Computers Programming
Course 8

Iulian Năstac

Recap from previous course

Instructions (Flow Control)
1. Introduction

Definition:

- Generally, an instruction is a part of the program, which can be directly executed.

- An instruction specifies an action of some kind.

Instructions (statements)

- There are several flow control statements in C programming language.
- Basically, C instructions can be organized as:
  - Selection instructions
  - Loop instructions
  - Jump instructions
  - Label instructions
  - Expression instructions
  - Block instructions

Recap from previous course

2. Selection instructions

- In the selection instructions we can meet two distinct forms: ***if ... else*** and ***switch***.

- Note that, in some certain circumstances, a selection instruction can be replaced by the conditional operator (…**?**... **:**…).

Remember about the simple version of the game "Guess the magic number."

2.1.3. Conditional operator (? :)

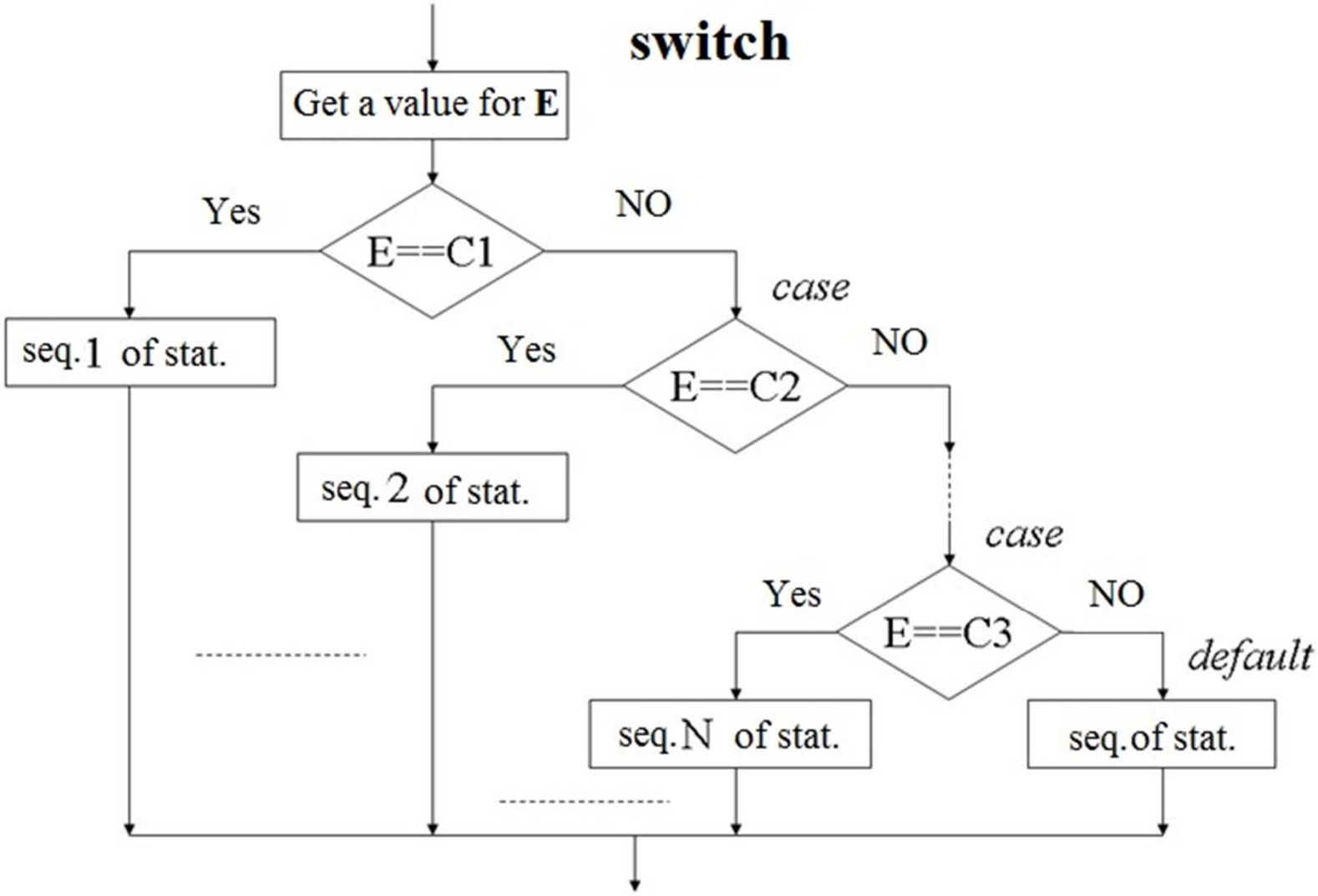
The conditional operator (*? :*) could replace *if-else*, in the following manner:

