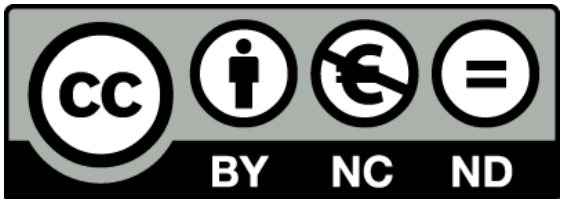
Laboratorio di Programmazione di Sistema

Programmazione Procedurale 4

Luca Forlizzi, Ph.D.

Versione 20.1



Parametri e Risultati di Tipo Puntatore o Aggregato

- Nelle precedenti presentazioni sono state presentati i concetti fondamentali relativi al passaggio di parametri e alla restituzione dei risultati, limitandosi a considerare parametri e risultati che hanno un tipo base
- I concetti fondamentali restano gli stessi anche per parametri e risultati che hanno un tipo diverso; cambiano però alcuni dettagli
- In questa presentazione si discutono in dettaglio le regole di C Standard relative a parametri di funzione e risultati di funzione che hanno uno dei seguenti tipi
  - Tipo struttura o unione
  - Tipo puntatore
  - Tipo array

Parametri e Risultati di un Tipo Struttura o Unione

- Variabili di un tipo struttura o unione possono essere parametri e risultati di funzione
- Presentiamo alcuni esempi, che utilizzano la seguente definizione di tipo

Definizione tipo MyS

```
#define N 5
struct MyS {
  double m1;
  int m2[ N ];
};
```

Parametri e Risultati di un Tipo Struttura o Unione

- Code1 mostra una funzione che ha un parametro di un tipo struttura

Code1

```
double average_MyS ( struct MyS s ) {
  return ( s.m1 + s.m2[ 0 ] ) / 2;
}
```

Parametri e Risultati di un Tipo Struttura o Unione

- Code2 mostra una funzione che ha il risultato di un tipo struttura

Code2

```
struct MyS build_MyS( double d ) {
  int i;
  struct MyS r;
  r.m1 = d;
  for ( i = 0 ; i < N ; i++ ) r.m2[ i ] = 0;
  return r;
}
```

Parametri e Risultati di un Tipo Struttura o Unione

- Code3 mostra una funzione che ha un parametro e il risultato di un tipo struttura

Code3

```
struct MyS set_m2_growing_MyS( int x, struct MyS s ) {
  for ( int i = 0 ; i < N ; i++, x++ )
    s.m2[ i ] = x;
  return s;
}
```

Parametri e Risultati di un Tipo Struttura o Unione

- Come per i tipi base, il passaggio di un parametro di un tipo struttura o di un tipo unione avviene per valore
- Ovvero, passare una variabile di tipo struttura o unione come parametro, comporta l'eseguire una copia dell'intera variabile
- Allo stesso modo, la restituzione di un risultato di tipo struttura o unione comporta l'eseguire una copia del risultato dell'espressione che segue l'istruzione `return`
- Le operazioni di copia necessarie per il passaggio di parametri per valore e per la restituzione di risultati, richiedono una certa quantità di tempo, che cresce con la dimensione dei valori copiati
- Dunque al crescere della dimensione di un tipo struttura o di un tipo unione, diminuisce l'efficienza di una chiamata

Parametri e Risultati di un Tipo Struttura o Unione

- Un'altra conseguenza rilevante è il fatto che le modifiche che una funzione opera su un parametro non hanno effetto sul corrispondente argomento, come in Code4

Code4

```
#include <stdio.h>
void set_m1_MyS( double d, struct MyS s ) {
  s.m1 = d;
}

int main( void ) {
  struct MyS dato = { 2.5, { 1, 2, 3, 1 } };
  set_m1_MyS( -4.8, dato );
  printf( "%f\n" , dato.m1 );
  /* stampa 2.5 ... è quello che
     il programmatore voleva? */
}
```

Parametri e Risultati di un Tipo Struttura o Unione

- Si noti che in Java, invece, il passaggio di parametri di tipo oggetto avviene per riferimento
- In C, è possibile ottenere una semantica simile al passaggio per riferimento, utilizzando dei puntatori alle variabili di tipo struttura o unione
- Utilizzando dei puntatori a variabile è possibile ottenere
  1. una maggiore efficienza, in quanto, nella maggior parte delle *ISA*, un puntatore ha una dimensione in byte paragonabile a quella di un singolo membro di una struttura
  2. una semantica analoga a quella di Java, in quanto i puntatori in C si comportano in modo simile ai riferimenti in Java

