is the character with 0 ASCII-code `'\0'`, the null-character.

'S'	'o'	'm'	'e'	' '	't'	'e'	'x'	't'	'\0'

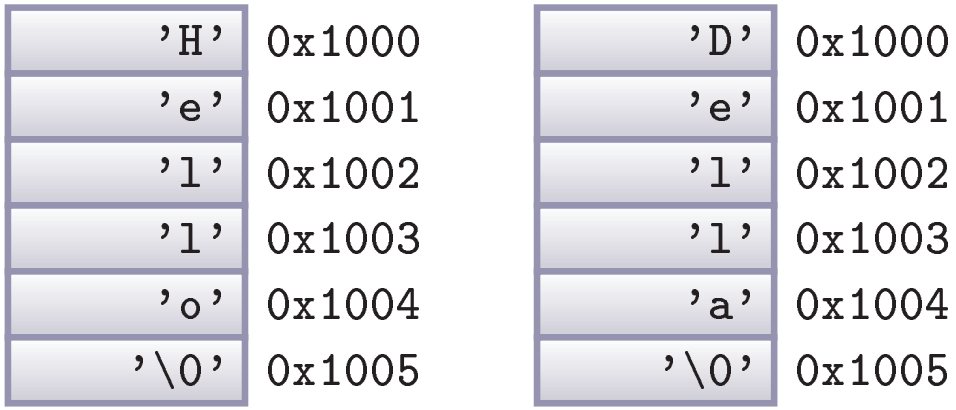
Defining strings as character arrays

- Definition of character array with initialization

```
1 char s[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- The same in a more simple way

```
1 char s[] = "Hello"; /* s array (const.addr 0x1000) */
```



- Elements of s can be accessed with indexing or with pointer-arithmetics

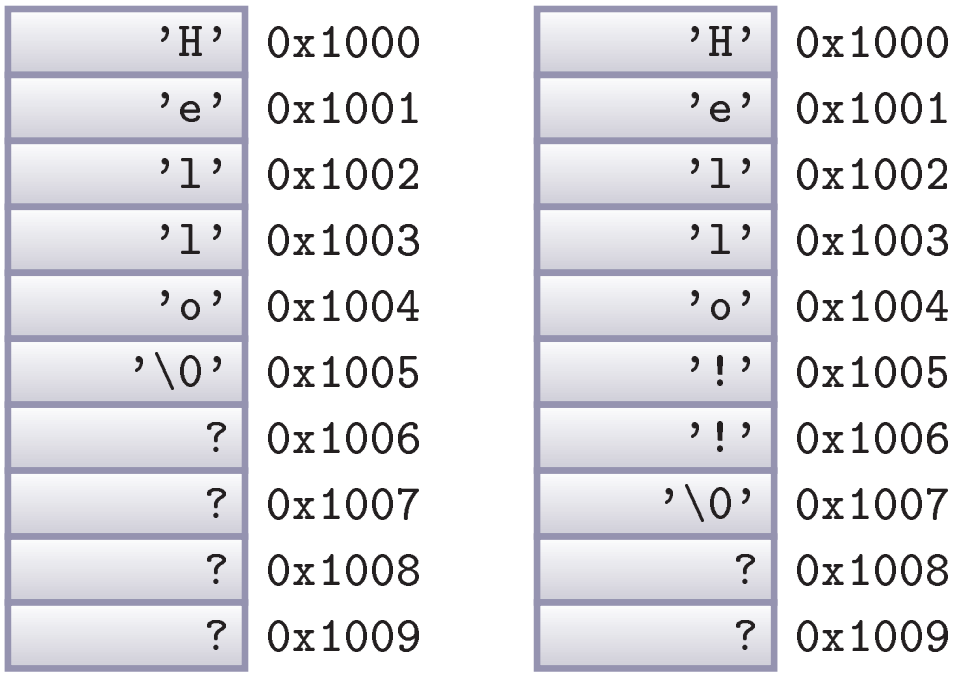
```
1 *s = 'D';    /* s is taken as pointer */
2 s[4] = 'a'; /* s is taken as array */
```

Defining strings as character arrays

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- We can allocate memory for a longer string than needed now, thus we have an overhead.

