

Programmazione

Andrea Passerini
passerini@dsi.unifi.it

Conoscenze informatiche e relazionali
Corso di laurea in Scienze dell'Ingegneria Edile

Linguaggi di programmazione

- Il calcolatore è in grado di comprendere solo istruzioni in linguaggio macchina.
- Il linguaggio macchina non è adatto alla scrittura di programmi poiché troppo distante dal linguaggio naturale.
- I linguaggi di programmazione permettono di scrivere programmi più agevolmente mantenendo le caratteristiche di non ambiguità e non ridondanza necessarie.
- Per poter essere eseguito dal calcolatore, un programma scritto con un linguaggio di programmazione deve essere *tradotto* in linguaggio macchina.

Classificazione dei linguaggi

In base al livello in cui si collocano tra macchina e uomo
basso livello

- E' il linguaggio assemblatore (*assembly*)
- c'è una corrispondenza uno ad uno tra istruzioni in assembly ed istruzioni in linguaggio macchina
- è specifico per ogni CPU
- permette di sfruttare appieno le potenzialità della CPU (il suo set di istruzioni)

Classificazione dei linguaggi

In base al livello in cui si collocano tra macchina e uomo
alto livello

- E' svincolato dalla struttura della macchina
- è più agevole per il programmatore in quanto più simile al linguaggio naturale
- evita di dover conoscere le caratteristiche specifiche della CPU
- permette di utilizzare lo stesso programma su CPU diverse (con opportuni traduttori)
- per ogni CPU deve esistere il relativo traduttore del linguaggio
- Il traduttore non sempre produce codice ottimizzato

Classificazione dei linguaggi

In base a ciò che il programmatore deve specificare

Linguaggi procedurali

- Il programmatore deve specificare il procedimento di calcolo (algoritmo) per ottenere il risultato voluto dati gli ingressi
- E' il tipo di programmazione di gran lunga più usata
- Esempi: Fortran, Basic, Pascal, Ada, C

Linguaggi non procedurali

- Il programmatore descrive “cosa” deve essere calcolato
- Il linguaggio associa alla descrizione del problema da risolvere (in termini di *fatti* e *regole*) una strategia di risoluzione tramite algoritmi predefiniti (tramite un *motore di inferenza*).

Classificazione dei linguaggi

In base alla struttura che devono avere i programmi

Programmazione strutturata

- si attiene ai concetti di diagrammi strutturati per gli algoritmi (Teorema di Böhm-Jacopini)
- introduce il concetto di *visibilità* di una variabile (*scope*) (variabili *locali* ad un blocco).
- permette maggiore riusabilità , leggibilità e manutenzione dei programmi.
- Esempi: Pascal, Ada, C

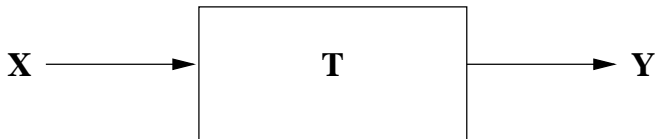
Classificazione dei linguaggi

In base alla struttura che devono avere i programmi

Programmazione orientata agli oggetti

- presuppone di raggruppare dati e procedure che operano su di essi in entità (*classi*)
- una classe è dotata di *attributi* (i dati) e *metodi* (le procedure) per accedere ed operare sugli attributi
- un *oggetto* è un'istanza di una determinata classe, con specifici valori per i suoi attributi (che possono variare durante l'esecuzione)
- E' il paradigma di programmazione più usato attualmente
- Esempi: C++, Java, Smalltalk, Python

Traduzione



- Dati:
 - una sequenza di istruzioni X descritte in un linguaggio *sorgente* (e.g. C).
 - un traduttore T in linguaggio macchina per una certa CPU C
- L'esecuzione di T con ingresso X produce in uscita una sequenza di istruzioni Y descritte in linguaggio *eseguibile* da C .

Traduzione

Esistono due possibili approcci alla traduzione

Compilazione

- Il programma sorgente viene compilato nel suo rispettivo eseguibile una volta per tutte.
- La versione eseguibile viene utilizzata ogni volta si debba eseguire il programma.
- Esempi di linguaggi compilati: Pascal, C

Interpretazione

- La traduzione in istruzioni in linguaggio macchina avviene solo al momento dell'esecuzione del programma.
- Il programma non viene eseguito direttamente dalla CPU ma tramite un *interprete* che traduce e manda in esecuzione le istruzioni del programma una per una.

