Modulo 3: Programmazione in linguaggio C

—

Curvatura informatica Classi Quarte
Liceo Scientifico ‘A. Pacinotti’
A.S. 2021/22

Prof.ssa Claudia Vacca

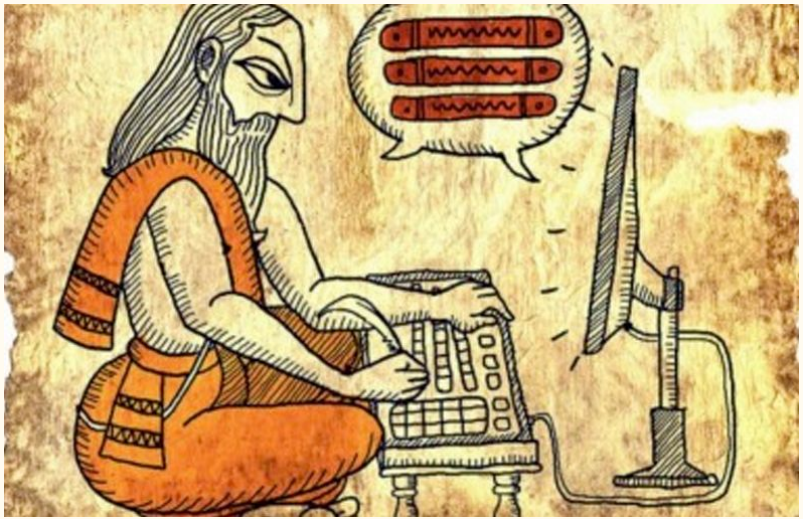
Il modulo:

- 9 ore lezione + 1 ora test
- 5 ore lavoro autonomo
- Calendario:
  - lezione ogni venerdì 15:30-17:00 (termine lezioni 29/04)
  - test finale venerdì 06/05

Lezione 1:

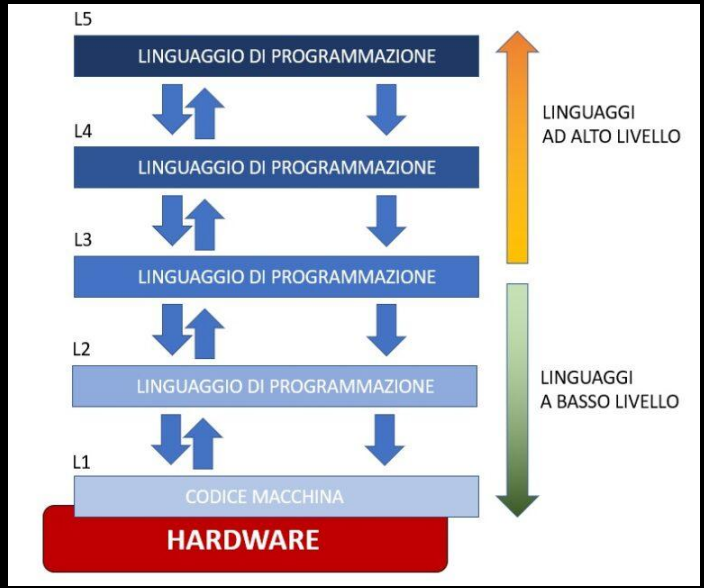
- Elementi di programmazione in linguaggio C
- Differenze principali C/Python
- Struttura di un programma in C (file sorgente, compilazione, file eseguibile)
- L'ambiente di lavoro: DevC++
- Variabili e tipi di dato

