

u3-s1-intro-c

Sistemi Operativi

Unità 3: Programmazione in C

Introduzione al linguaggio C

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Storia del C
 2. Compilazione in C
-

Caratteristiche Generali del C

Introduzione al C

Definizione del C.

Il **C** è un linguaggio di programmazione:

- *Ad alto livello*: non si scrive in istruzioni macchina
- *Imperativo*: il programma è una sequenza di istruzioni
- *Procedurale*: le istruzioni che svolgono una compito vengono raggruppate in *funzioni*, per permettere pulizia del codice e riuso

Livello C.

Tra i linguaggi di programmazione ad alto livello, il C è quello *più vicino al linguaggio macchina*.

- Libertà di utilizzo degli *indirizzi di memoria*, mediante i *puntatori*
- Utilizzato dentro *Linux* per scrivere il *kernel* e i *driver*

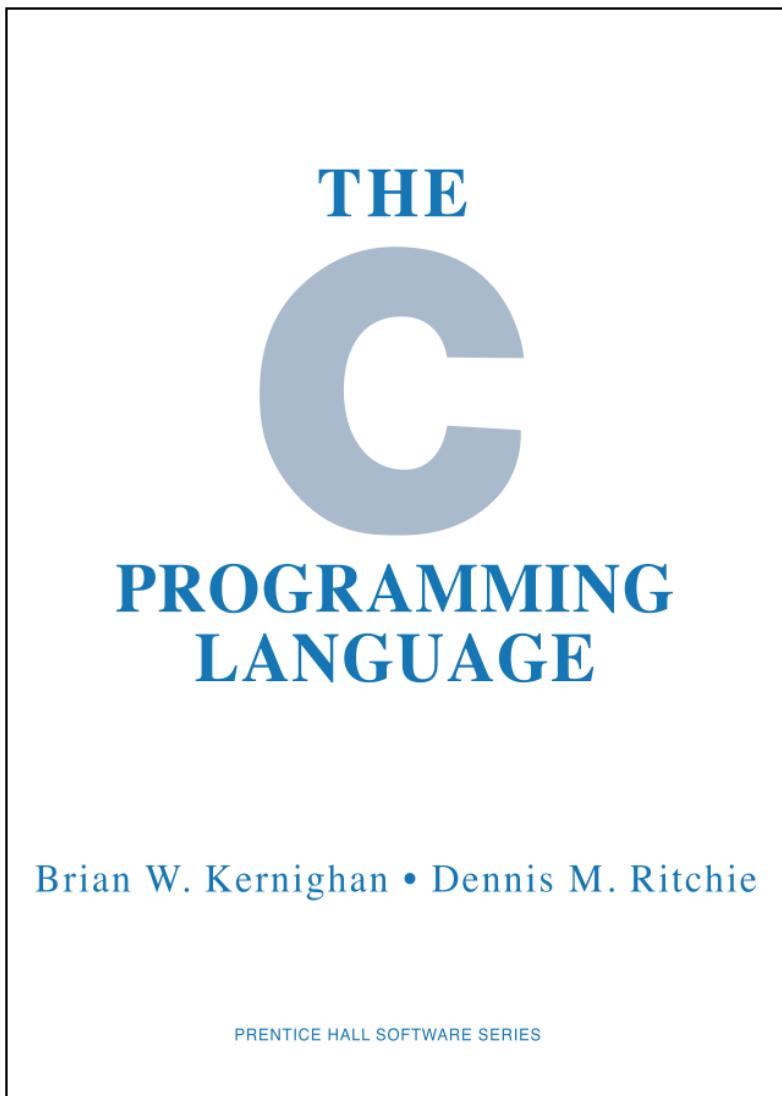
Caratteristiche del C:

- **Linguaggio minimalista**: pochi concetti semplici, vicini a quelli del linguaggio macchina
 - Molte istruzioni mappabili direttamente con una istruzione Assebly
 - Solo 32 parole riservate
 - **Ruolo centrale dei puntatori**: i puntatori sono variabili che contengono un indirizzo di memoria.
 - Permette perciò l'*indirizzamento indiretto*. Accedo a una variabile non tramite il suo nome, ma tramite il suo indirizzo.
 - Il programmatore ha un controllo molto elevato sulla memoria della macchina, consentendo di ottimizzare il codice
 - Tuttavia, questo potrebbe rappresentare una **vulnerabilità** nella sicurezza; infatti, le autorità statunitensi consigliano fortemente di **non** utilizzare il linguaggio C ([report ufficiale dettagliato](#))
 - **Tipizzazione statica**: ogni variabile ha un tipo di dato che deve essere esplicitamente dichiarato dal programmatore
-

Storia del C

Storia del C.

- Creato da Dennis Ritchie nel 1972 presso gli AT&T labs, col fine di scrivere il sistema operativo Unix
- Pubblicato nel 1978 col famoso libro *Il linguaggio C*
- Standardizzato a partire dal 1989. Standard ANSI X3.159-1989



THE C PROGRAMMING LANGUAGE

Brian W. Kernighan • Dennis M. Ritchie

PRENTICE HALL SOFTWARE SERIES

C oggi.

- Il C è in *continua evoluzione*. Si sono susseguiti vari standard negli anni.
 - Dalla prima versione **C89** (*anni '90*) ora siamo alla versione **C17**.
 - Nei prossimi anni ci sarà una nuova versione, per ora chiamata **C2x** (verosimilmente sarà **C23**)
- La standardizzazione garantisce la *portabilità* del *codice sorgente*. Uno stesso programma in C può essere compilato su *diversi SO* (Linux, Windows, MacOS).

Utilizzo del C

Applicazioni del C.

Attualmente il C è utilizzato per:

- Scrivere componenti di base di *Linux*, in particolare per i *kernel* e i *driver*
- Scrivere programmi che necessitano di *grande efficienza*
- Scrivere programmi in *domini critici*: telecomunicazioni, processi industriali, software real-time (ovvero dove c'è un *interazione a basso livello*)

Confronto del C con altri linguaggi

C vs Java:

C	Java
Compilato in codice macchina	Compilato in bytecode
Eseguito direttamente	Eseguito nella JVM
Gestione manuale di memoria	JVM gestisce la memoria
Generalmente Veloce	Lento

Compilazione in C

Linguaggi compilati e interpretati.

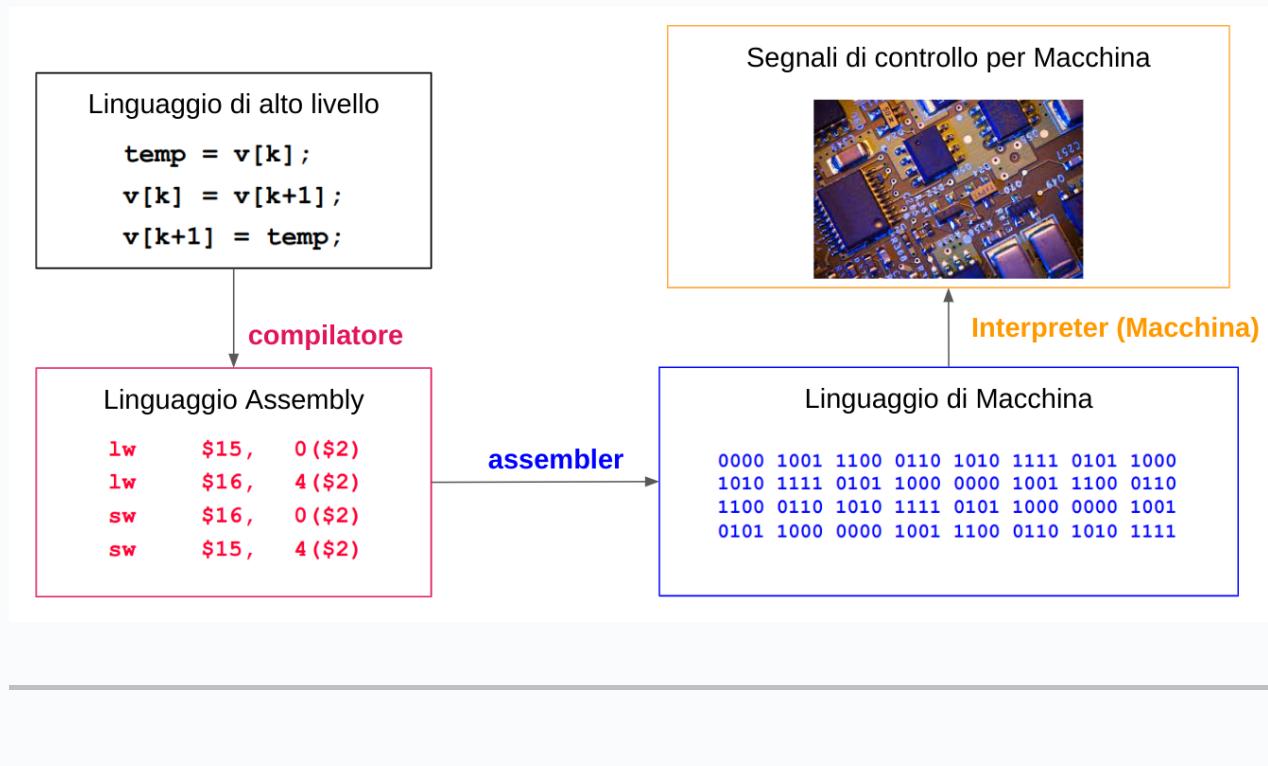
Il C è un *linguaggio compilato*.

- Un software chiamato *compilatore* traduce il codice sorgente in un eseguibile in *linguaggio macchina*
- Altri linguaggi compilati: C++, Go, Rust, ...

Il C non è un *linguaggio interpretato*

- Un linguaggio interpretato viene eseguito da un *interprete*, che legge ed esegue le istruzioni.
- Esempi di linguaggi interpretati: Python, R.

Figura: schema generale della compilazione.



Fasi della compilazione

Fasi della compilazione:

1. Il **Preprocessore** esegue eventuali sostituzioni testuali nel codice sorgente.
Necessario per costanti e macro. Osservare che qui non *entra in nessun modo* la **CPU**.
2. Il **Compilatore** crea il codice eseguibile per ogni file sorgente in C.
3. Il **Linker** assembla i codici eseguibili nel programma finale, collegando il programma alle funzioni di libreria.
 - Ogni linguaggio C fornisce varie funzioni di libreria per calcoli matematici, interazione col SO, realizzazione di interfacce grafiche.

Comandi per la Compilazione in C

Compilazione in Linux: si usa il compilatore standard **gcc**

Sintassi:

SHELL

```
gcc [<opzioni>] file1.c file2.c file3.c ... [-l librerie]
```

(Nota: "gcc" sta per "GNU C Compiler")

Normalmente vengono utilizzati come:

SHELL

```
gcc file.c      # compila e linka mettendo il codice eseguibile in a.out  
gcc file.c -c  # compila e non linka mettendo il codice oggetto in file.o  
gcc file.c -o outfile # compila e linka. codice exe in outfile  
gcc file.c -o outputfile -l libreria # compila e linka con libreria
```

Editor grafici: esistono molteplici IDE per il C. Uno semplice, snello e ben adatto a Ubuntu: **CodeBlocks**

Esempio di Compilazione in C

Primo programma in C: il seguente programma stampa a schermo la scritta **Hello World!**

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Per compilare ed eseguire, inserire il codice sorgente nel file **hello.c** ed eseguire i seguenti comandi:

SHELL

```
$ gcc hello.c -o hello
$ ./hello
Hello World!
```

Descrizione delle istruzioni dell'esempio.

- **#include <stdio.h>** Indica che usiamo la libreria standard di I/O, nella quale sono definite le principali funzioni per la gestione dell'input/output
- **int main() {** Definisce la funzione **main**, che costituisce il corpo principale di ogni programma. Deve esserci in ogni programma. Deve restituire un intero.
- **printf("Hello World!\n");** La funzione di libreria **printf** stampa a video
- **return 0;** Istruzione di ritorno dalla funzione **main**. Termina il programma. Il *valore di ritorno* del programma verso il chiamante è 0 (*no errore*)
- Le parentesi graffe **{ ... }** delimitano i *blocchi funzionali*, come ad esempio

```
int main()
{
    ... istruzioni...
}
```

- **Osservazione:** lo stesso approccio è usato per delimitare blocchi funzionali in tutti i costrutti

```
C  
if (condizione)  
{  
    ... istruzioni...  
}
```

Struttura di Programmazione in C

Struttura minima di un programma

```
C  
#include librerie  
int main(void) {  
    definizione variabili  
    istruzioni eseguibili  
}
```

Commenti:

I commenti sono testo che non viene analizzato dal compilatore
Servono per aumentare la leggibilità del codice.

Sintassi:

```
C  
/* commento  
multiriga */
```

```
C  
// commento su singola riga
```

Spaziatura: gli spazi e i ritorni a capo non hanno funzione in C

Le istruzioni che non iniziano un blocco sono terminate da ;. Si dice che la spaziatura è "zucchero sintattico".

```
C  
int main(){  
    printf("hello\\n");  
    return 0;  
}
```

equivale a

```
C  
int main(){ printf("hello\\n"); return 0;}
```

Utilizzo di librerie:

Le librerie si possono usare dopo averle menzionate con la direttiva:

```
C  
#include <libreria.h>
```

Soltamente per le *librerie di sistema* usiamo le *parentesi angolari* <>, altrimenti usiamo le apici ''.

Nota: Le istruzioni di include non vanno terminate con ; e non si possono inserire spazi a inizio riga

Librerie C

Librerie principali: ci sono in *tutti* i sistemi operativi e sono le seguenti.

