Capitolo 13

Stringhe

Introduzione

- Stringhe (due tipi):
  - stringhe *costanti* o *letterali,* per usare il gergo del C
  - stringhe *variabili*, che **possono cambiare** durante l'esecuzione del programma
- Le stringhe sono array di caratteri
  - Un carattere speciale—il carattere *nullo*—segnala la fine della stringa
- La libreria standard del C fornisce diverse funzioni per manipolare le stringhe

Stringhe costanti

- Una ***stringa costante*** è una sequenza di caratteri racchiusi fra doppi apici:

```
"When you come to a fork in the road, take it.”
```

- Possono contenere sequenze di escape

- Esempio

```
"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"
```

se stampata produce

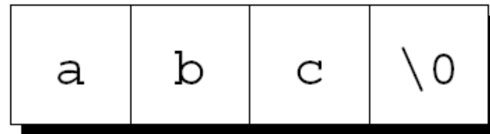
```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```

Memorizzazione delle stringhe costanti

- Quando il compilatore trova una stringa costante di lunghezza *n*, alloca *n* + 1 byte di memoria per la stringa
- Tale memoria conterrà gli *n* caratteri della stringa più un carattere aggiuntivo—il ***carattere nullo***—che ne segnala la fine
- Il carattere nullo ha codice ASCII 0 ed è rappresentato dalla sequenza di escape `\0`

Memorizzazione delle stringhe costanti

- La stringa letterale `"abc"` è memorizzata in un array di 4 caratteri:



- La stringa vuota è memorizzata in un array di 1 carattere:

Memorizzazione delle stringhe costanti

- Poiché una stringa è memorizzata in un array di caratteri, il compilatore la vede come `char *`
- Sia `printf` che `scanf` si aspettano valori di tipo `char *` per il primo argomento

- La seguente chiamata a `printf` passa l'indirizzo di `"abc"` (un puntatore alla locazione di memoria dove è memorizzato il carattere `a`):

```
printf("abc");
```

Operazioni sulle stringhe costanti

- Possiamo usare una stringa costante dovunque sia permesso usare un `char *`:

```
char *p;

p = "abc";
```

- Questo assegnamento **fa puntare** `p` **al primo carattere della stringa**

Operazioni sulle stringhe costanti

- È possibile usare un indice:

```
char ch;

ch = "abc"[1];
```

  Il valore di `ch` sarà il carattere `b`.

- Ecco una funzione che converte un numero fra 0 e 15 nell'equivalente cifra esadecimale:

```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Operazioni sulle stringhe costanti

- **Non è possibile** modificare una stringa costante:

```
char *p = "abc";

*p = 'd';    /*** WRONG ***/
```

- Un programma che cerca di cambiare una stringa costante può arrestarsi o avere comportamenti non definiti

Stringhe costanti e caratteri costanti

- Una stringa costante che contiene un solo carattere è diversa da una costante "carattere"
  - `"a"` è rappresentata da un *puntatore*
  - `'a'` è rappresentata da un *intero*

- Una chiamata corretta a `printf`:

```
printf("\n");
```

- Una chiamata sbagliata:

```
printf('\n');    /*** WRONG ***/
```

Stringhe variabili

- Un array monodimensionale di caratteri può essere usato per memorizzare una stringa
- ***Deve*** essere terminata dal carattere nullo
- Il C non ha un "tipo" stringa.
  - Le funzioni per le stringhe devono manipolare con attenzione il carattere (nullo) di fine stringa
  - Misurare la lunghezza di una stringa significa cercare il carattere nullo che ne indica la fine

Stringhe variabili

- Se una stringa può essere lunga 80 caratteri, deve essere dichiarata come array di 81 elementi:

```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

- La locazione aggiuntiva serve per il carattere di fine stringa \0
- Usare una macro che definisce la lunghezza della stringa e poi aggiungere il `+1` nella dichiarazione è una pratica comune

Stringhe variabili

- La dimensione dell'array ***non*** è la lunghezza della stringa
- La lunghezza della stringa dipende dalla posizione nell'array del carattere di fine stringa
- Un array di lunghezza `STR_LEN` + 1 può contenere stringhe di lunghezza fra 0 e `STR_LEN`

Inizializzazione di una stringa

- Una stringa variabile può essere inizializzata al momento della dichiarazione:

```
char date1[8] = "June 14";
```

- Il compilatore allocherà 8 byte per l'array ed inserirà il carattere di fine stringa

date1	J	u	n	e		1	4	\0

- In questo contesto, `"June 14"` ***non è*** una stringa costante ma un'abbreviazione dell'inizializzatore per l'array

Inizializzazione di una stringa

- Se l'inizializzatore ha meno caratteri della grandezza dell'array, il compilatore riempie il resto con caratteri nulli:

```
char date2[9] = "June 14";
```

La stringa `date2` sarà questa:

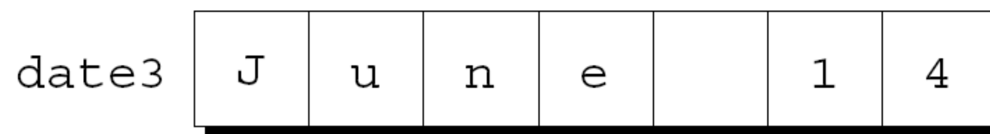
date2	J	u	n	e		1	4	\0	\0

Inizializzazione di una stringa

- L'inizializzatore non può essere più lungo della grandezza dell'array:

```
char date3[7] = "June 14";
```

- Non c'è spazio per il carattere di fine stringa quindi il compilatore ***non*** lo memorizzerà:



Inizializzazione di una stringa

- Si può omettere la lunghezza dell'array nella dichiarazione di una stringa variabile.
- Il compilatore calcolerà la dimensione necessaria:

```
char date4[] = "June 14";
```

- In questo caso il compilatore userà 8 caratteri per `date4`, sufficienti a memorizzare `"June 14"` ed il carattere nullo
- Omettere la lunghezza è utile in particolare quando la stringa è molto lunga

Array di caratteri e puntatori a caratteri

- La dichiarazione

  ```
  char date[] = "June 14";
  ```

  dichiara `date` come un *array*

- La dichiarazione

  ```
  char *date = "June 14";
  ```

  dichiara `date` come un *puntatore*

- Entrambe le variabili possono essere usate come stringhe

Array di caratteri e puntatori a caratteri

- Tuttavia ci sono differenze importanti fra le due versioni della variabile `date`.
  - Nella versione "array", i caratteri memorizzati nella variabile `date` **possono** essere modificati.
  - Nella versione "puntatore", `date` punta ad una stringa costante che **non può** essere modificata
  - Nella versione "array", `date` è il nome dell'array. Nella versione "puntatore", `date` è una variabile che **punta** ad una stringa

Array di caratteri e puntatori a caratteri

- La dichiarazione

  ```
  char *p;
  ```

  non alloca memoria per la stringa
- Prima di poter usare `p` come una stringa, dobbiamo far puntare `p` ad un array
- Per esempio:

  ```
  char str[STR_LEN+1], *p;

  p = str;
  ```

Array di caratteri e puntatori a caratteri

- Usare un puntatore non inizializzato può avere conseguenze disastrose
- Ecco un tentativo maldestro per la costruzione della stringa "`abc`":

```
char *p;

p[0] = 'a';      /*** WRONG ***/
p[1] = 'b';      /*** WRONG ***/
p[2] = 'c';      /*** WRONG ***/
p[3] = '\0';     /*** WRONG ***/
```

- Poichè la variabile `p` non è stata inizializzata il comportamento del programma non è definito

Leggere e scrivere stringhe

- Una stringa può essere stampata usando `printf` o `puts`.
- Leggere una stringa è un po' più complicato perchè l'input potrebbe essere più lungo dell'array in cui la stringa verrà memorizzata
- Per leggere una stringa intera possiamo usare `scanf` o `gets`.
- Oppure possiamo leggere un carattere alla volta

Stringhe con `printf` e `puts`

- La specifica di conversione `%s` permette di usare una stringa con `printf` :

```
char str[] = "Are we having fun yet?";

printf("%s\n", str);
```

L'output sarà

```
Are we having fun yet?
```

- `printf` scrive i carattere di una stringa uno alla volta fino a che non trova il carattere nullo

Stringhe con `printf` e `puts`

- Per stampare una parte di una stringa si può usare la specifica di conversione `%.`*p*`s`.

- *p* è il numero di caratteri da visualizzare

- L'istruzione

