



Funzioni e Sottoprogrammi

Informatica A AA 2024 / 2025

Giacomo Boracchi

25 Ottobre 2024

giacomo.boracchi@polimi.it

Slide credits Prof. Alessandro Campi



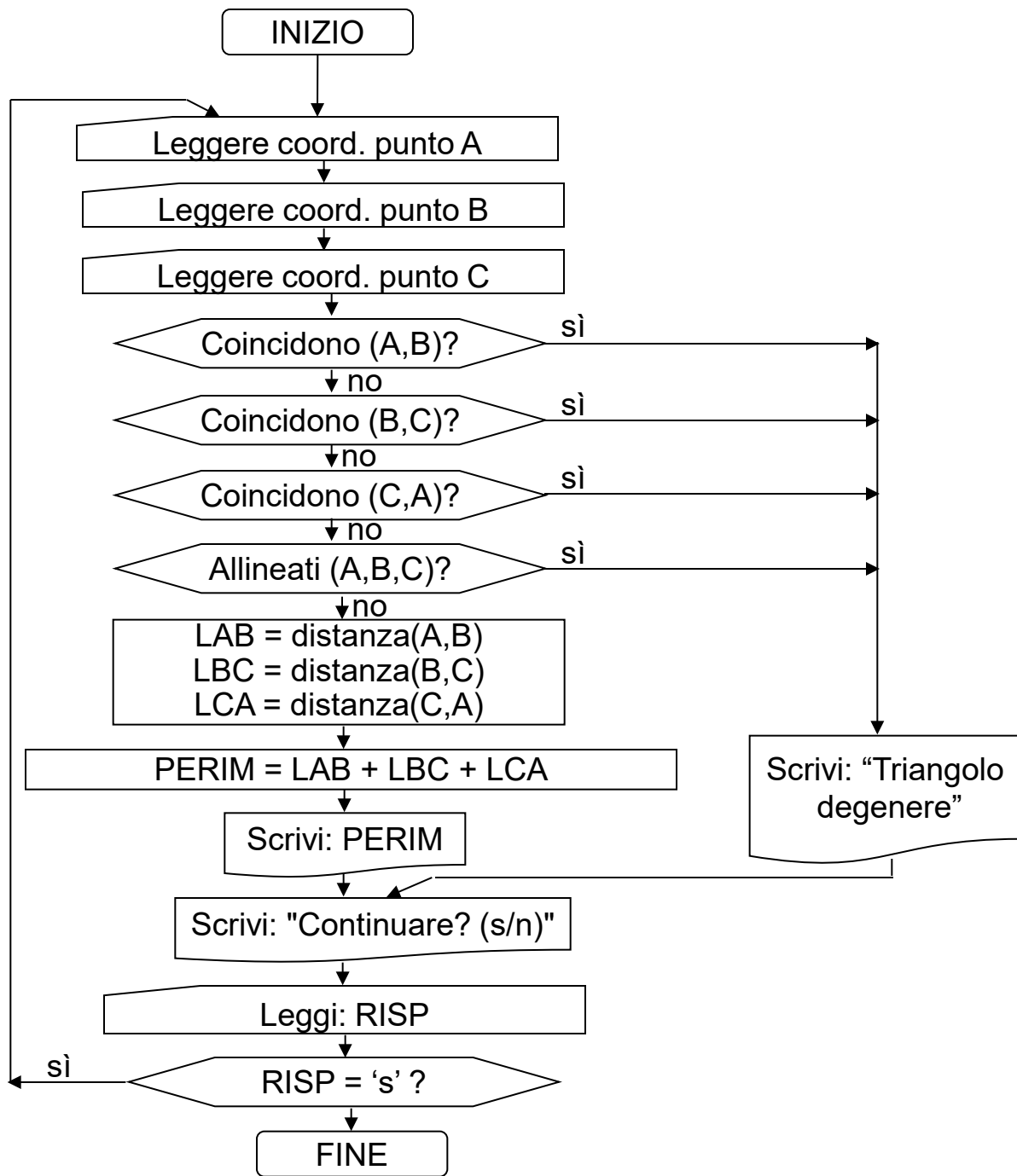
Funzioni

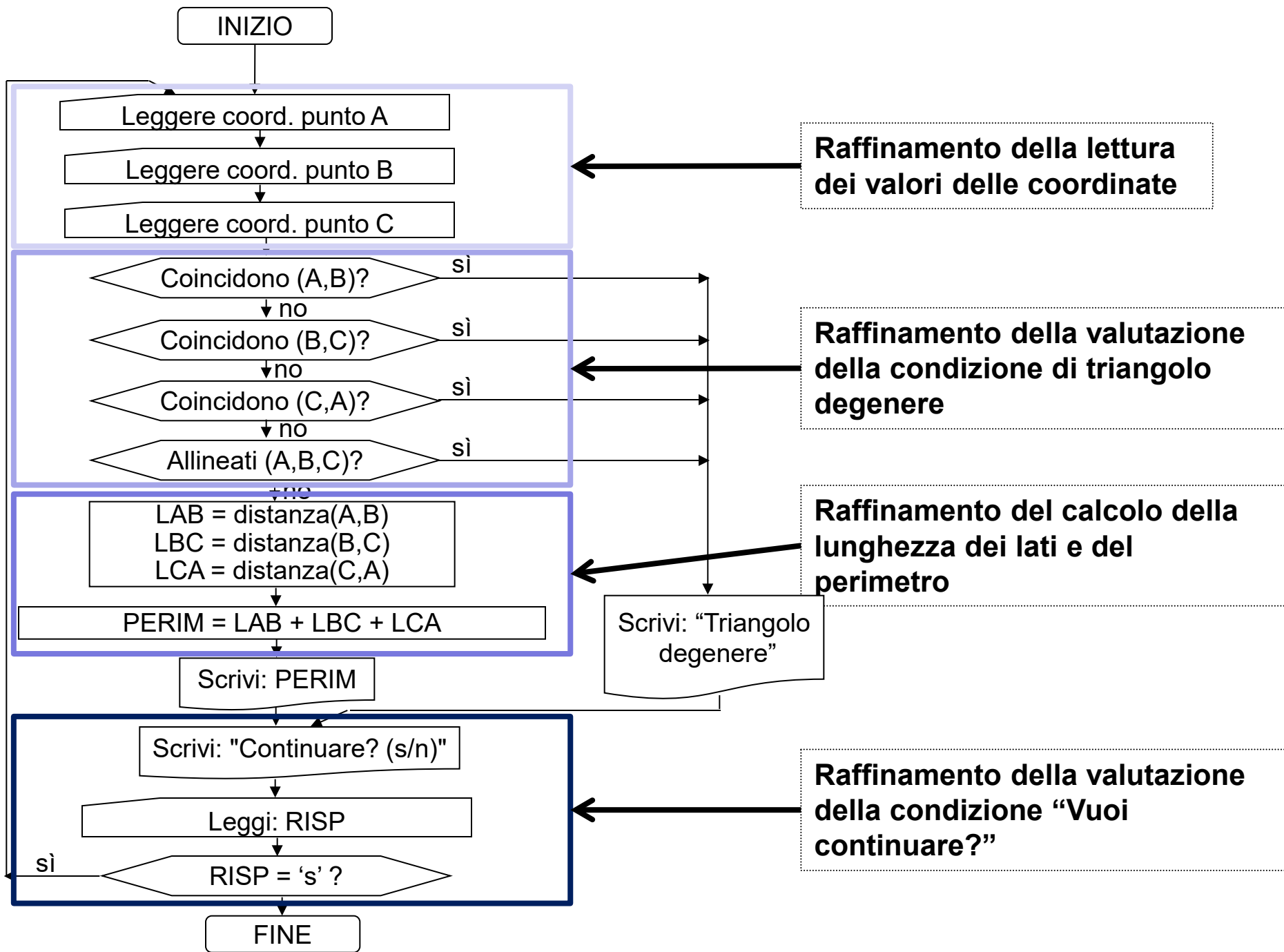
Perché le funzioni?



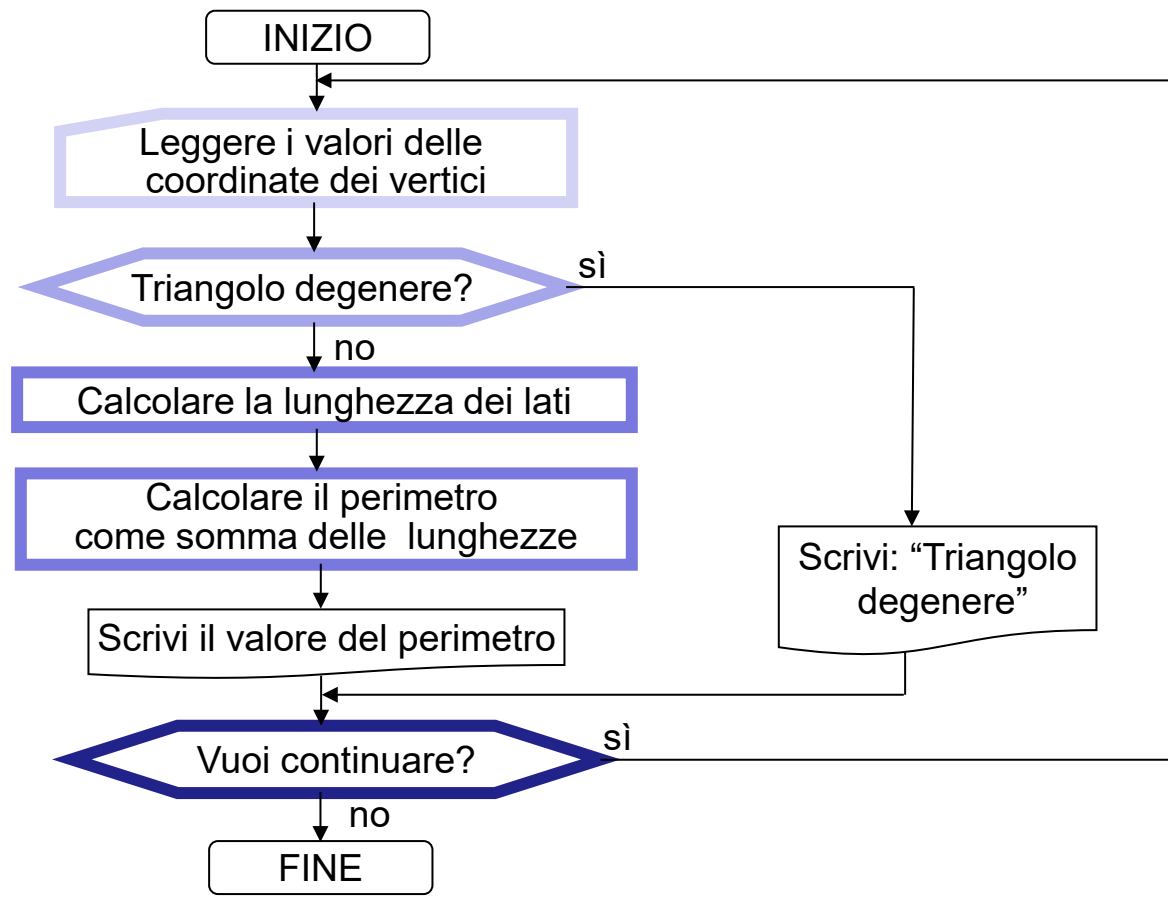
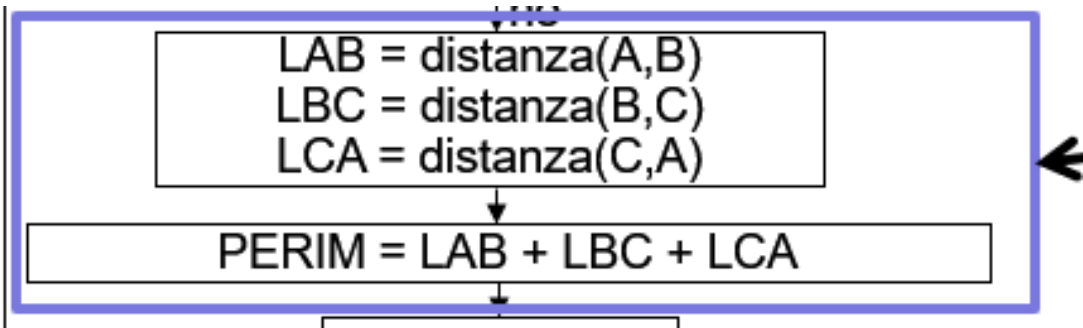
Problema

Date le coordinate di tre punti, riconoscere se sono i vertici di un triangolo non degenere, e nel caso calcolarne il perimetro





Così è più facile da leggere e da interpretare



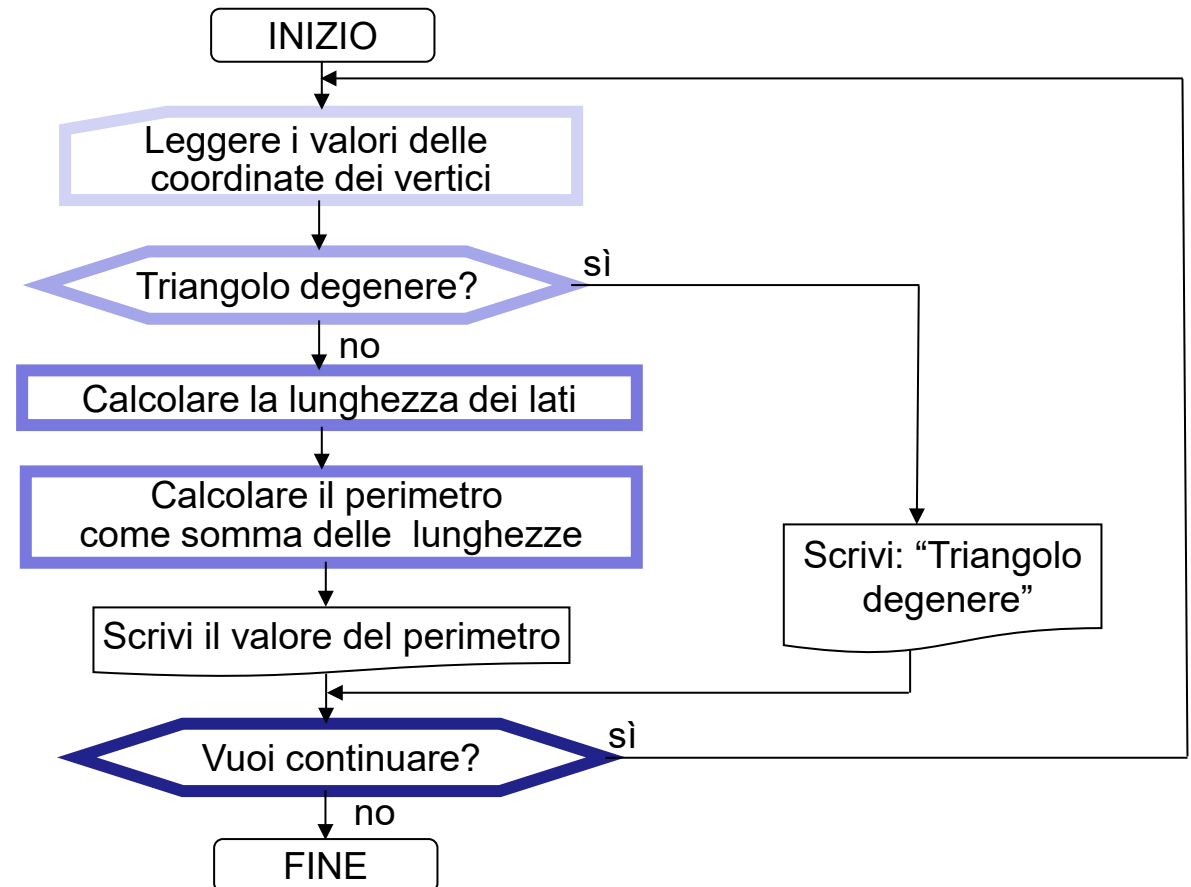
Sostituzione dei Blocchi

In un diagramma di flusso è possibile **sostituire dei blocchi che svolgono funzioni a basso livello con altri blocchi più ad «alto livello»**. E viceversa ovviamente.

La sostituzione avviene **collegando le frecce del flusso in ingresso ed in uscita**.

I **blocchi ad alto livello** rendono l'algorithmo **più comprensibile**.

In C, i blocchi ad alto livello sono le funzioni.





Abbiamo visto **sequenze di istruzioni che risolvono particolari “sottoproblemi”**: controllo della primalità di un numero, confronto di stringhe, etc.

È possibile **riutilizzare** queste sequenze?

È possibile **consentire ad altri di riutilizzare** queste sequenze?

È possibile **dare a queste sequenze un nome** che ne indichi la funzionalità?

È possibile **“svincolare”**

- lo sviluppo delle soluzioni ai sottoproblemi da
- lo sviluppo di una soluzione **“complessa”**?



Se fosse possibile...

...fare queste cose, i vantaggi sarebbero molti:

Potendo **riutilizzare** facilmente le sequenze, **aumenterebbe la produttività** (potremmo scrivere più codice in meno tempo)

Potremmo consentire ad altri di aumentare la loro produttività

Potremmo rendere il codice **più leggibile**

Potremmo “**spezzare**” la risoluzione di un problema in **sottoproblemi**



Il C offre, in risposta a queste esigenze, lo strumento delle **funzioni**

In C, una funzione è **una sequenza di comandi che:**

- **ha un nome**
- **può essere invocata** (cioè può esserne richiesta l'esecuzione)
- **può ricevere dei parametri** che ne influenzano l'esecuzione
- **produce un valore risultato**



Modularità nello sviluppo del codice

- Affrontare il problema per **raffinamenti successivi**

Riusabilità

- Scrivere **una sola volta** il codice e usarlo più volte
- Esempio: un algoritmo di ordinamento

Astrazione

- Esprimere in modo sintetico operazioni complesse
- Definire **operazioni** specifiche dei tipi di dato definiti dal programmatore
 - Esempio: calcolo “*totale + iva*” di un ordine
 - **punti, segmenti, poligoni, numeri complessi**



Sulla necessità dei sottoprogrammi

*I sottoprogrammi consentono di **scomporre** problemi complessi in moduli più semplici, sfruttabili poi, anche singolarmente, per la risoluzione di **problemi diversi**.*

*Se strutturati nel modo corretto, **nascondono** al resto del programma dettagli implementativi che non è necessario che esso conosca; in tal modo rendono **più chiaro** il programma nel suo complesso, e **assai più facile** la sua manutenzione [... o la sua scrittura corretta! NdA].*

B. W. Kernighan

D. M. Ritchie

Dennis Ritchie

From Wikipedia, the free encyclopedia

Dennis MacAlistair Ritchie (September 9, 1941 – October 12, 2011),^{[1][2][3][4]} commonly known by his **username** `dmr`, was an American **computer scientist** who "helped shape the digital era."^[1] He created the **C programming language** and, with long-time colleague, **Ken Thompson**, the **UNIX operating system**.^[1] Ritchie and Thompson received the **Turing Award** from the **ACM** in 1983, the **Hamming Medal** from the **IEEE** in 1990 and the **National Medal of Technology** from **President Clinton** in 1999. Ritchie was the head of **Lucent Technologies System Software Research Department** when he retired in 2007.

