Programmazione 1

Lezione 8

Vincenzo Marra

vincenzo.marra@unimi.it

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

3 maggio 2017

La ricorsione in programmazione

- La ricorsione è l'analogo informatico (ossia computazionale) della induzione matematica.

La ricorsione in programmazione

- La ricorsione è l'analogo informatico (ossia computazionale) della induzione matematica.
- La definizione induttiva del fattoriale di un numero intero $n \geqslant 0$ è:

$$n! := \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n \geqslant 1 \end{cases}$$

La ricorsione in programmazione

- La ricorsione è l'analogo informatico (ossia computazionale) della induzione matematica.
- La definizione induttiva del fattoriale di un numero intero $n \geqslant 0$ è:

$$n! := \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n \geqslant 1 \end{cases}$$

- Il caso $n = 0$ si chiama caso base. La definizione della funzione fattoriale, nel caso base, non contiene rimandi alla funzione fattoriale stessa. In generale vi possono essere più casi base, in numero finito.

La ricorsione in programmazione

- La ricorsione è l'analogo informatico (ossia computazionale) della induzione matematica.
- La definizione induttiva del fattoriale di un numero intero $n \geqslant 0$ è:

$$n! := \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n \geqslant 1 \end{cases}$$

- Il caso $n = 0$ si chiama caso base. La definizione della funzione fattoriale, nel caso base, non contiene rimandi alla funzione fattoriale stessa. In generale vi possono essere più casi base, in numero finito.
- Il caso $n > 0$ si chiama caso induttivo, o caso passo. Qui la definizione della funzione fattoriale rimanda alla funzione fattoriale stessa, applicata però a un valore strettamente minore del parametro n — in questo esempio, ad $n - 1$.

- Se scriviamo $\text{fatt}(n)$ per il fattoriale di n, usando l'ordinaria notazione funzionale, la definizione diventa:

$$\text{fatt}(n) := \begin{cases} 1 & \text{se } n = 0 \\ n\text{fatt}(n-1) & \text{se } n \geqslant 1 \end{cases} \qquad (*)$$

- Se scriviamo $\text{fatt}(n)$ per il fattoriale di n, usando l'ordinaria notazione funzionale, la definizione diventa:

$$\text{fatt}(n) := \begin{cases} 1 & \text{se } n = 0 \\ n\text{fatt}(n-1) & \text{se } n \geqslant 1 \end{cases} \qquad (*)$$

- Se poi interpretiamo questa definizione come la specifica di un algoritmo per calcolare il fattoriale, vediamo che il prototipo della funzione che implementa l'algoritmo è

```
int fatt(int n)
```

- Se scriviamo $\text{fatt}(n)$ per il fattoriale di n, usando l'ordinaria notazione funzionale, la definizione diventa:

$$\text{fatt}(n) := \begin{cases} 1 & \text{se } n = 0 \\ n\text{fatt}(n-1) & \text{se } n \geqslant 1 \end{cases} \qquad (*)$$

- Se poi interpretiamo questa definizione come la specifica di un algoritmo per calcolare il fattoriale, vediamo che il prototipo della funzione che implementa l'algoritmo è

```
int fatt(int n)
```

- Osserviamo inoltre che se il codice di `int fatt(int n)` fosse scritto seguendo la struttura logica della definizione induttiva (*), la procedura risultante avrebbe la peculiarità di invocare se stessa.

- Se scriviamo $\text{fatt}(n)$ per il fattoriale di n, usando l'ordinaria notazione funzionale, la definizione diventa:

$$\text{fatt}(n) := \begin{cases} 1 & \text{se } n = 0 \\ n\text{fatt}(n-1) & \text{se } n \geqslant 1 \end{cases} \qquad (*)$$

- Se poi interpretiamo questa definizione come la specifica di un algoritmo per calcolare il fattoriale, vediamo che il prototipo della funzione che implementa l'algoritmo è

```
int fatt(int n)
```

- Osserviamo inoltre che se il codice di `int fatt(int n)` fosse scritto seguendo la struttura logica della definizione induttiva (*), la procedura risultante avrebbe la peculiarità di invocare se stessa.
- La programmazione ricorsiva è appunto caratterizzata dalla circostanza che una data funzione invochi se stessa: e le funzioni codificate in questo modo si dicono ricorsive. Studiamo adesso dei primi esempi.

fatt.c

```c
#include <stdio.h>

int fatt(int);

int main(void) {
        int n;
        scanf("%d",&n);
        printf("%d\n", fatt(n));
        return 0;
}

int fatt(int n) {
        if (n==0)
                return 1;
        return (n*fatt(n-1));
}
```

Calcolo ricorsivo del fattoriale.

