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()
{
    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*.

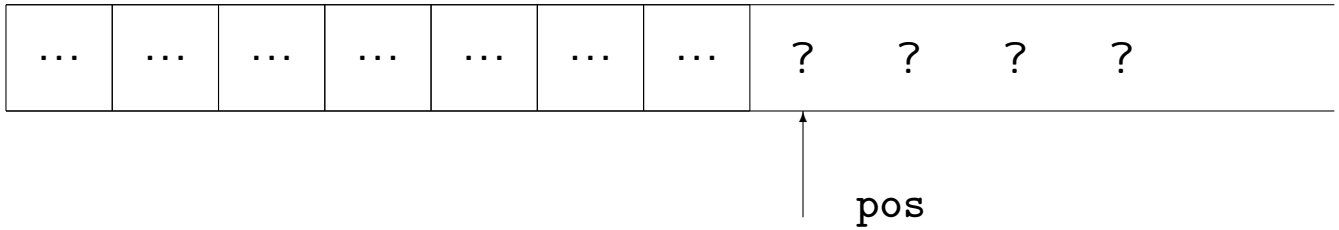
L'array `stack` di `char` conterrà gli elementi inseriti. La variabile `int pos` conterrà sempre l'indice della prima posizione *vuota* di `stack`, vale a dire, la posizione successiva alla *cima* dello stack.

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

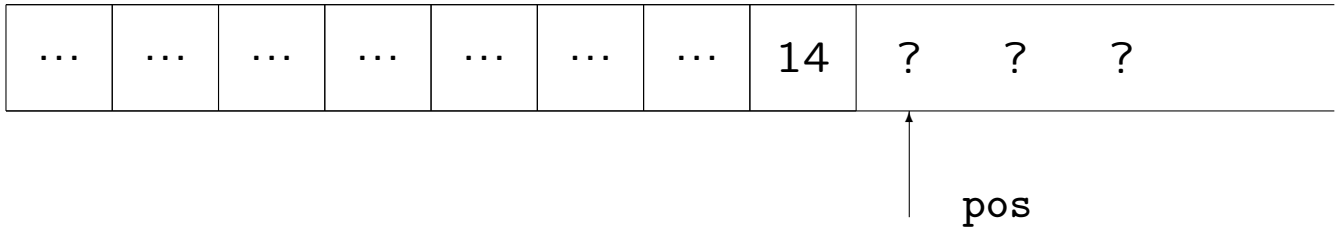
`push` pone l'elemento `e` in cima allo stack. Tale operazione è consentita solo se `stack` ha ancora posizioni vuote, solo se il valore di `pos` è `< SIZE`.

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

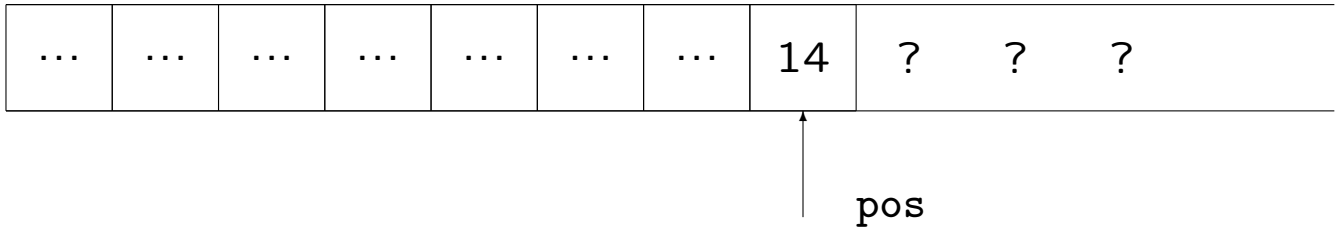
`pop` restituisce il valore contenuto sulla cima dello stack, vale a dire la posizione precedente alla prima posizione libera – quella contenuta in `pos`. L'elemento viene cancellato dallo stack, dunque `pos` viene decrementato per contenere la nuova posizione libera sulla cima dello stack.



`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);`).

