Introduction to programming
2014 - 2015

Corina Forăscu

corinfor@info.uaic.ro

http://profs.info.uaic.ro/~introp/

Course 4: agenda

- Arrays
- Strings
- Pointers

Arrays

- array = a fixed number of **elements** of the **same** `data_type` stored **sequentially in memory** - a mean to store multiple values together as one unit
- Array `dimension` = the size of the array
- Array declaration:

```
data_type arrayName [dimension];
```

- The elements of an array can be accessed by using an **index** into the array.
- Arrays in C++ are zero-indexed, so the first element has an index of **0**.

Array usage

- Powerful storage mechanism
- Avoids declaring multiple simple variables
- Can manipulate "list" as one entity

- One-dimensional (1 – dimensional): strings
- Bi-dimensional (2 - dimensional)
- …
- [Multidimensional (n-dimensional)]

Array declaration & selection

```
data_type arrayName [dimension];
```

- `dimension` must be a constant expression!!

```
(int dataValues[n]; //ERROR)
```

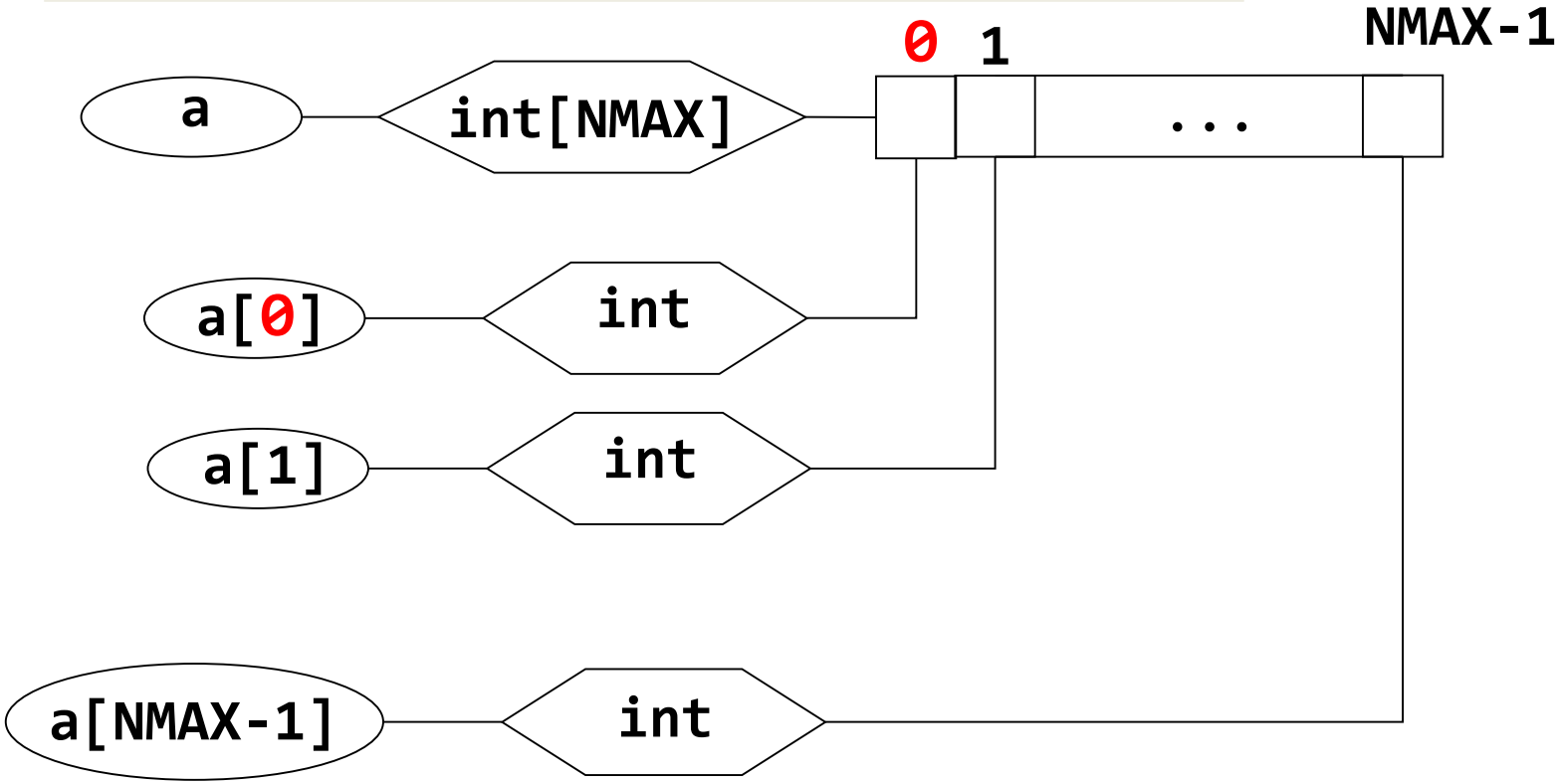
```
bytes = sizeof(data_type)* dimension
```

