
—

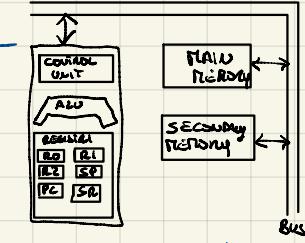


PROGRAMMAZIONE I

BIT > (b)

Binary Information, contiene 2 informazioni: 0 = spento, 1 = acceso

- $2^3 b = 1 \text{ byte (B)}$ capace di rappresentare 256 informazioni diverse
- $2^5 b = 1 \text{ word}$ Definita per ragione di efficienza di calcoli.

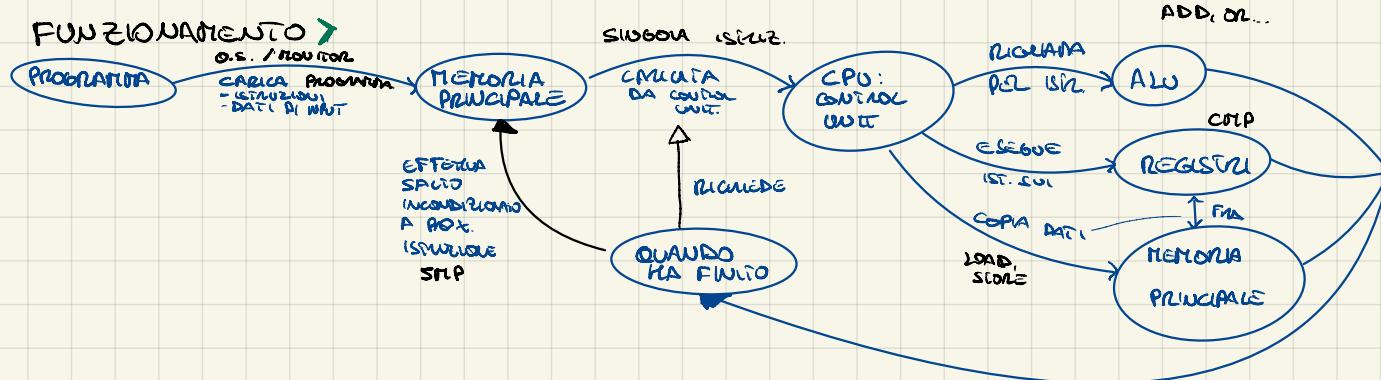


ARCHITETTURA DI VON NEUMANN

L'architettura di Von Neumann è l'architettura moderna utilizzata dai computer. Essa è composta

- **MEMORIA**: La memoria utilizzata dal sistema. Essa è divisa in:
 - MEM. PRINCIPALE: Memoria volatile in cui vengono caricati dati ed istruzioni.
 - MEM. SECONDARIA: Memoria duratura in cui vengono caricati i programmi.
- **BUS DI SISTEMA**: Sistema di comunicazione ad alte velocità tra le memorie principale e quelle secundarie.
- **CPU**: Parte elaborativa dell'architettura. Essa ha uno stato dato da quello dei suoi registri.
 - CONTROL UNIT: Preleva le istruzioni dalla memoria principale e le decodifica una alla volta. In base all'istruzione:
 - Attiverà una componente diversa dell'ALU.
 - Eseguita l'istruzione caricherà la prossima della memoria, oppure ne preleverà un'altra (salto incondizionato).
 - ALU: La Arithmetic Logic Unit esegue semplici istruzioni logico-arithmetiche.
 - REGISTRI: Memoria interna della CPU

FUNZIONAMENTO >



PROG 3

TIPI DI DATI >

- char (1 byte / 8 bit) : contiene il singolo carattere
- short (2 Byte (16 bit)) : contiene interi nel range $-2^{15}, 2^{15}-1$
- int (4 B (32b)) : $\text{--} \quad \text{--} \quad \text{--} \quad \text{--}$
- long (4 B (32b)) : $\text{--} \quad \text{--} \quad \text{--} \quad \text{--}$
- Long long (8 B (64b)) : $\text{--} \quad \text{--}$
- float (4 B, 32b) : contiene num. con virgola 3.4×10^{-38}
- double (8 B, 64b) : $\text{--} \quad \text{--}$ 1.7×10^{-308}

