



Linguaggi C e C++

- **Puntatori**
- **Passaggio di parametri**
- **Allocazione dinamica della memoria**



PUNTATORI

(Linguaggio C)



Puntatori: utilizzo

- Vettori \leftrightarrow Puntatori
- Passaggio di parametri per indirizzo
- Allocazione dinamica della memoria (heap)



Variabile

- Identificatore
- Tipo

Esempio:

```
int a = 5;
```

`&a` → indirizzo (di memoria) della variabile di nome *a*



Puntatore: definizione

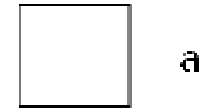
- L'indirizzo di memoria di una variabile può essere assegnato solo ad una speciale categoria di variabili dette puntatori.
- I puntatori sono quindi delle variabili abilitate a contenere un indirizzo (di memoria).

<tipo> * <variabile>;

↓
int, char, float, double, ...

Primi passi con i puntatori n. 1

int a;



char c;



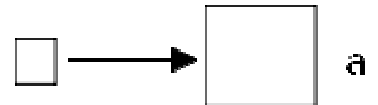
int * pi;



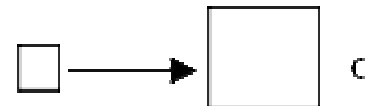
char * pc;



pi = &a;



pc = &c;



a = 5; oppure *pi = 5;

c = 'x'; oppure *pc = 'x';

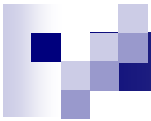


Operatori del puntatore

■ Operatori unari

- **&**

- ***** (operatore di indirezione)

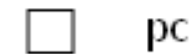


Primi passi con i puntatori n. 2

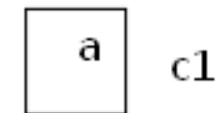
```
char c1, c2;
```



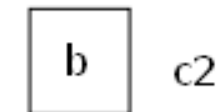
```
char * pc;
```



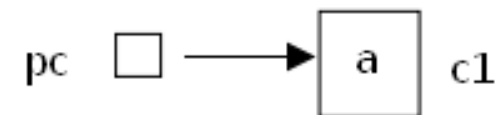
```
c1 = 'a';
```



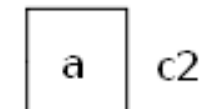
```
c2 = 'b';
```



```
pc = &c1;
```



```
c2 = *pc;
```



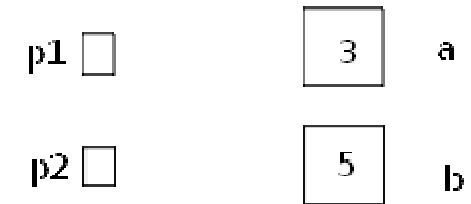
Primi passi con i puntatori n. 3

```
int a = 3;
```

```
int b = 5;
```

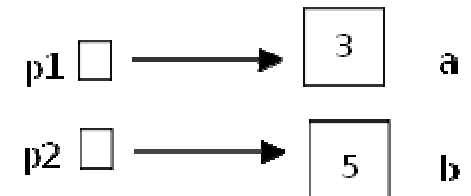
```
int * p1;
```

```
int * p2;
```



```
p1 = &a;
```

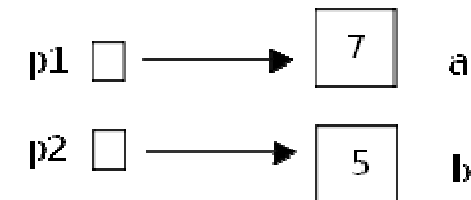
```
p2 = &b;
```



```
a = 7;
```

```
oppure
```

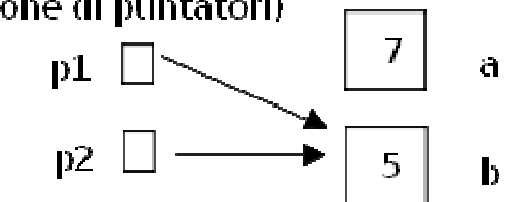
```
*p1 = 7;
```



```
p1 = p2; (assegnazione di puntatori)
```

```
oppure
```

```
p1 = &b;
```



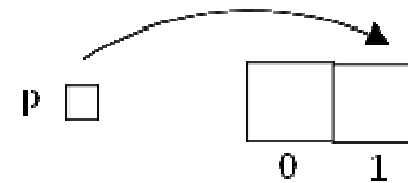
Primi passi con i puntatori n. 4

```
int buff[2];
```

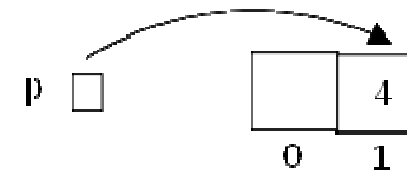


```
int *p;
```

```
p = &buff[1];
```

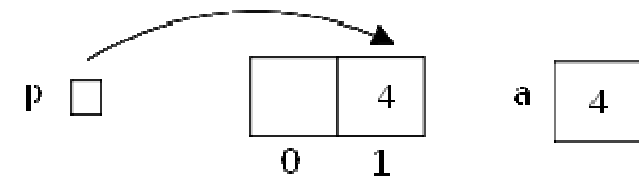


```
*p = 4;
```



```
int a;
```

```
a = *p;
```





Nota

```
int * p;  
char a;
```

`p = &a;` //NO! Devono essere dello stesso tipo



Array e puntatori

- In C c'è una forte relazione tra puntatori ed array derivante dal fatto che gli elementi di un array vengono allocati in memoria in “celle” consecutive.
- Il nome di un array è una costante il cui valore è l'indirizzo della prima componente dell'array.

```
char buf[5];
```

```
char * s;
```

```
s = &buf[0];    ≡    s = buf;
```



Espressioni con puntatori

- Assegnazione di puntatori
- Aritmetica dei puntatori (operatori: +, -, ++, --)
- Confronto fra puntatori



Assegnazione di puntatori

```
int a = 5;  
int *p, *p2;
```

```
p = &a;
```

```
p2 = p;           //assegnazione di puntatori
```

