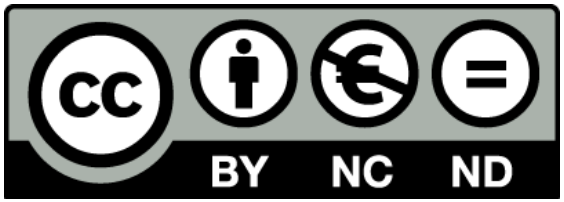
Laboratorio di Programmazione di Sistema

Array e Puntatori

Luca Forlizzi

Versione 20.1



Introduzione

- Nel linguaggio C vi è una relazione molto stretta tra array e puntatori
- Questa relazione è caratteristica del C e non la si trova in altri linguaggi di larga diffusione, ad eccezione del C++
- La relazione è piuttosto complessa e articolata in vari casi, ed è di non facile comprensione per i principianti
- Purtroppo, è una relazione che è necessario conoscere per scrivere programmi che abbiano un minimo di complessità, in quanto essa ha effetti sulla semantica che entrano in gioco in situazioni molto comuni

Introduzione

- L'inevitabilità di far conoscere questa relazione anche a programmatori principianti, ha alimentato il diffondersi di informazioni errate o solo parzialmente corrette circa il rapporto tra array e puntatori
- Abbastanza diffusa è la convinzione che array e puntatori siano "equivalenti"; il che, attribuendo al termine "equivalente" il comune significato, è falso
- Un'altra nozione diffusa, solo parzialmente vera, è quella secondo cui un array si trasformerebbe in "un puntatore costante al primo elemento"
- Per diventare buoni programmatori in C, è necessario comprendere con molta precisione la relazione tra array e puntatori

Introduzione

- La relazione tra array e puntatori si articola in 3 aspetti principali
  1. Aritmetica dei Puntatori
  2. Conversione Implicita Array-Puntatore in Espressione
  3. Conversione Implicita Array-Puntatore in Parametro
- In questa presentazione discutiamo i primi due aspetti, il terzo verrà illustrato negli approfondimenti dedicati alle funzioni
- Nel seguito, usiamo UB come abbreviazione di *undefined behavior*

Uguaglianza tra Puntatori

- Come sappiamo, è possibile applicare gli operatori == e != ad una coppia di puntatori che abbiano lo stesso tipo
- Se due puntatori ad oggetti dello stesso tipo, puntano lo stesso oggetto, essi risultano uguali

```
int x = 0, *p, *q;
p = &x; q = &x;
if (p != q) x += 100;
if (p == q) x += 666;
printf("%d", x); /* stampa 666 */
```

- Di solito è vero anche il viceversa, ma c'è un'eccezione

Puntatore a un Elemento di un Array

- Come mostrato nel capitolo 11 di **[Ki]** è possibile referenziare mediante un puntatore anche un elemento di un array

```
int a[10], *p;
float *p, b[3] = { 4.6, 2.5 }, *q;

p = &b[1];    /* p punta b[1] */
q = &b[2];    /* q punta b[2] */
*q = b[0] + *p;
printf("%f", b[2]); /* stampa 7.1 */
```

Puntatore *1-Oltre-Ultimo-Elemento* di un Array

- Allo scopo di definire l'aritmetica dei puntatori, chiamiamo *elemento 1-Oltre-Ultimo-Elemento* (in inglese *element One-Past-Last-Element*, abbreviato con elemento *OPLE*) di un array, un oggetto immaginario disposto in memoria subito oltre l'ultimo elemento dell'array
- In altre parole, se `N` è la lunghezza di un array, l'elemento OPLE è l'elemento che avrebbe indice `N` se l'array fosse dichiarato con lunghezza maggiore di `N`
- L'elemento OPLE non esiste, dunque è UB accedere ad esso
- L'unica operazione consentita sull'elemento OPLE di un array, è la creazione di un puntatore ad esso, chiamato *Puntatore 1-Oltre-Ultimo-Elemento* (in inglese *Pointer One-Past-Last-Element*, abbreviato con *POPLEL*)