32 bit

$-2^{15}, 2^{15}-1$
 $-2^{31}, 2^{31}-1$
 $-2^{31}, 2^{31}-1$
 $-2^{63}, 2^{63}-1$

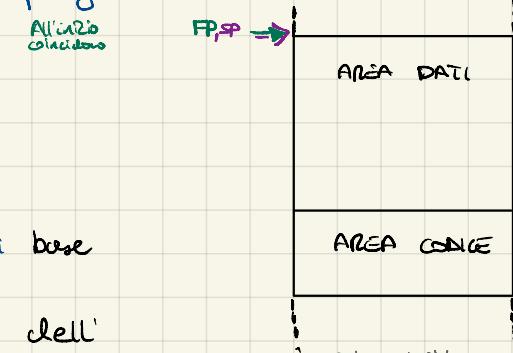
64 bit

$\text{--} \quad \text{--}$
 $\text{--} \quad \text{--}$
 $-2^{63}, 2^{63}-1$
 $\text{--} \quad \text{--}$
 $\text{--} \quad \text{--}$

MODELLO DI MEMORIA >

Quando un programma viene caricato nella memoria principale, viene **allocata** un'area di memoria per il codice e i dati del programma. L'area allocata nello stack è così divisa in

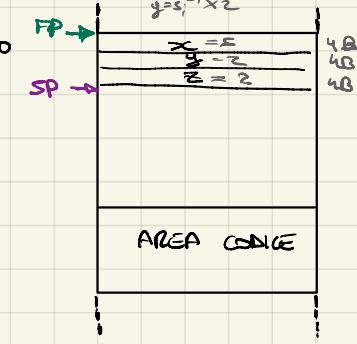
- area codice : di lunghezza fissa
- area dati : di dimensione massima, contenente i dati delle variabili



Inoltre, nel programma vengono impostati 2 puntatori:

- Frame Pointer: Il puntatore contenente l'indirizzo di base a cui allocare le variabili
- Stack Pointer: Il puntatore contenente l'indirizzo dell'ultima variabile (IND stack pointer \leq FP in quanto cara verso il basso)

A seconda del tipo, le variabili occuperanno l'area dati partendo dall'alto



PUNTATORI >

Siccome ogni variabile è assegnata ad un indirizzo nella memoria, è possibile ottenere l'indirizzo di memoria usando l'operatore `&`.

8.a → Indirizzo di memoria della variabile 'a'

È anche possibile definire delle variabili che contengano un indirizzo di memoria ed una variabile. Esse sono puntatori.

int a; int* pA = &a

I puntatori, indipendentemente dal tipo, sono di 4 B.

UTILITY CODICE >

Si utilizzano le librerie:

- `#include <stdbool.h>` per includere il tipo `bool`
- `#include <assert.h>` per poter fare assertions con `assert()`.

PREDICATI >

I predici vengono usati per descrivere lo status del codice ad un certo punto delle sue esecuzione. **3***

PRE / POST CONDIZIONI >

Sono dei predici che descrivono lo status prima/dopo una data istruzione.

Vi sono anche le pre/post condizioni del programma che determinano degli stati previsti veri nei rispettivi step.

```

// P1 PRE-COD. DI I1
// (C1) PI
a=x+b // I1
// P2 -> PRE COD. DI I2, POST DI I1
y=a+b // I2
// P3 -> POST COD. DI I2
  
```

```

#include <assert.h>
int main();
int a=3;
// C1 -> stato d'utopie a==3
assert(a==3);
int b=12; // stato d'utopie C1 c b==12
assert(a==3 && b==12);
int y=a+b;
// C2 stato da 1,2 e y==15
assert(a==3 && b==12 && y==15)
  
```

PRO 6 1 • RAGIONAMENTO BACKWARD

Per ragionamento Backward si intende la metodologia utilizzata per risolvere le condizioni di un programma partendo dal **baso**.

