Linguaggio C: approfondimento

Reti di Calcolatori A.A. 2023/24

Prof.ssa Chiara Petrioli - Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Michele Mastrogiovanni - Dipartimento di Ingegneria Informatica

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Controllo dell'esecuzione

Finora:

- Variabili
- Tipi e `unsigned` specifier
- Operatori comuni
- Standard I/O

In particolare, tutti i programmi visti finora però sono stati caratterizzati dalla loro esecuzione sequenziale.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

1. Controllo dell’esecuzione

Controllo dell'esecuzione

Esecuzione sequenziale

```
1 int age = 0;
2 char name[50];
3 printf("Enter age and name:\n");
4 fgets(name, 50, stdin);
5 sscanf(name, "%d", &age);
6 fgets(name, 50, stdin);
```

Le istruzioni vengono eseguite linearmente, una dopo l'altra, a partire dalla prima e terminando con l'ultima.

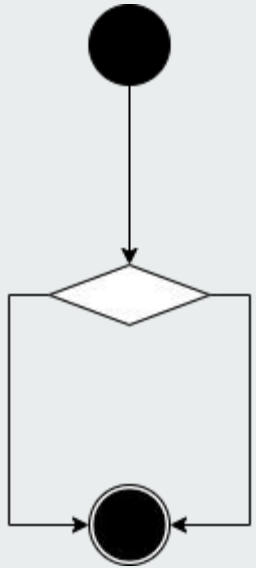
Spesso invece è necessario effettuare scelte, in modo da eseguire logica diversa a seconda dei valori riscontrati durante l'esecuzione.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione di selezione `if`

L'istruzione `if` permette l'esecuzione *condizionale* di blocchi di codice.

```
1 if(conditional_expression) {
2   // Blocco codice 1
3 }
4 else {
5   // Blocco codice 2
6 }
```



- *conditional_expression* è l'espressione la cui valutazione determina quale blocco di codice viene eseguito
- *Blocco codice 1:* eseguito solo se la valutazione è `true`
- *Blocco codice 2:* eseguito solo se la valutazione è `false`

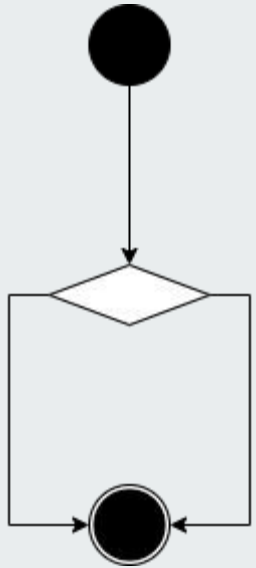
Problema: `bool` non è un tipo primitivo di C → valori `true` / `false` nell'istruzione `if`?

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione di selezione `if`

L'istruzione `if` permette l'esecuzione *condizionale* di blocchi di codice.

```
1 if(conditional_expression) {
2   // Blocco codice 1
3 }
4 else {
5   // Blocco codice 2
6 }
```



- *conditional_expression* è l'espressione la cui valutazione determina quale blocco di codice viene eseguito
- *Blocco codice 1:* eseguito solo se la valutazione è ~~`true`~~ → `!= 0`
- *Blocco codice 2:* eseguito solo se la valutazione è ~~`false`~~ → `== 0`

Problema: In C ogni espressione restituisce un valore (ad es. `a = b`)

```
1 int a = 0, b = 5, c = 0;
2 if((b = 0) || (a % 2 == 0)) c++;
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione di selezione `if`

Espressioni condizionali:

- Semplici

```
1 if(sizeof(name) > 0)
```

- Composte

```
1 if((b = 0) || (a % 2 == 0))
```

Evitare facilmente errori di valutazione:

- Invertire l'ordine del confronto

