Computers Programming
Course 7

Iulian Năstac

Recap from previous course

Operators in C

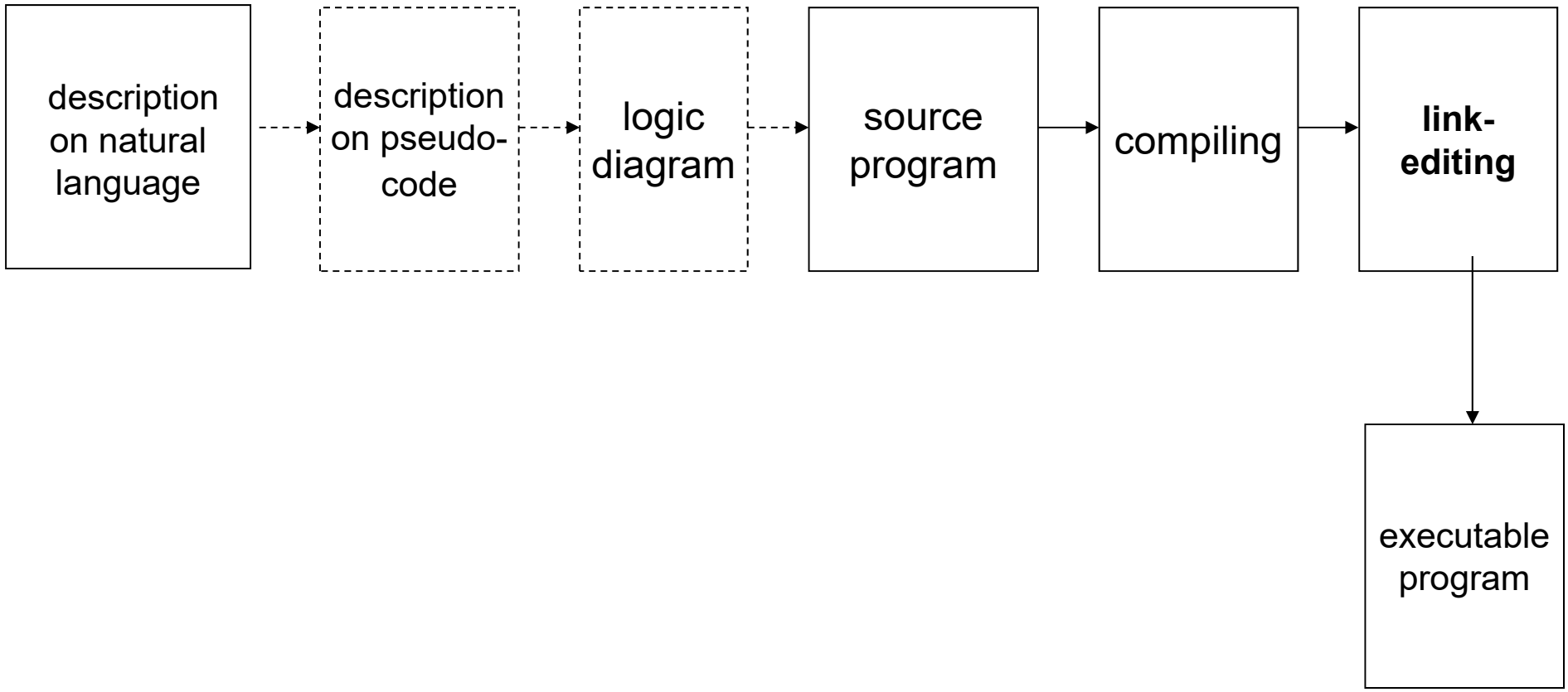
- Programming languages typically support a set of operators, which differ in the calling of syntax and/or the argument passing mode from the language's functions.

- C programming language contains a fixed number of built-in operators.

1	**() [] -> . ::**	Grouping, scope, array / member access
2	**! ~ - + * & sizeof *type cast* ++x - -x**	(most) unary operations, sizeof and type casts
3	*** / %**	Multiplication, division, modulo
4	**+ -**	Addition and subtraction
5	**<< >>**	Bitwise shift left and right
6	**< <= > >=**	Comparisons: less-than, ...
7	**== !=**	Comparisons: equal and not equal
8	**&**	Bitwise AND
9	**^**	Bitwise exclusive OR
10	**|**	Bitwise inclusive (normal) OR
11	**&&**	Logical AND
12	**||**	Logical OR
13	**?: = += -= *= /= %= &= |= ^= <<= >>=**	Conditional expression (ternary) and assignment operators
14	**,**	Comma operator

