PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024

STATEMENTS

STATEMENTS

- A *statement* specifies one or more actions to be performed, such as assigning a value to a variable, passing control to a function, or jumping to another statement. The sum total of all a program’s statements determines what the program does.
- Jumps and loops are statements that control the flow of the program. Except when those control statements result in jumps, statements are executed sequentially; that is, in the order in which they appear in the program.
- A statement is an expression followed by a semicolon:

  [*expression*] ;

EXAMPLES

- a + 3 is an expression
- a + 3; is a command

~~If (a + 3;) {~~
~~\\ Do something~~
~~}~~

~~int a= 3;~~
~~a= a + 2~~

IMPERATIVE PROGRAMMING

- In computer science, **imperative** programming is a programming paradigm that uses statements that change a program's state
- Procedural programming is a programming paradigm, derived from structured programming, based upon the concept of the procedure call. Procedures, also known as routines, subroutines, or **functions**
- **C is imperative and procedural**

BLOCK STATEMENT

- A *compound statement*, called a *block* for short, groups a number of statements and declarations together between braces to form a single statement:

  { [*list of declarations and statements*] }

- Unlike simple statements, block statements are not terminated by a semicolon **;**.
- When are they used? <u>A block is used wherever the syntax calls for a single statement, but the program's purpose requires several statements.</u>

```
{
  int a= 5, b= 4, c= 0;
  c= a + b;
}
```

EXAMPLE

```
int a= -4;
if  (a < 0)
    printf(“%d is less than 0”, a);
    a= -a;
    printf(”a now is %d”, a);
```

```
-4 is less than 0
a now is 4
```

~~int a= 4;~~
~~if (a < 0)~~
~~printf(“%d is less than 0”, a);~~
~~a= -a;~~
~~printf(”a now is %d”, a);~~

```
a now is -4
```

```
int a= 4;
if  (a < 0) {
    printf(“%d” is less than 0”, a);
    a= -a;
    printf(”a now is %d”, a);
}
```

EXAMPLE AND SCOPE

```
{   double result = 0.0, x = 0.0;
    long status = 0;
    int limit;

    ++x;                                    // Statements
    if ( status == 0 )
    {                                       // New block
        int i = 0;
        { int k = 3; }                      // Another block
     }
}
```

- Names declared within a block have *block scope*; in other words, they are visible only from their declaration to the end of the block.
- Within that scope, such a declaration can also hide an object of the same name that was declared outside the block

OTHER PARADIGMS

- In computer science, declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow
  - ✓ to minimize or eliminate side effects by describing **what** the program must accomplish, rather than **how** to accomplish it

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)      :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
                 ?- sibling(sally, erica).
                  Yes
```

DEFINITION STATEMENTS

EXAMPLES

- Used to define variables
- Definition of variables = creation in memory
- They need to have different names if the scope is the same

```
int main() {
    int a= 4;
}
```

```
int main() {
    int a= 4;
    float a= 5;
}
```

- They cannot be used as expressions