```
1 if((0 = b) || (0 == a % 2))
```

Operatori con ordine di valutazione specificato:

- Operatori `&&` e `||` → Da sinistra a destra, solo se necessario
- Operatore ternario `a ? b : c`
- Operatore `a,b` → Valuta `a` e lo scarta, poi valuta `b`

Tutti gli altri operatori non hanno un ordine di valutazione ben definito.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione di selezione `if`

L'istruzione `else` può essere combinata con altre istruzioni `if`.

- *Blocco codice 1*:
  Eseguito se *cond_expr1* viene valutata diversa da 0
- *Blocco codice 2*:
  Eseguito se *cond_expr1* viene valutata 0 e *cond_expr2* diversa da 0
- *Blocco codice 3*:
  Eseguito se *cond_expr1* e *cond_expr2* vengono valutate 0 e *cond_expr3* diversa da 0
- *Blocco codice 4*:
  Eseguito se tutte vengono valutate 0

```
1 if(cond_expr1) {
2   // Blocco codice 1
3 }
4 else if(cond_expr2) {
5   // Blocco codice 2
6 }
7 else if(cond_expr3) {
8   // Blocco codice 3
9 }
10 else {
11   // Blocco codice 4
12 }
```

In alcuni casi è possibile omettere le parentesi graffe intorno ai blocchi di codice: blocco di codice a singola istruzione, se un blocco di codice è a sua volta una coppia di istruzioni if-else, ecc.

Tuttavia è sconsigliato: sia per motivi di leggibilità che di esecuzione inaspettata dovuta a refactoring o debug.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione di selezione `switch`

Il costrutto `switch` permette la scelta tra più di due rami di esecuzione, ognuno identificato da un'apposita istruzione `case` o `default`.

- *expr* deve essere un'espressione con valore intero
  Ammessi: `char`, `signed`/`unsigned integer`, `enum`
- Ogni `case` deve essere associato ad una constant expression
  `value1`, `value2`, `value3` → non sono quindi variabili bensì literal!
- All'interno di ogni blocco codice deve essere presente un'istruzione `break`, altrimenti procederà la valutazione di altri `case`
- È ammessa al massimo 1 instruzione `default`

```
1  switch(expr){
2    case value1:
3    {
4      // Blocco codice 1
5      break;
6    }
7
8    case value2:
9    {
10     // Blocco codice 2
11   }
12   case value3:
13   {
14     // Blocco codice 3
15     break;
16   }
17
18   default:
19   {
20     // Blocco codice 4
21   }
22 }
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione di selezione `switch`

Esecuzione

1. *expr* viene valutata con valore `v_expr`
2. Se `v_expr` è uguale a *value1*, blocco codice 1 viene eseguito
3. Se `v_expr` è uguale a *value2*, blocco codice 2 viene eseguito; `break` mancante: farà eseguire anche il blocco codice 3
4. Se `v_expr` è uguale a *value3*, blocco codice 3 viene eseguito
5. Esauriti i case, il blocco codice 4 viene eseguito se nessun case ha eguagliato v_expr

L'esecuzione di più blocchi di codice dovuti all'assenza di `break` può essere intenzionale, ed è chiamato fallthrough.

```
switch(expr){
  case value1:
  {
    // Blocco codice 1
    break;
  }

  case value2:
  {
    // Blocco codice 2
  }
  case value3:
  {
    // Blocco codice 3
    break;
  }

  default:
  {
    // Blocco codice 4
  }
}
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Confronto: `if` vs `switch`

`if`

- Adatto a condizioni non strutturate
- Numero di opzioni relativamente limitato

```
1 if (n % 2 == 0){
2   cout << "The number is even" << endl;
3 }
4 else {
5   cout << "The number is odd" << endl;
6 }
```

`switch`

- Adatto a condizioni strutturate
- Numero di opzioni anche elevato

```
1 switch (n % 2) {
2   case 0:
3     cout << "The number is even" << endl;
4     break;
5   default:
6     cout << "The number is odd" << endl;
7     break;
8 }
```

