Puntatori in C

Lucidi della Pof.ssa Pazienza

http://www.uniroma2.it/didattica/FOI2/

Puntatori

- L’operatore di indirizzo &
- Indirizzi, puntatori
- Aritmetica dei puntatori
- L’operatore di dereferenziazione *

Operatore di indirizzo &

L'**operatore unario** di **indirizzo** **&** restituisce l'indirizzo della locazione di memoria dell'**operando**

Il valore restituito non va usato come **l-value**

(in quanto l'indirizzo di memoria di una variabile non può essere assegnato in un'istruzione, ma è predeterminato)

Operatore di indirizzo &

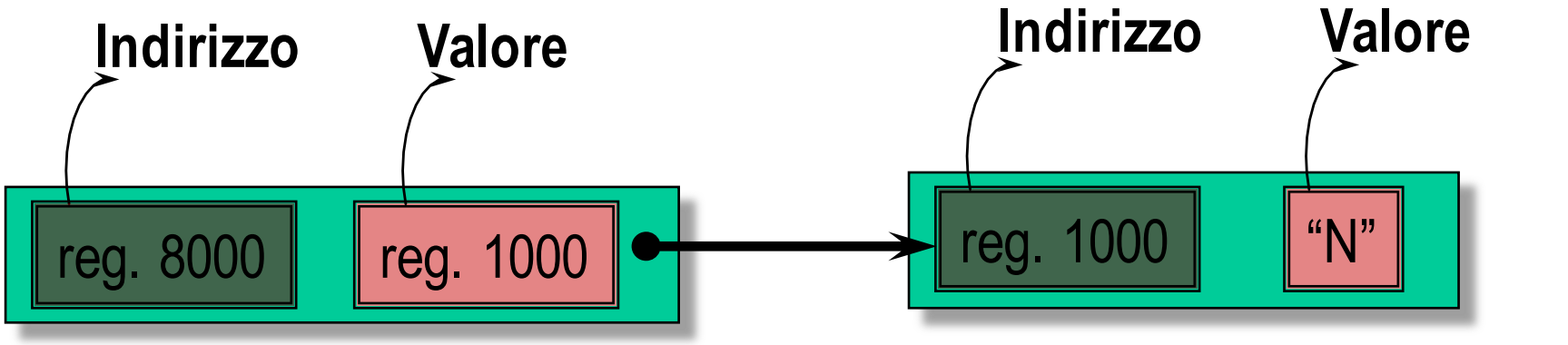
Es.:

&a	ammesso
&(a+1)	non ammesso
&a = b	non ammesso

Perché? (in quanto l'indirizzo di memoria di una variabile non può essere modificato in un'istruzione, ma solo usato come riferimento in quanto è predeterminato e non modificabile)

Puntatori

Una variabile di **tipo puntatore** é designata a contenere l'indirizzo di memoria di un'altra variabile (detta **variabile puntata**), la quale a sua volta può essere di qualunque **tipo**, anche non **nativo** (persino un altro **puntatore**!).



Variabile dato di tipo **puntatore**

Variabile dato di tipo **carattere**

Dichiarazione di variabile puntatore

Esistono puntatori per ogni tipo di variabile puntata

Un dato **puntatore** può puntare solo a un determinato **tipo** di variabili, quello specificato nella dichiarazione

Si usa il **qualificatore di tipo** (*suffisso*) *

Es. : **int* pointer;**

dichiara la variabile **pointer**, **puntatore** *ad una qualunque variabile di tipo* **int**

Si può anche dichiarare un **puntatore** a **puntatore**.

Es.: **double** pointer_to_pointer;**

Puntatore `void`

Un puntatore `void` è un puntatore generico che accetta come valore un indirizzo ma non punta ad un tipo dato ben determinato.

