Linguaggio C

Appunti per il corso di Laboratorio di Algoritmi e Strutture Dati

Stefano Aguzzoli

Alcune note introduttive

- Orario lezioni: Martedì: 18:30 – 21:30.
  Lezioni frontali in Aula 202,
  Esercitazioni in laboratorio in Aula 307,
  Settore Didattico, Via Celoria

- Frontali: 4,11,25 ottobre; 15,29 novembre; 13 dicembre; 10 gennaio.

- Laboratorio: 18 ottobre; 8,22 novembre; 6,20 dicembre; 17 gennaio.

- Eventuali Variazioni saranno comunicate per tempo sul sito: `http://homes.dsi.unimi.it/~aguzzoli/algo.htm`.

- Orario ricevimento: Mercoledì: 15:00 - 16:00

- Lucidi: in pdf su: `http://homes.dsi.unimi.it/~aguzzoli/algo.htm`

- Per supporto e altri strumenti relativi al C consultare anche il sito: `http://www.algoteam.dsi.unimi.it`

Programma del corso

Il corso verterà sull'insegnamento del linguaggio C e sull'uso del C per implementare strutture dati e algoritmi.

Il linguaggio C:

- Introduzione al C, panoramica sul linguaggio.
- Elementi lessicali, Tipi di dati fondamentali.
- Espressioni. Flusso del controllo.

- Funzioni

- Array, puntatori e stringhe

- Strutture e Unioni

- Strutture dati evolute

- Input e Output, C e UNIX

- Libreria Standard

Testi consigliati

- *Programmazione in C*
  (C Programming: A Modern Approach (second edition)),
  K. N. King.
  W. W. Norton & Company, 2008.

- *C Didattica e Programmazione* (A book on C, 4th edition),
  Al Kelley, Ira Pohl. Pearson, Italia, 2004.

- *Linguaggio C* (seconda edizione),
  Brian W. Kernighan, Dennis M. Ritchie.
  Nuova edizione italiana, Pearson, Italia, 2004.

Compilatori

Useremo solo compilatori aderenti allo standard ANSI.

Per Linux: `gcc`.
Già installato sulla maggior parte delle distribuzioni.

Per Windows: `gcc`.
`MinGW` contiene `gcc`.
Una “distribuzione” facile da installare e usare: `cs1300`.
Scaricatela da
`http://homes.dsi.unimi.it/~aguzzoli/algo.htm`
o da `http://www.cs.colorado.edu/~main/cs1300`

Modalità d’esame

- viene proposta la specifica di un problema da risolvere mediante tecniche viste a lezione
- avete alcuni giorni per sottoporre una soluzione in C (sorgente gcc-compilabile) e un annesso tecnico
- controllo di correttezza/originalità
- colloquio

Voto finale combinato di **corso+laboratorio**

Consigli per il progetto d'esame

- i temi riguardano problemi computazionalmente difficili: l'uso improprio delle strutture dati comporta un voto basso

- l'interfaccia utente è quasi nulla

- la portabilità è un fattore importante

- l'annesso tecnico è importante

- i commenti al codice sono importanti

Cosa è il C.

- Cenni Storici.
  - Progettato nel 1972 da D. M. Ritchie presso i laboratori AT&T Bell, per poter riscrivere in un linguaggio di alto livello il codice del sistema operativo UNIX.
  - Definizione formale nel 1978 (B.W. Kernighan e D. M. Ritchie)
  - Nel 1983 è iniziato il lavoro di definizione dello standard (ANSI C) da parte dell'American National Standards Institute.
  - Standard ANSI (ISO C89) rilasciato e approvato nel 1990.

Alcune caratteristiche (positive) del C

- Elevato potere espressivo:
  - Tipi di dato primitivi e tipi di dato definibili dall'utente
  - Strutture di controllo
  (programmazione strutturata, funzioni e procedure)

- Caratteristiche di basso livello
  (gestione delle memoria, accesso diretto alla rappresentazione, puntatori, operazioni orientate ai bit)

- (N.B. Stiamo parlando di un linguaggio imperativo e per la programmazione strutturata: non si parla proprio di programmazione orientata agli oggetti).

- Stile di programmazione che incoraggia lo sviluppo di programmi per passi di raffinamento successivi (sviluppo top-down)

- Sintassi definita formalmente. Linguaggio *piccolo*

- Codice efficiente e compatto

- Ricche librerie (standard) per operazioni non definite nel linguaggio (input/output, memoria dinamica, gestione stringhe, funzioni matematiche, ...)

- Rapporto *simbiotico* col sistema operativo (UNIX)

Lo standard ANSI:

- standard chiaro, consistente e non ambiguo

- modifica alcune regole del C tradizionale (Kernighan e Ritchie, K&R) e ne stabilisce altre.

- compromesso che migliora la portabilità mantenendo alcune caratteristiche machine-dependent.

- C++ si basa sullo standard ANSI C.

Alcuni difetti:

- La sintassi non è immediatamente leggibile e a volte ostica. (vedremo cosa succede quando si combinano array, puntatori, puntatori a funzione, ...)

- La precedenza di alcuni operatori è “sbagliata”.

- E’ possibile scrivere codice estremamente contorto.

Lo standard o gli standard ?

Il processo di standardizzazione non è terminato, e andrà avanti fino a quando il C si continuerà ad usare e presenterà aspetti migliorabili.

Quando parliamo di standard ANSI in questo corso ci riferiremo sempre allo standard ISO C89. Il corso illustrerà le caratteristiche del C conformi a ISO C89.

I compilatori, anche `gcc`, contemplano anche altri standard: estensioni successive di ISO C89.
Le caratteristiche del linguaggio aggiunte negli standard successivi non sono fondamentali per un utilizzo pienamente soddisfacente del C nell'implementazione di algoritmi.
Vedremo fra poco alcune opzioni di `gcc` relative alla gestione degli standard e delle varianti del linguaggio.

Una prima panoramica sul linguaggio C

Salve mondo!

Tradizionalmente, il primo programma C è il seguente:

```
/* prog1.c */
#include <stdio.h>

int main(void)
{
    printf("Salve mondo\n");
    return 0;
}
```

Il C è un linguaggio *compilato*.
Per compilare il programma dell'esempio:

```
gcc -o prog1 prog1.c
```

```
gcc -o prog1 prog1.c
```

`gcc` è il comando che richiama il compilatore.
`-o prog1` è un'opzione di `gcc` che specifica il nome dell'eseguibile.
`prog1.c` è il nome del file contenente il programma
I nomi dei file contenenti codice sorgente C hanno estensione `.c`

Se la compilazione ha successo viene prodotto un file eseguibile:
`prog1.exe` sotto Windows, `prog1` sotto Linux.

Per eseguire il programma:

`prog1` sotto Windows, `./prog1` sotto Linux.

Il programma stampa sul video la scritta

```
Salve mondo!
```

Analisi del programma:

```
/* prog1.c */
```

Commenti: I commenti in C iniziano con `/*` e finiscono con `*/`, tutto ciò che questi due delimitatori contengono è ignorato dal compilatore

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

Preprocessore: Le righe il cui primo carattere non di spaziatura è `#` sono *direttive* per il **preprocessore**. In questo caso si richiede di espandere sul posto il contenuto del file di intestazione `stdio.h`. Tale file contiene il *prototipo* della funzione `printf`.

```
int main(void)
{
```

Definizione di funzione: Si definisce la funzione `main` come una funzione che non ha alcun parametro formale (`void`), e che restituisce un valore di *tipo* intero `int`.

Ogni programma C definisce una funzione `int main(...)`: l'esecuzione del programma partirà invocando `main`, a sua volta `main` richiamerà altre funzioni e così via. Come vedremo, `main` può essere invocata con parametri, specificandone opportunamente la definizione prototipale.

Il *corpo* della funzione è racchiuso tra le parentesi graffe { e }.

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

Funzione di libreria standard: `printf` è una funzione contenuta nella libreria standard del C: per informare il compilatore del suo uso corretto in termini di numero e tipo dei parametri e del tipo di ritorno, bisogna includere il file di intestazione `stdio.h`. `printf` realizza un output formattato. `\n` è il carattere *newline*. Il carattere ; marca la fine di un istruzione.

```
return 0;
```

Istruzione `return`: L'istruzione `return` *expr* ritorna il valore di *expr* alla funzione chiamante. In questo caso, poiché `main` deve ritornare un intero (perché ?), si ritorna 0 al sistema operativo. `return 0` può essere omesso al termine della definizione di `main`.

Sistema C e organizzazione del codice

Il sistema C è formato dal linguaggio C, dal preprocessore, dal compilatore, dalle librerie e da altri strumenti di supporto.

Un programma C è costituito da un insieme di definizioni di funzioni. L'esecuzione parte dalla definizione della funzione `int main(...)`.

Un programma C può essere diviso su diversi file.

La produzione dell'eseguibile avviene in 3 fasi:
Nella prima fase, ogni file da compilare è trattato dal **preprocessore**. Le direttive al preprocessore determinano modifiche al file che le contiene, quali inclusione di file, espansione di macro, compilazione condizionale.

Prova: `gcc -E prog1.c > vedi` per vedere l'output del preprocessore.

Nella seconda fase ogni file da compilare preprocessato origina un file in formato *oggetto*, non ancora eseguibile.

Le librerie sono molto simili a file oggetto: infatti contengono la versione oggetto di moltissime funzioni predefinite.

La terza fase, a carico del *linker*, vede la produzione di un unico file eseguibile a partire dai file oggetto aggiornati e dalle parti necessarie delle librerie.

I file di intestazione (*name*.`h`) contengono le definizioni necessarie a ogni file sorgente per essere compilato in formato oggetto in modo tale da poter essere successivamente collegato agli altri file che fanno uso di definizioni condivise. Lo stesso discorso si applica all'uso di funzioni di librerie.

Nel nostro esempio precedente, la riga

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

è una direttiva al preprocessore che richiede l'inclusione del file di intestazione `stdio.h`.
L'effetto di tale inclusione è il rimpiazzamento nel file `prog1.c` della riga contenente la direttiva con il contenuto del file `stdio.h` stesso.
Tale file contiene definizioni necessarie alle funzioni di input/output della libreria standard, e di altre di uso comune. Ad esempio vi è il prototipo:

```
int printf(const char *format,...);
```

che informa il compilatore della natura degli argomenti di `printf` e di ciò che tale funzione può ritornare.

Il file `stdio.h` contiene non solo prototipi di funzione, ma in genere definisce elementi quali definizioni di tipi, di costanti, di macro, direttive al preprocessore. Ad esempio vi è la definizione della costante numerica `EOF` che rappresenta, in opportune situazioni, la fine di un file. Tipicamente:

```
#define EOF     (-1)
```

Le parentesi angolari < e > che racchiudono il nome del file da includere, comunicano al preprocessore di cercare il file in una serie di locazioni (directory) standard, in un ordine prefissato. Se si fosse scritto

```
#include "stdio.h"
```

la ricerca del preprocessore sarebbe iniziata dalla directory corrente.

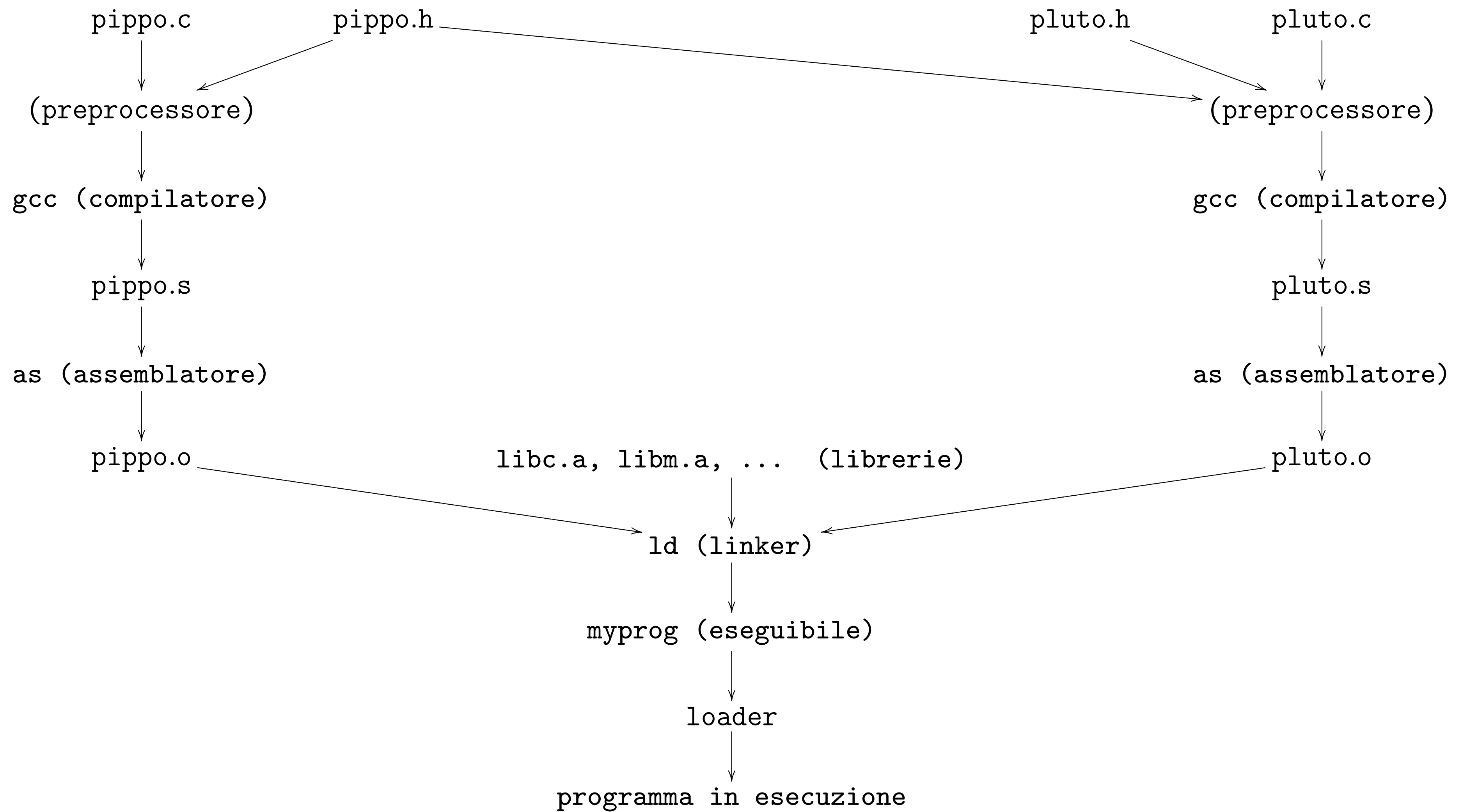
ESEMPIO:

Si supponga di compilare il progetto costituito dai due file `pippo.c`, `pluto.c` con i due file di intestazione `pippo.h` (supponiamo incluso da entrambi i due file `.c`) e, `pluto.h` (incluso solo da `pluto.c`):

Il progetto si compila con il comando `gcc -o myprog pippo.c pluto.c`

Si noti che non si devono specificare i nomi dei file di intestazione: essi sono inclusi dalle direttive al preprocessore `#include` contenute nei file sorgente `pippo.c`, `pluto.c`.

Ecco, nel prossimo lucido, una descrizione diagrammatica del processo di compilazione completo, dalla fase di preprocessamento all'esecuzione.


pippo.c
pippo.h
pluto.h
pluto.c
(preprocessore)
(preprocessore)
gcc (compilatore)
gcc (compilatore)
pippo.s
pluto.s
as (assemblatore)
as (assemblatore)
pippo.o
libc.a, libm.a, ... (librerie)
pluto.o
ld (linker)
myprog (eseguibile)
loader
programma in esecuzione

Alcune utili opzioni di `gcc`

`gcc` contempla diverse opzioni per controllare l'aderenza dei codici sorgenti allo standard ANSI ISO C89, o a sue successive estensioni.

Usare `-ansi` o `-std=c89` per controllare l'aderenza allo standard ISO C89.

Con queste opzioni `gcc` rimane ancora un po' *permissivo* e non si *lamenta* di alcuni usi non conformi allo standard (es. dichiarazioni inframmezzate a istruzioni, array con dimensioni non note a tempo di compilazione. etc...).

Usare `-pedantic` per forzare un controllo di stretta aderenza.

Usare `-Wall` per avere un rapporto su usi del linguaggio permessi ma che frequentemente sono abbinati a errori concettuali nella stesura del codice.

I messaggi segnalati da queste opzioni sono solo degli avvertimenti: la compilazione non si blocca a causa di questi problemi, e, a meno di altri errori, produce infine l'eseguibile.

È bene considerare sempre gli avvertimenti sollevati a causa di frammenti di codice non conformi allo standard, poiché tali frammenti pregiudicano la portabilità del codice.

Gli avvertimenti sollevati da `-Wall` vanno di volta in volta valutati confrontandoli con le intenzioni del programmatore: non sempre sono errori, ma usi intenzionali di alcune caratteristiche del C.

Prova a compilare i sorgenti allegati `warn1.c, ..., warn4.c`

Vediamo un altro esempio:

```
/* prog2.c */
#include <stdio.h>

#define LIRE_PER_EURO   (1936.27)

int main(void)
{
    const int n_euro = 5;
    double n_lire = n_euro * LIRE_PER_EURO;

    printf("%d euro valgono %.3f lire\n",n_euro,n_lire);
    return 0;
}
```

Lo si compili con

```
gcc -o prog2 prog2.c
```

Analisi del programma:

```
#define LIRE_PER_EURO   (1936.27)
```

Definizione di macro: La direttiva `#define` definisce una *macro*. Il preprocessore sostituirà dovunque nel file appaia la stringa `LIRE_PER_EURO` con la stringa `(1936.27)`. Le parentesi non sono obbligatorie nelle definizioni di macro, ma spesso sono necessarie per evitare alcuni effetti collaterali.

```
    const int n_euro = 5;
    double n_lire = n_euro * LIRE_PER_EURO;
```

Definizione di variabili: Queste righe definiscono due *variabili* locali alla funzione `main`.
La variabile `n_euro` è dichiarata di tipo `int`. Il tipo `int` è uno dei tipi base fondamentali del C.

Una variabile di tipo `int` può contenere valori interi positivi o negativi compresi in un intervallo dipendente dall'implementazione. `n_euro` è inizializzata con il valore intero 5.

La variabile `n_lire` è dichiarata di tipo `double`. Il tipo `double` è uno dei tipi base fondamentali. Una variabile di tipo `double` può contenere valori *in virgola mobile* positivi o negativi in *doppia precisione*. Per valori in *singola precisione* esiste il tipo `float`. `n_lire` è inizializzata con il risultato dell'espressione del membro destro.

Qualificatore `const`: La parola chiave `const` qualifica la variabile `n_euro` come costante. Il valore di `n_euro` non potrà essere cambiato nel corso del programma.
Nota: la moltiplicazione di `int` per `double` è automaticamente convertita in `double`.

```
printf("%d euro valgono %.3f lire\n",n_euro,n_lire);
```

Output formattato La funzione `printf` della libreria standard viene usata per stampare una stringa sullo schermo.

Il primo argomento di `printf` è una *stringa di formato* che specifica il numero e il tipo dei successivi argomenti. Se il tipo del $(k+1)$esimo argomento di `printf` combacia (o è automaticamente convertibile) col tipo richiesto dal $k$esimo argomento della stringa formato, allora il valore di tale argomento viene sostituito nella stringa che verrà stampata.

Gli argomenti della stringa formato, nella loro forma più semplice, vengono specificati dal carattere % seguito da un carattere che ne determina il significato.

Gli argomenti più comuni della stringa formato sono:
`%d` — intero decimale (vale a dire, in base 10)
`%c` — carattere
`%u` — intero decimale senza segno
`%s` — stringa (le vedremo)
`%x` o `%X` — intero esadecimale (base 16: cifre:0,1,...,9,a,b,...,f)
`%f` — numero in virgola mobile
`%e` o `%E` — numero in virgola mobile in notazione scientifica
`%g` o `%G` — numero in virgola mobile, il più corto tra `%e` e `%f`.

E' possibile specificare l'ampiezza e la precisione dei campi da stampare (esempio: `%.3f` specifica una precisione di 3 cifre decimali per il numero in virgola mobile).

Consideriamo ora il seguente programma organizzato su 3 file:

```
/* prog3.h */
#include <stdio.h>
#define LIRE_PER_EURO   (1936.27)

int geteuro(void);
void putlire(int, double);

/* prog3.c */
#include "prog3.h"

int main(void)
{
    int n_euro = geteuro();
    double n_lire = n_euro * LIRE_PER_EURO;

    putlire(n_euro,n_lire);
    return 0;
}
```

```c
/* aux3.c */
#include "prog3.h"

int geteuro(void)
{
    int ne;

    printf("Immetti numero di euro:\n");
    scanf("%d",&ne);
    return ne;
}

void putlire(int euro, double lire)
{
    printf("%d euro valgono %.3f lire\n",euro,lire);
}
```

Analisi del programma:

Il programma `prog3` è suddiviso su tre file.

- `prog3.h` è un file di intestazione che contiene le definizioni necessarie agli altri due file per poter essere collegati tra loro e con le funzioni di libreria utilizzate.

  In particolare, `prog3.h` include a sua volta il file `stdio.h`. Inoltre contiene la definizione della macro `LIRE_PER_EURO` e i prototipi delle due funzioni presenti in `aux3.c`.

  In questo modo, quando il compilatore trasformerà `prog3.c` in codice oggetto `prog3.o`, saprà quanti parametri e di quale tipo necessitano le funzioni usate in questo file ma definite in altri file. Inoltre conoscerà il tipo di ritorno di queste funzioni.

Notare che nei prototipi di funzione non è necessario riportare alcun nome per i parametri formali: tutto ciò che serve al compilatore è il loro tipo.

- `prog3.c` include il file di intestazione `prog3.h`.

  Questo file contiene un'unica funzione: `main`, che richiama le altre definite altrove.

  La funzione priva di argomenti `geteuro` restituisce un intero. Tale valore intero viene utilizzato per inizializzare la variabile `n_euro`.

  La funzione `putlire` non restituisce alcun valore, e necessita di due argomenti: il primo di tipo `int` e il secondo di tipo `double`.

- `aux3.c` contiene le definizioni delle due funzioni `geteuro` e `putlire`. Inoltre include il file di intestazione `prog3.h`.

  Nella funzione `geteuro` viene utilizzata la funzione `scanf` della libreria standard, il cui prototipo è in `stdio.h`. Tale funzione controlla l'input in dipendenza da una stringa di formato, che stabilisce che tipo di valori siano da leggere dall'input. `scanf` è duale di `printf`. La stringa formato determina numero e tipo dei valori da leggere, che vengono assegnati alle corrispondenti variabili presenti come argomenti successivi nell'invocazione di `scanf`, purché i tipi coincidano.

  Nel nostro esempio: `scanf("%d",&ne);` viene chiesto di leggere un valore intero decimale e di porre il valore letto nella variabile intera `ne`.

`&ne` applica l'operatore *indirizzo di ...* alla variabile `ne`. Infatti a `scanf` viene passato l'*indirizzo* della variabile, non la variabile stessa.

Il tipo dell'indirizzo di una variabile `int` è `int *`, vale a dire *puntatore a* `int`. Come vedremo, in C si fa largo uso esplicito dei puntatori.

L'utilizzo di parametri di tipo puntatore in `scanf` costituisce il modo standard del C per realizzare passaggi di parametri per referenza.

Infatti l'unica modalità nel passaggio di parametri realizzata direttamente nel C è il *passaggio per valore*.
Discuteremo diffusamente di ciò più avanti.

Come si compila `prog3` ?

Per compilare un programma su più file come `prog3`:

```
gcc -o prog3 prog3.c aux3.c
```

Si noti che il file di intestazione `prog3.h` non figura nel comando. Infatti si riportano solo i file di cui bisogna produrre una versione in formato oggetto: `prog3.o`, `aux3.o`.
Nel caso in cui, per esempio, `aux3.o` fosse già disponibile nella versione aggiornata, posso creare il file eseguibile con:

```
gcc -o prog3 prog3.c aux3.o
```

In questo caso `aux3.o` sfugge alle fasi di preprocessamento e compilazione, e viene solo usato dal linker per creare il file eseguibile.

Per creare uno (o più) file oggetto senza richiamare il linker:

```
gcc -c aux3.c
```

Questo modo di operare è spesso vantaggioso, soprattutto per progetti di molte migliaia di righe di codice: in questi casi, ripartire il codice su più file (in cui ogni file deve essere visto come un modulo contenente il codice per realizzare un determinato tipo di funzionalità) risulta conveniente per diverse ragioni.

Innanzitutto il codice così ripartito è più comprensibile e facilmente consultabile.

Inoltre si ha un effettivo vantaggio in termini di tempi di compilazione in quanto si devono ricompilare solo i file modificati dall'ultima compilazione, che, presumibilmente, in ogni momento dello sviluppo, non saranno molti.

L'utility `make` fornisce un valido aiuto nella gestione di progetti distribuiti su numerosi file. `make` è uno strumento generico e non si applica solo alla programmazione in C.

Nel file `makefile` si esplicitano le dipendenze fra i file che costituiscono il progetto.
Nel nostro caso il `makefile` apparirà così:

```
prog3: prog3.o aux3.o
    gcc -o prog3 prog3.o aux3.o

prog3.o: prog3.c prog3.h
    gcc -c prog3.c

aux3.o: aux3.c prog3.h
    gcc -c aux3.c
```

lanciando il comando `make` verranno effettuati solo i passi di compilazione strettamente necessari: i file aggiornati non verranno toccati.

Consideriamo l'esempio:

```
/* prog4.c: scrive la prima parola di input.txt su output.txt */
#include <stdio.h>

#define MAXWORD 80

int main(void) {
    FILE *fr = fopen("input.txt","r");
    FILE *fw = fopen("output.txt","w");
    char word[MAXWORD];

    fscanf(fr,"%s",word);
    fprintf(fw,"Parola letta: %s\n",word);

    /* Le ultime tre linee sono superflue in questo caso,
       poiche' sono le ultime istruzioni ad essere eseguite
       e i file verrebbero comunque chiusi automaticamente al
       termine dell'esecuzione del programma */
    fclose(fr);
    fclose(fw);
    return 0;
}
```

```
FILE *fr = fopen("input.txt","r");
FILE *fw = fopen("output.txt","w");
```

La struttura `FILE` è definita in `stdio.h` e permette l'accesso bufferizzato ai files.

La struttura `FILE` si utilizza attraverso puntatori: `fr` e `fw` sono di tipo: *puntatore a* `FILE`.

`fopen`, definita in `stdio.h`, restituisce un puntatore a `FILE`.
`fopen` apre il file il cui nome è il suo primo argomento, nella modalità specificata dal secondo argomento: "`r`": lettura, "`w`": scrittura, "`a`": accodamento.

Esistono anche le modalità “`r+`”, “`w+`”, “`a+`”, che aprono il file in aggiornamento (lettura *e* scrittura).

e anche “`rb`”, “`wb`”, “`ab`”, (e “`rb+`”, “`wb+`”, “`ab+`”), che specificano che il file sarà considerato un file binario: su UNIX/Linux non c’è differenza, nei sistemi Microsoft i file di testo sono rappresentati diversamente dai file binari.

`fscanf`, `fprintf`: analoghe a `scanf` e `printf`,
prototipo definito in `stdio.h`,
hanno un argomento aggiuntivo, il primo, che è un puntatore al file su cui operare: in lettura `fscanf` e in scrittura `fprintf`.

`fclose` chiude un file aperto.
Alla termine dell’esecuzione tutti i file aperti vengono chiusi automaticamente.

```
/* bubble.c */
#include <stdio.h>
#define DIM     10
void bubble(int [], int);  /* prototipo di bubble */

int main(void)
{
    int i, num[DIM];
    FILE *fr = fopen("numeri.txt","r");
    FILE *fw = fopen("ordinati.txt","w");
    /* leggo DIM numeri dal file */
    for(i = 0; i < DIM; i++)
        fscanf(fr,"%d",&num[i]);
    /* li ordino con bubblesort */
    bubble(num,DIM);
    /* li stampo sul file */
    for(i = 0; i < DIM; i++)
        fprintf(fw,"%d ",num[i]);
    fprintf(fw,"\n");
}
```

```
/* bubble ordina i primi n elementi
   dell'array di interi a[],
   usando bubblesort.
*/
void bubble(int a[], int n)
{
    int i,j,tmp;

    for(i = 0; i < n - 1; i++)
        /* sistemo l'i-esimo elemento piu' piccolo */
        for(j = n - 1; j > i; j--)
            if(a[j-1] > a[j]) {
                /* scambio */
                tmp = a[j-1];
                a[j-1] = a[j];
                a[j] = tmp;
            }
}
```

Analisi di `bubble.c`

```
void bubble(int [], int);
```

Prototipo della funzione `bubble`, definita dopo. Poiché `main` la usa, il compilatore deve già conoscerne il prototipo al momento di definire `main`.
Dichiarazione di tipo array: `int []` specifica che il primo argomento di `bubble` deve essere un *array* di `int`. Equivalentemente, avrei potuto usare `int *`, vale a dire *puntatore* a `int`. Vedremo come e perché gli *argomenti di funzione* di tipo array sono in realtà di tipo puntatore.

```
    int i, num[DIM];
```

Dichiarazione di variabili: locali a `main`. `i` è una variabile intera, `num` è un *array* di `DIM` interi. La dimensione di un array deve essere un'espressione costante intera.

```
for(i = 0; i < DIM; i++)
```

Istruzione di controllo del flusso: Il C, ovviamente, possiede istruzioni per controllare il flusso, vale a dire selezione ed iterazione. Il ciclo `for` in C è un costrutto molto flessibile e potente.

```
fscanf(fr,"%d",&num[i]);
```

Input formattato: legge un intero scritto in notazione decimale (`%d`) nella $i$esima componente di `num`. Notare che si passa a `scanf` l'indirizzo di `num[i]` (operatore `&`), per realizzare il passaggio per referenza.

Per accedere all'$i$esimo elemento di `num` si usa la notazione `num[i]`. Gli indici di un array C partono da 0.

```
    bubble(num,DIM);
```

Chiamata di funzione: viene invocata la funzione `bubble` passando l'array `num` (ne basta il nome) come primo argomento, e la sua dimensione come secondo.

```
void bubble(int a[], int n)
{ ... }
```

Definizione di funzione: viene definita `bubble`, si noti che qui i nomi dei parametri formali `a` e `n`, non sono tralasciati (cfr. con prototipo).

`bubble` è un'implementazione su `int` dell'algoritmo di ordinamento *bubblesort*, uno dei più ingenui e inefficaci algoritmi pensati per questo scopo. Il suo tempo di esecuzione è proporzionale al quadrato del numero di elementi da ordinare.

```
for(i = 0; i < n - 1; i++)
    for(j = n - 1; j > i; j--)
        ...
```

Corpo della definizione di `bubble`: Due cicli nidificati: il ciclo più interno, muove verso sinistra l'$i$esimo elemento più piccolo, attraverso confronti e, nel caso, *scambi*, fra il candidato attuale ad essere l'$i$esimo, e l'elemento immediatamente precedente.

```
        if(a[j-1] > a[j]) {
            /* scambio */
            tmp = a[j-1];
            a[j-1] = a[j];
            a[j] = tmp;
        }
```

Effettua lo scambio se la condizione `a[j-1] > a[j]` risulta vera.

NB. In C ogni espressione che restituisca un valore aritmetico può essere considerata una condizione: se il risultato è uguale a `0` allora la condizione risulta falsa, altrimenti risulta vera.

Viceversa, gli operatori logici, relazionali e d'uguaglianza restituiscono valori aritmetici (`1` per vero, `0` per falso).

Le prestazioni non sono buone:

Dopo $n-1$ iterazioni del ciclo più esterno tutti gli elementi sono al loro posto.

Numero quadratico di confronti: $ncfr = \frac{n(n-1)}{2}$.

Ovviamente il numero di scambi è $\leq ncfr$.

Vediamo ora di modificare leggermente il codice di `bubble` per visualizzare meglio come funziona l'algoritmo e per valutare sugli esempi il numero di scambi effettuati.

```
int bubble(int a[], int n, FILE *fw)
{
    int i,j,k,tmp,nswap = 0;

    for(i = 0; i < n - 1; i++) {
        for(j = n - 1; j > i; j--) {
            for(k = 0; k < j-1; k++)
                fprintf(fw,"%d%c",a[k],k == i-1 ? '|':' ');
            if(a[j-1] > a[j]) {
                tmp = a[j-1]; a[j-1] = a[j]; a[j] = tmp;
                fprintf(fw,"%d*%d ",a[j-1],a[j]);
                nswap++;
            } else
                fprintf(fw,"%d %d ",a[j-1],a[j]);
            for(k = j+1; k < DIM; k++)
                fprintf(fw,"%d ",a[k]);
            fprintf(fw,"\n");
        }
        fprintf(fw,"\n");
    }
    return nswap;
}
```

Commenti sulla versione modificata di `bubble`:

Il prototipo è stato modificato: ora `bubble` ritornerà un intero (il numero di scambi effettuato) e riceverà come terzo argomento un puntatore a `FILE` su cui stampare.

Sono state inserite istruzioni per meglio evidenziare la dinamica dell'algoritmo.

```
k == i-1 ? '|':' '
```

Operatore ternario: Funziona come una versione contratta di *if-then-else*. L'espressione: *expr* `?` *expr1* `:` *expr2* ha il valore di *expr1* quando *expr* $\neq 0$, di *expr2* altrimenti.

```
nswap++
```

Operatore di incremento: equivale a `nswap = nswap + 1`.

Parametri passati per indirizzo. Puntatori

Vogliamo modificare la parte del codice di `bubble` che realizza lo scambio dei valori:

```
tmp = a[j-1]; a[j-1] = a[j]; a[j] = tmp;
```

sostituiamo questa riga con un'invocazione a una funzione di prototipo `void swap(int a, int b)` che scambi i suoi due valori:

Proviamo la seguente versione:

```
void swap(int a, int b)
{
    int tmp = a;

    a = b;
    b = tmp;
}
```

Non funziona, perché ?

Il C supporta unicamente il passaggio di parametri per valore.

Dunque, nella chiamata `swap(x,y)` vengono passati i valori di `x` e `y`. Questi valori vengono localmente manipolati (scambiati) senza nessun effetto sull'ambiente chiamante.

La soluzione in C: Passaggio per referenza realizzato passando il *valore* degli *indirizzi*:

```
void swap(int *a, int *b)
{
    int tmp = *a;

    *a = *b;
    *b = tmp;
}
```

Il tipo *indirizzo* di una variabile di tipo intero è il tipo **puntatore** a intero.

Per dichiarare un puntatore a `int`:
`int *a;`
La dichiarazione è contorta: la si legga come: *l'oggetto puntato da* `a` *è un intero*.

Infatti, nelle espressioni, come `*b = tmp;` , `*` è l'operatore di *dereferenziazione* che restituisce l'oggetto puntato.

Si legga `*b = tmp;` come: *assegna all'oggetto puntato da* `b` *il valore contenuto in* `tmp`.

Per invocare questa versione di `swap` bisogna passare due indirizzi validi di oggetti di tipo `int`. Eventualmente usando l'operatore `&` (operatore *indirizzo di ...*, duale di `*`).