- arrays are blocks of static memory whose size must be determined at compile time.
- Array **selection** = identifying a particular element within an array:

```
array[index]
```

One-dimensional array declaration

```
#define NMAX 5
int a[NMAX]; // int a[5];
```



Similar to declaring five variables:

```
int a[0], a[1], a[2], a[3], a[4];
```

Array name

- Variable name
- Pointer to the first array element

a	same as **&a[0]**	***a**	same as **a[0]**
a+1	same as **&a[1]**	***(a+1)**	same as **a[1]**
a+2	same as **&a[2]**	***(a+2)**	same as **a[2]**
a+i	same as **&a[i]**	***(a+i)**	same as **a[i]**

Initializing arrays

```
int matr[4];


matr[0] = 6;
matr[1] = 0;
matr[2] = 9;
matr[3] = 6;
```

```
int matr[4] = { 6, 0, 9, 6 };
```

```
int matr[] = { 6, 0, 9, 6, 5, -9, 0 };
```

Auto-Initializing Arrays

- If fewer values than `dimension` are supplied:
  - The array is filled from the beginning
  - The "rest" is filled with zero of array base type
- If `dimension` is left out
  - The array is declared with the dimension indicated by the number of initialization values
- Given any array **`arr`**, the number of elements in `arr` can be computed using the expression

```
sizeof arr / sizeof arr[0]
```

Accessing arrays

- The operation with arrays are possible through their elements, using iterations (`for, while`)

```
for (i=0; i<n; i++) a[i]=0;
```

```
for (i=0; i<n; i++) c[i]=a[i]+ b[i];
```

- Accessing individual elements

```
arr[5];
arr[i];
arr[i+3];
```

Accessing arrays

/* 1st version */

```
for (i=0; i < n; ++i)
    suma += a[i];
```

/* 2nd version */

```
for (i=0; i < n; ++i)
    suma += *(a+i);
```

/* 3rd version */

```
for (p=a; p < &a[n]; ++p)
    suma += *p;
```

/* 4th version */

```
p=a;
for (i=0; i < n; ++i)
    suma += p[i];
```

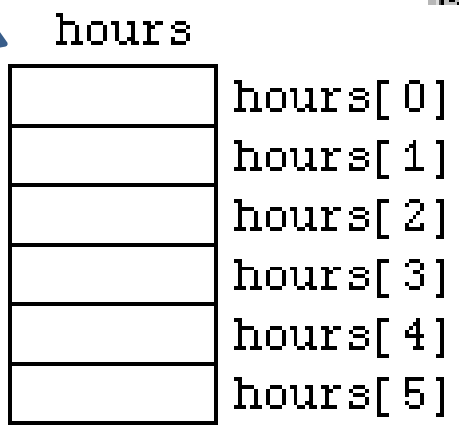
Arrays: example

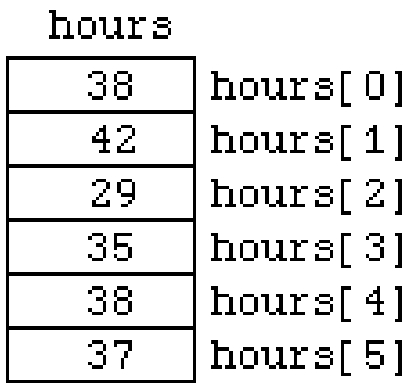
```
int hours[NO_OF_STUDS];
int count;

for (count = 1 ; count <= NO_OF_STUDS ; count++)
{cout << "Orele petrecute la laborator de
studentul cu numărul: " << count << ": ";
cin >> hours[count - 1];
}
```

```
Orele petrecute la laborator de studentul cu numĂrul: 1: 38
Orele petrecute la laborator de studentul cu numĂrul: 2: 42
Orele petrecute la laborator de studentul cu numĂrul: 3: 29
Orele petrecute la laborator de studentul cu numĂrul: 4: 35
Orele petrecute la laborator de studentul cu numĂrul: 5: 38
Orele petrecute la laborator de studentul cu numĂrul: 6: 37
Press any key to continue
```