```
int main() {
    int a= int b= 3;
}
```

LOOPS

LOOPS

- Use a loop to execute a group of statements, called the loop body, more than once. In C, you can introduce a loop by one of three *iteration statements*:

1. while,
2. do ... while,
3. for.

- The number of iterations through the loop body is controlled by a condition, the *controlling expression*.
- This is an expression of a scalar type; that is, an arithmetic expression or a pointer.
- The loop condition is true if the value of the controlling expression is not equal to 0; otherwise, it is considered false.

WHILE STATEMENTS

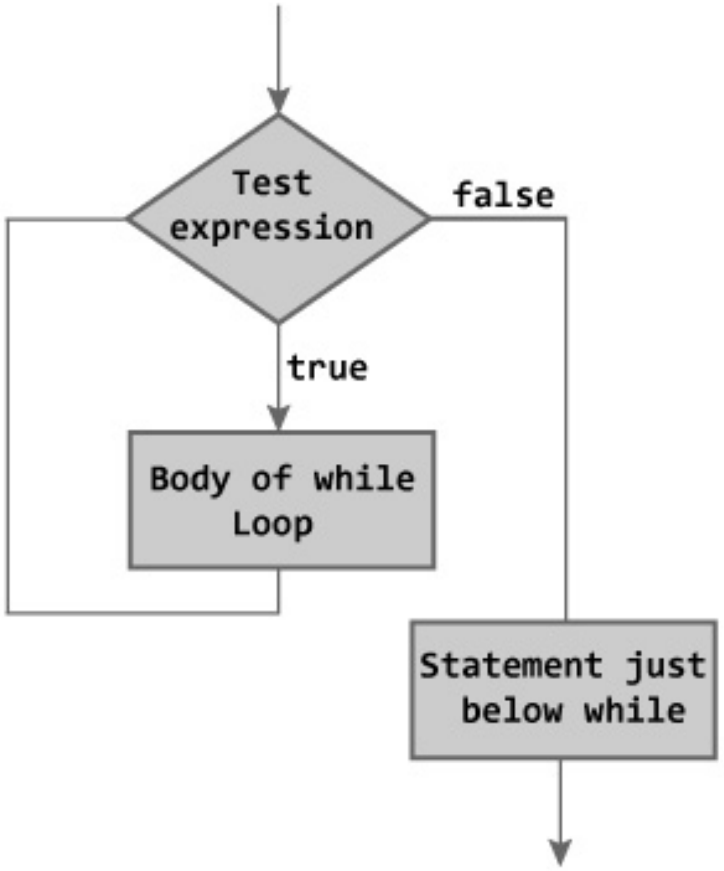
- A while statement executes a statement repeatedly as long as the controlling expression is true:

  while (expression) statement

- The while statement is a *top-driven* loop: first the loop condition (i.e., the controlling expression) is evaluated.
- If it yields true, the loop body is executed, and then the controlling expression is evaluated again.
- If the condition is false, program execution continues with the statement following the loop body.
- Syntactically, the loop body consists of one statement. If several statements are required, they are grouped in a block.

EXAMPLE

```
while (test expression) {
   commands in the body;
}
Statement just below while;
```



```c
#include <stdio.h>

int main () {

   int a = 10;

   while( a < 20 ) {
      printf("value of a: %d\n", a);
      a++;
   }

   return 0;
}
```

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

FOR STATEMENTS

- Like the while statement, the for statement is a top-driven loop, but with more loop logic contained within the statement itself:

for ([expression1]; [expression2]; [expression3]) statement

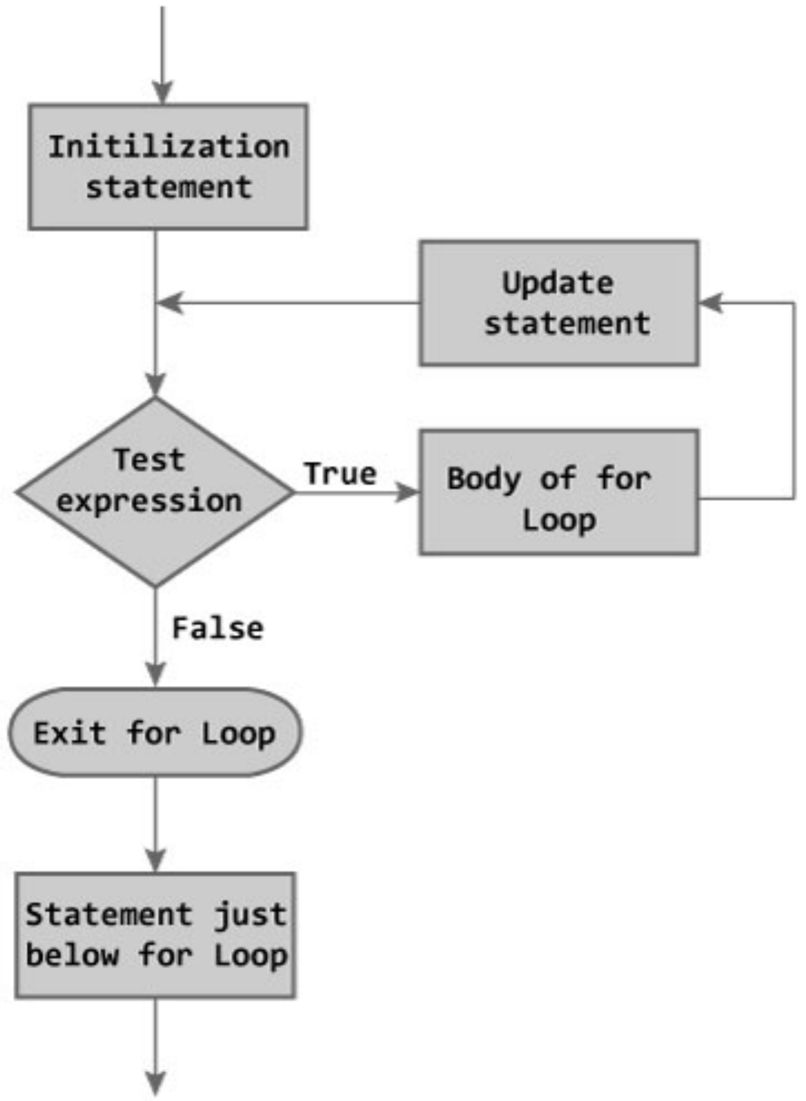
- ✓expression1: Initialization
  Evaluated only once, before the first evaluation of the controlling expression, to perform any necessary initialization.
- ✓expression2*:* Controlling expression
  Tested before each iteration. Loop execution ends when this expression evaluates to false.
- ✓expression3 : Adjustment
  An adjustment, such as the incrementation of a counter, performed after each loop iteration, and before expression2 is tested again.

EXAMPLE

