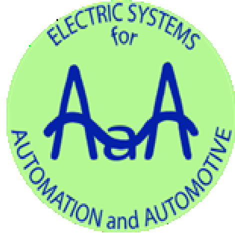
BASI DI LINGUAGGIO C



Tecnologie per il controllo di convertitori e azionamenti elettrici con laboratorio, a.a. 2020/2021

Prof. Manuele Bertoluzzo
Ing. Stefano Giacomuzzi
Laboratorio di Sistemi Elettrici per l'Automazione e la Veicolistica
Dipartimento di Ingegneria Industriale, Università degli Studi di Padova

manuele.bertoluzzo@unipd.it

Programmazione

- **Programmazione**: attività che porta a sviluppare un *algoritmo* che sia scritto in modo non ambiguo e conduca all'obbiettivo desiderato in un tempo finito.
- Il microprocessore può eseguire operazioni aritmetiche e logiche di base (addizione, sottrazione, moltiplicazione, negazione, congiunzione, ecc.); il programmatore deve scrivere una sequenza di tali operazioni che consentano al microprocessore di risolvere un problema.

Linguaggi di programmazine di *alto* e *basso livello*

- Il linguaggio di programmazione *C* è un linguaggio di *alto livello*, vale a dire che si avvicina alla terminologia umana, al contrario dei linguaggi di *basso livello*, che si avvicinano al linguaggio macchina (formato da 0 e 1). Un programma di basso livello è l'*Assembly*, mentre linguaggi di alto livello sono appunto il *C*, il *C++*, il *Java*, ecc.
- Il *C* è considerato il linguaggio maggiormente di basso livello tra i linguaggi di alto livello:
  - ha poche istruzioni
  - gestisce in maniera efficiente la memoria
  - è possibile inserire direttamente all'interno di un file in *C* del codice *Assembly*.

Linguaggi interpretati e compilati

- Il *file sorgente* (o *codice sorgente*) del programma è il file che contiene il testo scritto dal programmatore in un dato linguaggio di programmazione. Esso deve essere tradotto in linguaggio macchina, affinché l'elaboratore sia in grado di eseguirlo. Gli strumenti utilizzati a questo scopo possono essere *interpreti* o *compilatori*.
  - **Interprete**: programma in grado di leggere le istruzioni del codice sorgente, ne verifica la sintassi e le traduce ad una ad una in linguaggio macchina facendo eseguire al microprocessore della macchina il codice binario generato.

    Es.: *Basic*.
  - **Compilatore**: programma che legge le istruzioni del codice sorgente, ne verifica la sintassi e le traduce in linguaggio macchina trascrivendole in un secondo file (*object file, .obj*) che non è ancora eseguibile dal microprocessore. Serve un'altra fase di elaborazione, a cui provvede il *linker*, per incorporare nell'object file gli object files che contengono le funzioni esterne, solitamente raccolte nelle *librerie*. Il linker produce un terzo file direttamente *eseguibile* dal microprocessore.

    Es.: *C, C++*.
    - Vantaggi: velocità di esecuzione, semplicità di esecuzione.
  - Alcuni linguaggi sono prima **compilati** e poi **interpretati** da una *virtual machine.*

    Es.: Java, Python

Sintassi del linguaggio *C*

- *Case sensitive*: i caratteri maiuscoli sono considerati diversi dai minuscoli.
- *Istruzioni*: sequenze di caratteri terminate con un “;”.
- *Commenti*: annotazioni sul codice che vengono ignorate dal compilatore. Introdotti da *“//”, oppure aperti da “/*” e chiusi da “*/” nel caso di commenti multiriga.*
- Struttura di un programma *C*:

```
inclusione librerie
dichiarazioni di variabili esterne
lista funzioni
main()
{
    dichiarazioni di variabili locali
    istruzione 1
            …
    istruzione N
}
singole funzioni
```

```
#include <math.h>
float x;
int funzione1(float);
/* Lista funzioni */
main()
{
    long n;
    istruzione 1;
            …
    istruzione N;
}
int funzione1(float x)
{
    istruzione;
}
```

Variabili

Alla *dichiarazione* della variabile corrisponde anche la sua *definizione*, che fa sì che le venga riservato uno spazio in memoria (*allocazione*). Il *nome* di una variabile la identifica, il suo *tipo* ne definisce la dimensione e l'insieme delle operazioni che si possono effettuare su di essa.