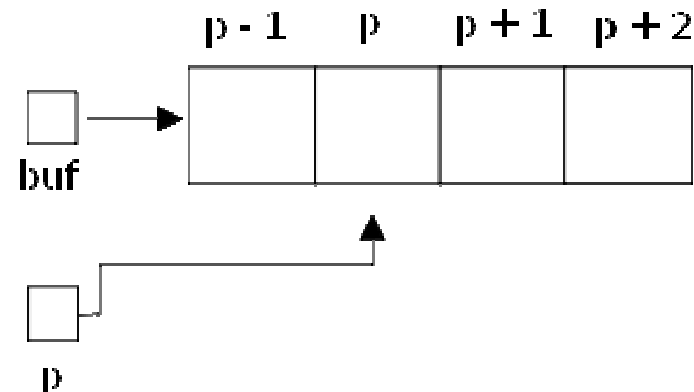
```
*p2 = 3;  ≡      *p = 3;  ≡      a = 3;
```

(vedi disegno alla lavagna)

Aritmetica dei puntatori

- Al valore di un puntatore è possibile sommare o sottrarre valori interi.
- La somma e la differenza tra un puntatore ed un valore intero viene intesa come lo spostamento (somma \rightarrow a dx, differenza \rightarrow a sx) logico del puntatore di un numero di locazioni di memoria pari al prodotto del numero intero per la dimensione del tipo di oggetto puntato.

- `int buf[4];`
- `int *p = &buf[1];`



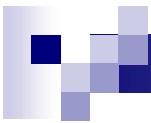


Aritmetica dei puntatori

```
char buf[5];  
char * p = buf;  
    p = p + 1;
```

```
double buf[5];  
double * p = buf;  
    p = p + 1;
```

Aggiungendo 1 ad un puntatore lo si fa puntare all'elemento successivo dell'array. Questo indipendentemente dalla dimensione di ciascun elemento.



Accesso ad un elemento di un vettore

- tramite
 - Indice
 - Puntatore

```
char buf[10];
```

```
char * s;
```

```
s = buf;
```

```
buf[7] = 'a';
```

≡

```
*(s+7) = 'a';
```



Domanda

- Date le seguenti definizioni:

char buf[10];

char * s;

qual è la differenza fra buf e s ?



Risposta

- buf è un puntatore costante (alla 1^a componente del vettore), s è un puntatore (non costante e quindi modificabile).



Accesso ad un elemento di un vettore

//accesso con indice

```
char buf[2];
```

```
int i;
```

```
    for (i=0; i<2;i++)
```

```
        buf[i] = 'k';
```

//accesso con puntatore

```
char buf[2];
```

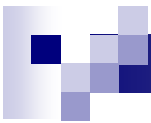
```
char *s;
```

```
int i;
```

```
    s = buf;
```

```
    for (i=0; i<2;i++)
```

```
        *(s + i) = 'k';
```



Accesso ad un elemento di un vettore

//accesso con puntatore

```
char buf[2];  
char *s;  
int i;  
    s = buf;  
    for (i=0; i<2;i++)  
        *s++ = 'k';
```

L'istruzione `*s++ = 'k';` opera nel seguente modo:

- a) copia 'k' nella locazione di memoria puntata da s;
- b) poi incrementa s di una unità (s quindi punta all'elemento successivo del vettore)



Ancora sull'aritmetica dei puntatori

- puntatore + un numero intero = puntatore
 - (se il numero intero è positivo spostamento a dx, altrimenti spostamento a sx)
- puntatore + puntatore = non è definito
- puntatore – puntatore = numero naturale
 - (il numero rappresenta la distanza ovvero il numero di posizioni tra i due elementi dell'array. La sottrazione fra puntatori è definita solo quando entrambi i puntatori puntano ad elementi dello stesso array.)



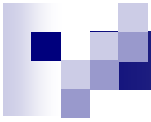
Esercizio

Dato il seguente frammento di pgm C:

```
int a[10] = {1, 6, 4, 3, 4, 5, 3, 6, 8, 1};  
int *p=a;  
int i, totale=0;  
for(i=0; i<10; i++)  
    if( a[9 - i] == (*p)) {  
        totale++;  
        p++;  
    }
```

indica quali delle seguenti affermazioni sono vere e quali false dopo l'esecuzione del ciclo for (giustifica le risposte attraverso disegni chiari e precisi):

```
a[totale] == totale  
(*p) - 1 == a[totale - 1]  
(*p) == 4  
*(p + a[0]) == a[totale - 2]
```



Passaggio di parametri (Linguaggio C)

Per il linguaggio C++ già visto in altre slide



Funzione scambia()

```
int main() {  
    int m,n;  
        //lettura  
        scambia(m,n);  
        //stampa  
}
```

```
void scambia(int a, int b) {  
    int temp;  
  
        temp = a;  
        a = b;  
        b = temp;  
}
```

(vedi disegno Stack alla lavagna)



Osservazioni

- Utilizzando il passaggio di parametri per valore non siamo riusciti a realizzare il compito della funzione `scambia()`.
- Soluzione:
- Passare non le variabili, ma l'indirizzo delle variabili.
- Questo tipo di passaggio di parametri è noto con il nome di passaggio di parametri per indirizzo.



Passaggio di parametri per valore

- I parametri formali sono delle locazioni di memoria nelle quali vengono copiati i contenuti dei parametri attuali.
- Nel passaggio di parametri per valore i parametri formali e parametri attuali fanno riferimento ad aree distinte di memoria.



Passaggio di parametri per indirizzo

- I parametri formali sono dei puntatori cui vengono associati gli indirizzi di memoria dei parametri attuali (durante l'esecuzione del sottopgm).
- Nel passaggio di parametri per indirizzo parametri formali e parametri attuali fanno riferimento a stesse di memoria.

