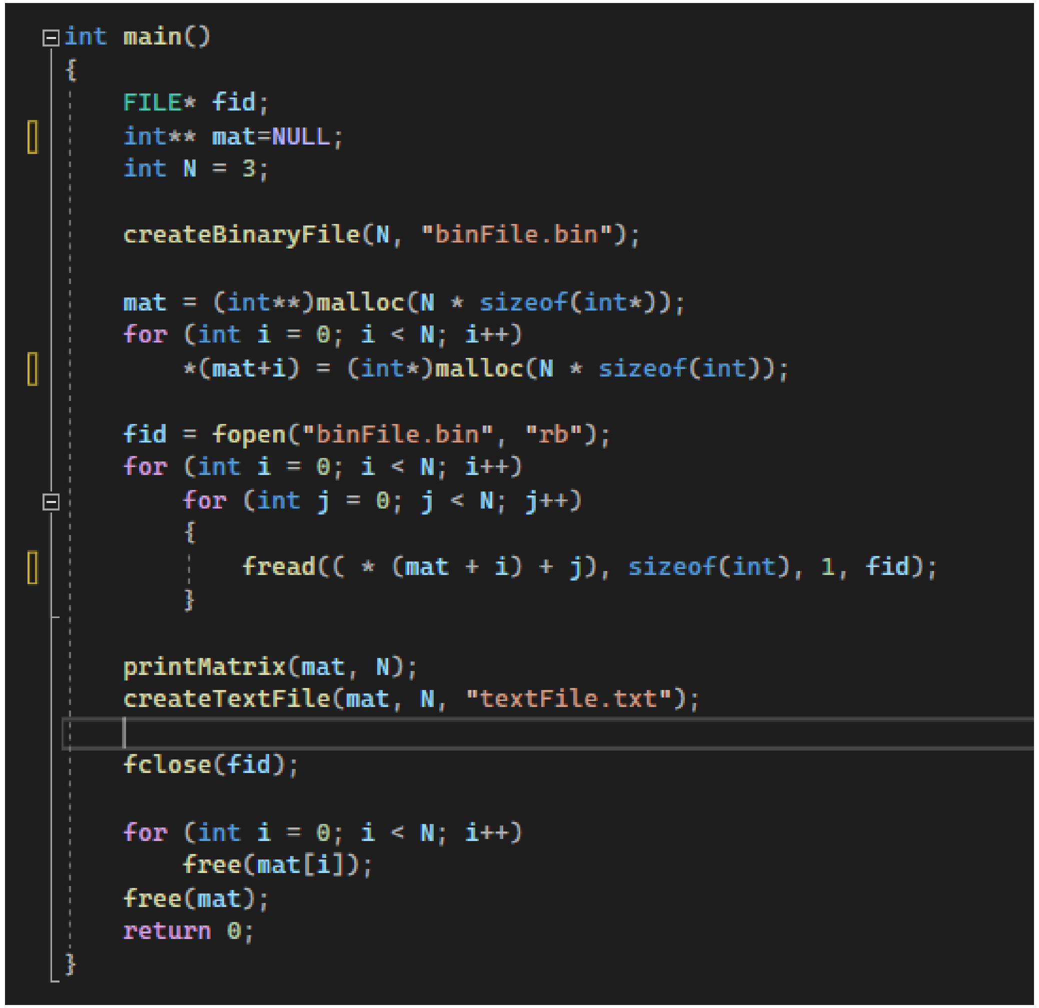
The C programming language

Ștefan-Adrian Toma

Military Technical Academy “Ferdinand I”

The C programming language

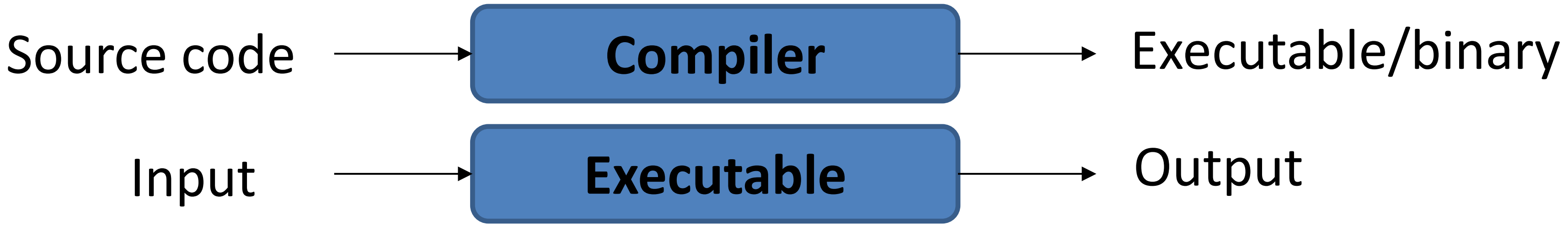
- C was developed at Bell Labs, in the 1970s, by Dennis Ritchie.
- C is a general-purpose, high level, structured language.
- The instructions are similar to algebraic expressions. Keywords are in English (e.g., *if*, *else*, *for*, *do* and *while)*.



```c
int main()
{
    FILE* fid;
    int** mat=NULL;
    int N = 3;

    createBinaryFile(N, "binFile.bin");

    mat = (int**)malloc(N * sizeof(int*));
    for (int i = 0; i < N; i++)
        *(mat+i) = (int*)malloc(N * sizeof(int));

    fid = fopen("binFile.bin", "rb");
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            fread(( * (mat + i) + j), sizeof(int), 1, fid);
        }

    printMatrix(mat, N);
    createTextFile(mat, N, "textFile.txt");

    fclose(fid);

    for (int i = 0; i < N; i++)
        free(mat[i]);
    free(mat);
    return 0;
}
```

The C programming language

- Learning C helps understand how computers work.
- The language is standardized: ANSI C, C99, C11, C17, C2x.
- It is a compiled language. This means that if the programs are written appropriately, it is very fast.



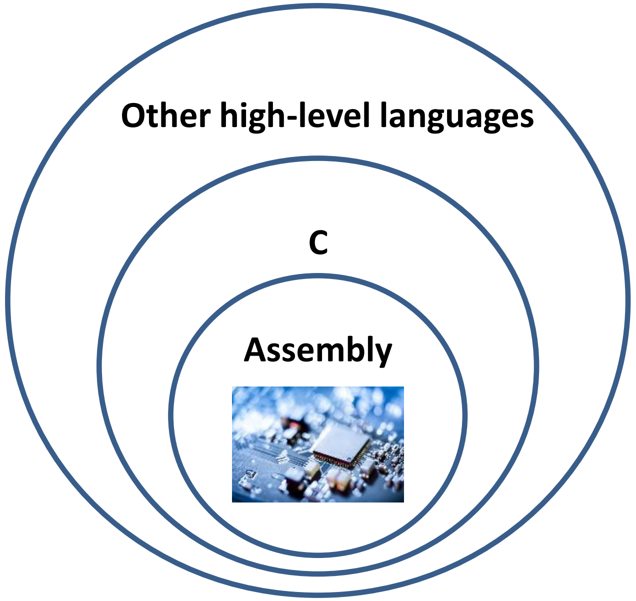
Source code → Compiler → Executable/binary

Input → Executable → Output

The C programming language

- The C programming language has some characteristics that make it a bridge between machine language and high-level languages. Hence, one can use C for both general purpose programs (e.g., *Doom*, *git*) and system programs (e.g., *the Linux kernel*, *device drivers*)

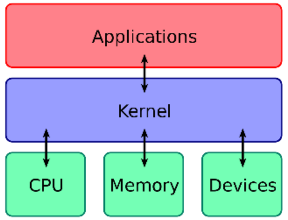
- It widely used in **embedded systems** (automotive, communications, radars etc.)



Other high-level languages

C

Assembly



Applications

Kernel

CPU

Memory

Devices

The C programming language

The elements of the C language are:

- Identifiers (variable, function, and array names)
- Keywords
- Constants (numerical, character, and string literals)
- Arrays (multiple values of the same type)
- Operators (+, -, *, /, =, &&, ||, etc.) – do operations with constants and variable
- Separators

- Variables – named locations in memory

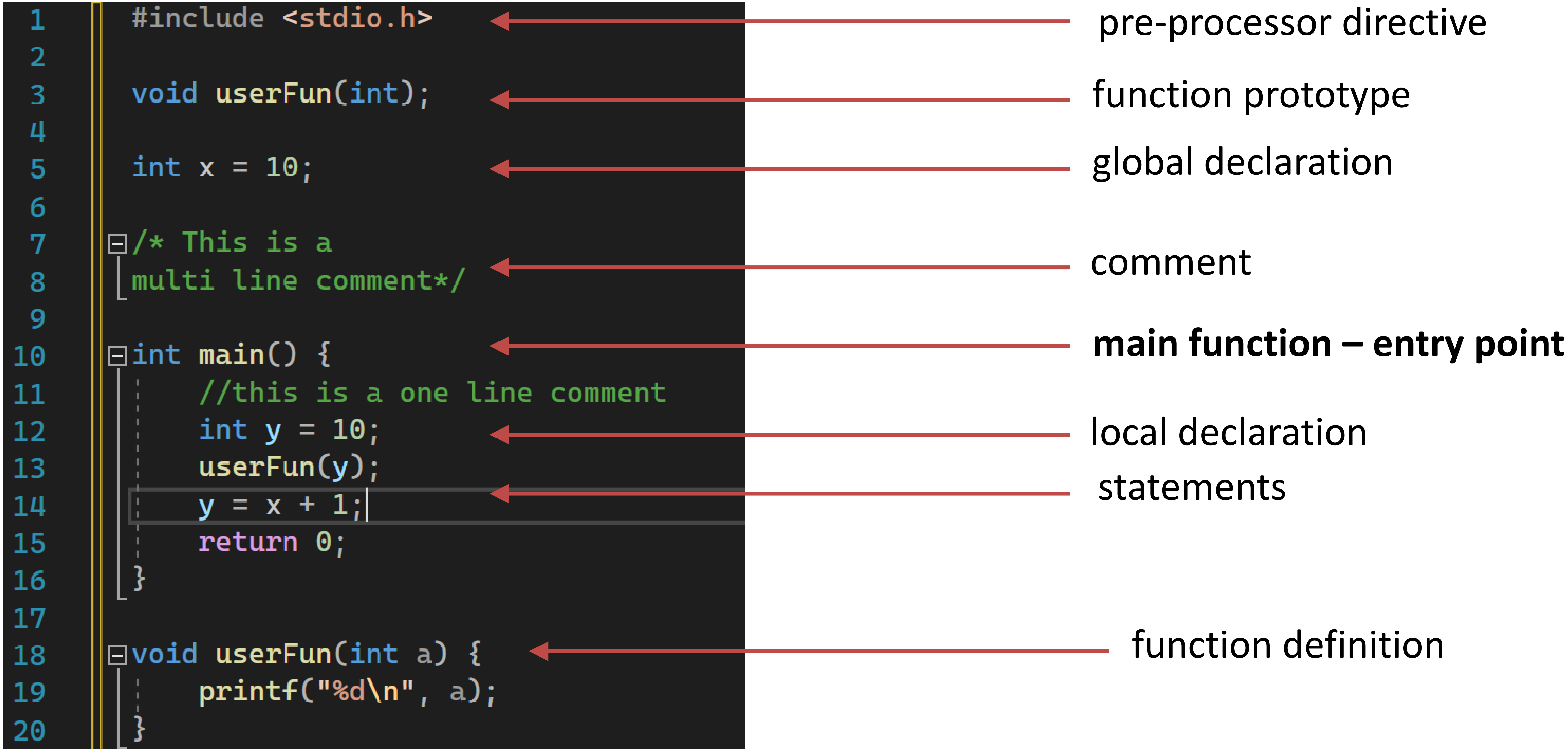
The C programming language - keywords

int	extern	double
char	register	float
unsigned	typedef	static
do	else	for
while	struct	goto
switch	union	return
case	sizeof	default
short	break	if
long	auto	continue
signed	const	void
enum	volatile	

The C programming language - separators

Separator	Name	Usage
{}	Curly braces	Defines blocks of code, such as functions, loops and conditionals.
[]	Square brackets	Used in array declarations and indexing.
()	Parentheses	Used in function calls, function declarations, and control flow statements.
,	Comma	Separates multiple variables, parameters, and control flow statements.
;	Semicolon	Terminates statements and separates declarations.
:	Colon	Used in labels
#	Hash (preprocessor)	Used for preprocessor directives (#include, #define).
“	Double quotes	Used for defining string literals.
‘	Single quotes	Used for defining character literals.

