Puntatori e allocazione dinamica

PASSAGGIO PER RIFERIMENTO: RIFLESSIONI

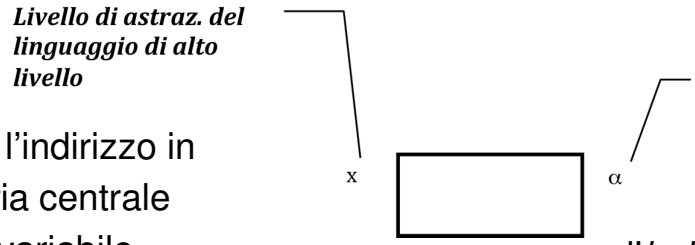
- Il passaggio per riferimento consente di superare i limiti della semantica per copia.
- Quando un parametro è passato per riferimento, la procedura (o funzione) riceve *non una copia del valore* del parametro attuale, ma *un riferimento a esso*.
- Nella corrispondente cella di memoria viene perciò inserito *l'indirizzo* della variabile che costituisce il parametro attuale, non il suo valore.

Gestire il passaggio per riferimento implica la capacità di *accedere* direttamente *agli indirizzi* delle variabili!

ACCESSO agli INDIRIZZI delle VARIABILI

Il C prevede meccanismi per:

- **ricavare l'indirizzo** di una variabile
  - operatore di **estrazione di indirizzo**: **&**
- **dereferenziare un indirizzo** di variabile, ossia "recuperare" la variabile dato il suo indirizzo.
  - operatore di **dereferenziamento**: *



Se α è l'indirizzo in memoria centrale di una variabile, *α denota tale variabile:

*α ≡ x

Se x è una variabile, &x denota l'*indirizzo in memoria centrale* di tale variabile:

&x ≡ α

PUNTATORI

- Un **puntatore** è il costrutto linguistico introdotto dal C (e da molti altri linguaggi) come forma di accesso alla macchina sottostante.

VARIABILI PUNTATORE (a un TIPO)

Un PUNTATORE (a T) è una variabile di tipo puntatore (a T) che può contenere l'indirizzo di una variabile (di tipo T).

```
<tipo> *    <nomeVariabile>
```

ESEMPIO

```
int  *p ;
int* p ;
```

oppure

```
typedef int* puntint;
puntint     p ;
```

PUNTATORI (cont.)

- Un puntatore è una variabile *destinata a contenere l'indirizzo di un'altra variabile*.
- Un puntatore *non può contenere un indirizzo qualunque*: esiste un vincolo di tipo.
- La sintassi traduce tale vincolo consentendo di definire *non un "puntatore qualunque"*, ma un *puntatore a un certo tipo T.*

ESEMPIO

```
int x = 8;  int *p;
p = &x;                              /* OK */
```

Da questo momento, *p e x sono due modi diversi di denotare *la stessa variabile.* Quindi si potrà, ad esempio, scrivere:

```
*p = 51;  x--;
```

Il valore di `*p` alias `x` è ora 50.

Operatore di deferenziamento "*"

- **Applicato ad una variabile puntatore fa riferimento all'oggetto puntato**

Esempio:

```
int *pi;        /* dichiarazione di un puntatore ad intero */
int a = 5, b;   /* dichiarazione di variabili intere */
pi = &a;        /* pi punta ad a ⇒ *pi è un altro modo di denotare a */
b = *pi;        /* assegna a b il valore della variabile puntata da pi,
                   ovvero il valore di a, ovvero 5 */
*pi = 9;        /* assegna 9 alla variabile puntata da pi, (a) */
```

- **N.B. Se `pi` è di tipo `int *`, allora `*pi` è di tipo `int`**
- **Non confondere le due occorrenze di "`*`":**

"`*`" in una dichiarazione serve per dichiarare una variabile di tipo puntatore

Es.: `int *pi;`

"`*`" in una espressione è l'operatore di dereferenziamento

Es.: `b = *pi;`

Operatori di deferenziamento ed indirizzo

- hanno priorità più elevata degli operatori binari
- "`*`" è associativo a destra
  - Es.: `**p` è equivalente a `*(*p)`
- "`&`" può essere applicato solo ad una variabile
  - `&a` non è una variabile ⇒ "`&`" non è associativo
  - "`*`" e "`&`" sono uno l'inverso dell'altro