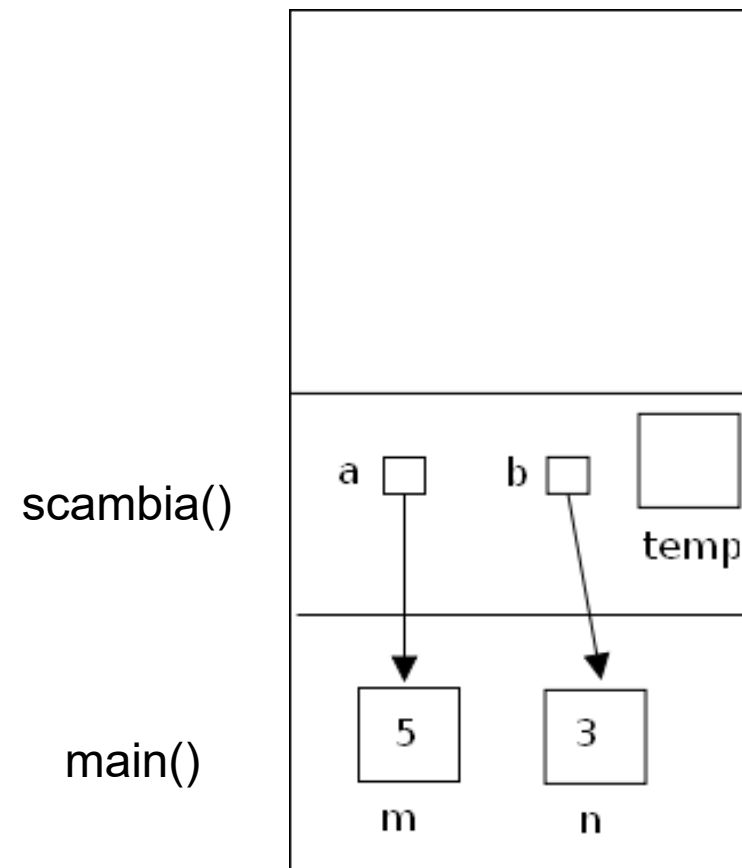


Funzione scambia()

```
int main(){  
    int m,n;  
        //lettura  
        scambia(&m,&n);  
        //stampa  
}
```

```
void scambia(int *a, int *b) {  
    int temp;  
  
        temp = *a;  
        *a = *b;  
        *b = temp;  
}
```

Disegno dello stack





Esercizio

- Scrivere un pgm che, immessi il numeratore e il denominatore di una frazione, verifichi se è ridotta ai minimi termini, facendo uso di una funzione C che calcoli il massimo comun divisore.



```
int main() {  
    int num, den, mcd;  
  
    //leggi frazione  
    mcd = maxComDiv(num,den);  
    if(mcd == 1)  
        printf("La frazione è ridotta ai minimi termini");  
    else {  
        num /= mcd;  
        den /= mcd;  
        printf("La frazione equivalente è %d/%d", num, den);  
    }  
}
```




```
int maxComDiv (int a, int b) {  
    int resto;  
    do {  
        resto = a%b;  
        a = b;  
        b = resto;  
    }while(resto != 0);  
  
    return a;  
}
```

(vedi disegno alla lavagna Stack)



Osservazione

- Mostrare che utilizzando il passaggio di parametri per indirizzo non è possibile poi stampare frazione equivalente, perché cambiano i valori di num e den (a meno di salvarli nel main()).



Esercizio

- Codificare la funzione `strlen()` in 3 modi diversi.



Soluzione 1° modo

```
int strlen(const char * str) {  
    int i;  
    for(i=0; str[i]; i++);  
    return i;  
}
```



Soluzione 2° modo

```
int strlen(const char * s){  
    int i=0;  
    while (*s++)  
        i++;  
    return i;  
}
```



Soluzione 3° modo

```
int strlen(const char * s){  
    char * q = s;  
    while (*q++);  
    return (q - s - 1 );           //differenza tra puntatori  
}
```



Esercizio

Scrivi una procedura ricorsiva

```
void printChar (const char *stringa);
```

che stampi, ricorsivamente, tutti i caratteri contenuti in stringa, un carattere per linea.



Soluzione

```
void printChar (const char *stringa){  
    If (*stringa != '\0'){  
        printf("%c\n", *stringa);  
        printChar(stringa+1);  
    }  
}
```




Esercizio

- Si definisca una funzione ricorsiva che dato un array `vet` di interi e la sua dimensione `dim`, restituisca il più piccolo intero maggiore di tutti gli elementi di `vet`. Se ad esempio l'array contiene : 15 100 0 -20, allora la funzione restituisce 101.
- N.B. Ogni chiamata ricorsiva deve risolvere il problema originario per una porzione dell'array.



Soluzione

```
int minimoMaggiorante(const int vet[], int dim) {  
    int res, act = vet[0]+1;  
    if(dim==1)  
        return act;  
    res = minimoMaggiorante(vet+1, dim-1);  
    if(res>act)  
        return res;  
    else  
        return act;  
}
```



Passaggio di un vettore

- In C il vettore può essere passato solo per indirizzo.

```
const int DIM = 10;
```

```
int main() {  
int vett[DIM];  
int numElem;
```

```
    //insert  
    stampa(vett, numElem);  
}
```



Passaggio di un vettore

■ Prototipi equivalenti:

- ☐ void stampa (int vett[], int numElem);
- ☐ void stampa (int vett[DIM], int numElem);
- ☐ void stampa (int *vett, int numElem);
- ☐ void stampa (int [], int);
- ☐ void stampa (int *, int);



Inserimento dati in un vettore

```
const int DIM = 10;
```

```
int main() {  
    int numeri[DIM];  
    int *p, i;  
  
        p = numeri;  
        ...  
}
```



Inserimento dati in un vettore

```
for (i = 0; i < DIM; i++)  
    scanf("%d", &numeri[i]);  
    oppure:  
    scanf("%d", numeri + i);  
    oppure:  
    scanf("%d", &p[i]);           //(*)  
    oppure:  
    scanf("%d", p + i);
```

(*) In C è possibile “assegnare” un indice ad un puntatore come se fosse un vettore.