Anatomy of a C program



```c
#include <stdio.h>

void userFun(int);

int x = 10;

/* This is a
multi line comment*/

int main() {
    //this is a one line comment
    int y = 10;
    userFun(y);
    y = x + 1;
    return 0;
}

void userFun(int a) {
    printf("%d\n", a);
}
```

pre-processor directive

function prototype

global declaration

comment

main function – entry point

local declaration

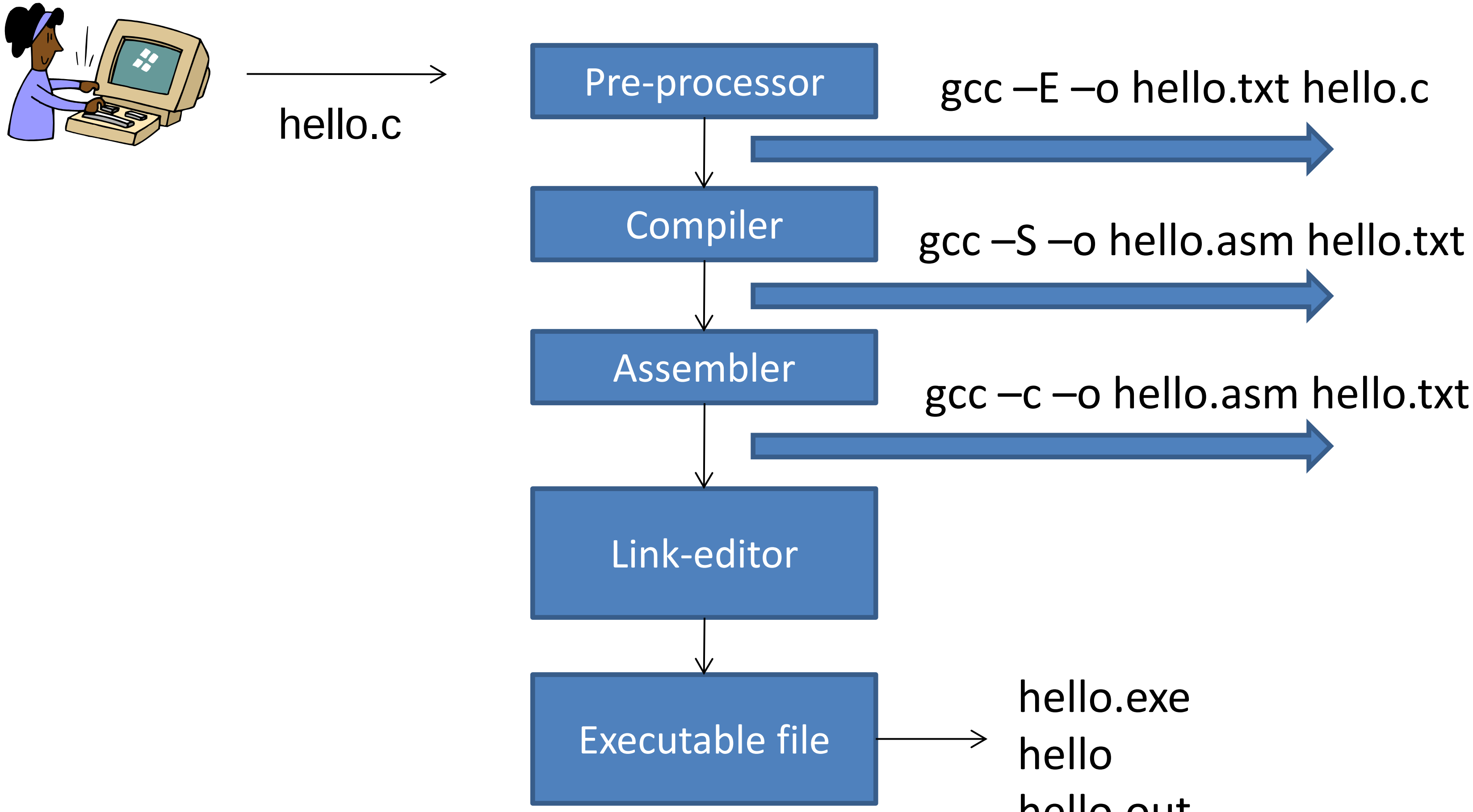
statements

function definition

How to run a program?

- MS Visual Studio Code
- Use the command line interface

How to run a program?



hello.c

Pre-processor

gcc –E –o hello.txt hello.c

Compiler

gcc –S –o hello.asm hello.txt

Assembler

gcc –c –o hello.asm hello.txt

Link-editor

Executable file

hello.exe
hello
hello.out

Data types

- Primitive (int, char, float, double)
- Aggregate (vectors/arrays, 2D arrays, 3D arrays …)
- Local and global

Operators

- Operators in C language are special symbols that perform operations on variables and values.

Type	Operators
Arithmetic	+ - * / %
Relational	== != > < >= <=
Logical	&& || !
Bitwise	& | ^ ~ << >>
Assignment	= += -= *= /= %=
Increment/decrement	++ -- (post, pre)
Ternary	? :
Special	sizeof & *

Logical operators

AND &&	x	y
1	1	1
0	1	0
0	0	1
0	0	0

OR ||	x	y
1	1	1
1	1	0
1	0	1
0	0	0

NOT !	x
0	1
1	0

Bitwise logical operators

AND &	x	y
1	1	1
0	1	0
0	0	1
0	0	0

OR |	x	y
1	1	1
1	1	0
1	0	1
0	0	0

NOT ~	x
0	1
1	0

OR ^	x	y
0	1	1
0	0	0
1	0	1
1	1	0

Operator precedence

Precedence	Operator	Description	Asoc
1	++ --	Suffix/postfix increment and decrement	Left to right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right to left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 1]	
	_Alignof	Alignment requirement(C11)	

Operator precedence

Precedence	Operator	Description	Asoc
3	* / %	Multiplication, division, and remainder	Left to right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	| |	Logical OR	

Operator precedence

Precedence	Operator	Description	Asoc
13	?:	Ternary conditional[note 3]	Right to left
14	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left to right

Flow control instructions

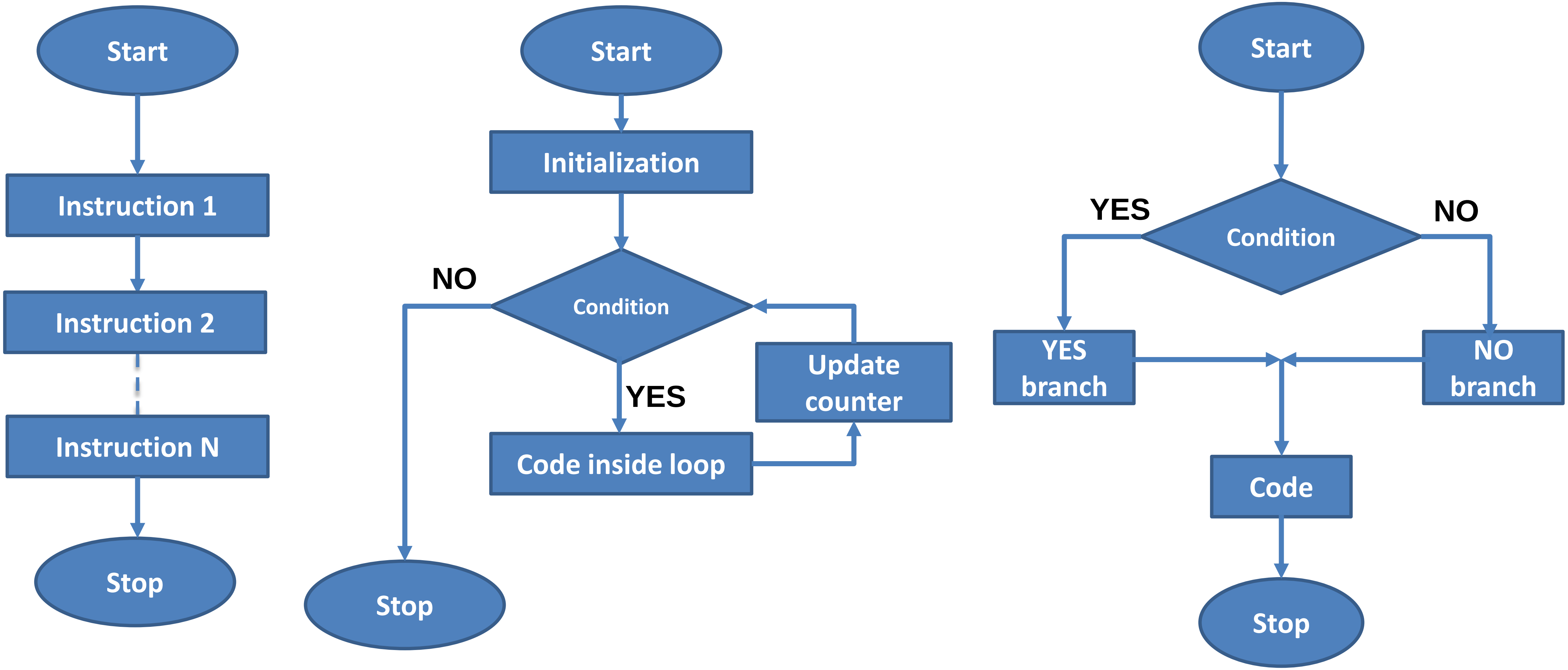
The **Böhm-Jacopini theorem** (1966) is a fundamental concept in computer science that states:

Any computable algorithm can be represented using only three basic control structures: sequence, selection, and iteration, without requiring the use of goto statements.

