Capitolo 22

Input/Output

Introduzione

- La libreria di input/output è la parte più importante della libreria standard del C
- L'intestazione `<stdio.h>` contiene le principali funzioni di input/output (e.g., `printf`, `scanf`, `putchar`, `getchar`, `puts`, `gets` …)
- Oggi cominceremo a trattare anche altre funzioni

Flussi (stream)

- In C il termine ***stream*** indica una *sorgente* di input o una *destinazione* per l'output
- Molti programmi (piccoli) ottengono il loro input da uno stream (ad es. la tastiera) e lo inviano ad un altro stream (ad esempio il video)
- Programmi più grandi possono avere necessità di usare più stream
- Gli stream spesso rappresentano file memorizzati da qualche parte (hard disk o altri tipi di memoria a lungo termine); in altri casi sono associati a periferiche (schede di rete, stampanti, etc)

Puntatori a file

- Per accedere ad uno stream, si usa un ***puntatore a file,*** il cui tipo è `FILE *`.
- Il tipo `FILE` è dichiarato in `<stdio.h>`
- Alcuni stream sono rappresentati da puntatori a file con dei nomi standard (es. `stdin`)
- Altri puntatori possono essere dichiarati in base alle necessità:

```
FILE *fp1, *fp2;
```

Flussi standard e redirezione

- `<stdio.h>` fornisce 3 flussi standard:

Nome (*FILE)	Flusso	Periferica di default
`stdin`	Standard input	Tastiera
`stdout`	Standard output	Video
`stderr`	Standard error	Video

- Questi flussi sono pronti per essere usati—non c'è bisogno di dichiararli e non c'è bisogno di aprirli o chiuderli

Flussi standard e redirezione

- Le funzioni di I/O viste nei capitoli precedenti ottengono l'input da `stdin` e spediscono l'output su `stdout`
- Molti sistemi operativi permettono di cambiare la periferica di default associata ai flussi standard con un meccanismo detto di ***redirezione (deviazione)***

Flussi standard e redirezione

- Un esempio di ***redirezione dell'input***:

```
demo < in.dat
```

l'input viene ottenuto dal file `in.dat` e non dalla tastiera:

- Un esempio di ***redirezione dell'output***:

```
demo > out.dat
```

l'output viene scritto nel file `out.dat` e non visualizzato a video

Flussi standard e redirezione

- La redirezione dell'input e dell'output possono essere usate insieme:

```
demo < in.dat > out.dat
```

- L'ordine non è importante. I seguenti comandi sono equivalenti

```
demo < in.dat > out.dat
demo > out.dat < in.dat
```

Flussi standard e redirezione

- Un problema con la redirezione dell'output è che viene rediretto *tutto l'output* nel file
- Per questo motivo è buona abitudine scrivere i messaggi di errore su `stderr` invece che su `stdout`
  - quando si redireziona l'output i messaggi di errore continuano ad apparire sul video
  - vedremo come fare per scrivere su `stderr`

File di testo e file binari

- In generale un file è una sequenza di byte
- I byte in un ***file di testo*** rappresentano caratteri (le lettere, in numeri, la punteggiatura, etc) e quindi possono essere stampati, letti e modificati con un "text editor"
  - Il codice sorgente di un programma C è memorizzato in un file di testo
- In un ***file binario,*** i byte possono assumere qualsiasi valore
  - Gruppi di byte potrebbero rappresentare altri tipi di dati, come interi, numeri in virgola mobile, strutture
  - Un programma compilato è memorizzato in un file binario

File di testo e file binari

- I file di testo hanno due caratteristiche che i file binari non hanno.
- ***I file di testo possono essere divisi in linee.*** Ogni linea finisce con uno o due caratteri speciali
  - Windows: usa due caratteri, il carattere carriage-return (`'\x0d'`) seguito dal carattere line-feed (`'\x0a'`)
  - UNIX e le nuove versioni di Mac OS: usano solo il carattere line-feed (newline)
  - Le vecchie versioni del Mac OS: solo il carattere carriage-return

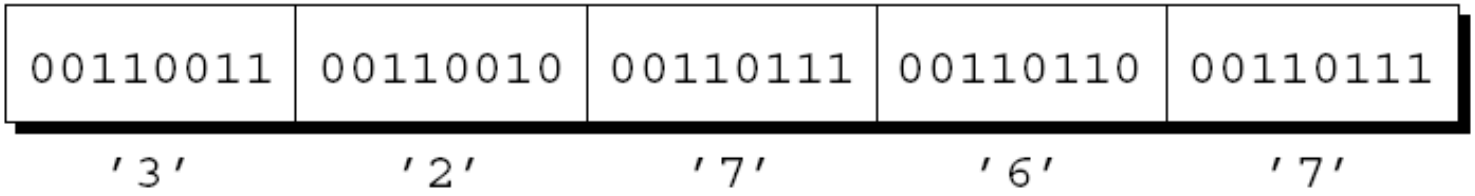
File di testo e file binari

- ***I file di testo possono avere un carattere speciale di "fine file"***
  - In Windows, questo carattere è `'\x1a'` (Ctrl-Z), ma non è strettamente necessario inserirlo
  - Molti altri sistemi operativi, incluso UNIX, **non** prevedono un carattere speciale alla fine del file

- In un file binario non ci sono né linee né caratteri di fine file. Tutti i byte sono trattati allo stesso modo.

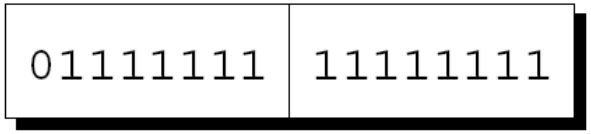
File di testo e file binari

- Quando i dati vengono scritti in un file, possono essere memorizzati in forma testuale oppure in forma binaria
- Un modo per rappresentare il numero 32767 in un file è quello di scrivere i caratteri 3, 2, 7, 6, e 7:



File di testo e file binari

- L'altra possibilità è quella di memorizzare direttamente la forma binaria, che necessiterebbe solo di 2 byte:



- Memorizzare i numeri in formato binario è più efficiente in termini di spazio utilizzato

File di testo e file binari

- I programmi che leggono o scrivono un file **devono sapere** quale formato utilizzare

Apertura di un file

- Per aprire un file ed usarlo come stream possiamo usare la funzione `fopen`
- Prototipo:

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

- `filename` è il nome del file da aprire
  - Può essere il nome completo o relativo
- `mode` è una stringa che specifica il tipo di operazioni che si intende fare
  - es. lettura, scrittura, scrittura in modalità “append”

Apertura di un file

- La parola chiave `restrict` compare due volte nel prototipo di `fopen`
- `restrict` fa parte del C99
- Il prototipo C89 di `fopen` non specifica `restrict`
- Non ha effetto sul comportamento di `fopen`, quindi può (solitamente) essere ignorata.

Apertura di un file

- `fopen` restituisce un puntatore al file che normalmente viene memorizzato in una variabile per poter accedere al file successivamente:

```
fp = fopen("in.dat", "r");
  /* opens in.dat for reading */
```

- Se non riesce ad aprire il file, allora `fopen` restituisce il puntatore nullo

Modalità d'apertura

- Per decidere la modalità in cui si vuole aprire un file con `fopen` occorre stabilire:
  - Quali operazioni devono essere effettuate
  - Se il file contiene testo o dati binari

Modalità d'apertura

- Modalità per i file di testo:

Stringa	***Significato***
`"r"`	Lettura
`"w"`	Scrittura
`"a"`	Scrittura in modalità "append"
`"r+"`	Lettura e scrittura
`"w+"`	Lettura e scrittura (azzera il file se esiste)
`"a+"`	Lettura e scrittura (accoda la scrittura se il file esiste)

Le operazioni di scrittura creano il file se il file non esiste.

Modalità d'apertura

- Modalità di apertura per file binari:

Stringa	***Significato***
`"rb"`	Lettura
`"wb"`	Scrittura
`"ab"`	Scrittura in modalità append
`"r+b"` or `"rb+"`	Lettura e scrittura
`"w+b"` or `"wb+"`	Lettura e scrittura (azzera il file se esiste)
`"a+b"` or `"ab+"`	Lettura e scrittura (accoda la scrittura se il file esiste)

Le operazioni di scrittura creano il file se il file non esiste.

