UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dispense del corso di Fondamenti di Informatica 1

Il linguaggio C - La libreria string.h

Aggiornamento del 13/11/2020

Ancora sulle stringhe

- Abbiamo già visto che lavorare sulle stringhe in C può diventare fastidioso abbastanza in fretta.
- Dobbiamo sempre scrivere funzioni apposta per fare qualsiasi operazione su di esse e così il lavoro in più e la probabilità di fare errori aumentano notevolmente.
- Per fortuna, essendo le stringhe utilizzate continuamente nella programmazione, esiste già nella libreria standard del C un file con tutte le funzioni per fare operazioni di base sulle stringhe.
- Attraverso queste funzioni potremo evitare di riscrivere ogni volta quelle di base e usare quelle già presenti come punto di partenza per scriverne altre più complesse relative al programma che si sta scrivendo.

La libreria string.h

- Il file da includere per avere a disposizione tutte le funzioni per lavorare sulle stringhe è il file string.h:

```
#include <string.h>
```

- In esso sono contenute varie funzioni utili per esaminare una stringa, modificarla, copiarla, concatenarla e per cercare in essa.
- In string.h sono anche presenti alcune funzioni per lavorare con la memoria a livello di byte in modo più generico, vedremo più avanti i dettagli anche di queste funzioni.
- Di seguito non elencheremo tutte le funzioni presenti in string.h ma solo quelle usate più di frequente, per i dettagli di tutte si può sempre consultare la reference.

Lunghezza di una stringa

- La lunghezza di una stringa si può determinare attraverso la funzione **strlen()**, la cui dichiarazione è:

```
size_t strlen (const char *str);
```

- La funzione strlen() ha un solo parametro, il puntatore al primo carattere della stringa, e restituisce la lunghezza della stringa stessa (escluso il terminatore!).
- Attenzione: se `str` non è un puntatore a un array zero terminato si genera un undefined behavior.
- Ad esempio:

```
#include <stdlib.h>
#include <string.h>
int main(void) {
      char s[] = "questa e' una stringa";
      size_t len = strlen(s);

      return 0;
}
```

Dopo l'esecuzione `len` varrà 21, ossia la lunghezza della stringa senza contare il terminatore.

Confronto tra stringhe

- Fare un confronto tra stringhe in C si traduce nel fare un confronto tra array.
- Di conseguenza non si può fare usando un operatore (**l'operatore '==' non ci serve a niente in questo caso**) ma serve una funzione.
- Il confronto tra stringhe in realtà è un caso particolare del confronto tra array, infatti si stanno comparando array di char zero terminati.
- Nella libreria string.h esiste già una funzione per eseguire questo tipo di confronti, la funzione **strcmp()**.
- La sua dichiarazione è:

```
int strcmp(const char *lhs, const char *rhs);
```

- Ha due parametri, i puntatori ai primi caratteri delle due stringhe da confrontare, e ritorna un intero con segno che contiene il risultato del confronto.
- Anche in questo caso si genera un undefined behavior se `lhs` o `rhs` non sono puntatori ad array zero terminati.

strcmp()

- Il valore di ritorno di strcmp() ci dice qual è il risultato del confronto:
  - Se il valore è uguale a 0 significa che le stringhe sono uguali.
  - Se il valore è negativo la prima stringa viene prima in ordine lessicografico.
  - Se il valore è positivo la seconda stringa viene prima in ordine lessicografico.
- Con ordine lessicografico si può considerare quello che viene normalmente chiamato ordine alfabetico.
- Nel caso della strcmp() il confronto tra due caratteri viene fatto sulla base dell'ordine in cui essi appaiono nella tabella ASCII.
- Si può quindi dire che due caratteri vengono confrontati proprio sulla base del loro valore in memoria.

strcmp()

- Vediamo un esempio di applicazione di strcmp():

```
#include <string.h>

int main(void) {
      char s1[] = "prima stringa";
      char s2[] = "altra stringa";
      char s3[] = "altra stringa";

      int r1, r2, r3;
      r1 = strcmp(s1, s2);
      r2 = strcmp(s2, s3);
      r3 = strcmp(s2, s1);

      return 0;
}
```

L'intero `r1` varrà un valore positivo (solitamente 1) dato che `s2` viene prima di `s2`.