Linguaggi di programmazione

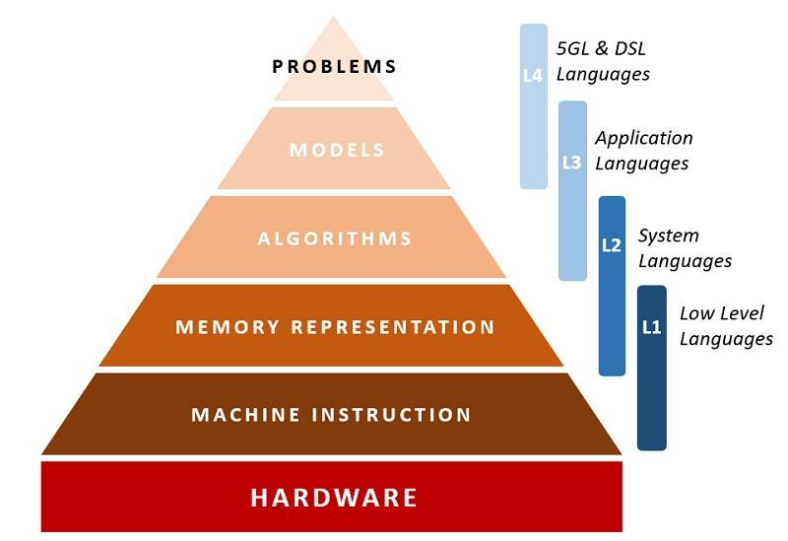


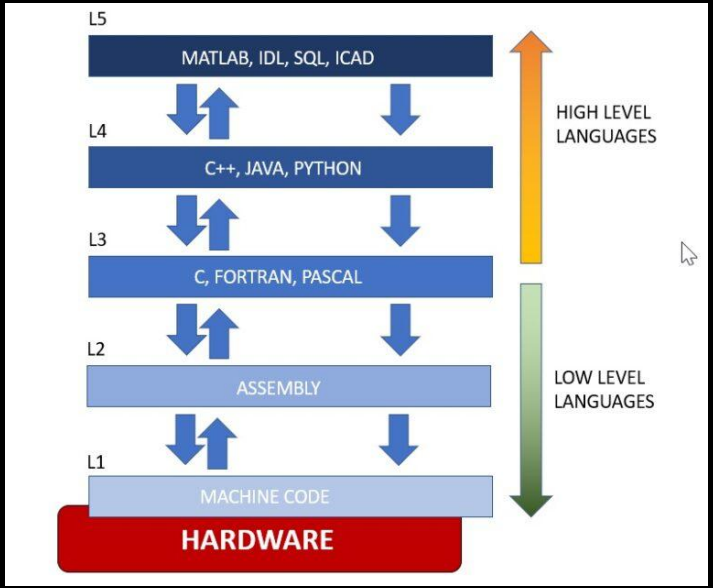
*Uomo e macchina che comunicano in sanscrito

- linguaggio macchina ➡ linguaggio naturale



Linguaggi di programmazione





Evoluzione storica...

- Nuove esigenze tecnologiche ➡ nuovi linguaggi di programmazione
- assembly: corrispondenza 1 ➡ 1 con istruzioni in linguaggio macchina

Evoluzione storica...

- 1956: FORTRAN (FORmula TRANslator, IBM)
- 1960: COBOL (COmmon Business Oriented Language) *commerciale/gestionale*
- 1968: Pascal, Politecnico di Zurigo, *didattico, versatile, general-purpose*
- 1972: linguaggio C, base dello UNIX

- ‘80: DBII, Oracle, linguaggi per gestione database
- ‘90: OOP (Object Oriented Programming), Java, C++
- ‘90-’00: HTML, XML, JavaScript per internet

Attenzione...

- *Evoluzione storica ≠ miglioramento in senso assoluto*
- Quasi tutti i linguaggi di programmazione sono ancora in uso
- Ogni linguaggio ha un campo di applicabilità diverso

- assembly/macchina: necessaria conoscenza specifica processore
  - CONTRO: no portabilità
  - PRO: massima efficienza nell'uso della macchina

Il linguaggio C

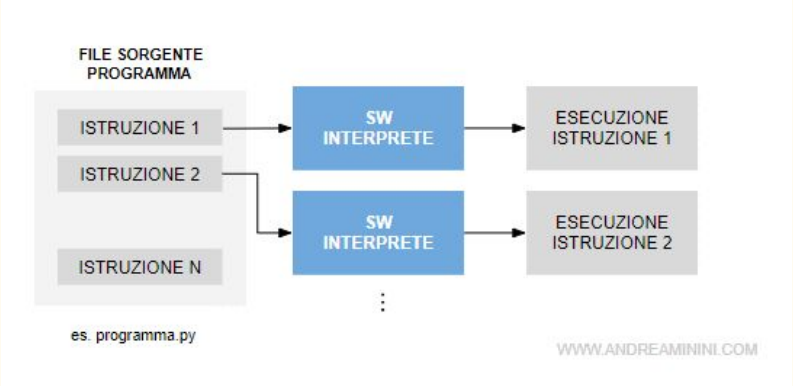
- linguaggio di medio/alto livello
- 1972, Dennis Ritchie
- implementazione sistemi operativi
- sostanzialmente invariato
- sintassi estesa (C++, oggetti)

Python e C a confronto

Python	C
1. 1989-91	1. 1972-73
2. livello alto	2. livello medio-alto
3. *general-purpose*	3. *general purpose*
4. linguaggio interpretato esecuzione lenta	4. linguaggio compilato esecuzione veloce
5. procedurale object-oriented (classi, oggetti)	5. procedurale imperativo (blocchi, funzioni)
6. librerie estese e pronte	6. meno librerie pronte
7. dinamico, sintassi semplice (no tipo di variabili, no puntatori, poche strutture dati, …)	7. sintassi più complessa (dichiarazione tipo di variabili, puntatori, molte e diversificate strutture dati, …)
8. migliore efficienza nell'uso della memoria (garbage collector)	8. allocazione della memoria manuale (maggiore complessità, più bug)
9. Debugging semplice (un'istruzione alla volta)	9. Debugging complesso (tutti i bug mostrati insieme)

