# Introduzione al linguaggio C


- Linguaggio
  - Alfabeto
  - Dizionario
  - Sintassi
  - Semantica
- Dall’algoritmo al programma
  - Problema → analisi → formalizzazione → soluzione → programmazione
- Programmazione
  - Rappresentazione dei dati
  - Istruzioni caratteristiche del linguaggio

# Un primo esempio

- Scrivere un programma che permetta di sommare due numeri introdotti da tastiera
  - Acquisire i dati
  - Eseguire la somma
  - Visualizzare il risultato
- Acquisire/visualizzare
  - I/O
- Eseguire la somma
  - algoritmo

```c
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv)
{
    int A,B,C;

    //ACQUISIRE I DATI
    printf("Inserisci A ");
    scanf("%d",&A);
    printf("Inserisci B ");
    scanf("%d",&B);

    //ESEDGUIRE L'OPERAZIONE
    C = A + B;

    //VISUALIZZARE IL RISULTATO
    printf("Il risultato è %d", C);

}

```

# Elementi di base del linguaggio C

```c
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv)
{
    int A,B,C;

    //ACQUISIRE I DATI
    printf("Inserisci A ");
    scanf("%d",&A);
    printf("Inserisci B ");
    scanf("%d",&B);

    //ESEDGUIRE L'OPERAZIONE
    C = A + B;

    //VISUALIZZARE IL RISULTATO
    printf("Il risultato è %d", C);

}
```

- Un programma in C consiste di funzioni e variabili
- Una funzione è costituita da
  - **Variabili**, che memorizzano i valori utilizzati durante il calcolo
  - **Istruzioni**, che indicano le operazioni da svolgere

(The C Programming Language, Second Edition pag 153)

# Elementi di base del linguaggio C

```c
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv)
{
    int A,B,C;

    //ACQUISIRE I DATI
    printf("Inserisci A ");
    scanf("%d",&A);
    printf("Inserisci B ");
    scanf("%d",&B);

    //ESEDGUIRE L'OPERAZIONE
    C = A + B;

    //VISUALIZZARE IL RISULTATO
    printf("Il risultato è %d", C);

}
```

Per il momento, li possiamo considerare come il “contenitore” del nostro programma. In realtà hanno un significato ben preciso

# Elementi di base del linguaggio C

```c
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv)
{
    int A,B,C;

    //ACQUISIRE I DATI
    printf("Inserisci A ");
    scanf("%d",&A);
    printf("Inserisci B ");
    scanf("%d",&B);

    //ESEDGUIRE L'OPERAZIONE
    C = A + B;

    //VISUALIZZARE IL RISULTATO
    printf("Il risultato è %d", C);

}
```

Sono le funzioni mediante le quali interagiamo con la tastiera ed il monitor

# Elementi di base del linguaggio C

```c
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv)
{
    int A,B,C;

    //ACQUISIRE I DATI
    printf("Inserisci A ");
    scanf("%d",&A);
    printf("Inserisci B ");
    scanf("%d",&B);

    //ESEDGUIRE L'OPERAZIONE
    C = A + B;

    //VISUALIZZARE IL RISULTATO
    printf("Il risultato è %d", C);

}
```