L'intero `r2` varrà 0 dato che le due stringhe `s2` e `s3` sono uguali.

L'intero `r3` varrà un valore negativo (solitamente -1) dato che `s2` viene prima di `s1`.

strncmp()

- Esiste anche una variante di strcmp() chiamata **strncmp()**.
- Questa funzione svolge esattamente lo stesso compito di strcmp() ma limita il confronto a un certo numero di caratteri.
- La sua dichiarazione è:

```
int strncmp(const char *lhs, const char *rhs, size_t count);
```

- Il parametro aggiuntivo `count` indica per quanti caratteri (e quindi per quanti byte) confrontare le due stringhe.
- in questa funzione si genera un undefined behavior quando la ricerca va oltre la fine di uno dei due array puntati da `lhs` o `rsh` (ossia se `count` è maggiore della lunghezza di una delle due stringhe).
- Il valore di ritorno di strncmp() contiene il risultato del confronto e funziona esattamente allo stesso modo di strcmp().

Funzioni di ricerca

- In string.h esistono anche alcune funzioni per cercare caratteri e sottostringhe in un'altra stringa.
- Le principali funzioni sono tre:

1. **strchr()**: cerca la prima occorrenza di un carattere.
2. **strrchr()**: cerca l'ultima occorrenza di un carattere.
3. **strstr()**: cerca la prima occorrenza di una stringa all'interno di un'altra.

strchr()

- La dichiarazione della funzione strchr() è la seguente:

```
char *strchr(const char *str, int ch);
```

- Il primo parametro è la stringa in cui effettuare la ricerca, il secondo parametro è il carattere da cercare (viene castato a unsigned char).
- La funzione ritorna un puntatore alla posizione della stringa in cui è stata trovata la prima occorrenza del carattere. Se il carattere non viene trovato la funzione ritorna `NULL`.
- Se `str` non è un array zero terminato si genera un undefined behavior.
- Ad esempio:

```
char s1[] = "prima stringa";
char *p = strchr(s1, 'i');
```

- In questo caso il puntatore `p` punterà al char che corrisponde alla «i» della parola «prima».

strrchr()

- La funzione strrchr() invece svolge lo stesso compito di strchr(), con la differenza che inizia la ricerca dalla fine della stringa.
- La sua dichiarazione è uguale a quella di strchr():

```
char *strrchr(const char *str, int ch);
```

- Ad esempio:

```
char s1[] = "prima stringa";
char *p = strrchr(s1, 'i');
```

- Questa volta il puntatore `p` punterà alla «i» della parola «stringa».

- Una particolarità di queste due funzioni è che si può anche cercare il terminatore.

strstr()

- La funzione strstr() serve invece per cercare una stringa all'interno di un'altra stringa.
- La sua dichiarazione è:

```
char *strstr(const char* str, const char* substr);
```

- Il primo parametro è la stringa in cui si effettua la ricerca, mentre il secondo è la stringa da cercare.
- Si genera un undefined behavior se uno dei due array non è zero terminato.

- Il valore di ritorno è un puntatore al primo carattere della sottostringa trovato in `str`.
- Se la sottostringa non viene trovata il puntatore ritornato è `NULL` e se `substr` è una stringa vuota viene ritornato `str` direttamente (il primo parametro).

strstr()

- Vediamo alcuni esempi:

```c
#include <string.h>

int main(void)
{
    char s[] = "uno due tre quattro";

    char *p, *q, *r;
    p = strstr(s, "due");
    q = strstr(s, "cinque");
    r = strstr(s, "");

    return 0;
}
```

`p` punterà alla «d» del «due» trovato nella stringa `s`.

`q` sarà un puntatore `NULL` perché la sottostringa «cinque» non è stata trovata nella stringa `s`.

