PROGRAMMAZIONEB
INGEGNERIA E SCIENZE INFORMATICHE - CESENA
A.A. 2021-2022

I PUNTATORI

ANDREA PIRODDI - ANDREA.PIRODDI@UNIBO.IT
CREDIT: PIETRO DI LENA

Chuck Norris facts:
1. *Chuck Norris can dereference a NULL pointer.*
2. *When Chuck Norris points to NULL, NULL scares.*

Introduzione

- Una dichiarazione (definizione) di variabile comporta l'allocazione di un certo numero di **celle contigue di memoria** (RAM).
  - L'informazione memorizzabile nelle celle allocate dipende dal tipo della variabile.
  - Ad ogni cella è associato un **indirizzo di memoria** univoco.

Introduzione

- Una dichiarazione (definizione) di variabile comporta l'allocazione di un certo numero di **celle contigue di memoria** (RAM).
  - L'informazione memorizzabile nelle celle allocate dipende dal tipo della variabile.
  - Ad ogni cella è associato un **indirizzo di memoria** univoco.
- Un **puntatore** è una variabile che permette di memorizzare un indirizzo di memoria.
  - Diciamo che un puntatore **referenzia** una cella di memoria.
  - Possiamo accedere al valore contenuto nella cella di memoria referenziata tramite **dereferenziazione** del puntatore.

Introduzione

- Una dichiarazione (definizione) di variabile comporta l'allocazione di un certo numero di **celle contigue di memoria** (RAM).
  - L'informazione memorizzabile nelle celle allocate dipende dal tipo della variabile.
  - Ad ogni cella è associato un **indirizzo di memoria** univoco.
- Un **puntatore** è una variabile che permette di memorizzare un indirizzo di memoria.
  - Diciamo che un puntatore **referenzia** una cella di memoria.
  - Possiamo accedere al valore contenuto nella cella di memoria referenziata tramite **dereferenziazione** del puntatore.
- Nella vita reale facciamo frequentemente uso di *puntatori*. Ad esempio:
  - La pagina di un libro può essere vista come una *cella di memoria*.
  - Il numero di pagina di un libro può essere visto come un *indirizzo di memoria*.
  - L'indice di un libro contiene *riferimenti* alle pagine del libro (*indirizzi di memoria*).
  - L'indice stesso è stampato su una o più pagine (*celle di memoria*) ed è quindi associato ad un numero di pagina (*indirizzo di memoria*).
  - Tramite l'indice possiamo accedere al contenuto di una pagina leggendo il numero della pagina nell'indice e sfogliando le pagine del libro fino a raggiungere la pagina cercata (*dereferenziazione*).

Puntatori: dichiarazione

- Una dichiarazione di variabile puntatore è una dichiarazione di variabile in cui il nome della variabile è preceduto dal simbolo asterisco `*`.

```
<Tipo> *<Nome>;
```

- Il tipo del puntatore indica il contenuto dell'indirizzo di memoria puntato.

Puntatori: dichiarazione

- Una dichiarazione di variabile puntatore è una dichiarazione di variabile in cui il nome della variabile è preceduto dal simbolo asterisco `*`.

```
<Tipo> *<Nome>;
```

- Il tipo del puntatore indica il contenuto dell'indirizzo di memoria puntato.
- Facciamo quindi distinzione tra variabili di un certo tipo e variabili puntatore ad un certo tipo

```
1 char  a; // variabile di tipo char
2 char *p; // variabile di tipo puntatore a char
3
4 int  x;  // variabile di tipo int
5 int *y;  // variabile di tipo puntatore ad int
```

Puntatori: dichiarazione

- Una dichiarazione di variabile puntatore è una dichiarazione di variabile in cui il nome della variabile è preceduto dal simbolo asterisco `*`.

```
<Tipo> *<Nome>;
```

- Il tipo del puntatore indica il contenuto dell'indirizzo di memoria puntato.
- Facciamo quindi distinzione tra variabili di un certo tipo e variabili puntatore ad un certo tipo

```
1 char  a; // variabile di tipo char
2 char *p; // variabile di tipo puntatore a char
3
4 int   x; // variabile di tipo int
5 int  *y; // variabile di tipo puntatore ad int
```

- Possiamo dichiarare puntatori *generici*, senza specificare il tipo puntato.

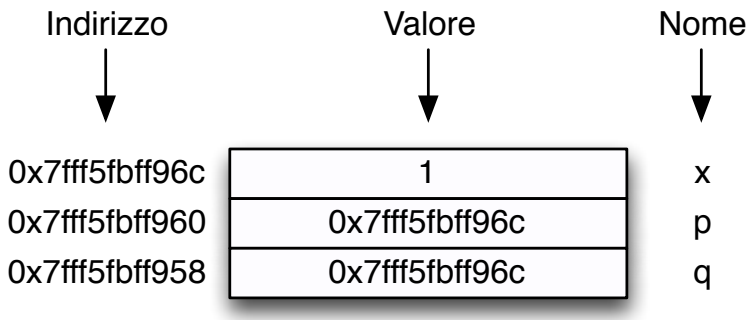
```
1 void *p; // variabile di tipo puntatore generico
```

- I puntatori generici sono utili quando non è noto a priori il tipo della variabile puntata.
- Per poterli utilizzare è necessario effettuare un cast al tipo specifico.

Puntatori: assegnamento

- Una variabile di tipo puntatore può contenere solo indirizzi di memoria o il valore costante `NULL`.
- Gli indirizzi di memoria che un puntatore può memorizzare dipendono dalla dimensione dello spazio di indirizzamento virtuale del processo.
- Gli indirizzi di memoria ottenuti tramite l'applicazione dell'operatore `&`, possono essere assegnati ad una variabile di tipo puntatore.

```
1 int  x = 1;
2 int *p = &x;    // Inizializza p con l'indirizzo di x
3 int *q = NULL; // Inzializza q con il valore NULL
4
5 q = &x;         // Assegna a q l'indirizzo di x
```



Puntatori: la costante NULL

- La costante `NULL` è una macro definita dall'implementazione in `stdlib.h`.
- Il valore `NULL` rappresenta un indirizzo di memoria *non accessibile*.
- Un puntatore a `NULL` viene inteso come un puntatore che non punta a nulla.

Puntatori: la costante NULL

- La costante `NULL` è una macro definita dall'implementazione in `stdlib.h`.
- Il valore `NULL` rappresenta un indirizzo di memoria *non accessibile*.
- Un puntatore a `NULL` viene inteso come un puntatore che non punta a nulla.
- Lo standard C impone una forte equivalenza tra la costante `0` e la costante `NULL`.
  - In operazioni che coinvolgono puntatori, un'**espressione intera costante con valore 0** viene convertita automaticamente dal compilatore nella costante `NULL`.
- Le seguenti espressioni di test del costrutto `if` sono quindi tutte equivalenti, indipendentemente da come viene rappresentata la costante `NULL`.

```c
int x, *p = &x;

if(p != NULL) { x = 1; }

// La costante 0 viene convertita in NULL
if(p != 0)    { x = 1; }

// Viene interpretato come p!=0 e quindi come p!=NULL
if(p)         { x = 1; }
```

Puntatori: la costante NULL

- La costante `NULL` è una macro definita dall'implementazione in `stdlib.h`.
- Il valore `NULL` rappresenta un indirizzo di memoria *non accessibile*.
- Un puntatore a `NULL` viene inteso come un puntatore che non punta a nulla.
- Lo standard C impone una forte equivalenza tra la costante `0` e la costante `NULL`.
  - In operazioni che coinvolgono puntatori, un'**espressione intera costante con valore 0** viene convertita automaticamente dal compilatore nella costante `NULL`.
- Le seguenti espressioni di test del costrutto `if` sono quindi tutte equivalenti, indipendentemente da come viene rappresentata la costante `NULL`.

```
1 int x, *p = &x;
2
3 if(p != NULL) { x = 1; }
4
5 // La costante 0 viene convertita in NULL
6 if(p != 0)     { x = 1; }
7
8 // Viene interpretato come p!=0 e quindi come p!=NULL
9 if(p)          { x = 1; }
```

- Tipicamente, la macro `NULL` è definita come

```
1 #define NULL ((void *)0)
```

oppure

```
1 #define NULL 0
```

Puntatori: tipo del puntatore

- I puntatori contengono solo indirizzi di memoria ma è necessario informare il compilatore del tipo di dato che si trova all'indirizzo di memoria puntato.
- Questa informazione è necessaria per indicare al compilatore:
  1. la dimensione (in byte) dell'oggetto puntato,
  2. come interpretare il contenuto della locazione di memoria puntata.

Puntatori: tipo del puntatore

- I puntatori contengono solo indirizzi di memoria ma è necessario informare il compilatore del tipo di dato che si trova all'indirizzo di memoria puntato.
- Questa informazione è necessaria per indicare al compilatore:
  1. la dimensione (in byte) dell'oggetto puntato,
  2. come interpretare il contenuto della locazione di memoria puntata.
- Una variabile puntatore al tipo `T` **dovrebbe** contenere unicamente indirizzi di memoria di oggetti di tipo `T`.
- Assegnamenti ad indirizzi di memoria non compatibili col tipo puntato non sono sintatticamente proibiti ma:
  - l'accesso a tale contenuto di memoria tramite il puntatore può causare comportamenti non definiti,
  - il compilatore avvisa con un warning assegnamenti di questo tipo.

```
1 int    x = 1;
2 int   *p = &x; // OK
3 char  *q = &x; // Warning: potenzialmente pericoloso
```

Puntatori: tipo del puntatore

- I puntatori contengono solo indirizzi di memoria ma è necessario informare il compilatore del tipo di dato che si trova all'indirizzo di memoria puntato.
- Questa informazione è necessaria per indicare al compilatore:
  1. la dimensione (in byte) dell'oggetto puntato,
  2. come interpretare il contenuto della locazione di memoria puntata.
- Una variabile puntatore al tipo `T` **dovrebbe** contenere unicamente indirizzi di memoria di oggetti di tipo `T`.
- Assegnamenti ad indirizzi di memoria non compatibili col tipo puntato non sono sintatticamente proibiti ma:
  - l'accesso a tale contenuto di memoria tramite il puntatore può causare comportamenti non definiti,
  - il compilatore avvisa con un warning assegnamenti di questo tipo.

```
1 int    x = 1;
2 int   *p = &x; // OK
3 char  *q = &x; // Warning: potenzialmente pericoloso
```

- Possiamo anche dichiarare puntatori al **tipo di dato funzione**:

```
1 int     f(int x); // funzione che prende un int e ritorna un int
2 int  (*q)(int x); // puntatore a funzione che prende un int
3                   // e ritorna un int
4
5 q = f;    // Assegnamento con l'indirizzo della funzione f
```

Puntatori: dimensione di memoria

- Esattamente come abbiamo visto per gli altri tipi di dato, la dimensione in byte di una variabile puntatore può essere determinata utilizzando l'operatore `sizeof()`.

```
1 int *p;
2
3 printf("%lu\n",sizeof(int *));
4 printf("%lu\n",sizeof(p));
```

Puntatori: dimensione di memoria

- Esattamente come abbiamo visto per gli altri tipi di dato, la dimensione in byte di una variabile puntatore può essere determinata utilizzando l'operatore `sizeof()`.

```
1 int *p;
2
3 printf("%lu\n",sizeof(int *));
4 printf("%lu\n",sizeof(p));
```

- Lo standard non impone vincoli sulla dimensione del tipo di dato puntatore.
- Inoltre, non impone che puntatori a tipi differenti abbiano tutti la stessa dimensione.
  - Ad esempio, su alcune architetture `sizeof(char *)` $\neq$ `sizeof(int *)`.
  - E' possibile che puntatori al tipo funzione e tipo di dato numerico abbiano dimensione differente (ad esempio, su architetture Harvard).