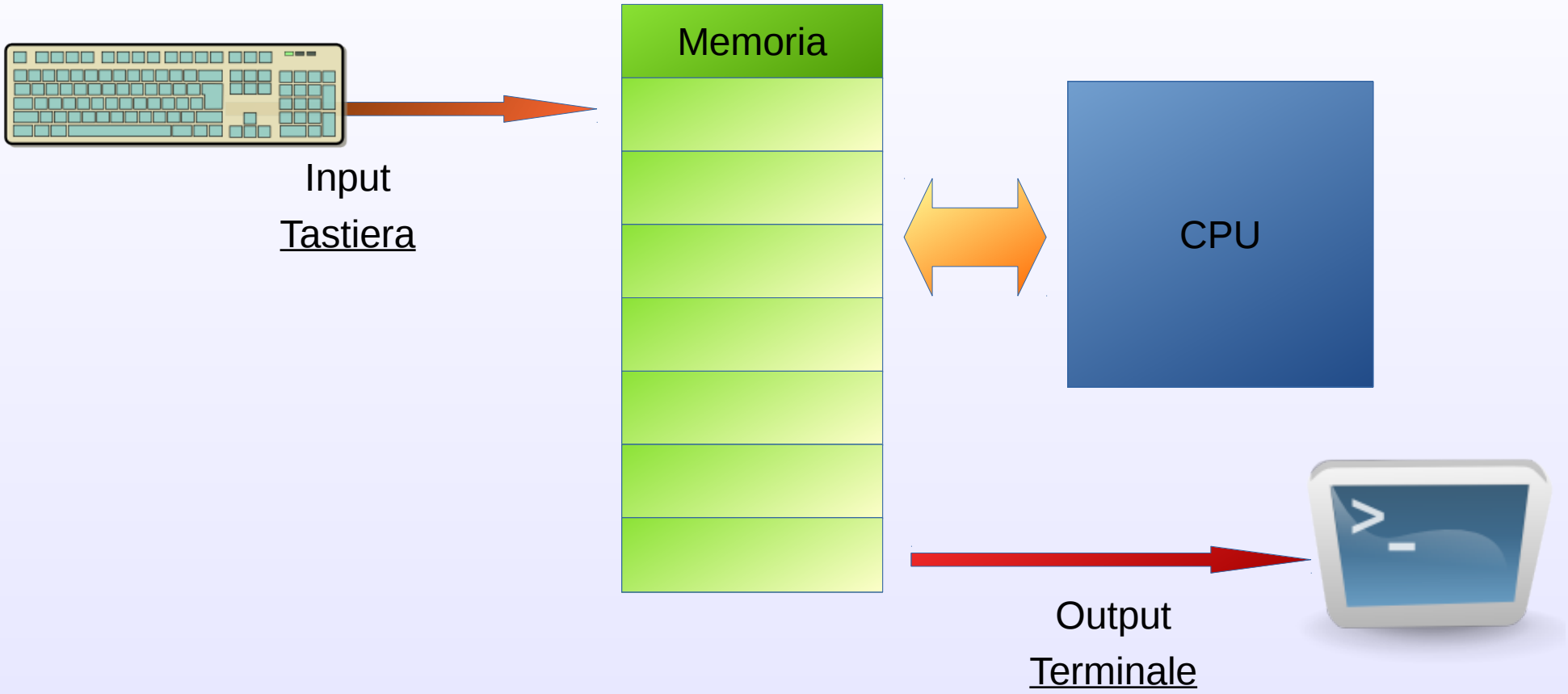
È il modo con cui comunichiamo all'elaborare che tipo di dato gli vogliamo fornire e dove deve memorizzarlo

# Elementi di base del linguaggio C

```c
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char** argv)
{
    int A,B,C;

    //ACQUISIRE I DATI
    printf("Inserisci A ");
    scanf("%d",&A);
    printf("Inserisci B ");
    scanf("%d",&B);

    //ESEDGUIRE L'OPERAZIONE
    C = A + B;

    //VISUALIZZARE IL RISULTATO
    printf("Il risultato è %d", C);

}
```

È il modo con cui comunichiamo all'elaborare cosa deve fare e dove deve memorizzare il risultato