if (cond) exp_1;
else exp_2; ⇔ *cond ? exp_1 : exp_2;*

Note: With this substitution, the subject for both "if" and "else" must be a simple expression.

Recap from previous course

Flowchart for *switch*



Observe the similarity with ***if-else-if*** chain.

3. Loop instructions

A loop instruction (iteration) allows a set of instructions to be executed repeatedly until a certain condition is satisfied.

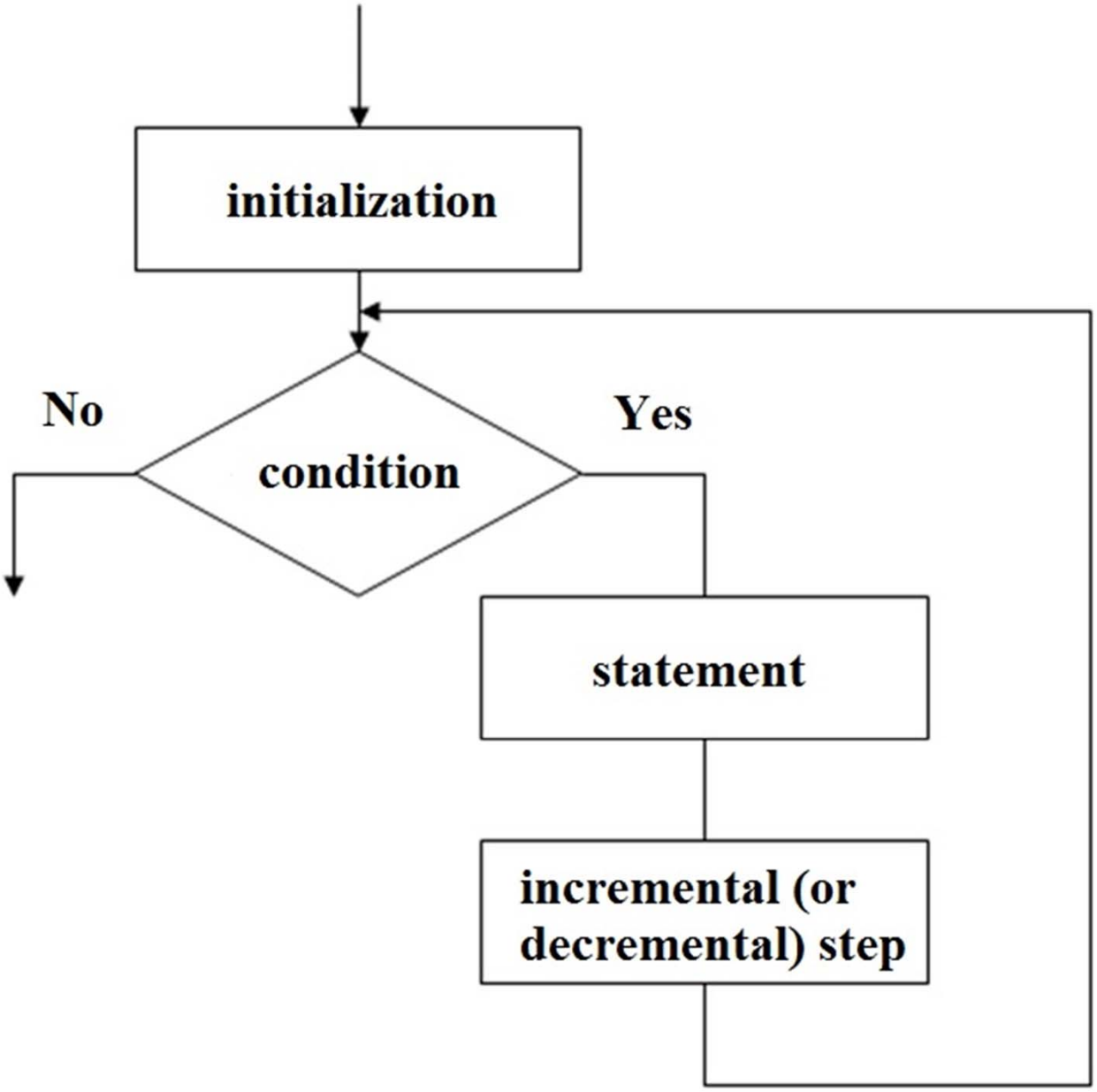
3.1. *for* statement

This instruction is found in most programming languages, but in C has a maximum flexibility.

Syntax:

```
for(initialization; test condition; increment/decrement)
{