Puntatori: dimensione di memoria

- Esattamente come abbiamo visto per gli altri tipi di dato, la dimensione in byte di una variabile puntatore può essere determinata utilizzando l'operatore `sizeof()`.

```c
int *p;

printf("%lu\n",sizeof(int *));
printf("%lu\n",sizeof(p));
```

- Lo standard non impone vincoli sulla dimensione del tipo di dato puntatore.
- Inoltre, non impone che puntatori a tipi differenti abbiano tutti la stessa dimensione.
  - Ad esempio, su alcune architetture `sizeof(char *)` $\neq$ `sizeof(int *)`.
  - E' possibile che puntatori al tipo funzione e tipo di dato numerico abbiano dimensione differente (ad esempio, su architetture Harvard).
- Come unica restrizione, lo standard impone la seguente proprietà:
  - Possiamo convertire un qualsiasi dato puntatore in tipo `void *` e poi riconvertirlo nuovamente nel tipo puntatore originario: il valore originale deve restare invariato.

```c
// Tipi puntatore
int x,*p = &x;
void  *q = p;
int   *r = q; // r = p
```

```c
// Tipi numerici
double  x = 1.5;
int     y = x;
double  z = y; // z != x
```

- Nota: questa proprietà non implica necessariamente che per qualsiasi tipo `T`,

$$\texttt{sizeof(void *)} \geq \texttt{sizeof(T *)}.$$

Puntatori: dereferenziazione

- Per poter *accedere* al contenuto dell'oggetto puntato dal puntatore utilizziamo l'operatore di **dereferenziazione** `*`.
- Il risultato dell'applicazione dell'operatore `*` è l'**oggetto puntato**.

```
1  int    x = 1;
2  int   *p = &x;
3
4  printf("%d\n",x);              // Stampa il contenuto di x
5  printf("%d\n",*p);             // Stampa il contenuto di x
6  printf("%p\n",(void *)&x);     // Stampa l'indirizzo di x
7  printf("%p\n",(void *)p);      // Stampa l'indirizzo di x
```

Nota: lo specificatore di formato %p si aspetta un puntatore a `void`.

Puntatori: dereferenziazione

- Per poter *accedere* al contenuto dell'oggetto puntato dal puntatore utilizziamo l'operatore di **dereferenziazione** `*`.
- Il risultato dell'applicazione dell'operatore `*` è l'**oggetto puntato**.

```
1 int    x = 1;
2 int   *p = &x;
3
4 printf("%d\n",x);              // Stampa il contenuto di x
5 printf("%d\n",*p);             // Stampa il contenuto di x
6 printf("%p\n",(void *)&x); // Stampa l'indirizzo di x
7 printf("%p\n",(void *)p);   // Stampa l'indirizzo di x
```

Nota: lo specificatore di formato %p si aspetta un puntatore a `void`.

- Possiamo modificare il valore di una variabile tramite la dereferenziazione.

```
1 int  x = 1;
2 int *p = &x; // p punta ad x
3
4 *p  = 2;  // x contiene 2
5 *p += 1;  // x contiene 3
6 x  *= *p; // x contiene 9
7 (*p)++;   // x contiene 10 (parentesi necessarie)
```

Possiamo accedere e modificare il contenuto della variabile `x` tramite riferimento diretto all'identificatore `x`, oppure tramite riferimento indiretto all'indirizzo di memoria di `x`, contenuto nella variabile puntatore `p`.

Puntatori: dereferenziazione scorretta

- E' necessario prestare molta attenzione alla semantica dell'operatore `*`.

```
1 int  x = 1;
2 int *p = &x; // OK
```

```
1 int  x = 1;
2 int *p;
3
4 p = &x; // OK
```

```
1 int  x = 1;
2 int *p;
3
4 *p = &x; // Warning
```

Nei primi due esempi copiamo correttamente l'indirizzo di `x` nella variabile puntatore `p`. Nel terzo esempio tentiamo di salvare l'indirizzo di `x` nella cella di memoria puntata da `p`, che non è stato inizializzato.

Puntatori: dereferenziazione scorretta

- E' necessario prestare molta attenzione alla semantica dell'operatore *.

```c
int  x = 1;
int *p = &x; // OK
```

```c
int  x = 1;
int *p;

p = &x; // OK
```

```c
int  x = 1;
int *p;

*p = &x; // Warning
```

Nei primi due esempi copiamo correttamente l'indirizzo di x nella variabile puntatore p. Nel terzo esempio tentiamo di salvare l'indirizzo di x nella cella di memoria puntata da p, che non è stato inizializzato.

- Dereferenziare puntatori non inizializzati, o inizializzati non correttamente, causa comportamento non definito.

```c
int  *p; // p non inizializzato

*p = -1;

// comportamento non definito
printf("%d\n",*p);
```

```c
int  *p;

p = -1; // indirizzo non legale

// comportamento non definito
printf("%d\n",*p);
```

Nel primo esempio, non sappiamo a cosa punti p. Nel secondo esempio, assegnamo a p un indirizzo non legale. In entrambi i casi, il programma potrebbe andare in crash.

Puntatori: dereferenziazione di puntatori a `void` e `NULL`

- Dereferenziare un puntatore inizializzato a `NULL` causa comportamento non definito.

```
1 int *p = NULL;
2
3 *p = 1;   // Comportamento non definito
```

Il programma nell'esempio andrà quasi sicuramente in crash.

Puntatori: dereferenziazione di puntatori a `void` e `NULL`

- Dereferenziare un puntatore inizializzato a `NULL` causa comportamento non definito.

```
1 int *p = NULL;
2
3 *p = 1;   // Comportamento non definito
```

Il programma nell'esempio andrà quasi sicuramente in crash.

- Per poter dereferenziare un puntatore a `void` è necessario prima convertirlo nel tipo appropriato.

```
1 int x;
2 void *p = &x;          // OK
3
4 *p = 1;                // Errore
5 printf("%d\n",*p);     // Errore
```

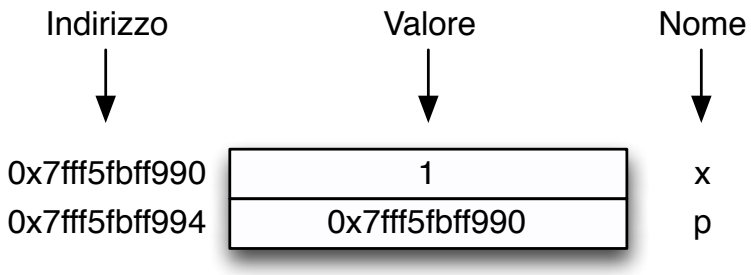
```
1 int x;
2 void *p = &x;                    // OK
3
4 *((int *)p) = 1;                 // OK
5 printf("%d\n",*(int *)p);        // OK
```

Nel primo esempio il compilatore rileva due errori di sintassi.

Puntatori: puntatori a puntatori 1/2

- Le variabili di tipo puntatore sono a tutti gli effetti variabili e risiedono in qualche locazione di memoria dello spazio di indirizzamento virtuale.
- Come per gli altri tipi di variabili, è possibile recuperare l'indirizzo di memoria di una variabile puntatore facendo uso dell'operatore &.

```
1 int   x = 1;
2 int *p = &x;
3
4 printf("%d\n",x);            // Stampa il contenuto della variabile x
5 printf("%d\n",*p);           // Stampa il contenuto della variabile x
6 printf("%p\n",(void *)&x); // Stampa l'indirizzo della variabile x
7 printf("%p\n",(void *)p);   // Stampa l'indirizzo della variabile x
8 printf("%p\n",(void *)&p); // Stampa l'indirizzo della variabile p
```

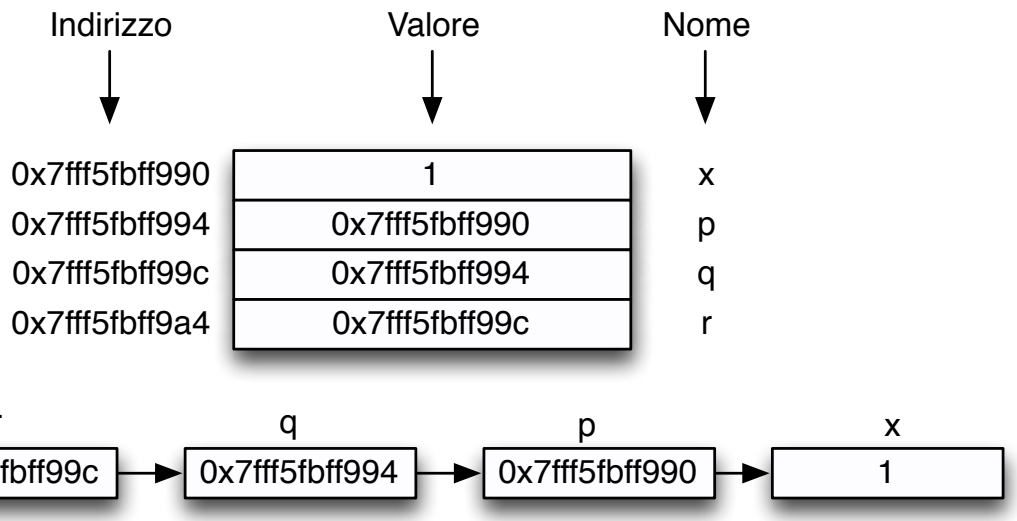


- Come facciamo a dichiarare variabili puntatore che contengano indirizzi di variabili puntatore?

Puntatori: puntatori a puntatori 2/2

- Per poter dichiarare variabili puntatore che contengono indirizzi di memoria di puntatori (puntatori a puntatori), utilizziamo la notazione asterisco *: facciamo precedere un asterisco dal tipo dell'oggetto puntato.
- Per poter dereferenziare puntatori a puntatori applichiamo l'operatore asterisco * un numero di volte pari ai livelli di dereferenziazione richiesti.

```
1 int     x = 1;   // x     una variabile di tipo int
2 int    *p = &x;  // p     un puntatore ad int
3 int   **q = &p;  // q     un puntatore a puntatore ad int
4 int  ***r = &q;  // r     un puntatore a puntatore a puntatore ad int
5
6 printf("%d\n",*p);    // Stampa il contenuto della variabile x
7 printf("%d\n",**q);   // Stampa il contenuto della variabile x
8 printf("%d\n",***r);  // Stampa il contenuto della variabile x
```



Puntatori: puntatori a `const`

- Puntatori a valori `const` (read-only): non possiamo modificare il contenuto dell'indirizzo di memoria puntato.

