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*:

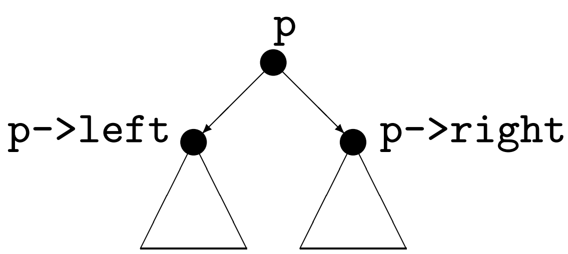
```
void forfewtimes(int i)
{
    static int k = 0;

    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.

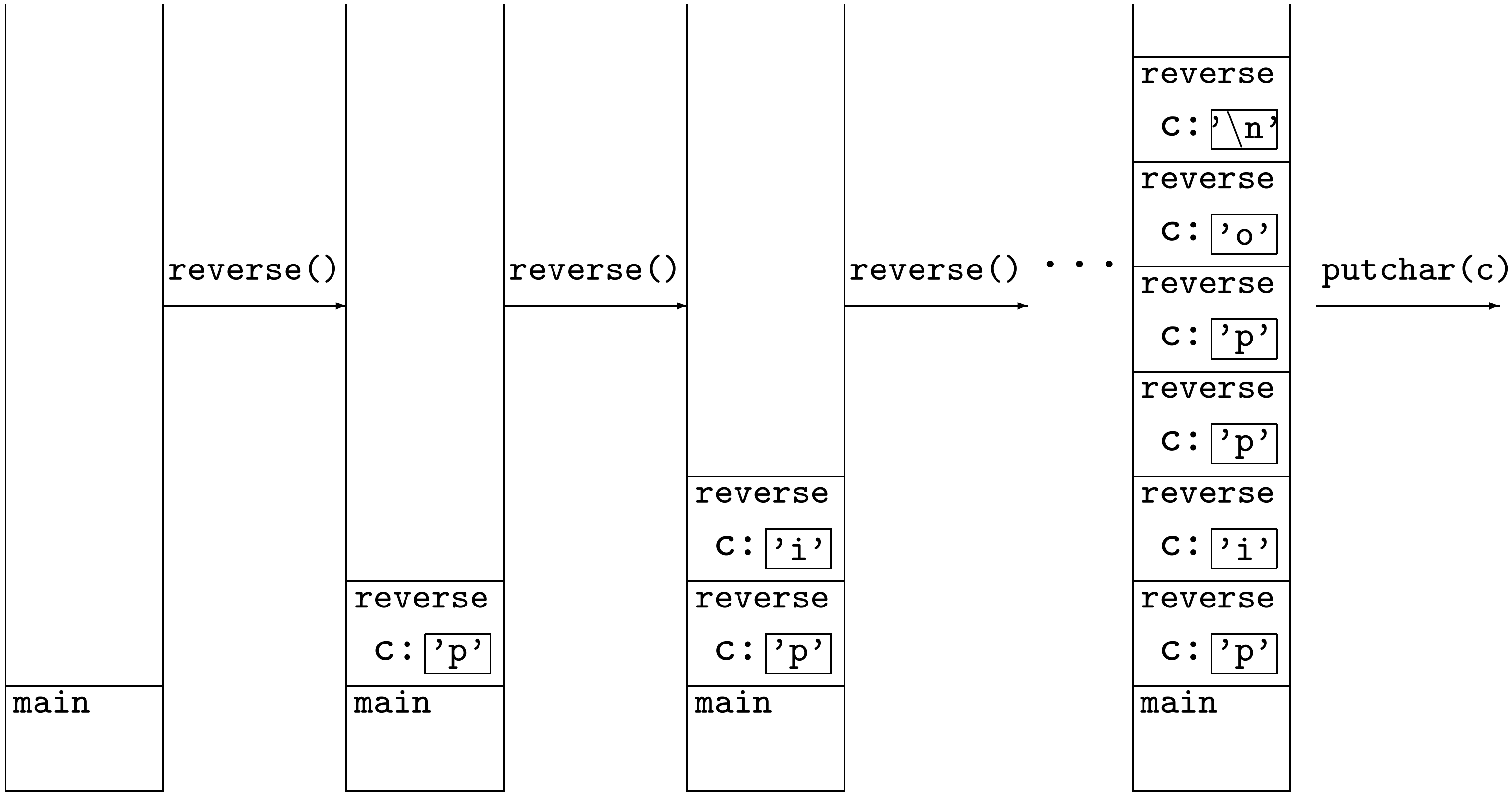
Ogni invocazione di `reverse` termina stampando sul video il carattere letto.