In particolare:

- Si parte dalla post-condizione del programma.
- Si tiene nota delle assegnazioni della condizione.
- Si sale verso l'istruzione precedente.
- Nella precondizione dell'istruzione precedente, sostituire la parte sx degli elementi.
- Ripetere dal 2° punto.

ESEMPIO >

```
// PRE a==x && b=y
{
    // (1) ???
    tmp = a;
    // (2) ???
    a = b;
    // (3) ???
    b = tmp;
    // (4) stato finale
    // POST a=y && b=x
```

```
// PRE a==x && b=y ↗ MATCH!
{
    // (1) ???
    b=y && a=x
    tmp = a;
    // (2) ???
    b=y && tmp=x
    a = b;
    // (3) ???
    a=y && tmp=x
    b = tmp;
    // (4) stato finale a=y && b=x
    // POST a=y && b=x
```

ESEMPI ESECZI DI ESERCIZI >

TIPO 1: BACKWARD + REFACTOR >

Ex. 5.9.2

Modificare il codice come segue:

- nei punti // (2), // (3) e // (4) specificare assert che servono per stabilire la weakest pre-condition dell'intero algoritmo al punto // (1);
- sostituire // (X) con un'assegnazione che assicura la verità della weakest pre-condition al punto // (1);
- aggiungere quanto necessario ad ottenere un sorgente C che non produce errori di alcun tipo.

Prog. 1 (C) A.A. 23/24

STRATEGIA

- Applicare LdF in backward.
- Includere semplificazioni.

SOLUZIONE

```
#include <assert.h>
void main(void)
{
    int x, y, z;
    y = 2 * x; // (X) sostituito
    assert(y == 6); // weakest pre-condition
    assert((y == 6) && (2 * y == 12)); // semplificato
    assert((z + y / 2 == 6) && (2 * (z * y / 2 + 1) == 14));
    y = 2 * z;
    assert((y / 2 == 6) && (2 * (y / 2 + 1) == 14));
    assert((y / 2 + 1 == 6) && (2 * (y / 2 + 1) == 14));
    z = y / 2 + 1;
    assert((z == 3) && (2 * z == 14)); // semplificato
    assert((x + 1 == 8) && (2 * (x + 1) == 16)); // semplificato
    x = x + 1;
    assert((x == 8) && (2 * x == 16)); // semplificato
    y = 2 * x;
    assert(x == 8 && y == 16); // lo stato finale è anche la post-condition
}
```

TIPO 2: STATO INIZIALE + REFACTOR >

Ex. 5.9.2

Modificare il codice come segue:

- nei punti // (2), // (3) e // (4) specificare assert che servono per stabilire la weakest pre-condition dell'intero algoritmo al punto // (1);
- sostituire // (X) con un'assegnazione che assicura la verità della weakest pre-condition al punto // (1);
- aggiungere quanto necessario ad ottenere un sorgente C che non produce errori di alcun tipo.

STRATEGIA

- Applicare LdF backward.
- Includere semplificazioni.

SOLUZIONE

```
#include <assert.h>
#include <iostream>
void main()
{
    // La precondizione principale "L=1" era identificata
    // come la più forte: 1 <= L <= 8
    // Prendiamo immediatamente y = 2
    y = 2;
    // Aggiungiamo la weakest pre-condition
    // sia vero, risulta che nello stato iniziale sia vero anche y == 8.
    // Dobbiamo quindi impostare da 8 a 1.
    // Imposta da 8 a 1:
    assert(8 <= y <= 1); // semplificato è la weakest pre-condition
    assert(8 <= y <= 1);
    int x = 1;
    assert(x == 1);
    int z = 1;
    assert(z == 1);
    int v = 8;
    assert(v == 8);
    assert(v == 8); // post-condition
}
```

FUNZIONI >

Le funzioni sono pezzi di codice ri-utilizzabili. Esse sono formate da:

- Tipo di ritorno, (void se non torna niente)
- Parametri formali, i parametri che la funzione si aspetta in ingresso.
 - o Reversibili secondo il libro
- Essi possono essere "passati" in 2 modi diversi
 - Per valore, il parametro attuale passato non subisce modifiche in quanto copia.
 - Per riferimento, il parametro attuale passato subisce modifiche nel corpo della funzione
- Istruzioni, della funzione (incluso return se non void)

Quando sono richiamate, le funzioni accettano da 0..N Parametri Attuali / Argomenti.

