Computers Programming
Course 9

Iulian Năstac

Recap from previous course

Cap. Procedural programming - Functions

- **Procedural programming** is a specific feature of high-level programming languages. If you want to execute a set of instructions with different data or in different places, then you can put all of them into a subroutine that will be called by a jump whenever needed. After this jump, will return to the instruction following the one in which he leaped and therefore differs from jumps performed with **goto** statement.

- This type of sequence organization above bears different names in programming languages: subprogram, subroutine, procedure, function, etc.

Recap from previous course

1. Function declarations

a) classic style :

type name_function() ;

No information on the parameters, so there may be no error checking.

b) modern syle :

type name_function(inf_p1, inf_p2, ... etc.);

where inf_p is declaring type of the variable used as an argument to the function.

2. Complete description of functions

a) Classic style

type name_function(name of parameters)
description of parameters
{
...
}

b) Modern style

type name_function(inf_p, inf_p, ...)
{
...
}

Recap from previous course

The logic of writing a program in "C" is:

- Including the header file;
- Declarations of global variables and constants;
- Declarations of functions;
- Describing the main function;
- Describing all functions declared.

3. Zone of influence for a C function

Definition: The sphere of influence of a language means the set of rules that determine how a code sequence can access another sequence of code or data.

In the C language, within the **sphere of influence** are adhered to the following principles:

1. Each function contains its own block of code and no other instruction from another function can not have access to it, excepting the function call. We cannot use **goto** to jump between functions.

2. If a function does not use global variables or data, it cannot affect other parts of the program. The code and data of a function cannot interact directly with the code and data of another function.

Principles (cont.)

3. In C programming, all functions have the same level of sphere of influence. C is not technically a sort of block structured language. It is forbidden to define a function in another function (but a instruction inside a function can call another function).

Principles (cont.)

4. Variables defined in a function are called local variables. A local variable is created within a function (or a inner block) and is destroyed on exit. So local variables do not retain their value between different calls of that function. The only exceptions are such local variables that are declared with **static**, which are not destroyed after leaving that function but are limited, as sphere of influence, inside the function.

Principles (cont.)

5. We want to call a function inside another function, by using an argument that is a variable.

Example:

```
type f1 (    )
{ type x;
   ...
   f2(x);
   ...
}
```

Function f2 gets a copy of the argument value. What happens inside the function f2 has no effect on the variable used as parameter (i.e. ***x*** from the function f1).

6. Different functions can have local variables with the same name which do not lead to mutual influence between functions.