  ```
  printf("%.6s\n", str);
  ```

  stamperà

  ```
  Are we
  ```

Stringhe con `printf` e `puts`

- Invece `%`*m*`s` stamperà la stringa in un "campo" di larghezza *m*.
- Se la stringa ha meno di *m* caratteri verrà allineata a destra
- Per forzare l'allineamento a sinistra possiamo usare il segno meno davanti ad *m*
- Le due specifiche di conversione possono essere usate insieme
- `%`*m*`.`*p*`s` stampa i primi *p* caratteri in un campo di grandezza *m*

Stringhe con **`printf`** e **`puts`**

- Oltre a `printf` si può usare la funzione `puts` per stampare una stringa:

  ```
  puts(str);
  ```

- Dopo aver scritto la stringa `puts` **aggiunge** sempre il carattere di new-line

Stringhe con `scanf` e `gets`

- Lo specificatore di conversione `%s` dice a `scanf` di leggere dall'input una stringa e di memorizzarla in **un array**:

```
scanf("%s", str);
```

- `str` viene interpretato come un puntatore quindi non c'è bisogno di `&` prima `str`
- La funzione `scanf` salta i caratteri "bianchi", dopodichè memorizza tutti i successivi caratteri nell'array `str` **fino al prossimo carattere bianco**
- `scanf` aggiunge sempre il carattere di fine stringa

Stringhe con `scanf` e `gets`

- Per leggere **un'intera linea** di input possiamo usare `gets`
- Proprietà di `gets`:
  - **Non salta** gli spazi ed i tab iniziali
  - Legge l'input fino al new-line
  - Il carattere di new-line stesso viene letto dall'input ma viene **eliminato**; al suo posto viene inserito il carattere nullo per terminare la stringa

Stringhe con `scanf` e `gets`

- Esempio:

```
char sentence[SENT_LEN+1];

printf("Enter a sentence:\n");
scanf("%s", sentence);
```

- Supponiamo che dopo il messaggio

```
Enter a sentence:
```

l'utente digiti

```
To C, or not to C: that is the question.
```

- `scanf` memorizzerà `"To"` nella variabile `sentence`

Stringhe con `scanf` e `gets`

- Usiamo ora `gets` al posto di `scanf`

```
gets(sentence);
```

- Quando l'utente inserisce la stessa stringa di prima, `gets` memorizza la stringa

  `" To C, or not to C: that is the question."`

  nella variabile `sentence`

Stringhe con `scanf` e `gets`

- `scanf` e `gets` non hanno modo di controllare se c'è spazio sufficiente nell'array
- Di conseguenza, se l'input è lungo potrebbero scrivere oltre la fine dell'array
  - comportamento non definito
- `scanf` può essere resa "sicura" specificando la lunghezza dell'array nello specificatore usando `%`*n*`s` invece di `%s`.
  - *n* è un intero che specifica il **numero massimo** di caratteri che possono essere memorizzati
- `gets` non può essere resa sicura; `fgets` è un'alternativa migliore

Leggere carattere per carattere

- A volte è necessario scrivere una “propria” funzione di input
- Occorre decidere
  - Cosa fare dello spazio “bianco” iniziale
  - Quando fermarsi nella lettura dell’input:
    - Carattere di new-line, spazio bianco, qualche altro carattere particolare? E poi, il carattere che causa lo stop, deve essere inserito nella stringa?
  - Cosa deve fare la funzione se l’input è troppo lungo
    - Eliminare i caratteri in eccesso o lasciarli per la prossima chiamata?

Leggere carattere per carattere

- Supponiamo che ci serva una funzione che
  - (1) non salti lo spazio bianco
  - (2) si fermi al primo new-line (che non viene messo nella stringa)
  - (3) elimini caratteri in eccesso
- Un possibile prototipo della funzione è

```
int read_line(char str[], int n);
```

- Se la linea contiene più di `n` caratteri, `read_line` eliminerà eventuali caratteri aggiuntivi
- `read_line` restituisce il numero di caratteri che memorizza nella stringa `str`

Leggere carattere per carattere

- `read_line` usa la funzione `getchar` per leggere caratteri e memorizzarli in `str`, ammesso che ci sia ancora spazio disponibile

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';    /* terminates string */
  return i;         /* number of characters stored */
}
```