PROTOTIPO >

I prototipi di funzione consentono di definire la firma della funzione senza doverla implementare. Essa viene poi implementata dopo il Main.

Ciò viene fatto per ragioni di comodità di lettura.

INFO: Il prototipo può omettere il nome dei parametri

REGOLE D'INGAGGIO >

NON mettere più di 1 return ovunque.

FUNZIONAMENTO NELLO STACK >

Ogni chiamata ad una funzione, main compreso, comporta la creazione di un Frame / Stack di Allocazione (Essendo una pila i nuovi frame si impilano sopra quelli già aperti).

Inoltre, ogni Frame / Stack di allocazione avrà le seguenti celle:

- RET: Sta per Return Line e contiene l'indirizzo alle prossime istruzioni da eseguire dopo il return della funzione.
- RETU: Sta per Return Value e contiene l'eventuale valore ritornato dalla funzione.
- 2 celle per ogni parametro della funzione chiamata.
- celle per le variabili dello stack.

Una volta terminate le istruzioni all'interno della funzione, lo Stack Pointer punterà al retu, deallocando il resto.

Eventualmente, il valore di retu verrà copiato nell'istruzione che richiede l'esecuzione tramite funzione. SP sarà incrementato, liberando anche retu.

Infine, il FP viene ripristinato al chiamante e il controllore di programma torna al valore di retl, eliminando il Frame della funzione chiamata.

void func(int a); ← Prototipo

int main() {

...
 ↗ Attuali / Argomenti

func(a); ← Invocazione

Definizione

Corpo

└ void func(int a);

...
 ↗ Parametri / P. formal

Chiamata a funzione square()

```
1 void main(void) {
2     int a = 10, asq = 0;
3     asq = square (a);
4     printf("%d squared:\n", a, asq);
5 }
6 int square (int x) {
7     int x2 = x * x;
8     return x2;
9 }
```

PC →

FP ret 7
SP asq 0
main

32 bit, 4 bytes

ma ha allocato a ed asq nello stack e sta per chiamare square()

Chiamata a funzione square()

```
1 void main(void) {
2     int a = 10, asq = 0;
3     asq = square (a);
4     printf("%d squared:\n", a, asq);
5 }
6 int square (int x) {
7     int x2 = x * x;
8     return x2;
9 }
```

PC →

ret 7
asq 0
main

32 bit, 4 bytes

L'indirizzo di ritorno, la linea 4, è salvato nello stack per primo come rett

Chiamata a funzione square()

```
1 Prima del ritorno dalla funzione chiamata square() al main(), la situazione è quella a destra
2 Una domanda d'esame sarà rappresentare lo stato della memoria della macchina prima del ritorno della funzione chiamata
```

ret 7
asq 100
main

32 bit, 4 bytes

Chiamata a funzione square()

```
1 void main(void) {
2     int a = 10, asq = 0;
3     asq = square (a);
4     printf("%d squared:\n", a, asq);
5 }
6 int square (int x) {
7     int x2 = x * x;
8     return x2;
9 }
```

PC →

ret 7
asq 100
main

32 bit, 4 bytes

L'operatore assegnazione copia il valore in cima allo stack di rett e in asq

Chiamata a funzione square()

```
1 void main(void) {
2     int a = 10, asq = 0;
3     asq = square (a);
4     printf("%d squared:\n", a, asq);
5 }
6 int square (int x) {
7     int x2 = x * x;
8     return x2;
9 }
```

PC →

ret 7
asq 100
main

32 bit, 4 bytes

La prossima linea eseguita sarà la 4

FUNZIONI: PASSARE VALORE PER RIFERIMENTO >

È possibile utilizzare un workaround per poter passare i valori per riferimento.