Quindi per scambiare due valori interi:

```
...
int x = 5;
int y = 7;

swap(&x,&y); /* ora x contiene 7 e y contiene 5 */
```

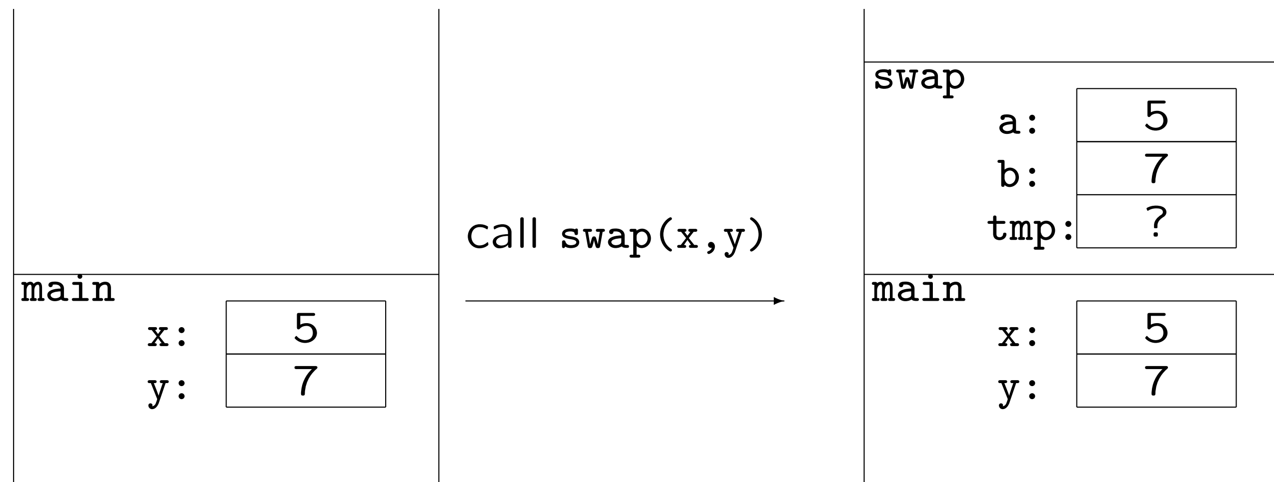
La funzione `bubble` diventa:

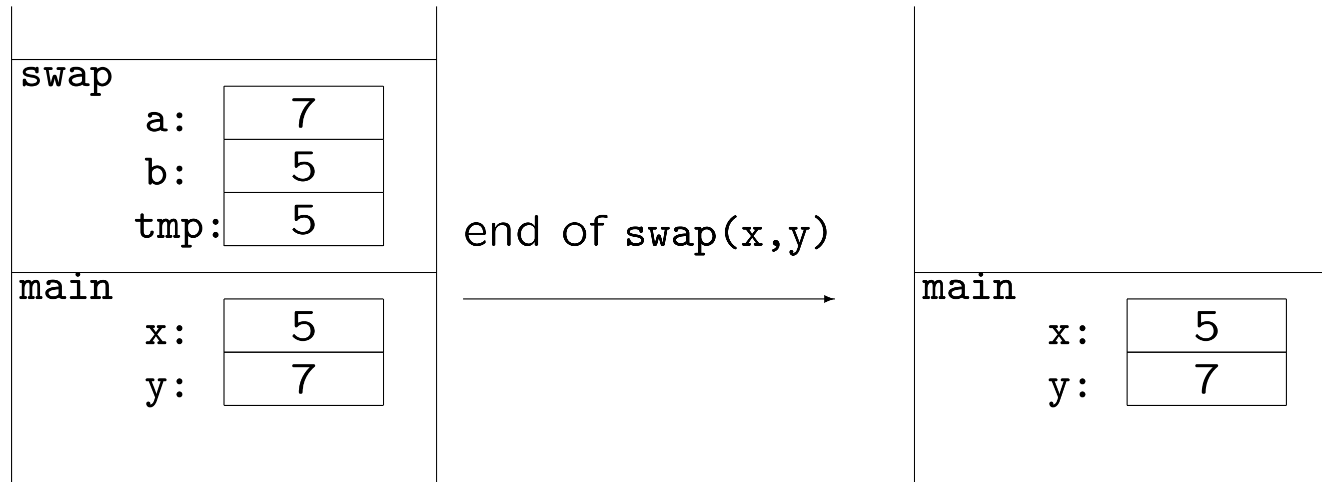
```
void bubble(int a[], int n)
{
    int i,j;

    for(i = 0; i < n - 1; i++)
        /* sistemo l'i-esimo elemento piu' piccolo */
        for(j = n - 1; j > i; j--)
            if(a[j-1] > a[j]) {
                /* scambio */
                swap(&a[j-1],&a[j]);
            }
}
```

Il comportamento di swap (nelle due diverse implementazioni)
Si consideri il frammento di codice seguente dove la versione di `swap` usata è quella con il prototipo `void swap(int a, int b)`:

```
int main(void) {
    int x = 5;
    int y = 7;
    swap(x,y);
}
```

Schematizziamo il contenuto dello stack dei record di attivazione:

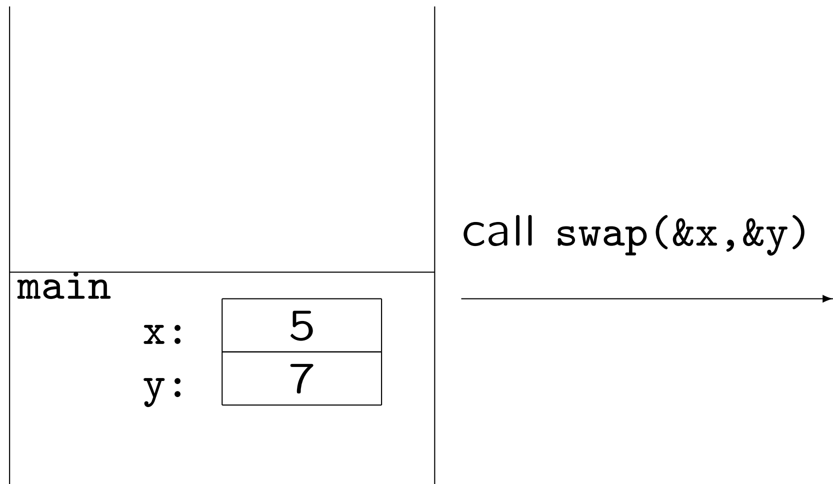


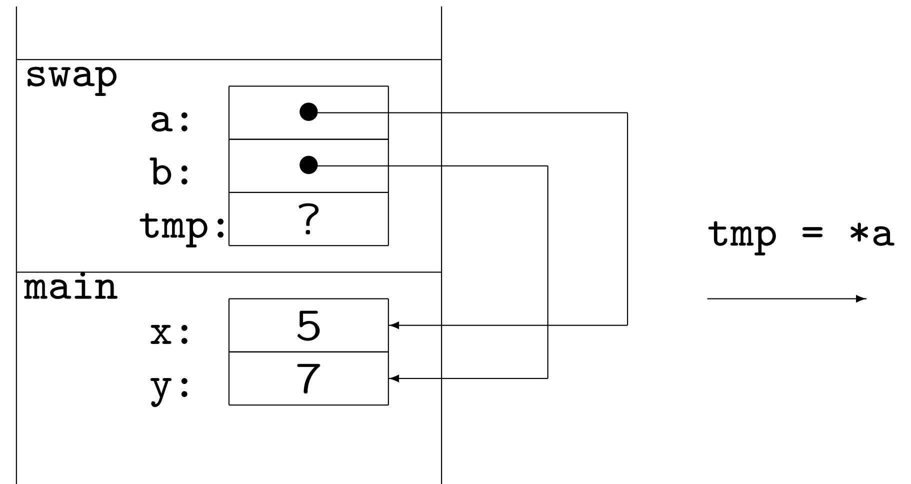


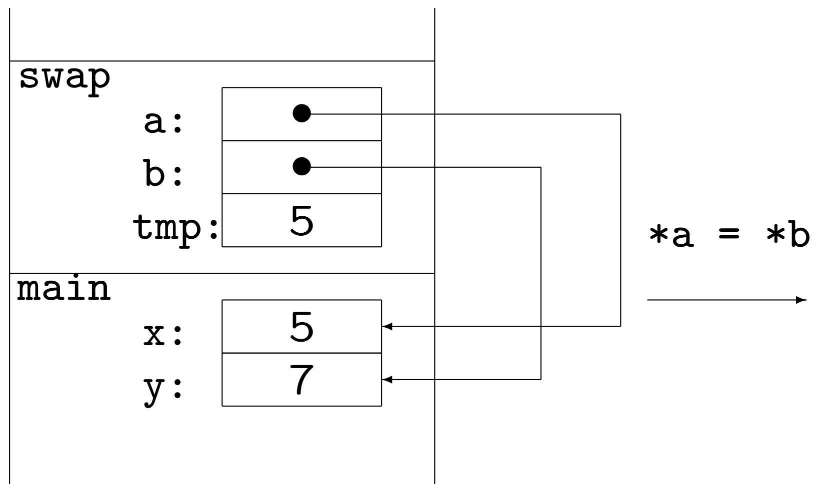
Non abbiamo ottenuto nulla da `swap`.

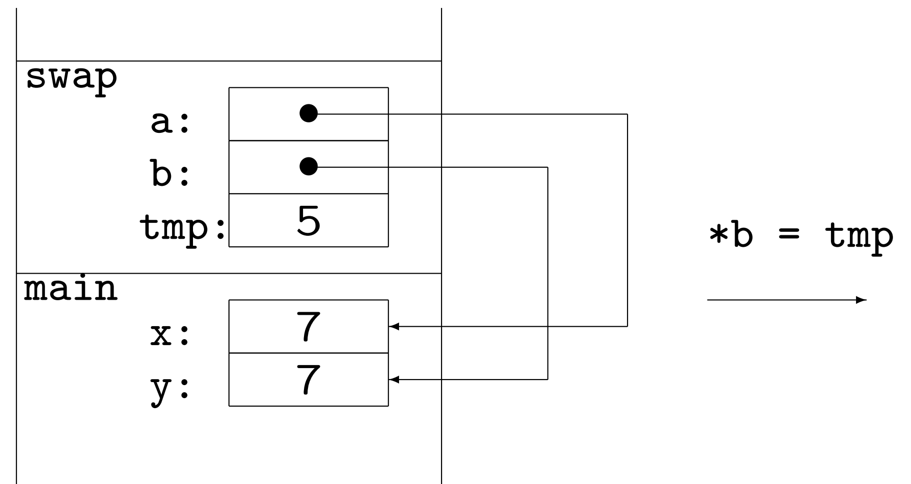
Si consideri ora la seconda definizione, quella con prototipo:
`void swap(int *a, int *b)`

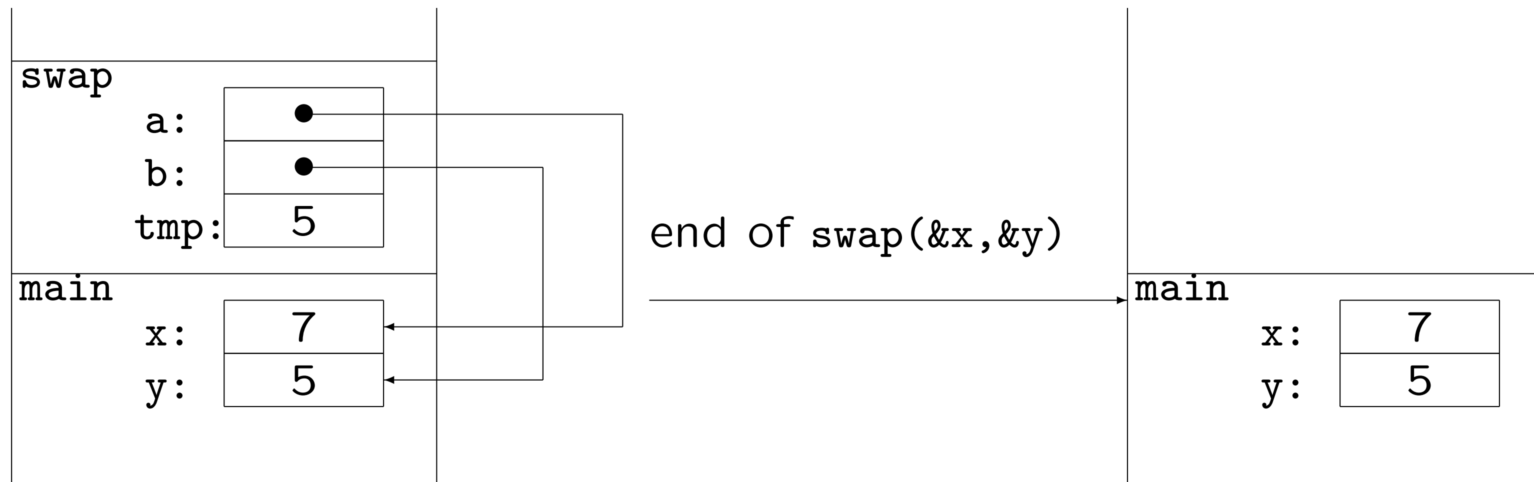
```
int main(void) {
    int x = 5;
    int y = 7;
    swap(&x,&y);
}
```


main
x:
5
y:
7
call swap(&x,&y)


swap
a:
b:
tmp:
?
main
x:
5
y:
7
tmp = *a


swap
a:
b:
tmp:
5
main
x:
5
y:
7
*a = *b


swap
a:
b:
tmp:
5
main
x:
7
y:
7
*b = tmp



Il passaggio (*sempre per valore*) degli indirizzi di `x` e `y` a `swap` ha permesso a questa funzione di accedere alla regione di memoria associata a `x` e `y`, attraverso l'uso dell'operatore di dereferenziazione `*`.

Questo è il modo standard per simulare un passaggio di parametri per riferimento in C.

Un accenno alla relazione tra array e puntatori

Il nome di un array è un puntatore al primo elemento dell'array stesso.

Il prototipo di `bubble` (1ª vers.) può infatti essere specificato come:

```
void bubble(int *a, int n)
```

(o anche `void bubble(int *, int)` se stiamo dichiarando solo il prototipo e non siamo in sede di definizione della funzione).

Quando si richiama `bubble`, nulla cambia:

```
bubble(num, DIM);
```

`num`, il nome dell'array, è un'espressione che punta al primo elemento dell'array stesso:

```
num == &num[0];
```

Le **stringhe** sono array di elementi di tipo `char`, il cui ultimo elemento è `0`, o, detto in modo equivalente, il carattere `'\0'`.

```
char s[] = "pippo";
```

è equivalente a

```
char s[6] = "pippo";
```

è equivalente a

```
char s[] = {'p','i','p','p','o','\0'};
```

è equivalente a (NB: non del tutto equivalente, vedremo in che senso più avanti):

```
char *s = "pippo";
```

Strutture
Per aggregare variabili di tipo diverso sotto un unico nome il C fornisce le strutture (e le unioni).

```
struct punto {
    int x;
    int y;
};

struct rettangolocoloratoepesato{
    int colore;
    double peso;
    struct punto bottomleft, topright;
};

struct punto sommapunti(struct punto a, struct punto b)
{
    a.x += b.x;
    a.y += b.y;
    return a;
}
```

Elementi lessicali, operatori, espressioni

Elementi lessicali

Un programma C dal punto di vista sintattico è una sequenza di caratteri che il compilatore deve tradurre in codice oggetto.

Il programma deve essere sintatticamente corretto.

Concentriamoci sul processo di compilazione vero e proprio (trasformazione da `.c` a `.o`).
La prima fase di questo processo contempla il riconoscimento degli elementi lessicali: Raggruppamento dei caratteri in *token*.

```
int main(void)
{
    printf("Salve mondo\n");
    return 0;
}
```

Token: `int`,`main`,`()`,`void`, { ,`printf`, `"Salve mondo\n"`,`;`,`return`,0, }

`int`,`void`,`return`: **parole chiave**

`main`,`printf`: **identificatori**

`"Salve mondo\n"`: **costante stringa**

`0`: **Costante intera**

`()`: **Operatore (chiamata di funzione)**

;, }, { : **Segni d'interpunzione**

Regole sintattiche

La sintassi del C può essere espressa formalmente mediante regole *Backus-Naur Form* (BNF).

Esempio: Definizione della categoria sintattica *digit* (*cifra*)

$$digit ::= 0|1|2|3|4|5|6|7|8|9$$

$\|$	alternativa
$\{\}_1$	scegliere esattamente una delle alternative nelle graffe
$\{\}_{0+}$	ripetere la categoria tra graffe 0 o più volte
$\{\}_{1+}$	ripetere la categoria tra graffe 1 o più volte
$\{\}_{opt}$	opzionale

Esempio: $alphanumeric_string ::= \{letter|digit\}_{0+}$

Identificatori:

$identifier ::= \{letter|_\}_1\{letter|_|digit\}_{0+}$

Costanti intere decimali:

$decimal_integer ::= 0|positive_decimal_integer$
$positive_decimal_integer ::= positive_digit\,\{digit\}_{0+}$
$positive_digit ::= 1|2|3|4|5|6|7|8|9$

Costanti stringa:

$string_literal ::= "\ \{character|escape_sequence\}_{0+}\ "$
$escape_sequence ::= \backslash'|\backslash"|\backslash?|\backslash\backslash|\backslash\mathtt{a}|\backslash\mathtt{b}|\backslash\mathtt{f}|\backslash\mathtt{n}|\backslash\mathtt{r}|\backslash\mathtt{t}|\backslash\mathtt{v}|\backslash octal|\backslash\mathtt{x}\,hexadecimal.$

dove *character* non è " ne' \

Espressioni

Le espressioni sono combinazioni significative di **costanti**,**variabili**, **operatori** e **chiamate di funzioni**.

Costanti, variabili e chiamate di funzioni sono di per se’ delle espressioni.

Un operatore combina diverse espressioni e produce un’espressione più complessa. il numero delle espressioni combinate è la *arità* dell’operatore.

Esempi:

```
i = 7
a + b
33 - bubble(num,DIM)
-4.207
"pippo"
```

Operatori

Ogni operatore possiede le seguenti proprietà:

arità: specifica il numero degli argomenti necessari per applicare l'operatore.

priorità: determina in che ordine vengono eseguite le operazioni in espressioni che coinvolgono altri operatori.

associatività: determina in che ordine vengono eseguite le operazioni in espressioni che coinvolgono altre occorrenze di operatori con la stessa priorità.

Per forzare un ordine diverso da quello determinato da priorità e associatività, si deve parentesizzare l'espressione con) e (.

Il C dispone degli ordinari operatori aritmetici, con le usuali arità, priorità e associatività.

```
(1 + 2 * 3) == (1 + (2 * 3))
```

```
(1 + 2 - 3 + 4 - 5) == ((((1 + 2) - 3) + 4) - 5)
```

Operatori aritmetici (in ordine di priorità):

+, -	(unari)	Associatività: da destra a sinistra
*, /, %		Associatività: da sinistra a destra
+, -	(binari)	Associatività: da sinistra a destra

Gli operatori aritmetici si applicano a espressioni intere o floating.
Vi è inoltre un'*aritmetica dei puntatori* (vedremo).

Operatori di incremento e decremento: `++` e `--`

`++` e `--` sono operatori unari con la stessa priorità del meno unario e associatività da destra a sinistra.

Si possono applicare solo a variabili (o meglio *lvalue*, come vedremo), di tipi interi, floating o puntatori, ma non a espressioni generiche (anche se di questi tipi).

```
int c = 5;

c++;  /* c == 6 */
c--;  /* c == 5 */
--c;  /* c == 4 */
++c;  /* c == 5 */
5++;  /* Errore! */
--(7 + bubble(num,DIM)); /* Errore! */
```

Semantica degli operatori di incremento/decremento

Postfisso: Il valore dell' espressione `c++` è il valore di `c`. Mentre `c` stesso è incrementato di 1.

Prefisso: Il valore dell' espressione `++c` è il valore di `c` incrementato di 1. Inoltre `c` stesso è incrementato di 1.

Analogo discorso vale per gli operatori di decremento.

```
int c = 5;

int b = 30 / c++; /* b == 6, c == 6 */
int d = 6 + --c;  /* d = 11, c == 5 */
```

Gli operatori prefissi modificano il valore della variabile cui sono applicati prima che se ne utilizzi il valore. Gli operatori postfissi modificano il valore della variabile dopo l'utilizzo del valore (vecchio) nell'espressione.

Operatori di assegnamento

In C, l'assegnamento è un'espressione:

```
int a,b,c;

a = b = c = 0;
printf("%d\n",b = 4); /* stampa: 4 */
```

Il valore di un'espressione *variabile* = *expr* è il valore assunto da *variabile* dopo l'assegnamento, vale a dire, il valore di *expr*.

Non ci si confonda con l'operatore di uguaglianza ==

```
int b = 0;

printf("%d,", b == 4); printf("%d,", b = 4); printf("%d\n", b == 4);
/* Che cosa stampa? */
```

Operatori di assegnamento

In C, l'assegnamento è un'espressione:

```
int a,b,c;

a = b = c = 0;
printf("%d\n",b = 4); /* stampa: 4 */
```

Il valore di un'espressione *variabile* = *expr* è il valore assunto da *variabile* dopo l'assegnamento, vale a dire, il valore di *expr*.

Non ci si confonda con l'operatore di uguaglianza ==

```
int b = 0;

printf("%d,", b == 4); printf("%d,", b = 4); printf("%d\n", b == 4);
/* stampa: 0,4,1 */
```

L'operatore di assegnamento ha priorità più bassa rispetto agli operatori aritmetici, e associa da destra a sinistra.

Operatori di assegnamento combinati:

```
+=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

La loro semantica è:
variabile op= expr equivale a *variabile = variabile op expr*
(ma *variabile* è valutata una volta sola, questo aspetto è da ricordare quando *variabile* è una componente di un array, di una struttura, etc., esempio: `a[i++] += 2;`)
variabile può essere qualsiasi *lvalue* (vedi lucido seguente)

Esempio:

```
int a = 1;

a += 5; /* a == 6 */
```

Variabili e *lvalue*

È il momento di essere più precisi nel parlare di variabili e oggetti.

Per *oggetto* intendiamo un'area di memoria alla quale è stato associato un nome.

Un *lvalue* è un'espressione che si riferisce a un oggetto.

Solo gli *lvalue possono comparire a sinistra* di un operatore di assegnamento.

Le variabili semplici sono *lvalue*, ma sono *lvalue* anche (i "nomi") di elementi di array (es . `a[i] = ...`), le espressioni consistenti nella dereferenziazione di un puntatore (es. `*ptr = ...`), i campi delle variabili `struct` (es. `p.x = ...`), e altre costruzioni ancora (le vedremo).

Esempio di utilizzo di operatori di assegnamento combinati

```
/* power2.c */
#include <stdio.h>

#define MAXEXP  10

int main(void)
{
    int i = 0, power = 1;

    while(++i <= MAXEXP)
        printf("%6d", power *= 2);
    printf("\n");
    return 0;
}
```

`while(++i <= MAXEXP)` : il ciclo viene ripetuto fino a quando `i` non sarà maggiore di `MAXEXP`.
`power *= 2` : ad ogni iterazione, il valore di `power` viene raddoppiato, e poi stampato.

Un esempio di utilizzo di operatori di incremento e di assegnamento

```
#include <stdio.h>

void copia(char *t, char *s)
{
    int i = 0;

    while((t[i] = s[i]) != 0)
        i++;
}

int main(void)
{
    char s[] = "pippo";
    char t[10];

    copia(t,s);
    printf(t);
    return 0;
}
```

Commenti:

```
while((t[i] = s[i]) != 0)
```

Si esce dal ciclo quando la *condizione* argomento di `while` diventa falsa, più precisamente, quando l'*espressione* argomento di `while` viene valutata `0`.

```
(t[i] = s[i]) != 0
```

Poiché l'assegnamento è un'espressione, qui se ne usa il valore: quando essa varrà 0 il ciclo terminerà. Essa varrà 0 quando varrà 0 la variabile `t[i]`, vale a dire quando varrà 0 la variabile `s[i]`.

```
i++
```

a ogni iterazione, il valore di `i` viene incrementato.

s:	'p'	'i'	'p'	'p'	'o'	0

i

t:	?	?	?	?	?	?	?	?	?	?

`t[i] = s[i]` vale `'p'`

s:	'p'	'i'	'p'	'p'	'o'	0

i

t:	'p'	?	?	?	?	?	?	?	?	?

`t[i] = s[i]` vale `'i'`

• • •

`t[i] = s[i]` vale `'o'`

s:	'p'	'i'	'p'	'p'	'o'	0

i

t:	'p'	'i'	'p'	'p'	'o'	0	?	?	?	?

`t[i] = s[i]` vale 0,

cioè la condizione del ciclo è *falsa*

Esercizio:

Cosa succede se rimpiazzo `copia` con:

```
void copia(char *t, char *s)
{
    int i = 0;

    while((t[i++] = s[i++]) != 0);
}
```

e se la rimpiazzo con:

```
void copia(char *t, char *s)
{
    int i = 0;

    while(t[i] = s[i])
        ++i;
}
```

Cosa succede se invece rimpiazzo `copia` con:

```
void copia(char *t, char *s)
{
    int i;

    for(i = 0;t[i] = s[i];i++);
}
```

Si noti che in questo caso l'istruzione `for` è immediatamente seguita da `;`: il ciclo è vuoto, nessuna istruzione viene ripetuta sotto il controllo del ciclo `for`. Solo l'espressione di controllo viene valutata ad ogni iterazione.

```
t[i] = s[i]
```

L'espressione di controllo è costituita semplicemente dall'assegnamento. Poiché l'assegnamento è un'espressione, il ciclo terminerà quando questa espressione varrà `0`. Cioè quando `s[i]`, che è il valore assegnato a `t[i]` ed è anche il valore di tutta l'espressione, sarà `0`.

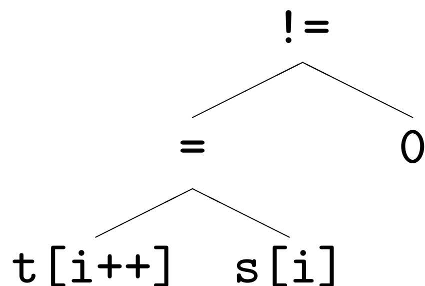
N.B. Abbiamo implementato `copia` per esercizio. La libreria standard contiene la funzione `strcpy`, il cui prototipo è in `string.h`.

Consideriamo questa versione di `copia`:

```
void copia(char *t, char *s)
{
    int i = 0;

    while((t[i++] = s[i]) != 0);
}
```

Funziona ?

E se rimpiazziamo la condizione del ciclo `while` con `(t[i] = s[i++]) != 0` ?



Associatività e Priorità determinano
la forma dell'albero di parsing,
ma non specificano del tutto l'ordine di valutazione.
L'espressione può essere valutata in due modi diversi
a seconda di quale sottoalbero di `=` viene valutato per primo.
Lo standard non risolve questo tipo di ambiguità.

Dichiarazioni, tipi fondamentali

Dichiarazione di variabili

In C tutte le variabili devono essere dichiarate prima di essere utilizzate.

```
int main(void) {
    int x,y,z;   /* dichiarazioni */
    double alpha = 3.5, beta, gamma; /* dichiarazioni con inizializzazione */
    int i = 0; /* dichiariamo un altro intero */
    ...
}
```

Le dichiarazioni permettono al compilatore di allocare lo spazio necessario in memoria: a una variabile di tipo T sarà riservato in memoria lo spazio necessario a contenere valori di tipo T.

Informano il compilatore su come gestire le variabili dichiarate a seconda del loro tipo: ad esempio il compilatore esegue la somma di due `int` in modo diverso dalla somma di due `double`, poichè la rappresentazione interna di questi due tipi è diversa.

Blocchi: Una porzione di codice racchiusa fra parentesi graffe { e } costituisce un **blocco**.
Le dichiarazioni, se presenti, devono precedere tutte le altre istruzioni del blocco.
Una variabile dichiarata in un blocco è visibile solo fino alla fine del blocco stesso. Può inoltre essere *occultata* in blocchi interni a quello in cui è dichiarata da un'altra variabile con lo stesso nome.

Cosa stampa il frammento seguente?

```
{
    int a = 2;
    {
        double a = 3.0;
        printf("a piu' interno -2.0: %f\n",a - 2.0);
    }
    printf("a meno interno -2.0: %f\n",a - 2.0);
}
```

Una variabile dichiarata in un blocco si dice *locale* al blocco.

Il blocco più esterno è quello dove si dichiarano le funzioni: le variabili dichiarate qui sono *globali*.

```
double b = 3.14; /* b e' una variabile globale */

int main(void)
{
        printf("b globale -2.0: %f\n",b - 2.0);
}
```

Le variabili globali sono visibili in tutto il file da dove sono state dichiarate in poi. La loro visibilità può essere modificata dichiarandole `extern` o `static` (vedremo in che senso).

Sintassi: Per la dichiarazione di variabili dei tipi fondamentali:

$$declaration ::= \quad type \quad declarator_list \,;$$
$$declarator_list ::= \quad declarator \quad | \quad \{\, , \, declarator\}_{opt}$$
$$declarator ::= \quad identifier \quad | \quad identifier \, = \, initializer$$

Vedremo poi come dichiarare puntatori, array, funzioni, strutture, etc.

$initializer$ è un'arbitraria espressione il cui valore appartiene al tipo $type$ della variabile $identifier$.

";" : Poichè una dichiarazione è un'istruzione, deve essere terminata da ;.
In una sola dichiarazione, però, si possono dichiarare più variabili dello stesso tipo, separandole con "," e inizializzandole o meno.

Inizializzazione e Assegnamento

L'inizializzazione non è necessaria.
Se non è presente: le variabili locali non dichiarate `static` conterranno un valore casuale, le variabili globali vengono inizializzate a 0. Alcuni compilatori inizializzano a 0 anche le variabili locali.

L'inizializzazione è concettualmente diversa dall'assegnamento.
L'assegnamento modifica il valore di una variabile già esistente.
L'inizializzazione è contestuale alla dichiarazione e fornisce il valore iniziale alla variabile appena creata.

Se dichiariamo una variabile come `const` non potremo più usare `a` come operando sinistro di un'operatore di assegnamento o di incremento/decremento:

```
const int a = 5;

a = 7;  /* Errore! */
--a;    /* Errore! */
```

Tipi Fondamentali

I tipi di dati fondamentali in C sono:

`char`	`signed char`	`unsigned char`
`signed short int`	`signed int`	`signed long int`
`unsigned short int`	`unsigned int`	`unsigned long int`
`float`	`double`	`long double`

I tipi derivati: puntatori, array, strutture, unioni, ..., sono costruiti a partire dai tipi fondamentali.

E' possibile usare alcune abbreviazioni:

`char`	`signed char`	`unsigned char`
`short`	`int`	`long`
`unsigned short`	`unsigned`	`unsigned long`
`float`	`double`	`long double`

I tipi fondamentali possono anche essere chiamati *tipi aritmetici*, poiché oggetti di questi tipi possono comparire come operandi di operatori aritmetici.

`float`,`double`,`long double` costituiscono l'insieme dei *tipi reali*: rappresentano mediante *virgola mobile* numeri con parte frazionaria.

Gli altri tipi aritmetici sono detti *tipi interi*.

I tipi interi `signed` possono rappresentare interi negativi. Di solito la rappresentazione interna è in *complemento a 2*.

I tipi interi `unsigned` rappresentano interi non-negativi.

Il tipo `char`

Quando si pensa ai tipi interi in C, è spesso utile pensare in termini di occupazione di memoria:

Un `char` in C può spesso essere visto come un intero "piccolo", e quindi utilizzato in contesti aritmetici. Viceversa, è possibile utilizzare qualunque tipo intero per rappresentare caratteri.

Un `char` occupa 1 byte di memoria.

In genere quindi, un `char` può memorizzare 256 valori differenti.

Le costanti carattere (es.: `'x'`,`'3'`,`';'`) *sono* di tipo `int`! (in C++ sono `char`). Comunque, una costante carattere può essere assegnata a un `char`:

```
char c = 'a';
```

Si noti la differenza:

```
char c;
c = '0';
c = 0;
```

`c = '0'`: Assegna alla variabile `c` di tipo `char` il valore corrispondente al carattere `'0'` (in ASCII: 48).

`c = 0`: Assegna a `c` il valore intero 0.

Possiamo fare lo stesso con ogni tipo intero:

```
short s;
c =  1;
s = '1';
printf("%d\n",s+c);
```

Cosa viene stampato ?

```
/* char.c */
#include <stdio.h>

int main(void)
{
    char c;
    short s;

    c = '0';
    printf("%s:  intero: %d, carattere: %c\n",
        "c = '0'",c,c);

    c = 0;
    printf("%s:  intero: %d, carattere: %c\n",
        "c = 0",c,c);

    c = 1;
    s = '1';
    printf("s+c:  %d\n",s+c);
}
```

Commenti:

```
printf("%s: intero: %d, carattere: %c\n","c = '0'",c,c);
```

Il primo argomento (`%s`) della stringa formato di `printf` è una stringa: in questo caso la stringa costante `"c = '0'"`.

Il secondo argomento è un intero da visualizzare in base 10: qui passiamo `c` che è di tipo `char`.

Il terzo argomento è un intero da visualizzare come carattere: qui passiamo ancora `c`.

```
printf("s+c: %d\n",s+c);
```

Qui passiamo come argomento `%d` il risultato della somma dello `short s` con il `char c`.

Vi sono alcuni caratteri non stampabili direttamente, ma attraverso una *sequenza di escape*:

carattere	**sequenza**	valore ASCII
allarme	\a	7
backslash	\\	92
backspace	\b	8
carriage return	\r	13
newline	\n	10
doppi apici	\"	34
form feed	\f	12
tab	\t	9
vertical tab	\v	11
apice	\'	39
carattere nullo	\0	0

Si può specificare il valore ASCII di una costante carattere in ottale (`es: '\007'`), o esadecimale (`es: '\x30'`).

Su macchine con byte di 8 bit:

`unsigned char` rappresenta l'intervallo di interi $[0, 255]$,
`signed char` rappresenta l'intervallo di interi $[-128, 127]$.
`char` equivale a uno fra `unsigned char` e `signed char`: dipende dal compilatore.

```
#include <stdio.h>

int main(void)
{
    char c = -1;
    signed char s = -1;
    unsigned char u = -1;

    printf("c = %d, s = %d, u = %d\n",c,s,u);
}
```

Provare a compilarlo con le opzioni `-funsigned-char` e `-fsigned-char` di gcc.

Il tipo `int`

L'intervallo di valori rappresentato dal tipo `int` dipende dalla macchina: solitamente un `int` viene memorizzato in una *parola* della macchina.

Assumiamo che un `int` occupi 4 byte (da 8 bit).
Allora `int` rappresenta l'intervallo di interi:

$$[-2^{31}, 2^{31} - 1] = [-2147483648, 2147483647]$$

Gli `int` sono rappresentati in complemento a 2.

Se durante la computazione si cerca di memorizzare in un `int` un valore esterno all'intervallo, si ha *integer overflow*. Solitamente l'esecuzione procede, ma i risultati non sono affidabili.

Le costanti intere possono essere scritte anche in ottale e esadecimale.

I tipi `short`, `long` e `unsigned`

`short` si usa quando si vuole risparmiare spazio e i valori interi in gioco sono "piccoli".

Spesso uno `short` occupa 2 byte. In tal caso `short` rappresenta l'intervallo $[-2^{15}, 2^{15}-1] = [-32768, 32767]$.

`long` si usa quando gli interi in gioco sono così grandi da non poter essere contenuti in un `int`: questa soluzione è efficace se effettivamente il tipo `long` occupa più byte del tipo `int`.

Lo standard ANSI assicura solo che `short` occupa al più tanti byte quanto `int`, che a sua volta occupa al più tanti byte quanto `long`.

In generale, se un tipo intero con segno T è memorizzato in k byte, allora T rappresenta l'intervallo $[-2^{8k-1}, 2^{8k-1} - 1]$.

I tipi `unsigned` sono memorizzati usando la stessa quantità di byte delle corrispettive versioni con segno.

Se il tipo con segno T rappresenta l'intervallo $[-2^{8k-1}, 2^{8k-1} - 1]$ allora `unsigned` T rappresenta l'intervallo $[0, 2^{8k} - 1]$.

Ad esempio, se gli `int` occupano 4 byte, allora gli `unsigned` rappresentano l'intervallo $[0, 2^{32} - 1] = [0, 4294967295]$.

I tipi `unsigned` sono trattati con l'aritmetica modulo 2^{8k}.

```
/* modulare.c */
#include <stdio.h>
#include <limits.h>

int main(void)
{
    int i;
    unsigned u = UINT_MAX;  /* valore massimo per tipo unsigned */

    for(i = 0;i < 10;i++)
        printf("%u + %d = %u\n",u,i,u+i);
    for(i = 0;i < 10;i++)
        printf("%u * %d = %u\n",u,i,u*i);
}
```

`limits.h`: File di intestazione che contiene macro che definiscono i valori limite per i tipi interi (es. `UINT_MAX`).

Ogni costante intera può essere seguita da un suffisso che ne specifichi il tipo.

Suffisso	**Tipo**	**Esempio**
`u` oppure `U`	`unsigned`	`37U`
`l` oppure `L`	`long`	`37L`
`ul` oppure `UL`	`unsigned long`	`37ul`

Se non viene specificato alcun suffisso per una costante, viene automaticamente scelto il primo fra i seguenti tipi:

`int`, `long`, `unsigned long`,

che possa rappresentare la costante.

I tipi reali `float`, `double`, `long double`

Per rappresentare costanti reali bisogna usare il punto decimale per differenziarle dalle costanti intere:

`3.14` è una costante `double`.
`3` è una costante `int`.
`3.0` è una costante `double`.

E' disponibile anche la notazione scientifica:

`1.234567e5` è `123456.7`
`1.234567e-3` è `0.001234567`

I tipi reali vengono rappresentati in virgola mobile.

Il numero di byte per rappresentare un `double` non è inferiore al numero di byte usato per rappresentare un `float`.

Il numero di byte per rappresentare un `long double` non è inferiore al numero di byte usato per rappresentare un `double`.

Su molte macchine, si usano 4 byte per `float` e 8 per `double`.

Con questa rappresentazione non tutti i reali in un dato intervallo sono rappresentabili, inoltre le operazioni aritmetiche non forniscono valori esatti, ma approssimati a un certo numero di cifre decimali.

La rappresentazione in virgola mobile è caratterizzata da due parametri: *precisione* e *intervallo*.

La *precisione* descrive il numero di cifre significative rappresentabili in notazione decimale.

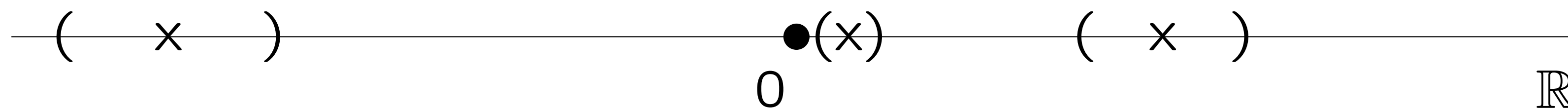
L'*intervallo* specifica il più piccolo e il più grande fra i valori reali rappresentabili.

Su macchine con `float` di 4 byte, la precisione del tipo `float` è di 6 cifre significative e un intervallo di circa $[10^{-38}, 10^{38}]$.

Su macchine con `double` di 8 byte, la precisione del tipo `double` è di 15 cifre significative e un intervallo di circa $[10^{-308}, 10^{308}]$.

La rappresentazione in virgola mobile è caratterizzata da due parametri: *precisione* e *intervallo*.

La *precisione* finita implica che non tutti i reali appartenenti all'*intervallo* sono rappresentabili: valori reali diversi possono essere rappresentati nello stesso modo (e quindi resi indistinguibili), se sufficientemente vicini rispetto al loro valore.



Nella figura gli intervalli fra tonde schematizzano insiemi di reali che in forma decimale coincidono per tutte le cifre significative. Ogni due numeri che appartengono allo stesso intervallo ( ) non sono distinguibili dalla rappresentazione in virgola mobile.

Rappresentiamo al più $2^{8*\texttt{sizeof(double)}}$ reali (in realtà razionali) diversi.

Le costanti reali in C sono di tipo `double`.

Per modificare questa impostazione di default, si può specificare un suffisso:

Suffisso	**Tipo**	**Esempio**
`f` oppure `F`	`float`	`3.7F`
`l` oppure `L`	`long double`	`3.7L`

In C, il tipo "naturale" per le operazioni sui reali è il tipo `double`.

Anche le funzioni della libreria matematica standard, solitamente restituiscono `double` e richiedono argomenti `double`.