fib.c

```c
#include <stdio.h>

int fib(int);

int main(void) {
        int n;
        scanf("%d",&n);
        printf("%d\n", fib(n));
        return 0;
}

int fib(int n) {
        if ( (n==1) || (n==2) )
                return 1;
        return ( fib(n-1)+fib(n-2) );
}
```

Calcolo ricorsivo della successione di Fibonacci.

esp.c

```c
#include <stdio.h>
int esp(int,int);
int main(void) {
        int b, e;
        printf("Base (>1): ");
        scanf("%d",&b);
        getchar(); //rimuove \n
        printf("Esponente (>=0): ");
        scanf("%d",&e);
        printf("%d^%d=%d\n",b,e,esp(b,e));
        return 0;
}
/* b>1,e>=0 */
int esp(int b, int e) {
        if (e==0)
                return 1;
        return ( b*esp(b,e-1) );
}
```

Calcolo ricorsivo della funzione esponenziale.

Ricorsione e record di attivazione

Per comprendere l'uso delle funzioni fino in fondo, ed in particolare la ricorsione, è necessario avere chiaro cosa succede quando una funzione invoca un'altra funzione. Abbiamo già visto come avviene il passaggio dei parametri.

Ricorsione e record di attivazione

Per comprendere l'uso delle funzioni fino in fondo, ed in particolare la ricorsione, è necessario avere chiaro cosa succede quando una funzione invoca un'altra funzione. Abbiamo già visto come avviene il passaggio dei parametri.

Ma sono richieste molte altre operazioni da parte del sistema operativo, oltre all'implementazione del passaggio dei parametri per copia, per portare a termine la chiamata e l'esecuzione di una funzione.

Ricorsione e record di attivazione

Per comprendere l'uso delle funzioni fino in fondo, ed in particolare la ricorsione, è necessario avere chiaro cosa succede quando una funzione invoca un'altra funzione. Abbiamo già visto come avviene il passaggio dei parametri.

Ma sono richieste molte altre operazioni da parte del sistema operativo, oltre all'implementazione del passaggio dei parametri per copia, per portare a termine la chiamata e l'esecuzione di una funzione.

Una trattazione sistematica di questo argomento va al di là degli scopi di un primo corso di programmazione. Ci limiteremo a qualche cenno, che ci permetterà comunque di apprezzare meglio il costo computazionale implicato dalle chiamate alle funzioni.

I concetti fondamentali sono il record d'attivazione di una funzione, il suo spazio di memoria locale, e il meccanismo col quale avviene il passaggio di parametri. Abbiamo già parlato degli ultimi due. Abbiamo visto che nel linguaggio C il passaggio di parametri avviene per copia (o per valore, come anche si dice), mentre altri linguaggi prevedono inoltre il passaggio dei parametri per riferimento.

I concetti fondamentali sono il record d'attivazione di una funzione, il suo spazio di memoria locale, e il meccanismo col quale avviene il passaggio di parametri. Abbiamo già parlato degli ultimi due. Abbiamo visto che nel linguaggio C il passaggio di parametri avviene per copia (o per valore, come anche si dice), mentre altri linguaggi prevedono inoltre il passaggio dei parametri per riferimento.

Il record d'attivazione della funzione chiamata fa parte della memoria locale della funzione, e contiene le informazioni essenziali richieste per avviare l'esecuzione della funzione.

I concetti fondamentali sono il record d'attivazione di una funzione, il suo spazio di memoria locale, e il meccanismo col quale avviene il passaggio di parametri. Abbiamo già parlato degli ultimi due. Abbiamo visto che nel linguaggio C il passaggio di parametri avviene per copia (o per valore, come anche si dice), mentre altri linguaggi prevedono inoltre il passaggio dei parametri per riferimento.

Il record d'attivazione della funzione chiamata fa parte della memoria locale della funzione, e contiene le informazioni essenziali richieste per avviare l'esecuzione della funzione.

Solitamente è organizzato come una pila (o in inglese, stack), una struttura di dati che descriveremo alla lavagna.

I concetti fondamentali sono il record d'attivazione di una funzione, il suo spazio di memoria locale, e il meccanismo col quale avviene il passaggio di parametri. Abbiamo già parlato degli ultimi due. Abbiamo visto che nel linguaggio C il passaggio di parametri avviene per copia (o per valore, come anche si dice), mentre altri linguaggi prevedono inoltre il passaggio dei parametri per riferimento.

Il record d'attivazione della funzione chiamata fa parte della memoria locale della funzione, e contiene le informazioni essenziali richieste per avviare l'esecuzione della funzione.

