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(;;)
```

```c
#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 parola, 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_group_1
        ...
        default:
             statement
             ...
             statement
        ...
        case_group_k
}
```

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 restituire valori appartenenti a *type*:

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

Se la funzione non deve restituire 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.

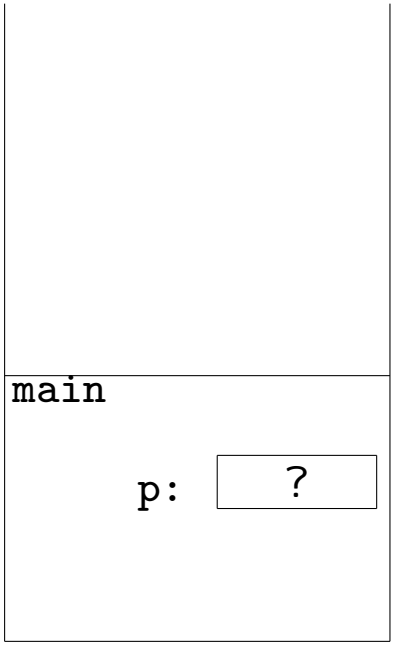
Se una funzione con tipo di ritorno diverso da `void` termina senza incontrare un'istruzione `return`, il valore di ritorno della funzione è indeterminato.

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

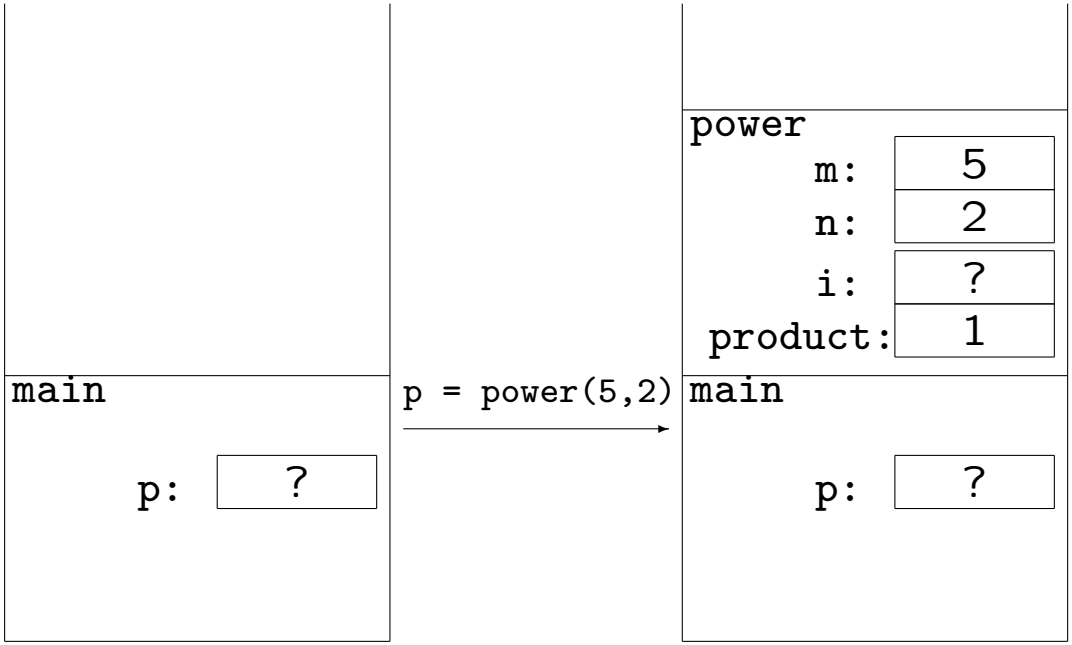
```
/* restituisce il numero di scambi effettuati) */
int bubble(int a[], int n, FILE *fw) { ... }

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

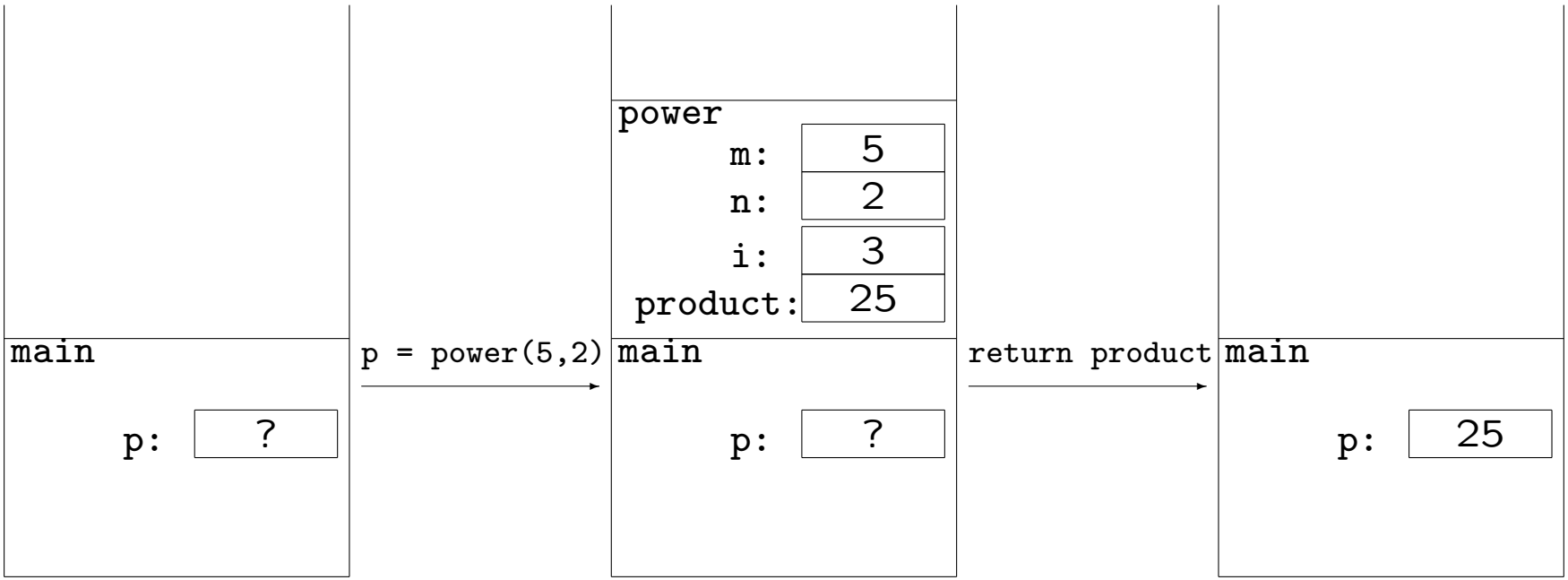
Invocazione di funzioni e stack dei record di attivazione:



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;
    }
}
```