```
double sqrt(double); /* prototipo per funz.  radice quadrata */
```

L'operatore `sizeof`

`sizeof(`A`)` fornisce la dimensione in byte dell'oggetto A o del tipo A.

`sizeof` ha la stessa priorità degli altri operatori unari.

Devono valere le seguenti relazioni:

```
sizeof(char) == 1
sizeof(short) <= sizeof(int) && sizeof(int) <= sizeof(long)
sizeof(unsigned) == sizeof(int)
sizeof(float) <= sizeof(double) && sizeof(double) <= sizeof(long double)
```

```c
#include <stdio.h>

void stampasizeof(char *s, int size)
{
    printf("%s:%d\n",s,size);
}

int main(void)
{
    stampasizeof("char",sizeof(char));
    stampasizeof("short",sizeof(short));
    stampasizeof("int",sizeof(int));
    stampasizeof("long",sizeof(long));
    stampasizeof("float",sizeof(float));
    stampasizeof("double",sizeof(double));
    stampasizeof("long double",sizeof(long double));
}
```

Un esercizio sull'uso del preprocessore:

```
#include <stdio.h>

#define stampa(s)   stampasizeof(#s,sizeof(s))

void stampasizeof(char *s, int size)
{
    printf("%s:%d\n",s,size);
}

int main(void)
{
    stampa(char);
    stampa(short);
    stampa(int);
    stampa(long);
    stampa(float);
    stampa(double);
    stampa(long double);
}
```

Commenti:

Si sono usate alcune caratteristiche del preprocessore per rendere più "eleganti" le chiamate alla funzione che stampa il nome del tipo (una stringa costante) e la sua dimensione.

```
#define stampa(s) stampasizeof(#s,sizeof(s))
```

Definizione di macro con parametri: `s` è un parametro della macro. Quando la macro è richiamata, come in `stampa(short)`, la stringa `short` viene sostituita (in fase di preprocessamento) a ogni occorrenza di `s` nel testo della macro `stampa`.

`#s` : direttiva al preprocessore che determina la sostituzione della stringa `short` con la stringa `"short"`.

Un altro esempio di *macro con parametri*

```
#define max(A,B)    ((A) >= (B) ? (A) : (B))

printf("Il maggiore tra %d e %d e' %d \n",9,7,max(7,9));
```

Perchè tante parentesi ?
Sono necessarie per evitare effetti collaterali della macro-espansione.
Si consideri infatti la definizione (errata):

```
#define maxbis(A,B)    A >= B ? A : B

int x = 5 * maxbis(7,9); /* il valore di x sara' 7 */
```

In questo esempio la macro viene espansa come
`5 * 7 >= 9 ? 7 : 9`. Poiché la priorità di `*` è maggiore della priorità di `?:`, e dato che $35 \geq 9$, il risultato dell'intera espressione è 7.

Conversioni e Cast

Ogni espressione aritmetica ha un tipo.

Promozioni: oggetti `char` e `short` possono essere utilizzati in tutte le espressioni dove si possono usare `int` o `unsigned`.
Il tipo dell'espressione è convertito in `int` se tutti i tipi coinvolti nell'espressione possono essere convertiti in `int`.
Altrimenti, il tipo dell'espressione è convertito in `unsigned`.

```
char c = 'A';

printf("%c\n",c);
```

Nel secondo argomento di `printf`, `c` è convertito (promosso) a `int`.

Conversioni Aritmetiche

Determinano il tipo di un'espressione in cui un operatore aritmetico ha operandi di tipi differenti:

- Se uno degli operandi è `long double` anche l'altro è convertito in `long double`.

- Altrimenti, se uno è `double`, l'altro è convertito in `double`.

- Altrimenti, se uno è `float`, l'altro è convertito in `float`.

- Negli altri casi si applicano le promozioni intere e:

Negli altri casi si applicano le promozioni intere e:

- Se uno degli operandi è `unsigned long` anche l'altro è convertito in `unsigned long`.

- Se uno è `long` e l'altro è `unsigned`:
  – se `sizeof(unsigned) < sizeof(long)` l'operando `unsigned` è convertito in `long`.
  – altrimenti entrambi convertiti in `unsigned long`.

- Altrimenti, se uno è `long`, l'altro è convertito in `long`.

- Altrimenti, se uno è `unsigned`, l'altro è convertito in `unsigned`.

- Altrimenti, entrambi gli operandi sono già stati promossi a `int`.

```
char c; short s; int i; long l;
unsigned u; unsigned long ul;
float f; double d; long double ld;
```

Espressione	Tipo
`c - s / i`	`int`
`u * 2.0 - 1`	`double`
`c + 3`	`int`
`c + 5.0`	`double`
`d + s`	`double`
`2 * i / l`	`long`
`u * 7 - i`	`unsigned`
`f * 7 - i`	`float`
`7 * s * ul`	`unsigned long`
`ld + c`	`long double`
`u - ul`	`unsigned long`
`u - l`	`dipendente dal sistema`

Cast

Per effettuare esplicitamente delle conversioni si usa il costrutto di *cast*.

Sintassi: (*type*) *expr*

Semantica: il valore dell'espressione *expr* viene convertito al tipo *type*.

```
double x = 5.3;
int i = (int) x; /* x viene convertito in int */
```

L'operatore di cast ha la stessa priorità degli altri operatori unari.

Input/Output con `getchar()` e `putchar()`

Quando si deve gestire l'input/output a livello di singoli caratteri, è preferibile usare le *macro* `getchar()` e `putchar()` definite in `stdio.h` piuttosto che `scanf` e `printf`.

`scanf` e `printf` realizzano input/output *formattato* (attraverso il loro primo argomento `char *format`), e sono molto meno efficienti di `getchar()` e `putchar()` che semplicemente leggono/scrivono un carattere alla volta.

```
int c;

c = getchar();  /* legge un carattere e lo pone in c  */
putchar(c);     /* scrive il carattere contenuto in c */
```

```
/* toupper.c */
#include <stdio.h>

int main(void)
{
    int c;

    while((c = getchar()) != EOF)
        if(c >= 'a' && c <= 'z')
            putchar(c + 'A' - 'a');
        else
            putchar(c);
    return 0;
}
```

Converte l'input trasformando le lettere minuscole in maiuscole.

(N.B.: la funzione `toupper()` è già nella libreria standard: per usarla includere `ctype.h`, vedi `toupperctype.c` nel file `.zip` relativo a questa lezione)

Forniamogli un testo di prova in input: `toupper < testo.txt`

Commenti su `toupper.c`:

`stdio.h` : Contiene le definizioni delle macro `EOF`,`getchar`,`putchar`.

`int c` : La variabile `c`, destinata a contenere un carattere alla volta del testo in input, è dichiarata `int`, perchè ?

`while((c = getchar()) != EOF)` :
— `c = getchar()` : viene posto in `c` il valore `int` restituito da `getchar()`. Il valore dell'espressione di assegnamento è il valore restituito da `getchar()`.
— `(c = getchar()) != EOF` : il valore dell'espressione di assegnamento viene confrontato con il valore `EOF` di tipo `int`: questo valore speciale viene restituito da `getchar()` quando raggiunge la fine del file.

– `(c = getchar()) != EOF` : Si noti la parentesizzazione: poiché l'operatore relazionale `!=` ha prorità maggiore dell'operatore di assegnamento, devo parentesizzare l'espressione di assegnamento. Cosa accadrebbe se non ci fossero le parentesi ?
– `while( ... )` : il ciclo viene ripetuto fino a quando `getchar()` non restituisce `EOF`.

`if(c >= 'a' && c <= 'z')` : La condizione è vera sse:

il valore di `c` è maggiore o uguale al codice ASCII del carattere `'a'`

e

il valore di `c` è minore o uguale al codice ASCII del carattere `'z'`.

`&&` è l'operatore logico `AND`.

`putchar(c + 'A' - 'a');` :

`putchar` richiede un `int` come argomento.
Il carattere il cui codice ASCII è dato dal valore dell'argomento di `putchar` viene stampato sullo standard output.

Le costanti carattere, come `'A'` e `'a'`, possono comparire in contesti numerici, il loro valore è il loro codice ASCII.

I caratteri `a,b,...,z` hanno codici ASCII consecutivi: `'a' == 97`, `'b' == 98`, `....` Allo stesso modo: `'A' == 65`, `'B' == 66`, `....`
Quando `c` contiene il codice di una minuscola, l'espressione `c + 'A' -'a'` esprime il codice della maiuscola corrispondente.

Input/Output su `FILE`: `fgetc` e `fputc`

`int fgetc(FILE *fp)`
Legge un carattere dal FILE (lo *stream*) associato al puntatore `fp`.

`int fputc(int c,FILE *fp)`
Scrive il carattere `c` sul FILE puntato da `fp`.

Entrambe restituiscono il carattere letto o scritto, oppure `EOF` se la lettura o scrittura non è andata a buon fine.

Il loro prototipo è in `stdio.h`, che contiene anche due macro equivalenti a `fgetc` e `fputc`. Tali macro sono `getc` e `putc`.

Le macro sono spesso più efficienti delle funzioni, ma sono meno affidabili a causa di effetti collaterali negli argomenti (vedi pag.115).

Il programma che segue converte il file `testo.txt` in maiuscolo, ponendo il risultato nel file `maiusc.txt`.

```
/* toupperfile.c */
#include <stdio.h>

int main(void)
{
    int c;
    FILE *fr = fopen("testo.txt","r");
    FILE *fw = fopen("maiusc.txt","w");

    while((c = fgetc(fr)) != EOF)
        if(c >= 'a' && c <= 'z')
            fputc(c + 'A' - 'a',fw);
        else
            fputc(c,fw);
    return 0;
}
```

Lo standard input e lo standard output sono rispettivamente associati a puntatori a `FILE` rispettivamente denotati con `stdin` e `stdout`.

`getchar()` equivale a `fgetc(stdin)`.

`putchar(c)` equivale a `fputc(c,stdout)`.

Esiste un terzo *stream* associato per default a un processo: `stderr`, *standard error*, che per default è associato all'output su schermo.

Si esegua il programma seguente con:

`touppermsg < testo.txt > maiusc.txt`

```
/* touppermsg.c */
#include <stdio.h>

int main(void)
{
    int c;

    fprintf(stderr,"Sto computando...\n");
    while((c = fgetc(stdin)) != EOF)
        if(c >= ’a’ && c <= ’z’)
            fputc(c + ’A’ - ’a’,stdout);
        else
            fputc(c,stdout);
    fprintf(stderr,"Finito!\n");
    return 0;
}
```

Flusso del controllo, istruzioni

Operatori relazionali e di uguaglianza:

Operatori relazionali: `<`, `>`, `<=`, `>=`.

Operatori di uguaglianza: `==`, `!=`.

Arità: binaria.

Priorità: i relazionali hanno priorità maggiore degli operatori di uguaglianza. Entrambi i tipi hanno priorità inferiore agli operatori `+` e `-` binari, e maggiore degli operatori di assegnamento.

Associatività: da sinistra a destra.

Gli operatori relazionali e d'uguaglianza restituiscono i valori booleani *vero*,*falso*.

In C il valore booleano *falso* è rappresentato dal valore di tipo `int` `0`, e dal valore `double` `0.0`. Inoltre, ogni modo di esprimere `0` rappresenta il valore booleano *falso*. In particolare: il carattere `'\0'` e il puntatore nullo `NULL`.

Il valore booleano *vero* è rappresentato da qualunque valore diverso da `0`.

```
if(5) printf("Vero!\n");
```

Gli operatori relazionali e d'uguaglianza restituiscono `1` quando la condizione testata è vera.
Restituiscono `0` altrimenti.

Avvertenze:

Poiché la priorità degli operatori relazionali e d'uguaglianza è minore di quella degli operatori aritmetici, si può scrivere senza uso di parentesi:

```
if(7 - 1 > 1) printf("Vero!\n"); /* la condizione equivale a (7 - 1) > 1 */
```

Poiché invece tale priorità è maggiore di quella degli operatori di assegnamento, si devono usare le parentesi per eseguire l'assegnamento prima del confronto. Altrimenti:

```
int c = 4, d = 4;

if (c -= d >= 1) printf("Vero: %d\n",c); else printf("Falso: %d\n",c);
```

Non si confonda l'operatore di assegnamento = con l'operatore di uguaglianza ==.

```
int c = 0, d = 0;
if(c = d) printf ("c uguale a d?":%d,%d\n",c,d);
    else printf("c diverso da d?:%d,%d\n",c,d);
```

Operatori logici `&&`, `||`, `!`

`&&`: `AND`. Binario, associa da sinistra a destra.
`c1 && c2` restituisce 1 sse `c1 != 0` e `c2 != 0`.
Restituisce 0 altrimenti.

`||`: `OR`. Binario, associa da sinistra a destra.
`c1 || c2` restituisce 1 se almeno uno tra `c1` e `c2` è diverso da 0.
Restituisce 0 altrimenti.

`!`: `NOT`. Unario, associa da destra a sinistra.
`!c` restituisce 1 se `c == 0`. Restituisce 0 altrimenti.

La priorità degli operatori logici è maggiore della priorità degli operatori di assegnamento. La priorità di `!` è la stessa degli altri operatori unari, la priorità di `||` è inferiore a quella di `&&` che è inferiore a quella degli operatori di uguaglianza.

Avvertenze:

Si voglia testare che il valore di `int j` cada nell'intervallo $[3, 5]$.

```
int j = 6;

if(3 <= j <= 5) printf ("SI'\n"); else printf("NO\n");      /* stampa SI' */

if(3 <= j && j <= 5) printf ("SI'\n"); else printf("NO\n"); /* stampa NO  */
```

Si noti anche:
Il valore di `!!5` è il valore di `!0`, vale a dire `1`.

Valutazione Cortocircuitata

`c1 && c2` :
Se `c1 == 0` non c'è bisogno di valutare `c2` per stabilire che
`c1 && c2 == 0`.

Analogamente:

`c1 || c2` :
Se `c1 != 0` non c'è bisogno di valutare `c2` per stabilire che
`c1 || c2 == 1`.

In C il processo evita di calcolare il valore di `c2` in questi casi.

```
int c;

while((c = getchar()) != EOF && dosomething(c)) ...
```

```
/* divisioni.c */
#include <stdio.h>

int main(void)
{
    int v[10] = { 2, 5, 12, 7, 29, 0, 3, 4, 6, 7 };

    int i, a = 300000;

    printf("%d",a);
    for(i = 0; i < 10; i++)
        if(v[i] && (a /= v[i]) >= 1)
            printf("/%d=%d",v[i],a);
    putchar('\n');
}
```

`if(v[i] && (a /= v[i]) >= 1)` :

Grazie alla valutazione cortocircuitata non viene eseguita l'espressione `a /= v[i]` quando `v[i] == 0`.

Istruzioni

Istruzione composta, o **blocco**: Sequenza di istruzioni racchiuse fra graffe. Le dichiarazioni possono apparire solo prima di ogni altro tipo di istruzione in un blocco.

$compound_statement ::= \{ \quad \{declaration\}_{0+} \quad \{statement\}_{0+} \quad \}$

Se in un costrutto C, come `while`,`for`,`if`, vogliamo porre più istruzioni, dobbiamo raccoglierle in un'istruzione composta.

Istruzione vuota: E' costituita dal solo simbolo `;`.

Utile quando a livello sintattico è necessaria un'istruzione, ma a livello semantico non vi è nulla da fare:

```
for(i = 0;t[i] = s[i];i++)
    ;
```

Istruzioni di selezione: `if`, `if-else`

```
if  (expr)
    statement
```

L'istruzione *statement* viene eseguita solo se il valore di *expr* è diverso da `0`.

```
if    (expr)
      statement1
else
      statement2
```

Se *expr* `!= 0` allora viene eseguita l'istruzione *statement1*, altrimenti viene eseguita l'istruzione *statement2*.

Avvertenze:

Si ricordi: in C ogni *espressione* costituisce una *condizione*: se tale espressione vale `0`, allora come condizione risulta *falsa*, altrimenti risulta *vera*.

Dangling Else:

```
if(a == 1) if(b == 2) printf("%d",b); else printf("%d",a);
```

equivale a

```
if(a == 1) {
    if(b == 2)
        printf("%d",b);
    else
        printf("%d",a);
}
```

il ramo `else` viene associato all'`if` più interno (il più vicino).

Per modificare la risoluzione standard del *Dangling Else* si utilizzano opportunamente le istruzioni composte:

```
if(a == 1)
{
    if(b == 2)
        printf("%d",b);
}
else
    printf("%d",a);
```

Serie di selezioni:

`if` (*expr*)
statement1
`else if` (*expr*)
statement2
...
`else if` (*expr*)
*statement*k

Istruzioni per realizzare iterazioni: **`while`**

`while` (*expr*)
statement

Si valuta *expr*: se il valore è diverso da `0` si esegue **`statement`**, dopodichè si ritorna all'inizio del ciclo: si valuta *expr* di nuovo.

Quando *expr* `== 0` il flusso abbandona il ciclo **`while`** e prosegue con l'istruzione successiva al ciclo stesso.

A volte non c'è bisogno di *statement*, poichè tutto il lavoro è svolto nella valutazione della condizione di controllo *expr*. In questi casi si usa l'istruzione vuota.

```
while((c = getchar()) != EOF && putchar(toupper(c)))
        ;
```

Esercizio sui costrutti iterativi:

Calcolo del fattoriale

Sia n un intero positivo:

$$0! = 1, \qquad n! = n((n-1)!) = n(n-1)(n-2)\cdots 2.$$

La funzione fattoriale cresce molto velocemente:

$$13! = 6227020800$$

AVVERTENZA:

-se scegliamo di rappresentare i valori del fattoriale in un `long` su implementazioni in cui `sizeof(long) == 4`, potremo calcolare correttamente il fattoriale di interi piuttosto piccoli (da 0! a 12!).

-se scegliamo di rappresentare i valori in un `double`, avremo solo valori approssimati del fattoriale.

-dovremmo implementare una sorta di tipo intero "esteso" per rappresentare i grandi valori interi della funzione fattoriale: vedremo in un esercizio come si può affrontare questo problema.

-per adesso scegliamo un'implementazione con `double`.

```c
#include <stdio.h>

int main(void)
{
    int n, i = 1;
    double factorial = 1.0;

    while(1) {

        printf("Immetti n:\n");
        if(!scanf("%d",&n)) {
            fprintf(stderr,"Errore in lettura!\n");
            return -1;
        }

        while(++i <= n)
            factorial *= i;

        printf("%d! = %.0f\n",n,factorial);
        i = 1; factorial = 1.0;
    }
}
```

Commenti su `factorial.c`

`while(1)` : L'espressione `1` è costantemente diversa da `0`: il ciclo viene ripetuto per sempre.
Questo è uno dei due modi standard del C per realizzare cicli infiniti.
Per uscire da un siffatto ciclo, o si interrompe il processo dall'esterno, oppure si usano delle istruzioni di uscita (da cicli o da funzioni) all'interno del ciclo.

`if(!scanf("%d",&n)) {`: Si usa il valore restituito da `scanf`, cioè il numero di argomenti della stringa formato letti con successo, per controllare che la lettura dell'intero `n` vada a buon fine.

```
fprintf(stderr,"Errore in lettura!\n");
return -1;
```

In caso `scanf` non abbia concluso con successo la lettura dell'intero, si stampa un messaggio d'errore su `stderr`, e poi si esce dalla funzione (e a fortiori) dal ciclo, con l'istruzione `return -1`.
Poiché la funzione in questione è `main` il processo termina e il valore `-1` è passato al sistema operativo.

```
while(++i <= n)
    factorial *= i;
```

Ciclo `while` che costruisce il valore del fattoriale.

Istruzioni per realizzare iterazioni: `do-while`

```
do
        statement
while  (expr);
```

Il costrutto `do-while` costituisce una variazione del costrutto `while`: *statement* viene eseguito prima di valutare la condizione di controllo *expr*.

Si noti che il `;` che segue il `while` indica la terminazione dell'istruzione `do-while`. Non è un'istruzione vuota.

```
do {
    printf("Immetti n:\n");
    if((ns = scanf("%d",&n)) != 1)  /* assumiamo di aver dichiarato char ns; */
        fprintf(stderr,"Errore in lettura! Riprova:\n");
    while(getchar() != '\n');  /* per svuotare lo stream di input */
} while(ns != 1);
```

In alcuni linguaggi di programmazione (e spesso nello pseudocodice usato per presentare gli algoritmi) è presente il costrutto `repeat-until`:

```
repeat
        statement
until   (expr);
```

Tale costrutto è simile al `do-while` del C.
La differenza:
– Nel costrutto `do-while` si opera una nuova iterazione se e solo se *expr* è vera.
– Nel costrutto `repeat-until` (ripeti-fino a quando) si opera una nuova iterazione se e solo se *expr* è falsa.

Istruzioni per realizzare iterazioni: `for`

```
for  (expr1;expr2;expr3)
     statement
```

Equivale a:

```
expr1;
while   (expr2) {
        statement
        expr3;
}
```

(solo quando *expr2* è presente e non vi è alcuna istruzione `continue` all'interno del ciclo (vedi p.159)).

Ognuna fra *expr1*,*expr2*,*expr3* può essere omessa
(ma non i caratteri ; separatori).

Nel caso si ometta *expr2*, il test corrispondente risulta sempre vero:

`for(`*expr1*`;;`*expr3*`)` *statement*   equivale a:

```
expr1;
while   (1) {
        statement
        expr3;
}
```

Il secondo modo standard per esprimere in C un ciclo infinito è:

```
for(;;)
```

```
#include <stdio.h>

int main(void)
{
    int n, i;
    double factorial;
    char ns;

    for(;;) {
        for(ns = 0;ns != 1;) {
            printf("Immetti n:\n");
            if((ns = scanf("%d",&n)) != 1)
                fprintf(stderr,"Errore in lettura! Riprova:\n");
            while(getchar() != '\n');
        }

        for(i = factorial = 1;i <= n;i++)
            factorial *= i;

        printf("n! = %.0f\n",factorial);
    }
}
```

Operatore virgola: ,

L'operatore virgola è un operatore binario.
Ha la priorità più bassa fra tutti gli operatori del C.
Associa da sinistra a destra.
I suoi due operandi sono espressioni generiche.

Sintassi e semantica:
expr1 , *expr2*

Vengono valutate prima *expr1* e poi *expr2*: il valore e il tipo dell'intera espressione virgola sono quelli di *expr2*.

L'operatore virgola si usa frequentemente nei cicli `for`:

```
for(i = 1, factorial = 1.0; i <= n; factorial *= i, i++);
```

Un'istruzione da non usare mai: `goto`

Il C contempla la famigerata istruzione `goto`.

Sintassi: Un'istruzione `goto` ha la forma: `goto` *label*.
label è un identificatore di etichetta.
Un'istruzione etichettata ha la forma: *label*: *statement*.

Semantica: L'istruzione

```
goto l0
```

trasferisce il controllo all'istruzione etichettata

`l0 :` *statement*.

L'istruzione `goto` e l'istruzione etichettata corrispondente devono comparire all'interno della stessa funzione.

```
int f() {
    ...
    {
        ...
        goto error;
        ...
    }
    ...
    error: printf("Errore!\n"); exit(-1);
    ...
}
```

Le etichette hanno uno proprio *spazio dei nomi* (Identificatori di variabili e di etichette possono coincidere, senza causare problemi al compilatore). Le etichette sono soggette alle stesse regole di visibilità degli altri identificatori.

Non si dovrebbe mai usare l'istruzione `goto`. Il suo uso scardina l'impianto strutturato del programma, alterando rozzamente il flusso del controllo.

L'istruzione `break`

Sintassi: `break;`

`break` deve comparire all'interno di un ciclo `while`, `do-while`, `for` o all'interno di un'istruzione `switch`.

L'istruzione `break` causa l'uscita dal più interno fra i costrutti di ciclo o `switch` che la contiene. Il controllo passa all'istruzione che segue immediatamente la fine di tale costrutto.

Modifichiamo `divisioni.c` in modo che il programma termini quando si tenta di dividere `a` per 0.

```
for(i = 0; i < 10; i++) {
    if(!v[i])
        break;
    printf("/%d=%d",v[i],a /= v[i]);
}
```

```
/* reverse.c */
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s[80];
    int i;

    while(1) {
        printf("Immetti stringa, oppure \"FINE\"\n");
        scanf("%s",s);
        if(strcmp(s,"FINE") == 0)
            break;
        for(i = strlen(s);i;i--)
            putchar(s[i-1]);
        putchar('\n');
    }
}
```

`#include <string.h>` : File d'intestazione contenente i prototipi delle funzioni per la manipolazione di stringhe, come:

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

`if(strcmp(s,"FINE") == 0)` : la funzione `strcmp` restituisce un `int` minore di `0` se il suo primo argomento stringa precede il secondo nell'ordine lessicografico, `0` se i due argomenti stringa sono uguali, un `int` maggiore di `0` se il secondo precede il primo.

`break;` : questa istruzione causa l'uscita dal ciclo più interno che la contiene. Poiché nessuna istruzione segue la fine del ciclo `while`, la funzione `main`, e quindi il programma, terminano.

`i = strlen(s)` : la funzione `strlen` restituisce la lunghezza in caratteri della stringa `s` (il carattere nullo di terminazione non viene contato).

L'istruzione `continue`

Sintassi: `continue;`
`continue` deve comparire all'interno di un ciclo `while`, `do-while`, `for`.

L'istruzione `continue` causa l'interruzione dell'iterazione corrente nel ciclo più interno che la contiene e causa l'inizio dell'iterazione successiva dello stesso ciclo.

Avvertenza:

```
for   (expr1;expr2;expr3) {
      ...
      continue;
      ...
}
```

equivale a:

```
expr1;
while    (expr2) {
         ...
         goto next;
         ...
next:
         expr3;
}
```

Modifichiamo `divisioni.c` in modo che il programma salti le divisioni di `a` per 0.

```
for(i = 0; i < 10; i++) {
    if(!v[i])
        continue;
    printf("/%d=%d",v[i],a /= v[i]);
}
```

Si noti la differenza con:

```
i = 0;
while(i < 10) {
    if(!v[i])
        continue;
    printf("/%d=%d",v[i],a /= v[i]);
    i++;
}
```

L'istruzione `switch`

Permette di realizzare una selezione a scelta multipla.

Sintassi:

```
switch  (integer_expr) {
        case_group_1
        case_group_2
        ...
        case_group_k
}
```

Dove *integer_expr* è una espressione di tipo `int`
mentre ogni *case_group*$_i$ è della forma:

```
case const_expr_i1 :   case const_expr_i2 :   ...   case const_expr_ih :
          statement
          statement
          ...
          statement
```

Ogni espressione *const_expr*$_{ij}$ deve essere una costante intera.

Nell'intera istruzione **`switch`** una costante intera non può apparire più di una volta come etichetta di **`case`**.

Inoltre, al massimo una volta in un'istruzione **`switch`** può apparire un gruppo **`default`**, solitamente come ultimo *caso* dell'istruzione **`switch`**:

```
switch  (integer_expr) {
        case_group1
        ...
        default:
            statement
            ...
            statement
        ...
        case_groupk
}
```

Solitamente, l'ultimo *statement* in un gruppo `case` o in un gruppo `default` è un'istruzione `break` (o `return`, se si vuole uscire non solo dallo `switch` ma dalla funzione che lo contiene).

Semantica:

-Viene in primo luogo valutata l'espressione *integer_expr*.

-Il controllo passa quindi alla prima istruzione *statement* successiva all'etichetta `case` la cui espressione *const_expr*$_{ij}$ ha valore coincidente con quello di *integer_expr*.

-Se tale etichetta non esiste, il controllo passa alla prima istruzione del gruppo `default`.

-Se il gruppo `default` non è presente, l'istruzione `switch` termina e il controllo passa all'istruzione successiva.

- - Una volta che il controllo è entrato in un gruppo `case` o `default`, l'esecuzione procede sequenzialmente, fino a incontrare un'istruzione `break` che determina l'uscita dall'istruzione `switch` (oppure un'istruzione `return` che determina l'uscita dalla funzione).

Avvertenza:

Se la successione di istruzioni in un gruppo `case` o `default` non termina con un'istruzione `break` (o `return`, o con qualche altro modo di uscire dalla funzione (es. funzione `exit()`)), il controllo, dopo l'ultima istruzione del gruppo, passerà alla prima istruzione del gruppo successivo.

`continue` non può apparire in un'istruzione `switch` (a meno che non appaia in un ciclo interno a `switch`).

```
#include <stdio.h>
int main(void) {
    int c, i, nwhite = 0, nother = 0, ndigit[10];
    for(i = 0; i < 10; i++) ndigit[i] = 0;
    while((c = getchar()) != EOF) {
        switch(c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c - '0']++;
            break;
        case ' ': case '\n': case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("Cifre.");
    for(i = 0; i < 10;i++) printf(" %d:%d,",i,ndigit[i]);
    printf(" spazi: %d, altri: %d\n",nwhite,nother);
}
```

`switch(c) {...}`: L'istruzione `switch` è applicata al valore `int` restituito da `getchar()`.

`case '0': ... case '9':`: Le etichette di `case` sono espressioni costanti `int` (Le costanti carattere sono di tipo `int`). Se il carattere letto è una cifra, allora il controllo passa all'istruzione che segue `case '9':`.

`ndigit[c - '0']++;`: viene selezionato e quindi incrementato l'elemento di `ndigit` corrispondente alla cifra letta.

`break;`: senza questa istruzione il controllo passerebbe all'istruzione `nwhite++` che segue `case '\t':`.

`default:`: se nessuna costante `case` coincide col valore di `c`, il controllo passa all'istruzione successiva a `default`.
Nota: l'ultimo `break` è superfluo.

L'operatore condizionale ternario `?:`

expr1 `?` *expr2* `:` *expr3*

expr1 viene valutata.
Se il valore è diverso da `0` allora viene valutata *expr2* e il suo valore è il valore dell'intera espressione.
Se il valore di *expr1* è `0`, allora viene valutata *expr3* e il suo valore è il valore dell'intera espressione.

Il tipo dell'intera espressione è il tipo adeguato sia a *expr2* che *expr3* determinato attraverso l'algoritmo per le conversioni standard.

La priorità dell'operatore ternario è immediatamente maggiore di quella degli operatori di assegnamento.
Associa da destra a sinistra.

Funzioni

Funzioni

Un programma C tipicamente contiene numerose funzioni *brevi*.

Le definizioni di funzioni possono essere distribuite su più file.

Le definizioni di funzioni non possono essere innestate.
Tutte le definizioni di funzioni appaiono al livello più esterno.

In un programma C deve essere definita una funzione `main`, dalla quale parte l'esecuzione del programma, e che richiama altre funzioni definite nel programma o appartenenti a qualche libreria.

Definizione di funzione

type *function_name* (*parameter_list*)
{
declaration_list

statement
...
statement
}

type è il tipo di ritorno della funzione.

La funzione dovrà ritornare valori appartenenti a *type*:

```
int f(...) {...}
```

Se la funzione non deve ritornare alcun valore, si specifica `void` come *type*.

```
void f(...) {...}
```

function_name è un identificatore. La funzione sarà usualmente invocata attraverso il suo nome. Il nome è anche un puntatore alla funzione stessa.

parameter_list è un elenco di dichiarazioni (senza inizializzazione) separate da virgole.

```
int f(char *s, short n, float x) {...}
```

In questo modo si specificano il numero, la posizione e il tipo dei parametri formali della funzione.

Nell'invocazione della funzione si deve fornire una lista di argomenti, corrispondenti nel numero e ordinatamente nei tipi alla lista dei parametri formali.

```
i = f("pippo", 4, 3.14f);
```

Per specificare che una funzione va invocata senza alcun argomento, nella dichiarazione si deve specificare `void` al posto della lista dei parametri.

```
int f(void) {...}
```

Nell'invocazione di siffatta funzione, non può essere tralasciata la coppia di parentesi `f()`.

```
i = f();
```

Il *corpo* di una definizione di funzione è un'istruzione composta.

Nel corpo, i parametri formali sono usati come variabili locali.

Nel corpo, i parametri formali sono usati come variabili locali.

```
/* restituisce m elevato alla n-esima potenza */
long power(int m, int n)
{
    int i;
    long product = 1L;

    for(i = 1; i <= n; i++)
        product *= m;
    return product;
}

/* funzione vuota */
void niente(void) {}
```

Nell'invocazione, i *valori* degli argomenti vengono utilizzati localmente al posto dei parametri corrispondenti.

```
long p = power(5,2); /* p == 25L */
/* in power, m == 5 e n == 2 */
```

Istruzione `return`

Sintassi: **return;** **return** *expr*;

Quando viene raggiunta un'istruzione **return** il controllo ritorna alla funzione chiamante.

Se l'istruzione è della forma **return** *expr*; il valore di *expr* viene restituito alla funzione chiamante come valore dell'espressione costituita dall'invocazione della funzione stessa.

```
double x = power(5,2) / 3.0;
/* il valore dell'espressione power(5,2) e' 25L */
```

Se necessario il tipo di *expr* viene convertito al tipo della funzione.

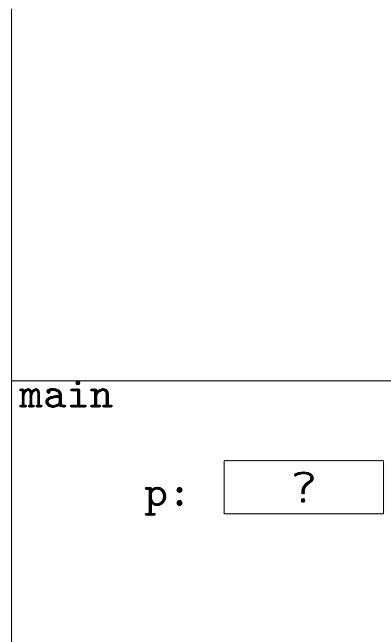
Se non viene raggiunta alcuna istruzione `return`, il controllo viene restituito alla funzione chiamante quando si è raggiunta la fine del blocco che definisce il corpo della funzione.

Se una funzione con tipo di ritorno diverso da `void` termina senza incontrare un'istruzione `return`, il valore di ritorno della funzione è indeterminato.

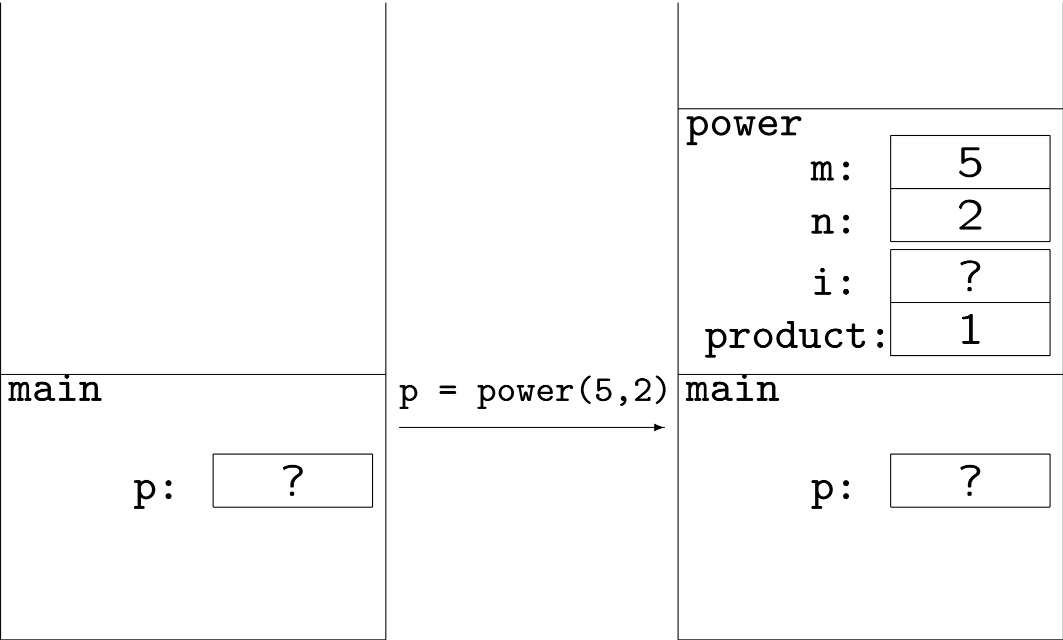
Il valore restituito da una funzione può anche non essere utilizzato:

```
/* restituisce il numero di scambi effettuati) */
int bubble(int a[], int n, FILE *fw) { ... }

bubble(a,7,stdout); /* ordina i primi 7 elementi di a */
                    /* non usa il valore di ritorno di bubble */
```

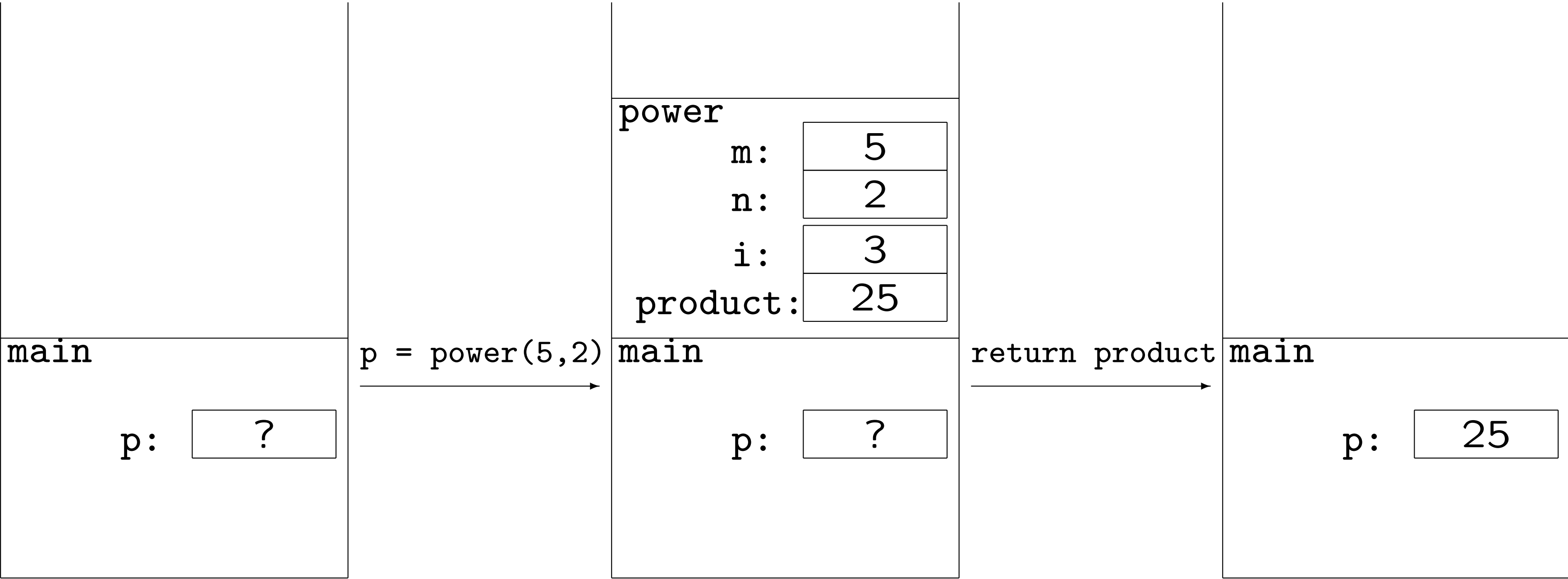
Invocazione di funzioni e stack dei record di attivazione:



Invocazione di funzioni e stack dei record di attivazione:



Invocazione di funzioni e stack dei record di attivazione:



ESERCIZIO:
Nel programma `factorial.c` avevamo erroneamente usato il tipo `double` per contenere il fattoriale di un numero dato. Ma `double` ha una precisione limitata (assumiamo 15 cifre decimali). `factorial.c` fornisce solo un'approssimazione del fattoriale.

D'altro canto se avessimo usato tipi interi, come `long`, avremmo potuto calcolare correttamente solo i fattoriali di numeri piuttosto piccoli (per `long` di 4 byte il massimo fattoriale rappresentabile è `12!`.

Per affrontare la situazione, decidiamo di rappresentare i numeri interi nonnegativi come array di caratteri.

Implementiamo la moltiplicazione fra siffatti interi con l'algoritmo per la moltiplicazione "a mano".
Usiamo poi quanto fatto per il calcolo del fattoriale.

Un "intero" sarà rappresentato come un array di DIGITS char, dove DIGITS è una macro (assumiamo DIGITS = 500).

Il numero 125 sarà rappresentato come
    { 5, 2, 1, -1, -1, ..., -1 }
(esercizio: `mult.c`: migliorare rappresentazione e implementazione: compattare: pensare in base 256)

```
/* frammento di mult.c */
#define DIGITS 500

typedef char INT[DIGITS]; /* interi rappresentati come array di DIGITS char */

/* moltiplica n1 * n2, mette il risultato in n3 */
void mult(INT n1, INT n2, INT n3)
{
    int i,j,r,l1,l2;

    /* "ripulisce" n3 */
    for(i = 0; i < DIGITS; i++)
        n3[i] = -1;
```

```
/* ciclo sul primo numero */
for(i = 0; i < DIGITS; i++) {
    /* se abbiamo raggiunto la cifra piu' grande
       del primo numero abbiamo finito */
    if((l1 = n1[i]) == -1)
        break;

    /* ciclo sul secondo numero */
    /* r e' il riporto */
    for(j = r = 0; j < DIGITS; j++) {
        /* dobbiamo aggiornare la (i+j)esima cifra del prodotto */

        /* abbiamo raggiunto la cifra piu' grande del secondo numero:
           sistemiamo il riporto e usciamo dal ciclo interno */
        if((l2 = n2[j]) == -1) {
            n3[i+j] = r;
            break;
        }
```

```
            /* La (i+j)esima cifra e' raggiunta per la prima volta ? */
            if(j >= i)
                /* se n3[i+j] == -1 dobbiamo porlo a 0 */
                n3[i+j] = max(n3[i+j], 0);

            /* aggiorniamo, tenendo conto del riporto */
            n3[i+j] += l1 * l2 + r;
            /* calcoliamo nuovo riporto */
            r = n3[i+j] / 10;
            /* se n3[i+j] e' piu' grande di 9: */
            n3[i+j] %= 10;
        }
    }

    /* elimina eventuali 0 che appaiano in testa */
    for(i = DIGITS - 1; i; i--)
        if(n3[i] != -1) {
            if(n3[i] == 0)
                n3[i] = -1;
            break;
        }
}
```

Prototipi di funzione

Il prototipo di una funzione costituisce una *dichiarazione* della funzione, e come tale fornisce al compilatore le informazioni necessarie a gestire la funzione stessa.

Nella *definizione* di una funzione, viene specificato anche ciò che la funzione deve fare quando viene invocata (questa informazione è data dal corpo della funzione).
Nella *dichiarazione* questa informazione non serve, infatti il prototipo di una funzione coincide con la riga di intestazione della funzione stessa, a meno dei nomi dei parametri formali, che possono anche essere omessi.