NO_OF__STUDS = 6



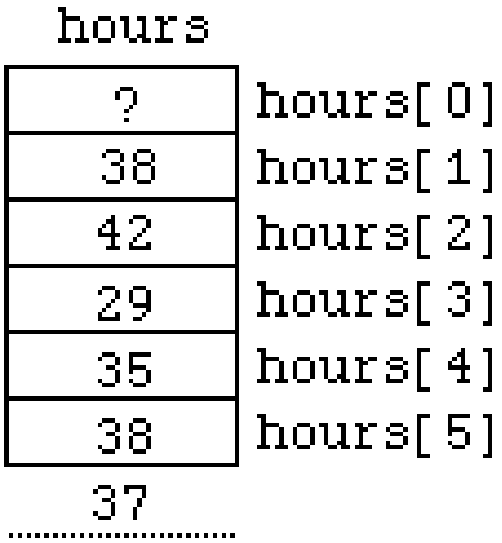


Arrays: example 2

- Array index starts at 0 and it is going till (dim-1)
- array dimension surpassed -> unpredictable results

```
for (count = 1 ; count <=
NO_OF_STUDS ; count++)
{cout << "Orele petrecute …cu numărul:
" << count << ": ";
cin >> hours[count];
}
```

NO_OF__STUDS = 6



Arrays: example 3

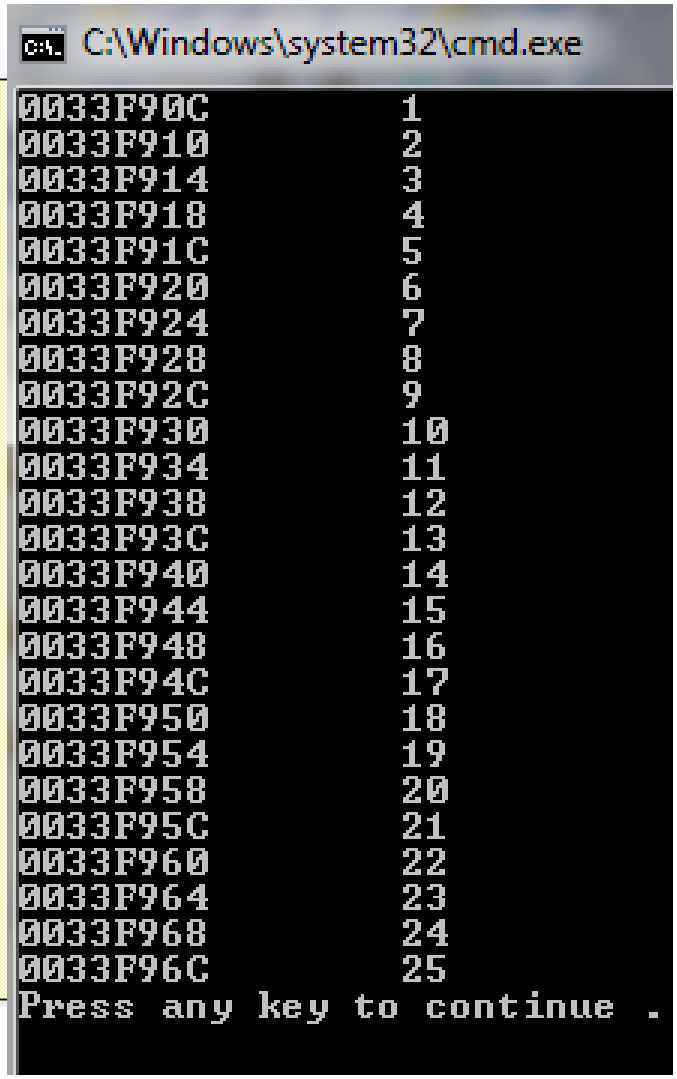
```
#include <iostream>
using namespace std;

#define NMAX 25
int main(){
        int a[NMAX], i;
        for (i=0; i<NMAX; i++){
        a[i] = i+1;
        cout << &a[i] << "\t" << *(a+i)<< "\n";
        }
        return 0;
}
```

Arrays: example 3

```
#include <iostream>
using namespace std;

#define NMAX 25
int main(){
        int a[NMAX], i;
        for (i=0; i<NMAX; i++){
        a[i] = i+1;
        cout << &a[i] << "\t" <<
                *(a+i)<< "\n";
        }
        return 0;
}
```



Arrays pitfalls

- Array indexes always start with zero!
- Zero is "first" number to computer scientists
- C++ will "let" you go beyond range, but
  - Unpredictable results
  - Compiler will not detect these errors

Arrays in functions

- Arguments in functions
  - Indexed variables
    - An individual "element" of an array can be a function parameter `funct(arr[i])`
  - Entire arrays
    - All array elements can be passed as "one entity“, by reference
- As return value from functions

Arrays as function arguments

- An array is NOT passed entirely as an argument
- Arrays are passed as function arguments through pointers to the first element of the array
- The formal parameter of a function with an array as argument can be declared as:

```
int main(void){
        int i[4];
        functia(i);
    return 0;
}
```