- data la dichiarazione `int a;`
  - `*&a` è un modo alternativo per denotare `a` (sono entrambi variabili)
- data la dichiarazione `int *pi;`
  - `&*pi` ha valore (un indirizzo) uguale al valore di `pi`
  - però: `pi` è una variabile

    `&*pi` non lo è (ad esempio, non può essere usato a sinistra di "`=`")

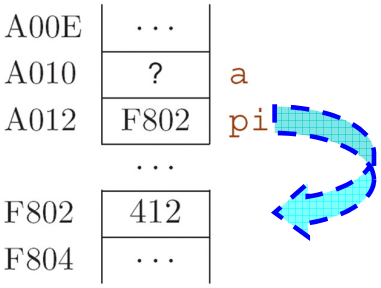
Inizializzazione di puntatori

- I puntatori (come tutte le altre variabili) devono essere inizializzati prima di poter essere usati

  ⇒ E' un errore dereferenziare una variabile puntatore non inizializzata

Esempio:

```
int a;
int *pi;
```



`a = *pi;` ⇔ ad `a` viene assegnato il valore 412

`*pi = 500;` ⇔ scrive 500 nella cella di memoria di indirizzo F802

Non sappiamo a cosa corrisponde questa cella di memoria!!!

⇒ la memoria può venire corrotta

PUNTATORI e COMPATIBILITA' DI TIPO

Un puntatore a T può contenere solo indirizzi di variabili di tipo T:

tipi di puntatori diversi sono *incompatibili* fra loro!

ESEMPIO

```
int x = 8;   float *q;
q = &x;                                  /* NO! */
```

oppure

```
int x = 8, *p;   float *q;
p = &x;
q = p;                                   /* NO! */
```

MOTIVO: l'informazione sul tipo *del puntatore* serve a dedurre il tipo *dell'oggetto puntato*, che è *indispensabile* per effettuare correttamente il dereferenziamento.

Operazioni sui puntatori

Sui puntatori si possono effettuare diverse operazioni

- **di dereferenziamento**

Esempio:

```
int *p, i;
...
i = *p;
```

  - **Il valore della variabile intera `i` è ora lo stesso del valore dell'intero puntato da `p`**

- **di assegnamento**

Esempio:

```
int *p, *q;
...
p = q;
```

  - **N.B. `p` e `q` devono essere dello stesso tipo (altrimenti bisogna usare l'operatore di cast)**
  - **Dopo l'assegnamento precedente, `p` punta allo stesso intero a cui punta `q`.**

- **di confronto**

Esempio: `if (p == q) ...`

Esempio: `if (p > q) ...`

  - **Ha senso? Finora no. Vedremo tra poco che ci sono situazioni in cui ha senso**

Aritmetica dei puntatori

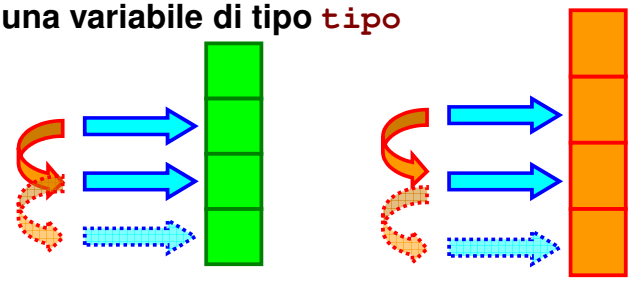
- Sui puntatori si possono anche effettuare operazioni aritmetiche, con opportune limitazioni
  - somma o sottrazione di un intero
  - sottrazione di un puntatore da un altro

Somma e sottrazione di un intero

- Se `p` è un puntatore a `tipo` e il suo valore è un certo indirizzo `ind`, il significato di `p+1` è il primo indirizzo utile dopo `ind` per l'accesso e la corretta memorizzazione di una variabile di tipo `tipo`

Esempio:

```
int *p, *q;
....
q = p+1;
```



- Se il valore di `p` è l'indirizzo `100`, il valore di `q` dopo l'assegnamento è `104` (assumendo che un intero occupi 4 byte)