- `ch` è di tipo `int` (nessun problema, `getchar` restituisce valore di tipo `int`)

Leggere carattere per carattere

- Dopo aver letto l'input `read_line` inserisce il carattere nullo per terminare la stringa

- Le funzioni della libreria standard, come `scanf` e `gets` automaticamente inseriscono il carattere nullo per terminare la stringa

Accedere ai caratteri di una stringa

- Poichè le stringhe sono memorizzate in degli array è possibile accedere ai singoli caratteri usando l'**indice**

- Per accedere a tutti i caratteri di una stringa `s` singolarmente, possiamo usare un ciclo che sfrutta un contatore `i` e seleziona i singoli caratteri con l'espressione `s[i]`

Accedere ai caratteri di una stringa

- Ecco una funzione che conta il numero di spazi in una stringa:

```
int count_spaces(const char s[])
{
  int count = 0, i;

  for (i = 0; s[i] != '\0'; i++)
    if (s[i] == ' ')
      count++;
  return count;
}
```

Libreria standard: funzioni per stringhe

- Alcuni linguaggi di programmazione forniscono operatori che possono copiare, confrontare, concatenare stringhe ed altro ancora
- Gli operatori del C sono invece **inutili** per operare sulle stringhe
- In particolare **non è** possibile copiare o confrontare delle stringhe

Libreria standard: funzioni per stringhe

- La libreria C fornisce un ricco insieme di **funzioni per operare sulle stringhe**
- I programmi che vogliono utilizzare le funzioni della libreria devono includere il file `string.h`:

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

- Negli esempi che seguono assumeremo che `str1` e `str2` siano array di caratteri usati come stringhe

La funzione `strcpy` (string copy)

- Prototipo:

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

- `strcpy` copia la stringa `s2` nella stringa `s1`
  - Per essere precisi dovremmo dire "`strcpy` copia la stringa puntata da `s2` nell'array puntato da `s1`."

- `strcpy` restituisce `s1` (un puntatore alla stringa di destinazione)

La funzione **`strcpy`** (string copy)

- Una chiamata a `strcpy` che memorizza `"abcd"` in `str2`:

```
strcpy(str2, "abcd");
  /* str2 now contains "abcd" */
```

- Una chiamata che copia il contenuto di `str2` in `str1`:

```
strcpy(str1, str2);
  /* str1 now contains "abcd" */
```

La funzione `strcpy` (string copy)

- Nella chiamata `strcpy(str1, str2)`, `strcpy` non ha modo di controllare che ci sia abbastanza spazio per copiare `str2` nell'array puntato da `str1`
- Se non c'è abbastanza spazio il comportamento del programma non è definito

La funzione `strncpy` (string copy sicura)

- Usare la funzione `strncpy` è più sicuro, anche se più lento
- `strncpy` ha un terzo argomento che specifica la grandezza dell'array di destinazione
- Una chiamata `strncpy` che copia `str2` in `str1`:

```
strncpy(str1, str2, sizeof(str1));
```

La funzione `strncpy` (string copy sicura)

- `strncpy` non scriverà il carattere nullo alla fine di `str1` se la lunghezza di `str2` è maggiore della lunghezza dell'array `str1`
- Ecco come ovviare al problema:

```
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

- L'istruzione dopo la chiamata alla funzione scrive in modo esplicito il carattere nullo nell'ultima posizione dell'array `str`

La funzione `strlen` (string length)

- Prototipo:

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

- `size_t` è un nome `typedef` che rappresenta un intero senza segno

La funzione `strlen` (string length)

- `strlen` restituisce la lunghezza della stringa `s`, **senza contare** il carattere nullo di fine stringa

- Esempi:

```
int len;

len = strlen("abc");   /* len is now 3 */
len = strlen("");      /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);    /* len is now 3 */
```

La funzione `strcat` (concatenation)

- Prototipo:

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

- `strcat` aggiunge il contenuto della stringa `s2` alla (fine della) stringa `s1`.
- Restituisce il valore di `s1` (il puntatore alla stringa risultato della concatenazione).
- Esempi:

```
strcpy(str1, "abc");
strcat(str1, "def");
  /* str1 now contains "abcdef" */
strcpy(str1, "abc");
strcpy(str2, "def");
strcat(str1, str2);
  /* str1 now contains "abcdef" */
```

La funzione **`strcat`** (concatenation)

- Il comportamento del programma non è definito se `strcat(str1, str2)` non ha abbastanza spazio in `str1` per inserire anche i caratteri di `str2`.
- Esempio:

```
char str1[6] = "abc";

strcat(str1, "def");    /*** WRONG ***/
```

- `str1` può contenere al massimo 6 caratteri, ma ne servono 7 e quindi `strcat` scriverà oltre la fine dell'array

La funzione `strcat` (concatenation)

- La funzione `strncat` è più sicura, anche se più lenta
- Come `strncpy`, ha un terzo parametro che specifica il numero massimo di caratteri che può copiare
- Ecco una chiamata a `strncat`:

```
strncat(str1, str2, sizeof(str1) - strlen(str1) - 1);
```

- `strncat` terminerà `str1` con un carattere nullo, che dobbiamo considerare quando calcoliamo il terzo argomento

La funzione `strcmp` (string comparison)

- `strcmp` considera **l'ordine lessicografico**: `s1` è minore di `s2` se una delle seguenti condizioni è soddisfatta:
  - I primi *i* caratteri di `s1` e `s2` sono uguali, ma il carattere (*i*+1)-esimo di `s1` è minore del carattere (*i*+1)-esimo di `s2`.
  - Tutti i caratterei di `s1` sono uguali a quelli di `s2`, ma `s1` è più corta di `s2`.

La funzione `strcmp` (string comparison)

- La relazione di ‘<’ fra caratteri è determinta dal codice ASCII del carattere
- Nella codifica ASCII:
  - A–Z, a–z, e 0–9 hanno codici consecutivi
    - ‘A’=65, ‘Z’=91
    - ‘a’=97 ‘z’=123
    - ‘0’=48, ‘9’=57
  - Le lettere maiuscole sono ‘<’ delle lettere minuscole
  - Le cifre sono ‘<’ delle lettere
  - Lo spazio (codice 32) è ‘<’ di tutti i caratteri stampabili

- `man ascii`

La funzione **strcmp** (string comparison)

- Prototipo:

```
int strcmp(const char *s1, const char *s2);
```

- `strcmp` confronta la stringa `s1` con `s2`, e restituisce un valore
  - minore di 0 se `s1` è (lessicograficamente) minore di `s2`
  - uguale a 0, se `s1` è uguale a `s2`
  - maggiore di 0, se `s1` è (lessicograficamente) maggiore di `s2`

La funzione **strcmp** (string comparison)

- Controllare se `str1` è minore di `str2`:

```
if (strcmp(str1, str2) < 0)    /* is str1 < str2? */
  …
```

- Controllare se `str1` è minore o uguale a `str2`:

```
if (strcmp(str1, str2) <= 0) /* is str1 <= str2? */
  …
```

- Usando il giusto operatore (<, <=, >, >=, ==, !=), possiamo controllare tutte le relazioni fra `str1` e `str2`.

Programma: stampare un promemoria

- Il programma `remind.c` stampa un promemoria di un mese
- L’utente può inserire una serie di promemoria, ognuno preceduto dal giorno del mese
- Quando l’utente inserisce 0 al posto di un giorno valido, il programma stampa la lista ordinandola per giorno
- La prossima slide mostra un esempio di utilizzo del programma

Programma: stampare un promemoria

```
Enter day and reminder: 24 Susan's birthday
Enter day and reminder: 5 6:00 - Dinner with Marge and Russ
Enter day and reminder: 26 Movie - "Chinatown"
Enter day and reminder: 7 10:30 - Dental appointment
Enter day and reminder: 12 Movie - "Dazed and Confused"
Enter day and reminder: 5 Saturday class
Enter day and reminder: 12 Saturday class
Enter day and reminder: 0