Modalità d'apertura

- Due modalità di apertura in scrittura:
  - *scrittura*
  - *append (accodamento)*
- Quando scriviamo dati in un file normalmente sovrascriviamo ciò che c'era
- Quando scriviamo in modalità "append" i dati invece vengono aggiunti, partendo dalla fine del file esistente

Modalità d'apertura

- Regole speciali vengono applicate quando apriamo il file sia in lettura che scrittura
  - **Non si può** passare da lettura a scrittura prima di chiamare una funzione per posizionarsi all'interno del file, a meno che l'operazione di lettura non sia arrivata alla fine del file
  - **Non si può** passare da scrittura a lettura senza chiamare una funzione per posizionarsi all'interno del file

Chiusura di un file

- La funzione `fclose` permette di chiudere un file quando non dobbiamo più effettuare operazioni
- L'argomento per la chiamata di `fclose` deve essere un puntatore precedentemente ottenuto con `fopen`
- `fclose` restituisce 0 se la chiusura avviene con successo
- Altrimenti restituisce il codice di errore `EOF` (una macro definita in `<stdio.h>`)

Chiusura di un file

- Una bozza di programma che apre un file per la lettura:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  …
  fclose(fp);
  return 0;
}
```

Chiusura di un file

- È abbastanza comune combinare `fopen` con la dichiarazione di `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

oppure il controllo del valore di ritorno

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) …
```

Nomi di file dalla linea di comando

- Ci sono molti modi per passare ad un programma i nomi dei file su cui operare
  - Scriverli nel codice sorgente (non è molto flessibile come soluzione)
  - Chiedere all'utente di inserire i nomi (scomodo per l'utente)
  - Prendere i nomi dalla linea di comandi (spesso è la soluzione migliore)
- Esempio: due nomi di file passati al programma `demo`:

```
demo names.dat dates.dat
```

Nomi di file dalla linea di comando

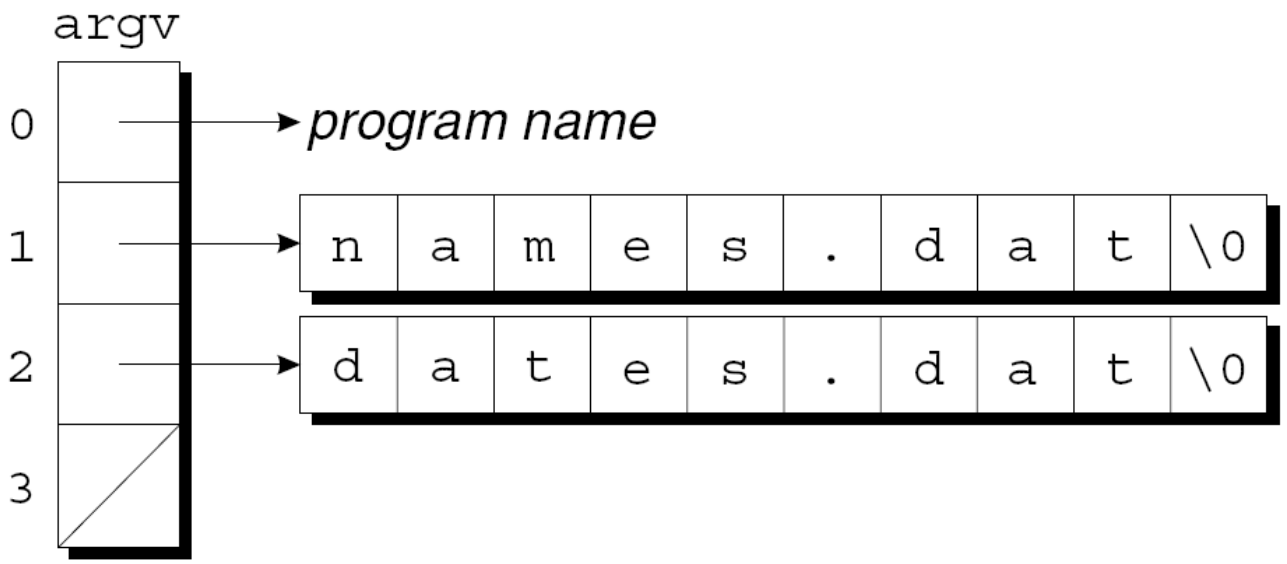
- Nel Capitolo 13, abbiamo visto come accedere ai parametri della linea di comando:

```
int main(int argc, char *argv[])
{
  …
}
```

- `argc` è il numero degli argomenti passati
- `argv` è un array di puntatori a stringhe (gli argomenti)

Nomi di file dalla linea di comando

- `argv[0]` punta al nome del programma, da `argv[1]` a `argv[argc-1]` ci sono i puntatori agli argomenti mentre `argv[argc]` è un puntatore nullo
- Nell'esempio del programma `demo`, `argc` è 3 e `argv` è fatto in questo modo:



Programma: Aprire un file

- Il programma `canopen.c` determina se un file esiste e può essere aperto per la lettura
- L'utente fornirà il nome del file come parametro nella linea comando:

  `canopen` *file*
- Il programma quindi stamperà o "*file* `can be opened`" oppure "*file* `can't be opened`"
- Se l'utente inserisce un numero di parametri sbagliato il programma stamperà `usage: canopen filename`.

canopen.c

```
/* Checks whether a file can be opened for reading */

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

int main(int argc, char *argv[])
{
  FILE *fp;

  if (argc != 2) {
    printf("usage: canopen filename\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("%s can't be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```

EOF e condizioni di errori

- Ogni flusso ha due indicatori ad esso associati: un ***segnalatore di errore*** ed un ***segnalatore di fine file***
- Questi indicatori vengono azzerati (`FALSE`) quando il file viene aperto
- Se si arriva alla fine del fine viene attivato (`TRUE`) il segnalatore di fine file
- Se si verifica un errore di input viene attivato il segnalatore di errore
  - Tale segnalatore viene attivato anche quando si verifica un errore di output

I/O in blocchi

- Le funzioni `fread` e `fwrite` permettono ad un programma di leggere e scrivere blocchi di dati in un solo colpo
- `fread` e `fwrite` vengono usate principalmente per file binari sebbene—con accortenza—è possibile usarle anche per file testuali

I/O in blocchi

- `fwrite` è progettata per copiare un array dalla memoria in uno stream
- Gli argomenti di `fwrite` sono:
  - L'indirizzo dell'array
  - Grandezza di un singolo elemento dell'array (in byte)
  - Numero di elementi da scrivere
  - Un puntatore a file
- Una chiama a `fwrite` che scrive l'intero contenuto dell'array `a` è:

```
fwrite(a, sizeof(a[0]),
       sizeof(a) / sizeof(a[0]), fp);
```

I/O in blocchi

- `fwrite` restituisce il numero di elementi effettivamente scritti
- Questo numero sarà minore del terzo argomento se si verifica un errore

I/O in blocchi

- `fread` leggerà gli elementi di un array da un flusso
- Ecco una chiamata a `fread` che legge il contenuto di un file e lo memorizza nell'array `a`:

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```

- Il valore restituito da `fread` è il numero di byte effettivamente letti dall'input
- Questo numero deve essere uguale al terzo argomento della funzione a meno che non si sia verificato un errore o il file sia finito

I/O in blocchi

- `fwrite` è comoda per programmi che devono memorizzare dati in un file prima di terminare l'esecuzione

- Successivamente, il programma stesso (o anche un altro programma) potrà usare la `fread` per riportare i dati in memoria

I/O in blocchi

- I dati non devono essere necessariamente in un array

- Ecco una chiamata a `fwrite` che scrive i dati di una struttura `s` nel file `fp`:

```
fwrite(&s, sizeof(s), 1, fp);
```

Programma: Modificare una parte di un file

- Le azioni eseguite dal programma `invclear.c` sono:
  - Aprire un file binario che contiene strutture di tipo `part`
  - Leggere le strutture in un array
  - Assegnare 0 al campo `on_hand` di ogni struttura
  - Riscrivere il file aggiornato

- Il programma apre il file in modalità `"rb+"` che permette sia la lettura che la scrittura

invclear.c

```
/* Modifies a file of part records by setting the quantity
   on hand to zero for all records */

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

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts;
```

```
int main(void)
{
  FILE *fp;
  int i;

  if ((fp = fopen("inventory.dat", "rb+")) == NULL) {
     printf("Can't open inventory file\n");
    exit(EXIT_FAILURE);
  }

  num_parts = fread(inventory, sizeof(struct part),
                    MAX_PARTS, fp);

  for (i = 0; i < num_parts; i++)
    inventory[i].on_hand = 0;

  rewind(fp);
  fwrite(inventory, sizeof(struct part), num_parts, fp);
  fclose(fp);

  return 0;
}
```

Posizionamento nel file

- La funzione `rewind` riassegna la posizione all'inizio del file
  - `rewind` non restituisce un valore ed azzera l'indicatore di errore del file `fp`.

I/O formattato

- Il gruppo seguente di funzioni della libreria utilizza stringhe di formato per controllare l'aspetto dell'output e stabilire quello dell'input

- `printf` e funzioni simili convertono dati numerici in formato testuale per poterli visualizzare

- `scanf` e funzioni simili possono convertire i dati da una forma testuale ad una forma numerica in fase di input

Le funzioni del gruppo "printf"

- Le funzioni **fprintf** e printf scrivono delle variabili su un flusso di output usando una stringa di formato per controllarne l'aspetto finale
- I prototipi per entrambe le funzioni finiscono con dei puntini sospensivi, che indicano che il numero di argomenti della funzione è variabile:

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

- Entrambe le funzioni restituiscono il numero di caratteri stampati; un valore negativo indica che si è verificato un errore

Le funzioni del gruppo "`printf`"

- `printf` scrive sempre su `stdout`, mentre `fprintf` scrive sullo stream indicato come primo argomento:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- Una chiamata a `printf` è equivalente ad una chiamata a `fprintf` con `stdout` come primo argomento

Le funzioni del gruppo "`printf`"

- `fprintf` funziona con un qualsiasi flusso di output
- Uno degli usi più comuni è quello di inviare i messaggi di errore su `stderr`:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

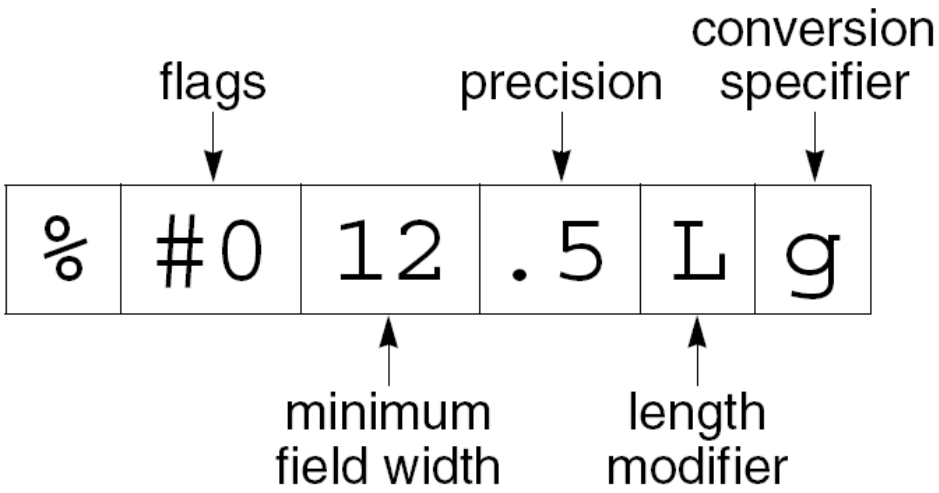
- Scrivere un messaggio su `stderr` garantisce che il messaggio apparirà sullo schermo anche se viene fatta la redirezione di `stdout`

Specificatori di conversione

- Sia `printf` che `fprintf` richiedono una stringa di formato che contiene dei normali caratteri stampabili e degli specificatori di conversione
  - I caratteri normali vengono stampati come appaiono nella stringa di formato
  - Gli specificatori di conversione invece descrivono in che modo i rimanenti argomenti passati alla funzione verranno convertiti in caratteri per essere visualizzati

Specificatori di conversione

- Uno specificatore di conversione è formato dal carattere % seguito da altri "oggetti" fino ad un massimo di 5
- Questi possibili "oggetti" sono;



Le funzioni ...`scanf`

- **`fscanf`** e `scanf` leggono dati da un flusso di input, in accordo ad una stringa di formato che specifica la formattazione dei dati
- Dopo la stringa di formato si devono specificare dei puntatori, ognuno dei quali servirà per memorizzare il dato corrispondente
- I dati in input verranno convertiti in accordo alla stringa di formato

```
scanf("%d%d", &i, &j);
```

Le funzioni ...`scanf`

- `scanf` legge i dati da `stdin`, mentre `fscanf` legge i dati dal flusso indicato come primo argomento

```
scanf("%d%d", &i, &j);
  /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);
  /* reads from fp */
```

- Una chiamata a `scanf` è equivalente ad una chiamata a `fscanf` con `stdin` usato come primo argomento

Le funzioni ...`scanf`

- Le funzioni ...`scanf` terminano prematuramente la propria esecuzione in caso di errori:
  - ***Errore di input*** (non è possibile leggere ulteriori byte)
  - ***Errore di formato*** (i dati in input non corrispondono alla stringa di formato)

Le funzioni ...`scanf`

- Le funzioni ...`scanf` restituiscono un intero pari al numero di dati in input che è stato possibile assegnare ad altrettanti oggetti (variabili)

- Restituiscono `EOF` se c'è un errore di input prima che si possa leggere un byte

Le stringhe di formato per ...`scanf`

- Le chiamate alle funzioni ...`scanf` somigliano alle chiamate alle funzioni ...`printf`
- Tuttavia le funzioni ...`scanf` funzionano in modo diverso
- Ricordo che la stringa di formato specifica un "pattern" che sarà usato dalla funzione per leggere i dati in input
  - Se l'input non corrisponde al formato, la funzione termina la sua esecuzione
  - I caratteri di input che creano questa non-corrispondenza vengono "rimessi" nel flusso di input e vengono "letti" alla prossima richiesta da parte del programma

Le stringhe di formato per ...`scanf`

- Una stringa di formato per le funzioni ...`scanf` può contenere 3 cose:
  - Specificatori di conversione
  - Caratteri “bianchi”
  - Caratteri “non-bianchi”

I/O di caratteri

- Le funzioni seguenti permettono invece di leggere o scrivere un singolo carattere (byte)

- Queste funzioni possono gestire sia file di testo che file binari

- Le funzioni trattano i caratteri come numeri, cioè come valori di tipo `int`, non di tipo `char`.

Funzioni di output

- `putchar` scrive un carattere nel flusso `stdout`:

```
putchar(ch);     /* writes ch to stdout */
```

- `fputc` e `putc` scrivono un carattere in un flusso specificato come argomento:

```
fputc(ch, fp);   /* writes ch to fp */
putc(ch, fp);    /* writes ch to fp */
```

- Se si verifica un errore queste 3 funzioni restituiscono `EOF` e attivano il segnalatore di errore.
- Altrimenti, restituiscono il carattere scritto

Funzioni di input

- `getchar` legge un carattere da `stdin`:

```
ch = getchar();
```

- `fgetc` e `getc` leggono un carattere da un flusso di input specificato nella chiamata:

```
ch = fgetc(fp);
ch = getc(fp);
```

Funzioni di input

- Le funzioni `fgetc`, `getc`, e `getchar` si comportano come le altre funzioni per quanto riguarda gli errori
- Alla fine del file attivano il segnalatore di fine file e restituiscono `EOF`
- Se si verifica un errore di input, attivano il segnalatore di errore di input e restituiscono `EOF`
- Per differenziare fra i due casi è necessario usare le funzioni `feof` o `ferror`.

Funzioni di input

- Uno degli usi più comuni di `fgetc`, `getc`, e `getchar` è quello di leggere caratteri da un file
- Ecco un tipico ciclo `while`:

```
while ((ch = getc(fp)) != EOF) {
   …
}
```

- La variabile `ch` è di tipo `int`

Funzioni di input

- La funzione `ungetc` "rimette" nel flusso di input il carattere letto
- Ecco un ciclo che legge una sequenza di cifre fermandosi al primo carattere che non è una cifra:

```
while (isdigit(ch = getc(fp))) {
   …
}
ungetc(ch, fp);
   /* pushes back last character read */
```

- Il carattere che non è una cifra è stato letto dal flusso ma viene rimesso nel flusso
- Verrà letto dalla prossima funzione che legge l'input

Programma: copia di un file

- Il programma `fcopy.c` crea una copia di un file
- Il nome del file originale e del nuovo file devono essere specificati sulla linea di comando quando il programma viene eseguito
- Un esempio di uso di `fcopy` per copiare `f1.c` in `f2.c`:

```
fcopy f1.c f2.c
```

- `fcopy` segnalerà un errore se non ci sono esattamente due nomi sulla linea di comando oppure se non riesce ad aprire i file

Programma: copia di un file

- Usare `"rb"` e `"wb"` come "modalità" di apertura dei file permette ad `fcopy` di copiare sia file di testo che file binary
- Se avessimo usato solo `"r"` e `"w"`, il programma avrebbe potuto avere problemi con i file binari

fcopy.c

```
/* Copies a file */

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

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;

  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }
```

```
  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  while ((ch = getc(source_fp)) != EOF)
    putc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

I/O per linee

- Le funzioni di libreria del prossimo gruppo leggono e scrivono linee di caratteri
- Queste funzioni sono usate principalmente con i file testuali

Funzioni di output

- La funzione `puts` scrive una stringa di caratteri su `stdout`:

```
puts("Hi, there!");  /* writes to stdout */
```

- Dopo aver scritto i caratteri della stringa la funzione `puts` **aggiunge** sempre il carattere new-line

Funzioni di output

- `fputs` è una versione più generale di `puts`.
- Il suo secondo argomento è un flusso sul quale scrivere l'output

```
fputs("Hi, there!", fp);  /* writes to fp */
```

- Diversamente da `puts`, la funzione `fputs` **non aggiunge** un carattere new-line (ovviamente se il carattere è presente nella stringa verrà scritto).
- Entrambe le funzioni restituiscono `EOF` se si verifica un errore in scrittura; altrimenti restituiscono un numero non negativo

Funzioni di input

- La funzione `gets` legge una linea di input da `stdin`:

```
gets(str); /* reads a line from stdin */
```

- `gets` legge caratteri uno alla volta, li memorizza nell'array puntato da `str`, fino a quando legge un new-line character (che viene eliminato).
- `fgets` è una versione più generale di `gets` che può leggere da un flusso qualsiasi
- `fgets` è anche più sicura di `gets`, in quanto limita il numero di caratteri che possono essere memorizzati

Funzioni di input

- Ecco una chiamata a `fgets` che legge una linea memorizzandola nell'array `str`:

```
fgets(str, sizeof(str), fp);
```

- `fgets` legge caratteri fino a quando trova un new-line oppure fino a `sizeof(str)` – 1 caratteri.
- Se si ferma perchè trova un new-line **lo memorizza** nella stringa

Funzioni di input

- Sia `gets` che `fgets` restituiscono un puntatore nullo se si verifica un errore di lettura oppure si raggiunge la fine dell'input senza avere letto nessun carattere.
- Altrimenti entrambe restituiscono il loro primo argomento, cioè un puntatore all'array in cui è stata memorizzata la stringa
- Le funzioni memorizzano il carattere di fine stringa alla fine della stringa

Funzioni di input

- `fgets` dovrebbe essere usata quasi sempre al posto di `gets`
- `fgets` leggerà da `stdin` se tale flusso viene usato come terzo argomento:

```
fgets(str, sizeof(str), stdin);
```

Appendice

- Per consultazione
- Ulteriori dettagli sulle funzioni descritte (e.g., specificatori di formato).
- Funzioni aggiuntive (e.g. posizionamento nei file, controllo degli errori etc…)

Altre operazioni sui file

- Le funzioni `remove` e `rename` permettono di cancellare e rinominare un file
- Diversamente dalle altre funzioni, `remove` e `rename` lavorano con i *nomi* dei file e non con i *puntatori* associati ai file
- Entrambe le funzioni restituiscono zero se non ci sono stati problemi ed un valore diverso da zero altrimenti

Altre operazioni sui file

- `remove` cancella un file:

```
remove("foo");
  /* deletes the file named "foo" */
```

- Se un programma crea un file temporaneo con `fopen`, si può usare `remove` per cancellarlo quando il programma termina
- È opportuno cancellare un file **quando non è aperto**: l'effetto della rimozione di un file aperto dipende dall'implementazione

Altre operazioni sui file

- `rename` cambia il nome di un file:

```
rename("foo", "bar");
  /* renames "foo" to "bar" */
```

- `rename` è comoda se si vuole rendere permanente un file temporaneo aperto con `fopen`
  - Se il nuovo nome corrisponde ad un file già esistente, il comportamento della funzione dipende dall'implementazione
- Anche in questo caso è meglio operare su un file non aperto: se il file è aperto `rename` potrebbe non essere in grado di cambiare il nome

I/O con stringhe

- Alcune funzioni descritte in precedenza possono essere usate con una stringa che funge da flusso di input o di output
- `sprintf` e `snprintf` riscrivono i caratteri in una stringa (non su `stdout`)
- `sscanf` legge i dati da una stringa, non da `stdin`

Funzioni di output su stringhe

- La funzione `sprintf` scrive l'output nella stringa puntata dal suo primo argomento
- Ecco una chiamata che scrive `"9/20/2010"` nella variabile `date`:

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```

- `sprintf` aggiunge il carattere nullo di fine stringa
- Restituisce il numero di caratteri scritti, senza contare il carattere nullo di fine stringa

Funzioni di output su stringhe

- `sprintf` può essere usata per formattare i dati salvandoli in una stringa prima di produrre effettivamente l'output (ad es. stampando la stringa)
- `sprintf` è comoda anche per convertire numeri in stringhe

Funzioni di output su stringhe

- La funzione `snprintf` (nuova nel C99) è come `sprintf`, ma prende un secondo parametro aggiuntivo `n`.
- Non più di `n` – 1 caratteri verrano scritti nella stringa di output, senza contare il carattere nullo di fine stringa che viene sempre scritto, a meno che `n` non sia 0
- Esempio:

```
snprintf(name, 13, "%s, %s", "Einstein", "Albert");
```

La stringa`"Einstein, Al”` viene scritta in `name`.

Funzioni di output su stringhe

- `snprintf` restituisce il numero di caratteri che avrebbe scritto (senza includere il carattere di fine stringa) se non ci fossero state limitazioni
- Se si verifica un errore, `snprintf` restituisce un numero negativo
- Per vedere se `snprintf` è riuscita a scrivere tutti i caratteri necessari si può controllare se il suo valore di ritorno è minore di `n`.