- Pointer

```
void functia(int *x)
```

- Array with a specific dimension

```
void functia(int tab[4])
```

- Array with no specific dimension

```
void functia(int tab[])
```

Arrays in functions: example 1

```
void insert_sort(int a[], int n)
{
  //...
}
```

```
/* utilizare */
int w[100];
// ...
insert_sort(w, 10);
```

Arrays in functions: example 2

```
double suma(double a[], int n);
// double suma(double *a, int n);

suma(v, 100);
suma(v, 8);
suma(&v[4], k-6);
suma(v+4, k-6);
```

Strings – arrays of characters

- Unidimensional arrays of chars
  - Each element in the array = char
  - LAST character in string – NULL character "\0" = indicates the end of the string

- `char sir[10];`
  - An array of char with 9 elements AND
  - The null character "\0“ at the end

String asignment and comparison

- Asignment “=” ONLY in declaration:

```
char sir[10];
sir = "Hello";// ILLEGAL

char sir[10] = "Hello";// OK
```

- Comparison not possible through the “==” operator

```
char sir_unu[10] = "alba";
char sir_doi[10] = "neagra";
sir_unu == sir_doi;   // warning: operator has no effect
```

- `strcmp` function is used instead

Strings – arrays of characters

- String declaration:

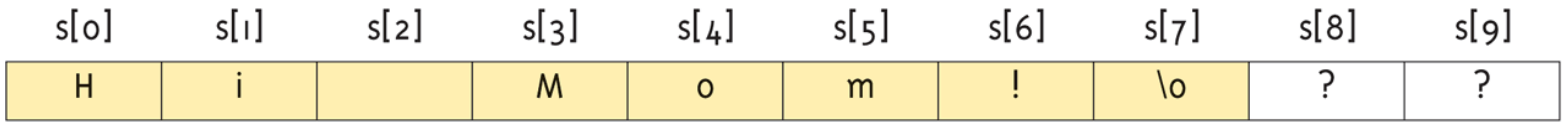
```
#define MAX_SIR 100 // big enough
...
char sirCaract[MAX_SIR];
```

- Declaration with initialization:

```
char s[] = "Hi Mom!";

char s[10]="Hi Mom!";

char s[10]= {'H', 'i', ' ', 'M', 'o', 'm', '!', '\0'};
```



Working with c-strings

- Declaration
  - Requires no C++ library
  - Built into standard C++
- Manipulation
  - Requires `<cstring>` library
  - Typically included when using c-strings
    - Normally want to do "fun" things with them

C-strings macros & functions

- **<cctype> (ctype.h)** (char handling functions)

```
isspace(c)
isdigit(c)
islower(c)
ispunct(c)
isalpha(c)
..
tolower(c)
toupper(c)
```

C-strings macros & functions

- **<cstring> (string.h)** (c-string & arrays manipulation)

```
char * strcat ( char * destination, const char * source );
int strcmp(const char *s1,const char*s2);
char * strcpy ( char * destination, const char * source );

size_t strlen ( const char * str );


const char * strchr ( const char * str, int character );
      char * strchr (       char * str, int character );
```

```
char mystr[100]="test string";
sizeof(mystr);  // 100
strlen(mystr);  // 11
```

