jgromero@inf.uc3m.es

Universidad
Carlos III de Madrid
www.uc3m.es

Lesson 5
Complex Data Types

Programming

Grade in Industrial Technology Engineering

Outline

1. Introduction
2. Arrays: definition and use
3. Arrays and pointers
4. Character strings
5. Structures: definition and use

Outline

1. **Introduction**
2. Arrays: definition and use
3. Arrays and pointers
4. Character strings
5. Structures: definition and use

Basic data types (Lesson 3)

- Single "cell"
- Store a single value
- Types:
  - Numerical: integer, real, etc.
  - Characters
  - Pointers

Complex data types

- Several "cells"
- Internal structure
- Types:
  - **Arrays** (vectors, matrices)
  - **Character strings**
  - **Structures** (sometimes [wrongly] called registers)

Outline

1. Introduction
2. **Arrays: definition and use**
3. Arrays and pointers
4. Character strings
5. Structures: definition and use

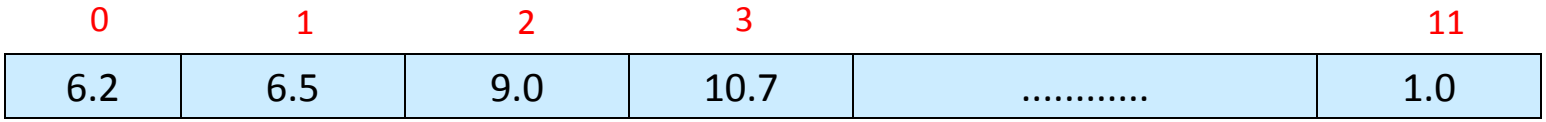
2. Arrays: definition and use

Example

Data structure to store the mean temperature of Madrid of each month of the year

```
float temperature[12]
```

One dimension (vector)



Array name: temperature

Array size: 12

Type of the elements of the array (`float`)

Same identifier: `temperature`

Each element has a different value

Each element is identified with and index: [0], [1], …, [11]

E.g.: Assign March temperature (third month)

```
temperature[2] = 9;
```

Beware!
Elements start counting at position 0

Collection of *elements* of the same type named with the same global identifier

Individual elements of the array are accessed with an index that identifies the position of the array

The index is **ALWAYS** an integer expression

Multiple-dimension arrays

- One-dimension array: vector or list
- Two-dimension array: matrix
  - Table of *n* rows and *m* columns

2. Arrays: definition and use

Example

Data structure to store the mean temperature of Madrid of each day of the year 2010

```
float temperature[365]
```

One dimension (vector)

0	1	2	3		364
6.2	6.5	9.0	10.7	.....	1.0

Beware!

Accessing elements outside the legal range of the array index results in unexpected results (usually, a runtime error)

2. Arrays: definition and use

Example

Data structure to store the mean temperature of Madrid, Barcelona, Seville, and Valencia of each month of the year

`temperature_cities[4][12]`

Two dimensions (matrix)

	0	1	2	3		11
0	6.2	6.5	9.0	10.7	.....	1.0
1	12.1	10.4	17.0	18.1	.....	13.4
2	18.1	20.1	24.7	26.4	.....	12.1
3	16.1	17.8	18.1	20.2	.....	15.5

Each element is identified with two indexes

Row ← → ***Column***

[0][0], [0][1], …, [0][11]
[1][0], [1][1], …, [1][11]
…
[3][0], [3][1], …, [3][11]

E.g.: Assign February temperature of Seville (third city, second month)

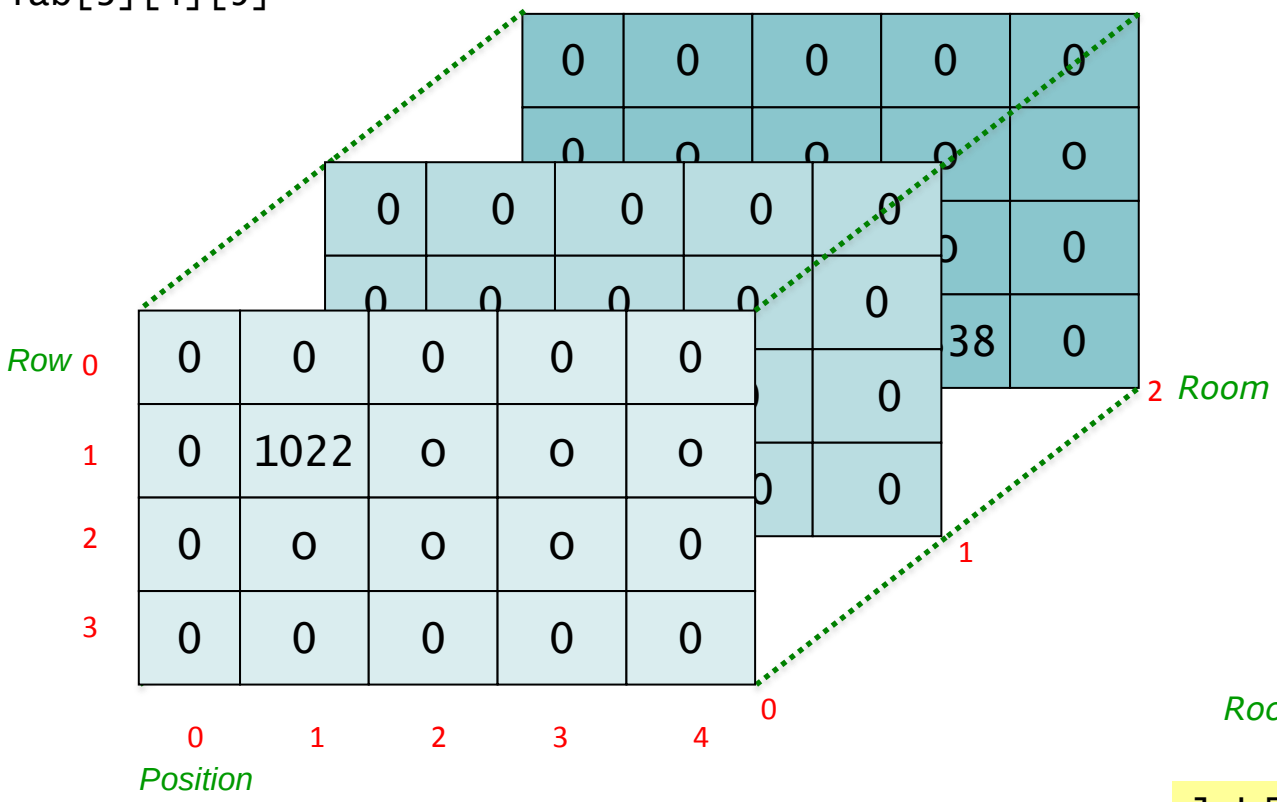
```
temperature_cities[2][1] = 14.3;
```