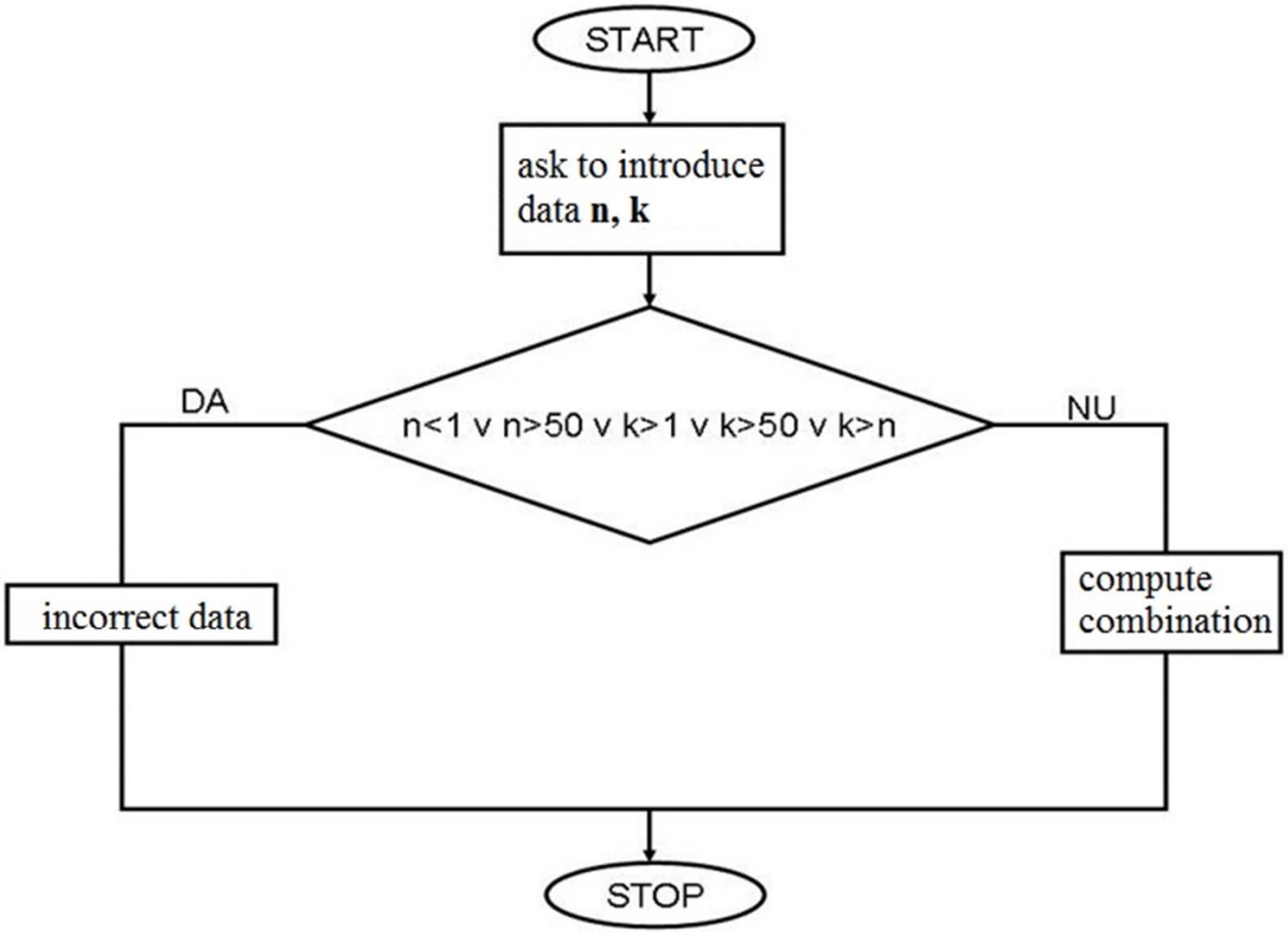
Example:

We will write a program that requires the introduction of variables n and k (as integers) from the keyboard and check if they belong to the interval [1,50]. The program calculates and provides the result:

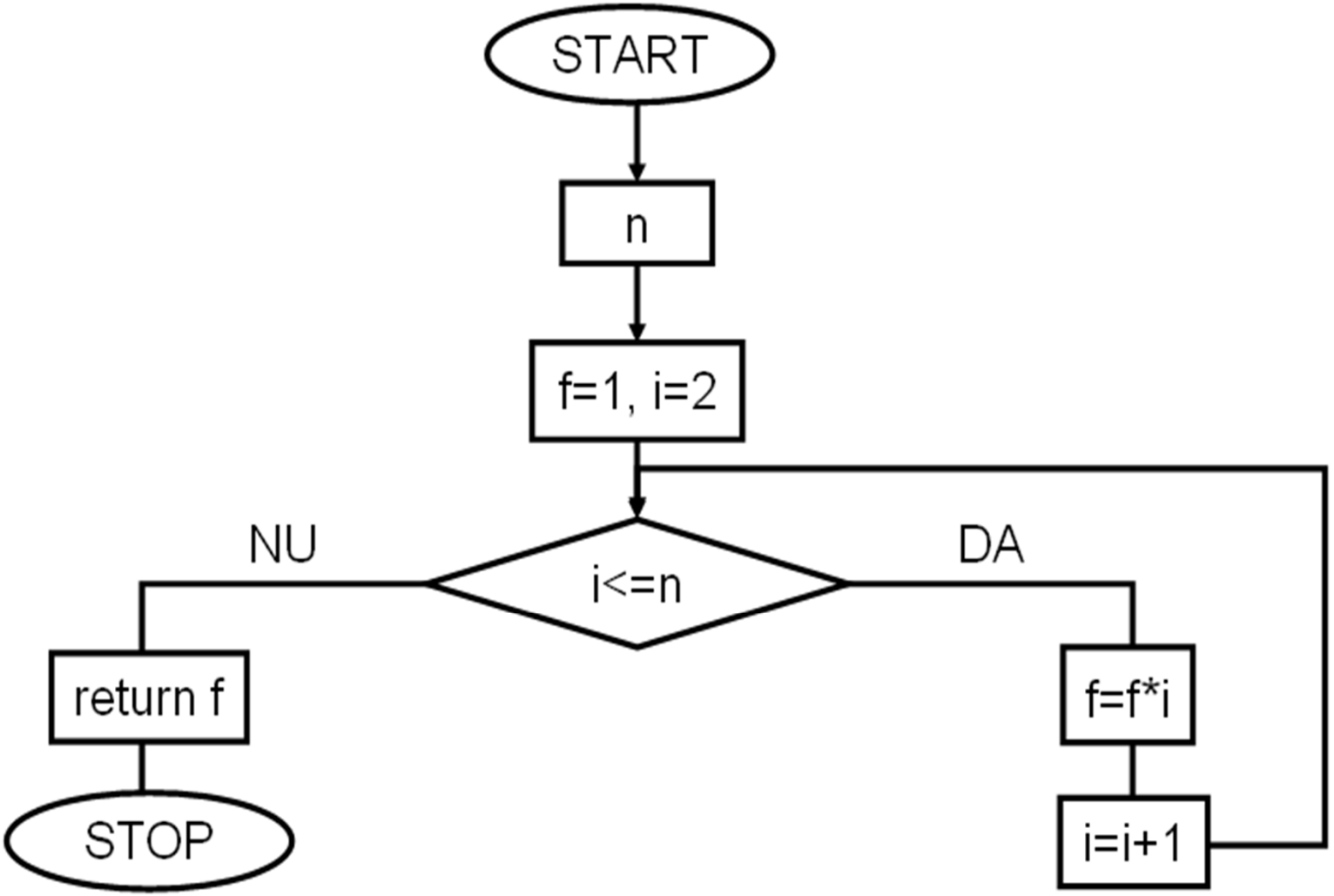
$$C_n^k = \frac{n!}{k!(n-k)!}$$

It must be verified that: k<n.

Flowchart of a program which shows only the evolution of the function ***main***.



The flowchart of ***fact*** function is:



```
#include <stdio.h>
double fact(int n);
int main(int)
{ int k,n;
  printf("\n Enter n=");
  scanf("%d",&n);
  printf("\n Enter k=");
  scanf("%d",&k);
  if (n<1||n>50||k<1||k>50||k>n)
          printf("\n Incorrect data");
  else printf("\n Result = %g",fact(n)/(fact(k)*fact(n-k)));
}
double fact(int n)
{ double f;
  int i;
  for (i=2,f=1.0;i<=n;i++)
          f*=i;
  return f;
}
```

4. Calling through the arguments of a function

Calling a function in C can be realized (relative to the nature of his arguments) in two ways:

- by value;

- by reference.

a. Calling by value

If data is passed by value, the data is copied from the variable used in (for example main()) to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

b. Calling by reference

When the function uses a pointer parameter then in that formal parameter it is actually copied an address of the memory. The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

Example of calling by value:

```
#include <stdio.h>

void call_by_value(int x)
{       printf("Inside call_by_value x = %d before adding 10.\n", x);
        x += 10;
        printf("Inside call_by_value x = %d after adding 10.\n", x);
}

int main()
{       int a=10;

        printf("a = %d before function call_by_value.\n", a);
        call_by_value(a);
        printf("a = %d after function call_by_value.\n", a);
        return 0;
}
```