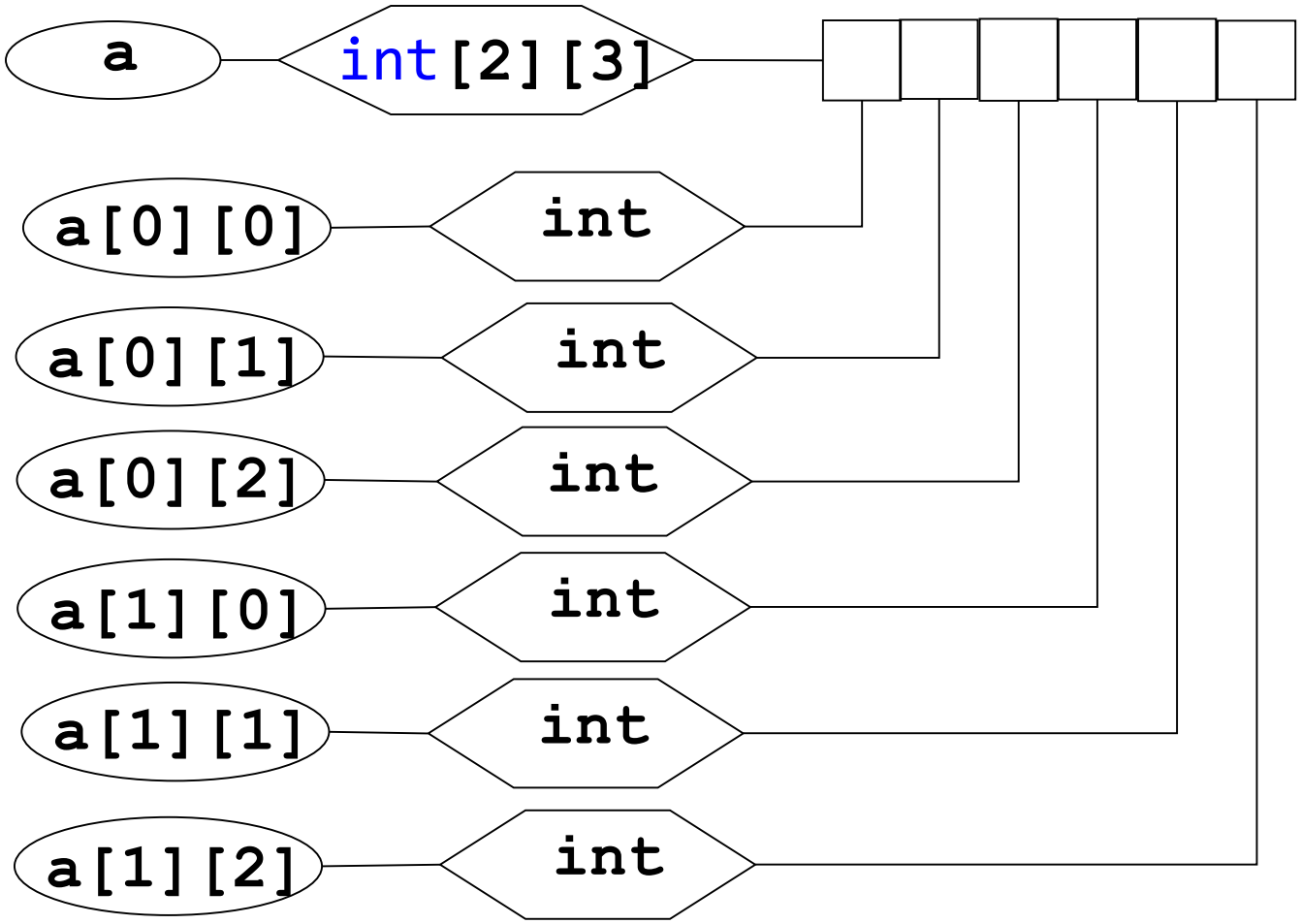
Bidimensional arrays

```
data_type arrayName[m][n];    int a[m][n];
```

- Contiguous memory of $m \times n$ locations
- Elements are identified through two indexes:
  - 1st index ranges in {0, 1, …, m -1}
  - 2nd index ranges in {0, 1, …, n -1}
  - Elements: a[0][0], a[0][1], …, a[0][n-1],
    a[1][0], a[1][1], …, a[1][n-1],
    …,
    a[m-1][0], a[m-1][1], …, a[m-1][n-1]
- The lexicographic order of indexes gives the elements' order in memory

Bidimensional arrays

```
int a[2][3];
```



Bidimensional arrays

```
double a[MMAX][NMAX]; // declaration
double suma;

/* . . . */

for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        cin >> a[i][j];

suma = 0;
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        suma += a[i][j];
```