L'indirizzo puntato da `r` sarà uguale a quello puntato da `s`, dato che è stata cercata una stringa vuota la funzione strstr() ha ritornato il primo parametro.

Funzioni per copiare e concatenare

- Nella libreria string.h sono presenti anche le funzioni per copiare e concatenare stringhe.
- A differenza delle funzioni viste in precedenza queste hanno bisogno di **copiare** dei byte da un array a un altro.
- Pertanto bisogna stare sempre attenti ad avere a disposizione spazio sufficiente in memoria dato che queste funzioni generano un undefined behavior quando lo spazio allocato non è grande abbastanza.

- Le funzioni principali sono due:
  - strcpy(): serve per copiare una stringa in un'altra.
  - strcat(): serve per concatenare due stringhe.

strcpy()

- La funzione **strcpy()** copia una stringa in un'altra, la sua dichiarazione è:

```
char *strcpy(char *dest, const char *src);
```

- Il primo parametro è la stringa di destinazione `dest` in cui copiare la stringa sorgente `src`.
- La funzione copia tutti i caratteri della stringa `src`, compreso il terminatore, nell'array puntato da `dest`.
- La funzione ritorna un puntatore che non è altro che una copia di `dest` (attenzione, una copia *del puntatore*, non della stringa!!).
- Vengono generati degli undefined behavior in vari casi (perciò bisogna fare particolare attenzione):
  - Se l'array `dest` non è grande abbastanza per contenere la stringa copiata.
  - Se le due stringhe si sovrappongono in memoria.
  - Se `src` non punta a un array zero terminato.

strcpy()

- In questo esempio vediamo i vari casi descritti in precedenza:

```
#include <string.h>

int main(void) {

      char s1[] = "stringa da copiare";
      char s2[] = "stringa da copiare molto più lunga della prima";

      char dest[25];

      strcpy(dest, s1);
      strcpy(dest, dest + 5);
      strcpy(dest, s2);

      return 0;
}
```

la stringa `s1` viene copiata in `dest` senza problemi perché `dest` è abbastanza grande per contenerla.

Undefined behavior! Sto provando a copiare la stringa che inizia a `dest+5` in `dest`. Le due stringhe si sovrappongono in memoria.

Undefined behavior! La stringa `s2` è troppo lunga per essere copiata in `dest`. Si scrive fuori da `dest`.

strcat()

- La funzione **strcat()** viene utilizzata per concatenare due stringhe. La sua dichiarazione è:

```
char *strcat(char *dest, const char *src);
```

- Il primo parametro è la stringa di destinazione `dest`, mentre la seconda stringa `src` è quella che viene appesa in coda a `dest`.
- La funzione copia tutti i caratteri di `src` (compreso il terminatore) in coda a `dest`. Il primo carattere di `src` (ossia `src[0]`) va a rimpiazzare il terminatore di `dest`.
- La funzione ritorna una copia del puntatore `dest`.
- Come per strcpy(), anche per strcat() si genera un undefined behavior in vari casi:
  - Se `dest` non è abbastanza grande da contenere `dest`, `src` e il terminatore.
  - Se le stringhe `dest` e `src` si sovrappongono in memoria.
  - Se `dest` o `src` non sono array zero terminati.

strcat()

- Ecco alcuni esempi dell'uso di strcat(). Supponiamo di usare un array di char di 28 byte chiamato `dest` per concatenare varie stringhe:

```
#include <string.h>

int main(void)
{
      char dest[28] = "Fondamenti";
      char s2[] = " di";
      char s3[] = " informatica";
      char s4[] = " 2016";

      strcat(dest, s2);
      strcat(dest, s3);
      strcat(dest, s4);

      return 0;
}
```

Provo a concatenare `dest` con `s2`. Tutto ok visto che in `dest` c'è abbastanza spazio. Bastano 14 byte per contenere «Fondamenti di» più il terminatore.

