ARCHITETTURA DI UN CALCOLATORE - CENNI

«**Informatica: arte e mestiere**»

Ceri, Mandrioli, Sbattella, Mc Graw-Hill

cap.2

Vista funzionale di un calcolatore

Le funzioni svolte da un calcolatore possono essere classificate in quattro tipologie:

1. **Elaborazione dati**: architettura generale in grado di coniugare flessibilità nel calcolo, scalabilità e standardizzazione dei componenti, abbattimento dei costi, …
2. **Memorizzazione dati**: persistente e per brevi periodi
3. **Trasferimento dati** da o verso l'esterno: tramite periferiche e trasmissione dati
4. **Controllo**: coordina le risorse del calcolatore

Dispositivo polivalente e adattabile, ad applicazioni diversificate: le sue **funzionalità** vengono **specializzate** mediante la **programmazione**.

L'hardware fornisce le funzionalità di base che consentono al software di realizzare tale specializzazione.

Architettura di un calcolatore

Con il termine "architettura" di un calcolatore intenderemo **l'insieme delle parti e delle loro interconnessioni che consentono determinate funzionalità "visibili" al programmatore**

- Es. un calcolatore mette a disposizione un'operazione per fare la somma di due numeri. Questa operazione fa parte dell'architettura del calcolatore e potrà essere usata dal programmatore

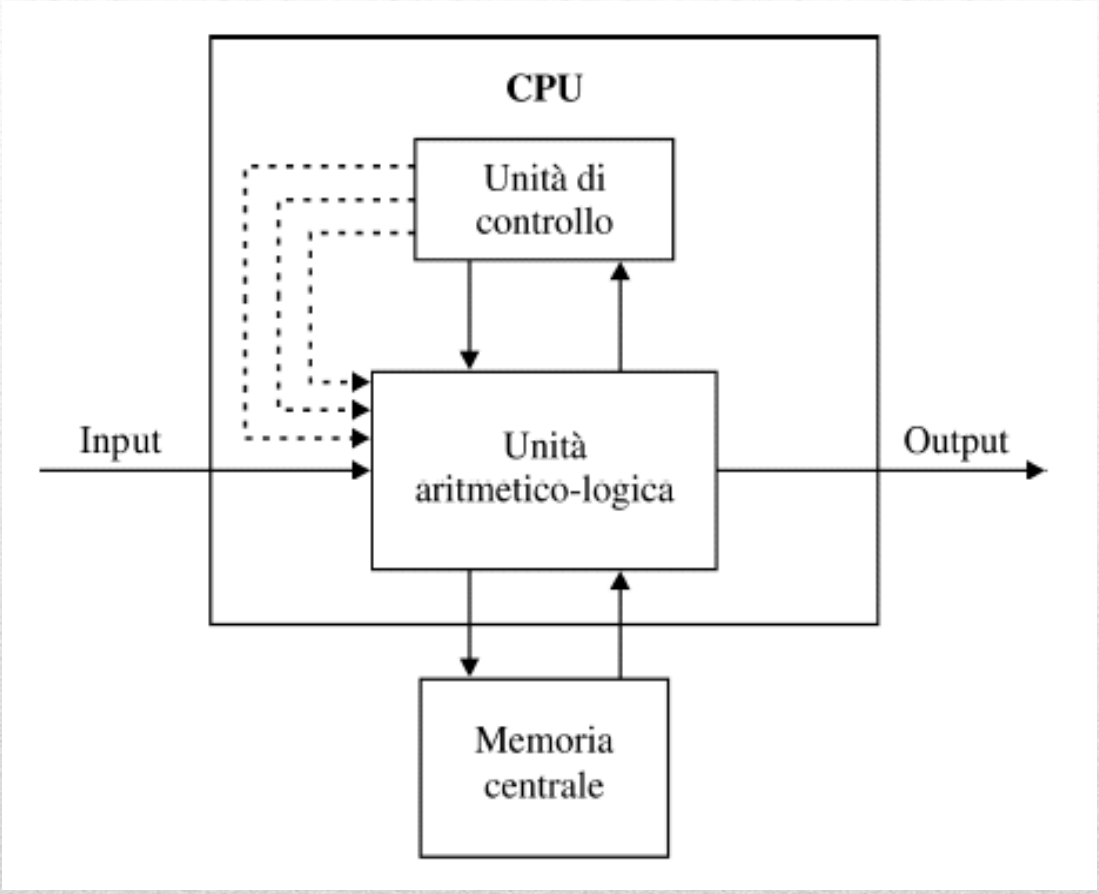
L'architettura può essere vista a vari **livelli di astrazione**

- Livello puramente "fisico": unità centrale, tastiera, monitor, disco, …
- Livello "logico" (nel senso "non fisico") o delle istruzioni: **architettura di Von Neumann**

Architettura di Von Neumann

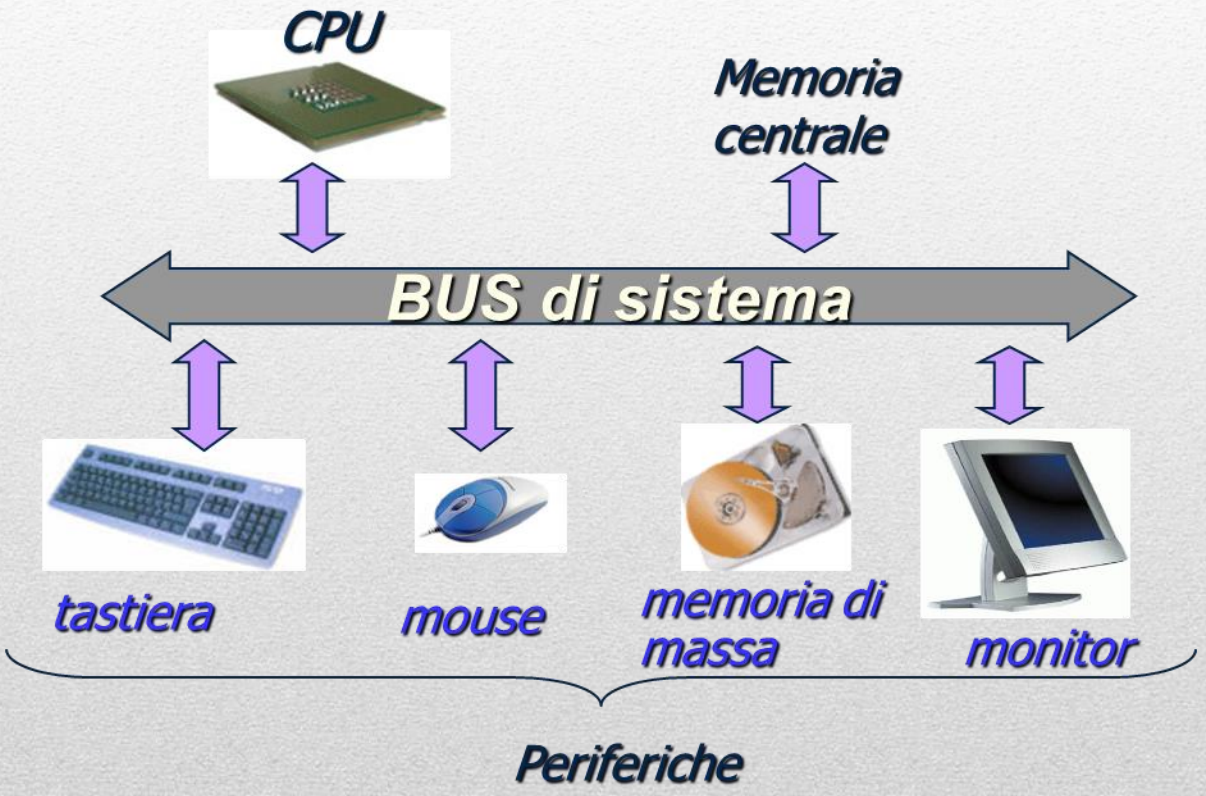
L'architettura di von Neumann (proposta dallo scienziato ungherese/statunitense John Von Neumann) è ancora quella dei sistemi di elaborazione di oggi

- dati in ingresso da trasformare in dati in uscita
- un programma da eseguire per effettuare la trasformazione
- una memoria in cui contenere il programma e i dati intermedi dei calcoli
- un agente che esegua le azioni programmate



Programmi e dati vengono memorizzati allo stesso modo nella stessa unità fisica, la memoria

Architettura di Von Neumann



Il modello di macchina di Von Neumann

Gli algoritmi che progetteremo si basano sulle caratteristiche e sulle capacità di base di un calcolatore basato sulla Macchina di Von Neumann, cioè un modello di calcolo ideale. Le componenti sono le seguenti:

- Le **unità di input** tramite cui la macchina acquisisce informazioni dall'esterno
- Le **unità di output** tramite cui la macchina produce (stampa) informazioni all'esterno
- **L'unità centrale di elaborazione** (CPU) che elabora le istruzioni del programma (i passi dell'algoritmo), composta da due elementi:
  - **Unità di controllo**: stabilisce l'ordine con cui devono essere eseguite le operazioni
  - **Unità logico-aritmetica**: esegue operazioni aritmetiche e risolve espressioni logiche
- La **memoria** in cui l'unità centrale deposita ed estrae le informazioni per poterle elaborare

Schema di funzionamento

I programmi sono composti da istruzioni codificate in binario:

- istruzioni di elaborazione (ad es. operazioni numeriche)
- istruzioni di trasferimento di dati tra due componenti della macchina

Un calcolatore esegue un programma sulla base dei seguenti principi:

- Dati e istruzioni sono memorizzati in una memoria *unica* che permette sia la scrittura che la lettura
- I contenuti della memoria sono indirizzati in base alla loro posizione, indipendentemente dal tipo di dato o istruzione contenuto
- Le istruzioni vengono eseguite in modo sequenziale

Il funzionamento della macchina di Von Neumann è un ciclo continuo:

1. la CPU estrae le istruzioni dalla memoria principale (**fetch**) ...
2. ...le decodifica (**decode**) determinando l'operazione da eseguire e i gli operandi ...
3. ...e le esegue (**execute**)

Il modulo di memoria

Le prestazioni della memoria possono influenzare in modo rilevante le prestazioni complessive di un calcolatore.

Per consentire un'**efficiente** esecuzione del programma, la memoria che contiene dati e istruzioni dovrebbe essere **veloce** quanto la CPU e dovrebbe avere **dimensioni** sufficienti a contenere sia il programma che i dati. Inoltre, dovrebbe assicurare una memorizzazione **persistente**.

1. **Memoria centrale**: ad elevata velocità di accesso , per programmi in esecuzione e relativi dati, supporto alla CPU (tecnologie elettroniche)
2. **Memoria di massa**: grandi moli di dati non utilizzati frequentemente, memorizzati in modo stabile (non volatile) e dal costo non proibitivo (tecnologie magnetiche, ottiche)

Il modulo di memoria

Quattro livelli:

- **Registri**, (interni alla CPU) capaci di memorizzare parole singole
  - Tipicamente dati “in transito” relativi ad un particolare dato o istruzione in esecuzione
- Memoria **cache** (integrata nella CPU)
  - Area di memoria ad accesso rapido finalizzata a contenere istruzioni e dati usati più frequentemente
- Memoria **centrale o primaria** (esterna alla CPU ma interna al calcolatore)
  - Contiene istruzioni e dati del programma in esecuzione
- Memoria **secondaria** (esterna al calcolatore)
  - Fa parte dei moduli periferici

La memoria centrale

Conserva le istruzioni e i dati dei programmi **in esecuzione**. Inevitabile ingresso/uscita delle informazioni dalla memoria di massa alla memoria centrale e viceversa.

Dati memorizzati in ***bit*** (*binary digit*): ogni unità elementare di memoria contiene un'informazione di tipo *binario*: 1 oppure 0

- realizzazione mediante dispositivi *fisici* a *due stati* (transistor a semiconduttori, due livelli di tensione)

E' organizzata come sequenza di *celle* o *parole*:

- **Parola**: insieme di più byte (una potenza di 2: tipicamente 1, 2, 4, 8)
- **Byte**: insieme di 8 bit

Mentre il bit rappresenta l'unità elementare di informazione, la parola di memoria è la più piccola quantità di memoria accessibile.

Le celle di memoria di un elaboratore hanno tutte la stessa capacità, mentre elaboratori differenti possono avere parole di lunghezza differente.