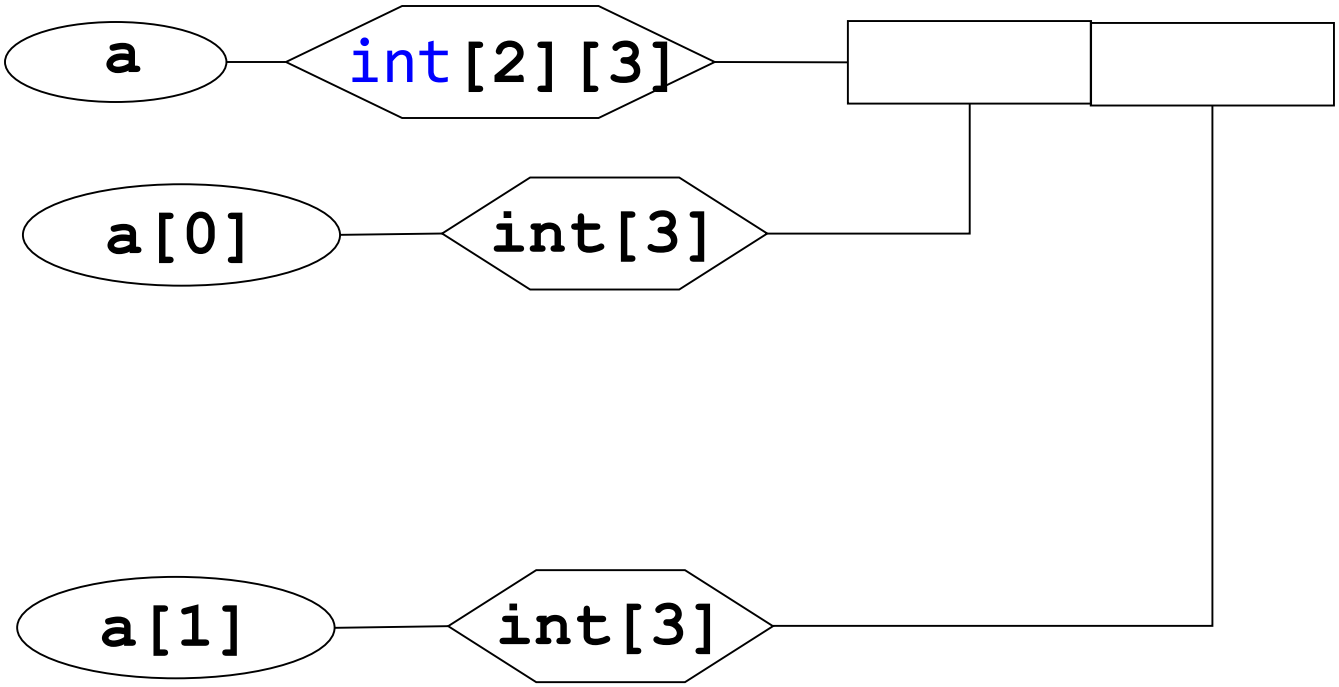
Bidimensional arrays

- Just like with matrices, a bidimensional array is an unidimensional array where each element is an unidimensional array.

- Notation

```
a[0][0], a[0][1], …, a[0][n-1],
…,
a[m-1][0], a[m-1][1],…, a[m-1][n-1]
```

Bidimensional arrays like unidimensional arrays



Bidimensional arrays

	Column 0	Column 1	. . .
Row 0	`a[0][0]`	`a[0][1]`	. . .
row 1	`a[1][0]`	`a[1][1]`	. . .
. . .	. . .	. . .	. . .

Equivalent expressions for `a[i][j]`

```
*(a[i] + j)
```

```
*((*(a + i)) + j)
```

```
(*(a + i))[j]
```

```
*(&a[0][0] + NMAX*i +j)
```

Bidimensional arrays as arguments

```
int minmax(int t[][NMAX], int i0, int j0,
           int m, int n)
{
  //...
}
```

```
/* utilizare */
if (minmax(a,i,j,m,n))
{
   // ...
}
```

Initialization of arrays

```
int a[] = {-1, 0, 4, 7};
 /* same as */
int a[4] = {-1, 0, 4, 7};
```

```
char s[] = "un sir";              /* same as */
char s[7] = {'u', 'n', ' ', 's', 'i', 'r', '\0'};
```

```
int b[2][3] = {1,2,3,4,5,6}  /* same as */
int b[2][3] = {{1,2,3},{4,5,6}} /* same as */
int b[][3] = {{1,2,3},{4,5,6}}
```

Pointers

- Pointer = variable containing the memory address where another object (usually a variable) is stored

- More flexible pass-by-reference
- Improved efficiency for some procedures
- Permit dynamic allocation (reserve new memory during program execution)
- Manipulate complex data structures efficiently, even if their data is scattered in different memory locations
- Use polymorphism – calling functions on data without knowing exactly what kind of data it is

- Uninitialised pointers
- Pointers with inadequate values

Pointers

- Declaring a pointer:

```
data_type *pointerName;
```

- `pointerName` e is a variable holding the memory address of a variable of type `data_type`
- the **base type** of a pointer is the type of the value to which a pointer points

  - A pointer is an address
  - An address is an integer
  - A pointer is NOT an integer!
- A pointers cannot be used as a number even though it “IS A" number

Pointers

```
int *p, i;  // int *p; int i;
p = 0;
p = NULL;
```

- Add "*" before variable name
- Pointer variable p "points to" ordinary variable of type int

Pointer operators

- Operator **&** - it returns the "**address of**" a variable

```
int *ptr = &x;
```

  - `ptr` “points to x”, “contains/ equals the address of x”, “references to x”.