Parametri e Risultati di un Tipo Puntatore

- Le funzioni possono avere parametri e risultato che hanno un tipo puntatore
- Attraverso un parametro di tipo puntatore, una funzione può modificare variabili esterne o locali ad altre funzioni
- Un esempio già incontrato in precedenza è la funzione `scanf`: gli argomenti successivi al primo sono puntatori a variabili, il cui contenuto viene modificato da `scanf`, come in Code5

Code5

```
#include <stdio.h>
int main( void ) {
  int x;
  scanf( "%d", &x );
}
```

Parametri e Risultati di un Tipo Puntatore

- Nella presentazione *Programmazione Procedurale 2* viene mostrata una funzione `conv_t` che tenta, in modo errato, di comunicare valori ai suoi argomenti; con parametri di tipo puntatore, si ottiene una versione corretta di `conv_t`

Code6

```
#include <stdio.h>
void conv_t( int tempo, int *ore, int *min, int *sec ) {
  *ore = tempo / 3600;
  *min = ( tempo % 3600 ) / 60;
  *sec = ( tempo % 3600 ) % 60;
}
int main( void ) {
  int h = 0, m = 0, s = 0;
  conv_t( 1354, &h, &m, &s );
  printf( "%d␣%d␣%d", h, m, s ); /* stampa 0 22 34 */
  return 0;
}
```

Parametri e Risultati di un Tipo Puntatore

- Una funzione può restituire come risultato un puntatore ad una variabile esterna o un parametro di tipo puntatore
- Code7 mostra una funzione che analizza 3 variabili e restituisce un puntatore a quella, tra le tre, che ha valore compreso tra quello delle altre

Code7

```
int e, a[ 4 ];
int* middle( int *x ) {
  if ( e <= *x && e <= a[ 3 ] )
    if ( *x <= a[ 3 ] ) return x; else return &a[ 3 ];
  if ( *x <= e && *x <= a[ 3 ] )
    if ( e <= a[ 3 ] ) return &e; else return &a[ 3 ];
  if ( e <= *x ) return &e; else return x;
}
```

Parametri e Risultati di un Tipo Puntatore

- Un errore tipico e insidioso, è restituire come risultato di una funzione, un puntatore ad una variabile automatica: ciò causa un undefined behavior in quanto la variabile automatica viene de-allocata al termine dell'esecuzione della funzione

Code8

```
double* add_pi( double x ) {
  double pi = 3.1415;
  pi += x;
  return &pi;
}

int main( void ) {
 double *p;
 p = add_pi( 1.2 ); /* undefined behavior */
 return 0;
}
```

Parametri e Risultati di un Tipo Puntatore

- Nella realizzazione di funzioni che manipolano strutture dati, è spesso preferibile usare, come parametri e risultati, puntatori a strutture invece che strutture
- In questo modo si ottiene maggiore efficienza, in quanto si evitano le operazione di copia di parametri e risultati, che possono portare rallentamenti sensibili nel caso di tipi struttura che hanno dimensione elevata
- Inoltre, attraverso puntatori a struttura, si ottiene una semantica simile al passaggio di parametri per riferimento usato, ad esempio, da Java per i tipi oggetto

Parametri e Risultati di un Tipo Puntatore

- Mostriamo quindi come modificare i precedenti esempi, in modo da ottenere funzioni che usano puntatori a strutture per parametri e risultati
- Il tipo `struct MyS` è definito esattamente come in precedenza
- Code9 mostra una versione modificata della funzione presente in Code1

Code9

```
double average_MyS_2 ( struct MyS *s ) {
  return ( s -> m1 + s -> m2[ 0 ] ) / 2;
 }
```

Parametri e Risultati di un Tipo Puntatore

- Code10 mostra una versione modificata della funzione presente in Code2
- Si noti che la funzione restituisce un puntatore ad una variabile locale statica: essendo statica non viene de-allocata al termine dell'esecuzione della funzione, quindi Code10 è corretto