Funzioni di input da stringhe

- La funzione `sscanf` è simile a `scanf` e `fscanf`.
- `sscanf` legge da una stringa, puntata dal suo primo argomento, invece di leggere da un flusso di input
- Il secondo argomento di `sscanf` è una stringa di formato uguale a quella di `scanf` e `fscanf`
- Anche i successivi argomento sono come per `scanf` e `fscanf`

Funzioni di input da stringhe

- `sscanf` è molto comoda per estrarre dati da una stringa (magari letta da un'altra funzione)
- Ecco un esempio che usa `fgets` per leggere una linea di input da `stdin`, e quindi usa `sscanf` per estrarre i dati:

```
fgets(str, sizeof(str), stdin);
  /* reads a line of input */
sscanf(str, "%d%d", &i, &j);
  /* extracts two integers */
```

Funzioni di input da stringhe

- Un vantaggio nell'usare `sscanf` è che si può esaminare la linea di input quante volte si vuole
- Rende più facile riconoscere forme alternative dell'input atteso
- Consideriamo il problema di leggere una data che può essere scritta sia nel formato *month*/*day*/*year* che nel formato *month*-*day*-*year*:

```
if (sscanf(str, "%d /%d /%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day, year);
else if (sscanf(str, "%d -%d -%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day, year);
else
  printf("Date not in the proper form\n");
```

Funzioni di input da stringhe

- Come `scanf` e `fscanf`, `sscanf` restituisce il numero di variabili che è riuscita ad assegnare
- `sscanf` restituisce `EOF` se la stringa di input finisce prima di permettere di trovare il primo argomento

Caratteri estesi

- Nel C99 alcune funzioni di I/O appartengono al file di intestazione `<wchar.h>`
- Le funzioni in `<wchar.h>` gestiscono il set di caratteri esteso
- Le funzioni in `<stdio.h>` che leggono o scrivono dati sono dette ***funzioni di input/output per byte (caratteri)***
- Le funzioni analoghe in `<wchar.h>` sono dette ***funzioni di input/output per caratteri estesi***

Le funzioni del gruppo "`printf`"

- Le funzioni `fprintf` e `printf` scrivono delle variabili su un flusso di output usando una stringa di formato per controllarne l'aspetto finale
- I prototipi per entrambe le funzioni finiscono con dei puntini sospensivi, che indicano che il numero di argomenti della funzione è variabile:

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
int printf(const char * restrict format, ...);
```

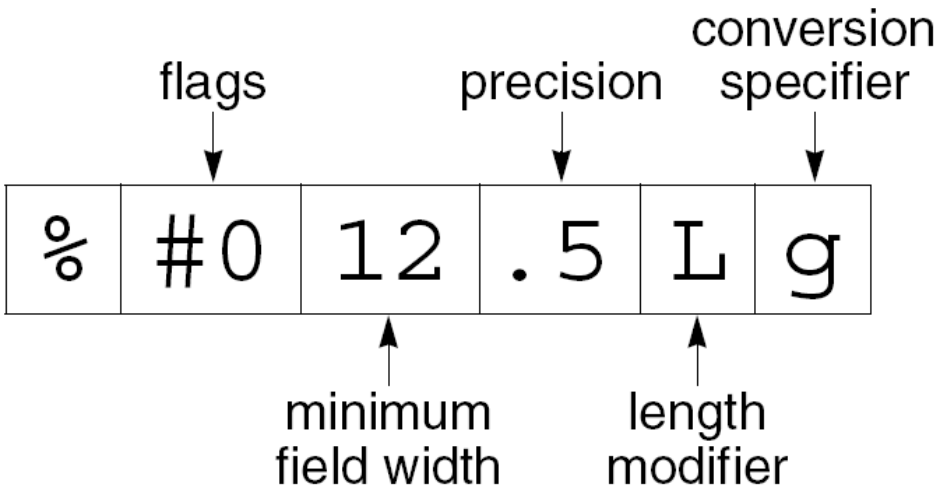
- Entrambe le funzioni restituiscono il numero di caratteri stampati; un valore negativo indica che si è verificato un errore

Specificatori di conversione

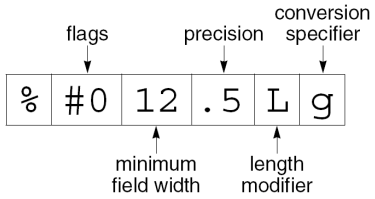
- Sia `printf` che `fprintf` richiedono una stringa di formato che contiene dei normali caratteri stampabili e degli specificatori di conversione
  - I caratteri normali vengono stampati come appaiono nella stringa di formato
  - Gli specificatori di conversione invece descrivono in che modi i rimanenti argomenti passati alla funzione verrano convertiti in caratteri per essere visualizzati

Specificatori di conversione

- Uno specificatore di conversione è formato dal carattere % seguito da altri "oggetti" fino ad un massimo di 5
- Questi possibili "oggetti" sono;

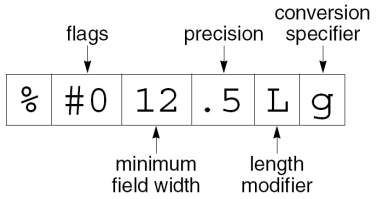


Specificatori di conversione



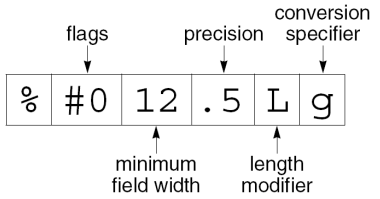
Flag	Significato
-	Allinea a sinistra all'interno del campo (default: allinea a destra)
+	Fai precedere sempre il segno al numero (normalmente solo il segno – viene mostrato)
(spazio)	I numeri non negativi prodotti dalle conversioni con segno vengono preceduti da uno spazio (il flag + annulla il flag *spazio*)
#	Esplicita la base del numero: ottali iniziano con 0, esadecimali diversi da 0 con 0x e 0X. I numeri a virgola mobile hanno sempre il separatore decimale. Lascia gli zero di "riempimento a sinistra" per le conversioni `g` e `G` (nmero di cifre significative)
0 *(zero)*	I numeri vengono completati con degli 0 di "riempimento" per occupare tutta la dimensione del campo. Il flag viene ignorato se è stato specificato una conversione `d,i,o,u,x,X` o se è stata specificata la precisione

Specificatori di conversione



- ***Larghezza minima*** (opzionale). Un valore che è più piccolo della larghezza minima verrà "allungato"
  - di default con degli zero alla sinistra
- Un valore che necessità di più caratteri userà più caratteri (è un minimo)
- La larghezza minima è o un intero o il carattere*
  - Se è * la larghezza viene ottenuta dall'argomento successivo

Specificatori di conversione



- ***Precisione*** (opzionale). Il significato della precisione dipende dal tipo di conversione:

  `d`, `i`, `o`, `u`, `x`, `X`: Numero minimo di cifre (vengono aggiunti zero di riempimento alla sinistra)

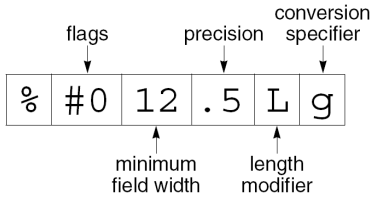
  `a`, `A`, `e`, `E`, `f`, `F`: Numero di cifre dopo il punto decimale

  `g`, `G`: Numero di cifre significative

  `s`: Numero massimo di byte

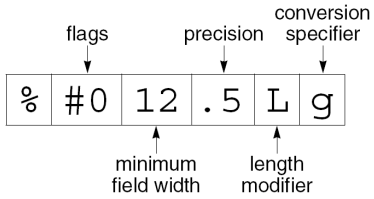
