Puntatori (in C)

Emilio Di Giacomo

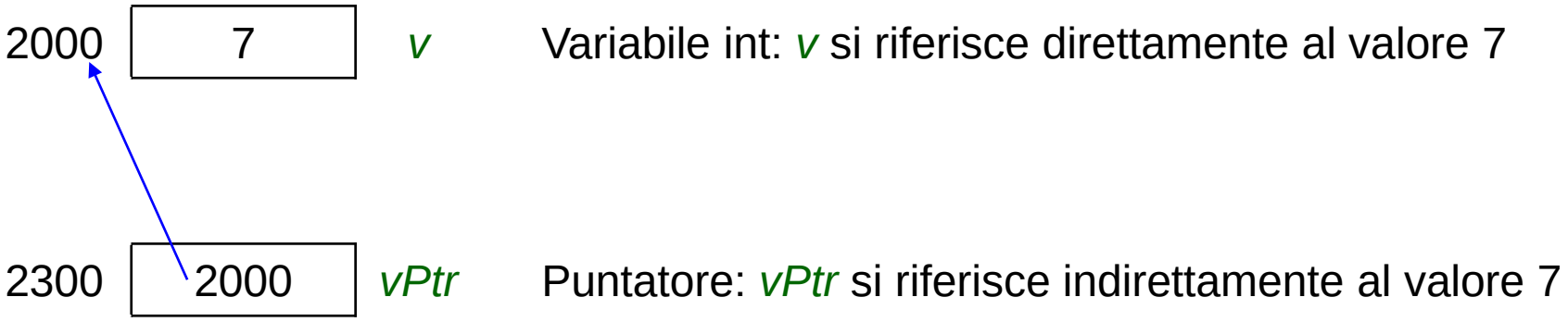
Puntatori

- In questa lezione parleremo di uno dei costrutti più potenti del C: i puntatori
- I puntatori vengono utilizzati per
  - realizzare il passaggio di parametri per riferimento
  - gestire strutture dati dinamiche
  - ...
- I puntatori sono una degli aspetti del C più difficili da padroneggiare
- Se adoperati male (intenzionalmente o accidentalmente) i puntatori posso portare a errori e a violazioni della sicurezza

Puntatori

- I puntatori sono un tipo di dato
- Una variabile di tipo puntatore memorizza un indirizzo di memoria
- Le variabili dei tipi primitivi (*int*, *double*, ...) memorizzano valori del tipo corrispondente
  - una variabile di questo tipo fa quindi direttamente riferimento a un valore
- Una variabile puntatore invece memorizza un indirizzo di memoria di una variabile
  - il puntatore quindi fa riferimento indirettamente a un valore

Puntatori



- In questo caso si dice che *vPtr* punta alla variabile *v* o, più semplicemente, punta a *v*

Puntatori

- Un puntatore memorizza l'indirizzo di una variabile di un certo tipo
  - esistono quindi puntatori a *int*, puntatori a *double*, ecc.
- Una variabile di tipo puntatore si dichiara con la seguente sintassi

  *<tipo> * <nome variabile>*
  - *<tipo>* è il tipo cui il puntatore fa riferimento
  - l'asterisco indica che la variabile è di tipo puntatore

Puntatori

- Esempio:

*int * p1;*

*double * p2;*

- *p1* è un puntatore a interi, cioè può memorizzare indirizzi di variabili *int*
- *p2* è un puntatore a *double*, cioè può memorizzare indirizzi di variabili *double*
- Attenzione:

*int * p, i;*

- solo *p* è un puntatore a interi, *i* è un intero

Assegnare valori ai puntatori

- Come possiamo assegnare un valore ad un puntatore?
- Ad un puntatore possiamo assegnare:
  - 0 (indica che il puntatore non punta a nulla)
  - NULL (come 0 ma preferibile)
    - NULL è una costante definita in *stddef.h*
  - L'indirizzo di una variabile
- Per assegnare ad un puntatore l'indirizzo di una variabile bisogna usare l'operatore di indirizzo *&*