Per poter fare ciò, si passa la variabile puntatore al posto delle variabili normali. In questo modo, il record di allocazione allocerà dello spazio con valore l'indirizzo della memoria precedente.

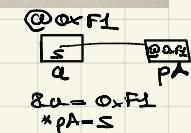
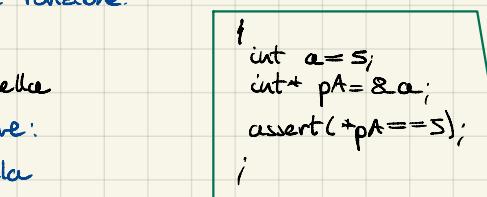
In questo modo, sarà possibile sfruttare il valore nella funzione.

DIFERENZIAMENTO >

Dato un puntatore, è possibile ottenere il valore della variabile puntata con l'operatore `*`. In particolare:

- QUANDO STA A SX: agisce come alias della variabile referenziata

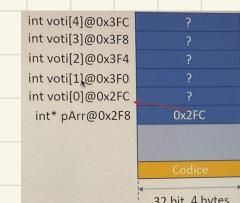
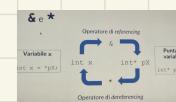
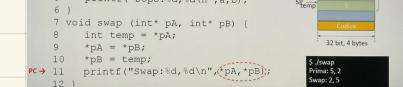
- QUANDO STA A DX: agisce come valore referenziato, ovvero valore dell'alias



Funzione swap() con puntatori

```

1 int main(void) {
2     int a = 5, b = 2;
3     printf("Prima:%d,%d\n",a,b);
4     swap (&a,&b);
5     printf("Dopo:%d,%d\n",a,b);
6 }
7 void swap (int* pA, int* pB) {
8     int temp = *pA;
9     *pA = *pB;
10    *pB = temp;
11    printf("Swap:%d,%d\n",*pA,*pB);
12 }
    
```



Indirizzo minore
nello stack sono
rappresentati dal
più grande al
più piccolo.

PROG 3 • 2023-10-25

ARRAY > `int arr[5]; int arr [] = {1,2,3,4,5}; int arr [] = {0};`

Gli array sono delle strutture dati in grado di contenere più elementi dello stesso tipo.

ALLOCAZIONE NELLO STACK >

Quando viene definito l'array, vengono riservati nello stack tante "righe" quanti sono gli elementi nell'array.

In particolare, il primo elemento con indice 0 sarà quello con indirizzo minore

nello stack sono
rappresentati dal
più grande al
più piccolo.

`SIZE_T > for(size_t i=0; i<n; i++)`

Durante le iterazioni degli elementi all'interno degli array, può essere dichiarato il tipo `size_t` come indice.

`size_t` è un tipo `int unsigned`, ovvero un intero senza segno ottimizzato.

ARRAY COME PARAMETRI > Regole d'ingaggio per esame

Il passaggio degli array NB: Non mettere prototipi nelle funzioni deve richiedere alla funzione chiamata:

- Il puntatore al 1° elemento dell'array: (basta passare il singolo array)
- La dimensione dell'array

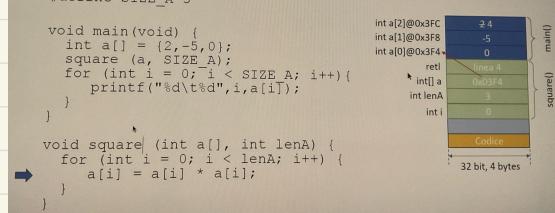
Passaggio di array come parametri

```

#define SIZE_A 3

void main(void) {
    int a[] = {2,-5,0};
    square (a, SIZE_A);
    for (int i = 0; i < SIZE_A; i++) {
        printf("%d\t%d",i,a[i]);
    }
}

void square (int a[], int lenA) {
    for (int i = 0; i < lenA; i++) {
        a[i] = a[i] * a[i];
    }
}
    
```



RICERCHE NEGLI ARRAY >

È possibile ricerca degli elementi in maniera negli array. È buona norma **ritornare -1** se non è stato trovato

VARIABLE LENGTH ARRAYS > int dim=s; int a[dim];

È possibile definire array di lunghezza dinamica semplicemente specificandone la dimensione a runtime.

