Programmazione in C

Unità
I/O Avanzato e File

I/O Avanzato e File

- Definizione di file
- File di testo in C
- Input robusto
- Formattazione avanzata
- Esercizi proposti
- Sommario

Riferimenti al materiale

- Testi
  - Kernighan & Ritchie: capitolo 7, appendice B
  - Cabodi, Quer, Sonza Reorda: capitoli 3, 8
  - Dietel & Dietel: capitoli 9, 11
- Dispense
  - Scheda: "I/O Avanzato in C"
  - Scheda: "Gestione dei file in C"

I/O Avanzato e File

Definizione di file

Definizione di file

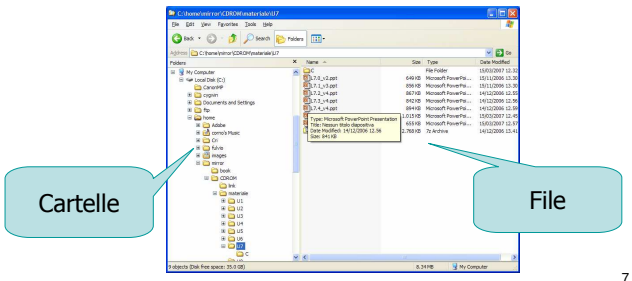
- Directory e file
- File binari e file di testo

Definizione di file

Directory e file

Directory e file

- Tutti i sistemi operativi permettono di organizzare le informazioni su hard disk secondo la metafora di cartelle (directory) e file



Definizioni

- File:
  - Una sequenza di byte
  - Memorizzata su un disco
  - Caratterizzata da uno specifico nome
  - Contenuta all'interno di una specifica directory
- Directory:
  - Un contenitore di file e di altre directory
  - Caratterizzata da uno specifico nome
  - Contenuta all'interno di un'altra directory

Identificazione di un file

- Per identificare un file occorre dunque conoscere:
  - Il nome assegnato al file
  - Il nome della directory in cui esso è memorizzato
- Una volta identificato un file, è possibile accedere al suo contenuto
  - Leggere la sequenza di byte di cui è composto
  - Modificare la sequenza di byte di cui è composto

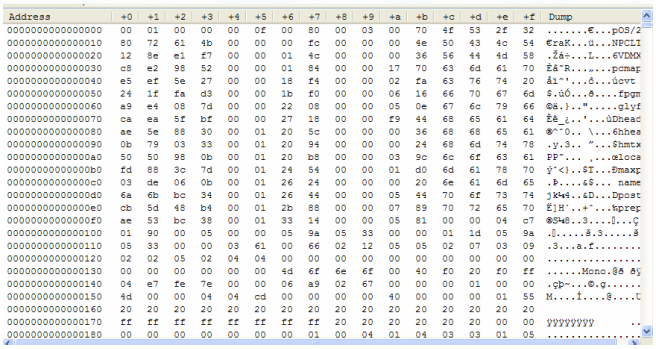
Operazioni permesse sui file

- Operazioni generiche
  - Cancellazione di un file esistente
  - Rinominazione di un file esistente
  - Copia di un file, creando un nuovo file con lo stesso nome in una diversa directory
  - Spostamento di un file, equivalente alla copia con cancellazione dell'originale
- Operazioni specifiche
  - Creazione di un nuovo file
  - Modifica del contenuto del file

Operazioni generiche o specifiche

- Le operazioni generiche sono solitamente svolte interagendo con il sistema operativo, e si possono applicare a qualsiasi file
- Le operazioni specifiche invece coinvolgono il contenuto del file, pertanto richiedono programmi specifici per ogni tipologia di file

File = Sequenza di byte



Address	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+a	+b	+c	+d	+e	+f	Dump
0000000000000000	00	01	00	00	00	0f	00	80	00	03	00	70	4f	53	2f	32	.....€...pOS/2
0000000000000010	80	72	61	4b	00	00	00	fc	00	00	00	4e	50	43	4c	54	€raK...ü...NPCLI
0000000000000020	12	8e	e1	f7	00	00	01	4c	00	00	00	36	56	44	4d	58	.Žá÷...L...6VDMX
0000000000000030	c8	e2	98	52	00	00	01	84	00	00	17	70	63	6d	61	70	Èâ˜R.....„....pcmap
0000000000000040	e5	ef	5e	27	00	00	18	f4	00	00	02	fa	63	76	74	20	åï^'...ô...úcvt
0000000000000050	24	1f	fa	d3	00	00	1b	f0	00	00	06	16	66	70	67	6d	$.úÓ...ð....fpgm
0000000000000060	a9	e4	08	7d	00	00	22	08	00	00	05	0e	67	6c	79	66	©ä.}.."......glyf
0000000000000070	ca	ea	5f	bf	00	00	27	18	00	00	f9	44	68	65	61	64	Êê_¿..'...ùDhead
0000000000000080	ae	5e	88	30	00	01	20	5c	00	00	00	36	68	68	65	61	®^ˆ0.. \...6hhea
0000000000000090	0b	79	03	33	00	01	20	94	00	00	00	24	68	6d	74	78	.y.3.. "...$hmtx
00000000000000a0	50	50	98	0b	00	01	20	b8	00	00	03	9c	6c	6f	63	61	PP˜... ¸...œloca
00000000000000b0	fd	88	3c	7d	00	01	24	54	00	00	01	d0	6d	61	78	70	ý‹<}...$T...Ðmaxp
00000000000000c0	03	de	06	0b	00	01	26	24	00	00	00	20	6e	61	6d	65	.Þ....&$... name
00000000000000d0	6a	6b	bc	34	00	01	26	44	00	00	05	44	70	6f	73	74	jk¼4..&D...Dpost
00000000000000e0	cb	5d	48	b4	00	01	2b	88	00	00	07	89	70	72	65	70	Ë]H´..+ˆ...‰prep
00000000000000f0	ae	53	bc	38	00	01	33	14	00	00	05	81	00	00	04	c7	®S¼8..3....Ç
0000000000000100	01	90	00	05	00	00	05	9a	05	33	00	00	01	1d	05	9a	.....š.3.....š
0000000000000110	05	33	00	00	03	61	00	66	02	12	05	05	02	07	03	09	.3...a.f........
0000000000000120	02	02	05	02	04	04	00	00	00	00	00	00	00	00	00	00	....
0000000000000130	00	00	00	00	00	00	4d	6f	6e	6f	00	40	f0	20	f0	ff	......Mono.@ð ðÿ
0000000000000140	04	e7	fe	7e	00	00	06	a9	02	67	00	00	00	01	00	00	.çþ~...©.g......
0000000000000150	4d	00	00	04	04	cd	00	00	00	00	40	00	00	00	01	55	M....Í....@....U
0000000000000160	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
0000000000000170	ff	ff	ff	ff	ff	ff	ff	ff	20	20	20	20	20	20	00	00	ÿÿÿÿÿÿÿÿ ...
0000000000000180	00	00	00	00	00	00	00	01	00	04	01	04	03	03	01	05	.....

Tipi di file

- Il significato dei byte memorizzati all'interno dei file è noto, solitamente, solamente al programma che lo ha creato
- Si parla di **tipi di file**, che sono solitamente indicati dall'**estensione** (ultime lettere del nome del file)
  - File prodotto da Microsoft Powerpoint → `.ppt`
  - File prodotto da Microsoft Word → `.doc`
  - File prodotto da Openoffice.Org Writer → `.odt`
  - ...

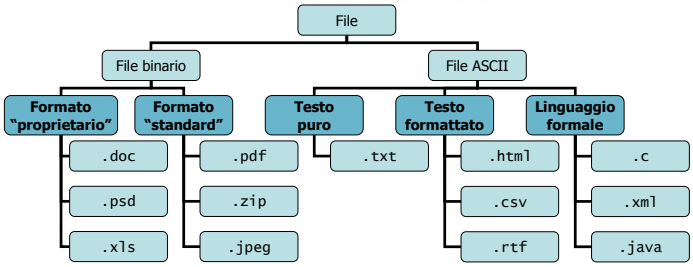
Conseguenza

- Non è possibile lavorare con file gestiti da altri programmi, a meno di non conoscere il formato del file
- **Eccezione**: se il formato del file è pubblicamente documentato, allora sarà possibile leggerlo e scriverlo interpretando correttamente i byte
  - Esempio: file di testo (ASCII)
    - La codifica ASCII è utilizzata in molti campi: testi (.txt), programmi in C (.c), pagine HTML (.html), ...
  - Esempio: file Acrobat (.pdf)
    - Struttura molto complessa, ma documentata

Definizione di file

File binari e file di testo

Vista d'insieme dei formati di file



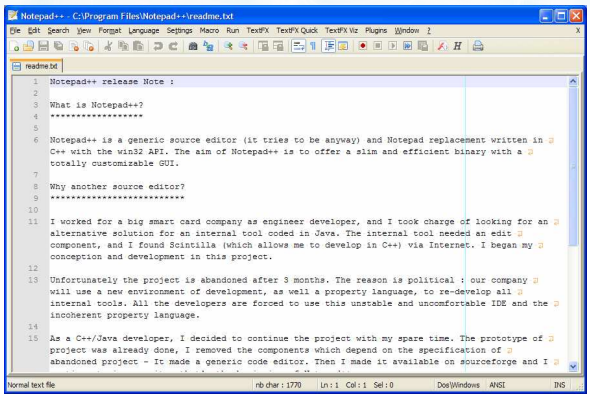
Differenze operative

- A livello di programmazione, esistono
  - Funzioni generali per accedere ai file
  - Funzioni specifiche per la lettura e la scrittura del contenuto di file binari
  - Funzioni specifiche per la lettura e la scrittura del contenuto di file ASCII
- Il corretto funzionamento di un programma dipende dalla perfetta conoscenza del formato del file stesso

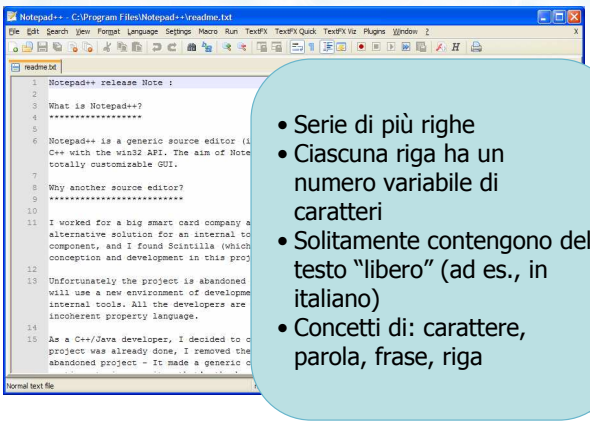
Scelta operativa

- In questo corso si tratteranno esclusivamente i file ASCII
  - Più semplici da comprendere
  - Facili da visualizzare e da creare con qualsiasi editor di testi
    - Notepad o molte sue alternative
    - L'ambiente di sviluppo in C
- Si userà indifferentemente la denominazione "file ASCII" o "file di testo"

File di "testo puro"