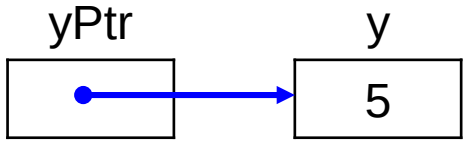
Operatore di indirizzo

- L'operatore di indirizzo *&* è un operatore unario che restituisce l'indirizzo di memoria del suo operando
- È quindi possibile scrivere:

  int y = 5;

  *int *yPtr = &y;*

  – L'indirizzo di memoria della variabile *y* viene assegnato al puntatore *yPtr* e quindi *yPtr* punta a *y*



Operatore di indirezione

- Per fare riferimento al valore della variabile puntata da un puntatore si può usare l'operatore di indirezione (o deferenziazione) *
- L'operatore di indirezione * è un operatore unario il cui operando è un puntatore e che restituisce il valore della variabile puntata dal puntatore
- Esempio:

  *printf("%d",*yPtr)*

  – stampa il valore della variabile *y*, cioè *5*

Operatore di indirizzo e indirezione

- Gli operatori di indirizzo e indirezione sono l'uno il complemento dell'altro
- Consideriamo il seguente programma

```
#include <stdio.h>
int main(){
        int y=5;
        int *yPtr=&y;
        printf("Indirizzo di y: %p valore di yPtr: %p\n", &y, yPtr);
        printf("Valore di y: %d valore di *yPtr: %d\n", y, *yPtr);
        printf("&*yPtr %p *&yPtr: %p\n", &*yPtr, *&yPtr);
}
```

- Lo specificatore *%p* è usato per visualizzare indirizzi di memoria

Operatore di indirizzo e indirezione

- L'output del programma predente è:

```
Indirizzo di y: 0028FEBC valore di yPtr: 0028FEBC
Valore di y: 5 valore di *yPtr: 5
&*yPtr 0028FEBC *&yPtr: 0028FEBC
```

- L'output conferma che
  - l'indirizzo di *y* e il valore di *yPtr* sono uguali
  - il valore di *y* e quello di **yPtr* sono uguali
  - gli operatori *&* e * possono essere applicati a *yPtr* in qualunque ordine dando lo stesso risultato

Passaggio di parametri

- Abbiamo visto come il passaggio di parametri ad una funzione avviene sempre per valore
- È possibile realizzare un passaggio per riferimento utilizzando gli operatori di indirizzo ed indirezione
- Supponiamo di voler scrivere un programma che data una variabile intera *a*, la modifica assegnandole il cubo di se stessa

Passaggio per valore

```
#include <stdio.h>

int cuboPerValore(int n){
    return n * n * n;
}

int main(){
    int a=5;
    printf("Valore iniziale di a: %d\n", a);
    a=cuboPerValore(a);
    printf("Nuovo valore di a: %d\n", a);
}
```

Passaggio per valore

- Il programma precedente passa la variabile *a* alla funzione *cuboPerValore*
- Poiché il passaggio è per valore la variabile *a* non può essere modificata all'interno della funzione
- Quindi per modificarla le si assegna il valore restituito dalla funzione
- Lo stesso effetto si potrebbe ottenere usando il passaggio per riferimento e modificando la variabile all'interno della funzione

Passaggio per riferimento

```
#include <stdio.h>

void cuboPerRiferimento(int *nPtr){
    *nPtr = *nPtr * *nPtr * *nPtr;
}

int main(){
    int a=5;
    printf("Valore iniziale di a: %d\n", a);
    cuboPerRiferimento(&a);
    printf("Nuovo valore di a: %d\n", a);
}
```