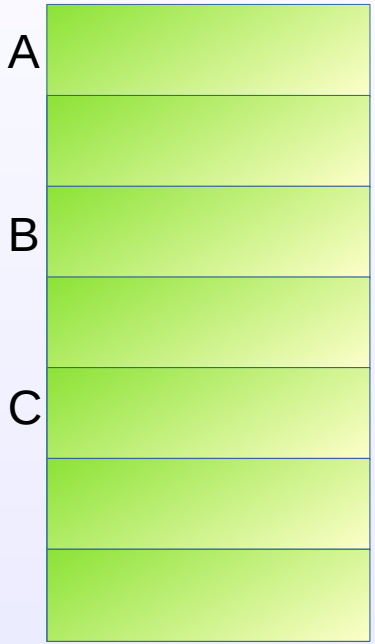
# Modello semplificato del calcolatore



# Variabili e tipi di dati

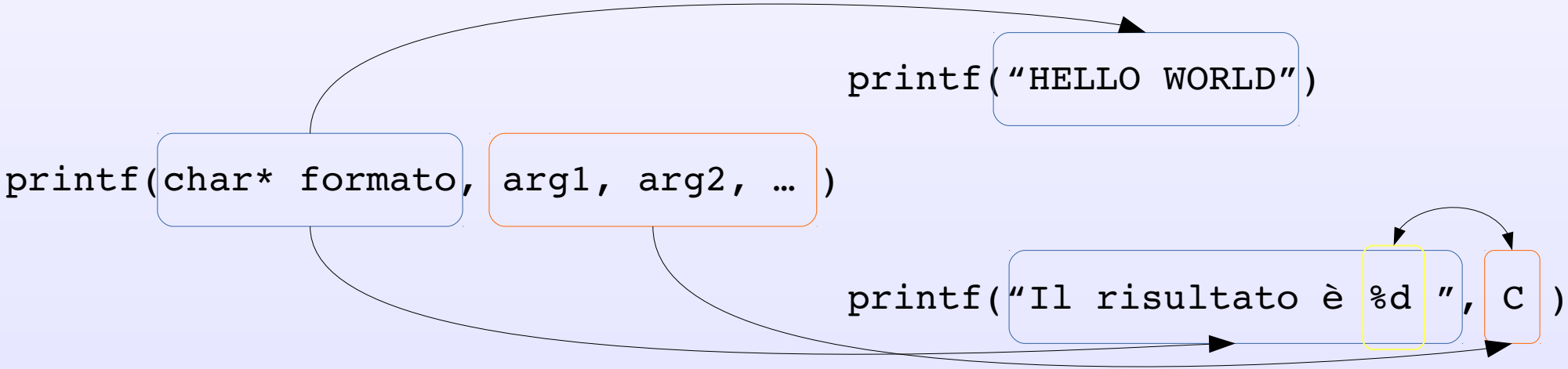
- Tipo di dato
  - I dati sono rappresentati nella memoria fisica del calcolatore come sequenze di 0 e 1. Tale sequenza non ha di per sé un significato. Indicando il tipo di dato, stiamo comunicando al calcolatore che interpretiamo quella sequenza di 0 e 1 come un dato specifico.
- Variabile
  - Porzione della memoria dove viene memorizzato un dato. È identificata dal nome

    **OCCHIO: In C, le variabili devono essere dichiarate prima di essere usate!**



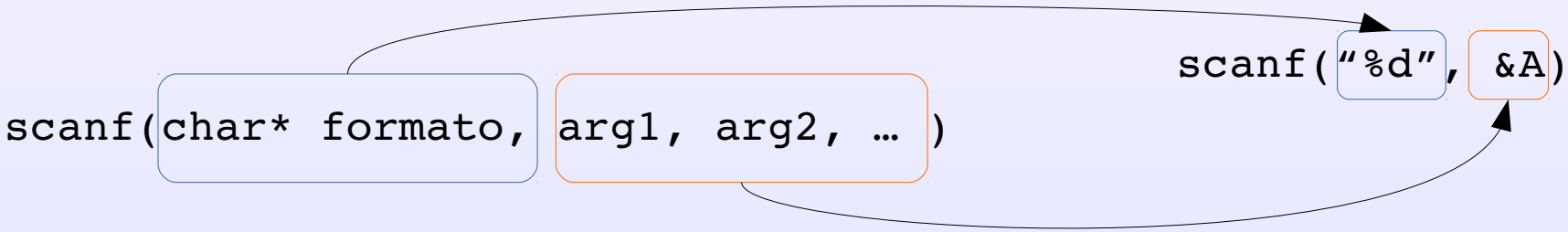
# Come “far parlare” i programmi

- Quando vogliamo che un programma comunichi con l’esterno, bisogna inserire esplicitamente un’istruzione di visualizzazione. Il processore può eseguire calcoli senza “far sapere” all’esterno cosa sta facendo.
- printf converte, formatta e stampa i suoi argomenti sullo standard output. (The C Programming Language, Second Edition pag 153)



# Come “parlare ai” programmi

- Quando vogliamo che un programma acquisisca dati dall’esterno, bisogna inserire esplicitamente un’istruzione di acquisizione. I dati acquisiti dall’esterno saranno memorizzati in una porzione di memoria indicata dal nome della variabile
- scanf legge i caratteri dallo standard input, li interpreta come specificato nel formato e memorizza il risultato attraverso i restanti argomenti. (The C Programming Language, Second Edition pag 153)



# Tipi

- Quando si dichiara una variabile bisogna sempre indicare il tipo e il nome
  - Fino ad ora abbiamo usato
    - `int` per indicare che una variabile è di tipo numerico intero
    - `float` per indicare che una variabile è di tipo numerico reale
- Tutte le variabili devono essere dichiarate prima dell'uso
- Una dichiarazione inizia con il tipo ed è seguita da un elenco di una o più variabili

```
int a, b, c;

int g;

float d, e f;
```

(The C Programming Language, Second Edition pag 35, 36, 40)

# Operatori (I)

- Operatori aritmetici
  - **+, -, *, /**
    - Si applicano sia a `float` che a `int`
    - **Attenzione: l'operatore / fornisce risultati diversi a seconda che gli operandi siano float o int**
  - **%**
    - Operatore resto della divisione intera (detto anche modulo)
    - Si applica a variabili di tipo int
- Operatori relazionali
  - **>, <, >=, <=, ==, !=**
    - Servono per confrontare numeri, secondo le consuete definizioni matematiche
- Operatori logici
  - **&&, ||**
    - operatore di AND logico (restituisce il valore VERO se entrambe le espressioni sono VERE)
    - Operatore OR logico (restituisce VERO se almeno una delle espressioni è vera)