- Dereference operator ***** - it returns the “**value pointed to**" by the variable

```
cout *ptr; //Prints the value pointed to by ptr
```

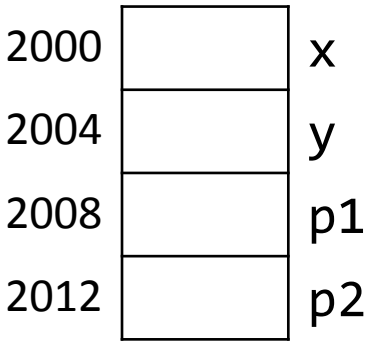
```
int *p, i;  // int *p; int i;
p = &i;
p = (int*) 232;
```

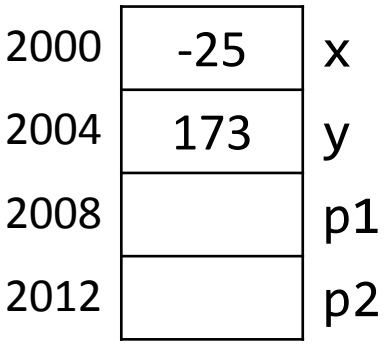
Pointers – details (1)

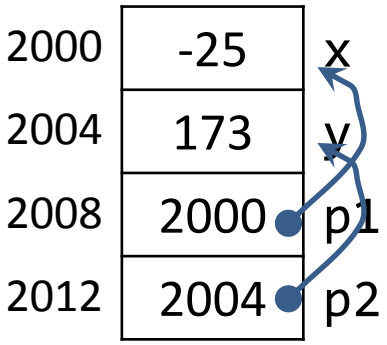
```
int x, y;
int *p1, *p2;
```

```
x = –25;
y = 173;
```

```
p1 = &x;
p2 = &y;
```

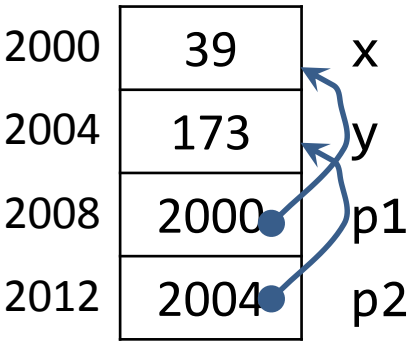




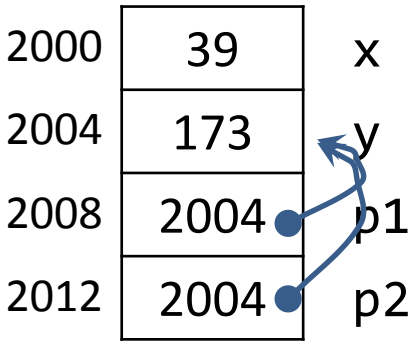


Pointers – details (2)

*p1 = 39;



p1 = p2;



`p1 = p2;` // **pointer assignment**

makes p1 and p2 point to the same location

`*p1 = *p2;` // **value assignment**

copies the value from the memory location addressed by p2 into the location addressed by p1.

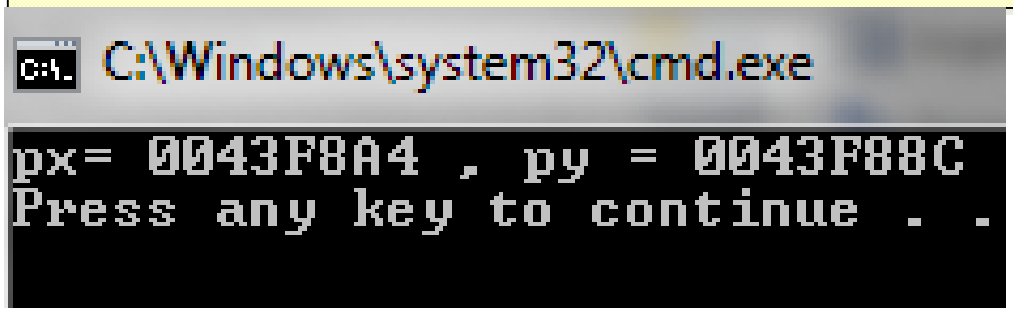
Pointers

- Pointers contain memory addresses, hence their memory size does not depend on the base type

```
sizeof(int*) = sizeof(double*) = ...
```

- Displaying a pointer:

```
int *px, x = 0, *py, y = 0;
px = &x;  py = &y;
cout <<"px= " << px << " , py = " << py << endl;
```



Pointers

```
int i = 3, j = 5;
int *p = &i, *q = &j, *r;
double x;
```

Expression	Same as	Value
`p = &i`	`p = (&i)`	`002EFEA4`
`**&p`	`*(*(&p))`	`3`
`r = &x`	`r = (&x)`	`error!`
	`(cannot convert from double* to int*)`	
`3**p/(*q)+2`	`(((3*(*p)))/(*q))+2`	`3`
`*(r=&j)*=*p`	`(*(r=(&j)))*=(*p)`	`15`

Pointers

```
int *i;    float *f;    void *v;
```

Correct expressions	Incorrect expressions
`i = 0;`	`i = 1;`
`i = (int*)1;`	`v = 1;`
`v = f; i=(int *)v;`	`i = f;`
`i = (int*)f;`	`&3;`
`f = (float*)v;`	`&(k+8);`
`*((int*)333);`	`*333;`