Passaggio per riferimento

- Il parametro della funzione *cuboPerRiferimento* è un puntatore a *int*
  - in questo modo viene passato l'indirizzo di *a* e non il suo valore
- All'interno della funzione si opera sulla variabile puntata da *nPtr* utilizzando l'operatore di indirezione
- Tale variabile viene modificata all'interno della funzione
  - non viene modificato il parametro ma la variabile da esso puntata
- All'atto dell'invocazione viene passato l'indirizzo di *a* tramite l'operatore di indirizzo

Passaggio parametri

- Quando usare il passaggio per riferimento?
- Quando è necessario che la variabile passata venga modificata internamente alla funzione
  - ciò accade ad esempio nella *scanf*
- Quando si devono passare molti dati e, per motivi di efficienza, si vuole evitare una copia completa limitandosi a copiare l'indirizzo
  - questo può succedere quando si usano le strutture (vedremo poi)

Esempio

- Supponiamo di voler scrivere una funzione che dati due interi effettui la divisione calcolando quoziente e resto
- Tale funzione dovrebbe restituire due valori interi
  - ciò però non è possibile
- Possiamo far in modo che la funzione scriva il risultato in due variabile passate per riferimento

Esempio

```
#include <stdio.h>
void dividi(int x, int y, int* quoziente, int* resto) {
        *resto = x % y;
        *quoziente = x/y;
}

int main(){
    int x, y;
    printf("immettere due interi\n");
    scanf("%d%d",&x,&y);
    int quoz, rest;
    dividi(x, y, &quoz, &rest);
    printf("%d%d", quoz,rest);
}
```

Passaggio di array

- Abbiamo visto che quando si passa un array ad una funzione il passaggio è un passaggio per riferimento
- Siamo adesso in grado di capire meglio la questione
- Come abbiamo visto una variabile di tipo array memorizza in realtà soltanto l'indirizzo della prima cella dell'array
  - essa è di fatto un puntatore alla prima cella dell'array

Passaggio di array

- Quando passiamo un array ad una funzione stiamo in realtà passando un puntatore
  - siamo quindi nella stessa situazione vista nel caso della funzione *cuboPerRiferiemnto*
- In effetti è possibile sostituire un parametro di tipo array (ad es. *int[]*) con un parametro di tipo puntatore (ad es. *int **)

void stampaArray(int a[], int size)

void stampaArray(int a, int size)*

Passaggio di array

- Nota: se una funzione ha un parametro di tipo puntatore (indicato come tipo puntatore o come tipo array) il compilatore non è in grado di sapere se essa si aspetta un array o l'indirizzo di una singola variabile

```
void stampaArray(int[] a, int size);

...

int b=10;

stampaArray(&b,4); // no errori in compilazione
```

Passaggio di array

- Nota: se una funzione ha un parametro di tipo puntatore (indicato come tipo puntatore o come tipo array) il compilatore non è in grado di sapere se essa si aspetta un array o l'indirizzo di una singola variabile

```
void stampaArray(int *a, int size);

...

int b=10;

stampaArray(&b,4); // no errori in compilazione
```

Ancora sul passaggio di puntatori

- Abbiamo visto che possiamo impedire la modifica di un parametro formale usando la parola chiave const
- Anche con i puntatori è possibile usare const
  - esso può però riferirsi sia al puntatore che ai dati puntati
  - ci sono quattro combinazioni possibili

Ancora sul passaggio di puntatori

- Esempi:

  *void funzione(int * p)*

  – puntatore non costante a dati non costanti

  *void funzione(const int * p)*

  – puntatore non costante a dati costanti

  *void funzione(int * const p)*

  – puntatore costante a dati non costanti

  *void funzione(const int * const p)*

  – puntatore costante a dati costanti

Aritmetica dei puntatori

- Sui puntatori si possono effettuare operazioni aritmetiche:
  - somma o sottrazione di un intero e quindi anche le operazioni di incremento e decremento
  - sottrazione tra due puntatori
- Il risultato di tali operazioni è quello di far puntare un puntatore ad una locazione di memoria successiva o precedente a quella iniziale
- Esse vengono svolte secondo regole che non sono quelle dell'aritmetica usuale
  - si parla di aritmetica dei puntatori