(The C Programming Language, Second Edition pag 35, 36, 40)

# Operatori (II)

- Operatore di assegnazione
  - **=**
    - Esegue il calcolo alla sua destra e lo memorizza nella variabile alla sua sinistra
  - Esempio

```
c = 7;
b = 4;
a = b + c;
```

    - Nella variabile `a` sarà memorizzato il valore `11`
  - Esempio

```
a = 2;
b = 4;
a = a + b;
```

    - Nella variabile `a` sarà memorizzato il valore `6`

(The C Programming Language, Second Edition pag 35, 36, 40)

# Espressioni e frasi

- Un’espressione è una combinazione valida di variabili ed operatori per produrre nuovi valori
- Un’espressione viene detta frase quando è seguita dal punto e virgola

(The C Programming Language, Second Edition pag 35, 36, 40)

# Blocchi di codice

- Un blocco di codice è costituito da tutte le frasi comprese fra due parentesi graffe
- Fino a questo momento i nostri programmi sono stati composti da un unico blocco di codice contenuto nel main

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void main(int argc, char** argv)
5  {
6      int A,B,C;
7
8      //ACQUISIRE I DATI
9      printf("Inserisci A ");
10     scanf("%d",&A);
11     printf("Inserisci B ");
12     scanf("%d",&B);
13
14     //ESEDGUIRE L'OPERAZIONE
15     C = A + B;
16
17     //VISUALIZZARE IL RISULTATO
18     printf("Il risultato è %d", C);
19
20 }
21
```

(The C Programming Language, Second Edition pag 35, 36, 40)

# Il costrutto `if`

- È un “meccanismo” che ci permette di eseguire porzioni di codice in base a determinate condizioni
- Possiamo vederlo come un “interruttore” in grado di attivare o disattivare blocchi di codice durante l’esecuzione di un programma

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, risultato;

    printf("A = ");
    scanf(&a);

    printf("B = ");
    scanf(&b);

    if(b != 0)
    {
        risultato = a / b;
        printf("%d/%d = %d", a, b, risultato);

    }

    if(b == 0)
    {
        printf("Impossibile eseguire una divisione per zero");

    }


}
```

(The C Programming Language, Second Edition pag 35, 36, 40)

# Il costrutto `if`

- Osservazione: all'interno di un blocco di codice posso inserire un altro blocco di codice. E così via...

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, risultato;

    printf("A = ");
    scanf(&a);

    printf("B = ");
    scanf(&b);

    if(b != 0)
    {
        risultato = a / b;
        printf("%d/%d = %d", a, b, risultato);

    }

    if(b == 0)
    {
        printf("Impossibile eseguire una divisione per zero");

    }


}
```

# Il costrutto `if - else`

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, risultato;

    printf("A = ");
    scanf(&a);

    printf("B = ");
    scanf(&b);

    if(b != 0)
    {
        risultato = a / b;
        printf("%d/%d = %d", a, b, risultato);

    }

    if(b == 0)
    {
        printf("Impossibile eseguire una divisione per zero");

    }


}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a, b, risultato;

    printf("A = ");
    scanf(&a);

    printf("B = ");
    scanf(&b);

    if(b != 0)
    {
        risultato = a / b;
        printf("%d/%d = %d", a, b, risultato);

    }
    else
    {
        printf("Impossibile eseguire una divisione per zero");

    }


}
```

- È equivalente a due costrutti if `if` in cui le condizioni del secondo sono complementari a quelle del primo
- Aiuta nella lettura del codice perché permette di individuare con più facilità la logica del programma

# Il concetto di ciclo

- Possiamo pensare ad un ciclo come ad un blocco di istruzioni che devono essere ripetute più volte.

  Consideriamo il seguente codice

```c
int main()
{
    printf("Conto alla rovescia...\n");

    int N = 5;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;
}
```

# Il concetto di ciclo

- Le istruzioni `printf` e `N = N – 1` possono essere logicamente raggruppate e considerate come un blocco di istruzioni che si ripete

```
int main()
{
    printf("Conto alla rovescia...\n");
    int N = 5;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;
}
```

# Il concetto di ciclo

- Abbiamo visto che l'esecuzione del codice è strettamente sequenziale. Informalmente possiamo dire che, con i costrutti visti fino ad ora, l'esecuzione del codice può "andare solo in avanti". Ciò vale anche nel caso degli `if`, in cui possiamo saltare l'esecuzione di un blocco di codice e passare al successivo.
- Sarebbe comodo, in casi come questo, avere a disposizione un comando che permetta di "tornare indietro" e ripetere alcune istruzioni senza la necessità di doverle riscrivere *esplicitamente* un *certo numero* di volte