Il puntatore `void` si riferisce ad oggetti di tipo diverso

```
int i; int* pi=&i;
char c; char* pc=&c;
void* tp;
tp=pi;
*(int*) tp=3;
tp=pc;
*(char*) tp='c';
```

Assegnazione di valore a un puntatore

NON si possono assegnare **valori** a un **puntatore**, salvo che in questi tre casi:

1. a un **puntatore** é assegnato il valore **NULL** (non punta a “niente”)
2. a un **puntatore** è assegnato l’**indirizzo** di una variabile esistente, restituito dall’operatore **&** (Es. **int a; int* p; p=&a;**)
3. é eseguita un’operazione di **allocazione dinamica della memoria**

Operazioni sui puntatori

Operazioni fondamentali:

Assegnazione di valore (indirizzo) ad un puntatore

int* pi = &i p = q

Riferimento all'oggetto puntato *pi=3

Confronto tra puntatori (==, !=)

Operazioni sui puntatori

E' possibile specificare che un dato puntatore non deve essere usato per modificare l'oggetto puntato attraverso la parola chiave `const`:

```
int i;
const int* p=&i;
```

Le istruzioni in cui si usa *p* per aggiornare *i* vengono segnalate come erronee

Operazioni sui puntatori

E' possibile specificare con la parola chiave `const` che un puntatore non deve essere modificato:

```
int i;
int* const p=&i;
```

Le istruzioni in cui si vuole modificare il *valore di p* vengono segnalate come erronee

Aritmetica dei puntatori (1)

Il **valore** assunto da un **puntatore** é un numero **intero** che rappresenta, in **byte**, un **indirizzo** di memoria

le operazioni di **somma** fra un **puntatore** e un **valore intero** (con risultato **puntatore**), oppure di **sottrazione** fra due **puntatori** (con risultato **intero**) *vengono eseguite tenendo conto del* ***tipo*** *della variabile puntata*

Es.: se si incrementa un **puntatore-**a-**float** di 3 unità, il suo valore viene incrementato di 12 byte.

Aritmetica dei puntatori (2)

Se l'espressione *p* rappresenta l'indirizzo di un oggetto di tipo T, allora l'espressione *p+1* rappresenta l'indirizzo di un oggetto di tipo T allocato *consecutivamente* in memoria.

In generale: se *i* è un intero, se p rappresenta l'indirizzo `addr` di *T* che occupa *n* locazioni di memoria, allora l'espressione `p+i` ha valore `addr+ n×i`

Aritmetica dei puntatori (3)

Le regole dell'**aritmetica** dei **puntatori** assicurano che il risultato sia sempre corretto, qualsiasi sia la lunghezza in byte della variabile puntata.

Es.: se **p** punta a un **elemento** di un **array**, **p++** punterà all'elemento successivo, qualunque sia il **tipo** (anche non **nativo**) dell'array

L'operatore di dereferenziazione *

L'**operatore unario** di **dereferenziazione** * di un **puntatore** restituisce il valore della variabile puntata dall'**operando**:

come **r-value**, esegue un'operazione di **estrazione**

Es.: **a = *p;** assegna ad **a** il valore della variabile puntata da **p**

come **l-value**, esegue un'operazione di **inserimento**

Es.: ***p = a;** assegna il valore di **a** alla variabile puntata da **p**

L'operatore di dereferenziazione *

l'operazione di **deref.** é inversa a quella di **indirizzo**.

Se assegniamo a un **puntatore p** l'**indirizzo** di una variabile **a**,

p = &a;

allora

***p == a**

cioè la **deref.** di **p** coincide con **a**

L'operatore di dereferenziazione *

ATTENZIONE: non é detto il contrario !!!

se si assegna alla **deref.** di **p** il valore di **a**,

***p = a ;**

ciò non comporta automaticamente che in **p** si ritrovi l'**indirizzo** di **a**, ma semplicemente che il valore della variabile puntata da **p** coinciderà con **a**

Puntatori e riferimenti

Un riferimento è un nome alternativo per un oggetto e va sempre inizializzato

Un riferimento è un puntatore costante che viene dereferenziato ogni volta che viene utilizzato

```
void g()
{
int ii=0;
int& rr=ii;      // inizializzazione
rr++;            // ii viene incrementato di 1
int *pp=&rr;     // pp punta ad ii
```

Attenzione!