```c
char s[10] = "Hello"; /* s array, (const.addr. 0x1000) */
```



- Modification:

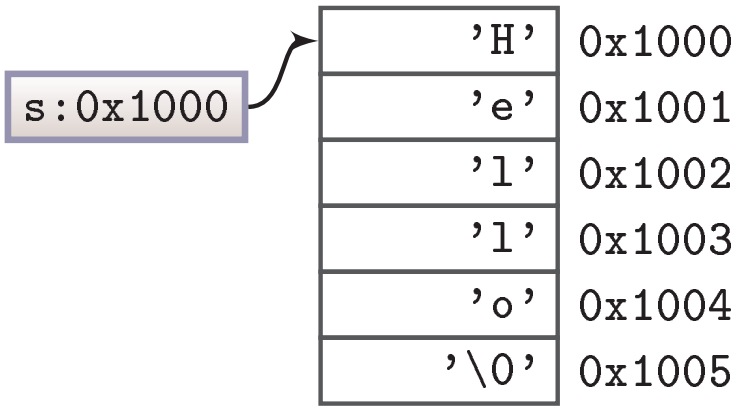
```c
s[5] = s[6] = '!';
s[7] = '\0';          /* must be terminated */
```

Defining strings as character arrays

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Defining a constant character array and a pointer pointing to it, with initialization.

```
1  char *s = "Hello"; /* s pointer */
```



- Here the so-called static part of memory is used to store the string. The content of the string cannot be changed.
- We can modify value of s, however it is not recommended, because this stores the address of our string.

Remarks

- Character or text?

```
1 char s[] = "A"; /* two bytes: {'A', '\0'} */
2 char c = 'A';   /* one byte: 'A' */
```

- A text can be empty, but there is no empty character

```
1 char s[] = "";  /* one byte: {'\0'} */
2 char c = '';    /* ERROR, this is not possible */
```

Reading and displaying strings

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Strings are read and displayed with format code %s

```
1 char s[100] = "Hello";
2 printf("%s\n", s);
3 printf("Enter a word not longer than 99 characters: ");
4 scanf("%s", s);
5 printf("%s\n", s);
```

```
Hello
Enter a word not longer than 99 characters:  ghostbusters
ghostbusters
```

- Why don't we have to pass the size for printf?
- Why don't we need the & in the scanf function?

Reading and displaying strings

- scanf reads only until the first whitespace character. To read text consisting of several words, use the gets function:

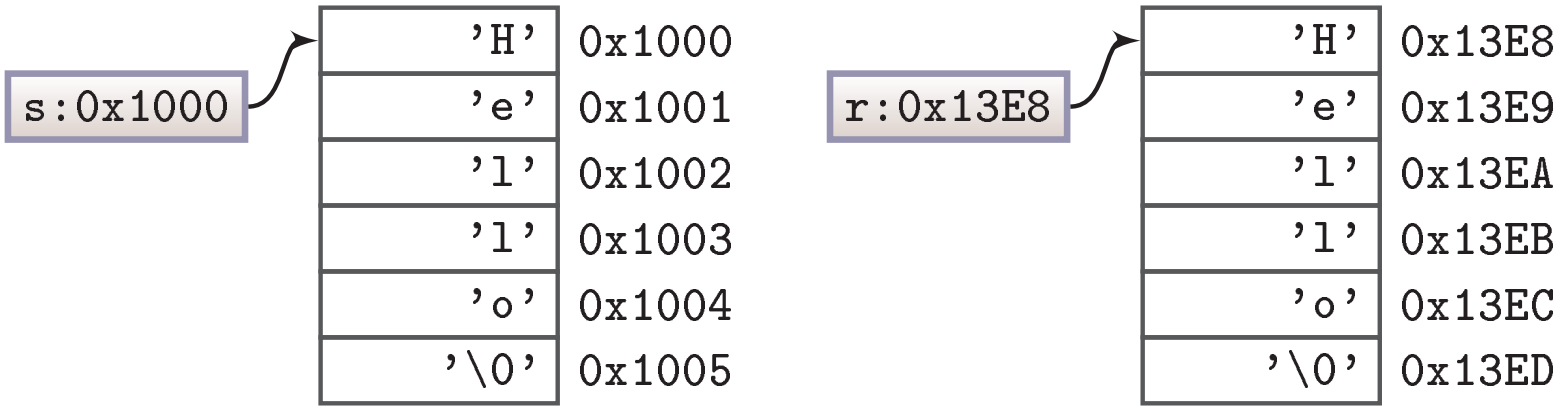
```
1 char s[100];
2 printf("Enter a text - max. 99 characters long: ");
3 gets(s);
4 printf("%s\n", s);
```