Recap from previous course

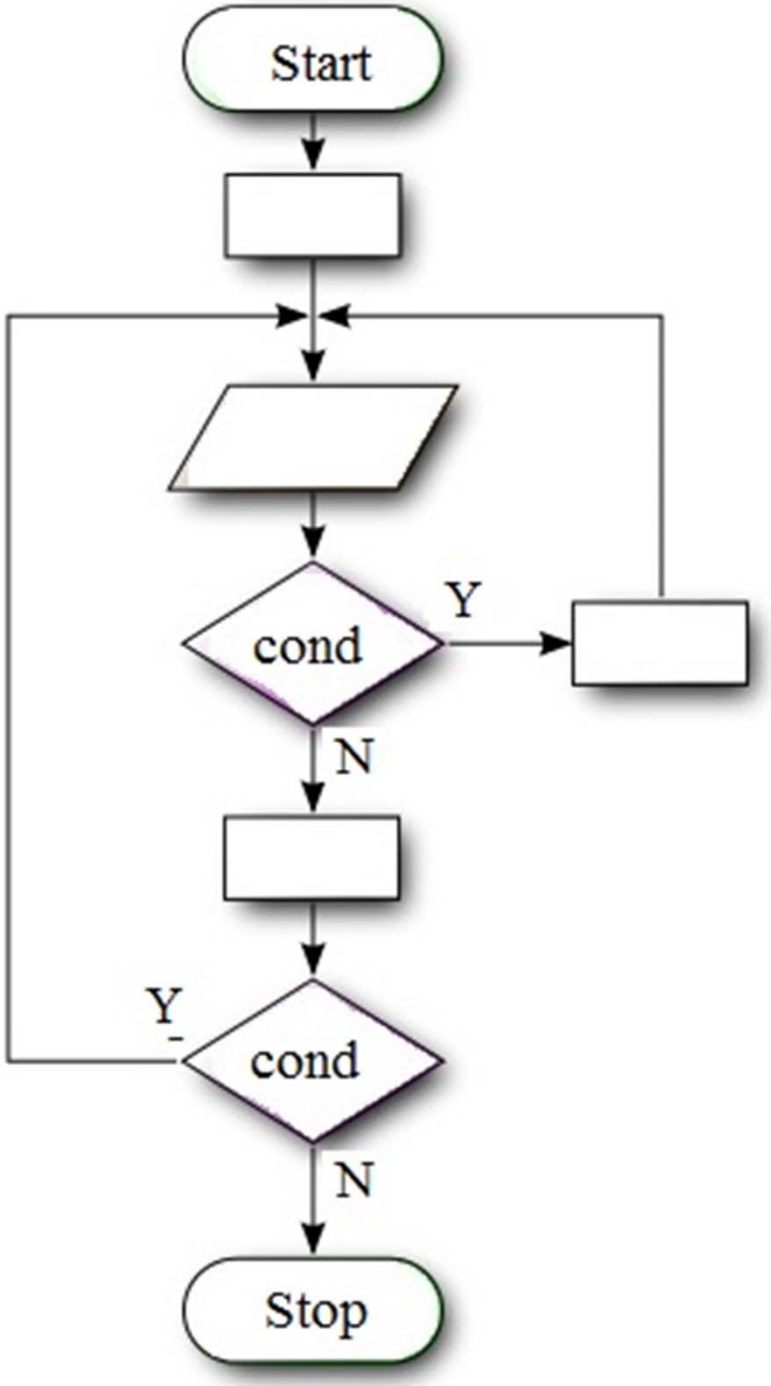
The design of a program includes several steps



Recap from previous course

Flow chart
(logic diagram)

- A **flow chart** is a schematic representation of an algorithm or a process, or the step-by-step solution of a problem.
- Flow charts use suitably annotated geometric figures connected by flow lines for the purpose of designing or documenting a process or program.



Flowchart start / stop

Flowchart process

Flowchart connector

Flowchart selection

Flow line

Recap from previous course

Pseudocode

- **Pseudocode** is an informal high-level description of the operating principle of a computer program or other algorithm.
- No standard for pseudocode syntax exists, as a program in pseudocode is not an executable program.
- A programmer who needs to implement a specific algorithm, especially an unfamiliar one, will often start with a pseudocode description, and then "translate" that description into the target programming language and modify it to interact correctly with the rest of the program.

Instructions (Flow Control)
1. Introduction

Definition:

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

- An instruction describes an action of a specific 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

True and false in C

- Many instructions in C are based on an expression of conditioning that determines the next action. A conditional expression is evaluated as true or false.

- In C, unlike other languages, it is considered as true any nonzero value (including negative numbers). A false value is equivalent with 0. True and false concepts allow a wide variety of routines that can be efficiently encoded.