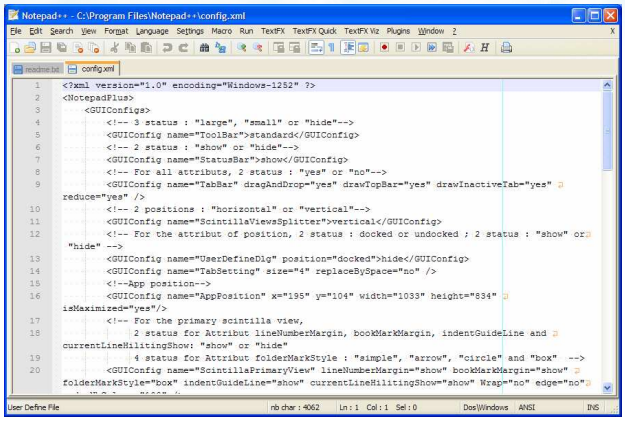


File di "testo puro"

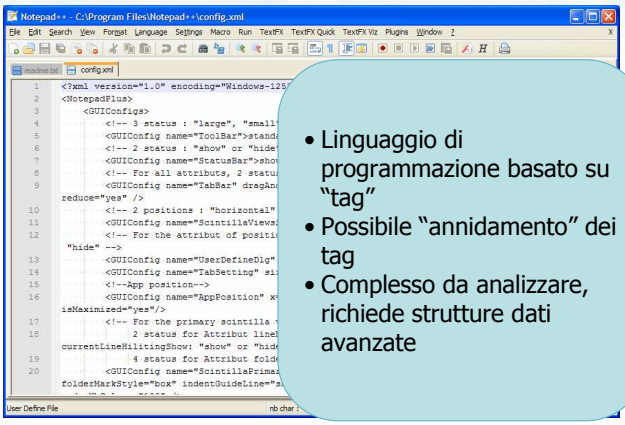


- Serie di più righe
- Ciascuna riga ha un numero variabile di caratteri
- Solitamente contengono del testo "libero" (ad es., in italiano)
- Concetti di: carattere, parola, frase, riga

File di "testo formattato" (es: XML)

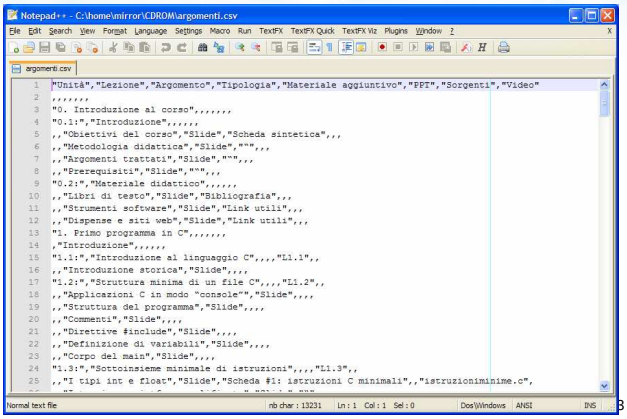


File di "testo formattato" (es: XML)

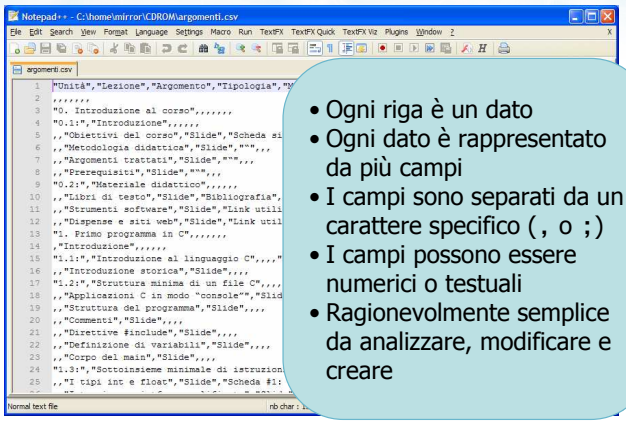


- Linguaggio di programmazione basato su "tag"
- Possibile "annidamento" dei tag
- Complesso da analizzare, richiede strutture dati avanzate

File di "testo formattato" (es: CSV)



File di "testo formattato" (es: CSV)

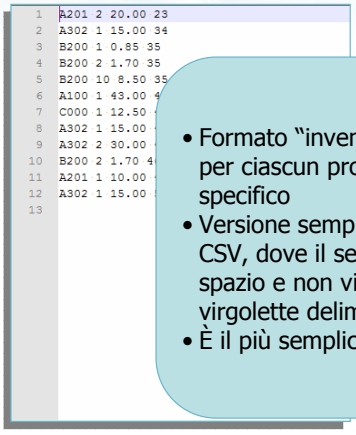


- Ogni riga è un dato
- Ogni dato è rappresentato da più campi
- I campi sono separati da un carattere specifico (, o ;)
- I campi possono essere numerici o testuali
- Ragionevolmente semplice da analizzare, modificare e creare

File di "testo formattato" (custom)

```
A201 2 20.00 23
A302 1 15.00 34
B200 1 0.85 35
B200 2 1.70 35
B200 10 8.50 35
A100 1 43.00 43
C000 1 12.50 44
A302 1 15.00 45
A302 2 30.00 46
B200 2 1.70 46
A201 1 10.00 47
A302 1 15.00 50
```


File di "testo formattato" (custom)



- Formato "inventato" ad hoc per ciascun programma specifico
- Versione semplificata del CSV, dove il separatore è lo spazio e non vi sono virgolette delimitatrici
- È il più semplice da gestire

I/O Avanzato e File

File di testo in C

File di testo in C

- Accesso ai file
- Funzioni `fopen/fclose`
- Funzioni `fget*/fput*`
- Funzioni `fprintf/fscanf`
- Condizione `feof`

File di testo in C

Accesso ai file

Accesso ai file (1/4)

- Un programma C può accedere ad alcuni file presenti sui dischi del calcolatore
  - File aperto: file al quale attualmente il programma ha accesso
  - File chiuso: file residente su disco al quale attualmente il programma non ha accesso

Accesso ai file (2/4)

- All'atto dell'apertura di un file, il programma deve dichiarare la modalità di accesso
  - Modalità di **lettura**: il programma può leggere il contenuto del file, ma non modificarlo
  - Modalità di **scrittura**: il programma può riscrivere da zero il contenuto del file
  - Modalità di **aggiunta**: il programma può aggiungere nuove informazioni al file
  - Modalità di **lettura/scrittura**: tutte le precedenti
- I successivi accessi al file devono essere compatibili con la modalità di accesso dichiarata

Accesso ai file (3/4)

- L'accesso ai file di testo è rigorosamente sequenziale
  - La lettura avviene dalla prima riga all'ultima, dal primo carattere all'ultimo
  - In scrittura, ogni riga o carattere scritto vengono posizionati dopo le righe o caratteri scritti in precedenza
    - A partire dal primo carattere, in modalità di scrittura
    - A partire dall'ultimo carattere esistente, in modalità di aggiunta

Accesso ai file (4/4)

- All'atto dell'apertura di un file, il programma deve dichiarare se il file è di tipo binario oppure di testo
  - La differenza consiste solamente nel trattamento "speciale" del carattere `'\n'` nel caso dei file di testo
  - In questo corso useremo sempre la modalità testuale

Stream associato ad un file

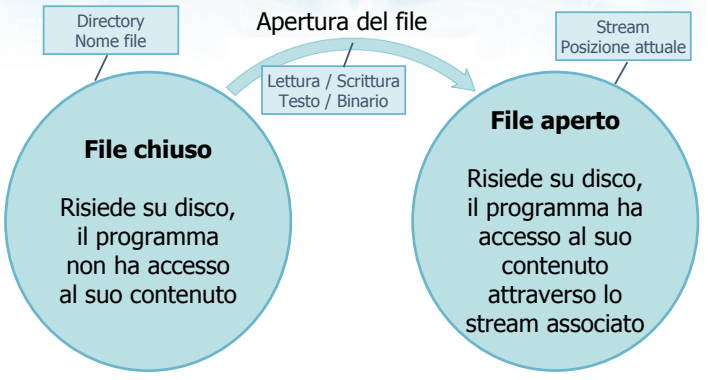
- In un programma C, esiste un tipo di dato specifico per rappresentare le informazioni relative ad un file aperto
  - Denominato: **file stream** (flusso associato ad un file)
  - Tipo di dato: `FILE *` (definito in `<stdio.h>`)
- "Aprire" un file significa quindi creare un nuovo stream ed associarlo ad uno specifico file sul disco

Significato di stream

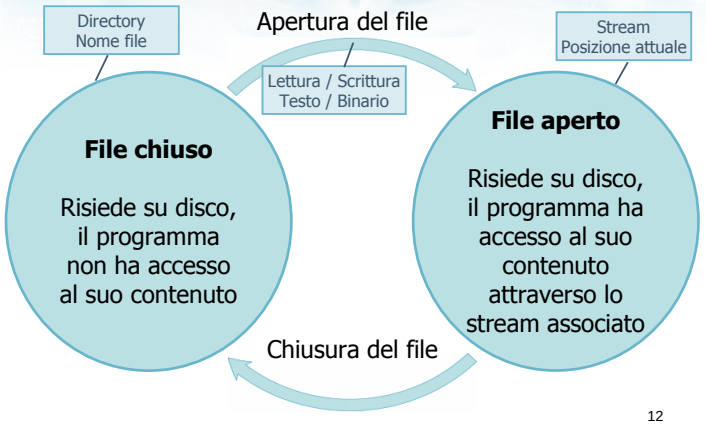
- Una volta che il file è aperto, il suo stream rappresenta
  - Un "collegamento" mediante il quale poter compiere delle operazioni sul contenuto del file
  - Le modalità di accesso scelte (testo/binario, lettura/scrittura/...)
  - La posizione attuale a cui si è arrivati nello scrivere o nel leggere il file
- Ogni operazione sul file avviene chiamando una funzione che riceve lo stream come parametro