Puntatori: puntatori a const

- Puntatori a valori const (read-only): non possiamo modificare il contenuto dell'indirizzo di memoria puntato.
- L'indirizzo di memoria puntato non deve necessariamente essere *read-only*, semplicemente non possiamo modificarne il valore puntato tramite dereferenziazione.

```
1 int         x = 1;   // x    una variabile di tipo int
2 const int *p = &x; // p    un puntatore ad un int read-only
3
4 x   = 2; // OK: la variabile x non    read-only
5 *p = 3; // Errore di sintassi: p punta ad un indirizzo read-only
```

Puntatori: puntatori a const

- Puntatori a valori const (read-only): non possiamo modificare il contenuto dell'indirizzo di memoria puntato.
- L'indirizzo di memoria puntato non deve necessariamente essere *read-only*, semplicemente non possiamo modificarne il valore puntato tramite dereferenziazione.

```
1 int         x = 1;   // x    una variabile di tipo int
2 const int *p = &x; // p    un puntatore ad un int read-only
3
4 x   = 2; // OK: la variabile x non    read-only
5 *p = 3; // Errore di sintassi: p punta ad un indirizzo read-only
```

- In perfetta simmetria, possiamo modificare il valore di una variabile read-only tramite dereferenziazione, se assegniamo l'indirizzo della variabile ad un puntatore (senza indicare che il tipo puntato è const). Poco pulito: genera warning del compilatore.

```
1 const int x = 1;   // x    una variabile di tipo const int
2 int       *p = &x; // p    un puntatore a int
3
4 x   = 2; // Errore di sintassi: x    una variabile read-only
5 *p = 3; // OK: p non punta ad un indirizzo read-only
```

Puntatori: puntatori a const

- Puntatori a valori const (read-only): non possiamo modificare il contenuto dell'indirizzo di memoria puntato.
- L'indirizzo di memoria puntato non deve necessariamente essere *read-only*, semplicemente non possiamo modificarne il valore puntato tramite dereferenziazione.

```
1 int         x = 1;   // x    una variabile di tipo int
2 const int *p = &x; // p    un puntatore ad un int read-only
3
4 x   = 2; // OK: la variabile x non    read-only
5 *p = 3; // Errore di sintassi: p punta ad un indirizzo read-only
```

- In perfetta simmetria, possiamo modificare il valore di una variabile read-only tramite dereferenziazione, se assegniamo l'indirizzo della variabile ad un puntatore (senza indicare che il tipo puntato è const). Poco pulito: genera warning del compilatore.

```
1 const int x = 1;   // x    una variabile di tipo const int
2 int       *p = &x; // p    un puntatore a int
3
4 x   = 2; // Errore di sintassi: x    una variabile read-only
5 *p = 3; // OK: p non punta ad un indirizzo read-only
```

- Nota: possiamo posizionare const prima o dopo il tipo di dato.

```
1 const int x; // Variabile read-only di tipo int
2 int const y; // Variabile read-only di tipo int
```

Puntatori: puntatori `const`

- Puntatori `cost` (read-only): non possiamo modificare l'indirizzo puntato.

Puntatori: puntatori const

- Puntatori cost (read-only): non possiamo modificare l'indirizzo puntato.
- Per dichiarare un puntatore read-only, la keyword const **deve precedere** l'asterisco.

```
1 int x;
2 int * const p = &x; // p     un puntatore read-only ad int.
3 int const  *q = &x; // q     un puntatore a un un int read-only
```

Puntatori: puntatori const

- Puntatori cost (read-only): non possiamo modificare l'indirizzo puntato.
- Per dichiarare un puntatore read-only, la keyword const **deve precedere** l'asterisco.

```
1 int x;
2 int * const p = &x; // p     un puntatore read-only ad int.
3 int const  *q = &x; // q     un puntatore a un un int read-only
```

- Possiamo inizializzare il puntatore read-only con un indirizzo di memoria e modificare il valore all'indirizzo puntato tramite dereferenziazione.

```
1 int   x;
2 int * const p = &x; // p     un puntatore read-only ad int.
3
4 *p = 0; // OK
```

Puntatori: puntatori const

- Puntatori cost (read-only): non possiamo modificare l'indirizzo puntato.
- Per dichiarare un puntatore read-only, la keyword const **deve precedere** l'asterisco.

```
1 int x;
2 int * const p = &x; // p     un puntatore read-only ad int.
3 int const  *q = &x; // q     un puntatore a un un int read-only
```

- Possiamo inizializzare il puntatore read-only con un indirizzo di memoria e modificare il valore all'indirizzo puntato tramite dereferenziazione.

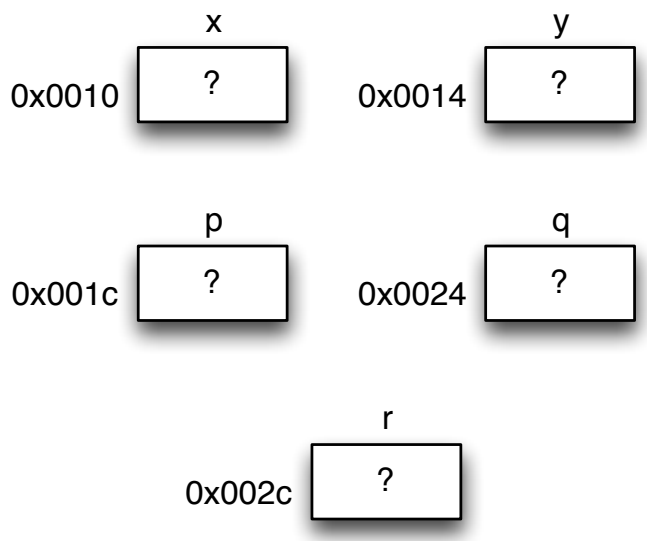
```
1 int  x;
2 int * const p = &x; // p     un puntatore read-only ad int.
3
4 *p = 0; // OK
```

- Possiamo solo inizializzare un puntatore read-only, ma non modificarne successivamente il valore.

```
1 int  x, y;
2 int * const p = &x; // p     un puntatore read-only ad int.
3
4 p = &y; // Errore di sintassi: p     read-only
```

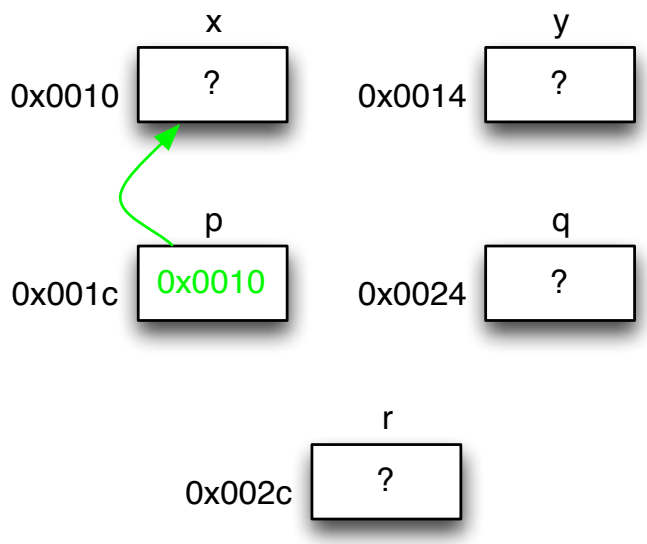
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



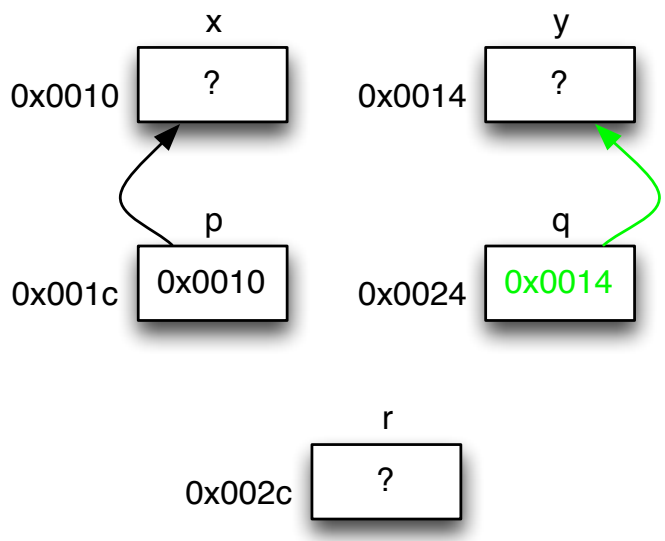
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



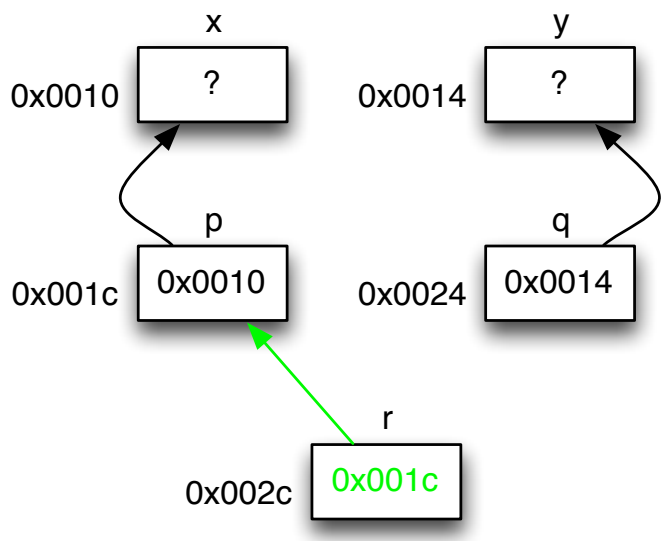
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



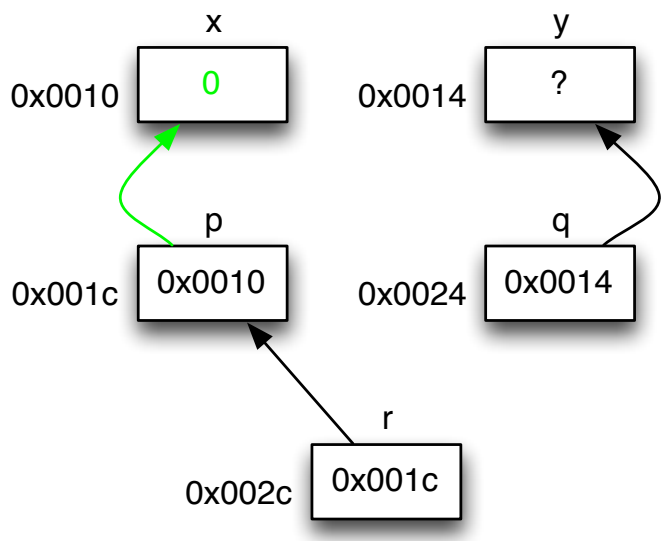
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



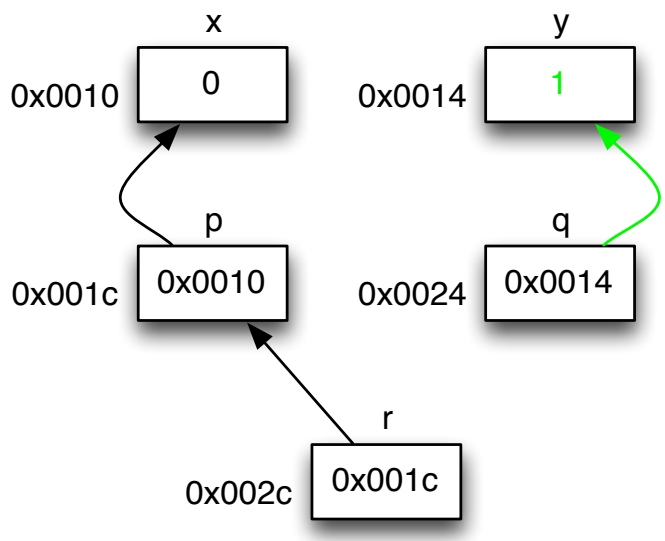
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