# Il concetto di ciclo

- Consideriamo ora un'altra situazione, in cui il numero di volte che si vuole ripetere un'istruzione non è noto a priori

```
int main()
{
    int N;

    printf("Conto alla rovescia personalizzato: inserisci il numero da cui iniziare\n");
    scanf("%d", &N);

    //QUANTE VOLTE DOVREI RIPETERE LE ISTRUZIONI
//    N = N - 1;
//    printf("%d", N);

    //?????????
}
```

# Il concetto di ciclo

- Nel momento in cui scriviamo il programma (compile time) non abbiamo alcuna informazione sui dati di input e quindi non sappiamo quante volte ripetere il blocco di codice.
  - Un caso estremo di soluzione sarebbe quella di scrivere un `if` per ogni possibile valore di N, con il risultato che il nostro codice avrebbe un numero enorme di istruzioni (*praticamente infinito*, perché?)
- Un’altra situazione interessante è quella in cui un codice deve essere eseguito in maniera indefinita. Pensiamo ad esempio ai server web o agli algoritmi di controllo degli impianti industriali o di un robot.
  - Dovremmo scrivere infinite righe di codice, contravvenendo quindi alla definizione di algoritmo (*sequenza finita di istruzioni elementari...*)

# Il concetto di ciclo

- Per risolvere i problemi di cui abbiamo parlato, c'è bisogno di introdurre un nuovo costrutto del linguaggio che mi consenta, dato un programma di "tornare indietro" e ripetere istruzioni già eseguite precedentemente
- Il C mette a disposizione un'istruzione particolare che permette di "saltare" ad un punto specifico del programma, anche se questo si trova in una posizione *precedente* all'istruzione correntemente in esecuzione.
- Con questo tipo di istruzione possiamo "alterare" il normale svolgimento del programma, "tornare indietro" quando necessario e di conseguenza eseguire più volte un blocco di istruzione.

# L'istruzione goto

- L'istruzione che permette di indicare un punto esatto del codice dal quale far ripartire l'esecuzione è
  - `goto <ETICHETTA>`
- Possiamo etichettare un blocco di codice con un nome a piacere e, utilizzando `goto`, indicare in qualunque punto del programma di ritornare a quel blocco invece di proseguire con l'istruzione successiva

```
int main()
{
BLOCCO :
goto blocco;
}
```

# L'istruzione goto

- L'effetto pratico è quello di ripetere in continuazione tutte le istruzioni che sono comprese fra l'etichetta e l'istruzione goto
- Si realizza quindi un'*esecuzione ciclica* di un blocco di istruzioni
- Utilizzato in combinazione con `if`, è però possibile regolare il numero di volte che queste istruzioni vengono eseguite!

```
int main()
{
BLOCCO :
goto blocco;
}
```

# L'istruzione goto

```c
int main()
{
    printf("Conto alla rovescia...\n");

    int N = 5;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;

    printf("%d", N);

    N = N - 1;
}
```

```c
int main()
{
    printf("Conto alla rovescia...\n");
    int N = 5;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;

    printf("%d", N);
    N = N - 1;
}
```

```c
int main()
{
    printf("Conto alla rovescia...\n");
    int N = 10;

blocco:
    {
        N = N - 1;
        printf("%d", N);
    }
    if(N!=0) goto blocco;
    return 0;
}
```

- Il blocco di codice indicato con l'etichetta `blocco` (che, ricordiamo, è un nome arbitrario) viene eseguito e successivamente viene verificata la condizione `N != 0`.
- Finché N assume un valore diverso da 0, il `goto` forza il programma a ritornare al punto etichettato con `blocco`.
- Quando la condizione dell'`if` non è più verificata, l'istruzione `goto` non viene più eseguita e il programma prosegue con l'istruzione successiva.

# Osservazioni

- L’istruzione `goto` è stata introdotta per poter dare ad un programma la possibilità di riprendere l’esecuzione da un punto qualsiasi del codice e quindi di alterare la “sequenzialità” del programma stesso
- Realizzare un ciclo significa quindi utilizzare l’istruzione `goto` in combinazione con un `if` che, al verificarsi o meno di una condizione, permette di ritornare su un blocco di istruzioni precedenti
- Nella pratica, tuttavia, i cicli sono realizzati mediante costrutti che “mascherano” la presenza del `goto`, rendendo i programmi più leggibili e meno soggetti ad errori.