Stampa dati di un vettore

```
for (i = 0; i < DIM; i++)  
    printf("%d\n", numeri[i]);  
    oppure:  
    printf("%d\n", *(numeri + i));  
    oppure:  
    printf("%d\n", p[i]);  
    oppure:  
    printf("%d\n", *(p + i));
```



Funzione insert()

```
const int DIM = 10;
```

```
int main() {  
    int vett[DIM];  
    int numElem;
```

```
        insert(vett, &numElem);  
        stampa(vett, numElem);  
}
```




Funzione insert()

```
void insert (int vett[ ], int *numElem) {                                //E' una procedura!  
    int i;  
    do {  
        printf("Quanti elementi vuoi inserire?\n");  
        scanf("%d", numElem);  
    }while(*numElem < 1 || *numElem > DIM);  
  
    for(i=0; i < *numElem; i++) {  
        printf("Inserisci l'elemento n. %d\n", i+1);  
        scanf("%d", &vett[i]);  
    }  
}
```



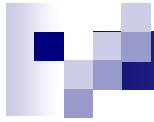
Nota

- insert() essendo una procedura, preferisco non usare “return”.
- Quindi scrivo:
 - void insert (int vett[], int *numElem);anziché
 - int insert (int vett []);



Versione con **return**

```
int insert (int vett [ ]) {  
    int i, numElem;  
    do {  
        printf("Quanti elementi vuoi inserire?\n");  
        scanf("%d", &numElem);  
    }while(numElem < 1 || numElem > DIM);  
  
    for(i=0; i < numElem; i++) {  
        printf("Inserisci l'elemento n. %d\n", i+1);  
        scanf("%d", &vett[i]);  
    }  
    return numElem;  
}
```



Funzione stampa()

```
void stampa (int vett[ ], int numElem) {  
    int i;  
    for(i=0; i<numElem; i++)  
        printf("%3d", vett[i]);  
}
```



Passaggio di una matrice

```
#define MAX_R 3  
#define MAX_C 4
```

```
int main() {  
    int matrix [MAX_R][MAX_C];  
    int numRighe, int numColonne;
```

```
        insert(matrix, &numRighe, &numColonne);  
        stampa(matrix, numRighe, numColonne);  
}
```



Funzione stampa()

```
void stampa(int matrix[ ][MAX_C], int numRighe, int
    numColonne) {
    int i, j;

    for(i=0; i<numRighe; i++) {
        for(j=0; j<numColonne; j++)
            printf("%3d", matrix[i][j]);
        printf("\n");
    }
}
```

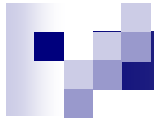


NOTA

■ Come vedi nel prototipo

```
void stampa(int matrix[ ][MAX_C], int numRighe, int numColonne);
```

- È obbligatorio indicare la dimensione delle colonne (vedi implementazione di una matrice in memoria centrale).



Esercizio

- Scrivi la funzione C `insert()` che inserisce i dati in una matrice.



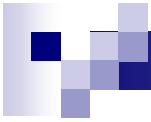
Esercizio

- Scrivi una funzione C che, dato un vettore di interi, calcola la media aritmetica.



Soluzione

```
double mediaAritmetica(int *vett int numElem) {  
    int i, somma=0;  
    for(i=0; i<numElem; i++)  
        somma += vett[i];  
    return (double)somma/numElem;  
}
```



■ Chiamata:

```
media = mediaAritmetica(vett, numElem);
```

Nota:

Si può fare anche così:

```
void mediaAritmetica(int *vett, int numElem, double *media);
```

ma, essendo una “vera” funzione preferisco l’uso del “return”.



Esercizio

- Scrivi una funzione C che ritorni sia la somma che la differenza di due argomenti.
- Nota: E' una procedura!



```
int main() {  
    int a, b;  
    int somma, diff;  
  
        //lettura di a e b  
        sommaDiff(a, b, &somma, &diff);  
        .....  
}
```



```
void sommaDiff(int x, int y, int *pS, int *pD) {  
    *pS = x + y;  
    *pD = x - y;  
}
```

(vedi disegno Stack alla lavagna)



Esercizio

- Scrivi una funzione C che, dato un vettore di interi, restituisce il massimo e la sua posizione.
- E' una funzione o una procedura?
Perché?




Regole di visibilità (scope rules)

- Un pgm C, come abbiamo più volte ricordato, è costituito da un insieme di funzioni.
- Ogni funzione definisce e utilizza internamente delle variabili proprie: una variabile definita internamente ad una funzione è detta variabile locale.




Regole di visibilità (scope rules)

- Esiste la possibilità di definire anche variabili globali: queste sono accessibili a tutte le funzioni del pgm (ti ricordo che noi al momento abbiamo visto pgm scritti su un unico file).



Regola pratica per stabilire la direzionalità di un parametro.

- Porsi idealmente all'interno del sottopgm e chiedersi:
 - ☐ Il valore del parametro viene fornito dall'esterno? (IN)
 - ☐ Il valore del parametro viene fornito all'interno? (**OUT**)
 - ☐ Il valore del parametro viene modificato all'interno del sottopgm? (IN/**OUT**)



Legame tra direzionalità e passaggio di parametri

- Si passano per valore tutti i parametri di solo ingresso purché non siano array o dati che occupano molta memoria.
- Si passano per indirizzo tutti i parametri di O o di I/O, gli array e i dati che occupano molta memoria.



Osservazione

- L'uso di **const** nella dichiarazione dei parametri impedisce al codice della funzione di modificare l'oggetto puntato dal parametro.
- Esempi:
 - `unsigned int strlen(const char * str);`
 - `int strcmp(const char * str1, const char * str2);`



Nota

- Ad un parametro (attuale/formale) vengono associati:
 - Un nome
 - Un tipo
 - Una posizione d'ordine
 - Una direzionalità (I, O, I/O)



Ricordiamo la definizione di tipo di dato

- Il tipo di una variabile è un attributo che specifica:
 - Un insieme di valori
 - Un insieme di operazioni/funzioni
 - L'implementazione in memoria



Esempio: tipo di dato stringa

- Insieme di valori: caratteri del codice ASCII
- Operazioni/funzioni:
 - gets(), puts()
 - strcmp()
 - strcpy()
 - strlen()
 - ...
- Implementazione: struttura sequenziale (array)

lvalue - rvalue

- `int a = 5;`

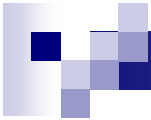
- **a**
 - **&a** (lvalue di a)
 - **5** (rvalue di a)

- L'lvalue è l'indirizzo di memoria di una variabile
- L'rvalue è il valore memorizzato all'lvalue
- Il compilatore ha bisogno degli lvalue per ottenere gli rvalue.



Funzioni che ritornano un puntatore

- Il tipo di ritorno di una funzione può essere un indirizzo a condizione che il valore restituito non sia un lvalue locale alla funzione.



