Pointers – Strings

Basics of Programming 1

DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

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

15 October, 2025

Content

Chapter 1

Pointers

Fundamental Theorem of Software Engineering (FTSE)

"We can solve any problem by introducing an extra level of indirection."
Andrew Koenig

Where are the variables?

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

Let's write a program that lists the address and value of variables

```
int a = 2;
double b = 8.0;
printf("address of a: %p, its value: %d\n", &a, a);
printf("address of b: %p, its value: %f\n", &b, b);
```

```
address of a: 0x7fffa3a4225c, its value: 2
address of b: 0x7fffa3a42250, its value: 8.000000
```

- address of variable: starting address of "memory block" containing the variable, expressed in bytes
- with the address-of operator we can create address of any variables[1] like this `&<reference>`

[1] more precisely left-values

The pointer type

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The pointer type is for storing memory addresses

Declaration of pointer

<pointed type> * <identifier>;

```
int*    p;  /*  p stores the address of one int data */
double* q;  /*  q stores the address of one double data */
char*   r;  /*  r stores the address of one char data */
```

it is the same, even if arranged in a different way

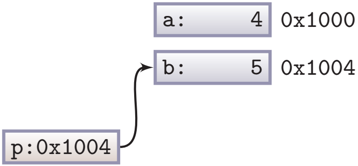
```
int     *p;  /*  p stores the address of one int data */
double  *q;  /*  q stores the address of one double data */
char    *r;  /*  r stores the address of one char data */
```

Operator of indirection

- If pointer p stores the address of variable a, then p ”points to a”
- If p points to a, then variable a can be accessed as *p. Here * is the operator of indirection (dereference operator).

```c
int a, b;
int *p; /* int pointer */

a = 2;
b = 3;
p = &a; /* p points to a */
*p = 4; /* a = 4 */
p = &b; /* p points to b */
*p = 5; /* b = 5 */
```



Address-of and indirection – summary

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

operator	operation	description
&	address-of	assigns its address to the variable
*	indirection	assigns variable to the address

- Interpreting declaration: type of *p is int

```
int *p;        /* get used to this version */
```

- Multiple declaration: type of a, *p and *q is int

```
int a, *p, *q; /* at least because of this */
```

Application – Function for exchanging two variables

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
void xchg(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void xchgp(int *px, int *py) {
  int tmp = *px;
  *px = *py;
  *py = tmp;
}

int main(void) {
  int a = 2, b = 3;
  xchg(a, b);
/* NO exchange */
  xchgp(&a, &b);/* exchange */
  return 0;
}
```

Application – returning value as parameter

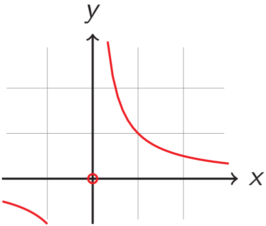
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- If a function has to calculate several values, then...

  ...we can use structures, but sometimes this seems rather unnecessary.
  Instead...

```
int inverse(double x, double *py)
{
  if (abs(x) < 1e-10) return 0;
  *py = 1.0 / x;
  return 1;
}
```

link




```
double y;          /* memory allocation for result */
if (inverse(5.0, &y) == 1)
  printf("Reciprocal of %f is %f\n", 5.0, y);
else
  printf("Reciprocal does not exist");
```

link

Application – return values as parameters

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Now we understand what this means

```c
int n, p;
/* return value as parameter */
scanf("%d%d", &n, &p); /* we pass the addresses */
```

Remarks:

- What is the use of having different pointer types for different types?
- Type = set of values + operations
- Obviously set of values is the same for all pointers (unsigned integer addresses)
- Operations are different!
- The operator of indirection (∗)
  - makes `int` from `int` pointer
  - makes `char` from `char` pointer
- Other differences are detailed in pointer-arithmetics...

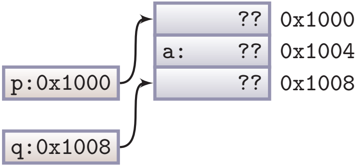
Pointer-arithmetics

If p and q are pointers of the same type, then

expr.	type	meaning
p+1	pointer	points to the next element
p-1	pointer	points to the previous element
q-p	integer number	number of elements between two addresses