Dennis MacAlistair Ritchie



Dennis Ritchie, 1999

Dennis Ritchie

From Wikipedia, the free encyclopedia

Dennis MacAlistair Ritchie (September 9, 1941 – October 12, 2011),^{[1][2][3][4]} commonly known by his **username** `dmr`, was an American **computer scientist** who "helped shape the digital era."^[1] He created the **C programming language** and, with long-time colleague, **Ken Thompson**, the **UNIX operating system**.^[1] Ritchie and Thompson received the **Turing Award** from the **ACM** in 1983, the **Hamming Medal** from the **IEEE** in 1990 and the **National Medal of Science** in 1999. He was elected a **Fellow of the ACM** in 1999 and a **Fellow of the IEEE** in 2007.



B. W. Kernighan

D. M. Ritchie

Dennis MacAlistair Ritchie



Dennis Ritchie, 1999

Dennis Ritchie

From Wikipedia, the free encyclopedia

Dennis MacAlistair Ritchie (September 9, 1941 – October 12, 2011),^{[1][2][3][4]} commonly known by his

~~username~~ ~~he~~ ~~was~~ ~~an~~ ~~American~~ ~~computer~~ ~~scientist~~ ~~who~~



Isaac Newton once said he stood on the shoulders of giants. Dennis Ritchie was a giant whose shoulders people like Steve Jobs stood on.

Dennis MacAlistair Ritchie



chie, 1999



Esempio...

```
#include <stdio.h>
int main() {
    char scelta;
    int valore,risultato;
    while(1) { /* menu */
        printf("Premere A per inserire un num tra 0 e 10 e calcolarne il cubo\n");
        printf("Premere B per inserire un num tra 11 e 20 e calcolarne il quadrato\n");
        printf("Premere C per inserire un num tra 21 e 30 e calcolarne il doppio\n");
        printf("Premere Q per uscire\n\n");
        scanf(" %c",&scelta);
        switch(scelta) {
            case 'a':;
            case 'A':
                printf("Inserisci valore\n");
                scanf("%d",&valore);
                risultato=valore*valore*valore;
                printf("risultato=%d\n\n",risultato);
                break;
```

```
case 'b':;
```

```
case 'B':
```

```
    printf("Inserisci valore\n");
```

```
    scanf("%d",&valore);
```

```
    risultato=valore*valore;
```

```
    printf("risultato=%d\n\n",risultato);
```

```
    break;
```

```
case 'c':;
```

```
case 'C':
```

```
    printf("Inserisci valore\n");
```

```
    scanf("%d",&valore);
```

```
    risultato=valore*2;
```

```
    printf("risultato=%d\n\n",risultato);
```

```
    break;
```

```
case 'q':;
```

```
case 'Q':
```

```
    return 0;
```

```
}
```

```
}
```

```
}
```



Miglioriamo...

```
#include <stdio.h>
void menu();
int main() {
    char scelta;
    int valore,risultato;
    while(1) {
        menu();
        scanf("%c",&scelta);
        .....
        .....
    }
```

```
void menu() {
    printf("Premere A per inserire un num tra 0 e 10 e calcolarne il cubo\n");
    printf("Premere B per inserire un num tra 11 e 20 e calcolarne il quadrato\n");
    printf("Premere C per inserire un num tra 21 e 30 e calcolarne il doppio\n");
    printf("Premere Q per uscire\n\n");
}
```



Ancora...

```
char menu();  
int main() {  
    char scelta;  
    int valore,risultato;  
  
    while(1) {  
        scelta=menu();  
  
        .....  
        .....  
    }  
}
```

```
char menu() {  
    char ch;  
    printf("Premere A per un num tra 0 e 10 e calcolarne il cubo\n");  
    printf("Premere B per un num tra 11 e 20 e calcolarne il quadrato\n");  
    printf("Premere C per un num tra 21 e 30 e calcolarne il doppio\n");  
    printf("Premere Q per uscire\n\n");  
  
    scanf(" %c",&ch);  
    return ch;  
}
```



Altro miglioramento...

```
char menu();
int leggi();
int main() {
    char scelta; int valore, risultato;
    while(1) {
        scelta = menu();
        switch(scelta){
        case 'a': case 'A':
            valore=leggi();
            risultato=valore*valore*valore;
            printf("risultato=%d\n\n",risultato);
            break;
        .....
        .....
    }
}

int leggi() {
    int v;
    printf("Inserisci valore\n");
    scanf("%d",&v);
    return v;
}
```



E i limiti?

```
int leggi(int min,int max) {
    int v;
    int ok=0;

    while(!ok) {
        printf("Inserisci valore\n");
        scanf("%d",&v);
        if(v>=min && v<=max)
            ok=1;
    }

    return v;
}
```

```
char menu();

int leggi(int min,int max);

int main() {
    char scelta;
    int valore,risultato;
    while(1) {
        scelta=menu();
        switch(scelta){
            case 'a': case 'A':
                valore=leggi(0,10);
                risultato=valore*valore*valore;
                printf("risultato=%d\n\n",risultato);
                break;
            case 'b': case 'B':
                valore=leggi(11,20);
                risultato=valore*valore;
                printf("risultato=%d\n\n",risultato);
                break;
```

Come si usano le funzioni in C?

... un po' di sintassi



Definizione di funzioni

```
<tipo restituito> <nome funzione> (<lista argomenti input>) {  
    <variabili locali>  
    <corpo della funzione>  
}
```

All'interno delle funzioni, l'istruzione *return* consente di *restituire* un solo valore
Il valore restituito deve essere di *<tipo restituito>*

<lista degli argomenti> è una sequenza di coppie: tipo, nomeVariabile



Esempio: cosa fa?

```
#include <stdio.h>

int main()
{
    int i,n;
    int primo;
    for (i=1;i<=100;i++)
    {
        primo=1;
        for (n=2;n<=i/2 && primo==1;n++)
            if ((i%n)==0)
                primo=0;
        if (primo==1)
            printf("%d\n",i);
    }
}
```



Esempio

```
#include <stdio.h>

int main()
{
    int i,n;
    int primo;
    for (i=1;i<=100;i++)
    {
        primo=1;
        for (n=2;n<=i/2 && primo==1;n++)
            if ((i%n)==0)
                primo=0;
        if (primo==1)
            printf("%d\n",i);
    }
    return 0;
}
```

La porzione di codice evidenziata calcola se il valore della variabile **i** è un numero primo

Il valore su cui 'lavora' è il valore della variabile **i**

Il suo 'risultato' è il valore della variabile **primo**



Definizione della funzione verifica_primo

```
int verifica_primo(int numero){  
    int primo=1, n;  
  
    for (n=2;n<=numero/2 && primo==1;n++)  
        if ((numero%n)==0)  
            primo=0;  
  
    return primo;  
}
```

Valore restituito



Esempio

```
#include <stdio.h>

int main(){
    int i,n;
    int primo;

    for (i=1;i<=100;i++){
        primo=verifica_primo(i);
        if (primo==1)
            printf("%d\n",i);
    }
}
```

Invocazione della funzione



Invocazione di una funzione: sintassi

Una funzione può essere vista come un' *espressione*, che può essere *valutata* (es: possiamo immaginare l'invocazione **somma (a , b)** equivalente ad **a+b**)

Quando, nell'esecuzione del codice, è necessario 'valutare' un'espressione-funzione, la funzione viene *invocata*

La sintassi per la formulazione di una espressione-funzione è semplicemente:

<nome funzione>(<lista argomenti>) ,

dove **<lista argomenti>** è una lista di **espressioni** (variabili, costanti, invocazione di funzioni) di tipo corrispondente a quelle della definizione



Quando una funzione viene invocata:

1. Le **espressioni** che ne costituiscono gli argomenti vengono **valutate**, e il valore reso disponibile alla funzione
2. La **funzione viene eseguita**, fino a quando non viene incontrato un comando **return**
3. Il “**valore**” dell'espressione-funzione è il valore dell'argomento del **return** che ha causato la terminazione



Esempio

```
int verifica_primo(int numero);

int main(){
    int i;

    for (i=1;i<=100;i++){
        if (verifica_primo(i)==1)
            printf("%d\n",i);
    }
    return 0;
}
```




Esempio

```
int verifica_primo(int numero);
```

```
int main(){
```

```
    int i;
```

```
    for (i=1;i<=100;i++){
```

```
        if (verifica_primo(i)==1)
```

```
            printf("%d\n",i);
```

```
    }
```

```
    return 0;
```

```
}
```

Programma identico al precedente perché l'argomento dell'espressione non cambia



Esempio

La funzione **controllaSePrimo** può essere utilizzata anche in altri codici, come nell'esempio che segue

```
#include <stdio.h>
#define L 10
int main()
{
    int i, n, j=0, flag, vet[L], pri[L];
    for(i = 0; i <L; i++){
        printf("\n inserire vet[%d]:", i);
        scanf("%d", &vet[i]);}

    for(i = 0; i < L; i++){
        flag=1;
        for(n=2;n<= vet[i]/2 && flag==1;n++)
            if((vet[i]%n)==0)
                flag=0;

        if(flag==1){
            pri[j] = vet[i];
            j++;
        }
    }
    printf("\n[");
    for(i = 0; i<j; i++)
        printf("%d, ",pri[i]);
    printf("]");

    return 0;
}
```

```
int controllaSePrimo(int num)
{
    int n, flag;

    flag=1;
    for(n=2; n<= num/2 && flag==1;n++)
        if((num%n)==0)
            flag=0;
    return flag;
}
```

```
int main()
{
    int i, j=0, flag, vet[L], pri[L];

    for(i = 0; i <L; i++){
        printf("\ninserire vet[%d]:", i);
        scanf("%d", &vet[i]);}

    for(i = 0; i <L; i++){
        if(controllaSePrimo(vet[i]) == 1){
            pri[j] = vet[i];
            j++;
        }
    }

    printf("\n[");
    for(i = 0; i<j; i++)
        printf("%d, ",pri[i]);
    printf("]");

    return 0;
}
```

Maggior leggibilità



I Sottoprogrammi

Quasi tutti i linguaggi di programmazione hanno una nozione di sottoprogramma

Esistono sottoprogrammi che restituiscono valori e sottoprogrammi che non lo fanno

Esempio:

```
double power (double val, int pow) {  
    double ret_val = 1.0;  
    int i;  
  
    for (i = 0; i < pow; i++)  
        ret_val = ret_val * val;  
  
    return ret_val;  
}
```



“Chiamata” al sottoprogramma precedente:

```
result = power (valore, esponente);
```

Il C suppone che tutti i sottoprogrammi restituiscano un valore. Se non si vuole che ciò accada, **occorre segnalarlo**

Un esempio di sottoprogramma **che non restituisce valori:**

```
void error_line (int line) {  
    printf ("Errore alla linea %d\n", line);  
}
```

Il sottoprogramma viene chiamato come segue:

```
error_line (line_number);
```



Funzioni e Procedure

In linguaggio informatico:

- I sottoprogrammi che restituiscono valori vengono chiamati **funzioni**
- I sottoprogrammi che **non** restituiscono valori vengono chiamati **procedure**

In C vengono chiamati tutti **funzioni**

Struttura di un programma con le funzioni

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

```
int power(int base, int n);
```

Dichiarazione della
funzione / prototipo

chiamate
di funzione

```
int main() {
```

```
    int i;
```

```
    for (i=0; i<10; i++)
```

```
        printf("%d %d %d\n", i, power(2,i), power(3,i));
```

```
    return 0;
```

```
}
```

```
int power(int base, int n) {
```

```
    int i, p=1;
```

```
    for(i=1; i<=n; i++)
```

```
        p=p*base;
```

```
    return p;
```

```
}
```

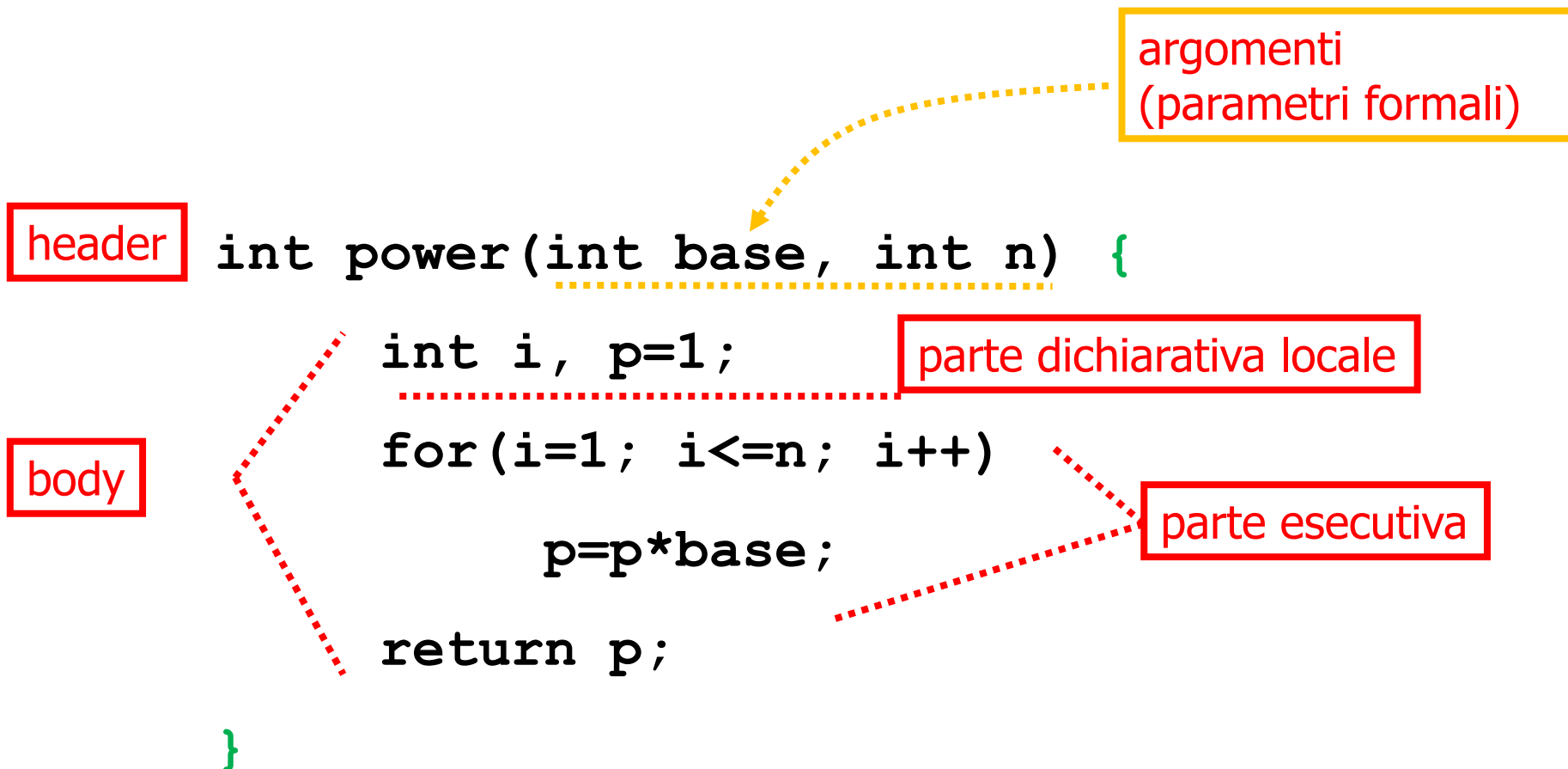
/*base elevata a n*/

definizione
di funzione

Definizione di variabili
locali alla funzione



Struttura di una funzione





Definizione di una funzione in C

Function header (testata):

- **tipo del risultato** (codominio della funzione)
- **identificatore del sottoprogramma** (nome della funzione)
- **elenco delle dichiarazioni dei parametri formali** in ingresso
 - sono **gli argomenti (il dominio) della funzione**, e ricevono un valore nel momento in cui la funzione viene attivata

Parentesi graffe che racchiudono:

- Parte dichiarativa locale.
- Corpo della funzione

```
double power (double val, int pow) {  
    double ret_val = 1.0;  
    int i;  
    for (i = 0; i < pow; i++)  
        ret_val = ret_val * val;  
    return (ret_val);  
}
```



Invocazione di una funzione in C

Ricordiamo:

```
result = power(valore, esponente) ;
```

- Identificatore della funzione
- Lista dei **parametri attuali** tra parentesi
 - parametri attuali: **valori degli argomenti** ai quali viene applicata la funzione
 - ogni **parametro attuale viene valutato come un'espressione** e può contenere altre chiamate di funzione
- Sintatticamente la chiamata è un'espressione

Corrispondenza posizionale

- al primo corrisponde il primo, al secondo il secondo, e così via ...



Parametri formali e attuali

Nella **definizione** della funzione si dà un nome ai **parametri formali** che la funzione riceve in ingresso

- In quel momento non vi è associato alcun valore

```
float power (float v, int e) { ... corpo ... }
```

Al momento della sua **invocazione** (chiamata) si definisce **il valore di ogni parametro attuale** corrispondente ad un' *espressione*

- Ogni espressione viene valutata e ha un tipo e un valore

```
result = power ( (strlen(s) - 3) * val) , 2) ;
```

```
result = power ( (power(6, 9) - 3) * val) , 2) ;
```

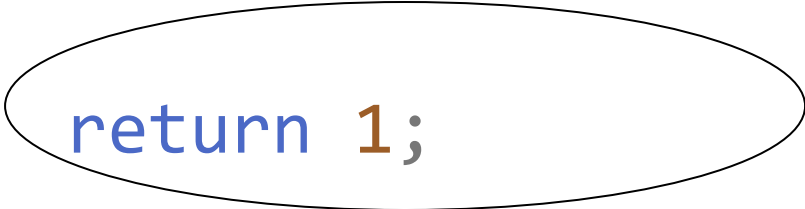
La corrispondenza tra parametri formali e attuali è **posizionale**

- Al primo corrisponde il primo, al secondo il secondo...
- I tipi devono corrispondere (eventualmente c'è cast implicito)



Definizione della funzione verifica_primo

```
int verifica_primo(int numero){  
    int n;  
  
    for (n=2;n<=numero/2 ;n++)  
        if ((numero%n)==0)  
            return 0;  
  
    return 1;  
}
```



Valore di ritorno



return espressione;

- Il valore dell'invocazione diventa il valore che segue *return*
- Termina l'esecuzione del sottoprogramma, il controllo torna al programma chiamante
- Visto dal programma chiamante, il valore restituito è il valore dell'espressione dell'invocazione

In un sottoprogramma possono esserci **nessuna o più istruzioni di *return***

- Ovviamente a ogni chiamata **se ne esegue al massimo una**

Le variabili dichiarate localmente “muoiono” all'uscita dal sottoprogramma



Void: funzioni senza return

Se l'istruzione *return* è assente, il sottoprogramma termina quando arriva alla } finale

Il **tipo speciale *void*** viene usato per indicare il caso di assenza di valori di uscita o di parametri.