La memoria centrale

Ogni cella è individuata da un indirizzo:

- numero che indica la posizione relativa rispetto alla prima cella, che ha indirizzo 0

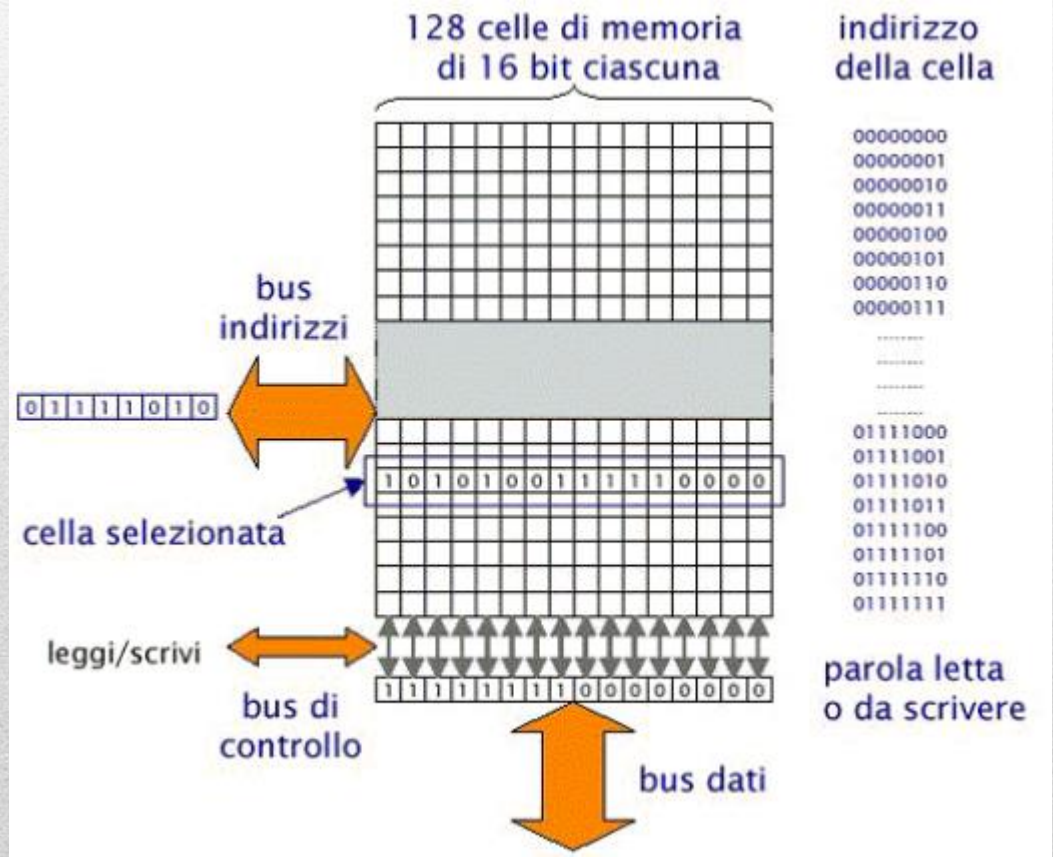
L'indirizzamento della memoria avviene tramite un opportuno registro, detto **registro degli indirizzi** (**MAR**). Se il registro indirizzi ha **k** bit si possono quindi indirizzare 2^k (da 0 a 2^k-1) celle di memoria. Quindi anche la dimensione della memoria è una potenza di 2.

Dal punto di vista dell'esecuzione delle istruzioni, alla memoria centrale si accede tramite le linee del bus:

- Il bus **indirizzi** trasferisce gli indirizzi delle celle cui si vuole accedere
- Sulle linee del bus **dati** vengono trasmessi i dati
- I segnali del bus di **controllo** specificano il tipo di operazione richiesta

Lettura/scrittura

Le operazioni che possono essere effettuate in memoria sono quelle di lettura e scrittura; in entrambi i casi è necessario utilizzare un secondo registro (**registro dati** - **MDR**) che possiede la stessa dimensione della parola e che viene utilizzato per contenere il dato letto/scritto.



Lettura

La lettura di una locazione di memoria consiste nel trasferimento fisico dei byte che costituiscono la locazione dalla memoria alla unità centrale di processo, senza modificare la locazione di memoria

Una operazione di lettura consiste nei seguenti passi:

- si scrive sul MAR l'indirizzo della locazione da leggere,
- questo poi viene trasferito al bus degli indirizzi che trasporta l'indirizzo in memoria,
- la quale scrive a sua volta sul bus dei dati il contenuto della locazione di memoria selezionata,
- che successivamente viene inserita o caricata (**load**) nel registro dei dati (MDR Memory Data Register) restando così disponibile alla CPU.

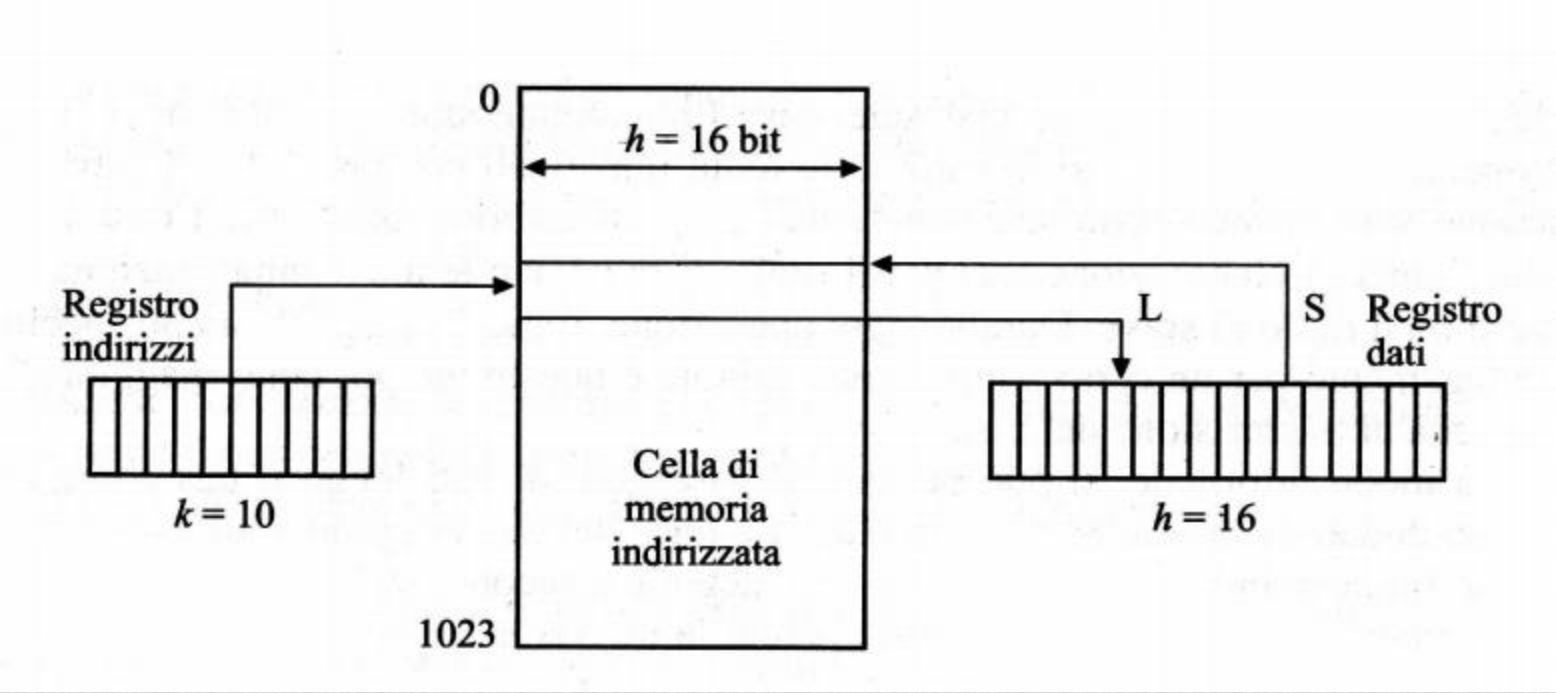
Scrittura

La scrittura in una locazione di memoria consiste nel trasferimento fisico del contenuto del registro dei dati (MDR) nella cella di memoria selezionata tramite il registro MAR.

Una operazione di scrittura consiste nei seguenti passi:

- si scrive sul registro MDR il dato da inserire o immagazzinare (**store**),
- si scrive sul MAR l'indirizzo della locazione da ricoprire (con il dato contenuto nel registro MDR),
- questo poi viene trasferito al bus che trasporta l'indirizzo in memoria,
- la quale scrive il dato che arriva dal bus dati nella locazione di memoria selezionata.

Schema di funzionamento della memoria



Caratteristiche della memoria centrale

Velocità di accesso elevata: decine di ns (10^{-9} sec)

Tempo di accesso indipendente dalla posizione del dato nella memoria

- RAM: Random Access Memory
- si contrappongono alle memorie ad accesso sequenziale, come i nastri magnetici

Dimensione limitata: oggi alcuni GB

- 2^{30} byte = 1073741824 byte ≈ 10^9 byte (un giga-byte)

L'informazione viene persa se si interrompe l'alimentazione elettrica (*volatilità*)

Memorie RAM e ROM

Un valore può essere memorizzato/recuperato dalla memoria specificando l'indirizzo

- il tempo di accesso è indipendente dall'indirizzo (ecco perché il nome di RAM)

Memorie ROM (*Read Only Memory)*

- sono memorie di sola lettura, pre-impostate dal fabbricante
- sono di fatto memorie RAM (ROM e RAM non sono termini contrapposti!) **ma** non sono volatili
- tipicamente contengono le istruzioni per l'avvio del calcolatore (firmware)
- sono usate anche in auto, elettrodomestici, ecc.

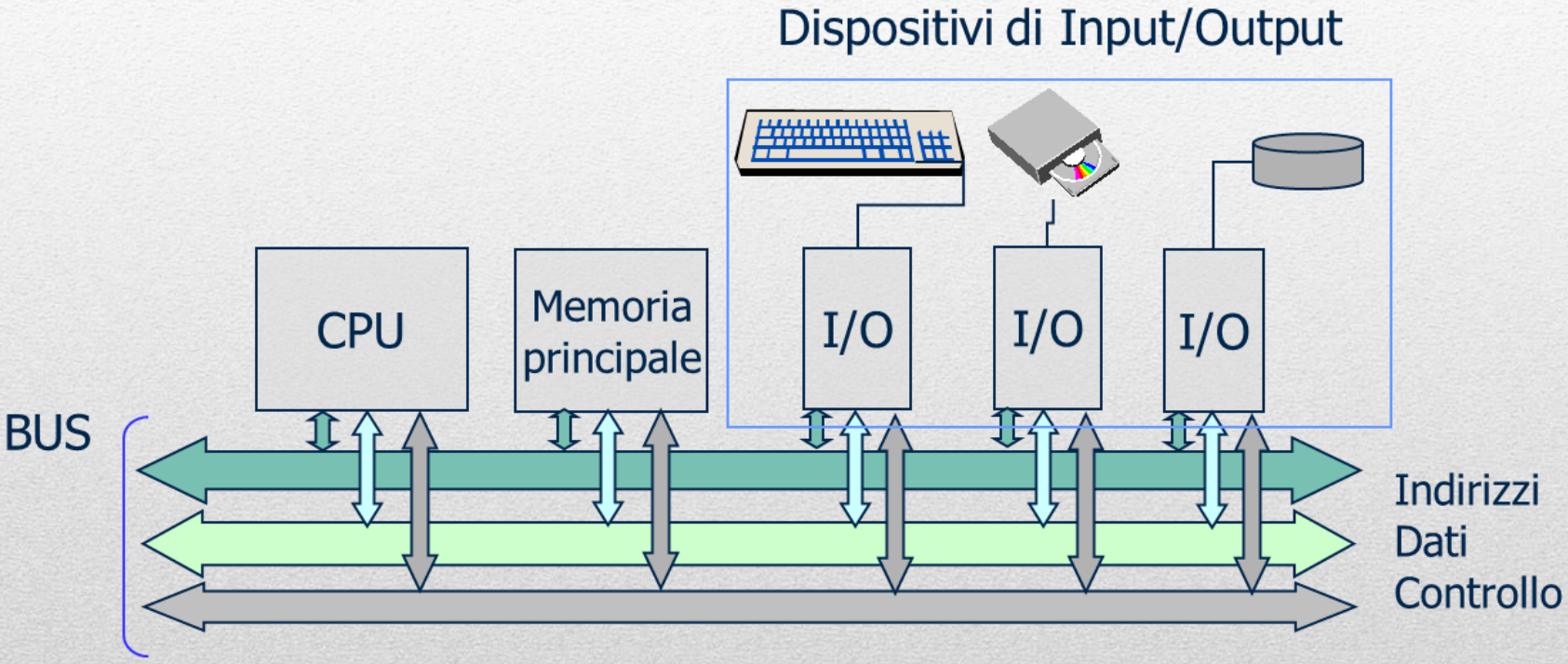
Il bus di sistema

Il bus di sistema è costituito da un insieme di connessioni lungo le quali viene trasferita l'informazione. Esso collega fra di loro l'unità di elaborazione, la memoria e le diverse interfacce di ingresso e di uscita.
Nelle implementazioni concrete il bus di sistema è costituito da tre parti distinte:

- Una mono-direzionale dal processore alla memoria detta ***bus degli indirizzi.*** *S*erve per trasmettere il contenuto del registro indirizzi alla memoria centrale selezionando così una specifica cella di memoria.
- Una bi-direzionale dal processore alla memoria e viceversa detta ***bus dei dati****.* Trasferisce dati dall'unità master all'unità slave o viceversa. I dati vengono trasferiti da una cella di memoria al registro dati a seguito di una operazione di lettura oppure dal registro ad una cella a seguito di una operazione di scrittura.
- Una bi-didirezionale dal processore alle altre unità funzionali e viceversa detto ***bus dei controlli****.* Trasferisce un codice corrispondente alla istruzione da eseguire dall'unità master all'unità slave e informazioni relative all'avvenuto espletamento dell'operazione richiesta in flusso contrario.