```c
int a, *p, *q;

p = &a;
p = p-1;
q = p+2;
printf("%d", q-p);
```




```
2
```

- At pointer-arithmetic operaitons addresses are ”measured” in the representation size of the pointed type, and not in bytes.[2]

[2]In this example we assume that size of int is 4 bytes

Pointer-arithmetic

- In the above example pointer-arithmetic is strange, as we don't know what is before or after variable a in the memory.
- This operation is meaningful, when we have variables of the same type, stored in the memory one afte the other.
- This is the case for arrays.

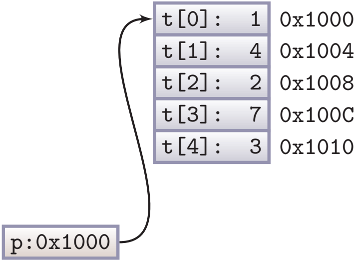
Pointers and arrays

- Traversing an array can be done with pointer-arithmetics.

```c
int t[5] = {1,4,2,7,3};
int *p, i;

p = &t[0];
for (i = 0; i < 5; ++i)
  printf("%d ", *(p+i));
```

```
1 4 2 7 3
```



- In this example `*(p+i)` is the same as `t[i]`, because `p` points to the beginning of array `t`

Pointers and arrays

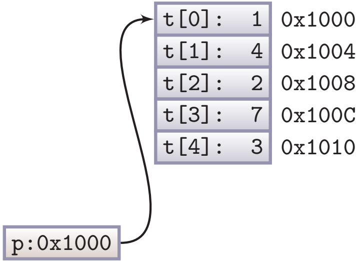
DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Pointers can be taken as arrays, this means they can be indexed.
  By definition `p[i]` is identical to `*(p+i)`

```c
int t[5] = {1,4,2,7,3};
int *p, i;

p = &t[0];
for (i = 0; i < 5; ++i)
  printf("%d ", p[i]);
```

```
1 4 2 7 3
```



- In this example `p[i]` is the same as `t[i]`, because `p` points to the beginning of array `t`

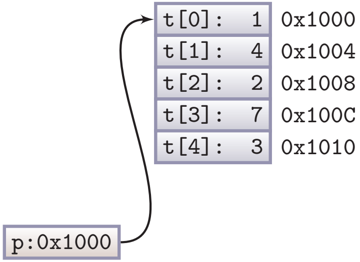
Pointers and arrays

- Arrays can be taken as pointers.
  The identifier (name) of array is the starting address of the array, in other words the value of expression `t` is `&t[0]`

```c
int t[5] = {1,4,2,7,3};
int *p, i;

p = t; /* &t[0] */
for (i = 0; i < 5; ++i)
  printf("%d ", p[i]);
```

```
1 4 2 7 3
```



- Pointer-arithmetics work for arrays too:
  `t+i` is identical to `&t[i]`

Pointers and arrays – summary

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Pointer can be taken as array, and array as a pointer.
- index operator is only a notation
  the compiler will always replace `a[i]` with `*(a+i)`,
  both if *a* is pointer, and also if *a* is array.
- Differences:
  - Elements of array have allocated space in memory (variables). No allocated elements belong to the pointer.
  - Starting address of array is constant, it cannot be changed. Pointer is a variable, the address stored in it can be modified.

```c
int array[5] = {1, 3, 2, 4, 7};
int *p = array;

/* the elements can be accessed via p and a */
p[0] = 2;                 array[0] = 2;
*p = 2;                   *array = 2;

/* p can be changed   array CANNOT */
p = p+1; /* ok */         array = array + 1; /* ERROR */
```

Passing arrays to functions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Let's use a function to determine the first negative element of array!
- Passing an array:
  - Address of first element `double*`
  - Size of the array `typedef unsigned int size_t`[3]

```
double first_negative(double *array, size_t size)
{
  size_t i;
  for (i = 0; i < size; ++i)  /* for each elems. */
    if (array[i] < 0.0)
      return array[i];

  return 0; /* all are non-negative */
}
```

link

```
double myarray[3] = {3.0, 1.0, -2.0};
double neg = first_negative(myarray, 3);
```

link

[3]defined in `stdio.h`

Passing arrays to functions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- To distinguish arrays and pointers in the parameter list, we can use the array-notation when passing an array.

