Array, puntatori e stringhe

Francesco Isgrò

Array, puntatori e stringhe

- Gli array sono un tipo di dato che usa variabili indicizzate
- Consentono di rappresentare una grande quantità di valori omogenei
- In C array e puntatori sono concetti correlati
- Il *nome* di un array è trattato come un puntatore costante
- I puntatori possono essere indicizzati

- Il C include un’aritmetica dei puntatori
- In C i puntatori sono usati per gestire le chiamate per riferimeto dei parametri delle funzioni
- Le stringhe sono array mono-dimensionali di caratteri

Array mono-dimensionali

- I programmi usano spesso dati omogenei
- Un programma che manipola gradi potrebbe avere

  ```
  int grade0, grade1, grad2;
  ```

- Se la quantità di variabili necessarie è grande, diventa scomodo dichiarare tutte variabili indipendenti

- Una variabile indicizzata è più comoda

  ```
  int grade[3];
  ```

- Un array di tre interi

  ```
  grade[0] = grade[1] = grade[2] = 0;
  ```

- Il primo indice degli array è sempre 0

- Dichiarazione di un array mono-dimensionale
  - tipo
  - identificatore
  - espressione costante intera fra parentesi quadre
- La dichiarazione

  ```
  int a[size];
  ```

  alloca spazio per a[0] . . . a[size-1]

- Buona norma di programmazione definire la dimensione del vettore come una costante simbolica

```
# define N 100
int main ( void )
{
    int a[ N ];
    .....
}
```

```
#include <stdio.h>

#define N 4

int main(void)
{
  int a[N], i;
  int sum = 0;
  ......

  for (i=0; i<N; i++)
    sum += a[i];

  printf(“Sum is %d \n”, sum);

  return 0;
}
```

Inizializzazione

- Gli array possono avere storage class automatic, external e static.
- Gli array non possono essere register
- In C tradizionale solo gli array external e statici potevano essere inizializzati esplicitamente
- In ANSI C anche gli array automatic possono essere inizializzati esplicitamente

- Esempio

  float v[5] = {0.0 , 1.0 , 2.0 , 3.0 , 4.0};

  - Inizializza il vettore con i valori

    v[0] = 0.0;
    v[1] = 1.0;
    v[2] = 2.0;
    v[3] = 3.0;
    v[4] = 4.0;

- L’inizializzazione

  int v[5] = {0};

  inizializza l’intero vettore a 0

- L’inizializzazione

  int v[5] = {1};

  assegna v[0] = 1 e pone a 0 i rimanenti

- Se un array di tipo external non è inizializzato esplicitamente, il sistema assegna 0 a tutti gli elementi
- Se un array di tipo static non è inizializzato esplicitamente, il sistema assegna 0 a tutti gli elementi
- Se un array di tipo automatic non è inizializzato esplicitamente, l’array non è necessariamente inizializzato a 0 dal sistema

```c
#include <stdio.h>

int v[5];

int main(void)
{
   static int a[5];
   int b[5];
   int i;

   for (i=0; i<5; i++)
     printf("%d ",a[i]);
   printf("\n");

   for (i=0; i<5; i++)
     printf("%d ",v[i]);
   printf("\n");

   for (i=0; i<5; i++)
     printf("%d ",b[i]);
   printf("\n");
}
```

```c
#include <stdio.h>

int v[5];

int main(void)
{
   static int a[5];
   int b[5];
   int i;

   for (i=0; i<5; i++)
     printf("%d ",a[i]);
   printf("\n");

   for (i=0; i<5; i++)
     printf("%d ",v[i]);
   printf("\n");

   for (i=0; i<5; i++)
     printf("%d ",b[i]);
   printf("\n");
}
```

Output

```
0 0 0 0 0
0 0 0 0 0
0 0 4195424 0 -758464656
```

- Se un array è dichiarato senza la size, ed è inizializzato, il sistema assegna implicitamente la size del numero dei valori di inizializzazione
- Esempio

```
int a[] = {1 , 2 , 3 , 4 , 5};
```

è equivalente a

```
int a[5] = {1 , 2 , 3 , 4 , 5};
```

- Per array di caratteri

  char s[] = " abc " ;

  è considerata dal compilatore come

  char s[] = { ’a ’ , ’b ’ , ’c ’ , ’ \0 ’ };

Indici

- Se a è un array, con una scrittura del tipo

  a[expr]

  dove expr è una espressione intera, si accede agli elementi di a
- Consideriamo la dichiarazione

  int a[N];

  dove N è una costante simbolica
- Si può accedere a un singolo elemento a[i] con

  i=0, ..., N-1

- Accedere a[i] con i al di fuori del range di validità è un comune errore di programmazione
- L’effetto di questo errore dipende dal sistema
- Se il programma non va in errore i risultati possono essere, chiaramente, erronei
- L’errore è dovuto al fatto che si accede a memoria che non è stata riservata per l’array.

Puntatori

- Una variabile è immagazzinata in un certo numero di byte a una particolare locazione di memoria, identificata da un indirizzo
- I puntatori sono usati per accedere la memoria e manipolare gli indirizzi
- Data una variabile v il suo indirizzo è indicato con &v
- Gli indirizzi sono valori che possono essere manipolati

- La dichiarazione
  - int *p;

  dichiara la variabile p come puntatore a int
- Il range di valori è lo zero e l'insieme di interi positivi che possono essere interpretati come indirizzi sul sistema

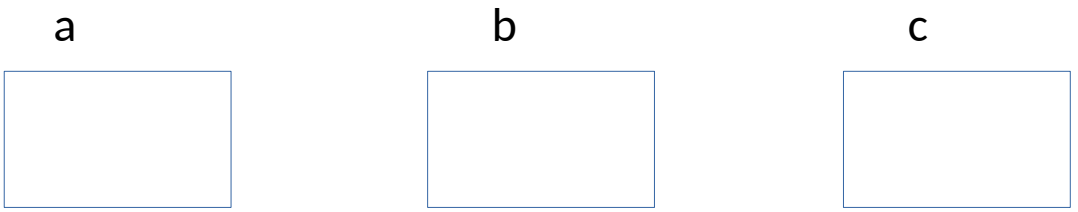
- Alcuni esempi di assegnamento

```
p = 0;
p = NULL;   //Equivalente a p = 0
p = &i;
p = ( int *) 1776; // indirizzo assoluto in memoria
```