Ricerca sequenziale in un vettore non ordinato

```
const int DIM = 10;
```

```
int main() {  
    int vett[DIM], numElem, elem;  
    int *punt;  
    .....  
    punt=ricercaSequenziale(vett, numElem, elem);    //chiamata  
    if(punt != NULL)  
        printf("Elemento trovato\n");  
        //printf("Elemento trovato nella posizione %d\n", (punt - vett) + 1);  
    else  
        printf("Elemento non trovato\n");  
}
```



Ricerca sequenziale in un vettore non ordinato

```
int * ricercaSequenziale(const int vett[], int numElem, int elem){  
    int i;  
  
    for(i=0; i<numElem; i++)  
        if(elem == vett[i])  
            return &vett[i];  
    return NULL;  
}
```

Error: invalid conversion from `const int*' to `int*'

Soluzione: **return (int *) &vett[i];**



Funzioni di libreria che ritornano un puntatore

- `char * gets(char * str);`
- `int atoi (const char * str);`
- Esempio di utilizzo del puntatore come tipo valore di ritorno di una funzione:
`int num = atoi(gets(str));`
- Nota: viene effettuata una conversione diretta (implicita) tra `(char *)` e `(const char *)`. Ti ricordo che non si può fare il viceversa se non ricorrendo ad una conversione esplicita tramite l'operatore di cast.



strchr()

- Prototipo:

char * strchr(const char *str, int car);

- La funzione strchr() restituisce un puntatore alla 1^a occorrenza di car individuata alla stringa puntata da str. Se l'operazione ha esito negativo la funzione restituisce un puntatore a NULL.

```
char *punt = strchr("Mappamondo",'m');  
printf(punt);           //Che cosa stampa?
```

- Codificare la funzione strchr() per esercizio.



strcat()

- Il suo prototipo è:

```
char * strcat(char *str1, const char * str2);
```

E se fosse stato:

```
void strcat(char *str1, const char * str2);  ?
```

Nota: strcat() è una procedura!



Usi di strcat()

- `strcat(str1, str2);` `//usato come procedura`
- `strcpy(str3, strcat(str1, str2));` `//usato come funzione`



abs()

- E' una funzione!
- Non ha senso usarlo come una procedura e cioè:
`abs(num);`

Ma ad esempio:

`val = abs(num);` oppure `printf("%d",abs(num));` oppure ...



Esercizio

- Codificare la funzione di libreria `strcat()`.



NOTA BENE

- Una caratteristica della gestione dei parametri in C è il fatto che essi sono passati alla funzione chiamata a partire dall'ultimo, cioè da destra a sinistra.
- Tale comportamento, nella maggior parte delle situazioni, è trasparente per il programmatore, ma possono verificarsi casi in cui è facile essere tratti in inganno.



Esercizio

Che cosa viene stampato (senza giustificare) dopo l'esecuzione delle seguenti istruzioni ?

```
int y [7] = { 1, 5, 6, 7, 8, 9, 12 };  
int *p;  
int s;  
    p = y + 1;  
    p += 2;  
    s = *p;  
    printf( "%d %d %d %d ", s, *(p + 1), *p++, *(p - 2) );
```



Puntatori a record

```
struct posiz esa[6];
```

```
struct posiz p = esa[5];  
struct posiz *pp;
```

```
pp = esa;  
pp = &esa[5];
```

```
double x = (*pp).x;           //la parentesi è obbligatoria
```

oppure

```
double x = pp→x;
```



Nota Bene

- L'operatore freccia \rightarrow è utilizzato al posto dell'operatore punto $.$ quando si accede ad un elemento della struttura mediante un puntatore.



Passaggio di una struttura per indirizzo

```
void posizioneOpposta(struct posiz *p) {  
    (*p).x = -(*p).x;    //p→x = - p→x;  
    (*p).y = -(*p).y;    //p→y = - p→y;  
}
```

Chiamata: `posizioneOpposta(&p1);`



Operazioni di I/O

```
struct posiz{
    double x;
    double y;
};

int main() {
    struct posiz esa[6];

        insert(esa);
        stampa(esa);

    return 0;
}
```

```
void insert(struct posiz esa[ ]) {
    int i;
        for(i=0; i<6; i++){
            printf("...");
            scanf("%f%f", &esa[i].x, &esa[i].y);
        }
}
```

```
void stampa(const struct posiz *esa) {
    int i;
        for(i=0; i<6; i++) {
            printf("%f %f\n", esa[i].x, esa[i].y);
        }
}
```



Confronta queste due funzioni

```
void stampa(const struct posiz *esa)
{
    int i;
    for(i=0; i<6; i++) {
        printf("%f %f\n", esa[i].x, esa[i].y);
    }
}
```

```
void stampa(const struct posiz *esa) {
    struct posiz *pp;

    for(pp=esa; pp<esa+6; pp++) {
        printf("%f %f\n", pp->x, pp->y);
    }
}
```




Osservazione

- I campi di un record possono essere di qualunque tipo (int, char, double, float), vettori, matrici, stringhe e per finire di tipo record.
- C'è anche la possibilità di avere dei campi di tipo puntatore (in questo caso, l'area di memoria a cui essi puntano è allocata nello heap).

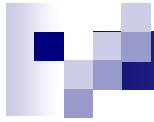


Passaggi di parametri

| | | |
|----------------|------------|---------------|
| Linguaggio C | Per valore | Per indirizzo |
| | Per valore | |
| Linguaggio C++ | Per valore | by reference |



Allocazione dinamica della memoria (Linguaggi C e C++)



Introduzione

- `int a[3];` → allocazione statica (viene stabilita dal compilatore)
- L'allocazione dinamica della memoria è l'allocazione eseguita durante l'esecuzione del pgm, in base ad esplicite istruzioni scritte nel pgm stesso.