- La precisione è specificata con un punto (`.`) seguito da un intero o dal carattere asterisco (`*`)
  - Se c'è `*` la precisione viene ottenuta dall'argomento seguente

Specificatori di conversione



- ***Modificatore di lunghezza*** (opzionale). Indica che l'oggetto da visualizzare ha una lunghezza che è maggiore o minore del normale
  - `%d` fa riferimento ad un valore `int`; `%hd` viene usato per visualizzare uno `short int` e `%ld` viene usato per visualizzare un `long int`.

Specificatori di conversione



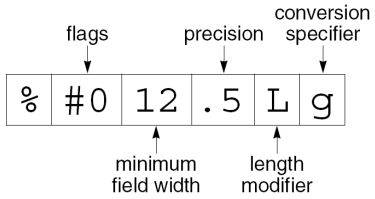
Mod.	*Conversione*	*Significato*
`hh`†	`d, i, o, u, x, X` `n`	`signed char, unsigned char` `signed char *`
`h`	`d, i, o, u, x, X` `n`	`short int, unsigned short int` `short int *`

`l` (*elle*)	`d, i, o, u, x, X`
	`N`
	`c`
	`s`
	`a, A, e, E, f, F, g, G`
		`long int, unsigned long int`
		`long int *`
		`wint_t`
		`wchar_t *`
		Nessun effetto

`ll`†	`d, i, o, u, x, X` `n`	`long long int, unsigned long long int` `long long int *`
`j`†	`d, i, o, u, x, X` `n`	`intmax_t, uintmax_t` `intmax_t *`
`z`†	`d, i, o, u, x, X` `n`	`size_t` `size_t *`
`t`†	`d, i, o, u, x, X` `n`	`ptrdiff_t` `ptrdiff_t *`
`L`	`a, A, e, E, f, F, g, G`	`long double`

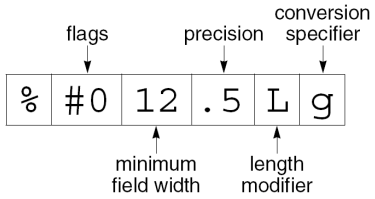
†C99

Specificatori di conversione



Lettera	Significato
`d`, `i`	Converte un `int` in formato decimale
`o`, `u`, `x`, `X`	Converte un `unsigned int` usando la base 8 (`o`), la base 10 (`u`), o la base 16 (`x`, `X`). `x` usa `a`–`f` mentre `X` usa `A`–`F`
`f`, `F`†	Converte un `double` in forma decimale. Se non è specificata una precisione, usa 6 cifre dopo il punto
`e`, `E`	Converte un `double` in notazione scientifica; nell'output ci sarà `e` o `E`
`g`, `G`	`g` converte un `double` alla forma `f` (esponente < -4) o alla forma `e`. `G` fa lo stesso con `F` ed `E`.
`a`†, `A`†	Converte un `double` in notazione esadecimale scientifica: [-]`0x`*h* . *hhhh*`p`±*d*. `a` usa `a`–`f`; `A` usa `A`–`F`. La scelta di `a` o `A` determina anche `x` o `X`

Specificatori di conversione



Lettera	Significato
`c`	Stampa un `int` come un carattere
`s`	Scrive i caratteri della stringa puntata dall'argomento. Si ferma se raggiunge il numero di caratteri specificato dalla precisione oppure quando trova il carattere di fine stringa (carattere nullo)
`p`	Converte un `void *` in un valore stampabile
`n`	L'argomento corrispondente deve puntare ad un oggetto di tipo `int`. Memorizza in questo oggetto il numero di caratteri stampati fino a quel momento; non produce nessun output
`%`	Scrive il carattere `%`

Specificatori di conversione (C99)

- Nel C99 ci sono
  - Ulteriori modificatori di lunghezza
  - Ulteriori tipi di conversione
  - La capacità di scrivere il valore “infinito” come NaN
  - Supporto per i caratteri estesi
  - Altre caratteristiche

Esempi di specificatori di conversione

- Esempi dell'effetto dei modificatori per la conversione `%d`:

Specificatore di conversione	***Risultato quando l'argomento è 123***	***Risultato quando l'argomento è –123***
`%8d`	`•••••123`	`••••-123`
`%-8d`	`123•••••`	`-123••••`
`%+8d`	`••••+123`	`••••-123`
`% 8d`	`•••••123`	`••••-123`
`%08d`	`00000123`	`-0000123`
`%-+8d`	`+123••••`	`-123••••`
`%- 8d`	`•123••••`	`-123••••`
`%+08d`	`+0000123`	`-0000123`
`% 08d`	`•0000123`	`-0000123`

Esempi di specificatori di conversione

- Altri esempio con il flag `#` per le conversioni `o`, `x`, `X`, `g`, e `G`:

Specificatore di conversione	***Risultato quando l'argomento è 123***	***Risultato quando l'argomento è 123.0***
`%8o`	`•••••173`	
`%#8o`	`••••0173`	
`%8x`	`••••••7b`	
`%#8x`	`••••0x7b`	
`%8X`	`••••••7B`	
`%#8X`	`••••0X7B`	
`%8g`		`•••••123`
`%#8g`		`•123.000`
`%8G`		`•••••123`
`%#8G`		`•123.000`

Esempi di specificatori di conversione

- Esempi con la dimensione minima e la precisione per la conversione `%s`:

Specificatore di conversione	***Risultato quando l'argomento è* "bogus"**	***Risultato quando l'argomento è* "buzzword"**
`%6s`	`•bogus`	`buzzword`
`%-6s`	`bogus•`	`buzzword`
`%.4s`	`bogu`	`buzz`
`%6.4s`	`••bogu`	`••buzz`
`%-6.4s`	`bogu••`	`buzz••`

Esempi di specificatori di conversione

- Esempi per la conversione `%g` che visualizza i numeri a volte come `%e` altre come `%f`:

Number	*Result of Applying* `%.4g` *Conversion to Number*
123456.	1.235e+05
12345.6	1.235e+04
1234.56	1235
123.456	123.5
12.3456	12.35
1.23456	1.235
.123456	0.1235
.0123456	0.01235
.00123456	0.001235
.000123456	0.0001235
.0000123456	1.235e-05
.00000123456	1.235e-06

Esempi di specificatori di conversione

- Normalmente la dimensione minima e la precisione sono specificate come costanti nella stringa di formato
- Si può usare il carattere `*` per specificarli come argomenti *dopo* la stringa di formato
- Esempi di `printf` equivalenti fra loro:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

Esempi di specificatori di conversione

- Il vantaggio di usare * è che ci permette di specificare il valore anche con una macro (facilmente modificabile) :