2. Selection instructions

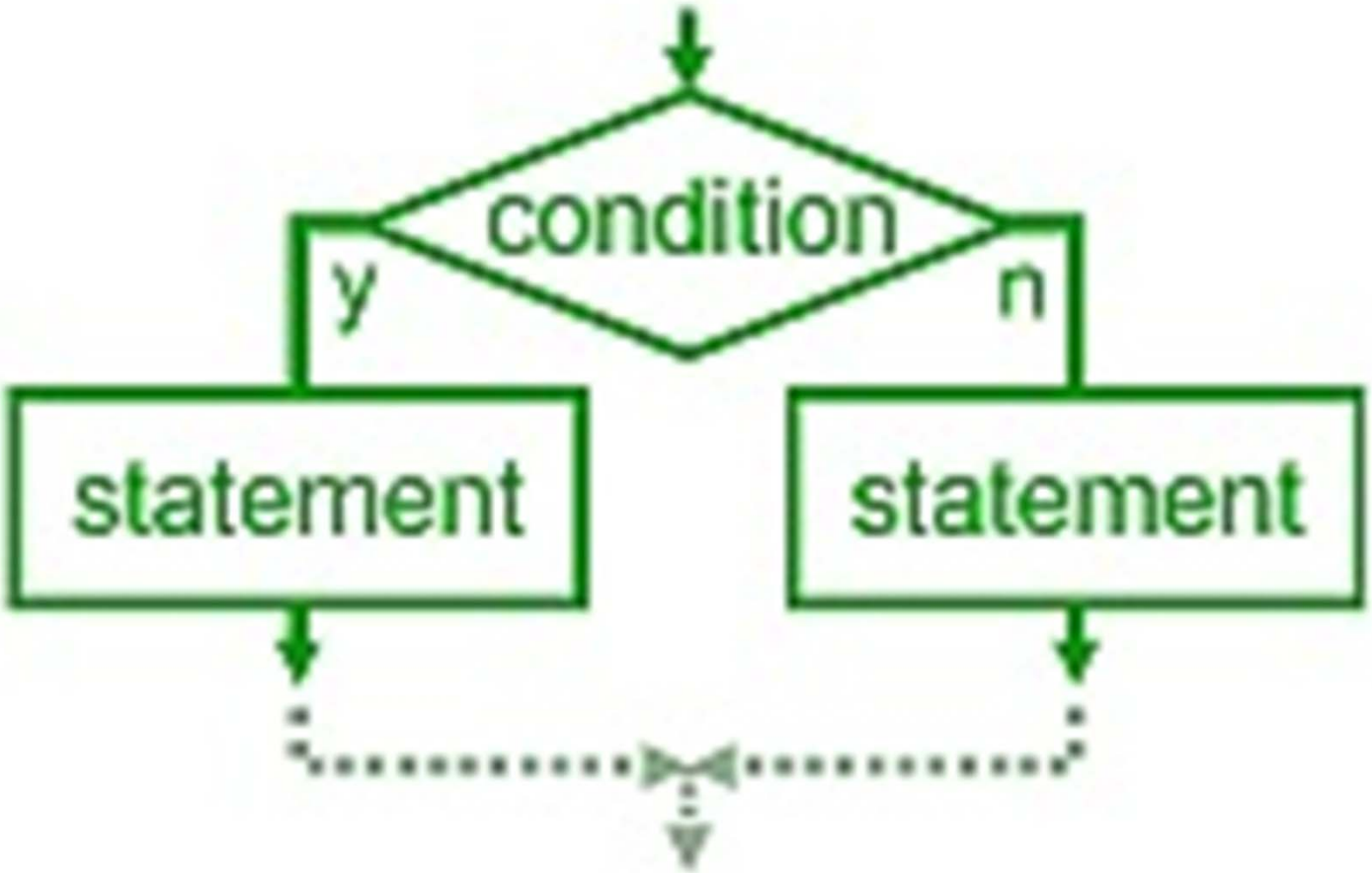
- In the selection instructions we can meet two distinct forms: ***if ... else*** and ***switch***.

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

2.1. *if statement*

Syntax:

```
if (condition is true)
        {
        /*1st block of statements*/
        }
else
        {
        /*2nd block of statements*/
        }
```


condition
y
n
statement
statement

Notes:

1) Only one instruction can be executed. If both instructions are running it seems that an ”**;**” is placed in a wrong position.

2) Conditioning that controls an "**if**" will cause a scalar result.

Example:

Simple version of the game "Guess the magic number."

- It displays **correct** if the player guesses the magic number.
- To choose the magic number, a random function is employed: *rand().*
- *This function generates a random number in the range* [0, RAND_MAX] where RAND_MAX = 32.767.

```
# include <stdio.h>
# include <stdlib.h>
int main()
{ int magic;
  int guess;
  magic = rand();
  printf (“\n Guess the magic number: “);
  scanf (“%d”, & guess);
  if (guess ==magic) printf (“\n Correct”);
  else printf (“Wrong”);
  getch();
}
```

2.1.1. Nested *if*

- *It is always possible to **nest** if-else statements, which means you can use one **if** or **else if** statement inside another **if** or **else if** statement(s).*

Ex:
```
if (i)  { if (i) instr1;
            if (k) instr2;
            else instr3;
          }
        else instr4;
```

Note: ANSI C allows a number of at least 15 levels of nesting. In practice, most compilers allow more levels. But, by using excessive nesting the whole program becomes unclear.