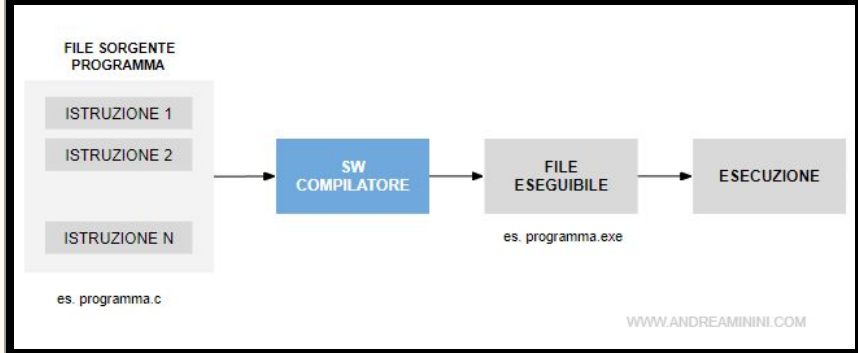
Python e C a confronto

Linguaggi interpretati (python, JavaScript, …)



1. interpretazione/traduzione ed esecuzione di una singola istruzione alla volta
2. se serve rieseguire il programma, è necessario ripetere anche l'intero processo di interpretazione

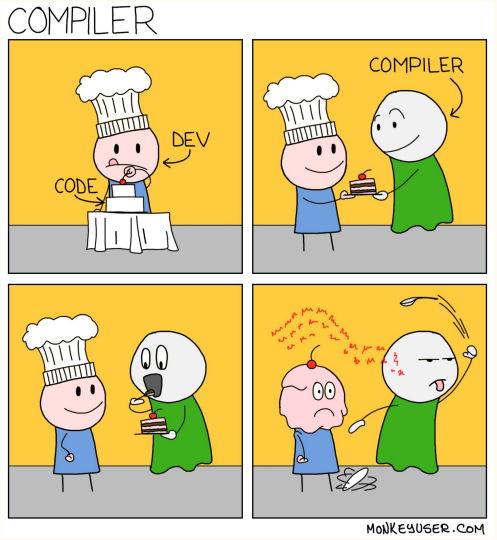
Linguaggi compilati (C, C++, FORTRAN, …)



1. analisi del codice nel file sorgente
2. compilazione/traduzione in linguaggio macchina di tutto il programma
3. file eseguibile (specifico della macchina), anche più volte senza necessità di ricompilare
4. esecuzione del programma intero

Come funziona un **compilatore**?

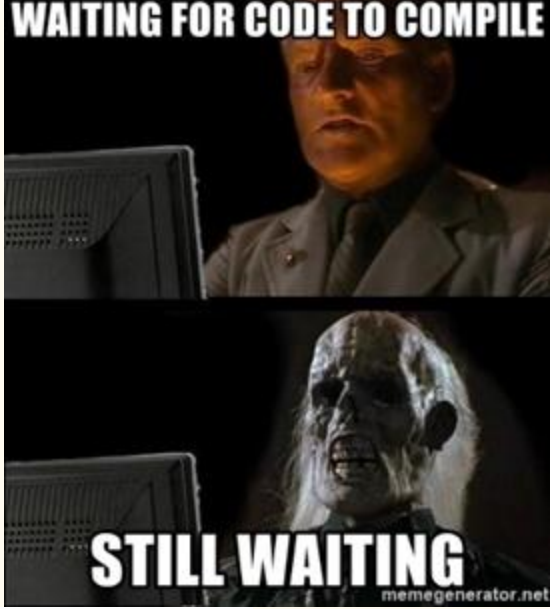
- **pre-processing:** rimozione dei commenti, inclusione di file
- **analisi lessicale** (*scanning*): riconoscimento lessemi (comandi, dati, simboli, ...)
- **analisi sintattica** (*parsing*): organizzazione/controllo sintassi grammaticale del linguaggio (istruzioni, dichiarazioni)
- **analisi semantica:** controllo coerenza/consistenza di quanto scritto

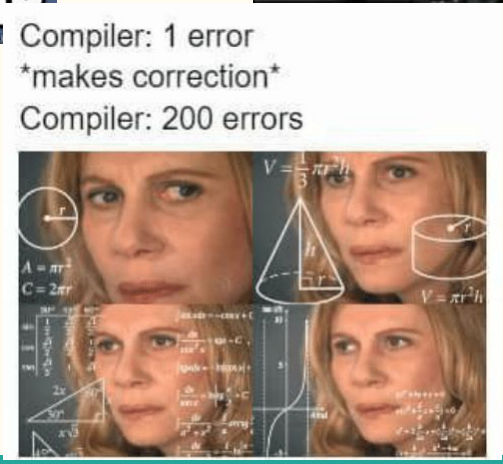


ERRORI:
- *fatal errors:* il compilatore sospende la compilazione, nessun file oggetto viene generato
- *warnings:* errori lievi che non compromettono la creazione del file oggetto

- **generazione codice:** traduzione in linguaggio assemblativo
- **ottimizzazione codice:** riesame del codice per migliorarne il tempo di esecuzione
- **linker:** collegamento alle librerie e conversione del programma oggetto in eseguibile (.obj ➡ .exe)

Come funziona un **compilatore realmente**?





E come andrà...