2. Arrays: definition and use

Example

Data structure to store information about the occupation of the computers of the faculty (NIA of the student)

```
int lab[3][4][5]
```




```
Room Row Position
lab[1][2][0] = 2525;
lab[2][3][3] = 3838;
lab[0][1][1] = 1022;
```

To **declare** an array, we have to specify:

- data type
- name
- number of dimensions
- number of elements per dimension

This is information is required by the compiler to automatically allocate memory for the array:

- *n* "variables" of the selected type are created and stored in **consecutive** address

Declaration of a one-dimension array

`<data type> <array name> [<size>] [= <init>];`

(`size` is an integer **LITERAL,** usually previously defined with **`#define`**)
[NOTE: `size` can be a variable in *C99*, but we will not use this feature]

Data type can be any C data type

Examples:

```
// usual array declaration
float temperature[365];
int num[10];
char vowels[5] = {'a', 'e', 'i', 'o', 'u'};

// declaration + initialization (size is not required!)
int countdown[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

Declaration of a **multiple-dimension array**

`<data type> <array name> [size][size]…[size];`

(`size` is an integer **LITERAL,** usually previously defined with **`#define`**)
First size value is optional if the array is initialized in the declaration

Examples:

Two-dimension array to store a 800x600 image (grayscale)

```
int image[600][800];
```

Three-dimension array to store information about the occupation of the computer labs in the faculty

```
int lab[20][15][10];
```

A two-dimension array can be regarded as a one-dimension vector whose elements are vectors

(This idea is extensive to more dimensions)

Value assignment

Elements are identified with the corresponding index

```
temperature[2] = 9;
temperature_cities[2][1] = 14.3;
image[0][5] = 225;
lab[1][4][2] = 1000123;
```

Only one value can be assigned to an element of the array

Assigning a value to all the elements of the array is not allowed

```
image = 0;
```

WRONG

Initialization

It is possible to initialize an array along with its declaration

Similar to basic data types

One-dimension arrays

Defining the size of the array

```
int numbers[6] = {4, 8, 15, 16, 23, 42};
```

Without defining the size of the array

```
int numbers[] = {4, 8, 15, 16, 23, 42};
```

Multiple-dimension arrays

Size must be defined (first dimension size can be omitted)

Each dimension sub-array is enclosed with { }

```
int matrix[2][3] = {
    {4, 8, 15} ,
    {16, 23, 42}
};
```

Array indexing with expressions

Element values can be accessed by using the proper indexes

```
int a;
a = (numbers[0] + numbers[1]) / 2;
```

Any integer expression can be used as index

```
int i = 1, j = 2;
float t = temperature_cities[i*2 + 1][j];
temperature_cities[i][j+1] = 13.3;
```

It is very common to use loops to access to the elements of an array

```
int i=0;
for(i=0; i<6; i++)
    printf("Value at position %i is %i \n", i, numbers[i]);
```

If we access to a position of an array that has not been allocated, we get a runtime error

```
int i;
int numbers[] = {4, 8, 15, 16, 23, 42};

numbers[6] = 100;

for (i=0; i<=5; i++) {
    printf("Element %i: %i \n", i, numbers[i]);
}

for (i=0; i<=6; i++) {
    printf("Element %i: %i \n", i, numbers[i]);
}
```

ERROR: Trying to access to position 6 of an array of 6 positions (indexes start from 0)

OK: Index 'i' takes values from 0 to 5

ERROR: Index 'i' takes values from 0 to 6

Complete arrays cannot be directly assigned

- Compilation error

Complete arrays cannot be directly compared

- The addresses of the first elements are compared

Complete arrays cannot be directly printed

- The address of the first element is printed

In general, **array data must be processed element by element**

Array values must be read one-by-one

```
int image[600][800];

printf("Enter pixel value of 3rd row - 2nd column: ");
scanf("%i", &image[2][1]);
```

Reading all the values of an array

```
int matrix[4][5];

int i, j;
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        printf("Enter value (%i, %i): ", i, j);
        scanf("%i", &matrix[i][j]);
    }
}
```

Array values must be printed one-by-one

```
int image[600][800];

printf("Printing pixel value of 3rd row - 2nd column: %i", image[2][1]);
```

Printing all the values of an array

```
int matrix[4][5];