STRINGHE IN C >

Le stringhe in C sono array formate dai caratteri specificati ai quali viene aggiunto il carattere di terminazione '\0' che rappresenta la fine della stringa.

USO IN OUTPUT > `printf("%s", "ciao");`

È possibile stampare le stringhe usando %s in printf();

USO IN INPUT >

È possibile

- definire le stringhe come array o literal: arr = {'c', 'i', 'o', ' ', '\0'} V cstr = "ciao";
- richiederle in input con %s con scanf();: scanf("%s", &s)
recorderà l'ellis...

BINARY SEARCH >

La binary search o "Ricerca Binaria" è un algoritmo che consente di effettuare ricerche ottimizzate con una Time complexity di O(log n)

Esso richiede che l'array sia ordinato.

ALG. NEL DETTAGLIO >

Nel dettaglio, dato un array ordinato ed un elemento x da cercare:

- Si confronta l'elemento a metà dell'array con x:
 - Se $x < m$: Bisognerà ripetere la ricerca con gli elementi dell'array da 0...m
 - Se $x > m$: Bisognerà ripetere la ricerca con gli elementi dell'array da m...dim
 - Se $x = m$: Elemento trovato

ARRAY CON PUNTOATORI >

Dato un array, è possibile accedervi dai suoi elementi in vari modi

- Tramite EI >

le EI si occupano in automatico di trovare l'indirizzo di base della variabile e di sommato alla moltiplicazione dell'i-esimo elem. per la sua dimensione

- Tramite & > int curr=&a[0]; int *a=curr; a=a+1*sizeof(int) V a=&a[1] | int*a = arr+i; i-esimo elem

Dato un array, è possibile accedere al puntatore al primo elemento. Da esso, è possibile accedere al successivo aggiungendo la dimensione del tipo dell'array

```
Accesso agli elementi dell'array

#define A_LEN 3

int main(void) {
    int a[A_LEN] = {17, 9, 80};
    int *plast = a + A_LEN - 1;
    for (int* pa = a; pa < plast; pa++) {
        printf("%d\n", *pa);
    }
}
```

MATRICI >

Le matrici in C sono array bi-dimensionali ovvero degli array che contengono al loro interno altri array.

In particolare:

- La prima parte contiene le righe.
- La seconda parte contiene le colonne.

INIALIZZAZIONE >

Come gli array, è possibile assegnare i valori alle matrice

- con [i]: dei singoli elementi si assegnano. $\text{mat}[1][0] = 2; \dots$
- con [i][j]: definita la dimensione delle matrice,
le grappe più interne rappresentano i valori delle colonne alla data riga.
Le grappe esterne contengono le righe.

In caso non tutti i valori siano stati riempiti, essi verranno pre-valorizzati con 0.

PASSAGGIO COGLE PARAMETRI > (int mat[1][1][1][1])
obb.

Nelle matrici, quando vengono dichiarati i parametri formali, essi devono dichiarare tutte le dimensioni (la prima è facoltativa)

$$A = r_1 \cdot c_1, B = r_2 \cdot c_2 \quad m=1$$

PRODOTTO DI MATRICI > $(A+B)_{ij} = \sum_{k=0}^m A_{ik} * B_{kj}$

Dette 2 matrici $A r_1 \cdot c_1, B r_2 \cdot c_2$ il loro prodotto è dato dalla matrice $C r_1 \cdot c_2$ che contiene per ogni colonna la sommatoria da $0 \dots c_2 - 1$ in cui si moltiplica:

- L'elemento di A con colonna K e riga l'-esima
- L'elemento di B con colonna j e riga k-esima.