Aritmetica dei puntatori

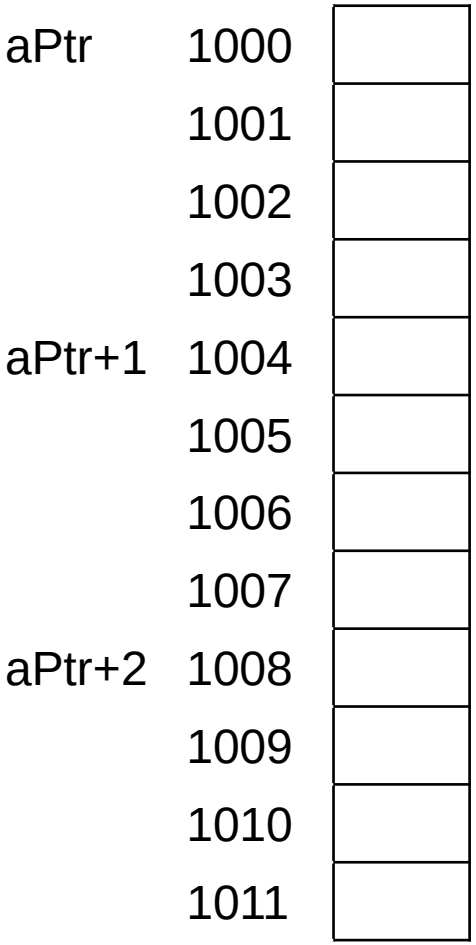
- Se *xPtr* è un puntatore ad un tipo *T* e il suo valore è un certo indirizzo *ind*, il risultato di *xPtr+1* si ottiene sommando a *ind* il numero di byte necessari a rappresentare il tipo *T*
- Esempio:

```
int *aPtr, *bPtr;
....
bPtr = aPtr+1;
```

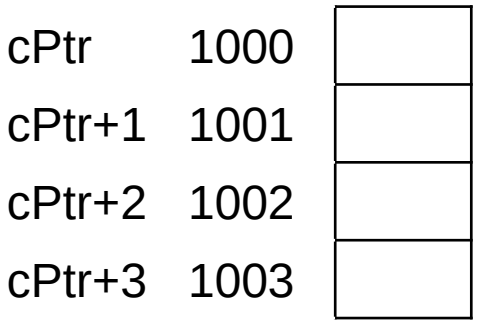
- Se il valore di *aPtr* è l'indirizzo 1000, il valore di *bPtr* dopo l'assegnamento è 1004 (assumendo che un intero occupi 4 byte)

Aritmetica dei puntatori

*int *aPtr*



*char *cPtr*



Aritmetica dei puntatori

- Supponiamo di aver definito due puntatori a *int* *xPtr* e *yPtr* e supponiamo che *xPtr* memorizzi l'indirizzo 1000 mentre *yPtr* memorizzi l'indirizzo 1008

  int x=yPtr-xPtr;

- L'istruzione precedente assegna a *x* il valore della differenza tra i due puntatori *yPtr* e *xPtr*
  - Tale valore non è 8 ma 2
  - In effetti *yPtr* è uguale a *xPtr+2*

Aritmetica dei puntatori: commenti

- L'aritmetica dei puntatori è indefinita a meno che non sia eseguita su un array
- Se infatti scriviamo

```
int a=7;
int *aPtr=&a;
aPtr++;
```

- dopo l'incremento del puntatore non abbiamo idea di quale sia la cella a cui sta puntando *aPtr* e di che cosa ci sia dentro
  - quindi qualunque operazione su **aPtr* ha un effetto non prevedibile