Puntatore *1-Oltre-Ultimo-Elemento* di un Array

- Il POPLEL di un array non può essere dereferenziato, in quanto ciò equivarrebbe ad effettuare un accesso all'elemento OPLE
- Il POPLEL di un array, se confrontato, mediante gli operatori == o !=, con un qualsiasi puntatore ad un vero elemento dello stesso array, risulta diverso
- Tuttavia, potrebbe accadere che un diverso oggetto `obj` venga ad occupare le parole di memoria teoricamente occupate dall'elemento OPLE di un array: in tal caso un puntatore ad `obj` risulterebbe uguale al POPLEL dell'array
- Il motivo per cui viene consentita la creazione di POPLEL di array saranno chiari nel seguito

Esempi di POPLEL

```
int a[10], x, *p, *q;

p = &a[10]; /* legale: p e' POPLEL di a */
*p = 2;     /* UB: accesso ad OPLEL */
a[0] = *p;  /* UB: accesso ad OPLEL */

q = &x;
printf("%d", p == &a[9]); /* stampa 0 */
printf("%d", p == q);     /* probabilmente
  stampa 0 ma potrebbe anche stampare 1 */
printf("%d", *p);         /* UB */
*p = x;                   /* UB */
```

Aritmetica dei Puntatori

- Con *scansione sequenziale* o *attraversamento* di un array si intende un algoritmo esamina tutti gli elementi di un array in ordine di indice crescente o decrescente, come ad esempio

```
for ( s = 0, i = 0 ; i < 10 ; i++ )
                        s += array[ i ];
```

- Come tradurre tale codice in *ASM* MIPS32 ?

Aritmetica dei Puntatori

- Una soluzione che rappresenta la variabile indice mediante un registro (ovvero t1)

```
        li      $t1,0           # i = 0
        li      $s2,0           # s = 0
        la      $s0,array
i_for:  bgeu    $t1,10,f_for    # test controllo ciclo

# calcola indirizzo elemento di indice i
        mulu    $t3,$t1,4       # elementi di 4 byte
        add     $t3,$s0,$t3

# s += array[ i ]
        lw      $t2,($t3)
        add     $s2,$s2,$t2

        add     $t1,$t1,1       # i++
        b       i_for
f_for:
```

Aritmetica dei Puntatori

- Una soluzione che usa un registro che contiene direttamente l'indirizzo dell'elemento cui accedere, e che viene modificato per effettuare la scansione

```
        li      $s2,0           # s = 0

# t0 contiene l'indirizzo elemento cui accedere
        la      $s0,array       # elementi di 4 byte
# t1 contiene l'indirizzo successivo a quello
# dell'ultimo byte di array
        la      $t1,array+40

i_for:  bgeu    $s0,$t1,f_for   # test controllo ciclo

# s += array[ i ]
        lw      $t2,($s0)       # accesso ad array[ i ]
        add     $s2,$s2,$t2

        add     $s0,$s0,4       # prossimo elemento

        b       i_for
f_for:
```

Aritmetica dei Puntatori

- La seconda soluzione impiega meno istruzioni e dunque, nella maggior parte delle implementazioni, è più efficiente
- Una situazione simile si verifica in molte altre architetture
- Tuttavia, non è semplice per un compilatore (specie per quelli realizzati negli anni 70) riconoscere che un determinato ciclo è una scansione lineare, e quindi tradurlo utilizzando l'idea mostrata nella seconda soluzione
- Pertanto, il C offre un meccanismo che permette ai programmatori di impiegare tale idea direttamente al livello di codice C

Aritmetica dei Puntatori

- Nel capitolo 11 di **[Ki]** e in vari esempi abbiamo mostrato che un puntatore può puntare anche ad elementi di un array e può essere usato per modificare tali elementi

```
int a[10], *p;
p = &a[1];
*p = 5;
printf("%d", a[1]); /* stampa 5 */
```