Code10

```
struct MyS *build_MyS_2( double d ) {
  int i;
  static struct MyS r; /* static local variable */
  r.m1 = d;
  for ( i = 0 ; i < N ; i++ ) r.m2[ i ] = 0;
  return &r;
}
```

Parametri e Risultati di un Tipo Puntatore

- Code11 mostra una versione modificata della funzione presente in Code3

Code11

```
struct MyS *set_m2_growing_MyS( int x, struct MyS *s ) {
  for ( int i = 0 ; i < N ; i++, x++ )
    s -> m2[ i ] = x;
  return s;
}
```

Parametri e Risultati di un Tipo Puntatore

- Code12 mostra una versione modificata della funzione presente in Code4

Code12

```
#include <stdio.h>
void set_m1_MyS_2( double d, struct MyS *s ) {
  s -> m1 = d;
}

int main( void ) {
 struct MyS dato = { 2.5, { 1, 2, 3, 1 } };
 set_m1_MyS_2( -4.8, dato );
 printf( "%f\n" , dato.m1 );
 /* stampa -4.8 */
}
```

Parametri e Risultati di un Tipo Array

- In C Standard, non è possibile definire funzioni che abbiano un tipo array come tipo del risultato: se si tenta di farlo si violano le regole sintattiche del linguaggio
- È invece possibile dichiarare parametri che abbiano un tipo array, ma tali parametri si comportano in modo difforme da parametri che hanno altri tipi, presentando numerose particolarità
- Il motivo è che, in realtà, non è possibile definire funzioni che abbiano parametri di tipo array: le dichiarazioni di parametri di tipo array non sono quello che sembrano
- Nella presentazione *Array e Puntatori*, abbiamo già notato qualcosa di strano in relazione alla chiamata di una funzione con un parametro di tipo array

Parametri e Risultati di un Tipo Array

- Code13 mostra un frammento di programma che chiama una funzione, dichiarata con un parametro di tipo array

Code13

```
void f13( int par_array[5], int n );
int array1[ 5 ] = { 1, 2, 3, 4, 5 };
f13( array1, 5 );
```

- Il primo argomento della chiamata di funzione è `array1`, che, essendo un espressione di tipo array, viene convertita dalla regola CIAPE in puntatore al primo elemento di `array1`
- Quindi tale argomento è un puntatore, non un array, ma il type checking non rileva incompatibilità

Conversione Implicita Array-Puntatore in Parametro

- Il motivo è che C Standard ha una regola, chiamata *Conversione Implicita Array-Puntatore in Parametro (CIAPP)*, in base alla quale ogni dichiarazione di parametro di tipo array viene trasformata automaticamente dal traduttore in una dichiarazione di tipo puntatore
- Una dichiarazione di un parametro che ha tipo *array-di-T* viene convertita in dichiarazione di tipo *puntatore-a-T*
- Si sottolinea che la conversione ha effetto sulle dichiarazioni di parametro di funzione ma non sulle dichiarazioni di variabile
- Quindi, i parametri che sono dichiarati di tipo array, in realtà non sono degli array, ma sono dei puntatori
- In altre parole, nella dichiarazione di un parametro, la sintassi `par[ ]` è semplicemente un modo alternativo di scrivere `*par`

Conversione Implicita Array-Puntatore in Parametro

- Le regole CIAPE e CIAPP sono simili ma diverse, in quanto la prima si applica ad espressioni e la seconda a dichiarazioni
- Operano in modo coordinato per fare in modo che, ad una funzione che ha un parametro di tipo array, si possa passare un'espressione di tipo array
- Infatti sia il parametro della funzione, sia l'espressione passata come argomento, vengono convertiti allo stesso tipo puntatore, come in Code14

Code14

```
int f14( int par[ 5 ], int n ); /* par ha tipo (int *) */
int main( void ){
  int a[ 5 ];
  f14( a, 5 ); /* corretto: a è convertito a (int *) */
  return 0;
}
```

Conseguenze di CIAPP

- CIAPP ha una serie di conseguenze relative a parametri dichiarati con tipo array monodimensionale, che è opportuno esaminare
  1. Sintassi alternative per le dichiarazioni
  2. Flessibilità degli argomenti ammissibili
  3. Apparente passaggio per riferimento
  4. Relazione con operatori con side effects
  5. Relazione con operatore `sizeof`
  6. Tempo impiegato per passaggio di parametro
- Per i parametri di tipo array multidimensionale la situazione è diversa, come spiegheremo in seguito