Example: extend previous program with a nesting level

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
  int magic;
  int guess;
  magic = rand();
  printf ("\n Guess the magic number: ");
  scanf ("%d", &guess);
  if (guess==magic) printf ("\n Correct");
  else    { printf ("Wrong. ");
            if (guess>magic) printf("Too big \n");
            else printf ("Too small \n");
           }
   …
}
```

2.1.2. *if – else – if* chain

The ***if – else – if*** chain is an usual construction in the case of multiple selections.

General form:

```
if (exp1) instr1;
else  if (exp2) instr2;
        else  if (exp3) instr3;
                else instr4;
```

Example:
Simplifying the previous program

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
  int magic;
  int guess;
  magic = rand();
  printf ("\n Guess the magic number: ");
  scanf ("%d", &guess);
  if (guess==magic) printf ("\n Correct");
   else  if (guess>magic) printf("too big \n");
            else printf ("too small \n");
}
```

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.

Examples:

1) *if (x>9) y=100;*
   else y=200; ⇔ *y=x>9 ? 100:200;*

2) *t ? f1(t)+f2(t) : printf("\n there is 0");*

Magic numbers again

```
# include <stdio.h>
# include <stdlib.h>
int main()
{
  int magic;
  int guess;
  magic = rand();
  printf ("\n Guess the magic number: ");
  scanf ("%d", &guess);
  if (guess==magic) printf ("\n Correct");
    else  (guess>magic) ? printf("too big \n") : printf ("too small \n");
}
```

2.2. *switch*

It is an instruction that successively tests the result of an expression or variable against a list of character or integer constants.

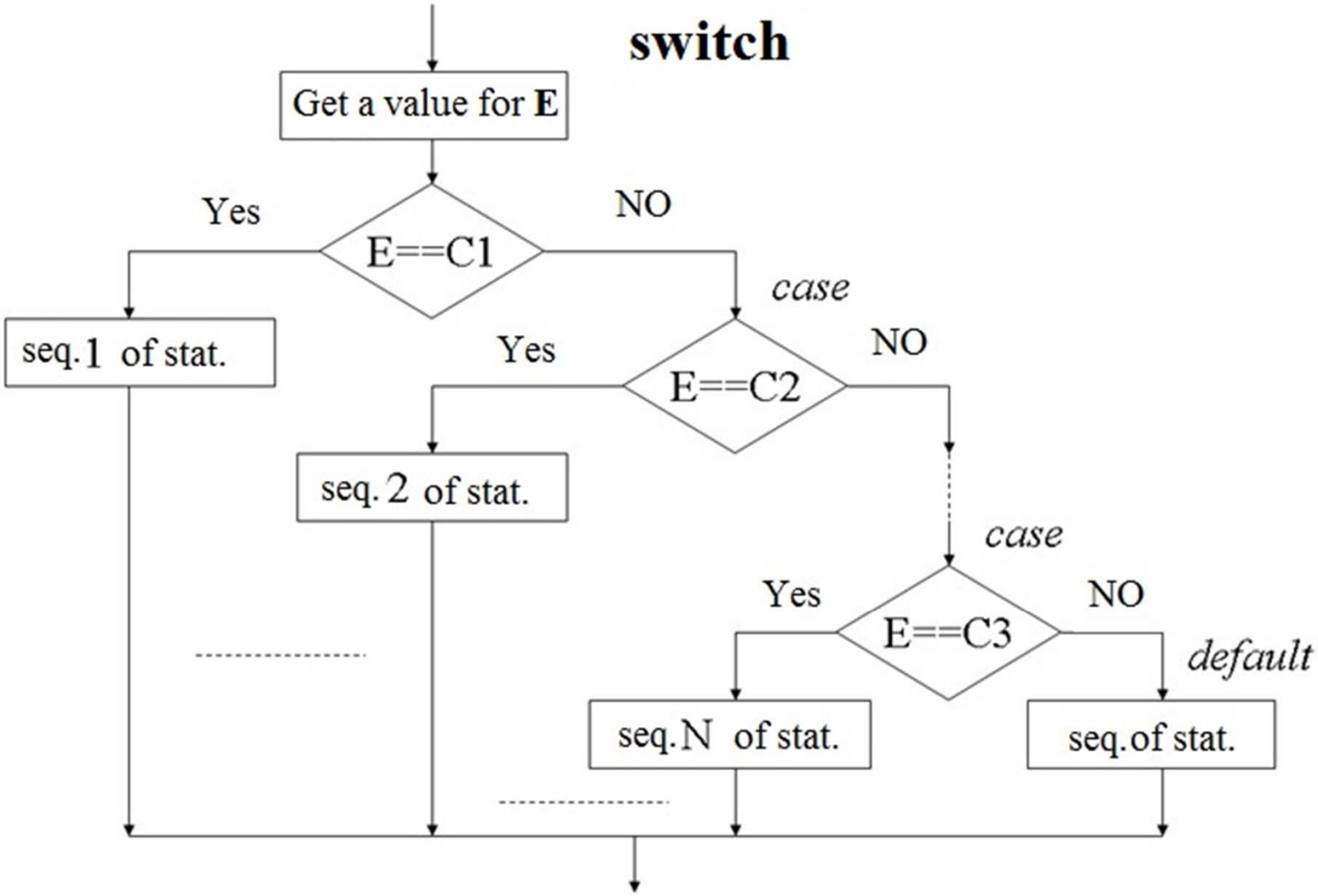
General form:

```
switch (exp)
    { case const1: sequence of instructions_1;
          break;
      case const2: sequence of instructions_2;
          break;
      …
      case constN: sequence of instructions_N;
          break;
      default: sequence of instructions_N+1;
    }
```