When you switch to C after coding
in python for 3 months
C compiler:
I am once again asking
for semicolon


COMPILER ISSUES
COMPILER ISSUES
EVERYWHERE
makeameme.org

Gli ambienti di sviluppo integrato IDE

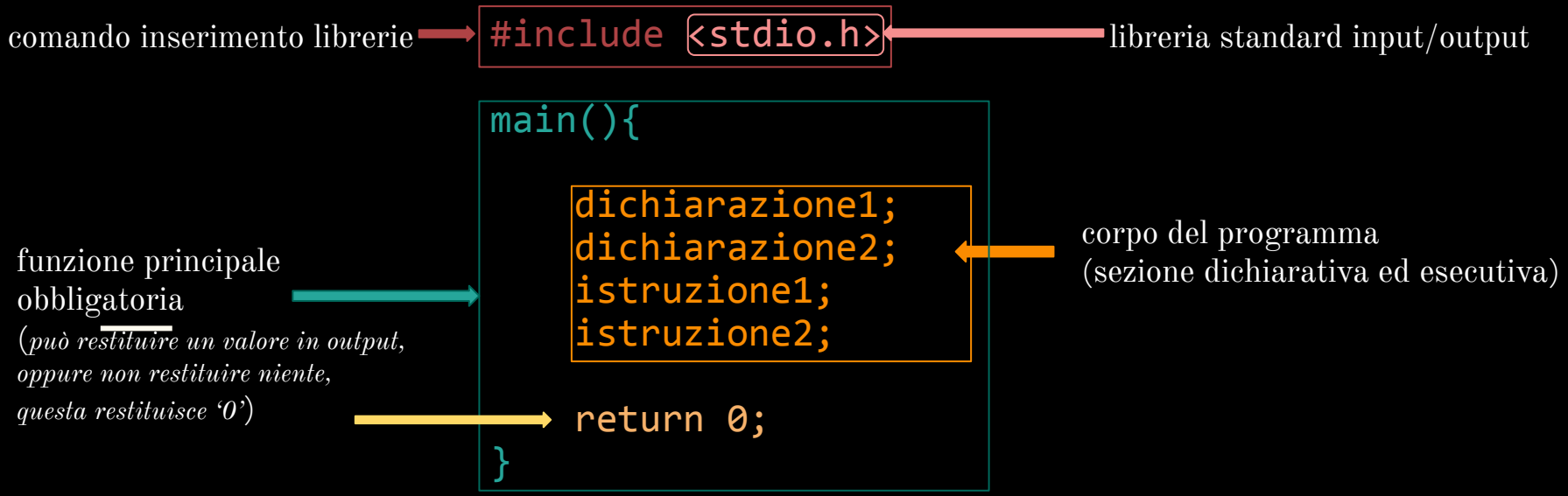
Software con tutti gli strumenti necessari in fase di collaudo del codice

- editing
- evidenziazione sintassi
- compilazione
- debugging
- esecuzione

Dev-C++

XCode, Code::Blocks (per Mac)

La struttura base di un codice



Dichiarazione di variabili

variabile: area della memoria destinata a contenere un particolare tipo di dato

creare una variabile:

- definirne un nome/identificatore attribuito in modo univoco *[case sensitive!]*
- definirne la natura/il tipo

```
int pippo;
pippo=3;
int pluto=2;

int chimera=pippo+pluto;
```

Successivamente:
-inizializzare/assegnare un valore/espressione alla variabile
-leggerla
-modificarla

I tipi primari

Tipo	**Dato**	**Bit**	**Intervallo di valori rappresentabili**
`char`	carattere ASCII	8	[-128 : 127]
`int`	intero	32	[-2147483648 : 2147483647]
`float`	virgola mobile	32	~[$-3{,}4\cdot10^{38}$: $3{,}4\cdot10^{38}$]
`double`	virgola mobile in doppia precisione	64	~[$-1{,}7\cdot10^{308}$: $1{,}7\cdot10^{308}$]
`void`	nessuno/qualunque		

I modificatori di tipo (per `char` e `int`)

Tipo	Modifica
`signed`	con segno
`unsigned`	intero
`long`	virgola mobile
`short`	minore estensione

I modificatori su `int`

`int`	Intero con segno, 32 bit
`short`	Intero con segno, 16 bit
`long`	Intero con segno, 64 bit
`unsigned int`	Come int, ma senza segno
`unsigned short`	Come short, ma senza segno
`unsigned long`	Come long, ma senza segno

I modificatori su `int`

Tipo	Dato	Bit	Intervallo di valori rappresentabili
`short/signed short`	intero con segno	16	[-32768 : 32767]
`unsigned short`	intero positivo	16	[0 : 65535]
`int/signed int`	intero con segno	32	[-2147483648 : 2147483647]
`unsigned int`	intero positivo	32	[0 : 4294967295]

Dichiarare e inizializzare variabili

Singoli apici!

char:

```
char lettera;
lettera = ‘c’;
char iniziale = ‘v’;
```

double:

```
double decimo, decimomuro;
decimo=3.1;
decimomuro=1.3;
```

int:

```
int  numero;
numero=3;
int Numero=4;
```

Case sensitive!

float:

```
float zattera=1.1;
```

PUNTO, no virgola !!!

