INFORMATICA A

A.A. 2018-19

Laboratorio n°4

Dott. Michele Zanella

Ing. Gian Enrico Conti

HEAP LAB

Puntatori

- I **puntatori** sono delle variabili che contengono l'indirizzo di memoria di un'altra variabile

```
int* nome_puntatore;
```

- "*" è chiamato **operatore di deferenziazione** e restituisce il contenuto dell'oggetto puntato dal puntatore; mentre l'operatore "&" restituisce l'indirizzo della variabile.

```
int var=0, var2, ind;

int* puntatore;

puntatore = &var;  //puntatore punta a var
var2 = *puntatore; //var2 == var
ind = puntatore;   //ind contiene l'indirizzo di var
*puntatore = 5     //var == 5
```

Operazioni sui puntatori

- È possibile effettuare operazioni sui puntatori.
  - Spostamento in avanti: *p* + *i*, sposta il puntatore di *i* posizioni avanti
  - Spostamento indietro: *p* – *i*, sposta il puntatore di *i* posizioni indietro
- **Un puntatore esiste sempre in funzione del tipo di oggetto puntato.**
  - Un puntatore ad *int* ha un blocco di memoria di 4 byte, a *char* di 1 byte, ecc…
  - Se sommo al puntatore un intero o utilizzo l'operatore **++,** il puntatore si sposterà in avanti di tanti byte quanti ne prevede il tipo di variabile puntata. Ad esempio: *p* + *i* equivale a *posizione* + *(i*dimensione tipo puntato)*
- Se ad una funzione passo dei puntatori eventuali operazioni su questi parametri saranno effettuate sulle variabili originali!
- È possibile agire sugli array come se si stesse agendo su un puntatore poiché entrambi sono blocchi contigui di memoria. **Attenzione:** devo comunque assegnare la prima cella dell'array ad un puntatore.
- Posso utilizzare i puntatori per puntare a strutture, in questo caso per accedere ai campi della *struct* utilizzo l'operatore **"->"**

Puntatori e tipi

- Puntatori ed array

```
char stringa[20];
char* pointer;

pointer= &stringa[0]; //pointer punta a stringa[0]
pointer=stringa;      //Uguale alla riga sopra
pointer++;            //Ora pointer punta a stringa[1]
```

- Nel dettaglio

```
int a[20]; // a è puntatore alla prima cella dell'array => a[0]
           // a[1] = *(a+1)
           // &[1] = (a+1)

scanf("%d", (a+1))     // &a[1] = &*(a+1)=(a+1)

for(int i=0; i<20; i++){
    printf("%d", a[i]); // Le due printf stampano la stessa cosa
    printf("%d", i[a]); // a[i] = *(a+i) = *(i+a) = i[a]!!!
}
```

Puntatori e tipi

- Puntatori e struct

```
struct punto {
    int x;
    int y;
} puntoA;

struct punto* pointer;

pointer = &puntoA; //pointer punta a
puntoA

pointer->x = 6;    //Assegno il campo x
pointer->y = 7;    //Assegno il campo y
```

Funzioni

- Hanno una "firma" (*signature*):
  - Tipo dell'output
  - Nome univoco
  - Parametri di input

```
int my_function(int a)
```

- Generalmente terminano con la restituzione, al *chiamante*, del risultato della computazione, attraverso l'istruzione **`return`**.
  - Esistono anche funzioni che non restituiscono nulla: quelle di tipo **`void`**

```
int somma(int a, int b){
    int s = a + b;
    return s;
}
```

- Vengono utilizzate per riutilizzare parti di codice e per migliorarne la leggibilità.
- **In C non è possibile utilizzare una funzione prima che sia dichiarata!**

Funzioni e puntatori

- Posso passare i parametri ad una funzione in due modi:
  - Per *copia*: copio il valore del parametro passato nella variabile locale (parametro)

```
int somma(int a, int b){
    int s = a + b;
    return s;
}
```

  - Per *referenza*: passo il puntatore (indirizzo) del parametro passato. **Se modifico il puntatore deferenziato modifico anche il valore del parametro all'esterno della funzione!!**

```
void somma(int a, int b, int* ris){
    *ris = a + b; //modifico direttamente il valore di ris
}
```