```
Enter a text - max. 99 characters long: this is text
this is text
```

Strings – typical mistakes

- Typical mistake: comparison of strings

```
1  char *s = "Hello";
2  char *r = "Hello";
3  if (s == r) /* what do we compare? */
4    ...
```



- The same mistake happens if defined as arrays

String functions

- Comparing strings
- the result
  - positive, if s1 stands after s2 alphabetically
  - 0, if they are identical
  - negative, if s1 stands before s2 alphabetically

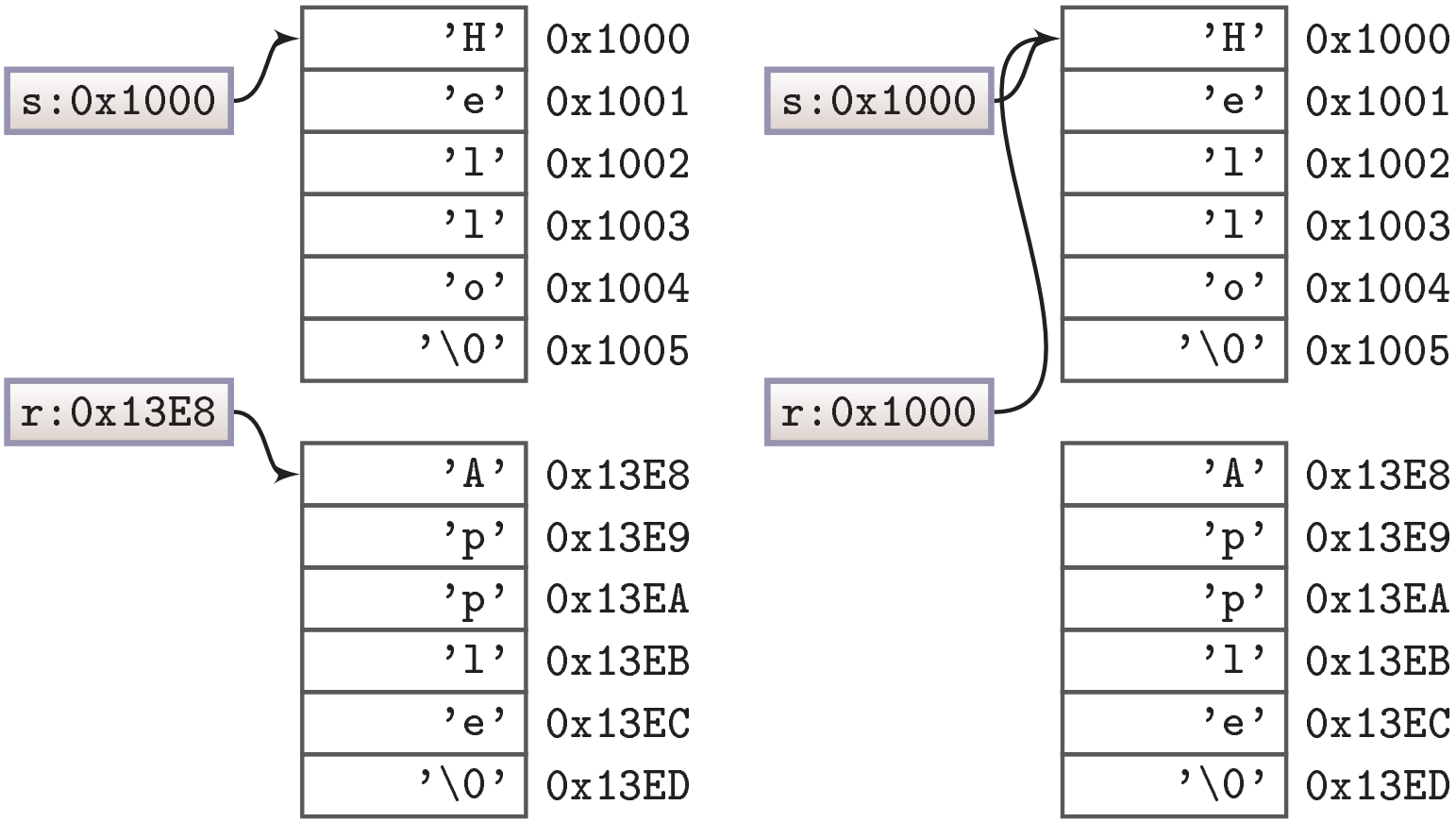
```
1  int strcmp(char *s1, char *s2) /* pointer-notation */
2  {
3    while (*s1 != '\0' && *s1 == *s2)
4    {
5      s1++;
6      s2++;
7    }
8    return *s1 - *s2;
9  }
```

- Is it a problem, that s1 and s2 was changed during the check?
- Remark: In the solution we made use of the information that \0 is the 0 ASCII-code character!

Strings – typical mistakes

- Typical mistake: string copy attempt

