UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - Gestione della memoria e array

Ultimo aggiornamento: 04/11/2020

Visibilità delle variabili

- Abbiamo visto che in C possiamo dichiarare variabili in diversi modi.
- Il primo modo è dichiarare la variabile al di fuori di ogni funzione.
- Questa variabile può essere utilizzata in tutte le funzioni definite nel file da quel punto in poi. Si dice che l'identificatore ha *file scope*, ovvero ha *visibilità a livello di file*.
- Vengono dette anche variabili globali, anche se è un termine poco preciso.
- Un secondo modo è quello di dichiarare la variabile all'interno di un blocco di istruzioni, ovvero tra due parentesi graffe. In questo caso la variabile può essere utilizzata solo all'interno di quel blocco o di blocchi in esso contenuti.
- In questo secondo caso si dice che l'identificatore ha *block scope*, ovvero ha *visibilità a livello di blocco*.
- Vengono dette anche variabili locali, anche se è un termine ancora meno preciso (vedremo perché).

Visibilità delle variabili

```
int x;

int f(void)
{
        int y;
        y = x;
        return y;
}

int main(void)
{
        int z;

        x = 25;
        z = f();
        // z = y; // Questo sarebbe un errore!

        return 0;
}
```

Variabile con visibilità a livello di file

Variabile con visibilità a livello di blocco

Uso di una variabile definita esternamente alla funzione: corretto, la variabile ha file scope.

Uso di una variabile definita esternamente alla funzione: errato, la variabile ha block scope e il blocco non è quello corrente.

Tempo di vita delle variabili

- La memoria in cui risiedono i dati di una variabile deve essere allocata (riservata) e successivamente liberata.
- Il tempo di esecuzione che intercorre dalla allocazione alla deallocazione di una variabile viene detto *tempo di vita* o *lifetime*.
- Per ora abbiamo sempre considerato la visibilità in diretta corrispondenza con il tempo di vita:
  - le variabili con file scope (globali) hanno tempo di vita pari alla durata del programma. Questa è detta *static storage duration*, o variabile statica.
  - le variabili con block scope (locali) hanno tempo di vita pari al tempo di esecuzione del blocco in cui si trovano. Questa è detta *automatic storage duration* , o variabile automatica.
- Le variabili statiche vengono inizializzate una sola volta all'inizio del programma (prima di chiamare la funzione main) e, se non si specifica l'inizializzazione, vengono inizializzate a 0.
- Le variabili automatiche invece vengono inizializzate ogni volta che si entra nel blocco in cui sono definite, solo se l'inizializzazione è esplicita, altrimenti contengono valori casuali.

Tempo di vita delle variabili

- In realtà questi due concetti possono essere separati in un caso specifico: è possibile definire variabili con block scope, ma «allungarne la vita», ovvero dare a queste una *static storage duration*.
- Per farlo si utilizza la parola chiave **static** davanti alla definizione.
- [Colloquialmente] Questo consente in pratica di avere variabili globali con visibilità locale.

```
int f(void) {
    static int v;
    return v++;
}

int main(void) {
    int x, y, z;
    x = f();
    y = f();
    z = f();
    return 0;
}
```

- Che cosa valgono x, y e z dopo gli assegnamenti corrispondenti?

Tempo di vita delle variabili

- In realtà questi due concetti possono essere separati in un caso specifico: è possibile definire variabili con *block scope*, ma «allungarne la vita», ovvero dare a queste una *static storage duration*.
- Per farlo si utilizza la parola chiave **static** davanti alla definizione.
- [Colloquialmente] Questo consente in pratica di avere variabili globali con visibilità locale.

```
int f(void) {
    static int v;
    return v++;
}

int main(void) {
    int x, y, z;
    x = f();
    y = f();
    z = f();
    return 0;
}
```

La variabile è statica, quindi viene inizializzata a 0 all'inizio del programma.

v viene incrementato, ma ritorniamo il suo valore prima dell'incremento.

Ad ogni chiamata di f(), ritorniamo il valore corrente di v, che ogni volta viene incrementato.