- 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*:

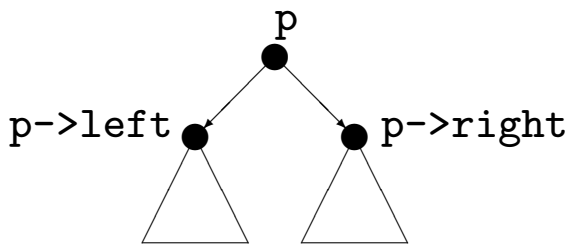
```
void forfewtimes(int i)
{
    static int k = 0;

    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.

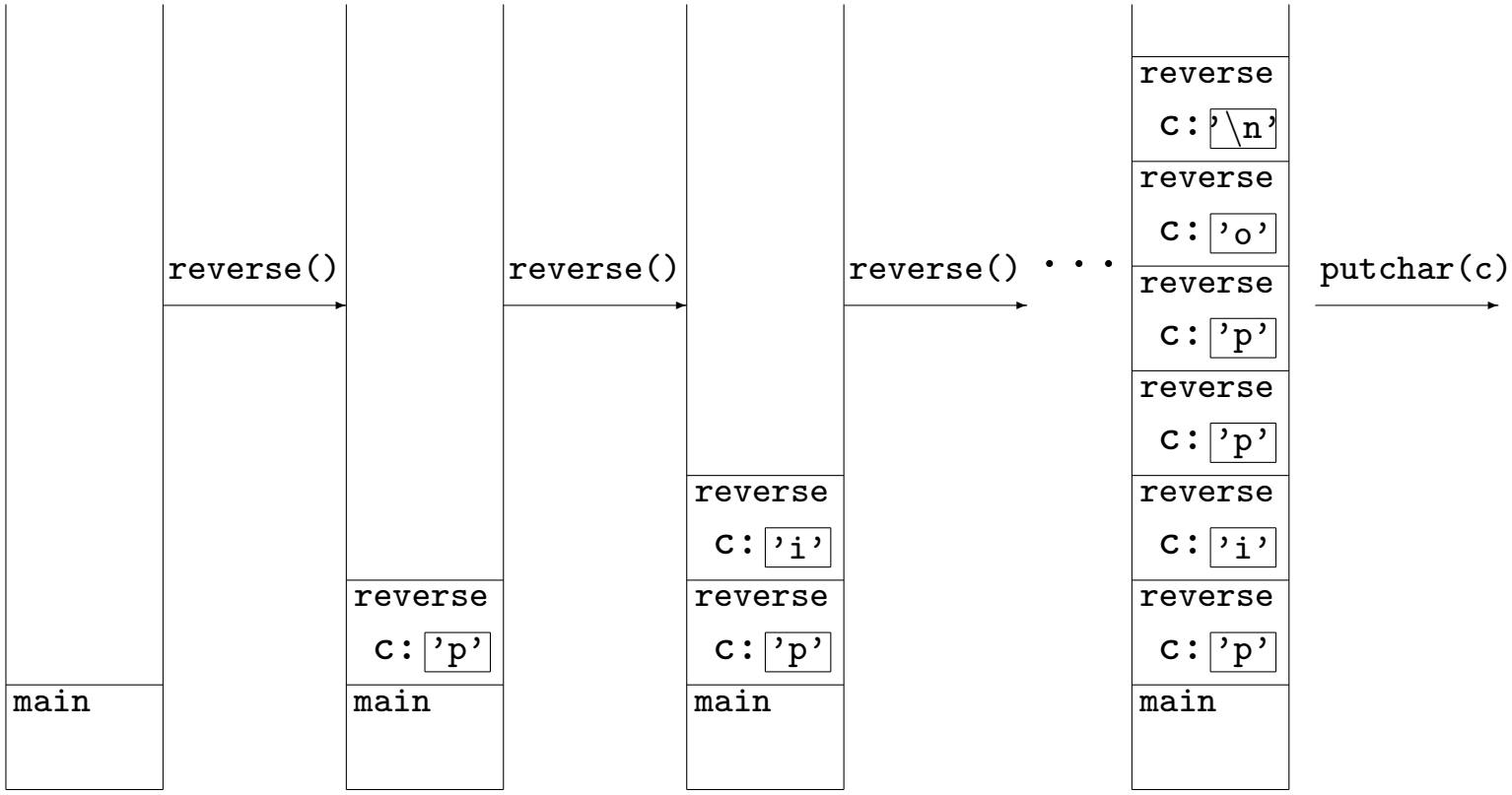
Ogni invocazione di `reverse` termina stampando sul video il carattere letto.