- <stdio.h>: Funzioni di lettura/scrittura su terminale e su file
- <stdlib.h>: Funzioni base per gestione di memoria, processi, conversione tra tipi di dato
- <math.h>: Funzioni matematiche
- <string.h>: Funzioni di manipolazione delle stringhe
- <ctype.h>: Manipolazione di caratteri

Altre librerie:

- <complex.h>: Manipolazione di numeri complessi
- <errno.h>: Gestione dei codici di errore di funzioni di libreria
- <time.h>: Per ottenere e manipolare date e orari
- <limits.h> e <float.h>: Costanti utili per lavorare su interi e numeri reali

Libreria solo per sistemi POSIX ([Linux](#), [UNIX](#), [Mac OS](#)), quindi non standard di C

- **<unistd.h>**: API standard di POSIX. Contiene le [System Call](#), qualora volessi usarle in una maniera diretta (1)

Librerie e System Call:

- Queste librerie (*ad esclusione di <unistd.h>*) sono raccolte nella [C standard library \(clib\)](#)
- Le funzioni di libreria [NON](#) sono delle System Call
 - Utilizzano al loro interno le System Call
- La **libc** è implementata su diversi SO
 - Utilizzando System Call diverse
- Permette di compilare lo stesso codice su SO diversi

Esempio: Aprire file in C

Per aprire un file si usa la funzione della **libc** chiamata **fopen**; a seconda del [sistema operativo](#) abbiamo una logica diversa per l'apertura del file. Infatti, si dice che questa funzione di libreria rappresenta un "*layer di compatibilità*".

- Su [Linux](#) utilizza la System Call **open**
- Su [Windows](#) utilizza la System Call **CreateFileA**

u3-s2-variabili-console

Sistemi Operativi

Unità 3: Programmazione in C

Variabili e utilizzo della console

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Variabili
2. Il tipo **int**
3. Il tipo **float**
4. Gli altri tipi
5. La funzione **printf**
6. La funzione **scanf**
7. Operazioni di base

Definizione di Variabile in C

Definizione di linguaggio tipizzato.

Ricordiamo che il C è un linguaggio *tipizzato* ([link da aggiungere!!!](#))

- Ogni variabile o costante ha un *tipo*
- Il tipo è *specificato esplicitamente* dal programmatore

Tipi di Variabili.

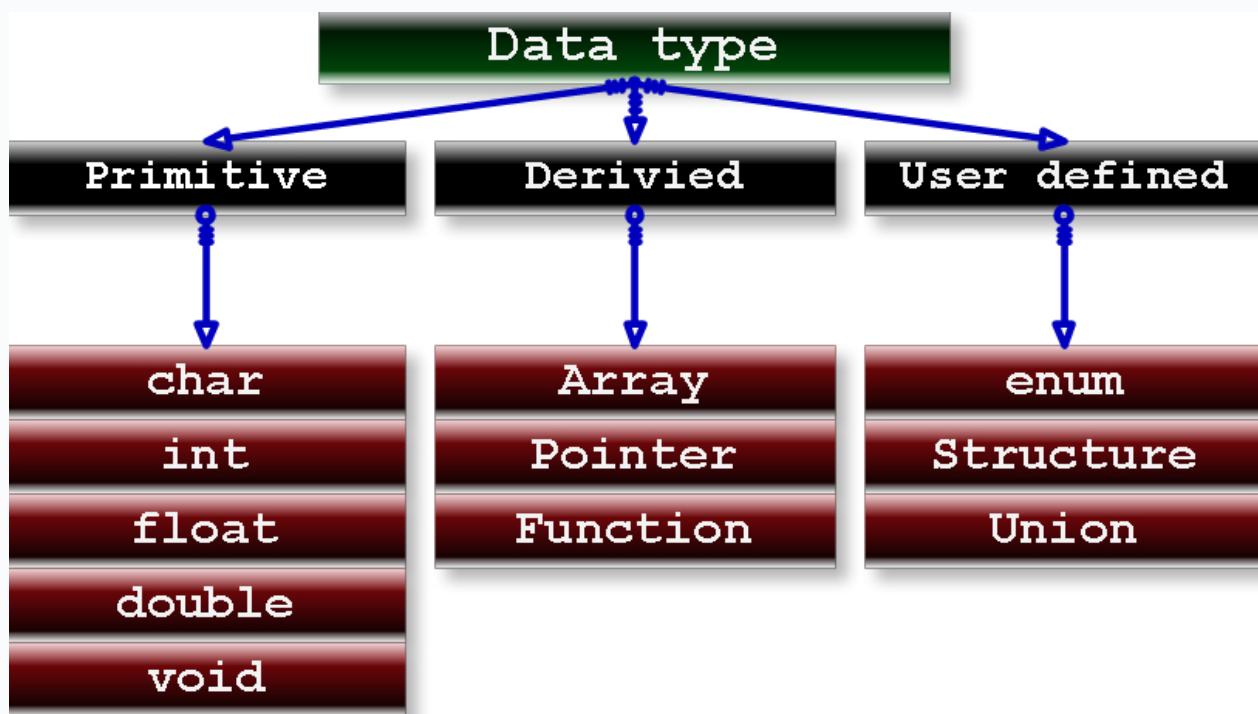
I tipi *principal*i sono:

- *Semplici*: `int`, `float`, `char`
- *Derivati*: insiemi di tipi semplici
 - Vettori (insiemi *omogenei*)
 - Struct (insiemi *non-omogenei*)
- Puntatori: contengono *indirizzi di memoria* a variabili di un certo tipo (come interi)

Nota. In C, i tipi di dato non hanno un'ampiezza standard, ma *varia da sistema a sistema*

- Ad esempio, un `int` può essere di 16, 32 o 64 bit; il linguaggio usa la *dimensione naturale* del *processore*
- Permette a ogni calcolatore di operare secondo la sua *dimensione naturale*
- Necessario fare *attenzione* in fase di scrittura del codice

FIGURA: Schema dei tipi di variabili



I Tipi Semplici

Il tipo **int**

- Rappresenta un *numero intero*.
- Rappresentato in *complemento a due su 16, 32 o 64 bit* (1).

SINTASSI

Dichiarazione:

```
int a;
```

Assegnazione:

```
a = 19;
```

Dichiarazione e Assegnazione:

```
int a = 19;
```

Il tipo **float** o **double**

- Rappresenta un *numero con la virgola*.
- Rappresentato con numero a *virgola mobile*, solitamente su 32bit (1).
- Esiste il tipo **double** che ha *precisione doppia*, solitamente su 64bit.

```
float a;
```

Assegnazione:

```
a = 3.14;
```

Altri Tipi di Base

I *tipi di base* in C sono:

- **int**: sono i numeri interi.
 - **float**: sono i numeri a virgola mobile
 - **double**: sono i numeri a virgola mobile a *precisione doppia*
 - **char**: sono le variabili che contengono *un carattere*.
 - **void**: *nessun tipo*, usato in situazioni particolari
-

Modificatori su Tipi

Possono essere usati dei *modificatori* sui tipi.

Esempio: **long int** indica un intero su più bit (ad es. 64 anziché 32).

CATEGORIA 1: LUNGHEZZA VARIABILI

- **long**: forza l'uso di un numero maggiore di bit
- **short**: forza l'uso di un numero minore di bit
 - **short int a;** indica un intero su 16 bit se di default è 32 bit.

CATEGORIA 2: SEGNO VARIABILI

- **signed**: indica che il tipo ha segno. Applicato di default
- **unsigned**: indica variabile che assume solo valori positivi

CATEGORIA 3: COSTANTI

- **const**: dichiara una costante.
 - **const float pi = 3.14**

Tipi **int** Speciali

Se si vuole avere il controllo sul *numero di bit di una variabile*, si possono usare i tipi:

- **int8_t**
- **int16_t**
- **int32_t**
- **int64_t**
- **uint8_t**
- **uint16_t**
- **uint32_t**
- **uint64_t**

A seconda dell'*architettura*, sarò *sicuro* sul numero dei bit. Questo è particolarmente

utile per scrivere *programmi molto efficienti*, o per *casi specifici* (come i *bitmask*)

Nota. Per usare questi tipi è necessario includere la libreria **#include**

<stdint.h>

Tipi Alias

Tipi di dato di sistema.

La libreria standard del C definisce dei tipi di dato *alias* (ovvero dei "soprannomi"), definiti nella Man Page **system_data_types**

- Aiutano la portabilità del codice.
- L'*alias* indica l'*obiettivo del tipo*, mentre su architetture diverse è *implementato con tipi diversi*; sono sostanzialmente dei *tipi di dati* con *scopi precisi*.

Esempi

- **size_t**: indica una lunghezza. E' solitamente **unsigned int**
- **off_t**: indica una offset di memoria. E' solitamente **int**

Ne esistono tanti: **pid_t uid_t gid_t time_t**

Operatori sui Tipi Dati

Operatore **sizeof**

L'operatore **sizeof** fornisce la *dimensione in Byte di un tipo di dato*.

- Ritorna un **size_t**

Importante! Perché la dimensione di un tipo dipende dalla macchina. Sarà utile quando useremo il comando **malloc** per allocare *dati* sullo *heap*.

Esempio: su PC 64bit

```
printf("%lu\n", sizeof(char)); // Stampa 1
printf("%lu\n", sizeof(int)); // Stampa 4
printf("%lu\n", sizeof(float)); // Stampa 4
printf("%lu", sizeof(double)); // Stampa 8
```

La funzione `printf`

Serve per stampare su *Standard Output* (1) (in genere *console*) del testo *arbitrario*.

- Per *interagire con utente*
- Per stampare il *risultato dell'elaborazione*
- Per stampare *informazioni* che sono *processate da altri programmi* tramite *pipe*

Contenuta nella libreria `stdio`.

Necessaria la direttiva:

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

Formato:

```
printf("formato", args...);
```

Il *formato* definisce il *testo da stampare*:

- *Tutti* i caratteri possono essere stampati
- Con `\n` si inserisce un ritorno a capo
- Per stampare il carattere `"` è necessario usare la sequenza di escape `\\"`
- Per stampare *valori numerici*:
 - Inserire le sequenze `%d` (per `int`) e `%f` (per `float`) nella *posizione desiderata*
 - Specificare negli `args` le variabili desiderate

Esempi:

```
printf("Hello ");
printf("World\n");
```

```
Hello World
```

C

```
int a = 14;
printf("Intero: %d\n", a);
```

Intero: 14

C

```
printf("Il numero %f ", 3.14);
printf("e' pi greco\n");
```

Il numero 3.14 e' pi greco

C

```
int a = 12;
float b = 1.1;
printf("a=%d\nb=%f\n", a, b);
```

a=12

b=1.1

La funzione **scanf**

Definizione.

La funzione **scanf** permette di **leggere** dallo **stdin** (1), tipicamente per **richiedere un input** all'utente da terminale.

Si possono leggere un **int** un **float** (o altri tipi)

Formato:

C

```
scanf("tipo", &variabile) ;
```

Tipo:

- Per leggere un **int**: **%d**.
- Per leggere un **float**: **%f**

Variabile:

Inserire una variabile di tipo **int** o **float** già dichiarate

- Preceduta dal simbolo **&**
 - Vedremo che il motivo è che la funzione **scanf** richiede un *puntatore*
 - Con **&variabile** si passa alla **scanf** l'indirizzo di **variabile**

Esempi:

Lettura di un **int**

```
C
int a;
scanf("%d", &a);
```

Lettura di un **float**

```
C
float b;
scanf("%f", &b);
```

Esempio Misto:

```
C
int a;
printf("Inserisci un numero: ");
scanf("%d", &a);
printf("Il quadrato del numero immesso è: %d\n", a*a);
```

Assegnazione

Assegnazione: si utilizza l'operatore **=**.

Esempi:

C

```
int a;
a = 12; // Assegnazione da costante
int b;
b = a; // Assegnazione da variabile
```

C

```
float f = 12; // Assegnazione assieme a dichiarazione
f = f + 12; // Assegnazione che incrementa
```

Operazioni Aritmetiche sui Numeri

Operazioni aritmetiche

- Somma: **a + b**
- Sottrazione: **a - b**
- Somma: **a * b**
- Divisione: **a / b**
 - Nota: se entrambi gli operandi sono **int** lo è anche il risultato.
- Resto della divisione: **a % b**
- Incremento: **i++**
- Decremento: **i--**

Casting

Conversione tra tipi: si chiama operazione di **casting**.

Sintassi. Il formato è: **(tipo) variabile**. Ad esempio: **(float) a**

Esempio:

C

```
int a = 5;
int b = 2;
float c;
c = a/b; // contiene 2
c = ( (float) a ) / ( (float) b ); // contiene 2.5
```

Parentesi: si possono utilizzare per annidare operazioni nella maniera desiderata.

Osservazione. Posso usare il *"casting implicito"*, ovvero la *conversione* dei tipi dati in mediante operazioni aritmetiche. Ad esempio **2+0.0** risulta un **float**.

Operazioni sui Bit

Operatori sui bit: eseguono operazioni logiche bit a bit

- **a & b**: *AND* bit a bit
- **a | b**: *OR* bit a bit
- **a ^ b**: *XOR* bit a bit
- **~a**: *NOT* bit a bit (operatore unario)

NOTA! Da non confondere con operatori logici (**&&**, **||**, **!**, che vedremo più avanti)

FIGURA: Schema degli operatori sui bit

X	Y	X&Y	X Y	X^Y	~(X)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Esercizi

Si scriva un programma che legge due interi da tastiera e stampa la loro somma.

```
#include <stdio.h>

int main(void)
{
    int a, b /* addendi */
    int c /* somma */

    /* LEGGI GLI ADDENDI A E B */
    printf("Somma due numeri\n\n");

    printf("Immetti il primo numero: ");
    scanf("%d", &a);

    printf("Immetti il secondo numero: ");
    scanf("%d", &b);

    /* CALCOLA LA SOMMA */
    c = a + b;

    /* STAMPA IL RISULTATO C */
    printf("La somma di %d + %d vale: %d\n", a, b, c);

    return 0; /* Valore di ritorno, significante no errore*/
}
```

Si scriva un programma che dato un numero di minuti, calcola a quante ore (e minuti rimanenti) equivale.

```
#include <stdio.h>

int main(void)
{
    int a; /* minuti input*/
    int b, c ; /* ore e minuti in output */
    /* LEGGI I MINUTI */
    printf("Calcolo delle ore\n\n");

    printf("Immetti il numero di minuti: ");
    scanf("%d", &a);

    /* CALCOLA LA SOMMA */
    b = a/60;
    c = a%60;

    /* STAMPA IL RISULTATO C */
    printf("Una quantità di %d minuti equivale a %d ore e %d minuti\n", a, b, c);

    return 0; /* Valore di ritorno, significante no errore*/
}
```

u3-s3-controllo-flusso

Sistemi Operativi

Unità 3: Programmazione in C

Controllo del flusso e cicli

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Controllo del flusso
 2. Cicli
-

Controllo del flusso

Definizione di Controllo del flusso

DEFINIZIONE. (*controllo di flusso*)

Un "controllo di flusso" è la possibilità di eseguire *set alternativi di istruzioni* a seconda del verificarsi di una *condizione* (ovvero una *espressione booleana* che assume solamente due valori, tradizionalmente scritti come 0 e 1).