Esempio:

```
void menu() {  
    printf("Premere A per inserire un num tra 0 e 10 e calcolarne il cubo\n");  
    printf("Premere B per inserire un num tra 11 e 20 e calcolarne il quadrato\n");  
    printf("Premere C per inserire un num tra 21 e 30 e calcolarne il doppio\n");  
    printf("Premere Q per uscire\n\n");  
}
```


Funzioni e la memoria: le variabili nelle funzioni



Parametri Formali e Parametri Attuali

Definizioni:

- I **parametri formali** sono le variabili usate come **argomenti nella definizione** della funzione ed il **valore restituito con return**
- I **parametri attuali** sono le variabili (o i valori) usati come **argomenti** o per raccogliere il valore restituito **dall'invocazione** della funzione

Tutti i parametri, in C, sono passati per **COPIA**

Esempio di invocazione

```
somma = sommaDivisori(x);
```

Comporta un assegnamento del valore restituito da **sommaDivisori(x)** nella variabile `somma`



Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un  
intero (escluso l'intero stesso)  
#include<stdio.h>
```

```
int sommaDivisori(int n);
```

```
int main()
```

```
{
```

```
    int x, somma;
```

```
    printf("inserire x: ");
```

```
    scanf("%d", &x);
```

```
    somma = sommaDivisori(x);
```

```
    printf("\n la somma dei divisori di %d e' %d", x, somma);
```

```
    return 0;
```

```
}
```

```
int sommaDivisori(int n)
```

```
{
```

```
    int i, s = 0;
```

```
    // tra le variabili ho già n
```

```
    for(i = 1; i <= n/2; i++)
```

```
        if(n % i == 0)
```

```
            s = s + i;
```

```
    return s;
```

```
}
```



Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un
intero (escluso l'intero stesso)
#include<stdio.h>
```

```
int sommaDivisori(int n);
```

```
int main()
```

```
{
```

```
    int x, somma;
```

```
    printf("inserire x: ");
```

```
    scanf("%d", &x);
```

```
    somma = sommaDivisori(x);
```

```
    printf("\n la somma dei divisori di %d e' %d", x, somma);
```

```
    return 0;
```

```
}
```

```
int sommaDivisori(int n)
```

```
{
```

```
    int i, s = 0;
```

```
    // tra le variabili ho già n
```

```
    for(i = 1; i <= n/2; i++)
```

```
        if(n % i == 0)
```

```
            s = s + i;
```

```
    return s;
```

```
}
```

Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un
intero (escluso l'intero stesso)
#include<stdio.h>
```

```
int sommaDivisori(int n);
```

```
int main()
```

```
{
```

```
    int x, somma;
```

```
    printf("inserire x: ");
```

```
    scanf("%d", &x);
```

```
    somma = sommaDivisori(x);
```

```
    printf("\n la somma dei divisori di %d e' %d", x, somma);
```

```
    return 0;
```

```
}
```

```
int sommaDivisori(int n)
```

```
{
```

```
    int i, s = 0;
```

```
    // tra le variabili ho già n
```

```
    for(i = 1; i <= n/2; i++)
```

```
        if(n % i == 0)
```

```
            s = s + i;
```

```
    return s;
```

```
}
```

Nel caso ci fossero più parametri

Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un  
intero (escluso l'intero stesso)
```

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

```
int sommaDivisori(int n);
```

```
int main()  
{
```

```
int x, somma;
```

```
printf("inserire x: ");
```

```
scanf("%d", &x);
```

```
somma = sommaDivisori(x);
```

```
printf("\n la somma dei divisori di %d e' %d", x, somma);
```

```
return 0;
```

```
}
```

Parametro(i) attuali (i)
(possono essere più di uno)

Parametro(i) formale(i)
(possono essere più di uno)

```
int sommaDivisori(int n)
```

```
{
```

```
int i, s = 0;
```

```
// tra le variabili ho già n
```

```
for(i = 1; i <= n/2; i++)
```

```
if(n % i == 0)
```

```
s = s + i;
```

```
return s;
```

```
}
```

Valore restituito dalla
funzione

Assegnamento del valore
restituito dalla funzione

Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un
intero (escluso l'intero stesso)
#include<stdio.h>
```

```
int sommaDivisori(int n);
```

```
int main()
```

```
{
```

```
    int x, somma;
```

```
    printf("inserire x: ");
```

```
    scanf("%d", &x);
```

```
    somma = sommaDivisori(x);
```

```
    printf("\n la somma dei divisori di %d e' %d", x, somma);
```

```
    return 0;
```

```
}
```

Passaggio **per copia**: all'invocazione, il valore di *x* viene copiato in *n*.
Il valore di *s*, al termine dell'esecuzione si sostituisce all'invocazione e viene assegnato a *somma*

```
int sommaDivisori(int n)
```

```
{
```

```
    int i, s = 0;
```

```
    // tra le variabili ho già n
```

```
    for(i = 1; i <= n/2; i++)
```

```
        if(n % i == 0)
```

```
            s = s + i;
```

```
    return s;
```

```
}
```



Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un intero (escluso l'intero stesso)
#include<stdio.h>

int sommaDivisori(int n);

int main()
{
    int x, somma;
    printf("inserire x: ");
    scanf("%d", &x);

    somma = sommaDivisori(x);

    printf("\n la somma dei divisori di %d e' %d", x, sommaDivisori(x));
    return 0;
}
```

```
int sommaDivisori(int n)
{
    int i, s = 0;
    // tra le variabili ho già n
    for(i = 1; i <= n/2; i++)
        if(n % i == 0)
            s = s + i;

    return s;
}
```

Il valore restituito potrebbe non venir raccolto da una variabile

Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un intero  
(escluso l'intero stesso)
```

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

```
int sommaDivisori(int);
```

```
/* non è importante il nome della variabile inserito nel prototipo */
```

```
// int sommaDivisori(int x);
```

```
// int sommaDivisori(int n);
```

```
// int sommaDivisori(int y);
```

```
int main()
```

```
{
```

```
int x, somma;
```

```
printf("inserire x: ");
```

```
scanf("%d", &x);
```

```
somma = sommaDivisori(x);
```

```
printf("\n la somma dei divisori di %d e' %d", x, sommaDivisori(x));
```

```
return 0;
```

```
}
```

```
int sommaDivisori(int n)
```

```
{
```

```
int i, s = 0;
```

```
// tra le variabili ho già n
```

```
for(i = 1; i <= n/2; i++)
```

```
    if(n % i == 0)
```

```
        s = s + i;
```

```
return s;
```

```
}
```

Parametri Formali e Parametri Attuali

```
// scrivere un programma calcola la somma dei divisori di un intero  
(escluso l'intero stesso)
```

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

```
int sommaDivisori(int);
```

```
/* non è importante il nome della variabile inserito nel prototipo */
```

```
// int sommaDivisori(int x);
```

```
// int sommaDivisori(int n);
```

```
// int sommaDivisori(int y);
```

```
int main()
```

```
{
```

```
    int n, somma;
```

```
    printf("inserire x: ");
```

```
    scanf("%d", &n);
```

```
    somma = sommaDivisori(n);
```

```
    printf("\n la somma dei divisori di %d e' %d", n, sommaDivisori(n));
```

```
    return 0;
```

```
}
```

```
int sommaDivisori(int n)
```

```
{
```

```
    int i, s = 0;
```

```
    // tra le variabili ho già n
```

```
    for(i = 1; i <= n/2; i++)
```

```
        if(n % i == 0)
```

```
            s = s + i;
```

```
    return s;
```

```
}
```

Se nel main ci fosse n non ci sarebbero problemi, sarebbe distinta da n in **sommaDivisori**



Osservazioni:

- La variabile da restituire (*s*) va dichiarata
- Il parametro formale (*n*) NON va dichiarato, perché già viene definito. Dichiararlo nuovamente è un errore.
- All'interno del corpo di questa funzione ho visibilità solo delle variabili *i*, *s* (che sono state dichiarate) e *n* (parametro formale). Queste appartengono ad uno spazio di memoria separato (Record di Attivazione) che viene distrutto terminata l'invocazione

```
int sommaDivisori(int n)
{
    int i, s = 0;
    // tra le variabili ho già n
    for(i = 1; i <= n/2; i++)
        if(n % i == 0)
            s = s + i;

    return s;
}
```



Omonimia di variabili: la variabile `i`

```
#include <stdio.h>

int power(int, int);

int main() {
    int i;
    for (i=0; i<10; i++)
        printf("%d %d %d\n", i, power(2,i), power(3,i));
    return 0;
}

int power( int base, int n ) {    /* base intera elevata a n
intero */
    int i, p=1;
    for(i=1; i<=n; i++ )
        p*=base;
    return p;
}
```

Omonimia di variabili: la variabile `i`

```
#include <stdio.h>

int power(int, int);

int main() {
    int i;
    for (i=0; i<10; i++)
        printf("%d %d %d\n", i, power(2,i), power(3,i));
    return 0;
}

int power( int base, int n ) {    /* base intera elevata a n
intero */
    int i, p=1;
    for(i=1; i<=n; i++ )
        p*=base;
    return p;
}
```

La variabile `i` locale alla funzione `power` non ha alcun rapporto con quella locale alla funzione `main`. L'omonimia era casuale, possiamo rinominare una delle due variabili per enfatizzarne la totale indipendenza.

Omonimia di variabili: la variabile `i`

```
#include <stdio.h>

int power(int, int);

int main() {
    int i;
    for (i=0; i<10; i++)
        printf("%d %d %d\n", i, power(2,i), power(3,i));
    return 0;
}

int power( int base, int n ) {    /* base intera elevata a n
intero */
    int contatore, p=1;
    for(contatore=1; contatore <=n; contatore++ )
        p*=base;
    return p;
}
```

La variabile `i` locale alla funzione `power` non ha alcun rapporto con quella locale alla funzione `main`. L'omonimia era casuale, possiamo rinominare una delle due variabili per enfatizzarne la totale indipendenza.



Variabili Locali e Globali

Ogni sottoprogramma può usare

- solo le variabili dichiarate al suo interno
- I parametri formali (che vengono già inizializzati all'invocazione)

Le variabili locali nascono e muoiono con la vita del sottoprogramma (lifetime)

- Dal momento in cui avviene il trasferimento del controllo al momento in cui il controllo è restituito al modulo "chiamante"

Variabili non dichiarate all'interno di un sottoprogramma si possono usare solo **se passate come argomenti**

Gestione Della Memoria



Funzioni: modello di esecuzione

Immaginiamo che esista **una macchina dedicata** a eseguire il programma `main()`

Immaginiamo che sia **creata** una nuova macchina dedicata, all'atto della **chiamata** di ogni funzione

Le macchine dedicate hanno una loro **memoria**

- per le variabili locali,
- per i valori dei parametri che ricevono, e
- per il risultato che eventualmente restituiscono

Tale memoria è anche detta ***ambiente*** della funzione

Vediamo come UNA SOLA MACCHINA può simulare questo modello di esecuzione



Riassumendo

Per ogni chiamata

- si crea *virtualmente* una macchina dedicata

Al termine del sottoprogramma

- la macchina dedicata viene *virtualmente* distrutta

Vediamo come queste "creazioni / distruzioni" sono ottenute, in pratica, nell'implementazione del linguaggio (simulazione delle "macchine virtuali dedicate")



Chiamata a sottoprogramma

In seguito a una chiamata a sottoprogramma, il programma in corso viene sospeso e **il controllo passa al sottoprogramma**

Al livello della macchina C:

- Salvataggio del program counter (PC) e del **contesto** del programma chiamante
- Assegnazione al PC dell'indirizzo del sottoprogramma
- Esecuzione del sottoprogramma
- Ritorno al programma chiamante con ripristino del suo **contesto**



Record di attivazione

Ogni sottoprogramma (incluso il main) ha associato un **record di attivazione**. Contiene:

- tutti i dati relativi **all'ambiente locale del sottoprogramma**
- **l'indirizzo di ritorno** nel programma chiamante
- altri dati utili

Per ogni attivazione di sottoprogramma si crea un nuovo record di attivazione



Perché è necessario?

In generale, una funzione può essere chiamata un numero imprecisato di volte

Ogni chiamata a procedura richiede allocazione di spazio di memoria per le sue variabili locali

- Il compilatore potrebbe “preparare” un ambiente per l’esecuzione di ogni funzione?
In generale no...

Vi sono procedure che richiamano se stesse (vedremo: ricorsione)

- **Possono esistere più “istanze” di una funzione, “addormentate” in attesa della terminazione di una “gemella” per riprendere l’esecuzione**

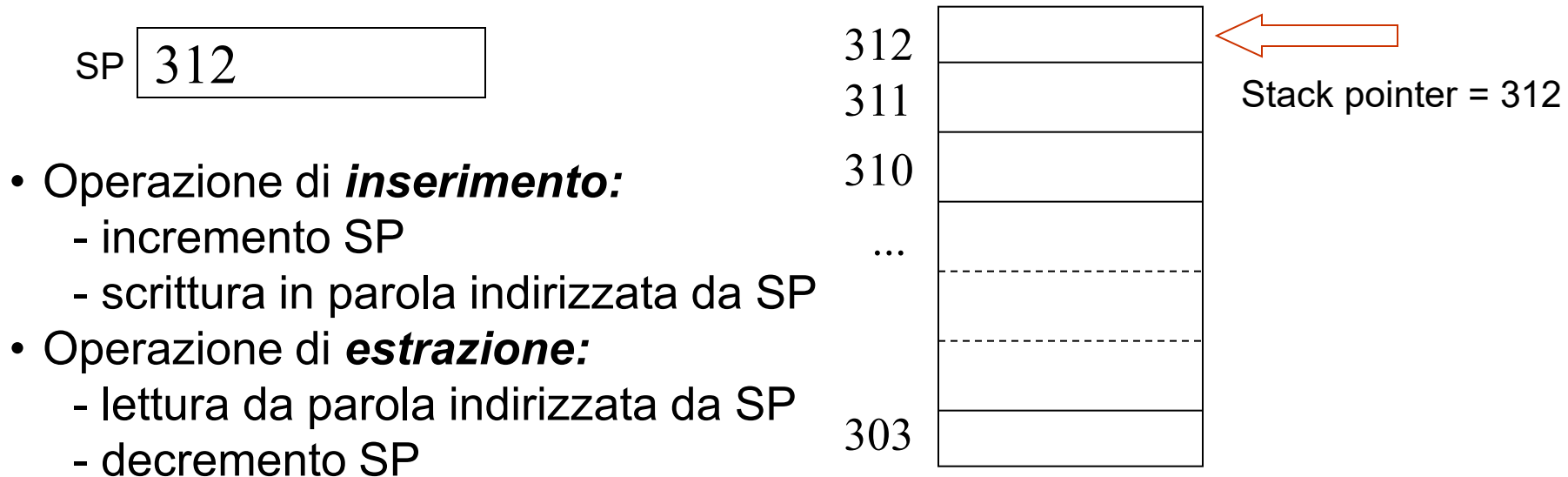
In questo ultimo caso **il compilatore non può sapere** quanto spazio allocare per le variabili del programma (negli ambienti che corrispondono alle varie invocazioni)



La pila (stack) di sistema

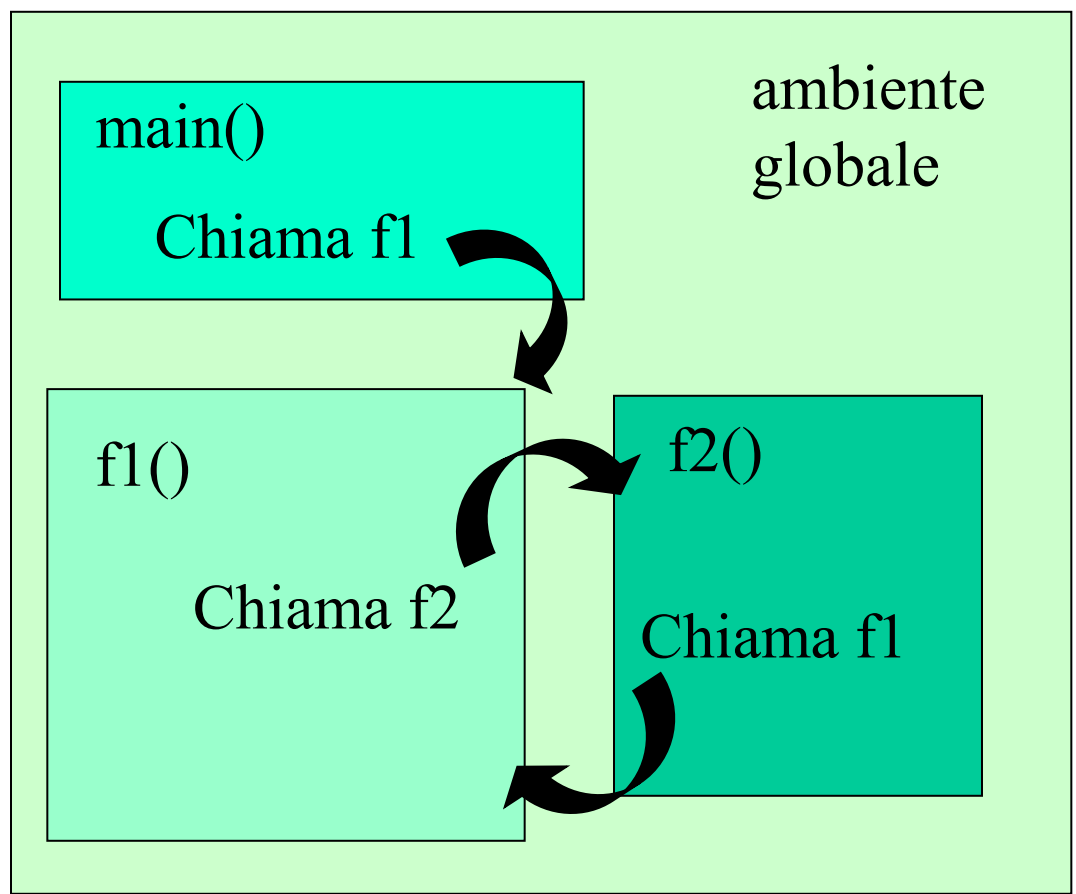
Una porzione della memoria di lavoro, chiamata **stack** (*pila*): modalità **LIFO** (Last in First Out) permette al sistema operativo di gestire i processi e di eseguire le chiamate a sottoprogramma

Lo **Stack Pointer** (puntat. alla pila) è un registro che contiene l'indirizzo della parola di memoria da leggere nello stack





Esempio

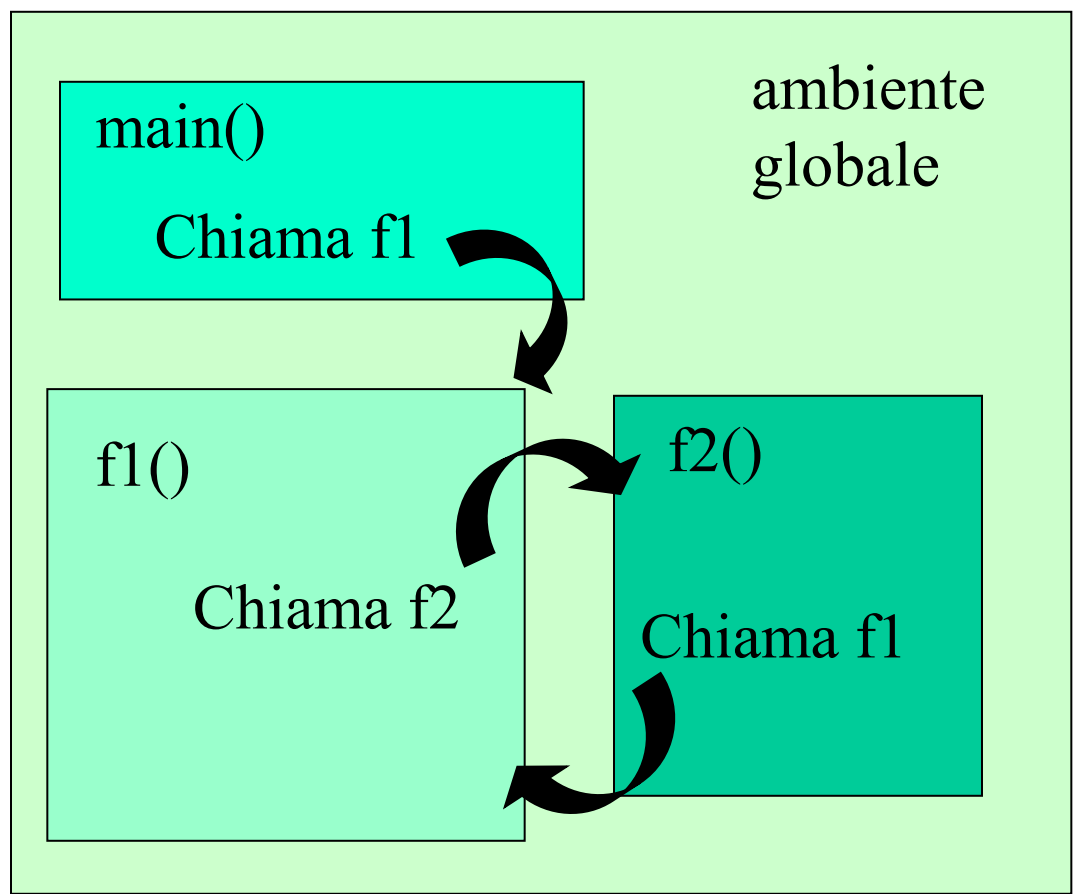


crescita verso l'alto 

...
r.d.a. di f1(3°)
r.d.a. di f2(2°)
r.d.a. di f1(2°)
r.d.a. di f2(1°)
r.d.a. di f1(1°)
r.d.a. di main()
variabili globali



Esempio

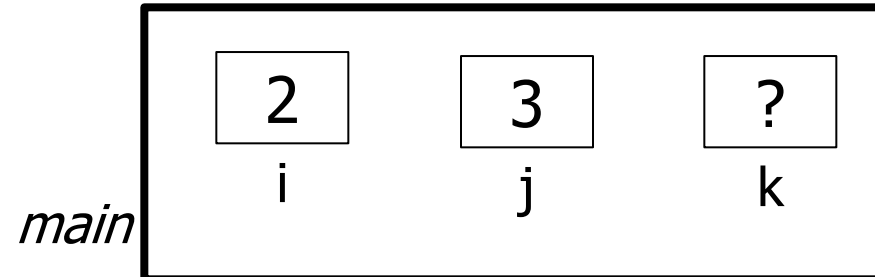


crescita verso l'alto 

...
r.d.a. di f1(3°)
r.d.a. di f2(2°)
r.d.a. di f1(2°)
r.d.a. di f2(1°)
r.d.a. di f1(1°)
r.d.a. di main()
variabili globali

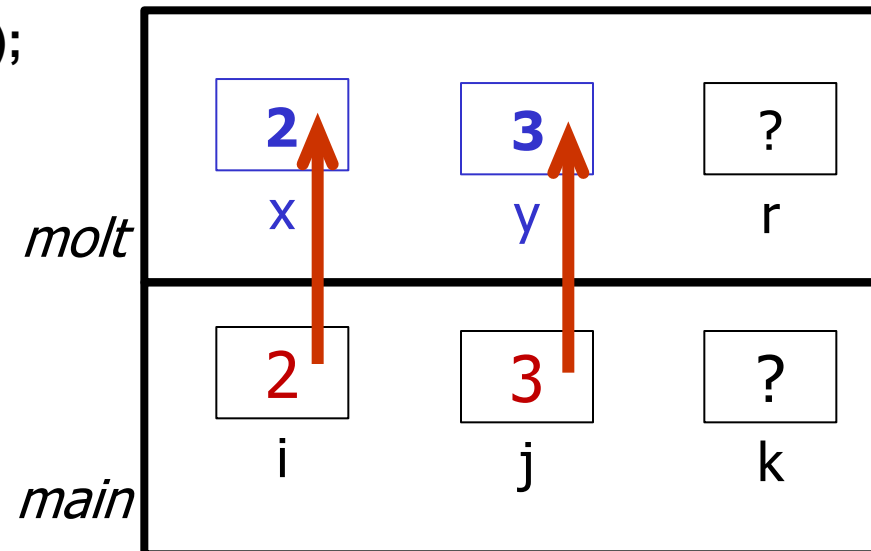
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



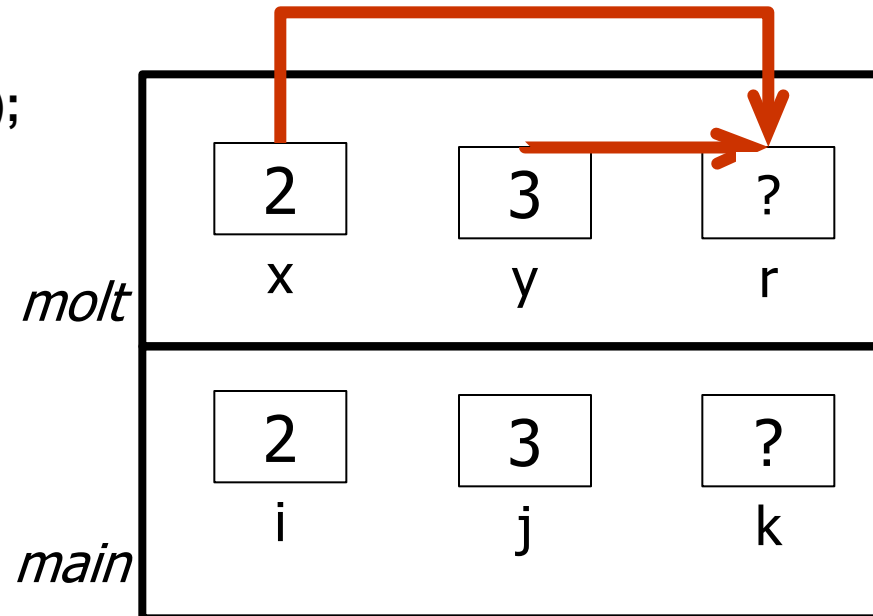
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica( i, j );  
    k=power(i,j);  
    return 0;  
}
```