Compilatori vs Interpreti

tempi di esecuzione L'interprete deve tradurre al volo ogni singola istruzione (impiegando tempo) e non può ottimizzare il codice

qualità della traduzione il compilatore, eseguendo la traduzione al di fuori dell'esecuzione del programma, ha tutto il tempo di ottimizzare la traduzione realizzando codice più efficiente

portabilità del codice un programma eseguibile non può funzionare su una CPU diversa da quella per cui è stato compilato

Compilatori vs Interpreti

modificabilità del codice un programma eseguibile non può essere modificato, e recuperare il sorgente a partire dall'eseguibile è estremamente difficile (*reverse code engineering*)

scrittura dei programmi la scrittura di software tipicamente comporta modifiche successive al codice sorgente via via che viene scritto. In un linguaggio compilato qualunque modifica al programma comporta la ricompilazione del tutto.

Compilatore + *Virtual Machine*

- è una soluzione ibrida tra compilazione ed interpretazione
- Il codice sorgente viene compilato in un linguaggio intermedio (*bytecode*) (una volta per tutte)
- il bytecode viene interpretato da una *virtual machine* (VM) ossia un interprete di basso livello che emula una CPU.
- per velocizzare la fase di esecuzione, il bytecode viene compilato in codice macchina (compilazione *just in time* (JIT)) all'inizio dell'esecuzione.
- vantaggi
 - Le ottimizzazioni più onerose vengono fatte durante la prima compilazione
 - Il bytecode è non modificabile
 - Lo stesso bytecode può essere eseguito su tutte le CPU per cui esiste un'implementazione della VM
 - E' più facile scrivere VM per CPU diverse che compilatori
- svantaggi
 - L'esecuzione di un programma implica un ritardo iniziale dovuto alla compilazione da bytecode a codice macchina

Linguaggio “C”

- Sviluppato da Dennis Ritchie nel 1972
- E' un linguaggio:
 - compilato
 - procedurale
 - strutturato
- Permette l'accesso a basso livello alla memoria ed è fatto per generare poche istruzioni macchina per ogni elemento basilare del linguaggio
- E' il linguaggio più usato per scrivere software di sistema
- Richiede una certa esperienza per essere usato efficacemente senza introdurre errori (*bugs*) nei programmi.
- E' stato standardizzato nel 1989 con il nome di ANSI-C, descritto in *The C Programming Language, Second Edition* by Brian W. Kernighan and Dennis M. Ritchie. Prentice Hall, Inc., 1988 (lo standard attuale è detto ISO C99).

Struttura di un programma

- Un algoritmo rappresentato da un diagramma a blocchi strutturato può essere tradotto in una serie di espressioni del linguaggio C
- Il C possiede 32 parole chiave per definire il tipo dei dati e le istruzioni di controllo
- Operazioni più complesse ma di uso comune sono realizzate tramite funzioni rese disponibili da apposite *librerie*
- Le librerie sono delle “raccolte” di funzioni o programmi utili e di uso generale

Struttura di un programma

- Un programma C è costituito da *funzioni e variabili*.
- Una funzione è costituita da un nome, zero o più parametri di ingresso, zero o un parametro di uscita, ed un blocco di istruzioni.
- Ogni istruzione deve essere terminata dal segno ';'.
- Ogni blocco di istruzioni deve essere racchiuso da {}.
- Una funzione specifica chiamata `main` viene utilizzata per delimitare l'intero programma.
- Gli spazi multipli e il ritorno a capo non hanno significato; in particolare non servono per delimitare/identificare le istruzioni (ma vengono usati per rendere leggibile il codice).