Il bus di sistema

Il bus è fisicamente realizzato tramite un insieme di conduttori elettrici



L'unità di elaborazione

L'unità centrale di elaborazione (CPU) è la parte del sistema che contiene gli elementi circuitali necessari al funzionamento dell'elaboratore. Questa esegue i programmi che risiedono nella memoria centrale, prelevando, decodificando ed eseguendo le istruzioni in essi contenute e coordinando il trasferimento dei dati tra le varie unità funzionali.

La CPU si compone di:

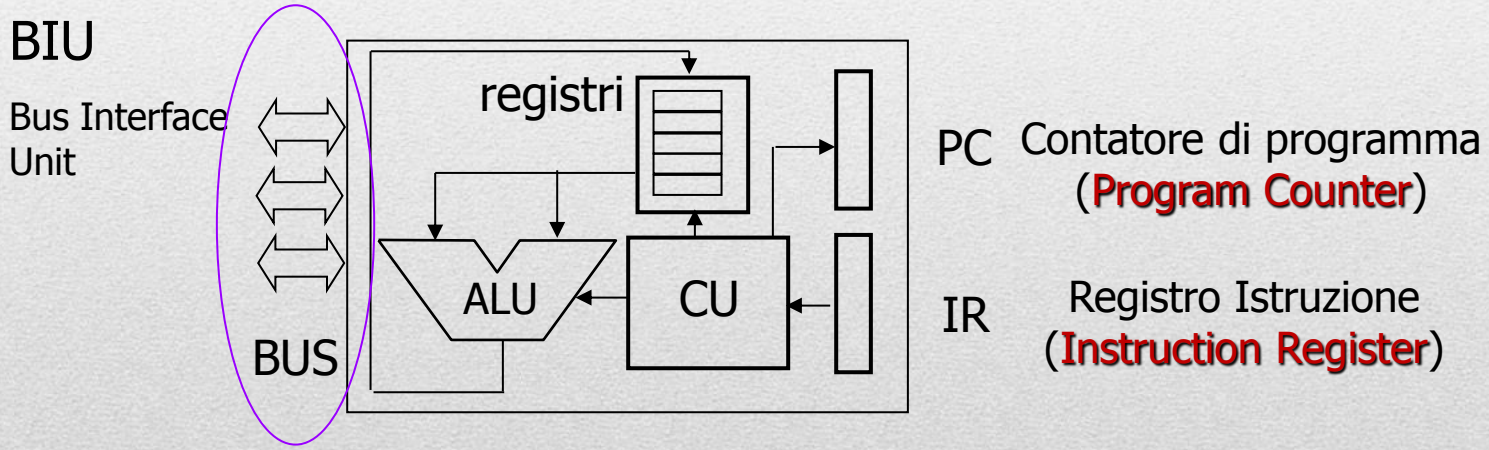
- **una unità di controllo (CU Control Unit),** che ha lo scopo di interpretare e attivare le risorse necessarie alla esecuzione delle istruzioni
- **una unità aritmetico-logica (ALU Arithmetic and Logic Unit)** in cui vengono effettuati i calcoli aritmetici e logici presenti nelle istruzioni (aritmetiche/logiche) del programma
- **alcuni dispositivi di memoria detti *registri*** che possono essere letti e scritti molto velocemente e che sono utilizzabili per memorizzare risultati parziali delle operazioni e informazioni necessarie al controllo del flusso del programma.
- **l'orologio di sistema (clock)** che sincronizza le operazioni rispetto ad una data frequenza

L'unità di elaborazione

A livello "macroscopico", ad ogni impulso di clock la CPU:

- "legge" il suo stato interno (determinato dal contenuto dei registri di stato) e la sequenza di ingresso (determinata dal contenuto dei registri istruzione e dati)
- produce un nuovo stato "dipendente" dallo stato in cui si trovava originariamente

In pratica, la CPU realizza una complessa funzione logica, con decine di ingressi e di uscite



Lo stato della CPU è costituito da informazioni memorizzate negli opportuni **registri** sui dati da elaborare, istruzione da eseguire, indirizzo in memoria della prossima istruzione da eseguire, eventuali anomalie/eventi verificatisi durante l'elaborazione.

Registri

I registri fondamentali presenti nella CPU sono:

- il registro degli indirizzi di memoria (***MAR Memory Address Register***), indica l'indirizzo della locazione di memoria che si vuole selezionare;
- il registro dei dati di memoria (***MDR Memory Data Register***), contiene il dato proveniente dalla locazione di memoria selezionata o il dato che si vuole memorizzare nella locazione di memoria selezionata;
- il contatore di programma (***PC Program Counter***) ha la funzione di guidare il flusso della esecuzione di un programma, infatti il suo contenuto indica l'indirizzo della prossima istruzione da eseguire;
- il registro della istruzione corrente (***IR Instruction Register***) che contiene l'istruzione da decodificare e eseguire;
- il registro delle interruzioni (***INTR Interrupt Register***) che contiene informazioni sullo stato di funzionamento delle periferiche

Le interfacce di I/O

Le interfacce di ingresso/uscita costituiscono gli elementi circuitali che consentono il collegamento dell'elaboratore con le varie periferiche. Esse tra loro sono molto diverse ma possiamo raggrupparle in tre categorie:

- **Unità di interazione**. Permettono all'utente di interagire con il sistema di calcolo (tastiera, mouse, scanner, stampanti, video, webcam)
- **Unità di memorizzazione**. Memorizzano in modo permanente le informazioni in esso contenute ed è per questo che vengono chiamate anche memorie permanenti (HD, CD, DVD, Floppy, ecc…)
- **Unità di comunicazione**. Permettono di collegare sistemi di calcolo diversi in modo da realizzare una rete (modem, schede di rete).

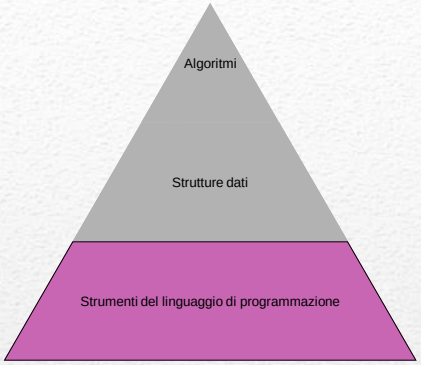
Una interfaccia contiene registri per inviare comandi alla periferica, scambiare dati, controllare il funzionamento della periferica. Una interfaccia generica potrebbe contenere i seguenti elementi:

- Un **registro dati** per scambiare dati con la periferica sia in ingresso che in uscita (PDR Peripherical Data Register)
- Un **registro comando** per contenere il comando che la periferica dovrà eseguire (PCR Peripherical Command Register)
- Un **registro di stato** della periferica (PSR Peripherical Status Register)

TIPI DI DATO

Manuale linguaggio C

Linguaggi di Computazione



Il "potere espressivo" di un linguaggio è caratterizzato da:

- quali **tipi di dati** consente di rappresentare (direttamente o tramite definizione dell'utente)
- quali **istruzioni di controllo** mette a disposizione (quali operazioni e in quale ordine di esecuzione)

PROGRAMMA = DATI + CONTROLLO

Dati

Un elaboratore è un manipolatore di simboli (segni)

L'architettura fisica di ogni elaboratore è intrinsecamente capace di trattare vari domini di dati, detti tipi primitivi

- dominio dei numeri naturali e interi
- dominio dei numeri reali (con qualche approssimazione)
- dominio dei caratteri
- dominio delle stringhe di caratteri

Tipi di Dato

Il concetto di tipo di dato viene introdotto per raggiungere due obiettivi:

- esprimere in modo sintetico
  - un insieme di valori
  - la loro rappresentazione in memoria, e
  - un insieme di operazioni ammissibili
- permettere di effettuare controlli statici (al momento della compilazione) sulla correttezza del programma.

Una medesima sequenza di bit può dunque rappresentare un numero intero positivo, un differente numero relativo, un numero razionale ovvero un carattere alfanumerico.

Per indicare alla macchina come dovrà essere trattata una certa sequenza di bit memorizzati in un determinato blocco della memoria, è necessario che il programmatore a priori **dichiari** il tipo di dato che intenderà associare ad una certa variabile nell'ambito di un intero programma o di una determinata funzione.

Le dichiarazioni

In C ogni variabile deve essere **dichiarata** prima di poter essere usata

La dichiarazione fornisce al compilatore le informazioni relative al numero di byte da allocare e alle modalità di interpretazione di tali byte

Per aumentare la concisione, è possibile dichiarare variabili dello stesso tipo separando i loro nomi con virgole

```
int j, k;          //variabili di nome j e k di tipo intero (vedi dopo)
float x, y, z;     //variabili di nome x, y e z di tipo reale
```

All'interno di un blocco, tutte le dichiarazioni devono apparire prima delle istruzioni eseguibili; l'ordine relativo delle dichiarazioni non è significativo

Tipi di Dato

Un tipo di dato T è definito come:

- un dominio di valori, D
- un insieme di funzioni F1,..,Fn sul dominio D
- un insieme di predicati P1,..,Pm sul dominio D

$$T = \{ D, \{F1,...,Fn\}, \{P1,...,Pn\} \}$$

Tipi di Dato

Esempio:

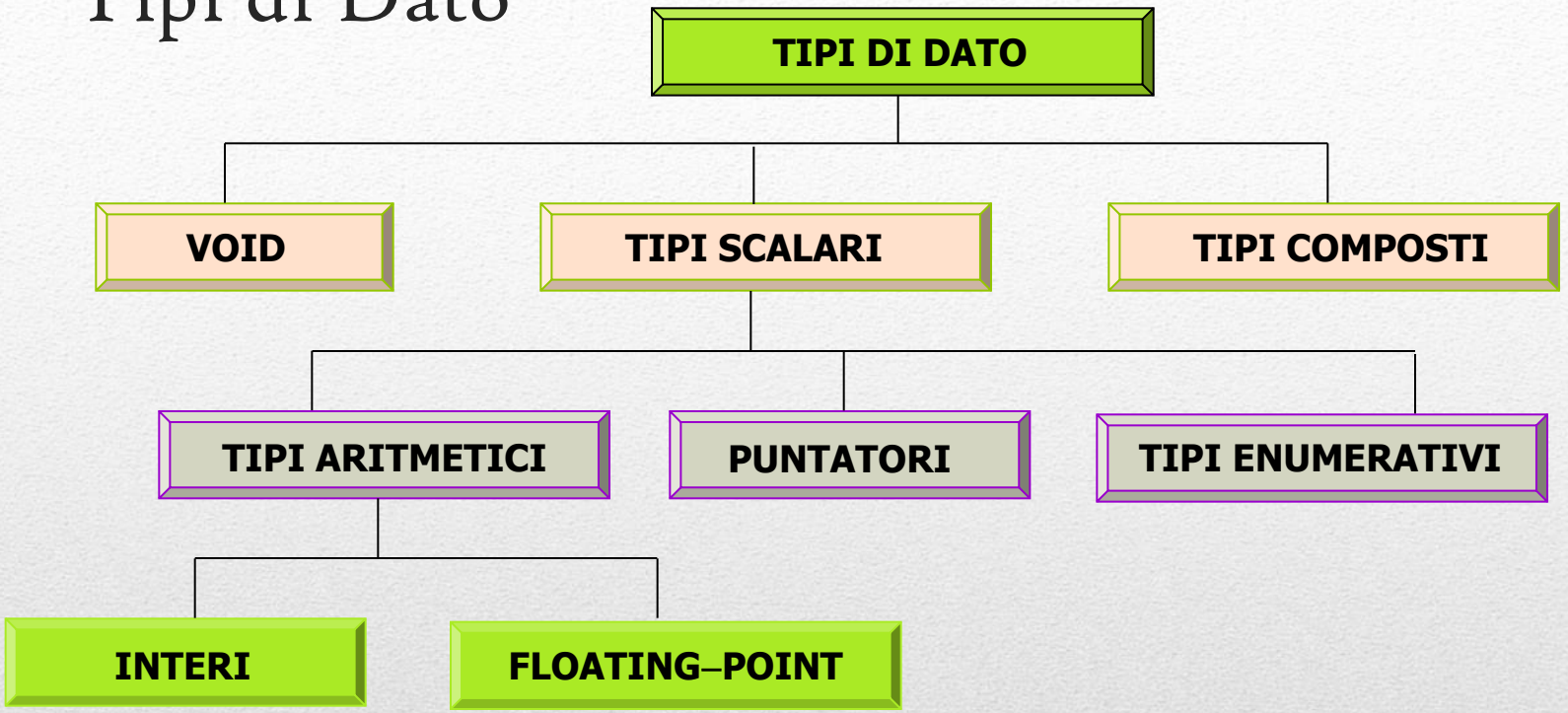
il tipo di dato intero è definito come:

- un dominio di valori, D
- un insieme di funzioni F1,..,Fn sul dominio D

  esempio SOMMA, SOTTRAZIONE, PRODOTTO ….
- un insieme di predicati P1,..,Pm sul dominio D

  ad esempio MAGGIORE, MINORE, UGUALE...

