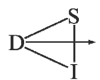
Introduzione al Linguaggio C

Parte 1

Prof. Alessandro Fantechi



Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze

Corso di Fondamenti di Informatica **Ingegneria Meccanica**

Indice

- 1. Elementi di base del Linguaggio C
  - Struttura di un programma C
  - Tipi di dati scalari
  - Dichiarazione di tipi di dati
  - Variabili
  - Operatori e operandi
- 2. Strutture di Controllo
  - Istruzioni iterative
  - Istruzioni condizionali
  - Salti

Testi di riferimento

- Un ottimo testo di base per apprendere i fondamenti della programmazione in linguaggio C è il seguente:

  “**Linguaggio C – guida alla programmazione”, seconda edizione**
  di *Alessandro Bellini, Andrea Guidi*
  Collana Workbooks, McGraw-Hill, 2003

- Il seguente testo è invece una guida completa al linguaggio C/C++, più adatto per programmatori esperti:

  “C++ La Guida Completa”, terza edizione
  di *Herbert Schildt*
  Collana Guide Complete, McGraw-Hill, 2003

- Infine, il seguente testo mette a confronto esempi di programmazione in linguaggio Pascal ed il corrispondente sorgente in C:

  “Pascal e C - Guida pratica alla programmazione”
  di *Marco Gori, Paolo Nesi, Elio Pasca*
  Collana Istruzione Scientifica, McGraw-Hill, 1996

Cenni storici

- Il linguaggio C venne sviluppato nel 1973 da Dennis M. Ritchie, degli AT&T Bell Labs, come linguaggio di programmazione di sistema
- Il linguaggio C doveva essere…
  - un linguaggio di livello sufficientemente alto per garantire ai programmi leggibilità e mantenibilità
  - …un linguaggio sufficientemente semplice da stabilire una corrispondenza immediata con la macchina sottostante
- Il linguaggio C si dimostrò così flessibile, ed il codice macchina prodotto così efficiente che, nel 1973, Ritchie e Ken Thompson riscrissero UNIX quasi completamente in C
- Nel *1977*, Ritchie e Brian Kernighan pubblicarono “The C Programming Language”, che formalizza lo *standard K&R*

Cenni Storici (cont.)

- Inizialmente il linguaggio C veniva usato soprattutto sui sistemi UNIX (PCC Portable C Compiler) ma, con la diffusione dei PC, compilatori C furono prodotti per nuove architetture e nuovi SO
- Col tempo, su diverse piattaforme, variazioni ed estensioni, introdotte dai diversi produttori, hanno finito col mettere a repentaglio la caratteristica prima del linguaggio: la portabilità del codice.
- Nel *1983*, l'*American National Standards Institute* (ANSI), costituì la commissione X3J11, che doveva formulare uno standard per il C, che includesse le nuove caratteristiche che il linguaggio aveva progressivamente maturato, mantenendone la portabilità
- La versione finale dello standard C fu approvata dall'ANSI nel 1989 che nel dicembre 1990 si traduce in uno standard ISO.

1. Elementi di base del linguaggio C

Componenti del Linguaggio

- Il nucleo si fonda su un ricco set di:
- Tipi di dati
  - Base
  - Derivati
- Strutture di controllo
  - Selezione (strutture decisionali)
  - Iterazione
- Costrutti di decomposizione del programma
  - Funzione
  - Unità di compilazione (modulo)
- Librerie Standard di corredo al compilatore
  - Standard I/O
  - Gestione stringhe
  - Gestione dinamica della memoria
  - Funzioni matematiche
  - Altre
- Si noti come, rispetto ad altri linguaggi, molte funzionalità siano rese disponibili come librerie "esterne" invece che come costrutti del linguaggio.

Componenti di un programma

- **Dichiarazione di variabili**
  In C, le variabili devono essere dichiarate prima di essere utilizzate
  Le dichiarazioni specificano il tipo e il nome di una variabile

  Esempio: **`int i`**;

- **Statement**
  Esempi:
  - **`printf("Enter a number:");`**
  - **`scanf(...);`**
  - **`result = sumup(n);`**
  - **`return 0;`**

  Il **;** è un terminatore di statement; deve essere aggiunto alla fine di ogni statement

Componenti di un programma (cont.)

- **Prototipi di funzione**
  Dicono al compilatore quale sarà il "formato" di una funzione
  Devono apparire prima che una funzione sia utilizzata
  Un prototipo di una funzione è diverso dalla sua definizione:
  - il primo descrive l' "interfaccia" della funzione
  - la seconda descrive l' "implementazione" della funzione

- **Definizione di funzione**
  Codifica l'algoritmo corrispondente ad un sottoprogramma (***subroutine***)

Componenti di un programma

- **Commenti:**
  Si affiancano a dichiarazioni di variabili o parti del programma per indicarne lo scopo e l'utilizzo;

  Esempio: `/* Questo è un commento */`
  Non possono essere nidificati

- **Parantesi graffe**
  Delimitano i ***blocchi di codice*** e costituiscono il costrutto primario di raggruppamento

Riferimento alle Librerie Standard

- Per ogni gruppo di funzioni esiste un file sorgente, chiamato *file header*, contenente le informazioni necessarie per utilizzare le funzioni
- I nomi dei file header terminano, per convenzione, con l'estensione ".h" (ad es., *stdio.h* è il file header dello standard I/O)
- Per includere un file header in un programma, occorre inserire nel codice sorgente l'istruzione

```
#include <nomefile.h>
```

Esempio:

- Per utilizzare *printf()*, che permette di visualizzare dati su terminale, è necessario inserire nel sorgente la linea di codice

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

- La direttiva *include* è una istruzione del linguaggio "C" ma è una rivolta ad un programma di espansione di macro noto come "preprocessore"

La funzione main

- Ogni programma eseguibile deve contenere una funzione speciale, chiamata ***main ()***, che indica il punto da cui inizia l'esecuzione del programma
- Le regole di scrittura della funzione ***main ()*** coincidono con quelle delle altre funzioni
- La funzione ***main ()*** può chiamare altre funzioni

Hello World

```
#include<stdio.h>

int main()

   {
     printf("Hello World\n");
     return 0;
   }
```

Include la libreria standard di I/O

Definisce una funzione ***main*** che non riceve alcun valore come argomento

main richiama la funzione di libreria ***printf*** per stampare la sequenza di caratteri specificata; ***n*** indica il ritorno a capo (*new line)*

Tipi di dati

- Il linguaggio C rende disponibili otto diversi tipi di numeri interi e tre tipi di numeri floating-point che, complessivamente, costituiscono i **tipi aritmetici**
- I tipi aritmetici, i **tipi enumerativi** ed i **puntatori** vengono detti ***tipi scalari***, poiché i valori che li compongono sono distribuiti su una ***scala lineare***, su cui si può stabilire una relazione di ordine totale
- Combinando i tipi scalari si ottengono i ***tipi composti*** che comprendono **array, strutture** ed **unioni**: servono per raggruppare variabili logicamente correlate in insiemi di locazioni fisicamente adiacenti
- Il tipo **void**, introdotto nello standard ANSI, non è né scalare, né composto e si applica, ad esempio, alle funzioni che non restituiscono valori

Tipi scalari

- **I tipi fondamentali sono:**
  - **char**: un singlo byte, che rappresenta uno dei caratteri del set locale;
  - **int**: un intero, il cui valore dipende dall'ampiezza degli interi sulla macchina utilizzata;
  - **float**: floating-point in singola precisione;
  - **double**: floating-point in doppia precisione.
- **In aggiunta, i qualificatori:**
  - **short**
  - **long**
  - **signed**
  - **unsigned**

  Esempio:
  ```
  short int indice;
  long int counter;
  ```

  L'ampiezza di un **int** riflette quella degli interi sulla macchina utilizzata (**short** equivale a 16 bit, **long** equivale a 32 bit). Ogni compilatore sceglie la dimensione appropriata.
- **Restrizioni:**
  - short >= 16 bit
  - int >= 16 bit
  - long >= 32 bit
  - short <= int <= long

Tipi scalari (cont.)

- I qualificatori **signed** e **unsigned** possono essere associati sia ai char che agli int.
  - unsigned char assume valori fra 0 e 255;
  - signed char assume valori fra –128 e 127.
- La dimensione di un tipo (o di una variabile) può essere determinata a runtime mediante la funzione **sizeof()**

Esempio:

```
int main()
   {
     printf("Size of double = %d byte(s)\n",
                              sizeof(long double));
     return 0;
   }
```

Gli header file standard `<limits.h>` e `<float.h>` contengono e costanti simboliche relative a tutte queste ampiezze (INT_MAX, ecc).

Tipi scalari

Type	Size
char, **unsigned char**, **signed char**	1 byte
short, **unsigned short**	2 bytes
int, **unsigned int**	4 bytes
long, **unsigned long**	4 bytes
float	4 bytes
double	8 bytes

Tipi scalari (cont.)

- I **tipi enumerativi** sono utili quando si vuole definire un insieme preciso di valori che possono essere associati ad una variabile

  Esempio:
  ```
  enum {red, blue, green, yellow} color;
  enum {bright, medium, dark} intensity;
  ```

- La sintassi per dichiarare i tipi enumerativi è introdotta dalla parola chiave enum, seguita dall’elenco dei nomi delle costanti fra parentesi graffe, e dal nome delle variabili
- Ai nomi delle costanti viene associato un valore intero di default, basato sulla loro posizione nell’elenco (a partire da 0)
- Tale valore può essere modificato esplicitamente indicando un valore diverso:
  ```
  enum {success=0, failure=1} status;
  ```
- Il compilatore ha il compito di allocare la memoria necessaria per un tipo enumerativo: a ***color*** dovrebbe essere allocato un singolo byte (solo quattro possibili valori)

Typedef

- Il C consente di associare ai tipi di dati nomi definiti dal programmatore, mediante la parola chiave **typedef**

  La sintassi è:

  typedef oldtype newtype

  Lo si usa per:
  - semplificare sintassi complesse (tipi strutturati)
  - creare tipi "portabili", che possano essere adattati all'architettura semplicemente cambiando gli header file

  Esempi:

```
typedef long int32;
typedef unsigned char byte;
```

  In combinazione con il preprocessore si possono fare cose più sofisticate tipo:

```
#ifdef (I386)
  typedef int32 int;
#endif
```


Tipo void

- Il tipo di dati `void` viene utilizzato per dichiarare funzioni che non restituiscono un valore:

```
void func(int a, int b)
{
      … …
}
```

- Il tipo void viene inoltre utilizzato per dichiarare puntatori generici, che verranno trattati successivamente.