```
double first_negative(double array[], size_t size)
                    /*  (double *array,  size_t size)  */
{
  ...
}
```

- In the formal parameter list `double a[]` is identical to `double *a`.
- In the formal parameter list we can use only empty `[]`, and size should be passed as a separate parameter!

Passing arrays to functions

- Let's use a function to determine the first negative element of array!
- The return value should be the address of the element found.

```
double *first_negative(double *array, size_t size)
{
  size_t i;
  for (i = 0; i < size; ++i)  /* for each elems. */
    if (array[i] < 0.0)
      return &array[i];

  return NULL; /* all are non-negative */
}
```

link

Null pointer

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- The null pointer (`NULL`)
  - It stores the `0x0000` address
  - Agreed that it "points to nowhere"

Chapter 2

Strings

Strings

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- 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

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

- The same in a more simple way

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

'H'	0x1000
'e'	0x1001
'l'	0x1002
'l'	0x1003
'o'	0x1004
'\0'	0x1005

'D'	0x1000
'e'	0x1001
'l'	0x1002
'l'	0x1003
'a'	0x1004
'\0'	0x1005

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

```
*s = 'D';    /* s is taken as pointer */
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.

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

'H'	0x1000	'H'	0x1000
'e'	0x1001	'e'	0x1001
'l'	0x1002	'l'	0x1002
'l'	0x1003	'l'	0x1003
'o'	0x1004	'o'	0x1004
'\0'	0x1005	'!'	0x1005
?	0x1006	'!'	0x1006
?	0x1007	'\0'	0x1007
?	0x1008	?	0x1008
?	0x1009	?	0x1009

- Modification:

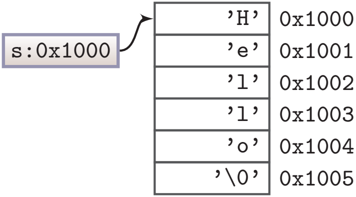
```
s[5] = s[6] = '!';
s[7] = '\0';          /* must be terminated */
```

Defining strings as character arrays

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

```
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

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Character or text?

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

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

```
char s[] = "";  /* one byte: {'\0'} */
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

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

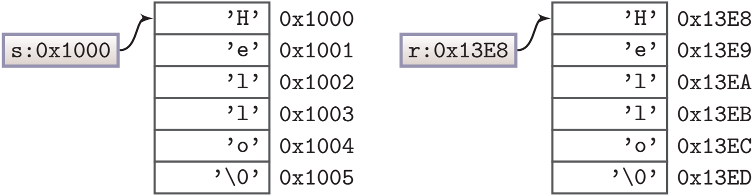
```
char s[100];
printf("Enter a text - max. 99 characters long: ");
gets(s);
printf("%s\n", s);
```

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

Strings – typical mistakes

- Typical mistake: comparison of strings

```c
char *s = "Hello";
char *r = "Hello";
if (s == r) /* what do we compare? */
  ...
```



- The same mistake happens if defined as arrays

String functions

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

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

```c
int strcmp(char *s1, char *s2) /* pointer-notation */
{
  while (*s1 != '\0' && *s1 == *s2)
  {
    s1++;
    s2++;
  }
  return *s1 - *s2;
}
```

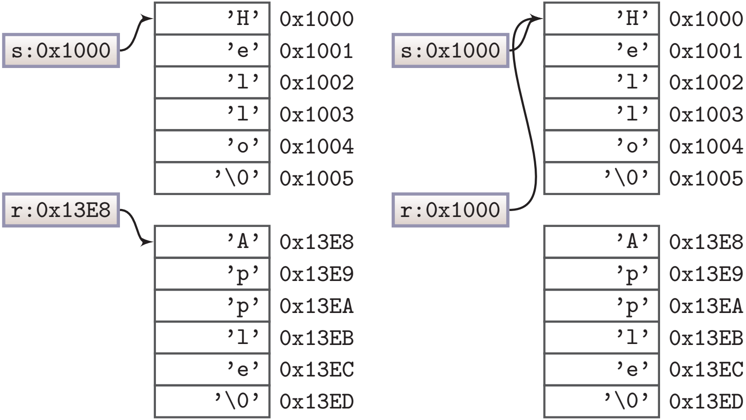
- Is it a problem, that s1 and s2 was changed during the check?
- Remark: In the solution we made use of the information that \0 is the 0 ASCII-code character!