Tipi di Dato



I tipi **aritmetici**, i **tipi enumerativi** ed i **puntatori** vengono detti **tipi scalari**, poiché i valori che li compongono sono distribuiti su una *scala lineare*, su cui si può stabilire una relazione di ordine totale

Combinando i tipi scalari si ottengono i **tipi composti** che comprendono array, strutture ed unioni: servono per raggruppare variabili logicamente correlate in insiemi di locazioni di memoria fisicamente adiacenti

Tipi di Dato Primitivi in C

- 4 tipi di dato primitivi
  - char (caratteri)
  - int (interi)
  - float (reali)
  - double (reali in doppia precisione)
- 4 qualificatori di tipo
  - signed
  - unsigned
  - short
  - long
- signed e unsigned possono essere applicati solo ai tipi char e int
- short può essere applicato solo a int
- long può essere applicato solo a int e a double

Tipi semplici predefiniti

- char
- signed char
- unsigned char

- float
- double
- long double

- [signed] short int
- [signed] int
- [signed] long int

- unsigned short int
- unsigned int
- unsigned long int

I qualificatori **short** e **long** condizionano lo **spazio** allocato dal compilatore per la memorizzazione delle variabili (lo spazio effettivamente utilizzato dipende dalla macchina)
I qualificatori **signed**, **unsigned**, **short** e **long** (nelle combinazioni possibili) condizionano l'insieme dei **valori** che la variabile può assumere nonché il suo valore minimo e massimo.

Tipi di Dato Primitivi in C

- caratteri
  - char caratteri ASCII
- interi con segno
  - short (int) -32768 ... 32767
  - int ???????
  - long (int) -2147483648 2147483647
- naturali (interi senza segno)
  - unsigned short (int) 0 ... 65535
  - unsigned (int) ???????
  - unsigned long (int) 0 ... 4294967295

ATTENZIONE: la dimensione di int non è fissa, dipende dal compilatore considerato. Unica certezza: short ≤ int ≤ long

… lo stesso vale per l'unsigned int

Per i caratteri alfanumerici (caratteri alfabetici, simboli di interpunzione, cifre numeriche ed altri simboli ancora) esiste una tabella di codifica standard che associa ad ogni carattere un codice numerico intero: la codifica **ASCII** (American Standard Code for Information Interchange). Ad esempio il carattere "a" è associato al codice 61, al carattere "b" il 62, e così via.

Tipi di Dato Primitivi in C

- reali
  - float singola precisione (32 bit)
  - double doppia precisione (64 bit)
- boolean
  - non esistono in C come tipo a sé stante
  - si usano gli interi:
    - zero indica FALSO
    - ogni altro valore indica VERO
  - convenzione: suggerito utilizzare uno per VERO

Informazioni numeriche

La memoria del calcolatore è un "casellario" molto grande suddiviso in locazioni di memoria, numerate progressivamente mediante degli indirizzi di memoria che ne identificano univocamente la posizione.

Ogni locazione è composta da un insieme di 8 bit che compongono un **byte**.

Con un solo byte è possibile rappresentare pochi numeri interi (compresi tra $0_{10} = 00000000_2$ e $255_{10} = 11111111_2$).

Per rappresentare numeri più grandi il calcolatore deve aggregare più locazioni di memoria contigue.

Tipi di dato: char

Al fine di attribuire significato ad una sequenza di bit occorre sapere quanti bit la compongono, e qual è la loro organizzazione al suo interno. La più ristretta sequenza di bit significativa per le macchine è il **byte**. In C, al byte corrisponde il tipo di dato **char**. Esso può assumere 256 valori diversi (2^8 = 256).

I valori del tipo char possono cambiare da computer a computer a causa del fatto che le varie macchine possono basarsi su un diverso set di caratteri. Attualmente, il set di caratteri più diffuso è quello ASCII

In C, la differenza tra carattere e numero è sfumata: il tipo di dati char è un valore intero rappresentato con un byte, che può essere utilizzato per memorizzare sia caratteri che interi

Si distinguono due tipi di character: il signed character, in cui l'ottavo bit funge da indicatore di segno (se è 1 il valore è negativo), e l'unsigned character, che utilizza invece tutti gli 8 bit per esprimere il valore, e può dunque esclusivamente assumere valori positivi. Un signed char può variare tra -128 e 127, mentre un unsigned char può esprimere valori tra 0 e 255.

Stringhe

Una stringa è una collezione di caratteri delimitata da virgolette, es:

"ciao" "Hello\n"

Le abbiamo usate anche come argomento della funzione printf. Nel dettaglio, in C vengono memorizzate come vettori (sequenze) di caratteri di cui l'ultimo, sempre presente in modo implicito, è il **carattere null** '\0'

"ciao" = {'c', 'i', 'a', 'o', '\0'}

Per una stringa di lunghezza n il compilatore alloca n+1 byte di memoria; in questo caso 5.

Impareremo che il vettore è trattato come un puntatore di tipo *char* *. Es.:

printf("abc");

Quando la printf viene invocata, le viene passato l'indirizzo di "abc" (un puntatore alla locazione di memoria che contiene la lettere *a*)

Tipi di dato: int

La sequenza di bit di ampiezza immediatamente superiore al byte è detta **word**. La dimensione della word dipende dal microprocessore che questa utilizza, e può essere, di 16, 32 o 64 bit.

E' di 16 bit su tutte le macchine MS-DOS o con versioni di Windows a 16 bit. E' di 4 byte (32 bit) su tutte le macchine con sistema operativo a 32 bit o 8 byte (64 bit).

Il tipo di dato C corrispondente alla word è l'**integer (int)**. Anche l'integer può essere signed o unsigned.

Per esprimere valori interi di notevole entità il C definisce il **long int**, che può essere signed o unsigned. Le variabili di tipo intero sono quindi:

short int
unsigned short int
int
unisgned int
long int
unsigned long int

Tipi di dato: int

Gli intervalli riportati in tabella NON sono definiti dallo standard C e possono **variare** da un compilatore all'altro. Ad esempio:

Tipo	**Dimensione**	**Range**
unsigned char	8 bit	da 0 a 255
char	8 bit	da -128 a 127
unsiged short	16 bit	da 0 a 65535
short int	16 bit	da -32768 a 32767
unsigned long	32 bit	da 0 a 4294976295
long int	32 bit	da -2147483648 a 2147483647

Tipi di dato: float

I numeri in virgola mobile (**floating-point**) vengono memorizzati in notazione esponenziale, riservando un certo numero di bit per la **mantissa** ed un altro per **l'esponente**. Le rispettive quantità e posizioni possono variare da sistema a sistema (consultare la documentazione del compilatore)

Il C fornisce tre **tipi floating point**:

float	floating point a singola precisione
double	floating point a doppia precisione
long double	floating point con precisione estesa

Il floating point può occupare 32 bit ed offre 7 cifre significative di precisione. I suoi valori hanno range 3.4E +/- 38

Il double precision può occupare 64 bit con 15 cifre di precisione. I suoi valori hanno range 1.7E +/- 308

Il long double precision 80 bit con 19 cifre di precisione. I suoi valori hanno range 1.2E +/- 4932

Tutti i tipi in virgola mobile sono dotati di segno.

Tipi di dato: float

Gli intervalli riportati in tabella NON sono definiti dallo standard C e possono **variare** da un compilatore all'altro. Ad esempio:

Tipo	**Dimensione**	**Range**
long	32 bit	da 0.29E-38 a 1.7E38
double	64 bit	da 0.29E-38 a 1.7E38

Tipi di dato: void

Il tipo di dato void è utilizzabile per esprimere l'assenza di dati o per evitare di specificare a quale tipo, tra quelli appena descritti, appartenga il dato.

Ha due funzioni: serve ad indicare che una **funzione** non restituisce nessun valore e serve per definire un **puntatore** che punta ad un dato generico

Può quindi essere utilizzato esclusivamente per dichiarare puntatori void e funzioni void.

Tipi di dato: ... e il boolean?

In C non esiste un tipo di dato boolean, cioè un dato che può assumere solo due valori (vero o falso, 0 o 1).

Come variabili di tipo booleano si possono utilizzare variabili char, int, long int sia signed che unsigned.

Quando questi dati vengono valutati per esprimere un valore di verità, sono considerati FALSO quando assumono valore 0 e VERO se assumono un qualsiasi valore diverso da 0.

Limiti nei tipi interi

I compilatori generalmente mantengono all'interno dello header file **<limits.h>** l'indicazione del range di valori rappresentabili nei tipi interi. Ad esempio:

SHRT_MAX 32.767

SHRT_MIN -32.768

USHRT_MAX 65.535

INT_MAX 32.767

INT_MIN -32.768

UINT_MAX 65.535

LONG_MAX 2.147.483.647

LONG_MIN -2.147.483.648

ULONG_MAX 4.294.967.295

(vedi programmi di stampa del range dei valori assunti dal tipo intero e dal tipo float più avanti)

... e quando si supera il range di memorizzazione?

Un dato in memoria ha a disposizione un certa dimensione finita e dunque esisterà un limite superiore ed uno inferiore. Es.:

```
signed char MyChar;   // [-128,127]

MyChar=130;
```

Questa sarà un condizione di overflow, e il numero risultante (che potrà essere uno contenuto nel range indicato) **potrebbe essere** -126 dato che dopo 127 il range e' terminato e riprende in maniera circolare da -128 (128), poi -127 (129) e -126 (130).

E' dunque di fondamentale importanza mettersi al sicuro considerando i limiti dei valori che si andranno ad usare mediante l'analisi preliminare dei requisiti richiesti dal programma, perchè in caso di overflow il **comportamento del programma non è definito**.