Header files

- Vengono utilizzati per spostare le definizioni delle variabili globali e dei tipi in un file esterno:
  - Possibilità di riutilizzare strutture dati in più file (es., librerie)
  - Migliorano la leggibilità del codice (meno righe nei sorgenti .c)
- Vengono salvati con un formato specifico: `my_header.h`
- Vengono inclusi nei file .c tramite la `#include "my_header.h"`

- Per il progetto creiamo un file `define_and_typedefs.h`

- Negli header files posso anche inserire dei **prototipi** delle funzioni, ossia una loro dichiarazione fatta senza specificare il corpo della funzione stessa.

Approccio per la soluzione dei problemi

1) Analisi dei requisiti, definizione del modello e dei tipi necessari
2) Stesura del Flow chart (o degli step) ad alto livello della soluzione
3) Definizione delle funzioni necessarie e organizzazione del codice
4) Implementazione e compilazione passo-passo (non aspetto alla fine di tutto!)
5) Testing

Esercizio 4.1: Somma vettori con puntatori

Scrivere una funzione che calcoli la somma di due vettori passati come puntatori e salvi il risultato in una variabile passata come puntatore.

```
void sommavp(int* v1, int* v2, int* ris);
```

Hint:

Considerate la dimensione dei vettori come prefissata

Esercizio 4.2: Gara di danza

Scrivere un programma che gestisca i punteggi di una gara di danza. calcoli il punteggio dell'esecuzione per ogni atleta di una gara di danza.

Requisiti:

- Ogni atleta è caratterizzato da un codice alfanumerico e dal suo punteggio finale
- Possibilità di inserire il codice degli atleti da input
- Possibilità di inserire i punteggi dei singoli giudici da input e calcolare il risultato finale
- Possibilità di stampare tutti gli atleti con i relativi punteggi finali

Hints:

- Utilizzare le funzioni della `stdio`
- Definire numero di giudici e atleti
- Definire i tipi e le funzioni necessarie

Esercizio 4.3: Mastermind

Scrivere un programma per giocare a **Mastermind**.

http://www.webgamesonline.com/mastermind/

http://www.webgamesonline.com/mastermind/rules.php

Regole di base:

- 2 giocatori (CPU e utente)
- CPU sceglie una sequenza di colori non nota all'utente
- L'utente deve indovinare l'esatta sequenza (posizione e colore)
- Ad ogni round l'utente inserisce una possibile sequenza
- CPU risponde in questo modo:
    - Per ogni posizione e colore indovinato nella sequenza, mostra un colore nero a destra (=black peg)
    - Per ogni colore indovinato (ma posizione sbagliata) nella sequenza, mostra una colore bianco (=white peg)

Esercizio 4.3: Mastermind (cont'd)

Altre regole:

- L'utente definisce la lunghezza della sequenza da indovinare
- Nella sequenza i colori si possono ripetere
- L'utente definisce il numero massimo di round in cui indovinare la sequenza
- L'utente vince se indovina l'esatta sequenza entro il numero massimo di round, altrimenti vince CPU.

Esercizio 4.3: Mastermind (cont'd)