- Alla *base* di quasi ogni programma.
 - E' necessario definire una condizione, ovvero una *espressione booleana* che può essere vera (**true**) o falsa (**false**)
 - Vedremo:
 - Operatori *di confronto*: `=`, `≠`, `<`, `>`, `<`, `>`, `<`
 - Operatori *di booleani* (logici): `&&`, `||`, `!`
-

Istruzione **if-else**

SINTASSI.

```
if ( condizione )
{
    A; // Ramo Vero
}
else
{
    B; // Ramo Falso
}
```

Concettualmente identico ai costrutti **if** in Python o Java.

Possibile avere più possibili rami.

```

if ( condizione1 )
{
    A; // Eseguito se condizione1
}
else if (condizione2)
{
    B; // Eseguito se condizione2
}
else
{
    C; // Eseguito altrimenti
}

```

Esercizio: si scriva un programma che legge da tastiera un intero e scrive se esso è positivo o negativo

```

#include <stdio.h>

int main(void)
{
    int a;
    printf("inserisci un numero: ");
    scanf("%d", &a);

    if (a>0){
        printf("%d e' positivo\n", a);
    }
    else
    {
        printf("%d e' negativo\n", a);
    }
}

```

Osservazione sull'Istruzione **if-else**

Osservazioni:

- E' possibile *omettere* il ramo **else**:

```
if ( condizione )
{
    A; // Ramo Vero
}
```

- Se un ramo è composto da una *sola istruzione*, è possibile *omettere* le {}

```
if ( condizione )
    A;
else
{
    B;
}
```

oppure

```
if ( condizione )
    A;
```

- ...
 - Questo vale per *tutti i costrutti* in C: cicli **for** e **while**, ...
 - In ogni caso è una buona idea *non* ometterli, per sicurezza

Esempio:

```
int a;
printf("inserisci un numero: ");
scanf("%d", &a);

if (a>0)
    printf("%d e' positivo\n", a);
else
    printf("%d e' negativo\n", a);
```

Errore comune: omettere le {} quando il blocco ha più di una istruzione!

Operatori di confronto

Sono equivalenti a quelli di *Java* o *Python*

- Uguaglianza: `a == b`
- Differenza: `a != b`
- Maggiore: `a > b`
- Maggiore o uguale: `a >= b`
- Minore: `a < b`
- Minore o uguale: `a <= b`

Nota. Questi valgono *solo* per i *tipi di dati esistenti*, quindi *non* stringhe!

Errore comune: confondere operatore di assegnazione `=` con quello di uguaglianza `==`

Operatori booleani

Gli operatori di confronto permettono di definire *condizioni semplici*.

Spesso è necessario *combinare condizioni semplici*.

Esempio: un valore è compreso in un intervallo?

`a>10 e a<20`

Per combinare condizioni semplici si usano gli *operatori booleani*.

- AND: `(cond1) && (cond2)`
- OR: `(cond1) || (cond2)`
- NOT: `!(cond1)`

Albero Sintattico degli Operatori Booleani.

Precedenza degli operatori nelle condizioni. *Ordine di priorità*:

- Operatori di confronto
- Operatore `!`
- Operatore `&&`
- Operatore `||`

Esempi:

```
if ( (a>10) && (b>10) )
/* Equivale a */
if ( a>10 && b>10 )
```

```
if ( !(a>10) || (b>10) )
/* Equivale a */
if ( !a>10 || b>10 )
```

Errore comune: Confondere operatore di AND booleano **&&** con l'operatore di **bitwise** AND **&**.

Annidamento di istruzioni **if else**

E' possibile annidare istruzioni **if else** per creare ramificazioni complesse.

```
if ( condizione1 )
    if ( condizione2 ){
        A;
    }
    else {
        B;
    }
else{
    C;
}
```

Esercizio

Si scriva un programma che risolve un'equazione di primo grado.

```

#include <stdio.h>
int main()
{
    float a, b ; /* coefficienti a e b */
    float x ; /* valore di x che risolve l'equazione */
    printf("Risoluzione di un'equazione di primo grado\n");
    printf("Equazione nella forma: ax + b = 0\n");
    /* LEGGI a e b */
    printf("Immetti coefficiente a: ");
    scanf("%f", &a);
    printf("Immetti coefficiente b: ");
    scanf("%f", &b);

    /* x VIENE CALCOLATO COME x=-b/a. SI DEVONO VERIFICARE I VALORI DI
    a E b */
    if( a != 0 ) {
        x = - b / a;
        printf("La soluzione e' x = %f\n", x);
    } else { /* CASO a==0 */
        if( b==0 ) {
            printf("Equazione indeterminata (ammette infinite soluzioni)\n");
        } else {
            printf("Equazione impossibile (non ammette soluzioni)\n");
        }
    }
}

```

Istruzione **switch**

Semplifica il codice in caso di *scelta in base al valore di una espressione*.

Sintassi:

```

switch (espressione)
{
    case v1:
        A;
        break;
    case v2:
        B;
        break;
    default:
        C;
}

```

Osservazioni:

- **espressione** può essere una **variabile** o un'**espressione**
- **v1**, **v2** devono essere una **costante**. Non possono essere una variabile.
- Necessario sempre delimitare ogni caso con **case** e **break**
 - *Errore comune*: dimenticare il **break**
- **default** si comporta come **else** nel costrutto **if**. E' opzionale.
- La sintassi è *molto arcarica*: questa istruzione è un relitto delle versioni vecchie del C
 - Infatti non utilizza **{}** ma **case** e **break** per *delimitare blocchi*
 - Relitto di istruzioni con **goto**.
- *Non* utilizzare se non serve strettamente.
 - Usata tipicamente per migliore performance rispetto a **if** quando ho tanti casi.

Operatore ternario

Permette di scrivere in *maniera concisa* un'espressione **if then else**.

Sintassi:

```
var = condizione ? espressione1 : espressione2;
```

Equivale a:

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

Limitazione. **espressione1** e **espressione2** possono essere solo **espressioni**, non istruzioni! L'operatore ternario può essere usato per fornire un'**espressione** in una istruzione

Esempi:

Assegnazione di valore minimo

```
c = (a < b) ? a : b; // Assegna a 'c' il minore tra 'a' e 'b'
```

Invocazione di una funzione (che è un'espressione):

```
a > b ? printf("%d\n", a) : printf("%d\n", b);
```

Cicli

Definizione di Ciclo

Definizione. (*ciclo*)

Hanno lo scopo di *ripetere in maniera controllata un blocco di istruzioni*.

Permettono di svolgere compiti ripetitivi senza duplicare il codice.

Solitamente organizzati in:

- Un blocco di istruzioni da eseguire ripetutamente.
- Una condizione che regola la fine del ciclo.

In C esistono *due tipi* di ciclo: **while** (e variante **do-while**) e **for**.

Ciclo **while**

Esegue finché *una condizione è vera*.

C

```
while ( C )
{
    A;
}
```

Comportamento:

1. Viene valutata C.
2. Se C è falsa, **salta** il blocco di istruzioni
3. Se C è vera, **esegue** il blocco di istruzioni A e **torna** al punto 1

Esempio:

C

```
int i = 1;
while ( i < 10 )
{
    printf("Numero = %d\n", i) ;
    i = i++; // Operatore di incremento
}
```

Risultato: stampa i numeri da 1 a 10 compresi.

Errore comune: sbagliare la condizione di terminazione e generare un ciclo infinito (anche se in realtà si potrebbe creare cicli infiniti di proposito).

Esercizio.

Si scriva un programma che calcoli la media di un numero di **float** specificato dall'utente.

```
#include <stdio.h>
int main()
{
    int i=0, n;
    float dato;
    float somma = 0.0;

    printf("Introduci n: "/* Leggi n */
    scanf("%d", &n) ;
    if( n>0 )
    {
        while(i < n) /* Esegui n volte */
        {
            printf("Valore %d: ", i+1);
            scanf("%f", &dato) ;
            somma = somma + dato ; /* Accumula la somma */
            i = i + 1; /* Contatore */
        }
        printf("Risultato: %f\n", somma/n) ;
    }
    else
        printf("Non ci sono dati da inserire\n");
}
```

Ciclo **for**

Definizione

Rende più facile eseguire un blocco N volte, determinato *a priori*.

- E' possibile anche col ciclo **while** ma è più prone a errori, siccome è necessario:
 1. inizializzare un *contatore*
 2. impostare la condizione del **while**
 3. *incrementare il contatore* a ogni operazione

Il ciclo **for** *sistematizza* queste operazioni

Sintassi

C

```
for ( I; C; A )
{
    B;
}
```

- **I** è l'**istruzione** di inizializzazione
- **C** è la **condizione** di terminazione
- **A** è l'**istruzione** di aggiornamento

Esempio:

C

```
int i;
for ( i=0; i<10; i=i++ )
{
    printf("Numero = %d\n", i);
}
```

equivale a:

C

```
int i;
i=0;
while ( i<10; )
{
    printf("Numero = %d\n", i);
    i++;
}
```

Osservazione.

E' semplice annidare cicli **for**, ad esempio per iterare su una tabella o matrice.

```

for( i=0; i<N; i++ )
{
    for( j=0; j<N; j++ )
    {
        printf("i=%d - j=%d\n", i, j);
    }
}

```

Esercizio: si crei un programma che determina su un numero inserito dall'utente è primo.

```

#include <stdio.h>
int main()
{
    int n, i;
    int primo=1;

    printf("inserisci il numero: ");
    scanf("%d", &n);

    for (i=2; i<n; i++)
        if(n%i==0)
            primo=0;

    if ( primo == 0)
        printf("Il numero %d non è primo\n", n);
    else
        printf("Il numero %d è primo\n", n);

}

```

Domanda: il numero 2 147 483 647 è primo? Quanto ci mette a calcolare?

Osservazioni

- Come nell'esempio precedente, se il blocco ha una sola istruzione, si possono omettere le {}
- Possibile annidare blocchi di una istruzione in più costrutti di flusso.
- Esempio:

```
for (i=2; i<n; i++)
    if(n%i==0)
        primo=0;
```

Istruzioni speciali sui Cicli

All'interno dei cicli **for** e **while** è possibile usare le seguenti istruzioni speciali.

- **break**: *termina il ciclo immediatamente*, passando alle istruzioni seguenti al ciclo.
- **continue**: *passa immediatamente alla iterazione successiva* senza eseguire le rimanenti istruzioni del blocco.

Nota. Da usare con parsimonia! Possono spesso, se inattenti, causare *cicli infiniti*.

Esercizio: quante volte viene invocata la **printf**?

```
for (i=0; i<10;i++){
    printf("Iterazione\n");
    if (i==3)
        break;
}
```

Esercizio: quante volte viene invocata la **printf**?

```
for (i=0; i<10;i++){
    if (i<3)
        continue;
    printf("Iterazione\n");
    if (i>5)
        break;
}
```

```
Dataview (inline field '='): Error:  
-- PARSING FAILED -----  
-----  
  
> 1 | =  
| ^  
  
Expected one of the following:  
  
'()', 'null', boolean, date, duration, file link, list ('[1,  
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),  
string, variable
```

```
Dataview (inline field '='): Error:  
-- PARSING FAILED -----  
-----  
  
> 1 | =  
| ^  
  
Expected one of the following:  
  
'()', 'null', boolean, date, duration, file link, list ('[1,  
2, 3]'), negated field, number, object ('{ a: 1, b: 2 }'),  
string, variable
```

u3-s4-variabili-derivate

Sistemi Operativi

Unità 3: Programmazione in C

Tipi complessi

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Vettori
 2. Le **struct**
 3. Le **union**
-

Osservazione Preliminare

Osservazione.

I tipo di dato semplici (**int** o **float**) possono contenere un solo dato alla volta.

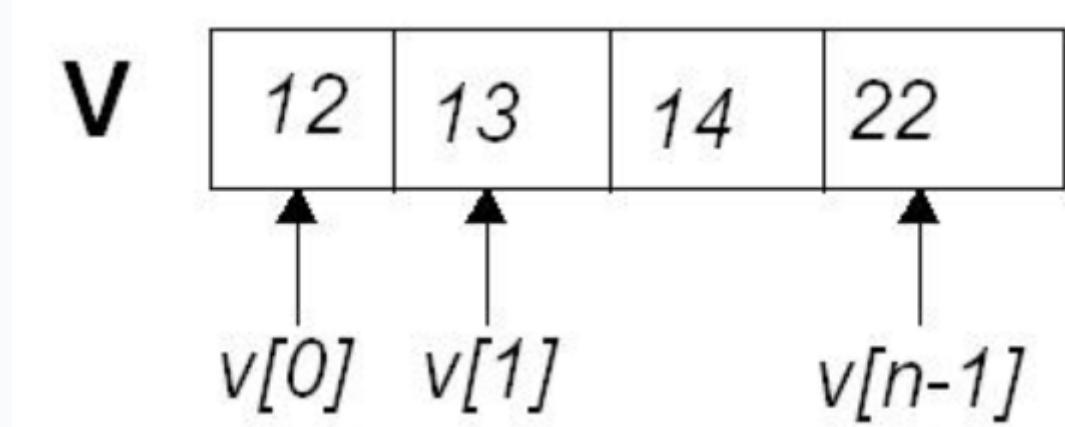
In C, si possono creare tipi di dato *complessi*, che contengono *più valori*. Noi vedremo:

- I vettori o **array**
 - Le strutture o **struct**
 - Le unioni o **union**
-

Vettori

Vettori: Definizione

Definizione: Un vettore o **array** è un insieme di variabili dello *stesso tipo*. E' composto di **N** celle, ognuna identificata da un *indice*.