IL SISTEMA OPERATIVO - CENNI

«**Informatica: arte e mestiere**»

Ceri, Mandrioli, Sbattella, Mc Graw-Hill

cap.13

Cos'è un sistema operativo

Il software può essere diviso in due grandi classi:

- i **programmi di sistema**, che gestiscono le funzionalità del sistema di calcolo
- i **programmi applicativi**, che risolvono i problemi degli utenti

L'insieme dei **programmi di sistema** viene comunemente identificato con il nome di **Sistema Operativo** (SO)

Definizione

Un sistema operativo è un programma che **controlla l'esecuzione di programmi applicativi** ed agisce come **interfaccia fra le applicazioni e l'hardware** del calcolatore

Cos'è un sistema operativo

Tutte le piattaforme hardware/software richiedono un sistema operativo

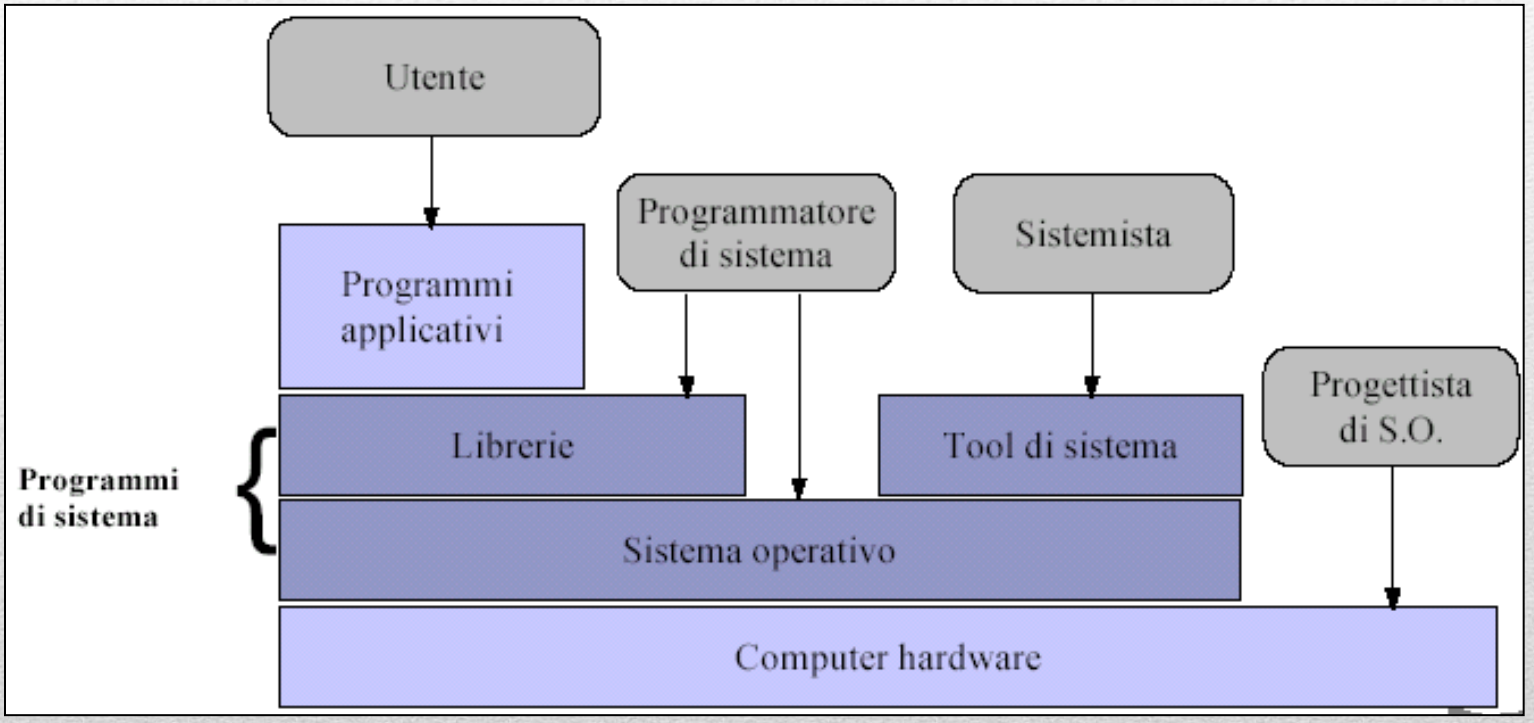
Quando si accende un elaboratore, occorre attendere alcuni istanti per poter iniziare a lavorare: durante questa pausa il computer carica il SO

Il sistema operativo deve:

- Garantire una gestione **EFFICIENTE** delle risorse del sistema di elaborazione
- Rendere **AGEVOLE** l'interfaccia tra l'uomo e la macchina

Il s.o. come macchina estesa

Visione a strati delle componenti hardware/software che compongo un sistema di elaborazione



Il s.o. come macchina estesa

Il SO può essere inteso come uno strumento che virtualizza le caratteristiche dell'hardware sottostante, offrendo all'utente la visione di una macchina astratta più potente e più semplice da utilizzare di quella fisicamente disponibile

In questa visione, un SO…

- …nasconde a programmatori/utenti i dettagli dell'hardware e fornisce un'interfaccia conveniente e facile da usare
- …agisce come intermediario tra programmatore/utente e hardware

Parole chiave

- Indipendenza dall'hardware
- Comodità d'uso
- Programmabilità

Il s.o. come macchina estesa

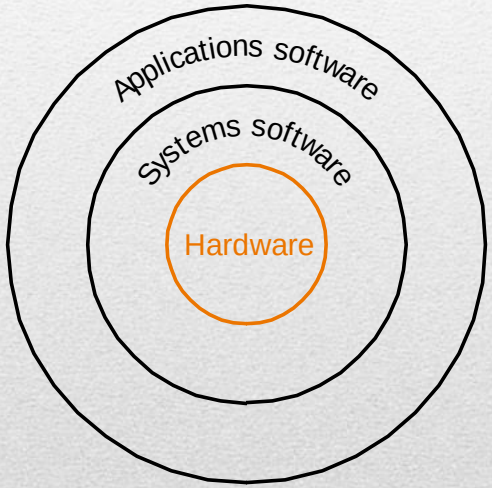
L'utente è in grado di utilizzare la macchina fisica senza conoscere i dettagli della sua struttura interna e del suo funzionamento

Hardware — fornisce le risorse fondamentali di calcolo (CPU, memoria, device di I/O)

Sistema Operativo — controlla e coordina l'utilizzo delle risorse hardware da parte dei programmi applicativi dell'utente

Programmi Applicativi — definiscono le modalità di utilizzo delle risorse del sistema, per risolvere i problemi di calcolo degli utenti (compilatori, database, video game, programmi gestionali)

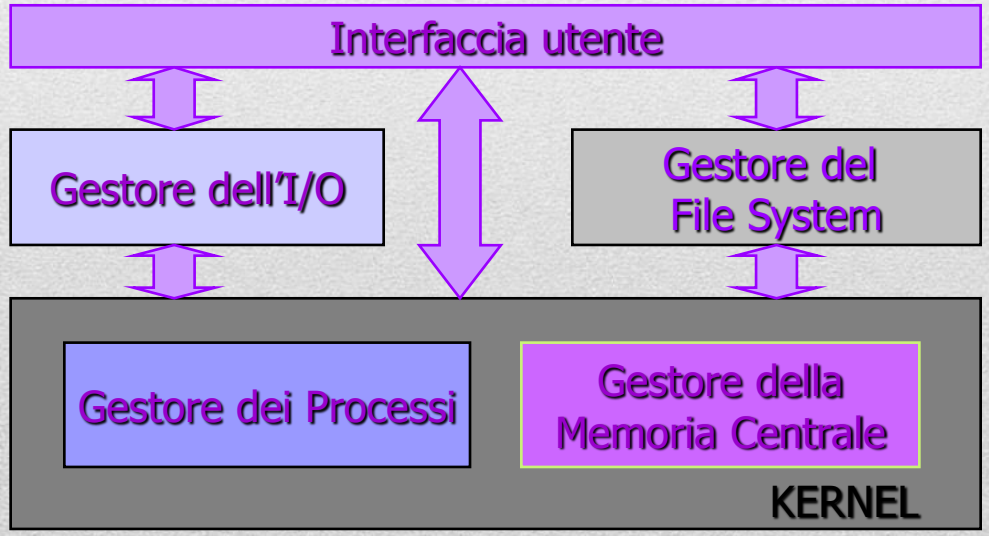
Utenti — persone, altri macchinari, altri elaboratori



Visione "a cipolla" del sistema di calcolo

Architettura del s.o.

- I SO sono costituiti da un insieme di moduli, ciascuno dedicato a svolgere una determinata funzione
- I vari moduli del SO interagiscono tra loro secondo regole precise, al fine di realizzare le funzionalità di base della macchina



L'insieme dei moduli per la gestione della CPU e della memoria centrale è il **kernel**

Riassumendo

Il sistema operativo fornisce un ambiente per eseguire programmi in modo conveniente ed efficiente; funge infatti da…

- Allocatore di risorse — controlla, distribuisce ed alloca le risorse (in modo equo ed efficiente)
- Programma di controllo — controlla l'esecuzione dei programmi utente e le operazioni sui dispositivi di I/O

Esempio: il filesystem

Si ha a che fare con file, directory, etc., e non ci si deve preoccupare di come i dati sono memorizzati sul disco

Compiti del s.o.

- Gestione dei processi
- **Gestione della memoria principale**
- Gestione della memoria di massa (file system)
- Realizzazione dell'interfaccia utente
- Protezione e sicurezza

La gestione della memoria principale

La memoria principale:

- è un “array” di byte indirizzabili singolarmente
- è un deposito di dati facilmente accessibile e condiviso tra la CPU ed i dispositivi di I/O

Il SO è responsabile delle seguenti attività riguardanti la gestione della memoria principale:

- Tenere traccia di quali parti della memoria sono usate e da chi
- Decidere quali processi caricare quando diventa disponibile spazio in memoria
- Allocare e deallocare lo spazio di memoria quando necessario

Il gestore di memoria “realizza” una macchina virtuale in cui ciascun programma opera come se avesse a disposizione una memoria dedicata

La gestione della memoria principale

L'organizzazione e la gestione della memoria centrale è uno degli aspetti più critici nel disegno di un SO

Il gestore della memoria è quel modulo del SO incaricato di assegnare la memoria ai task (per eseguire un task è necessario che il suo codice sia caricato in memoria)

La complessità del gestore della memoria dipende dal tipo di SO

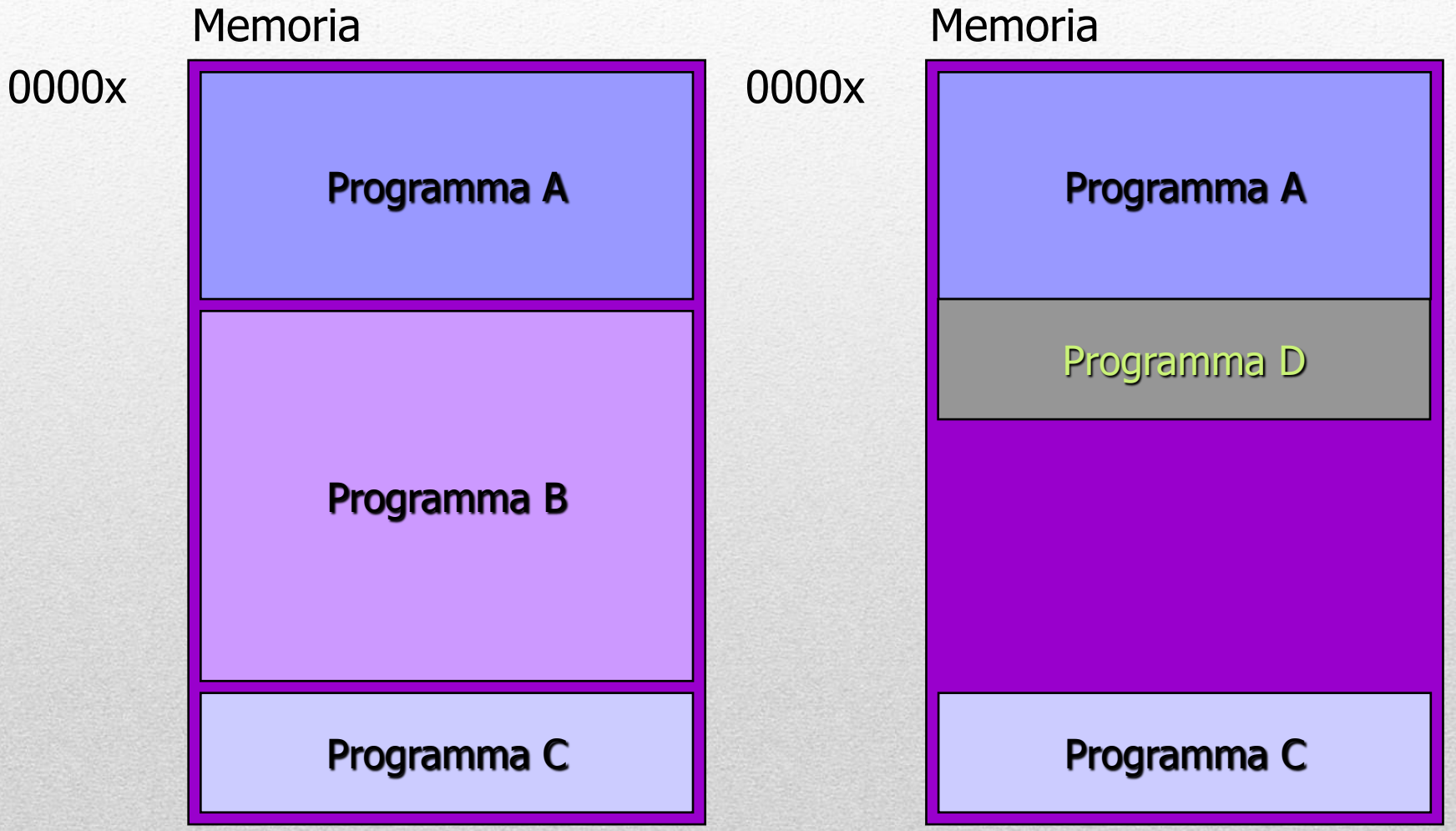
Nei SO multi–tasking, più programmi possono essere caricati contemporaneamente in memoria