int i, j;
for (i=0; i<4; i++) {
    for (j=0; j<5; j++) {
        printf("Printing value (%i, %i): %i", i, j, matrix[i][j]);
    }
}
```

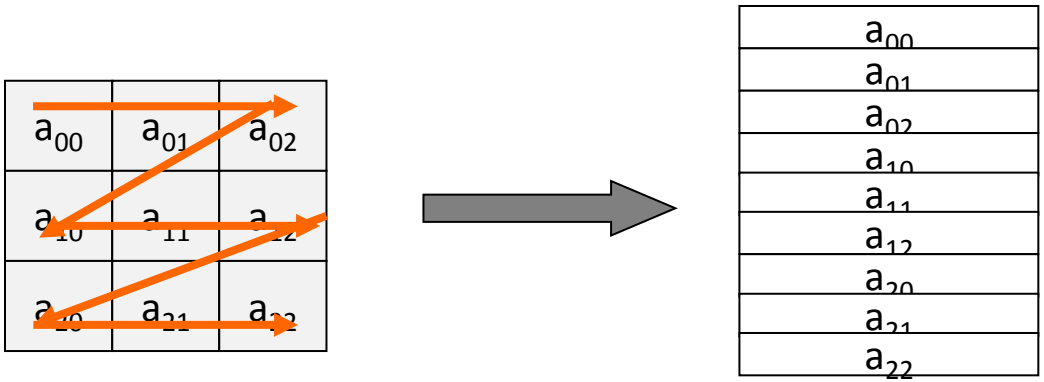
How can we compare two one-dimension arrays *a* and *b*?

How can we copy a one-dimension array *a* into another one-dimension array *b*?

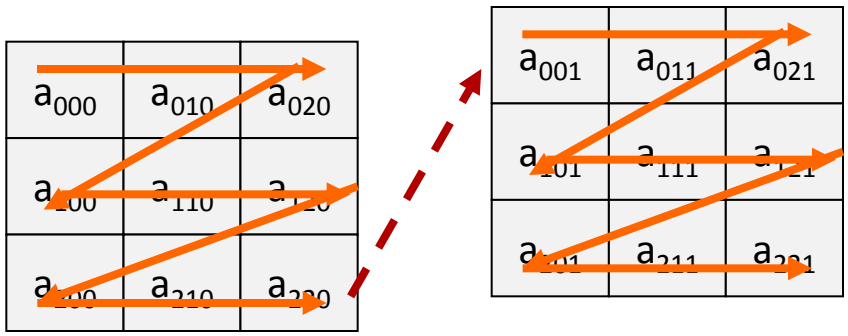
Array elements are stored in consecutive cells of the memory

One-dimension arrays: elements have consecutive addresses

Two-dimension arrays: Row major (the first row is stored, the second row is stored, etc.)



Three-dimension arrays: the first "page" is stored, the second page is stored, etc.



Outline

1. Introduction
2. Arrays: definition and use
3. **Arrays and pointers**
4. Character strings
5. Structures: definition and use

Arrays and pointers are very closely related

The name of an array is the address of the memory cell that stores the first element of the array

The elements of an array can be accessed with pointer operators from the first address

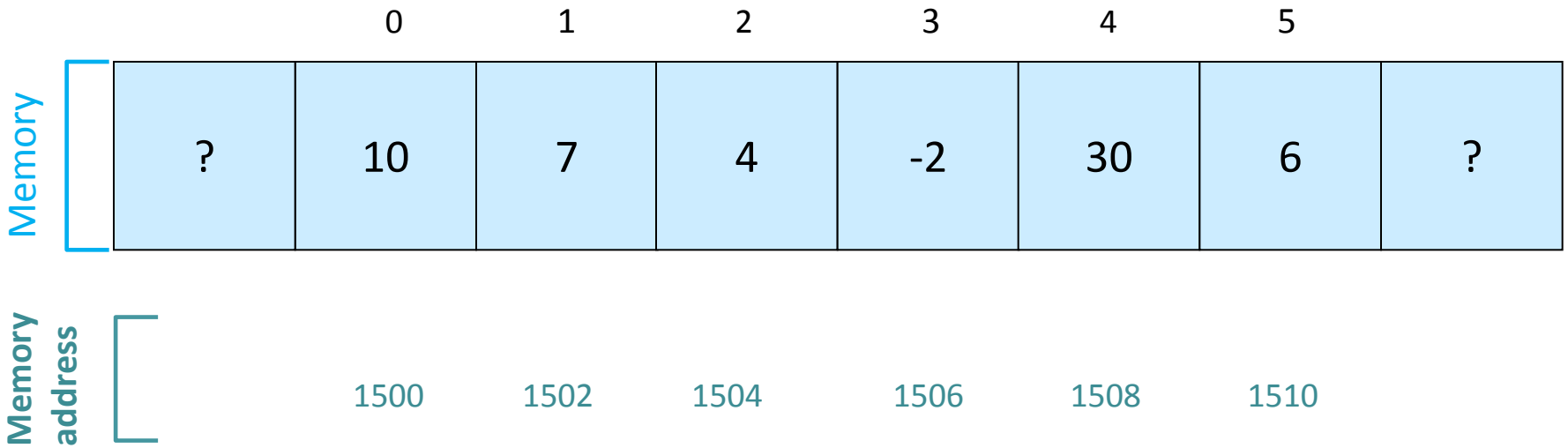
Let us assume the array definition:

```
int list[6] = {10, 7, 4, -2, 30, 6};
```

3. Arrays and pointers

Example

Assuming 2 bytes per integer `(sizeof(int) == 2)`

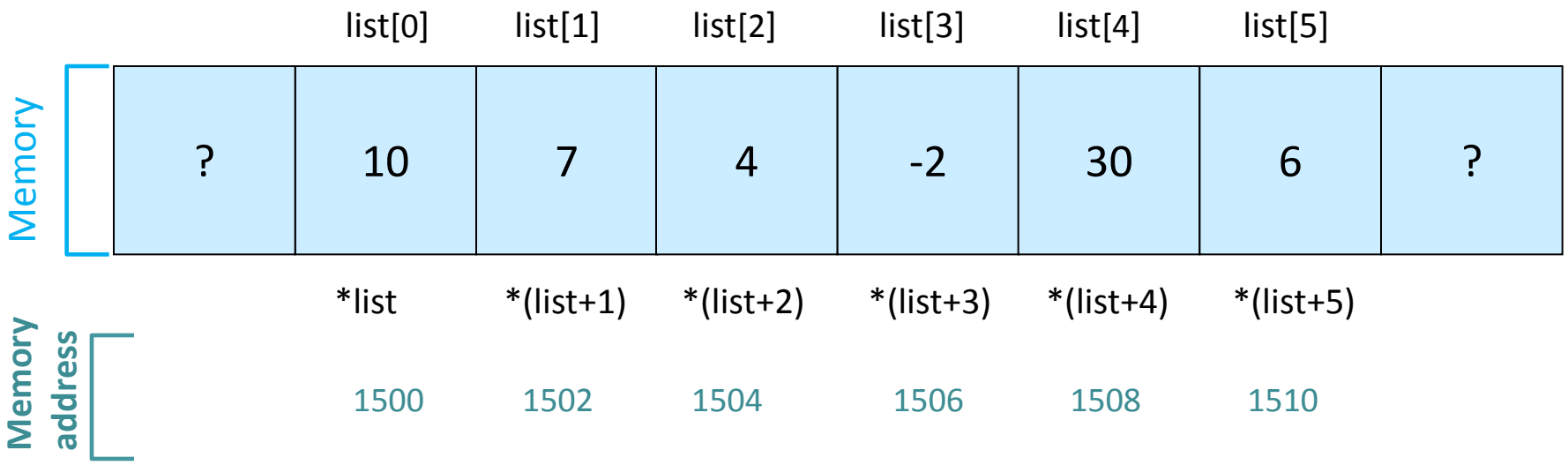


3. Arrays and pointers

Example

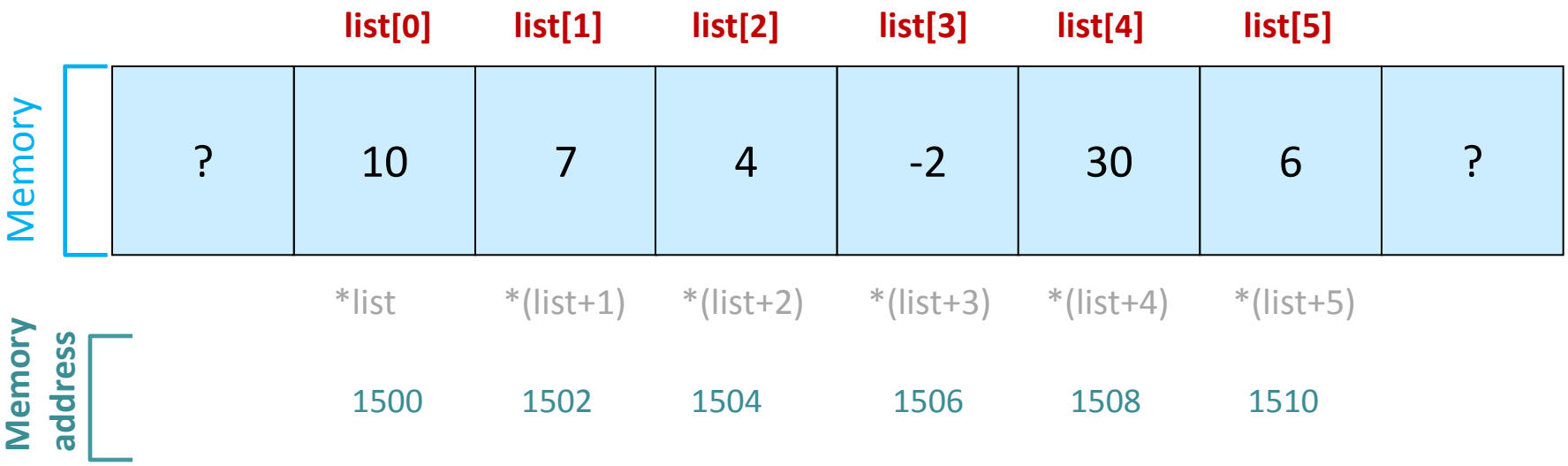
Array elements can be accessed in two different ways:

A. Index-based access
B. Address-based access



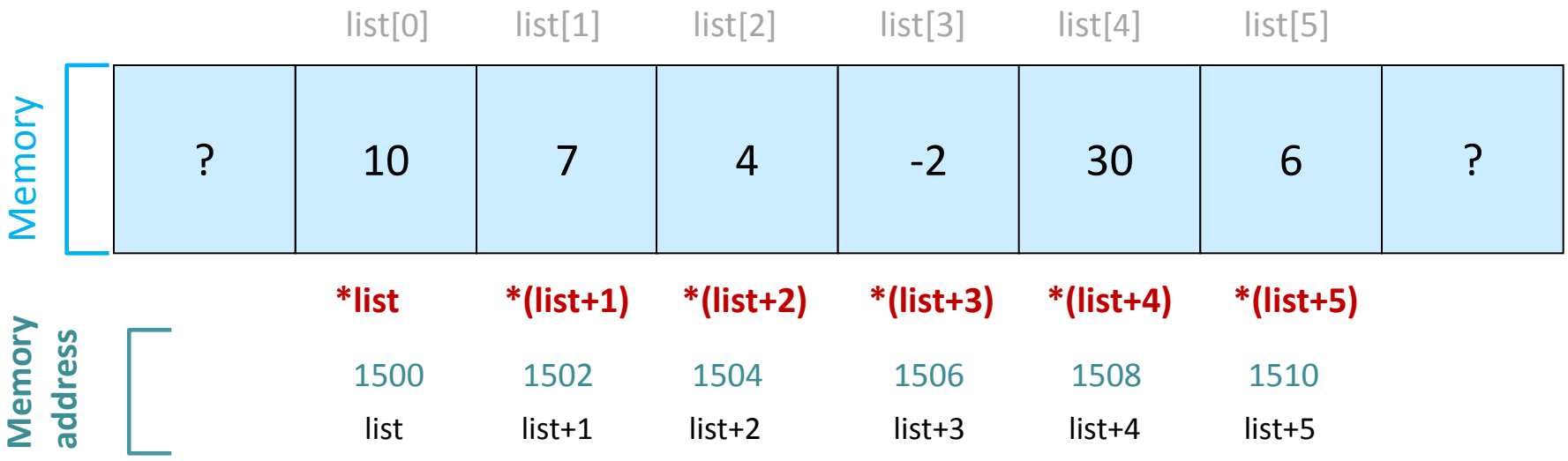
A. Index-based access

```
int x = list[1];    /* x = 7   */
int y = list[0];    /* y = 10  */
```



B. Address-based access

```
int x    = *(list+1);   /* x  = 7 */

int * p1 = &list[0];    /* p1 = 1500 ( &list[0] is the same as list ) */
int * p2 = list;        /* p2 = 1500 */
```

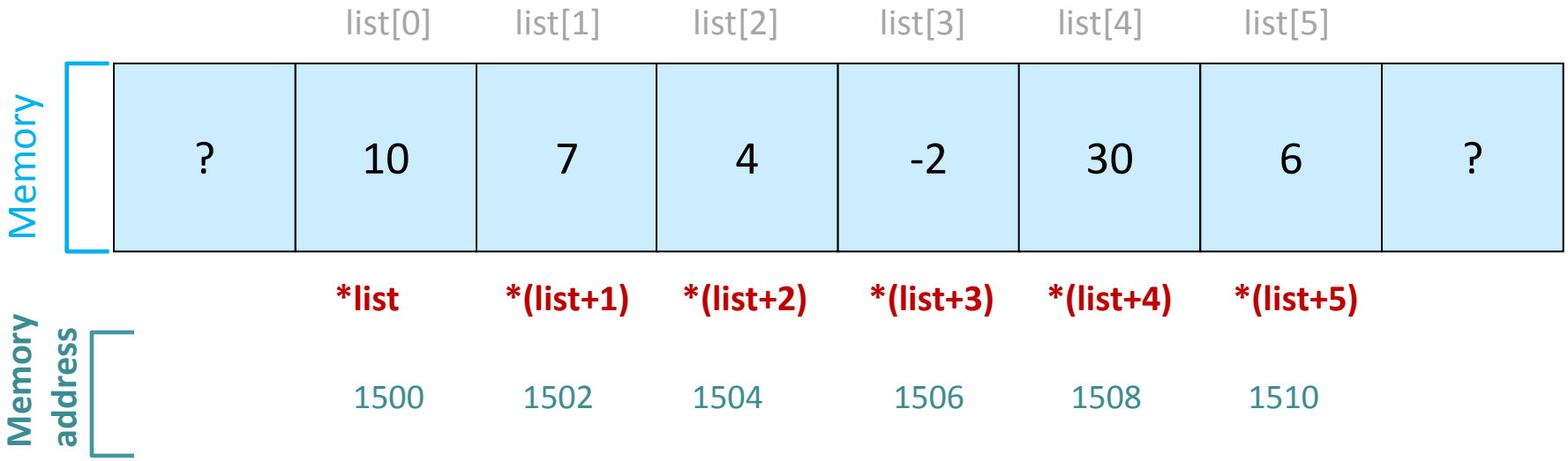


***list+1* is 1502!** Pointer arithmetic

Adding 1 to *list* means "point to the next value of the type of the pointer *list*"
The compiler automatically calculates the offset, which is 2 (assuming 2 bytes per int)

B. Address-based access