- L'aritmetica dei puntatori consente di utilizzare un puntatore ad un elemento di un array anche per accedere agli *altri* elementi dell'array

Aritmetica dei Puntatori

- L'aritmetica dei puntatori consente di applicare operatori aritmetici a puntatori in 3 modi
  1. Somma di un intero e di un puntatore
  2. Sottrazione di un intero da un puntatore
  3. Sottrazione tra due puntatori
- In **[Ki]**, questo argomento è trattato nelle sezioni 12.1 e 12.2

Somma di un intero e di un puntatore

- Se `p` è un puntatore all'elemento di indice `i` di un array `a`, `p+j` è un puntatore all'elemento di indice `i+j` di `a`

```
int a[10], *p, *q;
p = &a[1];
q = p + 3;        /* q punta a[4] */
*q = 6;
p = &a[6] + 3; /* p punta a[9] */
*p = *q + 2;
printf("%d␣%d", a[9], a[4]); /* stampa 8 6 */
```

- L'operazione è ben definita solo se sono vere entrambe le seguenti condizioni:
  1. `a` ha almeno `i` elementi oppure `a[i]` è l'elemento OPLE di `a`
  2. `a` ha almeno `i+j` elementi oppure `a[i+j]` è l'elemento OPLE di `a`

Sottrazione di un intero da un puntatore

- Se `p` è un puntatore all'elemento di indice `i` di un array `a`, l'espressione `p-j` è un puntatore all'elemento di indice `i-j` di `a`
- L'operazione è ben definita solo se sono vere entrambe le seguenti condizioni:
  1. `a` ha almeno `i` elementi oppure `a[i]` è l'elemento OPLE di `a`
  2. `a` ha almeno `i-j` elementi oppure `a[i-j]` è l'elemento OPLE di `a`

Sottrazione tra due puntatori

- Dati due puntatori che puntano ad elementi dello stesso array, è definita la sottrazione tra essi
- Il risultato è la distanza tra i due puntatori, misurata in numero di elementi
- Ovvero, se `p` punta `a[i]` e `q` punta `a[j]`, `p-q` ha lo stesso valore di `i-j`
- L'operazione `p-q` è ben definita solo se esiste un array `a` che rende vere entrambe le seguenti condizioni:
  1. `p` punta un elemento di `a` oppure è il POPLEL di `a`
  2. `q` punta un elemento di `a` oppure è il POPLEL di `a`

Pre/Post-Incremento/Decremento di un puntatore

- Dalle regole dell'aritmetica dei puntatori, segue anche la definizione degli operatori di pre-incremento, pre-decremento, post-incremento e post-decremento per i puntatori
- `p++` ha come risultato il valore di `p` precedente la valutazione e come side-effect l'assegnamento `p=p+1`
- `++p` ha come side-effect l'assegnamento `p=p+1` e come risultato il valore di `p` dopo che il side-effect si completa
- Gli operatori di incremento sono ben definiti solo se è ben definita `p=p+1`

Pre/Post-Incremento/Decremento di un puntatore

- `p--` ha come risultato il valore di `p` precedente la valutazione e come side-effect l'assegnamento `p=p-1`
- `--p` ha come side-effect l'assegnamento `p=p-1` e come risultato il valore di `p` dopo che il side-effect si completa
- Gli operatori di decremento sono ben definiti solo se è ben definita `p=p-1`

Operatori relazionali e puntatori

- Due puntatori che puntano ad elementi dello stesso array, possono essere confrontati mediante gli operatori <, <=, >=, >
- Se `p` punta `a[i]` e `q` punta `a[j]`, per ciascuno dei 4 operatori <, <=, >=, >, il confronto tra `p` e `q` ha un risultato uguale a quello prodotto dal confronto, mediante lo stesso operatore, tra `i` e `j`
- Il confronto tra `p` e `q` è ben definito solo se esiste un array `a` che rende vere entrambe le seguenti condizioni:
  1. `p` punta un elemento di `a` oppure è il POPLEL di `a`
  2. `q` punta un elemento di `a` oppure è il POPLEL di `a`