Tecnica delle costanti manifeste

Dichiarare come costanti tutti i numeri (non variabili) che compaiono nel codice.
Es. num_giorni_anno; tasso_conversione; tasso_interesse, ...

```
#include <stdio.h>

main(){

      const int giorni_anno=365;

      const int sec_giorno=86400;

      int sec_anno = giorni_anno*sec_giorno;

}
```

Terra o Venere?

Dichiarare e inizializzare costanti

Dichiarare variabili non modificabili

Costante: associare un identificatore (e un tipo) a un valore, *immodificabile* nel corso dell'esecuzione del programma

```
#include <stdio.h>

main(){
    const double PI=3.14;
}
```

```
#include <stdio.h>
#define PI 3.14;

main(){
}
```

Dichiarare e inizializzare costanti

Buona norma: utilizzare identificatori in caratteri MAIUSCOLI.

```
#include <stdio.h>

main(){
    const double PI=3.14;
}
```

-istruzione: punto e virgola necessario;
-allocazione spazio in memoria;
-costante tipizzata: valori ammissibili definiti dal tipo

```
#include <stdio.h>
#define PI 3.14;

main(){
}
```

-direttiva per il compilatore: punto e virgola facoltativo;
-il compilatore sostituisce a tutte le occorrenze dell'identificatore il valore;
-non occorre indicare il tipo

Funzioni di input-output

- libreria ‘`stdio.h`’
- output ⟶ `printf(), putchar()`
- input ⟶ `scanf(), getchar()`

- fondamentale specificare il formato/tipo del dato

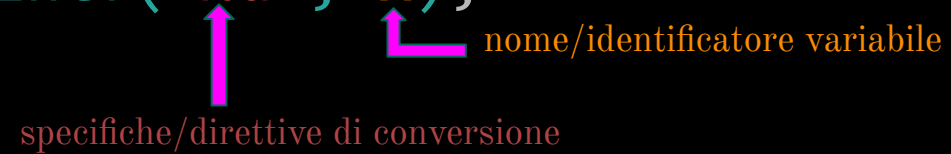
Output: `printf()`

- stampa di una stringa

```
printf(“No means no”);
```

- stampa del valore di una variabile

```
printf(“%d”, x);
```



Specifiche/direttive di conversione per `printf()`

Tipo	Espressione	A video
%c	char	singolo carattere
%d (%i)	int	intero con segno
%e (%E)	float or double	formato esponenziale
%f	float or double	reale con segno
%g (%G)	float or double	utilizza %f o %e in base alle esigenze
%o	int	valore base 8 senza segno
%p	pointer	valore di una variabile puntatore
%s	array of char	stringa (sequenza) di caratteri
%u	int	intero senza segno
%x (%X)	int	valore base 16 senza segno

Output: `printf()`

- stampa di una variabile intera

```
int x=4;
printf("%d", x);
```

- stampa di una variabile in formato decimale

```
float y=3.467;
printf("%f", y);
printf("%.2f", y);
```

Quante cifre dopo la virgola? ⟶

Output: printf()

- stampa di una variabile in formato decimale

```
float p=23.6789;
printf("%6.2f \n",p);
printf("%-6.2f \n",p);
printf("%8.2f \n",p);
printf("%-8.2f \n",p);
```

```
 23.68
23.68
   23.68
23.68
```

ampiezza minima del campo

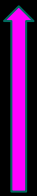
allineamento a sinistra

Output: `printf()`

- stampa di più variabili

```
int x=4;
double y=3.4;
printf(“%d %f”, x, y);
```

elenco delle specifiche



nomi delle variabili separati da virgole (separatori)
L’ordine è tutto!

Output: `printf()`

- stampa di stringhe e valori

```
int x=4;
printf(“La variabile vale %d”, x);
printf(“La x vale %d e ha valore intero”, x);
```

Elementi di tabulazione (*sequenze di escape*)

Sequenza	Rappresenta
\n	Nuova riga
\r	Ritorno a capo
\t	Tabulazione
\b	Backspace
\u	Visualizza il carattere seguente in maiuscolo
\l	Visualizza il carattere seguente in minuscolo
\\	Un carattere backslash letterale
\’	Un ‘ letterale all’interno di una stringa racchiusa tra apici singoli
\”	Un “ letterale all’interno di una stringa racchiusa tra apici doppi

Output: `printf()`

- stampa di stringhe con elementi di tabulazione

```
int x=4;
printf(“La x vale %d \n La x ha valore intero”, x);
printf(“Il mio nome e\’ \”Claudia\” \n”);
```

Buona norma:
inserire una tabulazione \n (newline) alla fine di ogni stringa.
Non è automatico!