```
int f(char *s, short n, float x) { ... }  /* definizione di f */

int f(char *, short, float);              /* prototipo di f */
```

Il prototipo o la definizione dovrebbero sempre precedere ogni utilizzo della funzione.

Avvertenze:

Nelle definizioni di funzioni (e quindi nei prototipi) vi sono alcune limitazioni concernenti i tipi di ritorno e la lista dei parametri:

- Il tipo di ritorno non può essere un array o una funzione, ma può essere un puntatore ad array o a funzione.

- Classi di memorizzazione: Una funzione non può essere dichiarata `auto` o `register`. Può essere dichiarata `extern` o `static`.

- I parametri formali non possono essere inizializzati, inoltre non possono essere dichiarati `auto`, `extern` o `static`.

Visibilità

Gli identificatori sono accessibili solo all'interno del blocco nel quale sono dichiarati.

L'oggetto riferito da un identificatore è accessibile da un blocco innestato in quello contenente la dichiarazione, a meno che il blocco innestato non dichiari un altro oggetto riutilizzando lo stesso identificatore.
In tal caso, la dichiarazione nel blocco innestato *nasconde* la dichiarazione precedente.

Da un dato blocco non sono accessibili gli identificatori dichiarati in un blocco parallelo.

```
/* siamo nel blocco piu' esterno dove dichiariamo le funzioni e le variabili globali */
double b = 3.14;

void f(void)
{
    int a = 2;
    {
        double a = 0.0, c = 3.0;
        /* da qui fino alla fine del blocco int a e' nascosta da double a */
        printf("a piu' interno: %f\n",a - 2.0);

        /* double b  e' visibile in questo blocco innestato
        in un blocco innestato in quello dove b e' stata dichiarata*/
        printf("b globale: %f\n",b - 2.0);
    }
    /* all'uscita dal blocco piu' interno, double a non e' piu' visibile
        e int a torna visibile */
    {
        /* int a e' ancora visibile poiche' non e' nascosta */
        printf("a meno interno: %f\n",a - 2.0);

        /* c qui non e' visibile poiche' dichiarata in un blocco parallelo */
        printf("c: %f\n",c - 2.0); /* ERRORE! */
    }
}
```

Le variabili dichiarate in un blocco sono risorse locali al blocco stesso, sia in termini di visibilità, sia in termini di memoria occupata. Per default, la memoria allocata per una variabile dichiarata in un blocco viene rilasciata quando il controllo abbandona il blocco.
Un blocco costituisce un ambiente di visibilità e di allocazione di memoria.

Le funzioni vengono definite nel blocco più esterno. Hanno visibilità globale. Se la funzione `f()` viene invocata in un file prima che sia stata dichiarata tramite il suo prototipo o la sua definizione, il compilatore assume implicitamente che `f` ritorni `int`; nessuna assunzione viene fatta sulla lista dei parametri.

Le variabili definite nel blocco più esterno hanno visibilità globale: sono visibili da tutte le funzioni dichiarate nel prosieguo del file.

Classi di memorizzazione

In C vi sono quattro *classi di memorizzazione* per funzioni e variabili:

`auto`, `extern`, `register`, `static`

Specificano le modalità con cui viene allocata la memoria necessaria e il ciclo di vita delle variabili.

`auto` : La parola chiave `auto` in pratica non si usa mai.
Per default sono `auto` le variabili dichiarate all'interno dei blocchi. La memoria necessaria a una variabile `auto` viene allocata all'ingresso del blocco e rilasciata all'uscita, perdendo il valore corrente della variabile. Al rientro nel blocco, nuova memoria viene allocata, ma ovviamente l'ultimo valore della variabile non può essere recuperato.
Il valore iniziale, se non specificato, è da considerarsi casuale.

`extern`: Le funzioni sono per default di classe `"esterna"`, così come le variabili globali.
Le variabili di classe `"esterna"` sopravvivono per tutta la durata dell'esecuzione.
Vengono inizializzate a `0` in mancanza di inizializzazione esplicita.

La parola chiave `extern` si usa per comunicare al compilatore che la variabile è definita altrove: nello stesso file o in un altro file.

Una **dichiarazione** `extern` è una **definizione** quando è presente un'inizializzazione (e in questo caso la parola chiave `extern` è sempre superflua.),
oppure una **dichiarazione** (che la variabile è **definita altrove**).
Possono essere contemporaneamente visibili più dichiarazioni `extern` della stessa variabile.

```
/* a.c */
int a = 1; /* def. di variabile globale, per default e' extern */

extern int geta(void); /* qui la parola extern e' superflua */

int main(void)
{
    printf("%d\n",geta());

}

/* b.c */
extern int a; /* cerca la variabile a altrove */
extern int a; /* ripetuta: no problem */

int geta(void)
{
    extern int a; /* questa dichiarazione e' ripetuta e superflua, non erronea */

    return ++a;
}
```

`register` : usando la parola chiave `register` per dichiarare una variabile, si *suggerisce* al compilatore di allocare la memoria relativa alla variabile nei registri della macchina.

Solitamente si usa `register` per variabili di tipo *piccolo* frequentemente accedute, come le variabili dei cicli.

```
register int i;

for(i = 0; i < LIMIT; i++) { ... }
```

I parametri delle funzioni possono essere dichiarati `register`.

Si noti che il compilatore potrebbe non seguire il *consiglio* di allocare la variabile nei registri.

I compilatori ottimizzanti rendono pressochè inutile l'uso di `register`.

Semanticamente, `register` coincide con `auto`.

`static` : La classe di memorizzazione `static` permette di creare variabili con ciclo di vita esteso a tutta la durata dell'esecuzione, ma con visibilità limitata.

Dichiarando `static` una variabile locale a un blocco, la variabile continua a esistere anche quando il controllo non è nel blocco. Tale variabile continua a non essere accessibile dall'esterno del blocco. Al ritorno del controllo nel blocco, la variabile, che ha mantenuto il proprio valore, sarà di nuovo accessibile.

```
int getst(void)
{
    static int st = 1;

    return ++st;
}

int main(void) { printf("%d,",getst());printf("%d\n",getst()); }
```

`static "esterne"`: Dichiarando **`static`** funzioni e variabili globali le si rende *private* alla porzione di file che segue la loro dichiarazione.
Anche dichiarandole altrove **`extern`** per indicarne l'esistenza in qualche punto del programma, tali funzioni e variabili rimarrano accessibili solo alle funzioni definite nella suddetta porzione di file.

Questa caratteristica permette di sviluppare componenti *modulari* costituite da singoli file contenenti gruppi di funzioni che condividono risorse non accessibili da nessun'altra funzione.

Es.:Uso di `static` `"esterne"`: Una prima implementazione di *stack*:

```
/* stk.c */
#define SIZE    100

static int pos = 0;
static char stack[SIZE];

void push(char e) {
    if(pos  < SIZE)
        stack[pos++] = e;
    else
        manageerror("Stack pieno",1);
}

char pop(void) {
    if(pos)
        return stack[--pos];
    else {
        manageerror("Stack vuoto",2);
        return 0;
    }
}
```

Commenti:

```
static int pos = 0;
static char stack[SIZE];
```

Definendo `static` queste due variabili `extern` si limita la loro visibilità al solo prosieguo del file.
In questo caso le variabili `pos` e `stack` sono visibili solo dalle funzioni `push` e `pop`.
In questo modo il file `stk.c` costituisce un *modulo*.

In questo modulo è fornita una prima semplicistica implementazione del tipo di dati astratto *stack*.

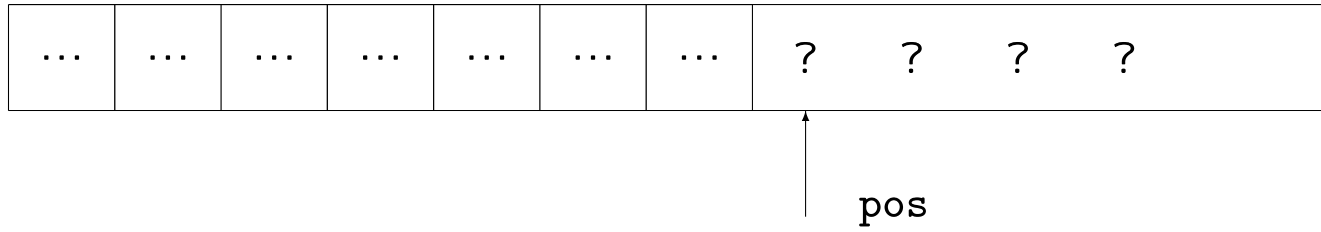
L'array `stack` di `char` conterrà gli elementi inseriti. La variabile `int pos` conterrà sempre l'indice della prima posizione *vuota* di `stack`, vale a dire, la posizione successiva alla *cima* dello stack.

```
void push(char e) {
    if(pos  < SIZE)
        stack[pos++] = e;
...
```

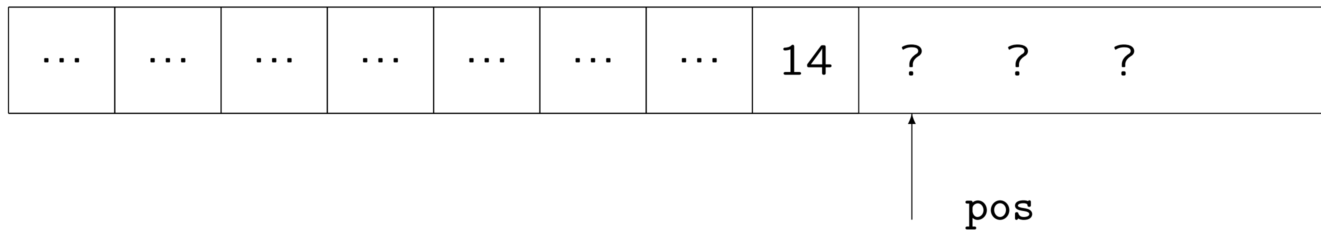
push pone l'elemento e in cima allo stack. Tale operazione è consentita solo se stack ha ancora posizioni vuote, solo se il valore di pos è < SIZE.

```
char pop(void) {
    if(pos)
        return stack[--pos];
...
```

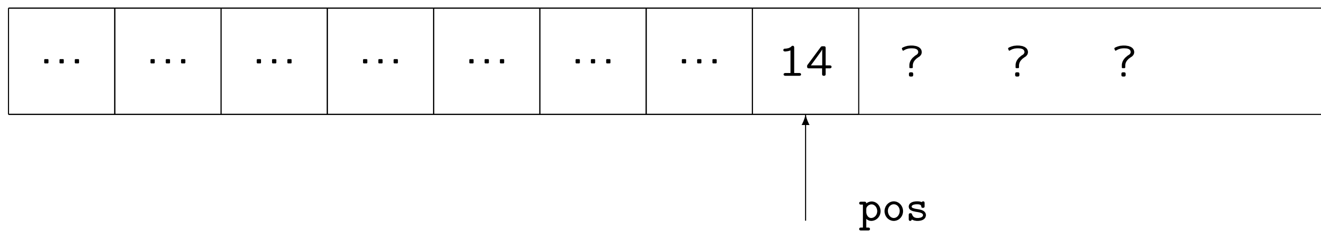
pop restituisce il valore contenuto sulla cima dello stack, vale a dire la posizione precedente alla prima posizione libera – quella contenuta in pos. L'elemento viene cancellato dallo stack, dunque pos viene decrementato per contenere la nuova posizione libera sulla cima dello stack.



push(14)



x = pop()



Esempio: controllo della parentesizzazione.

```
int main(void)
{
    int c;
    char d;

    printf("Immetti stringa:\n");
    while((c = getchar()) != EOF && c != ';') {
        if(!parentesi(c))
            continue;
        if(!empty())
            if(chiude(c,d = pop()))
                continue;
            else {
                push(d);
                push(c);
            }
        else
            push(c);
    }
    printf("%s\n", empty() ? "Corretto" : "Sbagliato");
}
```

Commenti:

```
while((c = getchar()) != EOF && c != ';') {
```

Stabiliamo che ';' marchi la fine dell'espressione parentesizzata.

```
if(!parentesi(c))
    continue;
```

Se `c` non è una parentesi, tralascia il resto del ciclo, e comincia una nuova iterazione.

```
if(!empty())
```

La funzione

```
char empty(void) { return !pos; }
```

è definita in `stk.c`.
Appartiene al modulo che implementa lo stack.
Restituisce 1 se lo stack è vuoto, 0 altrimenti.

• Se lo stack non è vuoto:

```
if(chiude(c,d = pop()))
    continue;
else {
    push(d);
    push(c);
}
```

`d = pop()` : viene prelevata la cima dello stack e assegnata a `d`.
`chiude` controlla che `d` sia una parentesi aperta di cui `c` sia la versione chiusa.
Se così è: `c` e `d` si elidono a vicenda, e si prosegue con la prossima iterazione (`continue;`).
Altrimenti: `d` è rimesso sullo stack, seguito da `c` (`push(d); push(c);`).

Facile esercizio: Riscrivere eliminando l'istruzione `continue`.

• Se lo stack è vuoto:

```
push(c);
```

si inserisce `c` sulla cima dello stack.

All'uscita dal ciclo:

```
printf("%s\n", empty() ? "Corretto" : "Sbagliato");
```

Se lo stack è vuoto, allora tutte le parentesi sono state correttamente abbinate.
Altrimenti la parentesizzazione è errata.

```
/* controlla che c sia una parentesi */
char parentesi(char c)
{
    return     c == '(' || c == ')'
            || c == '[' || c == ']'
            || c == '{' || c == '}';
}
/* controlla che la parentesi c chiuda la parentesi d */
char chiude(char c, char d)
{
    switch(d) {
    case '(':
        return c == ')';
    case '[':
        return c == ']';
    case '{':
        return c == '}';
    default:
        return 0;
    }
}
```

Ricorsione

Una funzione è detta *ricorsiva* se, direttamente o indirettamente, richiama se stessa.

```
void forever(void)
{
    printf("It takes forever\n");
    forever();
}
```

Per evitare che una funzione ricorsiva cada in una ricorsione infinita bisogna prevedere delle *condizioni di terminazione*:

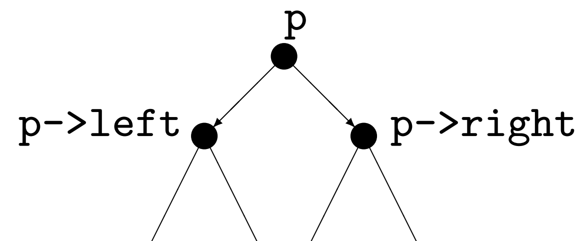
```
void forfewtimes(int i)
{
    static int k = 0;

    if(!k) k = i;
    if(i) {
        printf("Just %d times: %d\n",k,k-i+1);
        forfewtimes(i-1);
    }
}
```

Come vedremo la ricorsione è un concetto molto importante nello studio degli algoritmi e delle strutture dati.

Spesso il modo più naturale per realizzare algoritmi prevede l'utilizzo della ricorsione. Anche la maggior parte delle strutture dati sofisticate sono naturalmente ricorsive.

Esempio: visita ricorsiva *in-order* di un albero binario (la studieremo):

```
void inorder(struct node *p)
{
    if(p) {
        inorder(p->left);
        dosomething(p);
        inorder(p->right);
    }
}
```



Esempio: inversione di stringa:

```
void reverse(void)
{
    int c;

    if((c = getchar()) != '\n')
        reverse();
    putchar(c);
}
```

Commenti:

`if((c = getchar()) != '\n')`: Condizione di terminazione. `reverse` richiamerà se stessa fino a quando non sarà letto un carattere *newline*.

Ogni volta che `reverse` viene chiamata, viene allocato un nuovo record di attivazione, che comprende una nuova copia dell'ambiente locale. In particolare ci sarà una copia della variabile `c` per ogni chiamata attiva.
(Inoltre il record di attivazione contiene copia degli argomenti passati nella chiamata, spazio per allocare le variabili locali di classe `auto` e altre informazioni ausiliarie).

I record di attivazione vengono posti su uno stack.
Si effettua un'operazione `push` su questo stack ogni volta che una funzione viene invocata.
Si effettua un'operazione `pop` su questo stack ogni volta che una funzione invocata restituisce il controllo all'ambiente chiamante.

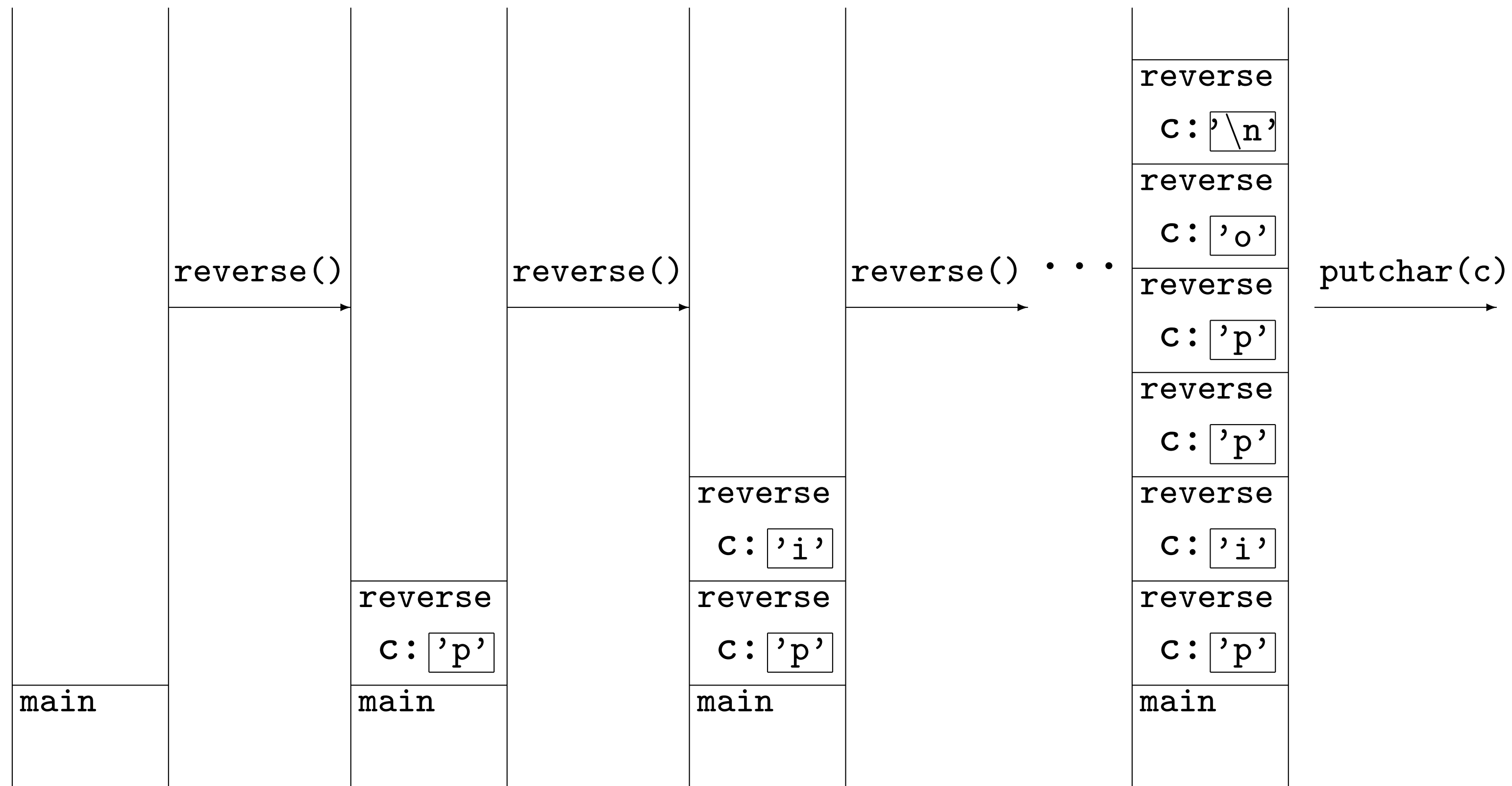
Ogni invocazione di `reverse` termina stampando sul video il carattere letto.

Poiché l'ordine di terminazione delle invocazioni innestate di `reverse` è inverso all'ordine in cui le invocazioni sono state effettuate, i caratteri letti saranno stampati in ordine inverso, a partire da `'\n'`.

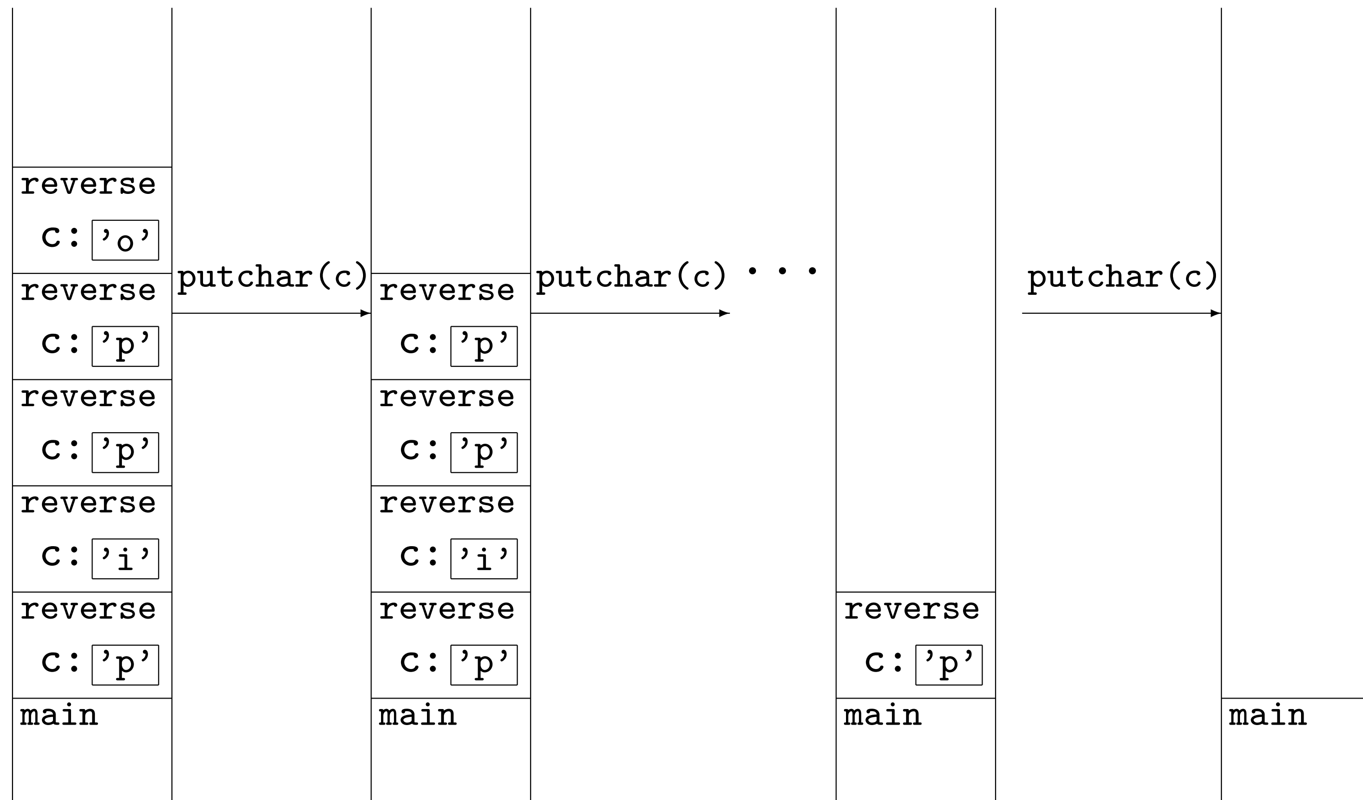
Provare la versione `reverse2.c` per visualizzare lo stack dei record di attivazione (limitatamente al valore di `c`).

Non è difficile dare una formulazione non ricorsiva di `reverse()`. Farlo per esercizio.

Stack dei record di attivazione su input "`pippo`":



Stack dei record di attivazione su input "pippo":



Considerazioni sull'efficienza

In generale, una funzione definita ricorsivamente può essere riscritta senza usare la ricorsione.
Infatti, il compilatore deve svolgere questo compito per produrre il codice in linguaggio macchina.

Spesso le versioni ricorsive sono più brevi ed eleganti delle corrispondenti versioni non ricorsive.

L'uso della ricorsione si paga con il costo aggiuntivo, in tempo e spazio, determinato dalla gestione di un numero elevato di record di attivazione innestati.

Quando questo costo si presenta elevato rispetto al resto del costo dovuto all'algoritmo implementato, si deve considerare la riscrittura non ricorsiva della funzione.

Esempio: I numeri di Fibonacci.

I numeri di Fibonacci sono definiti come segue:

$$f_0 = 0, \quad f_1 = 1, \quad f_i = f_{i-1} + f_{i-2}, \text{ per } i > 1$$

La successione degli f_i cresce esponenzialmente, anche se più lentamente di 2^i. Potremo calcolarne solo pochi valori, diciamo fino a f_{45}.

Ecco una funzione ricorsiva per il calcolo dei numeri di Fibonacci:

```
/* versione ricorsiva */
long fibor(int i)
{
    return i <= 1 ? i : fibor(i-1) + fibor(i-2);
}
```

Ecco una funzione iterativa per il calcolo dei numeri di Fibonacci:

```
/* versione iterativa */
long fiboi(int i)
{
    long f0 = 0L, f1 = 1L, temp;
    int j;

    if(!i)
        return 0;
    for(j = 2;j <= i;j++) {
        temp = f1;
        f1 += f0;
        f0 = temp;
    }
    return f1;
}
```

Mettendo `fibor` e `fiboi` a confronto osserviamo come `fiboi` sia meno elegante ma molto più efficiente.

`fibor` è inefficiente per due motivi:

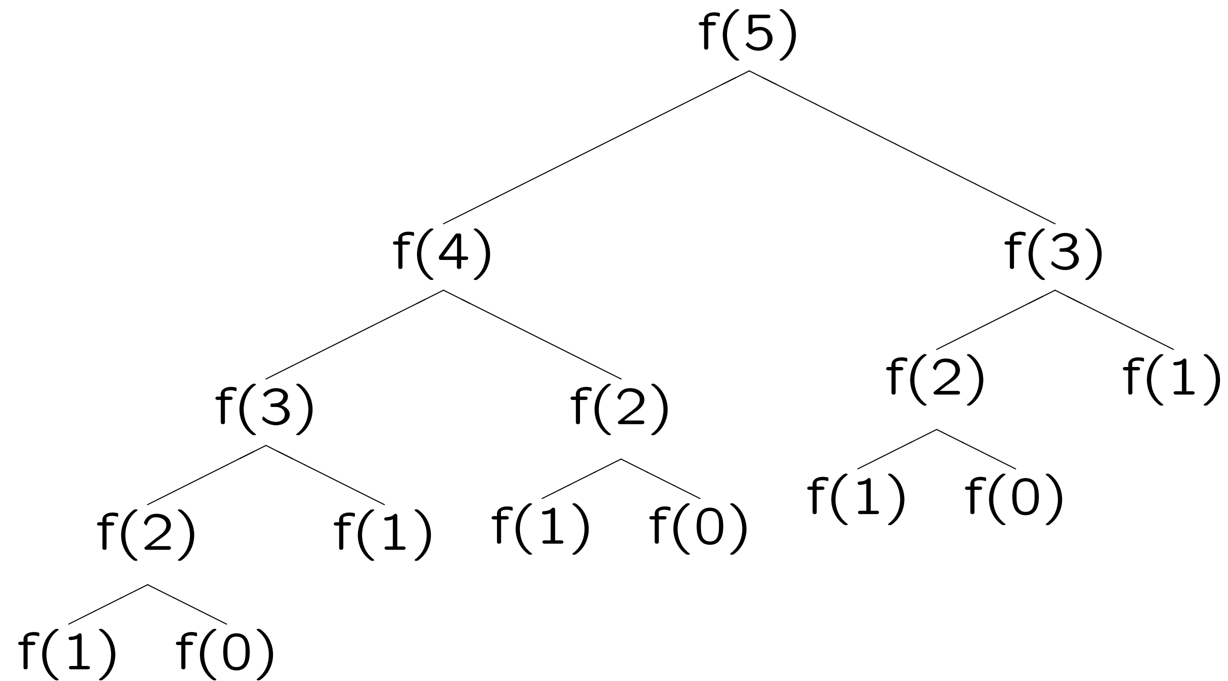
1: l'allocazione e deallocazione di numerosi record d'attivazione. Questo è il difetto ineliminabile delle funzioni ricorsive.

2: `fibor` effettua molte più chiamate a se stessa di quanto strettamente necessario: per esempio per calcolare f_9 è sufficiente conoscere il valore di $f_0, f_1, \ldots, f_8$.
`fibor(9)` richiama `fibor` 108 volte!

Non è raro il caso che alcuni accorgimenti rendano accettabili le prestazioni di una funzione ricorsiva eliminando difetti simili a **2**.

Albero di ricorsione per `fibor(5)`.

Confronta con `fibomon.c`.



Tecniche di memorizzazione dei valori

Il semplice accorgimento che migliora le prestazioni di `fibor` consiste nel memorizzare in un array i valori già calcolati.

```
/* versione ricorsiva con memorizzazione dei valori */
long fibor2(int i)
{
    static long f[LIMIT + 1];

    if(i <= 1)
        return f[i] = i;
    return f[i] = (f[i-1] ? f[i-1] : fibor2(i-1))
            + (f[i-2] ? f[i-2] : fibor2(i-2));
}
```

Per calcolare f_i, `fibor2` richiama se stessa al più i volte.

Backtracking

Spesso la soluzione (obiettivo) di un problema può essere trovata esplorando lo spazio di ricerca in modo da perseguire il raggiungimento di un numero finito di sotto-obiettivi.

L'intero processo di ricerca in molti di questi casi può essere descritto come la visita di un albero in cui ogni nodo rappresenta un sotto-obiettivo, e vi è un cammino da un sotto-obiettivo A a un sotto-obiettivo B, se B è raggiungibile dopo aver raggiunto A.

I figli del nodo A sono tutti e soli i sotto-obiettivi raggiungibili *direttamente* (con una *mossa*) da A.

In questi casi l'algoritmo di ricerca è formulato in maniera naturalmente ricorsiva.

Un algoritmo di backtracking visita ricorsivamente l'albero di ricerca esplorando tutti i cammini ammissibili.

Spesso l'albero di ricerca può essere *potato*: l'algoritmo usa dei criteri (*euristiche*) per riconoscere in anticipo alcune delle situazioni in cui la visita di un cammino (o di un sottoalbero) non porta ad alcuna soluzione; ovviamente un tale cammino (o un tale sottoalbero) viene escluso dalla ricerca.

Non appena si giunge a un nodo tale che ogni ramo cui il nodo appartiene viene riconosciuto come "vicolo cieco" (il sottoalbero di cui il nodo è radice non contiene foglie soluzione) tale nodo viene abbandonato e si risale (backtrack) nel ramo fino al nodo più profondo che ammetta mosse non ancora esplorate: da qui la ricerca riprende scegliendo un nuovo cammino attraverso una di queste mosse.

Backtracking: Il giro del cavallo

Scopo del gioco:

Data una scacchiera quadrata di lato N, esibire, se esiste, un percorso che parta dalla casella in alto a sinistra e, muovendosi come il cavallo degli scacchi, tocchi una e una sola volta tutte le rimanenti caselle.

	■		■	
■				■
		□		
■				■
	■		■	

1	28	33	20	3	26
34	19	2	27	8	13
29	32	21	12	25	4
18	35	30	7	14	9
31	22	11	16	5	24
36	17	6	23	10	15

```
#include <stdio.h>

#define N   8

short matrix[N][N];

char backtrack(short x, short y, short n);

int main(void)
{
    short x,y,n;

    matrix[x = 0][y = 0] = n = 1;
    if(backtrack(x,y,1))
        for(y = 0; y < N; y++) {
            for(x = 0; x < N; x++)
                printf("[%2d]",matrix[x][y]);
            putchar('\n');
        }
}
```

```
char checkpos(short x, short y)
{
    return !(x < 0 || x >= N || y < 0 || y >= N || matrix[x][y]);
}

char backtrack(short x, short y, short n)
{
    short i,xx,yy;
    static short m[][2] = { 1,-2, 2,-1, 2,1, 1,2, -1,2, -2,1, -2,-1, -1,-2 };

    if(n == N * N)
        return 1;
    for(i = 0; i < 8; i++)
        if(checkpos(xx = x + m[i][0], yy = y + m[i][1])) {
            matrix[xx][yy] = n+1;
            if(backtrack(xx,yy,n+1))
                return 1;
        }
    matrix[x][y] = 0;
    return 0;
}
```

Commenti: L'algoritmo consiste nell'esplorazione dell'albero delle mosse con la tecnica di backtracking. Una soluzione è un cammino (ramo) lungo $N * N$. Non appena un cammino (più corto di $N * N$) non ha più mosse percorribili, lo si abbandona e si risale al più profondo nodo del cammino per il quale esistono ancora mosse da esplorare.

```
#define N   8

short matrix[N][N];
```

La matrice `matrix` memorizza il cammino attualmente percorso. La dimensione del lato è data da una macro : in C non è possibile dichiarare array di dimensione variabile (a meno di ricorrere all'allocazione dinamica della memoria: lo vedremo).

```
int main(void)
{
    short x,y,n;

    matrix[x = 0][y = 0] = n = 1;
    if(backtrack(x,y,1))
        for(y = 0; y < N; y++) {
            for(x = 0; x < N; x++)
                printf("[%2d]",matrix[x][y]);
            putchar('\n');
        }
}
```

La funzione `main` inizializza `x`, `y` che sono le coordinate della mossa corrente, e `n` che è la profondità del nodo corrente.
Poi innesca la funzione ricorsiva `backtrack`, che ritorna `1` solo se vi è un cammino che prolunga quello corrente e che arriva a profondità `N*N`.
Se tale cammino è stato trovato, la matrice contiene la soluzione corrispondente, che viene stampata.

Analisi di `char backtrack(short x, short y, short n)`

```
static short m[][2] = { 1,-2, 2,-1, 2,1, 1,2, -1,2, -2,1, -2,-1, -1,-2 };
```

La matrice `m`, allocata staticamente, contiene la descrizione delle 8 mosse possibili partendo dalla casella (0,0).