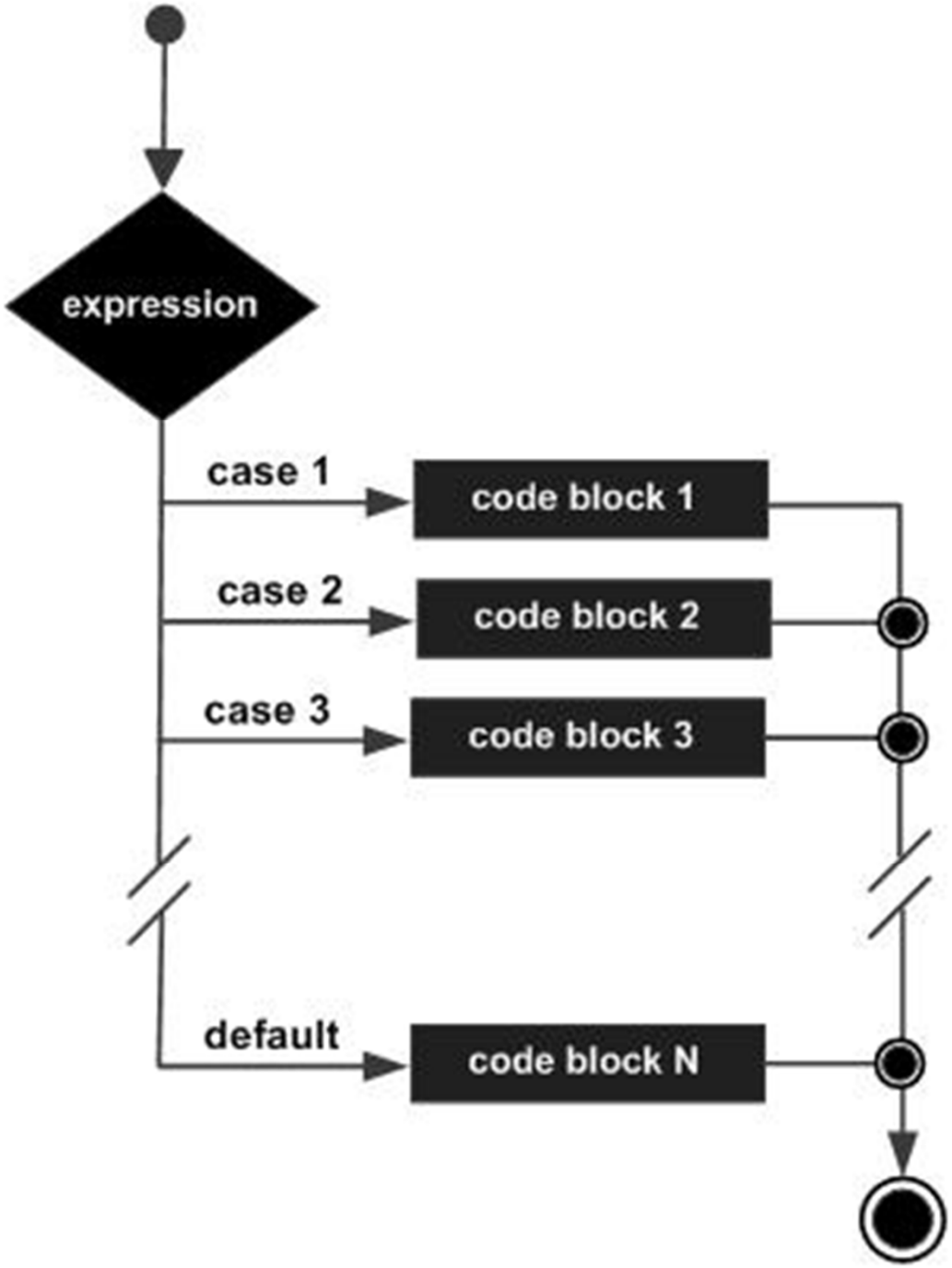
Notes:

1) ***break*** is optional, but its missing can leads to abnormal evolutions of the program.
2) ***default*** is optional.
3) ANSI Standard C states: C compilers must allow for a switch statement max. 257 cases. Practically there are no programs to use so many cases.
ANSI C++ - > 16.384 cases.
4) The sub-instruction ***case*** is forbidden outside a switch.
5) A ***switch*** is useful for menu selection.