/*block of statement*/
}
```

Recap from previous course



Recap from previous course

3.2. *while* statement

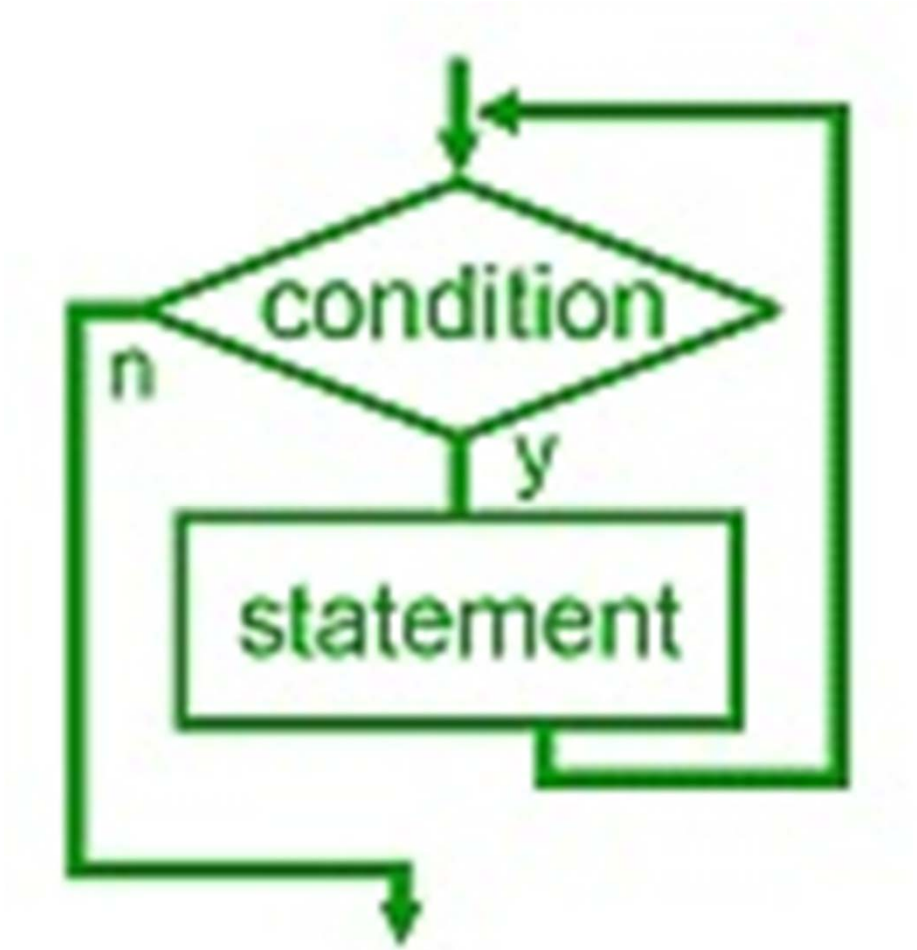
Syntax:

```
while(condition)
        {
                /*block of statement*/
        }
```

Loop repeats as long as the test condition is true.

Recap from previous course

The workflow for the while instruction



Equivalence **for** ⇔ **while**:

```
for (exp1; exp2; exp3)
        {
                /*block of statement*/
        }
```

⇔

```
exp1;
while (exp2)
        { /*block of statement*/
          exp3;
        }
```

Notes:

1) The condition is tested at the beginning.