Parte 2

Strutture di Controllo

Istruzioni e blocchi

- Ogni espressione diventa una **istruzione** quando è seguita da un punto e virgola, come in

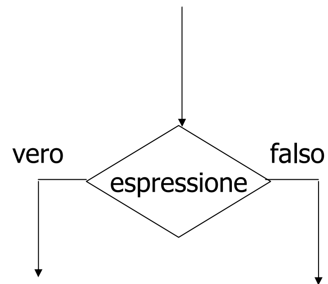
```
x = 0;
i++;
printf(…);
```

- Le parentesi graffe sono utilizzate per raggruppare dichiarazioni e istruzioni in un'unica **istruzione composta**, detta anche **blocco**. Sintatticamente, esse formano un'unica istruzione.
- La dichiarazione di variabili può avvenire dentro qualsiasi blocco.
- Le dichiarazioni vanno poste in testa al blocco medesimo "prima" di qualunque istruzione (i.e. non si possono annegare dichiarazioni di variabili nel codice di un blocco)

If-else

```
if (espressione)
     < BLOCCO_1 >
else
     < BLOCCO_2 >
```

opzionale (else / < BLOCCO_2 >)



dove la parte associata all'**else** è opzionale.

- L'espressione è valutata; se è vera (cioè se assume un valore non nullo) è eseguita la sequenza di istruzioni contenute in **BLOCCO_1**.
- Se è falsa (cioè se ha valore nullo), e se esiste la parte **else**, è eseguita la sequenza di istruzioni contenute in **BLOCCO_2**.
- In caso di ambiguità (mancanza di un else), questa viene risolta associando ogni else all'if più interno che ne è privo.

If-else

Esempio:

```
if (n > 0)
     if (a > b)
          z = a;
     else
          z = b;
if (n > 0)
{
     if (a > b)
          z = a;
}
else
     z = b;
```

If-else (cont.)

Esempio sbagliato:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0)
            {
            printf(…);
            return i;
            }
else
    printf("n è negativo");
```


Else-if

```
if (espressione)
      istruzione
else if  (espressione)
      istruzione
else if  (espressione)
      istruzione
else
      istruzione
```

opzionale (riferito a: else istruzione)

è il modo generale di realizzare una scelta plurima

- Le **espressioni** sono valutate nell'ordine in cui si presentano;

  se una di esse risulta vera, **l'istruzione** corrispondente è eseguita, terminando la catena di else-if.

Else-if (cont.)

```
int binsearch (int x, int v[ ], int n)
{
  int low, high, mid;
  low = 0;
  high = n - 1;
  while (low <= high)
     {
      mid = (low+high) / 2;
      if (x < v[mid])
          high = mid - 1;
      else if (x > v[mid])
          low = mid + 1;
      else
          return mid;
     }
  return -1;
}
```


Switch

```
switch (espressione)
  {
    case espressioni-costanti :
        istruzioni
    break;
    case espressioni-costanti :
        istruzioni
    break;
    default :
        istruzioni
    break;
  }
```

ogni caso possibile è etichettato da un insieme di costanti intere e/o di espressioni costanti.

Switch

- Se il valore di **(espressione)** coincide con uno di quelli dei vari casi, allora l'esecuzione inizia dal blocco di istruzioni relative a quel caso.
- Se il valore di **(espressione)** non coincide con nessuno di quelli dei vari casi, allora sono eseguite le istruzioni di **default**.
- **default** e' opzionale.
- L'istruzione **break** provoca l'uscita immediata dallo switch. Se non è presente, l'esecuzione dei casi è sequenziale.

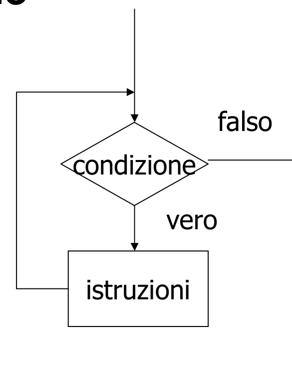
Switch (cont.)

Esempio:
frammento di programma per contare le occorrenze di ogni cifra, spazio, e degli altri caratteri (usando il costrutto switch)

```
…
while ((c = getchar( )) != '*')
    {
       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;
            }
    }
...
```

Ciclo while

```
while (espressione)
   {
  …
  istruzioni
  …
   }
```



- L'**espressione** è valutata;

  se il suo valore è vero, è eseguito il blocco **istruzioni** specificato, e l'espressione è valutata nuovamente;

  se l'espressione è falsa, l'esecuzione riprende dalla prima istruzione non controllata dal **while**.

Ciclo for

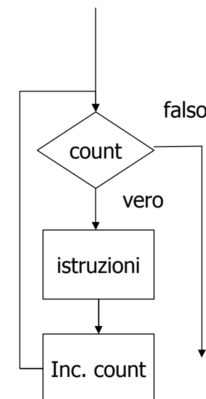
```
for (espr_1; espr_2;
espr_3)
   {
 …
 istruzioni
 …
   }
```

equivale a

```
espr_1;
while (espr_2)
  {
    …
    istruzioni
    …
    espr_3;
  }
```

- **Utilizzo generico del ciclo for:**

```
for (count=init_value;
       count<end_value;
       count++)
    {
       < sequenza
          istruzioni >
    }
/* compie un certo numero
   di iterazioni da
   init_value a end_value */
```



- In generale:
  - *espr_1* ed *espr_3* in un ciclo for sono assegnamenti o chiamate di funzioni;
  - *espr_2* è una espressione relazionale;
  - tutte possono mancare (ma non i ';'): `for(;;)` `< ciclo infinito >`

Ciclo for

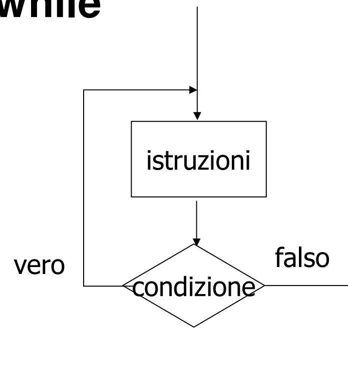
- **Nota:**
  - l'indice e la condizione limite di un ciclo **for** del C **possono** essere cambiati nel corpo del ciclo;
  - l'indice **conserva** il suo valore anche dopo l'uscita dal ciclo;

Esempio:

```
#include <ctype.h>
int alpha2int (char s[ ])
 {
   int       i, n, sign;
   for (i = 0; isspace(s[i]); i++)
      /*NOP*/;
   sign = (s[i] == '-') ? -1 : 1;
   if (s[i] == '+' || s[i] == '-')
      i++;
   for (n = 0; isdigit(s[i]); i++)
      n = 10 * n + (s[i] - '0');
   return sign * n
 }
```


Ciclo do…while

```
do
   {
     …
     istruzioni
     …
   }
while (espressione)
```



- E' eseguito il blocco **istruzioni**, e successivamente l'**espressione** è valutata;
  se è vera, il blocco contenuto nel costrutto **do** è eseguito nuovamente;
  quando l'espressione diventa falsa, il ciclo termina.

- Questo ciclo è equivalente al ciclo repeat-until del linguaggio Pascal.

Break e continue

- L'istruzione **Break** serve per uscire da un ciclo senza controllare la condizione di terminazione.
  Break provoca l'uscita incondizionata da un **for**, da un **while**,
  da un **do**, da uno **switch**.

  Esempio:

```
/* trim: rimuove dalla fine di una stringa gli spazi, i caratteri di
tabulazione e i newline */

int trim (char s[ ])
  {
     int n;
     for (n = strlen(s) -1; n >= 0; n--)
        {
         if((s[n] != `  `) &&
            (s[n] != `\t') &&
            (s[n] != `\n'))
             break;
        }
     s[n+1] = `\0';
     return n;
  }
```


Break e continue

- L'istruzione **continue** forza l'inizio dell'iterazione successiva di un **for**, di un **while**, di un **do**; cioè provoca l'esecuzione immediata della parte di controllo di un ciclo.
- Nel caso di un **for**, il controllo passa alla parte di incremento.
- **continue non** è **applicabile** a **switch**.

Goto e label

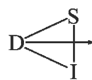
- Il C fornisce l'istruzione **goto** (e le **label**) per poter ramificare l'esecuzione.
- L'istruzione **goto introduce non-strutturazione** nel codice perché devia l'esecuzione dal normale flusso determinato dalle strutture di controllo (generalmente, si tende a limitarne l'uso).
- Unico utilizzo "nobile": *gestione degli errori*

```
for (…)
     for (…)
        {
            …
            if (errore_irreversibile)
            goto error;
        }
…
error: { gestione_errore }
```


Introduzione al Linguaggio C

Parte 2

Ing. Emilio Spinicci



Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze

Indice

- 3. Processo di compilazione
  - Preprocessore
  - Compilazione
- 4. Decomposione del programma, Variabili e operatori
  - Variabili
  - Funzioni
    - Printf
    - Scanf
  - Costanti
  - Operatori e operandi
  - Esempi di programmazione
- 5. Strutture dati complesse
  - Array, stringhe
  - Puntatori
  - Struct
- 6. Sommario delle principali librerie di sistema
  - Gestione dell'input/output
  - Gestione delle stringhe
  - Allocazione dinamica della memoria
  - Gestione dei File
  - Funzioni matematiche

3. Processo di compilazione

Preprocessore

- Il preprocessore C modifica il codice sorgente in base a "direttive del preprocessore" inserite nel codice
- le direttive del preprocessore iniziano con il carattere '#'
  il preprocessore crea un nuovo "codice sorgente" che è il testo che viene effettivamente compilato
- questo "codice intermedio" normalmente non viene mandato in output
- è possibile forzare il compilatore a dare in output questo codice per vedere cosa produce

Preprocessore (cont.)

- La direttiva #include prevede due varianti:
  `#include <nomefile>`
  `#include “nomefile”`
  tale direttiva include il contenuto di un file
- il file incluso può contenere a sua volte direttive **`#include`**
- nella variante **<nomefile>** ricerca il file solo negli include delle librerie standard (/usr/include)
- nella variante **“nomefile”** il file nella directory corrente e quindi negli altri path.
- è possibile specificare path di ricerca aggiuntivi per gli include file tramite opzioni di compilazione

Preprocessore (cont.)

- La direttiva **`#define`** può essere usata per definire macro sostituzioni.