- Che cosa valgono x, y e z dopo gli assegnamenti corrispondenti?
  x = 0, y = 1, z = 2

Quanto vale s alla fine del main()?

```
int main(void)
{
        int s = 0;

        for (int i = 0; i < 5; ++i) {
                int a = 0;
                ++a;
                s = s + a;
        }

        return 0;
}
```

Quanto vale s alla fine del main()?

```
int main(void)
{
        int s = 0;

        for (int i = 0; i < 5; ++i) {
                int a = 0;
                ++a;
                s = s + a;
        }

        return 0;
}
```

- **s = 5**

Quanto vale s alla fine del main()?

```
int main(void)
{
        int s = 0;

        for (int i = 0; i < 5; ++i) {
                static int a = 0;
                ++a;
                s = s + a;
        }

        return 0;
}
```

Quanto vale s alla fine del main()?

```
int main(void)
{
        int s = 0;

        for (int i = 0; i < 5; ++i) {
                static int a = 0;
                ++a;
                s = s + a;
        }

        return 0;
}
```

- **s = 15**

Ancora sul tempo di vita delle variabili

- Abbiamo detto che la storage duration può essere static o automatic. In ogni caso questa è definita al momento della compilazione.
- Ma noi abbiamo già visto un modo diverso di allocare memoria tramite specifiche funzioni: malloc(), calloc() e realloc().
- La storage duration dello spazio di memoria allocato da queste funzioni è detta, appunto, *allocated storage duration*.
- Il tempo di vita di questo spazio di memoria termina con la chiamata alla funzione free().

Gestione della memoria: variabili statiche

- Nel processore ADE8 abbiamo visto che si poteva dare un nome ad un byte, rendendolo quindi una variabile.
- Chi allocava e inizializzava quel byte in memoria?
- Nel simulatore lo facevamo noi, con il menù «Load Image…». Infatti al momento di caricare il programma, venivano caricati anche i dati presenti assieme alle istruzioni.
- Nei programmi C i dati non vengono mescolati alle istruzioni e li si mette tutti assieme prima o dopo le istruzioni.
- La parte del file binario che contiene il linguaggio macchina viene detta *segmento codice*, mentre quella che contiene i dati viene detta *segmento dati*. In inglese sono *text segment* e *data segment*.
- Il corrispondente in C delle variabili così create sono le variabili con static storage duration.
- **Le variabili globali e quelle dichiarate static finiscono quindi nel segmento dati.**

Gestione della memoria: variabili automatiche

- Le variabili con automatic storage duration invece devono essere allocate all'inizio di un blocco e deallocate alla fine, numerose volte e con dimensione variabile a seconda del blocco.
- Anche in ADE8 abbiamo avuto lo stesso problema e per questo abbiamo utilizzato il meccanismo dello stack.
- In C il compilatore genera il codice che si occupa di allocare e deallocare le variabili automatiche.
- Le variabili locali vengono riservate all'inizio della funzione con l'equivalente di qualche push sullo stack. Alla fine (dopo il return) si eseguono altrettante pop.
- Anche i parametri funzionano tramite lo stack: chi chiama la nostra funzione esegue prima la push dei parametri sullo stack e, dopo che la funzione è terminata, ne fa il pop.
- **I parametri e le variabili locali vengono messe sullo stack.**

- *In realtà il compilatore può comportarsi diversamente, ma questa semplificazione è spesso corretta e consente di evitare di complicare il discorso eccessivamente. Se siete interessati al tema, approfondirete in seguito.*

Gestione della memoria: variabili allocate

- Lo spazio di memoria con allocated storage duration viene esplicitamente allocato e deallocato dalla libreria standard, tipicamente tramite richieste al Sistema Operativo, che si occupa di tenere traccia di quale spazio è libero e quale invece è occupato.
- Questa memoria, è gestita in modi estremamente diversi e complessi dai diversi sistemi. Viene detta *memoria heap*, con riferimento ad un «mucchio» di cose.
- Tanto per chiarezza non abbiamo mai utilizzato la memoria in questo modo in ADE8.
- La memoria allocata sull'heap resta «in vita» (il Sistema Operativo la riserva per il nostro programma) finché non viene esplicitamente liberata con la free(), al contrario di quella utilizzata per lo stack, in cui i dati utilizzati per le funzioni vengono liberati quando esse ritornano al chiamante.
- **La memoria allocata dinamicamente con malloc(), calloc() e realloc() viene messa sull'heap.**