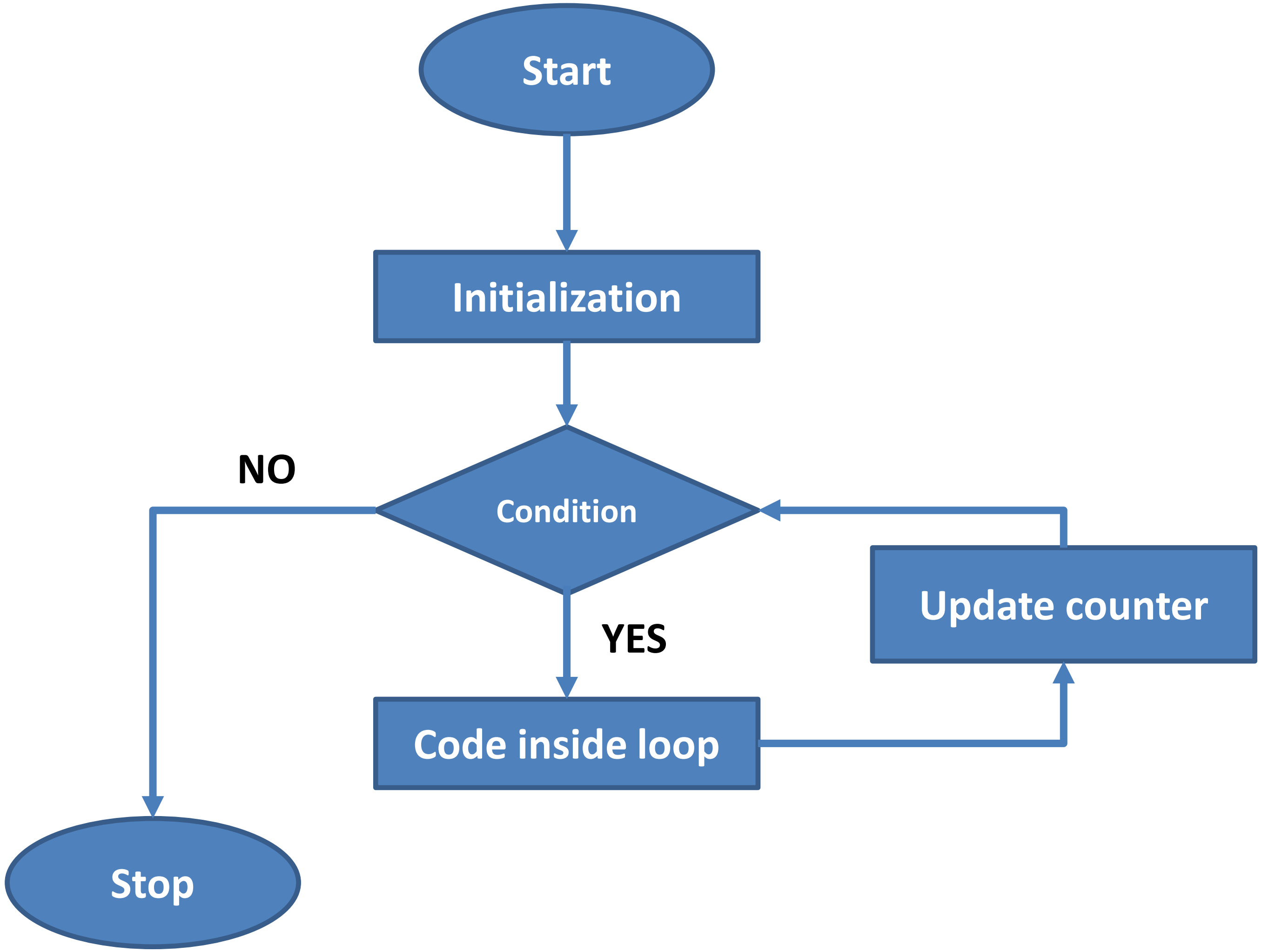
Foundation of Modern Programming Languages

- **Sequence**
- **Selection (decision making)**
- **Iteration**

Flow control instructions

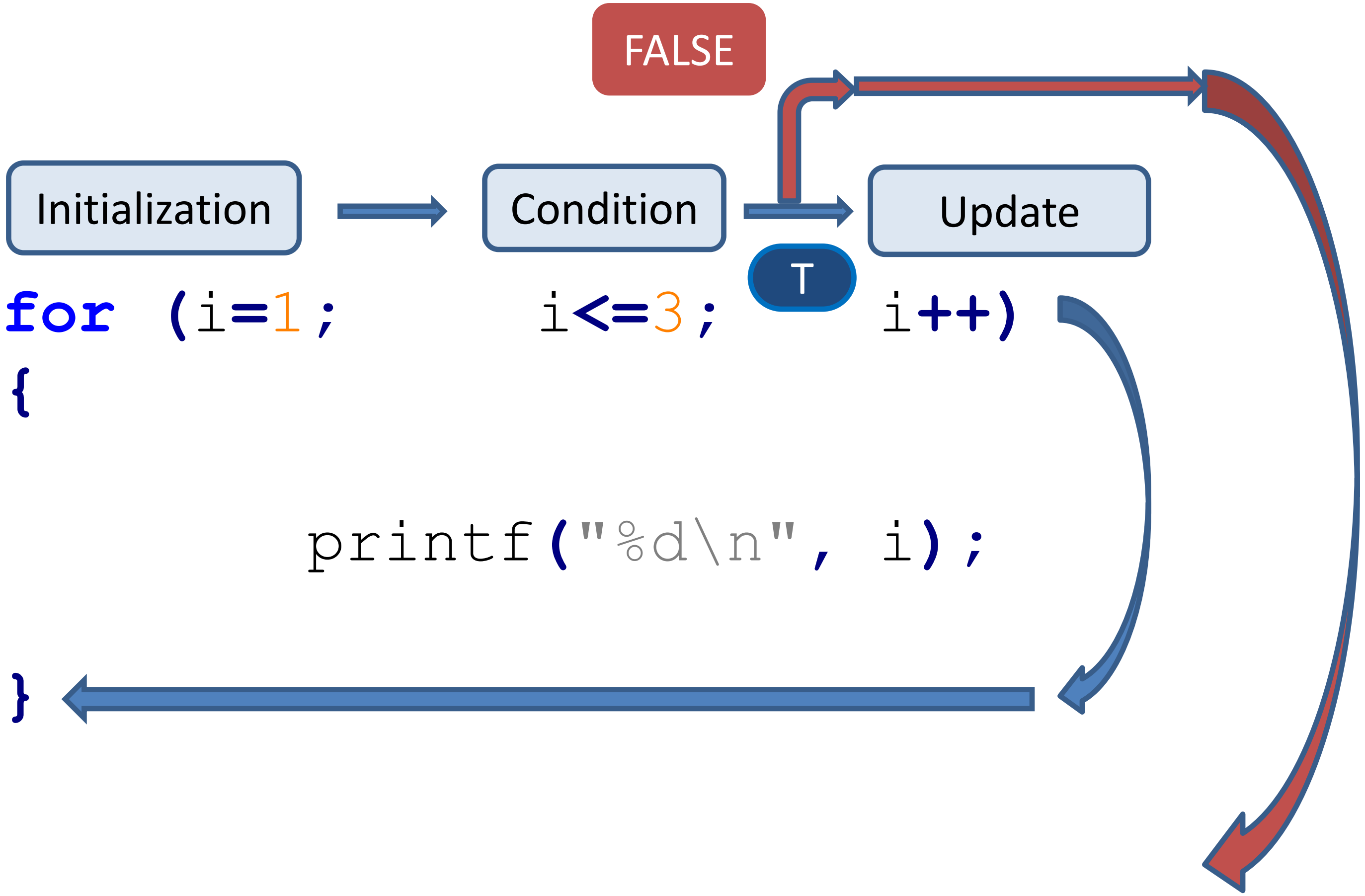


for



```
#include <stdio.h>

int main() {
        int i;
        for (i=1; i<=3; i++)
        {
            printf("%d\n", i);
        }
        return 0;
}
```