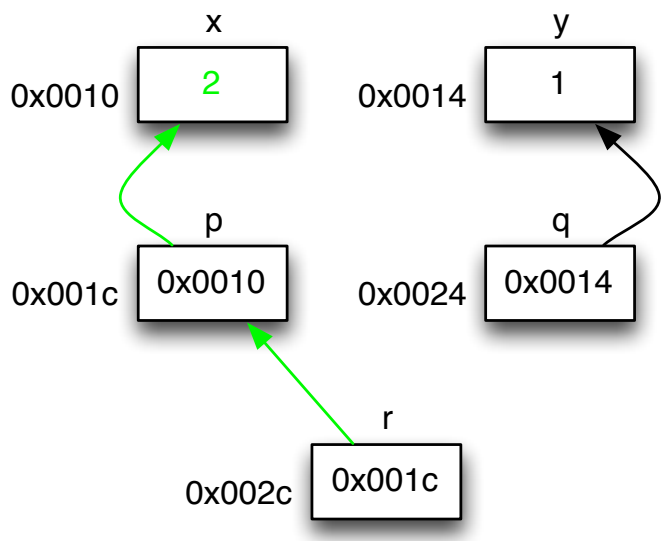
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



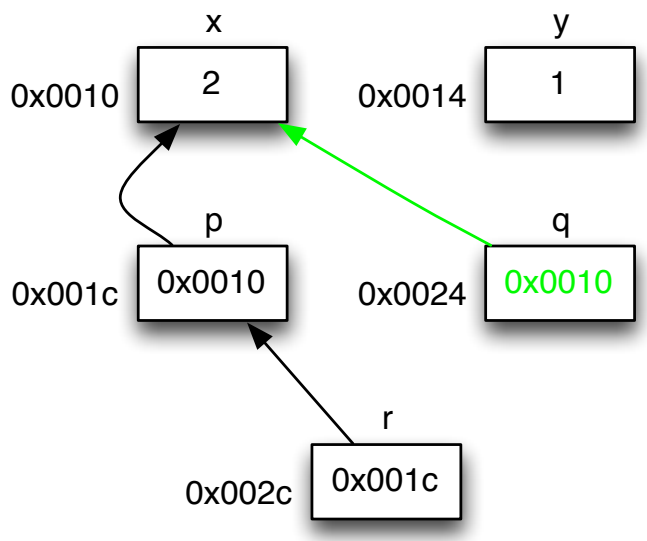
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



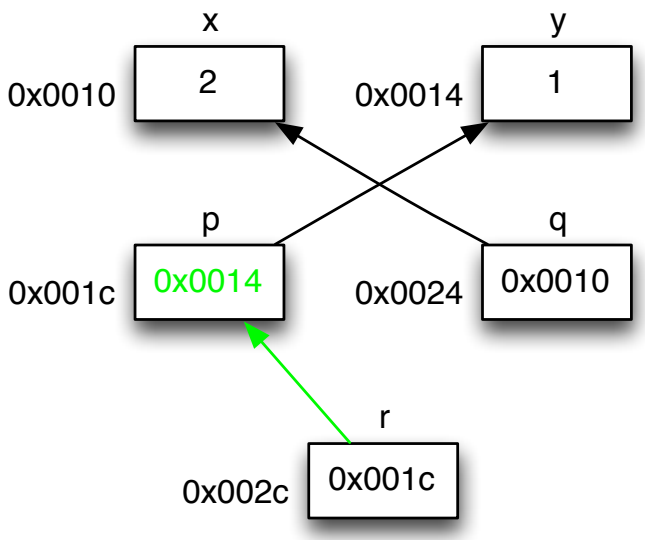
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



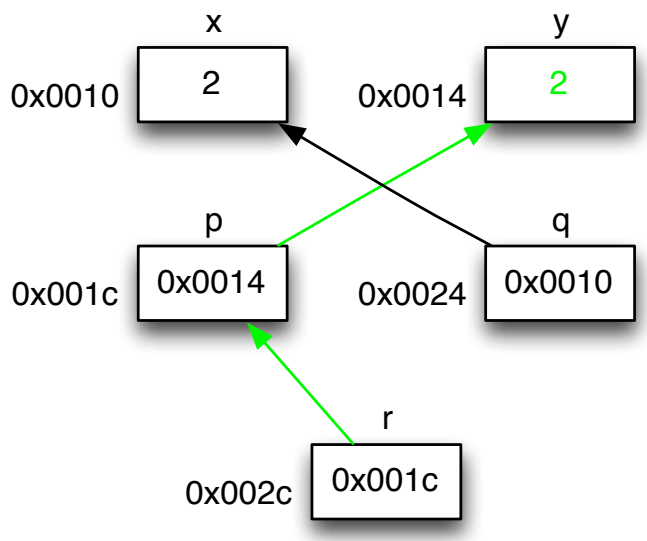
Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



Puntatori: esempio con notazione grafica

```
1  int main() {
2    int x, y;
3    int *p, *q;
4    int **r;
5
6      p = &x;
7      q = &y;
8      r = &p;
9
10   *p  = 0;
11   *q  = 1;
12   **r = 2;
13
14     q = &x;
15    *r = &y;
16   **r = 2;
17
18   return 0;
19 }
```



Aritmetica dei puntatori

- E' possibile eseguire diverse operazioni sulle variabili di tipo puntatore.
- **Sono ammesse** le operazioni di:
  - assegnamento tra puntatori e con tipi di dato interi
  - addizione e sottrazione tra puntatori e tipi numerici interi
  - sottrazione tra due puntatori,
  - confronto relazionale tra puntatori e con tipi di dato interi

Aritmetica dei puntatori

- E' possibile eseguire diverse operazioni sulle variabili di tipo puntatore.
- **Sono ammesse** le operazioni di:
  - assegnamento tra puntatori e con tipi di dato interi
  - addizione e sottrazione tra puntatori e tipi numerici interi
  - sottrazione tra due puntatori,
  - confronto relazionale tra puntatori e con tipi di dato interi
- **Non sono ammesse** (errore di sintassi) le operazioni di:
  - assegnamento con tipi in virgola mobile,
  - addizione e sottrazione tra puntatori e tipi numerici in virgola mobile,
  - addizione tra due puntatori e sottrazione tra puntatori di tipo differente
  - confronto relazionale con tipi in virgola mobile
  - le altre operazioni aritmetiche e le operazioni bitwise

Aritmetica dei puntatori

- E' possibile eseguire diverse operazioni sulle variabili di tipo puntatore.
- **Sono ammesse** le operazioni di:
  - assegnamento tra puntatori e con tipi di dato interi
  - addizione e sottrazione tra puntatori e tipi numerici interi
  - sottrazione tra due puntatori,
  - confronto relazionale tra puntatori e con tipi di dato interi
- **Non sono ammesse** (errore di sintassi) le operazioni di:
  - assegnamento con tipi in virgola mobile,
  - addizione e sottrazione tra puntatori e tipi numerici in virgola mobile,
  - addizione tra due puntatori e sottrazione tra puntatori di tipo differente
  - confronto relazionale con tipi in virgola mobile
  - le altre operazioni aritmetiche e le operazioni bitwise
- Sono sintatticamente **permesse ma** potenzialmente **pericolose** le operazioni di:
  - assegnamento e confronto tra puntatori di tipo differente senza utilizzare l'operatore di cast (fa eccezione il tipo `void *`),
  - assegnamento e confronto con tipi interi diversi da 0.

Aritmetica dei puntatori: assegnamento

- Possiamo assegnare ad una variabile puntatore il valore di un altro puntatore dello stesso tipo, oppure di tipo puntatore a `void`, e il valore intero 0 (equivale ad un assegnamento con `NULL`).

```
1 void *p = 0; // Assegnamento con tipo numerico intero 0
2 int  *q = p; // Assegnamento con puntatore a void
3 int  *r = q; // Assegnamento tra puntatori stesso tipo
4 void *s = r; // Assegnamento di puntatore a void con puntatore a int
```

Aritmetica dei puntatori: assegnamento

- Possiamo assegnare ad una variabile puntatore il valore di un altro puntatore dello stesso tipo, oppure di tipo puntatore a `void`, e il valore intero 0 (equivale ad un assegnamento con `NULL`).

```
1 void *p = 0; // Assegnamento con tipo numerico intero 0
2 int  *q = p; // Assegnamento con puntatore a void
3 int  *r = q; // Assegnamento tra puntatori stesso tipo
4 void *s = r; // Assegnamento di puntatore a void con puntatore a int
```

- Non possiamo assegnare ad una variabile puntatore un valore in virgola mobile.

```
1 void  *p = 1.2; // Errore di sintassi
2 int   *q = 1.2; // Errore di sintassi
3 float *r = 1.2; // Errore di sintassi
```

Aritmetica dei puntatori: assegnamento

- Possiamo assegnare ad una variabile puntatore il valore di un altro puntatore dello stesso tipo, oppure di tipo puntatore a `void`, e il valore intero 0 (equivale ad un assegnamento con `NULL`).

```
1 void *p = 0; // Assegnamento con tipo numerico intero 0
2 int  *q = p; // Assegnamento con puntatore a void
3 int  *r = q; // Assegnamento tra puntatori stesso tipo
4 void *s = r; // Assegnamento di puntatore a void con puntatore a int
```

- Non possiamo assegnare ad una variabile puntatore un valore in virgola mobile.

```
1 void  *p = 1.2; // Errore di sintassi
2 int   *q = 1.2; // Errore di sintassi
3 float *r = 1.2; // Errore di sintassi
```

- Sono sintatticamente corretti ma *non sicuri* gli assegnamenti tra puntatori di tipo differente (non `void`) e con interi diversi da 0.

```
1 int   *p = -10; // Warning
2 int   *q = 100; // Warning
3 float *r = q;   // Warning
```

Aritmetica dei puntatori: somma e sottrazione con tipi numerici

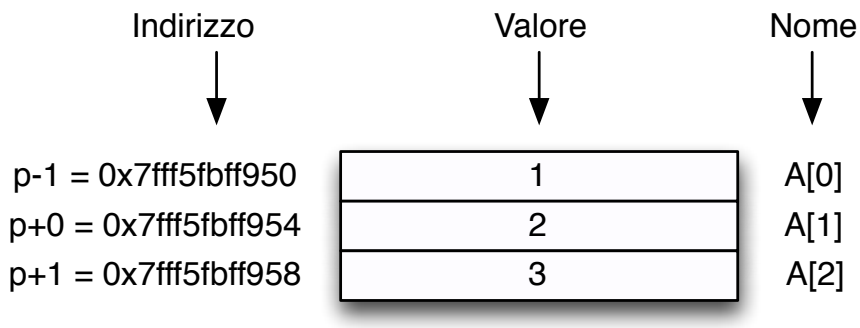
- Possiamo sommare/sottrarre valori interi ad un puntatore. Il nuovo indirizzo di memoria viene calcolato moltiplicando il valore intero per la dimensione del tipo puntato.