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'`.

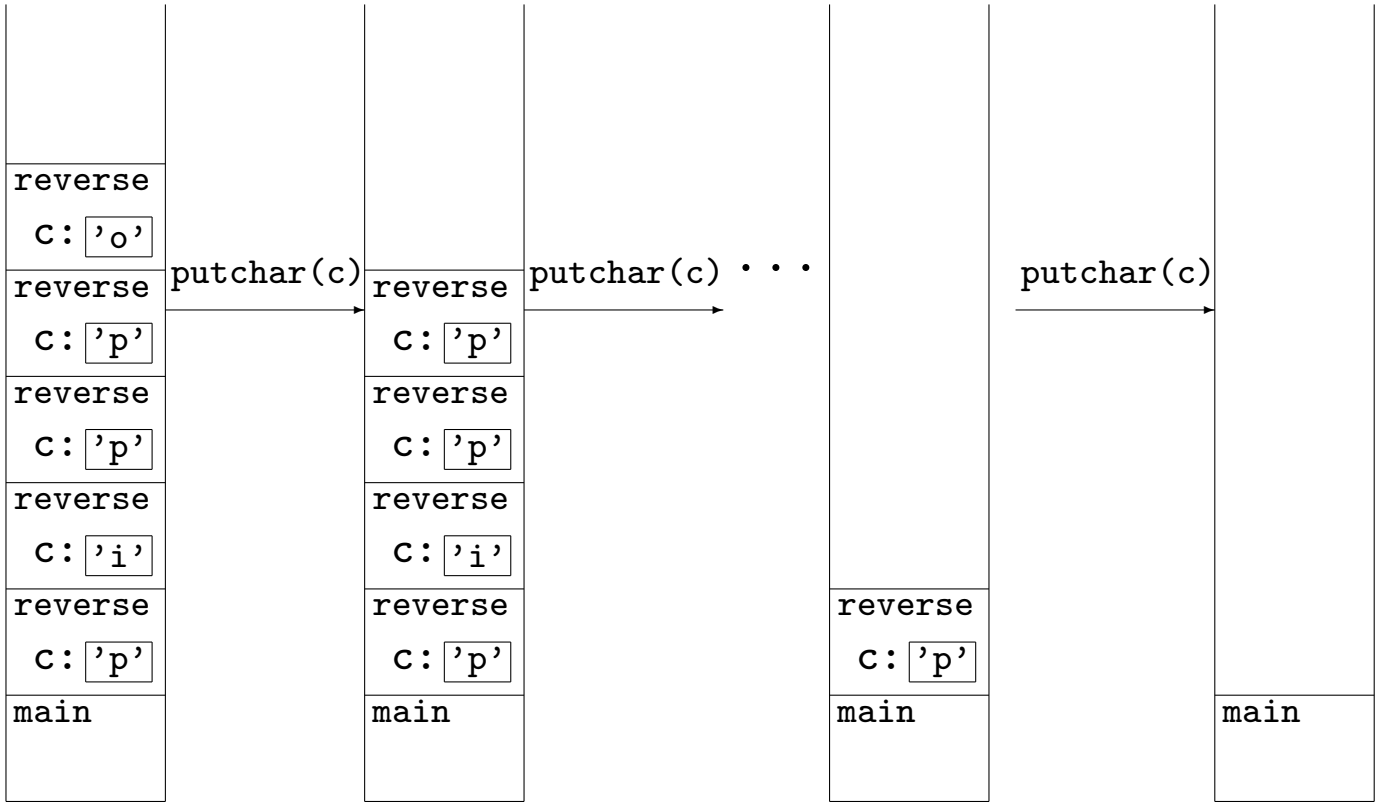
Provare la versione `reverse2.c` per visualizzare lo stack dei record di attivazione (limitatamente al valore di `c`).

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**.