```
1 char *s = "Hello";
2 char *r = "Apple";
3 r = s; /* what do we copy */
```



Other string functions

- `#include <string.h>`
  - `strlen` length of string (without `\0`)
  - `strcmp` comparing strings
  - `strcpy` copying string
  - `strcat` concatenating strings
  - `strchr` search for character in string
  - `strstr` search for string in string
- `strcpy` and `strcat` functions copy 'without thinking', the user must provide the allocated memory for the resulting string!

Chapter 2

Dynamic memory management

Dynamic memory management

- Let's read integer numbers and print them in a reversed order!
- The user will enter the number of the numbers to be read (count).
- Let's not use more memory than needed!

1. We read the count (`n`)
2. We ask memory from the operating system for storing `n` integer numbers
3. We read and store the numbers, and print them in reversed order
4. We give back (hand over) the reserved memory place to the operating system

Example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  int n, i;
2  int *p;
3
4  printf("How many numbers? ");
5  scanf("%d", &n);
6  p = (int*)malloc(n*sizeof(int));
7  if (p == NULL) return;
8
9  printf("Enter %d numbers:\n", n);
10 for (i = 0; i < n; ++i)
11   scanf("%d", &p[i]);
12
13 printf("Reversed:\n");
14 for (i = 0; i < n; ++i)
15   printf("%d ", p[n-i-1]);
16
17 free(p);
18 p = NULL;
```

link

p:0x0000

```
How many numbers?  5
Enter 5 numbers!
1 4 2 5 8
Reversed:
8 5 2 4 1
```

The malloc and free functions – <stdlib.h>

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void *malloc(size_t size);
```

- Allocates memory block of size bytes, and the address of the block is returned as void* type value
- The returned void* "is only an address", we cannot de-refer it. We can use it only if converted (eg. to int*).

```
1 int *p; /* starting address of int array */
2 /* Memory allocation for 5 int */
3 p = (int *)malloc(5*sizeof(int));
```

- If there is not enough memory avaible, the return value is NULL. This must be checked always.

```
1 if (p != NULL)
2 {
3   /* using memory, and releasing it */
4 }
```

The malloc and free functions – <stdlib.h>

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void free(void *p);
```

- Releases the memory block starting at address p
- The size of the block is not needed, the op.system knows it (it stored it just before the memory block, this is the reason for calling it with the starting address)
- free(NULL) is allowed (does not perform anything), so we can do this:

```
1 int *p = (int *)malloc(5*sizeof(int));
2 if (p != NULL)
3 {
4   /* using it */
5 }
6 free(p);  /* works even if NULL */
7 p = NULL; /* a useful step to remember */
```

- As a nullpointer points to nowhere, a good practice is to set a pointer to NULL after usage, so we can see it is not in use.

malloc – free

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- malloc and free go hand-in-hand,
- for each malloc there is a free

```
1 char *WiFi = (char *)malloc(20*sizeof(char));
2 int *Lunch = (int *)malloc(23*sizeof(int));
3 ...
4 free(WiFi);
5 free(Lunch);
```

- If we don't relelase the memory block, memory leak occurs
- Good practice rules:
  - Release in the same function where allocated
  - Don't modify the pointer that was returned by malloc, if possible, use the same pointer for releasing
- If we cannot keep these rules, make a note in the code about this (comment)