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'`.

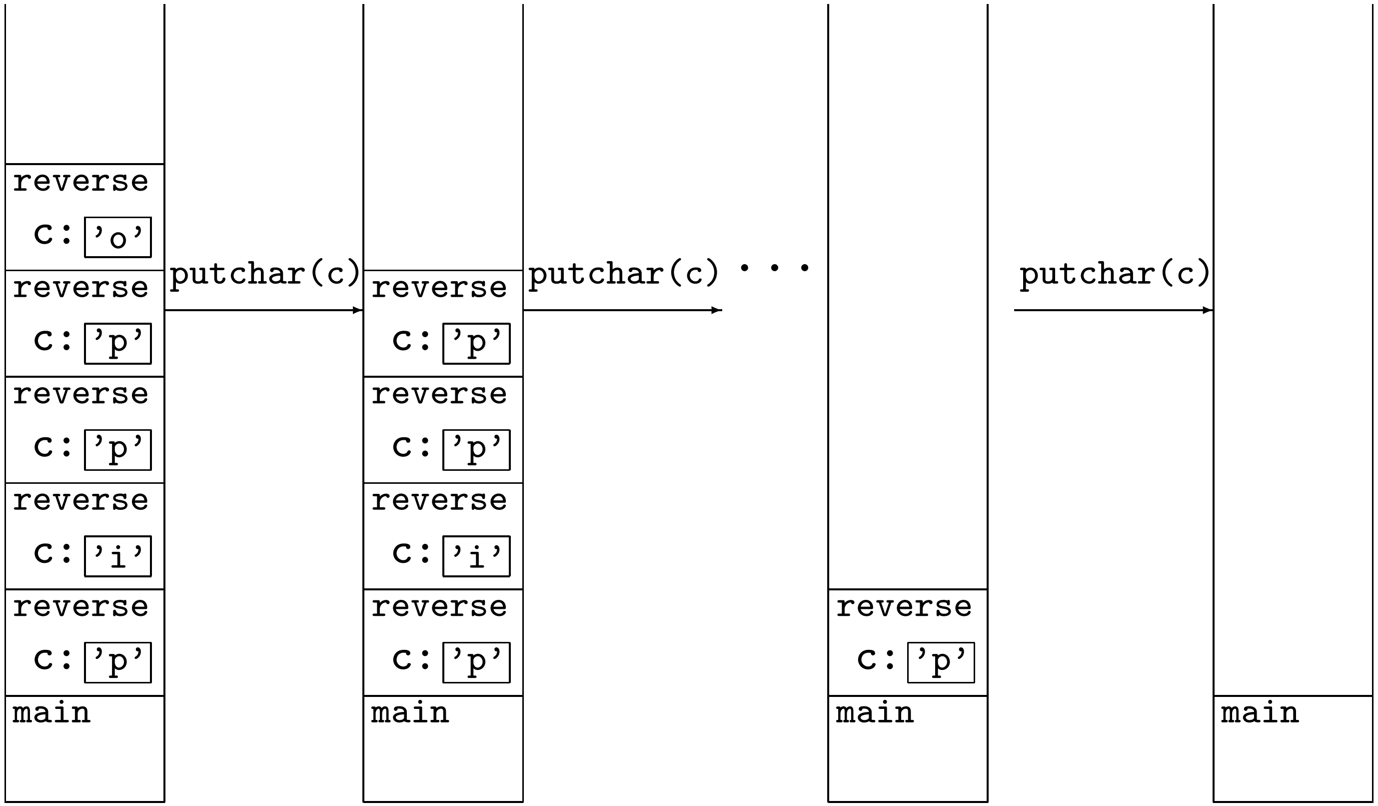
Provare la versione `reverse2.c` per visualizzare lo stack dei record di attivazione (limitatamente al valore di `c`).