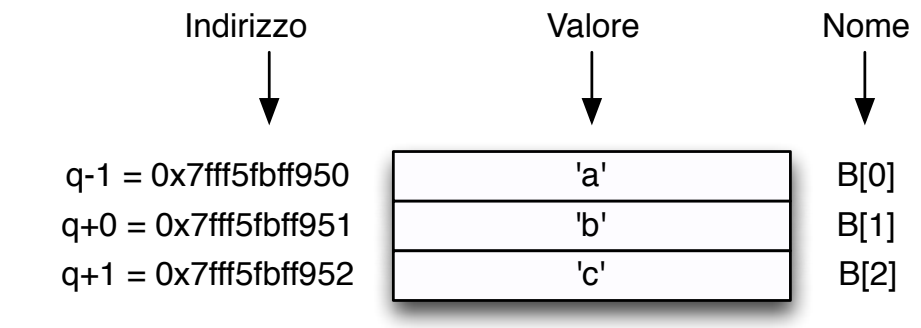
```c
int A[] = {1,2,3};
int *p = &A[0];

// incremento di sizeof(int)
p++;
printf("%d\n",*(p-1)); // Stampa 1
printf("%d\n",*(p+0)); // Stampa 2
printf("%d\n",*(p+1)); // Stampa 3
```

```c
char B[] = {'a','b','c'};
char *q = &B[0];

// incremento di sizeof(char)
q++;
printf("%c\n",*(q-1)); // Stampa a
printf("%c\n",*(q+0)); // Stampa b
printf("%c\n",*(q+1)); // Stampa c
```





Aritmetica dei puntatori: somma e sottrazione con tipi numerici

- Possiamo sommare/sottrarre valori interi ad un puntatore. Il nuovo indirizzo di memoria viene calcolato moltiplicando il valore intero per la dimensione del tipo puntato.

```c
int A[] = {1,2,3};
int *p = &A[0];

// incremento di sizeof(int)
p++;
printf("%d\n",*(p-1)); // Stampa 1
printf("%d\n",*(p+0)); // Stampa 2
printf("%d\n",*(p+1)); // Stampa 3
```

```c
char B[] = {'a','b','c'};
char *q = &B[0];

// incremento di sizeof(char)
q++;
printf("%c\n",*(q-1)); // Stampa a
printf("%c\n",*(q+0)); // Stampa b
printf("%c\n",*(q+1)); // Stampa c
```

Indirizzo	Valore	Nome
p-1 = 0x7fff5fbff950	1	A[0]
p+0 = 0x7fff5fbff954	2	A[1]
p+1 = 0x7fff5fbff958	3	A[2]

Indirizzo	Valore	Nome
q-1 = 0x7fff5fbff950	'a'	B[0]
q+0 = 0x7fff5fbff951	'b'	B[1]
q+1 = 0x7fff5fbff952	'c'	B[2]

- Non possiamo sommare o sottrarre valori in virgola mobile.

```c
int   x, *p = &x+1.2; // Errore di sintassi
float y, *q = &y-1.2; // Errore di sintassi
```

Aritmetica dei puntatori: somma e sottrazione tra puntatori

- Possiamo eseguire l'operazione di sottrazione tra due puntatori dello stesso tipo.

```
1 int A[] = {1,2,3,4,5};
2 int *p  = &A[0];
3 int *q  = &A[5];
4 int *r  = &A[3];
5
6 printf("%d\n",q-p); // Stampa  5
7 printf("%d\n",p-q); // Stampa -5
8 printf("%d\n",r-p); // Stampa  3
9 printf("%d\n",p-r); // Stampa -3
```

Lo standard ISO C89 dichiara inoltre che l'operazione di sottrazione tra due puntatori che non puntano a elementi dello stesso array (o, al limite, al primo byte dopo l'ultimo elemento dell'array) è non definita.

Aritmetica dei puntatori: somma e sottrazione tra puntatori

- Possiamo eseguire l'operazione di sottrazione tra due puntatori dello stesso tipo.

```
1 int A[] = {1,2,3,4,5};
2 int *p  = &A[0];
3 int *q  = &A[5];
4 int *r  = &A[3];
5
6 printf("%d\n",q-p); // Stampa  5
7 printf("%d\n",p-q); // Stampa -5
8 printf("%d\n",r-p); // Stampa  3
9 printf("%d\n",p-r); // Stampa -3
```

Lo standard ISO C89 dichiara inoltre che l'operazione di sottrazione tra due puntatori che non puntano a elementi dello stesso array (o, al limite, al primo byte dopo l'ultimo elemento dell'array) è non definita.

- Non possiamo sommare due puntatori, anche se dello stesso tipo.

```
1 int A[] = {1,2,3,4,5};
2 int *p  = &A[0];
3 int *q  = &A[5];
4 int *r  = p+q; // Errore di sintassi
```

Aritmetica dei puntatori: somma e sottrazione tra puntatori

- Possiamo eseguire l'operazione di sottrazione tra due puntatori dello stesso tipo.

```c
int A[] = {1,2,3,4,5};
int *p  = &A[0];
int *q  = &A[5];
int *r  = &A[3];

printf("%d\n",q-p); // Stampa  5
printf("%d\n",p-q); // Stampa -5
printf("%d\n",r-p); // Stampa  3
printf("%d\n",p-r); // Stampa -3
```

  Lo standard ISO C89 dichiara inoltre che l'operazione di sottrazione tra due puntatori che non puntano a elementi dello stesso array (o, al limite, al primo byte dopo l'ultimo elemento dell'array) è non definita.

- Non possiamo sommare due puntatori, anche se dello stesso tipo.

```c
int A[] = {1,2,3,4,5};
int *p  = &A[0];
int *q  = &A[5];
int *r  = p+q; // Errore di sintassi
```

- Non possiamo sottrarre due puntatori di tipo differente.

```c
int   x, *p = &x;
float y, *q = &y;

p-q; // Errore di sintassi
```

Aritmetica dei puntatori: operatori relazionali

- Possiamo utilizzare gli operatori relazionali tra puntatori dello stesso tipo

```
1 #define N 5
2 int A[N] = {1,2,3,4,5};
3 int *p;
4
5 // Stampa gli elementi dell'array
6 for(p=&A[0]; p<&A[N]; p++)
7   printf("%d\n",*p);
```

Aritmetica dei puntatori: operatori relazionali

- Possiamo utilizzare gli operatori relazionali tra puntatori dello stesso tipo

```
1 #define N 5
2 int A[N] = {1,2,3,4,5};
3 int *p;
4
5 // Stampa gli elementi dell'array
6 for(p=&A[0]; p<&A[N]; p++)
7   printf("%d\n",*p);
```

- Possiamo confrontare un puntatore con la costante 0 (equivale al confronto con `NULL`).

```
1 int *p = NULL;
2
3 if(p == 0) printf("Null pointer\n");
```

Aritmetica dei puntatori: operatori relazionali

- Possiamo utilizzare gli operatori relazionali tra puntatori dello stesso tipo

```
1 #define N 5
2 int A[N] = {1,2,3,4,5};
3 int *p;
4
5 // Stampa gli elementi dell'array
6 for(p=&A[0]; p<&A[N]; p++)
7   printf("%d\n",*p);
```

- Possiamo confrontare un puntatore con la costante 0 (equivale al confronto con NULL).

```
1 int *p = NULL;
2
3 if(p == 0) printf("Null pointer\n");
```

- Non possiamo confrontare un puntatore con valori in virgola mobile e non è *pulito* il confronto con valori interi diversi da 0 o tra puntatori non compatibili senza cast.

```
1 int  *p = NULL;
2 char *q = NULL;
3
4 if(p == 0.0)       // Errore di sintassi
5   printf("Null pointer\n");
6
7 if(p < 1 && p==q) // Warning: confronto con 1 e p,q non compatibili
8   printf("Null pointers\n");
```

Aritmetica dei puntatori: operatori non utilizzabili

- Non possiamo eseguire le seguenti operazioni aritmetiche e bitwise tra puntatori (anche dello stesso tipo) e tipi numerici.

```
1  int x, *p = &x;
2  int y, *q = &y;
3
4  p * q;   // Errore di sintassi: moltiplicazione
5  p * 1;   // Errore di sintassi: moltiplicazione
6  p / q;   // Errore di sintassi: divisione
7  p / 1;   // Errore di sintassi: divisione
8  p % q;   // Errore di sintassi: modulo
9  p % 1;   // Errore di sintassi: modulo
10
11 p & q;   // Errore di sintassi: AND bitwise
12 p & 1;   // Errore di sintassi: AND bitwise
13 p | q;   // Errore di sintassi: OR bitwise
14 p | 1;   // Errore di sintassi: OR bitwise
15 p ^ q;   // Errore di sintassi: XOR bitwise
16 p ^ 1;   // Errore di sintassi: XOR bitwise
17 ~p;      // Errore di sintassi: NOT bitwise
18 p << q;  // Errore di sintassi: SHIFT bitwise
19 p << 1;  // Errore di sintassi: SHIFT bitwise
20 p >> q;  // Errore di sintassi: SHIFT bitwise
21 p >> 1;  // Errore di sintassi: SHIFT bitwise
```

Puntatori e array: introduzione

- In C esiste una stretta relazione tra puntatori e array. Abbiamo già notato che:
  - un riferimento al nome di un array in una espressione è un riferimento all'indirizzo di memoria del primo elemento dell'array.
  - un riferimento al nome di una variabile puntatore è un riferimento all'indirizzo di memoria assegnato alla variabile puntata.
- Le similitudini tra puntatori e array vanno ben oltre questo aspetto ma ci sono anche profonde differenze.
  - Possiamo utilizzare puntatori facendo uso della notazione array.
  - Non possiamo sempre utilizzare i puntatori come tipi di dato array e viceversa.
- La principale difficoltà nel comprendere le relazioni tra array e puntatori risiede quasi unicamente nella difficoltà a comprendere il tipo dell'oggetto puntato.
  - La semantica delle operazioni aritmetiche tra puntatori e con puntatori dipende strettamente dal tipo dell'oggetto puntato.

Puntatori e array: preliminari

- **Puntatori ad int**.

```
1 int A[10];
2 int *p = A;
3 printf("%p\n",(void *)(p+1));
```

Puntatori e array: preliminari

- **Puntatori ad int**.

```
1 int A[10];
2 int *p = A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad `int`. L'espressione `A` è un riferimento all'indirizzo di un `int`. La `printf()` stampa l'indirizzo del primo elemento di `A[]` incrementato di `sizeof(int)`.

Puntatori e array: preliminari

- **Puntatori ad int**.

```
1 int A[10];
2 int *p = A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad `int`. L'espressione `A` è un riferimento all'indirizzo di un `int`. La `printf()` stampa l'indirizzo del primo elemento di `A[]` incrementato di `sizeof(int)`.

- **Puntatori ad array di int**.

```
1 int A[10];
2 int (*p)[10] = &A;
3 printf("%p\n",(void *)(p+1));
```

Puntatori e array: preliminari

- **Puntatori ad int**.

```
1 int A[10];
2 int *p = A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad `int`. L'espressione `A` è un riferimento all'indirizzo di un `int`. La `printf()` stampa l'indirizzo del primo elemento di `A[]` incrementato di `sizeof(int)`.

- **Puntatori ad array di int**.

```
1 int A[10];
2 int (*p)[10] = &A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad array contenente 10 `int`. L'espressione `&A` è un riferimento all'indirizzo di un array contenente 10 `int`. **Attenzione**: nonostante le espressioni `A` e `&A` producano lo stesso indirizzo, **il tipo dell'espressione è differente**. La `printf()` stampa l'indirizzo dell'array `A[]` incrementato di `10*sizeof(int)`.

Puntatori e array: preliminari

- **Puntatori ad int**.

```
1 int A[10];
2 int *p = A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad `int`. L'espressione `A` è un riferimento all'indirizzo di un `int`. La `printf()` stampa l'indirizzo del primo elemento di `A[]` incrementato di `sizeof(int)`.

- **Puntatori ad array di int**.

```
1 int A[10];
2 int (*p)[10] = &A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad array contenente 10 `int`. L'espressione `&A` è un riferimento all'indirizzo di un array contenente 10 `int`. **Attenzione**: nonostante le espressioni `A` e `&A` producano lo stesso indirizzo, **il tipo dell'espressione è differente**. La `printf()` stampa l'indirizzo dell'array `A[]` incrementato di `10*sizeof(int)`.

- **Array di puntatori ad int**.

```
1 int *p[10];
2 printf("%p\n",(void *)(p+1));
```

Puntatori e array: preliminari

- **Puntatori ad int**.

```
1 int A[10];
2 int *p = A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad `int`. L'espressione `A` è un riferimento all'indirizzo di un `int`. La `printf()` stampa l'indirizzo del primo elemento di `A[]` incrementato di `sizeof(int)`.

- **Puntatori ad array di int**.