```
1 if (n % 2 == 0){
2   cout << "Divisible by 2" << endl;
3 }
4 else if (n % 3 == 0){
5   cout << "Divisible by 3" << endl;
6 }
7 else {
8   cout << "Not divisible by 2 or 3" << endl;
9 }
```

In questo caso lo `switch` non sarebbe appropriato, siccome la condizione non è strutturata

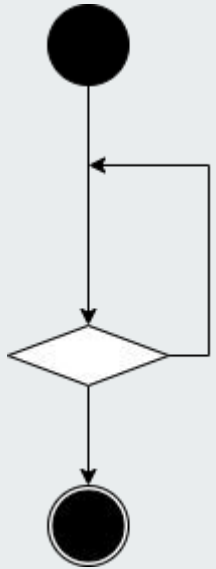
Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Ripetizione di istruzioni

In un programma potrà essere necessario ripetere un blocco di istruzioni diverse volte. Diverse istruzioni iterative o cicli sono perciò forniti dal linguaggio C.

- `while`
- `do while`
- `for`

Dal punto di vista funzionale sono pressoché equivalenti, permettendo la scelta di quale dei tre costrutti sia il più appropriato solo sotto l'aspetto stilistico.



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Ciclo `while`

Il ciclo `while` ripete l'esecuzione di un blocco di codice finché l'espressione condizionale è diversa da `0`.

```
1 while(cond_expr){
2   // Blocco codice
3 }
```

- Affinché l'esecuzione del blocco avvenga, *cond_expr* deve essere valutata diversa da `0` già all'inizio
- L'esecuzione del blocco viene ripetuta se, dopo l'esecuzione del blocco stesso, la condizione viene nuovamente valutata diversa da `0`
- Il corpo del ciclo deve gestire eventuali variabili usate in *cond_expr*

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Ciclo `do while`

Il ciclo `do while` esegue il blocco di codice almeno una volta, e l'esecuzione viene ripetuta finché l'espressione condizionale è diversa da `0`.

```
1 do{
2   // Blocco codice
3 } while(cond_expr);
```

- La prima esecuzione del blocco avviene prima che qualsiasi valutazione di *cond_expr* sia effettuata
- L'esecuzione del blocco viene ripetuta se, dopo l'esecuzione del blocco stesso, la condizione viene valutata diversa da `0`
- Il corpo del ciclo deve gestire eventuali variabili usate in *cond_expr*

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Ciclo `for`

Il ciclo `for` esegue il blocco di codice finché l'espressione condizionale è diversa da `0`.
Aggiuntivamente,

```
1 for(init_expr; cond_expr; iter_expr){
2   // Blocco codice
3 }
```

- Eventuali variabili usate in *cond_expr* possono essere inizializzate tramite *init_expr*
- *init_expr* viene eseguita esattamente una volta, prima dell'inizio del ciclo
- Affinché l'esecuzione del blocco avvenga, *cond_expr* deve essere valutata diversa da `0` già all'inizio
- L'esecuzione del blocco viene ripetuta se, dopo l'esecuzione del blocco stesso e l'esecuzione di ***iter_expr***, la condizione viene nuovamente valutata diversa da `0`
- Eventuali variabili usate in *cond_expr* possono essere aggiornate tramite *iter_expr*

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzioni di jump

Meccanismi di controllo dell'esecuzione finora:

- Istruzioni di selezione
- Istruzioni iterative (cicli)

} Il flusso del programma è soggetto alla valutazione di espressioni condizionali affinché il controllo dell'esecuzione cambi

In C esistono anche meccanismi di controllo dell'esecuzione che operano incondizionatamente:

- `break`
  Termina un ciclo indipendentemente dalla valutazione dell'espressione condizionale
- `continue`
  Ignora il resto delle istruzioni nello stesso blocco di codice, passando all'iterazione successiva
- `return`
  Termina l'esecuzione della funzione corrente, restituendo il controllo alla funzione chiamante
- `goto`
  Trasferisce il controllo dell'esecuzione in modo arbitrario, tramite un'etichetta (label)

} Generalmente invocati nel corpo di un'istruzione `if`, all'interno di un ciclo

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzioni di jump vs i diversi cicli

while