Aritmetica dei puntatori

- Quindi il valore calcolato in corrispondenza di un'operazione del tipo `p+i` dipende dal tipo di `p` (analogamente per un'operazione del tipo `p-i`)

Esempio:

```
int *pi;
*pi = 15;
pi=pi+1; ⇒ pi punta al prossimo int (4 byte dopo)
*pi = 20;
```

Esempio:

```
double *pd;
*pd = 12.2;
pd = pd+3; ⇒ pd punta a 3 double dopo (24 byte dopo)
```

Esempio:

```
char *pc;
*pc = 'A';
pc = pc - 5; ⇒pc punta a 5 char prima (5 byte prima)
```

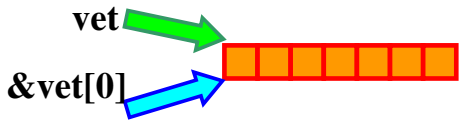
- Si può anche scrivere:

```
pi++;
pd += 3;
pc -= 5;
```

Relazione fra vettori e puntatori

- In generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella
  - L'unico caso in cui sappiamo quali sono le locazioni di memoria successive e cosa contengono è quando utilizziamo dei vettori

- In C il nome di un vettore è in realtà un puntatore, inizializzato all'indirizzo dell'elemento di indice 0



Esempio: `int vet[10];`
  - `vet` e `&vet[0]` hanno lo stesso valore (un indirizzo)
  - `printf("%p %p", vet, &vet[0]);` stampa 2 volte lo stesso indirizzo

- Possiamo far puntare un puntatore al primo elemento di un vettore

Esempio:
```
int vet[5];
int *pi;
pi = vet;
```
è equivalente a `pi = &vet[0];`

Accesso agli elementi di un vettore

Esempio:
```
int vet[5];
int *pi = vet;
*(pi + 3) = 28;
```
`pi+3` punta all'elemento di indice 3 del vettore (il quarto elemento)

  - 3 viene detto offset (o scostamento) del puntatore

- N.B. Servono le "`()`" perchè "`*`" ha priorità maggiore di "`+`"
  - Che cosa denota `*pi + 3` ?

- Osservazione:
  - `&vet[3]` equivale a `pi+3` equivale a `vet+3`
  - `*&vet[3]` equivale a `*(pi+3)` equivale a `*(vet+3)`

- Inoltre, `*&vet[3]` equivale a `vet[3]`
  - In C, `vet[3]` è semplicemente un modo alternativo di scrivere `*(vet+3)`

- Notazioni per gli elementi di un vettore:
  - `vet[3]` .... notazione con puntatore e indice
  - `*(vet+3)` .... notazione con puntatore e offset

Accesso agli elementi di un vettore

Riassumendo, si può accedere agli elementi di un vettore nei modi seguenti:

Esempio:

```
int vet[5] = {11, 22, 33, 44, 55};
int *pi = vet;
int offset = 3;

vet[offset] = 88;
*(vet + offset) = 88;
pi[offset] = 88;
*(pi + offset) = 88;
```

- **A differenza di un normale puntatore, il nome di un vettore è un puntatore costante ⇒ il suo valore non può essere modificato!**

```
int vet[10];
int *pi;
pi = vet; /* OK: pi assume l'indirizzo del primo elemento di vet */
pi++;     /* OK: pi assume l'indirizzo del secondo elemento di vet */
vet++;    /* NO: vet e' un puntatore costante! */
```


Modi alternativi per percorrere un vettore

```
int a[LUNG]= {.......};
int i, *p=a;
```

- **I seguenti sono tutti modi equivalenti per stampare i valori di `a`**

```
for (i=0; i<LUNG; i++)
     printf("%d", a[i]);
for (i=0; i<LUNG; i++)
     printf("%d", *(a+i));
for (p=a; p<a+LUNG; p++)
     printf("%d", *p);
```

- **Non è invece lecito un ciclo del tipo**

```
for (a; a<p+LUNG; a++)
       printf("%d", *a);
```

  - **perchè `a` non può essere modificato**

Differenza fra puntatori

- **Il parallelo tra vettori e puntatori ci consente di capire il senso di un'operazione del tipo `p-q` dove `p` e `q` sono puntatori allo stesso tipo**

Esempio:

```
int *p, *q;
int a[10]={0};
int x;
...
x=p-q;
```

- **Il valore di `x` è il numero di interi compresi tra l'indirizzo `p` e l'indirizzo `q`**

- **Quindi se nel codice precedente `...` ci sono le istruzioni:**

```
q = a;
p = &a[5];
```

- **il valore di `x` dopo l'assegnamento è 5**

PUNTATORI e STRUTTURE

Per le strutture, l'accesso all'indirizzo avviene nella maniera consueta.