Output: `putchar()`

- stampa di un solo carattere

```
char x='a';
putchar(x);
```
stampa di variabile `char`

—

```
putchar('\n');
```
stampa di sequenza di escape

```
putchar('c');
```
stampa di carattere singolo

```
putchar('vai');
```
stampa di stringa (sovrascrittura)

Input: `scanf()`

- inserimento di un valore in ingresso

```
int x;
scanf(“%d”,&x);
```

Specifiche di conversione per `scanf()`

`%d`	→ `int`
`%u`	→ `unsigned int`
`%x`	→ `unsigned int` in esadecimale
`%hd`	→ `short int`
`%ld`	→ `long int`
`%lu`	→ `unsigned long int`
`%f, %e, %g`	→ `float` (equivalenti)
`%lf`	→ `double`
`%Lf`	→ `long double`

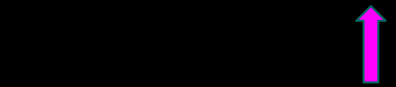
Input: `scanf()`

- viene prelevato il massimo numero di caratteri compatibili con il tipo richiesto dalla specifica
- non appena si incontra un carattere non compatibile con la specifica si ferma
- i caratteri rimanenti rimangono a disposizione per le successive letture “restano nel buffer della tastiera”, ossia in memoria

Input: `scanf()`

- inserimento di molti valori

```
int x;
double y;
scanf(“%d%lf”,&x,&y);
```

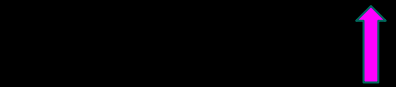


Lo spazio tra le specifiche non è necessario.
Se compare, in esecuzione ci si aspetta uno spazio tra i numeri inseriti!

Input: `scanf()`

- inserimento di molti valori

```
int x;
double y;
scanf(“%d%lf”,&x,&y);
```



Lo spazio tra le specifiche non è necessario.
Se compare, in esecuzione ci si aspetta uno spazio tra i numeri inseriti!

Input: `getchar()`

- inserimento di un solo carattere

```
char pippo;
pippo=getchar();
```
riempimento di variabile `char`

```
char pluto=getchar();
```
[NON TUTTI I COMPILATORI LO ACCETTANO]

I commenti...

- tutto ciò che deve essere ignorato dal compilatore
- BUONA NORMA: commentare i codici, per gli altri utenti ma anche per gli autori stessi!

/* *commento su*
più
linee */

// *commento su singola linea* [più tipico di C++]

Gli operatori

unari, binari, ternari

- aritmetici
- relazionali/logici
- assegnamento
- altri...

Operatori aritmetici

binari (addizione, sottrazione, moltiplicazione, divisione, modulo-*resto intero*, potenza)

unari (segno, incremento, decremento)

Purpose	C or C++
add	x + y
subtract	x - y
multiply	x*y
divide	x/y
modulus	x%y
exponentiations	pow(x,y)
unary plus	+x
unary minus	-y
postincrement	x++
preincrement	++x
postdecrement	x--
predecrement	--x

Operatori aritmetici: modulo `%`, `pow()`

- `int x,y;` solo variabili intere
  `int z=x%y;` resto intero della divisione tra x e y

- `pow(x,y);` potenza base x, esponente y

N.B. `#include <math.h>` libreria funzioni matematiche

Operatori aritmetici: incremento ++, decremento --

- `int x=9;` solo variabili intere
  `x++;` il valore di x viene incrementato di una unità -> x=10

- `int y=1;`
  `y--;` il valore di y viene decrementato di una unità -> y=0

Operatori aritmetici: pre-incremento, post-decremento

`x++ x--`

il valore di x viene in/decrementato dopo l'uso - POST-in/decremento

---

`++x --x`

il valore di x viene in/decrementato prima dell'uso - PRE-in/decremento

Operatori relazionali

Purpose	C or C++
less than	x < y
less than or equal	x <= y
greater than	x > y
greater than or equal	x >= y
equal	x == y
not equal	x != y

Risultati vero/falso, rappresentati da valori interi

VERO = 1

FALSO = 0

Operatori logici

Purpose	C or C++
false value	0
true value	non-zero
logical negation	!x
logical and	x && y
logical inclusive or	x || y

Operatori di assegnamento

Impostare/reimpostare il valore di una variabile con il risultato di un'operazione di tipo logico o matematico

Purpose	C or C++
assignment	x = y
add assignment	x += y
subtract assignment	x -= y
multiply assignment	x *= y
divide assignment	x /= y
modulus assignment	x %= y
right shift assignment	x >>= n
left shift assignment	x <<= n
and assignment	x &= y
or assignment	x |= y
xor assignment	x ^= y

Priorità/associatività

Priorità o precedenza: in presenza di più operatori, determina in quale ordine agiscono

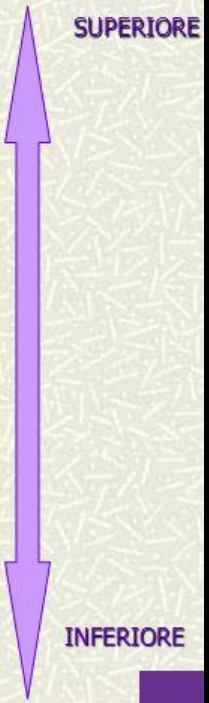
Associatività: in presenza di più operatori con la stessa priorità, determina in quale ordine agiscono