```
int * p4 = &list[3];          /* p4 = 1506 */
int * p5 = list + 3;          /* p5 = 1506 */
int z = *(p5+1);              /* z  = 30   */
int w = (*p5)+2;              /* w  = 0    */
```



Outline

1. Introduction
2. Arrays: definition and use
3. Arrays and pointers
4. **Character strings**
5. Structures: definition and use

A **string literal** is a sequence of characters delimited by double quotation marks

```
"Hello world!"
```

Blank spaces may appear

Escape sequences can be used (preceded by \)

`\n` : new line

`\"` : quotation mark character

A string variable is a one-dimension array of char with some particular properties:

- Strings contain meaningful text (name, phrase, etc.)
- There is a *null* character at the end of the string
  - The null character is `'\0'` (ASCII code 0)
    - Therefore, the length of an array that stores a string is (at least) the number of characters of the string plus one
    - E.g.: to store "table" string, 6 bytes are required –an array of size 6 or larger must be used to store "table\0"
    - Nevertheless, to store "table" as a plain array of characters, only 5 bytes are required
  - The null character marks the end of the useful text in the string
  - It is automatically when added when a string is initialized or read

```
char city[]  = "Madrid";  // M | a | d | r | i | d | \0
char city[9] = "Madrid";  // M | a | d | r | i | d | \0 | ? | ?
```

    - The characters of the string are stored in the first six elements of the array
    - The seventh element stores the '\0'

Strings have special functions to be managed as a whole: read, write, etc.

C mixes regular character arrays and strings of characters:

Normal array of characters

```
char a[5]; // No initialization
char a[5] = {'h', 'e', 'l', 'l', 'o'}; // element-by-element init.
char a[]  = {'h', 'e', 'l', 'l', 'o'}; // element-by-element init.
                                       // without size
```

String of characters

Size is specified: enough room for all characters must be available

```
char a[6] = "hello";      // \0 is automatically added
char a[6] = "hello\0";    // Explicitly adding \0 is not necessary
char a[5] = "hello";      // Wrong! Unexpected result

char a[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Size is not specified: room is automatically allocated for elements + \0

```
char a[] = "hello";       // 6 elements, since \0 is automatically added

char a[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

4. Character strings

Character strings and character arrays

strings.c

```
char string1[6] = "hello";       // string
char string2[]  = "hello";       // string
char string3[5] = {'h', 'e', 'l', 'l', 'o'};          // array of characters
char string4[]  = {'h', 'e', 'l', 'l', 'o'};          // array of characters
char string5[]  = {'h', 'e', 'l', 'l', 'o', '\0'};    // string
char string6[5] = "hello";       // bad formed string, unexpected results

printf("Character string string1: %s \n", string1);
printf("string1[0]: %c \n", string1[0]);
printf("string1[1]: %c \n", string1[1]);
printf("string1[2]: %c \n", string1[2]);
printf("string1[3]: %c \n", string1[3]);
printf("string1[4]: %c \n", string1[4]);
printf("string1[5]: %c \n", string1[5]);
printf("string1[6]: %c \n", string1[6]);     // out of bounds!

printf("Character string string2: %s \n", string2);
printf("string2[0]: %c \n", string2[0]);
printf("string2[1]: %c \n", string2[1]);
printf("string2[2]: %c \n", string2[2]);
printf("string2[3]: %c \n", string2[3]);
printf("string2[4]: %c \n", string2[4]);
printf("string2[5]: %c \n", string2[5]);
printf("string2[6]: %c \n", string2[6]);     // out of bounds!

printf("Character string string3: %s \n", string3);
printf("string3[0]: %c \n", string3[0]);
printf("string3[1]: %c \n", string3[1]);
printf("string3[2]: %c \n", string3[2]);
printf("string3[3]: %c \n", string3[3]);
printf("string3[4]: %c \n", string3[4]);
printf("string3[5]: %c \n", string3[5]);     // out of bounds!
```

```
Character string string1: hello
string1[0]: h
string1[1]: e
string1[2]: l
string1[3]: l
string1[4]: o
string1[5]:
string1[6]: X
```

```
Character string string2: hello
string2[0]: h
string2[1]: e
string2[2]: l
string2[3]: l
string2[4]: o
string2[5]:
string2[6]: h
```

```
Character string string3: hellohello
string3[0]: h
string3[1]: e
string3[2]: l
string3[3]: l
string3[4]: o
string3[5]: h
```

Complete string assignment can be only performed in the declaration of the string

```
char str[] = "hello";      ✔
str = "goodbye";           ✗
```

Element-by-element assignment can be performed

```
char str[50] = "helloworld";
printf("String: [%s]\n", str);

str[0] = 'b';
str[1] = 'y';
str[2] = 'e';
str[3] = '\0';
printf("String: [%s]\n", str);
```

```
String: [helloworld]
String: [bye]
```

The **size** of the array is different from the **length** of the string!

`printf` and `scanf`

Use the string format specifier `%s`

In the call to `scanf`, the '&' operator is not used

The name of the string is already a pointer

`scanf` with `%s` ends reading when a blank space is found