Day Reminder
  5 Saturday class
  5 6:00 - Dinner with Marge and Russ
  7 10:30 - Dental appointment
 12 Saturday class
 12 Movie - "Dazed and Confused“
 24 Susan's birthday
 26 Movie - "Chinatown"
```

Programma: stampare un promemoria

- Strategia:
  - Leggere una serie di linee, ognuna contenente un giorno ed un promemoria
  - Memorizzare le linee in ordine (in base al giorno)
  - Stamparle a video
- `scanf` verrà usata per leggere “il giorno”
- `read_line` verrà usata per leggere “il resto”

Programma: stampare un promemoria

- Le linee verranno memorizzate in un array di caratteri bidimensionale
- Ogni riga contiene una linea (stringa)
- Le azioni effettuate per ogni linea letta sono:
  - Ricerca della **posizione** nell'array in cui inserire la linea (promemoria), usando `strcmp`
  - Uso di `strcpy` per **spostare** tutte le linee di una posizione in avanti per "fare spazio" alla nuova linea
  - **Copia** della nuova linea nella posizione giusta; chiamata a `strcat` per aggiungere il promemoria al giorno

Programma: stampare un promemoria

- Useremo `scanf` per leggere il giorno in una variabile di tipo `int` e poi `sprintf` per riconvertirla in una stringa
- `sprintf` è simile a `printf`, ma al posto di stampare scrive il risultato in una stringa.
- Precisamente:

```
sprintf(day_str, "%2d", day);
```

scrive il valore di `day` in `day_str`, e

```
scanf("%2d", &day);
```

assicura che non ci siano più di 2 cifre nel giorno dato in input

remind.c

```
/* Prints a one-month reminder list */

#include <stdio.h>
#include <string.h>

#define MAX_REMIND 50   /* maximum number of reminders */
#define MSG_LEN 60     /* max length of reminder message */

int read_line(char str[], int n);

int main(void)
{
  char reminders[MAX_REMIND][MSG_LEN+3];
  char day_str[3], msg_str[MSG_LEN+1];
  int day, i, j, num_remind = 0;

  for (;;) {
    if (num_remind == MAX_REMIND) {
      printf("-- No space left --\n");
      break;
    }
```

```
    printf("Enter day and reminder: ");
    scanf("%2d", &day);
    if (day == 0)
      break;
    sprintf(day_str, "%2d", day);
    read_line(msg_str, MSG_LEN);

    for (i = 0; i < num_remind; i++)
      if (strcmp(day_str, reminders[i]) < 0)
        break;
    for (j = num_remind; j > i; j--)
      strcpy(reminders[j], reminders[j-1]);

    strcpy(reminders[i], day_str);
    strcat(reminders[i], msg_str);

    num_remind++;
  }

  printf("\nDay Reminder\n");
  for (i = 0; i < num_remind; i++)
    printf(" %s\n", reminders[i]);

  return 0;
}
```

```
int read_line(char str[], int n)
{
  int ch, i = 0;

  while ((ch = getchar()) != '\n')
    if (i < n)
      str[i++] = ch;
  str[i] = '\0';
  return i;
}
```

Nota: all’inserimento del primo promemoria, i due cicli `for()` interni non vengono eseguiti, essendo `num_remind=0`.

Pertanto, il promemoria viene costruito con `strcpy` e `strcat` e posto in `reminders[0]`;

Array di stringhe

- Esistono vari modi per memorizzare un array di stringhe.
- Una possibilità è quella di usare un array bi-dimensionale, e memorizzare una stringa in ogni riga