# do-while

```c
int main()
{
    int i = 10;
    do
    {
        printf("%d\n", i);
        i = i - 1;
    } while (i >= 0);
}
```

```c
int main()
{
    int i = 10;

    BLOCCO:
    {
        printf("%d\n", i);
        i = i - 1;
    } if (i >= 0) goto BLOCCO;
}
```

- Il do-while è immediatamente riconducibile all'uso del `goto` associato a `if` per il controllo della condizione di ripetizione
- È importante notare che le istruzioni del blocco sono eseguite almeno una volta

# while

```c
int main()
{
    int i = 10;

    BLOCCO:
        if(i>=0)
        {
            {
                printf("%d\n", i);
                i = i - 1;
            }
            goto BLOCCO;
        }
}
```

```c
int main()
{
    int i = 10;
    while (i >= 0)
    {
        printf("%d\n", i);
        i = i - 1;
    };
}
```

- A differenza del `do-while`, l'utilizzo di `while` esegue la verifica della condizione all'inizio dell'esecuzione.
- In questo caso le istruzioni del blocco potrebbero non essere eseguite

# for

```c
int main()
{
    int i = 10;

    BLOCCO:
        if(i>=0)
        {
            {
                printf("%d\n", i);

            }
            i = i - 1;
            goto BLOCCO;
        }
}
```

```c
int main()
{

    for (i = 10; i >= 0; i = i - 1)
    {
        printf("%d\n", i);
    }
}
```

- l'esecuzione di un ciclo mediante il costrutto `for` avviene in 4 fasi
  - Vengono eseguite le istruzioni di inizializzazione UNA SOLA VOLTA
  - Viene controllata la condizione
  - Viene eseguito il blocco
  - Vengono eseguite le istruzioni di fine blocco SEMPRE
    - **Si ripete dal passo giallo**

# Funzioni

- Una funzione non è altro che un **raggruppamento di una serie di istruzioni al quale è stato assegnato un nome**. Sebbene il termine "funzione" derivi dalla matematica, le funzioni C non sempre somigliano a funzioni matematiche.
- Nel C, una funzione non deve necessariamente avere degli argomenti e nemmeno deve restituire un valore.
- Le funzioni sono i blocchi che costituiscono i programmi C. Ogni funzione è essenzialmente un **piccolo programma con le sue dichiarazioni e le sue istruzioni**.
- Usando le funzioni possiamo **suddividere un programma in pezzi più piccoli** che sono più facili da scrivere e modificare (sia per noi che per gli altri).
- Le funzioni ci permettono di evitare la duplicazione del codice che viene usato più di una volta. Le funzioni, inoltre, sono riutilizzabili: in un programma possiamo usare una funzione che originariamente faceva parte di un programma diverso.
- I nostri programmi finora erano costituiti dalla sola funzione `main`. Vedremo come scrivere funzioni diverse dal `main` e impareremo nuovi concetti a riguardo del main stesso.

**(Programmazione in C,Kim N. King, APOGEO)**

# Definizione di una funzione

- Guardiamo alla forma generale di definizione di una funzione:

```
tipo_di_ritorno nome_funzione (lista_dei_parametri)
```

- Il tipo-restituito di una funzione è appunto il tipo del valore che viene restituito dalla funzione stessa.
- Specificare che il tipo restituito è `void` indica che la funzione non restituisce alcun valore
- Dopo il nome della funzione viene messo un elenco di parametri. Ogni parametro viene preceduto da uno specificatore che indica il suo tipo, mentre i diversi parametri sono separati da virgole.

Nota: per ogni parametro deve essere specificato il tipo separatamente, anche quando diversi parametri sono dello stesso tipo.

- Il corpo di una funzione può includere sia dichiarazioni sia istruzioni.
- Le variabili dichiarate nel corpo di una funzione appartengono esclusivamente a
- quella funzione, non possono essere esaminate o modificate da altre funzioni.

**(Programmazione in C,Kim N. King, APOGEO)**

# Chiamata a funzione

- Una chiamata a funzione è costituita dal nome della funzione seguito da un elenco di argomenti racchiusi tra parentesi:

```
media(x, y)
stampa_linea(i)
visualizza_nome()
```

- Una chiamata a una funzione `void` è sempre seguita da un punto e virgola che la trasforma in un'istruzione:

```
stampa_linea(N,i);
```

- Una chiamata a una funzione non-`void`, invece, produce un valore che può essere memorizzato in una variabile, analizzato, stampato e utilizzato in altri modi

```
int a;
a = conta_cifre(N);
printf(“%d”, a);
```