```
if(n == N * N)
    return 1;
```

Condizione di terminazione (con successo): si ritorna `1` se la profondità del nodo è `N*N`.

```
for(i = 0; i < 8; i++)
    ...
```

Si cicla tentando una alla volta le 8 mosse.

```
if(checkpos(xx = x + m[i][0], yy = y + m[i][1])) {
    ...
}
```

Si richiama `checkpos` per stabilire se la mossa è effettuabile oppure no.

```
matrix[xx][yy] = n+1;
if(backtrack(xx,yy,n+1))
    return 1;
```

Se la mossa si può fare la si effettua, e si prosegue il cammino richiamando ricorsivamente `backtrack` sulla nuova mossa.
Se questa invocazione a `backtrack` ritorna 1 allora la soluzione è stata trovata, e questo fatto viene comunicato a tutte le chiamate innestate: la pila ricorsiva si smonta e 1 viene restituito a `main`.

```
matrix[x][y] = 0;
return 0;
```

Altrimenti, nessun cammino proseguente dal nodo ha trovato soluzioni, per cui la mossa viene annullata e viene restituito `0` alla chiamata precedente: se quest'ultima è `main`, allora non sono state trovate soluzioni.

Poiché eventualmente si esplorano tutte le possibili configurazioni legali del gioco, se esiste una soluzione, l'algoritmo prima o poi la trova.
Ovviamente poiché l'albero di ricerca cresce in maniera esponenziale, anche il tempo di calcolo sarà esponenziale rispetto a `N`.

Ultimi commenti al codice:

```
char checkpos(short x, short y)
{
    return !(x < 0 || x >= N || y < 0 || y >= N || matrix[x][y]);
}
```

Questa funzione controlla se la mossa cade nella scacchiera e se la casella da occupare è attualmente libera.

ESERCIZIO:

- L'implementazione presentata fissa come mossa iniziale la occupazione della prima casella in alto a sinistra. Modificare il codice in modo da esplorare anche tutti i cammini che hanno una casella diversa come posizione iniziale (NB: occhio alle simmetrie).

- Una buona strategia per raggiungere "presto" una soluzione consiste nel cercare di occupare sempre le caselle libere più esterne. Modificare l'implementazione in modo che le mosse possibili siano esplorate non nell'ordine fissato dato, ma dando priorità a quelle più "esterne".

Array, Puntatori, Stringhe, Memoria Dinamica

Array

Quando è necessario elaborare una certa quantità di dati omogenei si possono usare variabili *indicizzate*:

```
int a0, a1, a2;
```

Il C supporta questo uso attraverso il tipo di dati array (o vettore):

```
int a[3]; /* definisce le variabili a[0],a[1],a[2] */
```

Il compilatore alloca una zona di memoria contigua sufficiente a contenere i 3 interi.
Poi associa a questa zona l'identificatore `a`.

Questi 3 interi sono accessibili in lettura e in scrittura attraverso la notazione `a[`*index*`]`, dove *index* $\in \{0, 1, 2\}$.

Nessun ulteriore supporto è offerto dal C per i tipi array.

Sintassi:
type *identifier*[*number_of_elements*]

Definisce un array di tipo *type* di *number_of_elements* elementi, e gli associa l'identificatore *identifier*.

Gli elementi hanno indici in $\{0, \ldots, number_of_elements - 1\}$, per accedere l'elemento $(i + 1)$esimo: *identifier*[*i*].

Gli elementi di un array sono *lvalue*: possono comparire come operandi sinistri di assegnamenti, e se ne può usare l'indirizzo.

Si possono definire array di ogni tipo.

Array di array: *type* *identifier* [n_1] [n_2] $\cdots$ [n_k],
per accedervi: *identifier*[i_1] [i_2] $\cdots$ [i_k].

Avvertenze:

Non vi è alcun controllo sui limiti degli array.
Se si cerca di accedere un elemento con indice inesistente, il risultato è imprevedibile: la macchina cercherà di accedere una locazione di memoria non allocata all'array, con esiti casuali, di solito catastrofici.

Nell'accesso a un elemento di un array si applica l'operatore `[]`, che ha la priorità più elevata e associa da sinistra a destra.

Gli array non possono essere `register`.
gli array `extern` e `static` sono per default inizializzati ponendo a 0 tutti gli elementi.

E' possibile inizializzare esplicitamente gli array, anche quelli di classe `auto`:

type *identifier*[*n*] = { $\mathtt{v}_0$, $\mathtt{v}_1$, ..., $\mathtt{v}_k$}

Per ogni indice $i \in \{0, \ldots, k\}$ si ha l'inizializzazione:

identifier[i] = $\mathtt{v}_i$

Gli elementi di indice $i \in \{k+1, \ldots, n-1\}$ sono inizializzati a 0.

In presenza di inizializzazione esplicita *number_of_elements* può essere omesso dalla definizione: il compilatore creerà un array con un numero di elementi uguale al numero dei valori inizializzanti.

`int a[] = {3,1,4};` equivale a `int a[3] = {3,1,4};`

Per array di `char` si può usare la notazione *stringa costante*:

`char c[] = "ciao";` equivale a `char c[5] = {'c','i','a','o',0};`

Puntatori

Sia `v` una variabile di tipo T.
Al momento della creazione della variabile `v` il sistema alloca memoria sufficiente a contenere i valori appartenenti al tipo T.

`&v` restituisce l'indirizzo della zona di memoria allocata per `v`.

L'operatore unario `&` restituisce l'indirizzo in memoria dell'argomento a cui è applicato. Ha la stessa priorità e associatività degli altri operatori unari (priorità minore degli operatori `[]` e `++`,`--` postfissi).

Il tipo dell'espressione `&v` è: *puntatore a* T.

Si possono dichiarare variabili di tipo puntatore: i valori contenuti in queste variabili saranno interpretati come indirizzi in memoria:

```
int v = 5;
int *pv = &v;
```

Sintassi:
type $*$*identifier*
definisce/dichiara la variabile *identifier* appartenente al tipo *puntatore* a *type*.

Per aiutare l'intuizione si legga la dichiarazione come:
" I valori puntati da *identifier* sono di tipo *type* "

Infatti, l'operatore unario di *dereferenziazione* `*` è duale all'operatore di estrazione di indirizzo `&`.
Dato un puntatore `p` al tipo T, l'espressione `*p` restituisce il valore di tipo T rappresentato nella zona di memoria che inizia dalla locazione il cui indirizzo è contenuto in `p`.
Per ogni variabile `v` (di qualunque tipo), vale: `v == *&v`

L'operatore $*$ ha priorità e associatività analoga agli altri operatori unari: più precisamente ha priorità minore degli operatori `[]` e `++`,`--` postfissi.

L'insieme dei valori possibili per un tipo puntatore è costituito da:
– indirizzi reali di oggetti in memoria.
– il valore speciale `0`, che rappresenta il *puntatore nullo*, vale a dire, la situazione in cui si assume che un puntatore non punti ad alcun indirizzo effettivamente dereferenziabile. Il puntatore nullo è anche denotato dal valore `NULL`.
– un insieme di interi positivi interpretati come indirizzi.

```
int v,*p;
double w;

p = 0;            /* assegna a p il puntatore nullo */
p = NULL;         /* equivale a p = 0 */
p = &v;           /* assegna a p l'indirizzo di v */
p = (int *)&w;    /* a p l'indirizzo di w, dopo la conversione a puntatore a int */
p = (int *)1776;  /* a p l'intero 1776, dopo la conversione in puntatore a int */
```

Nella definizione di un puntatore è possibile fornire un valore di inizializzazione:

```
int v = 5;
int *pv = &v; /*  pv inizializzato con l'indirizzo di v */
*pv = 3;      /*  ora v contiene 3 */
```

Attenzione!

`int *pv;` dichiara un puntatore a `int`.

`int *pv = &v;` il puntatore `pv` viene inizializzato con l'indirizzo di `v`.

`*pv = 3;` Il valore intero 3 viene assegnato all'`int` puntato da `pv`.

Nella definizione di `pv`, si dichiara "il tipo puntato è un intero" per definire il puntatore, che è `pv`, non `*pv`.

```
#include <stdio.h>

int main(void)
{
    int a = 1, b = 7, *p = &a;

    printf("int a = 1, b = 7, *p = &a;\n");
    printf("a:%d\tb:%d\t*p:%d\tp:%p\n",a,b,*p,p);

    p = &b;
    printf("\np= &b;\n");
    printf("a:%d\tb:%d\t*p:%d\tp:%p\n",a,b,*p,p);

    (*p)--;
    printf("\n(*p)--;\n");
    printf("a:%d\tb:%d\t*p:%d\tp:%p\n",a,b,*p,p);

    a = 3 * *p;
    printf("\na = 3 * *p;\n");
    printf("a:%d\tb:%d\t*p:%d\tp:%p\n",a,b,*p,p);

    *p--;
    printf("\n*p--;\n");
    printf("a:%d\tb:%d\t*p:%d\tp:%p\n",a,b,*p,p);
}
```

Avvertenze:

Non tutte le espressioni del C sono suscettibili all'estrazione e manipolazione dell'indirizzo.

Non è possibile puntare a:

Costanti numeriche:
`&3` : Errore.

Espressioni ordinarie:
`&(k + 99)` : Errore.
`&&v` : Errore.

Variabili `register`.
`register int v;`
`&v` : Errore.

Passaggio di parametri per indirizzo

Il C supporta solo il passaggio dei parametri per valore.

Per simulare il passaggio per referenza, si utilizzano tipi puntatore nei parametri formali: nella chiamata si devono passare gli indirizzi degli argomenti, utilizzando esplicitamente, laddove necessario, l'operatore `&`.

```
void swap(int *a, int *b)
{
    int tmp = *a;

    *a = *b;
    *b = tmp;
}

swap(&x, &y);
```

```
#include <stdio.h>

void swap(int *a, int *b)
{
    int tmp = *a;

    printf("\nswap(int *a, int *b)\n");
    printf("---> a = %p\t*a = %d\tb = %p\t*b= %d\n",a,*a,b,*b);
    printf("\nint tmp = *a;\n");
    printf("---> tmp = %d\n",tmp);
    *a = *b;
    *b = tmp;
    printf("\n*a = *b; *b = tmp;\n");
    printf("---> a = %p\t*a = %d\tb = %p\t*b= %d\n",a,*a,b,*b);
}

int main(void)
{
    int x = 5;
    int y = 7;
    printf("---> x = %d. y = %d.\n",x,y);
    printf("---> &x = %p\t&y = %p\n",&x,&y);
    swap(&x,&y);
    printf("---> x = %d. y = %d.\n",x,y);
}
```

Si possono definire puntatori a tutti i tipi del C.

Per definire un puntatore a puntatore al tipo `T`:

```
T **p;
```

Per scambiare il valore di due puntatori a `int`:

```
void swap(int **a, int **b)
{
    int *tmp = *a;

    *a = *b;
    *b = tmp;
}

int x, y, *p = &x, *q = &y;
swap(&p, &q);
```

Array e puntatori

Il nome di un array **è** un puntatore al suo primo elemento:

```
int a[10];

if(a == &a[0]) printf("Vero\n"); /* stampa "Vero" */
```

Più in generale, iniziando a usare l'aritmetica dei puntatori:

`&a[i]`	equivale a	`a + i`
`a[i]`	equivale a	`*(a + i)`

Possiamo applicare l'operatore [] anche ai puntatori

```
int a[10];
int *p = a;   /* assegna al puntatore p l'indirizzo del primo elemento di a */

p[2] = 5;     /* assegna 5 ad a[2] */
*(p + 2) = 5; /* assegna 5 ad a[2] */
*(a + 2) = 5; /* assegna 5 ad a[2] */
```

Avvertenze:

Vi sono differenze tra array e puntatori.

– In primo luogo definendo un array di `n` elementi di tipo `T` si alloca memoria contigua per `n * sizeof(T)` byte e si assegna l'indirizzo del primo di questi byte al nome dell'array.

– Il nome di un array è un puntatore costante, e ogni tentativo di modificarne il valore (del puntatore, non di ciò a cui esso punta) causerà un errore in compilazione.

```
int a[10];
int *p = a;   /* assegna al puntatore p l'indirizzo del primo elemento di a */

*++p = 3;     /* p viene incrementato, ora p == &a[1]. Poi assegna 3 ad a[1] */
*a = 4;       /* equivale ad a[0] = 4; */
*++a = 3;     /* ERRORE! il nome di un array e' un puntatore costante */
```

Aritmetica dei puntatori

Sia `p` un puntatore al tipo `T`: `T *p = &x;`

A `p` si può sommare un'espressione intera:
Le seguenti sono espressioni legali:
`p + 2 ,  --p ,  p += 3 ,  p -= 5`

Il valore dell'espressione `p + e` è l'indirizzo ottenuto sommando all'indirizzo di `p` il valore di `e * sizeof(T)`.

Si realizza uno spiazzamento rispetto a `p` del numero di byte richiesto dallo spazio necessario a memorizzare contiguamente `e` oggetti di tipo `T`.

Sia `q` un puntatore dello stesso tipo di `p`: `T *q = &y;`

L'espressione `p - q` restituisce il numero di oggetti di tipo `T` allocati nell'intervallo di memoria
contigua che va da `q` a `p`.
NB: `p` e `q` devono puntare ad elementi dello stesso vettore o al massimo il maggiore dei due può puntare al primo byte successivo alla zona di memoria allocata all'array, altrimenti il risultato non è ben definito.

Se `q` è maggiore di `p`, l'espressione `p - q` restituisce il numero negativo uguale a: `-(q - p)`.

Quando `p` e `q` puntano ad elementi dello stesso vettore, possono essere confrontati con gli operatori relazionali e d'uguaglianza:
`p > q` sse `p - q > 0`

Array come parametri di funzione

Nelle definizioni di funzione, i parametri di tipo array sono, a tutti gli effetti, parametri di tipo puntatore.

`void bubble(int a[], int n) { ... }` equivale a
`void bubble(int *a, int n) { ... }`

Quando si passa un array a una funzione, in realtà si passa un puntatore al suo primo elemento.
E' per questo che non si deve usare l'operatore `&` quando si desidera ottenere l'alterazione del valore dell'array —più precisamente: del valore dei suoi elementi— da parte della funzione chiamata:

```
char s[100];
int i;

scanf("%s%d", s, &i);  /* sarebbe stato sbagliato scrivere &s */
```

Allocazione dinamica della memoria

Il ciclo di vita della memoria allocata a una variabile è controllato dal sistema e determinato dalla classe di memorizzazione della variabile stessa.

E' possibile gestire direttamente il ciclo di vita di una zona di memoria per mezzo del meccanismo di

allocazione e *deallocazione dinamica*.

Il programma può richiedere al sistema di allocare il quantitativo di memoria specificato. Se tale richiesta ha successo, il sistema ritorna l'indirizzo del primo byte di tale zona.

La memoria così ottenuta rimane allocata fino a quando il processo termina oppure fino a quando il programma non comunica esplicitamente al sistema che essa può essere rilasciata e resa nuovamente disponibile.

La regione di memoria adibita dal sistema alla gestione delle richieste di allocazione/deallocazione dinamica — lo *heap* — solitamente è separata dalla regione di memoria usata per le variabili.

Poiché il controllo della memoria allocata dinamicamente spetta unicamente al programma, il suo uso è fonte di frequenti errori di programmazione che consistono nel tentativo di accedere una zona non ancora allocata, oppure già deallocata ed eventualmente riallocata ad un'altra risorsa.

Le funzioni `malloc`, `calloc`, `free`.

I prototipi di queste funzioni e la definizione del tipo `size_t` sono in `stdlib.h`

`size_t` coincide con `unsigned int` sulla maggior parte dei sistemi.

```
void *malloc(size_t size);
```

Richiede al sistema di allocare una zona di memoria di `size` byte. Se la richiesta ha successo viene restituito un puntatore di tipo `void *` che contiene l'indirizzo del primo byte della zona allocata. Se la richiesta fallisce viene restituito il valore `NULL`.

Un puntatore di tipo `void *` è considerato un puntatore di tipo "*generico*": esso può essere automaticamente convertito a puntatore al tipo T, per ogni tipo T.

L'uso corretto di `malloc` prevede l'uso contestuale dell'operatore `sizeof`.

```
int *pi = malloc(sizeof(int));
```

`malloc` richiede al sistema l'allocazione di una zona di memoria che possa contenere esattamente un valore di tipo `int`. L'indirizzo ottenuto (oppure `NULL`) viene assegnato, senza dover usare una conversione esplicita, alla variabile `pi` di tipo puntatore a `int`.

```
double *pd;

if( !( pd = malloc(5 * sizeof(double)) ) )
    error(...)
else
    use(pd);
```

`malloc` richiede l'allocazione di una zona che contenga 5 oggetti di tipo `double`. Si controlla che la richiesta abbia avuto successo.

`void *calloc(size_t n_elem, size_t elem_size);`
Richiede al sistema l'allocazione di un array di `n_elem` elementi di dimensione `elem_size`.

Se la richiesta ha successo:
– gli elementi vengono inizializzati a 0.
– viene restituito l'indirizzo del primo byte del primo elemento.

Se la richiesta fallisce: viene restituito `NULL`.

```
double *pd;

if( !( pd = calloc(5, sizeof(double)) ) )
    error(...);
else
    use(pd);
```

`malloc` non inizializza a 0 la zona allocata.

La memoria allocata con `malloc` o `calloc` non viene deallocata all'uscita dal blocco o dalla funzione.

Per deallocarla bisogna inoltrare una richiesta esplicita al sistema:

```
void free(void *p);
```

– Se `p` punta al primo byte di una zona di memoria allocata dinamicamente (da `malloc`, `calloc` o `realloc`) (e non ancora rilasciata (!)), allora tutta la zona viene rilasciata.
– Se `p` è `NULL`, `free` non ha effetto.
– Se `p` non è l'indirizzo di una zona di memoria allocata oppure è l'indirizzo di una zona già rilasciata, il processo cade in una situazione inconsistente, dagli effetti imprecisati: E' un errore da evitare.

Si noti che `p` non viene posto a `NULL` da `free(p)`.

Per modificare dinamicamente la dimensione di una zona di memoria allocata precedentemente (e non ancora rilasciata):

```
void *realloc(void *p, size_t size);
```

Se `p` è l'indirizzo base di una zona di memoria allocata di dimensione `sizeold`, `realloc` restituisce l'indirizzo di una zona di memoria di dimensione `size`.
—– Se `sizeold <= size` il contenuto della vecchia zona non viene modificato, e lo spazio aggiuntivo non viene inizializzato.
—– Se `size < sizeold` il contenuto dei primi `size` byte della vecchia zona non vengono modificati.

– `realloc`, se è possibile, non cambia l'indirizzo base della zona: in questo caso il valore ritornato è, per l'appunto, `p`.
– Se questo non è possibile, `realloc` alloca effettivamente una zona diversa e vi copia il contenuto della zona vecchia, poi dealloca quest'ultima e restituisce l'indirizzo base della nuova.

Esempio: Giro del cavallo modificato:
La dimensione del lato (e quindi della matrice che rappresenta la scacchiera) viene specificata da input, e la matrice allocata dinamicamente:

```
#include <stdio.h>
#include <stdlib.h>
short *matrix;  /* Nota: matrix e' short *, non piu' un array bidimensionale di short */
...
int main(void)
{
    short x,y,n,N;

    printf("\nImmetti dimensione lato:\n");
    scanf("%hd",&N);
    if(!(matrix = calloc(N * N, sizeof(short)))){fprintf(stderr,"Errore di allocazione\n");exit(-1);}
    x = y = 0;
    matrix[0] = n = 1;
    if(backtrack(x,y,1,N))            /* Si comunica a backtrack anche la dimensione del lato */
        for(y = 0; y < N; y++) {
            for(x = 0; x < N; x++)
                printf("[%2d]",matrix[x * N + y]); /* Nota come si accede ora a matrix[x][y] */
            putchar('\n');
        }
    free(matrix);
}
```

Esempio: Mergesort.

```
void mergesort(int *a, int *b, int l, int r)
{
    int i,j,k,m;

    if(r > l) {
        m = (r+l)/2;
        mergesort(a,b,l,m);
        mergesort(a,b,m+1,r);
        for(i = m+1; i > l; i--)
            b[i-1] = a[i-1];
        for(j = m; j < r; j++)
            b[r+m-j] = a[j+1];
        for(k = l; k <= r; k++)
            a[k] = b[i] < b[j] ? b[i++] : b[j--];
    }
}
void ordina(int *a, int l, int r)
{
    int *b = calloc(r - l + 1, sizeof(int));

    if(!b) { fprintf(stderr,"Errore di allocazione\n"); exit(-1); }
    mergesort(a,b,l,r);
    free(b);
}
```

`void ordina(int *a, int l, int r)`
L'algoritmo di ordinamento `mergesort` è implementato ricorsivamente. Ha inoltre bisogno di un array `b[]` di appoggio per ordinare l'array `a[]`. la funzione `ordina` crea l'array ausiliario `b[]` e richiama `mergesort(a,b,l,r)`.

`int *b = calloc(r - l + 1, sizeof(int));`
Si alloca dinamicamente un array di `r - l + 1` elementi `int` e se ne assegna l'indirizzo base a `b`.

`if(!b) {...}`
Si controlla che l'allocazione dinamica abbia avuto successo.

`free(b);`
Si rilascia la zona di memoria allocata dinamicamente il cui indirizzo base è contenuto in `b`.

Lo schema generico dell'algoritmo ricorsivo `mergesort` è:

```
void mergesort(int *a, int l, int r)
{
    int m;

    if(r > l) {
        m = (r+l)/2;      /* NB: e' il Floor del razionale (r+l)/2 */
        mergesort(a,l,m);    /* sulla prima meta'   */
        mergesort(a,m+1,r);  /* sulla seconda meta' */
        merge(a,l,m,r);      /* combina i due sottoarray ordinati */
    }
}
```

Dove la funzione `merge` deve "*fondere*" in un unico array ordinato i due sottoarray di `a` già ordinati.

E' un classico esempio di tecnica *Divide et Impera*.

La corrispondente equazione di ricorrenza è:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T_{\mathtt{merge}}(n) & \text{otherwise.} \end{cases}$$

Poiché, come vedremo,

$$T_{\mathtt{merge}}(n) = \Theta(n)$$

si può riscrivere:

$$T(n) = 2T(n/2) + \Theta(n)$$

La cui soluzione è

$$T(n) = \Theta(n \log_2 n).$$

Supponendo di fondere nell'array `c` i due array ordinati `a` di `m` elementi e `b` di `n` elementi:

```
void merge(int *a, int *b, int *c, int m, int n)
{
    int i = 0, j = 0, k = 0;

    while(i < m && j < n)      /* in c[k] il piu' piccolo tra a[i] e b[j] */
        if(a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    while(i < m)               /* esaurito b[], si accoda la parte restante di a[] */
        c[k++] = a[i++];
    while(j < n)
        c[k++] = b[j++];       /* esaurito a[], si accoda la parte restante di b[] */
}
```

Fino a quando non si è esaurito almeno uno degli array `a[]` e `b[]`, viene posto in `c[k]` il più piccolo fra `a[i]` e `b[j]`. Gli indici sono incrementati di conseguenza.
Infine, avendo esaurito uno tra `a[]` e `b[]` si accoda a `c[]` quanto rimane dell'altro.

Nel nostro caso si realizza la fusione (`merge`) nell'array `a[]` delle due parti dell'array ausiliario `b[]`.
Per prima cosa bisogna istanziare opportunamente l'array `b[]`:

```
for(i = m+1; i > l; i--)
    b[i-1] = a[i-1];
for(j = m; j < r; j++)
    b[r+m-j] = a[j+1];
```

Gli elementi da `a[l]` ad `a[m]` sono copiati nei corrispondenti elementi di `b[]`. Alla fine `i == l`.
Gli elementi da `a[m+1]` ad `a[r]` sono copiati negli elementi `b[m]`,...,`b[r]`, ma *in ordine inverso*. Alla fine `j = r`.

```
for(k = l; k <= r; k++)
    a[k] = b[i] < b[j] ? b[i++] : b[j--];
```

Queste righe realizzano la fusione. Il trucco di aver copiato la seconda parte in ordine inverso e il fatto che la scandiamo da destra a sinistra (`b[j--]`), ci permettono di non doverci occupare delle "code" degli array. La preparazione di `b[]` costa $\Theta(n)$. Ciò non incide sul comportamento asintotico.

Stringhe

Il tipo stringa in C è rappresentato da array di `char`, il cui ultimo carattere è il carattere nullo `'\0'` (cioè il valore intero `0`).

`char s[] = "pippo";` equivale a `char s[] = {'p','i','p','p','o',0};`

Una costante stringa in un'espressione è un puntatore al primo carattere della zona di memoria dove la stringa stessa è memorizzata.

```
char s[] = "ban";
printf("%s%s%c", s, s+1, *(s+1)); /* stampa banana */
```

Nota: `char *s = "stringa";` differisce da `char s[] = "stringa";`
Nel primo caso, `s` è una variabile puntatore a `char` inizializzata con l'indirizzo del primo byte della zona di memoria allocata alla costante `"stringa"`.
Nel secondo caso, `s` è un valore puntatore costante: `s` non può essere modificato.

Manipolare stringhe

La manipolazione di stringhe si effettua attraverso la manipolazione dei puntatori associati.

Per copiare (duplicare) una stringa `s` in una stringa `t`:
– allocare (dinamicamente o no), spazio sufficiente per la stringa `t` che deve ricevere la copia.
– copiare `s` in `t` carattere per carattere:

```
char *s = "stringa";
char t[10];
char *t1 = t, *s1 = s;

while(*t1++ = *s1++);
```

Cosa avrebbe comportato scrivere invece: `t = s;` ? E `s = t;` ?

Alcuni esempi:

```
/* copia s in t e restituisce t */
char *strcpy(char *t, const char *s)
{
    register char *r = t;

    while(*r++ = *s++);
    return t;
}
```

`const char *s` : il qualificatore `const` informa che il carattere puntato da `s` è costante e non può essere modificato.

`while(*r++ = *s++);` :
Viene scandita `s`. Si termina quando il carattere puntato è `0`.
Contestualmente il carattere viene copiato nel carattere puntato dalla scansione di `t`.
`;` :Il ciclo non ha corpo poiché tutto il lavoro è svolto nell'argomento di `while`.

```
/* restituisce la lunghezza della stringa s (non conta lo 0 finale) */
unsigned strlen(const char *s)
{
    register int i;

    for(i = 0; *s; s++, i++);
    return i;
}
```

`const char *s` : anche qui si indica che il carattere puntato da `s` è costante. Invece `s` stesso può essere modificato.

`for(i = 0; *s; s++, i++);` :
Si scandisce `s`. Ci si ferma quando il carattere puntato è `0`.
Simultaneameante a `s` si incremente il contatore `i` (nota l'uso dell'operatore virgola).
`;` :Anche in questo caso non c'è bisogno di fare altro e il corpo del ciclo è vuoto.

Versione alternativa di `strlen` che usa l'aritmetica dei puntatori:

```
unsigned strlen(const char *s)
{
    register const char *t = s;

    while(*t++);
    return t-s-1;
}
```

`register const char *t = s;` :
Dichiariamo un puntatore a carattere costante `t` dello stesso tipo di `s`. Non ci sono problemi a dichiararlo `register`. Non modifichiamo mai caratteri puntati da `t`.

`while(*t++);` :
Cicliamo fino a incontrare il valore `0` di terminazione della stringa. All'uscita dal ciclo `t` punta al carattere successivo allo `0`.

`return t-s-1;` :
Utilizziamo la differenza tra puntatori per determinare quanti caratteri abbiamo scandito.

La libreria standard contiene l'implementazione di versioni di `strcpy` e di `strlen`.

Per usare le funzioni per la manipolazione di stringhe, bisogna includere il file d'intestazione `string.h`.

Altre funzioni nella libreria standard (richiedono `string.h`):

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

concatena `s2` a `s1`. Restituisce `s1`.
`s1` deve puntare a una zona di dimensione sufficiente a contenere il risultato.

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

restituisce un intero `< 0` se `s1` precede `s2`, `0` se `s1 == s2`, `> 0` se `s1` segue `s2` nell'ordine lessicografico.

Array multidimensionali

Un array k-dimensionale è un array di array $(k-1)$-dimensionali.

`int matrix[10][10];` : definisce una matrice di 10 x 10 interi.

Per accedere all'elemento di posizione (j,k): `matrix[j][k]`.

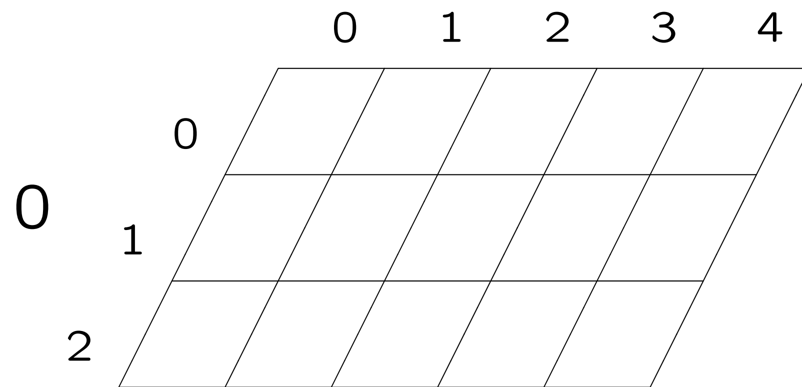
L'indirizzo dell'elemento base è `&matrix[0][0]`,
mentre `matrix == &matrix[0]`.

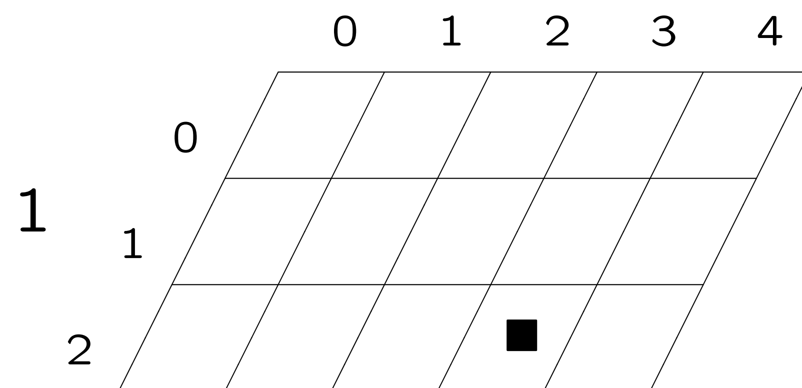
In genere, per l'array `int a[`k_1`][`k_2`]`$\cdots$`[`k_n`];` si ha: `a == &a[0]` e

`a[`i_1`][`i_2`]`$\cdots$`[`i_n`] ==`

`(&a[0][0]`$\cdots$`[0] +` $k_n * k_{n-1} * \cdots * k_2 * i_1 + k_n * k_{n-1} * \cdots * k_3 * i_2 + \cdots + k_n * i_{n-1} + i_n$`)`

Si dichiari `int a[2][3][5]`.
La mappa di memorizzazione ci dice che l'elemento `a[1][2][3]` è l'elemento in posizione $5*3*1+5*2+3 = 28$ sulle $5*3*2 = 30$ posizioni $0, \ldots, 29$ che costituiscono l'array `a`.





Nella definizione

`int a[7][5];`

il nome `a` è un puntatore ad un array di interi, con valore dato da `&a[0]`, mentre l'indirizzo del primo elemento dell'array bidimensionale è `&a[0][0]`: il valore è lo stesso, ma il tipo è diverso.

```
int a[7][5];
int *p,*q;
int (*pp)[5];

printf("&a[0]:%p\t&a[0][0]:%p\n",&a[0],&a[0][0]);
p = &a[0][0];
pp = &a[0];
q = *a;
q = *(&a[0]);
printf("p:%p\tpp:%p\tq:%p\n",p,pp,q);
```

Nelle dichiarazioni di parametri formali si può omettere la lunghezza della prima dimensione dell'array multidimensionale:
la formula vista prima per l'accesso agli elementi (la *mappa di memorizzazione*) non abbisogna del valore di k_1.

`int a[][5]` equivale a `int (*a)[5]` equivale a `int a[7][5]`

Le altre lunghezze non possono essere omesse.

Anche nell'inizializzazione esplicita la prima lunghezza può essere omessa:

```
int a[2][3] = {{1,2,3},{4,5,6}};    equivale a
int a[][3] = {{1,2,3},{4,5,6}};     equivale a
int a[2][3] = {1,2,3,4,5,6};
```

ESEMPIO: Ordinamento di parole in un testo

```
/* sw.c */

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

#define BUFFERSIZE 512
#define BLOCKSIZE 10

char *leggiparola(FILE *fp)
{
    static char buf[BUFFERSIZE];
    char *s = buf;
    char *t;

    while((*s = fgetc(fp)) != EOF && !isalnum(*s));
    if(*s != EOF)
        while(isalnum(*++s = fgetc(fp)));
    *s = 0;
    t = malloc(strlen(buf)+1);
    return strlen(buf) ? strcpy(t,buf) : NULL;
}
```

```
char **leggitesto(FILE *fp,int *pn)
{
    char **words = malloc(BLOCKSIZE * sizeof(char *));
    char *s;

    *pn = 0;
    while((s = leggiparola(fp)) && strcmp("END",s) != 0) {
        words[(*pn)++] = s;
        if(!(*pn % BLOCKSIZE))
            words = realloc(words,(*pn/BLOCKSIZE + 1) * BLOCKSIZE * sizeof(char *));
    }
    return words;
}

void mergesort(char **a, char **b, int l, int r)
{
    int i,j,k,m;

    if(r > l) {
        m = (r+l)/2;
        mergesort(a,b,l,m);
        mergesort(a,b,m+1,r);
        for(i = m+1; i > l; i--)      b[i-1] = a[i-1];
        for(j = m; j < r; j++)        b[r+m-j] = a[j+1];
        for(k = l; k <= r; k++)       a[k] = strcmp(b[i],b[j]) < 0 ? b[i++] : b[j--];
    }
}
```

```
int main(void)
{
    int i,n;
    char **w = leggitesto(stdin,&n);
    char **b = calloc(n, sizeof(char *));

    mergesort(w,b,0,n-1);
    for(i = 0;i < n;i++)
        printf("%s\n",w[i]);
}
```

Commenti: `sw.c` legge un testo e ne estrae le parole fino ad incontrare la fine del file. Poi elenca le parole estratte in ordine lessicografico.

L'elenco delle parole viene memorizzato in un array di stringhe allocato dinamicamente.

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

Per usare `malloc`,`calloc` e `realloc` includiamo `stdlib.h`.
Per usare `strcpy`,`strlen` e `strcmp` includiamo `string.h`.
Il file `ctype.h` contiene numerose macro per effettuare test su caratteri.
Qui usiamo `isalnum(c)` che è vera quando `c` è un carattere alfanumerico.

```
#define BUFFERSIZE 512
#define BLOCKSIZE 10
```

Consideriamo parole lunghe al più `BUFFERSIZE` caratteri.
Allochiamo l'array di puntatori a `char` `BLOCKSIZE` elementi per volta.

```
char *leggiparola(FILE *fp)
```

legge una parola dallo stream associato a `fp`.

```
    static char buf[BUFFERSIZE];
    char *s = buf;
    char *t;
```

`buf` è la memoria di lavoro dove si costruisce la parola letta. `s` scandisce `buf` carattere per carattere. `t` punterà alla parola letta allocata in memoria dinamica.

```
    while((*s = fgetc(fp)) != EOF && !isalnum(*s));
    if(*s != EOF)
        while(isalnum(*++s = fgetc(fp)));
    *s = 0;
```

Si "saltano" tutti i caratteri letti fino al prossimo alfanumerico.
Se non si è incontrato `EOF`, si accumulano caratteri in `buf`, grazie alla scansione `++s`, fintanto che si leggono alfanumerici.
`*s = 0` pone il carattere di terminazione della stringa letta in `buf`.

```
    t = malloc(strlen(buf)+1);
    return strlen(buf) ? strcpy(t,buf) : 0;
```

Viene allocata la memoria dinamica necessaria a contenere la stringa appena letta, e se ne assegna l'indizirizzo base a `t`.
Se `buf` è vuota si restituisce il puntatore nullo, altrimenti si restituisce `t`, dopo avervi copiato la stringa contenuta in `buf`.
N.B. per brevità non si è controllato l'esito di `malloc`: nelle applicazioni vere bisogna farlo sempre.

```
char **leggitesto(FILE *fp,int *pn)
```

`leggitesto` crea l'elenco delle parole estratte dallo stream associato a `fp`.
In `*pn`, passata per indirizzo, si memorizza il numero di parole lette.
La funzione restituisce un puntatore a puntatore a `char`: restituisce l'indirizzo base di un array di `char *` allocato dinamicamente.

```
char **words = malloc(BLOCKSIZE * sizeof(char *));
```

`words` è un puntatore a puntatore a `char`.
Gli si assegna l'indirizzo base di una zona di `BLOCKSIZE` elementi di tipo `char *`.

```
while((s = leggiparola(fp)) && strcmp("END",s) != 0) {
    words[(*pn)++] = s;
```

Si esce dal ciclo quando viene letta la parola vuota (da `leggiparola(fp)`) o quando la parola letta è `"END"`.
Si pone la parola letta nell'array `words`, nella posizione scandita da `*pn`.
NOTA gli assegnamenti:
`s = leggiparola(fp)` e `words[...] = s`.
Qui si maneggiano direttamente i puntatori, non si creano invece copie delle stringhe puntate.

```
    if(!(*pn % BLOCKSIZE))
        words = realloc(words,(*pn/BLOCKSIZE + 1) * BLOCKSIZE * sizeof(char *));
```

Se si è esaurita la capacità corrente dell'array `words`, lo si realloca con `BLOCKSIZE` elementi nuovi.
Anche in questo caso, bisognerebbe controllare che `realloc` abbia successo.

```
    return words;
```

Viene restituito l'array di parole così costruito.

```
void mergesort(char **a, char **b, int l, int r) {
```

Per ordinare l'array `words` si adatta a questo tipo (`char *`) l'algoritmo `mergesort`.

```
        a[k] = strcmp(b[i],b[j]) < 0 ? b[i++] : b[j--];
```

Per il confronto tra elementi si utilizza `strcmp`.

Nel `main` ci si deve preoccupare di creare un array di appoggio `b` di tipo e dimensioni adeguate.