Esempio di programma

```
#include <stdio.h>

int main()
{
    printf("Hello, ");
    printf("World\n");

    return 0;
}
```

Dichiarazione ed uso

- Variabili e funzioni devono essere *dichiarate* **prima** di poter essere usate, prima cioè di potersi riferire a esse
- La dichiarazione introduce un nome simbolico per l'entità e ne specifica delle caratteristiche:
 - Per una variabile le caratteristiche si riferiscono al tipo dei dati immagazzinati
 - Per una funzione le caratteristiche si riferiscono al tipo e numero di parametri in ingresso ed in uscita

Esempio

ERRATO

```
#include <stdio.h>

int main()
{
    a = 3;
    b = 5;
    printf("a+b=%d\n", a+b);
    int a;
    int b;

    return 0;
}
```

CORRETTO

```
#include <stdio.h>

int main()
{
    int a;
    int b;
    a = 3;
    b = 5;
    printf("a+b=%d\n", a+b);

    return 0;
}
```

Dichiarazione di variabili

- Le variabili sono dichiarate indicandone prima il tipo e poi il nome (e.g. `int a;`)
- Il tipo di una variabile specifica la natura del dato che potrà essere immagazzinato nella variabile
- Il valore di una variabile nel momento della dichiarazione è indefinito
- E' possibile inizializzare la variabile ad un certo valore mediante assegnamento.

Nome di variabile

- Il nome di una variabile è una sequenza di:
 - caratteri alfabetici maiuscoli o minuscoli
 - numeri (non come primo carattere)
 - il carattere '_' (si usa al posto dello spazio)

ERRATO

```
12var  
città  
data di nascita  
valore_in_-$
```

CORRETTO

```
var12  
citta  
data_di_nascita  
valore_in.dollari
```

Tipi fondamentali

char un byte, può rappresentare un qualsiasi carattere del set locale (e.g. ASCII o ISO Latin-1)

int un numero intero

float un numero reale in virgola mobile in singola precisione

double un numero reale in virgola mobile in doppia precisione

Costanti

- carattere Si scrive il carattere racchiuso tra apici singoli (e.g. `'A'`, `'0'`). Alcuni caratteri speciali vengono rappresentati tramite le sequenze di escape (e.g. il ritorno a capo: `'\n'`)
- interi si rappresentano con la sequenza di cifre decimali (e.g. `12008`)
- reali si rappresentano sia in notazione scientifica che non (e.g. `0.001`, `1e-3`)

Assegnamento

- L'operatore standard di assegnamento è =
- Ad una variabile si può assegnare:
 - una costante del tipo della variabile: $a=5$
 - il valore di una altra variabile: $b=a$;
 - il risultato di una espressione: $c=a+b*2$;

Nota

Nei diagrammi di flusso in genere si utilizza il simbolo \leftarrow per l'assegnazione, mentre = è utilizzato come operatore relazionale di uguaglianza nelle istruzioni condizionali.

Vettori

- Un insieme di valori tutti dello stesso tipo può essere memorizzato ed elaborato tramite un dato di tipo *vettore*
- un vettore si dichiara specificando il tipo di elementi, un nome per la variabile ed il numero di elementi, usando la sintassi:

tipo nome_vettore[dimensione]

- Ad esempio

```
int secondi[60];  
char nome[100];
```

Vettori

- Un vettore può essere inizializzato assegnando un valore a ciascuno dei suoi elementi.
- per farlo si devono indicare tutti i valori fra parentesi graffe separati da virgole:

```
tipo nome[n]={val1,...,valn}  
tipo nome[]={val1,...,valn}
```

- nel caso di inizializzazione si può non specificare il numero di elementi che viene calcolato automaticamente
- Ad esempio:

```
float vec[]={0.1,0.4,0.7,1.2};  
char citta[]={'f','i','r','e','n','z','e'};
```

Vettori

- I singoli elementi dei vettori si indicano con la notazione

`nome[indice]`

dove `indice` è la posizione nel vettore **a partire da zero**.