Solitamente è organizzato come una pila (o in inglese, stack), una struttura di dati che descriveremo alla lavagna.
Il record d'attivazione contiene, per esempio, i parametri passati alla funzione, e un puntatore al punto del codice al quale la funzione deve restituire il controllo: è in questo modo che si implementa, a basso livello, l'istruzione `return`.

ric1.c

```c
#include <stdio.h>
int main(void)
{
        void procA(int);
        procA(0);
        return 0;
}
void procA(int y) //procA ha una copia locale di y
{
        if (y>=5)
        {
                printf("y=%d (caso base)\n",y);
                return;
        }
        printf("y=%d\n",y);
        y++;
        procA(y);
}
```

ric2.c

```c
#include <stdio.h>
int main(void)
{
        void procA(int);
        procA(0);
        return 0;
}
void procA(int y) //procA ha una copia locale di y
{
        if (y>=5)
        {
                printf("y=%d (caso base)\n",y);
                printf("Termina chiamata y=%d\n",y);
                return;
        }
        printf("y=%d\n",y);
        y++;
        procA(y);
        printf("Termina chiamata y=%d\n",y-1);
}
```

ric3.c

```c
#include <stdio.h>
int main(void)
{
        void procA(int);
        procA(0);
        return 0;
}
void procA(int y) //procA ha una copia locale di y
{
        if (y>=5)
        {
                printf("y=%d (caso base)\n",y);
                printf("Termina chiamata y=%d\n",y);
                return;
        }
        procA(y+1);
        printf("y=%d\n",y);
        printf("Termina chiamata y=%d\n",y);
}
```

binomiale.c

```c
#include <stdio.h>
int main(void) //coeff. binomiale n su k; n,k>=0
{       int bin(int n, int k);
        int n,k; printf("Inserisci n: ");
        scanf("%d",&n); getchar();
        printf("Inserisci k: ");
        scanf("%d",&k);
        printf("C(n,k) = %d\n", bin(n,k)); return 0;
}
int bin(int n, int k) //n,k>=0
{
        if (n==0)
        {
                if (k==0)       return 1;
                else            return 0;
        }
        if (k==0)
                return 1;
        return ( bin(n-1,k)+bin(n-1,k-1) );
}
```

Array di puntatori

Abbiamo visto che i puntatori sono variabili di un tipo specifico. E' dunque possibile raccoglierli in array, come per i tipi `int`, `char`, etc.

Array di puntatori

Abbiamo visto che i puntatori sono variabili di un tipo specifico. E' dunque possibile raccoglierli in array, come per i tipi `int`, `char`, etc.

La dichiarazione:

$$\textit{tipo}\ \texttt{*vett[}\textit{dim}\texttt{];} \qquad (*)$$

stabilisce che `vett` è una variabile di tipo array di *dim* elementi, ciascuno dei quali è un puntatore a variabile di tipo *tipo*.

Array di puntatori

Abbiamo visto che i puntatori sono variabili di un tipo specifico. E' dunque possibile raccoglierli in array, come per i tipi `int`, `char`, etc.

La dichiarazione:

$$tipo\ \texttt{*vett[}dim\texttt{];} \qquad (*)$$

stabilisce che `vett` è una variabile di tipo array di *dim* elementi, ciascuno dei quali è un puntatore a variabile di tipo *tipo*.

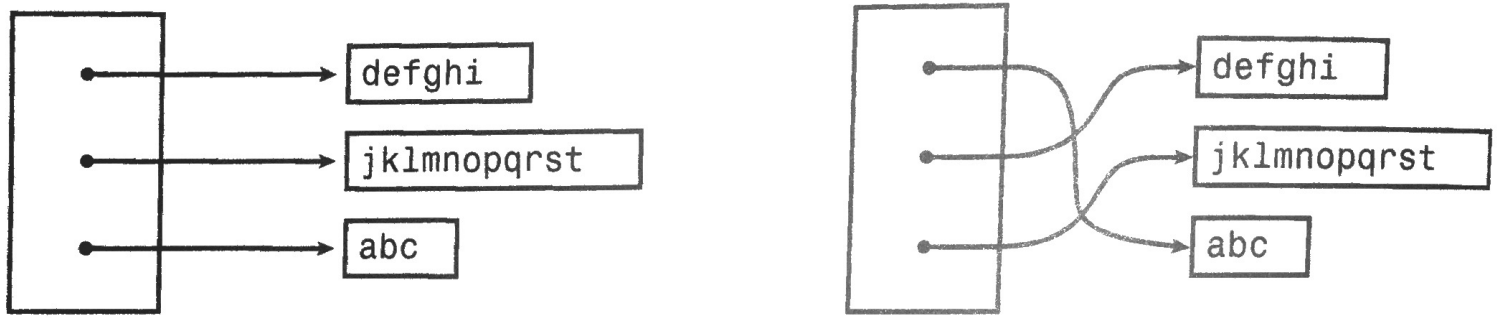
Nota. Bisogna stare attenti a non confondere (*) con un "puntatore ad array di *dim* elementi, ciascuno dei quali è di tipo *tipo*", ossia

$$tipo\ \texttt{(*vett)[}dim\texttt{];}$$

Si usano spesso gli array di puntatori a char, ossia gli array di stringhe. Ecco un'illustrazione tratta da K&R, p. 106:

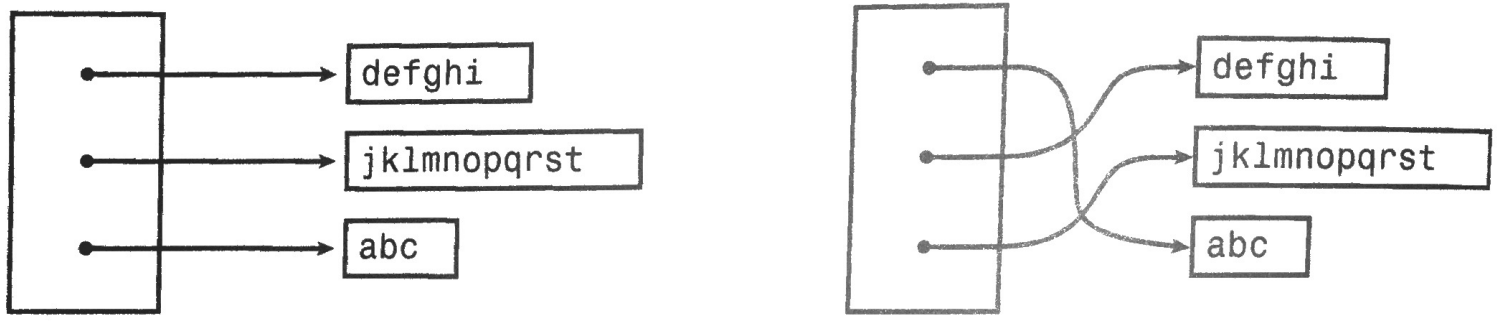
Si usano spesso gli array di puntatori a char, ossia gli array di stringhe. Ecco un'illustrazione tratta da K&R, p. 106:

rizzati in un vettore. Si ottiene il confronto tra due righe passandone i relativi puntatori a `strcmp`. Quando due righe fuori posto vanno scambiate, sono i puntatori a mutare collocazione nel vettore di puntatori e non le righe medesime.



Si usano spesso gli array di puntatori a char, ossia gli array di stringhe. Ecco un'illustrazione tratta da K&R, p. 106:

rizzati in un vettore. Si ottiene il confronto tra due righe passandone i relativi puntatori a `strcmp`. Quando due righe fuori posto vanno scambiate, sono i puntatori a mutare collocazione nel vettore di puntatori e non le righe medesime.



La dichiarazione potrebbe essere:

$$\texttt{char} * \texttt{v}[3];$$

arrpunt.c

```c
/* Array di puntatori a char */

#include <stdio.h>

int main(void)
{
        char *v[3] = {"Pinco", "Pallino", "Tizio"};

        printf("%s\n%s\n%s\n", v[0],v[1],v[2]);
        return 0;
}
```

Passaggio degli argomenti dalla riga di comando

Si possono passare argomenti al `main` al momento dell'invocazione del programma.

Passaggio degli argomenti dalla riga di comando

Si possono passare argomenti al `main` al momento dell'invocazione del programma.

Per esempio, scrivendo

```
./a.out par1 par2
```

si passano i due argomenti `par1` e `par2` alla funzione `main` di `a.out`.

Passaggio degli argomenti dalla riga di comando

Si possono passare argomenti al `main` al momento dell'invocazione del programma.

Per esempio, scrivendo

```
./a.out par1 par2
```

si passano i due argomenti `par1` e `par2` alla funzione `main` di `a.out`.

Si tenga presente che `"par1 par2"` rappresenta un singolo parametro contenente uno spazio, a causa delle virgolette, mentre `par1 par2` rappresenta due parametri distinti. (Ciò riguarda in realtà la shell, e non il linguaggio C.)

- Perché il passaggio dei parametri abbia effetto il prototipo della funzione `main` dev'essere:

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

- Il parametro

```
int argc
```

  detto *argument count*, è il numero di parametri (separati da spazi) presenti sulla riga di comando.
- Il parametro

```
char *argv[]
```

  detto *argument vector*, è un puntatore a un array di stringhe di caratteri che contengono gli argomenti presenti sulla riga di comando, uno per stringa.
- Per convenzione, `argv[0]` è il nome con cui il programma è stato invocato — nell'esempio precedente, è `./a.out`.