Allocare più dati dello stesso tipo

- Finora abbiamo visto solamente come allocare una serie di variabili dello stesso tipo che si troveranno a indirizzi consecutivi sfruttando l'allocazione dinamica della memoria (quindi l'area di memoria chiamata *heap*).
- È possibile fare la stessa cosa sfruttando invece lo stack o il segmento dati? Possiamo definire una serie di variabili dello stesso tipo automatiche o statiche?
- La risposta è sì. Ovviamente però non usando la funzione malloc(), dal momento che essa lavora solamente con l'heap e richiede la deallocazione.
- Per questo il C ci mette a disposizione altri strumenti sintattici.

Gli array

- In C è possibile definire una sequenza di variabili dello stesso tipo e riferirsi ad esse con un unico nome. Questo può essere fatto utilizzando un modificatore e una variabile di questo tipo è chiamata ***array***.
- La sintassi per definire un array utilizza le parentesi quadre ed è la seguente:

```
<tipo> <identificatore> [ <N> ] ;
```

- Dove *N* è un'espressione intera **costante** maggiore di zero che indica quanti elementi di tipo *tipo* si intende definire, ad esempio:

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

- Definisce un array di 5 int (che quindi occuperanno 20 byte in memoria).
- Anche in questo caso, come nell'allocazione dinamica, le variabili sono contigue in memoria, cioè si trovano a indirizzi consecutivi.

Uso degli array in espressioni

- Una volta dichiarato un array, come si può accedere ai valori in esso contenuti?
- In C quando l'identificatore di un array viene usato in una espressione, esso viene implicitamente convertito in un puntatore al suo primo elemento.
- In C inoltre non esistono operatori che siano specifici per gli array. Questi si usano semplicemente tramite puntatori.

```c
#include <stdlib.h>

int main(void)
{
	size_t n = 10;
	int *p = malloc(n * sizeof(int));

	for (size_t i = 0; i < n; ++i) {
		p[i] = 0;
	}

	free(p);

	return 0;
}
```

→

```c
#include <stdlib.h>

int main(void)
{
	int a[10];
	size_t n = 10;
	int *p = a;

	for (size_t i = 0; i < n; ++i) {
		p[i] = 0;
	}

	return 0;
}
```

Uso degli array in espressioni

- Per quanto detto però, non è indispensabile assegnare l'indirizzo del primo elemento di un array ad una variabile di tipo puntatore.
- Siccome l'uso del nome dell'array equivale all'uso di un puntatore al suo primo elemento, possiamo trattare gli array come se fossero puntatori (ricordando però che non lo sono!):

```c
#include <stdlib.h>

int main(void)
{
	int a[10];
	size_t n = 10;
	int *p = a;

	for (size_t i = 0; i < n; ++i) {
		p[i] = 0;
	}

	return 0;
}
```

→

```c
#include <stdlib.h>

int main(void)
{
	int a[10];
	size_t n = 10;

	for (size_t i = 0; i < n; ++i) {
		a[i] = 0;
	}

	return 0;
}
```

Uso degli array in espressioni

- Ci sono solamente due operatori che si comportano diversamente quando ricevono come parametro il nome di un array.
- L'operatore `sizeof` restituisce la dimensione complessiva in byte dell'array, non la dimensione di un puntatore. Dato il seguente codice:

```
int arr[10];
size_t s = sizeof(arr);
```

  il valore di s dopo l'esecuzione sarà 40, visto che l'array contiene 10 int che occupano 4 byte ciascuno.
- L'altro caso particolare è l'operatore `&`. Questo restituisce un «puntatore ad array di N elementi di tipo T». Cioè? Meglio aspettare un po' ad affrontare la cosa, ma sappiate che

```
int a[5];
int *p1 = a; // ok
int *p2 = &a; // errore: tipi incompatibili
```

Questa espressione è di tipo `int (*)[5]` non proprio una cosa facile da digerire!