```
int main(void)
{
    int i,n;
    char **w = leggitesto(stdin,&n);
    char **b = calloc(n, sizeof(char *));

    mergesort(w,b,0,n-1);
    for(i = 0;i < n;i++)
        printf("%s\n",w[i]);
}
```

Non ci si preoccupa di liberare con `free` la memoria allocata dinamicamente, poiché potremmo liberarla solo poco prima della fine del processo stesso, allorché viene rilasciata automaticamente.

Controllo dell'input orientato ai caratteri: `ctype.h`

Il file d'intestazione `ctype.h` contiene numerosi prototipi e macro relativi alla gestione dei caratteri:

```
int isalpha(int c); /* >0 se c e' una lettera, 0 altrimenti*/
int isupper(int c); /* >0 se c e' una lettera maiuscola, 0 altrimenti*/
int islower(int c); /* >0 se c e' una lettera minuscola, 0 altrimenti*/
int isdigit(int c); /* >0 se c e' una cifra, 0 altrimenti */
int isalnum(int c); /* >0 se c e' una lettera o una cifra, 0 altrimenti*/
int isspace(int c); /* >0 se c e' un carattere di spaziatura, 0 altrimenti */
int ispunct(int c); /* >0 se c e' un carattere di punteggiatura, 0 altrimenti*/
...

int tolower(int c); /* se c e' maiuscola restituisce
                       la corrispondente minuscola, altrimenti c */
int toupper(int c); /* se c e' minuscola restituisce
                       la corrispondente maiuscola, altrimenti c */
```

dove >0 è un valore positivo ma non fissato dallo standard.

Argomenti di `main`:

La funzione `main` può ricevere due argomenti:

– Il primo, di tipo `int` contiene il numero dei parametri con cui l'eseguibile è stato richiamato dalla riga di comando del sistema operativo.

– Il secondo, di tipo `char *[]` è un array di stringhe ognuna delle quali è una parola della riga di comando.
La prima di queste stringhe contiene il nome stesso dell'eseguibile.
(in alcuni sistemi DOS e Windows, il percorso intero dell'eseguibile).

Convenzionalmente il primo parametro è chiamato `int argc`,
il secondo `char *argv[]`.

Supponiamo di aver compilato `myprog.c`,
e supponiamo che il `main` sia:

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

Si esegua dal prompt del sistema operativo:

```
myprog par1 par2 par3
```

Allora:

`argc`	contiene	`4`
`argv[0]`	contiene	`"myprog"`
`argv[1]`	contiene	`"par1"`
`argv[2]`	contiene	`"par2"`
`argv[3]`	contiene	`"par3"`

Esempio: modifichiamo `sw.c`:

```
int main(int ac, char *av[])
{
    int i,n;
    char **w, **b;
    FILE *fr = stdin, *fw = stdout;

    switch(ac) {
    case 1:
        break;
    case 3:
        if(!(fw = fopen(av[2],"w")))
            errore("Impossibile scrivere sul file \"%s\"",av[2]);
    case 2:
        if(!(fr = fopen(av[1],"r")))
            errore("File \"%s\", di input non trovato",av[1]);
        break;
    default:
        errore("Numero di argomenti non permesso","");
    }
    w = leggitesto(fr,&n);
    b = calloc(n, sizeof(char *));
    mergesort(w,b,0,n-1);
    for(i = 0;i < n;i++)
        fprintf(fw,"%s\n\r",w[i]);
}
```

Nota:

L'intestazione di `main` si sarebbe potuta scrivere anche come:

```
int main(int ac, char **av)
```

infatti `char *av[]` dichiara un'array di puntatori a `char`, e nel contesto di una dichiarazione di parametri di funzione, ciò equivale a dichiarare `char **av`: cioè, puntatore a puntatore a `char`.

Esercizio:

Nella funzione `main` dell'esempio precedente, si è fatto un uso poco ortodosso dell'istruzione `switch`.

Motivate l'affermazione precedente.
Spiegate perchè il codice funziona.
Riscrivete il codice senza usare `switch`.

Puntatori a funzioni

In C, così come il nome di un array è un puntatore al suo primo elemento, il nome di una funzione è un puntatore alla funzione stessa.

Il C permette la manipolazione esplicita dei puntatori a funzione. La sintassi è intricata.

Sia `int f(short, double)`.
Allora il nome `f` è un puntatore di tipo:
`int (*)(short, double)`.

Dichiarando un puntatore dello stesso tipo, si può effettuare, ad esempio, un assegnamento:

```
int (*ptrtof)(short, double);  /* ptrtof puntatore a funzione */

ptrtof = f;                    /* assegna l'indirizzo di f a ptrtof */
(*ptrtof)(2,3.14);             /* invoca f */
```

Un uso tipico dei puntatori a funzione consiste nel passarli come argomenti di funzioni:
Modifichiamo `mergesort`:

```
void mergesort(int *a, int *b, int l, int r, char (*comp)(int, int))
{
    int i,j,k,m;

    if(r > l) {
        m = (r+l)/2;
        mergesort(a,b,l,m,comp);
        mergesort(a,b,m+1,r,comp);
        for(i = m+1; i > l; i--)
            b[i-1] = a[i-1];
        for(j = m; j < r; j++)
            b[r+m-j] = a[j+1];
        for(k = l; k <= r; k++)
            a[k] = (*comp)(b[i],b[j]) ? b[i++] : b[j--];
    }
}
```

Ora `mergesort` accetta un puntatore a una funzione per la comparazione di due elementi come ulteriore argomento.

Usiamo questa versione di `mergesort` con le seguenti funzioni di comparazione nel file `comp.c`:

```
char smaller(int a, int b)
{
    return a < b;
}

char greater(int a, int b)
{
    return a > b;
}

#include <stdlib.h>
#include <time.h>
char coin(int a, int b)
{
    static char flag = 0;

    if(!flag) { srand(time(NULL)); flag = 1; }
    return rand() % 2;
}
```

Ad esempio, per ordinare in ordine decrescente, invocheremo:

```
mergesort(a,b,l,r,greater);
```

Avvertenze:
Funzioni "corte" come `smaller` avremmo potuto scriverle come macro, ma in questo caso non avremmo avuto alcun puntatore a funzione, e non avremmo potuto invocare `mergesort` con `smaller` come parametro.

Poiché il nome di una funzione è già un puntatore, il C accetta che il prototipo di funzione sia specificato anche come:

```
void mergesort(int *a, int *b, int l, int r, char comp(int, int))
```

Il tipo di un puntatore a funzione è determinato anche dal tipo di ritorno e dalla lista di parametri: `int (*)(void)` è un tipo diverso da `int (*)(int)`.

Si presti attenzione alla sintassi: `int *f(int);` dichiara il prototipo di una funzione che restituisce un puntatore a intero.
`int (*f)(int);` dichiara un puntatore a funzione.

A dispetto della sintassi "poco intutitiva", gli array di puntatori a funzioni spesso sono di grande utilità.

```
int main(void)
{
    int i,e,a[] = { 14, 16, 12, 11, 5, 21, 17, 14, 12 };
    static char (*choice[])(int, int) = { smaller, greater, coin };

    printf("Array da ordinare:");
    for(i = 0; i < 9; i++) printf("%d ",a[i]);
    putchar('\n');

    do {
        printf("Scegli:\n");
        printf("0: ordina in modo crescente\n");
        printf("1: ordina in modo decrescente\n");
        printf("2: rimescola casualmente\n");
        if(!(e =(scanf("%d",&i) && i >= 0 && i <= 2))) {
            printf("\nImmetti o 0 o 1 o 2.\n");
            while(getchar() != '\n');/* per svuotare l'input */
        }
    } while(!e);

    ordina(a,0,8,choice[i]);
    for(i = 0; i < 9; i++) printf("%d ",a[i]);
    putchar('\n');
}
```

`static char (*choice[])(int, int) = { smaller, greater, coin };`
definisce un array di puntatori a funzioni che ritornano `char` e che richiedono due parametri `int`.
Inizializza l'array con i tre puntatori `smaller, greater, coin`.

`char smaller(int, int);`	dichiara il prototipo di funzione
`char (*f)(int, int);`	dichiara un puntatore a funzione
`char (*af[])(int, int);`	dichiara un array di puntatori a funzione

Leggi:

`af[i]`	è l'$i$esimo puntatore a funzione
`(*af[i])`	è una funzione che ritorna `char` e riceve due `int`.

Si può usare `typedef` per semplificare dichiarazioni/definizioni:

```
typedef char (*PTRF)(int, int); /*PTRF e' sinonimo del tipo puntatore a funzione*/
                                /*che ritorna char e prende due int             */
PTRF choice[] = { ... };   /* choice e' il nostro array di puntatori a funzione */
```

Strutture e Unioni

Strutture

Gli array sono tipi costruiti a partire dai tipi fondamentali che raggruppano dati omogenei: ogni elemento dell'array ha lo stesso tipo.
Gli elementi sono acceduti tramite un indice numerico: (`a[i]`).

Con le *strutture* si possono definire tipi che aggregano variabili di tipi differenti.
Inoltre, ogni membro della struttura è acceduto per nome: (`a.nome`).

Per dichiarare strutture si usa la parola chiave `struct`:

```
struct impiegato {
    int id;
    char *nome;
    char *indirizzo;
    short ufficio;
};
```

Il nome di un tipo struttura è costituito dalla parola `struct` seguita dall'etichetta.
Per dichiarare variabili della struttura appena introdotta:

```
struct impiegato e1, e2;
```

La definizione del tipo può essere contestuale alla dichiarazione di variabili del tipo stesso:

```
struct animale {
    char *specie;
    char *famiglia;
    int numeroesemplari;
} panda, pinguino;
```

Una volta definito un tipo struttura lo si può usare per costruire variabili di tipi più complicati: ad esempio array di strutture:

```
struct complex {
    double re, im;
} a[120];              /* dichiara un array di 120 struct complex */
```

I nomi dei membri all'interno della stessa struttura devono essere distinti, ma possono coincidere senza conflitti con nomi di membri di altre strutture (o di normali variabili).

Strutture con etichette diverse sono considerate tipi diversi, anche quando le liste dei membri coincidono.

```
struct pianta {
    char *specie;
    char *famiglia;
    int numeroesemplari;
} begonia;

void f(struct animale a) {}

int main(void) { f(begonia); } /* errore: tipo incompatibile */
```

Accesso ai membri di `struct`

Vi sono due operatori per accedere ai membri di una struttura.

Il primo di questi è l'operatore `.`:

struct_variable . *member_name*

```
panda.numeroesemplari++;   /* e' nato un nuovo panda */
```

La priorità di `.` è massima e associa da sinistra a destra.

```
struct complextriple {
    struct complex x, y, z;
} c;

c.x.re = c.y.re = c.z.im = 3.14;
```

Spesso le variabili dei tipi `struct` vengono manipolate tramite puntatori.

Il secondo operatore per l'accesso ai membri di una struttura (`->`) si usa quando si ha a disposizione un puntatore a una variabile di tipo `struct`:

pointer_to_struct `->` *member_name*

Questo costrutto è equivalente a:

(**pointer_to_struct*) . *member_name*

```
struct complex *add(struct complex *b, struct complex *c)
{
    struct complex *a = malloc(sizeof(struct complex));

    a->re = b->re + c->re;
    a->im = b->im + c->im;
    return a;
}
```

`->` ha la stessa priorità e la stessa associatività dell'operatore . .

La definizione di un nuovo tipo `struct` introduce i nomi dei membri della `struct` stessa: poiché questi devono essere noti al compilatore quando trasforma un file sorgente in formato oggetto, è buona norma collocare la definizione dei tipi `struct` in appositi file d'intestazione, da includere dove necessario.

```
/* file complex.h */

struct complex {
    double re, im;
};

/* file complex .c */
#include "complex.h"

struct complex *add(struct complex *b, struct complex *c)
{
    ...
}
```

Digressione: Uso di `typedef`

L'istruzione `typedef` introduce sinonimi per tipi costruibili in C. Spesso viene usata per semplificare dichiarazioni complesse e/o per rendere più intuitivo l'uso di un tipo in una particolare accezione.

Poiché introducono nuovi nomi di tipi che il compilatore deve conoscere per poter tradurre i file sorgente, le definizioni `typedef` si pongono di norma nei file d'intestazione.

Lo spazio dei nomi usato da `typedef` è diverso dallo spazio delle etichette per i tipi `struct`, quindi si può riutilizzare un'etichetta di un tipo `struct` come nome sinonimo per un tipo:

```
typedef struct complex complex;
```

La sintassi di `typedef` è:

`typedef` *dichiarazione di variabile*

Premettendo `typedef` alla dichiarazione (priva di inizializzazione) di un identificatore si definisce l'identificatore come sinonimo del tipo della dichiarazione.

Esempi d'uso di `typedef`

```
int m[10][10];          /* definisce la variabile m come array di array */
typedef int matrix[10][10]; /* definisce matrix come sinonimo di array di 10 array di 10 int */

typedef char (*PTRF)(int,int); /* definisce un tipo di puntatore a funzione */
PTRF a[10];                    /* definisce un array di quel tipo di puntatori a funzione */

typedef unsigned int size_t;  /* definisce size_t come sinonimo di unsigned int */

typedef struct complex complex; /* definisce complex come sinonimo di struct complex */
complex a;                      /* dichiara una variabile a di tipo struct complex */
```

Digressione: File di intestazione

Quali informazioni sono usualmente contenute nei file d'intestazione `.h` da includere nei file sorgente `.c`:

- definizioni di macro, inclusioni di file d'intestazione.
- definizioni di tipi `struct`, `union`, `enum`.
- `typedef`.
- prototipi di funzione.
- dichiarazioni `extern` di variabili globali.

Digressione: Già che ci siamo:

- I file sorgente `.c` **non** si devono mai includere!
- Per condividere informazioni fra diversi file sorgente `.c`, creare un apposito file di intestazione `.h` e includerlo nei sorgenti. Al resto ci pensa `gcc`: es. `gcc -o nome s1.c s2.c s3.c s4.c` .
- Un file di intestazione `.h` non contiene mai *definizioni* di funzioni, ma solo prototipi.
- Una macro con parametri, per poter essere usata da diversi file sorgenti `.c`, deve essere definita in un file `.h` incluso da questi sorgenti.
- Riducete al minimo necessario le variabili globali nei vostri programmi.
- Riducete la visibilita' di variabili globali e funzioni usate solo in un particolare file `.c` dichiarandole `static`.

In ANSI C è lecito l'assegnamento tra strutture:

```
struct s {
    int dato;
    char a[2000];
};

struct s s1 = { 15, "stringa di prova" }; /* inizializzazione di struct */
struct s s2 = s1;                         /* inizializzazione per copia di s1 */
struct s s3;

s3 = s1;                                  /* assegnamento: copia membro a membro */
```

L'assegnamento di una variabile struttura a un'altra avviene attraverso la copia membro a membro.
Se la struttura contiene un array come membro, anche l'array viene duplicato.
L'assegnamento `s3 = s1;` equivale a:

```
s3.dato = s1.dato;
for(i = 0; i < 2000; i++) s3.a[i] = s1.a[i];
```

In ANSI C, l'assegnamento di strutture ha la semantica della copia membro a membro.
Una situazione analoga si presenta nel passaggio di parametri alle funzioni: variabili di tipo `struct` possono essere passate come parametri alle funzioni e restituite dalle funzioni stesse.
Il passaggio di una variabile `struct` avviene quindi per valore.

```
struct s cambiadato(struct s s1, int dato)
{
    s1.dato = dato;
    return s1;
}
```

Questo è in netto contrasto con la gestione degli array, dove si passano e si assegnano i puntatori ai primi elementi degli array, non gli array stessi.

Quando si passa una variabile struttura a una funzione viene creata una copia locale di ogni membro della variabile stessa, membri di tipo array compresi.

Questo può risultare molto oneroso nei termini dello spazio necessario per memorizzare la copia e nel tempo necessario a realizzare la copia stessa.

Spesso una soluzione migliore prevede il passaggio di parametri di tipo `struct` attraverso puntatori.

```
void cambiadatobis(struct s *s1, int dato) /* s1 e' un puntatore a struct s */
{
    s1->dato = dato;
}
```

Inizializzazione:
Le variabili `struct` di classe `extern` e `static` sono per default inizializzate ponendo tutti i membri a `0`.
Le variabili `struct` possono essere esplicitamente inizializzate (anche quelle di classe `auto`), in maniera analoga al modo in cui si inizializzano gli array.

```
struct struttura {
    int id;
    char *s;
    int dato;
};

struct struttura v = { 15, "pippo" }; /* v.id = 15, v.s = "pippo", v.dato = 0 */
```

`struct` **anonime**:
E' possibile non specificare alcuna etichetta per una `struct`, in tal caso, le uniche variabili dichiarabili di quel tipo `struct` sono quelle dichiarate contemporaneamente alla `struct`:

```
struct {
    int id;
    char *s;
    int dato;
} v, w;
```

ESEMPIO: Rivisitiamo l'implementazione di `stack` e il suo uso nell'applicazione per il controllo della parentesizzazione:

```
/* stk.h */

struct stack {
    int pos;
    int dim;
    char *buf;
};

typedef struct stack stack;

void push(stack *s, char e);
char pop(stack *s);
char empty(stack *s);
stack *createstack(void);
void destroystack(stack *s);
```

Nel file d'intestazione riportiamo la definizione di `stack`, la `typedef` che introduce il nome di tipo `stack`, e i prototipi delle funzioni che manipolano lo `stack`.

Nella struttura `struct stack`:

`char *buf` è un array di caratteri allocato dinamicamente, che memorizzerà il contenuto dello `stack`.

`int dim` è la dimensione corrente dell'array `buf`, che viene allocato `BLOCKSIZE` byte per volta.

`int pos` è l'indice della posizione corrente d'inserimento/cancellazione (la "cima") dello `stack`.

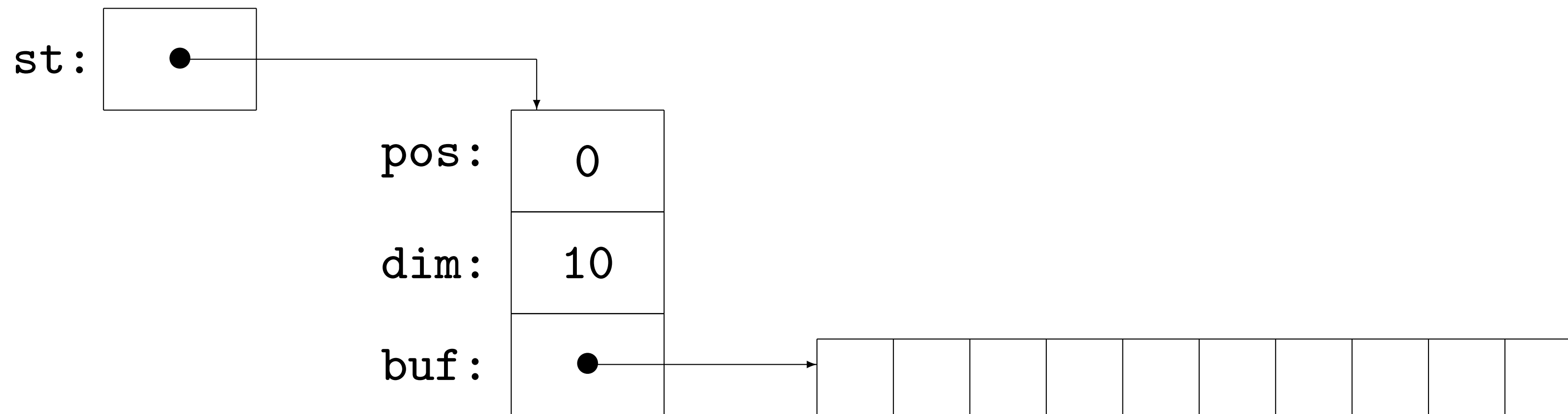
Il file `stk.c` contiene l'implementazione delle funzioni.

```
/* stk.c */

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

```
void manageerror(char *s, short code)
{
    fprintf(stderr,"Errore: %s\n",s);
    exit(code);
}

#define BLOCKSIZE   10

void push(stack *s, char e)
{
    if(s->pos == s->dim) {
        s->dim += BLOCKSIZE;
        if(!(s->buf = realloc(s->buf,s->dim)))
            manageerror("Impossibile riallocare lo stack",1);
    }
    s->buf[s->pos++] = e;
}

char pop(stack *s)
{
    if(s->pos)
        return s->buf[--s->pos];
    else
        return EOF;
}
```

```
char empty(stack *s)
{
    return !s->pos;
}

stack *createstack(void)
{
    stack *s = malloc(sizeof(stack));

    if(!s)
        managerror("Impossibile creare stack",2);
    if(!(s->buf = malloc(s->dim = BLOCKSIZE)))
        manageerror("Impossibile creare stack",3);
    s->pos = 0;
    return s;
}

void destroystack(stack *s)
{
    if(s) {
        free(s->buf);
        free(s);
    }
}
```

```
st = createstack();
```



Abbiamo modificato leggermente anche `paren.c`: eccone il `main`.
(Per compilare `gcc -o paren paren.c stk.c`)

```
int main(void)
{
    int c;
    char d;
    stack *s = createstack();

    printf("Immetti stringa:\n");
    while((c = getchar()) != EOF && c != ';') {
        if(parentesi(c)) {
            if(!empty(s)) {
                if(!chiude(c,d = pop(s))) {
                    push(s,d);
                    push(s,c);
                }
            } else
                push(s,c);
        }
    }
    printf("%s\n", empty(s) ? "Corretto" : "Sbagliato");
    destroystack(s);
}
```

Unioni

Le *unioni* sono strutture "*salva-spazio*".

La sintassi della dichiarazione di una unione è del tutto analoga alla dichiarazione di una struttura, ma si usa la parola chiave `union` al posto di `struct`.

```
union unione {
    int id, dato;
    char array[20];
} u;
```

In una `union` la memoria viene condivisa da ogni membro: nell'esempio la variabile `u` può contenere:
o l'intero `id`, o l'intero `dato`, o l'array di `char` array.

La dimensione di `u` è quella del più grande dei suoi membri.

Il problema di interpretare correttamente il valore attuale di una variabile `union` (vale a dire, a quale membro della `union` riferire il valore), è a carico esclusivo del programmatore.

```
union aux {
    unsigned intero;
    unsigned char cifre[sizeof(unsigned)];
};

int main(void)
{
    short i;
    union aux n;

    printf("Immetti intero\n");
    scanf("%u",&(n.intero));
    for(i = sizeof(unsigned); i; i--)
        printf("%u,",n.cifre[i-1]);
    putchar('\n');
}
```

Cosa fa questo programma ?

Il programmatore deve gestire esplicitamente l'informazione riguardante quale sia il membro di una variabile `union` con cui interpretare il valore della variabile stessa.
Spesso questa informazione è memorizzata insieme alla `union`:

```
struct elemento {
    char tipo;  /* 0: i, 1: c, 2: d, 3: p */
    union {
        int i;
        char c;
        double d;
        int *p;
    } dato;
};

void f(struct elemento *e) {
    switch(e->tipo) {
    case 0: useint(e->dato.i); break;
    case 1: usechar(e->dato.c); break;
    case 2: usedouble(e->dato.d); break;
    case 3: useptr(e->dato.p); break;
    }
}
```

Strutture che contengono strutture:

Se la definizione di un tipo `struct` contiene elementi di un altro tipo `struct`, allora la definizione di quest'ultimo tipo deve essere già nota al compilatore.

```
struct a {
    ...
    struct b sb;  /* ERRORE: struct b tipo sconosciuto */
};

struct b {
    ...
};
```

La definizione di `struct b` deve precedere quella di `struct a`:

```
struct b {
    ...
};

struct a {
    ...
    struct b sb;  /* OK */
};
```

Strutture autoreferenziali

Poiché la definizione di un tipo `struct` deve essere nota per potere includere membri di quel tipo in altri tipi `struct`, non è possibile che un tipo `struct` contenga membri dello stesso tipo:

```
struct r {
    ...
    struct r next; /* ERRORE: struct r tipo sconosciuto */
};
```

Ma poiché i tipi puntatori a dati hanno tutti la stessa dimensione in memoria, è possibile includere membri di tipo puntatore al tipo `struct` che stiamo dichiarando:

```
struct r {
    ...
    struct r *next; /* OK: next e' puntatore a struct r */
};
```

Le strutture autoreferenziantisi tramite puntatori sono lo strumento principe del C per costruire strutture dati dinamiche.

Strutture con riferimenti mutui

Usando i puntatori a tipi `struct` è possibile la mutua referenza:

```
struct a {
    ...
    struct b *pb;
};

struct b {
    ...
    struct a *pa;
};
```

In questo caso, `struct b` avrebbe potuto contenere una variabile di tipo `struct a` come membro, al posto del puntatore `struct a *pa`, poiché `struct b` conosce la definizione di `struct a`, mentre `struct a` non conosce la definizione di `struct b`.

Liste, Alberi, Grafi

Liste concatenate

La lista concatenata è una struttura che organizza i dati in maniera sequenziale.

Mentre gli elementi di un array sono accessibili direttamente, gli elementi di una lista devono essere acceduti sequenzialmente: se voglio accedere all'$i$esimo elemento devo scandire la lista dal primo all'$(i-1)$esimo elemento.

Mentre le dimensioni di un array sono rigide, le dimensioni di una lista variano dinamicamente, tramite le operazioni di inserzione e cancellazione di elementi.

Elementi logicamente adiacenti in una lista possono essere allocati in posizioni di memoria non adiacenti.

L’implementazione di liste in C può avvenire in vari modi: il modo standard prevede l’uso di strutture e puntatori.

Ecco la definizione del tipo elemento di una lista di interi.

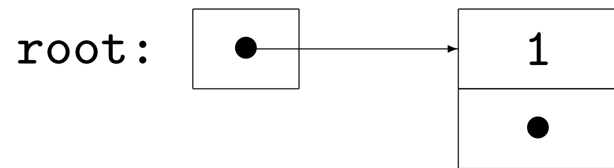
```
struct intlist {
    int dato;
    struct intlist *next;
};
```

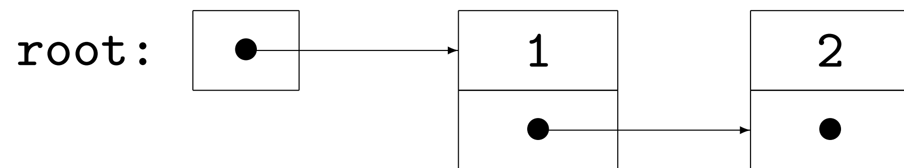
Una siffatta lista deve essere acceduta tramite un puntatore al primo elemento della lista:

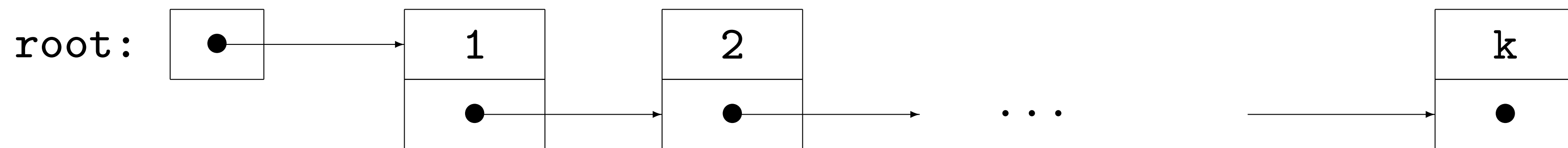
```
struct intlist *root = NULL; /* root punta a una lista vuota */

root = malloc(sizeof(struct intlist));
root->dato = 1;
root->next = malloc(sizeof(struct intlist));
root->next->dato = 2;
root->next->next = NULL; /* ora root punta a una lista di due elementi */
```

L’ultimo nodo della lista deve avere il suo campo `next` posto a `NULL`.

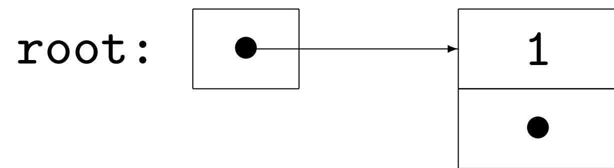
root:

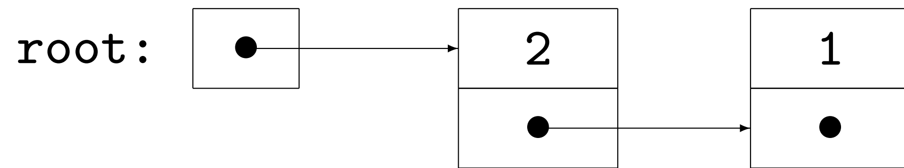

root:
1

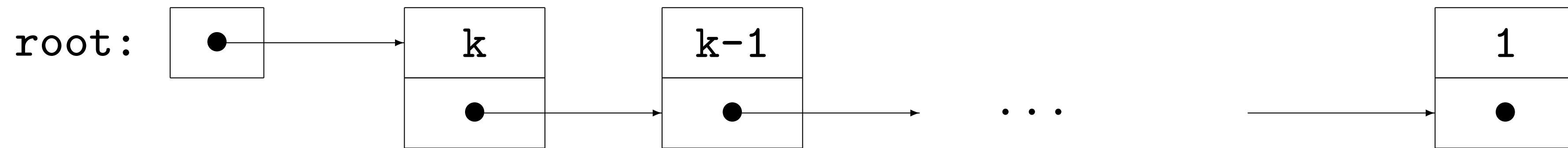

root:
1
2


root:
1
2
...
k

Con l'inserimento in testa ...







Operazioni sulle liste

Creiamo i file `intlist.h` e `intlist.c`.

In primo luogo introduciamo:

```
typedef struct intlist intlist;
```

Le operazioni: **Creazione**:
Con l'implementazione presentata la funzione per creare una lista è banale:

```
/* crea una lista vuota e ne restituisce il puntatore radice */
intlist *createlist(void)
{
    return NULL;
}
```

Se rivedessimo la nostra implementazione, potremmo dover fare operazioni più complesse per creare una lista inizialmente vuota: in tal caso riscriveremmo la funzione `createlist`.

Attraversamento:
L'attraversamento si ottiene seguendo i puntatori `next` fino a raggiungere il valore `NULL`.

```
/* visita una lista e esegue su ogni elemento la funzione op */
void traverse(intlist *p, void (*op)(intlist *))
{
    intlist *q;

    for(q = p; q; q = q->next)
        (*op)(q);
}
```

Esempio: per stampare il contenuto della lista puntata da `root`:

```
/* stampa l'elemento puntato */
void printelem(struct intlist *q)
{
    printf("\t-------\n\t|%5d|\n\t-------\n\t|  %c  |\n\t---%c---\n\t",
            q->dato, q->next ? '.' : 'X', q->next ? '|' : '-');
    if(q->next)
        printf("   |   \n\t   V\n");
}
```

Si richiama: `traverse(root,printelem);`

Inserimento in testa:
L'inserimento di un elemento in testa alla lista richiede solo l'aggiornamento di un paio di puntatori:

```
/* inserisce un elemento in testa alla lista */
intlist *insert(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    q->next = p;
    return q;
}
```

Per inserire un elemento di valore 15 in testa alla lista puntata da `root`:

`root = insert(root,15);`

Cancellazione:
La cancellazione di un elemento da una lista, con questa implementazione, risulta più complessa dell'inserzione, richiedendo la scansione della lista.

```
/* cancella l'elemento puntato da q dalla lista */
intlist *delete(intlist *p, intlist *q)  /* si assume q != NULL */
{
    intlist *r;

    if(p == q)
        p = p->next;
    else {
        for(r = p; r && r->next != q; r = r->next);
        if(r && r->next == q)
            r->next = r->next->next;
    }
    free(q);
    return p;
}
```

Per cancellare un elemento: `root = delete(root, q);`

Ricerca di un elemento:
Anche la ricerca del primo elemento di valore dato richiede la scansione della lista:

```
/* restituisce il primo elemento per cui check e' vera oppure NULL */
intlist *getcheckelem(intlist *p, char (*check)(intlist *, int), int a)
{
    intlist *q;

    for(q = p; q; q = q->next)
        if((*check)(q,a))
            return q;
    return NULL;
}
/* restituisce il primo elemento q per cui q->dato == elem */
intlist *geteleminlist(intlist *p, int elem) { return getcheckelem(p,checkexist,elem); }

char checkexist(intlist *p, int elem) { return p->dato == elem; }
```

Per cancellare il primo elemento di valore 25:

```
intlist *p = geteleminlist(root, 25);
root = delete(root,p);
```

Questo richiede due scansioni della lista: Esercizio: migliorare.

Distruzione della lista:
Si tratta di liberare tutta la memoria allocata alla lista:

```
/* distrugge la lista */
void destroylist(intlist *p) /* si assume p != NULL */
{
    while(p = delete(p,p));
}
```

Concatenazione di due liste:
Si scandisce la prima lista fino all'ultimo elemento, poi si collega il primo elemento della seconda all'ultimo elemento della prima.

```
/* concatena la lista con radice q alla lista con radice p */
intlist *listcat(intlist *p, intlist *q)
{
    intlist *r;

    if(!p)
        return q;
    for(r = p; r->next; r = r->next);
    r->next = q;
    return p;
}
```

Conteggio del numero di elementi:

```
/* ritorna il numero di elementi nella lista */
int countlist(intlist *p)
{
    int i;

    for(i = 0; p; p = p->next, i++);
    return i;
}
```

Inserimento di elementi in ordine:

```
/* inserisce un elem nella lista prima del primo elemento >= di elem */
intlist *insertinorder(intlist *p, int elem)
{
    intlist *q;

    if(!p || p->dato >= elem)
        return insert(p, elem);
    for(q = p; q->next && q->next->dato < elem; q = q->next);
    q->next = insert(q->next, elem);
    return p;
}
```

Provare l'esempio: `gcc -o list1 list1.c intlist.c`

Poiché la lista è una struttura dati definita ricorsivamente, è possibile riformulare le funzioni che effettuano una scansione dandone un'implementazione ricorsiva.

```
/* ritorna il numero di elementi nella lista: versione ricorsiva */
int rcountlist(intlist *p)
{
    return p ? rcountlist(p->next) + 1 : 0;
}

/* visita una lista e esegue su ogni elemento la funzione op: versione ricorsiva */
void rtraverse(intlist *p, void(*op)(intlist *))
{
    if(p) {
        (*op)(p);
        rtraverse(p->next,op);
    }
}

/* concatena la lista con radice q alla lista con radice p: versione ricorsiva */
intlist *rlistcat(intlist *p, intlist *q)
{
    if(p)
        p->next = rlistcat(p->next,q);
    return p ? p : q;
}
```

Modifichiamo la nostra implementazione di `intlist` aggiungendo un puntatore `prev` all'elemento precedente:

```
struct intlist {
    int dato;
    struct intlist *next, *prev;
};
```

In questo modo alcune funzioni di manipolazione risultano semplificate:

```
/* cancella l'elemento puntato da q dalla lista */
intlist *delete(intlist *p, intlist *q) /* si assume q != NULL */
{
    if(q->prev)
        q->prev->next = q->next;
    else
        p = q->next;
    if(q->next)
        q->next->prev = q->prev;
    free(q);
    return p;
}
```

Cancelliamo il primo elemento di valore 25:

```
intlist *p = geteleminlist(root, 25);

root = delete(root,p);
```

La funzione `geteleminlist` richiede una scansione lineare, mentre `delete` lavora in tempo costante.

La funzione `insert` va anch'essa modificata, ma continuerà a lavorare in tempo costante.

```
/* inserisce un elemento in testa alla lista */
intlist *insert(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    if(q->next = p)
        p->prev = q;
    q->prev = NULL;
    return q;
}
```

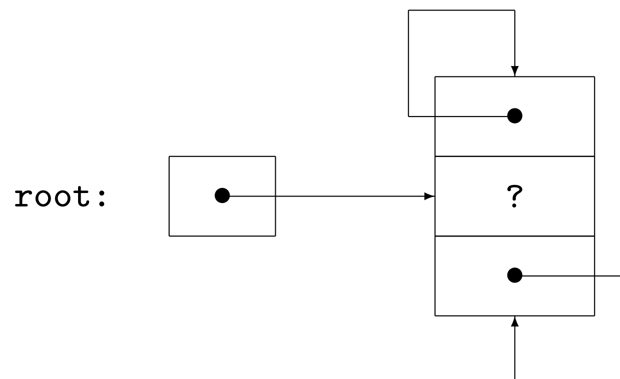
Un'ulteriore semplificazione si ottiene utilizzando un elemento *sentinella* (*dummy*) che non contiene informazione, ma serve a segnalare la fine (e l'inizio) di una lista.

Questa soluzione va contemplata quando le lunghezze stimate dalle liste usate sono significativamente maggiori delle dimensioni di un elemento.

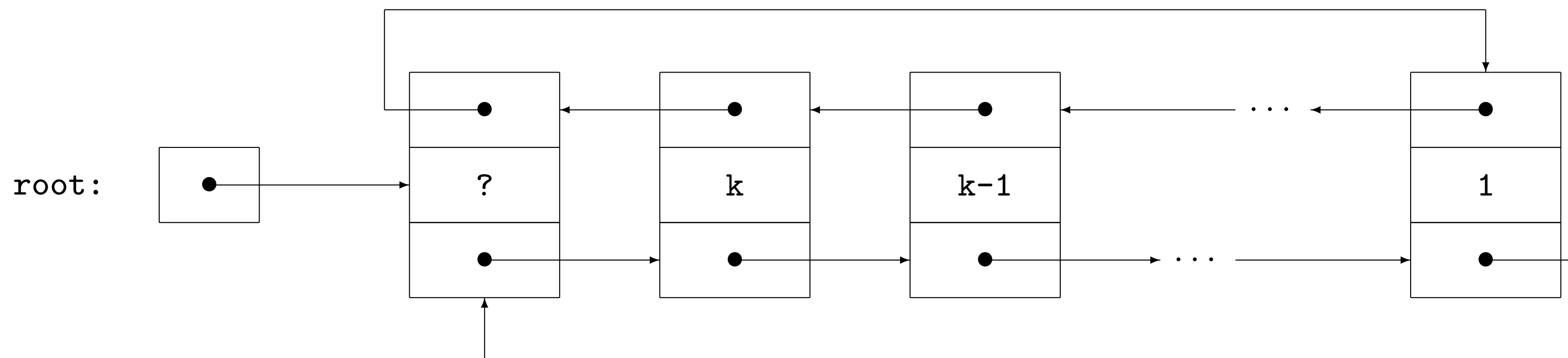
Bisogna in primo luogo allocare la sentinella al momento della creazione della lista.

```
intlist *createlist(void)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore di allocazione nella creazione della lista\n");
        exit(-1);
    }
    q->next = q->prev = q;
    return q;
}
```

```
root = createlist(); /* implementazione con sentinella */
```



dopo aver inserito in testa gli elementi $1, \ldots, k$:



Con l'introduzione della sentinella, inserimento e cancellazione diventano:

```
void insert(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    q->next = p->next;
    p->next->prev = q;
    p->next = q;
    q->prev = p;
}

void delete(intlist *q)
{
    q->prev->next = q->next;
    q->next->prev = q->prev;
    free(q);
}
```

Anche altre funzioni sono state modificate:

```
void listcat(intlist *p, intlist *q)
{
    if(q != q->next) {
        if(p == p->next) {
            p->next = q->next;
            q->next->prev = p;
        } else {
            p->prev->next = q->next;
            q->next->prev = p->prev;
        }
        p->prev = q->prev;
        q->prev->next = p;
    }
    free(q);
}
```

La funzione `listcat` non attua più la scansione per raggiungere la fine della prima lista, ma lavora in tempo costante.