Example of calling by reference:

```
#include <stdio.h>
...
void taking_over(float*p1,float*p2);
...
int  main()
 {
    float x, y, …;
    taking_over (&x, &y);
   ...
 }

void taking_over(float*p1, float*p2)
  {
     printf (" \nIntroduce the first number:  ");
     scanf ("%f", p1);
     printf ("\nIntroduce the second number:  ");
     scanf ("%f", p2);
  }
```

Example 2 of calling by reference:

```
#include <stdio.h>

void call_by_reference(int *y)
{       printf("Inside call_by_reference y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}

int main()
{       int b=10;

        printf("b = %d before function call_by_reference.\n", b);
        call_by_reference(&b);
        printf("b = %d after function call_by_reference.\n", b);

        return 0;
}
```

5. Values returned by a function

- A **return** statement causes execution to leave the current subroutine and resume at the point in the code immediately after where the subroutine was called, known as its return address.
- The return address is saved, usually on the process's call stack, as part of the operation of making the subroutine call.
- Return statements in C allow a function to specify a return value to be passed back to the code that called the function.

In C/C++, the syntax is:

return *exp*; /* where *exp* is an expression */

This is a statement that tells a function to return execution of the program to the calling function, and report the value of *exp*.

If a function has the return type void, the return statement can be used without a value, in which case the program just breaks out of the current function and returns to the calling one.

Notes:

1. In a function we can find one or more return instruction. Value or variable that follows the return statement must be of the function type (or converted to that type).

2. Often functions of void type lack return statement. However a function of type void can contain one or more simple return instructions, but unless accompanied by a return value.

Notes (continued):

3. A function cannot be assigned.
   Therefore:
   f(x,y)=100;
   is an incorrect statement. The parameters of a function cannot be initialized.

4. The type a function is not related to the types of its parameters.

5. Sometimes, the return statement is not necessary in specific functions (where the arguments are **called by reference**)… even if we expect something to be returned from such a function.

6. Recursive functions

- Recursion is a programming technique that allows the programmer to express operations in terms of themselves.
- In C, this takes the form of a function that calls itself.
- A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process".
- This makes it sound very similar to a loop because it repeats the same code, and in some ways it *is* similar to looping.

Notes:

- Recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task.
- Of course, it must be possible for the "process" to sometimes be completed without the recursive call.

Two variants (recursive and non-recursive) of the function for calculating the factorial of an integer:

Recursive:

```
int fact(int n)
{int f;
 if (n==1) return (1);
 f=fact(n-1)*n;
return(f);
}
```

Non-recursive:

```
int fact(int n)
{ int i,f;
 for(f=1,i=2;i<=n;i++)
      f*=i;
return f;
}
```

Notes:

1. A recursive routine does not significantly reduce code size nor improves memory usage.

2. Usually, recursive functions are slower than theirs iterative equivalent.

3. Misuse of recursion can cause overrunning the memory on a computer system.

4. However, a recursive version of a function could provide an advantage: creating simpler and clearer versions of algorithms (e.g.: sorting).

7. The efficiency of the functions

Functions are essential to write effective programs (unless very simple ones). However, in some special applications it is better to remove and replace a function with an **inline** code.

Inline code performs the same actions as the function without the disadvantage caused by a function call (including the decreasing of speed).

8. Variadic functions

- A variadic function is a function that accepts a variable number of arguments.

- A good example, among specific operations, that has been implemented as a variadic function in many languages is output formatting. The C function **printf** format is such an example.

- To portably implement variadic functions in the C programming language, the standard **stdarg.h** header file is used. The older **varargs.h** header is still in use for some compilers.

Variables

- Local
- Global

Note:

- Global variables have a definition and optionally, another one or more external variable declarations.

Storage classes of variables in C

- In C programming language, the scope and lifetime of a variable or function within a program is determined by its **storage class**.
- Each variable has a lifetime, or the context in which they store their value.
- Functions, along with variables, also exist within a particular scope, or visibility, which dictates which parts of a program know about and can access them.