```
1 int A[10];
2 int (*p)[10] = &A;
3 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un puntatore ad array contenente 10 `int`. L'espressione `&A` è un riferimento all'indirizzo di un array contenente 10 `int`. **Attenzione**: nonostante le espressioni `A` e `&A` producano lo stesso indirizzo, **il tipo dell'espressione è differente**. La `printf()` stampa l'indirizzo dell'array `A[]` incrementato di `10*sizeof(int)`.

- **Array di puntatori ad int**.

```
1 int *p[10];
2 printf("%p\n",(void *)(p+1));
```

La variabile `p` è un array contenente 10 puntatori ad `int`. La `printf()` stampa l'indirizzo del primo elemento dell'array `p` incrementato di `sizeof(int *)`.

Puntatori e array: puntatori visti come array

- L'equivalenza di tipo tra puntatori ad un tipo Type e array di Type, ci permette di effettuare qualsiasi operazione di indicizzazione degli array di Type tramite puntatori a Type: possiamo utilizzare la *notazione array* anche con i puntatori.

```
1 int A[] = {1,2,3};
2 int *p = A; // Copia l'indirizzo del primo elemento di A in p
3
4 printf("%d = %d\n",p[0],A[0]);
5 printf("%d = %d\n",p[1],A[1]);
6 printf("%d = %d\n",p[2],A[2]);
```

Puntatori e array: puntatori visti come array

- L'equivalenza di tipo tra puntatori ad un tipo Type e array di Type, ci permette di effettuare qualsiasi operazione di indicizzazione degli array di Type tramite puntatori a Type: possiamo utilizzare la *notazione array* anche con i puntatori.

```
1 int A[] = {1,2,3};
2 int *p = A; // Copia l'indirizzo del primo elemento di A in p
3
4 printf("%d = %d\n",p[0],A[0]);
5 printf("%d = %d\n",p[1],A[1]);
6 printf("%d = %d\n",p[2],A[2]);
```

- Possiamo accedere con un puntatore *compatibile* ad un qualsiasi punto dell'array.

```
1 int A[] = {1,2,3};
2 int *p = &A[1]; // Copia l'indirizzo di A[1] in p
3
4 printf("%d = %d\n",p[-1],A[0]);
5 printf("%d = %d\n",p[0], A[1]);
6 printf("%d = %d\n",p[1], A[2]);
```

Puntatori e array: puntatori visti come array

- L'equivalenza di tipo tra puntatori ad un tipo Type e array di Type, ci permette di effettuare qualsiasi operazione di indicizzazione degli array di Type tramite puntatori a Type: possiamo utilizzare la *notazione array* anche con i puntatori.

```
1 int A[] = {1,2,3};
2 int *p = A; // Copia l'indirizzo del primo elemento di A in p
3
4 printf("%d = %d\n",p[0],A[0]);
5 printf("%d = %d\n",p[1],A[1]);
6 printf("%d = %d\n",p[2],A[2]);
```

- Possiamo accedere con un puntatore *compatibile* ad un qualsiasi punto dell'array.

```
1 int A[] = {1,2,3};
2 int *p = &A[1]; // Copia l'indirizzo di A[1] in p
3
4 printf("%d = %d\n",p[-1],A[0]);
5 printf("%d = %d\n",p[0], A[1]);
6 printf("%d = %d\n",p[1], A[2]);
```

- Una variabile puntatore (anche se read-only) non è equivalente ad una variabile array.

```
1 int A[] = {1,2,3};
2 int * const p = A; // p     un puntatore read-only
3
4 printf("%lu\n",sizeof(A)); // Stampa 3*sizeof(int)
5 printf("%lu\n",sizeof(p)); // Stampa sizeof(int *)
```

Puntatori e array: array visti come puntatori

- Il compilatore trasforma implicitamente ogni espressione in *notazione array* in una equivalente *espressione aritmetica con tipo puntatore*:

$$\texttt{A[i]} \Longrightarrow \texttt{*(A+i)}$$

```
1 int A[] = {1,2,3};
2 int *p = A;
3
4 // Quattro modi equivalenti per accedere agli
5 // elementi di un array
6 printf("%d = %d = %d = %d\n",p[0],*(p+0),*(A+0),A[0]);
7 printf("%d = %d = %d = %d\n",p[1],*(p+1),*(A+1),A[1]);
8 printf("%d = %d = %d = %d\n",p[2],*(p+2),*(A+2),A[2]);
```

Puntatori e array: array visti come puntatori

- Il compilatore trasforma implicitamente ogni espressione in *notazione array* in una equivalente *espressione aritmetica con tipo puntatore*:

$$\texttt{A[i]} \Longrightarrow \texttt{*(A+i)}$$

```c
int A[] = {1,2,3};
int *p = A;

// Quattro modi equivalenti per accedere agli
// elementi di un array
printf("%d = %d = %d = %d\n",p[0],*(p+0),*(A+0),A[0]);
printf("%d = %d = %d = %d\n",p[1],*(p+1),*(A+1),A[1]);
printf("%d = %d = %d = %d\n",p[2],*(p+2),*(A+2),A[2]);
```

- La regola di conversione da *notazione array* ad *espressione aritmetica con tipo puntatore* viene applicata indipendentemente dal fatto che si lavori o meno con array.

```c
int  x = 1, y;
int *p = &x;

y = p[0];          // *(p+0)          = *p
y = 0[p];          // *(0+p)          = *p
y = (1-1)[p];      // *(1-1+p)        = *(0+p) = *p
y = (x-x)[p];      // *(x-x+p)        = *(0+p) = *p
y = (&p-&p)[p];    // *(&p-&p+p)      = *(0+p) = *p
```

In tutti i casi, alla variabile `y` verrà assegnato il valore della variabile `x`.

Puntatori e array multidimensionali 1/2

- La regola di conversione da *notazione array* ad *espressione aritmetica con tipo puntatore* viene applicata anche agli array bidimensionali.

$$\texttt{A[i][j]} \Longrightarrow \texttt{(*(A+i))[j]} \Longrightarrow \texttt{*(*(A+i)+j)}$$

Puntatori e array multidimensionali 1/2

- La regola di conversione da *notazione array* ad *espressione aritmetica con tipo puntatore* viene applicata anche agli array bidimensionali.

$$\texttt{A[i][j]} \Longrightarrow \texttt{(*(A+i))[j]} \Longrightarrow \texttt{*(*(A+i)+j)}$$

- Possiamo *facilmente* generalizzarla ad array multidimensionali.

$$\texttt{A[i][j][k]} \Longrightarrow \texttt{(*(A+i))[j][k]} \Longrightarrow \texttt{(*(*(A+i)+j))[k]} \Longrightarrow \texttt{*(*(*(A+i)+j)+k)}$$

Puntatori e array multidimensionali 1/2

- La regola di conversione da *notazione array* ad *espressione aritmetica con tipo puntatore* viene applicata anche agli array bidimensionali.

$$\texttt{A[i][j]} \Longrightarrow \texttt{(*(A+i))[j]} \Longrightarrow \texttt{*(*(A+i)+j)}$$

- Possiamo *facilmente* generalizzarla ad array multidimensionali.

$$\texttt{A[i][j][k]} \Longrightarrow \texttt{(*(A+i))[j][k]} \Longrightarrow \texttt{(*(*(A+i)+j))[k]} \Longrightarrow \texttt{*(*(*(A+i)+j)+k)}$$

- Concentriamoci sugli array bidimensionali (generalizziamo facilmente per array multidimensionali). Per riuscire a capire la semantica della regola di conversione per array bidimensionali è necessario prestare molta attenzione ai tipi di dato su cui lavoriamo.
  - Le operazioni aritmetiche nella conversione dipendono strettamente dal tipo di `A[][]`.
- In un array bidimensionale definito con la dichiarazione `int A[M][N]`:
  1. l'elemento `A[i]` è una variabile (read-only) di tipo **array di** `N int`,
  2. l'espressione `A` è sinonimo dell'indirizzo di memoria del primo elemento di `A` (equivalente a `&A[0]`).

Puntatori e array multidimensionali 1/2

- La regola di conversione da *notazione array* ad *espressione aritmetica con tipo puntatore* viene applicata anche agli array bidimensionali.

$$\texttt{A[i][j]} \Longrightarrow \texttt{(*(A+i))[j]} \Longrightarrow \texttt{*(*(A+i)+j)}$$

- Possiamo *facilmente* generalizzarla ad array multidimensionali.

$$\texttt{A[i][j][k]} \Longrightarrow \texttt{(*(A+i))[j][k]} \Longrightarrow \texttt{(*(*(A+i)+j))[k]} \Longrightarrow \texttt{*(*(*(A+i)+j)+k)}$$

- Concentriamoci sugli array bidimensionali (generalizziamo facilmente per array multidimensionali). Per riuscire a capire la semantica della regola di conversione per array bidimensionali è necessario prestare molta attenzione ai tipi di dato su cui lavoriamo.
  - Le operazioni aritmetiche nella conversione dipendono strettamente dal tipo di `A[][]`.
- In un array bidimensionale definito con la dichiarazione `int A[M][N]`:
  1. l'elemento `A[i]` è una variabile (read-only) di tipo **array di** `N int`,
  2. l'espressione `A` è sinonimo dell'indirizzo di memoria del primo elemento di `A` (equivalente a `&A[0]`).
- Vogliamo adesso risolvere il seguente quesito:
  - Come possiamo definire un puntatore che ci permetta di accedere correttamente agli elementi di un array bidimensionale utilizzando la `notazione array`?

Puntatori e array multidimensionali 2/2

- Quali delle seguenti dichiarazioni di puntatori `p,q,r,s` ci permette di accedere correttamente agli elementi dell'array bidimensionale `A[][]`?

```
1  int A[2][3] = {{1,2,3},{4,5,6}};
2
3  int (*p)[2]    = A;
4  int (*q)[3]    = A;
5  int (*r)[2][3] = A;
6  int **s        = A;
7
8  printf("%d = %d and %d = %d\n",p[0][0],A[0][0],p[1][0],A[1][0]);
9  printf("%d = %d and %d = %d\n",q[0][0],A[0][0],q[1][0],A[1][0]);
10 printf("%d = %d and %d = %d\n",r[0][0],A[0][0],r[1][0],A[1][0]);
11 printf("%d = %d and %d = %d\n",s[0][0],A[0][0],s[1][0],A[1][0]);
```

Nota: il codice è sintatticamente corretto anche se diversi assegnamenti non sono compatibili e in alcuni casi l'output della `printf()` è non definito (possibile crash).

Puntatori e array multidimensionali 2/2

- Quali delle seguenti dichiarazioni di puntatori `p,q,r,s` ci permette di accedere correttamente agli elementi dell'array bidimensionale `A[][]`?

```
1  int A[2][3] = {{1,2,3},{4,5,6}};
2
3  int (*p)[2]     = A;
4  int (*q)[3]     = A;
5  int (*r)[2][3]  = A;
6  int **s         = A;
7
8  printf("%d = %d and %d = %d\n",p[0][0],A[0][0],p[1][0],A[1][0]);
9  printf("%d = %d and %d = %d\n",q[0][0],A[0][0],q[1][0],A[1][0]);
10 printf("%d = %d and %d = %d\n",r[0][0],A[0][0],r[1][0],A[1][0]);
11 printf("%d = %d and %d = %d\n",s[0][0],A[0][0],s[1][0],A[1][0]);
```

  Nota: il codice è sintatticamente corretto anche se diversi assegnamenti non sono compatibili e in alcuni casi l'output della `printf()` è non definito (possibile crash).
- L'espressione `A` è l'indirizzo di memoria di un array di 3 `int`. L'unico assegnamento compatibile è quello a riga 4. L'unica stampa *corretta* è quella a riga 9.
- Il puntatore `p` accede correttamente solo al primo elemento della prima riga di `A[][]`.
- Il puntatore `r` accede all'indirizzo del primo elemento di `A[][]` e al primo indirizzo subito dopo l'ultimo elemento di `A[][]`.
- Il puntatore `s` confonde gli elementi interi di `A[][]` con indirizzi di memoria (crash).