Problema: allocare lo spazio in maniera ottimale

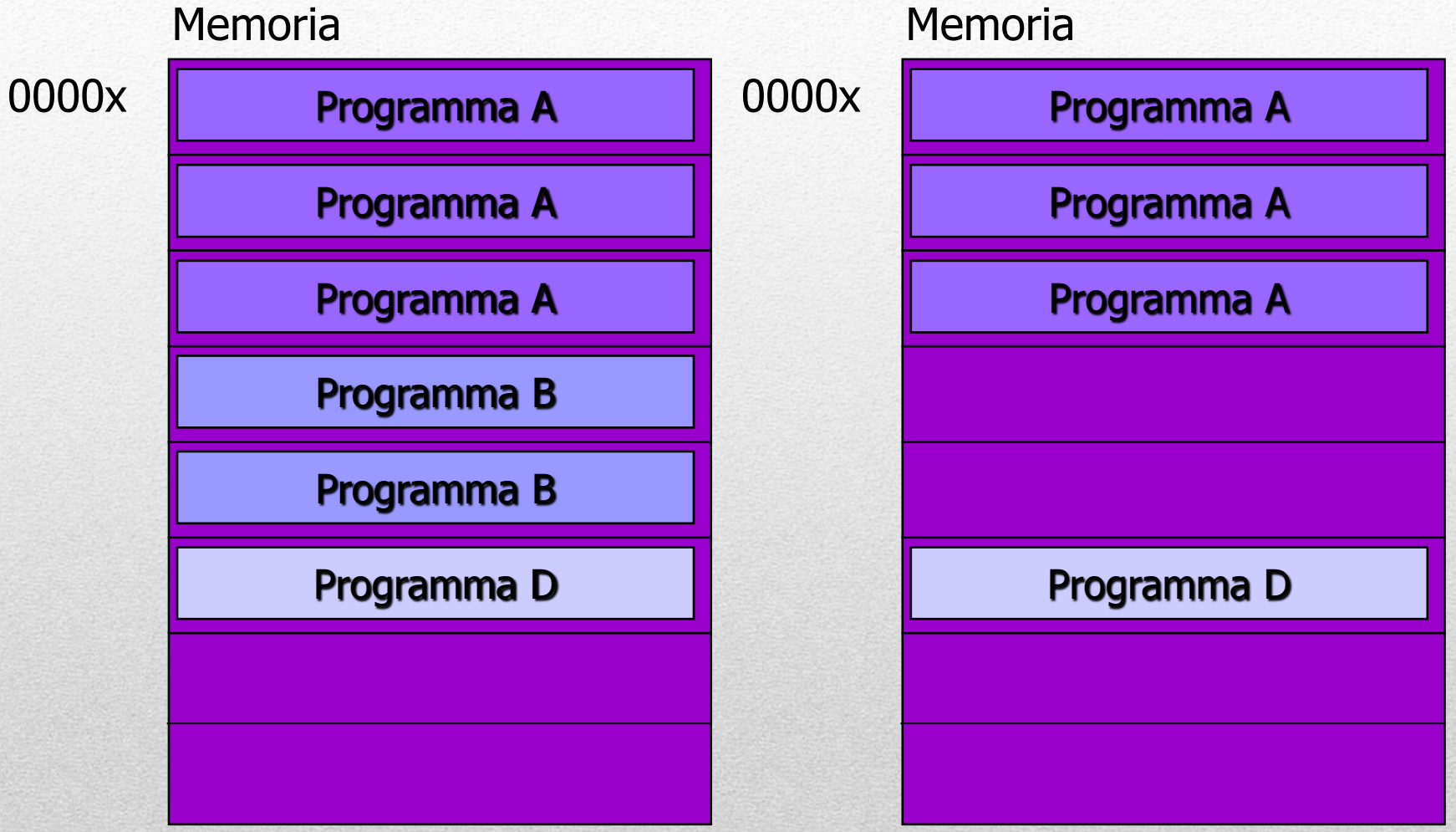
Allocazione lineare

PROBLEMA !!!!

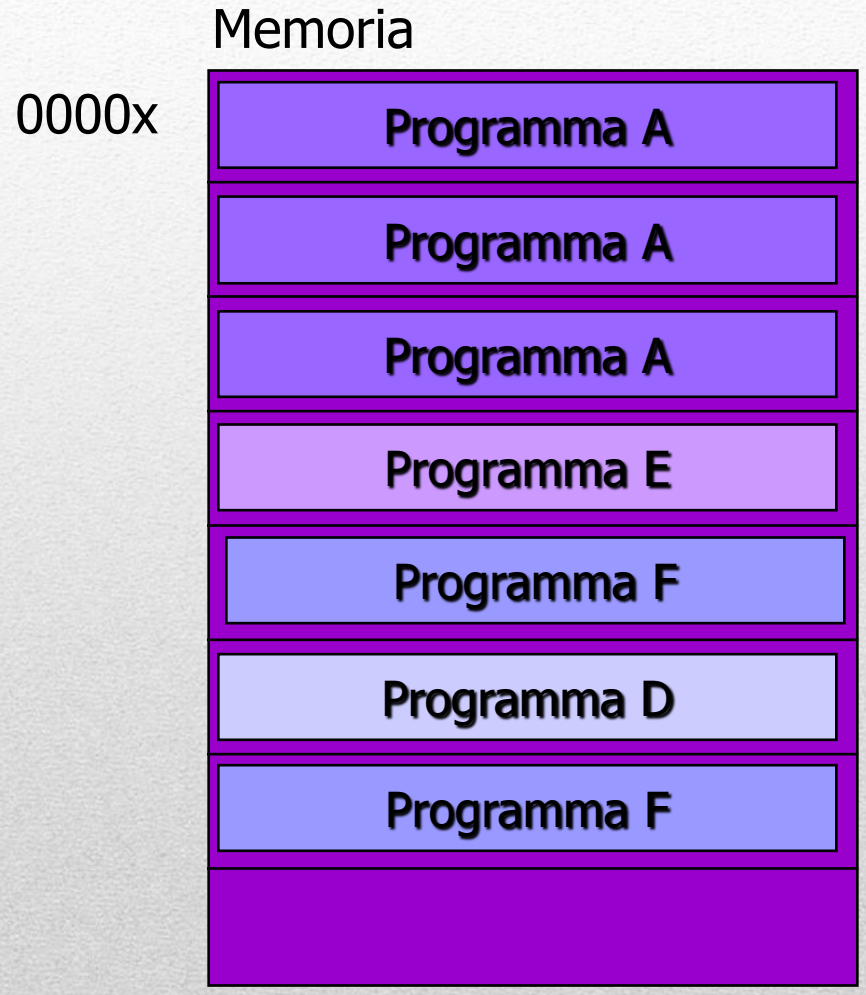
FRAMMENTAZIONE



Paginazione



Paginazione



La memoria virtuale

Spesso la memoria non è sufficiente per contenere completamente tutto il codice dei processi

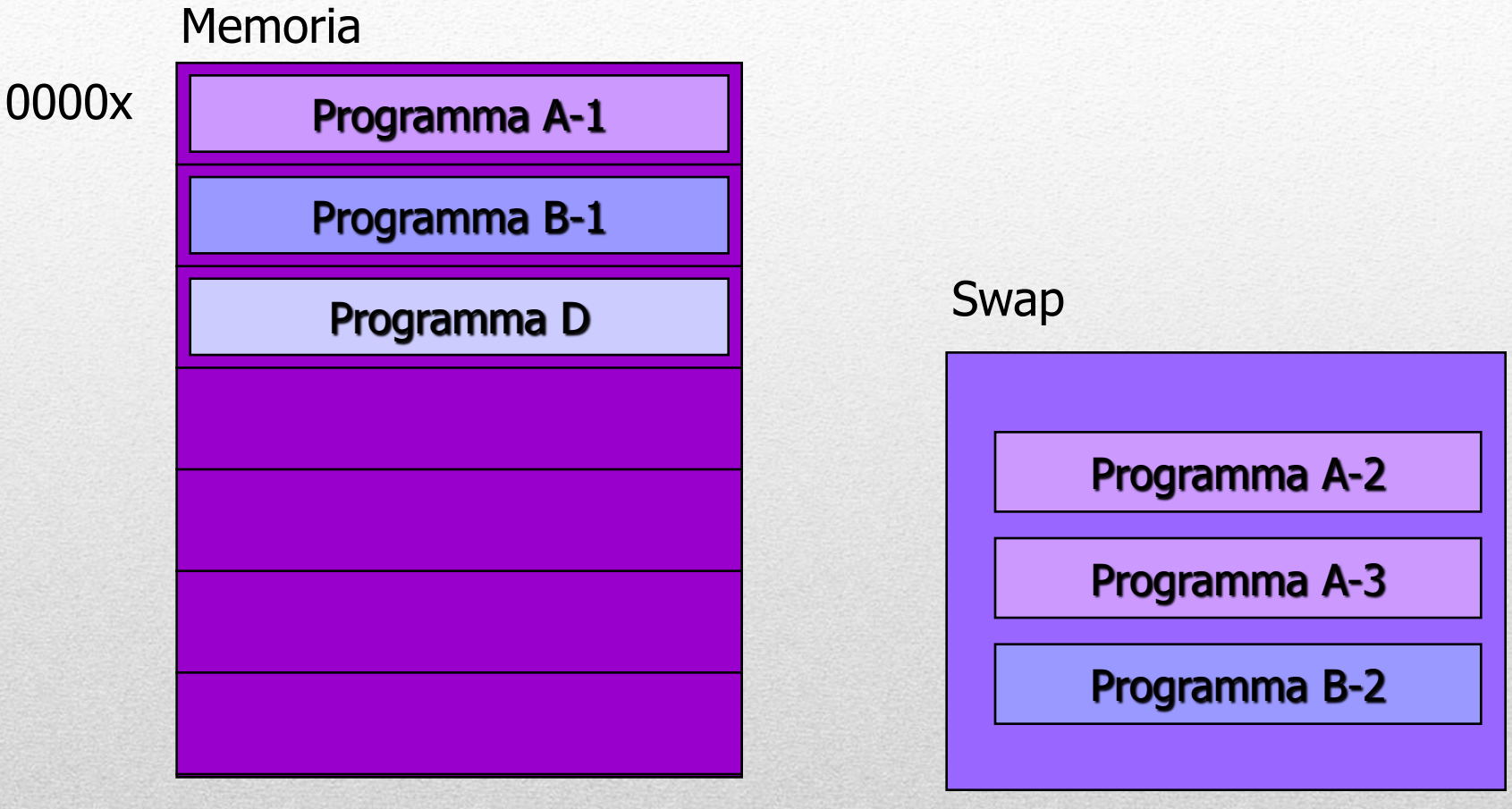
Si può **simulare** una memoria più grande tenendo nella memoria di sistema (RAM) solo le parti di codice e dati che servono in quel momento