- *p è il valore della variabile di cui p è l'indirizzo
- In pratica i e *&i sono equivalenti

```
int a, b, *p;

a = 1;
b = 2;
p = &a;

b = *p;
```

```
int a, b, *p;

a = 1;
b = 2;
p = &a;

b = *p;
```



```
int a, b, *p;

a = 1;
b = 2;
p = &a;

b = *p;
```

- L’istruzione

  ```
  b = *p;
  ```

  è equivalente a

  ```
  b = a;
  ```

```c
#include <stdio.h>

int main(void)
{
   int   i = 7, *p;

   p = &i;
   printf("%s%d\n%s%p\n",
      "   Value of i: %d", *p,
      "Location of i: %d",  p);
   return 0;
}
```

```
#include <stdio.h>

int main(void)
{
   int   i = 7, *p;

   p = &i;
   printf("%s%d\n%s%p\n",
      "   Value of i: %d", *p,
      "Location of i: %d",  p);
   return 0;
}
```

Output

```
   Value of i: 7
Location of i: 0x7fff717bfc0c
```

Dichiarazioni e inizializzazioni		
`int i = 3, j = 5, *p = &i, *q = &j, *r;` `double x;`		
Espressione	Espressione equivalente	Valore
`p == &i`		
`**&p`		
`*(r=&j) *= *p`		
`7**p / *q+7`		
`r = &x`		

Dichiarazioni e inizializzazioni		
`int i = 3, j = 5, *p = &i, *q = &j, *r;` `double x;`		
Espressione	Espressione equivalente	Valore
`p == &i`	`p == (&i)`	
`**&p`	`*(*(&p))`	
`*(r=&j) *= *p`	`(*(r=(&j))) *= *p`	
`7**p / *q+7`	`(((7*(*p))) / (*q))+7`	
`r = &x`	`r = &x`	

Dichiarazioni e inizializzazioni		
`int i = 3, j = 5, *p = &i, *q = &j, *r;` `double x;`		
Espressione	Espressione equivalente	Valore
`p == &i`	`p == (&i)`	`1`
`**&p`	`*(*(&p))`	`3`
`*(r=&j) *= *p`	`(*(r=(&j))) *= *p`	`15`
`7**p / *q+7`	`(((7*(*p))) / (*q))+7`	`11`
`r = &x`	`r = &x`	`illegale`

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *r;
  double x;

  r = &x;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *r;
  double x;

  r = &x;
}
```

Compilatore

```
prova.c:9:5: warning: assignment from incompatible pointer type
[enabled by default]
   r = &x;
```

- Non tutti i costrutti possono essere *puntati*
- Non si può puntare alle costanti

```
&3              // Illegale
```

- Non si può puntare a espressioni

```
&( k + 9)       // Illegale
```

- Non si può puntare a variabili register

```
register v ;
&v              // Illegale
```

Passaggio per indirizzo

- Una variabile passata come parametro di una funzione non cambia il proprio valore dopo l’uscita della funzione stessa
- Questo meccanismo si chiama passaggio per valore
- Altri linguaggi consentono il passaggio per riferimento: il valore della variabile passato come parametro può cambiare all’uscita della funzione
- Un uso appropriato dei puntatori può produrre un effetto simile al passaggio per riferimento

```
int f(int a, int b)
{
  a = a+1;
  b = b+1;
  return (a*b);
}

int main()
{
  int a = 1, b = 2, c;

  c = f(a,b);

  printf("a = %d\nb = %d\nc = %d \n", a, b, c);
}
```

```
int f(int a, int b)
{
  a = a+1;
  b = b+1;
  return (a*b);
}

int main()
{
  int a = 1, b = 2, c;

  c = f(a,b);

  printf("a = %d\nb = %d\nc = %d \n", a, b, c);
}
```

Output

```
a = 1
b = 2
c = 6
```

```
int f(int a, int b)
{
  a = a+1;
  b = b+1;
  return (a*b);
}

int main()
{
  int a = 1, b = 2, c;

  c = f(a,b);

  printf("a = %d\nb = %d\nc = %d \n", a, b, c);
}
```

Se invece volessimo Output?

```
a = 2
b = 3
c = 6
```

```
int f(int *a, int *b)
{
  a = a+1;
  b = b+1;
  return (a*b);
}

int main()
{
  int a = 1, b = 2, c;

  c = f(&a,&b);

  printf("a = %d\nb = %d\nc = %d \n", a, b, c);
}
```

```
int f(int *a, int *b)
{
  a = a+1;
  b = b+1;
  return (a*b);
}

int main()
{
  int a = 1, b = 2, c;

  c = f(&a,&b);

  printf("a = %d\nb = %d\nc = %d \n", a, b, c);
}
```

Output

```
a = 2
b = 3
c = 6
```

Swap di due variabili

- Lo swap di due variabili viene solitamente fatto usando una variabile buffer
- Esempio lo swap di due variabili intere

```
int a, b, tmp;
….
tmp = a;
a = b;
b = tmp;
```

- Scriviamo una funzione che faccia lo swap di due variabili di tipo intero, passate come parametri

```c
#include <stdio.h>

void   swap(int *, int *);

int main(void)
{
   int    i = 3, j = 5;

   swap(&i, &j);
   printf("%d  %d\n", i, j);       /* 5  3 is printed */
   return 0;
}
```

```c
#include <stdio.h>

void   swap(int *, int *);

int main(void)
{
   int    i = 3, j = 5;

   swap(&i, &j);
   printf("%d  %d\n", i, j);       /* 5  3 is printed */
   return 0;
}

void swap(int *p, int *q)
{
   int   tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Relazione fra array e puntatori

- Il nome di un array è un indirizzo, cioè il valore di un puntatore
- Un puntatore, come gli array, può essere indicizzato
- Un puntatore può prendere differenti indirizzi
- Il nome di un array è un indirizzo, o puntatore, che non può essere cambiato