```
#define text1 text2
```

chiede al preprocessore di trovare tutte le occorrenze di **`text1`** e di sostituirle con **`text2`**.

Esempio:

```
#define MAX            1000
```

- Si può usare per macro più complesse (con parametri):

```
#define square(x)            ((x)*(x))
```

- La direttiva **`#undef`** rimuove la definizione di una certa macro.

Preprocessore (cont.)

- Le direttive **`#if`**, **`#elif`**, **`#else`**, **`#endif`** informano il preprocessore di compilare solo alcune parti del codice, al verificarsi o meno di opportune ***condizioni***; possono essere utilizzate per creare codice più efficiente e più portabile:

```
#if condition_1
source_block_1
#elif condition_2
source_block_2
...
#elif condition_n
source_block_n
#else
default_source_block
#endif
```

Preprocessore (cont.)

- E' possibile testare “il valore” di macro definite con #define
  Esempio

```
#define ENGLAND       0
#define ITALY   1

#if ENGLAND
#   include "england.h"
#elif ITALY
#   include "italy.h"
#else
#   include "default.h"
#endif
```


Preprocessore (cont.)

- E' possibile testare l’esistenza, a prescindere dal suo valore di una macro con la direttiva

```
#ifdef(macroname) / #ifndef(macroname)

#ifdef(DEBUG)
      some useful debug code
#endif
```

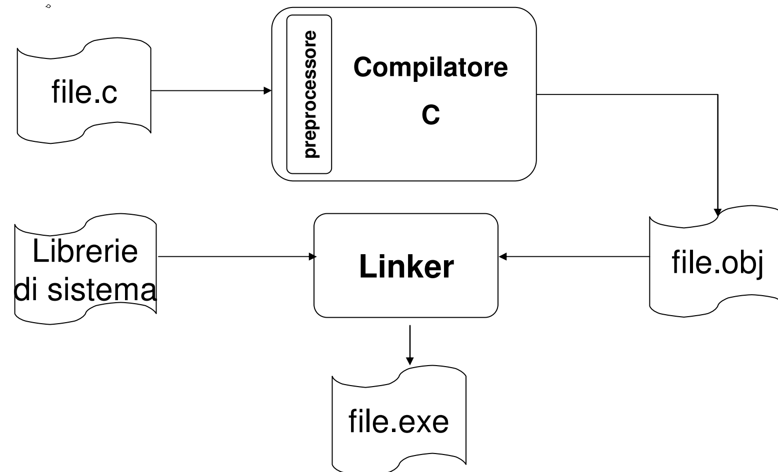
oppure con l’espressione `#if defined(macroname)`

```
#if defined(DEBUG)
      some useful debug code
#endif
```

- Uso tipico negli header files per prevenire la doppia inclusione, ad esempio nell’header file `myheader.h` si può scrivere:

```
#if !defined(_myheader_h_included_)
#define _myheader_h_included_
…
  contenuto dell’header file
..
#endif
```

Processo di compilazione



Per compilare

- La compilazione di un programma può avvenire in maniera automatica all'interno di un ambiente di sviluppo integrato (Microsoft VisualC++, DevC++), oppure da riga di comando;
- Principali compilatori in commercio:
  - **cl (Microsoft Visual C++)**
  - **bcc (Borland C++)**
  - **gcc (GNU C compiler, UNIX)**
- Riga di comando per generare il file oggetto:
  - **cl /c /TC /I<include path> /Fo<file>.obj <file>.c**
- Riga di comando per generare il file eseguibile:
  - **link <file>.exe <file1>.obj … <file*n* >.obj <lib1>.lib … <lib*n*>.lib**

4. Decomposione del programma, Variabili e operatori

Variabili

- **Dichiarazione di variabili**
  forma generica:
  ```
  typename varname1, varname2, ...;
  ```
  dichiara le variabili `varname1, varname2, ...`

  e gli associa il tipo `typename`

  meglio separare le dichiarazioni su più righe e magari commentarle:
  ```
  typename varname1; /* Comment */
  typename varname2; /* Comment */
  …
  ```

Variabili: nomi di variabile

- **I nomi di variabile**
  - possono contenere lettere, numeri e underscore ('_')
  - il primo carattere deve essere una lettera o un underscore
  - il linguaggio C è case-sensitive ('a' diverso da 'A')
  - le keyword del linguaggio non possono essere usate come nomi di variabili
  - si sconsiglia inoltre di utilizzare variabili che iniziano per underscore, usati spesso per variabili di sistema o del compilatore, e nomi maiuscoli comunemente usati per le costanti.

Funzioni

- E' possibile scomporre un programma in più sottoprogrammi o ***funzioni***, ognuna delle quali realizza (***implementa***) una parte del programma principale.
- Una funzione accetta dei dati in ingresso (i ***parametri***), li elabora secondo l'algoritmo di competenza del sottoprogramma che rappresenta, e restituisce un risultato.

- Formato della definizione di funzione:

```
<tipo ritornato> <nome funzione> (<lista param.>)
{
    <dichiarazioni>
    <istruzioni>

    return risultato;

}
```

Funzioni

Esempio: definire l'operatore esponenziale fra interi utilizzando una funzione.

```
#include <stdio.h>
int power(int m, int n);

int main ( )
  {
    int i;
    for (i = 0; i < 10; ++i)
     printf("%d %d %d\n", i, power(2, i), power(-3, i));
    return 0;
  }

int power(int base, int n)
  {
    int i, p;
    p=1;
    for (i=0; i<=n; ++i) p = p*base;
   return p;
  }
```

prototipo della funzione, in accordo con la definizione e l'uso della funzione

utilizzo della funzione

definizione della funzione

Valore ritornato dalla funzione

Funzioni

- La riga del main
  `printf("%d %d %d\n", i, power(2, i), power(-3, i));`

  contiene due chiamate alla funzione power, che è di tipo intero, come sono interi i due argomenti ad essa passati

- La prima linea della funzione power
  int power(int base, int n)

  dichiara i nomi ed i tipi dei parametri, e il tipo del valore restituito dalla funzione; i nomi usati dalla funzione sono locali a **power** stessa, e quindi non sono visibili a nessun'altra funzione. Lo stesso accade per **i** e **p** dichiarati nella funzione.

  - parametro (o argomento formale) ↔ variabile che compare nella lista relativa alla definizione della funzione
  - argomento (o argomento reale) ↔ valore che è usato nella chiamata della funzione stessa
  - il valore calcolato da una funzione viene restituito (opzionale) al main attraverso l'istruzione **return**, seguita da una espressione.
  - i parametri all'interno del prototipo sono opzionali; quindi si può scrivere

int power(int , int)

La funzione printf

- La funzione ***printf* ( )** visualizza sul dispositivo **stdout** (il display) messaggi e contenuto di variabili: **printf(< stringa di formato >, < argomenti >)**
- Può ricevere un numero variabile di argomenti: il primo argomento è un parametro speciale, detto ***stringa di formato***, che specifica il messaggio da visualizzare; gli altri argomenti sono le variabili che contengono i dati che si vuole stampare sullo schermo.
- La stringa di formato è racchiusa fra doppi apici e può contenere ***testo*** e ***specificatori di formato*** → sequenze speciali di caratteri che iniziano con il simbolo di percentuale (**%**) ed indicano le modalità di scrittura di un singolo dato

  Esempio: nell’istruzione

```
printf("Il valore di num è %d", num);
```

  - “Il valore di num è %d” è la stringa di formato;
  - ***%d*** è lo specificatore di formato per gli interi decimali

  num è la variabile intera decimale da stampare

La funzione printf

- Esistono specificatori anche per altri tipi di dati:

%c	dato di tipo carattere
%f	dato di tipo Floating-point
%.< *n* >f	dato reale con le prime 4 cifre decimali dopo la vrigola
%s	stringa (array di caratteri terminato da *null*)
%o	intero ottale
%x	intero esadecimale

La stringa di formato può contenere un numero qualunque di specificatori di formato, ma il loro numero deve coincidere con il numero dei dati da stampare, passati come argomenti

La funzione scanf

- La funzione ***scanf* ()** legge dati introdotti tramite il dispositivo **sdtin** (la tastiera) in opportune variabili: **scanf(< stringa di formato >, < argomenti >)**
- ***scanf ()*** può ricevere un numero qualunque di parametri preceduti da una stringa di formato
- I dati di ***scanf()*** devono essere ***lvalue*** (ovvero riferiti ad indirizzi di memoria), e devono pertanto essere preceduti dall'operatore indirizzo **&**

Esempio:

```
scanf("%d",&num);
```

richiede al sistema di leggere un intero da terminale e di memorizzare il valore nella variabile ***num***

Printf & Scanf

- **Esempio: calcolo dell'area di un rettangolo**

