Low-level programming

Lecture 4

Arrays

Marcin Radom, Ph.D.

Institute of Computing Science
Faculty of Computing and Telecommunication

One dimensional arrays (vectors)

Theory and examples

Arrays

- They are data structures which join together many values of the same type.
- **Array (table)** – chain of element of the same type, accessible under one name.
- In C language there is direct connection between pointers and array. Every operation which can be done using index of an array can be performed using pointers (and usually faster).
- **One-dimensional array declaration:**

```
type  identifier [ size ] ;
```

type **: numbers, signs, pointers or structure type**

size **: constant variable greater than zero**

(in *gcc* - integer)

Arrays

- **Example:**

```
int TAB[5];     // 0, 1, 2, 3, 4
```

  - In the example we define array *TAB* having size 5, in other words a chain of cells: *TAB[0], TAB[1], …, TAB[4]*
  - Notation *TAB[i]* means *i–th* array element.
  - Name of the array: *TAB* represents the location of its first element (it is however NOT a variable – it cannot be assigned with a new address).

```
const int sum = 50, multi = 120;

float RESULTS[ 2 * (sum + multi )]; // since C99
```

Arrays

- Calling single array element is done by index – it has range from zero to the size of the array minus 1, e.g.:

```
int table[10];       // 10-th elements array
int i;
i = TAB[3]; // TAB + 3 * sizeof int
RESULTS[i + 2] = RESULTS [i] + RESULTS [i + 1];
```

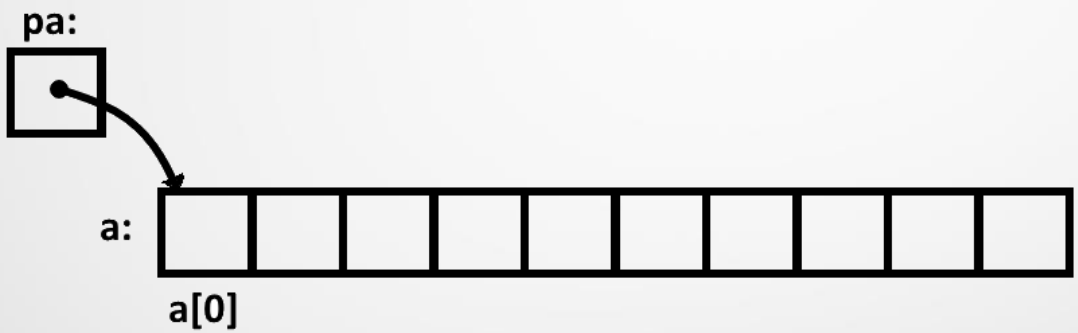
- In C language a common error occurs when index value exceeds the size of the array:

```
double TAB_DATA [ 7 ];
for (int ix = 0; ix < 12; ++ix ) // exceeding the range
    TAB_DATA [ ix ] = 48.74 ;
// program will compile, however there will be an error
// when the program reaches this code (program will be
// terminated by the operating system).
```

Pointer to the array element

- Example:

```
int a[10];
int *pa;   // integer pointer
pa = &a[0] // reads: pointer pa points to ADDRESS (&)
           // of the first element (index [0] ) of array a
```



The following code will copy content of *a[0]* into *x* (using pointer):

```
int x = *pa;
```

- If pointer ***pa*** points to the first element of an array **a** (a[0]), then: **(pa+1)** points to the next element of the array **a**: element a[1].

Pointer to the array element

- ***pa+i*** refers to the *i-th* element after ***pa*** (in the previous example: to the *i-th* element of array **a**, if **pa** points to the first element: a[0]), ***pa-i*** refers to *i-th* element before ***pa***.
- If one wants to call a value of the array element (for example with index [1]), then:

```
*(pa+1); // refers to the VALUE of table element (if pa points to array)
```

- ***pa+i*** is an address of *a[i]*, ****(pa+i)*** is the content (value) of *a[i]*
- This theory relates to every C array, no matter the type or size.
- **Value of the array variable (the name of the array) is by definition the address of the first element of the array:**

```
int t[10];
int *ptr;
ptr = t; // equal to ptr = &t[0]; then ptr points to element t[0];
```

- Calling value ***t[i]*** is equivalent of ****(ptr+i)***. In C language statement *t[i]* is transformed into **(t+i)*.

Arrays

- **&t[i]** – address of *i-th* element of an array, same as: **ptr+i** – this is also the address of element *t[i]* in previous example.
- Between array name and its pointer there is important difference. Pointer is A VARIABLE, so:

```
int some_table[100];
int *ptr = some_table;
ptr++;    // pointer movement into 2nd array element (if ptr pointed to 1st)
(*ptr)++; // not the 2nd element is incremented
          // () are necessary (operators priorities).
```

- **Array name IS NOT A NORMAL POINTER VARIABLE, so**:

```
some_table ++;    // ERROR
some_table = ptr; // ERROR
```

- Unary operators ***** and **&** bind stronger than arithmetic operators - they have higher priority, e.g.:

```
y = *ptr + 1;      // first a value from ptr address is taken, then this
                   // value is incremented than 1, then the result of
                   // addition is stored in y
```

Arrays

- Operations defined by unary operators ***** and **++** are being calculated from **right to left** (right-side joined with the same priority), so:

```
++*p;     // no parenthesis necessary, also: *p += 1 and *p = *p + 1
(*p)++    // same as above but with (necessary) parenthesis
```

- Pointers are normal variables, so they can be used for example like this (lets assume ***iq*** and ***ip*** are **int** type):

```
iq = ip;
```

The above line copies ***ip*** to ***iq***, in other words now ***iq*** points to the same address as ***ip***.