There are 4 storage classes in C:

- auto
- register
- static
- extern

auto

- Variables declared inside the function body are automatic by default. These variable are also known as local variables as they are local to the function and doesn't have meaning outside that function
- Since, variable inside a function is automatic by default, keyword **auto** are rarely used. There's a good chance you've never seen this keyword. That's because auto is the default storage class, and therefore doesn't need to be explicitly used often.
- Automatic variables are automatically allocated on stack memory when a program enters a block, and released when the program leaves that block. Access to automatic variables is limited to only the block in which they are declared, as well as any nested blocks.

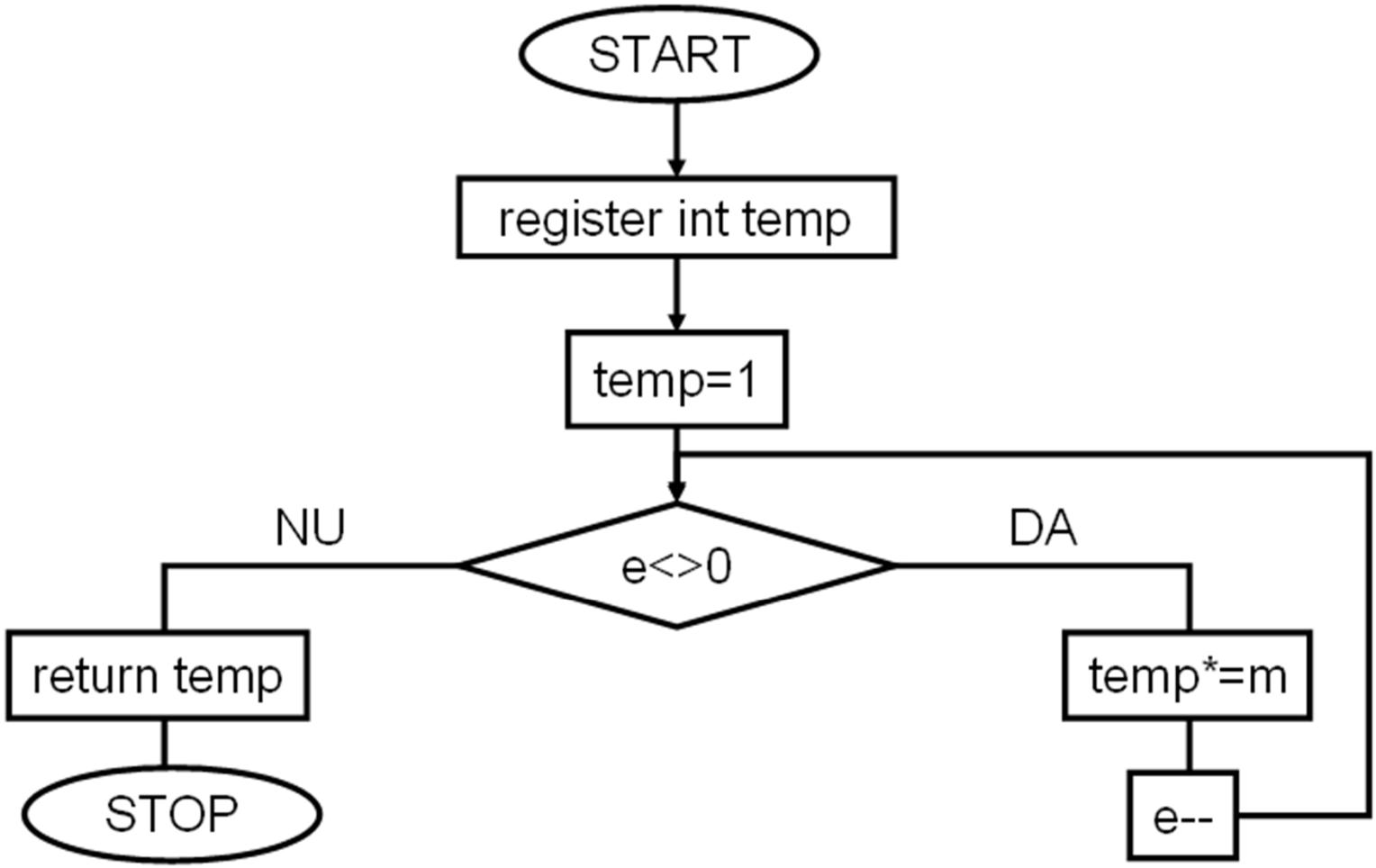
register

- ***register*** behaves just like ***auto***, except that instead of being allocated onto the stack, they are stored in a register.

- Registers offer faster access than RAM, but because of the complexities of memory management, putting variables in registers does not guarantee a faster program - in fact, it may very well end up slowing down execution by taking up space on the register unnecessarily.
- As it were, using register is actually just a *suggestion* to the compiler to store the variable in the register; implementations may choose whether or not to honor this.

Example:

```
int power(register int m, register int e)
{ register int temp;
  temp=1;
 for ( ; e; e--)
        temp=temp *m;
return temp;
}
```


START
register int temp
temp=1
NU
e<>0
DA
return temp
temp*=m
STOP
e--

static

When it comes to storage classes, the keyword **static** means one of two things:

1. A static variable inside a method or function retains its value between invocations.

2. A static variable declared globally can called by any function or method, so long as those functions appear in the same file as the static variable. The same goes for static functions.

extern

- Whereas static makes functions and variables globally visible within a particular file, extern makes them visible globally to *all files*.
- External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.
- In case of large program, containing more than one file, if the global variable is declared in file 1 and that variable is used in file 2 then, compiler will show error. To solve this problem, keyword **extern** is used in file 2 to indicate that, the variable specified is global variable and declared in another file.

Notes:

- Global variables (static and extern) are initialized to 0 automatically

- Local variables get arbitrary values, depending on actual bits' configuration on stack memory

Initialization

- Initialization is the assignment of an initial value for a data object or ***variable***.

- Syntax:

  class type name = expression;

Notes (again):

- Avoid to use local variables (auto) which are not initialized. These variables get arbitrary values when are initialized. This is due to the fact that on the stack when a variable is erased, this doesn't means that their position on the stack will become zero.
- Global and static variables always get zero value at declaration.
- **Function parameters cannot be initialized**.

Cap. **Matrices (Arrays)**

- Matrix representation is a method used by a computer language to store matrices of different dimension in memory.
- **In C programming language, a matrix is a collection of variables (of the same type) that can be called with the same name.**
- C programming language uses "Row Major" schema, which stores all the elements for a given row contiguously in memory.

Notes:

- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

- Syntax:

  type arrayName [lim_1] [lim_2] … [lim_n];

- where lim_i is the limit of index i (on dimension i)
- The index i can have the following values:

  0, 1, 2, … , lim_i - 1

Notes:

- Each lim_i (where i = 0, 1, 2, …, n) is a positive integer.
- We can access the elements of a matrix by using variables with indices.
- In C, the name of a matrix is in fact a pointer that indicates the address of its first element.
- At the declaration of a matrix, the compiler allows a memory zone to contiguous store all its elements.
- **Matrices and pointers are strong related**.

Example

- **int a[3][4];**

  this declares an integer array of 3 rows and 4 columns. Index of row will start from 0 and will go up to 2.

	Column 0	Column 1	Column 2	Column 3
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

One-dimensional matrices (vectors)

- The vectors are one-dimensional arrays.

- Syntax:

type name_vect [lim];

Notes:

- The declaration:

  char p [10];

  creates a matrix of 10 elements, from p [0] to p [9].

- The amount of memory required for recording a matrix is proportional to the type and its size. The total size in bytes is calculated:

  No_of_bytes = sizeof(type) * lim;

- C does not control the limits of the array. You can overcome both edges of a matrix and write in wrong place (of other variables). It remains the responsibility of the programmer to control limits, where necessary.