Flowchart for *switch*



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


expression
case 1
code block 1
case 2
code block 2
case 3
code block 3
default
code block N

Example:

A simple program that add and subtract

```
...
float x, y;
char op;
…
scanf("%f %f", &x, &y);
op=getch();
switch(op);
{ case '+': printf("\n Summ: %f \n", x+y);
            break;
  case '-': printf("\n Substract: %f \n", x-y);
            break;
  case '*':
  case '/': printf("\n Invalid operation\n");
            break;
  default: printf("\n Error \n");
 }
…
```

Notes:

1) There may be "**cases**" that haven't associated any sequence instructions. When this occurs the execution is placed to the next "**case**".
2) An instruction execution continues with the next **case** if there is no **break** statement. This prevents sometimes duplication of repetitive instructions and the result is a more efficient code.

2.2.1. Nested *switch*

In some programs there may be a "**switch**" instruction embedded in a sequence of another "**switch**". Even if some constants from the inner **switch** and the outer one also, contain similar values, there is no conflict anywhere.

```
...
switch(x)
  { case 1: switch(y)
                  { case 0: printf("message1");
                            break;
                    case 1: procesat(x,y);
                  }
          break;
    case 2: …;
  }
```

3. Loop instructions

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

3.1. *for* statement

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

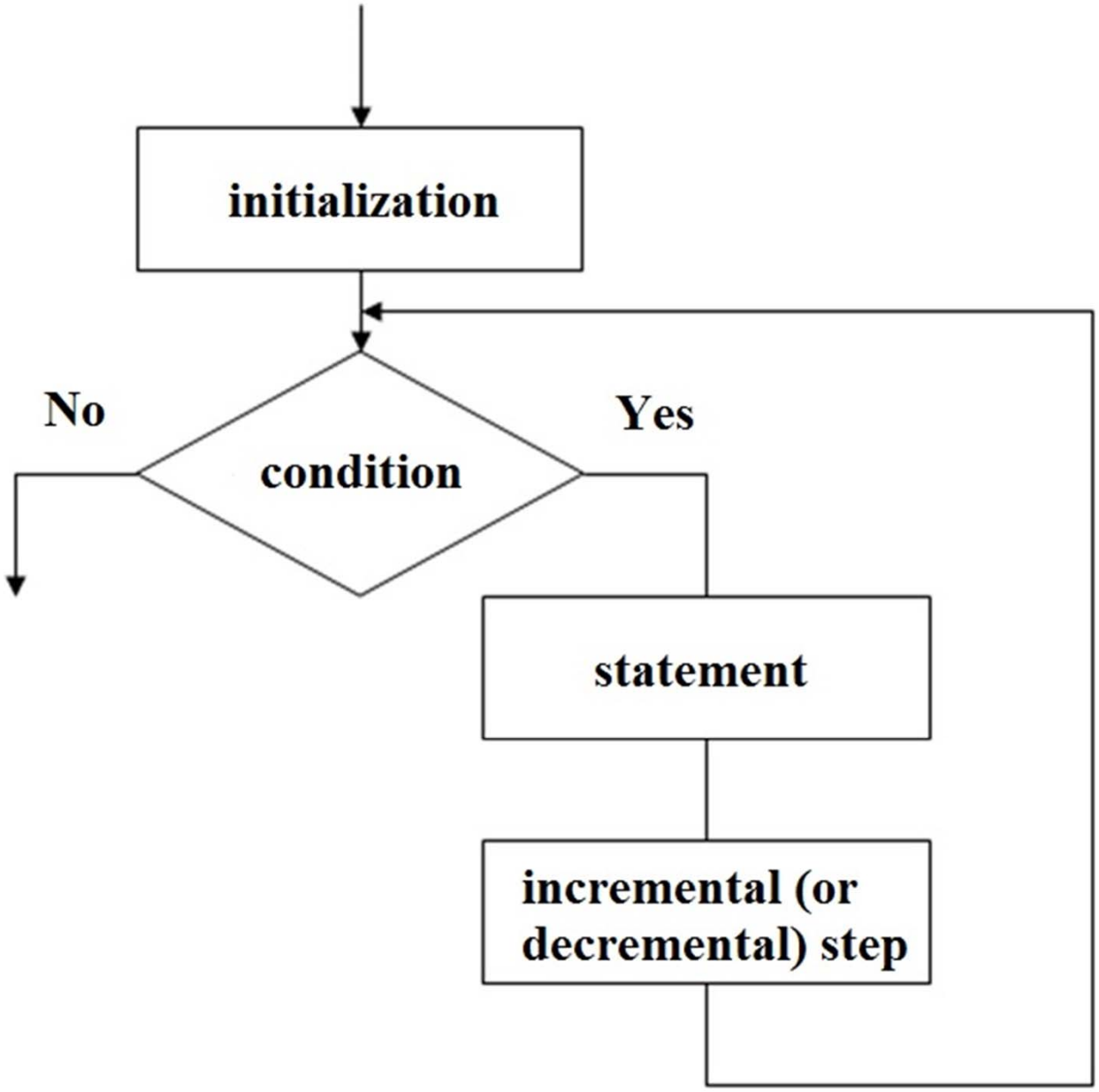
Syntax:

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

The for loop in C is executed as follows:

1. The initial counter value is initialized. This initialization is done only once for the entire for loop.
2. After the initialization, test condition is checked. Test condition can be any relational or logical expression. If the test condition is satisfied i.e. the condition evaluates to true then the block of statement inside the ***for*** loop is executed.
3. After the execution of the block of statement, increment/decrement of the counter is done. After performing this, the test condition is again evaluated.