Utilizzi: *vastissimi*. Permettono di trattare liste di oggetti *senza ripetere il codice*

- Effettuare operazioni matematiche: media, varianza
- Gestire flussi di dati
- Ripetere il codice è una pratica *sbagliata*; come ad esempio ho

```
int dato1, dato2, dato3, dato4, dato5 ;  
int dato6, dato7, dato8, dato9, dato10 ;  
  
scanf("%d", &dato1);  
scanf("%d", &dato2);  
scanf("%d", &dato3);  
...
```

- Questo è inutile e prone a errori!
- Una versione corretta sarebbe

```
int dato[10]; // Definizione di array  
  
for(i=0; i<10; i++)  
    scanf("%d", &dato[i]);  
  
for(i=9; i>0; i--)  
    printf("%d\n", dato[i]);
```

- Vedremo la sintassi esatta nelle prossime pagine.

Sintassi di Vettori

Definizione di un vettore:

```
tipo nome [N];
```

Esempio:

```
int vettore [10];
```

Definisce un array chiamato **vettore** composto da 10 interi (**int**).

- **Nota:** La lunghezza del vettore deve essere nota in *fase di compilazione*. Deve essere una *costante*!
- Il seguente codice è errato:

```
C
int N;
scanf("%d",&N);
float data[N];
```

- Questa è una grande *differenza* rispetto ad altri linguaggi di programmazione come Java o Python.
 - Esistono metodi per creare *array di lunghezza arbitraria* (i.c.d. "*array dinamici*") in C (la funzione **malloc**), che vedremo più avanti nel corso.

Costanti: esistono due modi in C per dichiarare delle costanti.

1. Tramite una direttiva **define**:

```
C
#define N 10
```

2. Tramite il modificatore **const** applicato a una variabile.

```
C
const int N = 10; // N non è modificabile
int dato[N];
```

Sintassi Alternativa: si può *definire ed inizializzare* allo stesso tempo un vettore.

```
C
int v[4] = {2, 7, 9, 10};
// equivalentemente
int v4[]; v[0]=2; v[1]=7; v[2]=9; v[3]=10;
```

In questo caso, si può *omettere la lunghezza*, che viene inserita *automaticamente* dal compilatore.

C

```
int v[] = {2, 7, 9, 10};
```

Operazioni sui Vettori

Indicizzazione

Accesso agli elementi: Si deve specificare l'*indice*. La sintassi è la seguente.

C

```
nomevettore[valoreindice]
```

Esempio:

C

```
int v [] = {4,5,6};
printf("%d\n", v[1]); // stampa 5
```

Indici:

- Partono da 0 e arrivano a $N - 1$
- Devono essere **int**
- Possono essere delle variabili o i risultati di una espressione

Importante! In C, *non viene controllato* che l'indice sia minore di $N - 1$. Se si accede con indici oltre i limiti, si va a leggere locazioni di *memoria arbitrarie*, che contengono *dati arbitrari*. Quindi attenti!

Esercizio: si leggano 5 interi e si stampino in ordine inverso.

```
#include <stdio.h>
#define N 5
int main (){
{
    int v[N];
    int i;

    for (i=0; i<N; i++){
        printf ("Inserisci l'elemento %d: ", i);
        scanf("%d", &v[i]);
    }
    for (i=N-1; i>0; i--)
        printf ("Elemento %d: %d\n", i, v[i]);

    return 0;
}
```

Copia di un Vettore

Copia di un vettore:

- Bisogna copiare il contenuto *elemento per elemento*
- Tra vettori della *stessa lunghezza e stesso tipo*.
- *Sbagliato* tentare di copiare usando una sola istruzione.

Corretto:

```
for (i=0; i<N; i++)
    v2[i] = v1[i];
```

Sbagliato:

```
v1 = v2;
v1[] = v2[];
```

Spiegazione.

- La variabile vettore è un *contenitore di elementi*
 - Di per se è *immutable*
 - Si possono solo *modificare* gli *elementi contenuti*
 - Approfondiremo quando vedremo i *puntatori* (infatti i vettori *sono* dei *puntatori* particolari)
-

Altri Utilizzi dei Vettori

- **Matrici:** un array di array è una *matrice*.

```
C  
int matrice [3][2];  
matrice[1][0]=12;  
float m[2][2]={{1,2},{3,4}};
```

Volendo, posso fare *array* di *array* di *array*, che è un *tensore*. Utile per le *reti neurali*.

- **Stringhe:** un array di **char** è una *stringa*.

Per definizione, in C le stringhe sono array di char, il cui ultimo elemento è 0 (o **'\0'**) , per convenzione; perché così se ne può *derivare la lunghezza*.

Molti usi e funzioni sulle stringhe. Vedremo più avanti.

```
C  
char s[4] = {'a', 'p', 'e', '\0'};
```

Esercizio sui Vettori

Esercizio: si un numero N da tastiera. Si leggano poi N interi e si stampi se essi includono duplicati.

```

#include <stdio.h>
#define MAXN 50 // Limite massimo del vettore
int main (){
{
    int v[MAXN]; // Vettore sovradimensionato
    int N, i, j;

    printf ("Si inserisca N: ");
    scanf("%d", &N); // Lunghezza effettiva del vettore

    if (N>MAXN){
        printf("N deve essere minore o uguale a %d\n", MAXN);
        return 1; // Ritorna un errore
    }

    for (i=0; i<N; i++){
        printf ("Inserisci l'elemento %d: ", i);
        scanf("%d", &v[i]);
    }

    for (i=0; i<N; i++)
        for (j=0; j<i; j++)
            if (v[i]==v[j])
                printf("L'elemento %d è duplicato dell'elemento %d\n", i,j);
    return 0;
}

```

Osservazione: si è scelto di sovradimensionare il vettore **v** rendendolo lungo **MAXN**, ma utilizzandolo fino all'elemento **N-1**.

Con la memoria dinamica che vedremo più avanti, questo work-around non sarà più necessario.

Le **struct**

Strutture: Definizione

Le **strutture** o **struct** sono collezioni che contengono variabili *non necessariamente dello stesso tipo* (ovvero possono essere di tipi diversi).

Funzionamento:

1. Si definisce la **struct**, un *nuovo tipo di dato* complesso formato da più campi
2. Si *creano* e si *usano* variabili del tipo appena creato.

Sintassi per le Strutture

1. Definizione di una **struct**:

```
struct nome {  
    campi  
};
```

Esempio:

```
struct punto{  
    float x;  
    float y;  
};  
  
struct lista  
{  
    int INFO;  
    lista* NEXT;  
}
```

2. Creazione di variabili **struct**:

Per creare nuove variabili di un tipo **struct** definito in precedenza.

```
struct nome variabile;
```

Esempio:

```
struct punto p1, p2;
```

3. Acesso ai campi **struct**:

```
variabile.campo
```

Esempio:

```
p1.x = 2.5;  
p1.y = 3.0;
```

Zucchero Sintattico per le **struct**

Utilizzo di `typedef`: per evitare di dover *premettere struct* ogniqualvolta si crea una variabile, si può usare la keyword **typedef**, con la seguente *sintassi*.

```
typedef struct {  
    campi  
} nome;
```

Esempio:

```
typedef struct{  
    float x;  
    float y;  
} punto;  
punto p1, p2; // Si può omettere struct
```

Esercizio sulle Struct

Esercizio: si crei un programma che effettua la somma vettoriale tra due vettori bidimensionali.

```
#include <stdio.h>

// La dichiarazione di una struct è solitamente fuori da ogni funzione
typedef struct{
    float x;
    float y;
} punto;

int main ()
{
    punto p1, p2, p3; // Tre variabili di tipo 'punto'

    // Lettura
    printf ("P1 → x: ");
    scanf("%f", &p1.x);
    printf ("P1 → y: ");
    scanf("%f", &p1.y);
    printf ("P2 → x: ");
    scanf("%f", &p2.x);
    printf ("P2 → y: ");
    scanf("%f", &p2.y);

    // Somma
    p3.x = p1.x + p2.x;
    p3.y = p1.y + p2.y;

    printf("La somma vettoriale è il punto: (%f, %f)\n", p3.x, p3.y);

}
```

Le **struct**: Osservazioni

Osservazioni

1. Non è possibile confrontare due **struct** con gli operatori **==** o **≠**. Infatti è necessario confrontare tutti i campi

```
C  
punto p1, p2;  
if (p1==p2) // Sbagliato! (Anche se in certi casi potrebbe funzionare, ma non è garantito)  
...  
if (p1.x==p2.x && p1.y == p2.y) // Corretto  
...
```

Inizializzazione: si può fare come coi vettori; ovvero *manualmente*

```
C  
punto p1 = {1.1, 2.4} // x=1.1 e y = 2.4
```

Usi delle Strutture

Le **struct** sono molto usate per creare *nuovi tipo di dato complesso*:

- Sono come record di un database
- *Esempi.* Numero complesso, indirizzo stradale, ecc...

Sono molto usate nelle *librerie del C*.

- Permettono di creare tipi di dato arbitrari
- Per rappresentare strutture del sistema operativo.
- *Esempi.* Variabili di sincronizzazione, pacchetti di rete, ecc...

Le **union**

Le Unioni: Definizione

Definizione.

Una **union** o *unione* è una variabile che può contenere *in momenti diversi* oggetti di tipo (e dimensione) *diversi*, con, *in comune*, il ruolo all'interno del programma.

- Le **union** servono per *risparmiare memoria*; si usa un'unica *cella di memoria*.
- Usate particolarmente in sistemi *embedded* con stretti vincoli di risorse
- Le **union** servono anche per avere un *tipo di dato generico* che ha tipo diverso a seconda della circostanza, quindi per *interpretare* zone di memorie *diversamente*.

Implementazione.

Occupava tanta memoria quanto il **campo più grande**, visto che esse non possono mai essere utilizzate contemporaneamente (la scelta di una esclude automaticamente le altre)

- I campi **condividono** il medesimo spazio di memoria.
- Se si cambia il valore a un campo, il valore di tutti gli altri campi viene sovrascritto

Sintassi delle Unioni

In C una unione viene definita tramite la **parola chiave union**

- La **definizione** di un'unione è molto **simile** a quella di una **struct**, ed è **medesimo** il **modo di accedervi**
- Si può usare **typedef** per evitare di dover premettere **union** ogniqualvolta si crea una variabile

Esempio.

```
C

union student
{
    uint64_t tessera_sanitaria;
    uint32_t matricola;
}; // Nota: Qui si prende automaticamente una cella di memoria da 64bit (8 byte)
```

- Si dichiarano e utilizzano come le **struct**

C

```
#include <stdio.h>
#include <stdint.h> /* Per uint32_t e uint64_t */
union student
{
    uint64_t tessera_sanitaria;
    uint32_t matricola;
};

int main( int argc, char *argv[] ){
    union student luca;
    luca.matricola = 1234;
    printf("Matricola: %d\n", luca.matricola);

    luca.tessera_sanitaria = 2897189786;
    /* Usare il formato %ld essendo in intero lungo */
    printf("Tessera Sanitaria: %ld\n", luca.tessera_sanitaria);
    return 0;
}
```

Nota. L'assegnazione di un campo **sovrascrive** il valore gli altri campi, che non potranno essere più letti.

C

```
union student luca;
luca.matricola = 1234;

// Sovrascrivo luca.matricola
luca.tessera_sanitaria = 2897189786;

// Leggo correttamente tessera_sanitaria
printf("Tessera Sanitaria: %d\n", luca.tessera_sanitaria);

// Leggo matricola, che è stata sovrascritta!
printf("Matricola di Luca: %d\n", luca.matricola); // Errore!
```

Casi d'uso delle Unioni

Si usano quando serve *dichiarare variabili* che possono assumere *tipo diverso a seconda delle circostanze*.

Esempio: Uno studente può essere identificato col *numero di tessera sanitaria* o con la *matricola* a seconda della situazione

```
C  
union student  
{  
    uint64_t tessera_sanitaria;  
    uint32_t matricola;  
};
```

La **union student** *non* può contenere *contemporaneamente* tessera sanitaria e matricola

- Il programma deve essere scritto di conseguenza

Rappresentazione delle Unioni nella Memoria

Come detto, una **union** occupa lo *spazio necessario al campo più grande*.

FIGURA: Schema

```
Dataview (inline field '='): Error:  
-- PARSING FAILED -----  
-----  
  
> 1 | =  
| ^  
  
Expected one of the following:  
  
'(', 'null', boolean, date, duration, file link, list ('[1,  
2, 3)'), negated field, number, object ('{ a: 1, b: 2 }'),  
string, variable
```

u3-s5-stringhe

Sistemi Operativi

Unità 3: Programmazione in C

Stringhe

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Il tipo **char**
 2. Stringhe
 3. Funzioni sulle stringhe
 4. Conversione tra stringhe e altri tipi
-

Il tipo **char**

Carattere e Stringa: Definizione

Recap.

Per ora abbiamo visto i tipi di dato:

- Intero: **int**
- Reale: **float**
- Vettori: **[]**
- Strutture: **struct**

Definizione di Carattere e Stringa.

Esiste il tipo **char** che rappresenta un *singolo carattere*.

Un *vettore di caratteri* è una *stringa*.

```
char s [10]; // Stringa di lunghezza 10
```

La convenzione ASCII

Il tipo **char** rappresenta un singolo carattere come un *numero*, codificato mediante la convenzione *ASCII*.