```
for (initializationStatement; testExpression; updateStatement)
{
      // statements in the body
}
```

i = 0
i = 1
i = 2
i = 3
~~i = 4~~

```
int a= 0;
int i;
       1          21              43
for ( i = 0;      i < 4;         ++i ) {

              a = a+2;
              printf("%d", a);   32
}

                 2 4 6 8
```



SAME BEHAVIOUR

i = 1
i = 2
i = 3
i = 4
~~i = 5~~

```
int a= 0;
int i;

for ( i = 1; i <= 4; ++i) {
                a = a+2;
                printf("%d", a);
}
```

```
2 4 6 8
```

FOR AND WHILE

- Any of the three expressions in the head of the for loop can be omitted. This means that its shortest possible form is: for (; ;)
- A missing controlling expression is considered to be always true, and so defines an infinite loop.
- for (; *expression*;) equivalent to a while (*expression*)
- Every for statement can also be rewritten as a while statement, and vice versa.

```
int a=0, i=0;
while (i < 4)
{
        a= a + 2;
        ++i;
}
```

DECLARATION IN PLACE OF EXPR1

- Since ANSI C99, a declaration can also be used in place of *expression1*. In this case, the scope of the variable declared is limited to the for loop. For example:

```
for ( int i = 0; i < 4; ++i ) {
        a = a + 2;
}
```

- The variable i declared in this for loop, no longer exists after the end of the for loop.

DO ... WHILE STATEMENTS

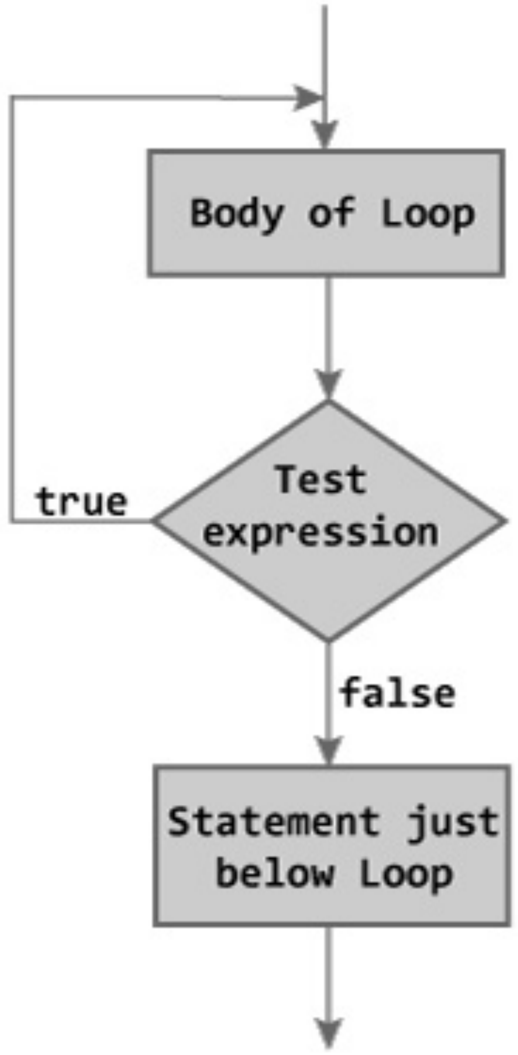
- The do ... while statement is a *bottom-driven* loop:

  do statement while (expression);

- The loop body statement is executed once before the controlling *expression* is evaluated for the first time.
- Unlike the while and for statements, do ... while ensures that at least one iteration of the loop body is performed.
- If the controlling expression yields true, then another iteration follows. If false, the loop is finished.

EXAMPLE