- Supponiamo che a sia un array e i un variabile int
- L’istruzione

  a[i] // i - esimo elemento del vettore
- è equivalente a

  *(a + i) // contenuto del puntatore a + i

- Se p è un puntatore l’istruzione

  p[i]

  è equivalente

  *(p + i)

- Consideriamo le dichiarazioni
  - #define N 100
  - int a[N] , i , *p , sum =0;
- Il sistema alloca memoria contigua per contenere 100 interi

- Consideriamo le dichiarazioni
  - #define N 100
  - int a[N] , i , *p , sum =0;
- Il sistema alloca memoria contigua per contenere 100 interi
- L’istruzione

  p = a;

  è equivalente a

  p = &a[0];

- Consideriamo le dichiarazioni
  - #define N 100
  - int a[N] , i , *p , sum =0;
- Il sistema alloca memoria contigua per contenere 100 interi
- L’istruzione

  p = a +1;

  è equivalente a

  p = &a[1];

Somma elementi di un array

```
for ( i =0; i < N ; i++)
    sum += a[i];
```

```
p = a;
for ( i =0; i < N ; i++)
    sum += p[i];
```

```
for ( i =0; i < N ; i++)
    sum += *(a + i);
```

```
for ( p=a; i < &a[N] ; p++)
    sum += *p;
```

- Il vettore a è un puntatore costante
- Le istruzioni

```
a = p;
++ a ;
a += 2;
```

sono illegali

Array come parametri di funzioni

- Nella definizione delle funzioni un parametro dichiarato come array è effettivamente un puntatore
- Quando un array è passato come argomento l’indirizzo base è passato come passaggio per valore
- Gli elementi dell’array non sono copiati
- Come esempio proviamo a scrivere una funzione che somma gli elementi di un array di double

```
double sum_array(double a[], int n) // n e' la dimensione di a[]
{
   int   i;
   double sum = 0.;

   for (i = 0; i < n; ++i)
      sum += a[i];
   return sum;
}
```

```
double sum_array(double *a, int n)
{
   int   i;
   double sum = 0;

   for (i = 0; i < n; ++i)
      sum += a[i];
   return sum;
}
```

`double v[100];`	
Varie maniere in cui la funzione sum_array() può essere chiamata	
Chiamata	Cosa viene calcolato
`sum_array(v, 100);`	`v[0] + v[1] + … + v[99];`
`sum_array(v, 88);`	`v[0] + v[1] + … + v[99];`
`sum_array(&v[7], k – 7);`	`v[7] + v[8] + . . . v[k-1];`
`sum_array(v+7, 2*k);`	`v[7] + v[8] + … + v[2*k+6];`

Esempio: bubble sort

```
void   swap(int *, int *);

void bubble(int *a, int n)     /* n is the size of a */
{
   int    i, j;

   for (i = 0; i < n - 1; ++i)
      for (j = n - 1; j > i; --j)
         if (a[j-1] > a[j])
            swap(a+j-1, a+j);
}
```

```
void   swap(int *, int *);

void bubble(int *a, int n)     /* n is the size of a[] */
{
   int    i, j;

   for (i = 0; i < n - 1; ++i)
      for (j = n - 1; j > i; --j)
         if (a[j-1] > a[j])
            swap(a+j-1, a+j);
}
```

Dati	7	3	66	3	-5	22	-77	2
Passo 1	-77	7	3	66	3	-5	22	2
Passo 2	-77	-5	7	3	66	3	2	22
Passo 3	-77	-5	2	7	3	66	3	22
Passo 4	-77	-5	2	3	7	3	66	22
Passo 5	-77	-5	2	3	3	7	22	66
Passo 6	-77	-5	2	3	3	7	22	66
Passo 7	-77	-5	2	3	3	7	22	66

Esempio: selection sort

```
void   swap(int *, int *);

void selection_sort(int *a, int n)     /* n is the size of a */
{
   int    i, j;

   for (i = 0; i < n - 1; ++i)
      for (j = i + 1; j < n ; j++)
         if (a[j] < a[i])
            swap(a+j, a+i);
}
```

Esempio: selection sort

```
void   swap(int *, int *);

void selection_sort(int *a, int n)      /* n is the size of a */
{
   int    i, j;

   for (i = 0; i < n - 1; ++i)
      for (j = i + 1; j < n ; j++)
         if (a[j] < a[i])
            swap(a+j, a+i);
}
```

Dati	7	3	66	3	-5	22	-77	2
Passo 1	-77	7	66	3	3	22	-5	2
Passo 2	-77	-5	66	7	3	22	3	2
Passo 3	-77	-5	2	66	7	22	3	3
Passo 4	-77	-5	2	3	66	22	7	3
Passo 5	-77	-5	2	3	3	66	22	7
Passo 6	-77	-5	2	3	3	7	66	22
Passo 7	-77	-5	2	3	3	7	22	66

Allocazione dinamica

- Il C fornisce le funzioni malloc() e calloc()
- I prototipi sono in stdlib.h
- malloc sta per *memory allocation*
- calloc sta per *contiguous allocation*
- Queste funzioni sono usate per creare spazio dinamicamente per array, struct e union

- La calloc() prende 2 parametri di tipo size_t
- Tipicamente size_t è unsigned int
- Una chiamata della forma

  calloc(n , el_size)

  alloca spazio contiguo per un array di n elementi di el_size byte (si usa la sizeof())

- Lo spazio è inizializzato con tutti i bit messi a 0
- Se la funzione ha successo ritorna un puntatore di tipo void * che punta alla base dell'array
- Se non ha successo ritorna NULL

- La malloc() si usa in maniera simile
- La funzione ha solo un argomento di tipo size_t
- Se la chiamata ha successo ritorna un puntatore di tipo void * che punta alla base della memoria richiesta
- Se non ha successo ritorna NULL
- La malloc() non inizializza a 0 la memoria, e può essere poco più veloce

```
a = (int *) calloc(n, sizeof(int));
a = (int *) malloc(n * sizeof(int));
```