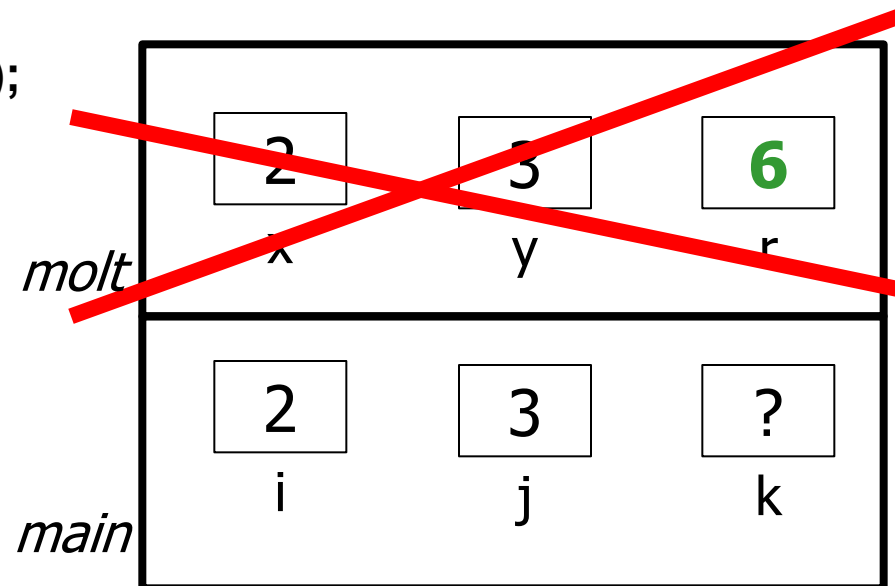
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



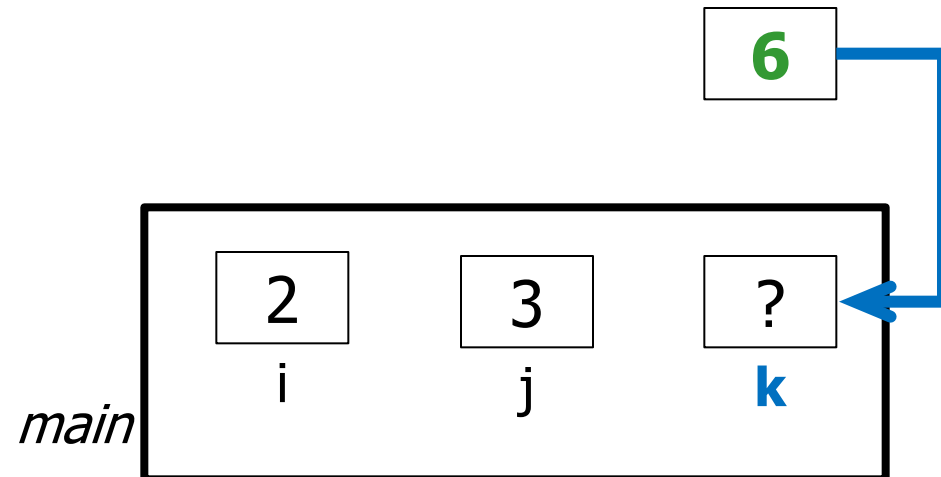
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



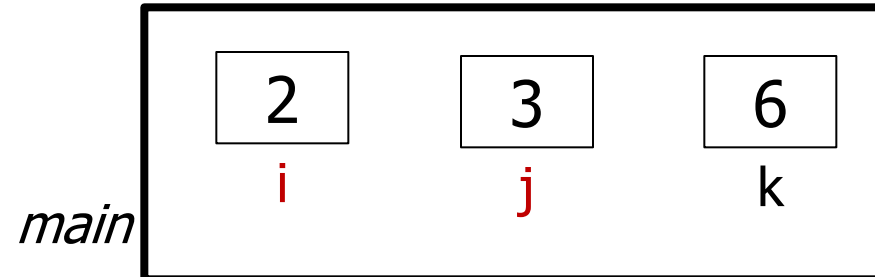
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



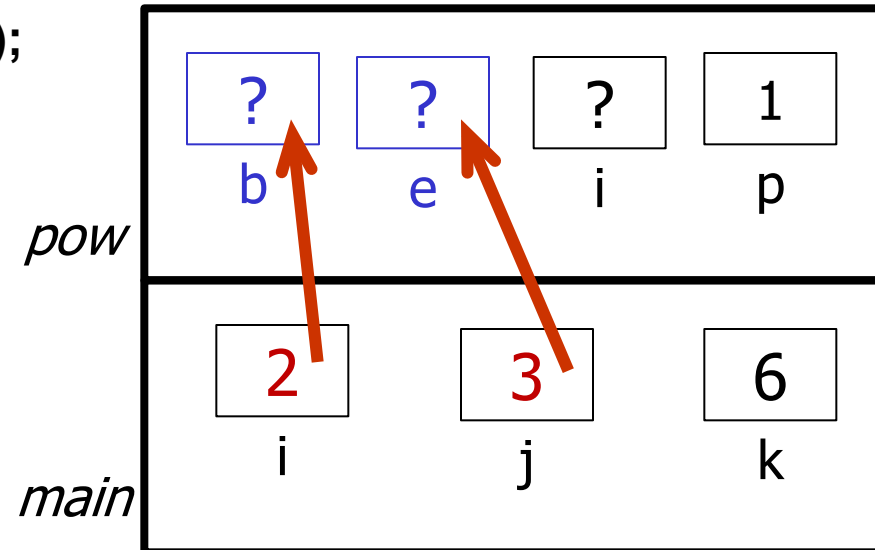
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k = power( i, j );  
    return 0;  
}
```



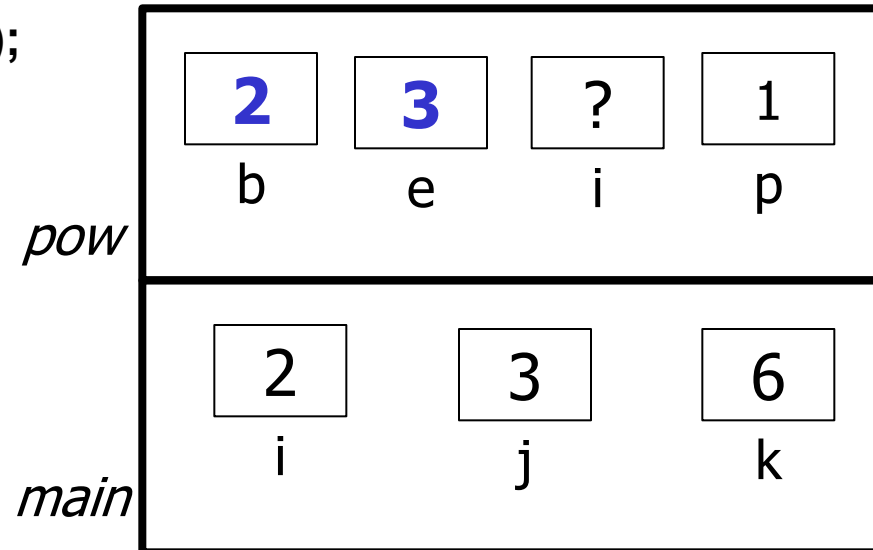
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k = power( i, j );  
    return 0;  
}
```



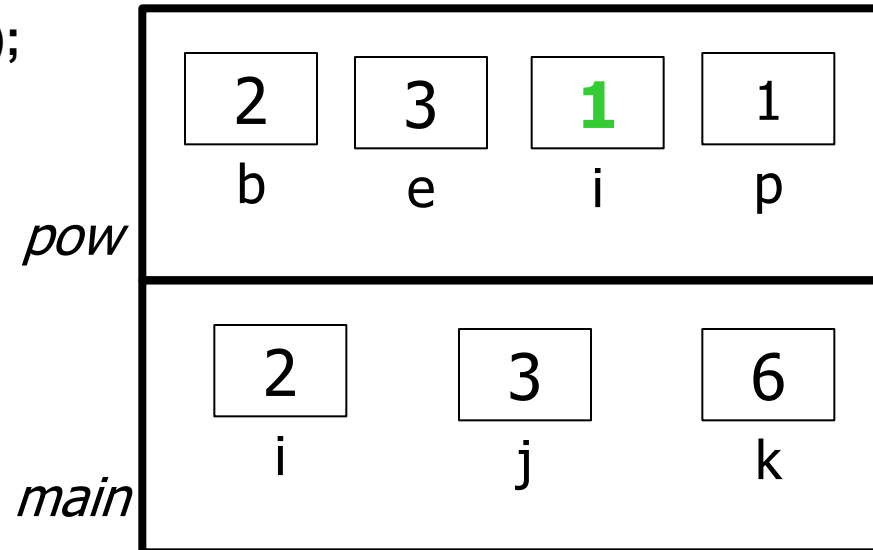
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k = power( i, j );  
    return 0;  
}
```



Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```

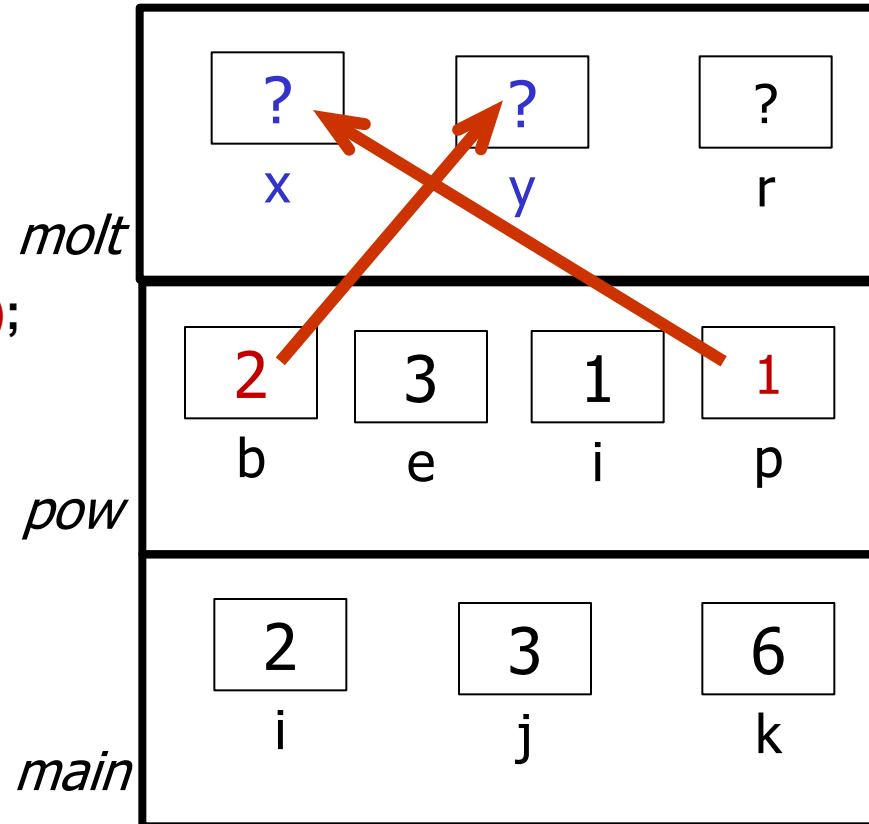


Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}
```