The step 2 and 3 are repeated till the test condition returns false.


initialization
No
condition
Yes
statement
incremental (or
decremental) step

Notes:

1) Initialization, condition and increment step are optional.

2) The associate statement may be invalid instruction (no instruction), single or multiple (in a block {…}).

Example 1:

```
for (x=100; x!=65; x-=5)
                { y=x*x;
                  printf(" \n Square of %d, is %f", x, y);
                }
```

Example 2:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i;
    clrscr();
    for(i=1; i<=5;i++)
    {
     printf("%d This will be repeated 5 times\n", i);
    }
    printf("End of the program");
    getch();
}
```

3.1.1. Variations of *for*

Variations from the standard loop can be obtained using the comma operator.

Example:

1)
```
...
        for (x=0, y=0; x+y<10; ++x)
        { … }
```

2)
```
...
for (i=1, j=rand(); i<j;  i++, j- -)
        { … }
```

3) Using a program access with password - the user can try up to three times to enter the password.

```
...
char sir[20];
int x;
…
for (x=0; x<3 && strcmp(sir, “pass”); x++)
        { printf(“\n Please enter the password :”);
          gets(sir);
        }
if(x==3) exit(1); /* otherwise the program continues */
…
}
```

4) Some parts of the general definition may be missing.

```
...
for (x=0; x!=123;   ) scanf(“%d”, &x);
```

In the latter example, if you type 123 then the loop is finished.

3.1.2. The infinite *for*

Format: *for(; ;) instruction;*

Example:
for(; ;) printf("This loop will run indefinitely \n");

Notes:
1) When the expression of conditioning is absent, it is assumed that it is true.
2) The construction ***for(; ;)*** does not necessarily guarantee a true infinite loop because the **break** statement used inside it determines its immediate ending.

```
...
ch='\0';
for( ; ; )
        { ch=getchar();
          if(ch=='A') break;
        }/* the loop is indefinitely repeated until the
        user will introduce an A from the keyboard*/
...
```

3.1.3. *for* without body of statement

It can be used to increase the efficiency of algorithms or create a delay loop.

Example: delay loop :