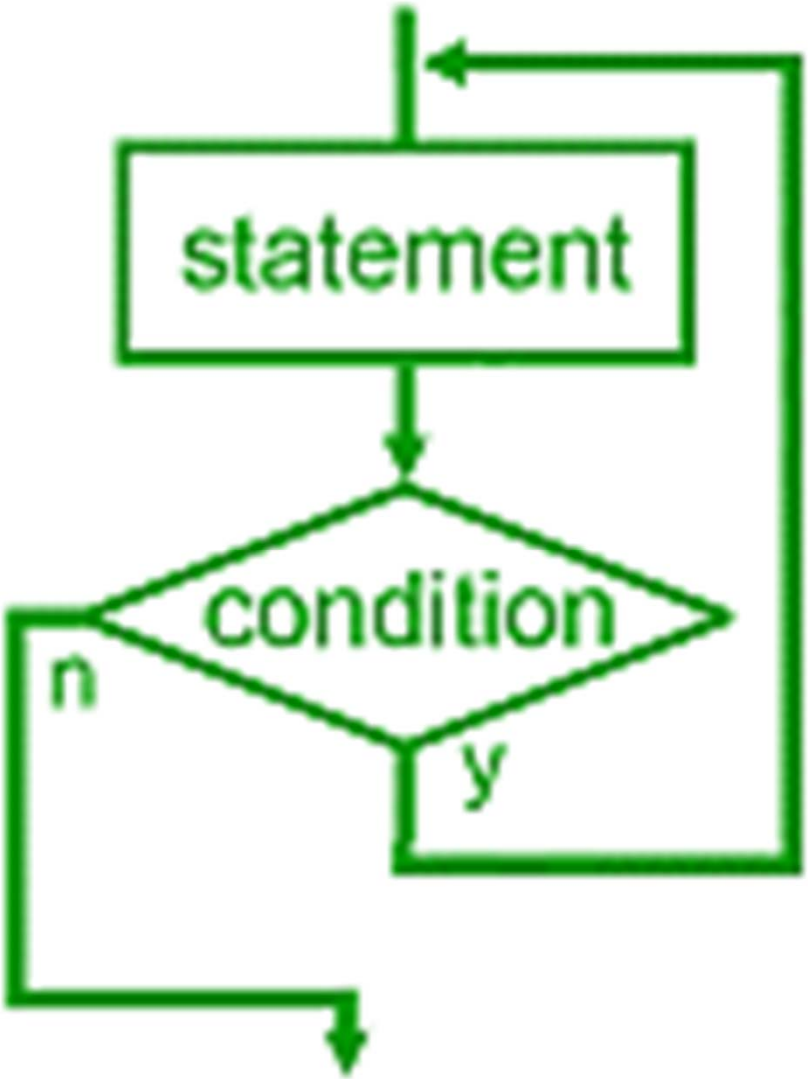
2) If the condition is initially false then the whole loop is ignored.

3.3. *do – while* statement

Syntax:

```
do
{     ….
      /*block of statement*/
} while(condition);
```

The workflow for the do-while instruction



Homework:

Rewrite the problem with magic numbers so that the program will ask the user to re-introduce a number, many times, till it will guess the one chosen by the computer (through **rand** function).

4. Jump statements

The jump statements include:

- ***return*** → may be located anywhere in the program .
- ***goto*** → may be located anywhere in the program .
- ***break*** → within looping instructions or switch statement.
- ***continue*** → within looping instructions.

4.1. *return* statement

It is used to return from a function.

Syntax:

return expression;

where **expression** is present only if the function is declared as returning a value. Value of the expression is converted to the function type.

Note: A function declared with **void** may not contain a **return** statement.

4.2. *goto* statement

Syntax: ***goto*** *label;*

...

label;

Remarks:

- this instruction is avoided because of the abuse to create programs that are not portable;
- There are not situations that require an exclussive goto statement. The label is a valid specifier, which must be in the same block with ***goto***. You can not jump between functions.

- The label can be placed before or after ***goto***.

Example:

```
x=1;
LABEL 1;
        x++;
if (x<100) goto LABEL1;
...
```

4.3. *break* statement

It has two uses:

- finish a case from a **switch** statement;
- **break** a loop.