- L'indice può essere una costante intera maggiore o uguale a 0 o una variabile intera, nel qual caso la posizione è data dal valore della variabile.
- Si possono utilizzare gli operatori aritmetici e di assegnamento sugli elementi di un vettore ma non sul vettore stesso.
- Il linguaggio non fa controlli sulla dimensione del vettore, per cui usare un indice maggiore della dimensione produrrà risultati indefiniti

Esempi

ERRATO

```
int a[] = {1,2,3};  
int b[] = {4,5,6,7,8};  
int d[3] = a; /* assegnazione */
```

```
a + b; /* somma */  
a < b; /* confronto */
```

CORRETTO

```
int a[] = {1,2,3};  
int b[] = {4,5,6,7,8};  
int c = 0;  
int d = 3;
```

```
a[1] = b[4] * (1 + a[2]);  
b[++d] += a[1] * a[c];  
a[d] = 3; /* out of size */  
b[--c] = 0; /* negative index */
```

Stringhe

- Le parole o le frasi sono dei vettori di caratteri in cui l'ultimo carattere è sempre ' \0 ' ed indica la fine della stringa.
- Le stringhe possono essere inizializzate con una sintassi particolare (il carattere di terminazione viene aggiunto automaticamente):

```
char nome[]="contenuto stringa";
```

- Ad esempio:

```
char frase[]="ecco una stringa";
```

Commenti

- Sono sequenze di caratteri comprese tra `/*` e `*/`
- Vengono ignorate dal compilatore e quindi non influenzano il comportamento del programma.
- Servono a documentare il codice inserendo spiegazioni accanto a dichiarazioni di variabili, funzioni, blocchi di istruzioni, etc.
- Ad esempio:

```
int a; /* numeratore */  
int b; /* denominatore */  
int r; /* resto */
```

```
/* calcolo il resto di a diviso b */  
r = a % b;
```

Operatori

Operatori aritmetici binari

+ - * / %

- La divisione intera tronca la parte frazionaria
- L'operatore % (modulo) è applicabile solo agli interi
- + - hanno precedenza inferiore rispetto a * / %
- tutti gli operatori aritmetici sono associativi da sinistra a destra

Operatori

Operatori relazionali e logici

> >= < <= == != && || !

- Restituiscono un valore vero o falso (si usano nelle istruzioni condizionate)
- Tutti gli operatori relazionali e logici (tranne !) hanno precedenza inferiore agli operatori aritmetici
- Gli operatori di disuguaglianza > >= < <= hanno precedenza superiore agli operatori di uguaglianza == (uguale) != (diverso)
- Gli operatori logici && (AND) || (OR) hanno precedenza inferiore a tutti gli altri operatori e sono associativi da sinistra a destra. && ha precedenza superiore a ||.
- L'operatore logico ! (NOT) ha precedenza superiore agli

Operatori

Operatori di incremento e decremento

++ --

- Sono operatori unari: sommano o sottraggono 1 al loro operando
- Possono essere usati in notazione prefissa (e.g. ++v) o postfissa (e.g. v--).
- In notazione prefissa prima la variabile viene modificata, poi se ne usa il valore.
- In notazione postfissa prima si usa il valore della variabile, poi si modifica.
- Ad esempio, se $n==5$, $x=n++$ assegna ad x 5, $x=++n$ assegna ad x 6 (in entrambi i casi ad n viene assegnato 6).

Operatori

Operatori di assegnamento

$+=$ $-=$ $*=$ $/=$ $\%=$

- Sono tutti della forma $op=$ con op operatore aritmetico.
- Date due espressioni $espr_1$ ed $expr_2$, si ha che:

$espr_1\ op=\ expr_2$

- equivale a:

$espr_1 = (espr_1)\ op\ (expr_2)$

- Ad esempio $x *= y+1$ equivale a $x = x * (y+1)$

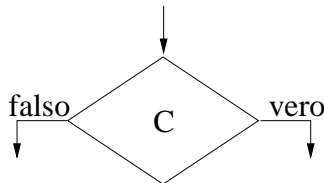
Istruzione condizionale

- L'istruzione condizionale si rappresenta con la sintassi:

```
if (condizione) espressione1;  
else espressione2;
```

- Ad Esempio:

```
if (a > b)  
    b++;  
else  
    a++;
```



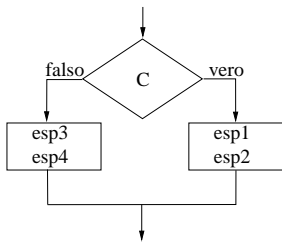
Istruzione condizionale

- All'istruzione può sempre essere sostituito un blocco di istruzioni (delimitato da { }), con la sintassi:

```
if (condizione) {  
    espressione1;  
    espressione2;  
}  
else {  
    espressione3;  
    espressione4;  
}
```