Provo a concatenare `dest` con `s3`. Ancora tutto ok visto che in `dest` c'è abbastanza spazio. Servono 26 byte per contenere «Fondamenti di informatica» più il terminatore.

Provo a concatenare `dest` con `s4`. Stavolta però si genera un undefined behavior dato che, per aggiungere anche « 2016» e il terminatore, `dest` dovrebbe essere lunga almeno 31 byte!

Le varianti con «n»

- Per le ultime due funzioni che abbiamo appena visto esiste una variante:

```
char *strncpy(char *dest, const char *src, size_t count);
char *strncat(char *dest, const char *src, size_t count);
```

- Queste funzioni hanno lo stesso comportamento di quelle già esaminate ma eseguono le loro operazioni (copiano o concatenano) al massimo su un certo numero di caratteri, indicati dal parametro `count`.

- Anch'esse ritornano una copia del puntatore `dest`.
- Tutti i casi in cui si genera un undefined behavior visti in precedenza sono ancora validi anche in queste due funzioni.

Le varianti con «n»

- ***Attenzione***: strncpy() è l'unica funzione ad avere un comportamento diverso dalle altre funzioni per le stringhe.
- Se durante la copia vengono raggiunti i `count` caratteri copiati l'array destinazione **non viene zero terminato!**
- Inoltre se il terminatore della stringa da copiare viene raggiunto prima di `count`, nei restanti byte vengono copiati degli zeri.
- La strncpy() esegue quindi sempre `count` scritture.

- strncat() invece aggiunge sempre alla fine della stringa concatenata il terminatore.
- strncat() quindi, se dopo `count` caratteri copiati non ha raggiunto il terminatore lo scrive lo stesso, per un totale di `count + 1` scritture.
- Un array di char su cui viene applicata la strncat() deve quindi essere grande almeno `strlen(dest) + count + 1`.

Altre funzioni di string.h

- Nella libreria string.h sono presenti anche altre funzioni che si occupano sempre di array di char ma non di stringhe C.
- Questo significa che lavorano su array di char ma non danno per scontato che questi siano zero terminati, si potrebbe pensare a queste funzioni come a funzioni che lavorano in modo più generico sulla memoria a livello di byte.
- Le funzioni sono molto simili a quelle per le stringhe ma non potendo contare sul terminatore per determinare la fine di una stringa avranno sempre la necessità di sapere la lunghezza in byte dell'array.
- Le funzionalità disponibili sono:
  - Riempimento di un array con un certo valore.
  - Ricerca di un certo byte in un array.
  - Confronto byte a byte tra array.
  - Copia di un array in un altro.
- A tutte le funzioni che vedremo, gli array verranno passati come puntatori a void. Come nel caso della malloc queste funzioni, lavorando a livello di byte non si preoccupano del tipo dei dati memorizzati negli array.

Riempimento di un array

- La funzione **memset()** ci permette, dato un array già allocato, di riempire tutti i suoi byte con un certo valore che possiamo specificare:

```
void *memset(void *dest, int ch, size_t count);
```

- Il parametro `dest` è il puntatore all'array che vogliamo riempire, `ch` è il valore con cui lo vogliamo riempire e `count` è il numero di byte che vogliamo riempire con il carattere `ch`.
- Il parametro `ch`, essendo un int, prima di essere utilizzato per riempire l'array viene castato a unsigned char.
- La funzione ritorna una copia del puntatore `dest`.
- La funzione genera un undefined behavior in due casi:
  - Se il puntatore `dest` è `NULL`.
  - Se `count` è maggiore della effettiva dimensione dell'array (quindi se proviamo a scrivere oltre la fine dell'array stesso).

memset()

- Vediamo alcuni esempi dell'uso di memset():

```
#include <string.h>

int main(void)
{
	char arr1[10];
	int arr2[5];

	memset(arr1, '8', 5);
	memset(arr1, 5, sizeof arr1);

	memset(arr2, 0x0a, sizeof arr2);
	memset(arr2, 0x0d, 5);

	memset(arr1, 'c', 20);

	return 0;
}
```

Riempio solo i primi 5 byte dell'array `arr1` con il valore 56.