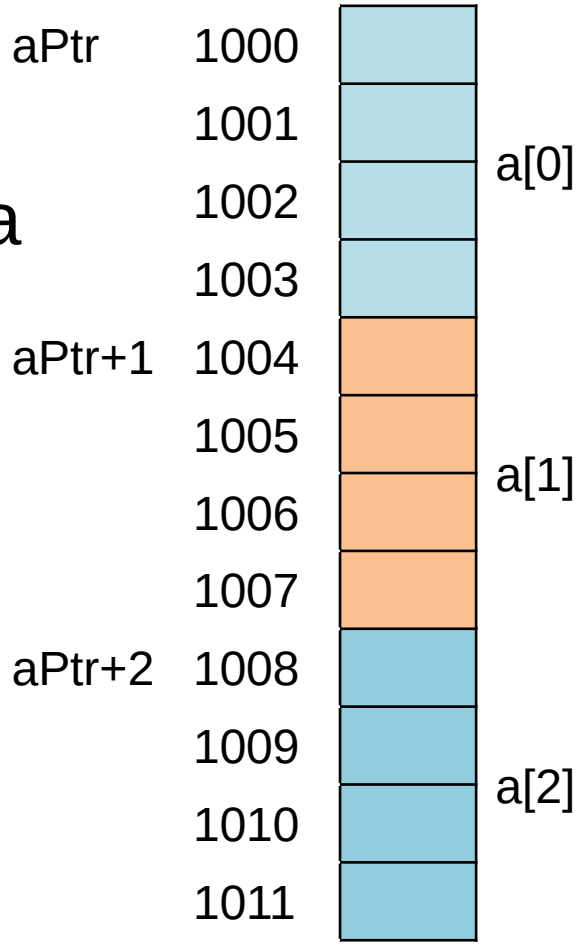
Aritmetica dei puntatori: commenti

- Consideriamo invece il seguente codice

  int a[3]={23, 54, 3};

  int aPtr=&a[0];*

- La situazione è illustrata in figura



- Se scriviamo

  **(aPtr+1)=6;*

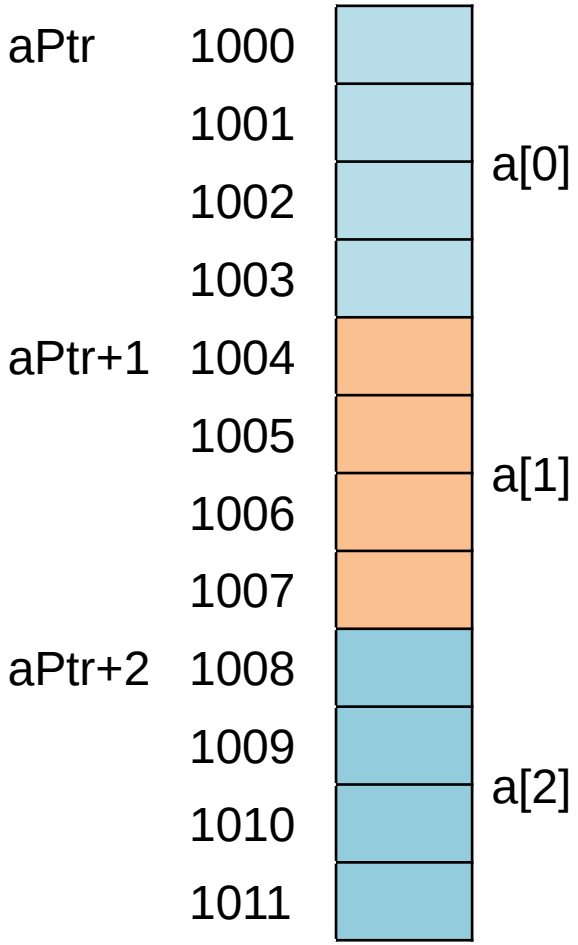
stiamo modificando *a[1]*

Aritmetica dei puntatori: commenti

- In altre parole, le seguenti due istruzioni sono equivalenti:

  a[1]=6;