Es.
```
int x;          // Definisce una variabile x di tipo integer
float y, z;     // Definisce due variabili (y e z) di tipo floating
```

Tipo	Dichiaratori	Tipo	Dichiaratori
character	`char`	long integer	`long int, long`
unsigned character	`unsigned char`	unsigned long integer	`unsigned long int, unsigned long`
short integer	`short int, short`	floating point	`float`
unsigned short integer	`unsigned short int, unsigned short`	double precision floating point	`double`
integer	`int`	long double precision floating point	`long double`
unsigned integer	`unsigned int, unsigned`	void type	`void`

Assegnazione di variabili e costanti

- L'assegnazione esplicita del valore iniziale alle variabili contestualmente alla definizione è detta *inizializzazione.*

  Es.

```
int x = 5;
float y = 7*x;
```

- Costanti intere ottali ed esadecimali (*hex*):
  - Una costante intera è considerata in base 8 quando è preceduta da uno `0`:

    Es. `014` vale 12_{10}
  - Una costante intera è considerata in base 16 (esadecimale, *hex*) se è preceduta da `0x` e può contenere le cifre da 0 a 9 e da A a F:

    Es. `0x2C` vale 44_{10}

- NB: Il formato *hex* viene utilizzato spesso per assegnare il valore dei bit nei registri:

  Es. 0000 0000 1100 0101 0001 1011 1110 0011 registro a 32 bit

  0 0 C 5 1 B E 3 0x00C51BE3

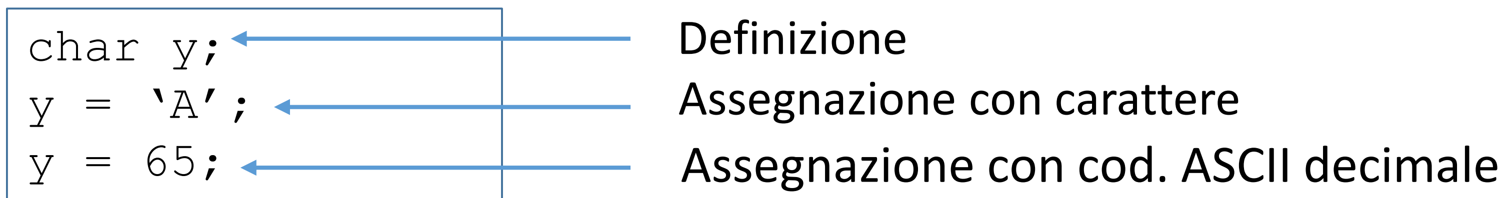
- Quando un certo valore costante viene utilizzato in modo ricorrente è opportuno assegnargli un nome simbolico, definendo all'inizio del programma un identificatore di costante mediante l'istruzione `#define`. Prima della compilazione, il preprocessore cerca i simboli definiti con direttive `#define`. I valori del comando `#define` possono essere di qualsiasi tipo.

  Es.

```
#define BASE 5        // Definisce il valore di BASE pari a 5
```

Caratteri e stringhe

- Codice ASCII: tabella che elenca le corrispondenze tra simboli grafici e numeri (es. A → 65). Per memorizzare un carattere, il *C* memorizza il numero intero corrispondente al suo codice ASCII.
- **Variabili carattere**: variabili di tipo intero su 8 bit, dette di tipo `char`.
- La variabile di tipo `char y` nell'esempio sottostante può essere usata sia come numero (65) che come lettera ('A').



- Le **stringhe** sono vettori di *char* terminati dal carattere `null` (di codice ASCII pari a 0, diverso dal carattere '0' che ha valore 48 in ASCII).
- Le **stringhe costanti** sono sequenze di *char* racchiuse da doppi apici "…"

Operatori in C

Un'*espressione aritmetica è* un insieme di variabili, costanti e richiami di funzione connessi da operatori aritmetici.

Operatori aritmetici	Descrizione
−	meno unario (negazione)
*	moltiplicazione
/	divisione
%	resto di divisione intera
+	addizione
−	sottrazione
=	assegnamento

Gerarchia operatori

Un'*espressione logica* è un'espressione che genera come risultato un valore vero o falso (per la precisione, 1 per vero e 0 per falso) e viene utilizzata dalle istruzioni di controllo.