```
…
 float a,b,area;

 printf("Inserire base e altezza: ");
 scanf("%f %f", &a, &b); area=(a*b);
 printf("\nArea = %.4f", area);
…
```

Messaggio visualizzato:

Inserire base e altezza: **2 5**   **< separati dallo spazio! >**

Area = **10.0000**

Chiamata per valore

- Gli argomenti forniti ad una funzione del C sono passati per valore, a differenza del Fortran (chiamata per riferimento) e del Pascal (parametri preceduti da var).
- In C, la funzione chiamata non può alterare direttamente una variabile nella funzione chiamante, ma solo modificarne una copia, privata e temporanea.
- E' possibile fare in modo che una funzione modifichi una variabile all'interno della routine chiamante. Il chiamante deve fornire l'indirizzo della variabile (il puntatore), mentre la funzione chiamata deve dichiarare il parametro come un puntatore con il quale accedere alla variabile
- Diversamente, se l'argomento è il nome di un vettore (un *array*), il valore passato alla funzione è l'indirizzo del vettore, non i suoi elementi.

Variabili (cont.)

- **Variabili locali vs variabili globali**
  Le variabili dichiarate fuori da ogni blocco (funzione) sono variabili globali
  Le variabili dichiarate all'interno di un blocco sono variabili locali a quel blocco

- **Visibilità:**
  Le variabili locali sono visibili solamente all'interno del blocco in cui sono dichiarate
  Le variabili globali sono visibili a partire dal punto in cui sono dichiarate, per tutto il resto del file
  Una dichiarazione di variabile locale nasconde le dichiarazioni con lo stesso nome definite all'esterno del blocco

Variabili (cont.)

Variabili esterne

E' possibile dichiarare una variabile globale in qualsiasi parte di un programma, anche in un file separato

In questo caso, bisogna distinguere fra:

- dichiarazione "effettiva": con la solita sintassi
- dichiarazione "extern": che prevede l'uso del qualificatore *extern* per informare il compilatore che ci si riferisce ad una variabile dichiarata altrove.

Esempio:

```
extern int var;
```


Variabili: classi di memoria

- **Le variabili locali sono automatiche:**

  esistono solo mentre il programma è in esecuzione nel range del blocco in cui sono definite ovvero:
  - quando l'esecuzione del programma entra nel blocco in cui la variabile è definita: la variabile viene creata
  - quando l'esecuzione del programma esce dal blocco in cui la variabile è definita: la variabile viene distrutta
  - nel caso l'esecuzione del programma esca da un blocco, per poi rientrarci, la variabile è da considerare come una nuova istanza

- **Variabili statiche**

  A volte è importante prolungare la vita di una variabile locale oltre la durata dell'esecuzione della funzione (o del blocco) che la contiene

  La variabile locale deve allora essere marcata ***static*** e diviene quindi "non automatica", ovvero, non viene distrutta dopo l'uso

Variabili: classi di memoria

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

void bogus()
{
   int x;
   print("%d ",x);
   x = 10;
}

void bogus2()
{
   int y;
   y = rand();
   print("%d ",y);
}

int main()
{
   int i;
   for(i=0;i<5;i++)
   {
     bogus();
     bogus2()
   }
   return 0;
}
```

Non inizializzata, contiene un valore non definito

Il valore viene perso all'uscita dal blocco valore

Entrambe stampano valori casuali!

Costanti

- E' possibile dichiarare variabili come costanti utilizzando il qualificatore **`const`**

  Esempio:

  ```
  const int maxlength = 2356;
  const int val = (3*7+6)*5;
  const double pi = 3.1415926;
  ```

- Il qualificatore **const**:
  - informa chi legge il codice che un certo valore non cambia
  - semplifica la lettura di codice di grande dimensione
  - informa il compilatore che un valore non cambia
  - il compilatore può ottimizzare il codice oggetto tenendo conto di questo fatto

Costanti (cont.)

- Il qualificatore **`const`** non era presente nella definizione originale del linguaggi e fu introdotto con lo standard ANSI.
- Fino ad allora per definire costanti si utilizzava il preprocessore C

  Esempio:
  ```
  #define  PI       (double) 3.14159
  ```
  Questa forma è ancora molto usata.

Costanti

- Una costante intera 1234, è un **int**.
- Una costante floating-point 123.4 o 1e-2 è di tipo **double**.
- Una costante **long** è seguita da una ‘l’ o da una ‘L’ terminali, come 123456789L.
- Le costanti **unsigned** sono seguite da una ‘u’ o da una ‘U’.
- ‘ul’ e ‘UL’ indicano gli **unsigned long**.
- I suffissi ‘f’ e ‘F’ indicano una costante **float**
- Un intero può essere scritto in
  - decimale: es. **31**
  - ottale: con uno **0** preposto, es. **037**),
  - esadecimale: con un **0X** o **0x** preposto, es. **0x1f** o **0x1F**).
- Esempio:
  `0xFUL` è una costante unsigned long con valore decimale 15.

Costanti (cont.)

- Una **costante carattere** è un intero, scritto sotto forma di carattere racchiuso fra apici singoli, come **‘A’**
- Il valore di una costante carattere è il valore numerico di quel carattere all’interno del set della macchina. Ad esempio, nel codice ASCII la costante carattere **‘0’** ha valore **48**, che non ha nessun legame con il valore numerico 0.
- Un’**espressione costante** è un’espressione costituita da sole costanti. Sono valutate al momento della compilazione, invece che run-time, e quindi possono essere inserite in ogni posizione nella quale può trovarsi una costante.

  Esempio:

```
#define MAXLINE 1000
char line[MAXLINE+1]
```

  oppure

```
#define LEAP 1
int days[31+28+LEAP+31+30+31+30….]
```


Costanti (cont.)

- Una **stringa costante**, o **costante alfanumerica**, è una sequenza di zero o più caratteri racchiusi fra doppi apici.

  Tecnicamente, una stringa costante è un vettore di caratteri.

  Poiché ogni stringa termina con il carattere nullo ‘\0’ (nella sua rappresentazione interna), la memoria richiesta per rappresentarla è pari al numero dei caratteri della stringa più uno.

  Nel’header file <string.h> sono definite le funzioni per il trattamento delle stringhe.

- Le **costanti enumerative** sono del tipo

```
enum boolean {FALSE, TRUE}
```

  il primo nome ha valore 0, il secondo 1, e così via, se non sono specificati valori espliciti.

```
enum mesi {GEN=1, FEB,MAR,APR,MAG,GIU,LUG,AGO}
```

  i nomi in enumerazioni diverse devono essere distinti. All’interno di un’unica enumerazione i valori non devono essere necessariamente distinti.

Operatori aritmetici

- Gli operatori aritmetici binari sono:
  +, **-**, *****, / *(divisione intera nel caso di operandi* `int`*)*,
  % *(operatore modulo, restituisce il resto della divisione intera)*.

  Esempio:

```
if ((year % 4 == 0 && year % 100 !=0) ||
     (year % 400 == 0))
   printf("il %d è bisestile", year);
else printf("il %d non è bisestile", year);
```

- L'operatore **%** non è applicabile ai float e ai double.
  Tutti gli operatori aritmetici sono associativi da sinistra a destra.

  Precedenza operatori:
  +, **-** *binari* **<** *, /, **%** **<** **+**, **-** *unari*

Operatori relazionali e logici

- Gli <u>operatori relazionali</u> sono:
  > **>=** **<** **<=**

- Gli <u>operatori di uguaglianza</u> sono:
  == (*uguale a*) **!=** (*diverso da*)

- Gli operatori relazionali hanno precedenza maggiore di quelli di uguaglianza, e precedenza inferiore agli operatori aritmetici.

  Esempio:
  i < lim-1 equivale a i < (lim-1)

Operatori relazionali e logici

- Gli <u>operatori logici</u> sono:

  && *(and)* || *(or)*

  Hanno precedenza maggiore rispetto agli operatori relazionali.

  La valutazione di espressioni connesse da && e|| avviene da sinistra a destra, e la valutazione si blocca non appena si determina la verità e falsità dell'intera espressione.

  Esempio:

```
for(i=0; ((i < lim-1) && (c=getchar())) != '\0'; ++i) s[i]=c;
```

  se il test (i < lim - 1) fallisce, non serve controllare il restante pezzo dell'operatore &&.

- <u>Per definizione</u>, il valore numerico di una espressione logica o relazionale è **1** se la relazione è vera, **0** se essa è falsa.
- <u>L'operatore unario di negazione</u>, !, converte un operando non nullo in uno 0, e un operando nullo in un 1.

Conversioni di tipo

- Con un operatore con operandi di tipo differente, gli operandi vengono convertiti in tipi comuni (in genere, da un tipo "più piccolo" a un tipo "più grande").
- Espressioni che provocano perdita di informazione producono un messaggio di **warning**, ma non sono illegali (ad esempio, assegnare un intero a uno short o un float ad un intero).
- Espressioni prive di senso non sono consentite (ad esempio, utilizzare un float come indice).

  - **Osservazione**: le variabili di tipo char sono numeri, ma lo standard non specifica se con o senza segno.

    Quando un char è convertito in un int, il risultato <u>può essere negativo</u>, a seconda dell'architettura della macchina.

Conversioni di tipo (cont.)

- Regole di conversione aritmetica (se non ci sono operandi unsigned):
  - ogni operando di tipo “inferiore” è convertito nell’operando di tipo “superiore” presente nella stessa espressione;
  - il risultato è di tipo “superiore”;
  - il valore del lato destro di un assegnamento è trasformato nel tipo del valore di sinistra, cioè nel tipo del risultato;
- E’ possibile forzare la conversione tramite l’operatore (unario) di ***casting***; nella costruzione

  (*nome del tipo***)** *espressione*

  l’espressione è convertita nel tipo specificato, utilizzabile successivamente. Ad esempio, la funzione di libreria ***sqrt***, che calcola la radice quadrata, deve operare su un argomento double. Se **n** è un intero, si può scrivere

  ```
  sqrt((double)n)
  ```

  per convertire il valore di n da intero a double, e poi applicare sqrt.

Operatori di incremento e decremento

- operatore di incremento: **++**
  operatore di decremento: **--**
  - **notazione prefissa**: la variabile è incrementata prima di essere usata nell’espressione;
  - **notazione postfissa**: la variabile è incrementata dopo il suo utilizzo nell’espressione;

  Esempio:
  se **n** è un **intero di valore 5**,

  ```
  x = n++; /*assegna a x il valore 5*/
  x = ++n; /*assegna a x il valore 6*/
  ```

  in entrambi i casi n assume valore 6.

Operatori bit a bit

- Il C fornisce sei operatori, applicabili ad operandi interi (char, short, int, long), con o senza segno:
  - **&** AND bit a bit
  - | OR inclusivo
  - **^** OR esclusivo
  - **<<** shift a sinistra
  - **>>** shift a destra
  - ~ complemento a uno (unario)
- L'operatore di AND bit a bit è spesso usato per azzerare insiemi di bit; l'operatore OR è spesso usato per attivare insiemi di bit.

Operatori bit a bit (cont.)