SOMMA TRA MATRICI >

Date 2 matrici A, B con lo stesso num di elementi, la loro somma è data da
la somma tra $A_{ij} + B_{ij}$

c1	c2	c3	c4
r1 [0][0] [0][1] [0][2] [0][3]			
r2 [1][0] [1][1] [1][2] [1][3]			
r3 [2][0] [2][1] [2][2] [2][3]			

PLUS: DIF C++ / H++

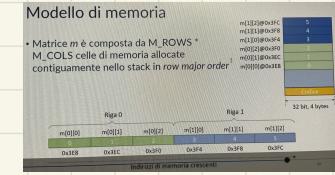
- C++ = assegna e incarta
- H++ = incrementa e assegna

MATRICI >

NELLA MEMORIA >

Nel modello di memoria, le matrici sono rappresentate seguendo la struttura degli array, ovvero:

- Per ogni riga, gli indirizzi delle colonne sono incrementali:
 $\&[r][0] < \&[r][1] < \&[r][2]$
- Le singole righe sono incrementali:
 $\&[0][0] < \&[1][0] < \dots < \&[r][0]$

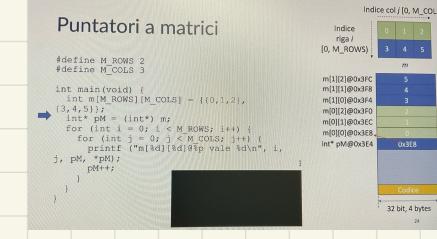
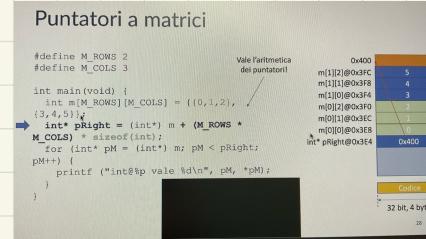


USO CON PUNTATORI >

Analogamente agli array, è possibile sfruttare l'infrastruttura del modello di memoria per scorrere gli elementi delle matrici.

Bisogna però ricordarsi che per accedere al 1° elemento va fatto il casting a puntatore:

`int* pMat = (int*) mat`



MATRICI RUGGED >

Per matrici rugged o matrici sfangiate si intendono delle matrici con numero di colonna diverso per ogni riga.

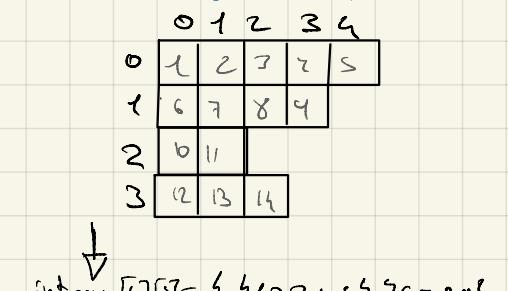
Naturalmente, il C non supporta questo tipo di matrici.

Ma è comunque possibile simularle.

IN C >

Per simularle in C, basta dichiarare la matrice con il massimo valore di righe e colonne per la matrice.

Per poi usare un array di supporto indicante le colonne per ogni riga.



`int arr[4] = {5, 4, 3, 2};`

`int arr[4] = {12, 13, 14, 15};`

`int arr[4] = {1, 2, 3, 4};`

`int arr[4] = {12, 13, 14, 15};`

`int arr[4] = {1, 2, 3, 4};`

HEAP >

La heap è un'altra memoria oltre allo stack che viene allocata per allocare dati dinamici.

ALLOCAZIONE >

L'allocamento viene gestito tramite la funzione `malloc()` di `stdlib.h`.

In particolare, `malloc()` richiede in input:

- La dimensione da allocare sulla heap, spesso usata con `sizeof()`, per allocare dimensioni riservate ai tipi.

In output

- L'indirizzo del primo byte associato. Spesso si fa il casting per assegnarlo a variabili puntatore.