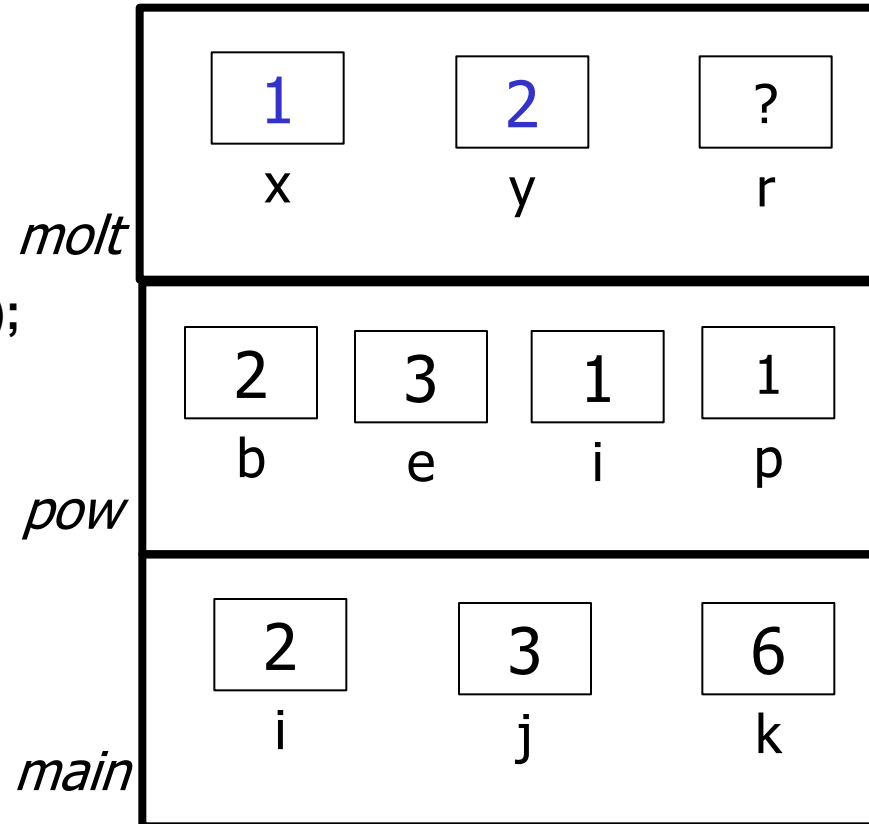
```
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}
```

```
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



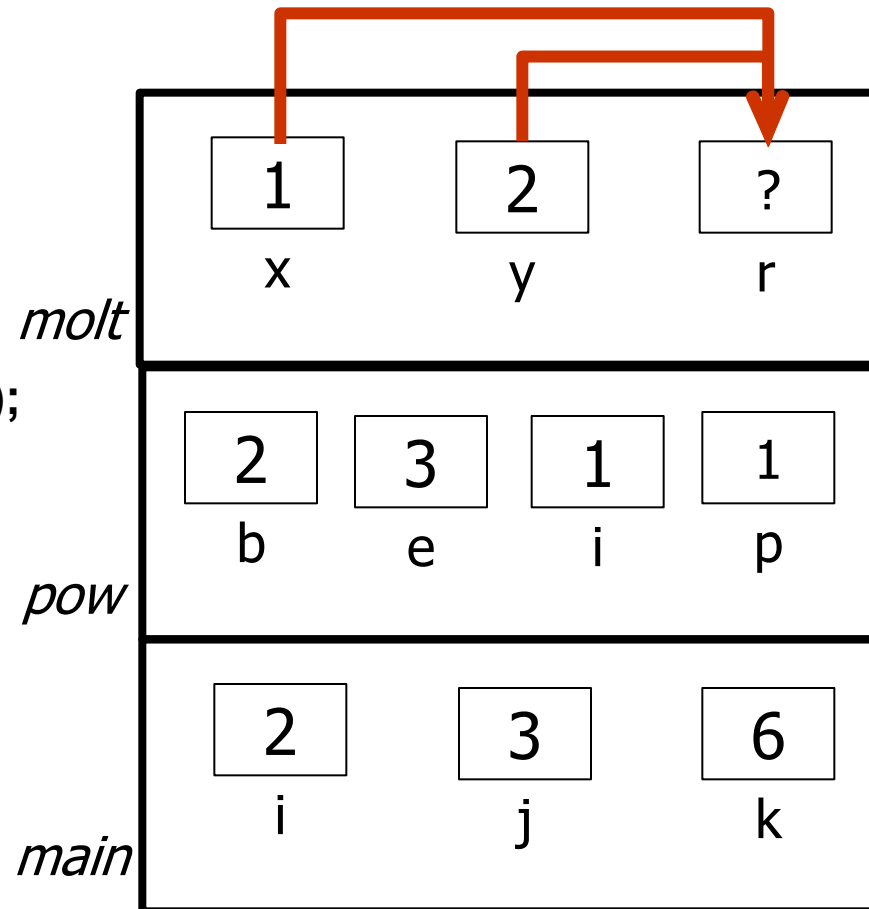
Esempio di codice

```
int multiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = multiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=multiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```

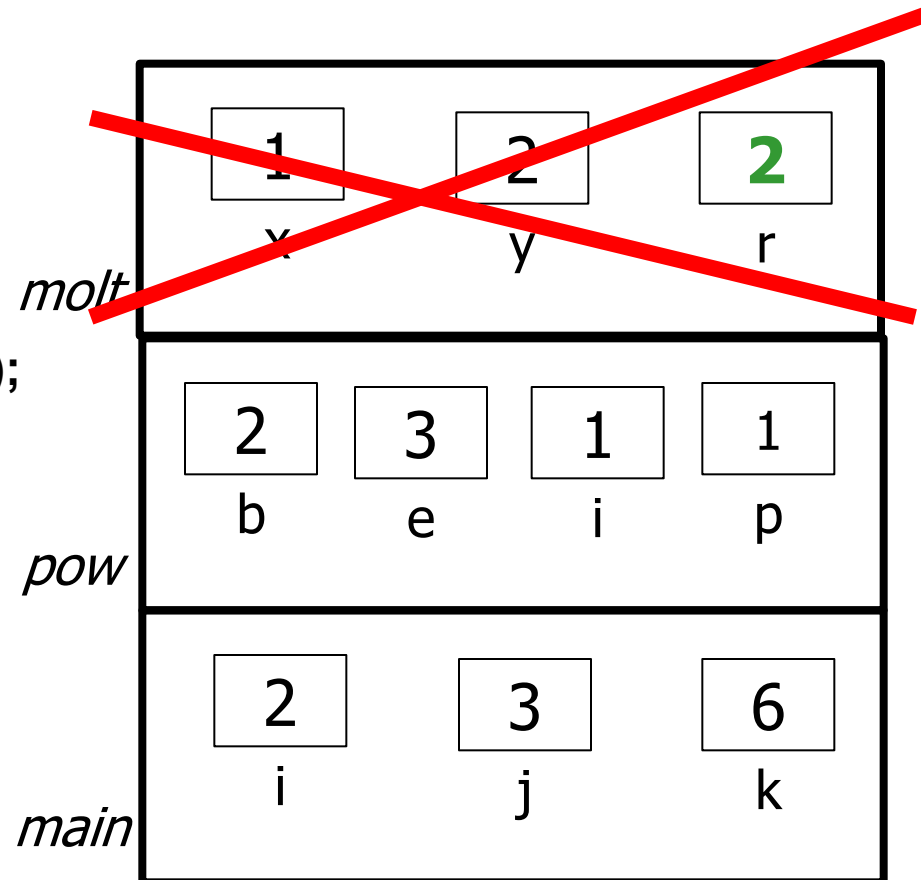


Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}
```

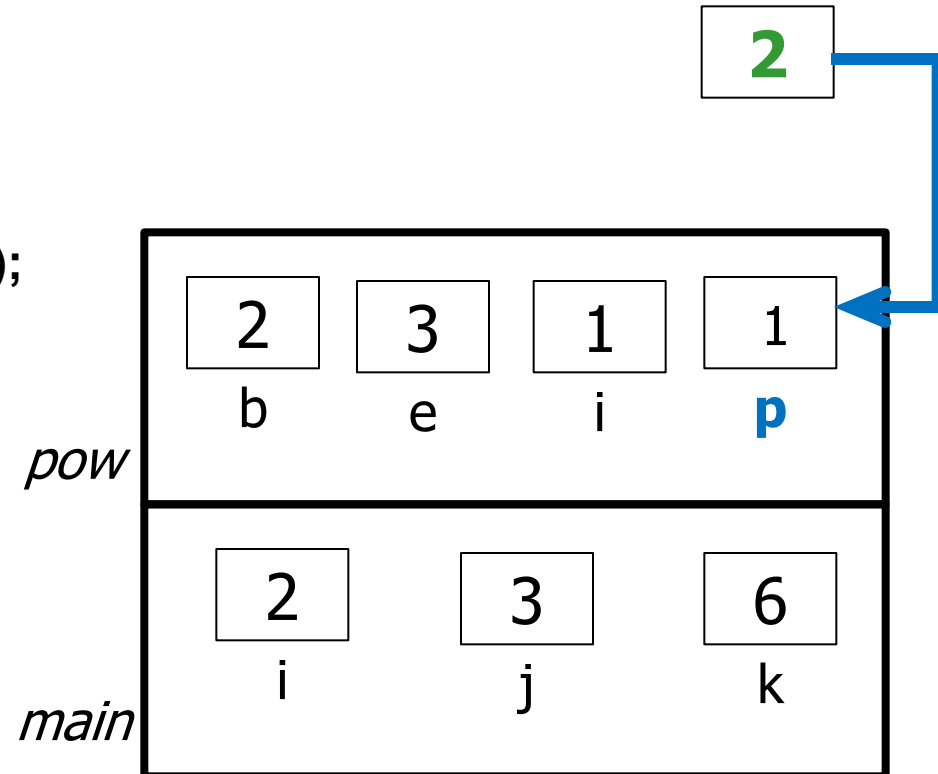
```
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}
```

```
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



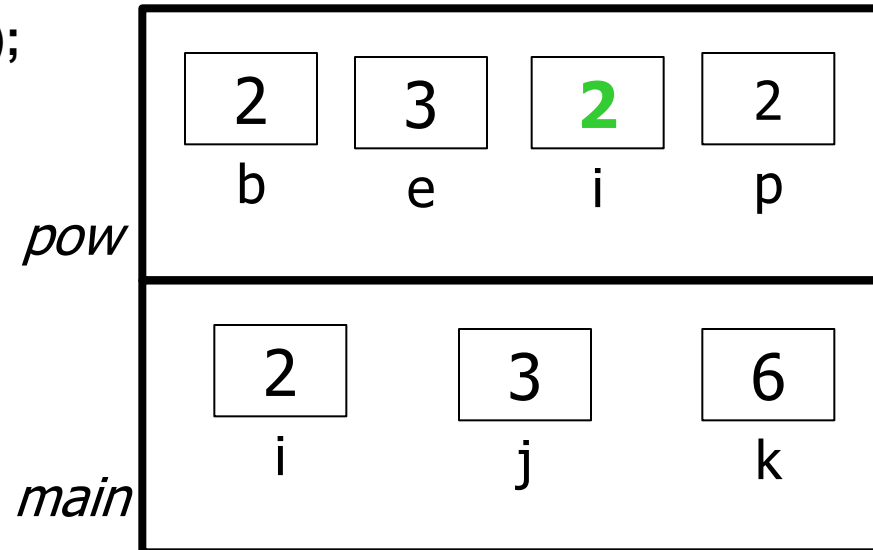
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```

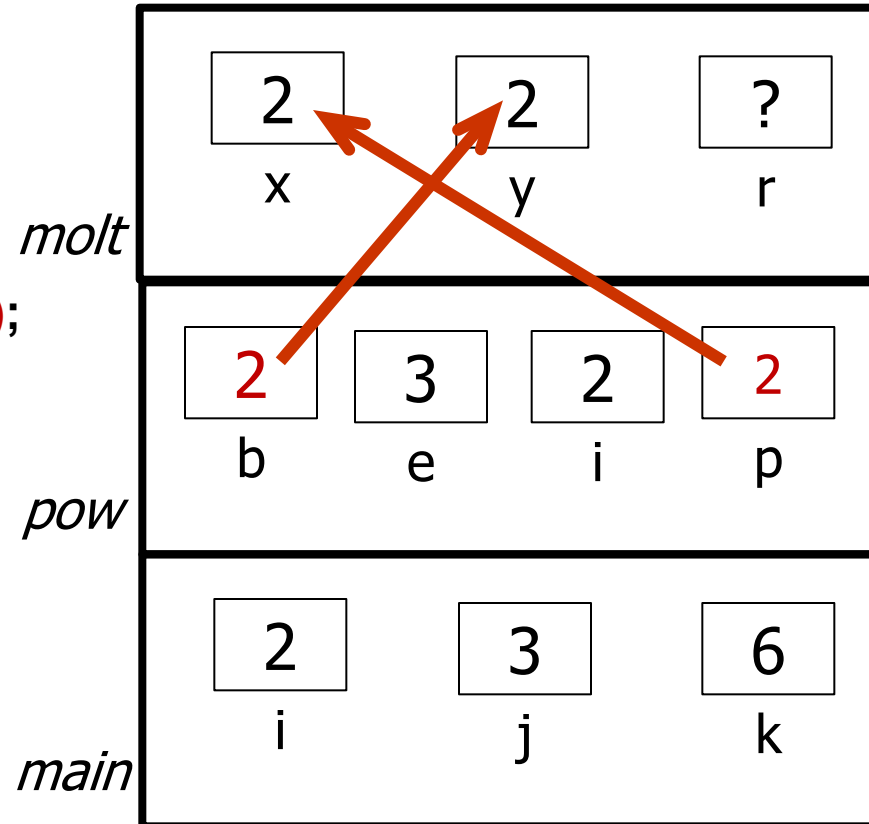


Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}
```

```
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}
```

```
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```

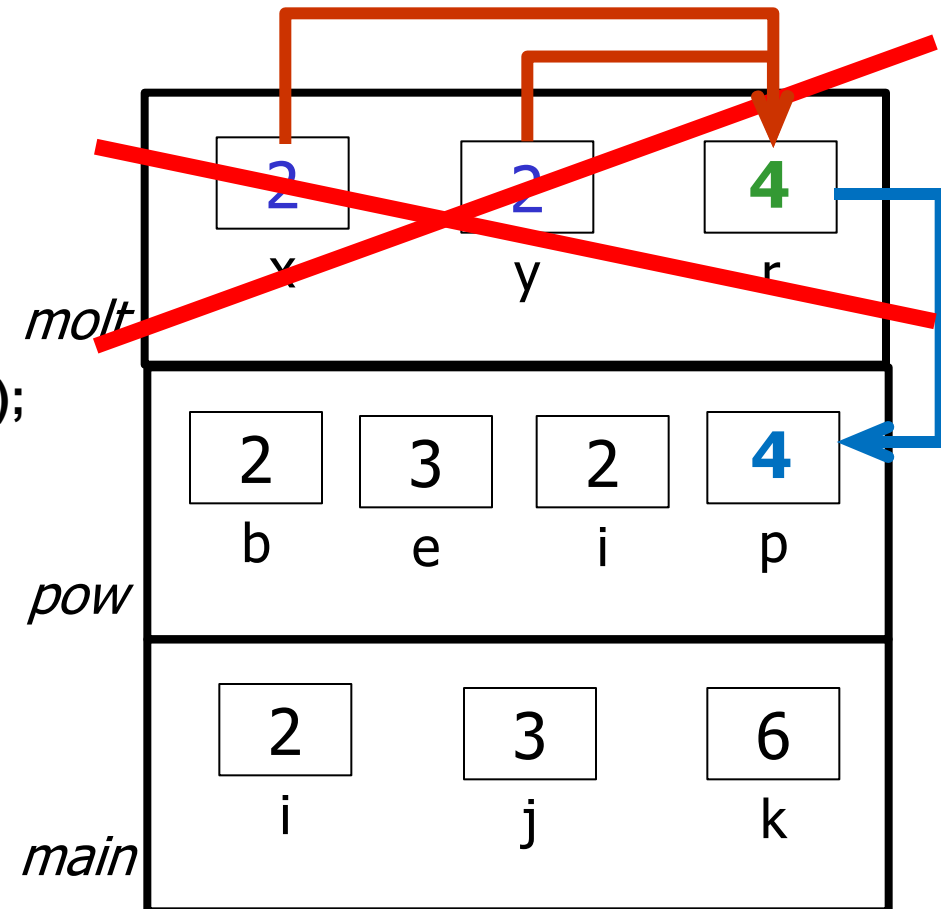


Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}
```

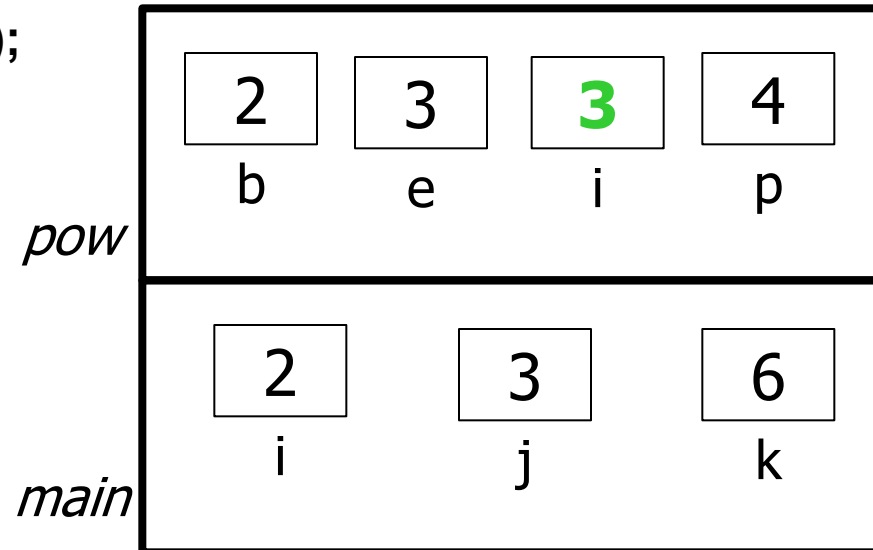
```
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}
```