ESEMPIO

```
typedef struct {
      char nome[20], cognome[20];
      int eta;} persona;
persona *pp;
persona p;
pp = &p;
```

PUNTATORI e STRUTTURE

È quindi possibile accedere a un singolo membro di un puntatore `p` a struttura attraverso gli operatori di dereferenziamento:

```
(*p).membro
```

dove le parentesi sono necessarie perché la precedenza di `.` è superiore a quella di `*`

```
p->membro
```

ESEMPIO

```
++ pp->eta; /* incrementa il val. di età */
pp->nome[0] /* accede al primo carattere */
```

NOTA `++pp->eta` = `++(pp->eta)`, in quanto la precedenza di `->` è superiore a quella di `++`.

DICHIARAZIONI STATICHE e VETTORI

- In C, le variabili fin qui viste devono essere **dichiarate staticamente**, ossia la loro esistenza e il loro nome devono essere *previsti a priori.*
- Mentre per le variabili di tipo scalare ciò non costituisce di norma un problema, può esserlo per variabili di tipo *vettore*.
- Infatti, in C *la dimensione di un vettore deve essere dichiarata come costante nota a priori*:

```
int v[4];
char nome[20];
```

- Ciò rappresenta un limite in quei casi in cui *è impossibile sapere a priori la quantità di dati in ingresso*.

ALLOCAZIONE DINAMICA della MEMORIA

PIU' IN GENERALE per superare la rigidità della dichiarazione statica, occorre un modo per "chiedere al sistema nuova memoria" *al bisogno*.

Questo è possibile grazie al concetto di **allocazione dinamica***!*

Come molti altri linguaggi, anche il C consente di "chiedere nuova memoria" *al momento del bisogno*, tramite una *primitiva di sistema*:

La funzione `malloc()`:

- chiede al sistema di **allocare un'area di memoria della dimensione specificata**
- restituisce l'**indirizzo dell'area di memoria allocata**
- Se l'allocazione non è stata possibile, `malloc()` restituisce un **puntatore NULL (0)**, *che segnala l'errore.*

ALLOCAZIONE DINAMICA della MEMORIA

Problema: Leggere e memorizzare N elementi interi positivi, ed in seguito calcolarne il massimo

Soluzione con vettori (I)

```
#include <stdio.h>
#define DIM 10
main()
{
  int a[DIM], i, max;
  for (i = 0; i < DIM; i++)
       scanf("%d", &a[i]);
  max = a[0];
  for (i = 1; i < DIM; i++)
       if (a[i] > max) max = a[i];
  printf("%d", max);
}
```

Limiti: il numero di interi in input deve essere pari al valore di DIM

ALLOCAZIONE DINAMICA della MEMORIA

Problema: Leggere e memorizzare N elementi interi positivi, ed in seguito calcolarne il massimo

Soluzione con vettori (II)

```
#include <stdio.h>
#define DIM 1000
main()
{
  int a[DIM], i=0, max;
  scanf("%d", &a[0]); i++;
  while ((i < DIM) || (a[i-1]<=0)){
       scanf("%d", &a[i]);i++;
  }
  max = a[0];
  for (j = 1; j < i; j++)
       if (a[j] > max) max = a[j];
  printf("%d", max);
}
```

Limiti: il numero di interi in input deve essere minore od uguale al valore di DIM

ALLOCAZIONE DINAMICA della MEMORIA

Problema: Leggere e memorizzare N elementi interi positivi, ed in seguito calcolarne il massimo

?

Limiti: nessun limite, se non quello della memoria del computer

ALLOCAZIONE DINAMICA della MEMORIA (cont.)