- Ad esempio:

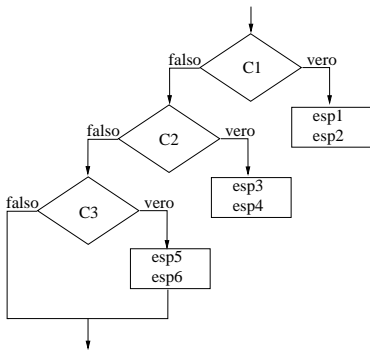
```
if(a > b && b != 0)  
{  
    a += a - b;  
    b = a;  
}  
else  
{  
    b = 0;  
    a++;  
}
```



Elenco di istruzioni condizionali

Le istruzioni condizionali possono essere usate in cascata per ottenere più alternative:

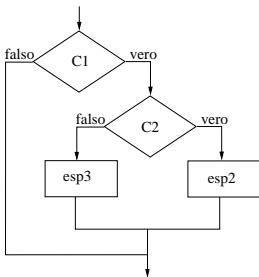
```
if(condizione1) {  
    espressione1;  
    espressione2;  
}  
else if(condizione2) {  
    espressione3;  
    espressione4;  
}  
else if(condizione3) {  
    espressione5;  
    espressione6;  
}
```



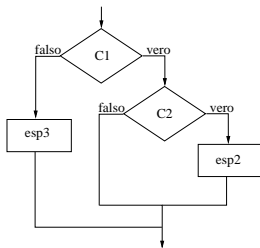
Nota

L'`else` si riferisce sempre all'`if` precedente che non sia annidato all'interno di un blocco:

```
if (condizione1)
  if (condizione2)
    espressione2;
  else
    espressione3;
```



```
if (condizione1)
{
  if (condizione2)
    espressione2;
}
else
  espressione3;
```



Valutazione delle condizioni

- In C, per convenzione si associa:
 - 0 a falso
 - tutto ciò che è diverso da 0 a vero
- Ad esempio:

```
int x = 0;
```

```
if(x)
```

```
    printf("x non contiene 0");
```

```
else
```

```
    printf("x contiene 0");
```

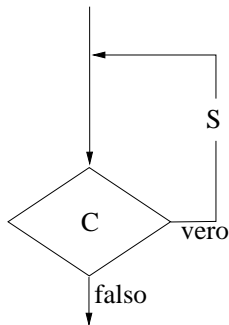
Ciclo

- Il blocco di iterazione strutturato con *controllo in testa* si rappresenta con la sintassi:

```
while (condizione)  
    espressione;
```

- che per blocchi di istruzioni diventa:

```
while (condizione)  
{  
    espressione1;  
    espressione2;  
}
```



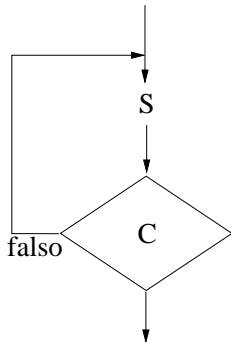
Ciclo

- Il blocco di iterazione con *controllo in coda* si rappresenta con la sintassi:

```
do  
  espressione;  
while (condizione) ;
```

- che per blocchi di istruzioni diventa:

```
do  
{  
  espressione1;  
  espressione2;  
}  
while (condizione) ;
```



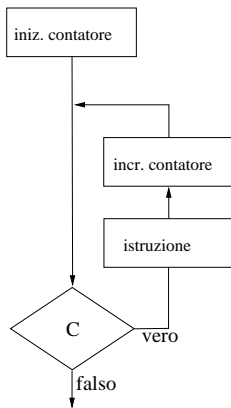
Ciclo enumerativo

- Il ciclo in cui è noto a priori il numero di iterazioni da eseguire si rappresenta in genere con l'istruzione `for`:

```
for (inizializzazione; condizione; incremento)  
    espressione;
```

- che per blocchi di istruzioni diventa:

```
for (inizializzazione; condizione; incremento)  
{  
    espressione1;  
    espressione2;  
}
```