```c
int n=0;
while(n<5){
  if(n%2==0)
    continue;
  printf("%d\n", n);
  n++;
}
```

Infinito!

do while

```c
int n=0;
do{
  if(n%2==0)
    continue;
  printf("%d\n", n);
  n++;
} while(n<5);
```

Infinito!

for

```c
for(int n=0; n<5; n++){
  if(n%2==0)
    continue;
  printf("%d\n", n);
}
```

Stampa 1 e 3, soprattutto termina!

L'istruzione `continue` è introdotta negli stessi punti del corpo di tutti e 3 i cicli, tuttavia:

- `while` e `do while` → variabile `n` (usata nella condizione) non viene aggiornata per via di `continue`
- `for` → Dopo l'esecuzione di `continue`, l'espressione di iterazione viene comunque eseguita, aggiornando `n`

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione `goto`

- Uso dovrebbe essere limitato e relegato a quando non ci sia altra scelta
- Se necessario, va usato con molta attenzione

La sintassi è molto semplice: il controllo dell'esecuzione passa esattamente al punto stabilito tramite un'etichetta

Sintassi `goto`

```
1 goto printLbl;
2 int n = 42;
3 printLbl: printf("%d\n", n);
```

Istruzione `printf` è uno statement etichettato.
Per definire un'etichetta si può anche usare solo il null statement `;` *Ad es.:* `printLbl: ;`

Etichetta `printLbl`

Cosa succede?

1. Non compila
2. Errore durante l'esecuzione
3. Legale per il C, compilazione ed esecuzione normale

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Istruzione `goto`

- Uso dovrebbe essere limitato e relegato a quando non ci sia altra scelta
- Se necessario, va usato con cautela

La sintassi è molto semplice: il controllo dell'esecuzione passa esattamente al punto stabilito tramite un'etichetta

Sintassi `goto`

```
1 goto printLbl;
2 int n = 42;
3 printLbl: printf("%d\n", n);
```

Istruzione `printf` è uno statement etichettato.
Per definire un'etichetta si può anche usare solo il null statement `;` *Ad es.:* `printLbl: ;`

Etichetta `printLbl`

Cosa succede?

1. ~~Non compila~~
2. ~~Errore durante l'esecuzione~~
3. Legale per il C, compilazione ed esecuzione *normale*

Variabile `n` non viene inizializzata. L'esecuzione è equivalente a:

```
1 int n;
2 printf("%d\n", n);
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

2.
Esercizi

Esercizi: controllo dell'esecuzione

1. Scrivere un programma che legga un intero positivo `n` dalla tastiera e faccia la somma dei primi `n` interi usando un ciclo `for`.

   Esempio: 5 → 1 + 2 + 3 + 4 + 5 = 15
2. Riprodurre Esercizio 1 utilizzando un ciclo `while`.
3. Riprodurre Esercizio 1 sostituendo il ciclo `for` con una o più istruzioni `goto`.

   Hint: implementare l'istruzione di inizializzazione, l'espressione condizionale, e l'istruzione di iterazione esplicitamente
4. Scrivere un programma che chieda all'utente un intero positivo e restituisca 0 se è un numero di Harshad o, alternativamente, 1 se non lo è.