Liberare memoria dinamica

- Lo spazio allocato dinamicamente non viene liberato automaticamente all'uscita dalle funzioni
- Si usa la free() per liberare esplicitamente la memoria
- Una chiamata della forma

```
free(( void *) ptr );
```

libera la memoria puntata dal ptr

- Se ptr è NULL la funzione non fa niente
- Se ptr non è NULL il puntatore deve essere l’indirizzo base della memoria allocata, altrimenti la funzione ritorna un errore
- L’effetto dell’errore dipende dal sistema

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void   fill_array(int *a, int n);
int    sum_array(int *a, int n);
void   wrt_array(int *a, int n);

int main(void)
{
   int   *a, n;

   srand(time(NULL));     /* seed the random number generator */

   printf("Input n: ");
   while (scanf("%d", &n) == 1 || n >= 1) {
      putchar('\n');

      /* allocate space for a[] */
      a = (int *) calloc(n, sizeof(int));
      fill_array(a, n);
      wrt_array(a, n);
      printf("sum = %d\n\n", sum_array(a, n));
      free((void *)a);
      printf("Input n: ");
   }
   printf("\nBye!\n\n");
   return 0;
}
```

```
void fill_array(int *a, int n)
{
   int   i;

   for (i = 0; i < n; ++i)
      a[i] = rand() % 19 - 9;
}

int sum_array(int *a, int n)
{
   int   i, sum = 0;

   for (i = 0; i < n; ++i)
      sum += a[i];
   return sum;
}


void wrt_array(int *a, int n)
{
   int   i;

   printf("a = [");
   for (i = 0; i < n; ++i)
      printf("%d%s", a[i], ((i < n - 1) ? ", " : "]\n"));
}
```

Stringhe

- Le stringhe sono vettori monodimensionali di caratteri
- In C una stringa è terminata dal carattere nullo \0
- \0 è un byte con tutti i bit off: il suo valore decimale è 0
- La dimensione di una stringa deve includere spazio anche per il carattere terminale
- Il vettore "a" ha quindi due elementi
- Il programmatore deve stare attento a non superare le dimensioni del vettore

- Una stringa costante è trattata dal compilatore come un puntatore
- Il codice

```
char * p = "abc" ;
printf ( " %s %s \ n " , p , p +1);
```

stamperà a schermo

```
abc bc
```

- Poiché una stringa costante è trattata come un puntatore si può scrivere

```
char a , b ;
a = " abc "[1];
b = *( "abc" +2);
printf ( " %c %c \ n " , a , b );
```

- che produrrà a schermo

```
b c
```

- Poiché una stringa costante è trattata come un puntatore si può scrivere

```
char a , b ;
a = " abc "[1];
b = *( "abc" +2);
printf ( " %c %c \ n " , a , b );
```

- che produrrà a schermo

```
b c
```

- Ovviamente espressioni di questo tipo non vanno mai usate nel codice

- Il carattere nullo di una stringa non va scritto come output
- Scrivere caratteri nulli su un file potrebbe causare problemi a programmi che li elaborano
- Scrivere invece stringhe nulle può essere utile

```
char   *s:
int    nfrogs;

.....  // get nfrogs from somewhere
s = (nfrogs == 1) ? "" : "s";
```

```c
/* Count the number of words in a string. */

#include <ctype.h>

int word_cnt(const char *s)
{
   int    cnt = 0;

   while (*s != '\0') {
      while (isspace(*s))     /* skip white space */
         ++s;
      if (*s != '\0') {           /* found a word */
         ++cnt;
         while (!isspace(*s) && *s != '\0')
            ++s;                  /* skip the word */
      }
   }
   return cnt;
}
```

La parola chiave const

- Introdotta in ANSI C e non disponibile in C tradizionale
- Una variabile dichiarata come const può essere inizializzata, ma non modificata
- Consideriamo

```
const int
k = 3;
k = 1;
```

- il compilatore segnalerà l'errore

```
error : assignment of read - only variable 'k '
```

- La dichiarazione

  ```
  const int *p;
  ```

- indica che p è un puntatore a una variabile const int
  - Il contenuto del puntatore non può essere modificato
  - Il puntatore può essere modificato

```c
/* Count the number of words in a string. */

#include <ctype.h>

int word_cnt(const char *s)
{
   int    cnt = 0;

   while (*s != '\0') {
      while (isspace(*s))     /* skip white space */
         ++s;
      if (*s != '\0') {           /* found a word */
         ++cnt;
         while (!isspace(*s) && *s != '\0') {
            *s = 'a';
            ++s;                  /* skip the word */
         }
      }
   }
   return cnt;
}
```

```c
/* Count the number of words in a string. */

#include <ctype.h>

int word_cnt(const char *s)
{
   int    cnt = 0;

   while (*s != '\0') {
      while (isspace(*s))     /* skip white space */
         ++s;
      if (*s != '\0') {           /* found a word */
         ++cnt;
         while (!isspace(*s) && *s != '\0') {
            *s = 'a';             // Compiler will complain *****
            ++s;                  /* skip the word */
         }
      }
   }
   return cnt;
}
```

Alcune funzioni per il trattamento dei caratteri

- Prototipi in ctype.h
- int isblank(int c)

  Ritorna vero se c è uno spazio o un tab. Non disponibile in Microsoft C
- int isdigit(int c)

  Verifica se c è una cifra
- int isalpha(int c)

  Verifica se c è una lettera

- int isalnum(int c)

  Verifica se c è una lettera o una cifra
- int islower(int c)

  Verifica se c è una lettera minuscola
- int isupper(int c)

  Verifica se c è una lettera maiuscola

- int tolower(int c)

  Se c è una lettera maiuscola la restituisce come minuscola, altrimenti restituisce lo stesso carattere

- int toupper(int c)

  Se c è una lettera minuscola la restituisce come maiuscola, altrimenti restituisce lo stesso carattere

Alcune funzioni per il trattamento delle stringhe

- Prototipi nel file string.h
- char *strcpy(char *s1, const char *s2)

  Copia la stringa s2 nell'array s1. Viene restituito il valore di s1
- char *strncpy(char *s1, const char *s2, size_t n)

  Copia al massimo n caratteri dalla stringa s2 nell'array s1. Viene restituito il valore di s1

- char *strcat(char *s1, const char *s2)

  Aggiunge la stringa s2 in coda all'array s1. Viene restituito il valore di s1

- char *strncat(char *s1, const char *s2, size_t n)

  Aggiunge al massimo n caratteri della string s2 in coda all'array s1. Viene restituito il valore di s1

- int strcmp(const char *s1, const char *s2)

  Confronta la stringa s1 con la stringa s2. Restituisce 0, un negativo, o un positivo se s1 è, rispettivamente, uguale, minore o maggiore di s2
- int strncmp(const char *s1, const char *s2, size_t n)

  Simile a strcmp(), ma confronta solo i primi n caratteri di s1 e s2

- char *strchr(const char *s, int c)

  Localizza la prima occorrenza del carattere c nella string s: se c viene trovato ritorna un puntatore a c in s, altrimenti restituisce NULL

- size_t strcspn(const char *s1, const char *s2)

  Determina e restituisce la lunghezza del segmento iniziale della stringa s1 costituito da caratteri non contenuti nella stringa s2