Puntatori e array multidimensionali 3/3

- Per poter comprendere interamente l'esempio dobbiamo innanzitutto capire, data la dichiarazione `int A[2][3]`, quali sono i tipi delle varie espressioni che riguardano l'identificatore `A`.

Espressione	Espressione equivalente	Tipo	Tipo convertito in
`&A`		`int (*)[2][3]`	`int (*)[2][3]`
`A`		`int [2][3]`	`int (*)[3]`
`*A`	`A[0]`	`int [3]`	`int *`
`&A[i]`		`int (*)[3]`	`int (*)[3]`
`A[i]`		`int [3]`	`int *`
`*A[i]`	`A[i][0]`	`int`	`int`
`&A[i][j]`		`int *`	`int *`
`A[i][j]`		`int`	`int`

- Le seguenti espressioni, che contengono l'identificatore `A`,

  `&A, A, &A[0], *A, A[0], &A[0][0]`

  hanno tutte lo stesso valore: l'indirizzo di memoria del primo elemento di `A[][]`.
- Il tipo delle espressioni è in alcuni casi molto differente: ad esempio per `&A` e `&A[0][0]`.
- L'aritmetica dei puntatori produce risultati differenti a seconda del tipo del puntatore: l'espressione `&A+1` ha un valore completamente differente da `&A[0][0]+1`.

Puntatori e array multidimensionali: esempi

- Ricordiamo che se `A[][]` è dichiarato come `int A[N][M]`:
  - l'espressione `A` è di tipo `int (*)[M]`,
  - l'espressione `*A` è di tipo `int *`.

Puntatori e array multidimensionali: esempi

- Ricordiamo che se `A[][]` è dichiarato come `int A[N][M]`:
  - l'espressione `A` è di tipo `int (*)[M]`,
  - l'espressione `*A` è di tipo `int *`.
- Come inizializzare un array 2D con una sola passata.

```
1 #define M 10
2
3 void init_matrix(int A[][M], int n, int init) {
4   int *p, *q;
5
6   // p punta al primo elemento di A[][], q all'ultimo
7   for(p=*A, q = *(A+n-1)+M-1; p<=q; p++) *p = init;
8 }
```

Puntatori e array multidimensionali: esempi

- Ricordiamo che se `A[][]` è dichiarato come `int A[N][M]`:
  - l'espressione `A` è di tipo `int (*)[M]`,
  - l'espressione `*A` è di tipo `int *`.
- Come inizializzare un array 2D con una sola passata.

```
1  #define M 10
2
3  void init_matrix(int A[][M], int n, int init) {
4    int *p, *q;
5
6    // p punta al primo elemento di A[][], q all'ultimo
7    for(p=*A, q = *(A+n-1)+M-1; p<=q; p++) *p = init;
8  }
```

- Come stampare il contenuto di un array 2D, utilizzando l'aritmetica dei puntatori.

```
1  void print_matrix(int A[][M], int n) {
2    int (*p)[M], (*q)[M], *r, *s;
3
4    // p punta al primo sotto-array di A[][], q all'ultimo
5    for(p=A, q=A+n-1; p<=q; p++) {
6      // r punta al primo elemento della riga, s all'ultimo
7      for(r=*p, s=*p+M-1; r<=s; r++)
8        printf("%d ",*r);
9      printf("\n");
10   }
11 }
```

Puntatori e array: stringhe

- Una dichiarazione di puntatore non alloca spazio di memoria per l'oggetto puntato ma unicamente per il puntatore.
- Abbiamo una eccezione particolare quando al puntatore è assegnata una stringa letterale.

```
1 char *s = "This is a string";
```

In questo caso, il compilatore alloca spazio di memoria (nel text segment) per la stringa letterale costante assegnata al puntatore.

Puntatori e array: stringhe

- Una dichiarazione di puntatore non alloca spazio di memoria per l'oggetto puntato ma unicamente per il puntatore.
- Abbiamo una eccezione particolare quando al puntatore è assegnata una stringa letterale.

```
1 char *s = "This is a string";
```

In questo caso, il compilatore alloca spazio di memoria (nel text segment) per la stringa letterale costante assegnata al puntatore.

- Questa eccezione è valida solo per le stringhe letterali, non per altri tipi di costanti.

```
1 double *d = 1.0;   // Errore
2 char   *c = '\0'; // Inizializza c a NULL
3 int    *p = 0;    // Inizializza p a NULL
```

Gli assegnamenti a riga 2 e 3 non allocano memoria per il valore costante ma inizializzano la variabile puntatore.

Puntatori e array: stringhe

- Una dichiarazione di puntatore non alloca spazio di memoria per l'oggetto puntato ma unicamente per il puntatore.
- Abbiamo una eccezione particolare quando al puntatore è assegnata una stringa letterale.

```
1 char *s = "This is a string";
```

In questo caso, il compilatore alloca spazio di memoria (nel text segment) per la stringa letterale costante assegnata al puntatore.

- Questa eccezione è valida solo per le stringhe letterali, non per altri tipi di costanti.

```
1 double *d = 1.0;   // Errore
2 char   *c = '\0'; // Inizializza c a NULL
3 int    *p = 0;     // Inizializza p a NULL
```

Gli assegnamenti a riga 2 e 3 non allocano memoria per il valore costante ma inizializzano la variabile puntatore.

- Non possiamo modificare il contenuto della stringa letterale, se questa è assegnata ad un puntatore. Possiamo farlo se è utilizzata per inizializzare un array.

```
1 char *s = "This is a string";
2
3 s[10]='S'; // Errore
```

```
1 char s[] = "This is a string";
2
3 s[10]='S'; OK
```

Puntatori e array: riassunto equivalenze

1. Il nome di un array in una espressione è trattato dal compilatore come un puntatore al primo elemento dell'array.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[0] = %d\n",*A);
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[0] = %d\n",*p);
```

Puntatori e array: riassunto equivalenze

1. Il nome di un array in una espressione è trattato dal compilatore come un puntatore al primo elemento dell'array.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[0] = %d\n",*A);
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[0] = %d\n",*p);
```

2. L'indicizzazione di un elemento di un array è equivalente ad una espressione aritmetica con tipo puntatore.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[2] = %d\n",*(A+2));
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[2] = %d\n",p[2]);
```

Puntatori e array: riassunto equivalenze

1. Il nome di un array in una espressione è trattato dal compilatore come un puntatore al primo elemento dell'array.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[0] = %d\n",*A);
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[0] = %d\n",*p);
```

2. L'indicizzazione di un elemento di un array è equivalente ad una espressione aritmetica con tipo puntatore.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[2] = %d\n",*(A+2));
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("A[2] = %d\n",p[2]);
```

3. Il nome di un array in una dichiarazione di funzione è trattato dal compilatore come un puntatore al primo elemento dell'array.

```c
void f(int A[][M], int n) {
  int i,j;

  for(i=0; i<n; i++)
    for(j=0; j<M; j++)
      A[i][j]=0;
}
```

```c
void f(int (*A)[M], int n) {
  int i,j;

  for(i=0; i<n; i++)
    for(j=0; j<M; j++)
      A[i][j]=0;
}
```

Puntatori e array: riassunto differenze

1. L'operatore `sizeof` è valutato in maniera differente se applicato ad array e puntatori.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A size = %lu\n",sizeof(A));
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("p size = %lu\n",sizeof(p));
```

Puntatori e array: riassunto differenze

1. L'operatore `sizeof` è valutato in maniera differente se applicato ad array e puntatori.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A size = %lu\n",sizeof(A));
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("p size = %lu\n",sizeof(p));
```

2. Non possiamo modificare una stringa letterale assegnata ad un puntatore. Possiamo farlo se la assegniamo ad un array.

```c
char s[] = "This is a string"

s[10] = 'S'; // OK
```

```c
char *s = "This is a string"

s[10] = 'S'; // Errore
```

Puntatori e array: riassunto differenze

1. L'operatore `sizeof` è valutato in maniera differente se applicato ad array e puntatori.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A size = %lu\n",sizeof(A));
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("p size = %lu\n",sizeof(p));
```

2. Non possiamo modificare una stringa letterale assegnata ad un puntatore. Possiamo farlo se la assegniamo ad un array.

```c
char s[] = "This is a string"

s[10] = 'S'; // OK
```

```c
char *s = "This is a string"

s[10] = 'S'; // Errore
```

3. Non possiamo assegnare valori ad una variabile di tipo array, possiamo farlo con una variabile puntatore.

```c
int A[] = {1,2,3};
int  *p = A;

A = NULL; // Errore
```

```c
int A[] = {1,2,3};
int  *p = A;

p = NULL; // OK
```

Puntatori e array: riassunto differenze

1. L'operatore `sizeof` è valutato in maniera differente se applicato ad array e puntatori.

```c
int A[] = {1,2,3};
int  *p = A;

printf("A size = %lu\n",sizeof(A));
```

```c
int A[] = {1,2,3};
int  *p = A;

printf("p size = %lu\n",sizeof(p));
```

2. Non possiamo modificare una stringa letterale assegnata ad un puntatore. Possiamo farlo se la assegniamo ad un array.

```c
char s[] = "This is a string"

s[10] = 'S'; // OK
```

```c
char *s = "This is a string"

s[10] = 'S'; // Errore
```

3. Non possiamo assegnare valori ad una variabile di tipo array, possiamo farlo con una variabile puntatore.

```c
int A[] = {1,2,3};
int  *p = A;

A = NULL; // Errore
```

```c
int A[] = {1,2,3};
int  *p = A;

p = NULL; // OK
```

4. Esistono altre differenze considerevoli legate alla keyword extern (non le vediamo).

Puntatori e strutture

- Possiamo definire strutture che contengano campi di tipo puntatore.

```
1 struct person {
2   char *name;
3   int age;
4   int height;
5   int weight;
6 };
```

- Possiamo dichiarare variabili di tipo puntatore a struttura.

```
1 struct person {
2   char *name;
3   int age;
4   int height;
5   int weight;
6 };
7
8 struct person *alice, *bob;
```

Il linguaggio C ha alcune regole sintattiche specifiche per:

- l'accesso ai campi di una struttura tramite puntatori.
- la dichiarazione di strutture autoreferenziali tramite puntatori.

Puntatori e strutture: accesso ai campi di una struttura tramite puntatori

- Per poter accedere ai campi di una struttura tramite puntatore possiamo usare l'operatore di dereferenziazione *, come già visto.

```
1 struct person {
2   char *name;
3   int age;
4   int height;
5   int weight;
6 } a, b, *alice=&a, *bob=&b;
7
8 (*alice).name = "Alice";
9 (*bob).name   = "Bob";
```

Le parentesi tonde sono necessarie: l'operatore . ha precedenza maggiore di *.

Puntatori e strutture: accesso ai campi di una struttura tramite puntatori

- Per poter accedere ai campi di una struttura tramite puntatore possiamo usare l'operatore di dereferenziazione *, come già visto.

```
1 struct person {
2   char *name;
3   int age;
4   int height;
5   int weight;
6 } a, b, *alice=&a, *bob=&b;
7
8 (*alice).name = "Alice";
9 (*bob).name   = "Bob";
```

Le parentesi tonde sono necessarie: l'operatore . ha precedenza maggiore di *.