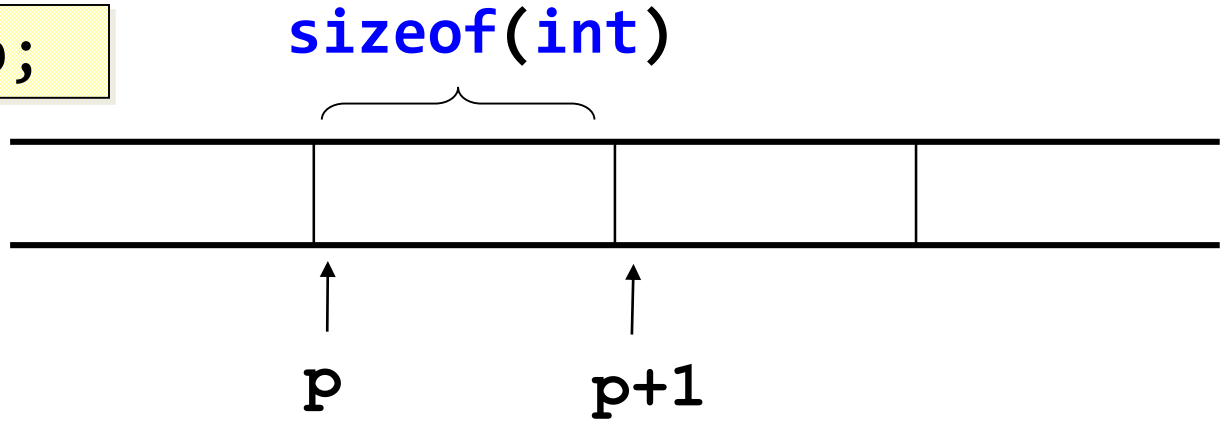
Pointers

```
#include <iostream>
using namespace std;
int main(void){
    int i=5, *p = &i;
    float *q;
    void *v;
    q = (float*)p;
    v = q;
    cout << "p = " << p << ", *p = " << *p << "\n";
    cout << "q = " << q << ", *q = " << *q << "\n";
    cout << "v = " << v << ", *v = " << *((float*)v)<<"\n";
return 0;
}
```

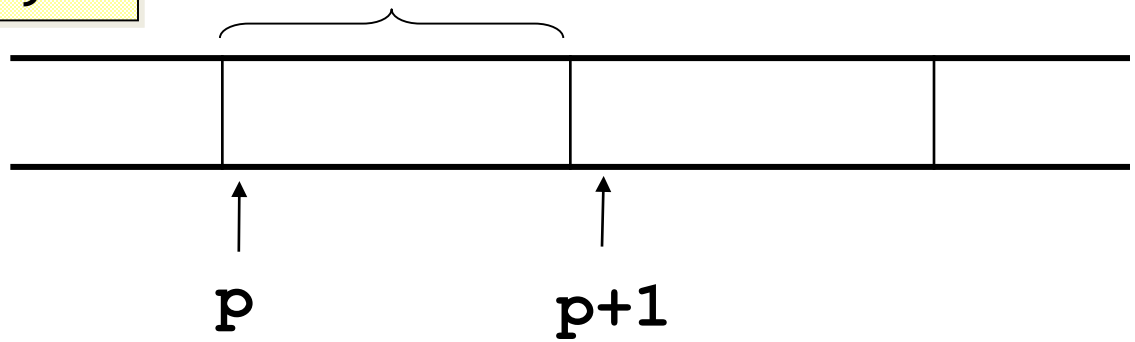
```
p = 0030F738, *p = 5
q = 0030F738, *q = 7.00649e-045
v = 0030F738, *v = 7.00649e-045
Press any key to continue . . .
```

Pointer arithmetics

```
int *p;
```




```
double *p;
```



p + 1 is the address of the next int / double in memory after p

Pointer arithmetics

```
int a[2], *p1, *q1;
```

```
p1 = a;
q1 = p1 + 1;
cout << "q1-p1 "<< q1-p1 << endl;
cout << sizeof(int) << (int)q1 - (int)p1 << endl;
```

q1 - p1 = 1

sizeof(int) = 4, (int)q1 - (int)p1 = 4

Scaling: when calculating the result of a pointer arithmetic expression, the compiler always multiplies the integer operand by the size of the object being pointed to.

Pointer arithmetics

```
double c[2], *p3, *q3;
```

```
p3 = c;
q3 = p3 + 1;
cout << q3-p3;
cout << sizeof(double), (int)q3 - (int)p3;
```

q3 - p3 = 1

sizeof(double) = 8, (int)q3 - (int)p3 = 8

Glossary

- Array
- Array dimension
- Array index
- Array selection
- C-string
- matrix
- Pointer
- Reference / dereference
- Base type of a pointer
- Pointer arithmetics