```
printf ( " %d \n " , strcspn ( "abc" , "abc" ));        //  0
printf ( " %d \n " , strcspn ( "abc" , "abcde" ));      //  0
printf ( " %d \n " , strcspn ( "djpc" , "lqrabc" ));    //  3
```

- size_t strspn(const chr *s1, const char *s2)
- Determina e restituisce la lunghezza del segmento iniziale della stringa s1 costituito solo da caratteri contenuti nella stringa s2

```
printf ( " %ld \n " , strspn ( "abc" , "abc" ));          //  3
printf ( " %ld \n " , strspn ( "abc" , "abcde" ));        //  3
printf ( " %ld \n " , strspn ( "djpc" , "lqrabc" ));      //  0
```

- char *strpbrk(const char *s1, const char *s2)

  Localizza la prima occorrenza di un carattere di s2 nella stringa s1. Restituisce il puntatore al carattere, se trovato, o NULL

- char *strrchr(const char *s, int c)

  Localizza l'ultima occorrenza di c in s. Restituisce il puntatore a c,altrimenti NULL

- char *strstr(const char *s1, const char *s2)

  Localizza la prima occorrenza della stringa s2 nella stringa s1. Se ritrovata restituisce il puntatore alla stringa in s1, altrimenti NULL

- Non c’è niente di particolarmente speciale in queste funzioni
- Possono essere scritte in C semplicemente
- Vediamo alcuni esempi

size_t strlen (const char *s)

size_t strlen (const char *s)

```
size_t strlen(const char *s)
{
   register size_t   n;

   for (n = 0; *s != '\0'; ++s)
      ++n;
   return n;
}
```

char *strcpy(char *s1, const char *s2)

char *strcpy(char *s1, const char *s2)

```
char *strcpy(char *s1, const char *s2)
{
   register char *p = s1;

   while (*p++ = *s2++);

   return s1;
}
```

char *strcpy(char *s1, const char *s2)

```
char *strcpy(char *s1, const char *s2)
{
   register char *p = s1;

   while (*p++ = *s2++);

   return s1;
}
```

```
#include <stdio.h>
#include <stdlib.h>

char *strcpy(char *s1, const char *s2);

int main(void)
{
  char *s1, *s2;

  s1 = (char *) malloc(3*sizeof(char));
  s2 = "abc";
  s1 = strcpy(s1, s2);

  printf("%s %s \n",s1,s2);
  return 0;
}
```

char* strcat (char *s1, const char *s2)

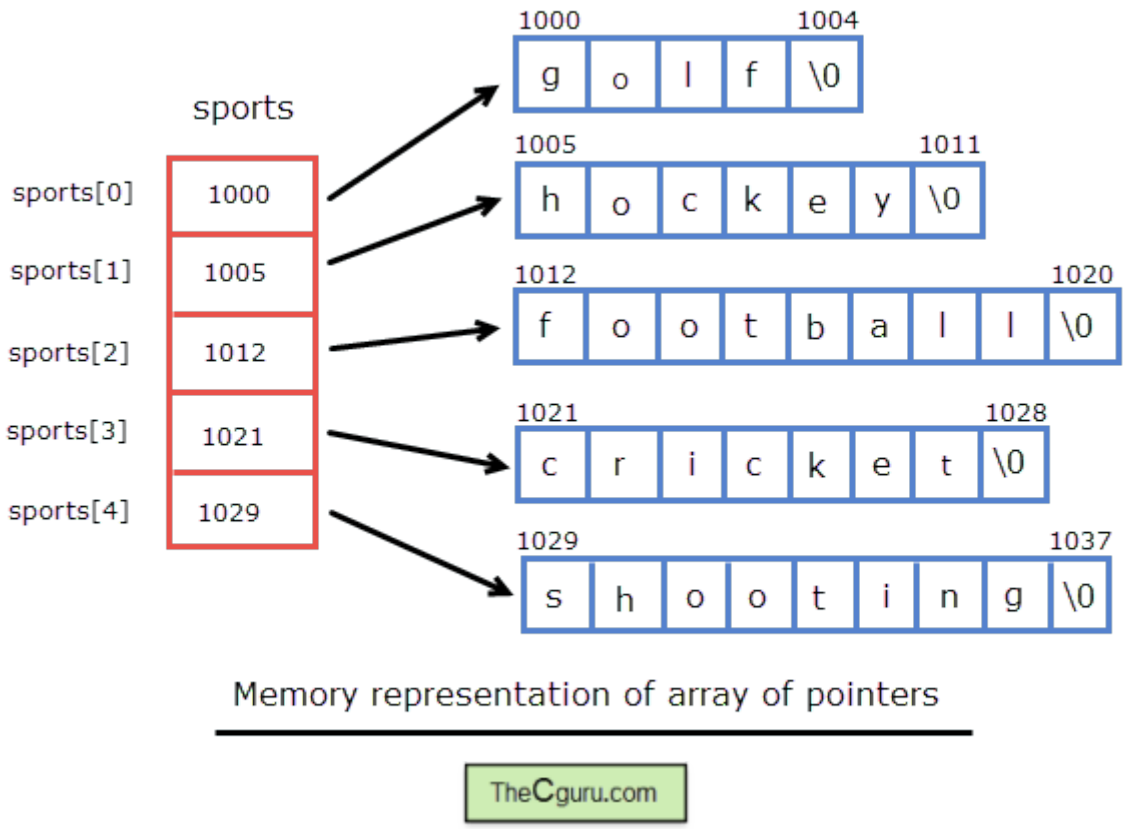
char* strcat (char *s1, const char *s2)