Riempio tutti i byte dell'array `arr1` con il valore 5.

Riempio tutti e 20 i byte dell'array `arr2` con il valore `0x0a` (i cinque interi varranno quindi `0x0a0a0a0a`).

Riempio solo i primi 5 byte dell'array `arr2` con il valore `0x0d` (il primo intero varrà `0x0d0d0d0d`, il secondo `0x0a0a0a0d` e i successivi ancora `0x0a0a0a0a`).

Undefined behavior, provo a riempire 20 byte con il valore di 'c' ma `arr1` è grande solo 10 byte.

Ricerca in un array

- In string.h è disponibile anche una funzione per cercare un certo valore di byte all'interno di un array, la funzione **memchr()**:

```
void* memchr(const void* ptr, int ch, size_t count);
```

- Il parametro `ptr` è il puntatore all'array in cui cercare, `ch` è il valore di byte da cercare e `count` è il numero di byte per cui effettuare la ricerca.
- Il parametro `ch`, prima di essere cercato, viene convertito a unsigned char.
- Così come per strchr(), Il valore di ritorno è un puntatore che punta al byte che corrisponde alla prima occorrenza del valore di `ch`. Il puntatore ritornato è `NULL` se non è stato trovato `ch`.
- Si genera un undefined behavior se:
  - `ptr` è un puntatore `NULL`.
  - Se la funzione accede oltre la fine dell'array puntato da `ptr` (ossia se `count` è maggiore della dimensione dell'array).

memchr()

- Ecco alcuni esempi di utilizzo di memchr():

```c
#include <string.h>

int main(void)
{
    char arr1[5] = { 3, 4, 5, 6, 7 };

    char *p1, *p2, *p3;
    p1 = memchr(arr1, 6, sizeof arr1);
    p2 = memchr(arr1, 10, sizeof arr1);
    p3 = memchr(arr1, 4, 10);

    return 0;
}
```

`p1` punterà al quarto byte di `arr1` (in altre parole: `arr+3` o `arr[3]`).

`p2` sarà un puntatore `NULL` dato che non c'è nessun 10 nell'array `arr1`.

Il valore di `p3` non ci interessa dato che la terza chiamata a memchr() genera un undefined behavior. Stiamo provando a cercare nei 10 byte puntati da `arr1` ma `arr1` è grande solamente 5 byte!

Confronto di array

- La funzione **memcmp()** ci permette di confrontare byte per byte due array:

```
int memcmp(const void* lhs, const void* rhs, size_t count);
```

- I parametri `rhs` e `lhs` sono i due array da confrontare e `count` è il numero di byte per cui eseguire il confronto.
- Il valore di ritorno è un intero il cui significato è analogo a quello ritornato da strcmp(), viene valutato anche in questo caso l'ordine lessicografico.
- Si genera un undefined behavior se la funzione accede oltre la dimensione di uno dei due array esaminati, ossia se `count` è maggiore della dimensione di uno dei due array.

memcmp()

- Vediamo qualche esempio di utilizzo di memcmp():

```
#include <string.h>

int main(void) {
     char arr1[5] = { 0x0a, 0x0a, 0x0a, 0x0a, 0x0a };
     char arr2[5] = { 10, 10, 10, 10, 10 };
     char arr3[4] = { 0x0a, 0x0b, 0x0c, 0x0d };
     int arr4[3] = { 0x0a0a0a0a, 0xffffff0a, 0xffff };

     int r1, r2, r3, r4;
     r1 = memcmp(arr1, arr2, sizeof arr1);
     r2 = memcmp(arr1, arr3, 4);
     r3 = memcmp(arr1, arr4, sizeof arr1);
     r4 = memcmp(arr3, arr4, sizeof arr4);

     return 0;
}
```

- Quanto varranno dopo l'esecuzione `r1`, `r2`, `r3` e `r4`?

memcmp()