Nel codice del programma C occorre quindi:

- specificare la dimensione dell'area desiderata (*in byte*)
- mettere *in un puntatore* l'indirizzo restituito da `malloc()`

Inoltre, poiché `malloc()` restituisce *un puro indirizzo*, senza informazioni di tipo, **è indispensabile un cast** per "etichettare" l'area di memoria come contenente dati di un certo tipo.

ESEMPIO

Ad esempio, per allocare spazio per cinque interi:

```
int *p;
p  =  (int*) malloc (5*sizeof(int));
```

NOTA

Per specificare la dimensione, è opportuno *non inserire mai valori numerici direttamente*, ma utilizzare invece l'apposito **operatore `sizeof()`.**

Funzione "sizeof"

- **La funzione `sizeof` restituisce l'occupazione in memoria in byte di una variabile o di un tipo.**
  - **Può essere applicata anche ad un tipo puntatore**
  - **Tutti i puntatori sono indirizzi ⇒ occupano lo spazio di memoria di un indirizzo**
  - **L'oggetto puntato ha invece la dimensione del tipo puntato**

Esempio:

```
char *pc;
int *pi;
double *pd;
printf("%d %d %d ", sizeof(pc), sizeof(pi), sizeof(pd));
printf("%d %d %d\n", sizeof(char *), sizeof(int *), sizeof(double *));
printf("%d %d %d ", sizeof(*pc), sizeof(*pi), sizeof(*pd));
printf("%d %d %d\n", sizeof(char), sizeof(int), sizeof(double));
```

```
4 4 4 4 4 4
1 2 8 1 2 8
```

ALLOCAZIONE DINAMICA della MEMORIA (cont.)

Attraverso l'istruzione

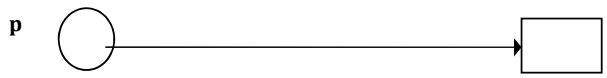
```
p = (<type>*) malloc(N*sizeof(<type>));
```

si ottiene un vettore dinamico di N elementi di tipo `<type>`!

- L'area di memoria allocata è usabile tramite il puntatore. Le due notazioni equivalenti sono:
  - o tramite la notazione `*p`
  - o tramite la notazione `p[i]`

ALLOCAZIONE DINAMICA della MEMORIA (cont.)

- Nel caso in cui **N**=1, si alloca spazio **per una singola variabile.**




```
int *p  =  (int*) malloc (sizeof(int));
```

- Quando non è più necessaria, l'area allocata dovrà essere **esplicitamente liberata** tramite la **funzione** `free()` della libreria **stdlib.h**:

```
free(p);
```

ALLOCAZIONE DINAMICA DELLA MEMORIA e STRUTTURE

- L'allocazione dinamica di una variabile di tipo struttura avviene nel modo consueto.

ESEMPIO

Dato
```
typedef struct {
      char nome[20], cognome[20];
      int eta;
      long stipendio;} persona;
```

All'interno di una qualsiasi funzione posso allocare dinamicamente un vettore di tipo persona

```
persona *p;
p =  (persona*) malloc (2*sizeof(persona));
```

- L'area allocata è utilizzabile tramite uno degli operatori di dereferenziamento (*, [] o, ->)

Esempio 1

Mostrare la differenza fra un vettore di interi *definito staticamente* e un vettore di interi *allocato dinamicamente*.

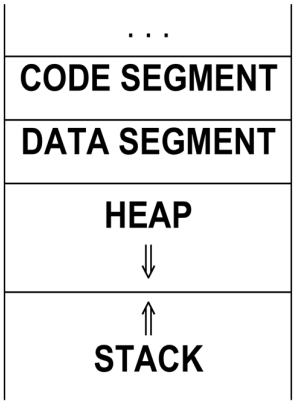
```
#include <stdio.h>
#include <stdlib.h>
main(){
 int *dynVect;       /* spazio per il punt. */
 int k, i;
 printf("Inserire la dimensione\n");
 scanf("%d", &k);
 dynVect = (int *) malloc(k * sizeof(int));

/*ora posso usare liberamente dynVect come vettore lungo k*/
/* i primi k quadrati */
 for (i=0;i<k;i++)
       dynVect[i] = i*i;
 free(dynVect);}
```

L'area Heap

- La memoria allocata dinamicamente viene allocata in un'apposita area, *distinta da tutte le altre fin qui viste:* **l'area heap**.