```
do {
   Body of the loop;
} while (test expression)
Statement just below Loop;
```



```c
#include <stdio.h>
int main()
{
    double number, sum = 0;

    // loop body is executed at least once
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &number);
        sum += number;
    }
    while(number != 0.0);

    printf("Sum = %.2lf",sum);

    return 0;
}
```

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

SELECTION STATEMENTS

SELECTION STATEMENTS

- A selection statement can direct the flow of program execution along different paths depending on a given condition. There are two selection statements in C:

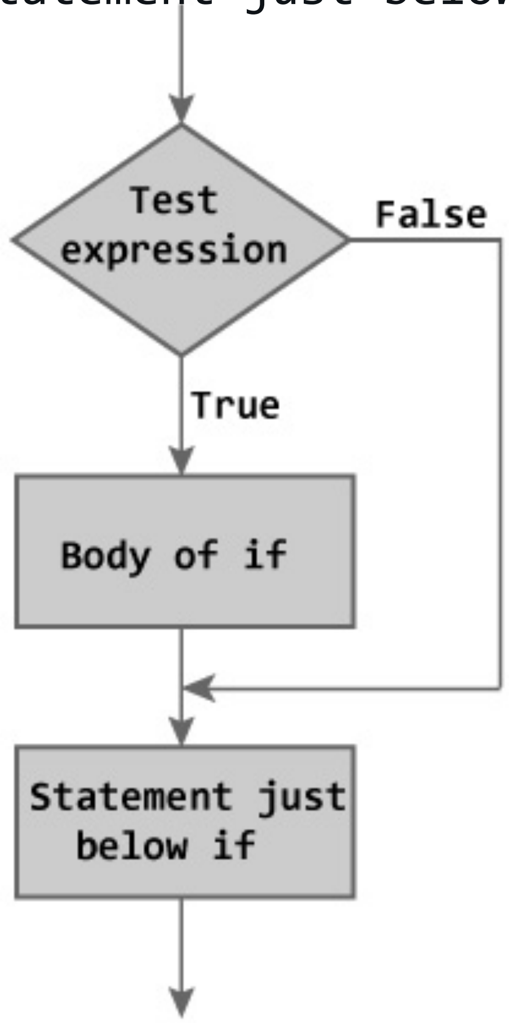
1. If,
2. switch.

IF STATEMENT

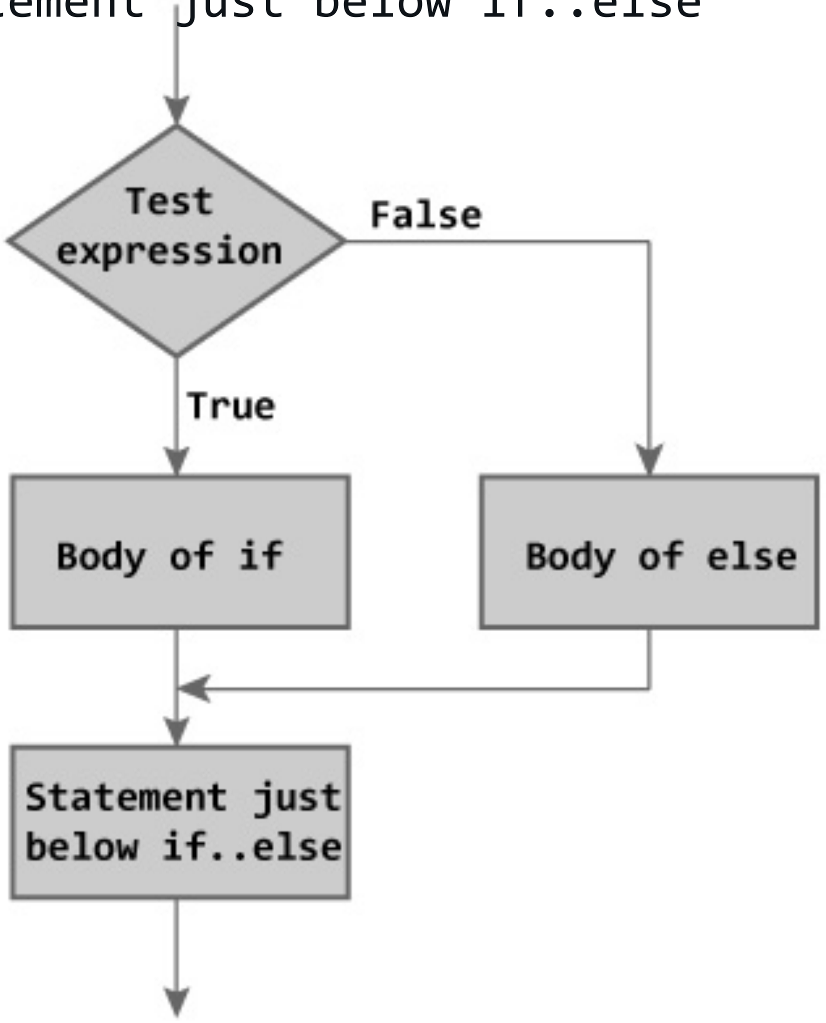
- An if statement has the following form:
  if (*expression*) *statement1* [else *statement2*]
- The **else** clause is optional.
- The *expression* is evaluated first, to determine which of the two statements is executed. This expression must have a scalar type.
- If its value is true -that is, not equal to 0- then *statement1* is executed. Otherwise, *statement2*, if present, is executed.

FLOW