```
char   *strcat(char *s1, const char *s2)
{
   register char   *p = s1;

   while (*p)
      ++p;
   while (*p++ = *s2++)
      ;
   return s1;
}
```

```
#include <stdio.h>
#include <stdlib.h>

char *strcat(char *s1, const char *s2);

int main(void)
{
  char *s1, *s2;

  s1 = (char *) malloc(10*sizeof(char));
  s1 = strcpy(s1,”abc”);
  s2 = "efg";
  s1 = strcat(s1, s2);

  printf("%s %s \n",s1,s2);
  return 0;
}
```

Array di puntatori

- La dichiarazione
  - int *a[20];
- definisce a come
  - un array di 20 elementi
  - ogni elemento è un puntatore a intero
- Può essere quindi considerato come un array di vettori
- Esempio: un programma per ordinare le parole in un file (leggendo da stdin)



Memory representation of array of pointers

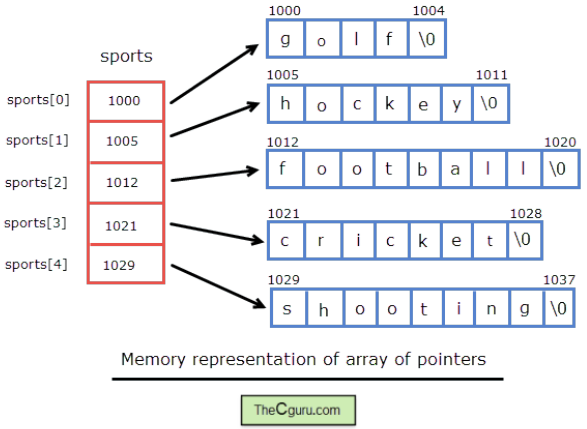
TheCguru.com

file sort.h

```
#define   MAXWORD   50          /* max word size */
#define   N         300         /* array size of w[] */

void   error_exit_calloc_failed(void);
void   error_exit_too_many_words(void);
void   error_exit_word_too_long(void);
void   sort_words(char *w[], int n);
void   swap(char **p, char **q);
void   wrt_words(char *w[], int n);
```

file main.c

```
/* Sort words lexicographically. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sort.h"

int main(void)
{
   char   word[MAXWORD];      /* work space */
   char   *w[N];              /* an array of pointers */
   int    n;                  /* number of words to be sorted */
   int    i;

   for (i = 0; scanf("%s", word) == 1; ++i) {
      if (i >= N)
         error_exit_too_many_words();
      if (strlen(word) >= MAXWORD)
         error_exit_word_too_long();
      w[i] = calloc(strlen(word) + 1, sizeof(char));
      if (w[i] == NULL)
         error_exit_calloc_failed();
      strcpy(w[i], word);
   }
   n = i;
   sort_words(w, n);          /* sort the words */
   wrt_words(w, n);           /* write sorted list of words */
   return 0;
}
```



file error.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sort.h"

void error_exit_calloc_failed(void)
{
   printf("%s",
      "ERROR: The call to calloc() failed to\n"
      "       allocate the requested memory - bye!\n");
   exit(1);
}

void error_exit_too_many_words(void)
{
   printf("ERROR: At most %d words can be sorted - bye!\n", N);
   exit(1);
}

void error_exit_word_too_long(void)
{
   printf("%s%d%s",
      "ERROR: A word with more than ", MAXWORD, "\n"
      "       characters was found - bye!\n");
   exit(1);
}
```

file wrt.c

```
#include <stdio.h>
#include "sort.h"

void wrt_words(char *w[], int n)
{
   int   i;

   for (i = 0; i < n; ++i)
      printf("%s\n", w[i]);
}
```

file swap.c

```
#include <stdio.h>
#include "sort.h"

void swap(char **p, char **q)
{
   char   *tmp;

   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

file sort_words.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sort.h"

void sort_words(char *w[], int n) /* n elements are to be sorted */
{
  int   i, j;

  for (i = 0; i < n; ++i)
    for (j = i + 1; j < n; ++j)
      if (strcmp(w[i], w[j]) > 0)
        swap(&w[i], &w[j]);
}
```

Array multidimensionali

- In C è possibile definire array di array

```
int a[100];          //array 1D
int b[2][7];         //array 2D
int c[5][3][2];      //array 3D
```

- Tutti gli elementi dell'array sono immagazzinati in memoria contigua
- La memoria contigua inizia dal primo elemento

```
a[0]          // 1D
b[0][0]       // 2D
c[0][0][0]    // 3D
```

- Gli array bidimensionali possono essere pensati naturalmente come una matrice
- Solitamente data la dichiarazione

```
int a[3][5];
```

3 indica il numero di righe e 5 il numero di colonne

- Ci sono varie maniere per accedere a un elemento a[i][j] di un array bidimensionale

```
*( a[i] + j )
(*( a + i ))[j]
*((*( a + i ))+ j )
*(& a[0][0] + 5* i + j )
```

Storage mapping function

- La funzione che per ogni array associa agli indici il valore è chiamata *storage mapping function*
- Per un array monodimensionale la funzione è specificata da

$$a[i] \rightarrow *(\&a[0] + i)$$

Storage mapping function

- La funzione che per ogni array associa agli indici il valore è chiamata *storage mapping function*
- Per un array bidimensionale la storage mapping function è specificata da

  a[i][j] → *(a[0][0] + N * i + j)

  dove N è il numero di colonne. Non viene usata la prima dimensione

- Vettori con più di due dimensioni si usano in maniera simile
- Se dichiariamo

  int A [7][9][2];

  il compilatore alloca spazio contiguo per 126 interi
- L'indirizzo base è &A[0][0][0]
- La storage mapping function è specificata da

  A[i][j][k] → *(&A[0][0][0] + 9*2* i + 2* j + k)
- Anche in questo caso non viene usata la prima dimensione