Esempio:

- `n = n & 077`

  azzera tutti i bit di n, esclusi i 6 meno significativi, mentre:
- `x = x | set_on`

  pone a uno, in x, i bit che sono ad uno nella variabile **set_on**.

- Gli operatori di shift **<<** e **>>** spostano verso sinistra e verso destra (rispettivamente) il loro operando sinistro di un numero di bit pari al valore dell'operando destro.
  - `x << 2` sposta a sinistra di 2 posizioni il valore di x
- L'operatore unario ~ produce il complemento a uno di un intero.
  - `x = x & ~077` pone a zero gli ultimi sei bit di x

Operatori bit a bit

- Gli operatori sui bit **&** e **|** sono diversi dagli operatori logici **&&** e **||**:
  Se **x** vale **1** e **y** vale **2**, allora
  - `x&y` vale **0**
  - `x&&y` vale **1** (ovvero ***true***).

Operatori di assegnamento ed espressioni

- Gli operatori binari:

  +, **-**, *, /, **%**, **<<**, **>>**, **&**, **^**, **!**

  hanno il corrispondente operatore di assegnamento '***op*=**'
  dove op è uno dei precedenti operatori.

- Se ***espr1*** e ***espr2*** sono espressioni, allora

  espr1 op*= *espr2

  equivale a

  ***espr1* = (*espr1*) *op* (*espr2*).**

  Esempio:

  `x *= y+1` equivale a `x = x * (y+1)`

Espressioni condizionali

- Espressione condizionale (operatore ternario):

  espr1* ? *espr2* : *espr3

  - è valutata *espr1*: se ha un *valore non nullo* (cioè se è vera) allora
  - viene valutata l'espressione *espr2*, e il risultato costituisce il valore dell'intera espressione condizionale
  - altrimenti viene valutata l'espressione *espr3*, e il risultato costituisce il valore dell'intera espressione condizionale

Esempio (massimo fra a e b):

```
z=(a > b) ? a : b
```

equivalente a

```
if (a > b)
      z = a;
else
      z = b;
```


Precedenza ed ordine di valutazione

OPERATORI	ASSOCIATIVITA'
() [] -> .	sx a dx
! - -- + ++ * & (tipo) sizeof	dx a sx
* / %	sx a dx
+ -	sx a dx
<< >>	sx a dx
< <= > >=	sx a dx
&	sx a dx
^	sx a dx
|	sx a dx
&&	sx a dx
||	dx a sx
?:	dx a sx
= += -= *= /=	sx a dx

Precedenza ed ordine di valutazione

- **Nota bene**: non è specificato l'ordine in cui vengono valutati "gli operandi" di un certo operatore.

  Nell'espressione

```
x = f(…) + g(…);
```

  quale delle due funzioni viene chiamata per prima?

Esempi di programmazione

- **Fattoriale di un numero *n***

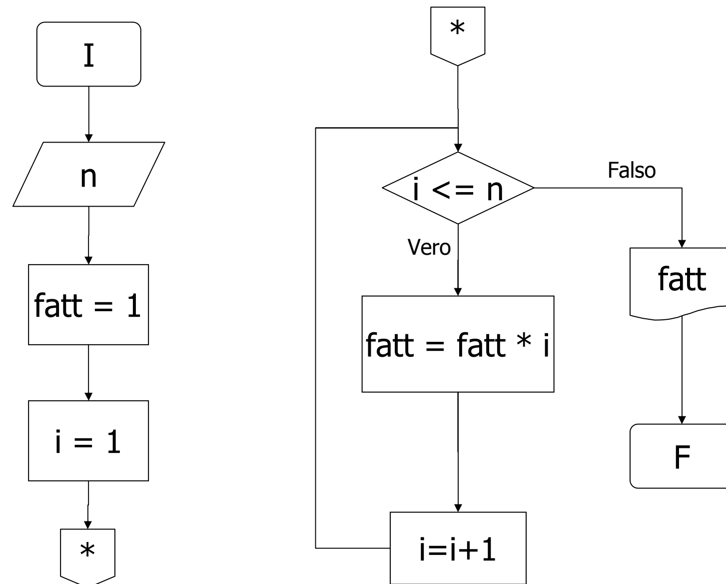
L'algoritmo si basa sulla proprietà del fattoriale:

$$n! = n \bullet (n-1) \bullet (n-2) \bullet \ldots \bullet 1 = n \bullet (n-1)!$$

Con un ciclo si scorrono tutti i numeri da 1 a n, ricalcolando il fattoriale ogni volta secondo la formula precedente

diagramma di flusso →

Fattoriale di un numero n



Codice del programma

```
#include <stdio.h>

int n;
unsigned long nfattoriale;

unsigned long fattoriale(int n)
{
  int i = 0;
  unsigned long fatt = 1;
  for(i = 1; i<=n; i++)
    fatt *= i  /* equivale a fatt = fatt *i; */

  return fatt;
}

main(void)
{
  printf("Numero: ");
  scanf("%d", &n);
  nfattoriale = fattoriale(n);
  printf("%d! = %ld\n", n, nfattoriale);
}
```

Alternativa: Fattoriale ricorsivo

- Definizione induttiva del fattoriale:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n \geq 1 \end{cases}$$

Funzioni ricorsive

- Un metodo ricorsivo possiede generalmente due parti.
  - **Una parte di terminazione che ferma la ricorsione, detta *caso di base.***
    - **Il caso di base deve avere una soluzione semplice o banale.**
  - **Una o più chiamate ricorsive, detto *caso ricorsivo*.**
    - **Il caso ricorsivo chiama a stessa funzione ma con argomenti più semplici o più piccoli.**

Function fattoriale()

```
unsigned long fattoriale(int n) {

  if (n == 0) {
      return 1;

  }

  else {
      return (n * fattoriale(n-1));
  }

}
```


Function fattoriale()

```
unsigned long fattoriale(int n) {
  if (n == 0) {
      return 1;          <--- Caso di base
  }
  else {
      return n * fattoriale(n-1);
  }
}
```

Caso di base

Caso ricorsivo

Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate

```
main()                      nfattoriale = fattoriale(n);
```


Invocazione ricorsiva

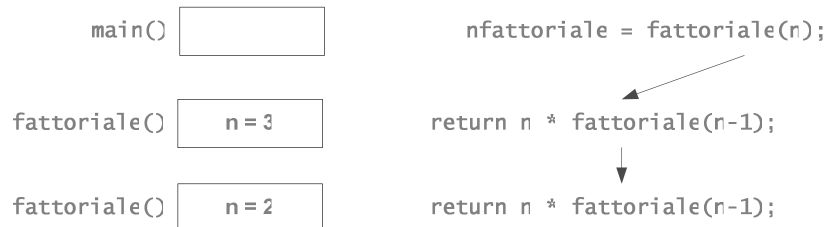
- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate

```
main()                      nfattoriale = fattoriale(n);

fattoriale()   n = 3        return n * fattoriale(n-1);
```

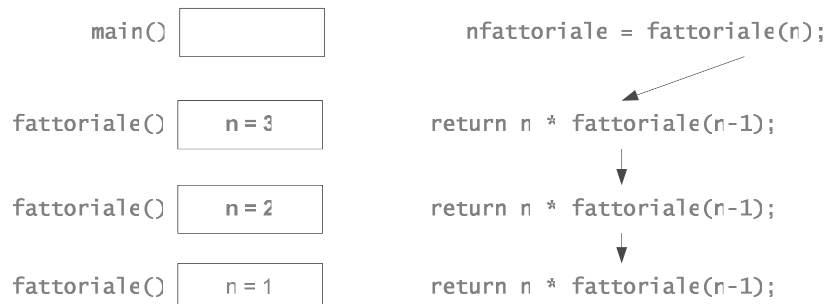
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



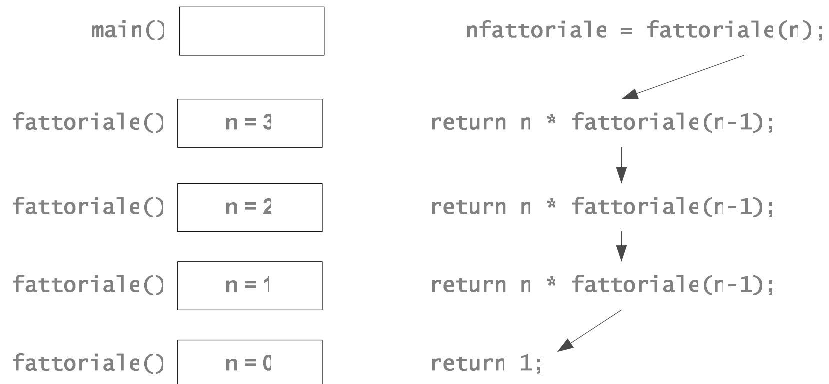
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



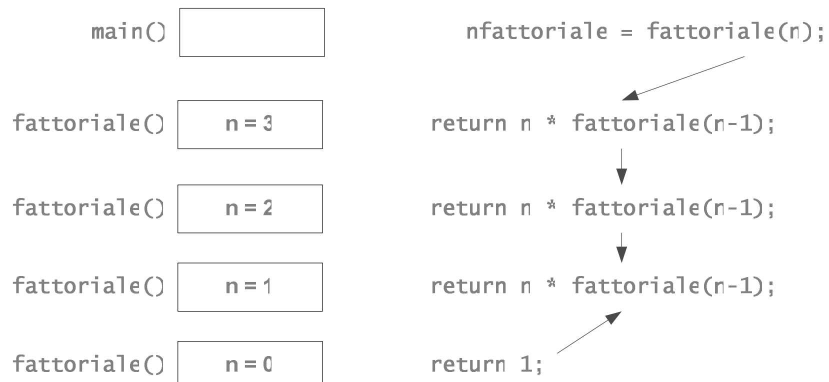
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