Stati di un file

Stati di un file

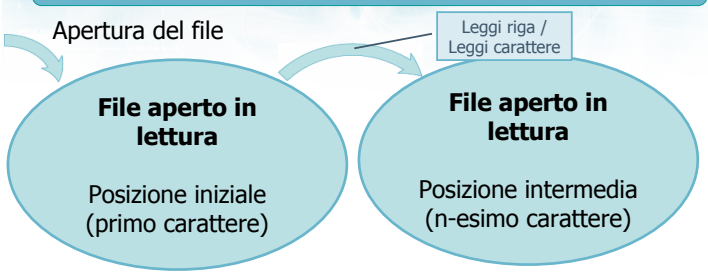


Stati di un file

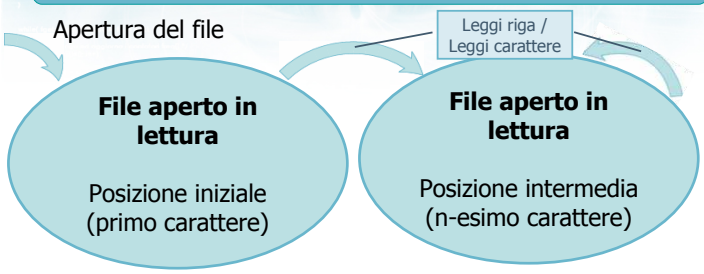


Lettura di un file

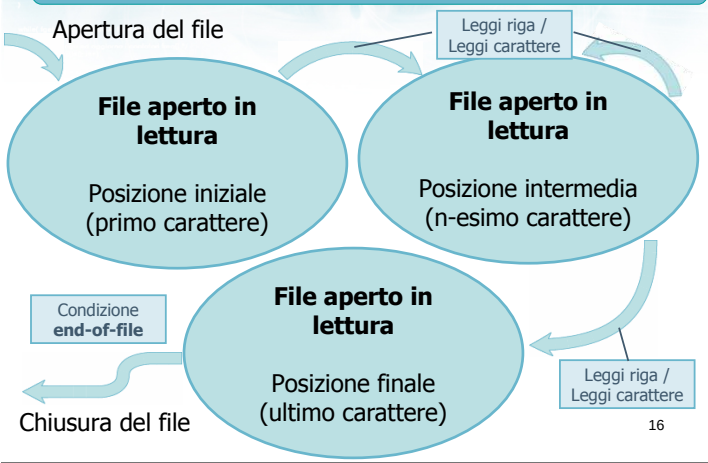
Lettura di un file



Lettura di un file

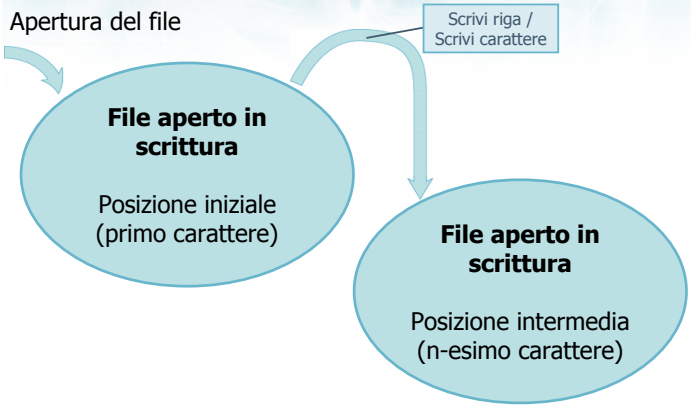


Lettura di un file

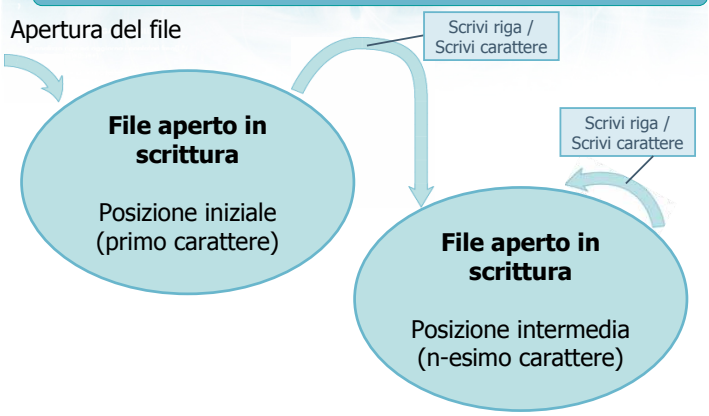


Scrittura di un file

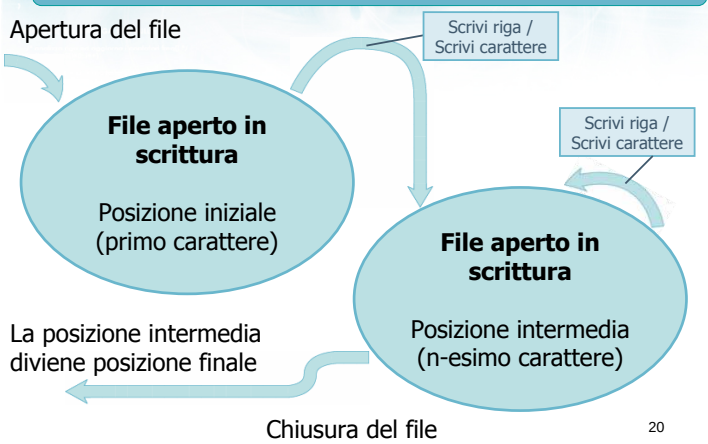
Scrittura di un file



Scrittura di un file

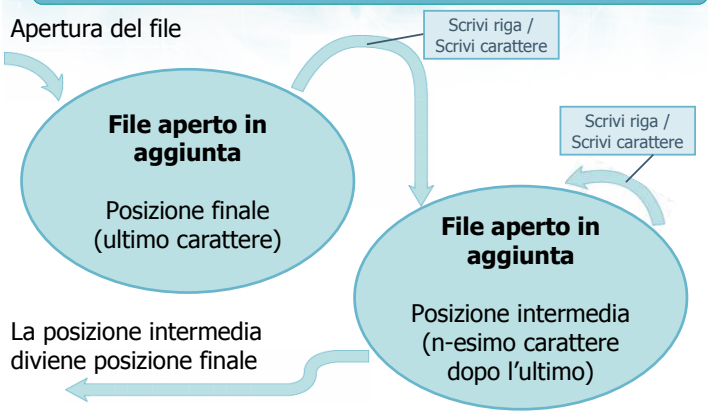


Scrittura di un file



Chiusura del file

Aggiunta ad un file



Chiusura del file

Funzioni C

- Tutte le funzioni per l'accesso ai file sono contenute in `<stdio.h>`
- Funzioni per apertura e chiusura: `fopen`, `fclose`
- Funzioni per la lettura: `fgetc`, `fgets`, `fscanf`
- Funzioni per la scrittura: `fputc`, `fputs`, `fprintf`
- Funzioni per lo stato del file: `feof`

File di testo in C

Funzioni `fopen/fclose`

Funzioni `fopen` e `fclose`

- Apertura del file (`fopen`):
  - La funzione `fopen` apre un file e restituisce una variabile stream
  - Richiede il nome del file e le modalità di apertura
  - Restituisce una nuova variabile di tipo `FILE *`
- Chiusura del file (`fclose`)
  - Quando il file non è più richiesto, si chiama la funzione `fclose` che chiude il file
  - Richiede come parametro lo stream, precedentemente restituito da `fopen`

fopen: sintassi

```
FILE * f ;

...

f = fopen( "nomefile", "modo" ) ;
```

Variabile stream di tipo FILE *

Stringa contenente il nome del file

Modalità di apertura (stringa)

Nome del file

```
f = fopen( "nomefile", "modo" ) ;
```

```
f = fopen( "dati.txt", "modo" ) ;
```


Nome del file

```
f = fopen( "nomefile", "modo" ) ;
```

```
f = fopen( "dati.txt", "modo" ) ;
```

```
f = fopen( "c:\\prog\\dati.txt",
           "modo" ) ;
```


Nome del file

```
f = fopen( "nomefile", "modo" ) ;
```

```
f = fopen( "dati.txt", "modo" ) ;
```

```
f = fopen( "c:\\prog\\dati.txt",
           "modo" ) ;
```

```
char nome[20] ;
gets(nome) ;
f = fopen( nome, "modo" ) ;
```


Nome del file

```
f = fopen( "nomefile", "modo" ) ;
```

```
f = fopen( "dati.txt", "modo" ) ;
```

```
f = fopen( "c:\\prog\\dati.txt",
           "modo" ) ;
```

```
char nome[20] ;
gets(nome) ;
f = fopen( nome, "modo" ) ;
```

```
f = fopen( argv[1], "modo" ) ;
```

Modo di apertura

```
f = fopen( "nomefile", "modo" ) ;
```

Modalità **lettura**, file di testo → `"rt"` `"r"`

Modalità **scrittura**, file di testo → `"wt"` `"w"`

Modalità **aggiunta**, file di testo → `"at"` `"a"`

Effetto della fopen (1/3)

- Modalità "r"
  - Se il file esiste, viene aperto ed f punta allo stream relativo, posizionato in lettura al primo carattere
  - Se il file non esiste, non viene creato nessuno stream e f==NULL

Effetto della fopen (2/3)

- Modalità "w"
  - Se il file non esiste, viene creato da zero ed f punta allo stream relativo, posizionato in scrittura al primo carattere
  - Se il file esiste già, viene innanzitutto cancellato e poi ricreato da zero, f punta allo stream relativo, posizionato in scrittura al primo carattere
  - Se non è possibile creare il file (perché la directory non esiste, o il disco è protetto in scrittura, ...), non viene creato nessuno stream e f==NULL

Effetto della fopen (3/3)

- Modalità "a"
  - Se il file non esiste, viene creato da zero ed f punta allo stream relativo, posizionato in scrittura al primo carattere
  - Se il file esiste già, non viene modificato, f punta allo stream relativo, posizionato in scrittura dopo l'ultimo carattere
  - Se non è possibile creare o modificare il file (perché la directory non esiste, o il disco è protetto in scrittura, ...), non viene creato nessuno stream e f==NULL

Controllo dell'errore

```
FILE * f ;

...

f = fopen( "nomefile", "r" ) ;
if( f == NULL )
{
  printf("Impossibile aprire file\n");
  exit(1) ;
}
```


Suggerimento

- Ogniqualvolta viene chiamata la funzione fopen, è indispensabile subito dopo controllare se il valore ritornato non è NULL
- È da considerarsi **errore** una chiamata ad fopen di cui non venga controllato il risultato
- In caso di errore, solitamente conviene interrompere il programma segnalando un codice di errore
  - Esempio: exit(1) ;

fclose: sintassi

```
FILE * f ;
...
f = fopen( "nomefile", "modo" ) ;

.../* accesso al file */

fclose(f) ;
```

Variabile stream

Avvertenze

- La funzione `fclose` può essere chiamata solamente su stream correttamente aperti
  - Mai chiamare `fclose` se `f==NULL`
- Dopo la chiusura del file, non è più possibile accedere allo stream
  - Eventualmente, ri-aprirlo nuovamente

Controllo dell'errore

```
int ris ;
...

ris = fclose(f) ;
if(ris!=0)
{
  printf("Impossibile chiudere\n") ;
  exit(1) ;
}
```


Suggerimento

- Conviene definire due funzioni aggiuntive, che chiamino le funzioni di libreria `fopen` e `fclose` e addizionalmente compiano i controlli d'errore
- Chiameremo `myfopen` e `myfclose` tali funzioni
- Nei programmi chiameremo sempre `myfopen` e `myfclose`, e mai direttamente le funzioni di libreria

Funzione `myfopen`

```
FILE * myfopen(char *name, char *mode)
{
  FILE * f ;

  f = fopen( name, mode ) ;
  if (f==NULL)
  {   printf("Impossibile aprire %s\n",
              name) ;
      exit(1) ;
  }
  return f ;
}
```