   Numero di Harshad: un numero intero che è divisibile per la somma delle sue cifre. Esempio: 30 → è un numero di Harshad → 0

   Esempio: 11 → non è un numero di Harshad → 1

   *Hint: utilizzare un ciclo **while**; l'ultima cifra di un qualsiasi numero intero in base 10 è il resto della divisione intera per 10*

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

3.
Array

Salvare in memoria più valori dello stesso tipo

Variabili:

- Contengono 1 solo valore di un determinato tipo
- Identificatore unico per ogni variabile

In caso di un elevato numero di valori da contenere in memoria, può diventare scomodo

```
1 int a, b, c;
2 a = 0;
3 b = 0;
4 c = 7;
```

Array

- Collezione di dati omogenei
- Locazioni di memoria contigue
- Dimensione fissa
- Accesso costante ai valori di un array

```
1 int arr[3];
2 arr[0] = 0;
3 arr[1] = 0;
4 arr[2] = 7;
```

Dichiarazione di un array di valori interi con dimensione 3

Accesso al primo elemento dell'array, posizione 0

Accesso all'ultimo elemento dell'array, posizione 2

Nei casi di inizializzazione degli array, è possibile lasciare al compilatore il compito di determinare automaticamente la dimensione.

```
1 int arr[] = {0,0,7};
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Array e puntatori

La variabile array `arr` è di tipo puntatore ad intero `int*`

```
1 int arr[5] = {5,4,3,2,1};
2 printf("%ld\n", arr);
3 printf("%ld\n", *arr);
4 printf("%ld\n", arr[0]);
```

Entrambe queste istruzioni stampano il primo elemento di `arr`

Gli array vengono interpretati come puntatori al primo elemento, in particolare quando sono passati come argomenti a funzioni.

```
1 int arr[5] = {5,4,3,2,1};
2 int v1 = arr[3];
3 int v2 = *(arr + 3);
```

Equivalenti

Non bisogna però confondersi tra array e array di puntatori.

```
1 int arr[3], *arrPtr[5];
```

Array di interi, tipo `int*`

Array di puntatori ad interi, tipo `int**`

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Array e puntatori

Aritmetica dei puntatori

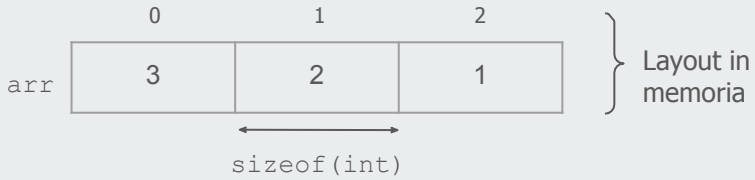
Alcuni operatori artimetici possono essere utilizzati direttamente sui puntatori, e nel caso degli array può risultare molto comodo. Bisogna considerare che il valore dei puntatori è un indirizzo di memoria, e quindi le operazioni aritmetiche assumono qualche significato in più.

Operatori: `++`, `--`, `+`, `-`

```
1 int arr[] = {3,2,1};
2 int *p = arr;
3 printf("%d\n", *(p+1));
4 p = &arr[2];
5 printf("%d\n", *(p-1));
```

Indirizzo dell'ultimo elemento di `arr`

Indirizzo contenuto in `p` viene incrementato e decrementato di 1 *unità di memoria*: `p` è puntatore ad interi, perciò generalmente 32-bit → 4 byte



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Array multidimensionali

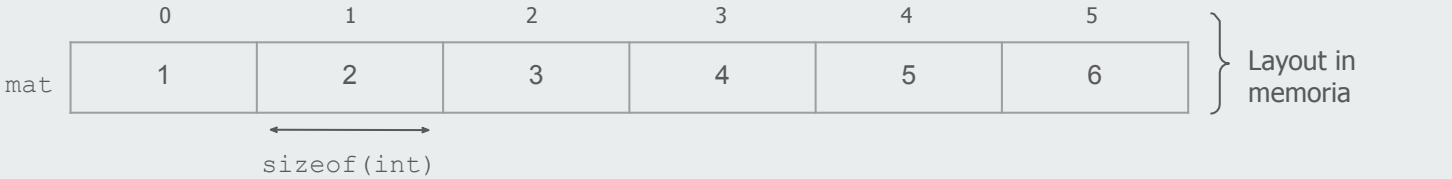
- Array di array
- Formato accesso: riga, colonna, …
- Riscrittura sotto forma di puntatore: puntatore alla prima riga

Array multidimensionale 3x2