  **(aPtr+1)=6;*



Array e puntatori

- In effetti array e puntatori in C sono due costrutti fortemente correlati
- Ad esempio possiamo assegnare il valore di una variabile array (che è un indirizzo!!) ad un puntatore

```
int b[5];
int *bPtr=b;
```

- L'ultima istruzione è equivalente a

```
int *bPtr=&b[0];
```

Array e puntatori

- Possiamo accedere agli elementi dell'array tramite indice o tramite l'aritmetica dei puntatori

  b[3]=7

  **(bPtr+3)=7*

- È possibile trattare un array come un puntatore

  **(b+3)=7*

- e un puntatore come un array

  bPtr[3]=7

Array e puntatori

- L'unica differenza tra un array e un puntatore è che il valore della variabile array non può essere modificato

  bPtr+=3; //OK

  b+=3; //NO

Esempio array/puntatori

```
#include <stdio.h>

int main(){

        int b[4]={10, 20, 30, 40};
        int *bPtr=b;

        printf("Array con indici\n");
        for(int i=0;i<4;i++)
                printf("b[%d]=%d\n",i,b[i]);

        printf("\nArray con offset\n");
        for(int i=0;i<4;i++)
                printf("*(b+%d)=%d\n",i,*(b+i));

        ...

}
```

Esempio array/puntatori

```
#include <stdio.h>

int main(){

        ...

        printf("\nPuntatore con indici\n");
        for(int i=0;i<4;i++)
                printf("bPtr[%d]=%d\n",i,bPtr[i]);

        printf("\nPuntatore con offset\n");
        for(int i=0;i<4;i++)
                printf("*(bPtr+%d)=%d\n",i,*(bPtr+i));

}
```

Esempio array/puntatori

- Output:

```
Array con indici
b[0]=10
b[1]=20
b[2]=30
b[3]=40

Array con offset
*(b+0)=10
*(b+1)=20
*(b+2)=30
*(b+3)=40
...
...

Puntatore con indici
bPtr[0]=10
bPtr[1]=20
bPtr[2]=30
bPtr[3]=40

Puntatore con offset
*(bPtr+0)=10
*(bPtr+1)=20
*(bPtr+2)=30
*(bPtr+3)=40
```

Assegnazione tra puntatori

- Un puntatore può essere assegnato ad un altro puntatore soltanto se i due puntatori sono dello stesso tipo

```
int * aPtr, *bPtr;
double * cPtr;
...
bPtr=aPtr   // OK
cPtr=aPtr // NO
```

Assegnazione tra puntatori

- Esiste anche il tipo puntatore a void (*void**)
- Tale tipo indica un puntatore generico che può memorizzare qualunque tipo di puntatore

```
void *aPtr;
int *bPtr;
double *cPtr;
...
aPtr=bPtr; // OK
aPtr=cPtr; // OK
bPtr=aPtr; // OK
cPtr=aPtr; // OK
```

Confronto tra puntatori

- È possibile confrontare puntatori utilizzando gli operatori relazionali
- Questi confronti hanno senso solo se i puntatori puntano allo stesso array
  - ad esempio se due puntatori puntano ad elementi dello stesso array un confronto può indicare che uno dei puntatori punta ad un elemento con indice più alto
- Il confronto di puntatori è comunemente usato per determinare se un puntatore è NULL

Altri usi dei puntatori

- I puntatori sono utili per la realizzazione di strutture dati dinamiche
  - ne riparleremo dopo aver introdotto le strutture
- In C inoltre esistono i puntatori a funzione che possono essere usati per
  - passare una funzione come parametro ad un'altra funzione
  - ottenere una funzione come risultato dell'invocazione di un'altra funzione
  - memorizzare funzioni in un array
- Non parleremo dei puntatori a funzione