“Oggetti” dinamici

- In C/C++ è il programmatore che decide la loro creazione e distruzione.
- Area di memoria: Heap
- `#define NULL 0` in `<stdio>`
- Operatore unario: `sizeof(tipo|var)` restituisce la dimensione in byte di un oggetto.
- L'operatore `sizeof`, simile ad una funzione, ritorna un tipo speciale, usato per le dimensioni di memoria: `size_t`
- In `<stdlib>` si trova: `typedef unsigned size_t;`



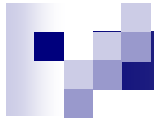
Allocazione/creazione di oggetti dinamici

- `#include <stdlib>`
- `void * malloc (size_t dim);`
- Alloca un blocco di memoria di dim byte
- Restituisce un puntatore generico al blocco di memoria (oppure NULL, se non c'è memoria disponibile).
- Testare il puntatore (se uguale a NULL chiamare la `exit()`)



Esempio

- `double * punt;`
- `punt = (double *) malloc (sizeof(double));`
- (v. disegno alla lavagna)



Accesso ad oggetti dinamici

- (v. disegno alla lavagna)



Allocazione/creazione di oggetti dinamici

```
void * calloc (size_t num, size_t size);
```

Alloca num elementi consecutivi di memoria di dimensione size (inizializzandola a zero).

Serve ad allocare sequenze contigue di aree di memoria (array).

Restituisce un puntatore generico al blocco di memoria (oppure NULL, se non c'è memoria a disposizione).

```
double * punt, * punt2;  
punt = (double *) calloc(10, sizeof(double));  
Più interessante:  
    punt2 = (double *) calloc(var, sizeof(double));
```

Si può ottenere la stessa cosa anche con la malloc():

```
punt2 = (double *) malloc(var* sizeof(double));
```

(v. disegno alla lavagna)



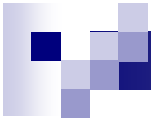
Deallocazione/distruzione di oggetti dinamici

- `void free (void * punt);`
- Rende nuovamente disponibile lo spazio di memoria (il blocco di memoria) puntato da `punt`. Se `punt` è uguale a `NULL`, la `free()` non fa nulla.
- E' buona norma "azzerare" i puntatori sui quali è stata effettuata una `free()`, in modo che non puntino più all'area ora libera.
`free(punt);`
`punt = NULL;`




realloc()

- `void * realloc (void * punt, size_t newSize);`
- Cambia la dimensione (si può solo aumentarla) di un blocco di memoria allocata.
- Per maggiori informazioni consulta l'Help.



Tecniche di recupero della memoria (heap)

- La vita di un oggetto dinamico inizia dalla sua creazione (allocazione, `malloc()`, `calloc()`) e finisce con la sua distruzione (deallocazione, `free()`) o alla fine del programma.
- Ci sono due tecniche di recupero della memoria:
 - 1) a carico del programmatore (Pascal, C, C++, ...)
 - 2) garbage collector (Lisp, Eiffel, Java, ...)



C++

■ Operatori:

- ☐ new
- ☐ delete
- ☐ delete []

■ Esempi:

```
punt = new tipoDiDato;  
delete punt;  
puntVett = new tipoDiDato [const|var];  
delete [ ] puntVett;  
(v. disegni alla lavagna)
```



Esempio

```
typedef struct{  
    char nome [21];  
    char cognome [25];  
    char indirizzo [30];  
    char professione [15];  
}persona;
```

```
persona * p;  
int i;  
p=(persona *) malloc (numElem*sizeof(persona));  
if(!p)  
    printf("Memoria insufficiente\n");  
else {  
    for(i=0; i < numElem; i++) {  
        gets((p+i) → nome);  
        ....  
    }  
}
```



Problemi relativi alla programmazione tramite puntatori

- Produzione di spazzatura (garbage)
- Dangling references (riferimenti fluttuanti/pendenti)
- Effetti collaterali



Produzione di spazzatura (garbage)

- Esempio:
- Se in una funzione, dopo aver allocato (nello stack) un puntatore e un'area dinamica (nello heap, assegnando al puntatore l'indirizzo dell'area), dimentichi di deallocare l'area prima che la funzione termini, produci garbage.
- Perché?



Produzione di spazzatura (garbage)

Altro esempio:

```
int *p, q;  
p=new int;  
p=&q;
```

(v. disegno alla lavagna)



Dangling references

- Un puntatore che punta a locazioni di memoria non significative prende il nome di dangling reference.
- Es. `int * puntTemp; //puntatore non init.`
- Per evitare di utilizzarlo, prima di avergli assegnato un indirizzo valido, è bene inizializzarlo a NULL.



Dangling references

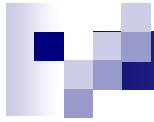
Altro esempio:

.....

p=q;

free(q);

(v. disegno alla lavagna)



Effetti collaterali

Esempio:

```
int *punt, *punt2;  
punt=punt2=new int;  
*punt2=4;  
delete punt;
```

E se ora io scrivo: `printf("%d",*punt2);` che cosa succede? (v. disegno alla lavagna)



Attenzione

- `int * num;`
- `delete num; //errore run time`

- `int * num=NULL;`
- `delete num; //nessun errore`



Esercizio

- Scrivi una funzione che legga una stringa, la allochi nello HEAP e faccia in modo che essa sia ancora disponibile al termine della funzione.



Soluzione n. 1

```
void funz (char *p) {  
    char str[80];  
    gets(str);  
    p=new char [strlen(str)+1];  
    strcpy(p,str);  
}
```

(v. disegno alla lavagna)

```
int main() {  
    char *p;  
    funz(p);  
    .....  
}
```

NO: anche se ho passato
un puntatore, il passaggio
è comunque per valore!!!
Ho inoltre creato garbage.



Soluzione n. 2

```
char * funz () {  
    char str[80];  
    char * p;  
        gets(str);  
        p=new char [strlen(str)+1];  
        strcpy(p,str);  
        return p;  
}
```

(v. disegno alla lavagna)

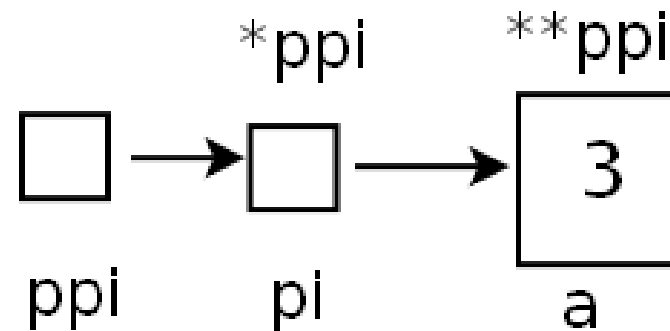
```
int main() {  
    char *p;  
        p=funz();  
        .....  
        delete [] p;  
}
```

OK!!!