```
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```

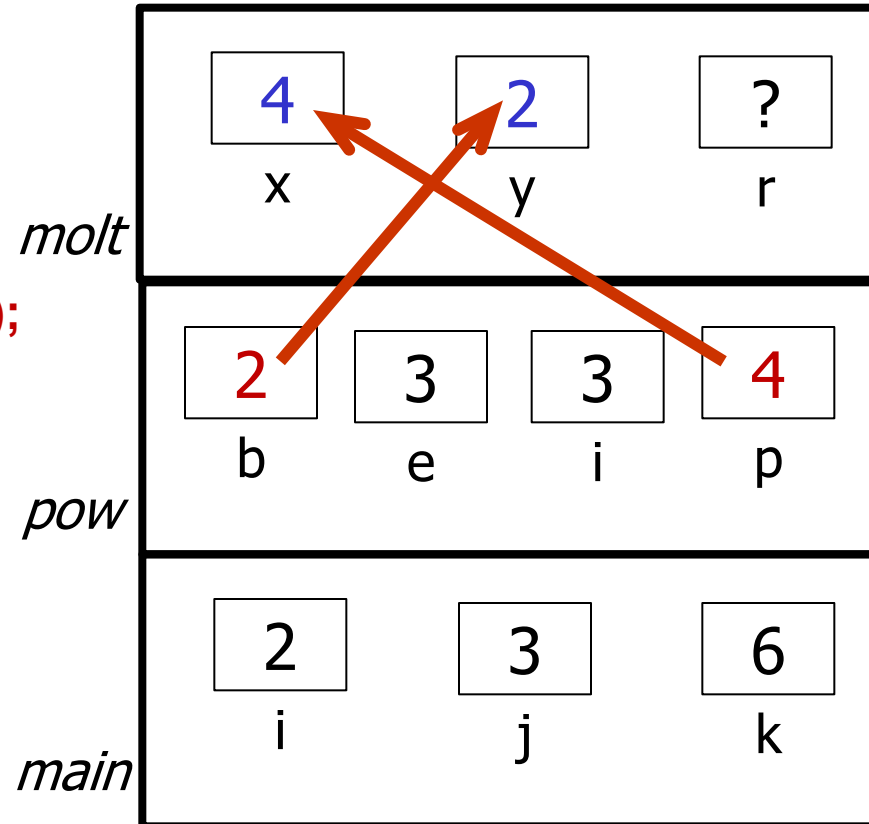


Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}
```

```
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}
```

```
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```

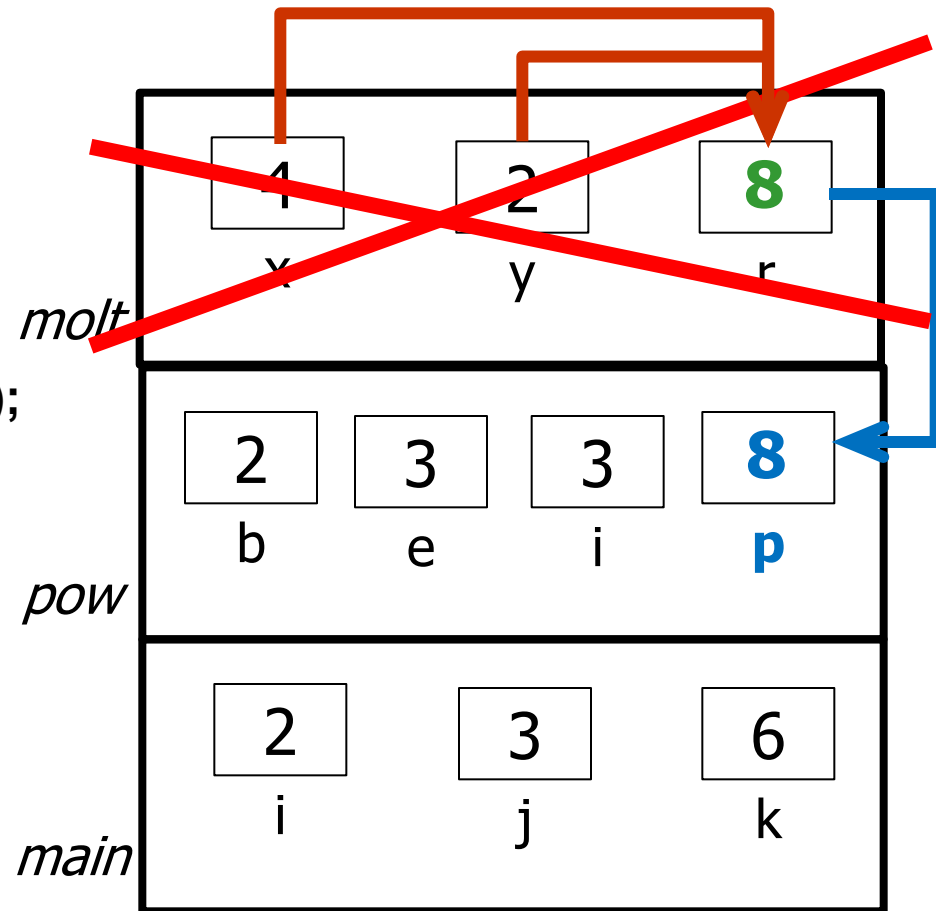


Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}
```

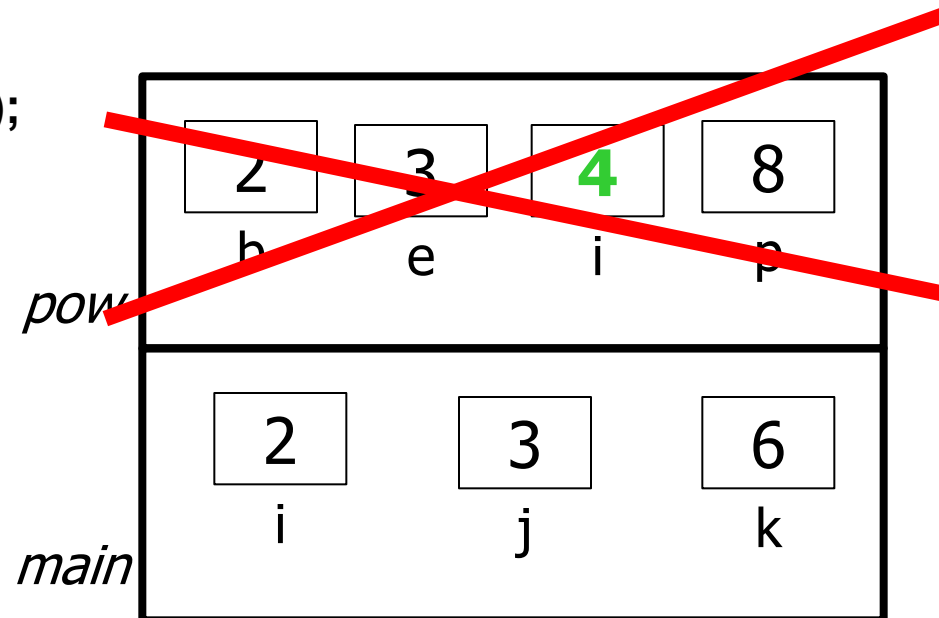
```
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}
```

```
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



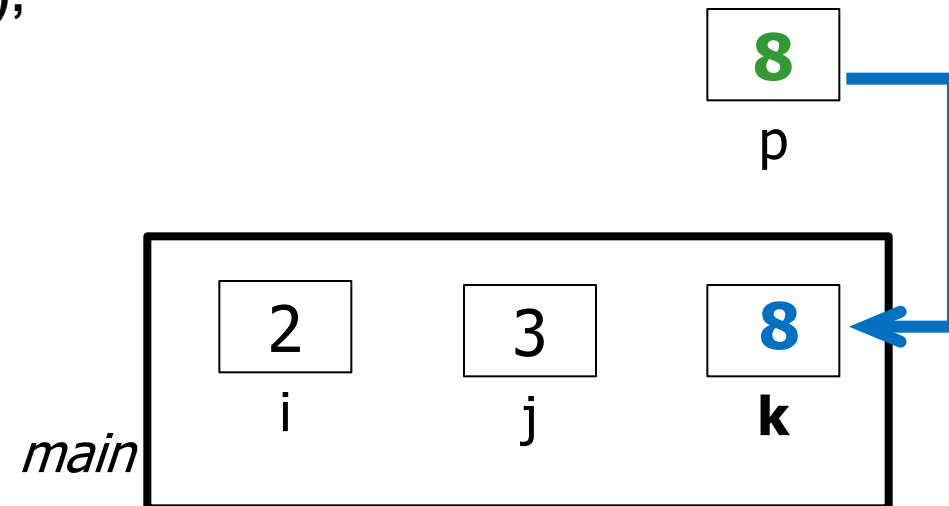
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



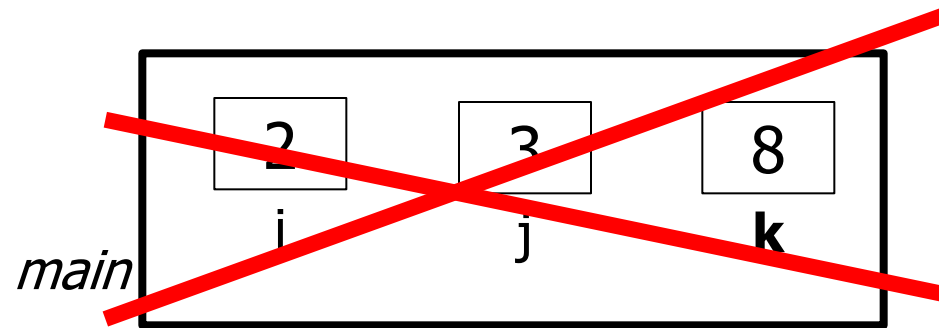
Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```



Esempio di codice

```
int moltiplica( int x, int y ) {  
    int r;  
    r = x * y;  
    return r;  
}  
  
int power( int b, int e ) {  
    int i, p=1;  
    for( i=1 ; i<=e ; i++ )  
        p = moltiplica( p, b );  
    return p;  
}  
  
int main() {  
    int i=2,j=3,k;  
    k=moltiplica(i,j);  
    k=power(i,j);  
    return 0;  
}
```





Esempio

Radice quad. intera, cioè il max int il cui quadrato è \leq par.

```
int RadiceIntera (int par) {
    int cont = 0;
    while (cont * cont <= par)
        cont++;
    return (cont - 1);
    /* NB: se si arriva qui cont * cont è > par */
}
```

Restituisco il valore dell'espressione **cont-1**

NB: se **par < 0** il risultato è **-1**

segnala un uso improprio della funzione (la radice sarebbe immaginaria)



Esempi di chiamate

```
x = sin(y) - cos(PI_GRECO - alfa);
```

```
/* PI_GRECO indica il valore costante  $\pi$  */
```

```
x = cos( atan(y) - beta );
```

```
RisultatoDiGestione = FatturatoTotale( ArchivioFatture ) -  
SommaCosti( ArchivioCosti );
```

```
OrdAlf = Precede( nome1, nome2 );
```



Esempio numeri perfetti

Scrivere un programma che chiede all'utente di inserire un numero positivo n (nel caso in cui il numero non è positivo ripetere inserimento) e verifica se questo è perfetto

Se n non è perfetto dice se è abbondante o difettivo e richiede un secondo numero intero positivo m e controlla se n ed m sono amici. Si stampa a schermo il risultato di questo controllo.

Un numero è perfetto se corrisponde alla somma dei suoi divisori, escluso se stesso (es. 6 è perfetto $1 + 2 + 3 = 6$)

Un numero è abbondante se è $>$ della somma dei suoi divisori (es 15 è abbondante $1 + 3 + 5 < 15$), altrimenti difettivo (es 12 è difettivo, $1+2+3+4+6 > 12$)

Due numeri a, b sono amici (o amicabili) se la somma dei divisori di a è uguale a b e viceversa (es 220 e 284)

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

```
int sommaDivisori(int);  
int controllaSePerfetto(int);  
int controllaSePerfettoRef(int, int*);  
int controllaSeAmici(int, int);  
int leggiInteroPositivo();
```

```
int main()  
{  
    int n, m, amici, perf, abb;  
  
    n = leggiInteroPositivo();  
    perf = controllaSePerfettoRef(n, &abb); // perf = controllaSePerfetto(n);  
  
    if(perf == 1)  
        printf("\n%d e' un numero perfetto", n);  
    else  
    {  
        if(abb > 0)  
            printf("\n%d e' un numero abbondante", n);  
        else  
            printf("\n%d e' un numero difettivo", n);  
  
        m = leggiInteroPositivo();  
  
        amici = controllaSeAmici(n, m);  
  
        if(amici == 1)  
            printf("\n%d e %d sono amici", n, m);  
        else  
            printf("\n%d e %d NON sono amici", n, m);  
    }  
}
```

```
int sommaDivisori(int x)
{
    int s=0, i;

    for(i = 1; i<x-1; i++)
        if(x%i == 0)
            s+= i;
    return s;
}
```

```
int controllaSePerfetto(int x)
{
    int flag, s;
    s = sommaDivisori(x);
    if(s == x)
        flag = 0;
    else
        if (x > s)
            flag = 1; // numero abbondante
        else
            flag = -1; // numero difettivo

    return flag;
}
```

```
int controllaSePerfettoRef(int x, int *abb)
{
    int flag = 0, s;
    s = sommaDivisori(x);
    if(s == x)
        flag = 1;
    else
        if (x > s)
            *abb = 1; // numero abbondante
        else
            *abb = -1; // numero difettivo
    return flag;
}
```

```
int controllaSeAmici(int a, int b)
{
    int flag = 0, sA = 0, sB = 0;

    sA = sommaDivisori(a);
    sB = sommaDivisori(b);

    if(a == sB && b == sA)
        flag = 1;

    return flag;
}
```

```
int leggiInteroPositivo()
{
    int x;
    float f;
    do
    {
        printf("\ninserire intero positivo: ");
        scanf("%f", &f);
        // x = (int) f; // casting esplicito
        x = f; // casting implicito dato dall'assegnamento
    } while (x < 0 || x != f);
    return x;
}
```



Dichiarazione delle Funzioni

È utile e raccomandato (standard ANSI C) riportare all'inizio del programma la “testata” (header) della funzione: prototipo

- Nella parte dichiarativa globale o nel programma che chiama la funzione

Serve a facilitare il lavoro del compilatore

- In particolare del parser: analisi sintattica

In pratica il prototipo si aggiunge alle dichiarazioni di variabili e costanti

//esempi

```
int sommaDivisori (int) ;  
int controllaSePerfetto (int) ;  
int controllaSePerfettoRef (int, int*) ;  
int controllaSeAmici (int, int) ;  
int leggiInteroPositivo () ;
```




Funzioni e Array

Ancora sui Parametri



Variabili automatiche:

- Sono quelle dichiarate nelle funzioni (inclusi i parametri) e nei blocchi di istruzione
- Sono **create** quando il flusso di esecuzione "entra" nel loro **ambito di visibilità**
- Sono **distrutte** all'uscita da tale ambito
- Sono allocate di volta in volta in celle differenti
- Non conservano i valori prodotti da precedenti esecuzioni della funzione o del blocco

```
int main()
{
    int n, m, amici, perf, abb;

    n = leggiInteroPositivo();
    [...]
    return 1;
}
```

```
int leggiInteroPositivo()
{
    int x;
    float f;
    do
    {
        printf("\ninserire intero positivo: ");
        scanf("%f", &f);
        x = f; // casting implicito dato dall'assegnamento
    } while (x < 0 || x != f);
    return x;
}
```

Lo scope delle variabili **x** ed **f**
corrisponde alla funzione
leggiInteroPositivo()



Tipo dei parametri e di return

Tipo dei parametri

- Può essere **built-in** o **user-defined**
- I **valori dei parametri attuali non sono modificabili tra le istruzioni del chiamante**
 - I sottoprogrammi lavorano su copie dei parametri attuali
 - **Fanno eccezione (apparentemente) gli array**
 - La ragione è che si passano dei "riferimenti alla prima cella"

Tipo di return

- Può essere **built-in** o **user-defined**
- **NON può essere un array**
 - **Ma può essere una struct** (anche se questa contiene array!!!)
- **PUÒ** essere un puntatore a qualsiasi tipo
 - Ci torneremo studiando i puntatori e la memoria dinamica



Tipo dei parametri e di return

Tipo dei parametri

- Può essere **built-in** o **user-defined**
- I **valori dei parametri attuali non sono modificabili tra le istruzioni del chiamante**
 - I sottoprogrammi lavorano su copie dei parametri attuali
 - **Fanno eccezione (apparentemente) gli array**
 - La ragione è che si passano dei "riferimenti alla prima cella"

Tipo di return

- Può essere **built-in** o **user-defined**
- **NON può essere un array** ← Tra poco vedremo perché
 - **Ma può essere una struct** (anche se questa contiene array!!!)
- **PUÒ** essere un puntatore a qualsiasi tipo
 - Ci torneremo studiando i puntatori e la memoria dinamica

Passaggio per Riferimento (a.k.a. passaggio per indirizzo)

... usiamo i puntatori insomma!



Modifica dei parametri attuali

Tutti i parametri, in C, sono passati per **COPIA**

Le variabili passate come parametri a una funzione, quindi, se alterate nell'ambiente della funzione, **non cambiano valore** nell'ambiente del chiamante (parametri passati *per valore*)

Esistono modi di modificare i parametri attuali??



Scambio dei valori di due interi

```
void swap (int p, int q) {  
    int temp; /* var. locale */  
    temp = p;  
    p = q;  
    q = temp;  
}
```

Chiamata: **swap** (**i**, **j**);

ERRORE: Alla fine **i** e **j** sono immutate

L'invocazione comporta i
seguenti assegnamenti
(passaggio dei parametri)

```
p = i;  
q = j;
```



Scambio dei valori di due interi

```
void swap (int p, int q) {  
    int temp; /* var. locale */  
    temp = p;  
    p = q;  
    q = temp;  
}
```

Chiamata: `swap (i, j);`

ERRORE: Alla fine `i` e `j` sono immutate perché vengono modificate nel record di attivazione di `swap` ma non nel record di attivazione del chiamante



Scambio dei valori di due interi

```
void swap (int p, int q) {  
    int temp; /* var. locale */  
    temp = p;  
    p = q;  
    q = temp;  
}
```

Chiamata: `swap (i, j);`



Scambio dei valori di due interi

```
void swapR(int *p, int *q) {  
    int temp; /* var. locale */  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

Chiamata: `swapR(&i, &j);`

L'invocazione comporta i seguenti assegnamenti (passaggio dei parametri)

`p = &i;`

`q = &j;`



Scambio dei valori di due interi

```
void swapR(int *p, int *q) {  
    int temp; /* var. locale */  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

Chiamata: `swapR(&i, &j);`



Passaggio per riferimento: Modifica dei parametri attuali

Tutti i parametri, in C, sono passati per *COPIA*



Passaggio per riferimento: Modifica dei parametri attuali

Tutti i parametri, in C, sono passati per **COPIA**

Se si vuole che una funzione agisca sulle variabili dell'ambiente del chiamante, occorre passare **l'indirizzo** di tali variabili (parametri passati *per locazione* o *per indirizzo*) \Rightarrow uso dei **puntatori**

In questo modo possiamo **modificare i parametri attuali**



Puntatori e funzioni

```
void fiddle (int x, int * y) {
    printf("Begin fiddle: x=%d, y=%d\n", x, *y);
    x = x+1;
    *y = *y+1;
    printf("End fiddle:    x=%d, y=%d\n", x, *y);
}

int main ( ) {
    int i = 0, j = 0;
    printf("Begin main:    i=%d, j=%d\n", i, j);
    fiddle(i, &j);
    printf("Returned, ending main: i=%d, j=%d\n", i, j);
}
```

N.B.: si estrae l'indirizzo durante l'invocazione usando «&» in `fiddle(i, &j);`



Traccia di esecuzione

Begin main: i=0, j=0

Begin fiddle: x=0, y=0

End fiddle: x=1, y=1

Returned, Ending main: i=0, j=1

All'uscita da fiddle il valore di *i* è *rimasto lo stesso*,
mentre quello di *j* (passato tramite puntatore) è *cambiato*

Se vogliamo poter modificare il valore di una variabile
in modo che *resti modificato* all'uscita dalla funzione, occorre

- Passare la variabile tramite un puntatore
- Oppure restituire e sovrascrivere durante l'invocazione il valore della variabile restituita



Modifica dei parametri attuali

Com'è possibile modificare – all'interno di una funzione -- una variabile nel main?

Nella **definizione** il parametro formale deve essere di tipo *puntatore* al tipo del parametro attuale di cui si vuole la modifica

Nella **chiamata** si deve passare l'indirizzo (usando $\&$) del parametro attuale da modificare

NOTA: il parametro attuale in questo caso deve essere una **variabile** (o una espressione che, valutata, restituisca un puntatore); **NON** una generica **espressione**.

Nel corpo della funzione si usa l'operatore * di *dereferenziazione* per riferirsi al parametro

Vantaggi del Passaggio per Riferimento



A cosa serve un passaggio per riferimento?

Permette di «restituire» più output

```
int controllaSePerfettoRef(int x, int *abb);
```

Questa funzione restituisce sia se il numero è perfetto sia (nel caso in cui non lo sia) se è abbondante o difettivo.

Come funziona?