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:

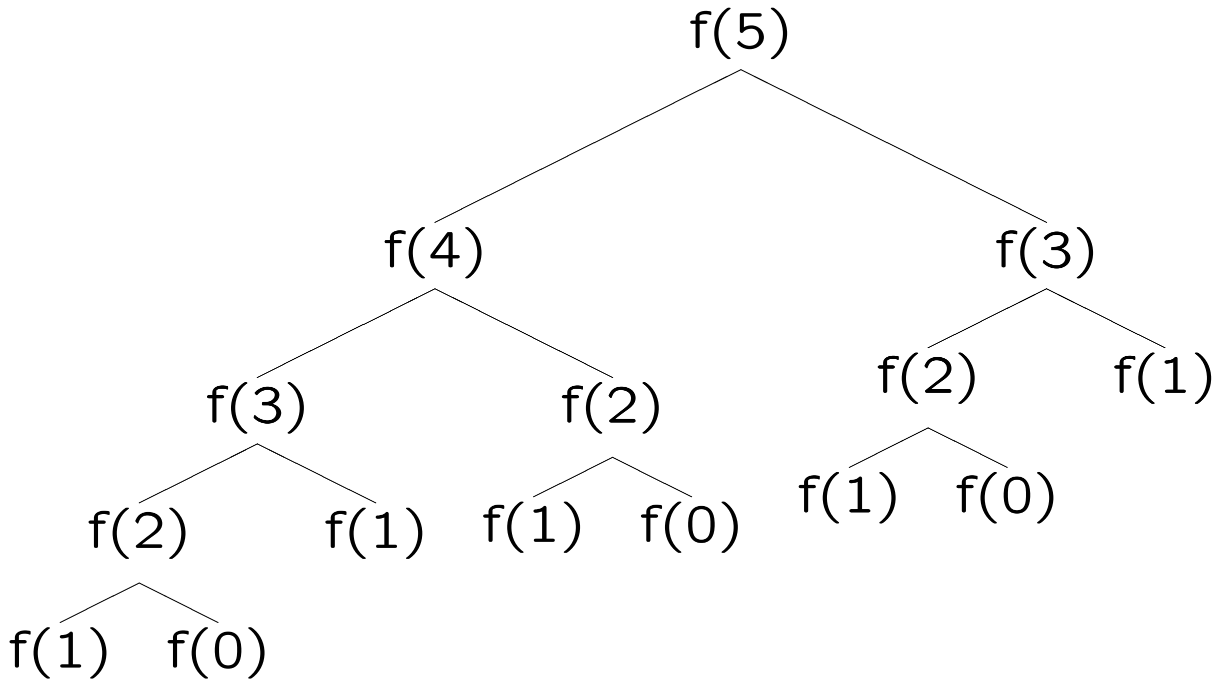
1: l'allocazione e deallocazione di numerosi record d'attivazione. Questo è il difetto ineliminabile delle funzioni ricorsive.

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.

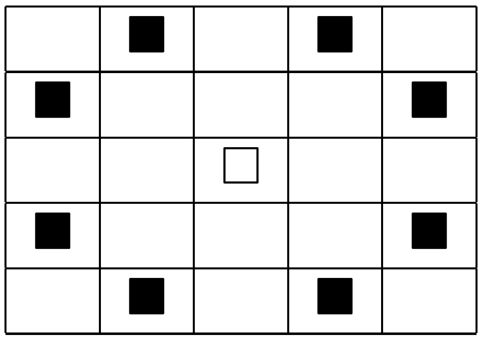
Spesso l'albero di ricerca può essere *potato*: l'algoritmo usa dei criteri (*euristiche*) per riconoscere in anticipo alcune delle situazioni in cui la visita di un cammino (o di un sottoalbero) non porta ad alcuna soluzione; ovviamente un tale cammino (o un tale sottoalbero) viene escluso dalla ricerca.

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

```c
#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');
        }
}
```

```c
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 $\quad *$*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`.

```c
#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);
```

```c
#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.