Funzione `myfclose`

```
int myfclose(FILE *f)
{ int ris ;

  if (f==NULL)
  { printf("ERRORE INTERNO\n") ;
    exit(1) ;
  }
  ris = fclose(f) ;
  if( ris!=0 )
  { printf("Impossibile chiudere\n") ;
    exit(1) ;
  }
  return ris ;
}
```

File di testo in C

Funzioni `fget*`/`fput*`

Lettura e scrittura su file

	Lettura	Scrittura
Carattere singolo	`fgetc`	`fputc`
Riga intera	`fgets`	`fputs`

Lettura e scrittura su file

	Lettura	Scrittura
Carattere singolo	`fgetc`	`fputc`
Riga intera	`fgets`	`fputs`

Legge prossimo elemento, fino alla fine del file

Scrive o aggiunge

Lettura e scrittura su file

	Lettura	Scrittura
Carattere singolo	`fgetc`	`fputc`
Riga intera	`fgets`	`fputs`

Parametro: `char`

Parametro: stringa

Legge prossimo elemento, fino alla fine del file

Scrive o aggiunge

`fgetc`: sintassi

```
int ch ;

ch = fgetc(f) ;
```

Stream aperto in lettura

Prossimo carattere del file; `EOF` se il file è finito

`fputc`: sintassi

```
int ch ;

fputc(ch, f) ;
```

Stream aperto in scrittura o in aggiunta

Carattere da aggiungere al file

`fgets`: sintassi

```
char str[80] ;

fgets(str, 79, f) ;
```

Max numero di caratteri letti

Stream aperto in lettura

Stringa nella quale viene letta la prossima riga del file (fino al \n compreso)

Fine del file

- La funzione `fgets` restituisce un valore `NULL` se ha tentato di leggere oltre la fine del file

```
char str[80] ;

while( fgets(str, 79, f) != NULL )
{
    /* elabora str */
}
```


fputs: sintassi

```
char str[80] ;

fputs(str, f) ;
```

Stream aperto in scrittura o in aggiunta

Stringa da aggiungere al file (solitamente termina con \n)

Esercizio: "Frequenza lettere"

- Sia dato un file di testo, contenente dei brani scritti da un utente
- Si scriva un programma in C che acquisisca sulla linea di comando il nome di tale file, e che stampi le frequenze con cui compaiono le varie lettere dell'alfabeto
- Si considerino equivalenti le maiuscole e le minuscole, e si ignorino i caratteri di spaziatura e punteggiatura

Analisi (1/2)

`manzoni.txt`

```
Quel ramo del lago di Como,
che volge a mezzogiorno,
tra due catene non interrotte di monti,
tutto a seni e a golfi,
a seconda dello sporgere e del
rientrare di quelli, vien, quasi
a un tratto, a ristringersi, e a
prender corso e figura di fiume,
tra un promontorio a destra,
e un'ampia costiera dall'altra parte
```


Analisi (2/2)

`manzoni.txt`

```
Quel ramo del lago di Como,
che volge a mezzogiorno,
tra due catene non interrotte di monti,
tutto a seni e a golfi,
a seconda dello sporgere e del
rientrare di quelli, vien, quasi
a un tratto, a ristringersi, e a
prender corso e figura di fiume,
tra un promontorio a destra,
e un'ampia costiera dall'altra parte
```

Prompt dei comandi

```
C:\prog>frequenza manzoni.txt
A : 26
B : 0
C : 6
D : 12
E : 32
F : 3
G : 7
H : 1
I : 21
J : 0
```


Soluzioni possibili

- Occorre calcolare un vettore di frequenze, in cui ciascuna posizione rappresenti la frequenza di ciascuna delle lettere alfabetiche
- Ci sono due approcci possibili alla lettura del file:
  - Soluzione **per caratteri**: il file viene letto un carattere alla volta, usando la funzione `fgetc`
  - Soluzione **per righe**: il file viene letto una riga alla volta, usando la funzione `fgets`, e tale riga viene poi esaminata con un ciclo interno al programma

Soluzione 1: per caratteri (1/3)

```
const int LETT = 26 ;
int freq[LETT] ; /* frequenze lettere */

FILE * f ;
int ch, i ;

if (argc!=2)
{
  printf("Numero argomenti errato\n") ;
  exit(1) ;
}

for(i=0; i<LETT; i++)
    freq[i] = 0 ;
```

frequenza.c

Soluzione 1: per caratteri (2/3)

```
f = myfopen( argv[1], "r" ) ;

ch = fgetc( f ) ;
while( ch!=EOF )
{
  if(isalpha(ch))
  {
    i = toupper(ch)-'A' ;
    /* posizione 0..25 della lettera */
    freq[i]++ ;
  }
  ch = fgetc( f ) ;
}
myfclose( f ) ;
```

frequenza.c

Soluzione 1: per caratteri (3/3)

```
for(i=0; i<LETT; i++)
{
  printf("%c : %d\n", i+'A', freq[i]) ;
}

exit(0) ;
```

frequenza.c

Soluzione 2: per righe

```
const int LUN = 200 ;
char riga[LUN+1];
...

while( fgets( riga, LUN, f ) != NULL )
{
  for(i=0; riga[i]!=0; i++)
  {
    if(isalpha(riga[i]))
    {
      freq[toupper(riga[i])-'A']++ ;
    }
  }
}
```

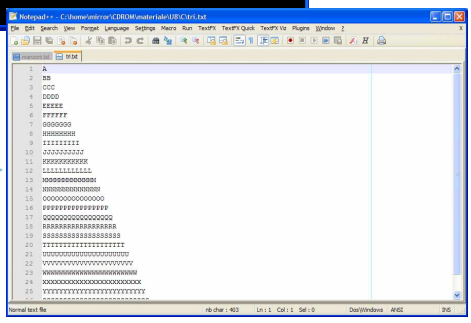
frequenza.c

Esercizio: "Triangolo alfabetico"

- Si realizzi un programma in C che crei un file di testo contenente tutte le lettere dell'alfabeto, con una disposizione "a triangolo"
  - La prima riga contiene una volta la lettera A
  - La seconda riga contiene 2 volte la lettera B
  - La terza riga contiene 3 volte la lettera C
  - ...
- Il nome del file viene passato come primo argomento sulla linea di comando

Analisi

```
c:\prog>triangolo tri.txt
```



Soluzione (1/2)

triangolo.c

```
FILE * f ;
int i, j ;
char ch ;

if (argc!=2)
{
  printf("Numero argomenti errato\n") ;
  exit(1) ;
}

f = myfopen( argv[1], "w" ) ;
```

Soluzione (2/2)

triangolo.c

```
for(i=0; i<26; i++)
{
  ch = i+'A' ;

  for(j=0; j<=i; j++)
    fputc( ch, f ) ;

  fputc( '\n', f ) ;
}

myfclose( f ) ;

exit(0) ;
```

File di testo in C

Funzioni fprintf/fscanf

Output formattato

- Qualora sia necessario creare file con più campi nella stessa riga, è scomodo ricorrere alle funzioni `fputc`/`fputs`
- È possibile utilizzare una variante della funzione `printf`, operante su uno stream aperto in scrittura
  - `fprintf(f, "formato", x, y, z) ;`

fprintf: sintassi

```
FILE * f ;

fprintf(f, "formato", variabili ) ;
```

Stream aperto in scrittura o in aggiunta

Elenco delle variabili da scrivere

Formato dei dati da stampare, usando gli stessi specificatori validi per `printf`

Input formattato

- Qualora sia necessario leggere file con più campi nella stessa riga
  - È scomodo ricorrere alla funzione `fgetc`
  - Il risultato della funzione `fgets` deve successivamente essere analizzato
- È possibile utilizzare una variante della funzione `scanf`, operante su uno stream aperto in lettura
  - `fscanf(f, "formato", &x, &y, &z) ;`

fscanf: sintassi

```
FILE * f ;

fscanf(f, "formato", &variabili ) ;
```

Stream aperto in lettura

Puntatori alle variabili da leggere

Formato dei dati da leggere, usando gli stessi specificatori validi per `scanf`

fscanf: una funzione pericolosa

- Nonostante la funzione `fscanf` sia prevista dalla libreria standard C, è considerata una funzione **pericolosa** nella lettura di file in generale
- In particolare, qualora il file non sia nel formato corretto (file contenente errori), allora il meccanismo di funzionamento di `fscanf` rende **impossibile** acquisire i dati in modo **affidabile**
- Suggerimento: **non usare mai** `fscanf`
- Nella prossima lezione vedremo una **soluzione robusta** al problema

File di testo in C

Condizione `feof`

End-of-File

- Un file aperto in lettura è inizialmente posizionato al primo carattere
- Ad ogni successiva lettura, avanza la posizione corrente all'interno del file
- Quando è stato letto l'ultimo carattere (o l'ultima riga) del file, non sono possibili ulteriori letture
  - In questo caso si dice che si è verificata una condizione di End-of-File (`EOF`)
  - Ulteriori tentativi di lettura genererebbero una condizione di errore

Tentativi di lettura

- Se si tenta di leggere oltre l'End-of-File
  - `fgets` restituisce `NULL`
  - `fgetc` restituisce `EOF`
  - `fscanf` restituisce `EOF`
- È possibile controllare tali valori di ritorno per controllare la fine del file
  - In tali casi, l'errore è già avvenuto, e l'operazione di lettura non è andata a buon fine

Funzione `feof`

- La funzione `feof` è specificatamente utile per verificare se uno stream `f` è già nella condizione di End-of-File **prima** di tentare operazioni di lettura
  - `if ( !feof( f ) ) { ... }`
- La funzione, partendo dallo stream `f`, restituisce:
  - **Vero**, se lo stream **è** già in End-of-File, e quindi le successive letture falliranno
  - **Falso**, se lo stream **non è** ancora in End-of-File, e quindi sono possibili ulteriori letture

Esempio

```
ch = fgetc( f ) ;

while( ch!=EOF )
{
   .../* elabora ch */

   ch = fgetc( f ) ;
}
```

```
while( !feof(f) )
{
   ch = fgetc( f ) ;

   .../* elabora ch */
}
```

I/O Avanzato e File

Input robusto

Input robusto

- Problemi nella lettura da file
- Soluzione basata su `fgetc`
- Funzione `sscanf`
- Soluzione basata su `fgets`

Input robusto

Problemi nella lettura da file

Lettura da file

- I file di testo contengono dati secondo un certo **formato**
- È semplice scrivere un programma in grado di leggere un file formattato correttamente
- Diviene molto complesso gestire eventuali **errori** di formato del file

Errori di formato

- Righe contenenti un numero errato di elementi
  - Elementi in eccesso
    - In fondo
    - All'inizio o in mezzo
  - Elementi in difetto
- Tipi di dato errati
  - Caratteri, stringhe, interi, reali
- Errori di coerenza interna

Esempio

- Un file di testo contiene i PIN dei bancomat dei membri di una famiglia
- Il file di testo contiene sulla prima riga il numero di bancomat descritti nel file
- Le righe successive contengono le informazioni su ciascun bancomat, uno per riga
- Ciascuna riga contiene 3 campi separati da spazi:
  - Il nome del proprietario del bancomat
  - Il numero del bancomat
  - Il PIN segreto (5 cifre)