- Come numero tra **0 e 127**
- Ogni numero rappresenta un possibile carattere
- **Non** ci sono caratteri *speciali, accentati o simili*
 - Lo standard per trattarli si chiama *Unicode*, non lo vedremo

FIGURA: Tabella ASCII

| Dec Chr |
|---------|---------|---------|---------|---------|
| 0 NUL | 26 SUB | 52 4 | 78 N | 104 h |
| 1 SOH | 27 ESC | 53 5 | 79 O | 105 i |
| 2 STX | 28 FS | 54 6 | 80 P | 106 j |
| 3 ETX | 29 GS | 55 7 | 81 Q | 107 k |
| 4 EOT | 30 RS | 56 8 | 82 R | 108 l |
| 5 ENQ | 31 US | 57 9 | 83 S | 109 m |
| 6 ACK | 32 : | 58 : | 84 T | 110 n |
| 7 BEL | 33 ! | 59 ; | 85 U | 111 o |
| 8 BS | 34 " | 60 < | 86 V | 112 p |
| 9 HT | 35 # | 61 = | 87 W | 113 q |
| 10 LF | 36 \$ | 62 > | 88 X | 114 r |
| 11 VT | 37 % | 63 ? | 89 Y | 115 s |
| 12 FF | 38 & | 64 @ | 90 Z | 116 t |
| 13 CR | 39 ' | 65 A | 91 [| 117 u |
| 14 SO | 40 (| 66 B | 92 \ | 118 v |
| 15 SI | 41) | 67 C | 93] | 119 w |
| 16 DLE | 42 * | 68 D | 94 ^ | 120 x |
| 17 DC1 | 43 + | 69 E | 95 _ | 121 y |
| 18 DC2 | 44 , | 70 F | 96 ` | 122 z |
| 19 DC3 | 45 - | 71 G | 97 a | 123 { |
| 20 DC4 | 46 . | 72 H | 98 b | 124 |
| 21 NAK | 47 / | 73 I | 99 c | 125 } |
| 22 SYN | 48 0 | 74 J | 100 d | 126 ~ |
| 23 ETB | 49 1 | 75 K | 101 e | 127 DEL |
| 24 CAN | 50 2 | 76 L | 102 f | |
| 25 EM | 51 3 | 77 M | 103 g | |

Sintassi per i Caratteri

Implementazione dell'ASCII.

Sono *sufficienti* 7bit per rappresentare *un carattere ASCII*; in C, ogni carattere occupa comunque 1B=8bit

Esempio:

Stringa: **ciao** è rappresentata come 4Byte:

```
99 105 97 111  
c i a o
```

Nota: non confondere numeri e caratteri

int a = 5; La variabile **a** contiene **5**

char c = '5'; La variabile **c** contiene **53**

Sintassi.

In C, un carattere si rappresenta con una *variabile* di tipo **char**.

Un **char** è molto simile a un **int** che occupa solo *1B di memoria*.

Rappresenta allo stesso tempo un carattere oppure un numero da 0 a 255.

```
char c;  
c = '5';  
c = 53; Equivalente!
```

Nota: necessario usare (solo) gli apici singoli **'**; gli apici doppi **"** racchiudono invece le *stringhe*, ovvero *vettori* di *caratteri*.

Sequenze di escape.

- Il carattere **** serve per introdurre un carattere speciale-
 - Ad esempio **\n** rappresenta il carattere di ritorno a capo
- **\n** è un singolo carattere
- Per rappresentare il carattere **** si usa la sequenza ****
- *ASCII* contiene alcuni caratteri non stampabili, detti di *speciali*. Per rappresentarli su C utilizziamo le seguenti sequenze.
 - 7 - BEL - **\a**: emetti un bip dall'altoparlante
 - 8 - BS - **\b**: cancella l'ultimo carattere
 - 9 - TAB - **\t**: tabulazione (spazio lungo)
 - 10 - LF - **\n**: avanza di una riga (*nota: questa funziona per Linux*)
 - 13 - CR - **\r**: torna alla prima colonna (*nota: funziona solo nel terminale*)

Esempio:

```
C  
char c = '\n'; //Contiene un ritorno a capo
```

Operazioni sui Caratteri

1. Stampare un carattere: 2 funzioni possibili

- Tramite **printf**:

```
C  
char c = 'a';  
printf("%c", c); // stampa: a
```

- Tramite **putchar**:

```
C  
char c = 'a';  
putchar(c); // stampa: a
```

2. Lettura di un carattere

- Tramite **scanf**:

```
C  
char c;  
scanf("%c", &c); // Legge da tastiera e mette in c
```

- Tramite **getchar**:

```
C  
char c;  
ch = getchar(); // Stesso comportamento
```

Nota: è complicato leggere un **solo** carattere. Bisogna gestire il carattere di **Invio**, che anch'esso è letto dalla **getchar**

Esempio di Operazioni sui Caratteri

Esempio: Stampare tutte le lettere maiuscole e minuscole

```
char ch;

// Maiuscole
for( ch = 'A' ; ch <= 'Z' ; ch++)
    putchar(ch);

// Minuscole
for( ch = 'a' ; ch <= 'z' ; ch++)
    putchar(ch);

putchar("\n");
```

Stringhe

Stringhe: Definizione

Definizione di Stringa.

Una *stringa* è una sequenza di caratteri.

Implementazione.

In C, si rappresenta tramite un *vettore di caratteri*.

Esistono una serie di *funzioni di libreria* per processare facilmente le *stringhe*

- Si possono anche manipolare *a mano* dei vettori di caratteri.
- Ma è *più veloce e sicuro usare le funzioni di libreria*.

Lunghezza di Stringhe

Lunghezza di una stringa: ogni funzione che processa stringhe deve conoscere il vettore su cui opera e la sua lunghezza. Questo rappresenta un *limite* per le eventuali funzioni che devono *manipolare stringhe*, dato che devono sapere tale lunghezza *a priori*.

Convenzione: Null-terminated string.

In C, per facilitare le operazioni si usano le *Null-terminated string*

- A una stringa si aggiunge sempre un **carattere terminatore**, di solito è il carattere **\0** (dopo vedremo perché).
- Quando la stringa viene processata, il carattere terminatore **indica che la stringa è finita**
- Non è necessario indicare anche la **lunghezza**
- Ogni stringa è lunga **un carattere in più**

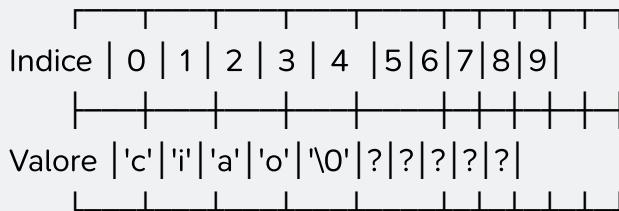
Il terminatore.

Il terminatore deve essere

- Un carattere **speciale non stampabile**, per non generare ambiguità
- Essere **ASCII** e rientrare in un **char** quindi compreso tra 0 e 255.
- Si utilizza per **convenzione** il carattere **\0** che corrisponde al numero 0

Errore comune: se si crea una stringa senza il terminatore, le funzioni di libreria hanno **comportamenti inaspettati** (di solito pericolosi!)

Esempio: si rappresenti in un vettore di lunghezza 10 la stringa **ciao**.



Non è importante il valore delle ultime 5 posizioni, **non verrà mai usato**.

Implementazione delle Stringhe

Definizione di stringhe: per definire in C una stringa ho vari modi.

1. Definendo un vettore di caratteri (**manuale**):

```
C
char s[] = {'c', 'i', 'a', 'o', '\0'}; // Il terminatore è messo dal programmatore
```

2. Usando le virgolette doppie **" "** per indicare una stringa (**automatica**):

```
C
char s[] = "ciao"; // Il terminatore è messo in automatico dal compilatore
```

Lettura di Stringhe

Principalmente ci sono *tre funzioni* per leggere stringhe: **scanf**, **gets**, **gets_s** o **fgets**

1. **Lettura di stringhe da `stdin`**: si usa la **scanf** con lo specificatore di formato **%s**.

- L'argomento deve essere un vettore di caratteri
- Non si usa l'operatore **&** (dato che ho già un *puntatore*)
 - L'operatore **&** si utilizza per passare come argomento l'indirizzo di una variabile
 - In C, passare come argomento un vettore già significa passarne l'indirizzo
- Legge fino al *primo spazio* o *ritorno a capo*.
- Termina la stringa letta col terminatore **'\0'**

Esempio: leggi una stringa

```
C  
char s[20];  
printf("Inserisci il tuo nome: ");  
scanf("%s", s); // Senza &
```

Importantissimo: se la stringa letta è più lunga di 19 caratteri, la **scanf** va a scrivere in zone di memoria arbitrarie.

Fonte di molte *vulnerabilità* software!

2. **Lettura di stringhe da `stdin`**: esiste anche la funzione **gets** che legge una stringa *fino al ritorno a capo*.

```
C  
char s[20] ;  
printf("Inserisci il tuo nome: ");  
gets(s) ;
```

Nota: ha lo stesso problema della **scanf**. Può andare a scrivere fuori dal vettore. Rimossa a partire da C11.

3. **Lettura di stringhe da `stdin`**: per scrivere un programma sicuro, utilizzare la funzione **gets_s(vettore, N)** che *non scrive più di N* caratteri su **vettore** (*compreso terminatore*)

```
char s[20];
printf("Inserisci il tuo nome: ");
gets_s(s, 20);
```

Nota: in Ubuntu, `gets_s` non è ancora implementata.

Tuttavia `gets_s(s,N)` equivale a `fgets(s,N,stdin)`.

Scrittura di Stringhe

Scrittura di stringhe su `stdout`: si usa la `printf` con lo specificatore di formato `%s`.

- L'argomento deve essere un *vettore di caratteri*
- Deve essere terminato da `'\0'`, altrimenti vengono stampati *caratteri casuali* finché non si incontra un `'\0'`

Esempio:

```
char nome [] = "Martino";
printf("Il mio nome: %s\n", nome);
```

Manipolazione delle Stringhe

Libreria Standard per le Stringhe

Le funzioni comuni su stringhe sono implementate nella *libreria standard del C*.

Necessario includere:

`#include <string.h>`

Permette di *non re-implementare* funzioni come *calcolo della lunghezza, copia, duplicazione, concatenazione*, eccetera... (quindi non scoprire l'acqua calda).

Lunghezza

Lunghezza: Si usa la funzione **strlen(s)**. Conta i caratteri finché trova il terminatore, *terminatore escluso*.

Esempio:

```
C  
char s [50];  
int l;  
printf("Inserisci una stringa: ");  
gets_s(s, 50);  
l = strlen(s);  
printf("La stringa e' lunga: %d\n", l);
```

Viene stampata la *lunghezza effettiva* della stringa immessa.

Copia di Stringhe

Copia di stringhe: si usa la funzione **strcpy(dst,src)**

Esempio:

```
C  
char s1[]="ciao";  
char s2[10];  
strcpy(s2,s1)
```

La stringa **s2** conterrà **ciao**, terminata da **'\0'**.

La **strcpy** copia *carattere per carattere*. Infatti, come abbiamo visto *non si può assegnare un vettore a un altro vettore* (**LINK DA FARE**).

```
C  
s2 = s1; // Sbagliato!
```

Concatenazione (o somma) delle Stringhe

Concatenazione: si usa la funzione **strcat(dst,src)**

Concatena **dst** e **src** e scrive tutto in **dst**

Nota! Il vettore **dst** deve essere sufficientemente lungo per contenere *la stringa risultante!*

Esempio sbagliato:

```
char s1[]="ciao";
char s2[]=" mondo";
strcat(s1, s2); // Errore! s1 è lunga 5
```

Esempio corretto:

```
char s1[15]="ciao";
char s2[]=" mondo";
strcat(s1, s2); // Corretto! s1 è lunga 15 > 4+6+1
```

Confronto tra le Stringhe

Confronto: si usa la **strcmp(a,b)** che *confronta le due stringhe carattere per carattere* e fornisce l'*ordinamento alfabetico*.

Essa ritorna un *valore numerico*, definito come:

- 0 se le stringhe sono uguali
- < 0 se **a** precede **b** in ordine alfabetico
- > 0 se **b** precede **a** in ordine alfabetico

Esempio:

```

char s1="ciao";
char s2={'c','i','a','o','\0'};
char s3="mondo";

strcmp(s1, s2); // ritorna 0
strcmp(s1, s3); // ritorna un numero <0
strcmp(s3, s1); // ritorna un numero >0

if (strcmp (s1, "ciao") ){ // Comparazione con costante
    ...
}

```

Funzioni Miste

Altre funzioni:

- *Ricerca di sotto stringhe*: **strchr** e **strstr**, **strspn**, **strcspn**
- Operazioni *su caratteri*: in **<ctype.h>** e non in **<string.h>**!
 - *Classificazione* di caratteri: **isalpha**, **isdigit**, **isupper**, **islower**
 - *Conversione* tra caratteri: **toupper**, **tolower**

Versioni sicure: le funzioni viste finora, hanno comportamenti imprevedibili se le stringhe fornite non sono terminate da **'\0'**.

- Ne esistono versioni *sicure*, in cui si forniscono la lunghezza del vettori coinvolti, per evitare di andare a *leggere o scrivere* oltre.
 - **strncpy(dst, src, n)**: come **strcpy**
 - **strncat(dst, src, n)**: come **strcat**
 - **strncmp(s1, s2, n)**: come **strcmp**
- Una buona norma è *utilizzare* queste versioni per pararsi da *stringhe malformate* di proposito.

Esercizio sulle Stringhe

Esercizio: si acquisisca una stringa da tastiera e si verifichi se è palindroma

```
#include <stdio.h>
#include <string.h>
#define MAXN 100

int main ()
{
    char s[MAXN];
    int len, i;
    printf("Inserisci una parola: ");
    scanf("%s", s);

    len=strlen(s);
    for (i=0; i<len; i++)
        if (s[i] ≠ s[len-1-i]){
            printf("Parola '%s' NON palindroma\n", s);
            return 0;
        }

    printf("Parola '%s' palindroma\n", s);
    return 0;
}
```

Conversione tra stringhe e altri tipi

Obiettivo

Esistono *funzioni* per *convertire una stringa* in un *numero intero o con virgola*.