```
char str[100];
printf("Enter string: ");
scanf("%s", str);
printf("String is: %s", str);
```

```
Enter string: hello world
String is: hello
```

Use `scanf("%[^\n]", s)` to read until the end of line

Other functions (unsafe, not recommended)

`gets(char [] str)`: reads a string until '\n' is found into `str`

'\n' is not included in `str`

`puts(char [] str)`: prints the string `str`

String copy

`strcpy(char dest[], char src[])`

- Function included in `<string.h>`
- Copies the string *src* into the string *dest*
- The destination string must be large enough to store all the characters of the source

```c
#include <stdio.h>
#include <string.h>

int main (void) {

    char str1[] = "helloworld";
    char str2[50];

    printf("String 1: [%s]\n", str1);
    printf("String 2: [%s]\n", str2);
    strcpy(str2, str1);
    printf("String 1: [%s]\n", str1);
    printf("String 2: [%s]\n", str2);

    return 0;

}
```

```
String 1: [helloworld]
String 2: [\371\277_\377]
String 1: [helloworld]
String 2: [helloworld]
```

4. Character strings

Other operations

`<string.h>`

Comparison

```
int strcmp(char str1[], char str2[])
```

Returns:
- 0 if *str1* is equal to *str2*
- 1 if *str1* is *greater* than *str2*
- -1 if *str1* is *less* than *str2*

Joining

```
char [] strcat(char str1[], char str2[])
```

Modifies *str1* by appending *str2* at the end
Returns *str1* (modified)

Length

```
int strlen(char str[])
```

Returns the number of *valid* characters in *str*, excluding \0

Finding

```
char * strchr(char str1[], char c)
```

Finds *c* in *str1*
Returns the address of the first occurrence of *c* in *str1* (NULL if not found)

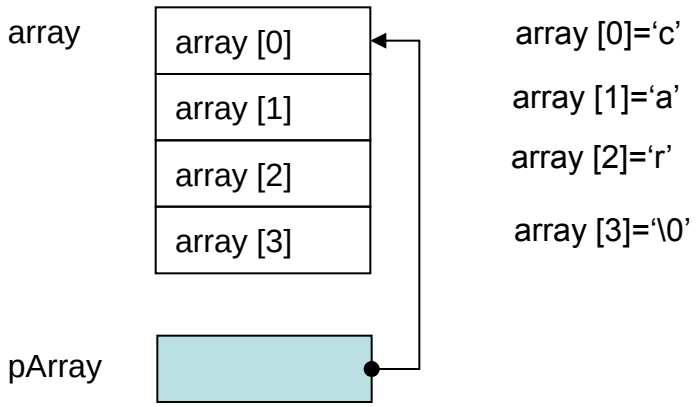
```
char * strstr(char str1[], char str2[])
```

Finds *str2* in *str1*
Returns the address of the starting occurrence of *str2* in *str1*

The relation between pointers and strings is the same as for pointers and regular arrays

- The name of the string is the address of the first element

```
char array[] = "car";
char *p_array = array;
```



Bibliography

Recommended lectures

Basic

- Stephen G. Kochan. *Programming in C.* Sams, 2004 (3rd Edition), Programming in C – Chapter 7
- Ivor Horton. *Beginning C: From Novice to Professional*. Apress, 2006 (4th Edition) – Chapter 5, Chapter 6 (sections 1, 2, 3, 4)

Additional information

- Ivor Horton. *Beginning C: From Novice to Professional*. Apress, 2006 (4th Edition) – Chapter 7 (sections 1, 2, 3)
- Stephen G. Kochan. *Programming in C.* Sams, 2004 (3rd Edition), Programming in C – Chapter 10 (all but section 5 on Character Strings, Structures, and Arrays)

Outline

1. Introduction
2. Arrays: definition and use
3. Arrays and pointers
4. Character strings
5. **Structures: definition and use**

5. Structures: definition and use

Definition

A **structure** is *a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling*. (Kernighan & Ritchie, *The C Programming Language*)

Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

Examples:

- Entry of an address book
  - Name
  - Surname
  - Phone number
  - E-mail
- 3D point
  - x coordinate
  - y coordinate
  - z coordinate

The components of a structure are named **members** or fields

- The members of the `AddressBookEntry` structure are `name`, `surname`, `phone` and `email`
- The members of the `Point3D` structure are `x`, `y`, `z`

Structure members are accessed by using their name – instead of the index, **which is used only for arrays**

Structures can be used as a whole or by accessing to the individual members

Structures are sometimes called registers, but this name must be avoided

Syntax

```
struct <structure_name> {
    <type> <member>;
    <type> <member>;
    …
    <type> <member>;
};
```

The `struct` definition is placed **out of the main method**

Example

```
struct Point3D {
    float x;
    float y;
    float z;
};
```

A structure declaration **creates a new data type**. It is a template for new data, rather than memory allocation to store data

New variables of the structured type can be declared and used

A `Point3D` variable can be declared

Syntax

```
struct <structure_name> <variable_name>;
```

Example

```
struct Point3D p1;
```

(`p1` is a new variable of the structured type `Point3D`)

To access to the members of a structure variable, the **point operator** (.) is used

To access to the coordinate `x` of the `Point3D p1` we use the syntax:

```
p1.x
```

`p1` is the name of the structure variable

`.` is the member access operator

`x` is the **name of the member** of the structure

Similarly

```
p1.y
p1.z
```

Values for the members of a structure can be assigned directly in the declaration, by structure assignment, or by member-to-member assignment.

1. Declaration and initialization of structures

Values are directly assigned

```
struct Point3D point1 = {2.1, 3.4, 9.8};
```

[Using *compound literals* is also possible in other parts of the code, but this is not studied in this course.]

2. Structure assignment

A copy of the structure is created

```
struct Point3D point2;
point2 = point1;
```

3. Individual member assignment

A copy of the member is assigned

```
struct Point3D point3 = {1.1, 2.3, -1.4};
struct Point3D point4;
point4.x = point3.x;
point4.y = point3.y;
point4.z = point3.z;
```

5. Structures: definition and use

Example