CIAPP: Sintassi Alternative per le Dichiarazioni

- CIAPP sancisce che un parametro dichiarato con tipo array è in realtà un parametro di tipo puntatore
- Quindi la dichiarazione di un parametro di tipo array è una sintassi alternativa per dichiarare un parametro di tipo puntatore
- Nella conversione di un parametro di tipo array a tipo puntatore, perde di significato la lunghezza del parametro array
- Infatti due parametri di tipo array con elementi dello stesso tipo ma lunghezze diverse, vengono convertiti allo stesso tipo puntatore

CIAPP: Sintassi Alternative per le Dichiarazioni

- Di conseguenza il traduttore, quando effettua il type checking, ignora del tutto le lunghezze dei parametri di funzione dichiarati come array, limitandosi a controllare che il tipo degli elementi dell'array corrisponda con il tipo degli oggetti puntati dall'argomento
- Quindi indicare esplicitamente la lunghezza per un parametro di tipo array non ha alcuna utilità per il traduttore, sebbene possa migliorare la leggibilità del programma
- Per questo motivo, C Standard consente anche di dichiarare un parametro di tipo array omettendo la lunghezza
- Ciò contrasta con il fatto che quando si dichiara una variabile di tipo array è invece obbligatorio specificare la lunghezza (la lunghezza può essere omessa se è presente un inizializzatore, ma solo in quanto essa viene calcolata automaticamente proprio usando l'inizializzatore)

CIAPP: Sintassi Alternative per le Dichiarazioni

- Code15 riporta una serie di dichiarazioni in forma di prototipo di una funzione, tutte equivalenti tra loro

Code15

```
void f15( int n, int *p );
void f15( int n, int p[ ] );
void f15( int n, int p[ 5 ] );
void f15( int n, int p[ 25 ] );
void f15( int n, int p[ * ] );
void f15( int n, int p[ n ] );
```

CIAPP: Sintassi Alternative per le Dichiarazioni

- Il fatto che la lunghezza di un parametro di tipo array non abbia significato semantico
  - ha un importante vantaggio: aumenta la flessibilità degli argomenti ammissibili per un parametro di tipo array, come discuteremo nel seguito
  - ha uno svantaggio: la funzione non ha modo di conoscere, attraverso il parametro, qual'è la lunghezza di un array passato come argomento (si potrebbe pensare di utilizzare l'operatore `sizeof` a tale scopo, ma invece non è possibile farlo neppure mediante tale operatore, come mostreremo nel seguito)

CIAPP: Sintassi Alternative per le Dichiarazioni

- Poiché un parametro di tipo array non fornisce informazioni sulla lunghezza di un corrispondente argomento di tipo array, se tale informazione è necessaria ad una funzione, la si deve comunicare in altro modo: ad esempio mediante un altro parametro, dedicato a tale scopo, come in Code16

Code16

```
int prod_array( int n, int a[ ]  ) {
  for ( int i = 0, p = 1 ; i < n ; i++ ) p *= a[ i ];
  return p;
}
int main( void ) {
  int r1, r2, a1[ 10 ], a2[ 5 ];
  r1 = prod_array( 5, a2 );
  r2 = prod_array( 10, a1 );
}
```

CIAPP: Flessibilità degli Argomenti Ammissibili

- Il fatto che la lunghezza di un parametro di tipo array non abbia significato semantico, rende possibile passare array di lunghezze diverse (ma elementi dello stesso tipo) ad uno stesso parametro di funzione, come già mostrato in Code16
- Inoltre, poiché l'argomento passato ad un parametro di tipo array è un puntatore, esso può anche essere un puntatore ad un elemento di un array diverso dal primo
- Ciò rende molto flessibili le funzioni in C, in quanto permette loro di accettare come parametri anche "fette" di array

CIAPP: Flessibilità degli Argomenti Ammissibili

- Code17 mostra possibili chiamate di una funzione che ha un parametro di tipo array e un ulteriore parametro attraverso cui passare la lunghezza dell'array argomento

Code17

```
void f17( int n, int par[ 5 ] );
int main( void ) {
  int a[ 10 ], b[ 5 ];

  f17( 10, a ); /* passa tutti gli elementi di a */
  f17( 5, a ); /* passa i primi 5 elementi di a */
  f17( 5, b ); /* passa tutti gli elementi di b */
  f17( 5, a + 3 ); /* passa da a[3] fino ad a[7] */
}
```