for



FALSE

Initialization

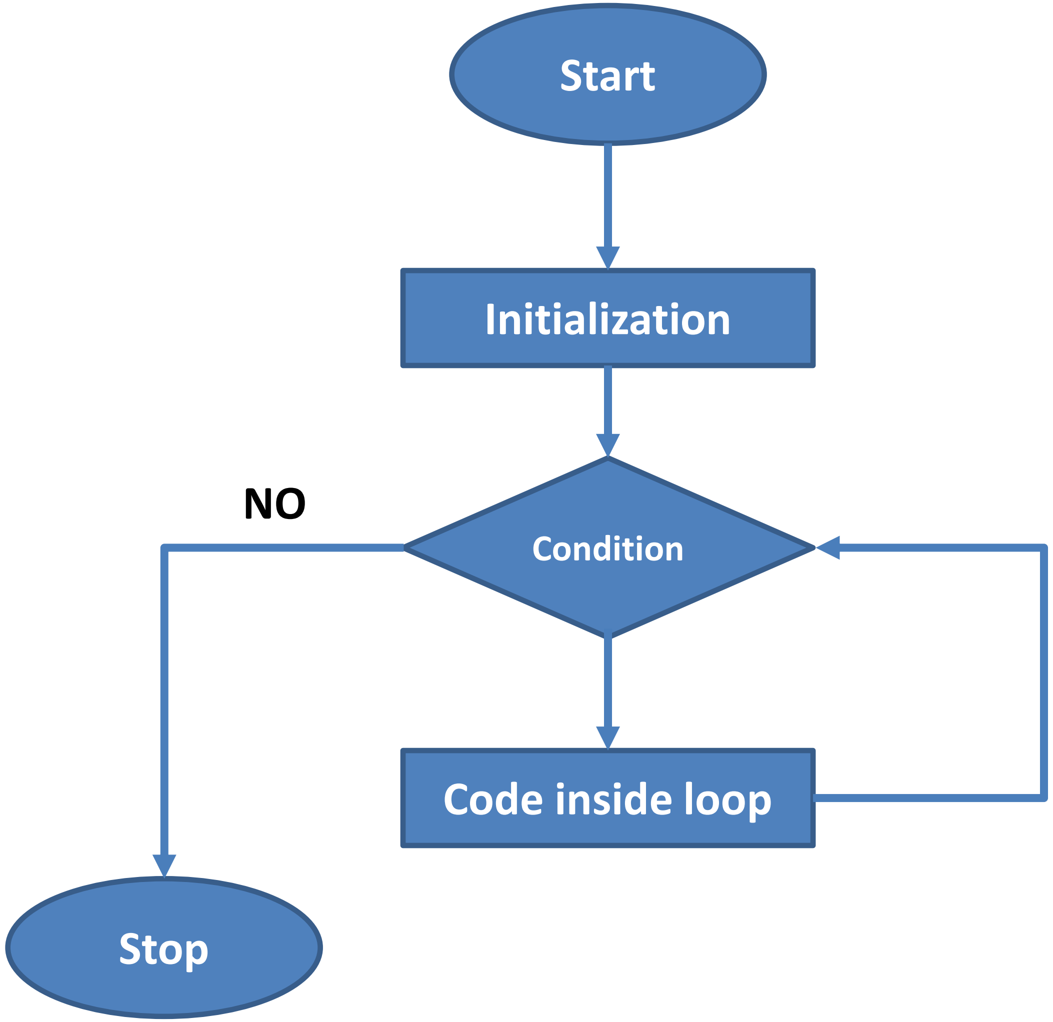
Condition

T

Update

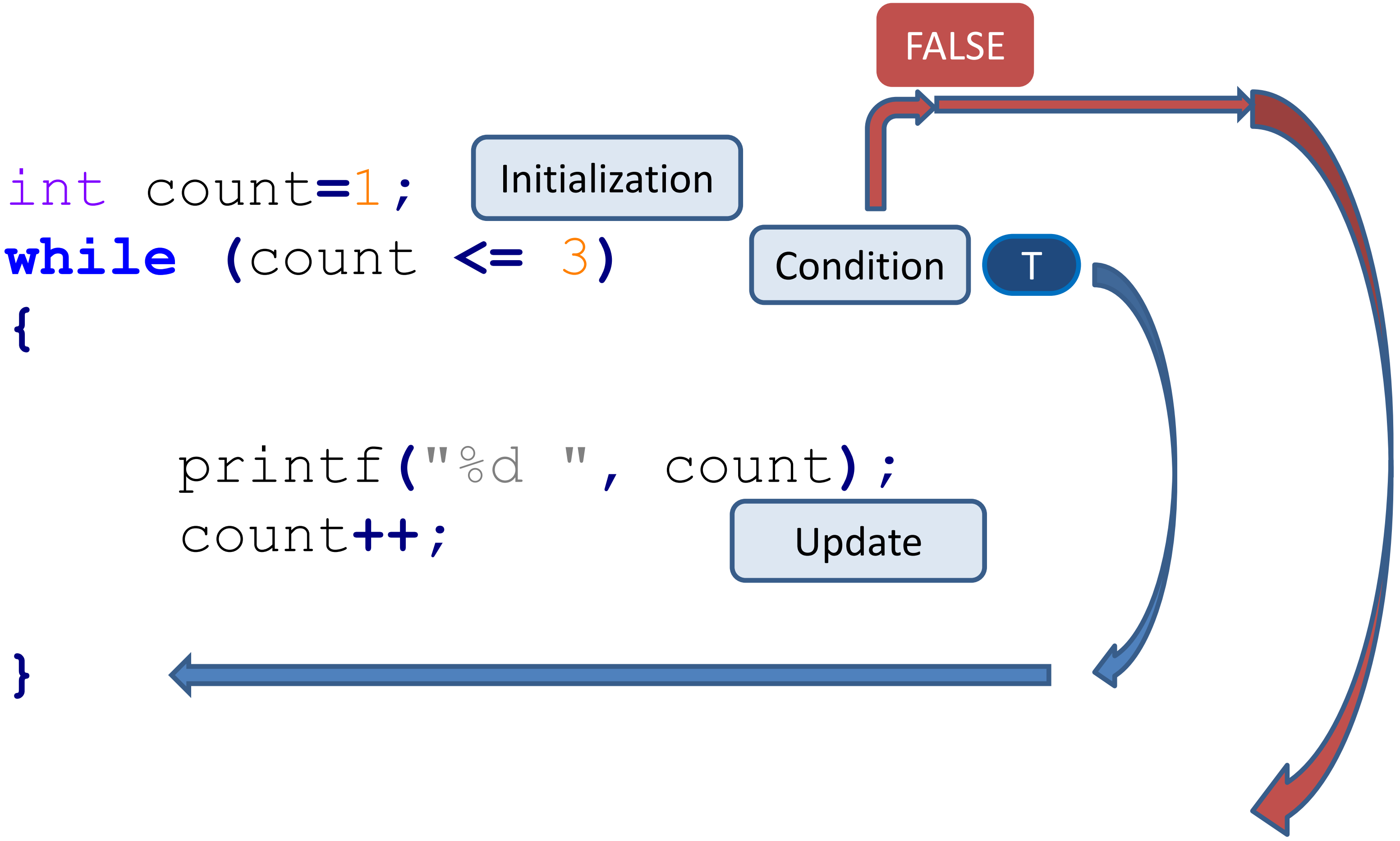
```
for (i=1;      i<=3;      i++)
{

        printf("%d\n", i);

}
```

while



```
#include <stdio.h>
int main() {
    int count=1;
    while (count <= 3) {
        printf("%d ", count);
        count++;
    }
    return 0;
}
```

while



```
int count=1;
while (count <= 3)
{

    printf("%d ", count);
    count++;

}
```

Initialization

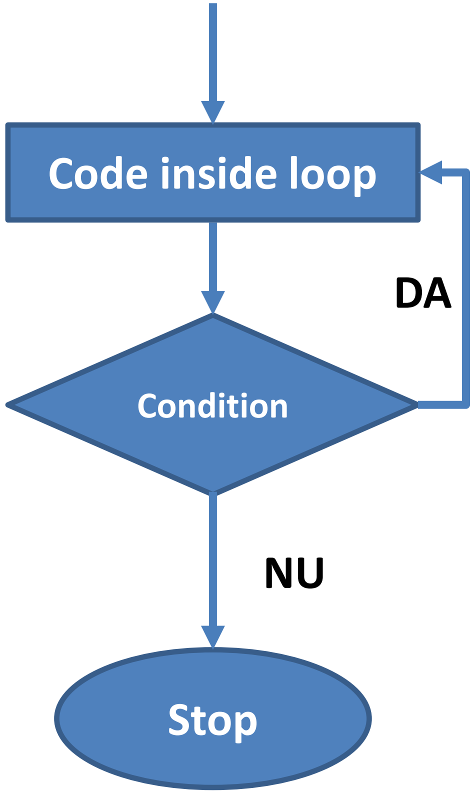
Condition

T

FALSE

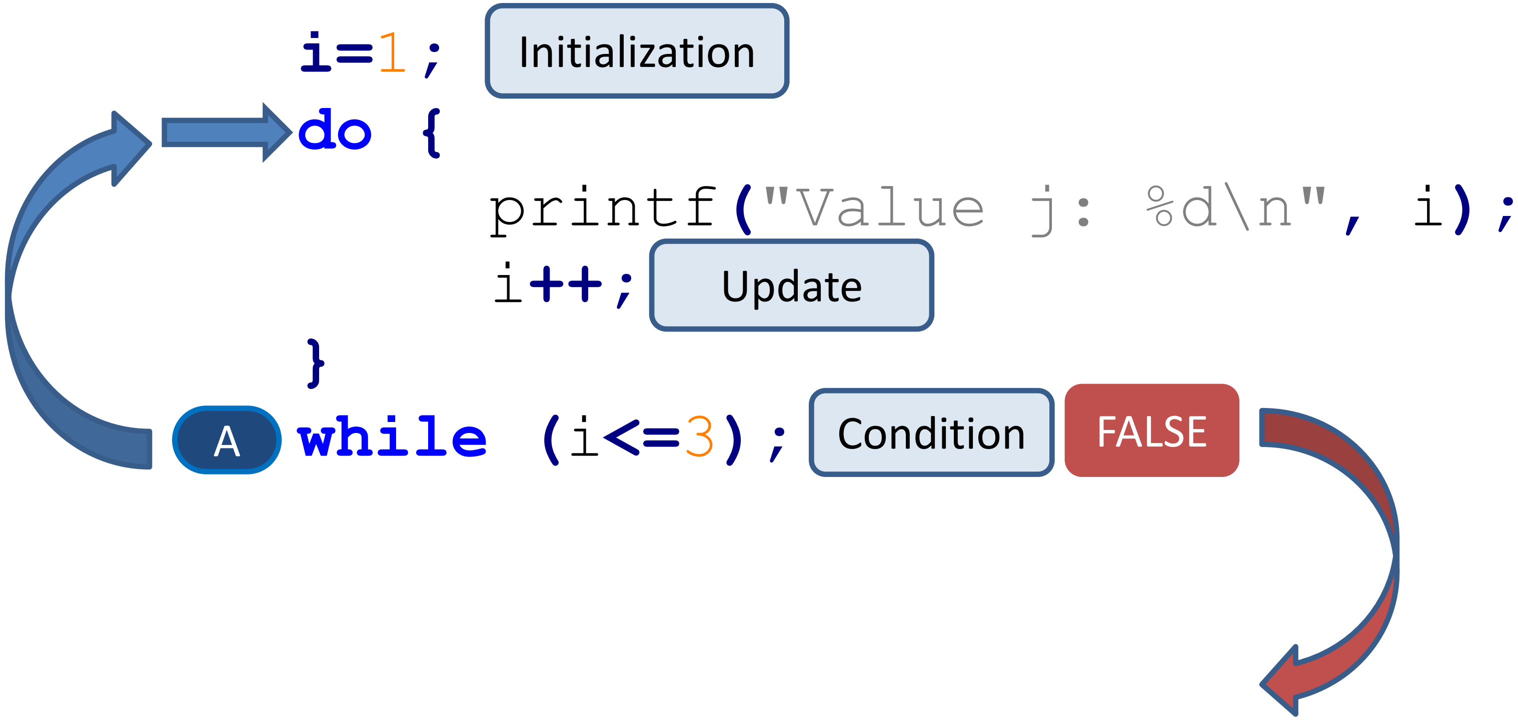
Update

do ...... while



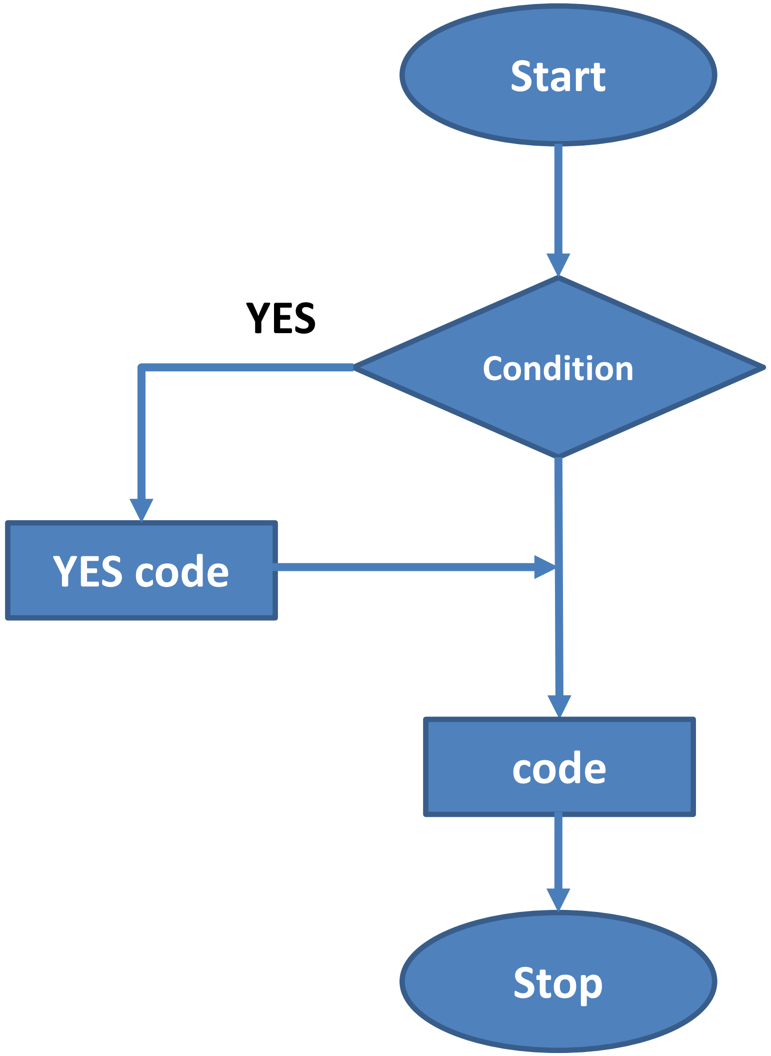
```
#include <stdio.h>
int main() {
     int i;
     i=1;
     do {
          printf("Value j: %d\n", i);
          i++;
     }
     while (i<=3);
     return 0;
}
```