```
1 int mat[][2] = {{1,2},
2                 {3,4},
3                 {5,6}};
4 printf("%d\n", mat[2][1]);
```

Equivalente:

```
1 *(*(mat + 2) + 1);
```



Inoltre:

- Il compilatore deve sempre conoscere la dimensione di una riga
- Gli array multidimensionali sono in realtà linearizzati in memoria

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Array multidimensionali

Non solo matrici…

- Array multidimensionali possono avere un numero arbitrario di dimensioni
- L'unica dimensione automaticamente determinata dal compilatore è la prima

```
1 // Dim. 3x2x1
2 int tensor[][2][1] = {{{1},{2}},
3                       {{3},{4}},
4                       {{5},{6}}};
```

Attenzione: l'accesso ad un array fuori i limiti di una qualsiasi dimensione non comporta alcun errore durante la compilazione od esecuzione.

Il risultato è solamente imprevisto; manipolare direttamente gli array perciò dev'essere fatto con cautela.

```
1 printf("%d\n", mat[5][7][22]);
```

Istruzione perfettamente legale dato l'array `mat` di cui sopra

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - Luca Iezzi Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

5.
Funzioni

Funzioni in C

Finora:

- Unica funzione `main`
- Tutto il codice necessariamente nello stesso punto
- Ripetizione di parti di codice per svolgere lo stesso compito in punti diversi

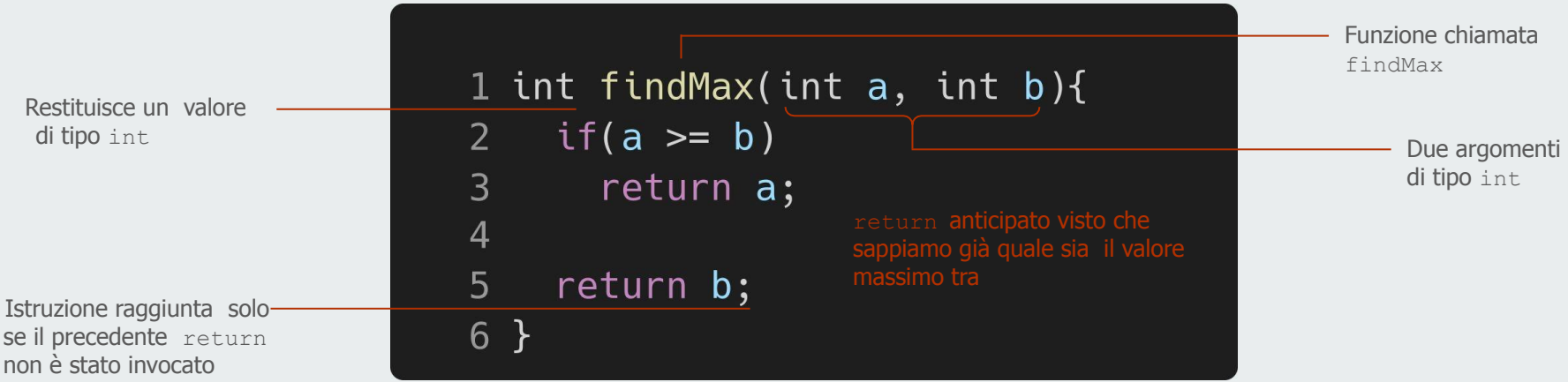
Insostenibile nel caso di programmi complessi: mancanza di organizzazione e difficoltà nell'eseguire debugging e/o aggiornamenti al codice.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Funzioni in C

Funzioni

- Un blocco di codice a cui viene associato un identificatore
- Generalmente svolge un preciso compito
- Può essere chiamata in qualsiasi altro punto del codice (anche da sé stessa!)



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Funzioni in C

Dichiarazione di `findMax`

Definizione di findMax