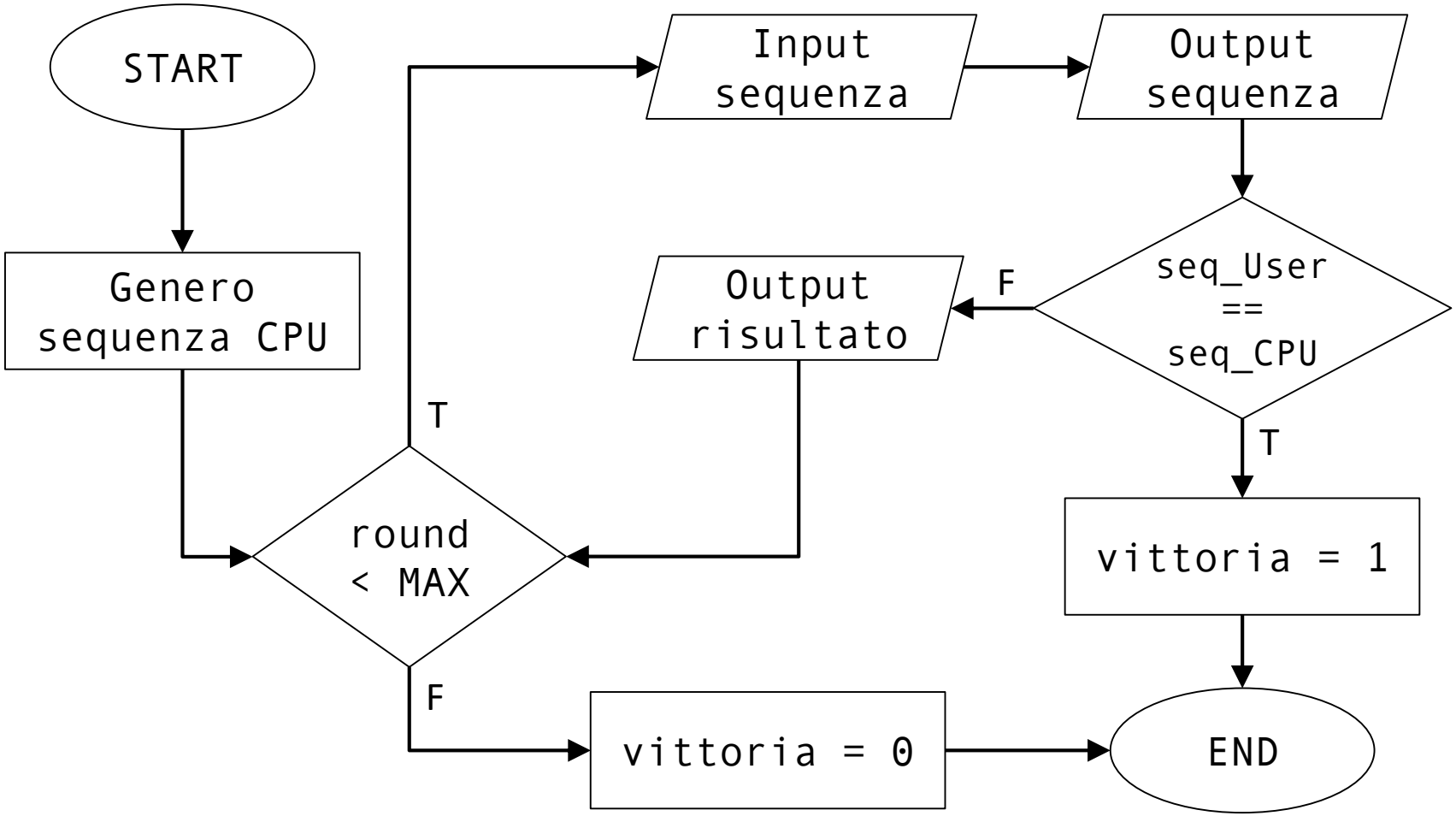
Requisiti:

- L'utente deve definire le impostazioni di gioco (lunghezza sequenza, numero massimo di round)
- CPU deve generare una sequenza casuale di colori
- L'utente deve poter inserire una possibile sequenza ad ogni round
- CPU deve mostrare la qualità del risultato del round (colori bianchi e/o neri)

Hints:

- Utilizzare solo testo: i colori vengono scritti come caratteri testuali (e.g., w = "pallino bianco")
- Definire un carattere per ogni colore per l'inserimento (e.g., 'y' = "giallo")
- 6 colori: rosso, giallo, verde, arancione, nero e bianco

Esercizio 4.3: Mastermind (Flowchart generale)



Esercizio 4.3: Mastermind (cont'd)

Altri hints:

- Utilizzare la funzione `rand()` della librearia `stdlib.h` per generare il codice CPU casuale.
- Definire ed implementare le seguenti funzioni (oltre a quelle per stampare risultato e codice):

```
void genera_codice(char* secret_code);
```

```
void indovina_codice(char* guess_code);
```

```
int controlla_codice(char* secret_code,
                     char* guess_code,
                     int* white_pegs,
                     int* black_pegs);
```

```
void stampa_codice(char* code);
```