Operatori relazionali	Descrizione
<	minore di
<=	minore o uguale a
>	maggiore di
>=	maggiore o uguale a
==	uguale a
!=	diverso da (not uguale a)

Operatori logici	Descrizione
!	not logico
&&	and logico
||	or logico

Altri operatori	Descrizione
()	chiamata di funzione
[]	indici di array
.	appartenenza a struttura
->	appartenenza a struttura referenziata da puntatore
~	complemento a uno
&	indirizzo di
*	indirezione
(tipo)	cast (conversione di tipo)
sizeof()	dimensione di
&	and su bit
^	xor su bit
[illegible]	[illegible]

Operatori aritmetici di assegnazione

Forma estesa	Forma compatta
x=x+1	x++
x=x-1	x--
x=x+5	x+=5
x=x-5	x-=5
x=x*5	x*=5
x=x/5	x/=5

Istruzioni condizionali

```
if (espressione)
    istruzione 1;
else
    istruzione 2;
```

Senza il ";"

```
espr1 ? espr2 : espr3
```

Se la valutazione di *espr1* restituisce vero, il risultato è uguale a *espr2*, altrimenti è uguale a *espr3*.

- Nell'istruzione **if** soltanto un'istruzione semplice viene controllata dalla valutazione di *espressione*.

Es.

```
if (espressione)
    istruzione 1;
    istruzione 3;
```

```
if (espressione)
    istruzione 1;
else
    istruzione 2;
    istruzione 3;
```

```
if (espressione)
    istruzione 1;
istruzione 3;
```

```
if (espressione)
    istruzione 1;
else
    istruzione 2;
istruzione 3;
```

In tutti i casi l'istruzione 3 viene eseguita indipendentemente dal verificarsi di *espressione*. Il posizionamento grafico di "`istruzione 3;`" non influisce sull'esecuzione del codice. Per includerla, bisogna comprendere tutte le istruzioni dopo l'`if` (o l'`else`) tra parentesi graffe.

Selezione multipla

- Se i risultati dell'espressione che indirizzano verso rami diversi sono valori costanti noti a priori, invece di *if…else*, si può utilizzare il più efficiente costrutto *switch*.
- Viene calcolata l'*espressione* e quindi il valore risultante cercato tra i vari *risultato x*, quindi:
  - se viene trovato un *risultato x* uguale al risultato dell'*espressione*, l'esecuzione prosegue alla prima istruzione del *blocco x* corrispondente;
  - se non viene trovato un *risultato x* uguale al risultato dell'*espressione*, se esiste il *blocco default* (opzionale), viene eseguito;
  - altrimenti nessun blocco viene eseguito.
- Le varie parti sono:
  - *espressione* è un'espressione che produce un risultato di tipo intero (char, short, int, long);
  - *risultato x* sono valori interi costanti noti al compilatore (numeri, #define, ecc.). L'ordine in cui sono elencati i diversi *risultato x* è ininfluente;
  - *blocco x* sono blocchi di codice (le parentesi graffe per i blocchi sono opzionali e in genere sono omesse se non contengono la definizione di variabili locali al blocco).
- Per eseguire il solo blocco corrispondente all'espressione e poi uscire dallo switch si utilizza l'istruzione `break` collocata come ultima istruzione di ciascun blocco.

```
switch (espressione)
{
case risultato 1:
    blocco 1;
    break;
case risultato 2:
    blocco 2;
    break;
...
default:
    blocco default:
    break;
}
```

Istruzioni iterative

- Le strutture iterative sono comunemente dette *cicli* o *loop*.
- In C i cicli sono controllati da una condizione di permanenza nel ciclo: fintantoché la condizione è vera, si esegue il corpo del ciclo (il blocco di codice da eseguire più volte).
- Tre possibili istruzioni per eseguire un ciclo:
  - Ciclo *while*
  - Ciclo *for*
  - Ciclo *do … while*

2020/21

Ciclo *while* e ciclo *for*

Ciclo *while*

Viene verificato che *espressione* sia vera, e in caso positivo viene eseguita *istruzione*. Il ciclo si ripete finché *espressione* risulta essere vera. Si può usare come espressione *1* per avere un ciclo infinito.

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

```
espr1;
   while (espr2)
      {
      blocco istruzioni
      espr3;
      }
```

```
i = 1;
   while (i<=5)
      {
      somma = somma+i;
      i++;
      };
```

Ciclo *for*

Fa eseguire il blocco di istruzioni finché *espr2* è vera.