```
if (test Expression)
{
   // Body of If
}
Statement just below if;
```



```
if (testExpression) {
    // Body of If
}
else {
    // Body of Else
}
Statement just below if..else
```



NESTED IF

- If several if statements are nested, then an else clause always belongs to the last if (on the same block nesting level) that does not yet have an else clause:

```
if ( n > 0 )
    if ( n % 2 == 0 )
        puts( "n is positive and even" );
    else                                        // This is the alternative
        puts( "n is positive and odd" );        // to the last if
```

NESTED IF (2)

- An else clause can be assigned to a different if by enclosing the last if statement that should not have an else clause in a block:

```
if ( n > 0 )
{
    if ( n % 2 == 0 )
        puts( "n is positive and even" );
}
else                                        // This is the alternative
    puts( "n is negative or zero" );        // to the first if
```

NESTED IF (3)

- To select one of more than two alternative statements, if statements can be cascaded in an **else if** chain. Each new if statement is simply nested in the else clause of the preceding if statement:

```
double spec = 10.0, measured = 10.7, diff;
diff = measured - spec;

if ( diff >= 0.0 && diff < 0.5 )
    printf( ”Rounding down: %.2f\n", diff );
else if ( diff >= 0.5 && diff < 1 )
    printf( ”Rounding down: %.2f\n", diff );
else
    printf( ”Difference greater than 1!\n" );
```

- As soon as one of these expression yields true, the corresponding statement is executed. The rest is discarded
- If none of the if conditions is true, then the last if statement’s else clause is executed, if present.

SWITCH

- A switch statement causes the flow of program execution to jump to one of several statements according to the value of an integer expression:

  switch (expression) statement

- *expression* has an integer type, and *statement* is the switch body, which contains case labels and at most one default label.
- The expression is evaluated once and compared with constant expressions in the case labels.
- If the value of the expression matches one of the case constants, the program flow jumps to the statement following that case label.
- If none of the case constants matches, the program continues at the **default** label, if there is one.

EXAMPLE