L'inserimento in coda alla lista risulta facile come l'inserimento in testa:

```
/* inserisce un elemento in coda alla lista */
void insertatend(intlist *p, int elem)
{
    intlist *q = malloc(sizeof(intlist));

    if(!q) {
        fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n");
        exit(-1);
    }
    q->dato = elem;
    q->prev = p->prev;
    p->prev->next = q;
    p->prev = q;
    q->next = p;
}
```

Si consideri questa implementazione con l'esempio rivisitato:

`gcc -o list3 list3.c intlistdummy.c`

Vi sono ancora funzioni che scandiscono la lista inutilmente:

```
int countlist(intlist *p)
{
    int i;
    intlist *q;

    if(p == p->next)
        return 0;
    for(i = 1, q = p->next; q->next != p; q = q->next, i++);
    return i;
}
```

In questo caso, per rimediare possiamo memorizzare l'informazione sul numero di elementi in un tipo `struct` che contiene anche il puntatore al primo elemento:

```
struct intlistheader {
    intlist *root;
    int count;
};
```

Per esercizio: modificare le funzioni di manipolazione di liste per implementare questa versione.

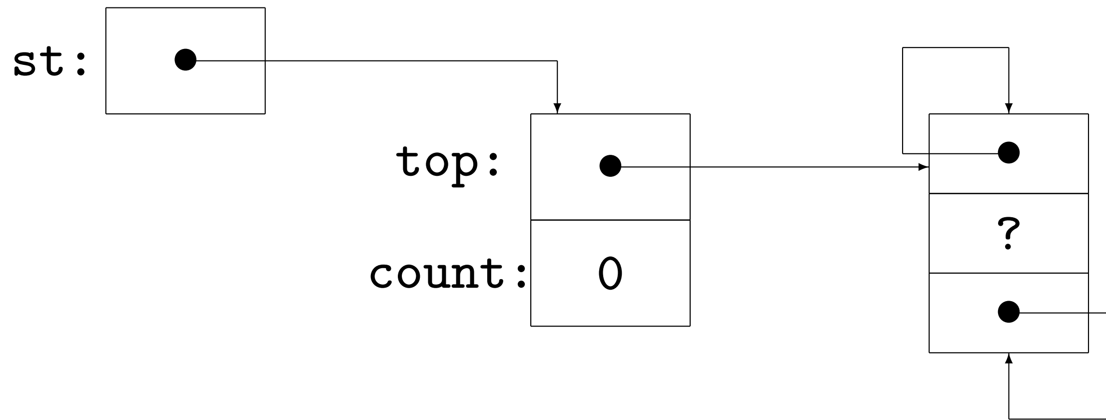
Poiché gli *stack* sono collezioni di dati per i quali l'accesso può avvenire solo attraverso la politica LIFO (Last In, First Out) è facile implementare stack attraverso liste concatenate:

```
struct intstack {
    intlist *top;
    int count;
};

typedef struct intstack intstack;

intstack *createstack(void)
{
    intstack *st = malloc(sizeof(intstack));

    if(!st) {
        fprintf(stderr,"Errore di allocazione nella creazione dello stack\n");
        exit(-1);
    }
    st->top = createlist();
    st->count = 0;
    return st;
}
```

Usiamo l'implementazione di `intlist` con sentinella che è contenuta in `intlistdummy.c`

```
void push(intstack *st, int elem)
{
    insert(st->top, elem);
    st->count++;
}

int pop(intstack *st)
{
    int e;

    if(!st->count){
        fprintf(stderr,"Errore: pop su stack vuoto\n");
        exit(-2);
    }
    e = head(st->top);
    deletehead(st->top);
    st->count--;
    return e;
}

char empty(intstack *st)
{
    return !st->count;
}
```

```
st = createstack();
```



Una *coda* (*queue*) è un tipo di dati astratto analogo allo *stack*.

Le code seguono una politica FIFO (First In, First Out).

Le code si possono implementare attraverso liste concatenate, a patto di avere un'implementazione in cui l'inserimento in fondo alla lista sia efficiente.

Usiamo ancora l'implementazione contenuta in `intlistdummy.c`

```
struct intqueue {
    intlist *head;
    int count;
};

typedef struct intqueue intqueue;
```

L'implementazione di `intqueue` è quasi identica a quella di `intstack`:
Solo l'inserimento di dati in `intqueue` differisce:

Si confrontino `push` ed `enqueue`:

```
void push(intstack *st, int elem)
{
    insert(st->top, elem);
    st->count++;
}

void enqueue(intqueue *q, int elem)
{
    insertatend(q->head, elem);
    q->count++;
}
```

Proviamo l'esempio di confronto:

```
gcc -o cfrstq cfrstq.c intstack.c intqueue.c intlistdummy.c
```

Alberi

La definizione in termini di teoria dei grafi:
Un grafo (non orientato) senza cicli e connesso è detto *albero*.
Un albero *radicato* è una coppia $< T, r >$ dove T è un albero e r è un suo vertice, detto *radice*.

La definizione ricorsiva:
Un albero radicato (non vuoto) è:
– o un singolo nodo
– o una radice connessa a un insieme di alberi, dove ogni albero è connesso tramite un unico arco alla radice.

Dalla semplicità della definizione ricorsiva consegue la semplicità di scrittura di algoritmi ricorsivi per la manipolazione di alberi.

Mentre **array, liste, stack, code** sono strutture dati intrinsecamente *monodimensionali* (un elemento ne segue un altro), gli **alberi** strutturano l'informazione in modo più complesso.

La modellizzazione dell'informazione tramite alberi è molto diffusa e spesso naturale e intuitiva:

– gerarchie, genealogie, organigrammi, tornei, espressioni, frasi

Moltissimi algoritmi usano gli alberi per strutturare i dati.

La soluzione più comune per l'implementazione di alberi in C è attraverso l'uso di strutture autoreferenziali tramite puntatori.

Alberi binari

Un *albero binario* è un albero radicato in cui ogni nodo interno ha al più due figli. Ogni figlio è distinto come figlio *sinistro* oppure figlio *destro*.

Definiamo la struttura corrispondente a un vertice (*nodo*) di un albero di elementi di tipo `int`:

```
struct inttree {
    int dato;
    struct inttree *left, *right;
};

typedef struct inttree inttree;
```

La *radice* di un albero con siffatti nodi sarà un puntatore a `inttree`. Ecco un albero vuoto:

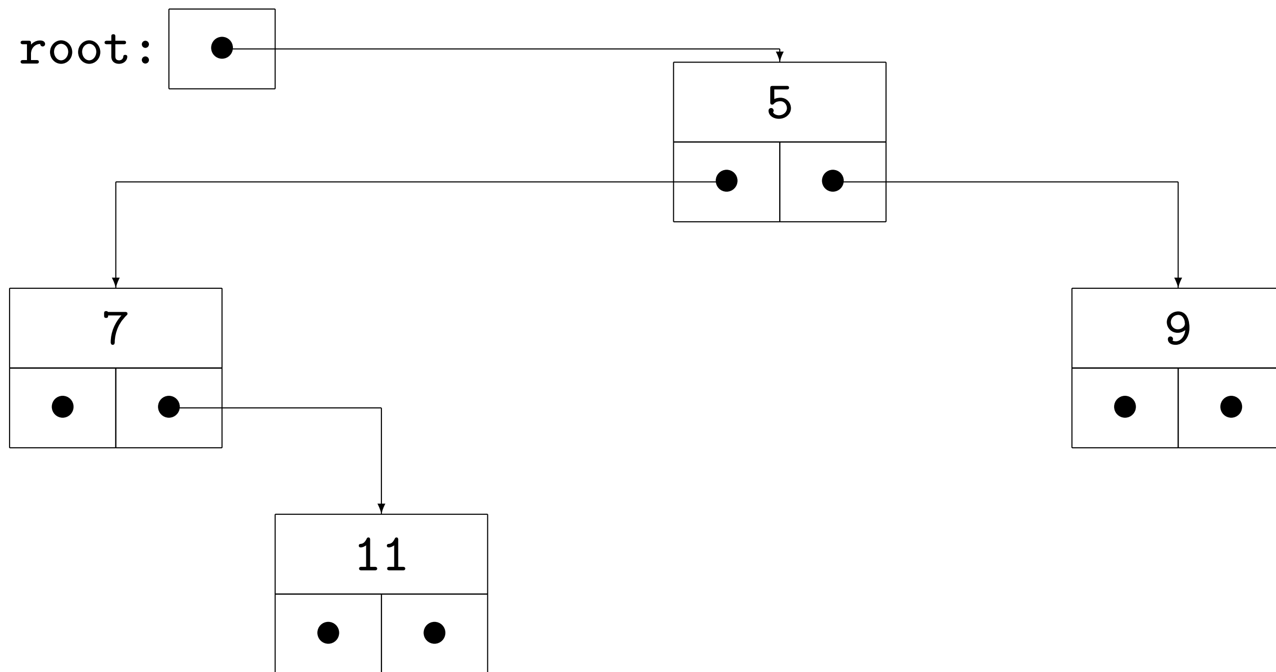
```
inttree *root = NULL;
```

I campi `left` e `right` della struttura `inttree` sono le radici dei sottoalberi sinistro e destro del nodo costituito dall'istanza della struttura.

Se un nodo non ha figlio sinistro (resp. destro), il suo campo `left` (resp. `right`) sarà posto a `NULL`.

Costruiamo manualmente un albero la cui radice sarà `root`:

```
root = malloc(sizeof(inttree));
root->dato = 5;
root->left = malloc(sizeof(inttree));
root->right = malloc(sizeof(inttree));
root->left->dato = 7;
root->right->dato = 9;
root->left->left = root->right->left = root->right->right = NULL;
root->left->right = malloc(sizeof(inttree));
root->left->right->dato = 11;
root->left->right->left = root->left->right->right = NULL;
```


root:
5
7
9
11

Attraversamento di alberi:
Un albero può essere attraversato in vari modi.
I tre modi più comuni hanno una semplice e diretta implementazione ricorsiva:

Visita in ordine **simmetrico** (**inorder**):
– sottoalbero sinistro
– radice
– sottoalbero destro

Visita in ordine **anticipato** (**preorder**):
– radice
– sottoalbero sinistro
– sottoalbero destro

Visita in ordine **posticipato** (**postorder**):
– sottoalbero sinistro
– sottoalbero destro
– radice

Visita **inorder**

```
void inorder(inttree *p)
{
    if(p) {
        inorder(p->left);
        dosomething(p);
        inorder(p->right);
    }
}
```

Visita **preorder**

```
void preorder(inttree *p)
{
    if(p) {
        dosomething(p);
        preorder(p->left);
        preorder(p->right);
    }
}
```

Visita **postorder**

```
void postorder(inttree *p)
{
    if(p) {
        postorder(p->left);
        postorder(p->right);
        dosomething(p);
    }
}
```

Per gestire in modo parametrico `dosomething(p)` almeno per funzioni di tipo `void f(inttree *p)` implementiamo gli attraversamenti con un secondo parametro di tipo puntatore a funzione:

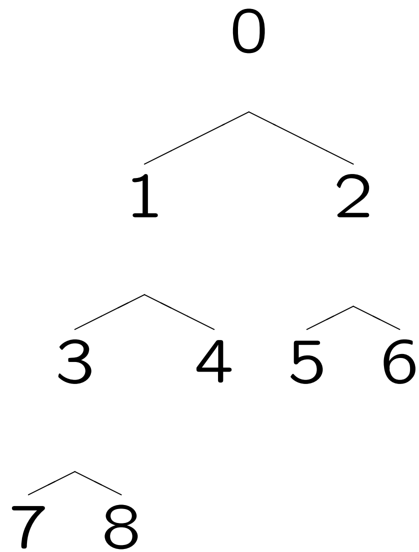
```
void postorder(inttree *p, void (*op)(inttree *))
{
    if(p) {
        postorder(p->left,op);
        postorder(p->right,op);
        (*op)(p);
    }
}
```

Costruzione di un albero binario a partire da un array

Un array di interi può essere visto come un albero binario in cui il figlio sinistro dell'elemento i-esimo è memorizzato nell'elemento di indice $2i+1$ e analogamente l'indice del figlio destro è $2i+2$.

```
inttree *buildtree(int *array, int i, int size)
{
    inttree *p;

    if(i >= size)
        return NULL;
    else {
        if(!(p = malloc(sizeof(inttree)))) {
            fprintf(stderr,"Errore di allocazione\n");
                exit(-1);
        }
        p->left = buildtree(array, 2 * i + 1, size);
        p->right = buildtree(array, 2 * i + 2, size);
        p->dato = array[i];
        return p;
    }
}
```

```
int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
```



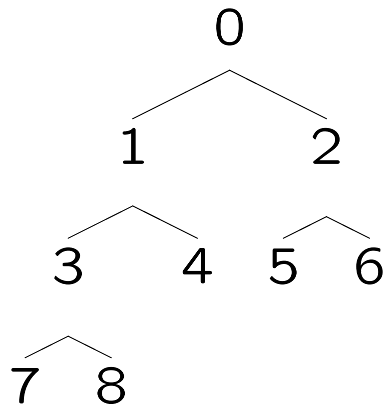
Esercizio: stampa di un albero

Vogliamo visualizzare la struttura dell'albero a video:

Poiché stampiamo una riga per volta, a ogni riga facciamo corrispondere un livello dell'albero.
Occorre stabilire:
– per ogni livello dell'albero quali nodi vi occorrono
– dove posizionare sulla riga questi nodi
La posizione del nodo sulla riga corrisponde al posto che il nodo ha nell'ordine simmetrico.

Utilizziamo un array di liste, indicizzato dai livelli per memorizzare le posizioni di stampa di ogni nodo.

Un elemento dell'i-esima lista contiene un puntatore a un nodo di livello i e la posizione di stampa per quel nodo.



					0			
			1				2	
	3			4		5		6
7		8						

Per ogni elemento, la sua collocazione (i, j) nella matrice dello schermo è così determinata:
- la riga i coincide con la profondità (livello) del nodo. Useremo una visita level-order per ottenere e gestire tale dato.
- il valore j per la colonna è stabilito dalla visita in-order dell'albero: l'elemento in esame è visitato per $(j + 1)$-esimo.

```
struct spaces {
    int col;
    inttree *ptr;
    struct spaces *prev, *next;
};

struct spaces *createlist(void)
{
    struct spaces *q = malloc(sizeof(struct spaces));

    if(!q) {
        fprintf(stderr,"Errore di allocazione nella creazione della lista\n");
        exit(-1);
    }
    q->next = q->prev = q;
    return q;
}

void delete(struct spaces *q)
{
    q->prev->next = q->next;
    q->next->prev = q->prev;
    free(q);
}
```

```
void destroylist(struct spaces *p)
{
    while(p->next != p) delete(p->next);
    free(p);
}

void insertatend(struct spaces *p, inttree *ptr, int col)
{
    struct spaces *q = malloc(sizeof(struct spaces));

    if(!q) { fprintf(stderr,"Errore nell'allocazione del nuovo elemento\n"); exit(-1); }
    q->ptr = ptr; q->col = col;
    q->prev = p->prev; p->prev->next = q; p->prev = q; q->next = p;
}

void levelvisit(inttree *p, int lev, struct spaces **sp)
{
    static int count = 0;

    if(!lev) count = 0;
    if(p) {
        levelvisit(p->left,lev + 1,sp);
        insertatend(sp[lev],p,count++);
        levelvisit(p->right,lev + 1,sp);
    }
}
```

```
void printtree(inttree *p)
{
    int lev,i,j;
    struct spaces *q;

    struct spaces **sp = calloc(lev = countlevels(p),sizeof(struct spaces *));

    for(i = 0; i < lev; i++)
        sp[i] = createlist();
    levelvisit(p,0,sp);
    for(i = 0; i < lev; i++) {
        j = 0;
        for(q = sp[i]->next;q != sp[i];q = q->next,j++) {
            for(;j < q->col;j++)
                printf("    ");
            printf("[%2d]",q->ptr->dato);
        }
        putchar('\n');
    }
    for(i = 0; i < lev; i++)
        destroylist(sp[i]);
    free(sp);
}
```

Commenti:

```
struct spaces {
    int col;
    inttree *ptr;
    struct spaces *prev, *next;
};
```

Creiamo un array di liste i cui elementi sono di tipo `struct spaces`:
`col` memorizza la colonna in cui stampare il dato puntato da `ptr`.
Abbiamo adattato le funzioni per l'implementazione di liste bidirezionali di interi (`intlistdummy.c`) al tipo `struct spaces`.

```
void levelvisit(inttree *p, int lev, struct spaces **sp)
{
    static int count = 0;

    if(p) {
        levelvisit(p->left,lev + 1,sp);
        insertatend(sp[lev],p,count++);
        levelvisit(p->right,lev + 1,sp);
    }
}
```

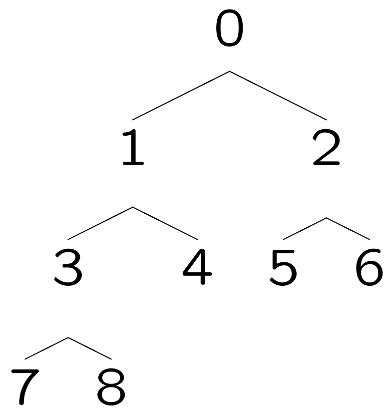
Usiamo una visita inorder dell'albero per assegnare le posizioni `col` dei nodi nella lista ausiliaria, *livello per livello*. La variabile `static int count` memorizza la posizione in ordine simmetrico del nodo correntemente visitato.

Per l'implementazione di `printtree.c` abbiamo utilizzato due semplici funzioni ricorsive, collocate in `inttree.c`, per il conteggio di nodi e livelli di un `inttree`:

```
int countnodes(inttree *p)
{
    return p ? countnodes(p->left) + countnodes(p->right) + 1 : 0;
}

#define max(a,b)    ((a) > (b) ? (a) : (b))

int countlevels(inttree *p)
{
    return p ? max(countlevels(p->left),countlevels(p->right)) + 1 : 0;
}
```

Notare che la parentesizzazione nella definizione della macro `max` è necessaria per un uso corretto della macro stessa.

Esercizio: Se ci accontentiamo di visualizzare l'albero in modo trasposto:



			7
		3	
			8
	1		
		4	
0			
		5	
	2		
		6	

allora, non c'e' bisogno di memorizzare la riga e la colonna dove stampare ogni elementi. Non appena si è in grado di calcolare riga e colonna di un elemento, tale elemento può essere stampato. Perché?

Implementazione non ricorsiva

Utilizzando esplicitamente uno stack, invece di affidarsi allo stack dei record di attivazione gestito dal sistema C, è possibile riformulare le funzioni ricorsive precedenti in modo iterativo.

```
void preorderit(inttree *p,void (*op)(inttree *))
{
    struct astack *s = createastack();

    apush(s,p);
    while(!aempty(s)) {
        p = apop(s);
        (*op)(p);
        if(p->right)
            apush(s,p->right);
        if(p->left)
            apush(s,p->left);
    }
    destroyastack(s);
}
```

Sostituendo lo stack `astack` con la corrispondente implementazione di coda `aqueue`, è immediato implementare la visita *level–order* dell'albero:

```
void levelorderit(inttree *p,void (*op)(inttree *))
{
    struct aqueue *s = createaqueue();

    aenqueue(s,p);
    while(!aemptyq(s)) {
        p = adequeue(s);
        (*op)(p);
        if(p->left)
            aenqueue(s,p->left);
        if(p->right)
            aenqueue(s,p->right);
    }
    destroyaqueue(s);
}
```

(Vedi `inttreeit.c` per la definizione di `struct astack`, etc.)

Alberi Binari di Ricerca

Gli alberi binari di ricerca (o, **alberi di ricerca binaria**) sono strutture dati che consentono, su un insieme di elementi con un ordine totale le operazioni di:

– **ricerca** di un elemento, **verifica** dell'appartenenza di un elemento a un sottoinsieme.
– **inserimento** e **cancellazione** di elementi.
– reperimento del **minimo** e del **massimo** elemento di un sottoinsieme.
– determinazione del **successore** e del **predecessore** di un elemento in un sottoinsieme.

Definizione:

Sia $< A, \leq >$ un insieme totalmente ordinato e sia $S \subseteq A$.

Un **albero di ricerca binaria** per S è un albero binario T i cui nodi sono etichettati da elementi di S.
Inoltre, detta $E(v)$ l'etichetta del nodo v di T valgono le seguenti:

– per ogni elemento s di S esiste uno e un solo v tale che $E(v) = s$.
– per ogni nodo v di T, se u appartiene al sottoalbero destro di v, allora $E(v) < E(u)$.
– per ogni nodo v di T, se u appartiene al sottoalbero sinistro di v, allora $E(v) > E(u)$.

Implementazione

Assumiamo che l'insieme A delle etichette (*chiavi*) siano gli interi.

Collochiamo nel file `searchtree.h` la seguente `typedef`

```
typedef int key;
```

Rappresenteremo i nodi dell'albero utilizzando una `struct` con tre campi puntatore: oltre ai puntatori `left` e `right` ai sottoalberi sinistro e destro, memorizzeremo nel puntatore `up` l'indirizzo del nodo *padre*.

```
struct searchtree {
    key v;
    struct searchtree *left, *right, *up;
};

typedef struct searchtree searchtree;
```

Ovviamente, oltre alla chiave, un nodo può contenere altre informazioni, che possono essere rappresentate aggiungendo nuovi campi membro o campi puntatore alla struttura `searchtree`.

Ad esempio, se ogni nodo deve essere associato a una stringa `nome`, modificheremo come segue la nostra definizione della struttura `searchtree`:

```
struct searchtree {
    key v;
    char *nome;
    struct searchtree *left, *right, *up;
};
```

NOTA: l'implementazione discussa gestisce "senza problemi" (quando l'unica cosa da gestire è il valore della chiave) alberi binari di ricerca in cui i valori per il campo `key v` possano trovarsi ripetuti in nodi distinti.

Operazioni sugli alberi binari di ricerca

Essendo a tutti gli effetti degli alberi binari, l'implementazione delle visite **inorder**, **preorder**, **postorder**, **level-order** può essere mutuata dalle corrispondenti funzioni per alberi binari generici:

```
void inorder(searchtree *p, void (*op)(searchtree *))
{
    if(p) {
        inorder(p->left,op);
        (*op)(p);
        inorder(p->right,op);
    }
}
```

La visita in ordine simmetrico di un albero di ricerca binaria produce un elenco ordinato delle etichette (*chiavi*) dell'albero.

Chiaramente, la visita in ordine di un albero con n nodi richiede $\Theta(n)$ passi.

Ricerca di una chiave in un albero:

```
searchtree *search(searchtree *p, key k)
{
    if(!p || k == p->v)
        return p;
    return search(k < p->v ? p->left : p->right, k);
}
```

`search` ritorna `NULL` se l'albero con radice `p` non contiene alcun nodo con chiave `k`, altrimenti ritorna un puntatore al (nodo più a sinistra) con chiave `k`.

`search` sfrutta in modo essenziale le proprietà definitorie di un albero di ricerca binaria: ricorsivamente viene esaminato il sottoalbero sinistro se `k < p->v`, il sottoalbero destro altrimenti.

Il tempo di esecuzione è $O(h)$ dove h è l'altezza dell'albero.

Un esercizio di ripasso sull'operatore ternario `?  :`
Versioni alternative di `search`:

```
searchtree *search2ndvers(searchtree *p, key k)
{
    if(!p || k == p->v)
        return p;
    return k < p->v ? search2ndvers(p->left,k) : search2ndvers(p->right,k);
}

searchtree *search3rdvers(searchtree *p, key k)
{
    if(!p || k == p->v)
        return p;
    if(k < p->v)
        return search3rdvers(p->left,k);
    else
        return search3rdvers(p->right,k);
}
```

Eccone una versione iterativa:

```
searchtree *itsearch(searchtree *p, key k)
{
    while(p && k != p->v)
        p = k < p->v ? p->left : p->right;
    return p;
}
```

Anche le operazioni di estrazione del **minimo** e del **massimo** elemento richiedono $O(h)$ passi, per h l'altezza dell'albero:

Semplicemente, per trovare il minimo si seguono solo i puntatori `left`:

```
searchtree *treemin(searchtree *p)   /* si assume p != NULL */
{
    for(;p->left;p = p->left);
    return p;
}
```

mentre per trovare il massimo si seguono solo i puntatori `right`:

```
searchtree *treemax(searchtree *p)  /* si assume p != NULL */
{
    for(;p->right;p = p->right);
    return p;
}
```

Successore e **Predecessore**
Per trovare il **successore** di una chiave in un albero, non è necessario passare la radice dell'albero stesso alla funzione: l'unico parametro è un puntatore al nodo di cui si vuole ottenere il successore:

```
searchtree *treesucc(searchtree *q) /* si assume q != NULL */
{
    searchtree *qq;

    if(q->right)
        return treemin(q->right);
    qq = q->up;
    while(qq && q == qq->right) {
        q = qq;
        qq = qq->up;
    }
    return qq;
}
```

`treesucc` restituisce il puntatore al nodo successore oppure `NULL` se tale nodo non esiste.

Il codice di `treesucc` è suddiviso in due casi:

```
if(q->right)
    return treemin(q->right);
```

Se il sottoalbero destro di `q` non è vuoto, il successore di `q` è il nodo più a sinistra del sottoalbero destro: tale nodo viene restituito da `treemin(q->right)`.

```
qq = q->up;
while(qq && q == qq->right) {
    q = qq;
    qq = qq->up;
}
return qq;
```

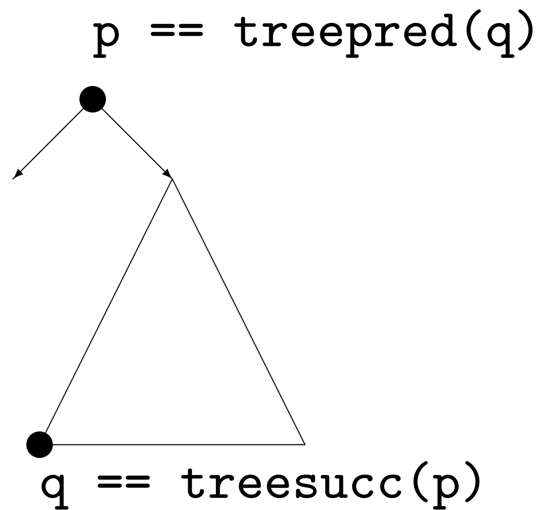
Altrimenti, se il successore esiste, allora deve essere l'antenato più "giovane" di `q` il cui figlio sinistro è pure antenato di `q`. Per determinarlo, `qq` punta al padre di `q`; la scansione fa risalire `q` e `qq`, fino a quando non si trova un antenato con le proprietà desiderate: essere figlio sinistro del padre.

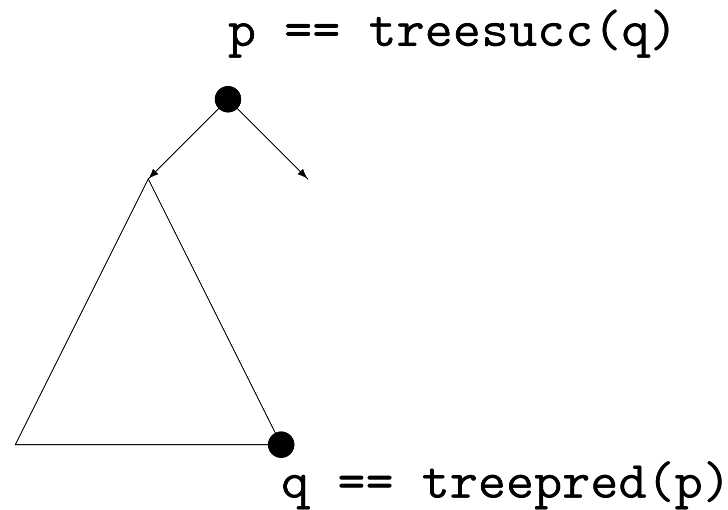
La funzione `treepred` che restituisce, se esiste, il puntatore al predecessore del nodo passato come parametro, è la *duale* di `treesucc`.

```
searchtree *treepred(searchtree *q) /* si assume q != NULL */
{
    searchtree *qq;

    if(q->left)
        return treemax(q->left);
    qq = q->up;
    while(qq && q == qq->left) {
        q = qq;
        qq = qq->up;
    }
    return qq;
}
```

`treesucc` e `treepred` richiedono tempo $O(h)$.

NOTA: Se un nodo ha due figli, allora il suo successore non ha figlio sinistro e il suo predecessore non ha figlio destro.





Nella prima figura `q` non ha figlio sinistro: `q == treemin(p->right)`.

Nella seconda figura `q` non ha figlio destro: `q == treemax(p->left)`.

Inserimento
La funzione `insert` crea un nodo con la chiave `k` e lo introduce "al posto giusto" nell'albero. Viene restituita la nuova radice.

```
searchtree *insert(searchtree *p, key k) {
    searchtree *q = malloc(sizeof(searchtree));
    searchtree *r = p;
    searchtree *s = NULL;

    if(!q) { fprintf(stderr,"Errore di allocazione\n"); exit(-1); }
    q->v = k;
    q->left = q->right = NULL;
    while(r) {
        s = r;
        r = k < r->v ? r->left : r->right;
    }
    q->up = s;
    if(!s)          return q;
    if(k < s->v)    s->left = q;
    else            s->right = q;
    return p;
}
```

```
searchtree *q = malloc(sizeof(searchtree));
...
if(!q) { fprintf(stderr,"Errore di allocazione\n"); exit(-1); }
q->v = k;
q->left = q->right = NULL;
```

In primo luogo viene allocato il nuovo nodo puntato da `q` nella memoria dinamica.

```
while(r) {
    s = r;
    r = k < r->v ? r->left : r->right;
}
```

`r` scende nell'albero scegliendo il ramo sinistro o destro a seconda dell'esito del confronto. `s` punta al padre di `r`.

```
q->up = s;
if(!s)          return q;
if(k < s->v)    s->left = q;
else            s->right = q;
return p;
```

Si collega il nodo puntato da `q` nel punto individuato da `s`. Se `s` è `NULL`, l'albero è vuoto e viene restituita la nuova radice `q`, altrimenti `q` viene agganciato come figlio sinistro o destro a seconda dell'esito del confronto fra le chiavi.

Cancellazione

La funzione `delete` cancella il nodo puntato da `q` dall'albero. Poiché tale nodo può essere la radice, `delete` restituisce la nuova radice.

```
searchtree *delete(searchtree *p, searchtree *q) /* si assume q != NULL */
{
    searchtree *r, *s, *t = NULL;

    if(!q->left || !q->right)   r = q;
    else                        r = treesucc(q);
    s = r->left ? r->left : r->right;
    if(s)                       s->up = r->up;
    if(!r->up)                  t = s;
    else
        if(r == r->up->left)    r->up->left = s;
        else                    r->up->right = s;
    if(r != q)                  q->v = r->v;
    free(r);
    return t ? t : (p != r ? p : NULL);
}
```

`delete` prende in esame tre casi:

1. Se `q` non ha figli, semplicemente si modifica il nodo padre, ponendo a `NULL` il puntatore che puntava a `q`.

2. Se `q` ha un unico figlio, si crea un collegamento tra il padre di `q` e il figlio di `q`.

3. Se `q` ha due figli, si determina il successore `r` di `q`: esso non ha figlio sinistro (vedi NOTA pag. 374). Si elimina `r` come nei casi 1 e 2, previa la sostituzione delle info contenute in `q` con quelle contenute in `r`.

```
searchtree *r, *s, *t = NULL;
```

r sarà uguale a q nei casi 1 e 2, sarà il successore di q nel caso 3. s sarà il figlio di r oppure NULL se r non ha figli. t sarà diverso da NULL solo se la radice dovrà essere modificata.

```
    if(!q->left || !q->right)   r = q;
    else                        r = treesucc(q);
    s = r->left ? r->left : r->right;
```

Viene determinato in quale dei tre casi siamo, e assegnati i valori corrispondenti a r e s.

```
    if(s)                       s->up = r->up;
    if(!r->up)                  t = s;
    else
        if(r == r->up->left)    r->up->left = s;
        else                    r->up->right = s;
```

Il nodo r viene estratto dall'albero, modificando opportunamente il padre di r e s. Si gestisce il caso in cui r è la radice e il caso in cui s è NULL.

```
    if(r != q)                  q->v = r->v;
    free(r);
    return t ? t : (p != r ? p : NULL);
```

Viene deallocato il nodo puntato da r. Se r != q siamo nel caso 3 e si copiano le info di r in q. L'ultima riga ritorna t se la radice è da modificare, p se non è da modificare, NULL se q era l'unico nodo dell'albero.

Anche le operazioni di inserimento e cancellazione dell'albero richiedono tempo $O(h)$, per h l'altezza dell'albero.

Distruzione di un albero di ricerca binaria:

(Esercizio: dire perché il codice seguente effettua troppe scansioni di rami dell'albero, e migliorarlo)

```
void destroysearchtree(searchtree *p) /* si assume p != NULL */
{
    while(p = delete(p,p));
}
```

Creazione di un albero di ricerca binaria:

```
searchtree *createsearchtree(void)
{
    return NULL;
}
```

Riepilogo delle Prestazioni

Le operazioni di ricerca, inserimento, cancellazione, minimo, massimo, predecessore, successore, richiedono tempo $O(h)$ per h l'altezza dell'albero.

Nel caso peggiore in cui l'albero degenera in una lista ho $h = n$.

Nel caso medio di un albero di ricerca binaria costruito in modo casuale con n chiavi (distinte) l'altezza h è $h \in O(\log_2 n)$.

Provare `rndtree.c`, per la generazione di permutazioni casuali di n chiavi e per la costruzione degli alberi binari di ricerca mediante inserimenti successivi nell'ordine specificato dalle permutazioni.

Potrete verificare sperimentalmente quale è l'altezza media degli alberi così costruiti.

Alberi Bilanciati

Le operazioni fondamentali sugli alberi di ricerca binaria:

Ricerca, inserimento, cancellazione, min, max, predecessore, successore,

richiedono tempo $O(h)$ dove h è l'altezza dell'albero, vale a dire la lunghezza del suo ramo più lungo.

Se potessimo garantire che ogni nodo ripartisca i suoi discendenti in modo tale che il suo sottoalbero sinistro contenga lo stesso numero di nodi del suo sottoalbero destro, allora avremmo un albero perfettamente *bilanciato*, la cui altezza sarebbe

$$h = 1 + \lfloor \log_2 n \rfloor,$$

dove n è il numero totale di nodi nell'albero.

Non si può richiedere che l'albero soddisfi perfettamente questa condizione.

Ma condizioni un poco più "rilassate" permettono implementazioni efficienti nella gestione della costruzione e della manipolazione di alberi sufficientemente bilanciati.

Agli effetti pratici parleremo di alberi **bilanciati** quando si garantisce che l'altezza dell'albero sia $O(\log_2 n)$.

Vi sono vari criteri che garantiscono questa proprietà: in dipendenza dal criterio definitorio utilizzato abbiamo vari tipi di alberi bilanciati.

Ad esempio possiamo richiedere che per ogni nodo le altezze h_l e h_r dei due sottoalberi sinistro e destro del nodo possano differire al più di 1.

Ogni albero che soddisfi questo criterio è bilanciato poiché ha altezza $O(\log_2 n)$.

Gli alberi che soddisfano questo criterio sono chiamati *Alberi AVL* (Adel'son, Vel'skiĭl, Landis). Gli alberi AVL sono uno dei primi esempi in letteratura di alberi bilanciati gestibili con algoritmi dalle prestazioni accettabili.

Vi sono numerosi tipi di alberi bilanciati: AVL, 2-3, 2-3-4, Red-Black, Splay, B-alberi, etc.

In genere, gli algoritmi per la manipolazione degli alberi bilanciati sono complicati, così come la loro implementazione in C.

Alberi Red-Black

Un albero *Red-Black* è un albero di ricerca binaria in cui ogni nodo contiene come informazione aggiuntiva il suo *colore*, che può essere **red** oppure **black**, e che soddisfa alcune proprietà aggiuntive relative al colore:

1) Ciascun nodo è **red** o è **black**.

2) Ciascuna foglia è **black**.

3) Se un nodo è **red** allora entrambi i figli sono **black**.

4) Ogni cammino da un nodo a una foglia sua discendente contiene lo stesso numero di nodi **black**.

Negli alberi Red-Black una foglia non contiene informazione sulla chiave, ma serve solo a segnalare la terminazione del ramo che la contiene.

Implementativamente, useremo un unico nodo "dummy" (o "sentinella") per rappresentare tutte le foglie.

Per semplicità implementativa è utile imporre un'ulteriore proprietà:

5) la radice è **black**.

Il numero di nodi **black** su un cammino da un nodo p (escluso) a una foglia sua discendente è detta la *b-altezza* del nodo $bh(p)$.

Un sottoalbero con radice p di un albero Red-Black contiene almeno $2^{bh(p)} - 1$ nodi interni.