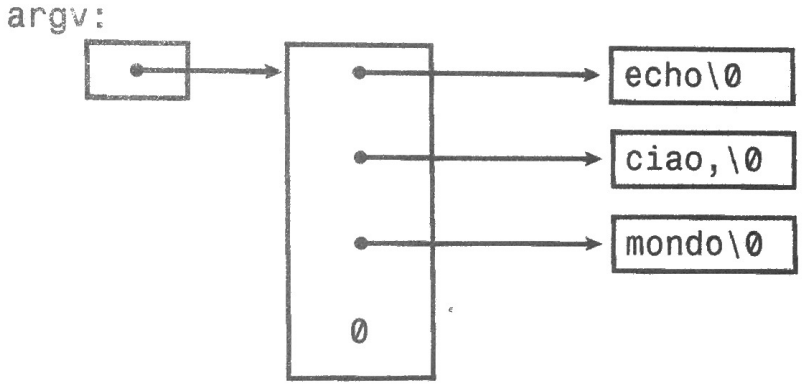
L'esemplificazione più semplice è data dal programma `echo`, che visualizza gli argomenti ricevuti dalla riga di comando, inframmezzati da spazi, su una riga singola. In pratica, il comando

```
echo ciao, mondo
```

produce

```
ciao, mondo
```

Per convenzione, `argv[0]` è il nome con cui il programma è stato invocato, quindi `argc` vale sempre almeno 1; quando è proprio uguale a 1, non si danno argomenti nella riga di comando dopo il nome del programma. Nell'esempio precedente, `argc` è 3, e `argv[0]`, `argv[1]`, `argv[2]` sono rispettivamente `"echo"`, `"ciao,"` e `"mondo"`. Il primo argomento facoltativo è `argv[1]` e l'ultimo è `argv[argc-1]`; inoltre, lo standard prescrive che `argv[argc]` sia un puntatore nullo.



La prima versione di `echo` tratta `argv` come un vettore di puntatori di caratteri:

ecoarg.c

```c
#include <stdio.h>

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

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

        return 0;
}
```

add.c

```c
/* Computa la somma dei primi due argomenti sulla riga di comando */

#include <stdio.h>
#include <stdlib.h> //Funzioni utilita' per le stringhe

int main(int argc, char *argv[])
{
  if (argc < 3)
  {
    printf("Errore: mancano gli argomenti.\n");
    return -1;
  }

  printf("%s+%s=%d\n",argv[1],argv[2],atoi(argv[1])+atoi(argv[2]));
  return 0;
}
```

Input/Output standard: i file

Il compilatore C assume l'esistenza nel sistema di tre flussi di dati standard:

1. `stdin` – associato d'ufficio ai dati in ingresso provenienti dalla tastiera.
2. `stdout` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore.
3. `stderr` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore. (Non ne parleremo in dettaglio.)

Input/Output standard: i file

Il compilatore C assume l'esistenza nel sistema di tre flussi di dati standard:

1. `stdin` – associato d'ufficio ai dati in ingresso provenienti dalla tastiera.
2. `stdout` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore.
3. `stderr` – associato d'ufficio ai dati in uscita visualizzati sul monitor del calcolatore. (Non ne parleremo in dettaglio.)

Un file che risieda nella memoria di massa del calcolatore può essere usato dai programmi in C come flusso di dati, sia in lettura che in scrittura. Le funzioni della libreria standard necessarie allo scopo sono definite nel file di intestazione `stdio.h`.

Apertura dei file

Prima di poter leggere da o scrivere su un file, è necessario aprirlo tramite la funzione:

```
FILE *fopen(char *nome, char *modo)
```

Apertura dei file

Prima di poter leggere da o scrivere su un file, è necessario aprirlo tramite la funzione:

```
FILE *fopen(char *nome, char *modo)
```

1. La stringa nome è il nome del file, comprensivo del percorso (assoluto o relativo al punto del file system in cui risiede il programma).

Apertura dei file

Prima di poter leggere da o scrivere su un file, è necessario aprirlo tramite la funzione:

```
FILE *fopen(char *nome, char *modo)
```

1. La stringa `nome` è il nome del file, comprensivo del percorso (assoluto o relativo al punto del file system in cui risiede il programma).
2. La stringa `modo` stabilisce la modalità di apertura del file — per esempio, in lettura — e sarà discussa fra poco.