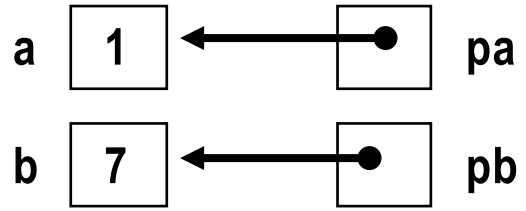
la dichiarazione di un **puntatore** comporta allocazione di memoria per la **variabile puntatore**, ma non per la **variabile puntata**.

Es.: **int* lista;**

alloca memoria per **lista** ma non per la variabile puntata da **lista**

Esempio

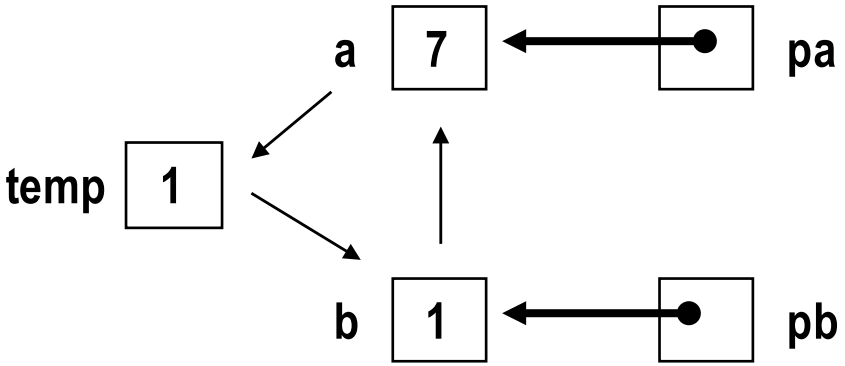
```
int a = 1;
int b = 7;
int* pa = &a;
int* pb = &b;
```




```
int temp;
temp = a;
a = b;
b = temp;
```

Oppure, usando pa e pb

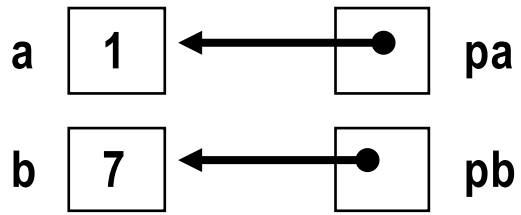
```
int temp;
temp = *pa;
*pa = *pb;
*pb = temp;
```



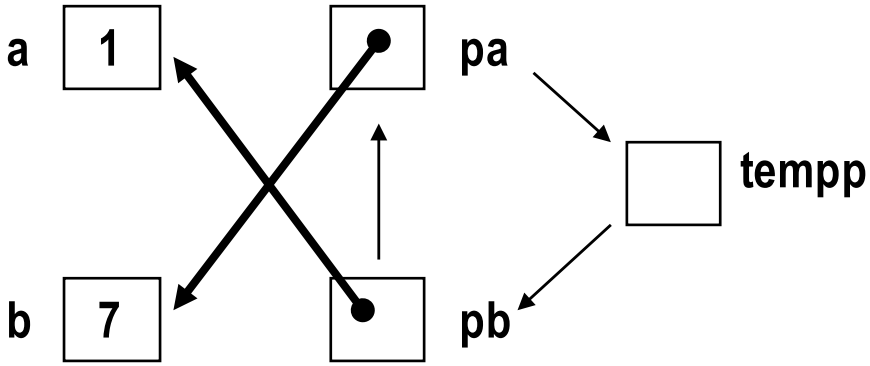
Scambio dei valori di a e b

Esempio

```
int a = 1;
int b = 7;
int* pa = &a;
int* pb = &b;
```




```
int* tempp;
tempp = pa;
pa = pb;
pb = tempp;
```



Scambio dei puntatori di a e b

Array e puntatori

Il ***nome di un array*** è in realtà un **puntatore costante** all'indirizzo di partenza dell'array stesso, per cui **non può essere modificato** nelle operazioni di aritmetica dei puntatori.