Ciclo enumerativo

inizializzazione inizializza il valore della variabile contatore

condizione controlla il verificarsi della condizione di fine ciclo

incremento incrementa (o decrementa) la variabile contatore
al termine di ogni iterazione

- Ad esempio

```
int i;
```

```
int V[100];
```

```
/* inizializza gli elementi del vettore a zero */
```

```
for(i = 0; i < 100; i++)
```

```
    V[i] = 0;
```

```
/* incrementa di uno tutti gli elementi dispari */
```

```
for(i = 99; i >= 0; i-=2)
```

```
    V[i]++;
```

Funzioni

- Per poter risolvere problemi complessi, è necessario dividerli in sottoproblemi da risolvere separatamente
- In C i sottoproblemi vengono gestiti tramite le *funzioni*.
- Una funzione prende dei dati in ingresso (parametri) e restituisce un'uscita (parametro di uscita).
- Una funzione può anche non avere parametri di ingresso e/o non avere il parametro di uscita.

Dichiarazione di funzioni

- Le funzioni (come le variabili) devono essere dichiarate prima di essere utilizzate.
- La dichiarazione consiste nello specificare il nome della funzione, il numero dei parametri in ingresso ed il tipo di ciascuno, ed il tipo del parametro di uscita, che se assente si indica con `void`
- La sintassi è la seguente:

tipo_uscita nome_funzione (tipo_ingresso₁, . . . , tipo_ingresso_n);

- Ad esempio:

```
float radice_quadrata(float x);  
int massimo_comune_denominatore(int a, int b);  
int time();  
void stampa(char);
```

Definizione di funzioni

- La definizione di una funzione consiste nella descrizione delle operazioni che la funzione svolge, secondo la sintassi:

```
tipo_out nome_funzione(tipo_in1 nome_in1, . . . , tipo_inn nome_inn)  
{  
    dichiarazioni ed istruzioni  
}
```

- Tipi di ingresso ed uscita e nome devono corrispondere a quelli nella dichiarazione.
- I parametri di ingresso ricevono un nome mediante il quale verranno utilizzati nel corpo della funzione
- Il corpo della funzione contiene dichiarazioni di variabili *locali* alla funzione stessa (non visibili all'esterno) ed istruzioni.
- Se la funzione ha un tipo di uscita non `void`, il suo corpo dovrà contenere almeno un'istruzione di `return`.

Esempi

```
int max(int V[], int size)
{
    int i; /* variabile locale */
    int max; /* variabile locale */

    if(size <= 0)
    {
        printf("Vettore vuoto!\n");
        return 0;
    }

    max = V[0];
    for(i = 0; i < size; i++)
    {
        if(V[i] > max)
            max = V[i];
    }
    return max;
}
```

Uso di funzioni

```
int max(int, int); /* dichiarazione */

/* definizione */
void main()
{
    int V[4] = {1,3,2,4};
    int max;

    max = max(V,4); /* chiamata a funzione */
}

/* definizione */
int max(int V[], int size)
{
    int i; /* variabile locale */
    int max; /* variabile locale */

    if(size <= 0)
    {
        printf("Vettore vuoto!\n");
        return 0;
    }

    max = V[0];
    for(i = 0; i < size; i++)
    {
        if(V[i] > max)
            max = V[i];
    }
    return max;
}
```

Librerie

- Funzioni di utilità generale vengono raccolte in *librerie* che sono rese disponibili ai programmatori
- Ogni libreria ha un file di intestazione (*header*) contenente le dichiarazioni delle funzioni implementate
- Per poter utilizzare le funzioni di una libreria si deve includere il corrispondente file di intestazione tramite l'istruzione (non terminata da ';'):

```
#include <nomefile>
```

- Tale istruzione dice al compilatore di includere il contenuto del file nel codice sorgente prima di trasformarlo in eseguibile

Librerie comuni

`stdlib.h` funzioni di allocazione memoria, conversioni
(`malloc`, `free`, `atoi`)

`stdio.h` per le funzioni di input/output (`printf`, `scanf`)

`math.h` per funzioni matematiche avanzate (`sin`, `cos`,
`exp`)

`string.h` per manipolazione di stringhe (`strcmp`, `strstr`)

Output

- La funzione `printf` permette di visualizzare del testo sullo schermo
- Permette di inserire nella stringa da stampare il contenuto di variabili
- Esempi:

```
printf("Una riga con ritorno a capo\n");  
printf("x=%d e y=%d\n", x, y);
```

Significato

```
printf("Una riga con ritorno a capo\n");
```

Stampa la stringa `Una riga con ritorno a capo` **e**
termina la riga con un ritorno a capo (carattere ' \n')

```
printf("x=%d e y=%d", x, y);
```

Se `x=10` **e** `y=56` **stampa la stringa** `x=10 e y=56`.