Esempi di Aritmetica Puntatori

```
int a[10], *p, *q, *r, *s, x = 3;
p = &a[5];
q = &a[10]; /* legale: q e' POPLEL di a */
r = &a[11]; /* UB */
r = p + 6; /* UB: p+6 e' maggiore di POPLEL */
s = p + 5; /* OK: s e' il POPLEL */
r = s - 2; /* OK: r punta a[8] */
a[0] = *r; /* OK */
a[1] = *s; /* UB: s e' il POPLEL */

s = &a[10] - 0; /* OK: s e' il POPLEL */
q = &a[11] - 3; /* UB: calcola &a[11] */
q = &a[8] + x - 2; /* UB, calcola &a[11]
                      (come passo intermedio) */
q = &a[8] + (x - 2); /* OK: q punta a[9] */
```

Esempio di Scansione Array con Puntatore

```
int a[10], *p, sum;

/* Notare l'utilizzo del POPLEL di a, per
   la condizione di uscita dal ciclo */
for (sum = 0, p = &a[0]; p < &a[10]; p++)
     sum += *p;
```

- Risulta piuttosto diffusa l'opinione secondo cui utilizzare un puntatore invece che un indice per scandire un array, produca un codice più efficiente
- In realtà dipende dall'implementazione
  - Nelle prime implementazioni del C, ciò era generalmente vero
  - Al giorno d'oggi, nella maggior parte dei casi, non vi è differenza nel codice prodotto

Esempio Utilizzo di * e ++

- L'espressione `a[x++] = 1` modifica sia un elemento dell'array che la variabile indice; risulta molto utile per scrivere codice succinto
- Un effetto analogo si ha anche con l'espressione `*p++` in cui al puntatore sono applicati due operatori, * e ++
- Le regole di precedenza stabiliscono che `*p++` viene interpretata come `*(p++)`: il valore di `p++` è pari al contenuto di `p` prima di fare l'incremento, quindi `*p++` accede l'elemento puntato da `p` prima dell'incremento

```
/* Scansione Array utilizzando *p++ */
int a[10], *p = &a[0], sum = 0;
while (p < &a[10]) sum += *p++;
```

Conversione Implicita Array-Puntatore in Espressione

- Il secondo fondamento costitutivo della relazione tra array e puntatori è una conversione implicita che converte alcune (quasi tutte per la verità) occorrenze di un espressione di tipo array, in un puntatore
- La regola è chiamata *Conversione Implicita Array-Puntatore in Espressione (CIAPE)*
- In **[Ki]**, questo argomento è trattato nella sezione 12.3

Conversione Implicita Array-Puntatore in Espressione

- Un'espressione E che ha tipo *array-di-T* viene convertita implicitamente in un'espressione di tipo *puntatore-a-T* che punta il primo elemento dell'array, tranne che nei 3 casi seguenti:
  1. E è operando di `sizeof`
  2. E è operando di `&`
  3. E è uno string literal usato per inizializzare un array
- Si sottolinea che la conversione ha effetto sulle espressioni, non sulle dichiarazioni
- Come tutte le conversioni di tipo, non modifica il tipo dell'oggetto a cui è applicata, ma modifica solo la semantica dell'operando della conversione, all'interno della espressione che contiene la conversione

Esempi CIAPE: dereferenziazione e aritmetica puntatori

```
int a[10], *p;

p = a; /* p punta a[0] */
*a = 1; /* scrive 1 in a[0] */
p = a+2; /* p punta a[2] */
*p = *a; /* copia a[0] in a[2] */
*(a+1) = 5; /* scrive 5 in a[1] */
/* assegna i restanti elementi di a */
for (p = a+3; p<&a[10]; p++) *p = *(a+2);

/* assegna a[0], a[1], a[2] */
*a = 1; *(p-9) = 2; *(a+2) = 3;
/* assegna i restanti elementi di a */
for (p = &a[3]; p<a+10; p++) *p = *(a+2);
```