```
switch ( menu() )               // Jump depending on the result of menu().
{
   case 'A':         action1();                    // Carry out action 1
                     break;                        // Don't do any other "actions."
   case 'B':         action2();                    // Carry out action 2
                     break;                        // Don't do the default "action."
   default:          putchar( '\a' );              // If no recognized command
}
```

INTERVALS

GCC extension

```
switch (value) {
    case 1 ... 8:
        printf("Hello, 1 to 8\n");
        break;
    default:
        printf("Hello, default\n");
        break;
}
```

```
switch(value)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
        printf("Hello, 1 to 8\n");
        break;
    default:
        printf("Hello, default\n");
        break;
}
```

ONE MORE EXAMPLE

```
switch (value) {
case 1:
    /* do 1 */
    break;
case 2:
    /* do 2 */
    break;
default:
    if (value >= 3 && value <= 8)
        printf("Hello, 3 to 8\n");
}
```

However, if you need to work with many intervals, use **IF**

ONE MORE EXAMPLE

```
Excellent
Good
OK
Mmmmm....
You must do better than this
What is your grade anyway?
```

```
#include <stdio.h>

main()
{
     int  Grade = 'A';

     switch( Grade )
     {
        case 'A' : printf( "Excellent\n" );
        case 'B' : printf( "Good\n" );
        case 'C' : printf( "OK\n" );
        case 'D' : printf( "Mmmmm....\n" );
        case 'F' : printf( "You must do better than this\n" );
        default  : printf( "What is your grade anyway?\n" );
     }
}
```

UNCONDITIONAL JUMPS

UCONDITIONAL JUMP STATEMENTS

- *Jump statements* interrupt the sequential execution of statements, so that execution continues at a different point in the program.
- A jump destroys automatic variables if the jump destination is outside their scope.
- There are four statements that cause unconditional jumps in C:
  - ✓break,
  - ✓continue,
  - ✓goto, and
  - ✓return.

BREAK STATEMENT

- The break statement can occur only in the body of a **loop** or a **switch** statement, and causes a jump to the first statement after the loop or switch statement in which it is immediately contained:

```
break;
```

- Thus the break statement can be used to end the execution of a loop statement at any position in the loop body.

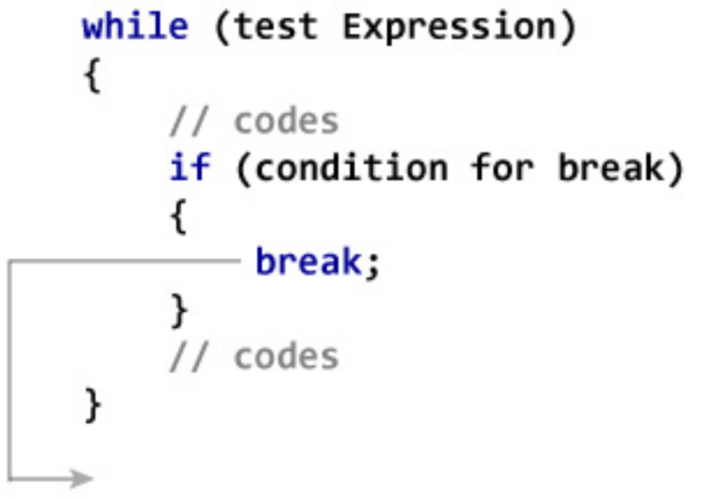
CONTINUE

- The continue statement can be used only within the body of a **loop**, and causes the program flow to skip over the rest of the current iteration of the loop:

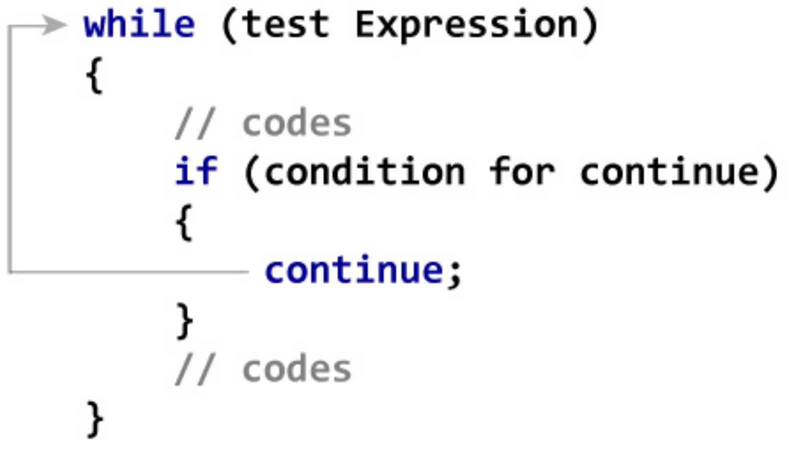
  ```
  continue;
  ```

- In a **while** or **do ... while** loop, the program jumps to the next evaluation of the loop's controlling expression.
- In a **for** loop, the program jumps to the next evaluation of the third expression in the **for** statement, containing the operations that are performed after every loop iteration.

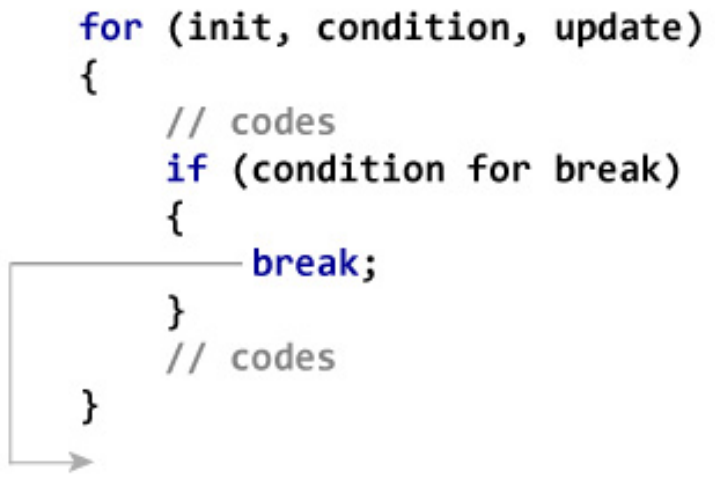
EXAMPLE



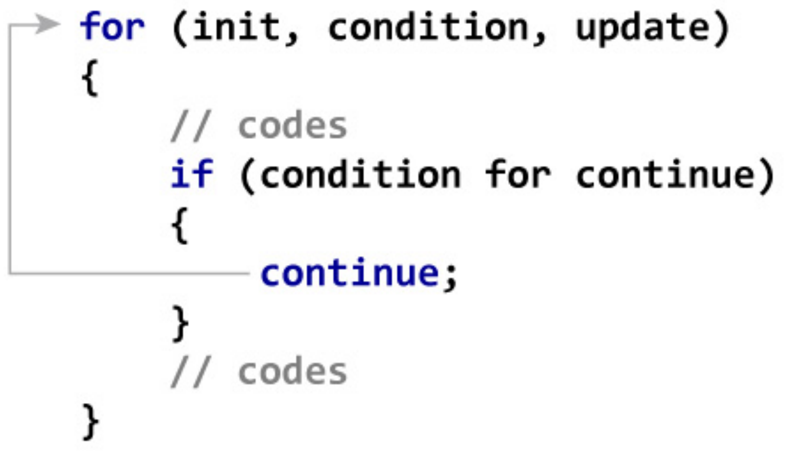
```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

EXAMPLE

```c
# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;

    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        // If user enters negative number, loop is terminated
        if(number < 0.0)
        {
            break;
        }

        sum += number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

```
Enter a n1: 2.4
Enter a n2: 4.5
Enter a n3: 3.4
Enter a n4: -3
Sum = 10.30
```

EXAMPLE

```c
# include <stdio.h>
int main()
{
    int i;
    double number, sum = 0.0;

    for(i=1; i <= 10; ++i)
    {
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);

        // If user enters negative number, loop is continued
        if(number < 0.0)
        {
            continue;
        }

        sum += number;
    }

    printf("Sum = %.2lf",sum);

    return 0;
}
```

```
Enter a n1: 1.1
Enter a n2: 2.2
Enter a n3: 5.5
Enter a n4: 4.4
Enter a n5: -3.4
Enter a n6: -45.5
Enter a n7: 34.5
Enter a n8: -4.2
Enter a n9: -1000
Enter a n10: 12