Esempio di file corretto

bancomat.txt

```
3
Aldo 123456789 12762
Giovanni 334422445 97864
Giacomo 887868083 32552
```


Definizioni

```
const char nomefile[] = "banco.txt" ;
const int MAX = 20 ;
          /* numero max di bancomat */
const int LUN = 15 ;
          /* lunghezza del nome */

int N ;
char nome[MAX][LUN+1] ;
int numero[MAX] ;
int pin[MAX] ;

FILE * f ;
int i ;
```

banco-bad.c

Lettura del file (solo se corretto)

```
f = myfopen(nomefile, "r") ;

fscanf(f, "%d", &N) ;

for(i=0; i<N; i++)
{
  fscanf(f, "%s %d %d",
   nome[i], &numero[i], &pin[i]) ;
}

myfclose(f) ;
```

banco-bad.c

Possibili errori nel file (1/3)

```
3
Aldo 123456789 12762
334422445 97864
Giacomo 887868083 32552
```

Campo mancante

Possibili errori nel file (1/3)

```
3
Aldo 123456789 12762
334422445 97864
Giacomo 887868083 32552
```

Campo mancante

```
3
Aldo 3212 123456789 12762
Giovanni 334422445 97864
Giacomo 887868083 32552
```

Campo extra nella riga

Possibili errori nel file (2/3)

```
3
Aldo 123456789 12762 3212
Giovanni 334422445 97864
Giacomo 887868083 32552
```

Campo extra a fine riga

Possibili errori nel file (2/3)

```
3
Aldo 123456789 12762 3212
Giovanni 334422445 97864
Giacomo 887868083 32552
```

Campo extra a fine riga

```
3
Aldo 123456789 12762
Giovanni 334422445 97864
Giacomo A32Z4324 32552
```

Tipi di dato errati

Possibili errori nel file (3/3)

```
3
Pier Aldo 123456789 12762
Giovanni 334422445 97864
Giacomo 887868083 32552
```

Spazi

Possibili errori nel file (3/3)

```
3
Pier Aldo 123456789 12762
Giovanni 334422445 97864
Giacomo 887868083 32552
```

Spazi

```
3
Aldo 123456789 12762
Giacomo 887868083 32552
```

Incoerenza interna

Osservazioni

- Di fronte a qualsiasi errore di formato, la funzione `fscanf`
  - Perde il "sincronismo" con le righe del file
  - "Blocca" la lettura in caso di stringhe incontrate quando si aspettava un numero
  - Non si "accorge" dell'End-of-File
- La funzione `fscanf` non è sufficientemente robusta per gestire la lettura da file di testo

Input robusto

Soluzione basata su `fgetc`

Lettura basata su `fgetc`

- Dovendo evitare l'utilizzo della funzione `fscanf`, si potrebbe optare per la funzione `fgetc`
- L'adozione di `fgetc` risolve i problemi di sincronizzazione e di lettura dei dati errati, ma introduce spesso una complessità eccessiva nel programma

Soluzioni basate su fgetc (1/4)

- Acquisizione di una stringa

```
char s[MAX] ;

i = 0 ;
ch = fgetc(f) ;
while( ch != EOF  && ch != '\n'
       && ch != ' ' && i < MAX-1 )
{
    s[i] = ch ;
    i++ ;
    ch = fgetc(f) ;
}

s[i] = 0 ; /* terminatore nullo */
```

Soluzioni basate su fgetc (2/4)

- Saltare tutti gli spazi (ma non gli a-capo)

```
ch = fgetc(f) ;
while( ch != EOF  && ch != '\n' &&
       ch == ' ' )
{
    ch = fgetc(f) ;
}
```

Soluzioni basate su fgetc (3/4)

- Acquisizione di un intero positivo

```
char s[MAX] ;

i = 0 ;
ch = fgetc(f) ;
while( ch != EOF  && isdigit(ch)
       && i < MAX-1 )
{
    s[i] = ch ;
    i++ ;
    ch = fgetc(f) ;
}
s[i] = 0 ; /* terminatore nullo */

x = atoi(s) ; /* converti in int */
```

Soluzioni basate su fgetc (4/4)

- Sono possibili tutti i controlli personalizzati, su
  - Lunghezza minima e massima dei campi
  - Tipo di caratteri permessi
- Alcuni tipi di dati sono complessi da acquisire
  - Intero relativo: -124
  - Numero reale: -3.14e+21
- Soluzione in generale completa, ma molto lavoro manuale
- Rischio: dimenticare alcuni casi particolari

Suggerimento

- Utilizzare la funzione fgetc quando occorre leggere dei testi in "formato libero"
  - Esempio: statistiche sul testo
  - Esempio: file di una sola riga
- Per file in formato custom, contenenti campi prefissati e/o di tipo numerico, occorre una soluzione più comoda

Input robusto

Funzione sscanf

Funzione sscanf

- La risposta a molti dei problemi sollevati viene da una nuova funzione di libreria: `sscanf`
- Tale funzione si può usare per analizzare il contenuto di una stringa, estraendone vari campi e memorizzandoli in variabili distinte
- Ha tutta la funzionalità di `scanf` e `fscanf`, ma lavora soltanto all'interno dei caratteri contenuti in una stringa
  - Potente e sicura

sscanf: sintassi

```
char str[80] ;

sscanf(str, "formato", &variabili ) ;
```

Stringa di caratteri

Puntatori alle variabili da leggere

Formato dei dati da leggere, usando gli stessi specificatori validi per `scanf`

Esempio

```
char str[80] ;
char nome[80] ;
int numero, pin ;

strcpy(str, "Aldo 91243213 1234\n");

sscanf(str, "%s %d %d",
       nome, &numero, &pin ) ;
```


Gestione degli errori

- La funzione `sscanf` non potrà mai leggere le righe successive di un file, in quanto la sua visibilità è confinata alla stringa passata
- Gli eventuali campi in eccesso a fine riga vengono quindi ignorati automaticamente
- Gli eventuali campi mancanti o di formato errato causano il mancato riconoscimento di quelli successivi
  - Condizione di errore facile da verificare analizzando il valore di ritorno di `sscanf`

Valore di ritorno

- La funzione sscanf restituisce al chiamante un valore intero:
  - Il valore è pari al numero di argomenti (specificatori %) correttamente riconosciuti e memorizzati nelle rispettive variabili

```
r = sscanf(str, "%s %d %d",
       nome, &numero, &pin ) ;
```


Esempio

```
char str[80] ;
char nome[80] ;
int numero, pin ;
int r ;

strcpy(str, "Aldo 91243213 1234\n");

r = sscanf(str, "%s %d %d",
       nome, &numero, &pin ) ;

if( r != 3 )
{ ... errore ... }
```

Suggerimenti

- Utilizzare **sempre** `sscanf` per analizzare una stringa
- Controllare **sempre** il valore di ritorno
- Non utilizzare più la funzione `atoi`, sostituirla con `sscanf(..."%d"...)`
- Per acquisire dati da tastiera, combinare con `gets`
- Per acquisire dati da file, combinare con `fgets`
- Nella prossima lezione vedremo come "istruire" `sscanf` a riconoscere formati più complessi

Input robusto

Soluzione basata su `fgets`

Input robusto da file di testo

- Affidiamo diversi ruoli alle varie funzioni
- `fgets`
  - Lettura del file riga per riga
  - Limite alla lunghezza max delle righe
  - Riconoscimento End-of-File
- `sscanf`
  - Analisi dei campi presenti in una riga
  - Controllo della correttezza del formato
  - Trasferimento nelle variabili/vettori del programma

Schema consigliato

```
const int LUNRIGA = 200 ;
int r, nr ;
char riga[LUNRIGA+1] ;

f = myfopen(nomefile, "r") ;

 /* Ciclo di lettura */

myfclose(f) ;
```


Ciclo di lettura

```
nr = 0 ;
while( fgets(riga, LUNRIGA, f)!=NULL )
{
  r = sscanf(riga, "%s %d %d",
             nome, &numero, &pin) ;
  if( r == 3 )
  {
    /* ...elabora la riga... */
  }
  else
    printf("Riga %d ignorata\n", nr+1);
  nr++ ;
}
```


Soluzione corretta (1/6)

```
const char nomefile[]="banco.txt";
const int MAX = 20 ;
const int LUN = 15 ;
const int LUNRIGA = 200 ;

int N ;
char nome[MAX][LUN+1] ;
int numero[MAX] ;
int pin[MAX] ;

FILE * f ;
int i, r, nr ;
char riga[LUNRIGA+1] ;
```

banco-ok.c

Soluzione corretta (2/6)

```
f = myfopen(nomefile, "r") ;
if(fgets(riga, LUNRIGA, f)==NULL)
{
  printf("Errore: file vuoto\n") ;
  exit(1) ;
}
r = sscanf(riga, "%d", &N) ;
if(r!=1)
{
  printf("Errore: La prima riga "
     "non contiene il numero\n");
  exit(1) ;
}
```

banco-ok.c

Soluzione corretta (3/6)

```
if( N<1 || N>MAX )
{
  printf("Errore: Num. bancomat "
      "%d non valido\n", N) ;
  printf("Valori ammessi: "
      "da 1 a %d\n", MAX) ;
  exit(1) ;
}
```

banco-ok.c

Soluzione corretta (4/6)

```
i = 0 ;
nr = 0 ;
while( fgets( riga, LUNRIGA, f )
          != NULL )
{
  if(i==N)
  {
    printf("Errore: troppe "
          "righe nel file\n" ) ;
    exit(1) ;
  }
```

banco-ok.c

Soluzione corretta (5/6)

```
  r = sscanf(riga, "%s %d %d",
    nome[i], &numero[i], &pin[i]);

  if( r == 3 )
    i++ ;
  else
  {
    printf("Riga %d ignorata\n",
          nr) ;
  }
  nr++ ;
}
```

banco-ok.c

Soluzione corretta (6/6)

```
if( i != N )
{
  printf("Errore: poche righe "
    " nel file\n" ) ;
  exit(1) ;
}

myfclose(f) ;
```

banco-ok.c

Conclusioni

- Prevedere tutti i possibili errori è difficile e pesante
  - La maggior parte delle linee di codice è dedicata alla gestione degli errori o delle anomalie
- Gli strumenti offerti dalla "coppia" `fgets` + `sscanf` sono validi ed efficaci

I/O Avanzato e File

Formattazione avanzata

Formattazione avanzata

- Modificatori di formato in output
- Modificatori di formato in input
- Stream predefiniti

Formattazione avanzata

Modificatori di formato in output

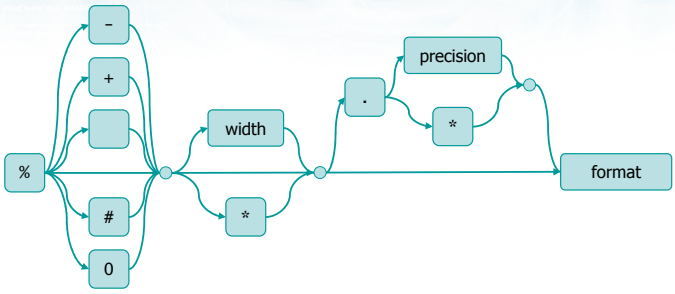
Formattazione dell'output

- L'output (su schermo o su file) viene formattato solitamente mediante la funzione `printf` (o `fprintf`)
- Ogni dato viene stampato attraverso un opportuno specificatore di formato (codici %)
- Ciascuno di questi codici dispone di ulteriori opzioni per meglio controllare la formattazione
  - Stampa incolonnata
  - Numero di cifre decimali
  - Spazi di riempimento
  - ...