Esempio:

- Stringa: "123" convertibile in **int** 123
- Stringa: "3.14" convertibile in **float** 3.14

Conversione Stringa → Numeri

Prime Funzioni.

- **n = atoi(s)**: converte stringa in **int**
- **f = atof(s)**: converte stringa in **float**

Nota! la stringa deve avere il **terminatore**. Non c'è controllo di errori: `atoi("ciao")` ritorna 0.

Funzione `sscanf`.

Si può usare la funzione **sscanf**. *Equivalente* alla funzione **scanf** ma ottiene i caratteri da **una stringa** e non da **stdin** (quindi si ha una situazione del tipo **stringa**→**sscanf**→**scanf**→**indirizzo variabile**). Sintassi: **sscanf(stringa, formato, argomenti)**

Esempio:

```
C  
char s[] = "314";  
int i;  
sscanf(s, "%d", &i);
```

Conversione Numeri → Stringa

Funzione `sprintf`.

Per convertire da **float** o **int** a stringa, si usa la funzione **sprintf(buffer, formato, argomenti)**, concettualmente *identica* alla **printf**, con la differenza che il risultato è salvato in **buffer** (e non stampato su **stdout**); similmente a **sscanf** si ha una situazione del tipo **numero**→**sprintf**→**printf**→**buffer**

Esempio:

```
C  
char s[100];  
int n = 425;  
sprintf(s, "%d", n); // s conterrà la stringa "425", terminata da '\0'
```

Esercizio

Esercizio: si acquisisca una riga da tastiera e si trasformi in **title case**.

Una stringa in **title case** ha le iniziali (e solo le iniziali) di ogni parola maiuscole.

Esempio: **Nel Mezzo Del Cammin Di Nostra Vita**

```

#include <stdio.h>
#include <string.h>
#include <ctype.h> // Necessario per isalpha e toupper

#define MAXN 100

int main ()
{
    char s[MAXN];
    int len, i;
    printf("Inserisci una frase: ");
    /* Notare che istruiamo fgets per leggere da standard input */
    fgets(s, MAXN, stdin);

    len=strlen(s);
    for (i=0; i<len; i++)
        /* Osservare la condizione seguente. Il secondo non genera mai
           errore perché eseguito solo se il primo è falso */
        if (i==0 || !isalpha(s[i-1])) {
            /* toupper semplicemente non ha
               effetto su numeri */
            s[i] = toupper(s[i]);
        }

    /* Non è necessario stampare '\n'. Con la fgets è incluso nella stringa */
    printf("Title Case: %s", s);
    return 0;
}

```

u3-s6-funzioni

Sistemi Operativi

Unità 3: Programmazione in C

Le funzioni

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Le funzioni
 2. La funzione **main**
-

Le funzioni

Le funzioni: Definizione

Definizione. (*funzione*)

Una *funzione* è un *insieme di istruzioni* che svolge un *completo compito*. Principalmente ha i seguenti scopi.

- Per rendere il *codice ordinato*
- Permettere il riuso del codice
- Avere codice *generico*

Quindi una funzione delimita un frammento di *codice riutilizzabile*.

- Può ricevere dei argomenti in ingresso
- Può fornire un valore di ritorno

Dopo essere definita la funzione viene *invocata*, ovvero utilizzata.

Il *copia e incolla* è da evitare!

- Disordinato
- Diventa difficile correggere errori

Esempio. (*funzione principale main*)

Il **main** è una funzione. Viene invocata dal SO quando viene *avviato il programma*.

- Riceve degli *argomenti* (non sempre, vedremo)
 - Ritorna un **int**
-

Le funzioni: Sintassi

Definizione di una funzione in C:

C

```
tipoDiRitorno nome (argomenti){
    ...
    istruzioni
    ...
    return valoreDiRitorno; // Opzionale
}
```

Esempio:

C

```
int somma (int a, int b){
    return a+b;
}
```

Argomenti di una Funzione

1. Argomenti di una funzione: Caso generale

Specificano i dati sui quali la funzione deve lavorare

- Rendono la funzione *generica*
 - Ma una funzione può non ricevere argomenti
- La funzione non deve operare *unicamente* sugli argomenti

Sintassi:

C

```
tipoDiRitorno nome (tipo nome, tipo nome, ...) {...}
```

Esempio:

C

```
float radice ( float numero ){...}
```

2. Caso Particolare: Assenza di Argomenti

Se la funzione non riceve argomenti, si indica **void**.

Esempio:

```
C  
int pigreco(void){  
    return 3.14;  
}
```

Altre funzioni che non richiedono parametri:

- Dimmi l'ora corrente: **int time(void)**

3. Caso Particolare: Argomenti Arbitrari:

Se non si indica niente come argomento **()**, quindi *non* l'opzione **void**, la funzione può ricevere un *numero arbitrario* di argomenti.

- Sistema utilizzato per funzioni come **printf** o **scanf**.
- Difficile creare funzioni con *numero variabile di argomenti*.
 - *Non ce ne occuperemo*

Esempio.

```
C  
void stampa(){  
    printf("ciao!\n");  
}  
  
stampa();  
stampa(2, 3); // Corretto, parametri ignorati
```

Valore di Ritorno di una Funzione

Valore di ritorno: Definizione

Specifica il tipo di dato ritornato dalla funzione come risultato

Se non deve ritornare un risultato, si indica **void**

Esempio:

```
C  
void stampaCiao(void){  
    printf("ciao\n");  
}
```

C

```
int somma(int a, int b){ return a+b;} // Ritorna un intero
```

Valore di ritorno:

L'istruzione **return** termina istantaneamente la funzione.

- Specifica il valore di ritorno (*se previsto*)
- *Non è necessaria* se la funzione non ha valore di ritorno (ritorna **void**)

C

```
int somma(int a, int b){
    return a+b; // Necessario ritornare un intero
}
```

C

```
void stampaCiao(void){
    printf("ciao\n");
    return; // Può essere omesso
}
```

Valore di ritorno: Uso Particolare

L'istruzione **return** può sempre essere usata per far terminare la funzione prima della fine delle istruzioni.

C

```
void stampaCiao(void){
    printf("ciao\n");      // Sempre eseguito
    return;
    printf("Mai Eseguito!\n"); // Non viene eseguito
}
```

```

int radice(int n){
    if (n<0)      // In caso di errore
        return -1; // Termina e ritorna -1
    ... calcolo...
    return r;     // Ritorna la radice se tutto ok
}

```

Le funzioni

Importanza della definizione:

La prima riga di una funzione definisce *il valore di ritorno e gli argomenti*.

Fondamentale per capire *input* e *output* della stessa.

Essa è utilizzata *per documentare* il codice

- Solo le istruzioni non sono incluse nella definizione
- Non interessa nella documentazione

Nota: in un codice C, le funzioni devono *prima* essere definite nel codice. Più avanti nel codice altre funzioni possono invocarle

Esempio:

int strlen(const char *s) calculate the length of a string

La funzione main

La funzione main: Definizione

La funzione **main** viene eseguita dal SO quando il *processo* viene avviato.

- Ovvero quando il programma *viene messo in esecuzione*

Riceve come argomenti i *parametri delle linea di comando*.

- Ovvero il testo scritto in coda al nome del programma quando lanciato

```
./myprog arg1 arg2 ...
```

Fornisce un **int** come valore di ritorno, detto *exit code*.

- Canale di comunicazione programma-SO per comunicare *errori di esecuzione*; se non ho errori ritorno 0, altrimenti ritorno un altro numero.

La funzione **main**: Sintassi

Definizione:

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

Argomenti:

- **int argc**: *numero di parametri* sulla riga di comando.
 - In assenza di parametri vale 1.
 - *Incrementato per ogni parametro*.
- **char* argv[]**: *vettore dei parametri*.
 - E' un *vettore di puntatori a carattere* (ovvero *vettore di stringhe*)
 - Ogni puntatore a carattere del vettore è un *argomento in forma di una stringa*
 - **argv[0]** è sempre il *nome del programma*. I parametri effettivi iniziano da **argv[1]**

Esempio:

```
C  
.myprog ciao mondo
```

argc vale 3

argv vale ".myprog", "ciao", "mondo"

```
C  
.myprog
```

argc vale 1

argv vale ".myprog"

Esempio: Stampa di **argc** e **argv**

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    printf("argc = %d\n", argc);
    for(i=0; i<argc; i++)
        printf("argv[%d] = \"%s\"\n", i, argv[i]);
    return 0;
}
```

Esecuzione:

SHELL

```
./sample
argc = 1
argv[0] = "./sample"
```

SHELL

```
./sample ciao mondo
argc = 3
argv[0] = "./sample"
argv[1] = "ciao"
argv[2] = "mondo"
```

La funzione **main**: Osservazioni

Osservazioni:

argv contiene un vettore di stringhe. Se essi devono essere interpretati *come numeri*, vanno convertiti tramite funzioni come **atoi**, **atof** o **sscanf**.

Se un programma *non ha necessità* di ricevere dei parametri, può definire il **main** senza argomenti.

```
int main(){}
int main(void){}
```

La funzione **main**: Valore di Ritorno

Valore di ritorno:

il **main** può restituire un **int** che viene esaminato dal SO.

Esso indica se c'è stato un errore nell'esecuzione. Per convenzione.

- 0 indica che non c'è stato errore
- Un numero diverso da 0 indica un errore.
 - Il significato del numero, è *specifico del programma*

Questo sistema viene molto utilizzato:

- In *script Bash* che necessitano di sapere se i programmi eseguiti hanno avuto errore
- Per i *moduli del SO*, che devono avviare servizi, demoni e programmi in background.

Esempio: si scriva un programma che accetta un solo parametro e lo stampa.

```
#include <stdio.h>
int main(int argc, char *argv[]){
    /* Con un parametro, argc=2,
       siccome il primo elemento è il nome del programma */
    if (argc!=2){
        printf("Numero di parametri errato\n");
        return 1; /* Il programma termina in questo punto */
    }

    printf("%s\n", argv[1]);
    return 0;
}
```

Lettura del valore di ritorno in Bash:

Per ottenere il valore di ritorno di un programma all'interno di uno script Bash si usa la variabile **\$?**.

Contiene il *valore di ritorno dell'ultimo programma lanciato*

```
./myprog ciao mondo
code=$? //Necessario salvarlo, altrimenti sovrascritto dopo echo
echo "MyProg ha fornito come codice di ritorno $code"
if (( $code == 0 )) ; then
    echo "Nessun errore"
else
    echo "Errore"
fi
```

Nota: il valore di `$?` viene scritto dopo ogni comando, anche dopo `echo`, dato che è un *programma!*

Osservazione: si può dichiarare la funzione `main` perché non ritorni nulla. Il programma funziona ma non è corretto. Il SO riceve un *valore di ritorno casuale*.

```
void main (int argc, char * argv[]){...}
```

Sarebbe da evitare.

Esercizio

Esercizio: si scriva un programma che riceve due interi come parametri e ne stampa la somma.

```

#include <stdio.h>
#include <stdlib.h> /* Necessario per atoi */
int main(int argc, char *argv[]){
    int a, b;

    if (argc!=3){ /* Considerare che argv[0] è il nome del programma */
        printf("Usage: somma a b\n");
        return 1; /* Codice di errore */
    }

    a = atoi(argv[1]); /* Conversione a int*/
    b = atoi(argv[2]);
    printf("%d\n", a+b);

    return 0; /* Nessun errore */
}

```

Nota: cosa succede se viene lanciato come **./somma 3 4** e come **./somma ciao mondo**?

u3-s7-puntatori

Sistemi Operativi

Unità 3: Programmazione in C

I puntatori

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. Puntatori in C
2. Puntatori e vettori
3. Puntatori e stringhe
4. Puntatori e funzioni
5. Puntatori a **struct**

Puntatori in C

Definizione di Puntatore

Definizione. (*puntatore*)

Un *puntatore* è una variabile che contiene un indirizzo di memoria.

Non ci interessa come è strutturato l'indirizzo (di solito è di 8byte)

- Esso è comunque un *indirizzo virtuale*, che viene tradotto in un *indirizzo fisico dalla Memory Management Unit*.

Implementazione in C.

In C, una *variabile puntatore* contiene un *indirizzo di memoria* dove è contenuta una variabile *di un certo tipo* (quindi sono tipizzati!).

- Tutte le volte che dichiaro un puntatore, devo dichiarare anche che tipo di dato contiene l'indirizzo di memoria che contiene. Questo perché il compilatore dev'essere in grado di *interpretare l'indirizzo* dato.
 - *Fondamentale per l'utilizzo pratico*
-

Puntatori in C: Sintassi

1. Dichiarazione di un puntatore: **tipo * nome;**

Esempio:

```
int * pi; // Puntatore a int
float * pf; // Puntatore a float
```

Il tipo **int *** indica una variabile puntatore a intero: contiene l'indirizzo di memoria alla quale troviamo una variable intera.

2. Assegnazione un puntatore: si può usare l'operatore **&** per ottenere l'indirizzo di una variabile esistente.

Esempio:

```
int a = 5;  
int * pi;  
pi = &a; // pi contiene l'indirizzo di a
```

Nota: gli indirizzi sono dei numeri interi. Non è dato sapere quanto lunghi, dipende dall'architettura.

3. **Accesso alla variabile puntata** (oppure *dereferenziazione*): l'operatore ***** applicato a un puntatore serve per accedere alla variabile puntata. Detto *operatore di dereferenziazione*.

Esempio:

```
int a = 5;  
int * pi;  
pi = &a; // pi contiene l'indirizzo di a  
int b = *pi; // b contiene il valore 5
```

L'operatore ***** è l'inverso di **&**:

Pertanto: ***(&a) == a** e **&(*pi) == pi**

4. **Scrittura nella variabile puntata:**

Usando l'operatore ***** si può sia leggere che *scrivere nella zona di memoria*.

Esempio:

```
int a = 5;  
int * pi = &a; // pi contiene l'indirizzo di a  
*pi = 10; // Modifico il contenuto di a  
printf("%d", a); // Stampa 10
```

Esempio Generale di Puntatori

Esempio:

```
#include <stdio.h>

int main ()
{
    int a=5, *p;

    p = &a;
    printf("a=%d\n", a); // Stampa: 5
    printf("p=%p\n\n", p); // Stampa un indirizzo, e.g., p=0x7ffc6de0703c

    printf("&a=%p\n", &a); // Stampa lo stesso indirizzo, e.g., p=0x7ffc6de0703c
    printf("*p=%d\n", *p); // Stampa: 5

    return 0;
}
```

Puntatori in C: Osservazione sulla funzione scanf

Osservazione: ora appare più chiaro perché nella funzione **scanf** bisogna usare l'operatore **&** per passare gli argomenti in lettura.