Si usa il concetto di **memoria virtuale**

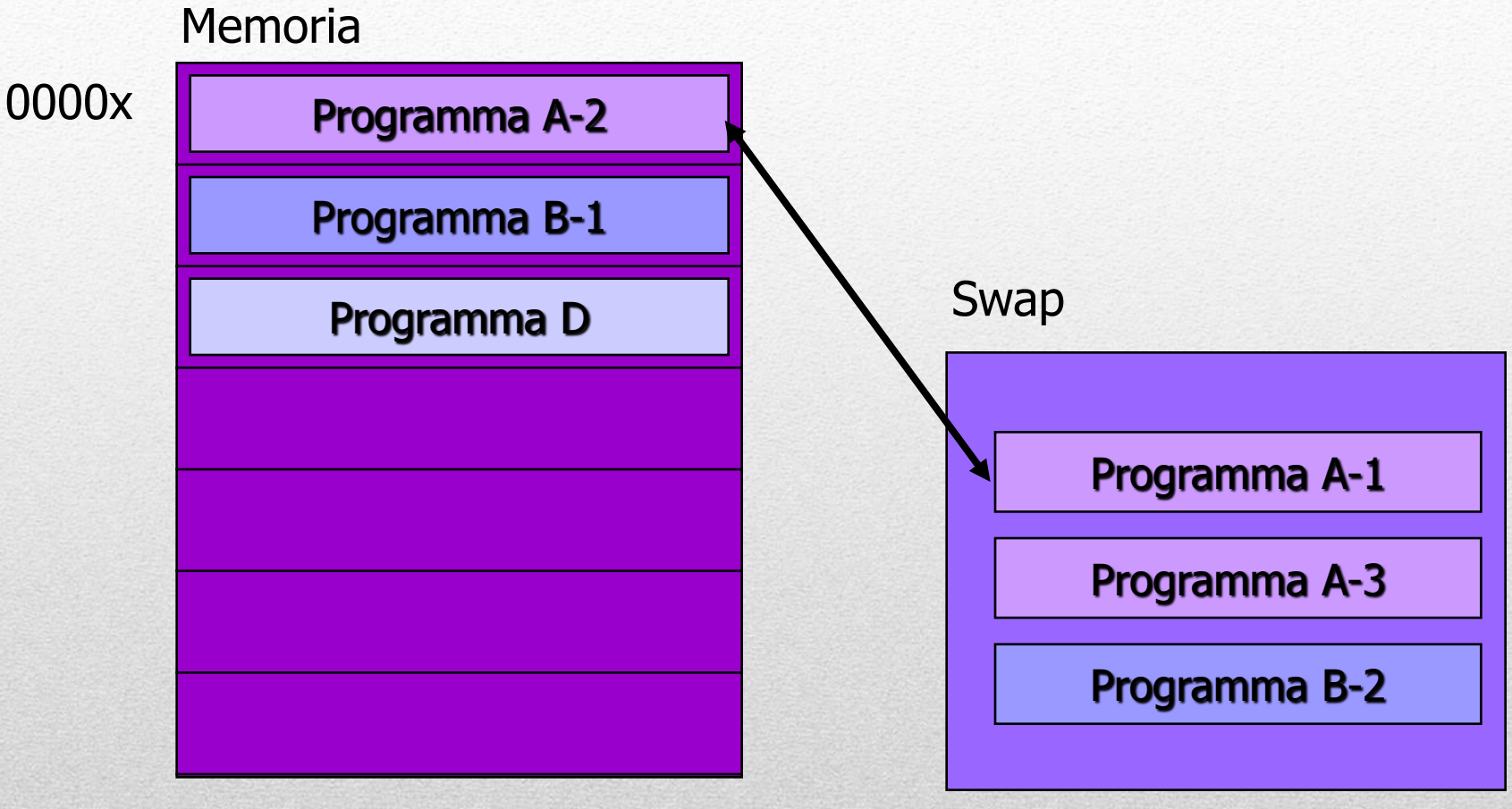
I dati e le parti di codice relativi a programmi non in esecuzione possono essere tolti dalla memoria centrale e "parcheggiati" su disco nella cosiddetta area di swap

I processori moderni sono dotati di meccanismi hardware per facilitare la gestione della memoria virtuale

La memoria virtuale



La memoria virtuale



La gestione della memoria secondaria

Poiché la memoria principale è volatile e troppo piccola per contenere tutti i dati e tutti i programmi in modo permanente, un computer è dotato di memoria secondaria

- In generale, la memoria secondaria è data da hard disk e dischi ottici

Il SO garantisce una visione logica uniforme del processo di memorizzazione:

- Astrae dalle caratteristiche fisiche dei dispositivi per definire un'unità di memorizzazione logica – il *file*
- Ciascuna periferica viene controllata dal relativo device driver, che nasconde all'utente le caratteristiche fisiche variabili dell'hardware: modalità e velocità di accesso, capacità, velocità di trasferimento

La gestione della memoria secondaria

Il SO è responsabile delle seguenti attività riguardanti la gestione della memoria secondaria:

- Allocazione dello spazio
- Gestione dello spazio libero
- Ordinamento efficiente delle richieste di accesso al disco (disk scheduling)

LE VARIABILI

Manuale linguaggio C

Le variabili

Una variabile è un'astrazione della cella di memoria

Le variabili sono dei "contenitori" in memoria capaci di immagazzinare tipi diversi di dati (numeri interi, reali, caratteri, …).

Costituiscono la struttura di dati più semplice del linguaggio su cui agiscono operatori e funzioni.

tipo nome

```
int numero_studenti, i;
float media_voti;
char ch;

int j=14;

i=200;
numero_studenti=i;
media_voti=25.45;
ch='a';
ch=65;
```

dichiarazioni

dichiarazione + assegnamento

assegnamenti

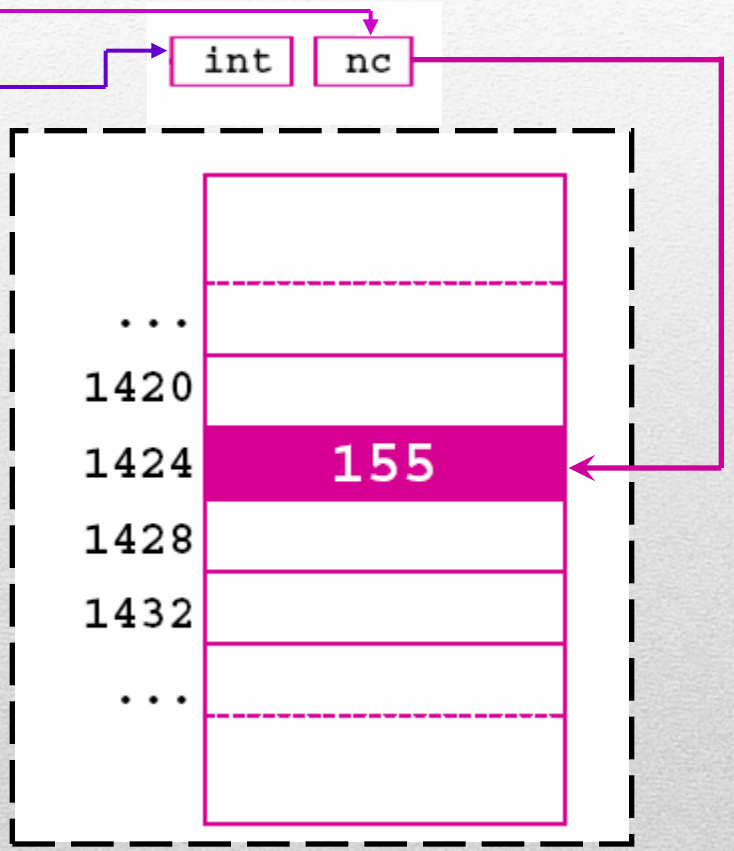
Variabili scalari

Una variabile scalare:

- Ha un **nome**
- Ha un **tipo**
  - Numero intero
  - Numero reale
  - Carattere
  - ...

Corrisponde ad un'area di memoria di dimensione adatta

Contiene un dato semplice



Variabili complesse

Variabili composte

Collezioni di dati omogenei

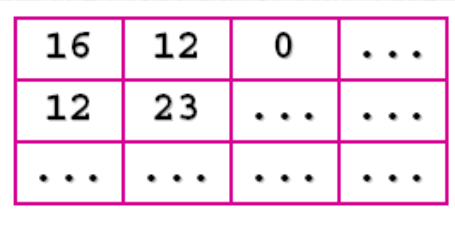
- Vettori
- Matrici

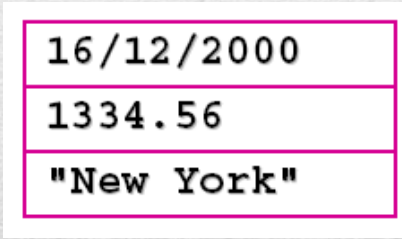
Collezioni di dati eterogenei

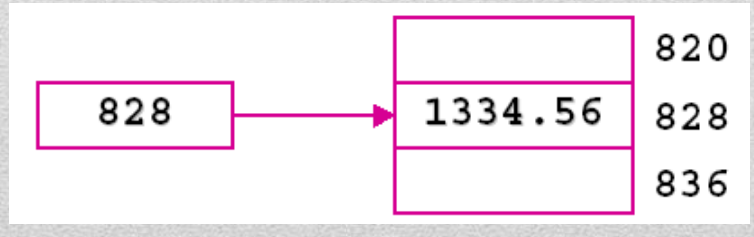
- Strutture (record)
- Unioni

Riferimenti ai dati

- Puntatori







La dichiarazione delle variabili

Il C consente di individuare una certa area di memoria mediante un nome arbitrario che le viene attribuito con un'operazione detta **definizione della variabile**. La variabile è l'area di RAM così identificata.

Ogni riferimento al nome della variabile è in realtà un riferimento al valore in essa contenuto. Si noti che nella definizione della variabile viene specificato il tipo di dato associato a quel nome (e dunque contenuto nella variabile).

(Random Acces Memory, Memoria ad Accesso Casuale)
Memoria nella quale i dati vengono letti e scritti in ordine sparso o meglio non sequenzialmente.
I dati in memoria vengono "scritti" in formato binario cioè in sequenze di 0 o 1 e vengono impressi con un impulso elettrico; i dati risiedono nella RAM fino a che è presente una tensione (la corrente del computer); quando la tensione viene a mancare (spegnimento) i dati in memoria vengono persi

In tal modo il programmatore può gestire i dati in RAM senza conoscerne la posizione e senza preoccuparsi (entro certi limiti) della loro dimensione in bit e dell'organizzazione interna dei bit, cioè del significato che ciascuno di essi assume nell'area di memoria assegnata alla variabile.

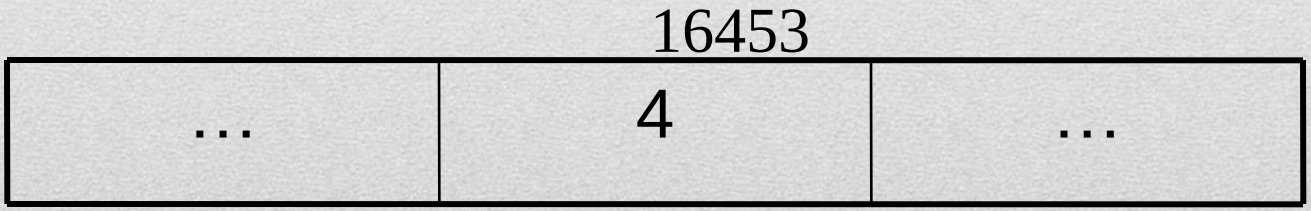
Le variabili

Una variabile è un'astrazione della cella di memoria.

Formalmente, è un simbolo associato a un indirizzo fisico (L-value)...

simbolo	indirizzo
x	16453

Perciò, l' L-value di x è 16453 (fisso e immutabile!) che denota un valore (R-value).



..e l' R-value di x è attualmente 4 (può cambiare).

I nomi delle variabili

Non è possibile dichiarare più variabili con lo stesso nome in una medesima funzione (ma in funzioni diverse sì). A rendere differente il nome è sufficiente una diversa disposizione di maiuscole e minuscole.

I nomi delle variabili devono cominciare con una lettera dell'alfabeto o con l'underscore ("_") e possono contenere numeri, lettere e underscore.

La loro lunghezza massima varia a seconda del compilatore; le implementazioni commerciali più diffuse ammettono nomi composti di oltre 32 caratteri.

Non sono ammessi caratteri di punteggiatura o altro, che hanno significati speciali.

Non sono ammessi nomi uguali a parole chiave riservate del C o a nomi di variabili o a funzioni delle librerie usate.

I nomi delle variabili

Esempio – Nomi di variabile corretti:

j
j5
_system_name
variable_name
NoMe_CoN_lEtTeRe_MiNuScOlE_e_MaIuScOlE

Esempio – Nomi di variabile scorretti:

5j	i nomi non possono iniziare con una cifra
$name	i nomi non possono contenere il simbolo $
int	int è una parola riservata
bad%#*name	i nomi non possono contenere nessun carattere speciale eccetto "_"

Le variabili

Tutte le variabili devono essere dichiarate prima di poter essere utilizzate.