Formato

- Nella stringa da stampare si possono inserire caratteri speciali (quelli che cominciano con `\`) e specificatori di conversione (quelli che cominciano con `%`).
- I caratteri speciali indicano ad esempio ritorno a capo (`\n`) o tabulazione (`\t`).
- Gli specificatori di conversione devono essere sostituiti nella stringa stampata dal contenuto della variabile in posizione corrispondente tra gli argomenti della funzione successivi alla stringa.

Specificatori di conversione

`%c` carattere

`%s` stringa

`%d` intero

`%f` reale

- Lo specificatore di conversione deve rispecchiare il tipo della variabile in posizione corrispondente
- Esempio

```
int x = 3;
```

```
float y = 0.5;
```

```
char s[5] = "word";
```

```
printf("x=%d, y=%f, s=%s\n", x , y , s);
```

Input

- La funzione `scanf` permette di acquisire dati dalla tastiera
- La sintassi della stringa di acquisizione è simile a quella per `printf`, ma contiene in genere solo specificatori di conversione e spazi.
- Uno specificatore di conversione indica come interpretare la porzione corrente di input fino al prossimo spazio o alla terminazione dell'input (ottenuta premendo "INVIO")
- Il valore ottenuto interpretando la porzione corrente tramite lo specificatore viene memorizzato nella variabile nella posizione corrispondente.

Esempi

```
scanf ("%d", &x) ;  
scanf ("%d %f %c", &x, &y, &z) ;  
scanf ("%s", w) ;
```

Nota

- Le variabili (tranne quelle di tipo stringa) devono essere precedute da &
- Il carattere speciale & indica un *puntatore* alla variabile di cui vogliamo aggiornare il contenuto.
- La variabile stringa deve essere abbastanza grande da contenere la stringa letta (altrimenti si verificherà una condizione di *buffer overflow*)

Esempio di programma di input/output

```
#include <stdio.h>

int main()
{
    int x;
    char buffer[5];

    printf("Inserire un numero intero: ");
    scanf("%d",&x);
    printf("Il numero inserito e': %d\n", x);
    printf("Inserire una parola: ");
    scanf("%s",buffer);
    printf("La parola inserita e': %s\n", buffer);

    return 0;
}
```

Calcolo del M.C.D.

```
#include <stdio.h>

int mcd(int, int);

int main()
{
    int a,b,r;

    printf("Inserire un numero intero A: ");
    scanf("%d",&a);
    printf("Inserire un numero intero B: ");
    scanf("%d",&b);

    if(a < b)
    {
        int tmp = a; /* variabile di scambio */
        a = b;
        b = tmp;
    }

    r = mcd(a, b);
    printf("Il M.C.D tra %d e %d e': %d\n", a , b , r);
    return 0;
}
```

Definizione della funzione `mcd`

```
int mcd(int a, int b)
{
    int r = a % b;

    while( r != 0)
    {
        a = b;
        b = r;
        r = a % b;
    }
    return b;
}
```

Compilazione ed esecuzione

- Il codice sorgente deve essere compilato tramite il compilatore C della macchina su cui si vuole eseguire il programma
- Ad esempio su UNIX si usa il compilatore `gcc`
- Tale compilatore è disponibile per Windows all'interno dell'ambiente *Cygwin* (<http://www.cygwin.com/>)
- Per un file `mcd.c` il comando di compilazione è

```
gcc -o mcd mcd.c
```
- Il programma generato (`mcd`) è direttamente eseguibile scrivendo il nome del file sulla linea di comando di un terminale
- Il nome del file deve essere scritto completo del percorso rispetto alla directory in cui siamo. Ad esempio se si trova nella directory corrente si scrive:

```
./mcd
```