\+ e − sono associativi a sinistra (stessa priorità)
+= e −= sono associativi a destra (stessa priorità)

a − b + c ➡ (a−b)+c
a −= b += c ➡ a −= (b += c)

Tabella di priorità/associatività

Classe	Operatori	Associatività
primari	() [] → .	Da sinistra a destra
unari	cast sizeof & * - + ~ ++ -- !	Da destra a sinistra
moltiplicativi	* / %	Da sinistra a destra
additivi	+ -	Da sinistra a destra
scorrimento	<< >>	Da sinistra a destra
relazionali	< <= > =>	Da sinistra a destra
uguaglianza	== !=	Da sinistra a destra
AND tra bit	&	Da sinistra a destra
XOR tra bit	^	Da sinistra a destra
OR tra bit	|	Da sinistra a destra
AND logico	&&	Da sinistra a destra
OR logico	||	Da sinistra a destra
condizionale	?:	Da destra a sinistra
assegnamento	= += -= *= /= %= >>= <<= &= ^=	Da destra a sinistra
virgola	,	Da sinistra a destra



Espressioni

-qualunque combinazione valida di costanti variabili e operatori, con valore associato

```
int a,b,c;
4*a+b-c/3;
(3a==b) || (c<=4);
```

Operatore “virgola”

- agisce tra due espressioni
- il valore risultante è quello dato dall’ultima

—

```
int i = (j=2, j+4);
```

il risultato è i=6

Operatore ternario “? :”

```
espr1 ? espr2 : espr3
```

-valuta la verità di espr1
-se vera, il risultato è espr2
-se falsa, il risultato è espr3

```
int h,i=1,j=2,k=3;
h = (i==1) ? 2*j : 2*k;
h = (i!=2) ? 3*j : 4*j;
```

il risultato è ...
il risultato è ...

Conversioni di tipo - Casting

Possibilità di convertire una variabile da un tipo a un altro

Casting implicito

Conversione automatica

- **promozione**
  no perdita di dati
  tipo “inferiore” → “superiore”

- **coercizione**
  perdita di dati
  tipo “superiore” → “inferiore”

Casting esplicito

Indicazione esplicita del tipo di destinazione nel quale si vuole convertire la variabile di partenza

- promozione

```
int i=45;
float f;
f=(float)i;
```

[f=45.00]

- coercizione

```
int i;
float f=27.2;
i=(int)f;
```

[i=27]

Casting implicito

Promozione automatica

- in presenza di costanti/variabili di diverso tipo
- il risultato dipende dal tipo “maggiore”
- in presenza di `char/int/float/double` → `double`
- in presenza di `char/int` → `int`

```
int i;
char c;
double d;
float f;
d=i*c + d/f;
```

i*c → int subito (domina i)
d/f → double subito (domina d)
i*c → double alla somma (domina il 2° addendo d/f)

Casting implicito

Coercizione automatica

- perdita di dati

```
int i;
double d=48.89;
i=d;
```

→ i registra solo la parte intera di d (48)

Operatore sizeof()

- operatore unario
- valore restituito è intero senza segno (unsigned int)
- restituisce la dimensione dell'operando (type o espressione)
- parentesi non necessarie in caso di espressione

```
sizeof(int);
```
restituisce la dimensione del tipo/identificatore int

```
sizeof i;
```
restituisce la dimensione della variabile i

```
int i;
sizeof i;
```
equivalente a
```
sizeof(int);
```

Istruzioni di controllo

Programmazione strutturata

- Paradigma di programmazione entro la programmazione procedurale
- Flusso di esecuzione evidente dalla struttura sintattica
- Metodo top-down → Suddividere problemi in sottoproblemi

Strutture di controllo

- Modificare l'ordine di esecuzione sequenziale delle istruzioni
- Controllare il flusso di esecuzione

- **selezione** (`if`, `if-else`, `if-else if`, `switch`)
- **iterazione/cicli** (`for`, `while`, `do-while`)
- **salto** (`break`, `continue`)

Istruzioni di salto - break

- Provoca l'uscita immediata dal blocco (switch, for, while, do-while)

```
while(x<100){
    if(x<50) break;
    printf("ciao");
}
```

eseguita solo se x>50 (e anche minore di 100)

Istruzioni di salto - continue

- Nei cicli (while. do-while, for) permette di passare all'iterazione successiva

```
while(x<100){
    if(x<50) break;
    printf("ciao");
}
```

eseguita solo se x>50 (e anche minore di 100)

Istruzioni di salto - goto

- Permette il salto a qualunque altra istruzione all'interno del programma, preceduta da una label che la identifichi
- Fortemente osteggiata e bandita dalla programmazione strutturale
- Sovente causa dei "*codici-spaghetti*" ingarbugliati e incomprensibili

```
goto label;
```

Boss: How is the italian guy doing?