**(Programmazione in C,Kim N. King, APOGEO)**

# Dichiarazione di funzione (prototipi)

- La dichiarazione di una funzione deve essere consistente con la sua definizione.
- Un prototipo fornisce una descrizione completa su come chiamare una funzione: quanti argomenti fornire, di quale tipo debbano essere e quale sarà il tipo restituito. Per inciso, il prototipo di una funzione non è obbligato a specificare il nome dei parametri, è sufficiente che sia presente il loro tipo:

```
double media (double, double);
```

- In ogni caso di solito non è bene omettere i nomi dei parametri, sia perché questi aiutano a documentare lo scopo di ogni parametro, sia perché ricordano al programma tore l'ordine nel quale questi devono comparire quando la funzione viene chiamata. Tuttavia ci sono delle ragioni legittime per omettere il nome dei parametri e alcuni programmatori preferiscono comportarsi in questo modo.

**(Programmazione in C,Kim N. King, APOGEO)**

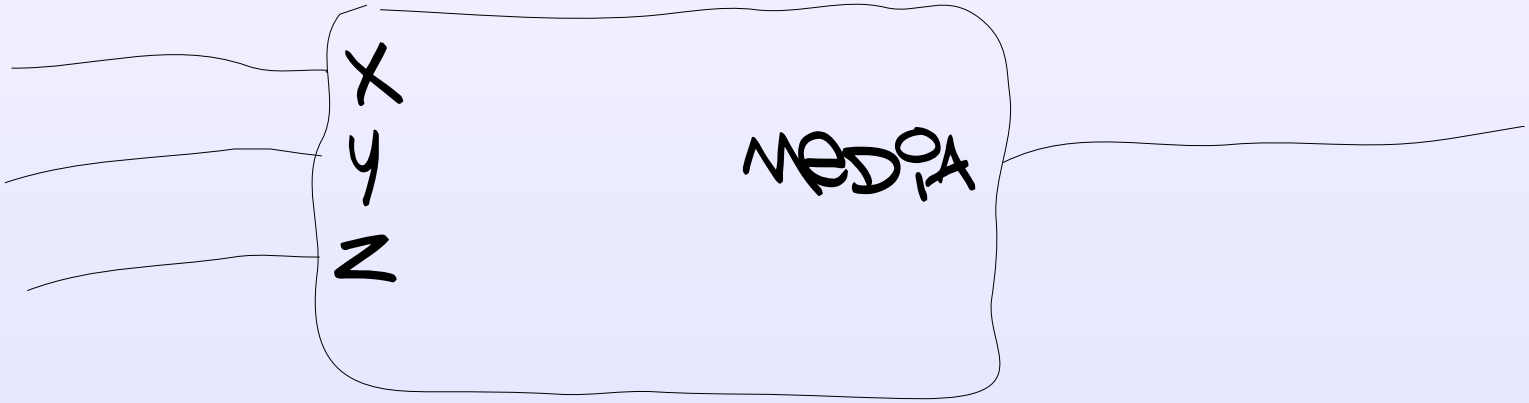
# Argomenti e parametri

- Concentriamoci sulla differenza tra parametri e argomenti. I parametri compaiono nelle definizioni delle funzioni e sono dei nomi che rappresentano i valori che dovranno essere forniti alla funzione quando questa verrà chiamata.
- Gli argomenti sono delle espressioni che compaiono nelle chiamate alle funzioni.
- Nel C gli argomenti vengono passati per valore: quando una funzione viene chiamata, ogni argomento viene calcolato e il suo valore viene assegnato al parametro corrispondente.
- Dato che i parametri contengono una copia del valore degli argomenti, ogni modifica apportata ai primi durante l'esecuzione della funzione non avrà alcun effetto sui secondi.
- Agli effetti pratici ogni parametro si comporta come una variabile che è stata inizializzata con il valore dell'argomento corrispondente.