```c
#include <stdio.h>

int findMax(int a, int b);

int main(){
  printf("%d\n", findMax(42, 89));
  return 0;
}

int findMax(int a, int b){
  if(a >= b)
    return a;
  return b;
}
```

Funzioni devono essere dichiarate o interamente definite prima della funzione `main`

Se una funzione è stata solamente dichiarata, al massimo 1 definizione deve essere fornita

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Funzioni in C

A cosa fare attenzione:

- Passaggio argomenti
  Per default vengono copiati i valori, ogni aggiornamento dei valori non verrà mantenuto all'uscita della funzione; se c'è questa necessità, usare puntatori come parametri
- Allocare memoria all'interno di una funzione e restituirne un puntatore tramite `return`

```
1 int* findMax(int a, int b){
2   int max = (a >= b) ? a : b;
3   return &max;
4 }
```

Segmentation fault
Il puntatore restituito fa riferimento ad una locazione di memoria che non è più allocata. Le variabili locali ad una funzione vengono infatti distrutte all'uscita di quest'ultima. Ciò non avviene con *malloc*…

- Puntatori a funzione

```
1 int (*fnPtr)(int,int) = findMax;
2 printf("%d\n", fnPtr(42, 89));
```

```
1 int (*fnPtrArr[2])(int,int) = {findMax, findMax};
2 printf("%d\n", fnPtrArr[1](42, 89));
```

Array di 2 puntatori a funzione con ritorno `int` e due argomenti `int`

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

6. Esercizi

Esercizi: array

5. Scrivere un programma che chieda all'utente un intero positivo `n > 2`, crei un array di dimensione `n`; scrivere una funzione che riempia l'array con i primi `n` numeri della successione di Fibonacci.

Successione di Fibonacci: considerando F_n l'n-esimo numero, per definizione $F_0 = 0$ ed $F_1 = 1$, mentre un generico $F_n = F_{n-1} + F_{n-2}$ Esempio: 5 → 0, 1, 1, 2, 3

Esempio: 2 → errore

6. Scrivere un programma che chieda all'utente un intero `n` e lo usi per inizializzare il crivello dei numeri fortunati; scrivere una funzione che esegua il crivello fino alla condizione di terminazione.

Crivello dei numeri fortunati Esempio: 15

Fase 1: primi n numeri interi positivi	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Fase 2: eliminazione di ogni secondo numero	1		3		5		7		9		11		13		15
Fasi successive: il prossimo numero sopravvissuto (es. x) viene usato per eliminare ogni x-esimo numero partendo dall'inizio	1		3				7		9				13		15
Termina: non appena nessun numero viene eliminato	1		3				7		9				13		15

…

17

Hint: utilizzare un array per mantenere in memoria i candidati a numeri fortunati; sostituire i numeri eliminati con 0

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Riferimenti

- https://en.cppreference.com/w/c/language/if
- https://en.cppreference.com/w/c/language/switch
- https://en.cppreference.com/w/c/language/constant_expression
- https://en.cppreference.com/w/c/language/while
- https://en.cppreference.com/w/c/language/do
- https://en.cppreference.com/w/c/language/for
- https://en.cppreference.com/w/c/language/break
- https://en.cppreference.com/w/c/language/continue
- https://en.cppreference.com/w/c/language/return
- https://en.cppreference.com/w/c/language/goto
- https://en.wikipedia.org/wiki/Harshad_number
- https://en.cppreference.com/w/c/language/array
- https://www.cs.swarthmore.edu/~richardw/classes/cs31/s18/offsite/pointer.html
- https://en.cppreference.com/w/c/language/functions
- https://en.wikipedia.org/wiki/Lucky_number