Arrays

- Example:

```
char buffer[8];
char *pp;
pp = buffer;              /* equivalent
pp = & buffer[0];            statements */
```

- *(TAB + 4) **and** TAB [4] **are equivalent**

```
const int ele = 25;
short    TS[ele];
int      TI[ele];
double   TD[ele];
for (int i = 0; i < ele; ++i ) {
     *(TS + i) = 1;
     *(TI + i) = 1;
     *(TD + i) = 1.0;
}
```

One-dimensional array initialization

- Array may be initialized by the assignment operator = and by placing comma separated values in curly brackets:

  type **identifier [size] = { list_of_elements };**

  e.g.:

```
int days_of_year[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

- In such a case the number of array elements does not have to be provided by the programmer – the compiler will calculate it by counting elements in brackets.
- If the number of elements in the brackets is lower than given size of the array, all others elements (not initialized in { }) will be given 0 value.
- **Example:**

```
long MM[3] = { 154835L, 337782L, 0L }
/* MM[0] == 154835,
   MM[1] == 33782,
   MM[2] == 0 */
```

One-dimensional array initialization

- **Examples:**

```
const float F1 = 3.5E7F, F2 = 33.E8F;
float DATA[ ] = { F1 + F2, F1 / (F1 - F2)};
/* array DATA has two elements */

double PQ[150] = { 1.5, 3.8, 3.8, 2.7 };
/* PQ[0] == 1.5, PQ[1] == 3.8, PQ[2] == 3.8,
   PQ[3] == 2.7, all the rest == 0 */

double Empty[1200] = {0};          // zero-initialization

char NN[5] = { "alfa" };  // NN[4] == \0
char MM[4] = { "beta" };  // error, table too small
```

One-dimensional arrays initialization

- **Example:**

```
int TabCount [ 125 ];
for (int i = 0; i < 125; ++i)
     scanf("%d", &TabCount[i]);

double Real [ 12 ];
for (i = 0; i < 12; ++i)
     scanf("%lf", &Real[ i ]);
```

- **Obtaining array size:**

```
long A[5];
int area, element, elements;
area      = sizeof A;                 // == 20 (5 elements 4 bytes each)
element   = sizeof A[0];              // == 4 (1 element = 4 bytes)
elements  = sizeof A / sizeof A[0];   // == 5 elements (20 / 4)
```

One-dimensional arrays

- Character arrays can be initialized differently – by using a string constants, e.g.:

```
char pattern[] = "yes";      // array size 3 + 1 ( /0 )
```

different form, same result:

```
char pattern[] = {'y', 'e', 's', '\0'};
```

- **Extensions in gcc:**
  - Since **C99** (as **C89** standard extension) elements of the array can be given in mixed order if followed with the index:

```
[index] = element_value
```

```
int T[5] = { 3, [2] = 5, [4] = 2 };
/* T[0] == 3, T[1] == 0, T[2] == 5, T[3] == 0, T[4] == 2 */
for (int i = 0; i < 5; ++i)
     printf("\t%d", T[i]); //  Writes down: 3  0  5  0  2
```

One-dimensional arrays – example

- **Example** – program reads elements for one dimensional array, then writes them down in order: first the ones on even index, then with odd index (*A[i]* = *value*)

```
int main() {
      double Tab[100];      // table of numbers
      int n, i;             // numbers total, counter
      n = 101;              // wrong initial value
      while (n < 1 || n > 100) {
           printf("Enter number of elements [1-100] : ");
           scanf("%d", &n);
      }
      for (i = 0; i < n; ++i) {
           printf("Enter value A[%d] : ", i);
           scanf("%lf", &Tab[i]);
      }
      printf("\nEven index elements: n"); // even header
      for (i = 0; i < n; i += 2)
           printf("A[%d] = %.3lf\n", i, Tab[i]);
      printf("\nOdd index elements.\n"); // odd header
      for (i = 1; i < n; i += 2)
           printf("A[%d] = %.3lf\n", i, Tab[i]);
      printf("\n");
      return 0;
}
```

One-dimensional arrays – addresses arithmetic

- Example: *p* is a pointer to some array element:
  - ***p++*** - points to next element of an array
  - ***p += i*** – points to the element places *i* positions ahead of a current *p* address
- Pointers can be compared with zero and also assigned the zero value.
- Usually instead of zero we use symbolic constant **NULL** (defined in **<stdio.h>**), to stress that we are using pointers syntax:

```
if( p != 0 ) { … } equivalent to    if ( p != NULL ) { … }
```

- With some restriction pointers can be compared with operators <, >, <=, ==, != and >=, i.e.
  - if *p* and *q* points to some elements of an array then

    ```
    p > q
    ```

    is true, if *p* points to the further positioned element than *q*.
- Any pointer can be always compared with 0.

One dimensional arrays – addresses arithmetic

- Pointer + integer value **p + q** signifies address of the *n-th* element from the place pointed by *p* (no matter the type of pointer). The real size (in bytes) of this step is calculated using pointer type, e.g., variables type double given *n* = 8 bytes.
- Lets assume that *p* points to some element of array type int. For example: *p = &table[o]*
  - Then *p + 5* moves forward by 5 elements ahead, on the 6-th element of the array (table[5]).
  - Counting in bytes: *p + 5* moves the pointed address 20 bytes ahead (int is coded in 4 bytes, so 4*5 is 20).
- Subtraction of pointers is also legal. If *p* and *q* points to the elements of the same array and *p < q* then *q – p + 1* is the number of elements BETWEEN *p* and *q*.
- Example of a function counting characters in array:

```
/* strlen: return length of string s */
int strlen(char *s) {
      char *p = s;  // p points to first element of an array
      while (*p != '\0')
            p++;
      return p - s;
}
```

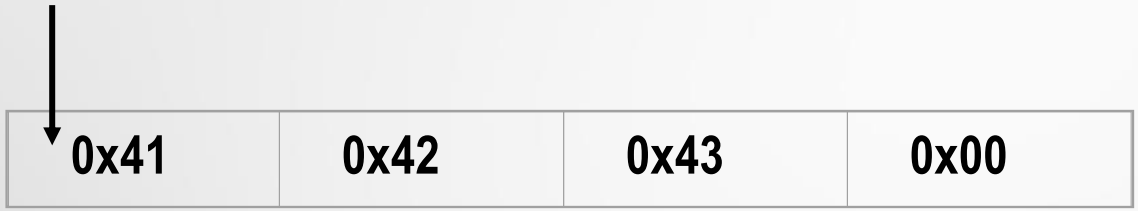
Arrays and strings processing

Functions from <**string.h**>

Arrays – string processing

- String constant **"I am text"** is an array of characters (with \0 sign at the very end, added automatically by compiler), e.g.:

```
char *Letters = "ABC";
```



- String constants are often given as function arguments:

```
printf("I am string");
```

- When such a string is in code, access to it is performed by a character pointer. String constant is then accessible by a pointer to its first character.
- Other example:

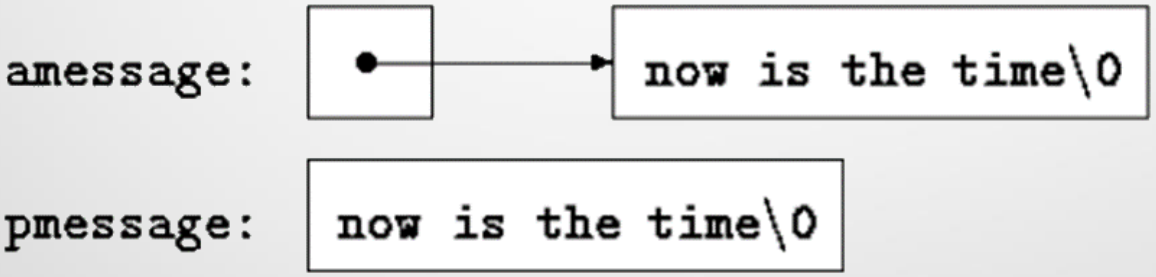
```
char *p_msg;
p_msg = "I am string"; // p_msq stores address of the first sign, i.e.: I
```

Arrays – string processing

- There is important difference between an array and a pointer in texts:

```
/* Every character in this array can be changed, amessage points ALWAYS to same area of
memory */
char amessage[] = "now is the time";    // Array
/* Pointer has been initialized to points on string constant. That pointer can be
reassigned to different memory address, but trying to change any particular character in
its string constant is generally not possible */
char *pmessage = "now is the time";     // Pointer
```

- Characters in ***amessage*** can be changed.



- Pointer points to a particular string. It can be later used to point on something different, however we cannot use it to change any single character in its pointed string.

Arrays – string processing

- **Example:**

```
char *Text = "Results.";  // string constant
*(Text + 1) = 'y';        // error

char Text [16] = { "Error." };
                          // copying text to an array
Text[3] = 'y';            // ok
*(Text + 3) = 'y';        // ok
```

Arrays – string processing

- **Example**: copying string arrays.

```
/* strcpy: copy t to s; array subscript version */
void strcpy(char *s, char *t){
     int i;
     i = 0;
     while ((s[i] = t[i]) != '\0')
          i++;
}
```

- To compare: another version using pointer syntax:

```
/* strcpy: copy t to s; pointer version */
void strcpy(char *s, char *t){
     int i;
     i = 0;
     while ((*s = *t) != '\0') {
          s++;     // move pointer s to the next character
          t++;     // move pointer t to the next character
     }             // in short: while (*s++ = *t++) ;
}                  // *s++ read value and move further
```

Arrays – string processing

- Effect of instruction ***s = t;*** where ***s*** and ***t*** are arrays of signs, is one array pointer by both ***s*** and ***t.***
- Comparing two strings – **strcmp** (*string compare*).

```
/* strcmp: return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t) {
    int i;
    // moves pointer to the position where s and t vary
    for (i = 0; s[i] == t[i]; i++){
          if (s[i] == '\0')
              return 0;
    }
    return s[i] - t[i];
}
```

Arrays – string processing

- **Example:**

```
char Text[ 16 ] = "Somebody.";
int i = 0;
while(Text[i] != 0 ){ // to the end of an array
     // change all 'o' to 'a' in Text
     if (Text[i] == 'o') Text[i] = 'a';
          ++i;
}
// same thing, but with pointers:
char *ptr = Text;
while( *ptr != 0 ){     // to the end of an array
     if ( *ptr == 'o')
          *ptr = 'a';
     ++ptr;
}
```

Arrays – string processing, Example 1

- **Example:** encrypting program which changes letters in a given text with the following rules:
  - instead of letters a - z writes the next letter (for z -> a),
  - instead of letters A - Z writes the next letter (forA -> Z),
  - all other characters remain unchanged.

  Text can have multiple words, in must be one line, with \n as EOL

  Program accepts 4 orders:
  - N | n – read new text,
  - K | k – encrypt text,
  - D | d – decrypt text,
  - Q | q – quit.