Apertura dei file

Prima di poter leggere da o scrivere su un file, è necessario aprirlo tramite la funzione:

```
FILE *fopen(char *nome, char *modo)
```

1. La stringa `nome` è il nome del file, comprensivo del percorso (assoluto o relativo al punto del file system in cui risiede il programma).
2. La stringa `modo` stabilisce la modalità di apertura del file — per esempio, in lettura — e sarà discussa fra poco.
3. `FILE` è un tipo (definito da `stdio.h`, non primitivo come `int` o `char`) i cui valori descrivono file del file system.

Apertura dei file

Prima di poter leggere da o scrivere su un file, è necessario aprirlo tramite la funzione:

```
FILE *fopen(char *nome, char *modo)
```

1. La stringa `nome` è il nome del file, comprensivo del percorso (assoluto o relativo al punto del file system in cui risiede il programma).
2. La stringa `modo` stabilisce la modalità di apertura del file — per esempio, in lettura — e sarà discussa fra poco.
3. `FILE` è un tipo (definito da `stdio.h`, non primitivo come `int` o `char`) i cui valori descrivono file del file system.
4. La funzione restituisce un puntatore a `FILE`. Esso è `NULL` se, e solo se, si è riscontrato un errore.

Modalità di apertura dei file

Una chiamata alla funzione:

```
FILE *fopen(char *nome, char *modo)
```

stabilisce la modalità di apertura del file tramite la stringa `modo`.

Modalità di apertura dei file

Una chiamata alla funzione:

```
FILE *fopen(char *nome, char *modo)
```

stabilisce la modalità di apertura del file tramite la stringa `modo`.

1. Se `modo` è `"r"` (*read*), il file è aperto in lettura. Se esso è inesistente la funzione restituisce `NULL` e si ha un errore.

Modalità di apertura dei file

Una chiamata alla funzione:

```
FILE *fopen(char *nome, char *modo)
```

stabilisce la modalità di apertura del file tramite la stringa `modo`.

1. Se `modo` è `"r"` (*read*), il file è aperto in lettura. Se esso è inesistente la funzione restituisce `NULL` e si ha un errore.
2. Se `modo` è `"w"` (*write*), il file è aperto in scrittura. Se esso già esiste viene interamente cancellato. Se invece è insistente, viene creato.

Modalità di apertura dei file

Una chiamata alla funzione:

```
FILE *fopen(char *nome, char *modo)
```

stabilisce la modalità di apertura del file tramite la stringa `modo`.

1. Se `modo` è `"r"` (*read*), il file è aperto in lettura. Se esso è inesistente la funzione restituisce `NULL` e si ha un errore.
2. Se `modo` è `"w"` (*write*), il file è aperto in scrittura. Se esso già esiste viene interamente cancellato. Se invece è insistente, viene creato.
3. Se `modo` è `"a"` (*append*), il file è aperto in modalità di scrittura *accodamento*: le operazioni di scrittura aggiungono i dati da scrivere alla fine del file. Se esso è insistente, viene creato.

Modalità di apertura dei file

Una chiamata alla funzione:

```
FILE *fopen(char *nome, char *modo)
```

stabilisce la modalità di apertura del file tramite la stringa `modo`.

1. Se `modo` è `"r"` (*read*), il file è aperto in lettura. Se esso è inesistente la funzione restituisce `NULL` e si ha un errore.
2. Se `modo` è `"w"` (*write*), il file è aperto in scrittura. Se esso già esiste viene interamente cancellato. Se invece è insistente, viene creato.
3. Se `modo` è `"a"` (*append*), il file è aperto in modalità di scrittura *accodamento*: le operazioni di scrittura aggiungono i dati da scrivere alla fine del file. Se esso è insistente, viene creato.
4. Se si tenta di eseguire una di queste operazioni, ma non se ne hanno i permessi, la funzione restituisce `NULL`.

1. Nota. Le modalità appena viste non permettono di eseguire lettura da e scrittura su il medesimo file aperto tramite un singolo puntatore a FILE. Letture e scritture concorrenti sono possibili in C tramite altre modalità di apertura, cui accenneremo solo brevemente.

1. Nota. Le modalità appena viste non permettono di eseguire lettura da e scrittura su il medesimo file aperto tramite un singolo puntatore a `FILE`. Letture e scritture concorrenti sono possibili in C tramite altre modalità di apertura, cui accenneremo solo brevemente.
2. Se `modo` è `"r+"` (*read update*), il file è aperto in lettura e scrittura.
3. Se `modo` è `"w+"` (*write update*), il file è aperto in scrittura e lettura.
4. Se `modo` è `"a+"` (*append update*), il file è aperto in modalità di lettura e accodamento.