1. **abb** conterrà l'indirizzo di una cella del main (è un puntatore)

```
int abbondante;
```

```
controllaSePerfettoRef(7, &abbondante);
```

2. All'interno di `controllaSePerfettoRef` scrivo il secondo valore da restituire direttamente nel record di attivazione nel main, tramite **abb**

```
*abb = 1; // risultato da inserire
```

3. Terminata l'esecuzione della funzione, nel main accedo alla variabile `abbondante` che sarà stata così popolata



Passaggio per riferimento

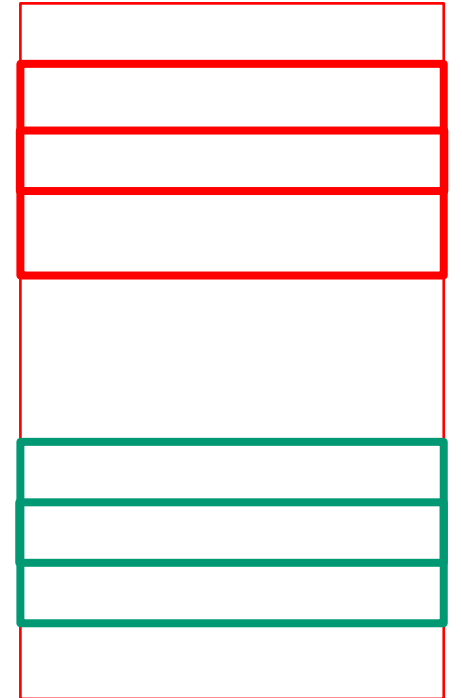
```
int controllaSePerfettoRef(int x, int *abb);
```

Record
attivazione
funzione

***abb**

Record
attivazione
main

res

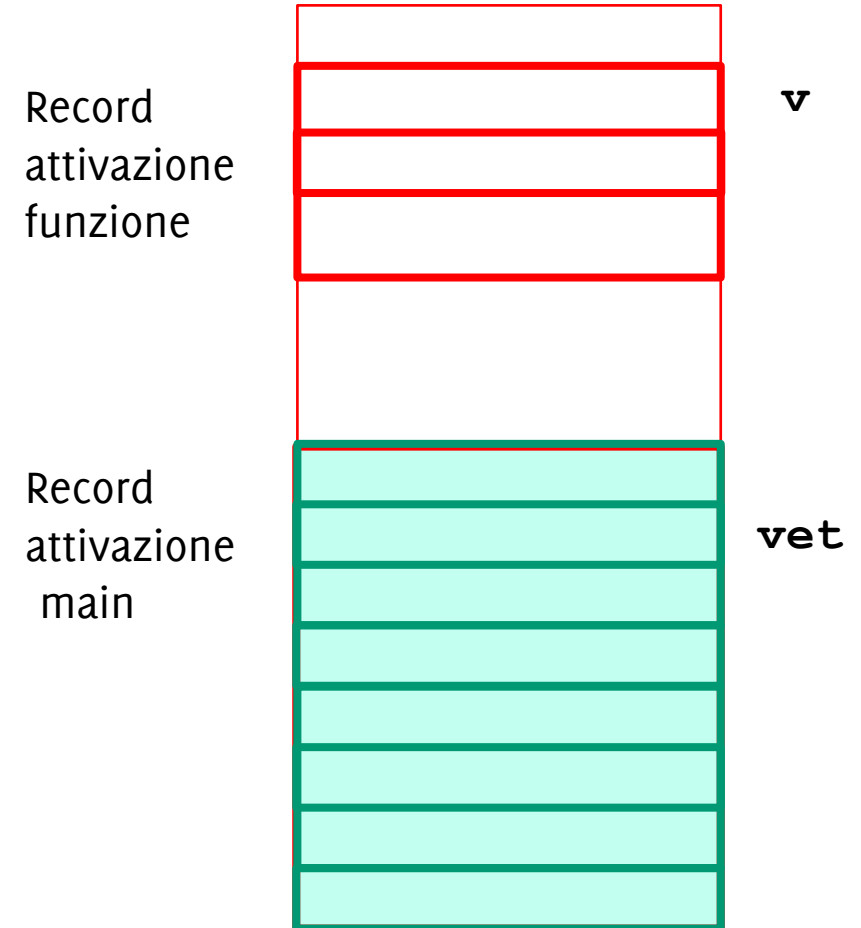


A cosa serve un passaggio per riferimento?

Permette di evitare di copiare grandi quantità di dati (ad esempio array)

- Tengo **una sola copia dell'array nel main**
- Opero su questa mediante puntatori all'interno della funzione invocata

Oss: si tratta sempre di un'operazione di copia ... ma di indirizzi! Non dei valori dell'array





Intercambiabilità di procedure e funzioni

Funzione:

```
int f (int par1) {  
    ... (calcola valore  
di variabile "risultato") ...  
    return (risultato);  
}
```

chiamata:

```
y = f (x) ;
```

Procedura:

```
void f (int par1, int * par2)  
{  
    ... (calcola valore di  
variabile "risultato")  
    ...  
    *par2 = risultato;  
}
```

chiamata:

```
f (x, &y) ;
```

a questo punto a y è stato assegnato il
valore

Funzioni ed Array



Parametro attuale di tipo array

- Viene **passato l'indirizzo** di base dell'array
 - È l'indirizzo della prima cella del primo elemento
- **Gli elementi dell'array NON sono copiati** nel parametro formale, rimangono nel record di attivazione del main e durante l'esecuzione della funzione sono accessibili tramite puntatori
- Non viene allocato nel record di attivazione della funzione un array per i parametri formali (ma è possibile dichiararne altri)



strcmp(s1, s2)

E se non ci fosse?

- controlliamo un carattere alla volta
- interrompiamo il controllo appena sono diverse

```
int mystrcmp(char s1[ ], char s2[ ])
{
    int i = 0;
    while ( s1[i] == s2[i] && s1[i] != '\0' )
        ++i;
    return s1[i] - s2[i];
}
```

```
#include<stdio.h>
#include<string.h>
#define N 30

int mystrcmp(char [], char []);

int main()
{
    char s[N] = "ciao mamma";
    char t[N] = "ciao";

    printf("\nstrcmp(%s, %s) = %d", s, t, strcmp(s,t));
    printf("\nmystrcmp(%s, %s) = %d", s, t, mystrcmp(s,t));

    return 0;
}

int mystrcmp(char s1[], char s2[])
{
    int i = 0;
    while(s1[i] == s2[i] && s1[i] != '\0')
        i++;

    return s1[i] - s2[i];
}
```

```

#include<stdio.h>
#include<string.h>
#define N 30

int mystrcmp(char [], char []);

int main()
{
    char s[N] = "ciao mamma";
    char t[N] = "ciao";

    printf("\nstrcmp(%s, %s) = %d", s, t, strcmp(s,t));
    printf("\nmystrcmp(%s, %s) = %d", s, t, mystrcmp(s,t));

    return 0;
}

int mystrcmp(char s1[], char s2[])
{
    int i = 0;
    while(s1[i] == s2[i] && s1[i] != '\0')
        i++;

    return s1[i] - s2[i];
}

```

All'invocazione della funzione `mystrcmp` passo gli array che sono degli indirizzi. Quindi sto copiando l'indirizzo di `s` in `s1` e l'indirizzo di `t` in `s2` con l'invocazione

Il codice di `mystrcmp` contiene istruzioni identiche a quelle che avremmo messo nel `main`. I parametri formali `s1` e `s2` sono puntatori e li uso per scorrere gli array.

```

#include<stdio.h>
#include<string.h>
#define N 30

int mystrcmp(char [], char []);

int main()
{
    char s[N] = "ciao mamma";
    char t[N] = "ciao";

    printf("\nstrcmp(%s, %s) = %d", s, t, strcmp(s,t));
    printf("\nmystrcmp(%s, %s) = %d", s, t, mystrcmp(s,t));


    return 0;
}

int mystrcmp(char s1[], char s2[])
{
    int i = 0;
    while(s1[i] == s2[i] && s1[i] != '\0')
        i++;

    return s1[i] - s2[i];
}

```

Nonostante la sintassi
 (**char** s1[], **char** s2[])
 sono dei puntatori, non degli array!
 Altrimenti sarebbero costanti e non
 potrebbero essere modificati
 nell'invocazione (qui comunque non
 vengono modificati)



"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018_Informatica_A\Lez12\stringhe.exe"

```
strcmp(ciao mamma, ciao mamma) = 0  
mystrcmp(ciao mamma, ciao mamma) = 0  
strlen(ciao mamma) = 10  
mystrlen_ref(ciao mamma) = 10
```

```
Process returned 0 (0x0)    execution time : 0.045 s  
Press any key to continue.
```

"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018_Informatica_A\Lez12\stringhe.exe"

```
strcmp(ciao, ciao mamma) = -1  
mystrcmp(ciao, ciao mamma) = -32  
strlen(ciao) = 4  
mystrlen_ref(ciao) = 4
```

```
Process returned 0 (0x0)    execution time : 0.083 s  
Press any key to continue.
```

```
#include<stdio.h>
#include<string.h>
#define N 30

int mystrcmp(char [], char []);

int main()
{
    char s[N] = "ciao mamma";
    char t[N] = "ciao";

    printf("\nstrcmp(%s, %s) = %d", s, t, strcmp(s,t));
    printf("\nmystrcmp(%s, %s) = %d", s, t, mystrcmp(s,t));

    return 0;
}

int mystrcmp(char s1[], char s2[])
{
    int i = 0;
    while(s1[i] == s2[i] && s1[i] != '\0')
        i++;

    return s1[i] - s2[i];
}
```

```
int mystrlen(char *);  
int mystrlen_ref(char *);
```

```
int mystrlen(char *s)  
{  
    int i = 0;  
    while (s[i] != '\0')  
        i++;  
    return i;  
}
```

```
int mystrlen_ref(char *s)  
{  
    char *q = s;  
    while (*(q++) != '\0');  
    return q-s-1;  
}
```




E se modificassi la stringa in `mystrlen`?

```
int mystrlen(char *);  
int mystrlen_ref(char *);
```

```
int mystrlen(char *s)  
{  
    int i = 0;  
    while(s[i] != '\0')  
        i++;  
    return i;  
}
```

```
int mystrlen_ref(char *s)  
{  
    char *q = s;  
    while(*(q++) != '\0');  
    return q-s-1;  
}
```



Altro esempio: somma dei primi n elementi di un array di double

```
/* n-1 rappresenta la posizione occupata dell'ultimo
elemento dell'array che contiene un valore significativo (la
" coda " da n+1 a DIMENSIONE non è considerata significativa)
*/
double sum(double a[], int n) {
    int i;
    double ris = 0.0;
    for ( i = 0; i < n; i++ )
        ris = ris + a[i];
    return ris;
}
```



Rivediamo la copia di stringhe

Copia con sovrascrittura

- Copia di un carattere alla volta fino a '`\0`' (incluso)
- **ATTENZIONE:** la memoria per `s1` deve già essere stata esplicitamente allocata [array di dimensioni sufficienti]

```
void strcpy( char s1[], char s2[] ) {
    int i = 0;
    if ( s1 != NULL && s2 != NULL ) {
        while ( s2[i] != '\0' ) {
            s1[i] = s2[i];
            i++;
        }
        s1[i] = '\0';
    }
}
```

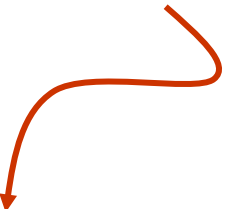
```
void strcpy2( char s1[ ], char s2[ ] ) {
    while( s1 && s2 && ( *(s1++)=*(s2++) ) != '\0' )
        ;
}

/* ...una versione assai più... "acrobatica"! */
/* "Bella" però è codice "poco leggibile"! */
```



Ordinamento di array – bubblesort –

```
void bubblesort (int param[], int size) {  
    for (i=0; i<size-1; i++)  
        porta in fondo il massimo in i..size-1  
}
```



```
for ( j=0 ; j<size-1-i ; j++ )  
    if ( param[j] > param[j+1] )  
        swap( &param[j], &param[j+1] );
```

Attenzione: si usa la funzione **swap** passandole gli indirizzi degli elementi da scambiare



Attenzione: tramite i puntatori, durante l'esecuzione della funzione è possibile modificare la variabile nel record di attivazione del main!

```
modificaArray(v, 4);
```

Parametro formale di tipo array

- Puntatore al primo elemento. Ci sono diverse opzioni per il prototipo

```
void modificaArray(int vett[], int dim);
```

```
void modificaArray(int *vet, int dim);
```

```
void modificaArray(int vett[N], int dim);
```

```
#include <stdio.h>

void modificaInt(int i);

void modificaArray(int vett[],int dim);

int main() {
    int a = 0, i, v[4] = {0,1,2,3};
    printf("Valori prima\n");
    printf("  a = %d\n", a);
    for (i=0; i<4; i++)
        printf("  v[%d] = %d\n", i, v[i]);
    printf("\nChiamo le funzioni...\n");
    modificaInt(a);
    modificaArray(v,4);
    printf("\nValori dopo:\n");
    printf("  a = %d\n", a);
    for (i=0; i<4; i++)
        printf("  v[%d] = %d\n", i, v[i]);
}
```

```
void modificaInt(int i) {
    i = i + 100;
}

void modificaArray(int vett[],int dim) {
    int i;
    for (i=0; i<dim; i++)
        vett[i] = vett[i] + 100;
}
```

Cosa stampa?



«Restituire» array?

Come si fa' a fare in modo che una funzione operi su un vettore per poi riportare al chiamante – spesso il main – un secondo array?

Ad esempio:

// scrivere una funzione che prende in ingresso un vettore e riporta al chiamante un vettore contenente i soli numeri pari

- Non è possibile fare return di un array!
- Gli array sono solo passati per indirizzo (riferimento)



«Restituire» array?

```
#include<stdio.h>
#define N 20
void pari(int*, int, int*, int*);
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;
// len rappresenta le dimensioni effettive di vet

pari(vet, len, vet_pari, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);

return 0;
}

void pari(int x[], int len_x, int y[], int* len_y)
{
int i,j=0;
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }

*len_y = j;
}
```




«Restituire» array?

```
#include<stdio.h>
#define N 20
void pari(int*, int, int*, int*);
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;
// len rappresenta le dimensioni effettive di vet

pari(vet, len, vet_pari, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);

return 0;
}

void pari(int x[], int len_x, int y[], int* len_y)
{
int i,j=0;
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }
*len_y = j;
}
```



«Restituire» array?

"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2021_Informatica_A_Boracchi\Lez12\passaggioArray.exe"

```
vet: [1,2,3,4,5,6,7,8,9,10,];
```

```
vet_pari: [2,4,6,8,10,];
```

```
Process returned 0 (0x0)    execution time : 0.077 s
```

```
Press any key to continue.
```



«Restituire» array?

```
#include<stdio.h>
#define N 20
void pari(int*, int, int*, int*);
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;
// len rappresenta le dimensioni effettive di vet

pari(vet, len, vet_pari, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);

return 0;
}
```

Quando passo un vettore di interi è necessario passare anche la sua lunghezza (**len**), altrimenti non ho modo di sapere quali elementi sono significativi

```
void pari(int x[], int len_x, int y[], int* len_y)
{
int i,j=0;
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }
*len_y = j;
}
```

«Restituire» array?

```
#include<stdio.h>
#define N 20
void pari(int*, int, int*, int*);
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;
// len rappresenta le dimensioni effettive di vet

pari(vet, len, vet_pari, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);

return 0;
}
```

vet_pari è il vettore che raccoglierà il risultato delle istruzioni eseguite nella funzione. Deve essere dichiarato nel main, di modo che il suo spazio sia allocato in questo record di attivazione dove deve essere accessibile

```
void pari(int x[], int len_x, int y[], int* len_y)
{
int i,j=0;
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }
*len_y = j;
}
```

«Restituire» array?

```
#include<stdio.h>
#define N 20
void pari(int*, int, int*, int*);
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;
// len rappresenta le dimensioni effettive di vet

pari(vet, len, vet_pari, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);

return 0;
}
```

`len_pari` è un intero che conterà le dimensioni effettive di `vet_pari`. Non c'è modo di conoscerle altrimenti nel main. Per `len_pari` avrei potuto anche usare return perché è una variabile intera

```
void pari(int x[], int len_x, int y[], int* len_y)
{
    int i,j=0;
    for(i = 0; i < len_x; i++)
        if(x[i] % 2 == 0)
        {
            y[j] = x[i];
            j++;
        }
    *len_y = j;
}
```



E se facessi con return ...

```
#include<stdio.h>
#define N 20
int* pariSbagliata(int*, int, int*);

int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;

vet_pari = pariSbagliata(vet, len, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);
return 0;
}
```

```
int* pariSbagliata(int x[], int len_x, int* len_y)
{
int i,j=0;
int y[N];
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }
*len_y = j;
return y;
}
```



E se facessi con return ...

```
#include<stdio.h>
#define N 20
int* pariSbagliata(int*, int, int*);
```

```
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int vet_pari[N];
int len = 10, len_pari, i;
```

```
vet_pari = pariSbagliata(vet, len, &len_pari);
```

```
for(i=0; i < len; i++)
    print
```

```
for(i=0;
    print
```

```
return 0;
}
```

```
int* pariSbagliata(int x[], int len_x, int* len_y)
{
int i,j=0;
int y[N];
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }
```

error: assignment to expression with array type

`vet_pari = ...`

Non è possibile assegnare niente, nemmeno un indirizzo ad un array!
Va considerato come una costante!



E se usassi un puntatore?

```
#include<stdio.h>
#define N 20
int* pariSbagliata(int*, int, int*);
```

```
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int *p_vet_pari;
int len = 10, len_pari, i;

p_vet_pari = pariSbagliata(vet, len, &len_pari);

for(i=0; i< len; i++)
    printf("%d,", vet[i]);

for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);
return 0;
}
```

```
int* pariSbagliata(int x[], int len_x, int* len_y)
{
int i,j=0;
int y[N];
for(i = 0; i < len_x; i++)
    if(x[i] % 2 == 0)
    {
        y[j] = x[i];
        j++;
    }
*len_y = j;
return y;
}
```


E se usassi un puntatore?

```
#include<stdio.h>
#define N 20
int* pariSbagliata(int*, int, int*);
```

```
int main()
{
int vet[N]={1,2,3,4,5,6,7,8,9,10};
int *p_vet_pari;
int len = 10, len_pari, i;
```

```
p_vet_pari = pariSbagliata(vet, len, &len_pari);
```

```
for(i=0; i< len; i++)
    printf("%d,", vet[i]);
```

```
for(i=0; i< len_pari; i++)
    printf("%d,", vet_pari[i]);
```

```
return 0;
```

```
}
```

```
int* pariSbagliata(int x[], int len_x, int* len_y)
{
    int i,j=0;
    int y[N];
    for(i = 0; i < len_x; i++)
        if(x[i] % 2 == 0)
        {
            y[j] = x[i];
            j++;
        }
    *len_y = j;
    return y;
}
```

Sintatticamente è corretta, ma è concettualmente sbagliata e a runtime non funziona
Perché? **Dangling pointer!**



Dangling pointer!

"C:\Users\Giacomo\Dropbox (DEIB)\Didattica\2021_Informatica_A_Boracchi\Lez12\passaggioArray.exe"

```
vet: [1,2,3,4,5,6,7,8,9,10,];
```

```
vet_pari: [4223008,0,268501009,0,7341912,];
```

```
Process returned 0 (0x0)    execution time : 0.073 s
```

```
Press any key to continue.
```





Spiegazione

Stack all'invocazione di
`pariSbagliata`

Stack al termine di
`pariSbagliata`

dopo



All in all..

Non è possibile restituire un array anche se usassi un puntatore.

Infatti, se dichiarassi un array all'interno della funzione, questo verrebbe istanziato all'interno del record di attivazione della funzione, che viene de-allocato al momento in cui la funzione termina e torna al chiamante.

Darei luogo ad un **dangling pointer** con l'invocazione della funzione

Per dichiarare l'array all'interno di una funzione e renderlo disponibile al termine dell'invocazione, **bisogna ricorrere alla gestione della memoria dinamica.**

In questo modo lo spazio allocato rimane tale anche quando l'invocazione alla funzione si conclude.



Nota importante

Nel linguaggio parlato, si è soliti dire «*una funzione che prende in ingresso un vettore e restituisce un secondo vettore*»

A fare una corrispondenza 1:1 con il C la frase suona sbagliata, perché

- **La funzione non può fare return di un vettore**
- **La funzione dovrà prendere in input anche la lunghezza effettiva del vettore**

Tuttavia, i programmatori C non comunicano «dettando prototipi delle funzioni» e queste espressioni vanno interpretate in senso più generale per cui

- **Restituire -> rende disponibile dopo l'esecuzione, un vettore nel R.A chiamante**
- **Prendere in ingresso un vettore -> ... ed eventuali altre variabili aggiuntive**



Nota importante

Nel linguaggio parlato, si è soliti dire «*una funzione che prende in ingresso un vettore e restituisce un secondo vettore*»

A fare una corrispondenza 1:1 con il C la frase suona sbagliata, perché

- **La funzione non può fare return di un vettore**
- **La funzione dovrà prendere in input anche la lunghezza effettiva del vettore**

Chi interpreta la frase in questo modo dimostra di non conoscere i meccanismi di base delle funzioni e della gestione della memoria del C!