```
int main(void){
      char Text[64];
      char* ptr;
      char option;
      bool go_on= true;
```

Arrays – string processing, Example 1

```
while (go_on) {
     printf("Option:[N, K, D, Q] : ");
     fflush(stdin);
     scanf("%c", &option);

     switch(option & 0x5F){ //change small character on large, e.g. 'n' into 'N'
          case 'N' :
               printf("Type new text: \n");
               fflush(stdin);
               scanf("%63[ -~]", Text);  //&Text[0], with spaces
               break;
          case 'K' :
               printf("Encrypted text: \n"); // code message
               ptr = Text;// &Text[0];
               while(*ptr){
                    if (*ptr >= 'a' && *ptr < 'z') // except letter z
                         printf("%c", *ptr + 1); // increase code for character
```

Arrays – string processing, Example 1

```
            else if (*ptr > 'A' && *ptr <= 'Z') // except A
                printf("%c", *ptr - 1);     // previous letter
            else if (*ptr == 'z')       // for z change to a
                printf("%c", 'a');
            else if (*ptr == 'A')       // for A change to Z
                printf("%c", 'Z');
            else printf("%c", *ptr);         // plain text
                ++ptr; // move pointer to the next character
        }
        printf("\n");
        break;
    case 'D':
        printf("Decrypted message: \n"); // decode
        ptr = Text;   // &Text[0]; first array element
        while(*ptr){
            if (*ptr > 'a' && *ptr <= 'z') // except a
                printf("%c", *ptr - 1);     // previous letter
            else if (*ptr >= 'A' && *ptr < 'Z') // except Z
                printf("%c", *ptr + 1);     // next letter
```

Arrays – string processing, Example 1

```
                else if (*ptr == 'a') // previous to a is z
                     printf("%c", 'z’);
                else if (*ptr == 'Z') // next for Z is A
                     printf("%c", 'A’);
                else
                      printf("%c", *ptr);
                // plain text
                ++ptr;   // move pointer to the next character
           }
           printf("\n");
           break;
      case 'Q’:
           go_on = false; // quit
           break;
      default:
           printf("Unknown option.\n");
      }
  }
  return 0;
}
```

Library: string.h

- In library header **<string.h>** there are some interesting functions for strings

```
typedef unsigned int size_t;
```

-
```
char* strcat( char* Destination, const char* Source );
  // Adds up texts Source and Destination
  // (Source to the end of Destination). Result: pointer to
  // string with the resulting superstring
```

-
```
char* strncat( char* Dest, const char* Source, size_t Count );
  // Same as above but copies no more character from Source
  // then Count, copies to the end of Destination string
```

-
```
size_t strlen( const char* String );
  // Returns string length
```

Library: string.h

- ```
  char* strchr( const char* String, int C );
    // Finds first position of character C in String
    // Result: pointer to found character or NULL
  ```

- ```
  char* strrchr( const char* String, int C );
    // Same as above but starts searching from the last letter of
    // String, move to first, gives pointer to first found.
  ```

- ```
  char* strpbrk( const char* String, const char* CharSet );
    // Search first founded substring CharSet within String, returns
    // pointer to the first sign of found pattern, NULL if not
    // found.
  ```
  ```

Library: string.h

- ```
  int strcmp( const char* String1, const char* String2 );
    // Compares two string :
    // < 0 - String1 smaller than String2
    // = 0 - String1 and String2 are identical
    // > 0 - String1 larger than String2
  ```

- ```
  int strncmp(const char* String1, const char* String2, size_t Count);
    // Same as above but compares only Count number of characters
  ```

Library: string.h

- ```
  char* strcpy( char* Destination, const char* Source );
   // Move Source text to the Destination text (effectively
   // erasing the latter). Results: pointer to new Destination
   // string (with Source content).
  ```

- ```
  char* strncpy( char* Dest, const char* Source, size_t Count );
   // Same as above but only copies first Count characters
   // of Source into Destination. The rest of Destination characters
   // are replaced with \0.
  ```

Library: string.h – example

```c
#include <stdio.h>    #include <stdlib.h>   #include <string.h>
int main(int argc, char *argv[]) {
      char Tab[128];
      int size = 0;
      char* ptr = Tab;
      char searched;
      int counter = 0;
      printf("Enter multiword text:\n");
      scanf("%127[ -~]", Tab);    // with spaces, max 127 signs
      printf("Enter search character: ");
      fflush(stdin);              //cleaning keyboard buffer!
      scanf("%c", &searched);

      size = strlen(Tab);
      while( ptr != NULL ) {      // returns pointer to the first found text
            ptr = strchr(ptr, searched);
            if( ptr != NULL) {
                  ++counter;
                  ++ptr;
            }
      }
      printf("Sign %c has been found %d times.\n", searched, counter);
      system("PAUSE");
      return 0;
}
```

Dynamic memory allocation

Arrays of pointers, multidimensional arrays, jagged arrays, etc.