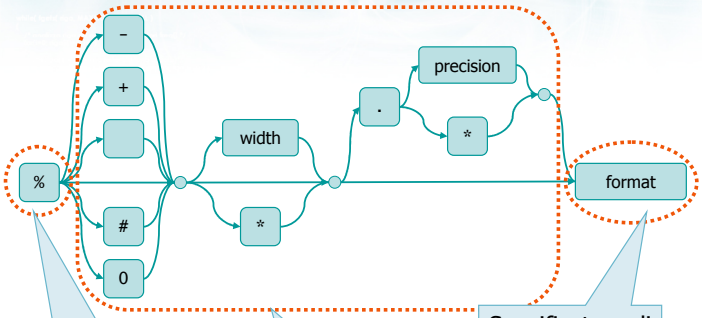
Specificatori di formato

Tipo	printf
char	%c %d
int	%d
short int	%hd %d
long int	%ld
unsigned int	%u %o %x
unsigned short int	%hu
unsigned long int	%lu
float	%f %e %g
double	%f %e %g
char []	%s

Forma completa degli specificatori



Forma completa degli specificatori

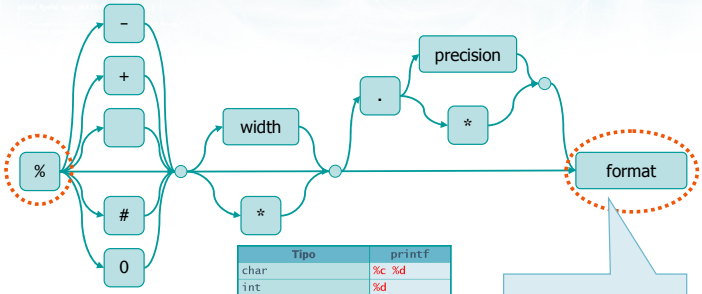


% obbligatorio

Modificatori opzionali

Specificatore di formato obbligatorio

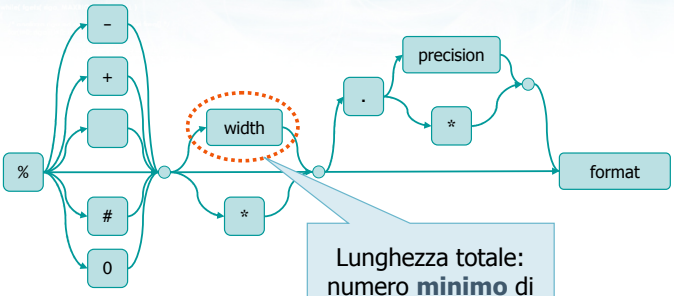
Forma completa degli specificatori



Tipo	printf
char	%c %d
int	%d
short int	%hd %d
long int	%ld
unsigned int	%u %o %x
unsigned short int	%hu
unsigned long int	%lu
float	%f %e %g
double	%f %e %g
char []	%s

Specificatori già noti

Forma completa degli specificatori

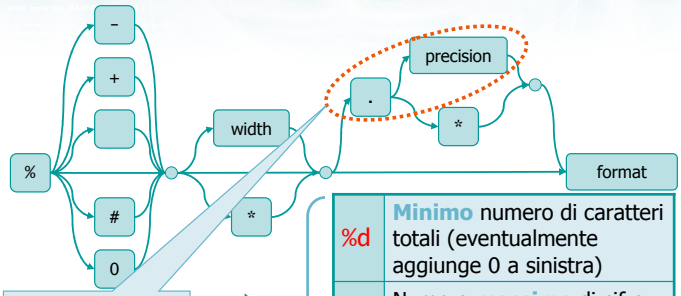


Lunghezza totale: numero **minimo** di caratteri stampati

Esempi

Istruzione	Risultato
`printf("%d",  13) ;`	13
`printf("%1d", 13) ;`	13
`printf("%3d", 13) ;`	␣13
`printf("%f",  13.14) ;`	13.140000
`printf("%6f", 13.14) ;`	13.140000
`printf("%12f", 13.14) ;`	␣␣13.140000
`printf("%6s", "ciao") ;`	␣␣ciao

Forma completa degli specificatori



Precisione (dipende...)

%d	**Minimo** numero di caratteri totali (eventualmente aggiunge 0 a sinistra)
%f	Numero **massimo** di cifre dopo la virgola
%s	**Massimo** numero di caratteri (stringhe più lunghe vengono troncate)

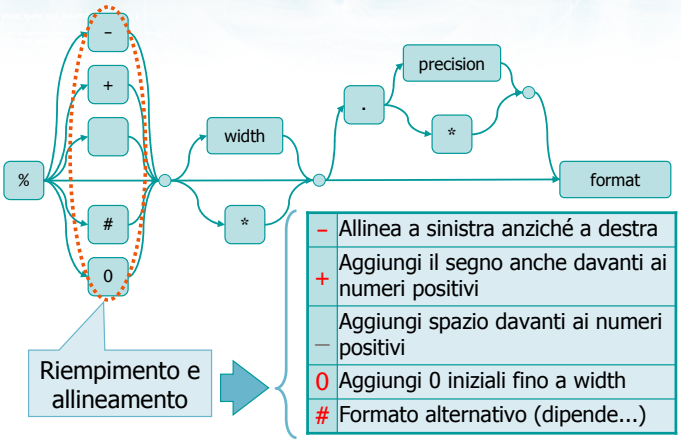
Esempi (1/2)

Istruzione	Risultato
`printf("%.1d",  13) ;`	13
`printf("%.4d", 13) ;`	0013
`printf("%6.4d", 13) ;`	␣␣0013
`printf("%4.6d", 13) ;`	000013
`printf("%.2s", "ciao") ;`	ci
`printf("%.6s", "ciao") ;`	ciao
`printf("%6.3s", "ciao") ;`	␣␣␣cia

Esempi (2/2)

Istruzione	Risultato
`printf("%.2f", 13.14) ;`	13.14
`printf("%.4f", 13.14) ;`	13.1400
`printf("%6.4f", 13.14) ;`	13.1400
`printf("%9.4f", 13.14) ;`	␣13.1400

Forma completa degli specificatori



Riempimento e allineamento

-	Allinea a sinistra anziché a destra
+	Aggiungi il segno anche davanti ai numeri positivi
␣	Aggiungi spazio davanti ai numeri positivi
0	Aggiungi 0 iniziali fino a width
#	Formato alternativo (dipende...)

Esempi (1/2)

Istruzione	Risultato
`printf("%6d", 13) ;`	␣␣␣␣13
`printf("%-6d", 13) ;`	13␣␣␣␣
`printf("%06d", 13) ;`	000013
`printf("%6s", "ciao") ;`	␣␣ciao
`printf("%-6s", "ciao") ;`	ciao␣␣

Esempi (2/2)

Istruzione	Risultato
`printf("%d", 13) ;`	13
`printf("%d", -13) ;`	-13
`printf("%+d", 13) ;`	+13
`printf("%+d", -13) ;`	-13
`printf("% d", 13) ;`	␣13
`printf("% d", -13) ;`	-13

Formattazione avanzata

Modificatori di formato in input

Approfondimenti su scanf

- Tipologie di caratteri nella stringa di formato
- Modificatori degli specificatori di formato
- Valore di ritorno
- Specificatore %[]

Stringa di formato (1/2)

- Caratteri stampabili:
  - `scanf` si aspetta che tali caratteri compaiano esattamente nell'input
  - Se no, interrompe la lettura
- Spaziatura ("whitespace"):
  - Spazio, tab, a capo
  - `scanf` "salta" ogni (eventuale) sequenza di caratteri di spaziatura
  - Si ferma al primo carattere non di spaziatura (o End-of-File)

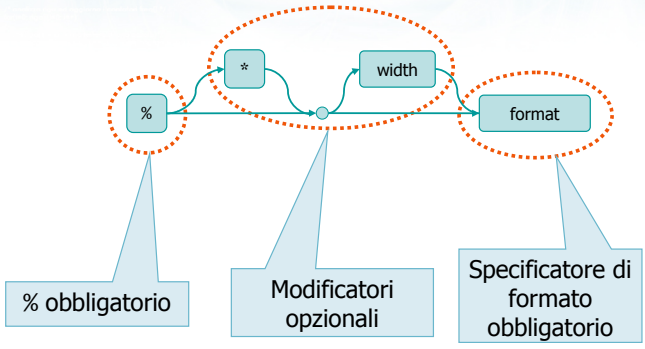
Stringa di formato (2/2)

- Specificatori di formato (%-codice):
  - Se il codice non è %c, innanzitutto `scanf` "salta" ogni eventuale sequenza di caratteri di spaziatura
  - `scanf` legge i caratteri successivi e *cerca* di convertirli secondo il formato specificato
  - La lettura si interrompe al primo carattere che non può essere interpretato come parte del campo

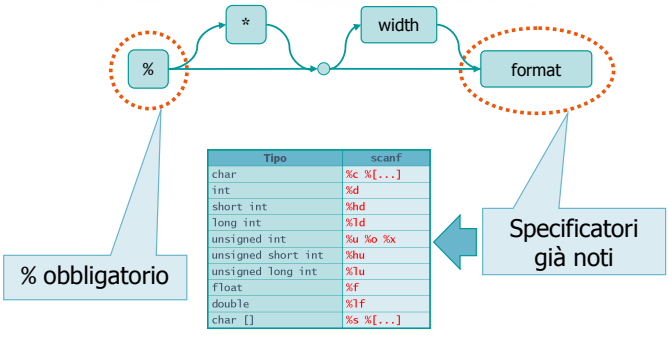
Specificatori di formato

Tipo	scanf
char	%c %[...]
int	%d
short int	%hd
long int	%ld
unsigned int	%u %o %x
unsigned short int	%hu
unsigned long int	%lu
float	%f
double	%lf
char []	%s %[...]