```
float a;
scanf("%f", &a);
```

Significa che la funzione **scanf** riceve come argomento l'indirizzo di una variabile **float**.

La funzione scriverà in quell'indirizzo il valore letto da tastiera.

Internamente effettuerà un'operazione del tipo:

```
*pf = valore;
```

Puntatori e vettori: Similitudini

Osservazione. (relazione puntatori-vettori)

In C, puntatori e vettori hanno una *stretta relazione*. Infatti, sono praticamente le *stesse*.

Il nome di un vettore senza indice, ritorna l'indirizzo del primo elemento del vettore.

Esempio:

```
C  
int v[5] = {5,6,7,8,9};  
int * pi;  
pi = v; // Operazione consentita  
printf("%d\n", *pi); // Stampa: 5
```

Dato il vettore: **int v[5];**, sono equivalenti

- **v** \Leftrightarrow **&(v[0])**
- **v[0]** \Leftrightarrow ***v**

Corollario. (Aritmetica dei puntatori)

E' possibile *sommare interi a un puntatore*, per accedere a *locazioni contigue*.

Ogni incremento di 1 di un puntatore, fa accedere al blocco successivo *di lunghezza del tipo del puntatore*.

Esempio

- Il puntatore **int * pi** contiene l'indirizzo 1000
- Il tipo **int** è su $32bit = 4B$
- Allora: **pi+1** indica l'indirizzo 1004, **p+2** indica 1008
- In generale: **pi+N** indica l'indirizzo $1000 + N \times 4$ (ovvero vado alla "*prossima cella*")

Osservazione.

Sono quindi equivalenti:

- **&(v[2])** e **v+2**
- **v[2]** e ***(v+2)**
- In generale ***(v+i)** \Leftrightarrow **v[i]**, per tali valori *i* consentiti.

Si può iterare su un vettore facendo:

```
for (i=0; i<N; i++)
    v[i] = ...
    ... equivale a ...
    *(v+i) = ...
```

Puntatori e vettori: Differenze

Differenze tra puntatori e vettori

- Notiamo che i *puntatori e vettori* sono comunque *diverse*.
- Un *puntatore* può essere *riassegnato* per puntare a un altro indirizzo
- Un *vettore* è un contenitore *immutabile*.
 - Tecnicamente è un *puntatore costante*. Non gli può essere assegnato un altro valore.

```
char v[10], *pv;
pv = v; // Consentito
pv = v + 3; // Consentito
v = pv; // Errore!
v = pv + 2; // Errore!
```

Corollario. (*Rappresentazioni delle stringhe*)

Sappiamo che una *stringa* è un vettore di **char** terminato dal terminatore '\0'. Abbiamo due modi per rappresentare tale stringa:

```
char stringa [] = "ciao";
char * ps = stringa;
```

- Un puntatore a **char** può *riferirsi a una* stringa.
- Il puntatore **ps** contiene l'*indirizzo del primo elemento* di **stringa**.
Tutte le *funzioni di manipolazione delle stringhe* prendono come argomento *un puntatore* a **char**

```
strlen(stringa);
```

La funzione **strlen** prende come argomento un **char ***

Puntatori e stringhe: Errori gravi

Errori gravi:

1. Dereferenziare un puntatore non inizializzato: *comportamenti imprevedibili*, al meglio un *segmentation fault*

```
int * pi;
int a = *pi; // Errore! pi contiene un indirizzo a caso!
```

2. Dereferenziare un intero (oppure qualsiasi cosa che *non sia* un puntatore): *comportamenti ancora più imprevedibili*

```
int i = 12;
int j = *i; // Errore: i contiene 12, non un indirizzo
```

In questo caso, il compilatore *solleva un errore (segmentation fault)*.

Puntatori e funzioni: Passaggio dei Parametri per Riferimento

Definizione. (*passaggio di parametri per riferimento*)

I puntatori si usano per passare parametri *per riferimento*.

In questo modo, la funzione *può modificare gli argomenti* che riceve e il chiamante vederne gli effetti.

- Come la **scanf** che *modifica il valore* di una variabile argomento.
- Tecnica usata quando una funzione deve ritornare *più di un valore* (ricordiamo che una funzione può tornare al massimo *un solo* valore)
 - I valori di ritorno aggiuntivi *sono puntatori forniti dal chiamante*
 - In cui la funzione *colloca il risultato*

Esempio: si scriva una funzione che calcola la lunghezza di una stringa terminata da '\0'.

```
C  
int len ( char * s ){  
    int i = 0;  
    while ( *(s+i) != '\0' ) // Aritmetica dei puntatori  
        i++;  
    return i;  
}
```

Utilizzo:

```
C  
char stringa [] = "ciao!";  
int a;  
a = len(stringa); /* Non è necessario l'operatore &.  
Il nome di una variabile vettore  
già indica l'indirizzo del primo elemento */
```

Esempio: si scriva una funzione che prende due interi per riferimento e ne scambia il valore.

```
C  
void swap ( int * a, int * b ){  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

Utilizzo:

```
C  
int i = 5;  
int j = 7;  
swap (&i, &j);
```

Dopo l'esecuzione: **i=7** e **j=5**

Puntatori a **struct**

Puntatori a struct: un puntatore può tranquillamente puntare una **struct**.

```
C  
struct punto {float x; float y;}  
struct punto p1 = {1, 4};  
struct punto * pp;  
pp = &p1; // Assegno a pp l'indirizzo di p1
```

Se può accedere agli elementi di una struct *tramite puntatore*. Alternativamente è possibile usare la sintassi →, spiegata come segue.

```
C  
(*pp).x; // Contiene il valore 1  
(*pp).y; // Contiene il valore 4
```

Operatore →: si utilizza su puntatori a **struct**.

permette di accedere direttamente a un campo della **struct** puntata.

Sintassi: **puntatore→campo** ($\equiv (*\text{puntatore}).\text{campo}$)

Esempio:

```
C  
struct punto {float x; float y;}  
struct punto p1 = {1, 4};  
struct punto * pp = &p1;
```

Le seguenti istruzioni si equivalgono e ritornano l'**int** 1.

```
C  
(*pp).x;  
pp→x;
```

Nota. Con la prima sintassi, le parentesi sono *fondamentali*, dato che la precedenza viene data prima all'operatore **.**

Puntatori a funzione: Definizione

Osservazione preliminare.

E' possibile passare delle **funzioni** come **argomento a una funzione**.

- Si usa per rendere il **codice generico e modulare**, per il **multithreading**, ecc...
- Useremo quando ci occuperemo di **thread**.

Esempio di applicazione: Voglio creare una funzione che riceve come argomento un vettore e una **funzione**.

- Essa applica la **funzione** fornita su ogni elemento del vettore (funzione nota come "*mapping*" o "*mappatura*")

Definizione. (**puntatore a funzione**)

- Un **puntatore a funzione** è una variabile che rappresenta la posizione di una funzione.
- **Dereferenziarlo** significa **invocare** la funzione.

Un puntatore a funzione è tipizzato:

- Può puntare funzioni che ritornano un tipo ben definito
- E accettano un certo tipo di argomenti

Puntatori a funzione: Sintassi in C

Dichiarazione:

La sintassi è:

tipoDiRitorno (* nome) (tipoArg1, tipoArg2, ...)

Esempio:

```
C  
int (*pf) (int, int);  
// Il puntatore accetta solo funzioni del tipo f: int x int → int
```

Dichiara il puntatore a funzione di nome **pf** che può **puntare a funzioni** che:

- accettano due **int** come argomento
- ritornano un **int**
- ovvero una funzione del tipo $f : \text{int} \times \text{int} \rightarrow \text{int}$

Puntatori a funzione: Operazioni

1. **Assegnazione e utilizzo:** si usano gli operatori `=` per assegnazione (*senza &!*) e `*` per dereferenziazione, come di consueto.

Esempio:

```
C  
int somma (int a, int b){return a+b;}  
...  
int (*pf) (int, int); // Dichiarazione  
  
pf = somma; // Assegnazione. Non serve &  
res = (*pf)(3,5); // Invoca la funzione  
// res contiene 8
```

Caso d'Uso Frequenti

Funzioni che accettano come argomento puntatori a funzione:

E' uno degli *utilizzi più frequenti*. Rendono *generiche* le funzioni il più possibile.

Esempi:

- Funzione che *ordina* secondo un criterio fornito dall'utilizzatore
- Funzione del SO che avvia un *thread* che esegue una funzione fornita dall'utente

Puntatori a funzione: Esempi

Esempio: si crei e si usi una funzione che applica a ogni elemento di un vettore di interi una funzione fornita dal chiamante.

```
#include <stdio.h>
void apply(int * v, int n, int (*f)(int) ){
    for (int i=0; i<n; i++) // Dichiaro i nel loop
        v[i] = (*f)(v[i]);
}

int square(int a) {return a*a;}

int main(){
    int vec [] = {1,2,3,4,5};
    apply(vec, 5, square);
    for (int j=0; j<5; j++)
        printf("vec[%d]=%d\n", j, vec[j]);
    return 0;
}
```

Esempio: si crei e si usi una funzione che combini due interi usando una funzione fornita dal chiamante e stampi il risultato.

```
#include <stdio.h>

void combineAndPrint(int a, int b, int (*comb)(int,int) ){
    int p = (*comb)(a, b); // Dereferenziazione di comb
    printf("Combinazione: %d\n", p);
}

int add(int a, int b) {return a+b;}
int mult(int a, int b){return a*b;}

int main(){
    combineAndPrint(3, 4, mult); // Stampa 12
    combineAndPrint(3, 4, add); // Stampa 7
    return 0;
}
```

Nota:

combineAndPrint(3, 4, mult); e **combineAndPrint(3, 4, &mult);** sono equivalenti

Operazioni sui file

[Martino Trevisan](#)

[Università di Trieste](#)

[Dipartimento di Ingegneria e Architettura](#)

Argomenti

1. File
 2. Apertura e chiusura file
 3. Lettura/scrittura su file
 4. Gestione degli errori in C
-

File in C

Premessa

In C è possibile interagire coi File.

In C, come in tutti i linguaggi, è possibile *leggere e scrivere su file*.

Esistono varie funzioni per farlo.

1. **Funzioni di libreria:** portabili tra SO. In queste pagine vedremo queste, contenute in `<stdio.h>`; sono le *stesse* in tutti i *sistemi operativi*.
2. **System Call** per *Linux*

Tipi di File.

I file possono essere:

- Binari: contenere *sequenze di bit arbitrarie*.
 - |◦ Immagini, file compressi
- Testuali: contengono *solo caratteri stampabili*
 - |◦ File `.txt`, `.html`, sorgenti di programmi `.c`

Noi vedremo principalmente file testuali.

Operazioni di base sui file.

Le operazioni di base sono:

- Apertura di un file; informo l'**S.O.** che sto per farci qualcosa
- Lettura o scrittura nel file; *effettuo le operazioni necessarie*
- Chiusura del file; *concludo e salvo le modifiche*

Vedremo le funzioni principali per queste operazioni.

- Esistono *anche altre operazioni*, che vedremo nel corso quando parleremo di file e **file system**
-

Apertura e chiusura file

Apertura di un File

Apertura di un File in generale.

Un programma può accedere a file su disco, tramite il loro **path** (ovvero quella stringa che indica la posizione del file sul disco).

Prima di leggere o scrivere un file, il programma deve *aprirllo*.

- Indicare al sistema operativo che accederà a tale file e in che **modalità**

La **modalità** può essere:

- Lettura (read)
- Scrittura (write)
- Aggiunta (o *append*)

L'accesso (scrittura/lettura) ai file è **sequenziale**.

Si inizia a leggere o scrivere dall'inizio e si prosegue

- Simile all'idea di **cursore** degli editor grafici; infatti c'è un "**cursore virtuale**", che rappresenta la **posizione del file aperto**.
- Esistono funzioni per riposizionare il **cursore**, vedremo più avanti

Quando si apre un file, bisogna *indicare* se esso è **binario o testuale**.

- Se testuale, le funzioni si aspettano **\n** per delimitare le righe

Implementazione in C.

Per indicare un **file aperto**, su cui è possibile effettuare operazioni, in C si usa il tipo **FILE**

- Tecnicamente esso è *un puntatore* a una variabile di tipo **FILE**.
- Non ci interessa che tipo è **FILE**; precisamente sarebbe una *struct*, ma è in ogni caso una questione che non ci riguarda.

FILE * è un cosiddetto *handle opaco*: è un puntatore. *Ma non ci interessa a cosa punta*.

Solo le funzioni di libreria hanno interesse ad accedere al dato, al *programmatore* non ne interessa niente.

Pertanto il contenuto di **FILE** può cambiare o non seguire uno standard.

Apertura di File in C

Per aprire un file si usa la funzione **fopen(path, modo)**. Essa ritorna un **FILE ***.

- **path** può essere assoluto o relativo.
- **modo** indica se apriamo in lettura (**r**), scrittura (**w**) o aggiunta (**a**). Di default la modalità è *testuale*. Per indicare *modalità binaria*, aggiungere **b** (es. **rb** o **wb**).
- Nota benissimo: si deve inserire una *stringa* in entrambi i casi!