```
...
for (t=0; t<N; t++)   ;
...
```

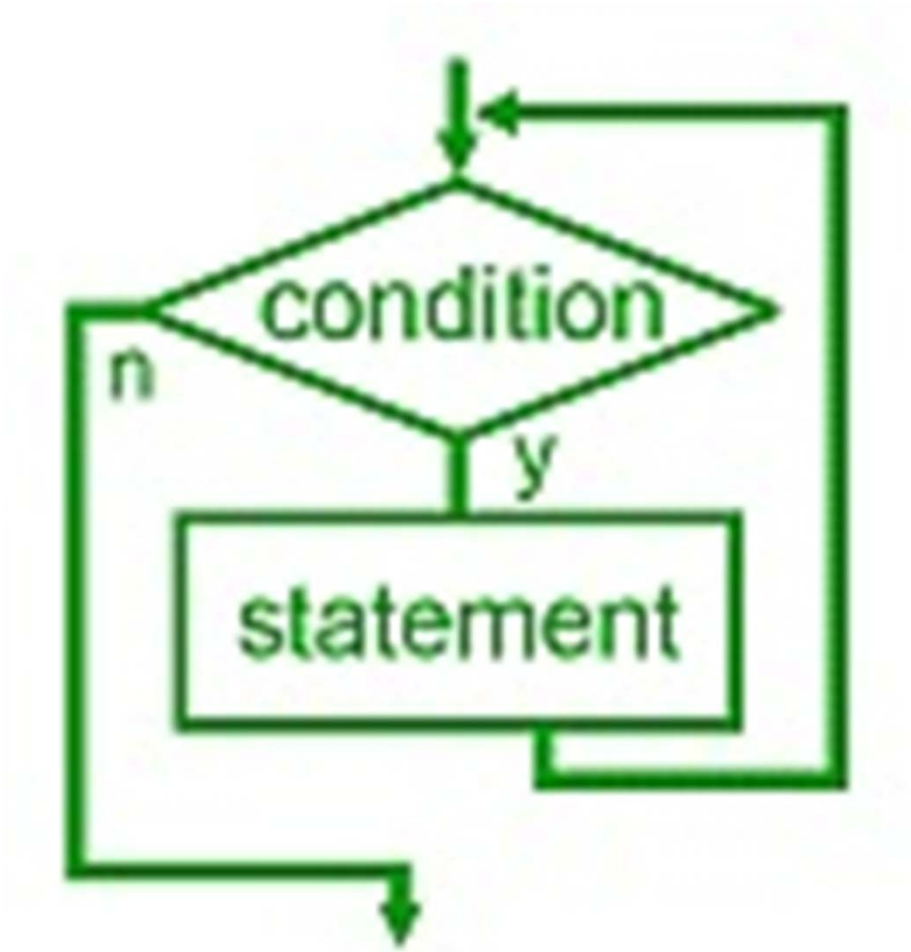
3.2. *while* statement

Syntax:

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

Loop repeats as long as the test condition is true.

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.

Example:

1)
```
char ch;
ch = '\0';
…
while (ch != 'A') ch=getchar();
…
```

The loop is indefinitely repeated until the user will introduce an A from the keyboard.

2) The block of statement can be missing:

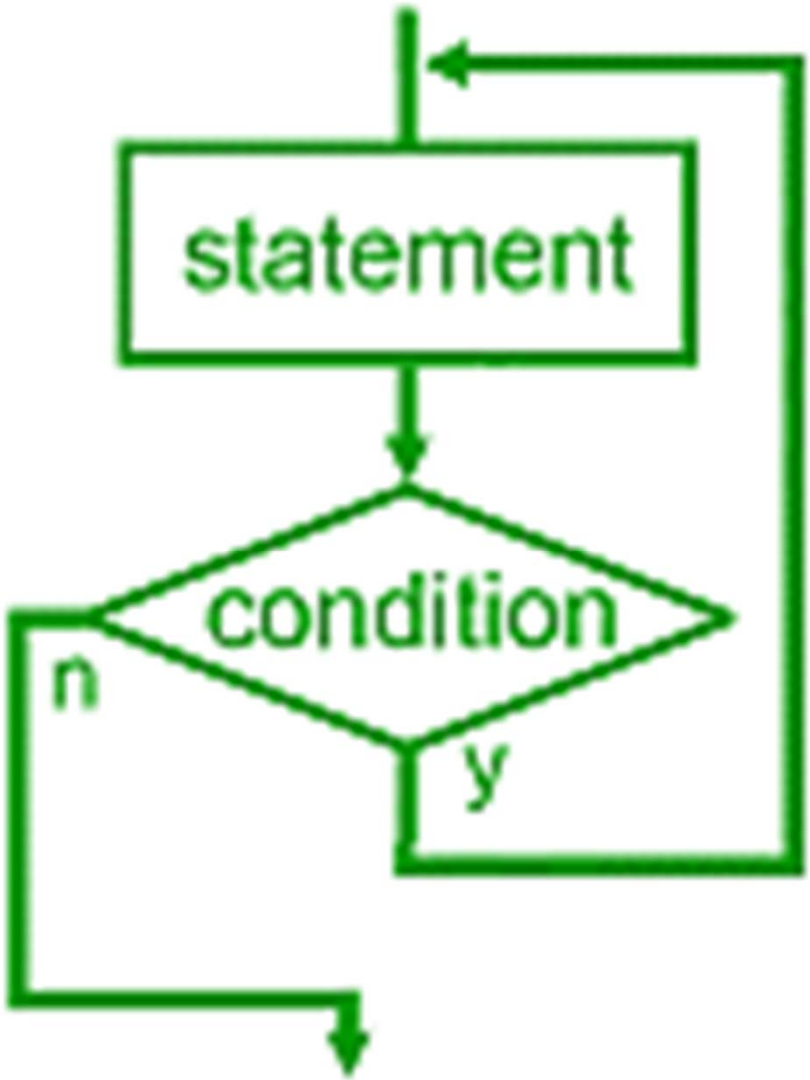
```
while ((ch=getchar()) != 'A');
```

3.3. *do – while* statement

Syntax:

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

The workflow for the do-while instruction



- Here, while and do are the keywords which is know to the compiler.
- Condition can be any expression.
- This is very similar to while loop.

Notes:

- The block of statement enclosed in the opening and closing braces after the keyword do is executed at least once.

- After the execution of the block of statement for the first time, the condition in the do-while is checked.

- If the conditional expression returns true, the block of statement is executed again. This process is followed till the conditional expression returns false value.

Example: selection from a menu

```
...
char ch;
…
printf("\n 1. Option 1 ");
printf("\n 2. Option 2 ");
printf("\n 3. Option 3 ");
printf("\n Enter an option ");
do   { ch=getchar();
       switch(ch);
          { case '1': block of statement  1; break;
            case '2':  block of statement 2; break;
            case '3':  block of statement 3; break;
          }
     } while (ch!='1' && ch!='2' && ch!='3');
…
```

Homework:

Rewrite the problem with magic numbers so that the program will ask to re-entry a number until the user will guess the number chosen by the ***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.