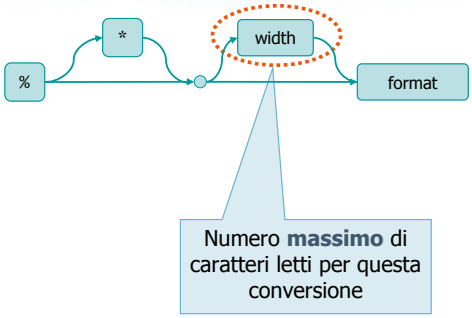
Forma completa degli specificatori



Forma completa degli specificatori



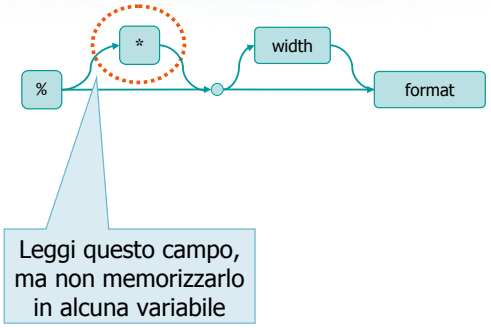
Forma completa degli specificatori



Esempi

Istruzione	Input	Risultato
`scanf("%d", &x) ;`	134xyz	x = 134
`scanf("%2d", &x) ;`	134xyz	x = 13
`scanf("%s", v) ;`	134xyz	v = "134xyz"
`scanf("%2s", v) ;`	134xyz	v = "13"

Forma completa degli specificatori



Esempi

Istruzione	Input	Risultato
`scanf("%d %s", &x, v) ;`	10 Pippo	x = 10 v = "Pippo"
`scanf("%s", v) ;`	10 Pippo	x immutato v = "10"
`scanf("%*d %s", v) ;`	10 Pippo	x immutato v = "Pippo"

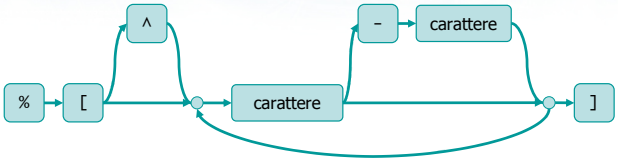
Valore di ritorno

- La funzione `scanf` ritorna un valore intero:
  - Numero di elementi (%) effettivamente letti
    - Non conta quelli "saltati" con %*
    - Non conta quelli non letti perché l'input non conteneva i caratteri desiderati
    - Non conta quelli non letti perché l'input è finito troppo presto
      - End-of-File per `fscanf`
      - Fine stringa per `sscanf`
  - `EOF` se l'input era già in condizione End-of-File all'inizio della lettura

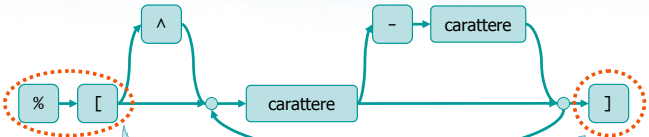
Lettura di stringhe

- La lettura di stringhe avviene solitamente con lo specificatore di formato %s
  - Salta tutti i caratteri di spaziatura
  - Acquisisci tutti i caratteri seguenti, fermandosi al primo carattere di spaziatura (senza leggerlo)
- Qualora l'input dei separatori diversi da spazio, è possibile istruire `scanf` su quali siano i caratteri leciti, mediante lo specificatore %[*pattern*]

Struttura di un pattern

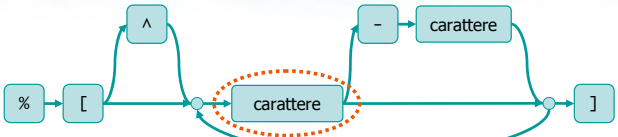


Struttura di un pattern



Definizione di pattern nella stringa di formato

Struttura di un pattern

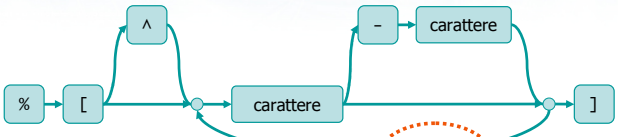


Carattere di cui può essere composta la stringa da leggere

Esempi

Pattern	Effetto
`%[r]`	Legge solo sequenze di `'r'`

Struttura di un pattern

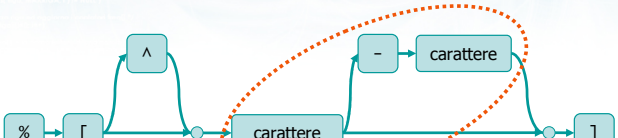


Caratteri o intervalli possono essere ripetuti più volte

Esempi

Pattern	Effetto
`%[r]`	Legge solo sequenze di `'r'`
`%[abcABC]`	Legge sequenze composte da a, b, c, A, B, C, in qualsiasi ordine e di qualsiasi lunghezza

Struttura di un pattern

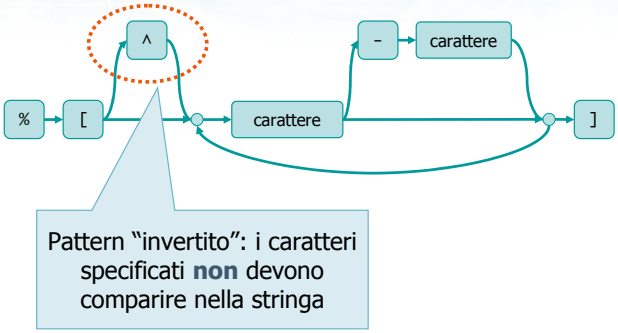


Intervallo di caratteri di cui può essere composta la stringa da leggere

Esempi

Pattern	Effetto
%[r]	Legge solo sequenze di 'r'
%[abcABC]	Legge sequenze composte da a, b, c, A, B, C, in qualsiasi ordine e di qualsiasi lunghezza
%[a-cA-C]	Idem come sopra
%[a-zA-Z]	Sequenze di lettere alfabetiche
%[0-9]	Sequenze di cifre numeriche
%[a-zA-Z0-9]	Sequenze alfanumeriche

Struttura di un pattern



Pattern "invertito": i caratteri specificati **non** devono comparire nella stringa

Esempi

Pattern	Effetto
%[r]	Legge solo sequenze di 'r'
%[abcABC]	Legge sequenze composte da a, b, c, A, B, C, in qualsiasi ordine e di qualsiasi lunghezza
%[a-cA-C]	Idem come sopra
%[a-zA-Z]	Sequenze di lettere alfabetiche
%[0-9]	Sequenze di cifre numeriche
%[a-zA-Z0-9]	Sequenze alfanumeriche
%[^x]	Qualunque sequenza che non contiene 'x'
%[^\n]	Legge fino a file riga
%[^,;.!?]	Si ferma alla punteggiatura o spazio

Osservazioni

- Ricordare che i pattern devono sempre essere associati a dati di tipo stringa (vettori di caratteri)
- Il comune specificatore "%s" equivale al pattern "%[^ \t\n]"

Esempio

- Il file /etc/passwd, presente in tutti i sistemi operativi derivati da Unix, contiene i dati degli utenti nel seguente formato:

```
corno:w3tce34:501:401:Fulvio Corno:/home/corno:/bin/bash
```

  - Campi separati da ':'
  - Nome utente, password: stringhe prive di spazi
  - User ID, Group ID: interi
  - Nome reale: stringa generica (con spazi)
  - Home directory e shell: stringhe generiche

Soluzione

```
f = myfopen("/etc/passwd", "r") ;

while(fgets(riga, MAX, f)!=NULL)
{
  sscanf(riga,
    "%[^:]:[%^:]:%d:%d:"
    "%[^:]:%[^:]:%[^:]",
    login, pass, &uid, &gid,
    realname, home, shell ) ;
  /* elabora i dati ... */
}
myfclose(f) ;
```

Formattazione avanzata

Stream predefiniti

Stream predefiniti

- L'istruzione `fopen` permette di aprire nuovi stream, associati a file esistenti sui dischi dell'elaboratore
- All'avvio di un programma in C, sono già stati aperti in modo automatico 3 stream predefiniti
  - `stdin`
  - `stdout`
  - `stderr`

`stdin`

- `stdin` è detto lo "standard input" di un programma
- Normalmente è associato al canale di input del terminale (o della console) nel quale il programma è avviato
  - In pratica, la tastiera del P.C.
- L'input può essere rediretto, prendendolo da un file anziché dalla tastiera, avviando il programma da linea di comando con l'operatore <
  - `prog < file.txt`

`stdout`

- `stdout` è detto lo "standard output" di un programma
- Normalmente è associato al canale di output del terminale (o della console) nel quale il programma è avviato
  - In pratica, il video del P.C.
- L'output può essere rediretto, salvandolo su un file anziché su video, avviando il programma da linea di comando con l'operatore >
  - `prog > file.txt`

`stderr`

- `stderr` è detto lo "standard error" di un programma
- Normalmente è associato al canale di output del terminale (o della console) nel quale il programma è avviato
  - In pratica, il video del P.C.
- È uno stream distinto ed indipendente da `stdout`
- Solitamente l'output di `stderr` non viene rediretto, per permettere ai messaggi di errore di essere sempre visti dall'utilizzatore

Uso comune

- `stdin` viene usato per acquisire i dati
  - Può essere rediretto da un file
- `stdout` viene usato per presentare i risultati
  - Può essere rediretto su un file
- `stderr` viene usato esclusivamente per i messaggi di errore
  - Rimane visibile sulla console

Equivalenze

La funzione…	è equivalente a…
`scanf("formato", …);`	`fscanf(stdin, "formato", …);`
`printf("formato", …);`	`fprintf(stdout, "formato", …);`
`ch=getchar();`	`ch=fgetc(stdin);`
`putchar(ch);`	`fputc(ch, stdout);`

Stampa dei messaggi di errore

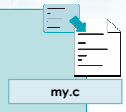
```
if(...condizione errore fatale...)
{
  fprintf(stderr,
    "Messaggio di errore\n") ;
  exit(1) ;
}
```


Suggerimento

- Tutti i messaggi di errore devono essere stampati sullo stream `stderr`
- Conviene definire una funzione `myerror`
  - Stampa un messaggio di errore
  - Interrompe il programma

```
void myerror(char *message) ;
```


Funzione `myerror`




```
int myerror(char *message)
{
    fputs( message, stderr ) ;
    exit(1) ;
}
```

I/O Avanzato e File

Esercizi proposti

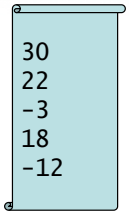
- Esercizio "Somma numeri"
- Esercizio "Bersagli"
- Esercizio "Consumi toner"

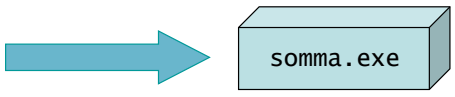
Esercizi proposti

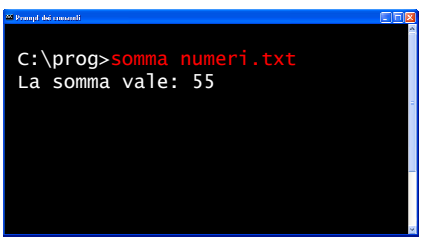
Esercizio "Somma numeri"

- Un file di testo contiene una serie di numeri interi (positivi o negativi), uno per riga
- Si scriva un programma C che:
  - Acquisisca da linea di comando il nome del file
  - Calcoli la somma di tutti i numeri presenti nel file
  - Stampi in output il valore di tale somma

numeri.txt







sommafile.c

```
int main(int argc, char *argv[])
{
  const int MAX = 80 ;

  FILE * f ;
  char nomefile[MAX] ;
  char riga[MAX] ;
  int r, num ;

  int somma ;
```