Dynamic memory allocation in C++

- For memory allocation we use **new** command (C++), e.g.:

```
int *wsk;
wsk = new int;     // creates new int object, no name, but its address is given
                   // to wsk pointer
```

- To free memory previously dynamically allocated we use **delete** command (C++).

```
delete wsk;  // deletes object pointed by wsk (if created using new)
```

- Creating array:

```
float *tab;
tab = new float[15];
     // creates new 15-elements float array. It has no name, but
     // pointer tab points to its first elements.
delete [] tab;     // delete the array
```

Dynamic memory allocation

- Examples:

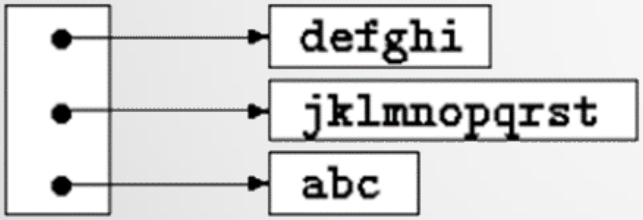
```
float *taba  =  new float [ 150 ];
delete [ ] taba;

int  size;
scanf( "%d" , &size);
long *Table =  new long [size];

double *TD1 = new double[150];
double x = TD1[53];                    // ok

double *TD2 = new double [20][30];     // error
double *TD2 = new double [600];        // ok
double y = TD2[3][21];                 // error, TD2 is one dimensional
double z = TD2[3 * 30 + 21];           // ok
```

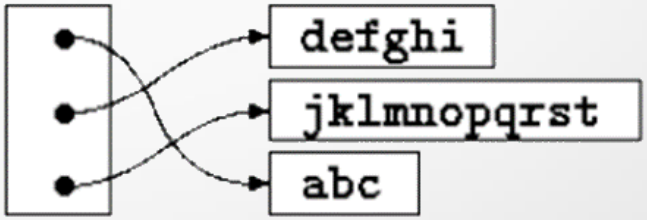
Arrays of pointers

- Pointers are variables in C, so they can be stored in arrays.
- Such an array is called pointers array and it contains multiple addresses for some areas in memory.
- **Example:**
  - We have many lines of text and want to sort them.
  - Example solution: put them all in a single long array, one text after another (so e.g.: 3 strings: `"she" "has" "dogs"` we pun into single string : `"she\0has\0dogs"`) – not very convenient (moving substrings withing such an array requires many extension / compression operations).
  - Another solution:
    - Using pointers array.
    - All lines (words) are stored in a string array, and each line will be stored using pointer to its first character.
    - Such pointers will be stored in another array (pointers array).

Arrays of pointers

- Lines can be compared using ***strcmp*** (providing two pointers for two lines of text).
- When one wants to exchange two lines, all one needs to do is to switch their pointers in the pointers array. The lines remain in the same positions in the text array.
- Sorting is performed using pointers array only.





- Pseudocode:
  - read all lines of text
  - sort them
  - write them down in a sorted sequence

Arrays of pointers – example

```c
#include <stdio.h>
#include <string.h>
#define MAXLINES 5000 /* max lines to sort*/
#define MAXLEN 1000 /* max length of any input line */
//FUNCTIONS PROTOTYPES:
char *lineptr[MAXLINES]; /* array of pointers */
int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
void qsort(char *lineptr[], int left, int right);
char line[MAXLEN]; /* current text line */
char longest[MAXLEN];
int getline(char line[], int maxline);
int max; /* maximum size of text array to the current
moment */

/* sort input lines */
main(){
   int nlines; /* number of lines to sort */
       if ((nlines = readlines(lineptr, MAXLINES))>= 0){
       qsort(lineptr, 0, nlines-1);
       writelines(lineptr, nlines);
       return 0;
    } else {
       printf("error: input too big to sort\n");
       return 1;
    }
}
```

```c
/* readlines: */
int readlines(char *lineptr[], int maxlines){
   int len, nlines;
   char *p, line[MAXLEN];
   nlines = 0;
   while ((len = getline(line, MAXLEN)) > 0){
      // if nlines >= maxlines or memory cannot
      // be allocated:
      if (nlines>=maxlines || (p=malloc(len))== NULL){
         return -1;
      } else {
         line[len-1] = '\0'; /* delete newline */
         strcpy(p, line);
         lineptr[nlines++] = p;
      }
    }
    return nlines; /* number of text lines*/
}

/* writelines: */
void writelines(char *lineptr[], int nlines){
    int i;
    for (i = 0; i < nlines; i++)
       printf("%s\n", lineptr[i]);
} /* lineptr[i] is a pointer to first character of i-th
line of text */
```

```c
/* getline: */
 int getline(char s[],int lim){
    int c, i;
    for (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i) {
       /* while i < lim - 1 and not encountered sign \n or the end of buffer
          reads all characters of line to sort */
       s[i] = c;
    }
    if (c == '\n'){
       s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
 }
 /* qsort: sort v[left]...v[right] in ascending order*/
 void qsort(char *v[], int left, int right){
    int i, last;
    void swap(char *v[], int i, int j);
    if (left >= right) /* do nothing if array contains */
       return; /* fewer than two elements */ // pivot moves to the beginning of a range, index left
    swap(v, left, (left + right)/2); // pivot is middle element
    last = left; // index pointing to place to change
    for (i = left+1; i <= right; i++){
       if (strcmp(v[i], v[left]) < 0) /* compare texts */
          swap(v, ++last, i); /* change texts in pointers array */
    }
    swap(v, left, last); // move pivot to the proper position
    qsort(v, left, last-1);
    qsort(v, last+1, right);
 }
 /* swap: switch v[i] with v[j] */
 void swap(char *v[], int i, int j){
    char *temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
 }
```