Puntatore di puntatore

Un puntatore di puntatore è una variabile abilitata a mantenere l'indirizzo di una variabile puntatore.

```
int **ppi;  
int a = 3;  
int *pi;  
    pi = &a;  
    ppi = &pi;  
    printf("%d", **ppi);
```





Soluzione n. 3

```
void funz (char **p) {  
    char str[80];  
    gets(str);  
    *p=new char [strlen(str)+1];  
    strcpy(*p,str);  
}
```

(v. disegno alla lavagna)

```
int main() {  
    char *p;  
    funz(&p);  
    .....  
    delete [] p;  
}
```

OK!!!



Puntatori

- L-value: è l'indirizzo di memoria di un elemento
- R-value: è il valore memorizzato all'L-value.



Un errore frequente

```
int main() {  
    int x, *p;  
    x = 10;  
    *p = x;           //NO: non ho allocato l'area!!!  
}
```



Capire la differenza

- `char str[10]`
- `char str []`
- `char * str`



char str[10]

- `char str[10]="prova";` `//inizializzazione`
- `char str[10];`
- `str="prova";` `//NO: Il nome di un array è una costante!!!`
- `char str[10];`
- `strcpy(str,"prova");` `//assegnazione`



char str []

- `char str [] = "prova";`
- `char str[] = { 'p', 'r', 'o', 'v', 'a', '\0' };`



char * str

- `char * str;`
- `str = new char [6];`
- `strcpy(str, "prova");`

- `char * str="prova";` // deprecated conversion from string constant to 'char*'

- `char * str;`
- `gets(str);` //NO: non ho allocato l'area di memoria!!!



Vettori di puntatori

- `char mesi[12][10];` `//array di stringhe`
 `//matrice di caratteri`
- `char mesi[12][10]={“gennaio”, “febbraio”, ...};`

(v. disegno alla lavagna)

- Spreco di memoria



Vettori di puntatori

- mesi[2] è un (char *)
- gets(mesi[i]);
- printf(“%s\n”,mesi[i]); //puts(mesi[i]);



Vettori di puntatori

- `char * mesi[12];` //vettore di puntatori
- Allocazione dinamica delle stringhe contenenti i mesi.

```
char mese[10];
int i;
for(i=0; i<12; i++) {
    gets(mese);
    *(mesi+i)= new char[strlen(mese)+1];    // ≡ mesi[ i ] = .....
    strcpy(*(mesi+i),mese);
}
for(i=0; i<12; i++)
    printf("%s\n",mesi[i]);
```

Vantaggio: miglior sfruttamento della memoria , ho però lo spazio occupato dal vettore di puntatori.

N.B. `char **punt=mesi;`



Vettori di puntatori

```
char * mesi[]={"Gennaio", "Febbraio", "Marzo", "Aprile", "Maggio",  
               "Giugno", "Luglio", "Agosto", "Settembre", "Ottobre",  
               "Novembre", "Dicembre" };
```

//solo in fase di inizializzazione

//deprecated conversion from string constant to 'char*'

```
printf("%d\n",sizeof(mesi));           //48 byte
```

```
printf("%d\n",sizeof(mesi[2]));        //4 byte
```

```
printf("%d\n",sizeof(*mesi[2]));       //1 byte
```

```
printf("%d\n",strlen(mesi[2]));        //5 byte
```



Vettori

| | Chiamata | Parametri formali |
|----------------|----------|-----------------------------|
| int vett [10] | vett | int vett[] int * vett |
| int * vett[10] | vett | int * vett[] int **vett |



Vettori di puntatori

- `int * vett[10];`

- `vett[5]` \rightarrow è un `(int *)`

- `*(vett[5])` \rightarrow è un `(int)`



Esercizio

- Scrivere una funzione a cui viene passato un array di puntatori a double e restituisce il maggiore fra gli n di tali double.



Soluzione

```
int main() {  
    double * vett[MAX_DIM];  
    int numElem;  
    double massimo;  
  
    insert(vett,&numElem);  
    massimo=max(vett,numElem);  
  
    printf("Max = %f\n",massimo);  
  
    getchar();  
    return 0;  
}
```




Soluzione

```
void insert(double * vett[ ], int *numElem){
char num[10];
int i;

do {
    printf("Quanti elementi vuoi inserire ?\n");
    *numElem=atoi(gets(num));
}while(*numElem < 0 || *numElem > MAX_DIM);

for(i=0; i<*numElem; i++) {
    vett[i]=new double;
    printf("Digita un numero reale\n");
    *vett[i]=atof(gets(num));
}
}
```



Soluzione

```
double max(const double * vett[ ], int numElem) {  
    double massimo;  
    int i;  
  
    massimo=*vett[0];  
    for(i=1; i<numElem; i++)  
        if(*vett[i] > massimo)  
            massimo=*vett[i];  
    return massimo;  
}
```



Parametri della funzione main()

- Poiché i parametri del main() vengono passati tramite la linea di comando che manda in esecuzione il programma, essi rappresentano uno strumento di comunicazione fra il S.O., che interpreta il comando, ed il pgm che viene mandato in esecuzione.



Parametri della funzione main()

- Due sono i possibili parametri:
 - **argc**: indica il numero di parametri che sono stati immessi nella linea di comando (compreso il nome del pgm stesso);
 - **argv**: rappresenta una lista di stringhe ognuna delle quali, ordinatamente, contiene un parametro (compreso il nome del pgm stesso). Inoltre, il parametro argv, dato che rappresenta una struttura di lunghezza variabile, viene terminato mediante una stringa nulla.

int main(int argc, char * argv[])

Parametri della funzione main()

■ Esempio (linea di comando):

> prova 30 ciao 5 a pippo

Valore di argc → 6

Valore di argv[0] → prova

Valore di argv[1] → 30 //E' una stringa!

Valore di argv[2] → ciao

Valore di argv[3] → 5 //E' una stringa!

Valore di argv[4] → a //E' una stringa!