CIAPP: Flessibilità degli Argomenti Ammissibili

- Il prezzo da pagare per tanta flessibilità, naturalmente è la mancanza di controllo da parte del traduttore sulla "validità" della lunghezza dell'argomento array comunicata attraverso un altro parametro
- Se, ad una funzione che ha un parametro di tipo array e un ulteriore parametro attraverso cui comunicare la lunghezza dell'array argomento, viene comunicata una lunghezza dell'argomento maggiore di quella effettiva, anche una funzione scritta correttamente potrebbe effettuare accessi out-of-bounds all'argomento array

CIAPP: Flessibilità degli Argomenti Ammissibili

- Code18 mostra un esempio di chiamata di funzione con parametro array che viola un constraint, e un esempio di chiamata che genera un undefined behavior

Code18

```
int prod_array( int n, int a[ ]  ) {
  for ( int i = 0, p = 1 ; i < n ; i++ ) p *= a[ i ];
  return p;
}
int main( void ) {
  int r, a1[ 10 ];
  r = prod_array( 5, a1[ ] ); /* constraint violation */
  r = prod_array( 5, a1 ); /* corretta */
  r = prod_array( 10, a1 ); /* corretta */
  r = prod_array( 11, a1 ); /* undefined behavior */
  r = prod_array( 6, a1 + 2 ); /* corretta */
  r = prod_array( 6, a1 + 5 ); /* undefined behavior */
}
```

CIAPP: Flessibilità degli Argomenti Ammissibili

- In C99 e versioni successive, è possibile dichiarare parametri di tipo VLA
- La lunghezza del parametro VLA è un'espressione che contiene una o più variabili
- Tra le variabili, possono esserci altri parametri della funzione, purché precedano il VLA nella lista dei parametri
- Code19 è una modifica di Code16, che usa questa possibilità

Code19

```
int prod_array( int n, int a[ n ]  ) {
  for ( int i = 0, p = 1 ; i < n ; i++ ) p *= a[ i ];
  return p;
}
int main( void ) {
  int a1[ 10 ], r = prod_array( 10, a1 );
}
```

CIAPP: Flessibilità degli Argomenti Ammissibili

- Nelle dichiarazioni pure, la lunghezza di un parametro VLA può essere indicata o con un'espressione che contiene i nomi di parametri che precedono il VLA, come nelle definizioni, oppure con il simbolo $*$
- Code20 mostra diversi modi, tutti equivalenti, di scrivere una dichiarazione pura della funzione mostrata in Code19

Code20

```
int prod_array( int n, int a[ n ]  );
int prod_array( int n, int a[ * ]  );
int prod_array( int , int [ * ]  );
```

CIAPP: Flessibilità degli Argomenti Ammissibili

- Utilizzare un parametro VLA che ha come lunghezza un altro parametro, permette di rendere esplicito nel codice sorgente il legame tra il VLA e il parametro che viene usato per passare la lunghezza del VLA
- Ma l'unica utilità di esprimere questo legame è quella di rendere il codice più leggibile per un umano
- Infatti questa informazione non ha valore semantico, in particolare non viene controllato se il valore del parametro corrisponde davvero alla lunghezza del VLA
- Come nel caso dei parametri array con lunghezza statica, la lunghezza del VLA potrebbe essere diversa dal valore del parametro che la dovrebbe comunicare alla funzione
- Nel caso di parametri array multidimensionali, i vantaggi dei VLA sono più significativi, come mostriamo nel seguito

CIAPP: Apparente Passaggio per Riferimento

- Una funzione può modificare il contenuto di un array passato come parametro
- Diversamente da ciò che accade con tutti i parametri di tipo diverso da array, la modifica ha effetto sul corrispondente argomento, come in Code21

Code21

```
void f21( n, int a[ ] ) {
  a[ 0 ] = a[ n - 1 ] = 0;
}
int main( void ) {
  int x, y, a1[  ] = { 1, 2, 3, 2, 1 };
  x = a1[ 0 ] + a1[ 4 ];  /* x vale 2 */
  f21( 5, a1 );
  y = a1[ 0 ] + a1[ 4 ];  /* y vale 0 */
}
```