- `r1` varrà 0. Abbiamo infatti confrontato `sizeof arr1` byte (5 byte) tra gli array `arr1` e `arr2`. Entrambi gli array hanno 5 byte tutti dello stesso valore (`0x0a` e `10` sono lo stesso numero).
- `r2` varrà -1. Abbiamo confrontato 4 byte dagli array `arr1` e `arr3`. `arr1` contiene `0x0a` in tutti i suoi primi 4 byte mentre `arr2` contiene valori maggiori a partire dal secondo byte. Il risultato è che `arr1` precede `arr3` in ordine lessicografico.
- `r3` varrà 0. abbiamo confrontato `sizeof arr1` byte (5 byte) tra `arr1` e `arr4`. Anche se `arr4` è un array di int e non di char i suoi primi 5 byte sono comunque uguali a quelli di `arr1` (ricordarsi di cosa significa little-endian). Una volta che `arr1` e `arr4` vengono passati a memcmp() diventano puntatori a void e si perde l'informazione sul tipo originario dei due puntatori.
- Il valore di `r4` non ci importa granché dato che il quarto confronto genera un undefined behavior. `sizeof arr4` è 12 byte che è maggiore della dimensione di `arr3`. L'undefined behavior si genera perché accediamo oltre l'ultima cella di memoria allocata per `arr3`.

Copia di array

- In string.h c'è anche una funzione che ci permette di copiare il contenuto di un array in un altro, la funzione **memcpy()**:

```
void* memcpy(void *dest, const void *src, size_t count);
```

- Il parametro `dest` è il puntatore all'array in cui copiare i dati, `src` è il puntatore all'array da copiare e `count` è il numero di byte che devono essere copiati.
- La funzione ritorna una copia del puntatore `dest`.
- Bisogna fare particolare attenzione, prima di effettuare la copia bisogna essere sicuri che la memoria puntata da `dest` sia allocata e grande abbastanza per contenere `count` byte.
- Si genera un undefined behavior nei seguenti casi:
  - Se si accede oltre l'ultimo byte allocato per `dest` o `src`.
  - Se i due array si sovrappongono in memoria.
  - Se uno tra `dest` e `src` è un puntatore `NULL`.

memcpy()

- Vediamo alcuni esempi dell'uso della memcpy():

```c
#include <string.h>

int main(void)
{
	int arr1[5] = { 5, 10 };
	int arr2[5];
	char arr3[5] = { 3, 4, 5, 6, 7 };

	memcpy(arr2, arr1, sizeof arr1);
	memcpy(arr2, arr3, sizeof arr3);

	memcpy(arr3, arr3 + 1, 3);
	memcpy(arr3, arr1, sizeof arr1);

	return 0;
}
```

Copio 20 byte da `arr1` in `arr2`. Tutto bene, `arr2` è grande abbastanza.

Copio 5 byte da `arr3` in `arr2`. Anche in questo caso tutto ok, `arr2` è grande abbastanza.

Undefined behavior! Sto provando a copiare 3 byte di `arr3` su sé stesso. I due array così individuati si sovrappongono in memoria.

Undefined behavior, sto provando a copiare 20 byte da `arr1` a `arr3` ma `arr3` ne può contenere solamente 5.

memmove()

- In string.h esiste anche un'altra funzione per copiare array di byte, la funzione **memmove()**:

```
void* memmove(void* dest, const void* src, size_t count);
```

- La dichiarazione di memmove() è uguale a quella di memcpy() e anche i suoi parametri e il suo valore di ritorno hanno lo stesso significato.
- Ma qual è allora la differenza tra memcpy() e memmove()?
- La memmove() risolve il problema del copiare zone di memoria sovrapposte, che nella memcpy() generava un undefined behavior.
- In questo caso la copia tra i due array avviene come se i dati passassero prima per un array temporaneo.
- Un undefined behavior si genera comunque se:
  - Uno tra `src` o `dest` è un puntatore `NULL`.
  - Se la funzione accede oltre lo spazio allocato per uno dei due array (ossia se `count` è maggiore della dimensione di `dest` o `src`).