- Viene calcolata *espr1* (solo la prima volta);
- Viene valutata *espr2*:
  - Se *espr2* è vera: - esegue il *blocco istruzioni*
    - esegue *espr3*;
    - torna a valutare la condizione *espr2*.
  - Se *espr2* è falsa: - passa ad eseguire le istruzioni successive al *blocco istruzioni*.
- Il ciclo *for* è un ciclo *while* riscritto in modo tale da raggruppare tra le parentesi tutto ciò che gestisce l'indice: inizializzazione (*espr1*), controllo (*espr2*) e aggiornamento (*espr3*).

Quando il numero di iterazioni è noto a priori (e quindi il ciclo è controllato da un indice), è preferibile (per chiarezza e stilisticamente) utilizzare un ciclo *for* che raggruppa in un punto solo l'inizializzazione, il controllo e l'aggiornamento dell'indice.

```
for (espr1; espr2; espr3)
    {
    blocco istruzioni
    }
```

```
for (i=1; i<=5; i++)
    {
    somma = somma+i;
    }
```

Ciclo *do ... while*

- Fa eseguire un blocco di codice fintantoché una certa condizione è vera.
- Valuta la condizione **dopo** aver eseguito il blocco.
- Anche se la *condizione* è inizialmente falsa, il *blocco* viene eseguito almeno una volta.

```
do
    {
    blocco istruzioni
    }
while (espressione);
```

Con il “;”

I puntatori

- Il *puntatore* è una variabile che contiene l'indirizzo di memoria di un oggetto (variabile o costante) → ad ogni variabile infatti corrisponde un nome, una locazione di memoria, e l'indirizzo della locazione di memoria.
- L'informazione relativa al tipo è necessaria per permettere ai puntatori di conoscere la dimensione dell'oggetto.
- L'operatore di indirizzo `&` applicato ad una variabile restituisce l'indirizzo di memoria di quella variabile, ossia ne restituisce il puntatore.
- L'operatore di *deriferimento* (*dereference*) o di *indirezione* (*indirection*) `*` applicato ad un indirizzo di memoria restituisce il valore contenuto in quell'indirizzo.

```
float x;          //dichiara una variabile float
float *punt;      //dichiara un puntatore ad una variabile float
x = 7;            // assegna un valore alla variabile x
punt = &x;        // assegna al puntatore l'indirizzo della variabile
```

```
int x = 7;
int *punt = &x;
```

`&punt` esprime l'indirizzo di `punt`. L'*indirezione* di un indirizzo restituisce il valore memorizzato a quell'indirizzo → `*&punt` esprime il valore memorizzato all'indirizzo di `punt`, cioè il valore contenuto in `punt`, che corrisponde al'indirizzo di `x`.
`punt` esprime l'indirizzo di `x`, quindi la sua *indirezione* `*punt` rappresenta il contenuto di `x`. L'espressione `&*punt` equivale all'indirizzo del contenuto di `x` cioè l'indirizzo di `x`, che è uguale a `punt`.

Le funzioni

- Le funzioni *ritornano* (o *restituiscono*) un valore al programma chiamante a partire da una o più variabili di ingresso. Esse vanno indicate come segue:

  tipo_di_risultato nome_funzione (argomenti)

- Sintassi di *dichiarazione* e *definizione* di una funzione:

```
#include <stdio.h>
double cubo(float);
main()
{
     float a = 10;
     double b = 4;
     b = cubo(a);
}
double cubo(float c)
{
return (c*c*c);
}
```

Dichiarazione della funzione

Invocazione della funzione

Definizione della funzione

Ritorno di un valore

- Il **tipo** di una funzione è determinato dal tipo del valore restituito e dal tipo, numero e ordine dei suoi parametri.
- Lo **scope** di una funzione si estende dal punto in cui viene definita fino a fine file: la funzione può essere utilizzata solo dalle funzioni che nello stesso file seguono la sua definizione.
- Il **prototipo** di una funzione è una dichiarazione che estende lo scope della funzione.

Le funzioni

- Se la funzione non restituisce valori bisogna indicare il tipo `void`. Se non si mette nulla viene supposto `int`.
- Se la funzione non ha parametri, si indica `void` tra le parentesi. Se non si indica nulla, il compilatore C non attua alcun controllo sui parametri (invece per un compilatore C++ è come se l'argomento fosse esplicitamente void).
- Si chiama una funzione indicandone il nome seguito da una coppia di parentesi contenenti i valori da elaborare (separati da virgole); se la funzione non richiede variabili, le parentesi sono vuote o indicano `void`, ma devono esserci.