Esempi CIAPE: dereferenziazione e aritmetica puntatori

```
/* CIAPE si applica anche
   nell'inizializzatore */
int a[10], *p = a+3, sum2 = 0, sum3, i;

/* altri 2 modi per fare la somma
   degli elementi di a */
p = a;
while (p < a+10) sum2 += *p++;

for (sum3 = 0, i = 0; i < 10; i++)
    sum3 += *(a+i);
```

Esempi CIAPE: operatori con side-effects

- Come per tutte le conversioni di tipo, CIAPE modifica il valore dell'espressione cui è applicato e il valore da essa prodotto viene utilizzato nel proseguo della valutazione dell'espressione entro cui si applica CIAPE
- Ma quando si applica CIAPE ad una variabile, il contenuto della variabile non viene modificato
- Ad esempio in `p = a+2` (dove `a` è un array e `p` è un puntatore dello stesso tipo degli elementi dell'array) l'espressione a viene convertita in puntatore, ma solo ai fini del calcolo del valore da assegnare a `p`: la variabile a rimane di tipo array e il suo contenuto non viene modificato
- Poiché il risultato della conversione è un valore, e non un oggetto, esso non può essere modificato dai side-effects di un operatore

Esempi CIAPE: operatori con side-effects

- Pertanto utilizzare un'espressione di tipo array come operando sinistro di = o come operando di ++ o di -- è constraint violation, ovvero una violazione delle regole del C Standard, segnalata in fase di traduzione

```
int a[10], *p;

p = a + 2; /* corretto */
a = p + 2; /* constraint violation */
a++;       /* constraint violation */
p = ++a + 2;
           /* constraint violation */
```

Esempi CIAPE: `sizeof`

- Se un espressione di tipo array è operando di `sizeof` non viene convertita
- Quindi `sizeof` si applica ad un array (non ad un puntatore al primo elemento dell'array): pertanto viene calcolata la dimensione dell'array (espressa in byte)

```
/* dichiarazione di un array di float
   e di un puntatore a float */
float a[5], *pf;

pf = a;

/* Stampa 2 numeri diversi */
printf("%d␣%d", (int) sizeof a,
                          (int) sizeof pf);
```

Esempi CIAPE: &

- Se un espressione di tipo array è operando di & non viene convertita
- Questo vuol dire che se E è un'espressione di tipo *array-di-T*, l'espressione &E ha tipo *puntatore ad array-di-T* (se invece E venisse convertita al tipo puntatore a T, &E avrebbe tipo puntatore a puntatore a T)
- Ad esempio, se a è un array di K elementi che hanno tipo T, l'espressione &a è un puntatore all'intero array a e non un puntatore al primo elemento di a

Esempi CIAPE: &

- Si noti che se a è un array di K elementi che hanno tipo T, le espressioni a e &a sono diverse tra loro: in particolare a ha tipo *array di K elementi che hanno tipo T* e viene convertita da CIAPE al tipo *puntatore a T*, mentre &a ha tipo *puntatore ad array di K elementi che hanno tipo T*
- Inoltre non viene effettuata nessuna conversione implicita tra tali tipi: se si tenta di assegnare un valore di uno di tali tipi ad una variabile dell'altro, si viola un constraint

Esempi CIAPE: &

```
/* dic. array di float e puntatore a float */
float a[5], *pf;
/* dic. puntatore ad array di 5 float */
float (*paf)[5];

/* Le seguenti istruzioni mostrano la differenza
   tra le espressioni a e &a */
pf = a; /* corretto: a viene convertito
           al tipo ( float * ) */
pf = &a; /* constraint violato: il tipo
            di &a non è ( float * ) */
paf = a; /* constraint violato: il tipo a cui
            viene convertito a è ( float * ) ma
            il tipo di paf è ( float (*)[5] ) */
paf = &a; /* corretto */
```