Tuttavia, i programmatori C non comunicano «dettando prototipi delle funzioni» e queste espressioni vanno interpretate in senso più generale per cui

- **Restituire -> rende disponibile dopo l'esecuzione, un vettore nel R.A chiamante**
- **Prendere in ingresso un vettore -> ... ed eventuali altre variabili aggiuntive**



Parametri: pro e contro

Modi di passaggio caratteristiche	per valore ("per copia")	per indirizzo ("per riferimento") ("per puntatore")
Tempo e spazio necessari a trasferire (copiare) i dati	grandi (per parametri di grandi dimensioni)	piccoli (la dimensione dei puntatori è fissa, non dipende dai dati)
C'è rischio di effetti collaterali indesiderati?	No (i parametri attuale e formale sono distinti)	Sì (i parametri attuale e formale "di fatto" coincidono)
Permette la restituzione di valori al chiamante?	No	Sì



Il perimetro con le funzioni

```
#include <math.h>
#include <stdio.h>
#define N 5
typedef struct { float x; float y; } punto;

float dist ( punto p1, punto p2 ) {
    return sqrt( pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2) ); }

float perimetro ( punto poligono[], int dim ) {
    int i; float per = 0.0;
    for (i=1; i<dim; i++)
        per = per + dist(poligono[i-1], poligono[i]);
    return per + dist(poligono[0], poligono[dim-1]); }

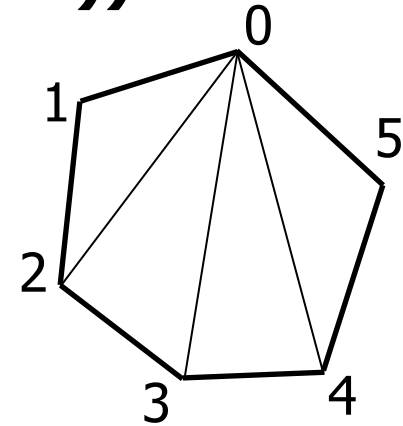
int main() {
    punto pol[N]; ... acquisizione delle coordinate dei punti ...
    printf("%d lati di lunghezza totale %f", N, perimetro(pol, N));
    return 0;
}
```


Scomposizione del poligono in triangoli (a ventaglio)

∀ triangolo (Erone): **$A = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$**

p: semiperimetro. **a, b, c**: misura dei lati A:area

```
float erone( punto p1, punto p2, punto p3 ) {  
    punto tri[3]; float p;  
    tri[0]=p1; tri[1]=p2; tri[2]=p3;  
    p = 0.5 * perimetro(tri, 3);  
    return sqrt(p*(p-dist(p3, p1))*(p-dist(p1, p2))*(p-dist(p2, p3))); }
```

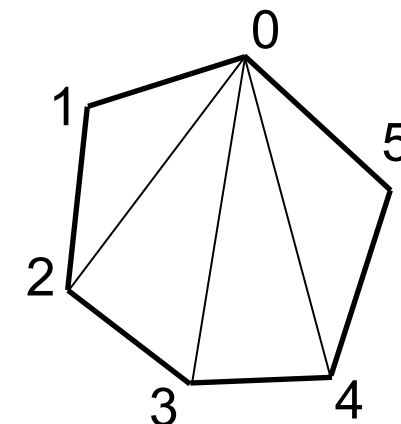


```
float areapol( punto polig[], int dim ) {  
    int i; float area = 0.0;  
    for (i=2; i<dim; i++)  
        area = area + erone( polig[0], polig[i-1], polig[i] );  
    return area;  
} /* Così funziona solo con i poligoni convessi */
```



Dal perimetro all'area (variante)

```
float erone( punto p1, punto p2, punto p3 ) {  
    float p, a, b, c;  
    a = dist(p1, p2); b = dist(p2, p3); c = dist(p1, p3);  
    p = (a+b+c) / 2;  
    return sqrt(p*(p-a)*(p-b)*(p-c));  
}
```



```
float areapol( punto polig[], int dim ) {  
    int i; float area = 0.0;  
    for ( i=2; i<dim; i++ )  
        area = area + erone( polig[0], polig[i-1], polig[i] );  
    return area;  
}
```

/ Ancora funziona solo coi poligoni convessi */*



Esempio TDE Settembre 2024

Si scriva una funzione **conta** che riceve in ingresso una stringa **S** contenente più parole separate da virgole e contenenti solamente lettere minuscole (oltre ad eventuali parametri aggiuntivi utili per lo sviluppo della funzione). La funzione **conta** deve restituire al chiamante la lettera dell'alfabeto che compare più frequentemente tra le iniziali delle parole (si supponga per semplicità sia unica).

Si scriva quindi una seconda funzione **elimina** che riceve in ingresso la stringa **S** (contenente più parole e solo caratteri minuscoli), un carattere **C** (oltre ad eventuali parametri aggiuntivi ritenuti utili per lo sviluppo della funzione), e riporta al chiamante una stringa **T** contenente le sole parole di **S** che iniziano per **C**, separate da virgole. Nel main si stampi il vettore restituito dalla funzione **elimina**.

Si parta dal file

<https://www.dropbox.com/scl/fi/wnpma7pkccnxu5mycwzkg/arrayTesto.c?rlkey=vwqnzi66ftgogaw7qdhjt79y4&dl=0>

Usando la stringa nel file allegato il programma stamperà:

Lettera maggiormente frequente: a

Parole che iniziano con a: albero,acqua,amico

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

#define MAX_STR_LEN 1000

void vettIniziali(char s[], char iniz[]);
char conta(char s[]);
void elimina(char s[], char t[], char c);

int main() {
    char s[MAX_STR_LEN] = "acane,gatto,albero,fiore,acqua,sole,amico,giardino,pietra,vento";
    char selez[MAX_STR_LEN];

    printf("%s\n",s);
    printf("carattere più presente: %c\n", conta(s));

    elimina(s, selez, conta(s));

    printf("\n%s\n",selez);

    return 0;
}
```

```

char conta(char s[])
{
    char iniz[MAX_STR_LEN], max_char;
    int max_val, cnt, i, l, j;
    vettIniziali(s, iniz);

    l = strlen(iniz);
    max_val = 1; max_char = iniz[0];

    for(i = 0; i < l; i++)
    {
        cnt = 0;
        for(j = 0; j < l; j++)
            if(iniz[i] == iniz[j])
            {
                cnt++;
                if (cnt > max_val)
                {
                    max_val = cnt;
                    max_char = iniz[i];
                }
            }
    }

    return max_char; }

```

```

void vettIniziali(char s[], char iniz[])
{
    int i, j=0, l;

    iniz[j] = s[0];
    j++;

    l = strlen(s);

    for(i = 0; i < l; i++)
    {
        if(s[i] == ',')
        {
            iniz[j] = s[i+1];
            j++;
        }
    }
    iniz[j] = '\0';
}

```

```

void elimina(char s[], char t[], char c)
{
    int i, l, k = 0;

    l = strlen(s);

    i = 0;
    if(s[i] == c)
    {
        while(s[i] != ',')
        {
            t[k] = s[i];
            k++;
            i++;
        }
        t[k] = s[i];
        k++;
    }
}

```

```

while(i < l)
{
    if(s[i] == ',')
    {
        i++;
        if(s[i] == c)
        {
            while(s[i] != ',')
            {
                t[k] = s[i];
                k++;
                i++;
            }
            t[k] = s[i];
            k++;
        }
    }
    i++;
}
t[k] = '\0';
return;
}

```



Esercizio 3 (8 punti) TDE 1 Marzo 2017

Scrivere una funzione che riceve due stringhe: *parola* e *elimina*. La funzione cerca in *parola* tutti i caratteri che compongono la stringa *elimina*, e li rimuove solamente se li trova tutti e nell'ordine in cui compaiono in *elimina*, anche se non consecutivi. L'operazione viene ripetuta finché in *parola* è contenuta un'intera istanza della stringa *elimina*. La funzione, oltre a modificare l'array *parola*, restituisce il numero di volte che ha eliminato i caratteri dell'intera stringa *elimina* da *parola*. Nell'eliminare caratteri non devono essere lasciati buchi, ma *parola* dev'essere ricompattata.

Esempio

parola: amaarrreeemmmarrreeaaaarrmae

elimina: mare

La funzione trova tutte le lettere di "mare" in "aMAaRrrEeeemmmarrreeaaaarrmae" che diventa "aarreeemmmarrreeaaaarrmae"

La funzione trova tutte le lettere di "mare" in "aarreeeMmmmARrrEeaaaarrmae" che diventa "aarreeemmmrreeaaaarrmae"

La funzione trova tutte le lettere di "mare" in "aarreeemmMrreAaaaRrrmaE" che diventa "aarreeemmrreeaaarrma"

Poi non trova più tutte le lettere e restituisce 3

```
#include<stdio.h>
#define N 40

int eliminaParola(char *, char *);
int trovaIndici(char *, char *, int *);
void rimuoviIndici(char *, int *, int);

int main()
{
    char parola[N] = "amaarrreeeemmmmmarrreeaaaarrrmae";
    char elimina[N] = "mare";

    int n;

    printf("parola: %s,\nelimina: %s", parola, elimina);
    n = eliminaParola(parola, elimina);
    printf("\n\nparola: %s,\nelimina: %s\n n = %d", parola, elimina, n);

    return 1;
}
```



```
int eliminaParola(char parola[], char elimina[])
{
    int indx[N], n = 0, i;
    while(trovaIndici(parola, elimina, indx))
    {
        /*for(i = 0; i < strlen(elimina); i++)
            printf("\n indx[%d] = %d", i, indx[i]); */

        rimuoviIndici(parola, indx, strlen(elimina));
        n++;
        printf("\nparola: %s, \nelimina: %s\n n = %d", parola, elimina, n);

    }
    return n;
}
```

```

int trovaIndici(char parola[], char elimina[], int indx[])
{
    int i, j = 0;
    for(i = 0; i < strlen(parola); i++)
        if(parola[i] == elimina[j])
        {
            //printf("\n found %c at %d", elimina[j], i);
            indx[j] = i;
            j++;
        }
    // printf("%d, %d", j, strlen(elimina));
    if(j == strlen(elimina))
        return 1;
    else
        return 0;
}

```

```

void rimuoviIndici(char parola[], int indx[], int n)
{
    int i, j, len_p;
    for(i = n; i >= 0; i--) // parti dal fondo se no cambiano gli indici da rimuovere
    {
        printf("\n rimuovo %d (%d)", indx[i], n);
        len_p = strlen(parola); // devo ricalcolare la lunghezza dopo che taglio
        // rimuovo l'i-simo elemento da parola
        for(j = indx[i]; j < len_p; j++)
            parola[j] = parola[j+1];
    }
}

```



Si una funzione riordina che prende in ingresso due vettori **u** e **v** (più eventuali variabili necessarie) e che riordina gli elementi del primo vettore passato in input (**u** nel nostro caso) nel seguente modo:

- tutti gli elementi di **u** che non compaiono in **v** saranno nelle prime posizioni di **u** dopo l'invocazione
- tutti gli elementi di **u** che compaiono in **v** saranno nelle ultime posizioni di **u** dopo l'invocazione

Si faccia riferimento all'esempio sotto per eventuali dubbi su come modificare **u**.

Si invochi quindi la funzione relativamente al seguente codice e si stampi il contenuto del vettore **u** a schermo.

N.B non è possibile stampare all'interno della funzione riordina



```
#include<stdio.h>
#define N 5

int main()
{
    int u[N] = {4,1,3,7,0}, v[N] = {1,6,8,4};

    // TODO stampa u e v

    // TODO invocazione riordina

    // TODO stampa u dopo invocazione

    return 0;
}
```

Risultato atteso

u = 4 1 3 7 0

v = 1 6 8 4

dopo invocazione u = 3 7 0 4 1



Il tipo Popolo definisce un popolo con nome e numero di abitanti.

```
typedef struct{
    char nome[N];
    int pop;
}Popolo;
```

Si scriva una funzione `analizzaPopoli` che prende in ingresso un array di elementi di tipo `Popolo` -- ed ulteriori variabili aggiuntive se necessarie-- e riporta al chiamante il nome del popolo con più abitanti tra quelli contenuti nell'array ed il nome del popolo con il numero di abitanti più vicino alla media degli abitanti dei vari popoli. Per identificare il popolo più vicino alla popolazione media, si consideri la differenza di popolazione in valore assoluto.

Si invochi la funzione `analizzaPopoli` nel main e si stampino i nomi dei due popoli selezionati.

In riferimento all'array popolato nel main nel seguente codice (da utilizzare per sviluppare la soluzione)
https://www.dropbox.com/s/c0ckq4moeggmwpr/indiani_TODO.c?dl=0



```
strcpy(indiani[0].nome, "Cayuga");  
indiani[0].pop = 652;  
strcpy(indiani[1].nome, "Oneida");  
indiani[1].pop = 2123;  
strcpy(indiani[2].nome, "Seneca");  
indiani[2].pop = 3521;  
strcpy(indiani[3].nome, "Onondaga");  
indiani[3].pop = 1763;  
strcpy(indiani[4].nome, "Mohawak");  
indiani[4].pop = 4512;
```

ci si aspetta la seguente stampa dal main:

Più numerosi: Mohawak, più vicini alla media: Oneida



Si scriva una funzione `rimuoviValori` che prende in ingresso una matrice `M` di interi di dimensioni $N \times N$ (con `#define N 4`), un intero `x` e ulteriori variabili aggiuntive, se necessarie.

La funzione identifica i valori di `M` che sono a distanza minima da `x` e modifica `M` mettendo a zero tutti i valori che stanno sulla parallela alla diagonale principale passante per i valori sopra identificati. Si guardino gli esempi sotto per evitare fraintendimenti su quali elementi mettere a zero.

N.B. Qualora ci fossero più valori in `M` alla stessa distanza da `x`, la funzione `rimuoviValori` ripete l'operazione per tutti i valori a distanza minima.

N.B. Si consiglia fortemente l'uso di funzioni ausiliarie per produrre un codice semplice, leggibile e compatto

Si scriva nel `main` l'invocazione alla funzione `rimuoviValori` e si stampi la matrice nel `main`, non in `rimuoviValori`.



Esercizio Gennaio 2022

M =

1	3	5	3
5	6	7	5
5	6	2	1
3	6	9	4

rimuoviValori invocata su M con $x = 7$

1	0	5	3
5	6	0	5
5	6	2	0
3	6	9	4

M =

1	3	5	3
5	6	7	5
5	6	2	1
3	6	9	4

rimuoviValori invocata su M con $x = 8$

1	0	5	3
0	6	0	5
5	0	2	0
3	6	0	4

Funzioni e Matrici



Parametri di tipo array e struct

Per passare a una funzione un parametro di tipo array occorre passarne l'indirizzo base, perciò di fatto **gli array sono sempre passati per indirizzo**

- Analogamente, una funzione **non può restituire un array** (inteso come "il suo contenuto"), **ma solo un puntatore a un array** (cioè il puntatore al suo primo elemento)

Un parametro di tipo struct si può passare **sia per indirizzo sia per valore** (anche se la struct contiene campi di tipo array!)

- Analogamente, una funzione **può restituire una variabile di tipo struct** (anche se la struct contiene degli array)

Altre particolarità degli array con le funzioni:

Esempio: `typedef double TipoArray [DIMENSIONE]`

Tre prototipi equivalenti:

```
double sum (TipoArray a, int n)
double sum (double a[], int n)
double sum (double *a, int n)
```

N.B.: non si deve specificare la dimensione del vettore; *n* rappresenta la porzione dell'array da considerare valida

La funzione `sum(V, n)` somma i primi *n* elementi di un array. Supponiamo di avere un array `V[50]`:

```
sum (V, 50)      restituisce V[0] + V[1] + ... V[49]
sum (V, 30)      restituisce V[0] + V[1] + ... V[29]
sum (&V[0], 30)  restituisce V[0] + V[1] + ... V[29]
sum (&V[5], 7)   restituisce V[5] + V[6] + ... V[11]
sum (&V[0]+5, 7) restituisce V[5] + V[6] + ... V[11]
sum (V+5, 7)     restituisce V[5] + V[6] + ... V[11]
```



Ancora sui parametri di tipo array

Per gli array mono-dimensionali:

Dichiariamo i parametri formali come puntatori al tipo degli elementi dell'array

Dichiarazione del parametro formale

```
... f ( UnTipo vet[] )
```

Non occorre, sintatticamente, specificare la dimensione statica degli array: il compilatore può eseguire il calcolo dello spiazzamento in base al tipo puntato

- **UnTipo v**[N] ;

- **v**[i] ≡ * (**v**+i)

- serve conoscere solo **sizeof (UnTipo)**, **N** non serve

Dichiarando in alternativa:

```
... f ( UnTipo * vet )
```

il compilatore può comunque risolvere l'espressione **vet**[i]



Ancora sui parametri di tipo array

Per gli array multi-dimensionali

Il calcolo dello spiazzamento richiede di conoscere alcune dimensioni intermedie

- `UnTipo m[X][Y], c[X][Y][Z];`
- $m[i][j] \equiv *(*(m+i)+j) \approx m + Y \cdot i + j$
 - serve conoscere `sizeof(Tipo)` e `Y` (`X` non serve)
- $c[i][j][k] \equiv *(*(*c+i)+j)+k \approx c + Y \cdot Z \cdot i + Y \cdot j + k$
 - servono `sizeof(Tipo)`, `Y` e `Z` (`X` non serve)

Sintatticamente occorre specificare tutte le dimensioni meno l'ultima

Dichiarazioni corrette: ... `g(UnTipo mat[][Y])` ... `h(UnTipo cube[][Y][Z])`

Che si può invocare con la chiamata ...`g(m)` ;

Se dichiaro **erroneamente**:

... `g(UnTipo * mat)` ... `h(UnTipo * cube)`

il compilatore non sa risolvere le espressioni `mat[i][j]` e `cube[i][j][k]`




Modificare una matrice con una funzione

```
#include<stdio.h>
#define N 30
void tabellina(int [][][N], int);
void stampaTab(int [][][N], int);

int main()
{
    int tab[N][N], n;
    do
    {
        printf("inserire quante tabelline: ");
        scanf("%d", &n);
    }while(n > N || n < 0);

    tabellina(tab, n);
    stampaTab(tab, n);
    return 0;
}
```



```
void tabellina(int t[][N], int n)
{
    int i, j;
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            t[i][j] = (i + 1) * (j + 1);
}
```

```
void stampaTab(int t[][N], int n)
{
    int i, j;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
            printf("%5d", t[i][j]);
        printf("\n");
    }
}
```



Funzioni e struct

I tipi struct vengono passate in input e restituite dalle funzioni come i normali tipi base.

- Questo vale perché è possibile fare assegnamenti tra struct
- Vale anche quando una struct contiene un array tra i propri campi
- È sempre possibile utilizzare il passaggio per riferimento anche con i tipi strutturati