do ...... while



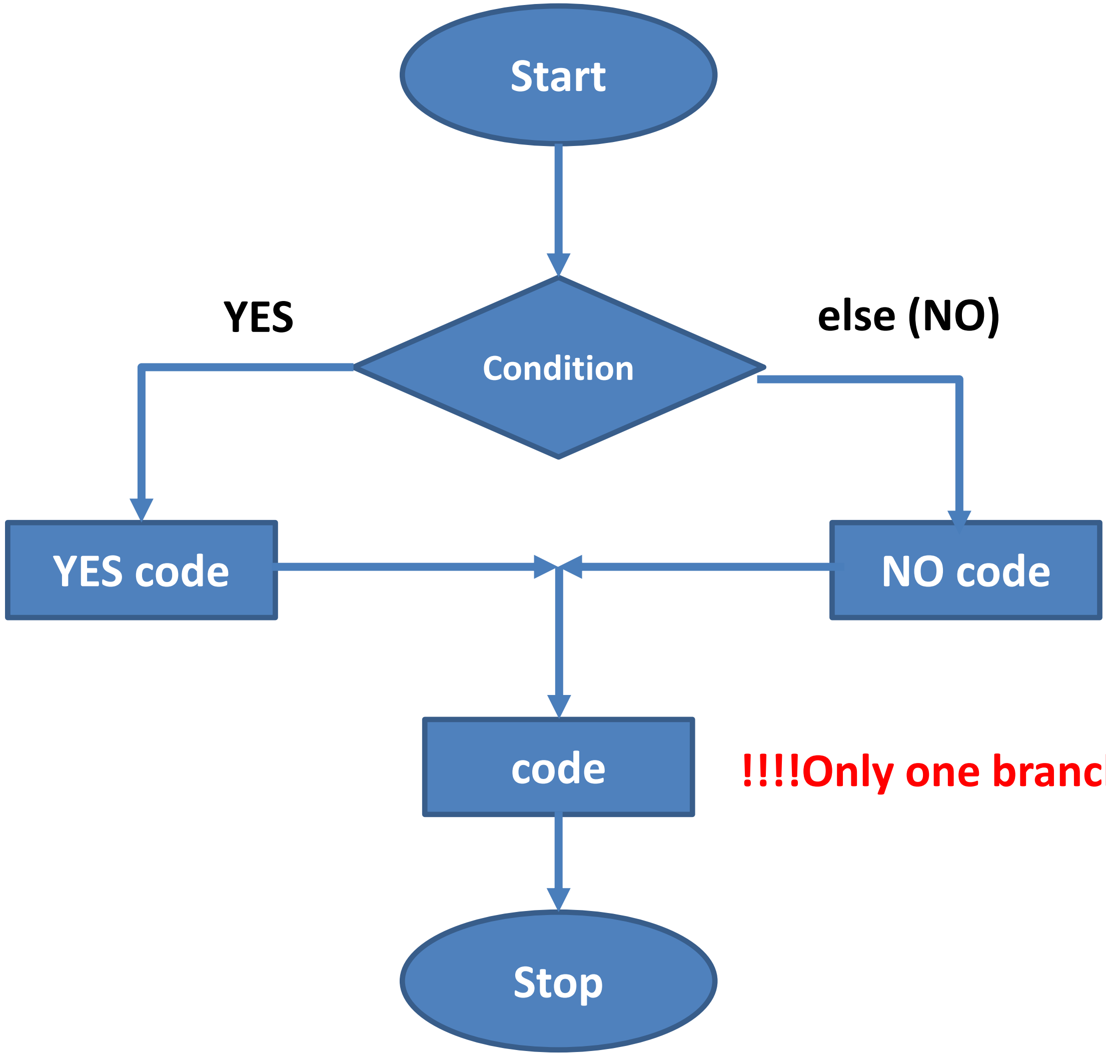
```
i=1;
do {
    printf("Value j: %d\n", i);
    i++;
}
while (i<=3);
```

if



```
#include <stdio.h>
int main()
{
        int i=10;
        if (i<11)
        {
                printf("Salut!"); //YES
        }
        printf(“Good bye!"); //NO
        return 0;
}
```

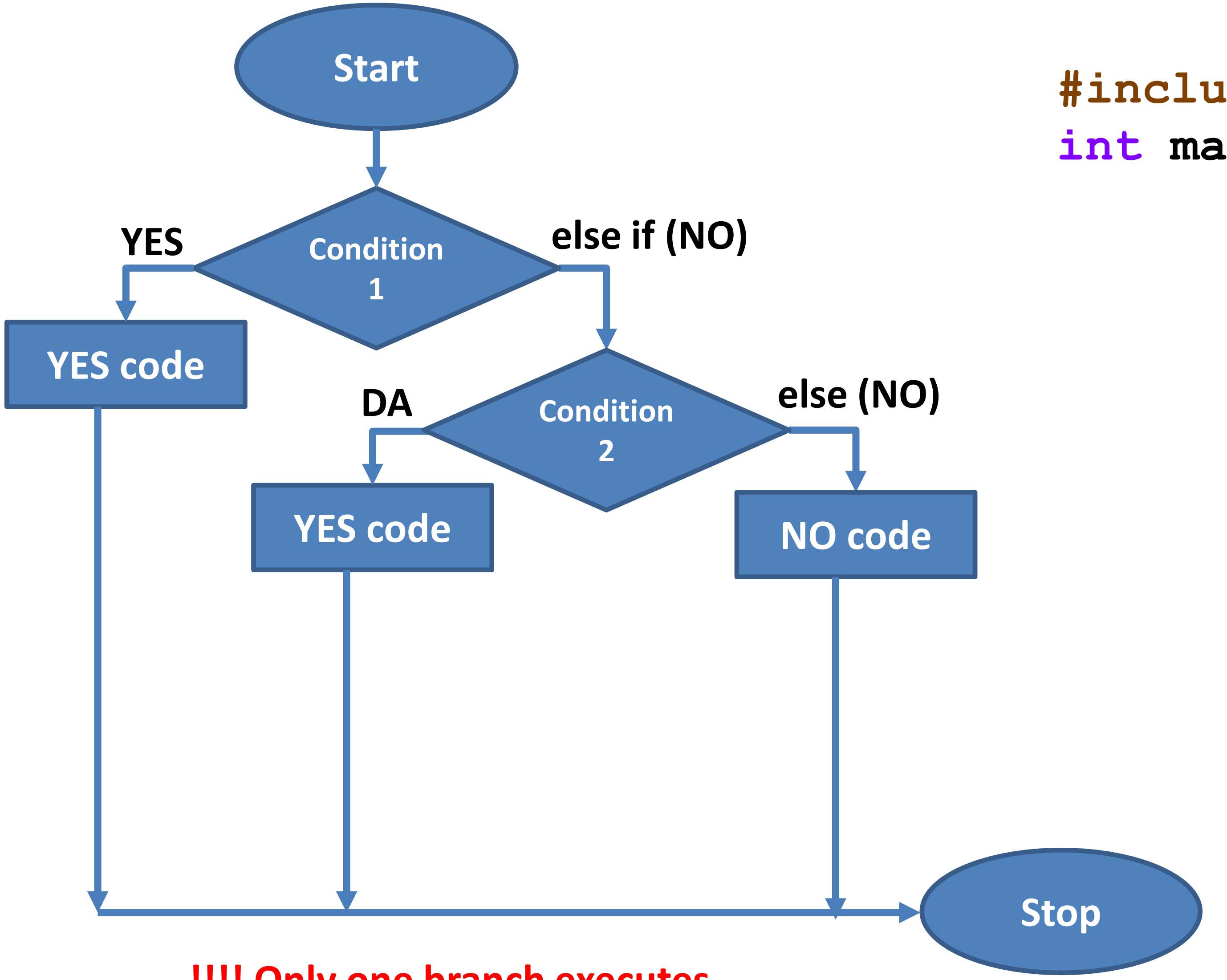
if … else



!!!!Only one branch executes.

```
#include <stdio.h>

int main()
{
      int i=10;
      if (i<11)
      {
            printf("Salut!"); //YES
      }
      else
      {
            printf("Hello!"); //NO
      }
      printf("La revedere!"); //cod

      return 0;
}
```