ESEMPIO > `int* a = malloc(sizeof(int));` `int* pArr = malloc(sizeof(int)*5);`

`*a = 5; // sulla heap`

`(int*) seta`

`*pArr = 5; arr[0] = 5;`



esempio allocazione array con

`// int a[8]; -- stack`

`int* a = (int*) malloc(8 * sizeof(int));`

`a[0] = 10;`

`a[1] = 20;`

NOTA BENE: Se non si riesce

ad allocare la memoria

se esce errno se si prova ad

`malloc();` restituirà `NULL`

PROG-1 • MATRICI - 2023-11-20

HEAP >

FREE >

Dopo che è stato riservato dello spazio, è possibile liberarlo col metodo `free()`.
Esso accetta un puntatore in ingresso.

Per good practice, si assegna nullo alle variabili dopo che sono stati definiti. `int* pA = (int*)malloc(...);`
`pA = NULL;`
`free(pA);`
`/*more*/`

```
free
• lascia la memoria puntata dal puntatore dato
precondizione: il puntatore deve fare riferimento a un blocco
memoria allocato con malloc o realloc e non ha già fatto free
...
...
free(a)
• è considerata una buona pratica impostare in puntatore a NULL dopo
averlo utilizzato
free(a);
/*more*/
```

PROG-1 • FUNZIONI RICORSIVE • 2023-11-27

Le funzioni ricorsive sono delle funzioni che vengono richiamate se stesse.

In particolare, la funzione implementa

- Un caso base, risolvibile
- Un caso ricorsivo, che viene semplificato richiamando una parte risolvibile ed una parte semplificata ricorsiva

PROG 1 • Funzioni Ricorsive II - 2023-11-28

TIPOLOGIE DI RICORSIONE >

CONTRO-VARIANTE >

La ricorsione contro-variente parte da 0 fino a N

CO-VARIANTE >

La ricorsione co-variente parte da N fino a 0

DILO COMICA >

La ricorsione di locomica usa 2 indici, dei quali:

- i rappresenta il limite sx, che va da 0 fino a $\lfloor d/2 \rfloor$
- j rappresenta il limite dx che va da $(\lfloor d/2 \rfloor) + 1$ fino a N

Se $dx = sx$, allora si ha raggiunto la fine della ricorsione

] ricorsioni a destra

RICERCA DICOERICA >

È possibile utilizzare l'algoritmo di ricorsione di locomica per poter effettuare delle ricerche.

Questa tipologia di ricerca è detta binary search ed è

la versione ricorsiva della ricerca binaria

```
int binarySearch(int arr[], const int len, const int val, const int s, const int d) {
```

```
if(s == d) //caso base non trovato
    return arr[s] == val ? s : -1;
```

```
{else {
```

```
    int middle = (s+d)/2;
    if(val < arr[middle]) {
```

```
        return binarySearch(arr, len, val, s, middle);
```

```
    } else {
```

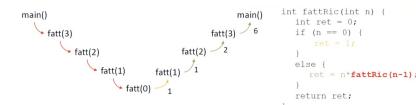
```
        return binarySearch(arr, len, val, middle, d);
```

```
}
```

```
}
```

Fattoriale ricorsivo

- La chiamata a funzione avviene per n decrescenti
- Il calcolo di $n!$ avviene per n crescenti al ritorno della ricorsione



```
int fattRic(int n) {
    int ret = 0;
    if(n == 0) {
        ret = 1;
    } else {
        ret = n*fattRic(n-1);
    }
    return ret;
```

contr-variente (arr, len, 0):

```
void contr_variente(int arr[], int len, int i) {
```

```
if(i == len) {
    /*caso base
    */
    else,
```

```
, contr_variente(arr, len, i+1);
```

co-variente (arr, len)

```
void co_variente(int arr[], int i) {
```

```
if(i == 0) {
    /*caso base
    */
    else,
    co_variente(arr, i-1);
```

di locomica (arr, s, d)

```
if(s == d) {
```

//caso base

```
{else {
```

```
    diLoco(arr, left, (s+d)/2);
```

```
    diLoco(arr, (s+d)/2, right);
```