```
printf("%*d", WIDTH, i);
```

- … o come variabile se il valore dipende dall'esecuzione:

```
printf("%*d", page_width / num_cols, i);
```

Esempi di specificatori di conversione

- La conversione `%p` viene usata per stampare un puntatore

```
printf("%p", (void *) ptr);
  /* displays value of ptr */
```

- I puntatori vengono normalmente visualizzati in notazione ottale o esadecimale

Esempi di specificatori di conversione

- La conversione `%n` viene usata per sapere quanti caratteri sono stati stampati, fino a quel punto, dalla funzione

- Ad esempio, dopo la seguente chiamata, il valore di `len` sarà 3:

```
printf("%d%n\n", 123, &len);
```

Le stringhe di formato per ...`scanf`

- Le chiamate alle funzioni ...`scanf` somigliano alle chiamate alle funzioni ...`printf`
- Tuttavia le funzioni ...`scanf` funzionano in modo diverso
- La stringa di formato specifica un "pattern" che sarà usato dalla funzione per leggere i dati in input
  - Se l'input non corrisponde al formato, la funzione termina la sua esecuzione
  - I caratteri di input che creano questa non-corrispondenza vengono "rimessi" nel flusso di input e quindi saranno "letti" alla prossima richiesta da parte del programma

Le stringhe di formato per ...`scanf`

- Una stringa di formato per le funzioni ...`scanf` può contenere 3 cose:
  - Specificatori di conversione
  - Caratteri “bianchi”
  - Caraterri “non-bianchi”

Le stringhe di formato per ...`scanf`

- ***Specificatori di conversione.*** Sono simili a quelli delle funzioni ...`printf`
- La maggior parte degli specificatori buttano via eventuali caratteri "bianchi" all'inizio dell'input (le eccezioni sono `%[`, `%c`, e `%n`).
- I caratteri "bianchi" successivi ad ogni singolo dato vengono lasciati nell'input e saranno letti dal successivo specificatore

Le stringhe di formato per ...`scanf`

- ***Caratteri "bianchi".*** **Uno o più** caratteri bianchi in una stringa di formato corrispondono a **zero o più** caratteri bianchi nel flusso di input
- ***Caratteri "non-bianchi".*** Un carattere non bianco, diverso da `%`, corrisponde allo stesso carattere nel flusso di input

Le stringhe di formato per ...`scanf`

- La stringa di formato `"ISBN %d-%d-%ld-%d"` specifica che l'input dovrà consistere di:
  - i caratteri "I", "S", "B" e "N"
  - eventualmente dei caratteri bianchi
  - un intero
  - il carattere "-"
  - un intero (eventualmente preceduto da caratteri bianchi)
  - il carattere "-"
  - un intero long (eventualmente preceduto da caratteri bianchi)
  - il carattere "-"
  - un intero (eventualmente preceduto da caratteri bianchi)

Specificatori di conversione ...`scanf`

- Uno specificatore di conversione per ...`scanf` consiste del carattere `%` seguito da:
  - `*`
  - Grandezza massima del campo
  - Modificatore di lunghezza
  - Specificatore del tipo di dati
- `*` (facoltativo). Significa ***non assegnare il dato:*** un dato in input viene letto ma non viene assegnato a nessun oggetto
  - I dati soppressi con `*` non vengono conteggiati nel valore di ritorno della funzione

Specificatori di conversione ...`scanf`

- ***Grandezza massima*** (facoltativo). Limita il numero di caratteri
  - I caratteri bianchi ignorati all'inizio dell'input non vengono conteggiati
- ***Modificatore di lunghezza*** (facoltativo). Indica che l'oggetto in cui il dato verrà memorizzato ha una lunghezza che è maggiore o minore di quella normale

Specificatori di conversione ...`scanf`

Modificatore di lungheza	*Carattere di conversione*	*Significato*
`hh`†	`d, i, o, u, x, X, n`	`signed char *, unsigned char *`
`h`	`d, i, o, u, x, X, n`	`short int *, unsigned short int *`

`l` (*ell*)	`d, i, o, u, x, X, n`
	`a, A, e, E, f, F, g, G`
	`c, s,` or `[`
		`long int *, unsigned long int *`
		`double *`
		`wchar_t *`

`ll`† (*ell-ell*)	`d, i, o, u, x, X, n`	`long long int *, unsigned long long int *`
`j`†	`d, i, o, u, x, X, n`	`intmax_t *, uintmax_t *`
`z`†	`d, i, o, u, x, X, n`	`size_t *`
`t`†	`d, i, o, u, x, X, n`	`ptrdiff_t *`
`L`	`a, A, e, E, f, F, g, G`	`long double *`

†C99 only

Specificatori di conversione ...`scanf`

- Deve essere uno dei seguenti caratteri.

Carattere	*Significato*
`d`	Un intero decimale, l'argomento corrispondente deve essere `int *`
`i`	Un intero, l'argomento corrispondente deve essere un `int *`. L'intero sarà considerato in base 8 se inizia con 0, in base 16 se inizia con 0x o 0X, altrimenti in base 10
`o`	In intero ottale, l'argomento corrispondente deve essere `unsigned int *`
`u`	Un intero decimale, l'argomento corrsipondente deve essere `unsigned int *`

Specificatori di conversione ...`scanf`

Carattere	*Significato*
`x`, `X`	Un intero esadecimale, l'argomento corrispondente deve essere `int *`
`a`†, `A`†, `e`, `E`, `f`, `F`†, `g`, `G`	Un numero in virgola mobile, l'argomento corrispondente deve essere `float *`
`c`	Un carattere, oppure *n* caratteri dove *n* è il modificatore di lunghezza massima. L'argomento deve essere un puntatore ad un carattere o ad un array di caratteri.
`s`	Una sequenza di caratteri non-bianchi. Alla fine verrà aggiunto il carattere di fine stringa. L'argomento deve essere un puntatore ad un array di caratteri
`[`	Stringa su un insieme di caratteri %[***charset***] Come %s ma i caratteri devono far parte dell'insieme *charset*
`p`	Puntatore nella forma in cui lo scrive la ...`printf`. L'argomento deve essere `void *`
`n`	Numero di caratteri letti da ...`scanf` fino a questo punto. L'argomento deve essere `int*`. Non viene consumato nessun carattere dall'input ed il valore di ritorno non è influenzato da questo assegnamento
`%`	%% permette di specificare il carattere % nella stringa di formato

Specificatori di conversione ...`scanf`

- I dati numerici possono sempre iniziare con il segno (+ o −).
- I caratteri di conversione `o`, `u`, `x`, e `X` convertono i dati nei formati unsigned, quindi normalmente non vengono utilizzati per leggere numeri negativi

Specificatori di conversione ...`scanf`

- Il carattere di conversione `[` è una versione un po' più complicata (ma più flessibile) del carattere di conversione `s`
- La forma completa è `%[`*set*`]` or `%[^`*set*`]`, dove *set* è un insieme arbitrario di caratteri
- `%[`*set*`]` corrisponde a sequenze di caratteri in *set* (detto anche ***scanset,*** insieme di scansione).
- `%[^`*set*`]` corrisponde a sequenze di caratteri **non** in *set*

Esempi:

`%[abc]` sequenze contenenti solo `a`, `b`, e `c`.

`%[^abc]` sequenze che non contengono `a`, `b`, o `c`.

Specificatori di conversione ...`scanf`

- Molti degli specificatori di conversione delle funzioni ...`scanf` sono strettamente correlati alle funzioni numeriche in `<stdlib.h>`.

- Queste funzioni convertono stringhe di caratteri (es `"-297"`) nell'equivalente numero (–297).

- Lo specificatore `d`, per esempio, cerca un segno facoltativo, + o –, seguito da cifre decimale; questo formato è lo stesso richiesto dalla funzione `strtol`

Specificatori di conversione ...`scanf`

- Corrispondenza fra specificatori ...`scanf` e funzioni di conversioni da stringa a numero:

Convesione scanf	***Funzione di conversione numerica***
`d`	`strtol`, con parametro base=10
`i`	`strtol` con parametro base=0 (che significa base 8,10 o 16)
`o`	`strtoul` con parametro base=8
`u`	`strtoul` con parametro base=10
`x, X`	`strtoul` con parametro base=16
`a, A, e, E, f, F, g, G`	`strtod`

```
long strtol(const char * str, char ** endptr, int base);
```

Specificatori di conversione ...`scanf` C99

- C99 ha introdotto per gli specificatori di conversione delle funzioni `scanf` e `fscanf`:
  - Nuovi modificatori di lunghezza
  - Nuovi specificatori di conversione
  - Capacità di gestire il "numero" infinito e NaN
  - Il supporto per i caratteri multi-byte

Esempi per `scanf`

- Nel seguito sono riportati degli esempi con `scanf`.
- I caratteri che sono stampati con la ~~barretta di cancellazione~~ vengono "consumati" dalla funzione

Esempi per `scanf`

Chiamata a `scanf`	*Input*	*Variabili*
`n = scanf("%d%d", &i, &j);`	~~12•~~,•34¤	n: 1 i: 12 j: unchanged
`n = scanf("%d,%d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged
`n = scanf("%d ,%d", &i, &j);`	~~12•,•34~~¤	n: 2 i: 12 j: 34
`n = scanf("%d, %d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged

Esempi per `scanf`

`scanf` *Call*	***Input***	***Variables***
`n = scanf("%*d%d", &i);`	~~`12•34`~~¤	`n: 1` `i: 34`
`n = scanf("%*s%s", str);`	~~`My•Fair`~~`•Lady`¤	`n: 1` `str: "Fair"`

`n = scanf("%1d%2d%3d", &i, &j, &k);`	~~`12345`~~¤	`n: 3`
		`i: 1`
		`j: 23`
		`k: 45`

`n = scanf("%2d%2s%2d", &i, str, &j);`	~~`123456`~~¤	`n: 3`
		`i: 12`
		`str: "34"`
		`j: 56`

Esempi per `scanf`

`scanf` *Call*	***Input***	***Variables***
`n = scanf("%i%i%i", &i, &j, &k);`	~~`12•012•0x12`~~¤	`n`: 3 `i`: 12 `j`: 10 `k`: 18
`n = scanf("%[0123456789]", str);`	~~`123`~~`abc`¤	`n`: 1 `str`: `"123"`
`n = scanf("%[0123456789]", str);`	`abc123`¤	`n`: 0 `str`: unchanged
`n = scanf("%[^0123456789]", str);`	~~`abc`~~`123`¤	`n`: 1 `str`: `"abc"`
`n = scanf("%*d%d%n", &i, &j);`	~~`10•20`~~`•30`¤	`n`: 1 `i`: 20 `j`: 5

EOF e condizioni di errori

- Se chiediamo alle funzioni ...`scanf` di leggere un certo numero *n* di dati, è intuitivo aspettarsi un valore di ritorno pari a *n*
- Il valore di ritorno può essere minore di *n* se la funzione non è riuscita ad assegnare tutti i valori a causa di:
  - ***EOF (End-of-file).*** La funzione ha trovato il segnalatore di fine file nell'input
  - ***Errore di lettura.*** La funzione non riesce a leggere altri caratteri dal flusso di input
  - ***Manca corrispondenza di formato.*** L'input non corrisponde al formato

EOF e condizioni di errori

- Quando un segnalatore (fine file o errore) viene attivato, rimane attivo fino a quando non viene azzerato esplicitamente con una chiamata alla funzione `clearerr`
- `clearerr` azzera sia il segnalatore di fine file che quello di errore di lettura:

```
clearerr(fp);
  /* clears eof and error indicators for fp */
```

- `clearerr` è necessaria di rado, poichè altre funzioni della libreria azzerano i segnalatori come effetto collaterale

EOF e condizioni di errori

- Le funzioni `feof` e `ferror` possono essere usate per controllare se si è verificata la condizione di errore
- La chiamata `feof(fp)` restituisce un valore nonzero se il segnalatore EOF è stato attivato dalla precedente operazione sullo stream `fp`.
- La chiamata `ferror(fp)` restituisce un valore nonzero se il segnalatore di errore di input è attivato

EOF e condizioni di errori

- Quando `scanf` restituisce un valore più piccolo di quello atteso, `feof` e `ferror` possono essere usate per capire cosa è successo
  - Se `feof` restituisce `TRUE` allora il file è finito
  - Se `ferror` restituisce `TRUE` allora si è verificato un errore di input
  - Se nessuna delle due funzioni restituisce `TRUE` allora c'è stata una mancata corrispondenza dell'input con la stringa di formato
- Il valore di ritorno della `scanf` indica quante variabili sono state assegnate prima del verificarsi del problema

EOF e condizioni di errori

- La funzione `find_int` è un esempio che sfrutta `feof` e `ferror`
- `find_int` cerca una linea che inizia con un intero:

```
n = find_int("foo");
```

- `find_int` restituisce il valore dell'intero nella linea oppure un codice di errore
  - –1 Il file non può essere aperto
  - –2 Errore di lettura
  - –3 Nessuna linea inizia con un intero

```
int find_int(const char *filename)
{
  FILE *fp = fopen(filename, "r");
  int n;

  if (fp == NULL)
    return -1;                  /* can't open file */

  while (fscanf(fp, "%d", &n) != 1) {
    if (ferror(fp)) {
      fclose(fp);
      return -2;                /* read error */
    }
    if (feof(fp)) {
      fclose(fp);
      return -3;                /* integer not found */
    }
    fscanf(fp, "%*[^\n]");      /* skips rest of line */
  }

  fclose(fp);
  return n;
}
```

Posizionamento nel file

- Ogni flusso ha una ***posizione*** ad esso associata
- Tale posizione indica il prossimo byte su cui operare
- Quando un file viene aperto la "posizione" è inizilizzata al primo byte del file
  - In modalità "append", la posizione iniziale potrebbe essere anche alla fine del file, dipende dall'implementazione
- Quando viene effettuata una operazione di lettura o scrittura la posizione avanza automaticamente fornendo quindi un accesso ***sequenziale*** ai dati del file

Posizionamento nel file

- Sebbene l'accesso sequenziale sia sufficiente per la maggior parte delle applicazioni, in alcuni casi è necessario poter "saltare" da un parte all'altra di un file
- Se un file contiene una serie di record (strutture), potremmo voler leggere direttamente una determinata struttura senza leggere tutte quelle che sono memorizzate prima
- La libreria `<stdio.h>` fornisce 5 funzioni che permettono di gestire la posizione all'interno di un file

Posizionamento nel file

- La funzione `fseek` cambia la posizione all'interno di un file specificato come primo argomento (un puntatore a file).
- Il terzo argomento è una delle seguenti 3 macro:

`SEEK_SET`	Inizio del file
`SEEK_CUR`	Posizione attuale
`SEEK_END`	Fine del file

- Il secondo argomento, di tipo `long int`, è un conteggio (che può essere negativo) di byte che permette di spostarsi relativamente alla posizione specificata con il secondo argomento

Posizionamento nel file

- `fseek` per spostarsi:

```
fseek(fp,  0, SEEK_SET); //Inizio file

fseek(fp, 10, SEEK_SET);
   //10 byte dall'inizio file

fseek(fp, 0, SEEK_END); //Fine file
fseek(fp, -10, SEEK_END);
   //10 byte prima della fine del file

fseek(fp, -10, SEEK_CUR);//10 byte indietro

fseek(fp, 10, SEEK_CUR); //10 byte avanti
```

- Normalmente `fseek` restituisce 0. Se c'è un errore (ad esempio la posizione richiesta non esiste), `fseek` restituisce un valore diverso da 0

Posizionamento nel file

- Le funzioni per il posizionamento in un file sono molto utili per i file binari
- Non è vietato usarle per i file di testo, ma esistono delle limitazioni
- Per flussi di testo, `fseek` può essere usata per muoversi solo all'inizio o alla fine del file o per ritornare in un punto visitato precedentemente
- Una limitazione per flussi binari: `fseek` potrebbe non supportare chiamate in cui il terzo argomento è `SEEK_END`.

Posizionamento nel file

- La funzione `ftell` restituisce la posizione attuale
- Il valore restituito da una `ftell` può essere usato in una chiamata a `fseek`:

```
long file_pos;
…
file_pos = ftell(fp);
  /* saves current position */
…
fseek(fp, file_pos, SEEK_SET);
  /* returns to old position */
```

Posizionamento nel file

- Se `fp` punta ad un flusso di un file binario, la chiamata `ftell(fp)` restituisce la posizione corrente come numero di byte, dove 0 rappresenta l'inizio del file
- Se `fp` è un flusso di testo, `ftell(fp)` non è necessariamente il numero di byte (se si usano i caratteri multi-byte)
- Pertanto è meglio non effettuare operazioni aritmetiche usando il valore restituito da `ftell`.

Posizionamento nel file

- La funzione `rewind` riassegna la posizione all'inizio del file
- La chiamata `rewind(fp)` è praticamente equivalente a `fseek(fp, 0L, SEEK_SET)`.
  - Piccola differenza: `rewind` non restituisce un valore ed azzera l'indicatore di errore del file `fp`.

Posizionamento nel file

- `fseek` ed `ftell` sono limitate dal fatto che usano un long int per rappresentare la posizione all'interno di un file
- Per lavorare con file molto grandi, il C fornisce le funzioni `fgetpos` e `fsetpos`.
- Queste funzioni usano il tipo `fpos_t` per rappresentare la posizione
  - Un valore `fpos_t` non è necessariamente un intero, ptrebbe essere una struttura

Posizionamento nel file

- La chiamata `fgetpos(fp, &file_pos)` memorizza la posizione attuale di `fp` nella variabile `file_pos`
- La chiamata `fsetpos(fp, &file_pos)` cambia il valore della posizione del file `fp` nel nuovo valore `file_pos`.
- Se si verifica un errore `fgetpos` e `fsetpos` memorizzano un codice di errore nella variabile gloable `errno`.
- Entrambe restituiscono 0 quando non ci sono errori ed un valore diverso da 0 in caso di errore

Posizionamento nel file

- Ecco un esempio che usa `fgetpos` e `fsetpos` per memorizzare una posizione e poi ripristinarla

```
fpos_t file_pos;
…
fgetpos(fp, &file_pos);
  /* saves current position */
…
fsetpos(fp, &file_pos);
  /* returns to old position */
```