I puntatori possono essere utilizzati in tutte le operazioni di indicizzazione di un array

Array e puntatori

La dichiarazione di un **array** comporta allocazione di memoria di una **variabile puntatore** (il **nome** dell'**array**), e dell'**area puntata**, di lunghezza predefinita

*il **puntatore** é dichiarato **const*** e inizializzato con l'**indirizzo** dell'**area puntata** (cioè del primo elemento dell'**array**)

Es.: **int lista[5];**

- alloca memoria per il **puntatore costante lista**;
- alloca memoria per **5** valori di tipo **int**;
- inizializza **lista** con **&lista[0]**

Array e puntatori (1)

Il **nome** (usato da solo) di un **array** ha il significato di **puntatore** al primo **elemento** dell'array

Ogni altro elemento é accessibile tramite la **deref.** del **puntatore-array** incrementato di una quantità pari all'**indice/offset** dell'**elemento** ovvero le espressioni (dato un **array A**):

A[i] e ***(A+i)**

conducono ad identico risultato anche se sono due formalismi diversi.

Array e puntatori (2)

Se A[N] è un array di indirizzo ind,

A[i] il suo elemento i-esimo (i intero),

l.'indirizzo di A[i] sarà dato dall'espressione *(ind+i)

Array di puntatori

I puntatori, come qualsiasi altra variabile, possono essere raggruppati in array e dichiarati come:

Es.: **int* A[10] ;**

dichiara un array di 10 puntatori a int

Come il nome di un array equivale a un puntatore, così un array di puntatori equivale a un puntatore a puntatore (con in più l'allocazione della memoria puntata, come nel caso di array generico)

Array di Stringhe

Il caso più frequente di array di puntatori é quello dell'array di stringhe, che consente anche l'inizializzazione (atipicamente) delle stringhe che costituiscono l'array

Es.: **char* colori[3] = {''Blu'', ''Rosso'', ''Verde''};**

Le stringhe possono essere di differente lunghezza; in memoria sono allocate consecutivamente e, sono riservati tanti bytes quant'è la rispettiva lunghezza (terminatore compreso).

Array di puntatori

`suit [4]` indica un array di 4 elementi

```
char* suit[4] = {"Heart",
  "Diamond", "Clubs", "Spades"}
```

Ogni elemento dell'array è un puntatore ad un **`char`**, l'array contiene in realtà soltanto i puntatori al primo carattere di ogni stringa. Pur avendo l'array **`suit`** dimensione fissa, si può accedere a stringhe di qualsiasi lunghezza.

Stringhe

In memoria le stringhe sono degli array di tipo char, con una particolarità in più, che le fa riconoscere da operatori e funzioni come stringhe e non come normali array: l'elemento dell'array che segue l'ultimo carattere della stringa deve contenere il carattere NULL o '\0' (detto in questo caso terminatore); si dice pertanto che una stringa é un "array di tipo char null terminated".

Dichiarazione di stringhe

Es.: **`char MiaVar[30];`**

Dichiara la variabile MiaVar come array di tipo char con massimo 30 elementi. Affinché MiaVar sia identificata (in lettura) da operatori e funzioni come stringa, dobbiamo inserire nell'array una serie di caratteri terminati da un NULL.

Se vogliamo che MiaVar presenti a operatori e funzioni la stringa "Ciao", dobbiamo scrivere invece le istruzioni:

```
MiaVar[0]='C'; MiaVar[1]='i';
MiaVar[2]='a'; MiaVar[3]='o';
MiaVar[4]='\0';
```

impegnando così **5** elementi dell'array dei **30** disponibili (i rimanenti 25 saranno ignorati).

Inizializzazione di variabili stringa

Se nella dichiarazione-inizializzazione si omette la dimensione dell'array, questa viene automaticamente definita dalla lunghezza della costante stringa aumentata di uno, per far posto al terminatore (in questo caso la stringa non può più essere "allungata"!):

```
char Saluto[] = ''Ciao'';
```

allocato in memoria **array** con 5 **elementi**

Considerazioni

ATTENZIONE:

In caso che si creino delle stringhe con un numero di caratteri (compreso il terminatore) maggiore di quello dichiarato, il programma non produce direttamente messaggi di errore, ma invade zone di memoria non di sua pertinenza, con conseguenze imprevedibili (spesso si verifica un errore fatale a livello di sistema operativo).