  Es.
  ```
  int massimo (int m,n)
  float potenza (float x, int n)
  void stampa (int *punt)
  void scrivi ()
  ```
- La funzione *massimo* usa due variabili intere (m,n) e fornisce un valore intero. La funzione *potenza* usa una variabile reale (`x`) e una variabile intera (`n`) e fornisce un valore reale. La funziona *stampa* usa un puntatore ad una variabile intera (`punt`) e non fornisce nessun risultato (`void`). La funzione *scrivi* non ha variabili (`void`) e non fornisce nessun risultato (parentesi vuota o, in alternativa, `void`).
- È importante sapere che il programma viene eseguito a partire dalla parola chiave *main* che caratterizza la funzione "programma principale". La funzione *main* è una funzione come tutte le altre, l'unica sua particolarità è che deve esistere nel testo di ogni programma. Può essere messa alla fine, come ultima funzione. Se si vuole mettere invece il *main* all'inizio del programma occorre definire le funzioni che verranno usate, dopo i comandi del precompilatore e prima del *main* (*dichiarazione* della funzione).

Variabili locali ed esterne

- Le variabili locali vengono definite all'interno della funzione ed hanno classe di allocazione **automatica**: vengono create ogni volta che si esegue la funzione ed eliminate ogni volta che questa termina (perdendone il valore).
- Le variabili locali di classe di allocazione **statica** invece non vengono mai rimosse dalla memoria per cui non perdono il loro valore quando la funzione termina (resta quello che aveva al termine della chiamata precedente). Es. `static int x = 0;`
- Le variabili esterne vengono definite esternamente al corpo delle funzioni:
  - In testa al file, dopo gli `#include` ed i `#define`;
  - In alternativa, tra una funzione e l'altra.
- Le variabili esterne sono visibili e condivisibili da tutte le funzioni. Hanno classe di allocazione statica: non vengono mai rimosse dalla memoria e, salvo inizializzazione esplicita, vengono inizializzate automaticamente a 0.
- Una variabile locale con lo stesso nome di una esterna copre la visibilità di quella esterna, alla quale quindi la funzione non può accedere con quel nome.
- Lo *specificatore di classe di allocazione* `extern` permette di estendere la visibilità delle variabili esterne: viene premessa ad una definizione di variabile per trasformarla in dichiarazione (non riserva memoria). Indica al compilatore che la variabile è *definita* (con allocazione di memoria, senza `extern`) altrove (più avanti nello stesso file o in un altro file) → `extern int x;`
- All'interno di una funzione, il comando `extern` associato ad una variabile, la identifica come variabile esterna definita dopo la funzione stessa o in altro file.

Passaggio dei parametri nelle funzioni

```
#include <stdio.h>
void swap(int *, int *);
main()
{
      int x = 5, y = 8;
      swap(&x, &y);
}
void swap (int *a, int *b);
{
      *a = 15;
      *b = 24;
}
```

- Il *main* alloca `x` e `y`, assegna loro il valore 5 e 8, rispettivamente, e chiama la funzione `swap` passandole l'indirizzo di `x` e `y` (`&x` e `&y`) presente in una variabile temporanea.
- Alla chiamata di `swap`, gli indirizzi di `x` e `y` vengono copiati by-value in `a` e `b`.
- La funzione `swap` accede a `x` e `y` come `*a` e `*b`, modificandoli in 15 e 24.
- Quando `swap` termina, `x` e `y` valgono 15 e 24.

I puntatori a funzione

- Si può chiamare una funzione utilizzando l'indirizzo di memoria dal quale inizia il codice eseguibile della funzione stessa.
- Definisce la variabile `nome` come puntatore a una funzione che restituisce un valore del *tipo* indicato e richiede i *parametri* indicati

```
tipo (*nome) (parametri);
```

```
int (*puntfunz) (float);
```

- `int`: Indica che l'*indirezione* del puntatore restituisce un intero
- `*`: Indica la natura di puntatore
- `(float)`: Contiene la lista dei parametri della funzione: sono proprio queste parentesi ad indicare che `puntfunz` è un puntatore a funzione
- `( )` attorno a `*puntfunz`: Sono indispensabili per distinguere la dichiarazione di un puntatore a funzione da un prototipo di funzione

```
int *puntfunz (float);
```

Costituisce il prototipo di una funzione che restituisce un puntatore di tipo `int` e richiede un parametro di tipo `float`

Thanks for your kind attention