The italian guy he hired:

```
#define PASTA_LA_VISTA 0;

int main(void){
                goto Lasagna;
Tagliatelle:    goto Spaghetti;
Lasagna:        goto Fusilli;
Spaghetti:      return PASTA_LA_VISTA;
Fusilli:        goto Tagliatelle;
}
```

Co-Worker: Let's create some well organized and structured code.

Me:

Nobody:
My code:

What are you doing?
Me: Coding.

Teorema di Jacopini-Bőhm

Con i tre costrutti fondamentali, **concatenazione(sequenza)**, **iterazione** e **selezione**, è possibile codificare tutti gli algoritmi computabili.

Qualunque linguaggio ammetta queste tre figure strutturali fondamentali è definito completo.

Tutti i linguaggi che ammettano le tre figure strutturali fondamentali sono equipotenti.

Selezione semplice - `if`

- Test di verità su condizione logica o istruzione (V/F)
- Se VERO → esegui istruzione o blocco di istruzioni
- Se FALSO → non esegue niente

```
if(x==8)
    y=x+2;
```

[su singola istruzione, graffe non necessarie]

```
if(x==8){
    y=x+2;
    x++;
}
```

Selezione doppia - `if else`

- Test di verità su condizione logica o istruzione (V/F)
- Se VERO → esegui istruzione/blocco_1
- Se FALSO → esegui istruzione/blocco_2

```
if(x==8){
    y=x+2;
}else{
    x++;
}
```

[graffe non necessarie se singola istruzione, ma super-consigliate!]

Piccolo esercizio...

- inserire due numeri interi da tastiera
- determinare il maggiore
- stampare un messaggio di comunicazione all'utente del risultato ottenuto

Selezione nidificata - `if- else if`

Selezione dentro la selezione

- Test di verità su condizione_1
  - Se VERO → esegui istruzione/blocco_1
  - Se FALSO → Test di verità su condizione_2
    - Se VERO→ esegui istruzione/blocco_2
    - *Se FALSO → esegui istruzione/blocco_3* *facoltativo!*

Selezione nidificata - `if- else if`

```
if(x==8){
    y=x;
}else if(x>8){
    y=--x;
}else{
    y=++x;
}
```

Dandling else...

Le graffe diventano fondamentali in alcuni casi

```
if(a>0) if(b>0) printf(“b positivo”); else printf(“ciao”);
```

compilato come:

```
if(a>0)
     if(b>0)
          printf(“b positivo”);
     else
          printf(“ciao”);
```

se l’else era invece da assegnare al primo if:

```
if(a>0){
     if(b>0)
          printf(“b positivo”);
}else{
          printf(“ciao”);
}
```

Selezione multipla - `switch`

- Valutazione di un'espressione e del suo valore
- L'espressione può assumere solo valori `int/char`
- Ogni possibile valore è associato a un blocco di istruzioni, che deve terminare con un `break`

- Qualora nessuna delle opzioni proposte si riveli vera, si inserisce un blocco di istruzioni di `default` (opzionale)

Selezione multipla - `switch`

```
switch(x){
    case 1:
        printf(“x vale 1”);
        break;

    case 2:
        printf(“x vale 2”);
        break;

    default:
        printf(“x non vale 1 e nemmeno 2”);
        break;
}
```

Iterazione precondizionata - `while`

- Istruzioni che devono essere ripetute/iterate
- La condizione perché l'iterazione proceda è testata **prima**
- Le istruzioni potrebbero anche non essere mai eseguite

```
while(x<9){
    x++;
}
```

[per evitare *loop infiniti*, occorre modificare il valore della variabile di controllo nel blocco di istruzioni]

Iterazione postcondizionata - do while

- Istruzioni che devono essere ripetute/iterate
- La condizione perché l'iterazione proceda è testata **dopo** *(cond. di uscita)*
- Le istruzioni sono eseguite almeno una volta

```
do{
    printf("x vale %d",x);
    x++;
}
while(x<9)
```

Cicli a conteggio - `for`

- while e do-while sono "*a iterazione indeterminata*"
- non si può sapere a priori quante volte sarà eseguito il blocco
- il numero di iterazioni non è definito esplicitamente dal programmatore

---

- Nel ciclo `for` il *numero di iterazioni è noto a priori*, come dato iniziale

Cicli a conteggio - `for`

Tre elementi fondamentali (espressioni):

- inizializzazione contatore *(variabile* `int`*)*
- condizione di test/uscita
- incremento/decremento del contatore di un valore costante (*passo del ciclo*)

```
for(int i=0; i<100; i++){
    istruzioni;
}
```

Cicli a conteggio - for

Il ciclo for e il loop infinito

```
for(;;){
  printf(“Questa frase sara’ stampata infinite volte”);
}
```

Equivalenza tra ciclo for e ciclo while

```
for(int i=0;i<10;i++){
    istruzione;
}
```

```
int i=0;
while(i<10){
    istruzione;
    i++;
}
```

Altro in laboratorio...

Altre funzioni matematiche (radici, generatori random)

Progettazione algoritmi

Applicazioni