**(Programmazione in C,Kim N. King, APOGEO)**

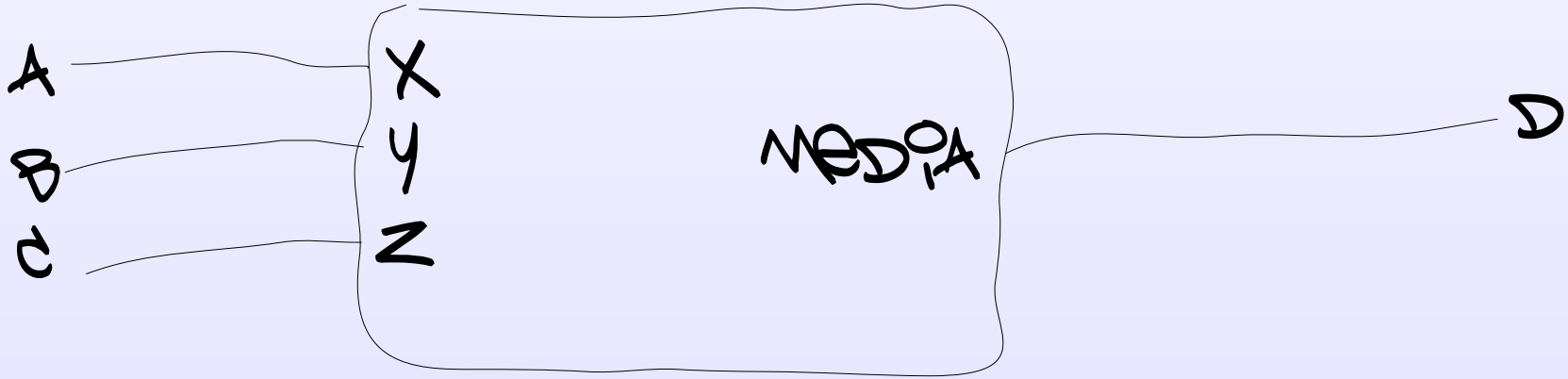
# Argomenti e parametri

```
float media_di_tre_numeri(float x, float y, float z)
{
    float somma, media, N = 3;
    somma = x + y + z;
    media = somma / N;
    return media;
}
```



# Argomenti e parametri

```
void main()
{
    float a = 7, b = 15, c = 2, d;
    d = media_fra_tre_numeri(a,b,c);
}
```



# Gestione e organizzazione del codice

- Sebbene alcuni programmi C siano sufficientemente brevi da essere posti in un singolo file, la maggior parte non lo sono. Programmi che sono costituiti da più di un file sono la regola e non l'eccezione. In questo capitolo vedremo che un tipico programma è costituito da diversi file sorgente e tipicamente anche da alcuni file header.
- I file sorgente contengono le definizioni delle funzioni e delle variabili esterne.
- I file header contengono le informazioni che devono essere condivise tra i file sorgente.
- Fino a questo momento abbiamo assunto che un programma C sia costituito da un singolo file. In realtà un programma può essere diviso su un qualsiasi numero di file sorgente.
- Per convenzione i file sorgente hanno estensione .c. Ogni file sorgente contiene parte del programma, principalmente definizioni di funzioni e variabili. Un file sorgente deve contenere una funzione chiamata `main` che fa da punto di partenza per il programma.

**(Programmazione in C,Kim N. King, APOGEO)**

# Gestione e organizzazione del codice

Suddividere un programma in più file sorgente presenta vantaggi significativi:

- raggruppare funzioni e variabili collegate all'interno di un singolo file aiuta a rendere chiara la struttura del programma;
- ogni file sorgente può essere compilato separatamente (un grosso risparmio tempo se il programma è grande e deve essere modificato di frequente, una cosa piuttosto comune durante lo sviluppo);
- le funzioni diventano facilmente riutilizzabili in altri programmi quando vengono raggruppate in file sorgente separati.

Quando suddividiamo un programma in diversi file sorgente si presentano dei problemi: come fa una funzione di un file a chiamare una funzione che è stata defitita in un altro file?

La risposta risiede nella direttiva #include, che rende possibile la condivisione delle informazioni tra i file sorgente.

**(Programmazione in C,Kim N. King, APOGEO)**

# Gestione e organizzazione del codice

- La direttiva `#include` dice al processore di aprire una specifico file e di inserire il suo contenuto all'interno del file corrente. Quindi, se vogliamo che diversi file sorgente abbiano accesso alla stessa informazione, mettiamo questa informazione in un file e poi utilizziamo la direttiva `#include` per inserire il contenuto di questo all'interno di ogni file sorgente.
- I file che vengono inclusi in questo modo vengono chiamati file header.
- Per convenzione i file header hanno estensione . h.
- Nota: lo standard C utilizza il termine "file sorgente" per fare riferimento a tutti i file scritti dal programmatore, sia i file .c che quelli .h. Noi utilizzeremo il termine "file sorgente" solo per riferirci ai file .c.

**(Programmazione in C,Kim N. King, APOGEO)**

# Gestione e organizzazione del codice

```
Dichiarazione funzione1()
Dichiarazione funzione2()
Dichiarazione funzione3()
...
```

**file1.h**

```
#include “file1.h”
main()
{
}
```

**programma_principale.c**

```
#include “file1.h”
Definizione Funzione1()
Definizione Funzione2()
Definizione Funzione3()
...
```

**file1.c**