```c
#include <stdio.h>

struct Point3D {
    float x;
    float y;
    float z;
};

int main (void) {
    struct Point3D pA, pB;

    printf("Enter x coordinate of pA: ");
    scanf("%f", &(pA.x));
    printf("Enter y coordinate of pA: ");
    scanf("%f", &(pA.y));
    printf("Enter z coordinate of pA: ");
    scanf("%f", &(pA.z));

    printf("\nEnter x coordinate of pB: ");
    scanf("%f", &(pB.x));
    printf("Enter y coordinate of pB: ");
    scanf("%f", &(pB.y));
    printf("Enter z coordinate of pB: ");
    scanf("%f", &(pB.z));

    if ( (pA.x == pB.x) && (pA.y == pB.y) && (pA.z == pB.z) ) {
       printf("pA and pB are the same point");
    } else {
       printf("pA and pB are different points");
    }

    return 0;
}
```

(See:. **Point3DExample.c**)

distance between pA and pB?

An **array of structures** stores a **list of entities**

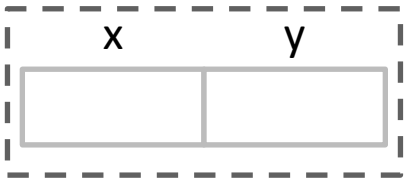
Different from an array inside a structure!

Syntax

```
struct <structure_name> <array_name>[<size>];
```

The entities of the array are grouped together and can be accessed with indexes as any other array element

```
struct Point2D {
    float x;
    float y;
};
```




```
int main(void) {

    /* Declare array of structure Point2D */
    struct Point2D points[3];
```

	x	y
points[0]		
points[1]		
points[2]		

```
    /* Access to array structure members */
    points[0].x = 1.0;
    points[0].y = 2.1;

    points[1].x = 3.1;
    points[1].y = 4.5;

    points[2].x = 5.0;
    points[2].y = 4.1;
```

	x	y
points[0]	1.0	2.1
points[1]	3.1	4.5
points[2]	5.0	4.1

The **members** of a structure can be basic or complex data types:

integer, character, pointer…
array, structure

Example

Entry of an address book (arrays inside a structure)

```
struct AddressBookEntry {
        char name[256];
        char surname[256];
        char email[256];
        int phone[4];
};
```

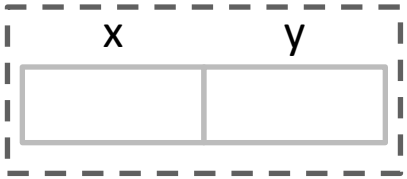
Example (structures inside a structure, array of structures inside a structure)

```
struct Point2D {
    float x;
    float y;
};

struct Triangle {
    struct Point2D a;
    struct Point2D b;
    struct Point2D c;
};

struct Dodecahedron {
    struct Point2D points[12];
};
```

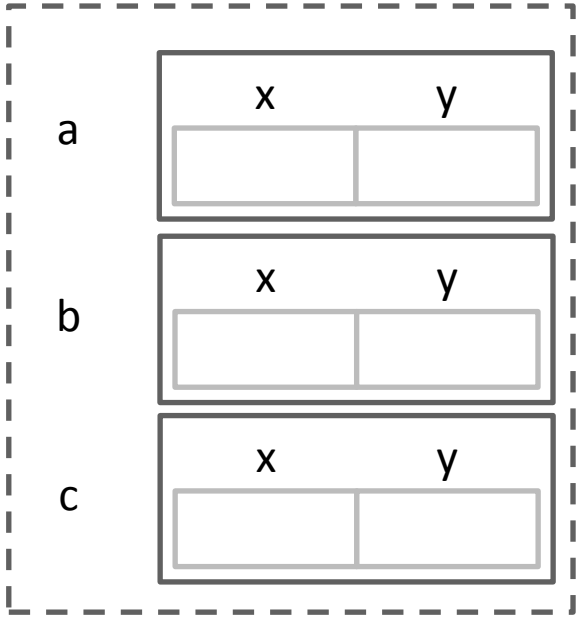
To access to nested members, the point operator is used several times. If arrays are involved, brackets must be used

(See: **Shapes1.c, Shapes2.c**)

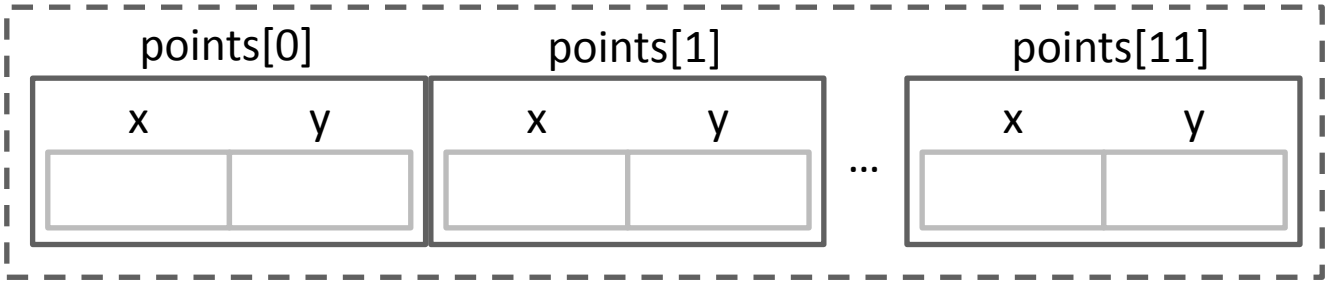
```
/* Structure */
struct Point2D {
    float x;
    float y;
};
```




```
/* Structure with nested structure*/
struct Triangle {
    struct Point2D a;
    struct Point2D b;
    struct Point2D c;
};
```



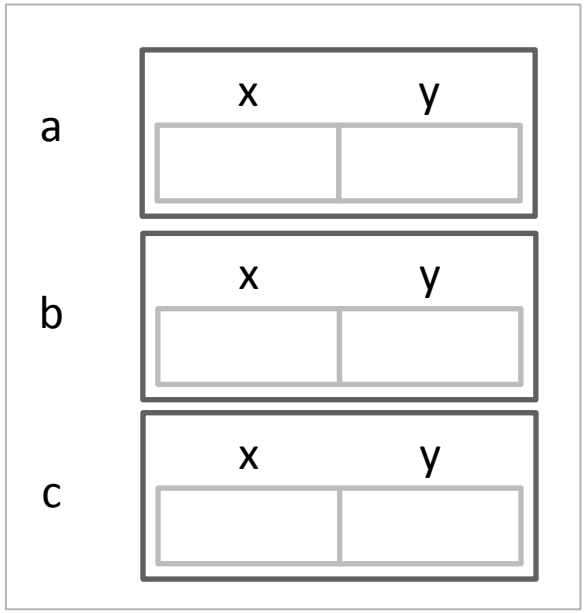