Example:

```
...
while(1)
        {
        ... /* some operations are executed from a menu */
        …
        printf (”\Do you want to leave this menu?”);
        if ((c=getche())==’d’) break;
        }
...
```

Note: A **break** causes the output only from the innermost loop in which it exist.

Example:

Display numbers from 1 to 10, 100 times.

```
...
        int t, contor;
        for(t=0; t<100; t++)
                { contor=1;
                  for( ; ; )
                   { printf(" %d",contor);
                          contor++;
                          if(contor==11)
                            {
                              printf("\n");
                              break;
                            }
                    }
                }
...
```

Function *exit()*

The function ***exit()*** is presented as a sort of generalized break instruction. This function determines the immediate stop of a program.

Syntax:

void exit(stop_code);

Notes on exit function parameter:

- ***stop_code*** is an ***int***;
- "**0**" is used to indicate the normal completion of the program.
- function exit () can be used when the program takes a bad turn.

Example: Sequence of a program that requires a special graphics adapter.

```
...
int main()
{
…
if(!virtual_graphics()) exit(1);
        ... /* play */
        }
```

4.4. *continue* statement

Continue statement is used only in loops. This instruction forces with the next iteration of the loop, ignoring the rest of the code in the current iteration.

The effect of this instruction is:

a. inside the ***for*** loop, ***continue*** causes direct execution of the incrementing sequence, and then execution of the conditioning test.

b. for **while** and **do-while** loops, the program control passes to the conditioning test.

Example 1

A program that displays all the 1-100 factorials of the consecutive numbers. The program displays the sequence of 15 numbers, then waiting a key to display the next 15 factorials.

```
# include <stdio.h>
# include<conio.h>
int main()
{       int i,j,k;
        double f;
        for(i=1;i<=100;i++)
        { for(f=1.0, j=2;j<=i;j++)
                f*=j;
          printf("\n %d factorial is %g",i,f);
          k=i%15;
          if (k) continue;
          getch();
        }
        getch();
}
```

Example 2

Encoding a message, by changing all characters you type with the next letter in ASCII code (for instance A becomes B). The program stops when you type *$*.

```
...
char Finish, ch;
...
Finish = 0;
while (! Finish)
        {  ch=getch();
           if(ch=='$')
                   {Finish = 1;
                    continue;
                   }
        putchar(ch+1);
        }
...
```

5. Label instructions

These are valid labels encountered during execution of a program.

In C, the label instructions are of two types:

- ***case*** and ***default*** - discussed in the **switch** statement;
- *Valid* ***labels*** - discussed in the **goto** statement.

6. expression type statements

An expression instruction is any valid expression followed by a semicolon (;). This includes a null instruction.

An empty statement is reduced to only a semicolon (;). It has no effect, but is frequently used in certain alternative codes.

7. block statements

A block statement is a sequence of instructions (of different kinds) enclosed in braces.

Syntax:

```
{
   inner successive statements;
}
```

Block instructions are groups of instructions that are treated as a unit.

Remarks:
Any switch is followed by a block statement.
The most commonly situation is when used several instructions to create a multiple instruction as object of another instruction.

Cap. Procedural programming - Functions

Introduction

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 (to a specific memory address) whenever needed. After this jump, the program will return to the next instruction (after the one in which it leaped or sometimes even in the same instruction, in the case of a complex instruction) and therefore differs from the jumps performed with **goto** statement.

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

Many programming languages contain two types (or categories) of procedures:

a) procedures that define a return value;

b) procedures that doesn't use a return value.

In C, the using of functions in a program involves:

- **Declare functions**;

- **Defining functions** (complete description of functions).

RETURN_TYPE name_of_function (PARAMETER_TYPE name_of_param1,
PARAMETER_TYPE name_of_param2, … etc.);

```
// here are some examples of prototypes used at the top of a file:


float sqrt( float x );

float average( int grades[ ], int length );
```

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.

Function Declaration and Function Prototypes

- All identifiers in C need to be declared before they are used.
- This is true for functions as well as variables.
- For functions the declaration needs to be before the first call of the function.
- A full declaration includes the return type and the number and type of the arguments.
- This is also called the function prototype.

Note:

- Older versions of the C language didn't have prototypes, the function declarations only specified the return type and did not list the argument types.

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, ...)
{
...
}

Notes:

1. These two styles define the developmental stages of language C. Most compilers today only use programs written in modern style. The programs written in the classical style can be easily corrected.