Soluzione (2/4)

sommafile.c

```
if(argc != 2)
  myerror("Num. argomenti errato\n") ;

strcpy(nomefile, argv[1]) ;

f = myfopen( nomefile, "rt" ) ;

somma = 0 ;
```

Soluzione (3/4)

sommafile.c

```
while( fgets( riga, MAX, f ) != NULL)
{
  r = sscanf( riga, "%d", &num ) ;

  if(r==1)
    somma = somma + num ;
  else
    printf("Riga ignorata\n") ;
}
```

Soluzione (4/4)

sommafile.c

```
  myfclose(f) ;

  printf( "La somma vale: %d\n", somma ) ;

  exit(0) ;
}
```

Esercizi proposti

Esercizio "Bersagli"

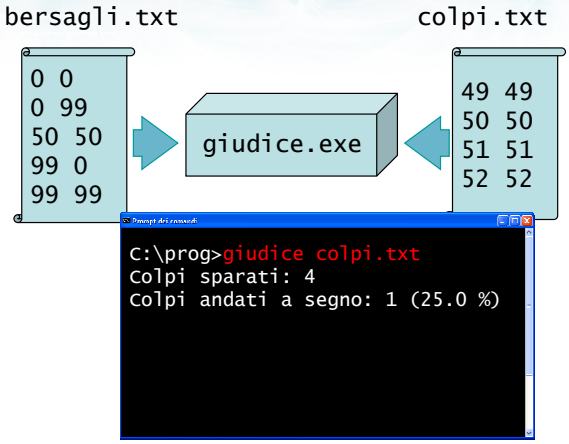
Esercizio "Bersagli" (1/2)

- Si desidera creare un programma in grado di calcolare il numero di colpi andati a segno in un'esercitazione di tiro
- I bersagli sono descritti tramite le coordinate cartesiane del punto in cui sono posizionati all'interno di una griglia 100 × 100. Le coordinate sono rappresentate solo da numeri interi, compresi tra 0 e 99. La posizione dei bersagli è contenuta nel file di testo `bersagli.txt`: ogni riga di tale file contiene le coordinate X e Y di un singolo bersaglio

Esercizio "Bersagli" (2/2)

- I colpi sparati sono descritti anch'essi tramite le loro coordinate X e Y e sono memorizzati in un file di caratteri il cui nome è passato come primo parametro sulla linea di comando. Ogni riga di tale file contiene le coordinate X e Y del punto in cui è stato sparato un colpo
- Si scriva un programma che legga dai file succitati la posizione dei bersagli ed i colpi sparati e quindi calcoli il numero di colpi andati a segno, sia come valore assoluto sia come percentuale dei colpi sparati

Analisi



Algoritmo

- Acquisire dal file `bersagli.txt` tutte le coordinate, memorizzandole in due vettori paralleli `Bx[]` e `By[]`. Lunghezza dei vettori: `Nb`
- Acquisire dal file `argv[1]` le coordinate dei vari colpi `Cx`, `Cy`. Numero colpi: `Nc`
  - Per ciascun colpo, verificare se le coordinate coincidono con quelle di almeno un bersaglio
  - Se sì, incrementare `Ncc`
- Stampare `Ncc` e `Ncc/Nc*100`

Soluzione (1/4)

bersagli.c

```
int main(int argc, char *argv[])
{
  const int MAXB = 100 ;
      /* massimo numero di bersagli */
  const int MAX = 80 ;
      /* lunghezza riga del file */
  const char FILEB[] = "bersagli.txt";

  int Nb ; /* numero di bersagli */
  int Bx[MAXB], By[MAXB] ;
          /* coordinate dei bersagli */

  int Nc ; /* numero colpi sparati */
  int Ncc ; /* numero di colpi centrati */
```

Soluzione (2/4)

bersagli.c

```
FILE *f ;
char riga[MAX] ;
int Cx, Cy ;
int i, r, trovato ;

/* 1: acquisizione coordinate bersagli */
f = myfopen( FILEB, "rt") ;
Nb = 0 ;
while( fgets(riga, MAX, f) != NULL )
{
  r=sscanf(riga,"%d %d",&Bx[Nb],&By[Nb]);
  if( r!=2 )
    myerror("Formato errato\n") ;
  Nb ++ ;
}
myfclose(f);
```

Soluzione (3/4)

bersagli.c

```
/* 2: analisi coordinate dei colpi */
if( argc != 2 )
  myerror("ERR: manca nome file\n");
f = myfopen( argv[1], "rt") ;

Nc = 0 ;
Ncc = 0 ;
while( fgets(riga, MAX, f) != NULL )
{
  r = sscanf( riga, "%d %d", &Cx, &Cy );
  if(r!=2) myerror("Formato errato\n") ;
  Nc ++ ;
  /* Ricerca del bersaglio */
}
myfclose(f);
```

Soluzione (3/4)

bersagli.c

```
/* 2: analisi coordinate dei colpi */
    trovato = 0 ;
    for(i=0; i<Nb && trovato==0; i++)
      if( Cx==Bx[i] && Cy==By[i] )
        trovato = 1 ;

    if(trovato==1)
      Ncc ++ ;

  Nc ++ ;
  /* Ricerca del bersaglio */
}
myfclose(f);
```

Soluzione (4/4)

bersagli.c

```
  /* 3: stampa risultati */
  printf("Colpi sparati: %d\n", Nc) ;
  printf("Colpi andati a segno: %d ", Ncc);
  if(Nc!=0)
    printf("(%.2f%%)", Ncc*100.0/Nc) ;
  printf("\n");

  exit(0) ;
}
```

Esercizi proposti

Esercizio "Consumi toner"

Esercizio "Consumi toner" (1/3)

- Si desidera analizzare la statistica dei consumi di toner di un'azienda per ottimizzare gli acquisti futuri
- La quantità di cartucce di toner prelevate dal magazzino ogni giorno è riportata all'interno di un file di testo il cui nome è passato come primo parametro sulla riga di comando

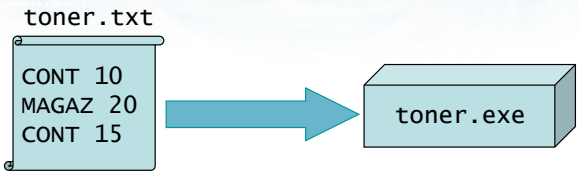
Esercizio "Consumi toner" (2/3)

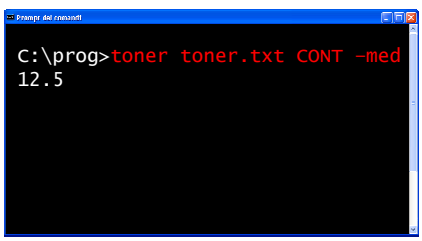
- Il file contiene una riga per ogni giorno. Ogni riga contiene in sequenza:
  - Il nome del dipartimento che ha prelevato il toner (una stringa lunga al massimo 5 caratteri)
  - Un numero intero (valore minimo 1 e massimo 99) che indica la quantità di cartucce di toner prelevate in quel giorno da quel dipartimento
- Non è noto il numero di righe presenti nel file

Esercizio "Consumi toner" (3/3)

- Il programma riceve inoltre come secondo argomento sulla linea di comando il nome di un dipartimento per il quale calcolare l'indicatore statistico dato come terzo argomento sulla linea di comando secondo la seguente codifica:
  - `-min` indica che si desidera il valore minimo
  - `-max` indica che si desidera il valore massimo
  - `-med` indica che si desidera il valore medio (da stamparsi in output con un cifra dopo la virgola)

Analisi





Argomenti del programma

```
C:\prog>toner toner.txt CONT -med
```

argv[1]
Nome del file contenente i consumi

argv[2]
Dipartimento da analizzare

argv[3]
Operazione statistica:
-min
-med
-max

Soluzione (1/4)

toner.c

```
int main(int argc, char *argv[])
{
  const int LUNDIP = 5 ;
  const int MAX = 80 ;

  char dip[LUNDIP+1], dipf[LUNDIP+1] ;
  int stat ;
    /* tipo di statistica:
         1=min, 2=max, 3=med */
  FILE * f ;
  int qtaf, r ;
  int min, max, tot, cont ;
  char riga[MAX+1] ;
```

Soluzione (2/4)

toner.c

```
if(argc!=4)
  myerror("Numero parametri errato\n");
/* Acquisisci il nome del dipartimento */
strcpy(dip, argv[2]) ;
/* Acquisisci tipo statistica */
if( strcmp( argv[3], "-min") == 0 )
  stat = 1 ;
else if ( strcmp( argv[3], "-max") == 0 )
  stat = 2 ;
else if ( strcmp( argv[3], "-med") == 0 )
  stat = 3 ;
else
  myerror("Statistica sconosciuta\n");
```

Soluzione (3/4)

toner.c

```
f = myfopen(argv[1], "rt") ;
tot = 0 ;
cont = 0 ;
min = 100 ;
max = 0 ;
while( fgets(riga, MAX, f) != NULL )
{
  r = sscanf(riga, "%s %d", dipf, &qtaf);
  if(r!=2)
    printf("Riga ignorata\n");
  else
  {
    /* Aggiorna statistiche */
  }
}
myfclose(f) ;
```

Soluzione (3/4)

toner.c

```
    if(strcmp(dip, dipf)==0)
    {
      if( qtaf < min )
        min = qtaf ;
      if( qtaf > max )
        max = qtaf ;
      tot = tot + qtaf ;
      cont++ ;
    }
  else
  {
    /* Aggiorna statistiche */
  }
}
myfclose(f) ;
```

Soluzione (4/4)

toner.c

```
/* Stampa il valore della statistica */
if(cont==0)
  printf("Nessun elemento\n");
else if( stat==1 )
  printf("%d\n", min) ;
else if( stat ==2 )
  printf("%d\n", max) ;
else if( stat==3 )
  printf("%.1f\n", (float)tot/cont) ;

exit(0) ;
}
```

I/O Avanzato e File

Sommario

Argomenti trattati (1/2)

- File
  - File binari
  - File di testo
- Gestione dei file in C
  - Apertura/chiusura
  - Lettura/scrittura
  - Gestione degli errori
  - Il problema degli errori di formattazione

Argomenti trattati (2/2)

- Formattazione avanzata
  - Funzione `sscanf`
  - Opzioni degli specificatori di formato
    - In output
    - In input
    - Pattern di input
  - Stream predefiniti
- Input robusto
  - Utilizzo combinato di `fgets` e `sscanf`

Tecniche di programmazione

- Gestire i file, in lettura e scrittura
- Verificare gli errori che possono incorrere nelle operazioni di I/O
- Utilizzare le funzioni `myfopen`, `myfclose`, `myerror`
- Utilizzare `sscanf` per analizzare righe anche dal formato complesso
- Utilizzare `printf`/`fprintf` per controllare l'ampiezza dei campi di output

Materiale aggiuntivo

- Sul CD-ROM
  - Testi e soluzioni degli esercizi trattati nei lucidi
  - Scheda sintetica
  - Esercizi risolti
  - Esercizi proposti
- Esercizi proposti da altri libri di testo