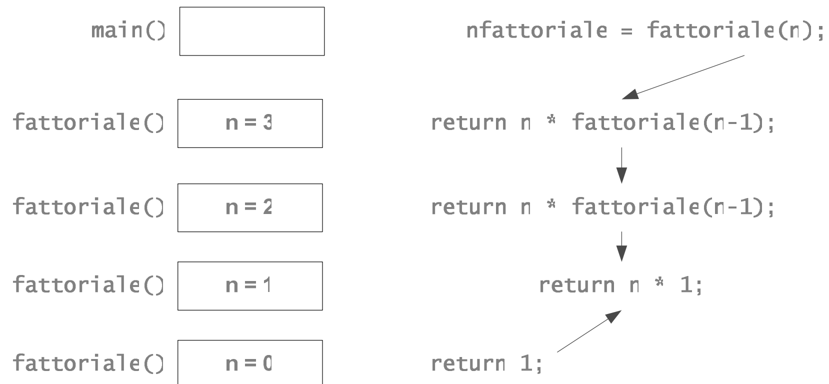
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



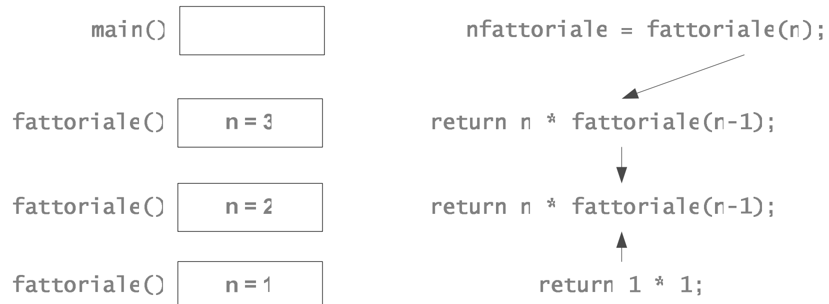
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



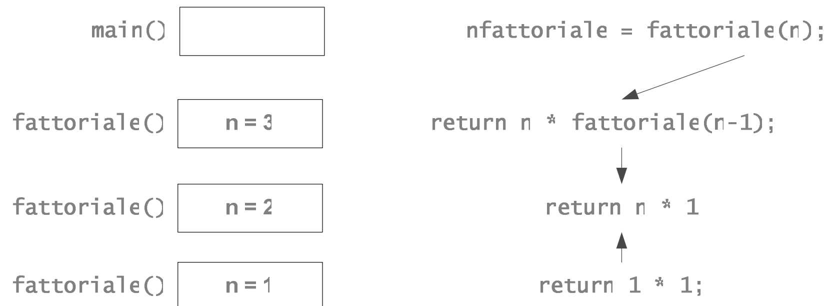
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



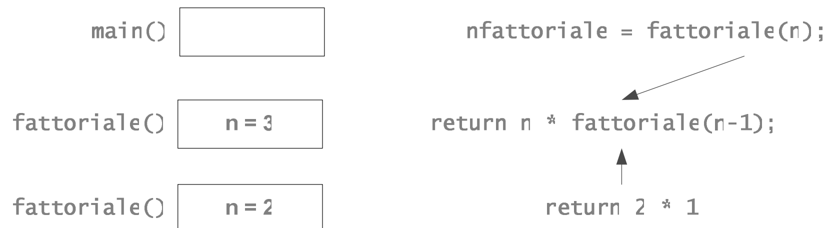
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate

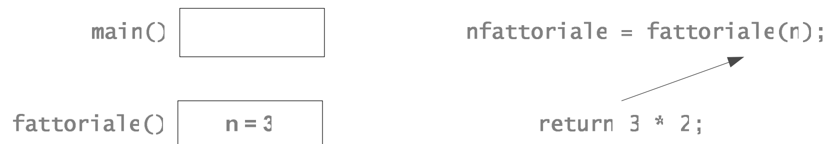


Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate

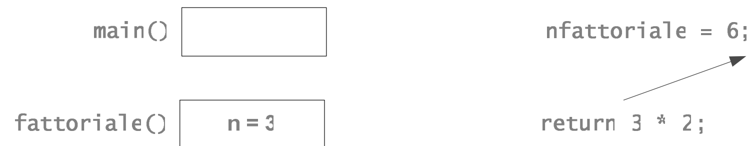
Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate



Invocazione ricorsiva

- *Ogni* invocazione della funzione viene inserita nello *Stack* delle chiamate

Esempi di programmazione

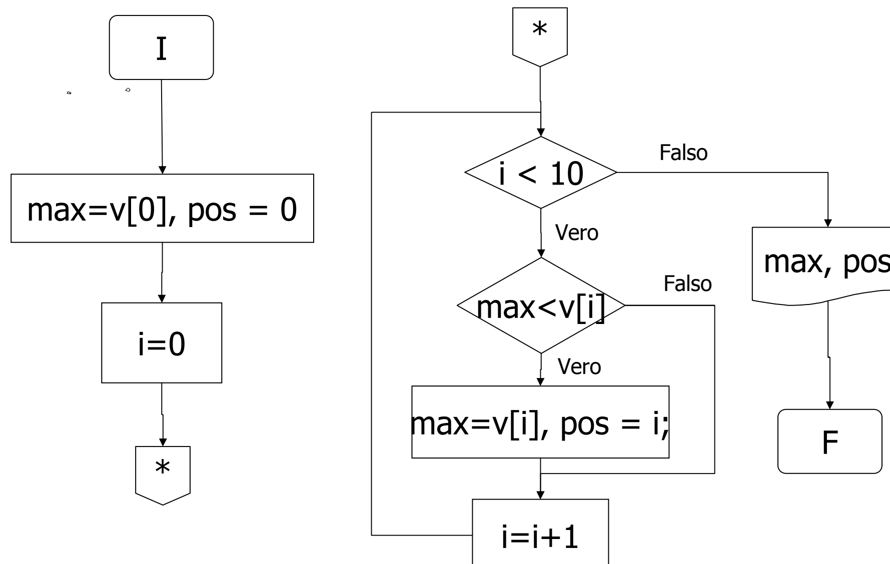
- **Ricerca del massimo elemento in un vettore**

Si supponga di avere un vettore V di 10 elementi, contenente numeri interi;

Algoritmo di scansione sequenziale: si controllano tutti gli elementi del vettore; se un elemento è maggiore del max trovato fino a quel momento, si aggiorna il max e se ne memorizza la posizione che occupa all'interno del vettore; si prosegue fino alla fine del vettore.

diagramma di flusso →

Ricerca del massimo elemento in un vettore



Ricerca del massimo elemento in un vettore

```
#include <stdio.h>

#define MAX_DIM = 10;
typedef vector int[MAX_DIM];

int ricerca_max (vector v)
   {
     int max, i, pos;
     max = v[0]; pos =0;
     for (i = 0; i < MAX_DIM; i++)
        {
          if (v[i] > max)
               {
                 max = v[i];
                     pos = i;
               }
        }
     return pos;
   }
```

```
main(void)
{

    int v[10] = {13, 124, 324, 12, 214, 24,
                 325, 6543, -32, -2};

    int pos = 0;
    int max = 0;

    printf("Ricerca elemento massimo in un
            vettore\n");

    pos = ricerca_max(v);
    max = v[pos];

    printf("Valore massimo:  %d\n
            posizione:  %d\n", max, pos);

}
```


Esempi di programmazione

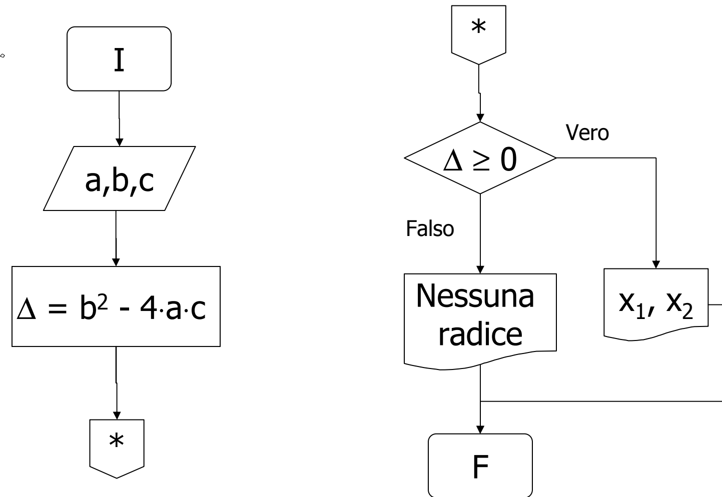
- **Risoluzione di un'equazione di 2° grado**

$$ax^2 + bx + c = 0$$

- Si inseriscono i coefficienti a,b,c ;
- Si calcola $\Delta = b^2 - 4 \cdot a \cdot c$;
  - $\Delta \geq 0$ → 2 radici reali
    ($\Delta = 0$ → radici coincidenti);
  - $\Delta < 0$ → nessuna radice reale ;

diagramma di flusso →

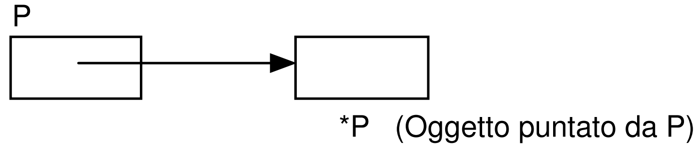
Equazione di 2° grado



5. Strutture dati complesse

Puntatori

- Un puntatore è una informazione che consente di raggiungere una determinata locazione di memoria
  - Un puntatore contiene quindi un indirizzo di memoria.



*P (Oggetto puntato da P)

  - L'operatore di ***deferenziazione*** ***** precedente l'identificatore del puntatore consente l'accesso all'oggetto puntato.
  - ***P** indica il contenuto della cella di memoria il cui indirizzo è contenuto nel puntatore P
  - L'operatore **&** precedente l'identificatore di una variabile restituisce l'indirizzo della variabile stessa.
  - Supponendo di avere un intero x (`int x`), **&x** indica l'indirizzo della cella di memoria che contiene il valore intero x.