Specificare la dimensione degli array

- Osservando il programma precedente si nota che abbiamo dovuto scrivere la dimensione dell'array due volte: una tra le parentesi quadre e una nell'inizializzazione della variabile n.
- Non sarebbe stato meglio fare così?

```
#include <stdlib.h>

int main(void)
{
      size_t n = 10;
      int a[n];

      for (size_t i = 0; i < n; ++i) {
           a[i] = 0;
      }

      return 0;
}
```

Non si può fare!!! n non è una costante.

- Nella definizione di un array bisogna utilizzare **una costante** tra le parentesi quadre.
- Questa è una estensione del C99 poco supportata, nota come *variable length array*.

Specificare la dimensione degli array

- Sarebbe allora bastato fare così?

```
#include <stdlib.h>

int main(void)
{
      const size_t n = 10;
      int a[n];

      for (size_t i = 0; i < n; ++i) {
           a[i] = 0;
      }

      return 0;
}
```

Non si può fare!!! n non è una costante.

- Una variabile di sola lettura non è una costante. **Una variabile const, non è una costante!**
- Quindi? Non c'è un modo per evitare di specificare due volte la dimensione?

Specificare la dimensione degli array

- Una soluzione è la seguente:

```
#include <stdlib.h>

int main(void)
{
	int a[10];
	size_t n = sizeof a / sizeof *a;

	for (size_t i = 0; i < n; ++i) {
		a[i] = 0;
	}

	return 0;
}
```

- L'espressione `sizeof a` calcola la dimensione dell'array in byte, mentre l'espressione `sizeof *a` calcola la dimensione del primo elemento.
- Dimensione totale diviso dimensione di un elemento fa il numero di elementi.
- Però…

Specificare la dimensione degli array

- Una soluzione è la seguente:

```c
#include <stdlib.h>

int main(void)
{
        int a[10];
        size_t n = sizeof a / sizeof *a;

        for (size_t i = 0; i < n; ++i) {
                a[i] = 0;
        }

        return 0;
}
```

NON FATELO

Specificare la dimensione degli array

- Una soluzione è la seguente:

```
#include <stdlib.h>

int main(void)
{
        int a[10];
        size_t n = sizeof a / sizeof *a;

        for (size_t i = 0; i < n; ++i) {
                a[i] = 0;
        }

        return 0;
}
```

NON FATELO

- Perché no? Perché gli studenti capiscono (sbagliando) che possono sempre calcolare la dimensione degli array in quel modo. Anche quando non si stanno più utilizzando degli array.
- In questo corso **è vietato utilizzare questo modo** di far calcolare al compilatore la dimensione di un array. Cioè **vengono tolti punti** a chi usa l'espressione durante l'esame.

Inizializzazione di array

- Supponiamo di dichiarare in un blocco di istruzioni un array di 10 interi a 32 bit con segno:

```
int arr[10];
```

- In questo caso i 10 interi quanto varranno? Anche questa volta la risposta è «boh». Come per le variabili locali i valori contenuti in un array non inizializzato conterranno valori a caso.
- In C è però possibile inizializzare i valori contenuti in un array utilizzando le parentesi graffe, usando le sintassi:

```
<tipo> <identificatore>[<N>] = { <espressione> , ... };
```

- In questo caso i valori dell'array vengono inizializzati con i valori delle espressioni contenute tra parentesi graffe, ogni espressione è separata dalla successiva da una virgola.

Inizializzazione di array

- Quando si usa l'inizializzazione di array così

```
<tipo> <identificatore>[<N>] = { <espressione> , ... };
```

- ci sono alcuni dettagli da tenere presente:
  - Quando è presente l'inizializzazione, N è opzionale e, nel caso venga omesso, l'array conterrà esattamente tanti valori quanti sono presenti nella lista.
  - Se N è presente, i valori contenuti nella lista non possono essere più di N.
  - Se N è presente e i valori contenuti nella lista sono meno di N, i restanti verranno inizializzati a 0.