- La sintassi del linguaggio C ci mette a disposizione dell'**operatore freccia** ->, specifico per accedere ai campi di una struttura tramite puntatore.

```
1 struct person {
2   char *name;
3   int age;
4   int height;
5   int weight;
6 } a, b, *alice=&a, *bob=&b;
7
8 alice->name = "Alice";
9 bob->name   = "Bob";
```

Puntatori e strutture: strutture autoreferenziali

- Per alcuni problemi specifici è utile avere la possibilità di definire **strutture autoreferenziali**, cioè che contengano campi il cui tipo è la struttura stessa.
- In C non è possibile definire campi di una struttura il cui tipo sia la struttura stessa ma è possibile definire campi di tipo puntatore alla stessa struttura.

```
1 struct int_list {
2   int x;
3   struct int_list *next; // autoreferenza
4 };
```

Puntatori e strutture: strutture autoreferenziali

- Per alcuni problemi specifici è utile avere la possibilità di definire **strutture autoreferenziali**, cioè che contengano campi il cui tipo è la struttura stessa.
- In C non è possibile definire campi di una struttura il cui tipo sia la struttura stessa ma è possibile definire campi di tipo puntatore alla stessa struttura.

```c
struct int_list {
  int x;
  struct int_list *next; // autoreferenza
};
```

- E' uno dei pochi esempi di **forward declaration** in C (come per i prototipi). E' necessario che la struttura abbia un tag: non si applica alla `typedef`.

```c
typedef struct {
  int x;
  int_list *next; // Errore
} int_list;
```

```c
typedef struct int_list {
  int x;
  struct int_list *next; // OK
} int_list;
```

- Le strutture autoreferenziali tramite puntatori sono lo strumento principale del linguaggio C per la costruzione di strutture dinamiche.

Puntatori e funzioni

- Possiamo definire/dichiarare funzioni che abbiano parametri formali e/o valore di ritorno di tipo puntatore.
- Le funzioni C supportano unicamente il passaggio dei parametri per valore. Il **passaggio per riferimento** può essere *simulato* facendo uso di puntatori.
  - Passiamo alla funzione l'indirizzo di memoria del dato da modificare.
  - La funzione riceve e mantiene l'indirizzo di memoria passato in input in una variabile locale (argomento formale) di tipo puntatore.
  - All'interno della funzione possiamo modificare il contenuto della variabile esterna tramite dereferenziazione.

Puntatori e funzioni

- Possiamo definire/dichiarare funzioni che abbiano parametri formali e/o valore di ritorno di tipo puntatore.
- Le funzioni C supportano unicamente il passaggio dei parametri per valore. Il **passaggio per riferimento** può essere *simulato* facendo uso di puntatori.
  - Passiamo alla funzione l'indirizzo di memoria del dato da modificare.
  - La funzione riceve e mantiene l'indirizzo di memoria passato in input in una variabile locale (argomento formale) di tipo puntatore.
  - All'interno della funzione possiamo modificare il contenuto della variabile esterna tramite dereferenziazione.
- Un **valore di ritorno di tipo puntatore** è un indirizzo di memoria.
  - Dobbiamo ricordare che indirizzi di variabili locali (non `static`) alla funzione corrispondono a locazioni di memoria nel record di attivazione della funzione.
  - Il contenuto di tali locazioni di memoria è impredicibile non appena la funzione termina la sua esecuzione.
  - Ad esempio, non è semanticamente corretto (ma sintatticamente permesso) ritornare l'indirizzo di un array locale al blocco della funzione.

Puntatori e funzioni: esempio

- Un esempio classico per mostrare l'utilità del passaggio per riferimento è quello di definire una funzione che permetta di scambiare il valore di due variabili.
- Esempio non corretto: `swap()` scambia i valori delle proprie variabili locali `x` e `y`.

```
1  void swap(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6  int main() {
7    int x = 0, y = 1;
8    swap(x,y);
9    return 0;
10 }
```

Puntatori e funzioni: esempio

- Un esempio classico per mostrare l'utilità del passaggio per riferimento è quello di definire una funzione che permetta di scambiare il valore di due variabili.
- Esempio non corretto: `swap()` scambia i valori delle proprie variabili locali `x` e `y`.

```
1  void swap(int x, int y) {
2    int tmp = x;
3    x = y;
4    y = tmp;
5  }
6  int main() {
7    int x = 0, y = 1;
8    swap(x,y);
9    return 0;
10 }
```

- Esempio corretto: passiamo a `swap()` gli indirizzi delle variabili da scambiare.

```
1  void swap(int *x, int *y) {
2    int tmp = *x;
3    *x = *y;
4    *y = tmp;
5  }
6  int main() {
7    int x = 0,  y = 1;
8    swap(&x,&y);
9    return 0;
10 }
```

Puntatori e funzioni: passaggio di parametri al `main()`

- Nello standard ISO C89, abbiamo due sole possibilità per definire la funzione `main()` (ricordiamo che non è necessario dichiararne il prototipo).

```
1 int main(void);
```

oppure

```
1 int main(int argc, char *argv[]);
```

In questo secondo caso, i nomi `argc` e `argv` sono utilizzati di default, per quanto possano essere scelti altri nomi per gli argomenti.

Puntatori e funzioni: passaggio di parametri al `main()`

- Nello standard ISO C89, abbiamo due sole possibilità per definire la funzione `main()` (ricordiamo che non è necessario dichiararne il prototipo).

```
1 int main(void);
```

oppure

```
1 int main(int argc, char *argv[]);
```

In questo secondo caso, i nomi `argc` e `argv` sono utilizzati di default, per quanto possano essere scelti altri nomi per gli argomenti.

- Il passaggio dei parametri al `main()` viene effettuato da *riga di comando*.
  - Il parametro `argc` contiene il numero di argomenti passati.
  - Il parametro `argv[]` è un vettore (di lunghezza `argc`) di puntatori a stringhe. La stringa `argv[0]` *contiene* il nome dell'eseguibile.

```
1 #include <stdio.h>
2
3 // Stampa tutti gli argomenti passati
4 int main(int argc, char *argv[]) {
5   int i;
6   for(i=0; i<argc; i++)
7     printf("argv[%d] = %s\n",i,argv[i]);
8   return 0;
9 }
```

Puntatori e funzioni: funzioni per copiare stringhe 1/2

- Facendo uso della notazione puntatore possiamo provare a vedere come implementare la funzione di libreria `strcpy()` per copiare stringhe.
- Implementiamo esattamente il prototipo definito nell'header `string.h`.
- Versione 1.

```
1 char *strcpy(char *s1, const char *s2) {
2   int i=0;
3
4   for(i=0; i < strlen(s2); i++)
5     s1[i] = s2[i];
6   s1[i]='\0';
7   return s1;
8 }
```

Puntatori e funzioni: funzioni per copiare stringhe 1/2

- Facendo uso della notazione puntatore possiamo provare a vedere come implementare la funzione di libreria `strcpy()` per copiare stringhe.
- Implementiamo esattamente il prototipo definito nell'header `string.h`.
- Versione 1.

```
1 char *strcpy(char *s1, const char *s2) {
2   int i=0;
3
4   for(i=0; i < strlen(s2); i++)
5     s1[i] = s2[i];
6   s1[i]='\0';
7   return s1;
8 }
```

Estremamente lenta: la lunghezza di s2 viene ri-calcolata ad ogni ciclo.

- Versione 2.

```
1 char *strcpy(char *s1, const char *s2) {
2   int i=0;
3   while((s1[i]=s2[i])!='\0') i++;
4   return s1;
5 }
```

Puntatori e funzioni: funzioni per copiare stringhe 1/2

- Facendo uso della notazione puntatore possiamo provare a vedere come implementare la funzione di libreria `strcpy()` per copiare stringhe.
- Implementiamo esattamente il prototipo definito nell'header `string.h`.
- Versione 1.

```
1 char *strcpy(char *s1, const char *s2) {
2   int i=0;
3
4   for(i=0; i < strlen(s2); i++)
5     s1[i] = s2[i];
6   s1[i]='\0';
7   return s1;
8 }
```

Estremamente lenta: la lunghezza di s2 viene ri-calcolata ad ogni ciclo.

- Versione 2.

```
1 char *strcpy(char *s1, const char *s2) {
2   int i=0;
3   while((s1[i]=s2[i])!='\0') i++;
4   return s1;
5 }
```

Molto più efficiente e compatta ma possiamo fare a meno della variabile contatore e non abbiamo la necessità di eseguire un test sul carattere nullo.

Puntatori e funzioni: funzioni per copiare stringhe 2/2

- Versione 3.

```
1 char *strcpy(char *s1, const char *s2) {
2   char *s = s1;
3   while(*s1 = *s2) {
4     s1++;
5     s2++;
6   }
7   return s;
8 }
```

Puntatori e funzioni: funzioni per copiare stringhe 2/2

- Versione 3.

```
1 char *strcpy(char *s1, const char *s2) {
2   char *s = s1;
3   while(*s1 = *s2) {
4     s1++;
5     s2++;
6   }
7   return s;
8 }
```

Iniziamo a intravedere la potenza dell'aritmetica dei puntatori. Possiamo compattare ulteriormente il codice.

- Versione 4.

```
1 char *strcpy(char *s1, const char *s2) {
2   char *s = s1;
3   while(*s1++ = *s2++);
4   return s;
5 }
```

Puntatori e funzioni: funzioni per copiare stringhe 2/2

- Versione 3.

```
1 char *strcpy(char *s1, const char *s2) {
2   char *s = s1;
3   while(*s1 = *s2) {
4     s1++;
5     s2++;
6   }
7   return s;
8 }
```

Iniziamo a intravedere la potenza dell'aritmetica dei puntatori. Possiamo compattare ulteriormente il codice.

- Versione 4.

```
1 char *strcpy(char *s1, const char *s2) {
2   char *s = s1;
3   while(*s1++ = *s2++);
4   return s;
5 }
```

L'operatore incremento ha precedenza maggiore rispetto a l'operatore di dereferenza * quindi l'incremento viene effettuato sull'indirizzo di memoria e non sul valore puntato.

Puntatori e funzioni: funzioni per calcolare la lunghezza di stringhe

- Facendo uso della notazione puntatore possiamo provare a vedere come implementare la funzione di libreria `strlen()` per calcolare la lunghezza di una stringa.
- Implementiamo esattamente il prototipo definito nell'header `string.h`.
- Versione 1.

```
1 size_t strlen(const char *s) {
2   size_t i;
3   for(i=0; s[i] !='\0'; i++);
4   return i;
5 }
```

Funziona correttamente ed è efficiente. Possiamo riscriverla completamente facendo uso unicamente dell'aritmetica dei puntatori.

Puntatori e funzioni: funzioni per calcolare la lunghezza di stringhe

- Facendo uso della notazione puntatore possiamo provare a vedere come implementare la funzione di libreria `strlen()` per calcolare la lunghezza di una stringa.
- Implementiamo esattamente il prototipo definito nell'header `string.h`.
- Versione 1.

```
1 size_t strlen(const char *s) {
2   size_t i;
3   for(i=0; s[i] !='\0'; i++);
4   return i;
5 }
```

Funziona correttamente ed è efficiente. Possiamo riscriverla completamente facendo uso unicamente dell'aritmetica dei puntatori.

- Versione 2.

```
1 size_t strlen(const char *s) {
2   const char *t = s;
3   while(*s++);
4   return s-1-t;
5 }
```

La lunghezza della stringa viene calcolata come offset tra l'indirizzo del primo (`t`) ed ultimo carattere non nullo (`s-1`) della stringa.