The calloc function – <stdlib.h>


```
void *calloc(size_t num, size_t size);
```

- Allocates memory block for storing `num` pieces of elements, each with `size` size, the allocated memory block is cleared (set to zero), and the address of the block is returned as `void*` type value
- Usage is almost the same as of `malloc`, except this performs the calculation `num*size`, and removes the garbage.
- The allocated block must be released in the same way: with `free`.

```
1 int *p = (int *)calloc(5, sizeof(int));
2 if (p != NULL)
3 {
4   /* using it */
5 }
6 free(p);
```

The realloc function – <stdlib.h>


```
void *realloc(void *memblock, size_t size);
```

- resizes to size bytes a memory block that was earlier allocated
- the new size can be smallero r larger than the earlier size
- if needed, the earlier content is copied to the new place, the elements are not initialized
- its return value is the starting address of the new place

```
1 int *p = (int *)malloc(3*sizeof(int));
2 p[0] = p[1] = p[2] = 8;
3 p = realloc(p, 5*sizeof(int));
4 p[3] = p[4] = 8;
5 ...
6 free(p);
```

Example

- Let's create a function that concatenates the strings received as parameters. The function should allocate memory for the resulting string, and should return with its address.

1. The function determines the length of the two strings,
2. allocates memory for the result,
3. copies the first string into the result string,
4. copies the second string after it.

- Of course, this function cannot release the allocated memory, this must be done in the calling program segment

Example


```c
/* concatenate -- concatenating two strings
  Dynamic allocation, returning with address.
*/
char *concatenate(char *s1, char *s2){
    size_t l1 = strlen(s1);
    size_t l2 = strlen(s2);
    char *s = (char *)malloc((l1+l2+1)*sizeof(char));
    if (s != NULL)     {
        strcpy(s, s1);
        strcpy(s+l1, s2); /* or strcat(s, s2) */
    }
    return s;
}
```

link

Example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Usage of the function

```c
char word1[] = "partner", word2[] = "ship";

char *res1 = concatenate(word1, word2);
char *res2 = concatenate(word2, word1);
res2[0] = 'w';

printf("%s\n%s", res1, res2);

/* The function did allocate memory, release it! */
free(res1);
free(res2);
```

link

```
partnership
whippartner
```

Example: Create smart array object

- Let us create a double dynamic arary that knows its size, and can be handled through functions

1. int length(S_array sarr)
2. void print(S_array sarr)
3. void push back(S_array* sarr,double what)
4. int pop(S_array* sarr)
5. double get_element(S_array sarr, int index)
6. double* set_element(S_array sarr, int index)
7. void delete(S_array sarr)

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct {
  double* Array;
  int size;
}S_array;
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  int length(S_array sarr) {
2    return sarr.size;
3  }
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  void print(S_array sarr) {
2    for (int i = 0; i < sarr.size; i++)
3      printf("%f\t", sarr.Array[i]);
4  }
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  void push_back(S_array* arr, double what) {
2    int i = (*arr).size;
3    (*arr).Array = (double*)realloc((*arr).Array, (i + 1)
4      * sizeof(double));
5    if ((*arr).Array == NULL)
6      exit(-1);
7    (*arr).Array[i] = what;
8    (*arr).size++;
9  }
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  void pop(S_array* arr) {
2    (*arr).size--;
3  }
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  double get_element(S_array sarr, int idx) {
2    return sarr.Array[idx];
3  }
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  double* set_element(S_array sarr, int idx) {
2    return &sarr.Array[idx];
3  }
```

link

Smart array example


```
1  void delete(S_array sarr) {
2    free(sarr.Array);
3  }
```

link

Smart array example

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  int main() {
2    S_array smart = { NULL,0 };
3    push_back(&smart, 3.0);
4    push_back(&smart, 13);
5    push_back(&smart, -12.3);
6    print(smart);
7    pop(&smart);
8    print(smart);
9    *set_element(smart, 0) = -100;
10   print(smart);
11   delete(smart);}
```

link

```
3.000000 13.000000 -12.300000
3.000000 13.000000
-100.000000 13.000000
```

Thank you for your attention.