```
char planets[][8] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
```

- Il numero di righe nell'array può essere omesso, ma occorre specificare il numero di colonne

Array di stringhe

- L'array `planets` contiene vari byte non utilizzati:

	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

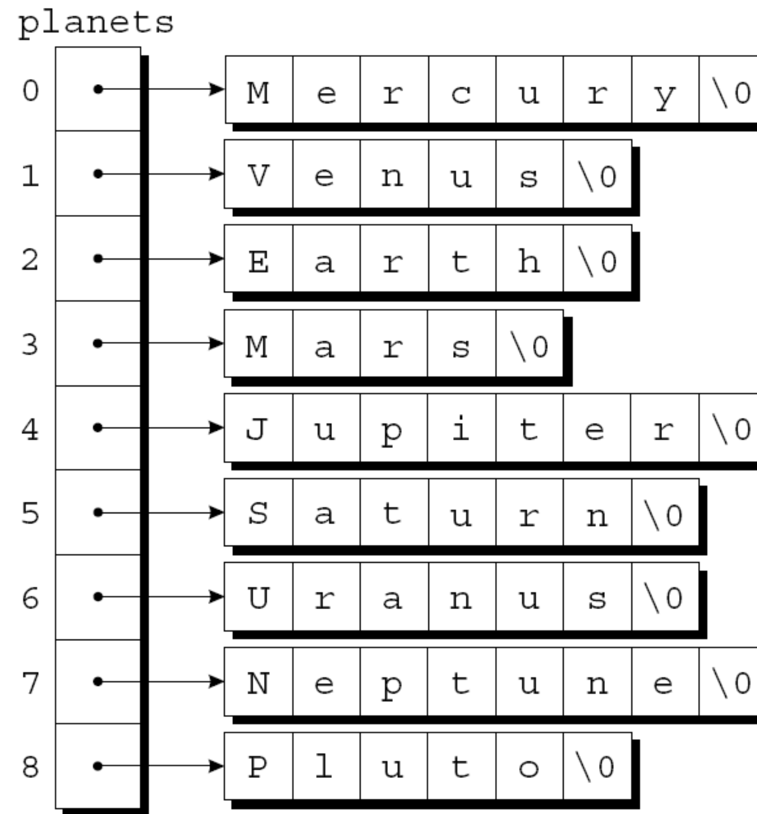
Array di stringhe

- Spesso quando dobbiamo gestire molte stringhe ci troveremo in questa situazione:
  - Alcune stringhe più lunghe altre più corte
- Potremmo usare un ***array irregolare (ragged array)*** in cui le righe possono avere lunghezze diverse
- Possiamo "simulare" un tale array utilizzando un array di *puntatori* alla stringhe:

```
char *planets[] = {"Mercury", "Venus", "Earth",
                   "Mars", "Jupiter", "Saturn",
                   "Uranus", "Neptune", "Pluto"};
```

Array di stringhe

- Ecco come verranno memorizzate le stringhe `planets`:



Array di stringhe

- Per accedere ad una delle stringhe dobbiamo usare l'indice per selezionare il puntatore dell'array `planets`
- Dopodiché possiamo accedere ai caratteri della stringa come si fa con un normale array bidimensionale
- Ecco un ciclo che cerca le stringhe che iniziano per M

```
for (i = 0; i < 9; i++)
   if (planets[i][0] == 'M')
      printf("%s begins with M\n", planets[i]);
```

Argomenti sulla linea di comando

- Quando eseguiamo un programma, spesso abbiamo la necessità di fornire al programma delle informazioni
- Ad esempio il nome di un file oppure un *flag* che dice al programma come comportarsi
- Esempio:

```
ls
ls -l
ls -l remind.c
```

Argomenti sulla linea di comando

- Gli argomenti passati sulla linea di comandi sono accessibili da tutti i programmi C
- Per accedere a tali argomenti il `main` deve avere due parametri:
-
```
int main(int argc, char *argv[])
{
   …
}
```
- Gli argomenti passati sulla linea di comando sono detti ***parametri del programma***

Argomenti sulla linea di comando

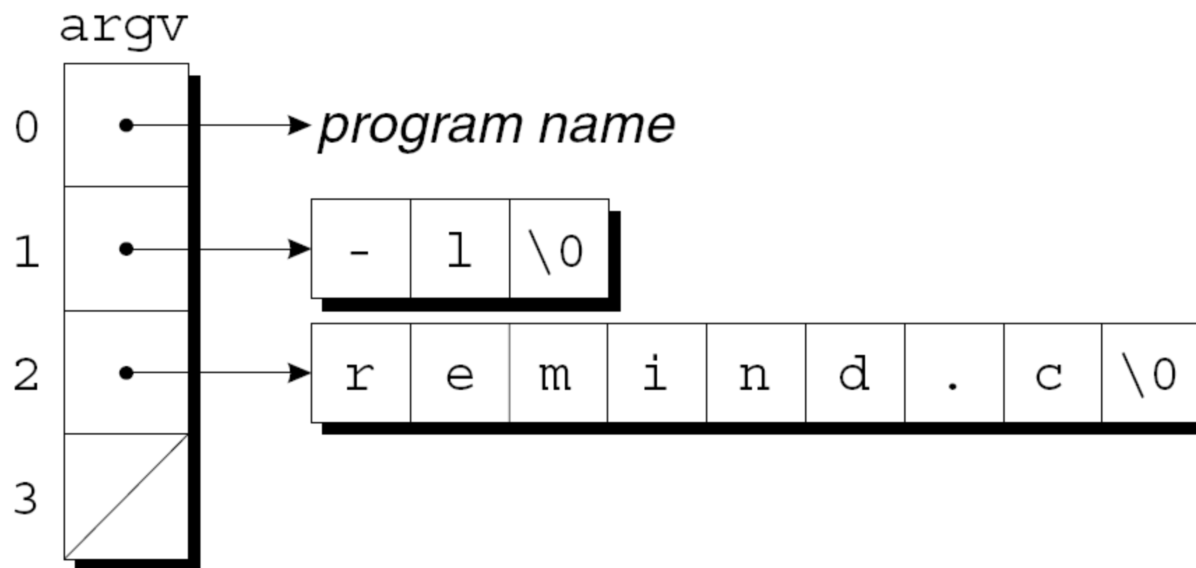
- Il parametro `argc` (che sta per "argument count") specifica il numero di parametri
- Il parametro `argv` ("argument vector") è un array di puntatori a stringhe, ognuna delle quali contiene un parametro del programma
- `argv[0]` punta al ***nome del programma***
- I puntatori da `argv[1]` a `argv[argc-1]` puntano ai parametri del programma
- `argv[argc]` è un ***puntatore nullo***
  - La macro `NULL` rappresenta il valore di un puntatore nullo

Argomenti sulla linea di comando

- Se l'utente digita la linea di comando

  `ls -l remind.c`

  allora `argc` vale 3, mentre `argv è:`



Argomenti sulla linea di comando

- Poiché `argv` è un array di puntatori accedere ai parametri del programma è semplice
- Tipicamente un programma che si aspetta di ricevere dei parametri utilizza un ciclo per esaminare i parametri passati
- Un modo per scrivere un tale ciclo è il seguente:

```
int i;

for (i = 1; i < argc; i++)
  printf("%s\n", argv[i]);
```

Argomenti sulla linea di comando

- Un’altra tecnica (in cui potreste imbattervi) è quella di usare un puntatore a `argv[1]`, e quindi incrementare il puntatore ripetutamente:

```
char **p;
```

(…non è un errore, puntatore a puntatore!)

```
for (p = &argv[1]; *p != NULL; p++)
  printf("%s\n", *p);
```

(…uso avanzato dei puntatori, solo per info, non approfondiremo)

Programma: controllo dei nomi dei pianeti

- Il programma `planet.c` illustra l'accesso ai parametri del programma
- Il programma controlla se i parametri passati come argomenti sono nomi di pianeti

```
planet Jupiter venus Earth fred
```

- Per ognuno dei parametri il programma dirà se è il nome di un pianeta (e ne specificherà il numero) oppure no:

```
Jupiter is planet 5
venus is not a planet
Earth is planet 3
fred is not a planet
```

planet.c

```
/* Checks planet names */

#include <stdio.h>
#include <string.h>

#define NUM_PLANETS 9

int main(int argc, char *argv[])
{
  char *planets[] = {"Mercury", "Venus", "Earth",
                     "Mars", "Jupiter", "Saturn",
                     "Uranus", "Neptune", "Pluto"};
  int i, j;
```

```
  for (i = 1; i < argc; i++) {
    for (j = 0; j < NUM_PLANETS; j++)
      if (strcmp(argv[i], planets[j]) == 0) {
        printf("%s is planet %d\n", argv[i], j + 1);
        break;
      }
    if (j == NUM_PLANETS)
      printf("%s is not a planet\n", argv[i]);
  }

  return 0;
}
```

Obiettivo principale del corso

- Programmazione in linguaggio C

1. Fondamenti di C ✓
2. Input/Output ✓
3. Espressioni ✓
4. Istruzioni cond. ✓
5. Cicli ✓
6. Tipi di base ✓
7. Array ✓
8. Funzioni ✓
9. Puntatori (light) ✓
10. Stringhe ✓
11. Strutture
12. Cenni I/O e gestione dei file