Multidimensional array

- In C language there are rectangular multidimensional arrays, they are however usually less frequent than pointers arrays.
- Multidimensional arrays are arrays which elements are other arrays.
- Example of two-dimensional array (in C such an array is still 1-dimensional array, but with elements being other arrays):

```
int tab[4][2]; // 4 elements array, where each element is
               // 2-elements array, in other words: 4 rows with 2 elements each
```

  - Elements are arranged in such a way in memory:

tab[0][0] tab[0][1] tab[1][0] tab[1][1] tab[2][0] tab[2][1] tab[3][0] tab[3][1]

- Attention – WRONG SYNTAX **`tab[i,j]`** compiler will interpret as **`tab[j]`**, because comma operator ',' is left-side joined.
- Two dimensional array initialization – list of initial values can be placed in curly brackets (each row is a sub-list):

```
int tab[4][2] = {{0, 1}, {2, 3}, {4, 5}, {6, 7}};
int tab[4][2] = {0, 1, 2, 3, 4, 5, 6, 7}; // not recommended but still ok
```

Multidimensional array

- **Examples:**

```
float MATRIX[10][20];
/* 10 rows, 20 columns */
MATRIX[10[nw][nk] // nw – number of rows, nk – number of columns

const int kwa = 30;
long SQUARE[kwa][kwa];

for (int i = 0; i < kwa; i ++)
     for (int j = 0; j < kwa; j ++)
         SQUARE [ i ] [ j ] = i == j ? 0 : 1;
// main diagonal will be zero, all other elements = 1
```

Multidimensional array

- **Examples:**

```
int BB[2][3] = { { 1, 2, 3} , {4, 5, 6} };
/*
 BB[0][0] == 1, BB[0][1] == 2, BB[0][2] == 3
 BB[1][0] == 4, BB[1][1] == 5, BB[1][2] == 6
*/

float CC[3][2] = { { 8.5 } , { 3.2 } };
/*    CC[0][0] == 8.5, CC[0][1] == 0
      CC[1][0] == 3.2, CC[1][1] == 0
      CC[2][0] == 0  , CC[2][1] == 0   */

long LL[2][3];
sizeof LL;                    // (2 * 3) * 4 bytes = 24
sizeof LL[0];                 // 3 (row has 3 elements) * 4 bytes = 12
sizeof LL[0][0];              // 4 bytes
```

Multidimensional array

- **Reading matrix line by line:**

```
const int Wie = 10, Kol = 5;
int MM [ Wie ][ Kol ];
for (int i = 0; i < Wie; ++i)
     for (int j = 0; j < Kol; ++j)
          scanf("%d", & MM[ i ][ j ];

// the same as above but with pointers
int ile = Wie * Kol, k;
int *p = &MM[0][0];

k = 0;
// 2-dimensional array is in fact 1-dimensional array
// (see: previous explanation)
while (k++ < ile)
     scanf("%d", p++);
```

Multidimensional array

- **Example of 3-dimensional array:**

```
float TTT[5][10][20];
// 5 matrices, each 10 rows, 20 columns

for (int mac = 0; mac <  5; mac ++)
     for (int wie = 0; wie < 10; wie ++)
          for (int kol = 0; kol < 20; kol ++)
               TTT[mac][wie][kol] = 1.0F;
```

Multidimensional array – Example 1

```
int main(){
     double A[100], B[100], C[100];
     int n = 102;
     int i;
     while (n > 100 || n < 1){
          printf("Enter size of an rray [1-100]: ");
          scanf("%d", &n);
     }

     printf("Enter %d elements of array A (real numbers) : ", n);

     for (i = 0; i < n; ++i)
          scanf("%lf", &A[i]);

     printf("Enter %d elements of array B (real numbers) : ", n);

     for (i = 0; i < n; ++i)
          scanf("%lf", &B[i]);
```

Multidimensional array – Example 1

```
    for (i = 0; i < n; ++i)
        if (A[i] > B[i])
            C[i] = 2 * A[i] + B[i] + 1;
        else
            C[i] = A[i] - B[i] - 1;

    for (i = 0; i < n; ++i)
        printf("C[%d] = %8.2lf\n", i, C[i]);

    return 0;
}
```

Multidimensional array – Example 2

- **Example** – compute sum of elements for lower matrix triangle and upper matrix triangle (elements of main diagonal are not part of the sum).

```
int main( ){
     int Size = 200;
     float Matrix[100][100];
     int Row, Column;
     float SumUp= 0.0f, SumDown= 0.0f;
     while (Size < 1 || Size> 100){
          printf("\nEnter matrix size: ");
          scanf("%d",&Size);
     }
 printf("\nEnter elements line by line: \n");
 for (Row = 0; Row < Size; ++Row )
     for (Column= 0; Column < Size; ++Column)
          scanf("%f", &Matrix[Row][Column]);
```

Multidimensional array – Example 2

```
    for (Row = 0; Row < Size; ++Row )
         for (Column = 0; Column < Size; ++Column )
              if (Column > Row)
                   SumUp+= Matrix[Row][Column];
              else if (Column < Row)
                   SumDown+= Matrix[Row][Column];
    printf("\nSum up= %10.2f\nSum down= %10.2f\n", SumUp, SumDown);
    return 0;
}
```