CIAPP: Apparente Passaggio per Riferimento

- Il fatto che le modifiche agli elementi dell'array fatte nel corpo della funzione abbiano effetto sull'argomento, sembra apparentemente contraddire la regola generale di C Standard, secondo la quale le variabili sono passate per valore
- In realtà, tecnicamente non c'è contraddizione, in quanto, per CIAPE e CIAPP, l'argomento che viene passato non è un array, ma è un puntatore all'array
- Tale puntatore viene passato per valore e può essere usato per modificare l'array puntato, che è una variabile non locale alla funzione
- Quindi, tali modifiche hanno effetto all'esterno della funzione

CIAPP: Relazione con Operatori che Hanno Side Effects

- Un array passato come argomento ad una funzione, viene convertito in un puntatore
- Tale puntatore è passato per valore e memorizzato nel parametro
- Le modifiche fatte al parametro puntatore non hanno effetto all'esterno della funzione (diversamente dalle modifiche fatte all'array puntato dal parametro)

CIAPP: Relazione con Operatori che Hanno Side Effects

- Pertanto è possibile applicare ad un parametro array operatori con side effects, come in Code22
- Tali operatori modificano il parametro ma le modifiche non hanno effetto all'esterno del corpo della funzione
- Si ricorda che, invece, non è possibile applicare operatori con side effects a variabili di tipo array

Code22

```
void f22( int n, int par_arr[ 5 ] ) {
  int x, var_arr[ 5 ];
  /* par_arr è un parametro, var_arr è una variabile */

  x = *par_arr; /* corretto */
  par_arr = var_arr + 1; /* corretto */
  var_arr = par_arr + 1; /* constraint violation */
  par_arr++; /* corretto */
  var_arr++; /* constraint violation */
}
```

CIAPP: Relazione con Operatore `sizeof`

- Come noto, l'operatore `sizeof` applicato ad una variabile, restituisce la quantità di byte occupata da essa
- È possibile calcolare la lunghezza di una variabile `v` di tipo array, con l'espressione `sizeof v / sizeof v[ 0 ]`
- Si potrebbe quindi pensare di utilizzare tale espressione su un parametro di tipo array, per conoscere la lunghezza del corrispondente argomento, evitando così la necessità di dotare una funzione di un parametro ulteriore attraverso cui comunicare la lunghezza di una argomento array
- Purtroppo l'espressione non fornirebbe la lunghezza corretta

CIAPP: Relazione con Operatore `sizeof`

- Il motivo è che, poiché il parametro è in realtà un puntatore, l'operatore `sizeof` applicato ad un parametro calcola la quantità di byte occupata da un puntatore, come in Code23

Code23

```
void f23( int n, char par_arr[ 5 ] ) {
  char var_arr[ 5 ];

  /* la chiamata seguente stampa 5 */
  printf( "%zu", sizeof var_arr / sizeof var_arr[ 0 ] );

  /* la chiamata seguente stampa il numero di byte
     usati per memorizzare un puntatore */
  printf( "%zu", sizeof par_arr / sizeof par_arr[ 0 ] );
}
```

CIAPP: Tempo Impiegato per Passaggio di Parametro

- Se invece che un puntatore, venisse passato davvero un array per valore, sarebbe necessario eseguire una copia dell'argomento nel parametro
- Tale copia richiederebbe un certo tempo, non trascurabile nel caso di array molto lunghi
- Inoltre tale copia richiederebbe un tempo diverso in base alla dimensione (ovvero alla lunghezza) dell'argomento: di conseguenza, anche se il corpo della funzione eseguisse operazioni non dipendenti dalla lunghezza dell'argomento, chiamate della funzione con argomenti di lunghezza diversa richiederebbero tempi diversi

CIAPP: Tempo Impiegato per Passaggio di Parametro

- La regola CIAPP rende il C più efficiente e uniforme
- Code24 mostra un esempio in cui ad una funzione vengono passati array di lunghezze molto diverse

Code24

```
int f24( int n, int par[ ] ) {
  /* il calcolo eseguito nel corpo non dipende
     dalla lunghezza del parametro array */
  return par[ 0 ] + par[ n - 1 ];
}
int main( void ){
  int x, a[ 10 ], b[ 100000 ];

  /* le due chiamate richiedono circa lo stesso tempo */
  x = f24( 10, a );
  x = f24( 100000, b );
}
```