Esempio:

```
FILE * f;  
f = fopen("file.txt", "r");  
if (f==NULL){ /*Errore*/ }
```

In caso di errore, la **fopen** ritorna il puntatore nullo **NULL**.

Note. (*Comportamenti in casi di non-esistenza del file*)

- In modalità *scrittura e aggiunta*, se il file non esiste, *viene creato*.
- In modalità *lettura*, se il file non esiste, la **fopen** ritorna **NULL**.
- In modalità *scrittura*, se il file esiste, il suo contenuto *viene cancellato all'apertura*

Chiusura di File

Chiusura file: quando non si accede più a un file, bisogna *chiuderlo* e dismettere il corrispondente **FILE ***

Si chiama la funzione **fclose(file)** che accetta come argomento un **FILE ***.

- Mai chiamare la **fclose** con un **FILE *** invalido settato a **NULL**!
- Si può chiudere un file aperto *solo una volta*

Se un file non viene chiuso, la chiusura è effettuata *automaticamente dal SO* quando il programma termina.

Lettura/scrittura su file

Lettura su File

Le funzioni che si usano simili a quelle che si usano per leggere da tastiera e scrivere su console.

1. Lettura:

- **fgetc(file)**: legge *un carattere* e lo fornisce come *valore di ritorno*
 - Ritorna la costante **EOF** se il file è finito ("*End of File*")
- **fgets(buffer, N, file)**: legge *una stringa* di massimo **N** caratteri.
 - Legge fino a quando ha letto **N** caratteri o trova **\n**, che è *incluso* nella stringa ritornata e terminata da **\0**
 - Ritorna **NULL** (= 0) se il file è finito, altrimenti ritorna **buffer**

Scrittura su file

2. Scrittura:

- **fputc(carattere, file)**: scrive un *carattere* su file
- **fputs(stringa, file)**: scrive una *stringa* su file

Nota: queste funzioni possono leggere e scrivere *anche su terminale*. E' sufficiente dire loro di leggere dal file **stdin** e scrivere su file **stdout**.

- **fputc('a', stdout) ≡ putc('a')**
- Abbiamo già visto che la **fgets** è una valida alternativa alla insicura **gets**.
- Infatti queste funzioni sono le "*versioni generiche*" delle funzioni appena menzionate.

Esempio: si legga un path da tastiera e se ne stampi il contenuto come file di testo

```
#include <stdio.h>

int main ()
{
    char s[100], buffer [100];
    FILE * f;

    printf("Inserisci un path: ");
    scanf("%s", s);

    f = fopen(s, "r");
    if (f==NULL){
        printf("Impossibile aprire %s\n", s);
        return 1; /* Errore */
    }

    /* Non è importante la lunghezza di buffer */
    while ( fgets(buffer, 100, f) )
        fputs(buffer, stdout); // Equivale a printf("%s", s);

    fclose(f);
    return 0;
}
```

Comandi Generali di Lettura/Scrittura su File

Comandi Alternativi.

- Alternativamente si possono usare le funzioni **fprintf** e **fscanf** che sono equivalenti a **printf** e **scanf** con la differenza che scrivono *su file* e *non da console*.

Sintassi:

fprintf(file, formato, argomenti)
fscanf (file, formato, &argomenti)

Queste funzioni sono particolarmente utili per leggere e scrivere *numeri interi e reali*

Esempio:

```
fprintf(f, "%d\n", 123); // scrive il numero 123 e un ritorno a capo
```

Esempio: si legga un numero N da tastiera e si stampi sul file **numeri.txt** i numeri da 1 a N

```
C

#include <stdio.h>

int main ()
{
    int n, i;
    FILE * f;

    printf("Inserisci un numero: ");
    scanf("%d", &n);

    f = fopen("numeri.txt", "w");
    if (f==NULL){
        printf("Impossibile aprire numeri.txt\n");
        return 1;
    }

    for (i=1; i<=n; i++)
        fprintf(f, "%d\n", i);

    fclose(f);
    return 0;
}
```

Nota su **fgetc** e **fgets**

Nota.

In caso di **lettura**, abbiamo visto che **fgetc** e **fgets** hanno *comportamenti diversi*. In pratica, in caso di file completamente letto:

- **fgetc** ritorna la costante **EOF**
- **fgets** ritorna la costante **NULL**
- **fscanf** ritorna la costante **EOF**
- Non abbiamo **numeri** 1 o 0
Questo è un retaggio storico delle versioni vecchie.

E' possibile usare la funzione **eof(FILE *)** per verificare se il file è stato letto completamente.

- Ritorna **TRUE** / **FALSE**

Formati per comandi `printf` e `scanf`

Abbiamo visto che `(f/s)printf` e `(f/s)scanf` permettono di *specificare il formato* come `%d %f %s %c`.

Riassumiamo:

- Carattere `char`: `%c`
- Stringa `char []`: `%s`
- Intero `int`: `%d`
- Reale `float`: `%f`

Lista completa: <https://man7.org/linux/man-pages/man3/printf.3.html>

Esistono modificatori per definire numero di cifre decimali, padding per interi, ecc..

Esercizio su Lettura/Scrittura di File

Esercizio: si legga `persone.txt` che contiene su ogni riga nome ed età di una persona, separati da `' '`. Si calcoli l'età media. Esempio di file:

```
martino 31
```

```
andrea 37
```

```
#include <stdio.h>

int main (){
{
    int n=0, s, e;
    FILE * f;
    char nome[100];

    f = fopen("persone.txt", "r");
    if (f==NULL){
        printf("Impossibile aprire persone.txt\n");
        return 1;
    }

    /* La fscanf si aspetta di trovare su ogni riga una parola e un intero */
    while (fscanf(f, "%s %d\n", nome, &e) != EOF){
        n++; s+=e; /* Accumula i contatori */
    }

    printf("La media è %f\n", (float)s/n ); /* Notare il casting */
    return 0;
}
```

Gestione degli errori in C

Problematica della Gestione degli Errori

Osservazione.

Abbiamo visto che le funzioni di libreria segnalano un eventuale errore tramite il valore di ritorno

Esempio:

```
f = fopen("file.txt", "r");
if (f==NULL){
    /*Errore*/
}
```

Problema.

Con questo meccanismo, non è possibile sapere niente su *quale sia stato l'errore*.

- File non esistente?
- No permessi di lettura?
- Non si sa

Metodo Errno

Soluzione: Metodo Errno

La *libreria standard del C* utilizza il seguente meccanismo per *specificare la causa* di errore

- Ogni programma in C ha la *variabile globale int errno*
- Una funzione di libreria che fallisce, setta **errno** con un *codice di errore esplicativo*
- Il chiamante invoca una funzione di libreria.
 - Tramite il valore di ritorno, rileva se c'è stato un errore
- Se c'è stato un errore, il chiamante legge in **errno** il codice di errore

Necessaria includere la libreria

```
#include <errno.h>
```

Definizione: Errno

La variabile globale **errno** è *intera* e contiene un *codice di errore*.

- il manuale di Linux (**man errno**) contiene la descrizione dei codici di errore
- *Esempi:* **fopen** può fallire con **ENOENT**= 2 (file inesistente), **EACCES**= 13 (permessi insufficienti) e molti altri

Stampa dell'errore

Implementazione di Errno.

Tutti i codici di errore sono costanti definite nella libreria standard. Adesso basta trovare un modo per *associare* il codice all'*errore* effettivo.

- Il programmatore può confrontare **errno** con le costanti definite in **<errno.h>** per identificare l'errore

Esempio:

```
C  
FILE * f = fopen("file.txt", "r");  
if (f==NULL){  
    if (errno == ENOENT)  
        printf("File inesistente\n");  
    else if (errno == EACCES)  
        printf("Permessi insufficienti\n");  
    else  
        printf("Errore generico\n");  
    return 1;  
}
```

- Esistono delle funzioni di libreria per semplificare la gestione dell'errore, tra cui la funzione **errno**.

```
C  
#include <stdio.h>  
void perror(const char *s);
```

Stampa il messaggio di errore relativo al valore corrente di **errno**, premettendo la stringa **s**

Esempio:

```
C  
FILE * f = fopen("file.txt", "r");  
if (f==NULL){  
    perror("Error opening file.txt");  
    return 1;  
}
```

Stampa: **Error opening file.txt: No such file or directory**

- Funzione **strerror**

```
#include <string.h>
char *strerror(int errnum);
```

Ritorna *una stringa* che spiega l'errore dal codice **errnum**

Esempio:

```
FILE * f = fopen("file.txt", "r");
if (f==NULL){
    printf("Impossibile aprire il file. Errore: %s\n", strerror(errno));
    return 1;
}
```

Stampa: **Impossibile aprire il file. Errore: No such file or directory**

Gestione degli errori in C: i limiti

Problemi.

La gestione degli errori tramite **errno** è considerata obsoleta.

- I linguaggi più moderni usano i costrutti **try catch** (oppure **try**, **except** in Python)

Casi d'uso problematici.

La gestione degli errori tramite la variabile globale **errno** è una *tecnica problematica*, in caso di:

- In caso di segnali (*vedremo*)
- Fortunatamente **errno** è thread safe (*vedremo*)

u3-s9-esercizi

Sistemi Operativi

Unità 3: Programmazione in C

Esercizi

[Martino Trevisan](#)

[Università di Trieste](#)

Argomenti

1. Stampa di file
 2. Area di un triangolo
 3. Calcolo del minimo
 4. Calcolo della media
 5. Somma di vettori bidimensionali
-

Stampa di file

Si scriva un programma che riceve il nome di un file da riga di comando e ne stampa il contenuto.

```
C  
#include <stdio.h>  
int main(int argc, char *argv[]){  
    FILE * f;  
    char buffer[100];  
  
    if (argc!=2){ /* Controllo degli argomenti */  
        printf("Uso: ./stampa path\n");  
        return 1;  
    }  
  
    f = fopen(argv[1], "r");  
    if (f == NULL){ /* Controllo sul file */  
        printf("Impossibile aprire il file\n");  
        return 1;  
    }  
  
    /* Stampa finchè non termina il file */  
    while (fgets(buffer, 100, f) !=NULL)  
        printf("%s", buffer);  
  
    fclose(f);  
    return 0;  
}
```

Area di un triangolo

Si scriva un programma che riceve base e altezza di un triangolo da riga di comando e stampa la sua area. Base e altezza sono numeri con virgola.

```
C

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    float base, altezza;

    if (argc!=3){ /* Controllo degli argomenti */
        printf("Uso: ./area base altezza\n");
        return 1;
    }

    /* Conversione */
    base = atof(argv[1]);
    altezza = atof(argv[2]);

    /* Controllo base e altezza maggiori di 0 */
    if (base<0 || altezza <0){
        printf("Parametri non validi. Devono essere maggiori di 0.\n");
        return 1;
    }

    printf("Area: %f\n", base*altezza/2);

    return 0;
}
```

Calcolo del minimo

Si scriva un programma che riceve come parametro il nome di due file:

- Il primo file è di input e contiene un intero *positivo* per riga
- Il secondo file è di output e vi viene scritto il numero minimo del file di input

Si crei una riga di comando in bash che svolge lo stesso compito, ipotizzando che il file di input sia **in.txt** e quello di output **out.txt**

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    FILE * fin, *fout;
    int n, min;

    if (argc!=3){ /* Controllo degli argomenti */
        printf("Uso: ./minimo filein fileout\n");
        return 1;
    }

    fin = fopen(argv[1], "r");
    if (fin == NULL){ /* Controllo sul file */
        printf("Impossibile aprire il file in input\n");
        return 1;
    }

    fout = fopen(argv[2], "w");
    if (fout == NULL){ /* Controllo sul file */
        printf("Impossibile aprire il file di output\n");
        return 2;
    }

    fscanf(fin, "%d\n", &min); /* Valore iniziale per il minimo */
    while (fscanf(fin, "%d\n", &n) != EOF ) /* Cerca il minimo */
        if (n<min)
            min = n;

    fprintf(fout, "%d\n", min);
    fclose(fin);
    fclose(fout);
    return 0;
}
```

La versione in bash è molto più compatta

SHELL

```
cat in.txt | sort | head -1 > out.txt
```

Calcolo della media

Si scriva e si testi una funzione che calcola la media di un vettore

```
#include <stdio.h>

float media(int n, float * v){
    float s = 0;
    int i;
    for (i=0; i<n; i++)
        s+=v[i];
    return s/n;
}

int main(int argc, char *argv[]){
    float lista [] = {1.5, 2.5, 4};
    printf("Media: %f\n", media(3, lista) );
    return 0;
}
```

Somma di vettori bidimensionali

Si scriva e si testi una funzione che calcola la somma di due vettori bidimensionali.
Se ne fornisca una versione con e senza l'uso delle **struct**.

Senza **struct**

E' necessario l'uso dei puntatori, siccome la funzione deve ritornare due valori.

```
#include <stdio.h>

void sommaV(float x1, float y1, float x2, float y2,
            float * pxres, float * pyres) {
    *pxres = x1+x2;
    *pyres = y1+y2;
}

int main(int argc, char *argv[]){
    float punto1x = 1.1, punto1y = 2.0, punto2x = 3.6, punto2y = 2.7;
    float puntoSx, puntoSy;

    sommaV(punto1x, punto1y, punto2x, punto2y, &puntoSx, &puntoSy);
    printf("La somma vettoriale e': (%f, %f)\n", puntoSx, puntoSy);

    return 0;
}
```

Con **struct**

Utilizziamo una **typedef** per evitare di ripetere molte volte la keyword **struct**.

```
#include <stdio.h>

typedef struct {
    float x;
    float y;
} punto;

punto sommaP(punto p1, punto p2){
    punto risultato;
    risultato.x = p1.x + p2.x;
    risultato.y = p1.y + p2.y;
    return risultato;
}

int main(int argc, char *argv[]){
    punto p1 = {1.4, 4.2};
    punto p2 = {3.2, 5.9};

    punto s = sommaP(p1, p2);
    printf("La somma vettoriale e': (%f, %f)\n", s.x, s.y);

    return 0;
}
```