1. Nota. Le modalità appena viste non permettono di eseguire lettura da e scrittura su il medesimo file aperto tramite un singolo puntatore a `FILE`. Letture e scritture concorrenti sono possibili in C tramite altre modalità di apertura, cui accenneremo solo brevemente.
2. Se `modo` è `"r+"` (*read update*), il file è aperto in lettura e scrittura.
3. Se `modo` è `"w+"` (*write update*), il file è aperto in scrittura e lettura.
4. Se `modo` è `"a+"` (*append update*), il file è aperto in modalità di lettura e accodamento.
5. Per poter usare queste modalità di apertura occorre gestire la posizione corrente all'interno del file fra le letture e le scritture, tramite funzioni quali `fseek` o `fflush`.

1. Nota. Le modalità appena viste non permettono di eseguire lettura da e scrittura su il medesimo file aperto tramite un singolo puntatore a `FILE`. Letture e scritture concorrenti sono possibili in C tramite altre modalità di apertura, cui accenneremo solo brevemente.
2. Se `modo` è `"r+"` (*read update*), il file è aperto in lettura e scrittura.
3. Se `modo` è `"w+"` (*write update*), il file è aperto in scrittura e lettura.
4. Se `modo` è `"a+"` (*append update*), il file è aperto in modalità di lettura e accodamento.
5. Per poter usare queste modalità di apertura occorre gestire la posizione corrente all'interno del file fra le letture e le scritture, tramite funzioni quali `fseek` o `fflush`.
6. I dettagli sono in K&R, Cap. 7 e §1 dell'Appendice B.

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

1. Il parametro `pf` è il puntatore al file da chiudere.

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

1. Il parametro `pf` è il puntatore al file da chiudere.
2. La funzione restituisce 0 se l'operazione di chiusura è stata eseguita con successo, e `EOF` altrimenti.

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

1. Il parametro `pf` è il puntatore al file da chiudere.
2. La funzione restituisce 0 se l'operazione di chiusura è stata eseguita con successo, e `EOF` altrimenti.
3. Un errore durante la chiusura si può riscontrare, per esempio, perché `pf` è `NULL` o perché esso non è correttamente associato a un file aperto.

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

1. Il parametro `pf` è il puntatore al file da chiudere.
2. La funzione restituisce 0 se l'operazione di chiusura è stata eseguita con successo, e `EOF` altrimenti.
3. Un errore durante la chiusura si può riscontrare, per esempio, perché `pf` è `NULL` o perché esso non è correttamente associato a un file aperto.
4. I file che risultino ancora aperti al momento della terminazione (non anomala) del programma sono automaticamente chiusi.

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

1. È però sempre una buona idea chiudere esplicitamente i file che non servono più, perché le risorse del sistema operativo sono limitate ed occorre usarle efficientemente.

Chiusura dei file

Quando un file aperto non è più usato dal programma, lo si può chiudere tramite la funzione:

```
int *fclose(FILE *pf)
```

1. È però sempre una buona idea chiudere esplicitamente i file che non servono più, perché le risorse del sistema operativo sono limitate ed occorre usarle efficientemente.
2. Oltre a chiudere il file, la chiamata alla funzione forza la scrittura fisica sul file dei dati in uscita che ancora risiedano nella memoria tampone (buffer) in attesa di essere trasferiti. Tutti i sistemi operativi moderni usano tecniche più o meno sofisticate di *bufferizzazione dell'I/O*, che noi però non potremo spiegare in dettaglio. Si tratta di argomenti che si studiano nei corsi di Sistemi Operativi.

Lettura e scrittura di caratteri singoli

- `int getc(FILE *pf)`
- Restituisce il successivo carattere del file `pf`, che deve essere aperto in lettura, come intero (ossia il suo codice ASCII del carattere), oppure l'intero `EOF` se incontra la fine del file o riscontra un errore.
- Nota. A differenza della lettura da `stdin`, la lettura da file non è mai bloccante. Ciò vale anche per tutte le altre funzioni di lettura da file.

Lettura e scrittura di caratteri singoli

- `int getc(FILE *pf)`
- Restituisce il successivo carattere del file `pf`, che deve essere aperto in lettura, come intero (ossia il suo codice ASCII del carattere), oppure l'intero `EOF` se incontra la fine del file o riscontra un errore.
- Nota. A differenza della lettura da `stdin`, la lettura da file non è mai bloccante. Ciò vale anche per tutte le altre funzioni di lettura da file.

- `int putc(int c, FILE *pf)`
- Scrive il carattere `c`, codificato come `int`, sul file `pf`, che deve essere aperto in scrittura. Restituisce il carattere scritto oppure `EOF` se incorre in un errore.