Valore di argv[5] → pippo

Valore di argv[6] → la stringa nulla ('\0')

↙ argv[4][0] oppure *(argv[4])

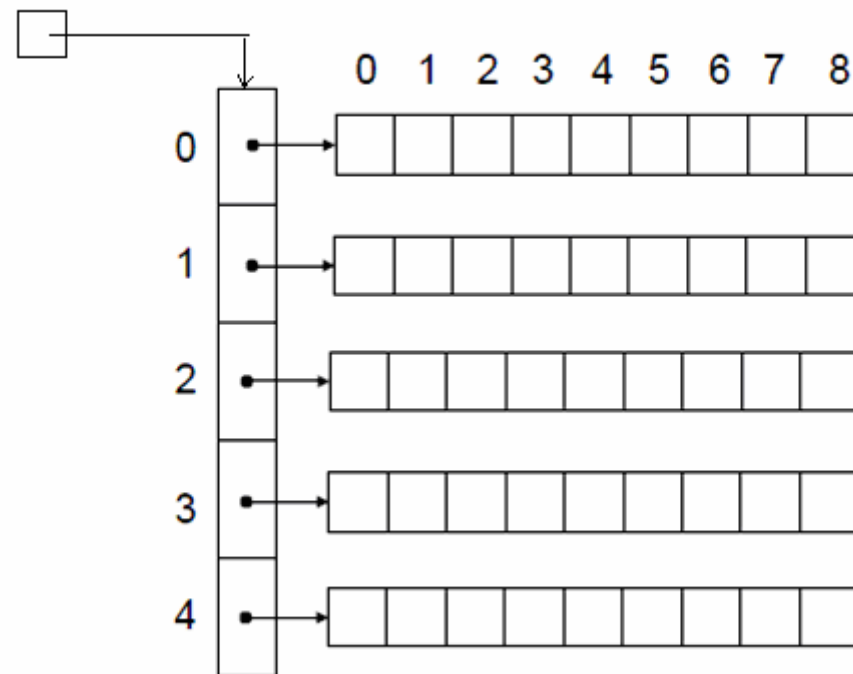


Matrici – Puntatori – Allocazione dinamica

■ ALLOCAZIONE DINAMICA DI UNA MATRICE

- Possiamo rappresentare una matrice come un vettore di vettori. Cioè l'elemento i -esimo di un vettore è un vettore che rappresenta le colonne della matrice.
- Per costruire una matrice dinamica di r righe e c colonne basterà eseguire le seguenti due operazioni:
 - 1. Allocare dinamicamente un vettore di r puntatori (vettore delle righe).
 - 2. Per ogni riga allocare un vettore di c elementi del tipo base considerato (vettore colonne).

- La figura mostra la rappresentazione grafica di una matrice dinamica.





- 1^ versione: (non usare le funzioni)
 - Main: Richiesta dimensioni (r, c) → Allocazione → Inserimento dati → Stampa → Deallocazione
- 2^ versione: (usare le funzioni)
 - Richiesta dimensioni (r, c) e allocazione
 - Inserimento dati
 - Stampa
 - Deallocazione
- 3^ versione: uso typedef e passaggio by reference (C++)



Appendice

■ Vettori

$$\square \text{array}[i] \equiv *(\text{array} + i)$$

■ Matrici

$$\square \text{array}[i][j] \equiv *(\text{array}[i] + j) \equiv *((*(\text{array} + i)) + j)$$



- Normalmente per accedere ad una componente di una matrice scrivo:
 - **`a[i][j]`**

- Se la matrice viene allocata (staticamente o dinamicamente) con un vettore unidimensionale (per righe) per accedere ad una componente scrivo:
 - **`a[i*DIM_COLONNE+j]`**
(variabile numColonne se è dinamica)



Esercizio svolto

Si assuma che una matrice sia memorizzata in un file per righe, in accordo al seguente formato:

$r \quad c \quad a_{11} \ a_{12} \ \dots \ a_{1n} \ a_{21} \ a_{22} \ \dots \ a_{2n} \ \dots \ a_{m1} \ a_{m2} \ \dots \ a_{mn}$

con a_{ij} elemento generico della matrice.

Scrivere un programma che:

- legge una matrice di double da un file di ingresso (che si assume nel formato corretto);
- scrive su standard output il valore della somma degli elementi della matrice;
- scrive in un file di uscita la matrice trasposta.

I nomi dei file di ingresso e di uscita sono letti da standard input e sono costituiti da un numero di caratteri pari ad, al più, MAX_LUN_NOME.



```
const int M = 10;
const int MAX_LUN_NOME = 30;
int main() {
    fstream ingr, usc;
    char nomeingr[MAX_LUN_NOME], // stringhe per
    nomeusc[MAX_LUN_NOME];       // i nomi dei file
    double m[M][M], t[M][M];    // la matrice e la trasposta
    int r, c;

    cin >> nomeingr >> nomeusc; // si leggono i nomi dei file

    ingr.open(nomeingr, ios::in); // si aprono i file in lettura e scrittura, rispettivamente
    usc.open(nomeusc, ios::out);

    ingr >> r >> c;

    for (int i = 0; i < r; i++)
        for (int j = 0; j < c; j++)
            ingr >> m[i][j];
    ingr.close();

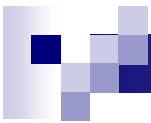
    // Si legge la matrice dal file d'ingresso
    // 1) si leggono il numero di righe ed il numero di colonne
    // 2) si leggono, per riga, gli elementi della matrice
```



```
// Si scrive la somma degli elementi della matrice su standard. output
cout << sommaElementi((double *)m, r, c);
// Si calcola la trasposta
trasposta((double *)m, (double *)t, r, c);
// Si salva la matrice trasposta sul file di uscita
// 1) Si salvano il numero di riga e di colonna
usc << c << r;
// 2) Si salvano, per riga, gli elementi della matrice
for (int i = 0; i < c; i++)
    for (int j = 0; j < r; j++)
        usc << t[i][j];
usc.close(); // Il file di uscita non serve più
}
```



```
double sommaElementi(double *m, int r, int c) {  
    double sommaElem = 0.0;  
    for (int i = 0; i < r; i++)  
        for (int j = 0; j < c; j++)  
            sommaElem += *(m + i * M + j);  
    return sommaElem;  
}
```



```
void trasposta (double *m, double *t, int r, int c) {  
    for (int i = 0; i < r; i++)  
        for (int j = 0; j < c; j++)  
            *(t + j * M + i) = *(m + i * M + j);  
}
```