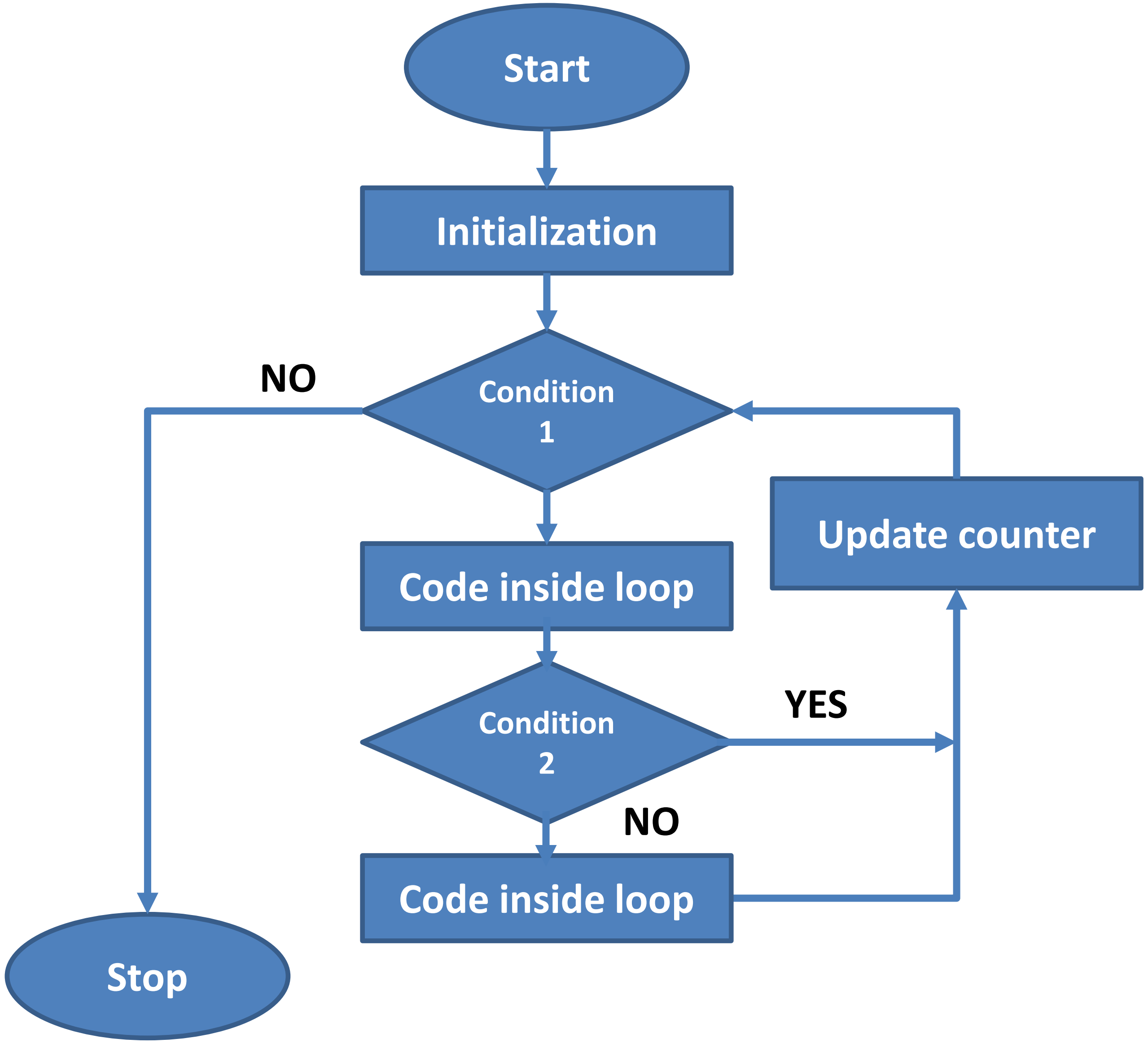
if … else if … else



```
#include <stdio.h>
int main(){
        int i=10;
        if (i<11){
                printf("Salut!"); //YES
        }
        else if (x==11){
                printf("Buna!"); //NO
        }
        else{
                printf("Hello!"); //else code
        }

        printf("La revedere!");
        return 0;
```

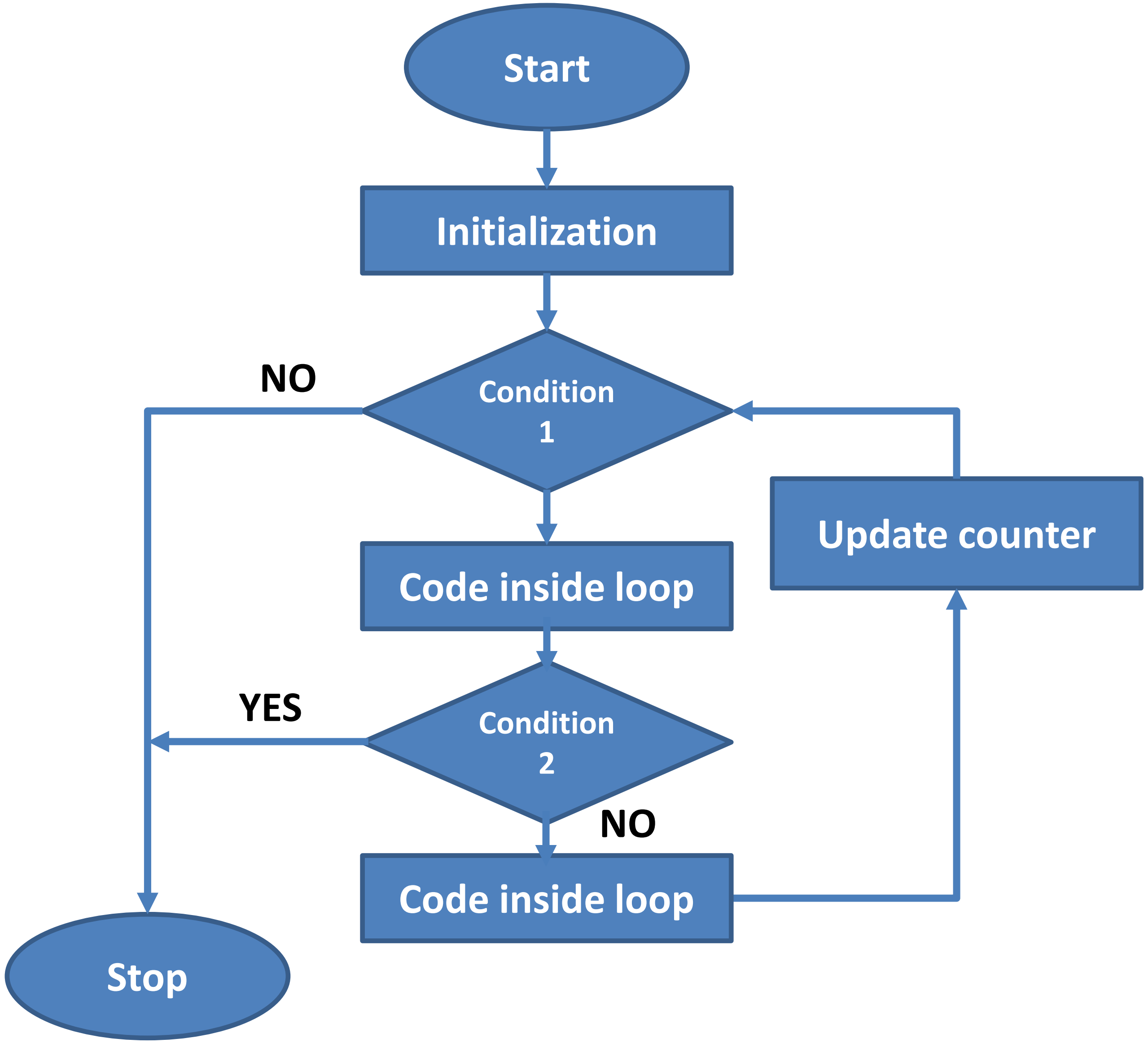
!!!! Only one branch executes.

continue



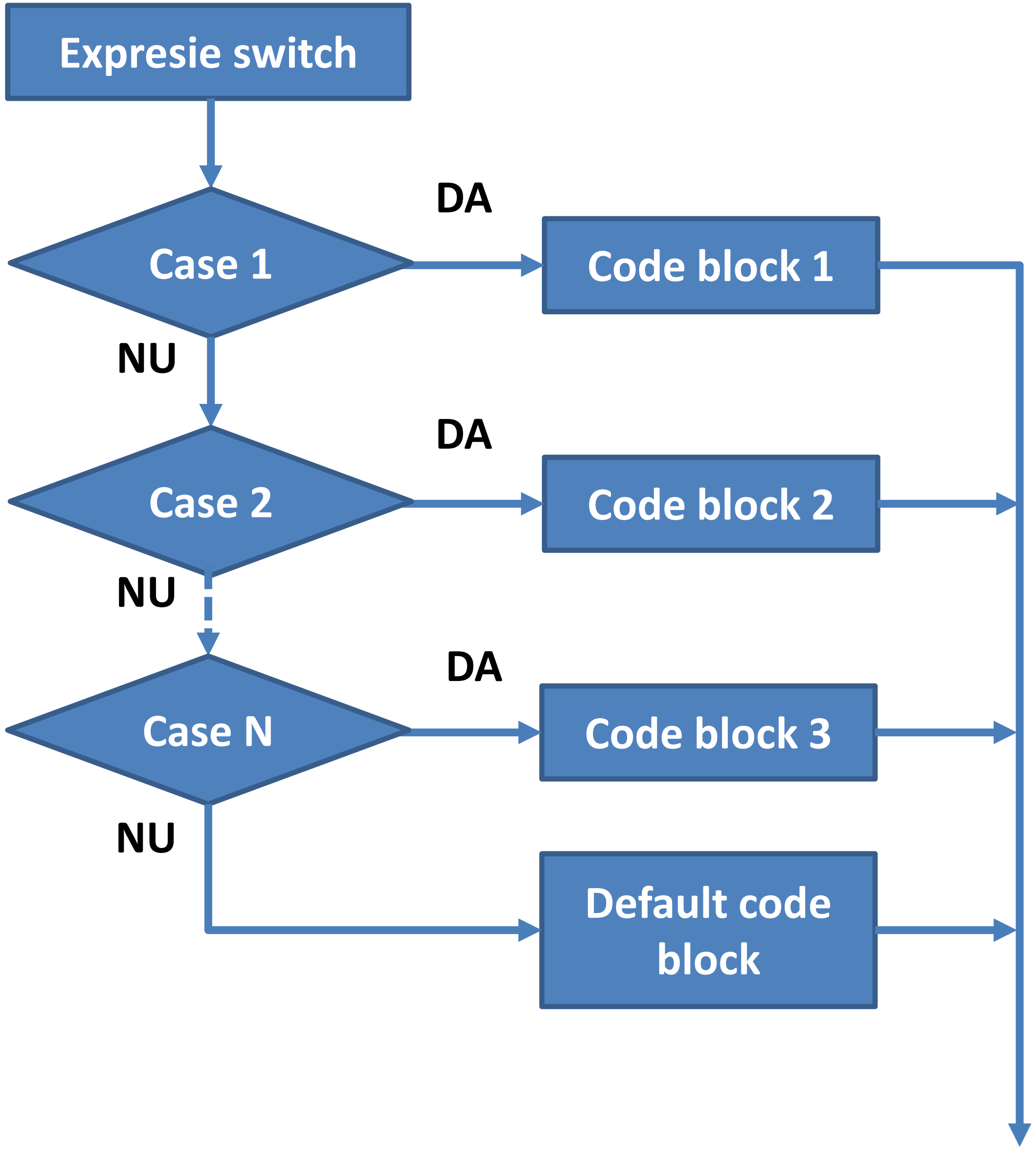
```
#include <stdio.h>

int main()
{
        for(int i=0;i<10;i++)
        {
                printf("Hello!");
                if (i==5) //condition 2
                {
                        continue;
                }
                printf("Goodbye!");
        }
        return 0;
}
```

break



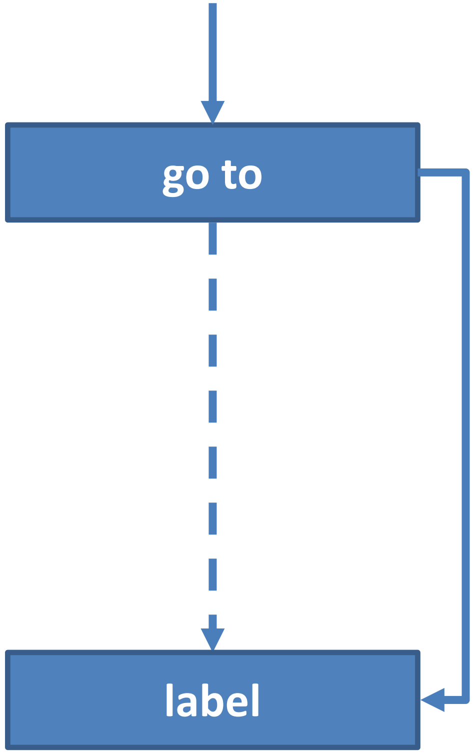
```
#include <stdio.h>

int main()
{
        for(int i=0;i<10;i++)
        {
                printf("Hello!");
                if (i==5) //condition 2
                {
                        break;
                }
                printf("Goodbye!");
        }
        return 0;
}
```

switch . . . . case: . . . default



```
#include <stdio.h>
int main(){
    int i=0;
    switch(i){
      case 1:
         printf(“The variable i is 1\n");
         break;
      case 0:
         printf(“The variable i is 0\n");
         break;
      default:
         printf(“The variable i is %d\n",i);
     }
     return 0;
}
```

goto



UNCONDITIONAL JUMP

```
int test=0;
for(int i=0;i<20;i++){
    for(int j=1;j<5;j++){
       test=test+3;
       if (test==1000){
           goto label;
       }
     }
}

label:
printf("Variable test is big");
```

When do we use go to?