Inizializzazione di array

- Alcuni esempi di inizializzazione di array:

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

float b[5] = { 1.1f, 3.2f, 4.3f };

int c[] = { 12, 1.4 };

double d[2] = { 5.4, 3.87, 3.55 };


int x = 1, y = 2, z = 3;

int e[] = { x, y, z };
```

Array contenente 3 int tutti inizializzati, rispettivamente a 1, 2 e 3.

Array contenente 5 float, di cui i primi tre inizializzati esplicitamente e gli ultimi due implicitamente a zero.

Array di due int. Warning: attenzione ai tipi dei letterali inseriti nella lista di inizializzazione! Qui 1.4 è un double che viene convertito automaticamente in un intero.

ERRORE: ho dichiarato un array di 2 double ma nella lista ne ho inseriti 3.

Array di 3 int inizializzato con i valori contenuti in 3 variabili locali.

Inizializzare gli array

- Il programma di prima, che riempiva l'array di zeri, può allora diventare:

```c
#include <stdlib.h>

int main(void)
{
	int a[10];
	size_t n = 10;

	for (size_t i = 0; i < n; ++i) {
		a[i] = 0;
	}

	return 0;
}
```

→

```c
#include <stdlib.h>

int main(void)
{
	int a[10] = { 0 };
	size_t n = 10;

	return 0;
}
```

- Adesso utilizziamo questi array per fare qualcosa, come ad esempio calcolare la media dei valori contenuti nell'array.

Media intera dei valori di un array di int

- Ecco una semplice soluzione:

```
#include <stdlib.h>

int main(void)
{
	int a[] = { 7, 9, 1, 13, 8 };
	size_t n = 5;

	int m = 0;
	for (size_t i = 0; i < n; ++i) {
		m += a[i];
	}
	m /= n;

	return 0;
}
```

- Adesso però vale la pena fare una funzione che esegua il compito.
- È chiaro che la funzione dovrà ritornare un int, ma quanti e di che tipo dovranno essere i parametri? Di sicuro la funzione dovrà accedere all'array.

Come dichiarare un parametro di tipo array?

- Non si può.
- «No, dai. Davvero.»
- Ripeto: **in C non esistono parametri di tipo array**.
- «Ma come, Prof.? Io l'ho sempre fatto e funziona. Ad esempio:»

```
void func(int a[4]) { … }
```

- «`a` è un parametro di tipo array di 4 int!»
- **Falso**. `a` è un parametro di tipo `int*`, ovvero puntatore a `int`. Cioè è identico a scrivere

```
void func(int *a) { … }
```

- O anche

```
void func(int a[]) { … }
```

Come passare gli array alle funzioni

- Supponiamo di voler scrivere una funzione alla quale vogliamo passare un array di 4 int come parametro, il C ci concede tre modi per farlo:

```
void func(int *a)
{ … }
```

```
void func(int a[4])
{ … }
```

```
void func(int a[])
{ … }
```

- Tra queste definizioni non c'è alcuna differenza! Le funzioni dichiarate sono esattamente identiche, per il C il parametro `a` sarà sempre un puntatore a int.
- Questo è perfettamente ragionevole se pensiamo che durante la chiamata alla funzione potremo passare come valore per l'inizializzazione dei parametri il nome dell'array. Usare il nome dell'array in una espressione, quindi anche in una chiamata a funzione, fa sì che questo nome diventi un puntatore a int che punterà al suo primo elemento.
- All'interno della funzione, potremo utilizzare *a, *(a+1) , *(a+2) , *(a+3), o gli equivalenti a[0], a[1], a[2], a[3]. Infatti sappiamo che gli array sono di dimensione 4.
- Ma se non sapessimo la dimensione dell'array?

Come passare gli array alle funzioni

- La prima soluzione a cui uno studente pensa è qualcosa di questo tipo:

```
void func(int a[]) {
        size_t n = sizeof a / sizeof *a;
        …
}
```

NOOOO!!!!!!!
Non funziona!

- Quanto vale n? (compilando a 32 bit)
- a è di tipo int*, quindi sizeof a vale 4
- *a è di tipo int, quindi sizeof *a vale 4
- n vale 1! Sempre!!! Indipendentemente dal numero di elementi dell'array che è stato utilizzato per inizializzare il parametro.
- Di conseguenza sarà necessario passare alla funzione anche il numero di elementi contenuti nell'array.
- La definizione della funzione diventerà quindi una delle due:

```
void func(int *a, size_t n)
{
        …
}
```

```
void func(int a[], size_t n)
{
        …
}
```

Funzione per la media intera di un array di int

- Vediamo allora la funzione per il calcolo della media:

```
#include <stdlib.h>

int media(int *a, size_t n) {
      int m = 0;
      for (size_t i = 0; i < n; ++i) {
           m += a[i];
      }
      return m / n;
}

int main(void) {
      int a[] = { 7, 9, 1, 13, 8 };

      int m = media(a, 5);

      return 0;
}
```

- Ricordate: **in C non si possono passare array alle funzioni**. Si può passare solo un puntatore al primo elemento.

Assegnamento tra array

- In C NON c'è alcun modo per assegnare direttamente il contenuto di un array a un altro, nessun operatore ci permette di farlo.
- Questo non significa che non sia possibile copiare il contenuto di un array in un altro. Significa solamente che sarà necessario scrivere una funzione per farlo, adattandola al caso specifico del programma che si sta realizzando.
- Se provassimo a scrivere

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

Il compilatore ci darà un errore. Non è possibile usare i nomi degli array per fare assegnamenti.

- Bisogna sempre utilizzarli come puntatori.
- **Il nome di un array non è mai un lvalue.**

Assegnamento tra array

- Supponiamo ad esempio di voler copiare il contenuto di un vettore di double in un altro:

```
void copia_arr(double *src, double *dst, size_t n)
{
    for (size_t i = 0; i < n; ++i)
        dst[i] = src[i];
}

int main(void)
{
    double a1[] = {2.3, 6.5, 1.1, 4};
    double *a2 = malloc(sizeof(double)* 4);

    copia_arr(a1, a2, 4);

    free(a2);
    return 0;
}
```

- Notare che non c'è alcuna differenza tra un array automatico e la memoria allocata da malloc, nel momento in cui il puntatore arriva ad una funzione.

Assegnamento tra array

- La funzione non ha bisogno di modificare il vettore sorgente, quindi possiamo scrivere:

```
void copia_arr(const double *src, double *dst, size_t n)
{
    for (size_t i = 0; i < n; ++i)
        dst[i] = src[i];
}
```

- Cosa abbiamo cambiato?
- Il primo parametro è stato dichiarato di tipo `const double *`.
- Questo è un «puntatore a const double», ovvero le espressioni `*src` o `src[i]` sono di tipo `const double`.
- Se scrivessimo

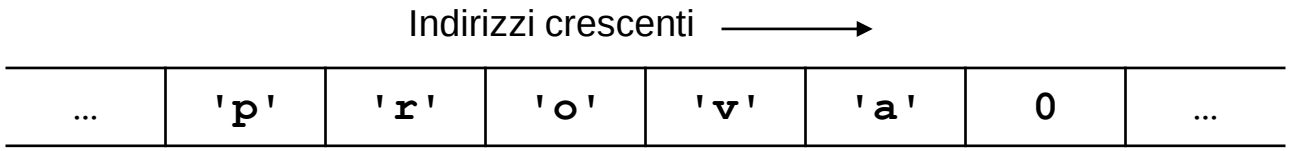
```
src[i] = 3.0;
```

  il compilatore segnalerebbe errore.

- **In generale se un parametro di tipo puntatore a qualcosa non deve modificare i dati a cui punta è buona norma dichiararlo const.**

Gli array di char zero terminati

- Durante la programmazione capita spesso di dover lavorare su «stringhe», ossia sequenze di caratteri.
- In C le stringhe non sono altro che array di char.
- Questi array hanno però una particolarità, sono «zero terminati», ossia utilizzano il valore 0 per marcare la fine della stringa. Il valore 0 è chiamato *terminatore*.
- Ad esempio la stringa «prova», per essere una stringa C, sarà composta da:



- **Le stringhe C NON sono «array speciali»!** In C non esiste niente del genere. Sono solamente array di char in cui il valore convenzionale per determinare la fine della stringa è il valore 0.
- Il terminatore è un char come tutti gli altri, anche se non può essere stampato, pertanto bisogna sempre prevedere anche lo spazio in memoria per inserirlo al termine della stringa.

Inizializzazione di stringhe

- In C è possibile inizializzare una stringa usando la seguente sintassi:

`char s[<N>] = "<contenuto-della-stringa>" ;`

- In questo caso bisogna fare attenzione ad allocare spazio sufficiente per contenere tutti i caratteri della stringa, **compreso il terminatore!**
- Ecco alcuni esempi:

```
char s1[3] = { 'a', 'b', 'c' };

char s2[6] = { 'p', 'r', 'o', 'v', 'a', 0 };


char s3[11] = "fondamenti";

char s4[] = "di";

char s5[15] = "informatica";

char s6[4] = "2016";
```

Array senza terminatore, valido ma non è una stringa C.

Array con terminatore, è una stringa C.

N è sufficiente per contenere tutti i caratteri compreso il terminatore.

Lascio decidere N al compilatore che allocherà spazio anche per il terminatore.

N è più grande del numero di caratteri più il terminatore, i successivi saranno tutti inizializzati a 0.

Errore: N è sufficiente per i caratteri ma non per il terminatore.

Le costanti di tipo array di char zero terminati

- Come per i numeri in C è possibile anche usare costanti di tipo stringa, che per il compilatore non sono altro che array di char zero terminati.
- Tutto ciò che nella slide precedente era compreso tra doppi apici (`"…"`) faceva parte di un *letterale* di tipo stringa, quindi di un letterale di tipo «array di char zero terminato».
- Di per sé, un letterale non è modificabile, ma quando il suo contenuto viene utilizzato per inizializzare un array di char allora abbiamo a disposizione una copia modificabile della stringa:

```
char s[] = "fondamenti";
```

- Un ulteriore caso possibile è il seguente:

```
char *s = "fondamenti";
```

- In questo caso ci accorgeremmo presto che il contenuto della stringa puntata da `s` non è modificabile. Abbiamo infatti inizializzato un puntatore a char con l'indirizzo di una *costante* di tipo stringa. Il puntatore `s` punterà quindi a una zona di memoria NON modificabile e un tentativo di farlo provocherà un crash del programma.

Letterali di tipo stringa

- Tutte le sequenze di escape dei caratteri sono utilizzabili all'interno dei letterali stringa.
- Ad esempio è possibile scrivere:

```
"prova\n"
```

- ovvero la scritta *prova* seguita da un *a capo*. Adesso però la domanda è: di che tipo è l'espressione `"prova\n"`?
- È di tipo `char[7]`, ovvero array di 7 char. Perché?

'p'	'r'	'o'	'v'	'a'	10	0

- 6 caratteri più uno zero a terminare la sequenza.
- Di che tipo è l'espressione: **`"!\xff\\!"`**
- È di tipo `char[5]`, ovvero array di 5 char.
- Di che tipo è l'espressione: **`""`**
- È di tipo `char[1]`, ovvero array di 1 char.

Letterali di tipo stringa

- Letterali di tipo stringa accostati senza nulla in mezzo vengono concatenati assieme in un'unica stringa:

```
"Ciao " "mi " "chiamo " "Costantino."
```

è come scrivere

```
"Ciao mi chiamo Costantino."
```

- Lo stesso vale per

```
"Ciao "
"mi "
"chiamo "
"Costantino."
```

- È invece un errore inserire degli a capo all'interno delle stringhe:

```
"Ciao mi
chiamo
Costantino."
```

Errore! Il compilatore non accetta stringhe con a capo all'interno.