```
/* Structure with nested array of structures */
struct Dodecahedron {
    struct Point2D points[12];
};
```



5. Structures: definition and use

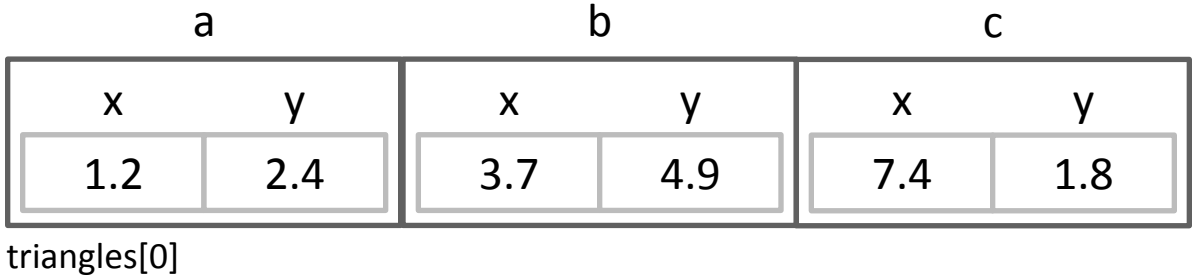
Complex structure members

```
int main(void) {
    …
    struct Triangle tri;
    …
    /* Create triangle with vertex: (0, 0) */
    tri.a.x = 0;
    tri.a.y = 0;
```

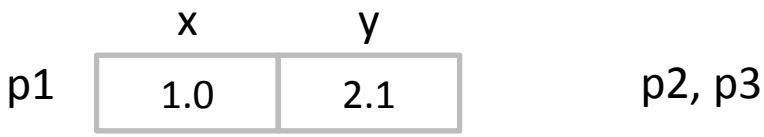


```
/* Declare array of complex structure Triangle */
struct Triangle triangles[2];
```

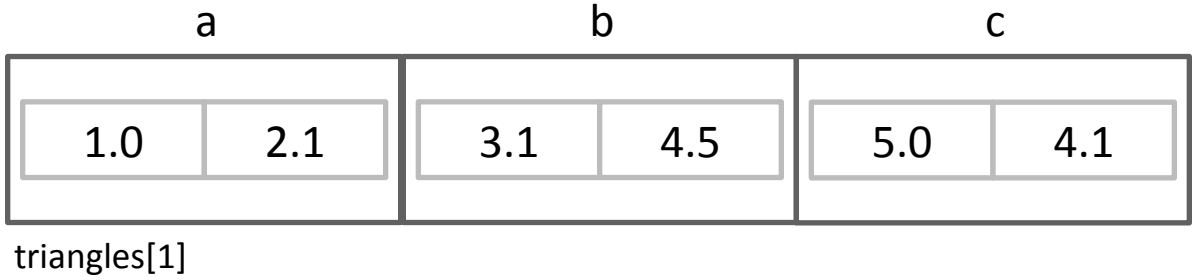
```
triangles[0].a.x = 1.2;
triangles[0].a.y = 2.4;
triangles[0].b.x = 3.7;
triangles[0].b.y = 4.9;
triangles[0].c.x = 7.4;
triangles[0].c.y = 1.8;
```




```
struct Point2D p1 = {1.0, 2.1};
struct Point2D p2 = {3.1, 4.5};
struct Point2D p3 = {5.0, 4.1};
```




```
triangles[1].a = p1;
triangles[1].b = p2;
triangles[1].c = p3;
```

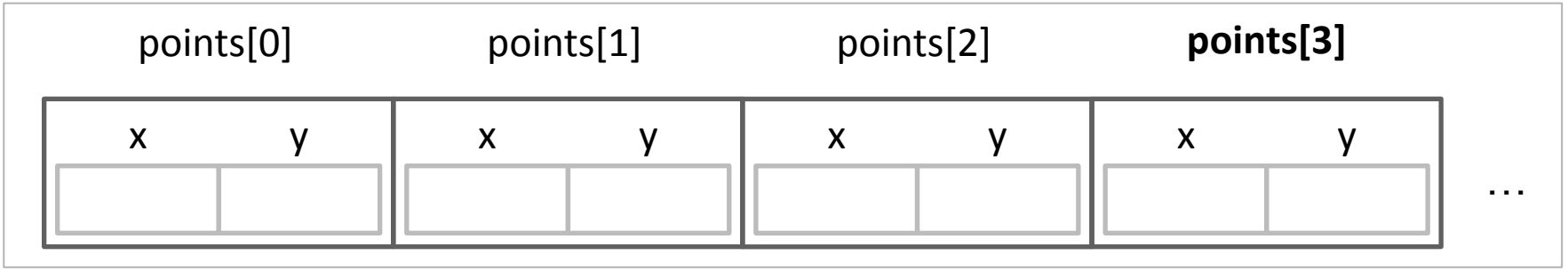


5. Structures: definition and use

Complex structure members

```
/* Structure with nested structure*/
struct Dodecahedron {
    struct Point2D points[12];
};

int main(void) {
      …
      struct Dodecahedron dod;
      …
      /* Assign point 3 of dodecahedron */
      dod.points[3].x = 1;
      dod.points[3].y = 1;
```

dod



Pointers to structures can be declared and used

Syntax

```
struct <structure_name> *<pointer_name>;
```

Example

```
struct Point2D *pp;
pp = &p1;  // p1 has been declared as struct Point2D p1
```

Pointers allow access to the structure members with the arrow (->) operator

Example

```
printf("\nPoint 1: (%f, %f)", pp->x, pp->y);
```

Bibliography
Recommended lectures

Basic

- Stephen G. Kochan. *Programming in C.* Sams, 2004 (3rd Edition), Programming in C – Chapter 9
  - Skip *Functions and Structures* section until next lesson
  - Notice that some examples use functions

Additional information

- Ivor Horton. *Beginning C: From Novice to Professional*. Apress, 2006 (4th Edition) – Chapter 11
- Stephen Prata. *C Primer Plus.* Sams, 2004 (5th Edition) – Chapter 14 (until *Structures: What Next?*)

Outline

1. Introduction
2. Arrays: definition and use
3. Arrays and pointers
4. Character strings
5. Structures: definition and use