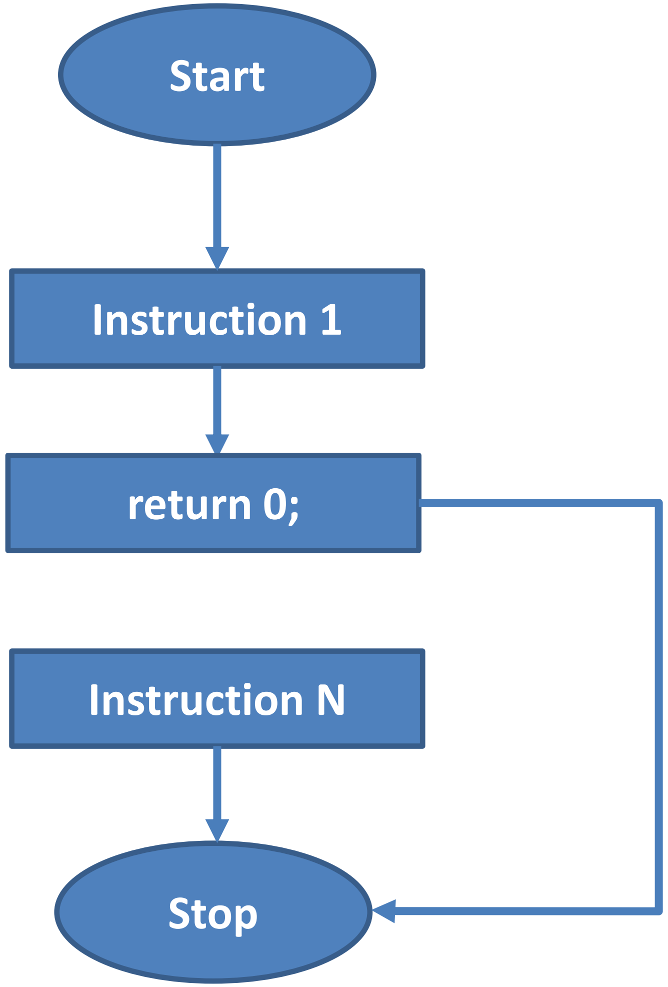
- **NEVER. IT IS BAD PRACTICE!**
- It makes code harder to read (spaghetti code).
- A structured alternative (if, for, while, break, continue) works better.
- It creates unintended jumps, making debugging difficult.

- **HOWEVER ….**

Use case	Why?
Error handling	Centralizes cleanup code (e.g., memory deallocation, file handling).
Breaking out of deeply nested loops	Avoids complex flags and improves clarity.
State machines	Efficient transitions between states (common in low-level systems).

return

- exits the program or function
- see slides on functions



Examples

```
int i = 0;
switch (3/2) {
       case 1: //integer number (int, char)
              printf("Case 1 - Variable i is 1\n");
       //break;
       case 0:
              printf("Case 2 - Variable i is 0\n");
              break;
       default:
              printf("Variable i is %d\n", i);
}
```

Examples

```
int i = 0;
switch (i) {
      case i:
            printf("Case 1 - Variable i is 1\n");
      break;
      case i+1:
            printf("Case 2 - Variable i is 0\n");
            break;
      default:
            printf("Variable i is %d\n", i);
}
```

Examples

```
for (;;)
{
printf("Hello!\n");
}
```

```
while (1)
{
printf("Hello!\n");
}
```

```
do
{
printf("Hello");
} while (1);
```

```
for (int i = 0, j = 0; i < 5,j<3; j++,i++)
{
printf("%d %d\n", i, j);
}
```

```
for (int i = 0, j = 0; i < 3,j<5; j++,i++)
{
printf("%d %d\n", i, j);
}
```

```
for (int i = 0, j = 0; i < 3, j < 5; j++, i++);
{
printf("%d %d\n", i, j);
}
```

```
int i, j;
for (i = 0, j = 0; i < 3, j < 5; j++, i++);
{
printf("%d %d\n", i, j);
}
```

Examples

```
int i = 0, j = 0;
for (i = 0; i < 5; i++)
{
        for (j = 0; j < 4; j++)
        {
                if (i > 1)
                break;
        }
        printf("Hi \n");
}
```

```
int i = 2, j = 2;
while (i + 1 ? --i : j++)
{
        printf("%d", i);
}
```

```
extern int x;
int main()
{
        do
        {
                do
                {
                printf("%o", x);
                }
                while (!- 2);
        }
        while (0);
return 0;
}
int x = 8;
```

```
static int i;
for (++i; ++i; ++i)
{
        printf("%d ", i);
        if (i == 4) break;
}
```

Examples

```
int i = 0, j = 0;
for (i = 0; i < 5; i++)
{
        for (j = 0; j < 4; j++)
        {
                if (i > 1)
                break;
        }
        printf("Hi \n");
}
```

```
int i = 0, j = 0;
for (i = 0; i < 5; i++)
{
        for (j = 0; j < 4; j++)
        {
                if (i > 1)
                        goto xxx;
        }
printf("Hi \n");
}
xxx:printf("Salut!");
```

```
void foo();
int main() {
        foo();
        printf("Salut!");
        return 0;
}
void foo(){
 int i = 0, j = 0;
 for (i = 0; i < 5; i++)
 {
   for (j = 0; j < 4; j++)
   {
   if (i > 1)
          return;
   }
   printf("Hi \n");
 }
}
```

Functions

- A function in C is a block of reusable code that performs a specific task. Functions help in breaking down a program into smaller, manageable parts, improving readability, reusability, and debugging.

- C functions are broadly classified into two types:

Library (Built-in) Functions

- Provided by C standard libraries (e.g., printf(), scanf(), strlen(), sqrt(), etc.).
- Require #include directives for usage (e.g., #include <stdio.h> for printf()).

User-defined Functions

- Created by programmers for specific tasks.

Examples

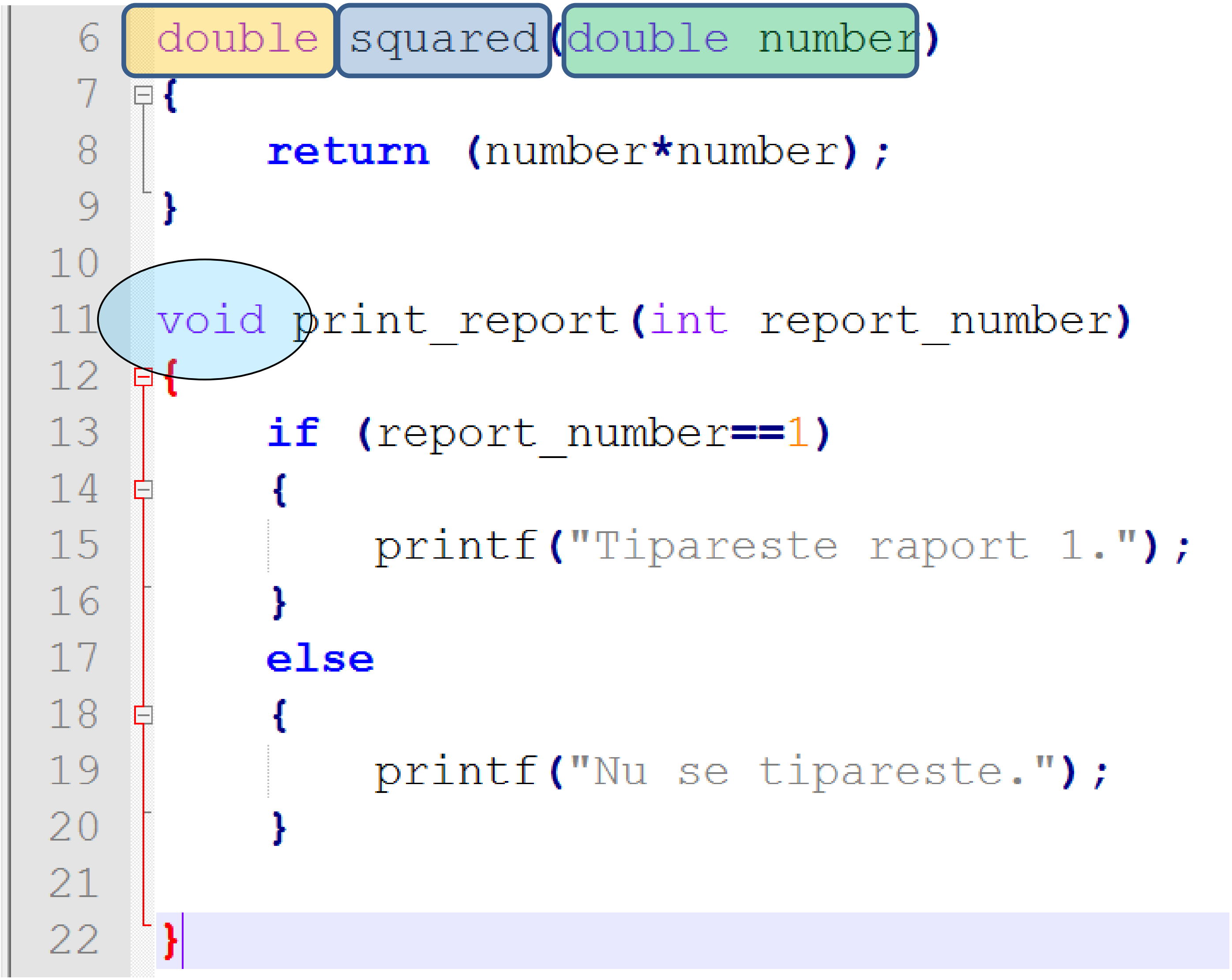
```c
#include <stdio.h>

int twice(int); //PROTOTYPE
void foo();

int main()
{
        int x = 10;
        int y = 0;
        y = x + x;
        y = twice(x);
        foo();
        return 0;
}
```

```c
int twice(int x)
//DEFINITION (IMPLEMENTATION)
{
        return x + x;
}

void foo()
{
        printf("Hello");
}
```

Functions – main concepts

Concept	Description
Function	A reusable block of code performing a task.
Function Prototype	Declares a function before using it.
Function Definition	The actual implementation.
Function Call	Executes the function.
Types of Functions	With/without parameters & return values.
Recursion	A function calling itself.
Inline Functions	Functions optimized for performance.