Puntatori

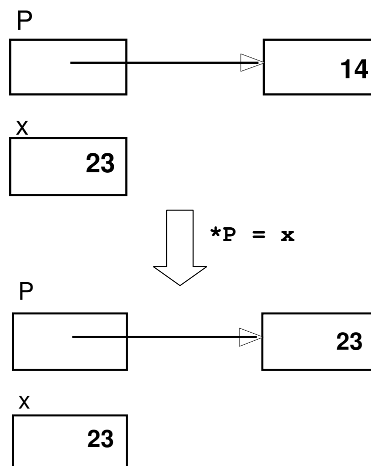
- Supponiamo di avere il seguente codice:

```
int *P;

/* P punta una cella
   contenente
   un valore intero */

int x = 23;
.
.
.
*P = 14;
```

- Eseguendo il comando *P = x si ottiene:



Puntatori

- Considerando ancora lo stesso codice della slide precedente:

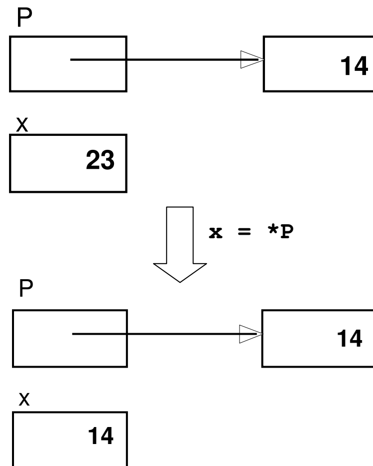
```
int *P;

/* P punta una cella
   contenente
   un valore intero */

int x = 23;
.
.
.
*P = 14;
```

- Se adesso si esegue invece

  x = *P si ottiene:

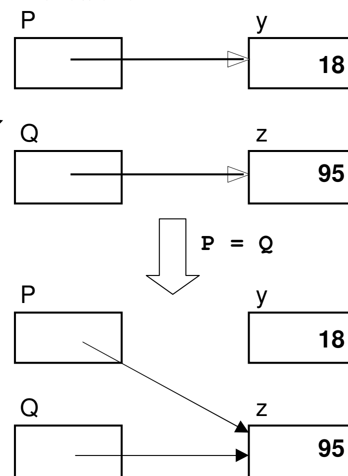


Puntatori

- Supponiamo di avere il seguente codice:

```
int *P, *Q;
int y = 18; int z = 95;
.
.
.
P = &y;
Q = &z;
;
```

- Eseguendo il comando P = Q si ottiene:



Puntatori e memoria

>> Errato → Corretto

```
#include
   <stdio.h>

int main()
{
  int *p;

  scanf("%d", p);

  return 0;
}
```

```
#include <stdio.h>

int main()
{
  int *p;
  int a;

  p = &a;
  scanf("%d", p);

  return 0;
}
```


Puntatori e memoria

- Spiegazione: un puntatore deve puntare ad una zona di memoria effettiva;
  un puntatore cui non si vuole assegnare alcuna area di memoria deve assumere il valore della costante **NULL** (void * 0)

- Oltre all'assegnamento di una locazione di memoria (appartenente ad una variabile) tramite l'operatore **&**, un puntatore può essere associato ad un'area di memoria attraverso le funzioni di libreria:

```
void * malloc(size_t size);
void free(void * p);
```

Dove:
- **malloc** alloca ***size*** bytes nello heap (la zona di memoria dedicata alla memoria dinamica)
- **free** libera la memoria allocata

Puntatori e memoria

- **Esempio:**

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

int main()
   {
     int *p;
     /* Allocate 4 bytes */
     p = (int *) malloc( sizeof(int) );
     scanf("%d", p);
     printf("%d", *p);
     /* Free memory; important !!! */
     free(p);
   }
```


Puntatori e memoria

- Altri dettagli...
  - per ogni chiamata a malloc, deve esserci (prima o poi) la corrispondente chiamata free
  - una tabella all'interno della libreria standard C tiene traccia dei blocchi allocati (indirizzi iniziali e loro dimensioni)
  - quando un indirizzo viene passato a free, viene cercato nella tabella e il blocco corrispondente viene "deallocato“
- Non è possibile deallocare:
  - un blocco di memoria già deallocato
  - parte di un blocco di memoria

Puntatori e memoria (cont.)

- Esempio:

```
int *p;
p = (int *)malloc(100*sizeof(int));
for(i=0; i<100; i++)
    p[i] = i;
p[100]=3;       /* sbagliato! */
free(p+10);     /* sbagliato! */
free(p);
p[0]=5;         /* già deallocata! */
free(p);        /* già deallocata! */
```


Array e stringhe

- ***Array***: definiscono vettori i cui elementi appartengono tutti allo stesso tipo: ciascun elemento è contraddistinto da un indice

```
int  v[32];  /* definisce un vettore di interi di 32
                elementi, con indice variabile in
                0..31 */
```

- E' possibile definire tipi di dati ***array*** con ***typedef***:

```
typedef int matrice[3][3]; /* tipo matrice 3x3 */
```

- Le ***stringhe*** sono codificate in C come vettori di caratteri (***array di char***):

```
typedef string char[];
```

  - Le stringhe di ***n*** caratteri contengono ***n+1*** elementi; l'ultimo elemento, di indice ***n***, è posto al valore **\0** (carattere di fine stringa)

Strutture (struct)

- Strutture
  - Le strutture sono il meccanismo C per raggruppare insiemi di dati eterogenei in singole unità "maneggevoli"
  - Sono anche il meccanismo "base" su cui è costruito gran parte del C++ (classi)
  - Le struct corrispondono ai record in Pascal.
- Per definire un tipo struttura:

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

- Questo definisce un nuovo tipo di nome **struct coord**

Dimensione di una struttura

- La dimensione di una struttura è pari alla somma delle dimensioni dei suoi membri?

  Esempio:

```
struct prova {
        char c;
        int i;
};
```

occupa un byte

occupa quattro byte

- la struttura potrebbe anche occupare otto byte invece di cinque, a causa di problemi/vincoli di allineamento su oggetti differenti.
- La dimensione corretta/effettiva di una struttura può essere ricavata con l'operatore **sizeof().**

Definizione di variabili struct

- Primo approccio:

```
struct coord {
              int x;
              int y;
             } first,second; /*Definizione variabili*/
```

- Secondo approccio:

```
struct coord {
               int x;
               int y;
             };
struct coord first,second;
```

- Terzo approccio:

```
struct coord {
               int x;
               int y;
             };
typedef struct coord coordinate;

coordinate first,second;
```


Struct – Accesso ai campi

- Per accedere ai campi di una struttura, si utilizza l'operatore "**.**"
  Sitassi:
  structure_var.member_name
  Esempi:

```
first.x = 50;
second.y = 100;
```

- La sintassi *structure_var***.***member_name* può essere utilizzato ovunque può essere utilizzata una variabile:

```
printf ("%d , %d", second.x , second.y );
scanf("%d, %d", &first.x, &first.y);
```

- E' possibile copiare il contenuto di una struttura con un solo statement:

```
second = first;
```

- invece di scrivere

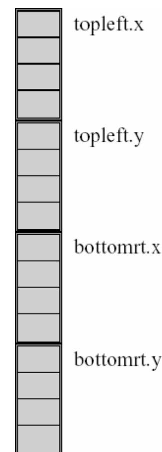
```
second.x = first.x; second.y=first.y;
```

Strutture Annidate

- Oggetti di qualunque "tipo" possono far parte di una struttura:

  Esempio (rettangolo come coppia di punti):

```
struct rectangle {
      struct coord topleft;
      struct coord bottomrt;
} ;
struct rectangle mybox ;
mybox.topleft.x = 0 ;
mybox.topleft.y = 10 ;
mybox.bottomrt.x = 100 ;
mybox.bottomrt.y = 200 ;
mybox.bottomrt.x = 100 ;
mybox.bottomrt.y = 200 ;
```



Strutture e Array

- E' possibile inserire array in una struttura, come qualsiasi altro oggetto

  Esempio:

```
struct record {
        float x;
        char y[5] ;
                  };
struct record p;
p.x = 1.0;
p.y[0] = 0;
```

- Allo stesso modo è possibile definire array di strutture.

  Esempio (1000 entry cognome-nome-telefono):

```
struct entry {
   char fname [20] ;
   char lname [20] ;
   char phone [10] ;
            } ;
struct entry list[1000];
```

Strutture - Inizializzazione

```
struct sale {
  char customer [20] ;
   char item [20] ;
  int amount ;
};

struct sale mysale = {
  " Lanificio Rossi ",
  " Lana grezza Lana grezza ",
  1000
};
```

- Per le strutture annidate:

```
struct customer {
  char firm [20] ;
  char contact [25] ;
};

struct sale {
  struct customer buyer ;
  char item [20] ;
  int amount ;
}

struct sale mysale = {
  {" Lanificio Rossi ", "
   Mario Rossi "},
  " Mario Rossi ",
  1000
};
```


Strutture - Inizializzazione (cont)

- Per gli array di strutture:

```
struct customer {
  char firm [20] ;
  char contact [25] ;
};

struct sale {
  struct customer buyer;
  char item [20] ;
  int amount ;
};
```

```
struct sale y1990[2] =
  {
   {
    {"Lanificio Rossi",
     "Mario Rossi"} ,
     "Lana grezza",
     1000
   },
   {
    {"TuttoSport",
     "Elena Baggio"},
     "Attrezzature per
      basket",
     290
   }
  };
```

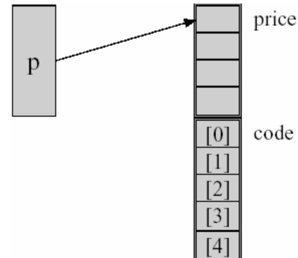
Puntatori a struttura

```
struct part {
  float price ;
  char code[5] ;
};

struct part *p;
struct part thing;

p = &thing;
   /* The following three
      statements are equivalent */

thing.price = 50;
(*p).price = 50;
p -> price = 50;
```



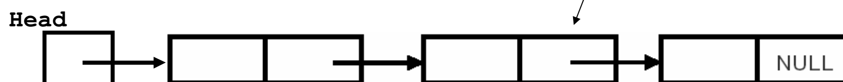
Puntatori come membri di struttura

```
struct node{
             int data;
             struct node *next;
};

struct node *Head,a,b,c;
Head = &a;
a.next = &b;
b.next = &c;
c.next = NULL;
a.data = 1;
a.next->data = 2; /* b.data =2 */
a.next->next->data = 3; /* c.data = 3 */
```

definizione di una struttura di dimensioni dinamiche denominata lista; il puntatore Head punta l'inizio della lista, mentre il puntatore next permette di raggiungere l'elemento successivo

Head



Parametri sulla riga di comando

- Fino ad ora abbiamo sempre visto la funzione **main** nella forma:

```
int main()
```

- Esiste, però, una forma più estesa che ci permette di accedere ai parametri passati al programma sulla command line,

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

dove

- **argc** è il numero dei parametri passati sulla command line più uno
- **argv[]** è un vettore di argc puntatori a stringa contennente:
  - in **argv[0]** il path name con il quale il programma è stato invocato
  - in **argv[1] .. argv[argc – 1]** i parametri passati sulla linea di comando.

Parametri sulla riga di comando

- Esempio

```
int main(int argc, char *argv[])
{
  int i;
  printf("%s e' stato chiamato con parametri\n",
          argv[0]);
  for(i = 1; i < argc; ++i)
       printf("\t%s\n", argv[i]);
  return 0;
}
```

6. Sommario delle principali librerie di sistema

Input e output standard

- Le funzionalità di input e output non fanno parte del C, ma sono implementate attraverso la libreria standard, ed altre librerie più sofisticate (**`<stdio.d>`**, **`<string.h>`**, **`<ctype.h>`**, **`<conio.h>`**).
  - **int getchar(void)** è il più semplice meccanismo di input; ad ogni chiamata, getchar restituisce il prossimo carattere in input.
  - **int putchar(int)** è usata per l'output; ad ogni chiamata, putchar(c) invia il carattere c allo standarrd output (il video).
  - ogni sorgente che utilizza una funzione di input/output della libreria standard deve contenere la linea **`#include <stdio.h>`.**

Output formattato - printf

- **int printf(char *format, arg1, arg2, …)** converte, formatta e stampa sullo standard output i suoi argomenti sotto il controllo di format.
- La stringa di formato contiene: caratteri ordinari (copiati sull'output) e specifiche di conversione (che cominciano con %, finiscono con un carattere di conversione e convertono e stampano gli argomenti).
- Tra il % ed il carattere di conversione possiamo trovare:
  - Un segno meno (allineamento a sinistra);
  - Un numero per l'ampiezza minima del campo;
  - Un punto, per la precisione;
  - La precisione (numeri da stampare dopo il punto decimale, oppure numero massimo di caratteri da stampare di una stringa);
  - Una h, se il numero deve essere stampato come short;
  - Una l, se il numero deve essere stampato come long.

Output formattato - printf

- Di seguito è riportato l'effetto di alcune specifiche sulla stampa della stringa di 12 caratteri "salve, mondo".

%s	:salve, mondo:
%10s	:salve, mondo:
%.10s	:salve, mon:
%.15s	:salve, mondo:
%-15s	:salve, mondo :
%15.10s	: salve, mon:
-15.10s	:salve, mon :

- **int sprintf(char * string char *format, arg1, arg2, …)**
  converte, formatta e stampa su **string** i suoi argomenti sotto il controllo di format.

Input formattato - scanf

- **int scanf(char *format, …)** legge caratteri dallo standard input, li interpreta in base al contenuto di format, e memorizza il risultato negli argomenti che seguono.
- Gli argomenti che seguono format devono essere dei puntatori ai registri dove memorizzare l'input formattato.
- scanf termina quando esaurisce la sua stringa di formato; il suo valore di ritorno è il numero di elementi letti in input e registrati correttamente.

- **int sscanf(char * string char *format, arg1, arg2, …)**
  legge una stringa (sotto il controllo di format) da **string**.
- La stringa di formato contiene le specifiche di conversione:
  - Spazi o caratteri di tabulazione (ignorati);
  - Caratteri normali (non %) che dovrebbero corrispondere al successivo carattere in input;
  - %, *, un numero per l'ampiezza massima, h, l o L per la dimensione, un carattere di conversione.

Accesso a file

- **Problema**:
  scrivere un programma che accede a un file.
- **fopen** legge un nome esterno e restituisce un puntatore da utilizzare nelle chiamate a lettura e scrittura del file.

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

- la prima dichiarazione afferma che **fp è un puntatore a FILE**
- `fopen` è una funzione che restituisce un puntatore a FILE;
- **FILE** è un nome di tipo, e non il tag di una struttura.

```
fp = fopen(name, mode)
```

Nome del file; se non esiste in scrittura o in append, viene creato.

Indica come aprire il file secondo la modalità indicata in *mode*:

r (lettura), w (scrittura), a (aggiunta)

Accesso a file

- l'apertura in scrittura o in append di un file che non esiste provoca la sua creazione;
- l'apertura in scrittura di un file che già esiste provoca la perdita di tutti i dati;
- l'apertura in append di un file che già esiste non provoca la pedita dei dati;
- in qualsiasi caso di errore, fopen ritorna **NULL**;

- **`int getc(FILE *fp);`**
  è la definizione della funzione che ritorna un carattere da un file.
- **`int putc(int c, FILE *fp);`**
  è la definizione della funzione che scrive il carattere c sul file fp e ritorna il carattere scritto.
- **`int fscanf(FILE *fp, char *format, …);`**
- **`int fprintf(FILE *fp, char *format, …);`**
  sono uguali a scanf e printf, ma leggono e scrivono su files.
- **`int fclose(FILE *fp);`**
  interrompe la connessione creata da fopen fra il puntatore e il suo nome esterno

Accesso a file (cont.)

- **Esempio**:
  leggere due file carattere per carattere e mischiarli (merging).

```
#include <stdio.h>
main()
{
  FILE *in1,*in2,*out;
  FILE *cur;
  int c,N;

  in1=fopen("f1","r");
  in2=fopen("f2","r");
  out=fopen("output","w");
  cur=in1;
  while ((c=fgetc(cur))!=EOF)
       if (c=='*')
          cur=(cur==in1)? in2 : in1;
       else
          fputc(c,out);
       fclose(in1);
       fclose(in2);
       fclose(out);
}
```

Input e output di linee

- **`char *fgets(char *line, int maxline, FILE *fp)`**
  legge una linea di input dal file fp e la registra nel vettore line; legge al massimo maxline-1 caratteri. Ritorna line.

- **`char *fputs(char *line, FILE *fp)`**
  scrive la linea line sul file fp. Ritorna zero.

Funzioni standard utili

<u>Operazioni sulle stringhe</u> (modulo <string.h>)

strcat(s, t)	concatena t alla fine di s;
strncat(s, t, n)	concatena n caratteri di t alla fine di s;
strcmp(s, t)	ritorna un valore negativo, nullo o positivo per s<t, s==t, s>t;
strncmp(s, t, n)	come strcmp, ma confronta solo i primi n caratteri;
strcpy(s, t)	copia t in s;
strlen(s)	ritorna la lunghezza di s;
strchr(s,c)	ritorna un puntatore al primo c in s, NULL se c non compare in s;
strrchr(s,c)	ritorna un puntatore all'ultimo c in s, NULL se c non compare in s;

Funzioni standard utili

Controllo e conversione della classe di un carattere

isalpha(c)	diverso da zero se c è alfabetico, 0 altrimenti
isupper(c)	diverso da zero se c è maiuscolo, 0 altrimenti
islower(c)	diverso da zero se c è minuscolo, 0 altrimenti
isdigit(c)	diverso da zero se c è una cifra, 0 altrimenti
isalnum(c)	diverso da zero se c è vero isalpha(c) o isdigit©, 0 altrimenti
isspace(c)	diverso da zero se c è un carattere di spaziatura, 0 altrimenti
toupper(c)	ritorna c convertito in maiuscolo
tolower(c)	ritorna c convertito in minuscolo

Funzioni standard utili

Ungetc

int ungetc(int c, FILE *fp)	rideposita il carattere c nel file fp, restituendo c

Esecuzione dei comandi

system(char *s)	esegue il comando contenuto nella stringa s, e riprende l'esecuzione del programma corrente

Funzioni standard utili

Gestione della memoria (modulo <malloc.h>)

void *malloc(n)	ritorna un puntatore a n byte di memoria non inizializzata, oppure NULL se la richiesta non è soddisfacibile
void *calloc(n, size)	ritorna un puntatore a un'area sufficiente a contenere un vettore di n oggetti di ampiezza size, oppure NULL se la richiesta non è soddisfacibile
void free(p)	libera lo spazio puntato da p, dove p è un puntatore ottenuto in precedenza da una chiamata a malloc o calloc

Funzioni standard utili

- Il puntatore restituito da malloc e calloc ha l'allineamento corretto per gli oggetti, ma deve essere forzato al tipo appropriato.

Esempio:
```
int *ip;
ip = (int *)calloc(n, sizeof(int));
```

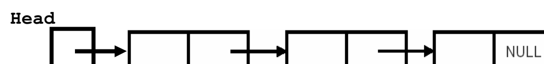
- In generale non esistono restrizioni sul puntatore passato a free, ma è un errore liberare aree di memoria che non sono state ottenute con malloc o calloc.
- E' anche un errore usare aree di memoria già rilasciate in precedenza.

Esempio sbagliato: un ciclo che libera elementi di una lista;
```
for(p = head; p != NULL; p = p->next)
     free(p);
```

Esempio esatto: un ciclo che libera elementi di una lista, ma salva tutto ciò che serve prima della free;

```
for(p = head; p != NULL; p = q)
{
  q = p->next;
  free(p);
}
```

Head



Funzioni standard utili

<u>Funzioni matematiche</u> (modulo <math.h>)

Nell'header **`<math.h>`** sono dichiarate molte funzioni matematiche; gli argomenti e i valori ritornati sono di tipo double.

sin(x)	seno di x in radianti	**log(x)**	logaritmo naturale
cos(x)	coseno di x in radianti	**log10(x)**	logaritmo in base 10
atan2(y,x)	Arcotangente di y/x in radianti	**pow(x,y)**	x elevato a y
		sqrt(x)	radice quadrata
exp(x)	Esponenziale	**fabs(x)**	valore assoluto

<u>Numeri casuali</u>

Nell'header **`<stlib.h>`** è definita la costante **RAND_MAX**; la funzione **`rand()`** calcola una sequenza di interi pseudo-casuali nell'intervallo **[0, RAND_MAX]**.