E.g. * upper triangle, + lower triangle

```
  0 1 2 3 4 5 6
0 - * * * * * *
1 + - * * * * *
2 + + - * * * *
3 + + + - * * *
4 + + + + - * *
5 + + + + + - *
6 + + + + + + -
```

Multidimensional array – Example 3

- **Multiplication of matrices: C = A * B**

```
const int Aw = 2;// number of rows for A
const int Ak = 3;// number of columns for A
const int Bw = 3;// number of rows for B
const int Bk = 2;// number of columns for B
int main (){
     int i, j, m;
     double s = 0.0;
     double A[Aw][Ak], B[Bw][Bk], C[Aw][Bk];
     // number of elements in a row of first matrix must be equal
     // to the number of columns in second matrix
     if (Ak != Bw){
          printf("Wrong size.\n");
          return;
     } else {
          for (i = 0 ; i < Aw ; ++i){ // read elements of A
               for (j = 0 ; j < Ak ; ++j){
                    printf("A[%d][%d] = ", i, j);
                    scanf("%lf", &A[i][j]);
               }
          }
```

Multidimensional array – Example 3

```
printf("\n");
for (i = 0 ; i < Bw ; ++i){ // read elements of B
     for (j=0 ; j < Bk ; ++j){
          printf("B[%d][%d] = ", i, j);
          scanf("%lf", &B[i][j]);
     }
}
// multiply matrices A * B. Result is stored in matrix C
for (i = 0 ; i < Aw ; ++i)
     for(j = 0 ; j < Bk ; ++j){
          s = 0.0;
          for (m = 0 ; m < Ak ; ++m){ // row from A times column from B
               s += (A[i][m] * B[m][j]);
          }
          C[i][j] = s; // s – sum for rows * columns
     }
```

Multidimensional array – Example 3

```
        printf("\nMatrix A\n\n"); // write down elements of A
        for (i = 0 ; i < Aw ; ++i){
             printf("\n");
             for (j = 0 ; j < Ak ; ++j) printf("%5.2lf\t",  A[i][j]);
        }
        printf("\n\nMatrix B\n\n"); // write down elements of B
        for (i = 0 ; i < Bw ; ++i){
             printf("\n");
             for (j = 0 ; j < Bk ; ++j) printf("%5.2lf\t", B[i][j]);
        }
        printf("\n\nMatrix C\n\n"); // write down elements of C
        for (i = 0 ; i < Aw ; ++i){
             printf("\n");
             for (j = 0 ; j < Bk ; ++j) printf("%5.2lf\t", C[i][j]);
        }
        printf("\n\n");
    }
    return 0;

}
```

Multidimensional array – Example 4

- Simulation of cyclic buffer for storing ***int*** numbers in array. The length of the array should be 2^N, for easy checking if index is in range.

```
int main (){
     const int Size = 4;// 2 ** N
     const int Mask = Size - 1;
     int Bufor [ 4 ] = {0}; // fill with 0
               // index for array Bufor:
     int Head = 0; // first empty (place for new element)
     int Tail = 0; // first to read (element to read)
     int Tmp;
     char Order;
     int Again = 1;
     while (Again){
          printf("\nChoose operation [R, W, Q] : ");
          fflush(stdin);
          scanf("%c", &Order);
```

Multidimensional array – Example 4

```
switch(Order & 0x5F){
     case 'R' :
          if (Head == Tail)
               printf("\nEmpty buffer"); // condition for empty buffer
          else { // read by tail and increase by 1
               printf("\n>%4d",Bufor[Tail++]);
               Tail &= Mask; // possible loop by buffer/tail
               // for Tail > 3 assign 0 to Tail
               // Tail = 4 is 00000100
               // Mask = 3 is 00000011
               // Tail = Tail & Mask is 0
          }
          break;
   case 'W' :
          Tmp= Head; // head copy
          Tmp++; // increase copy
          Tmp &= Mask; // possible loop
          // when buffer gets more than 4 numbers
          // will assign to Tmp 0
          // (for Tmp <= 3 no changes, for Tmp > 3 set to 0)
```

Multidimensional array – Example 4

```
                if (Tmp == Tail)
                     printf("\nFull buffer.");
                else {
                     printf("\nProvide element: ");
                     // write in head position
                     scanf("%d", &Bufor[Head]);
                     Head = Tmp; // head update
                }
                break;
            case 'Q' :
                Again = 0;
                break;
            default :
                printf("\nWrong operation [R, W, Q]\n");
        } //end switch

    } //end while
    printf("\n\n");
    return 0;

}
     // for example if by operation w we will provide: 23, 34, 12,
     // the r will read (and remove from buffer) elements
     // in the following order: 23, 34, 12
```