Per induzione: base: se p è la foglia allora non vi e' alcun nodo interno da conteggiare e infatti $2^0 - 1 = 0$: Ok.
Passo: I figli di p hanno b-altezza $bh(p)$ o $bh(p) - 1$ a seconda del loro colore. Quindi: il numero di nodi interni al sottoalbero di radice p deve essere
$\geq (2^{bh(p)-1} - 1) + (2^{bh(p)-1} - 1) + 1 = 2^{bh(p)} - 1$: Ok.

Poiché su ogni ramo almeno metà dei nodi sono black per la proprietà 3, allora la b-altezza della radice deve essere $\geq h/2$, per h l'altezza dell'albero.
Quindi il numero di nodi interni n dell'albero è tale che:

$$n \geq 2^{bh(root)} - 1 \geq 2^{h/2} - 1.$$

Si conclude:

Un albero Red-Black con n nodi (interni) ha altezza

$$h \leq 2 \log_2(n+1).$$

Dunque, gli alberi Red-Black sono alberi bilanciati, e le operazioni fondamentali di ricerca, min, max, predecessore,successore sono eseguite in tempo $O(\log_2 n)$.

Per quanto riguarda inserimento e cancellazione: dobbiamo vedere come implementare queste operazioni.

Implementazione di alberi Red-Black.

Nell'header file `rbtree.h` riportiamo le definizioni di tipo necessarie:

```
typedef int key;

typedef enum { red, black } color;

struct rbnode {
    key v;
    color c;
    struct rbnode *left, *right, *up;
};

typedef struct rbnode rbnode;

typedef struct {
    rbnode *root, *nil;
} rbtree;
```

DIGRESSIONE: Tipi enumerativi

In C è possibile definire *tipi enumerativi*:

```
enum giorno { lun, mar, mer, gio, ven, sab, dom };

enum giorno lezione;
lezione = mer;
```

Si possono dichiarare variabili di un tipo enumerativo e assegnare loro i valori del tipo stesso.

I valori vengono implicitamente interpretati come costanti di tipo `int`. Per default il primo di questi valori vale 0, e gli altri assumono via via i valori successivi (`lun == 0`, `mar == 1`, `...` , `dom == 6`). Si può altresì specificare esplicitamente il valore numerico dei simboli:

```
typedef enum { lun = 1, mar, mer, gio, ven, sab = 10, dom } giorno;

giorno lezione = mer;  /* mer == 3  */
giorno riposo = dom;   /* dom == 11 */
```

```
struct rbnode {
    key v;
    color c;
    struct rbnode *left, *right, *up;
};
```

Modifichiamo la definizione di nodo di albero binario di ricerca aggiungendo il campo `color c`.

```
typedef struct {
    rbnode *root, *nil;
} rbtree;
```

Una variabile di tipo `rbtree` costituisce un *header* per un albero Red-Black: contiene due puntatori, `root` punterà alla radice dell'albero, mentre `nil` rappresenterà *ogni* foglia dell'albero: è il nodo sentinella.

Creazione dell'albero Red-Black:

```
rbtree *createrbtree(void)
{
    rbtree *t = malloc(sizeof(rbtree));

    if(!t) {
        fprintf(stderr,"Errore di allocazione A\n");
        exit(-1);
    }
    if(!(t->root = malloc(sizeof(rbnode)))) {
        fprintf(stderr,"Errore di allocazione B\n");
        exit(-2);
    }
    t->nil = t->root;
    t->nil->left = t->nil->right = t->nil->up = t->nil;
    t->nil->c = black;
    return t;
}
```

`createrbtree` in primo luogo alloca l'header `t` e il nodo sentinella, controllando l'esito delle allocazioni.

All'inizio l'albero è vuoto, quindi esiste solo il nodo sentinella: `t->root` e `t->nil` punteranno alla sentinella, così come tutti i puntatori della sentinella stessa.

Il colore della sentinella (che rappresenta ogni foglia) è `black`.

Rotazioni

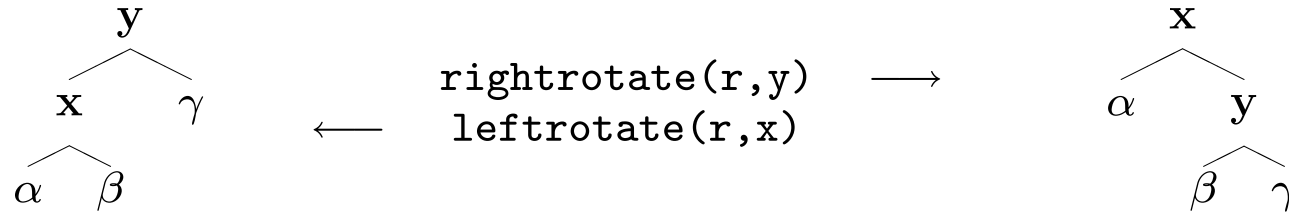
Poiché gli alberi Red-Black sono una versione specializzata degli alberi di ricerca binaria, tutte le operazioni fondamentali implementate per gli alberi di ricerca binaria possono essere applicate agli alberi Red-Black.

Però, le operazioni che modificano l'albero, *Inserimento* e *Cancellazione*, in genere violano le proprietà Red-Black.

Potremo ristabilire le proprietà violate modificando localmente la struttura dell'albero e la colorazione di alcuni nodi senza intaccare l'ordine di visita degli elementi.

Per questo ci dotiamo delle operazioni di *rotazione sinistra* e *rotazione destra*.

Le rotazioni non modificano l'ordinamento delle chiavi secondo la visita inorder.




```
void leftrotate(rbtree *r, rbnode *x)
{
    rbnode *y = x->right;

    x->right = y->left;
    if(y->left != r->nil)
        y->left->up = x;
    y->up = x->up;
    if(x->up == r->nil)
        r->root = y;
    else
        if(x == x->up->left)
            y->up->left = y;
        else
            y->up->right = y;
    y->left = x;
    x->up = y;
}
```

```
void rightrotate(rbtree *r, rbnode *x)
{
    rbnode *y = x->left;

    x->left = y->right;
    if(y->right != r->nil)
        y->right->up = x;
    y->up = x->up;
    if(x->up == r->nil)
        r->root = y;
    else
        if(x == x->up->right)
            y->up->right = y;
        else
            y->up->left = y;
    y->right = x;
    x->up = y;
}
```

Inserimento

La procedura di inserimento di un elemento nell'albero Red-Black richiede tempo $O(\log_2 n)$.

In primo luogo si alloca il nodo `x` contenente la chiave `k` e si inserisce `x` nell'albero usando la normale procedura `insert` per alberi di ricerca binaria (qui è chiamata `simpleinsert`).
Il nodo inserito è colorato `red`.

Quali proprietà Red-Black possono essere violate ?
1) NO. Il nodo `x` è `red`.
2) NO. Il nodo `x` non è una foglia (non è `*nil`).
3) SI'. Dipende da dove è stato inserito il nodo.
4) NO. Si è sostituita una foglia `black` con un nodo `red` che ha due figli foglie `black` (`*nil`).

La proprietà 3) è violata quando il padre di `x` è `red`.

Per ristabilire la proprietà 3) faremo risalire la "violazione" nell'albero, preservando le altre proprietà: all'inizio di ogni iterazione `x` punterà a un nodo `red` con padre `red`.

Vi sono sei casi da considerare, ma i 3 casi "destri" sono simmetrici ai 3 casi "sinistri".

I 3 casi sinistri (risp., destri) corrispondono a situazioni in cui il padre di `x` è un figlio sinistro (risp., destro).

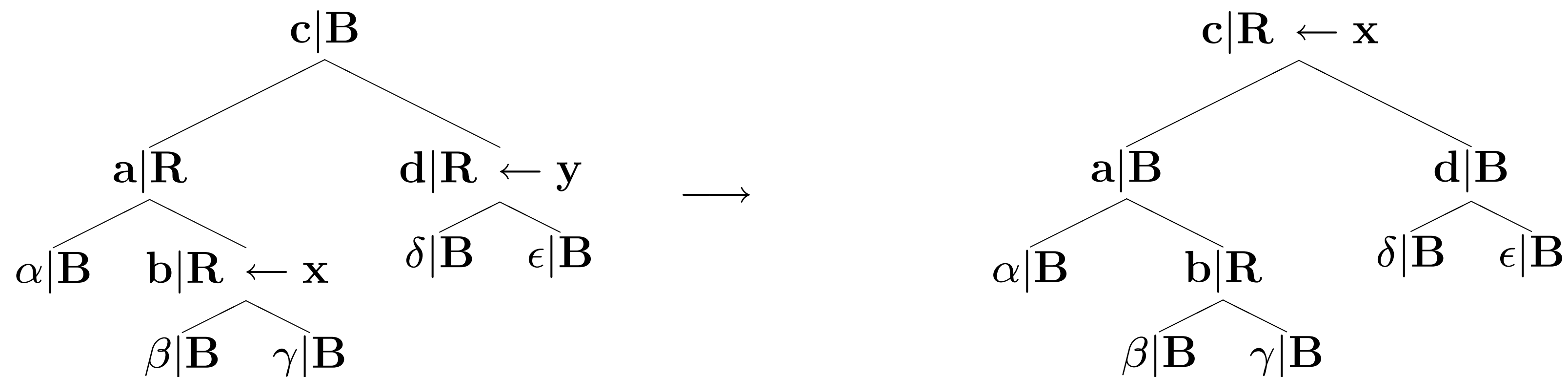
La proprietà 5) garantisce che il nonno di `x` esista sempre.

Assumiamo che il padre `x->up` di `x` sia figlio sinistro.
Il nonno di `x` è `black` perchè vi è una sola violazione nell'albero.
Sia `y = x->up->up->right` lo "zio destro" di `x`.

Caso 1) Lo zio `y` è `red`.
Si colorano `black` i nodi `x->up` e `y` e si colora `red` il nonno di `x`.
A questo punto può essere il nonno `x->up->up` l'unico nodo che viola RB3). Si compie un'altra iterazione.
Caso 2) Lo zio `y` è `black` e `x` è figlio destro.
Una `leftrotate(x->up)` ci porta al caso 3) senza modificare le b-altezze dei nodi ne' violare RB4) poiché sia `x` che `x->up` sono `red`.
Caso 3) Lo zio `y` è `black` e `x` è figlio sinistro.
Una `rightrotate(x->up->up)` e le ricolorazioni del padre e del nonno terminano l'esecuzione, in quanto non ci sono più nodi rossi adiacenti.

Commenti supplementari alla procedura di inserimento.
Ripristino di RB3): i tre casi.

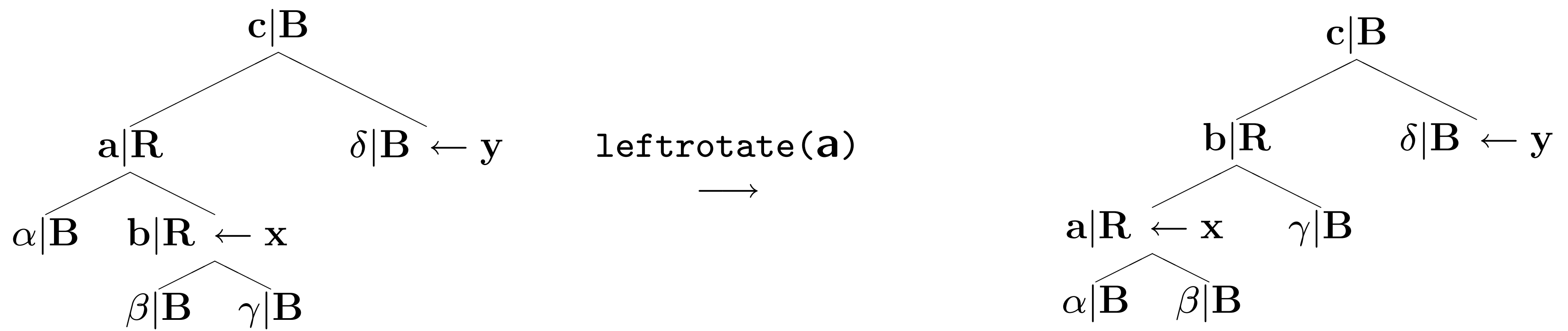
Caso 1)



Si applica anche quando `x` è figlio sinistro del padre.

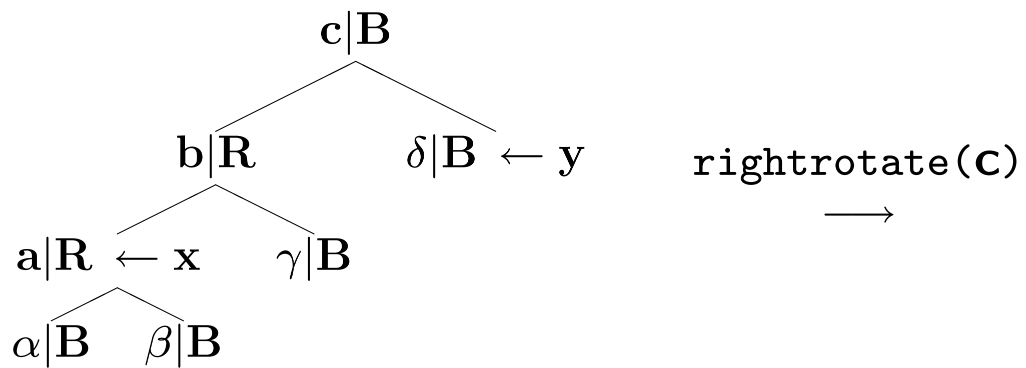
Si itera col nuovo valore di `x`.

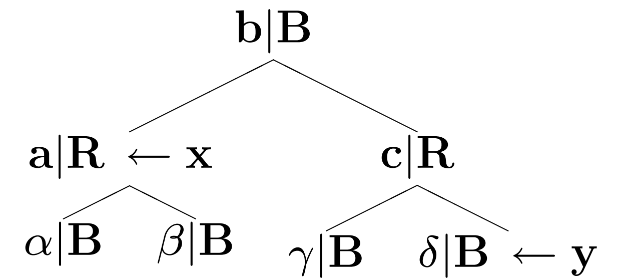
Caso 2)



Porta al caso 3).

Caso 3)





La proprietà RB3) è ripristinata.

```
void rbinsert(rbtree *tree, key k)
{
    rbnode *x = simpleinsert(tree, k);    /* inserisce k come in albero binario */
    rbnode *y;                            /* y sara' lo zio di x */

    while(x != tree->root && x->up->c == red) {
        if(x->up == x->up->up->left) {              /* caso L */
            y = x->up->up->right;
            if(y->c == red) {
                x->up->c = black;                   /* caso 1L */
                y->c = black;                       /* caso 1L */
                x->up->up->c = red;                 /* caso 1L */
                x = x->up->up;                      /* caso 1L */
            } else {
                if(x == x->up->right)               /* caso 2L */
                    leftrotate(tree,x = x->up);     /* caso 2L */
                x->up->c = black;                   /* caso 3L */
                x->up->up->c = red;                 /* caso 3L */
                rightrotate(tree,x->up->up);        /* caso 3L */
            }
        } else {                                    /* caso R */
            ...
        }
    }
    tree->root->c = black;         /* colora black la radice */
}
```

```
rbnode *simpleinsert(rbtree *tree, key k)
{
    rbnode *q = malloc(sizeof(rbnode));
    rbnode *r = tree->root;
    rbnode *s = tree->nil;

    if(!q) {
        fprintf(stderr,"Errore di allocazione C\n");
        exit(-4);
    }
    q->v = k;
    q->left = q->right = tree->nil;
    q->c = red;                      /* colora red il nodo da inserire */
    while(r != tree->nil) {
        s = r;
        r = k < r->v ? r->left : r->right;
    }
    q->up = s;
    if(s == tree->nil)
        return tree->root = q;
    if(k < s->v)
        s->left = q;
    else
        s->right = q;
    return q;                        /* ritorna un puntatore al nodo inserito */
}
```

Commenti sulle prestazioni.

Le procedure di rotazione `leftrotate` e `rightrotate` richiedono tempo $O(1)$.

Nella funzione `rbinsert` il ciclo `while` è ripetuto solo se si esegue il caso 1 con conseguente spostamento verso la radice del puntatore `x`.

Tale ciclo può essere eseguito al più $O(\log_2 n)$ volte, dunque `rbinsert` richiede tempo $O(\log_2 n)$.

Si noti che `rbinsert` effettua al più due rotazioni, poiché se si esegue il caso 2 o 3 la funzione termina.

Cancellazione
La procedura di cancellazione di un nodo da un albero Red-Black richiede tempo $O(\log_2 n)$.
E' però considerevolmente più complicata dell'operazione di inserimento.

In primo luogo si richiama la procedura di cancellazione per nodi da alberi di ricerca binaria, modificata leggermente:
-Non c'e' bisogno di gestire alcuni casi terminali grazie all'uso della sentinella.
-Se il nodo estratto è `black`, viene violata la proprietà RB4).
-Se il figlio del nodo estratto è `red`, può essere momentaneamente violata RB3): la si ripristina subito colorando `black` il figlio.

Prima di uscire da `delete` si invoca un'apposita funzione `fixup` sul figlio del nodo estratto, che ripristinerà RB4) (e la momentanea violazione di RB3)) preservando le altre proprietà Red-Black.

```
void rbdelete(rbtree *tree, rbnode *q)
{
    rbnode *r, *s;

    if(q->left == tree->nil || q->right == tree->nil)
        r = q;
    else
        r = treesucc(tree,q);
    s = r->left != tree->nil ? r->left : r->right;
    s->up = r->up;           /* non si controlla che s non sia nil */
    if(r->up == tree->nil)
        tree->root = s;
    else
        if(r == r->up->left)
            r->up->left = s;
        else
            r->up->right = s;
    if(r != q)
        q->v = r->v;
    if(r->c == black)          /* se il nodo estratto e' black */
        fixup(tree, s);        /* richiama fixup sul figlio s  */
    free(r);
}
```

```
void fixup(rbtree *tree, rbnode *x) {
    rbnode *w;
    while(x != tree->root && x->c == black) {
        if(x == x->up->left) {                                  /* caso L */
            if((w = x->up->right)->c == red) {
                w->c = black;   x->up->c = red;                 /* caso 1L */
                leftrotate(tree,x->up);                         /* caso 1L */
                w = x->up->right;                               /* caso 1L */
            }
            if(w->left->c == black && w->right->c == black) {
                w->c = red;                                     /* caso 2L */
                x = x->up;                                      /* caso 2L */
            } else {
                if(w->right->c == black) {
                    w->left->c = black; w->c = red;             /* caso 3L */
                    rightrotate(tree,w);                        /* caso 3L */
                    w = x->up->right;                           /* caso 3L */
                }
                w->c = x->up->c;    x->up->c = black;           /* caso 4L */
                w->right->c = black;                            /* caso 4L */
                leftrotate(tree,x->up);                         /* caso 4L */
                x = tree->root;                                 /* caso 4L */
            }
        } else { /* caso R */ }
    }
    x->c = black;                       /* colora black il nodo rosso o la radice */
}
```

Cancellazione.

Analisi della funzione `fixup(rbtree *tree, rbnode *x)`

La procedura `fixup` ripristina le proprietà Red-Black dopo la cancellazione di un nodo colorato **black**.

Al momento dell'eliminazione del nodo, l'unica proprietà che può essere violata è RB4) (NB: anche RB3 se il nodo `x` e suo padre sono **red**, ma questo si risolve subito colorando `x` **black**):

RB4) Ogni cammino da un nodo a una foglia sua discendente contiene lo stesso numero di nodi **black**.

Dobbiamo ripristinarla senza violare le altre proprietà.

RB4) è violata:

Ogni cammino che conteneva il nodo **black** estratto, ora ha un nodo **black** di meno. Ogni antenato del nodo estratto viola RB4).

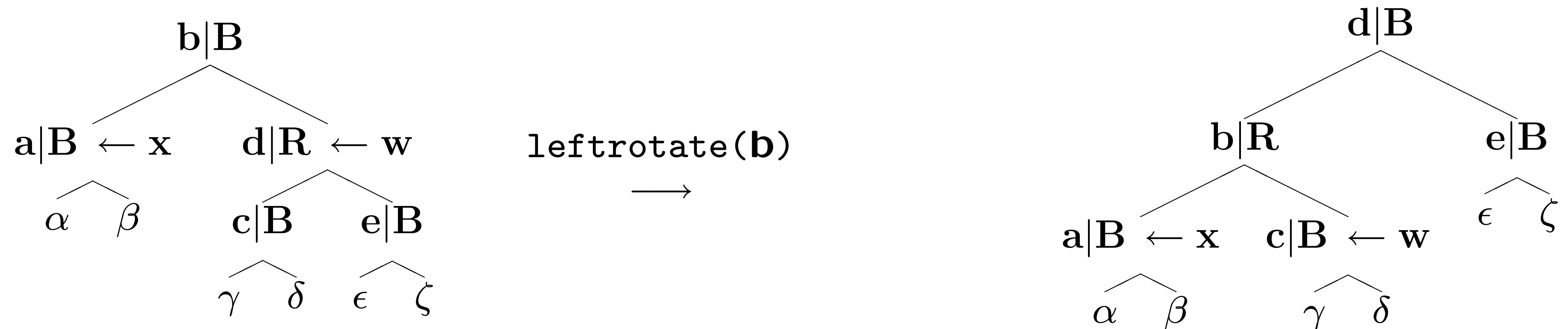
Se (l'unico) figlio del nodo estratto è **red** lo ricoloriamo **black** e così facendo abbiamo ristabilito RB4).

Se invece il figlio era **black**, si risale nell'albero fino a quando, o si trova un nodo **red** da poter colorare **black**, o si raggiunge la radice, o si possono eseguire opportune rotazioni e ricolorazioni che risolvano il problema.

A ogni iterazione `x` punta a un nodo **black**.
Se `x` è figlio sinistro vi sono 4 casi da trattare.
Dualizzando si ottengono i 4 casi per `x` figlio destro.

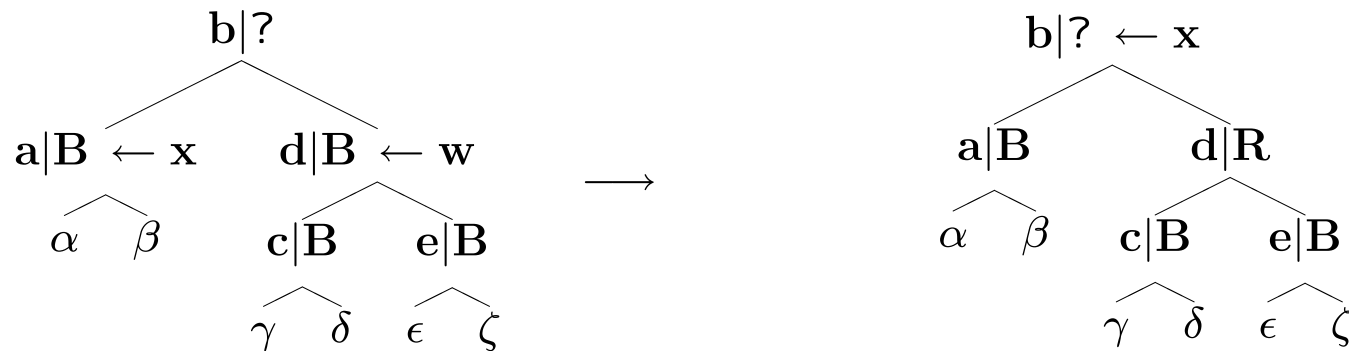
Sia `w` il fratello di `x`. `w` non è foglia, altrimenti RB4) era già violata prima della cancellazione.

Caso 1) `w` è **red** (il padre di `x` deve quindi essere **black**).



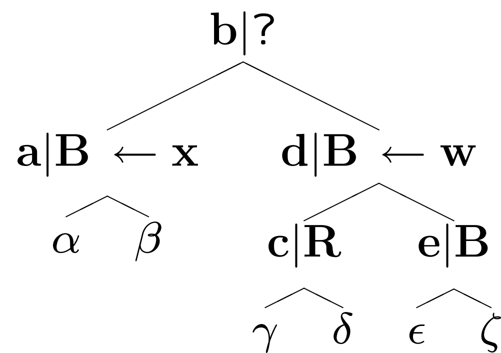
Porta a uno dei casi 2), 3), 4).

Caso 2) `w` è **black**. Entrambi i figli di `w` sono **black**.
Si può colorare `w` **red**. Il problema della violazione di RB4) si sposta sul padre di `x`.

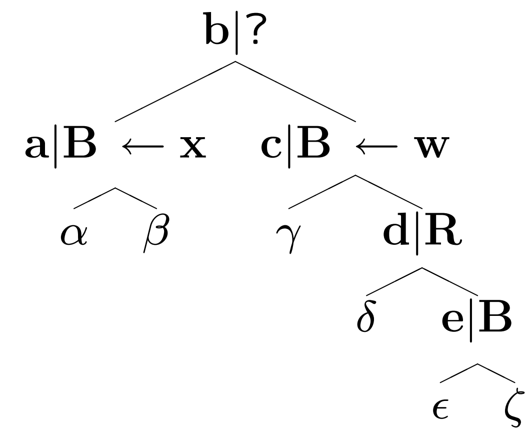


Si itera sul nuovo valore di `x`.

Caso 3) `w` è **black**. il suo figlio sinistro è **red** e invece il suo figlio destro è **black**.

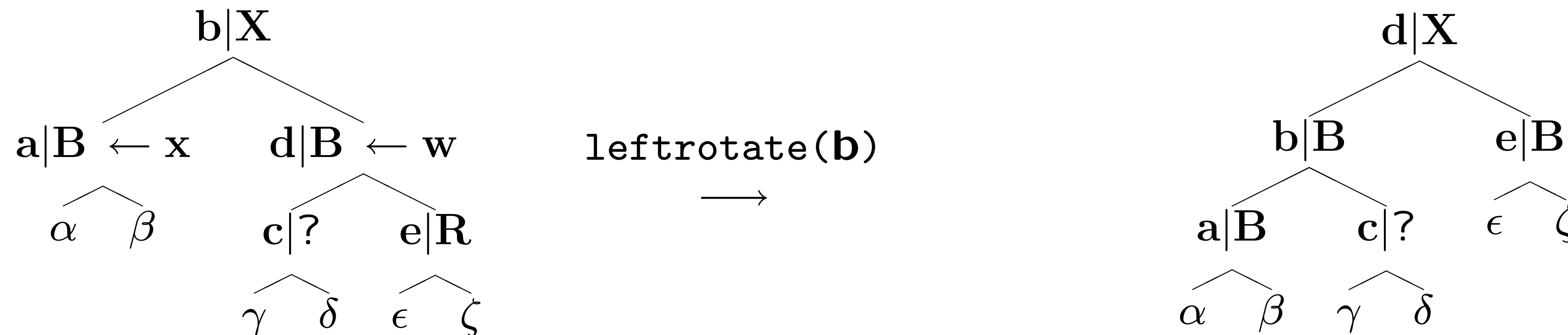


`rightrotate(`**`d`**`)`
$\longrightarrow$



Porta al caso 4).

Caso 4) `w` è **black**. il suo figlio destro è **red**.



`x` viene posto = `root`.

La proprietà RB4) è ripristinata.

Infatti ogni cammino che passa per il nodo che era puntato da `x` incontra un `black` in più. Il numero di nodi `black` su ogni cammino che non tocca `x` non è stato modificato.

Commenti sulle prestazioni.

La cancellazione del nodo, senza `fixup` richiede $O(\log_2 n)$.
Per quanto riguarda `fixup`:
– Ogni rotazione richiede tempo $O(1)$.
– I casi 3) e 4) eseguono al più due rotazioni, e poi la procedura termina.
– Il caso 1) richiede una rotazione, poi:
Se il caso 1) porta al caso 2) non si verifica alcuna ulteriore iterazione poiché il padre di `x` sarà sicuramente **red**.
Altrimenti il caso 1) porta al caso 3) o 4).
– Il caso 2) può dare luogo a una serie di iterazioni che portano il puntatore `x` a risalire nell'albero. Al massimo si raggiunge la radice dopo $O(\log_2 n)$ iterazioni.

Dunque la cancellazione di un elemento da un albero Red-Black richiede $O(\log_2 n)$ passi e al più 3 rotazioni.

Rappresentazione di Grafi (non orientati)

Vi sono molti modi di rappresentare grafi non orientati.

Quando si deve scegliere come implementare grafi (orientati o meno) in una certa applicazione, bisogna considerare gli eventuali vincoli — impliciti o espliciti — e le possibili operazioni che caratterizzano la classe di grafi adatta all'applicazione.

Consideriamo due rappresentazioni alternative per grafi non orientati generici:
– con *matrice* d'adiacenza.
– con *liste* d'adiacenza.

Matrice d'adiacenza.

Consideriamo un grafo non orientato $G = (V, E)$. Numeriamo i vertici di V: Indichiamo con v_i il vertice di V la cui etichetta è $i \in \{1, 2, \ldots, |V|\}$.

La **matrice di adiacenza** di G è la matrice simmetrica `M` con $|V| \times |V|$ valori booleani in cui `M[x][y] == 1` se e solo se l'arco $(v_{\mathtt{x}}, v_{\mathtt{y}}) \in E$.

La rappresentazione con matrici di adiacenza sono accettabili solamente se i grafi da manipolare *non* sono *sparsi* (A G mancano "pochi" archi per essere il grafo completo su V).

Lo spazio necessario con questa rappresentazione è $O(V^2)$. Anche il tempo richiesto da molti degli algoritmi fondamentali è $O(V^2)$ con questa rappresentazione.

Liste d'adiacenza.

Quando il grafo è sparso, diciamo $|E| = O(|V| \log |V|)$, la rappresentazione attraverso **liste di adiacenza** è spesso da preferire.

In questa rappresentazione vi è una lista per ogni vertice $v_i \in V$. Ogni elemento di una tale lista è a sua volta un vertice $v_j \in V$. v_j appartiene alla lista associata a v_i se e solo se $(v_i, v_j) \in E$.

I dettagli implementativi variano a seconda delle necessità applicative.
Ad esempio, le liste di adiacenza possono essere contenute in un array statico o dinamico, una lista concatenata, una tabella hash, un albero di ricerca, etc...
Anche le informazioni contenute nei nodi delle liste di adiacenza dipendono dalle esigenze dell'applicazione.

Consideriamo una semplice implementazione con un array allocato in memoria dinamica di liste d'adiacenza.

Questa implementazione non è adatta ad applicazioni che modifichino frequentemente i grafi.

```
struct node {               /* nodo di lista di adiacenza */
    int v;
    struct node *next;
};

struct graph {              /* struttura associata a ogni grafo */
    int V;                  /* numero nodi  */
    int E;                  /* numero archi */
    struct node **A;        /* array di liste di adiacenza */
};
```

Creazione del grafo

```
struct graph *creategraph(int nv, int ne)
{
    struct graph *g = malloc(sizeof(struct graph));

    if(!g) {
        fprintf(stderr,"Errore di Allocazione\n");
        exit(-1);
    }
    g->E = ne;
    g->V = nv;
    if(!(g->A = calloc(nv,sizeof(struct node *))))  {
        fprintf(stderr,"Errore di Allocazione\n");
        exit(-2);
    }
    return g;
}
```

La funzione `creategraph` richiede il numero di vertici e il numero di nodi del grafo da creare. Alloca spazio per ogni lista di adiacenza, inizialmente vuota.

Lettura del grafo:

Dopo aver creato la struttura delle liste di adiacenza, bisogna inserire gli archi nelle liste stesse.

```
void readgraph(struct graph *g, FILE *fp) {
    int i,v1, v2;

    for(i = 0; i < g->E; i++) {
        fscanf(fp,"%d %d",&v1,&v2);
        g->A[v1-1] = vertexinsert(g->A[v1-1],v2);
        g->A[v2-1] = vertexinsert(g->A[v2-1],v1);
    }
}
```

Usiamo questa funzione che legge da un file gli archi. Nel file ogni riga contiene un unico arco specificato come coppia di vertici.

La funzione per inserire i vertici nelle liste di adiacenza è una normale funzione per l'inserzione in testa a liste concatenate:

```
struct node *vertexinsert(struct node *p, int k) {
    struct node *q = malloc(sizeof(struct node));

    if(!q) { fprintf(stderr,"Errore di Allocazione\n"); exit(-3); }
    q->v = k;
    q->next = p;
    return q;
}
```

Si noti che per ogni arco si devono eseguire due chiamate a `vertexinsert` su due liste diverse dell'array `g->A`.
Dato un arco $(v, w) \in E$, si deve inserire v nella lista associata a w, così come w nella lista associata a v.

Attraversamento in Profondità

L'attraversamento in profondità (**depth-first search, DFS**) di un grafo non orientato consiste nel visitare ogni nodo del grafo secondo un ordine compatibile con quanto qui di seguito specificato:

Il prossimo nodo da visitare è connesso con un arco al nodo più recentemente visitato che abbia archi che lo connettano a nodi non ancora visitati.

L'attraversamento DFS, fra le altre cose, permette l'individuazione delle componenti connesse di un grafo.

Implementiamo la visita DFS tramite una semplice funzione ricorsiva:

```
void dfs1(struct graph *g, int i, int *aux) {
    struct node *t;
    aux[i] = 1;
    for(t = g->A[i]; t; t = t->next)
        if(!aux[t->v - 1]) {
            printf("%d,",t->v);
            dfs1(g,t->v-1,aux);
        }
}

void dfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { fprintf(stderr,"Errore di Allocazione\n"); exit(-4); }
    for(i = 0; i < g->V; i++)
        if(!aux[i]) {
            printf("\n%d,",i+1);
            dfs1(g,i,aux);
        }
    free(aux);
}
```

La procedura ricorsiva implementata per la visita DFS usa un array di appoggio per memorizzare quando un vertice è già stato incontrato.

Quando `dfs1` è richiamata da `dfs` si entra in una nuova componente connessa.

`dfs1` richiamerà se stessa ricorsivamente fino a quando tutta la componente è stata visitata.

Poiché `dfs1` contiene un ciclo sulla lista di adiacenza del nodo con cui è richiamata, ogni arco viene esaminato in totale due volte, mentre la lista di adiacenza di ogni vertice è scandita una volta sola.

La visita DFS con liste di adiacenza richiede $O(|V| + |E|)$.

Attraversamento in Ampiezza

L'attraversamento in ampiezza (**breadth-first search, BFS**) è un modo alternativo al DFS per visitare ogni nodo di un grafo non orientato.

Il prossimo nodo da visitare lo si sceglie fra quelli che siano connessi al nodo visitato meno recentemente che abbia archi che lo connettano a nodi non ancora visitati.

Vediamone un'implementazione non ricorsiva che memorizza in una coda i nodi connessi al nodo appena visitato.

Sostituendo la coda con uno stack si ottiene una (leggera variante della) visita DFS.

```
void bfs1(struct graph *g, int i, int *aux) {
    struct node *t;
    intqueue *q = createqueue();
    enqueue(q,i);
    while(!emptyq(q)) {
        i = dequeue(q);
        aux[i] = 1;
        for(t = g->A[i]; t; t = t->next)
            if(!aux[t->v - 1]) {
                enqueue(q,t->v -1);
                printf("%d,",t->v);
                aux[t->v-1] = 1;
            }
    }
    destroyqueue(q);
}

void bfs(struct graph *g) {
    int i, *aux = calloc(g->V,sizeof(int));
    if(!aux) { fprintf(stderr,"Errore di Allocazione\n"); exit(-4); }
    for(i = 0; i < g->V; i++)
        if(!aux[i]) {
            printf("\n%d,",i+1);
            bfs1(g,i,aux);
        }
    free(aux);
}
```

La procedura implementata per la visita BFS usa un array di appoggio per memorizzare quando un vertice è già stato incontrato.

Quando `bfs1` è richiamata da `bfs` si entra in una nuova componente connessa.

`bfs1` usa una coda per memorizzare da quali vertici riprendere la visita quando la lista di adiacenza del vertice corrente è stata tutta esplorata.

Ogni lista è visitata una volta, e ogni arco due volte.

La visita BFS con liste di adiacenza richiede $O(|V| + |E|)$.

Complementi: Qualche nozione ulteriore di Input/Output e gestione di stringhe.

Nel file di intestazione `stdio.h` si trovano i prototipi delle funzioni della famiglia di `scanf` e `printf`.

Ad esempio vi sono le funzioni `sscanf` e `sprintf`.
Funzionano come le rispettive funzioni per l'input/output, ma il loro primo argomento è una stringa di caratteri che sostituisce lo stream di input/output.

```
char s[80];
sprintf(s,"%d, %d, %d, stella!",1,2,3); /* s contiene 1, 2, 3, stella! */

sscanf(s,"%d",&i);                      /* i contiene 1 */
```

Accesso diretto ai file.
Il prototipo delle seguenti funzioni è in `stdio.h`.

```
int fseek(FILE *fp, long offset, int place)
```
Riposiziona il cursore per la prossima operazione sul file binario puntato da `fp`.
La nuova posizione del cursore è `offset` byte da:
– Dall'inizio se `place` è `SEEK_SET`
– Dalla posizione corrente prima di `fseek` se `place` è `SEEK_CUR`
– Dalla fine del file se `place` è `SEEK_END`

```
long ftell(FILE *fp)
```
Restituisce la posizione attuale del cursore associato al file `fp` come numero di byte dall'inizio del file stesso.

```
void rewind(FILE *fp)
```
Riporta il cursore all'inizio del file puntato da `fp`.

```
size_t fread(void *a_ptr, size_t el_size, size_t n, FILE *fp)
```

Legge al più `n * el_size` byte dal file puntato da `fp` e li pone nell'array puntato da `a_ptr`.
Viene restituito il numero di elementi effettivamente scritti con successo in `a_ptr`.
Se si incontra la fine del file, il valore restituito è minore di `n * el_size`.

```
size_t fwrite(const void *a_ptr, size_t el_size, size_t n, FILE *fp)
```

Scrive `n * el_size` byte nel file puntato da `fp`, prelevandoli dall'array puntato da `a_ptr`.
Viene restituito il numero di elementi scritti con successo nel file.
In caso di errore viene restituito un valore minore di `n * el_size`.

Consultare il libro, e/o il sito del corso (o altre fonti) per la documentazione relativa alle funzioni di libreria standard i cui prototipi sono in `stdio.h`, `stdlib.h`, `ctype.h`, `string.h`.

Altri file di intestazione interessanti: `assert.h`, `time.h`.