Inizializzazione

- Un array multidimensionale può essere inizializzato in molte maniere
- Queste inizializzazioni sono equivalenti

```
int A[2][3] = {1 ,2 ,3 ,4 ,5 ,6};
int A[2][3] = {{1 ,2 ,3} , {4 ,5 ,6}};
int A[][3] = {{1 ,2 ,3} , {4 ,5 ,6}};
int A[][3] = {1 ,2 ,3 ,4 ,5 ,6};
```

- Se non ci sono parentesi interne gli elementi sono inizializzati per riga nell'ordine A[0][0] , A[0][1] , A[0][2] , A[1][0] , ...
- Se i valori sono meno degli elementi dell'array i rimanenti elementi sono inizializzati a 0
- Se la prima parentesi è vuota il compilatore prende la dimensione dal numero di parentesi interne
- Tutte le dimensioni devono essere esplicitate, eccetto la prima

- Queste inizializzazione sono equivalenti

```
int A[2][2][3] = {
    {{1 ,1 ,0} , {2 ,0 ,0}} ,
    {{3 ,0 ,0} , {4 ,4 ,0}}
};
int A[][2][3] = {{{1 ,1}, {2}} , {{3} , {4 ,4}}};
```

- In generale se un array di storage class automatic non è esplicitamente inizializzato, allora gli elementi potrebbero contenere qualsiasi valore
- Una semplice maniera di inizializzare tutti gli elementi a 0 è

```
int A[2][2][3] = {0};
```

- In questo caso vanno dichiarate tutte le dimensioni.

Dichiarazione di parametri formali

- Un array multidimensionale può essere usato come parametro di una funzione
- Il compilatore nel generare il codice oggetto della funzione ha bisogno della storage mapping function per l’array
- Per questo motivo nella dichiarazione del parametro array è necessario esplicitare tutte le dimensioni, eccetto la prima

- Ricordiamo che data, per esempio, la dichiarazione

  int A[7][9][2];

  la storage mapping function è data da

  A[i][j][k] → *(& A[0][0][0] + 9*2* i + 2* j + k)

- Supponiamo la dichiarazione
  - int A[3][5];
- Scriviamo una funzione che calcoli la somma degli elementi della matrice

Somma elementi di una matrice

```
int sum(int a[3][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

Somma elementi di una matrice

```
int sum(int a[3][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

```
int sum(int a[][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

Somma elementi di una matrice

```
int sum(int a[3][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

```
int sum(int a[][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

```
int sum(int (*a)[5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

Somma elementi di una matrice

```
int sum(int a[3][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

```
int sum(int a[][5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

Vanno bene solo per matrici di dimensione 3x5

```
int sum(int (*a)[5])
{
   int   i, j, sum = 0;

   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

```
int sum(int a[][5], int m)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

```
int sum(int a[][5], int m)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

Questa versione va bene per array di dimensioni mx5

Si potrebbe pensare di ...

```
int sum(int **a, int m, int n)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < n; ++j)
         sum += a[i][j];
   return sum;
}

int main()
{
   int A[3][5], s;

   .....
   s = sum(A,3,5);
   .....
}
```

Si potrebbe pensare di ...

```
int sum(int **a, int m, int n)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < n; ++j)
         sum += a[i][j];
   return sum;
}

int main()
{
   int A[3][5], s;

   .....
   s = sum(A,3,5);
   .....
}
```

```
prova.c: In function 'main':
prova.c:17:4: warning: passing argument 1 of 'sum' from
incompatible pointer type [enabled by default]
prova.c:3:5: note: expected 'int **' but argument is of
type 'int (*)[5]'
```

Si potrebbe pensare di ...

```
int sum(int **a, int m, int n)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < n; ++j)
         sum += a[i][j];
   return sum;
}

int main()
{
   int A[3][5], s;

   .....
   s = sum(A,3,5);
   .....
}
```

```
prova.c: In function 'main':
prova.c:17:4: warning: passing argument 1 of 'sum' from
incompatible pointer type [enabled by default]
prova.c:3:5: note: expected 'int **' but argument is of
type 'int (*)[5]'
```

Non si generalizza quanto succede per gli array monodimensionali

- Un nome di un array è un indirizzo che non può essere cambiato, un puntatore invece si
- Se consideriamo

  ```
  int A[3][5];
  ```

  il contenuto di ogni A[i] è un array monodimensionale. A è quindi un puntatore ad array monodimensionali

- Con

  int **a ;

- a è un puntatore a puntatori a int.
- si può fare

  a[i] = ... ;

- Quindi A[n][m] e **a non sono equivalenti

- In generale è meglio usare i puntatori e allocare la memoria dinamicamente
- Un’altra maniera di rendere la funzione sum più generale è usare l’indirizzo base dell’array e le dimensioni

```
int sum(int **a, int m, int n)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < n; ++j)
         sum += a[i][j];
   return sum;
}

int main()
{
   int **A, s;

   //alloca dinamicamente A
   .....
   s = sum(A,3,5);
   .....
}
```

Si potrebbe pensare di ...

```
int sum(int **a, int m, int n)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < n; ++j)
         sum += a[i][j];
   return sum;
}

int main()
{
   int A[3][5], s;

   .....
   s = sum(A,3,5);
   .....
}
```

```
int sum(int *a, int m, int n)
{
   int   i, j, sum = 0;

   for (i = 0; i < m; ++i)
      for (j = 0; j < n; ++j)
         sum += *(a + n*i + j);
   return sum;
}

int main()
{
   int A[3][5], s;