Two-dimensional array – array of rows

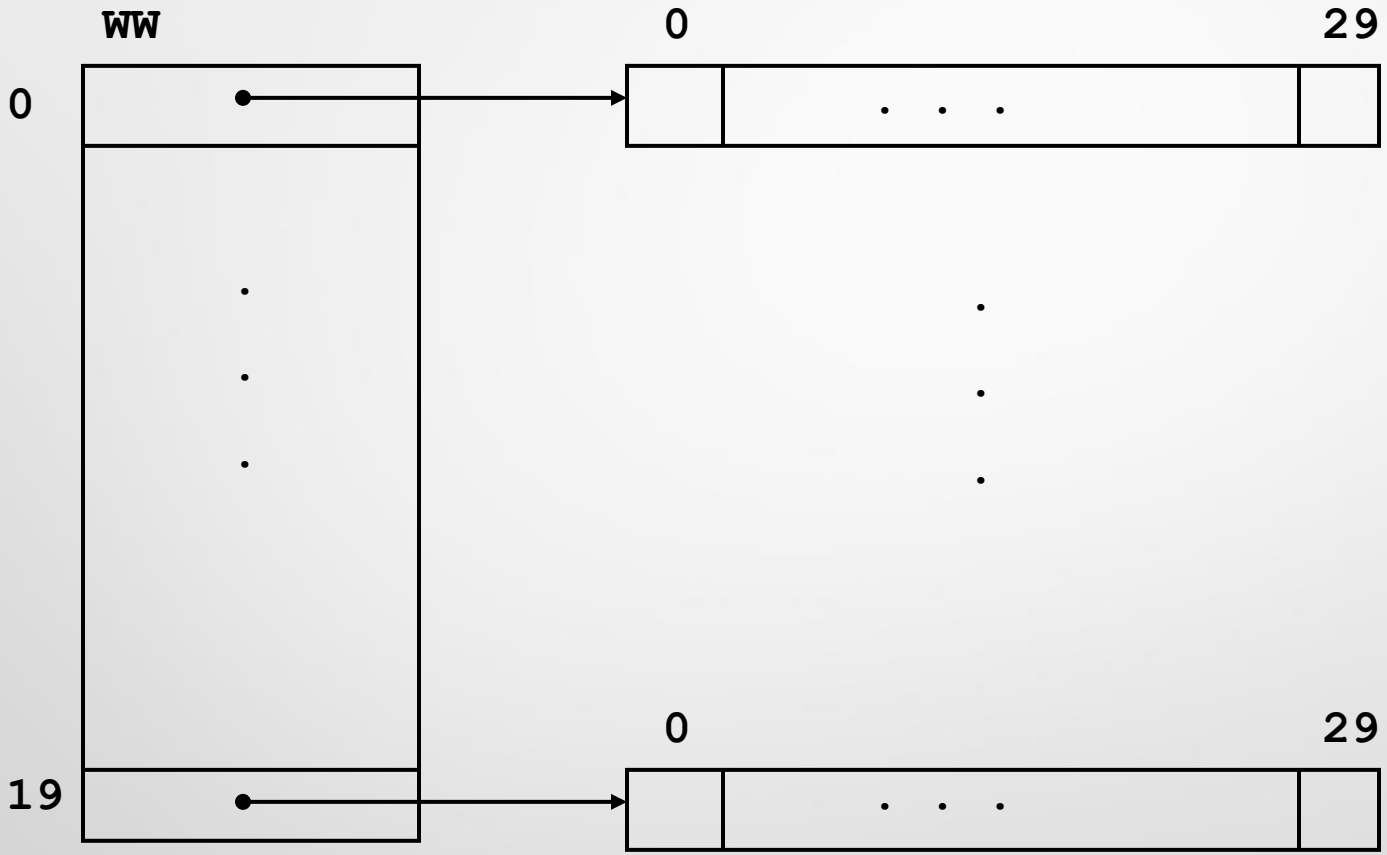
- After definitions:

```
int a[10][20];// real two-dimension array, to find specific
              // location we use pattern 20 * row + column
int *b[10];   // assign 10 cells for pointers,
              //without their initialization
```

- **Example** for pointers array:

```
int** WW = new int*[20];  // assign 20 cells for pointers
for (int i = 0; i < 20; ++i)
    WW[i] = new int[30];  // assign memory for 30 elements type
                          // int. Each pointer from WW[i] will
                          // point to 30-elements array
WW[4][5] = 17;
int x = WW[4][5];         // ok
```

Two-dimensional array – array of rows



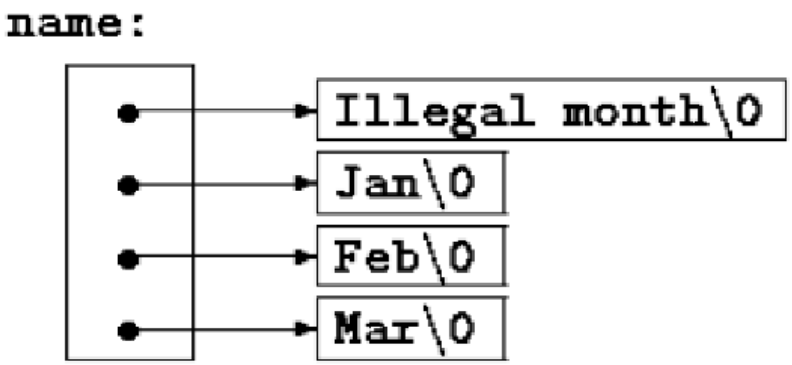
Two-dimensional *jagged* arrays

- Array of pointers allow creating arrays where rows have different sizes.
- Usually we use it to store text having, obviously, different lengths
- **Example:**

```
int DL [ ] = {5, 15000, 7, 2000, 3};
int N = sizeof(DL) / sizeof(DL[0]); // number of elements of array DL
int** WW = new int*[N];          // pointers array of size N
for (int i = 0; i < N; ++i)
     WW[i] = new int[DL[i]];     // assigns memory for next lines / rows
                                 // having size: 5, 15000, 7, 2000, 3
WW[1][12547] = 17;               // this row has size 15000, so ok
int x = WW[1][12547];
```

Two-dimensional *jagged* arrays

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```




```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

aname:

Illegal month\0 Jan\0	Feb\0	Mar\0
0 15	30	45

Questions?