Definition of a function



```
double squared(double number)
{
    return (number*number);
}

void print_report(int report_number)
{
    if (report_number==1)
    {
        printf("Tipareste raport 1.");
    }
    else
    {
        printf("Nu se tipareste.");
    }

}
```

Return values

- To return a value from a function we use the *return* instruction;
- Sintaxa instrucțiunii *return*:
  - ***return expression;***
  - Expression can be any valid C expression of the type defined in the function head.
  - Use type cast …
  - Can use multiple *return* instructions in a single function (de ex.: is an “if-then-else” …)

Return values

Function type	Description	Example

With return & parameters	Takes input, returns output.	`int add(int a, int b) {`
		`     return a+b;`
		`}`

With parameters, no return	Takes input, prints result.	`void printSum(int a, int b){`
		`     printf(“%d”,a+b);`
		`}`

No parameters, with return	No input, returns value.	`int getValue(){`
		`     return 10;`
		`}`

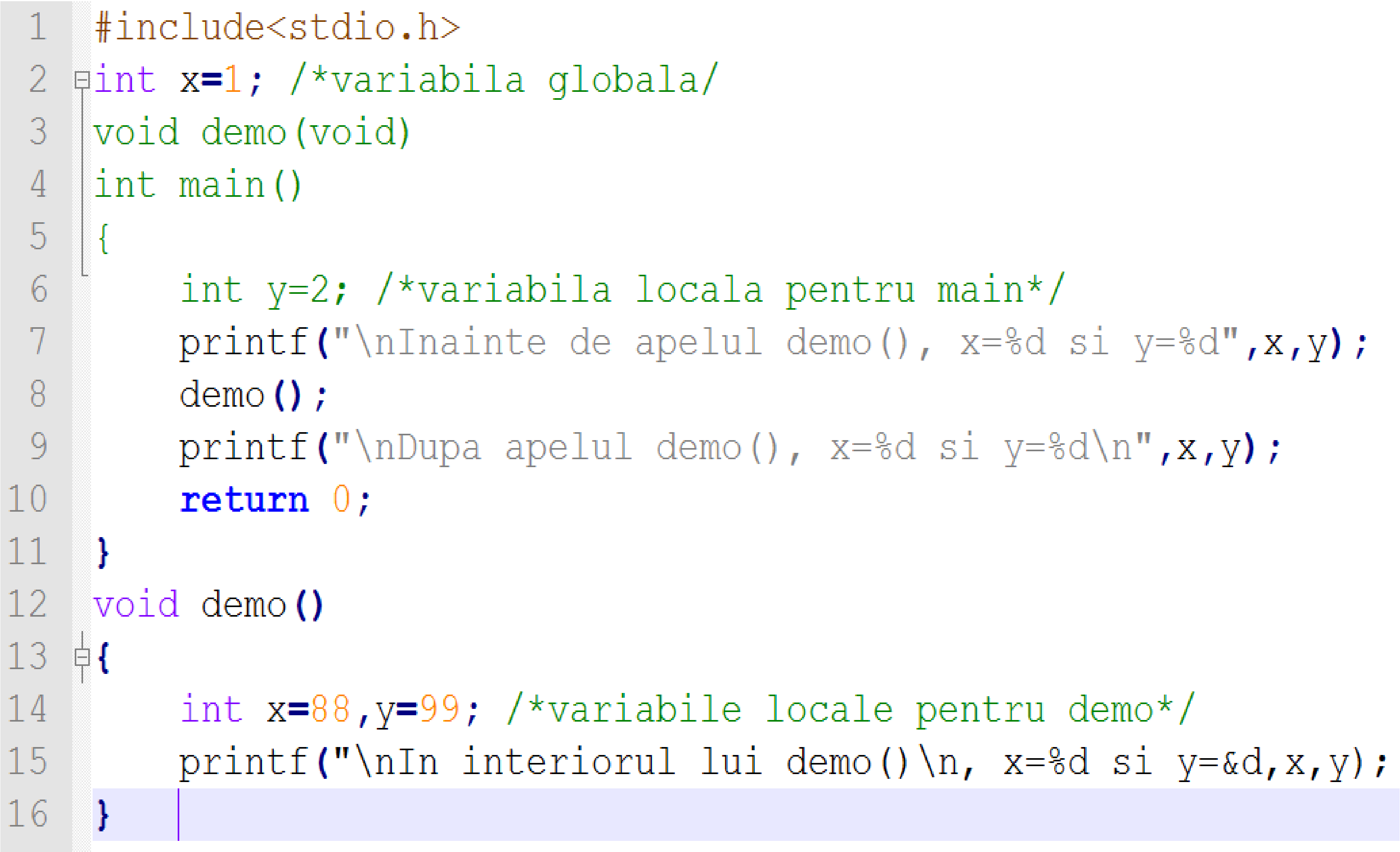
No parameters, no return	Just executes code.	`void greet(){`
		`     printf(“Hello!”);`
		`}`

Local vs. global variables

Use local variable when	Use global variable when
The variable is needed only in a specific function	The variable is used by multiple functions.
To prevent accidental modifications by other functions	To share data between functions without passing parameters.
When you want memory to be freed automatically after function execution	When you need the variable to persist throughout the program.

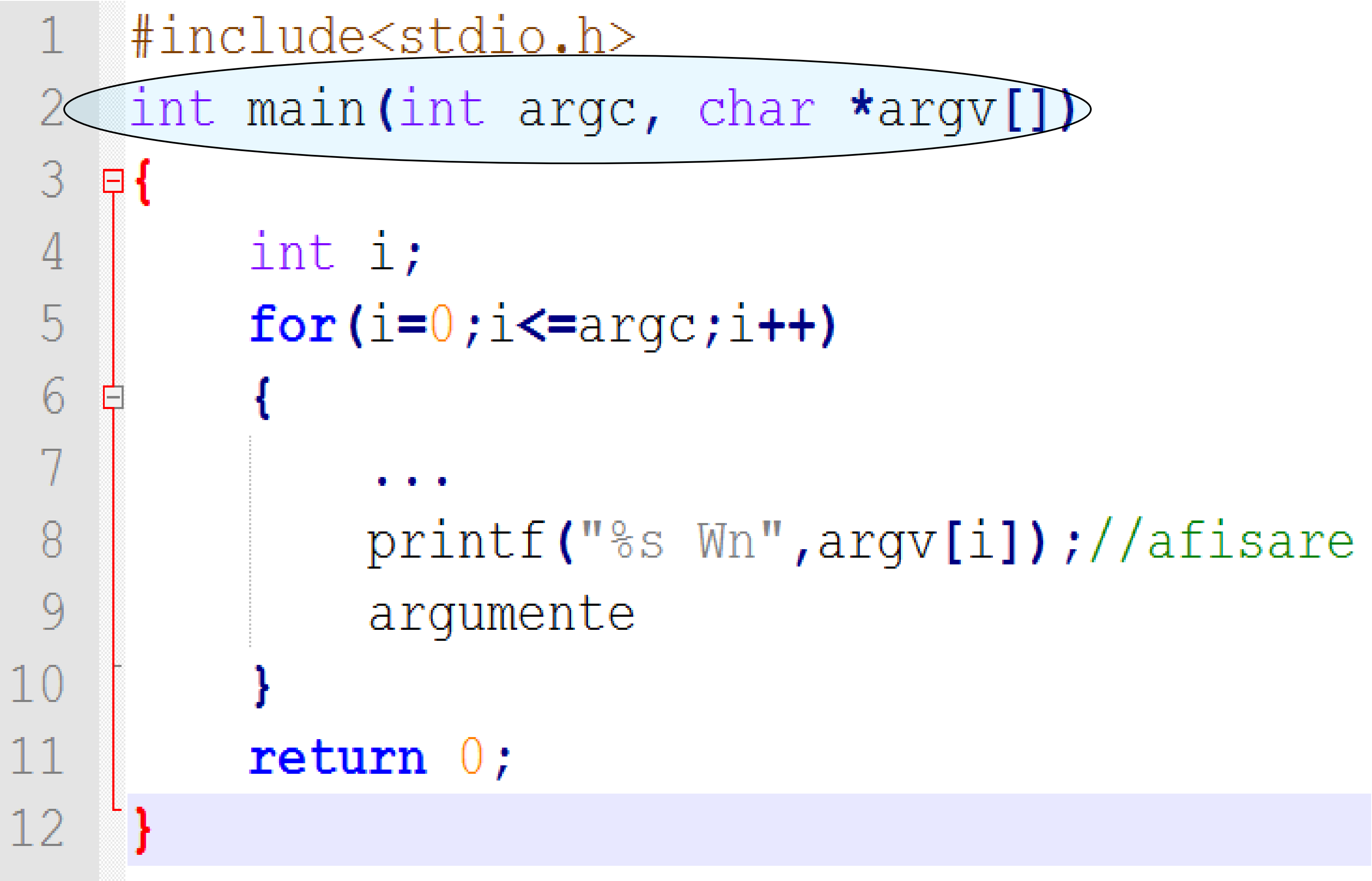
Using global variables in C is generally discouraged because they can introduce several issues that make code harder to manage, debug, and maintain. Key reasons why one should avoid global variables are *reduced maintainability*, *increased risk of name clashes*, *poor modularity*, *thread safety issues*, *unintended side effects*, *increased memory usage* and *debugging complexity*.

Global variables - example



```c
#include<stdio.h>
int x=1; /*variabila globala/
void demo(void)
int main()
{
    int y=2; /*variabila locala pentru main*/
    printf("\nInainte de apelul demo(), x=%d si y=%d",x,y);
    demo();
    printf("\nDupa apelul demo(), x=%d si y=%d\n",x,y);
    return 0;
}
void demo()
{
    int x=88,y=99; /*variabile locale pentru demo*/
    printf("\nIn interiorul lui demo()\n, x=%d si y=&d,x,y);
}
```

Local variables - example



```
#include<stdio.h>
int main(int argc, char *argv[])
{
      int i;
      for(i=0;i<=argc;i++)
      {
            ...
            printf("%s Wn",argv[i]);//afisare
            argumente
      }
      return 0;
}
```

Passing arguments by value

- In C, function arguments (parameters) can be passed in two ways:

Pass by Value (default) – A copy of the argument is passed.

Pass by Reference – A reference (memory address) is passed using pointers.

- If an argument is passed by value, the copy becomes a local variable in the called function.

Strings in C

- Strings in C are character 1D arrays.

Structures, unions, enumerations

- struct, union, enum

Files

- fopen, fclose
- fprintf, fscanf, ftell, fseek, fread, fwrite

Preprocessor

- #include, #define …

To be continued ….

- Pointers
- Character arrays

We discuss pointers in detail, in other lectures.

Questions ???