Parametri di Tipo Array Multidimensionale

- Nel caso di parametri array multidimensionali, vi è una sostanziale differenza semantica tra la prima dimensione e le successive
- Il motivo è che
  - tecnicamente, in C Standard un array con k dimensioni è un array i cui elementi hanno tipo array con $k - 1$ dimensioni
  - la regola CIAPP non si applica ricorsivamente agli elementi di un parametro array

Parametri di Tipo Array Multidimensionale

- Ad esempio, quindi, un parametro dichiarato come array bidimensionale di elementi di tipo `int`, viene convertito da CIAPP in puntatore ad array di elementi di tipo `int` e non ad un puntatore ad un puntatore ad un `int`
- Di conseguenza, le lunghezze delle dimensioni successiva alla prima di un parametro array, sono prese in considerazione dal traduttore al momento del type checking
  - Pertanto, nelle dichiarazioni di funzione solo la lunghezza della prima dimensione di un parametro array può non essere indicata
  - Le altre devono essere indicate

Parametri di Tipo Array Multidimensionale

- Code25 mostra esempi di dichiarazioni di funzioni con parametri di tipo array multidimensionale; alcune dichiarazioni sono corrette, altre violano constraint

Code25

```
int f25_1( int a[ 10 ] [ 10 ]  ); /* corretto */
int f25_2( int a[ ] [ 10 ]  ); /* corretto */
int f25_3( int a[ 10 ] [ ]  ); /* constraint violation */
int f25_4( int a[ ] [ ]  ); /* constraint violation */
```

Parametri di Tipo Array Multidimensionale

- Nelle chiamate di funzione, le lunghezze delle dimensioni successiva alla prima di un argomento di tipo array multidimensionale, devono essere uguali a quelle del corrispondente parametro, altrimenti si hanno constraint violation, come in Code26

Code26

```
void f26( int a[ 6 ] [ 6 ]  );
int a1[ 6 ][ 6 ], a2[ 4 ][ 6 ], a3[ 9 ][ 6 ];
int a4[ 6 ][ 4 ], a5[ 6 ][ 8 ];
f26( a1 ); /* corretto */
f26( a2 ); /* corretto */
f26( a3 ); /* corretto */
f26( a4 ); /* constraint violation */
f26( a5 ); /* constraint violation */
```

Parametri di Tipo Array Multidimensionale

- Dunque l'argomento corrispondente ad un parametro array multidimensionale, deve essere un array che ha le lunghezze successive alla prima esattamente uguali alle corrispondenti lunghezze del parametro
- Ciò limita la flessibilità delle funzioni, in quanto rende impossibile scrivere una funzione che accetta come argomenti array che hanno lunghezze diverse per le dimensioni oltre la prima

Parametri di Tipo Array Multidimensionale

- La funzione mostrata in Code27 accetta come argomenti array bidimensionali che hanno un qualunque numero di righe, ma esattamente 4 colonne
- Questo è stato a lungo un limite del C, rispetto al Fortran o ad altri linguaggi, rilevante sopratutto per le applicazioni matematiche

Code27

```
/* C89 */
int sum_mat_C89( int n, int a[ ] [ 4 ]  )
{
  int i, j, r = 0;
  for ( i = 0 ; i < n ; i++ )
    for ( j = 0 ; j < 4 ; j++ )
      r += a[ i ][ j ];
  return r;
}
```

Parametri di Tipo Array Multidimensionale

- Tale limite è stato superato dal C99, in quanto, in tale versione e nelle successive, è possibile scrivere funzioni che accettano come argomenti array con qualunque lunghezza in tutte le dimensioni, grazie a parametri di tipo VLA multidimensionali, come in Code28

Code28

```
// C99 and later versions
int sum_mat_C99( int n, int m, int a[ n ] [ m ]  )
{
  int r = 0;
  for ( int i = 0 ; i < n ; i++ )
    for ( int j = 0 ; j < m ; j++ )
      r += a[ i ][ j ];
  return r;
}
```

Materiale Didattico

- Per assimilare e approfondire i contenuti di questa presentazione, si consiglia di studiare anche
  - Sezioni 9.3, 11.4, 11.5, 12.3, 12.4, 12.5, 16.2 di **[Ki]**