- La memoria allocata nello heap con `malloc()` resta riservata fino a che non viene esplicitamente deallocata con `free()`.
- In particolare, un'area allocata dentro a una funzione *sopravvive alla funzione che l'ha allocata*, ed è disponibile in tutto il programma (fino a quando non viene distrutta).



Esempio 2

Scrivere una funzione che inserisce in un vettore un numero n (non fissato a priori) di valori e restituisca al main il puntatore al vettore creato e la sua dimensione.

L'INTESTAZIONE DELLA FUNZIONE

La funzione può restituire il vettore creato *come risultato* e la dimensione come parametro passato per riferimento.

```
int* creaVett(int* pnum);
```

L'IMPLEMENTAZIONE DELLA FUNZIONE

```
#include <stdio.h>
#include <stdlib.h>
int* creaVett(int* pnum){
 int i;
 int *v;

 printf("Quanti valori? ");
 scanf("%d", pnum);
 v = (int*) malloc(*pnum * sizeof(int));
 for (i=0; i< *pnum; i++){
       printf("v[%d] = ", i);
   scanf("%d", v+i);}
 return v;}
```

Esempio 3

Scrivere una procedura per visualizzare un vettore di interi e aggiungerla al `main()` in modo da vedere il vettore creato.

L'INTESTAZIONE DELLA FUNZIONE

```
void mostraVett(int* v, int dim);
```

L'IMPLEMENTAZIONE DELLA FUNZIONE

```
#include <stdio.h>
#include <stdlib.h>
void mostraVett(int* v, int dim){
 int i;
 for (i=0; i<dim; i++)
       printf("v[%d] = %d\n", i, v[i]);}
```

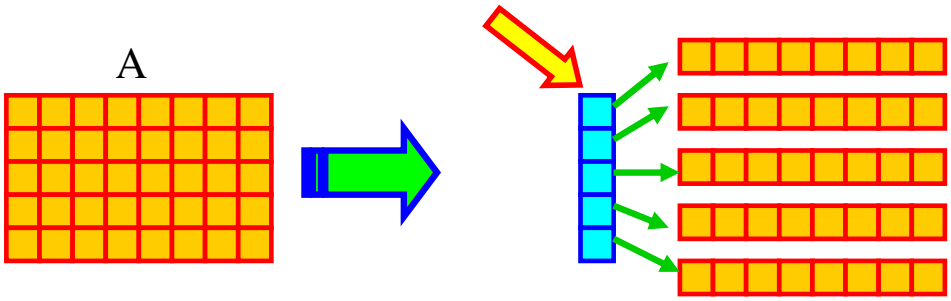
IL CLIENTE

```
main(){
 int *pv, num;
 pv = creaVett(&num);
 mostraVett(pv, num);
 free(pv);}
```


ALLOCAZIONE DINAMICA della MEMORIA (matrice bidimensionale)

Problema: Allocare correttamente una matrice bidimensionale

- **Una matrice bidimensionale può essere vista come un vettore di vettori**
  - **in altre parole, come un vettore di puntatori, che puntano ognuno ad un vettore**




```
#define DIM1 5
#define DIM2 8
int main()
{int a[DIM1][DIM2];
 ...
}
```

```
#define DIM1 5
#define DIM2 8
int main()
{int **a;
 ...
}
```

ALLOCAZIONE DINAMICA della MEMORIA (matrice bidimensionale)

Problema: Allocare correttamente una matrice bidimensionale

- Una matrice bidimensionale può essere vista come un vettore di vettori
  - in altre parole, come un vettore di puntatori, che puntano ognuno ad un vettore

```
#include <stdio.h>
#define DIM1 5
#define DIM2 8
main()
{ int **a,i;
  ...
  a=(int **)calloc(DIM1, sizeof(int *));
  for (i = 0; i < DIM1; i++)
      a[i]=(int *)calloc(DIM2,sizeof(int));
  ...
  a[i][j]=i*j;
  ...
}
```