   .....
   s = sum(&A[0][0],3,5);
   .....
}
```

Allocazione dinamica di matrici

- L’uso degli array multidimensionali non è molto flessibile
- Non possono essere passati direttamente come parametri di funzioni senza ledere in generalità o semplicità
  - Funzioni per matrici a dimensioni fisse

    ```
    int f( int a[][3]);
    ```

  - Funzioni a cui non si passa la matrice, ma l’indirizzo base

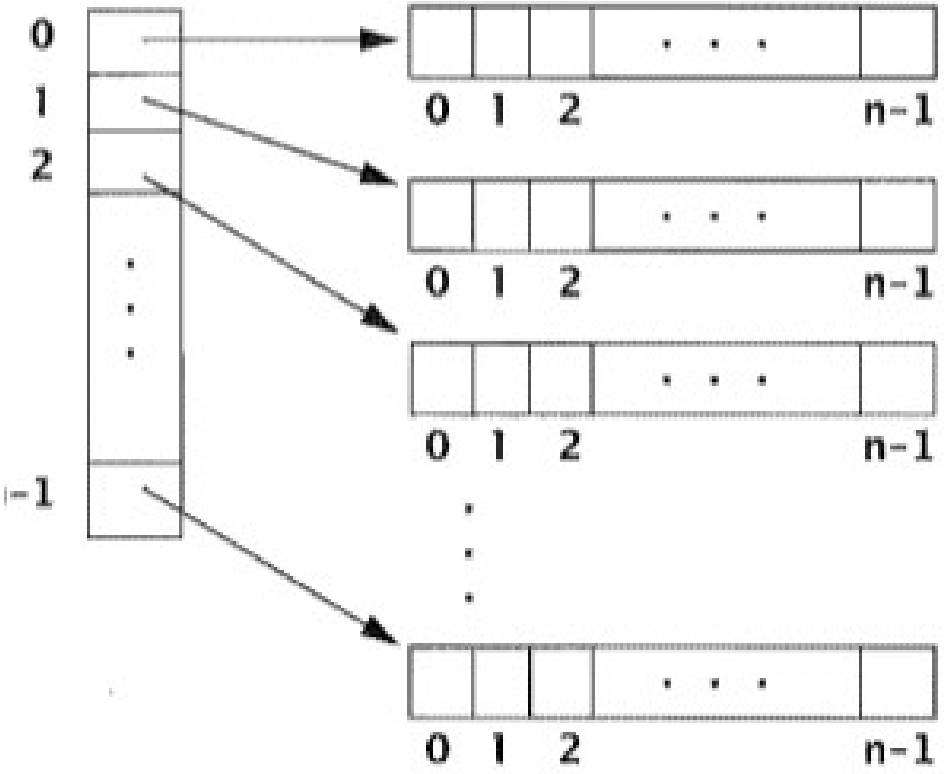
    ```
    int f( int *a , m , n );
    f(&a[0][0] , 2 ,3);
    ```

- Più semplice poter avere funzioni a cui si passa direttamente la matrice e, se necessarie, le dimensioni

```
int  ( int **a , int m , int n );
```

```
int i, m, n;
double **a;

.....
a = malloc(m*sizeof(double *));
for (i=0; i<n; i++)
  a[i] = malloc(n*sizeof(double));
```

```
int i, m, n;
double **a;

.....
a = malloc(m*sizeof(double *));
for (i=0; i<n; i++)
  a[i] = malloc(n*sizeof(double));
```



- La memoria allocata per la matrice non è contigua
- Al generico elemento si accede con la comoda scrittura a[i][j]
- La memoria si libera facendo una free per ogni riga

```
double **matrix_alloc(int m, int n)
{
  int i;
  double p;

  p = malloc(m*sizeof(double *));
  for (i=0; i<m; i++)
    p[i] = malloc(n*sizeof(double));

  return p;
}


void free_matrix(double **a, int m)
{
  int i;

  for (i=0; i<m; i++)
    free(a[i]);

  free(a);
}
```

Allocazione con memoria contigua

Allocazione con memoria contigua

```
double **matrix_alloc(int m, int n)
{
  int i;
  double **a;

  a = malloc(m*sizeof(double *));
  a[0] = malloc(m*n*sizeof(double));
  for (i=1; i<m; i++)
    a[i] = a[i-1] + n;

  return a;
}

double free_matrix(double **a)
{
  free(a[0]);
  free(a);
}
```

Variable length array

- In C99 è stato introdotto il meccanismo degli array a lunghezza variabile
- Meccanismo per usare array come variabili auto anche quando la dimensione si conosce solo a run-time
- Comodo ma ci sono dei contro

```c
#include <stdio.h>

int main(void)
{
   printf("%s", "Enter size of a one-dimensional array: ");
   int arraySize; // size of 1-D array
   scanf("%d", &arraySize);

   int array[arraySize]; // declare 1-D variable-length array

   printf("%s", "Enter number of rows and columns in a 2-D array: ");
   int row1, col1; // number of rows and columns in a 2-D array
   scanf("%d %d", &row1, &col1);

   int array2D1[row1][col1]; // declare 2-D variable-length array

   printf("%s",
      "Enter number of rows and columns in another 2-D array: ");
   int row2, col2; // number of rows and columns in another 2-D array
   scanf("%d %d", &row2, &col2);

   int array2D2[row2][col2]; // declare 2-D variable-length array

   printf("\nsizeof(array) yields array size of %d bytes\n",
      sizeof(array));      // test sizeof operator on VLA

   for (size_t i = 0; i < arraySize; ++i) {
      array[i] = i * i;
   }  // assign elements of 1-D VLA

   for (size_t i = 0; i < row1; ++i) {
      for (size_t j = 0; j < col1; ++j) {
         array2D1[i][j] = i + j;
      } // assign elements of first 2-D VLA
   }

   for (size_t i = 0; i < row2; ++i) {
      for (size_t j = 0; j < col2; ++j) {
         array2D2[i][j] = i + j;
      }
   } // assign elements of second 2-D VLA

   puts("\nOne-dimensional array:");
   print1DArray(arraySize, array); // pass 1-D VLA to function

   puts("\nFirst two-dimensional array:");
   print2DArray(row1, col1, array2D1); // pass 2-D VLA to function

   puts("\nSecond two-dimensional array:");
   print2DArray(row2, col2, array2D2); // pass other 2-D VLA to function
}
```

Stack e heap

- Lo stack è la memoria messa da parte come spazio di lavoro per programma. Quando viene chiamata una funzione, viene riservato un blocco in cima allo stack per le variabili locali e alcuni dati.
- L'heap è la memoria riservata per l'allocazione dinamica.
- Lo stack è tipicamente molto piccolo rispetto all’heap.

- I VLA sono allocati nello stack
  - Non se ne conosce la dimensione in fase di compilazione
  - Potenziale rischio di stack overflow
- Comunque utilizzabili per piccole dimensioni