Esercizio: Eco da file

Si scriva un programma che accetti da riga di comando il nome di un file, e lo visualizzi sul terminale carattere per carattere. Se il nome del file non è specificato o è inesistente o non può essere aperto, il programma termina con un messaggio d'errore appropriato

feco.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc < 2)
        {
                printf("Errore nel numero degli argomenti.\n");
                return -1;
        }

        FILE *pf;
        if ( (pf=fopen(argv[1], "r")) == NULL )
        {
                printf("Errore nell'apertura del file: \"%s\".\n", argv[1]);
                return -1;
        }

        /* Esegue l'eco a video, carattere per carattere */
        char c;
        while ( (c=getc(pf)) != EOF )
                putchar(c);

        return 0;
}
```

Lettura e scrittura di dati formattati

- `int fscanf (FILE *pf, const char *formato, ...)`
- `int fprintf (FILE *pf, const char *formato, ...)`
- Queste due funzioni sono identiche alle loro controparti `scanf` e `printf`, eccetto che il primo argomento indica il file dal o sul quale leggere o scrivere, rispettivamente.
- In altre parole,
  - `int fscanf (stdin, const char *formato, ...)`
  - `int fprintf (stdout, const char *formato, ...)`

  sono identiche a `scanf` e `printf`, che già conosciamo.

Lettura e scrittura di stringhe

- `char *fgets (char *s, int max, FILE *pf)`
- L'abbiamo già citata. Legge la prossima riga da `pf`, fino al più a `max-1` caratteri, pone quanto letto in `s`, aggiungendo in coda il terminatore `'\0'`.
- Nota. Include anche l'eventuale `'\n'` fra i caratteri letti.
- Restituisce il puntatore `s`, oppure `NULL` se incontra la fine del file o se riscontra un errore.

Lettura e scrittura di stringhe

- `char *fgets (char *s, int max, FILE *pf)`
- L'abbiamo già citata. Legge la prossima riga da `pf`, fino al più a `max-1` caratteri, pone quanto letto in `s`, aggiungendo in coda il terminatore `'\0'`.
- Nota. Include anche l'eventuale `'\n'` fra i caratteri letti.
- Restituisce il puntatore `s`, oppure `NULL` se incontra la fine del file o se riscontra un errore.

- `int fputs (char *s, FILE *pf)`
- Scrive la stringa `s` sul file `pf`.
- Restituisce un valore non negativo se la chiamata va a buon fine, e `EOF` se riscontra un errore.

scrivileggi.c

```c
#include<stdio.h>

int main(void)
{
        FILE *pf; //puntatore a file
        char letta[BUFSIZ]; //buffer di lettura
        char s[]="Ego te absolvo..."; //una stringa

        if ((pf=fopen("test", "a")) == NULL) //pf e' associato al file "test"
        {
                printf("Errore I/O.\n");
                return 0;
        }

        fprintf(pf, "Ecco un intero: %d\n", 45);
        fprintf(pf, "Ecco un double: %lf\n", 3.1415);
        fprintf(pf, "Ecco una stringa: %s\n", s);

        fclose(pf);

        if ((pf=fopen("test", "r")) == NULL) //pf e' associato al file "test"
        {
                printf("Errore I/O.\n");
                return 0;
        }

        while ( fgets(letta, BUFSIZ, pf)!= NULL ) //stampa su stdout una riga alla volta.
                printf("%s",letta);
        fclose(pf);

    return 0;
}
```

Lettura di stringhe: funzioni deprecate e sicurezza

Abbiamo già citato la funzione

$$\texttt{char} * \texttt{gets}(\texttt{char} * \texttt{s})$$

Essa è "deprecata" dagli standard C recenti, e non va mai usata in programmi reali. (Si usa `fgets` al suo posto, anche per leggere da `stdin`.) E' quindi consigliabile abituarsi fin da subito all'uso di `fgets`.

Lettura di stringhe: funzioni deprecate e sicurezza

Abbiamo già citato la funzione

$$\texttt{char} * \texttt{gets}(\texttt{char} * \texttt{s})$$

Essa è "deprecata" dagli standard C recenti, e non va mai usata in programmi reali. (Si usa `fgets` al suo posto, anche per leggere da `stdin`.) E' quindi consigliabile abituarsi fin da subito all'uso di `fgets`.

Il motivo per cui l'uso di `gets` è deprecato è che i programmi che la usano possono andare in buffer overflow: per cenni, ciò accade perché `gets` può leggere più caratteri dello spazio di memoria (*buffer*) che il programmatore ha allocato per memorizzarli, il che può permette a un programma malevolo (*malware*) di attaccare il programma in questione e accedere al sistema su cui esso gira. Sono argomenti trattati nei corsi di Sicurezza Informatica.