Strings – typical mistakes

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Typical mistake: string copy attempt

```
char *s = "Hello";
char *r = "Apple";
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!

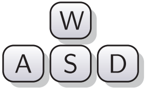
Chapter 3

The enumerated type

The enumerated type – Motivation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- We are writing a game, in which the user can control direction of the player with 4 keys.



- As the input from user needs to be read (checked) frequently, we create a `read_direction()` function for this task.
- This function reads from the keyboard and returns the direction to the calling program segment.
- What type should the function return with?

The enumerated type – Motivation

- Idea Nr. 1: Let's return with the key pressed. ('a','s','w','d'):

```
char read_direction(void)
{
  char ch;
  scanf("%c", &ch);
  return ch;
}
```

link

- Problems:
  - We have to decode characters into directions many times at different parts of the source code.
  - If we change to use the arrow keys $\leftarrow \downarrow \uparrow \rightarrow$ for control, we have to modify the source code a thousand time and place.
- Solution:
  - We have to decode in place (inside the function), and should return with direction.
  - But how can we do that?

The enumerated type – Motivation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Idea Nr. 2: Let's return with `int` values 0,1,2,3:

'a'	0	←
'w'	1	↑
'd'	2	→
's'	3	↓

```c
int read_direction(void) {
  char ch;
  scanf("%c", &ch);
  switch (ch) {
  case 'a': return 0; /* left */
  case 'w': return 1; /* up */
  case 'd': return 2; /* right */
  case 's': return 3; /* down */
  }
  return 0; /* default is left :) */
}
```

- Problem:
  - In other parts of the program we have to use numbers 0-3 for the directions, so the programmer must remember the number-direction assignments.

The enumerated type – Motivation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- We need a type named `direction`, that can store `LEFT`, `RIGHT`, `UP`, `DOWN` values.
- We can do such thing in C!

  Declaration of the appropriate enumerated type (`enum`):

```c
enum direction {LEFT, RIGHT, UP, DOWN};
```

- How to use the type:

```c
enum direction d;
d = LEFT;
```

The enumerated type – Motivation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- The final solution with the new type

```
enum direction {LEFT, RIGHT, UP, DOWN};
typedef enum direction direction; /* simplification */

direction read_direction(void)
{
  char ch;
  scanf("%c", &ch);
  switch (ch)
  {
  case 'a': return LEFT;
  case 'w': return UP;
  case 'd': return RIGHT;
  case 's': return DOWN;
  }
  return LEFT;
}
```

link

The enumerated type – Motivation

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Usage of the function:

```
direction d = read_direction();
if (d == RIGHT)
  printf("You were eaten by a tiger\n");
```

link

- Without the enumerated type, it would look like this:

```
int d = read_direction();
if (d == 2) /* "magic" constant, what does it mean? */
  printf("You were eaten by a tiger\n");
```

link

- The enumerated type...
  - replaces "magic constants" with informative code,
  - focuses on content instead of representation,
  - allows a higher level programming.

The enumerated type – Definition

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

The enumerated (enum) type

Joins into one type integer type constants referenced by symbolic names.

```
enum [&lt;enumeration label&gt;]opt
{ <enumeration list> }
[<variable identifiers>]opt;
```

```
enum direction {LEFT, RIGHT, UP, DOWN} dir1, dir2;
```

enum examples

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
enum month {
  JAN,  /*  0 */
  FEB,  /*  1 */
  MAR,  /*  2 */
  APR,  /*  3 */
  MAY,  /*  4 */
  JUNE, /*  5 */
  JULY, /*  6 */
  AUG,  /*  7 */
  SEPT, /*  8 */
  OCT,  /*  9 */
  NOV,  /* 10 */
  DEC   /* 11 */
};

enum month m=OCT; /*9*/
```

```
enum {
  RED,        /*  0 */
  BLUE = 3,   /*  3 */
  GREEN,      /*  4 */
  YELLOW,     /*  5 */
  GRAY = 10   /* 10 */
} c;

c = GREEN;
printf("c: %d\n", c);
```

```
c:  4
```

Thank you for your attention.