Sum = 59.70
```

GOTO

- The goto statement causes an unconditional jump to another statement in the same function. The destination of the jump is specified by the name of a label:

  goto label_name;

- A *label* is a name followed by a colon:

  label_name*: *statement

- Labels have a name space of their own, which means they can have the same names as variables or types without causing conflicts.
- Labels may be placed before any statement, and a statement can have several labels. Labels serve only as destinations of goto statements, and have no effect at all if the labeled statement is reached in the normal course of sequential execution.

EXAMPLE

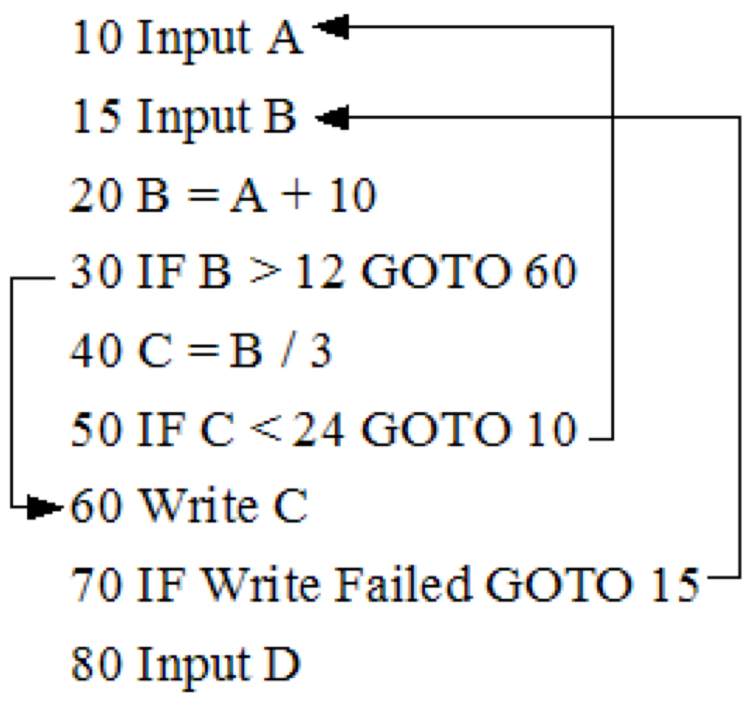
```
void calculate(int a)
{
    if ( a < 1 || a > 5 )
        goto here;

    printf(“a between 1 and 5!\n”)

    return;

    here: printf(“a < 1 or a > 5!\n”);
    return;
}
```

GOTO CAVEAT

- Because code that makes heavy use of goto statements is hard to read, you should use them only when they offer a clear benefit, such as a quick exit from deeply nested loops.
- Any C program that uses goto statements can also be written without them!

SPAGHETTI CODE

- Spaghetti code is a pejorative phrase for source code that has a complex and tangled control structure, especially one using many GOTO statements, exceptions, threads, or other "unstructured" branching constructs.



```
10 Input A
15 Input B
20 B = A + 10
30 IF B > 12 GOTO 60
40 C = B / 3
50 IF C < 24 GOTO 10
60 Write C
70 IF Write Failed GOTO 15
80 Input D
```

Assembly language, Fortran, BASIC

TYPEDEF

MAKE IT SIMPLE!

- The easy way to use types with complex names is to declare simple synonyms for them.
- In a declaration that starts with the keyword typedef, each declarator defines an identifier as a synonym for the specified type.
- The identifier is then called a *typedef name* for that type.
- Except for the keyword typedef, the syntax is exactly the same as for a declaration of an object or function of the specified type.

EXAMPLES

- In the scope of these declarations,
  - ✓UINT is synonymous with unsigned int, and
  - ✓State is synonymous with enum state

```
typedef unsigned int UINT;
typedef enum state {DEAD,ALIVE} State;
```

- The variable ui has the type **unsigned int**, and **enum state** is a pointer to unsigned int.
  - ✓UINT ui = 10;
  - ✓State s= DEAD;

SU LIBRO

- Scope o campo di azione Sezione 5.13
- Sezioni 3.4-3.10, Sezioni 4.1-4.8, Sezione 15.9 (da poi scordare immediatamente)
- Sezione 10.6 (typedef)