2. When complete describing the functions, the first line will be identical to the prototype (the declaration) with the observation that on this definition does not appear semicolon (**;**).

3. At the present stage we can say that the logic of writing a program in "***C***" is the following:

- Including the header file;

- Declarations of global variables and constants;

- Declarations of functions;

- Describing the main function;

- Describing all functions declared.

4. By declaring functions (prototypes), the compiler makes preliminary checks on function parameters.

5. The declaration of the functions follows the order of their calling within the program.

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.

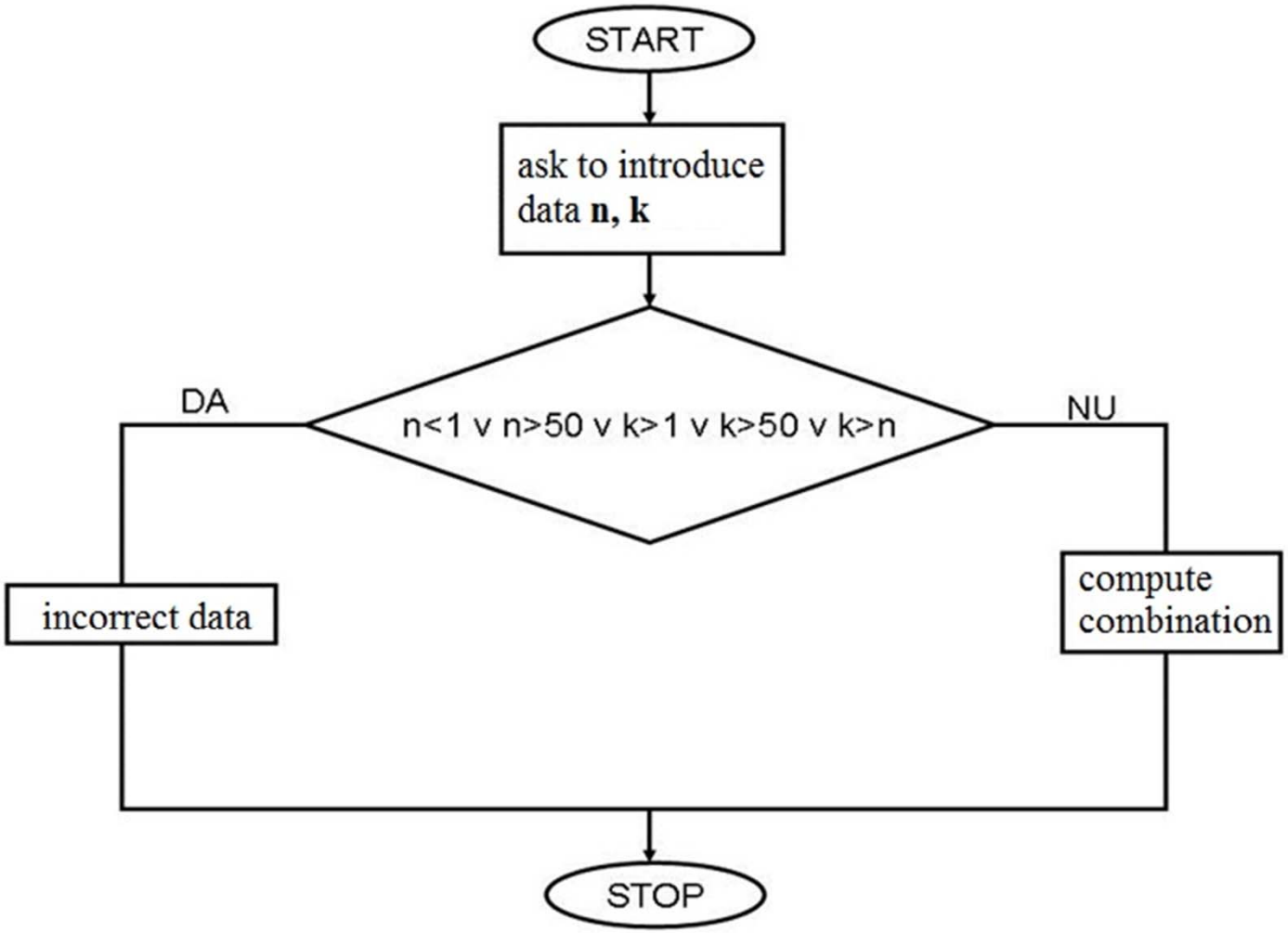
Example:

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:

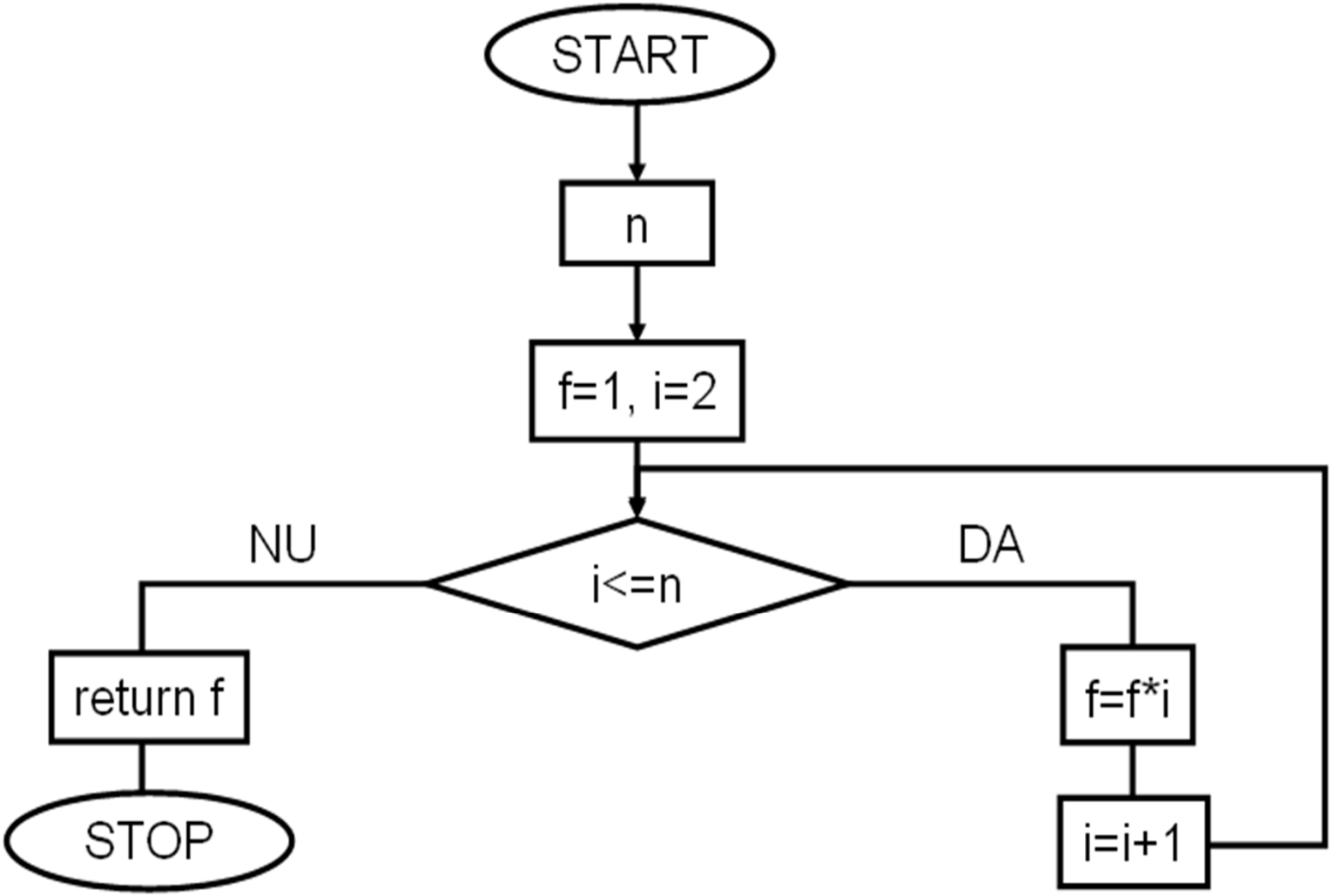
$$C_n^k = \frac{n!}{k!(n-k)!}$$

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.