Esempi CIAPE: string literal

- Uno string literal usato per inizializzare un array non viene convertito
- Questo caso viene analizzato nel cap. 13 di **[Ki]**

Esempi CIAPE: indicizzazione

- Come noto, per accedere all'elemento di indice `i` dell'array `a` si usa l'espressione `a[i]`
- `a[i]` contiene come sotto-espressione `a`, che essendo un espressione di tipo array, viene convertita in puntatore
- Ma allora come è possibile accedere ad un elemento dell'array `a`, se, all'interno dell'espressione `a[i]`, `a` non è più un array?
- **Verità sconvolgente:** C Standard non definisce `[ ]` come operatore che si applica ad array

Esempi CIAPE: indicizzazione

- L'operatore `[ ]` è definito come operatore che ammette due operandi, uno di tipo *puntatore a T* e il secondo di tipo intero, e che ha un risultato di tipo *T*
- Date un'espressione `p` di tipo puntatore a T e un'espressione `i` di tipo intero, il valore dell'espressione `p[i]` è, per definizione, lo stesso dell'espressione `*(p+i)`
- Ne segue che l'espressione `p[i]` è ben definita solo se esiste un array `a` tale che sono vere entrambe le seguenti condizioni:
  1. `p` è un puntatore ad un elemento di `a` oppure è il POPLEL di `a`
  2. `p+i` è un puntatore ad un elemento di `a`

Esempi CIAPE: indicizzazione

- Dunque l'operatore `[ ]` può essere applicato anche a puntatori
- Si ricordi che è UB dereferenziare il POPLEL di un array, quindi se `p+i` è un POPLEL, allora `p[i]` causa un UB
- Si noti che se `p` punta un elemento di un array con indice maggiore di 0, l'espressione `p[i]` è definita anche per determinati valori negativi di `i`

```
unsigned long a[8] = { 1, 2, 3, 4, 5}, *p;
p = a + 2; /* p punta a[2] */
a[6] = p[2]; /* corretto: copia a[4] in a[6] */
p[1] += p[-1]; /* corretto: somma a[1] a a[3] */
p[-2] = 0; /* corretto: scrive 0 in a[0] */
p[-3] = 0; /* UB: scrive fuori dell'array */
p[5] = -1; /* corretto: scrive -1 in a[7] */
p[6] = 1; /* UB: scrive 1 nell'elemento OPLE */
```

Esempi CIAPE: chiamata di funzione

- Si consideri una funzione `f` che prende un parametro di tipo array; la definizione di `f` potrebbe essere, ad esempio: `int f(int par[5], int n);`
- Per chiamare la funzione con argomenti l'array `a` e il numero 5, si usa l'espressione `f(a,5)`
- `f(a,5)` contiene come sotto-espressione `a`, che essendo un espressione di tipo array, viene convertita in puntatore
- Ne segue che l'argomento passato ad `f` è un puntatore, non un array

Esempi CIAPE: chiamata di funzione

- Come mai il type checking non rileva una incompatibilità tra il tipo dell'argomento (puntatore) e quello del parametro (array)?
- Si potrebbe pensare che la regola CIAPE si applichi anche al parametro, ma non è così in quanto la regola CIAPE si applica alle espressioni, ma il costrutto `int par[5]` presente all'interno della definizione di `f` non è un espressione, è una dichiarazione
- C'è invece una regola simile, ma con alcune differenze, che si applica alle dichiarazioni dei parametri di funzione, chiamata *Conversione Implicita Array-Puntatore in Parametro (CIAPP)*
- La regola CIAPP verrà presentata in dettaglio in occasione di un futuro approfondimento sulle funzioni