Per aumentare la concisione, è possibile dichiarare variabili dello stesso tipo separando i loro nomi con virgole.

```
…
int voto_scritto, voto_orale;
float media_voti;
…
```

All'interno di un blocco, tutte le dichiarazioni devono apparire prima delle istruzioni eseguibili; l'ordine relativo delle dichiarazioni non è significativo

Dichiarazione e inizializzazione

Una dichiarazione consente di allocare la memoria necessaria per una variabile, ma alla variabile non viene automaticamente associato nessun valore:

se il nome di una variabile viene utilizzato prima che sia stata eseguita un'assegnazione esplicita, il risultato non è prevedibile

Esempio:

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

int main()
{
   int m;

   printf("Il valore di m è: %d\n", m);
   return 0;
}
```

Il risultato del programma non è "certo": *m* assume il valore lasciato nella locazione di memoria dall'esecuzione di un programma precedente

Dichiarazione e inizializzazione

```
int main(void)
{
  int i, j, k;
  float x, y, z;

  printf("Value of i: %d\n", i);
  printf("Value of j: %d\n", j);
  printf("Value of k: %d\n", k);

  printf("Value of x: %g\n", x);
  printf("Value of y: %g\n", y);
  printf("Value of z: %g\n", z);

  return 0;
}
```

Se compilato ed eseguito, potrebbe visualizzare:

Value of i: 5618848

Value of j: 0

Value of k: 6844404

Value of x: 3.98979e-34

Value of y: 9.59105e-39

Value of z: 9.59105e-39

I valori stampati dipendono da molteplici fattori, quindi la probabilità che otteniate esattamente questi numeri è bassa.

Lo specificatore %g stampa un valore a virgola mobile; a differenza di %f non verranno visualizzati zeri aggiuntivi, inoltre, se il valore che deve essere stampato non ha cifre dopo la virgola, %g non stampa il separatore decimale. Utile quindi per visualizzare numeri la cui dimensione non può essere predetta durante la scrittura del programma oppure tende a variare considerevolmente per dimensione. %g utilizza il formato a virgola fissa per stampare numeri non troppo piccoli o grandi, altrimetni passa al formato esponenziale

La funzione printf() (vedi laboratorio)

La funzione *printf()* può avere un numero variabile di argomenti

Il primo argomento è un parametro speciale, detto stringa di formato, che specifica il numero di argomenti che contengono i dati da stampare e le modalità di formattazione dei dati

La stringa di formato è racchiusa fra doppi apici e può contenere testo e specificatori di formato — sequenze speciali di caratteri che iniziano con il simbolo di percentuale (%) ed indicano le modalità di scrittura di un singolo dato

Esempio: nell'istruzione

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

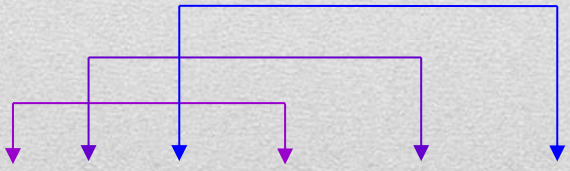
- "Il valore di num è %d" è la stringa di formato
- %d è lo specificatore di formato per gli interi decimali
- num è la variabile intera decimale da stampare

La funzione printf()

Esistono altri specificatori per altri tipi di dati:

%c	dato di tipo carattere
%f	dato di tipo floating–point
%s	array di caratteri terminato da null (stringa)
%o	intero ottale
%x	intero esadecimale

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



```
printf("Stampa tre valori: %d %d %d",num1,num2,num3);
```

La funzione printf()

I dati da stampare possono essere espressioni

```
printf("Il quadrato di %d è %d\n", num, num*num);
```

Il simbolo speciale \n è una sequenza di escape

Quando le sequenze di escape sono inviate ad un dispositivo di uscita sono interpretate come segnali che controllano il formato della visualizzazione

\n forza il sistema ad effettuare un ritorno a capo (newline)

La funzione scanf() (vedi laboratorio)

La funzione *scanf()* legge dati introdotti da tastiera

scanf() può ricevere un numero qualunque di parametri preceduti da una stringa di formato

I parametri di *scanf()* devono essere *lvalue*, e devono pertanto essere preceduti dall'operatore indirizzo &

Esempio:

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

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

```
1  /      /* SOMMA DI DUE NUMERI INTERI */
2
3  #include <stdio.h>
4
5  int main()
6  {
7     int integer1, integer2, sum;         /* declaration */
8
9     printf( "Enter first integer\n" );   /* prompt */
10    scanf( "%d", &integer1 );            /* read an integer */
11    printf( "Enter second integer\n" );  /* prompt */
12    scanf( "%d", &integer2 );            /* read an integer */
13    sum = integer1 + integer2;           /* assignment of sum */
14    printf( "Sum is %d\n", sum );        /* print sum */
15
16    return 0;  /* indicate that program ended successfully */
17 }
```

1. Dichiarazione di tipo
2. Input dei dati
3. Calcolo della somma
4. Stampa dei risultati
5. Output del programma

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Commenti al programma

- Già visti:
  - Commenti, **`#include <stdio.h>`** e **`main`**
- **`int integer1, integer2, sum;`**
  - Dichiarazione di variabili
    - Variabili: locazioni di memoria dove vengono memorizzati dei valori
    - **`int`** significa che la variabile può contenere valori interi (es.: **`-1`**, **`3`**, **`0`**, **`47`**)
    - **`integer1`**, **`integer2`**, **`sum`** Nomi di variabili (identificatori)
      - **Identificatori**: consistono di lettere, numeri (non possono iniziare con un numero), trattini; sono "case sensitive"
    - Le dichiarazioni devono essere poste prima delle istruzioni eseguibili, altrimenti viene segnalato un errore di sintassi dal compilatore

Commenti

- **`scanf( "%d", &integer1 );`**
  - Acquisisce i dati dall'utente
    - **`scanf`** usa l'input standard (in genere la tastiera)
  - Qui **`scanf`** ha due argomenti:
    - **`%d`** – indica che il dato deve essere un intero
    - **`&integer1`** – locazione di memoria dove memorizzare la variabile
    - **`&`** per ora ci basti sapere che si tratta dell'indirizzo di memoria della variabile in cui memorizzare il valore inserito da tastiera
  - L'utente risponde allo **`scanf`** digitando un numero e poi premendo il tasto enter (return)

Commenti

- **=** (operatore di assegnazione)
  - Assegna un valore ad una variabile
  - E' un operatore "binario" (cioè ha due operandi)

    ```
    sum = variable1 + variable2;
    ```

    a **sum** è assegnato il risultato dell'operazione **variable1 + variable2;**
  - La variabile sum che riceve il risultato deve essere sulla sinistra dell'operatore =
- **printf("Sum is %d\n", sum);**
  - Simile a **scanf** - **%d** significa che verrà stampato un valore intero
    - **sum** specifica quale valore verrà stampato
  - I calcoli possono essere effettuati all'interno dell'istruzione **printf**

    ```
    printf( "Sum is %d\n", integer1 + integer2 );
    ```

Le variabili

Le variabili assumono caratteristiche diverse, in particolare rispetto alla loro visibilità (**scope**), cioè in base alla posizione che occupa nel programma la loro dichiarazione.

```
double globale;
int main() {
 float automatica;
 int i=0;
 ….
 return 0;
}
void f (double z) {
 static int i;
 …
}
```

Lo stack di memoria

Lo stack è una sequenza dinamica di word scritte nella memoria RAM tipica dei programmi in esecuzione. L'area di memoria in cui lo stack si sviluppa è una parte di quella affidata al processo stesso.

Con il termine dinamica si intende che il numero di word dipende dall'istruzione che il processore si accinge ad eseguire, in funzione della quale può crescere, rimanere costante o diminuire.

La sua organizzazione è di tipo LIFO

Variabili locali

Le variabili **locali** possono essere dichiarate all'interno di un qualsiasi blocco all'inizio del blocco stesso.

Le variabili locali sono *visibili* e possono essere *utilizzate* solo all'interno del blocco stesso.

Il *ciclo di vita* delle variabili locali inizia nel momento in cui il controllo entra nel blocco, e termina quando il controllo esce dal blocco.

Le variabili locali sono caricate sullo stack ed eliminate in corrispondenza del loro ciclo di vita.

Quando una variabile è dichiarata nel corpo di una funzione, è locale alla funzione e quindi nasce e muore ad ogni chiamata.

Variabili globali

Le variabili **globali** sono quelle dichiarate fuori da tutte le funzioni.

Le variabili globali sono *visibili* e possono essere *utilizzate* da tutte le funzioni che stanno **nello stesso file** dopo la dichiarazione della variabile stessa.

Le variabili globali sono *visibili* e possono essere *utilizzate* da tutte le funzioni che stanno **in altri file** dopo la dichiarazione della variabile se essa è dichiarata come **extern**. (es.: extern tipo NomeVar)

Una dichiarazione *extern* dice al compilatore che la variabile non è dichiarata nel file attuale (dove compare extern) ma in un qualche altro file (che il compilatore cercherà) ma che la variabile può essere utilizzata anche nel file attuale.

La dichiarazione *extern* rappresenta quindi solo un riferimento per il linker e la variabile viene fisicamente allocata solo nel file in cui compare la dichiarazione senza extern.

Variabili statiche

Se vogliamo che una certa variabile *globale* **non** sia utilizzata da funzioni presenti in un **altro file**, dobbiamo modificare la sua dichiarazione facendola precedere dalla parola chiave **static** (es.: static tipo NomeVar)

In tale modo la variabile statica sarà visibile e potrà essere utilizzata solo dalle funzioni del suo file.

Lo specificatore **static** applicato ad una variabile **locale** ordina al compilatore di collocare la variabile non più nello stack all'atto della chiamata alla funzione, ma in una locazione di memoria permanente per tutta la durata del programma, come se fosse una variabile globale. Quindi la variabile locale static:

- viene inizializzata una sola volta (la prima volta che si chiama la funzione)
- mantiene il suo valore anche dopo l'uscita dalla funzione.

Caratteristiche delle variabili

Lo scope (la parte di programma in cui la variabile è nota e può essere manipolata) è:

- in C, Pascal: determinabile staticamente
- in LISP: determinabile dinamicamente

tipo: specifica la classe di valori che la variabile può assumere (e quindi gli operatori applicabili)

tempo di vita: è l'intervallo di tempo in cui rimane valida l'associazione simbolo/indirizzo fisico (L-VALUE)

- in FORTRAN: allocazione statica
- in C, Pascal: allocazione dinamica

valore: è rappresentato (secondo la codifica adottata) nell'area di memoria associata alla variabile

ESPRESSIONI

Manuale linguaggio C

Le espressioni

Espressioni: formule che mostrano come calcolare un valore.

Espressioni semplici: **variabili** (valore che deve essere calcolato) e **costanti** (valore che non verrà modificato)

Espressioni più complicate: applicano degli **operatori** sugli operandi

Alcuni dei più importanti sono: operatori **aritmetici**, operatori **relazionali**, operatori **logici**, di **incremento** e **decremento**.

Operatori aritmetici

Operatore e operandi	Descrizione
++op	Incrementa di un'unità l'operando prima che venga restituito il suo valore.
op++	Incrementa di un'unità l'operando dopo averne restituito il suo valore.
--op	Decrementa di un'unità l'operando prima che venga restituito il suo valore.
op--	Decrementa di un'unità l'operando dopo averne restituito il suo valore.
+op	Non ha alcun effetto (nel K&R non esiste) specifica che una costante numerica è positiva
-op	Inverte il segno dell'operando (prima di restituirne il valore).
op1 + op2	Somma i due operandi.
op1 - op2	Sottrae dal primo il secondo operando.
op1 * op2	Moltiplica i due operandi.
op1 / op2	Divide il primo operando per il secondo (se operandi interi tronca il risultato).
op1 % op2	Calcola il resto della divisione tra il primo e il secondo operando, richiede operandi interi.
var = valore	Assegna alla variabile il valore alla destra.
op1 += op2	op1 = (op1 + op2)
op1 -= op2	op1 = (op1 - op2)
op1 *= op2	op1 = (op1 * op2)
op1 /= op2	op1 = (op1 / op2)
op1 %= op2	op1 = (op1 % op2)

Operatori aritmetici

Gli operatori +, -, *, / ammettono sia operandi interi che float; è ammesso anche mischiare i tipi, nel qual caso il risultato è di tipo float (6.7f / 2 ha valore 3.35)

L'operatore /:

quando entrambi gli operandi sono interi l'operatore tronca il risultato omettendo la parte frazionaria (1/2 è 0 e non 0.5)

L'operatore %:

richiede operandi interi; se anche uno solo non lo è, il programma non verrà compilato.

Utilizzare lo 0 come operando destro di uno dei due opertori / e % provoca un **comportamento non** definito

Comportamento non definito: il programma potrà assumere comportamenti differenti a seconda del compilatore usato. Inoltre, il programma potrebbe non essere compilabile, se venisse compilato, potrebbe non essere eseguibile, e nel caso in cui fosse eseguibile potrebbe andare in crash, comportarsi in modo errato o produrre risultati senza senso. **Comportamento da evitare**

Comportamento definito dall'implementazione

Descrivere il risultato del caso in cui / e % vengono utilizzati con un opeando negativo è complesso:

C89: il risultato di *i/j* può essere arrotondato sia per eccesso che per difetto (-9 / 7 può essere sia -1 che -2)

C89: il risultato di *i%j* dipende dall'implementazione (-9 % 7 può essere sia -2 che 5)

C99: il risultato di *i/j* viene sempre arrotondato verso lo 0 (-9 / 7 è -1)

C99: il risultato di *i%j* ha il segno di i (-9 % 7 è -2)

Comportamento definito dall'implementazione: per favorire l'efficienza (cioè avvicinarsi al comportamento dell'hardware sottostante) lo standard C non specifica deliberatamente alcune parti del linguaggio intendendo lasciare all'implementazione (software necessario su una particolare piattaforma per compilare, fare il linking ed eseguire i programmi) il compito di occuparsi dei dettagli.

Precedenza degli operatori e associatività

Operatore	Descrizione	Associatività
( ) [ ] . -> ++ --	Parentesi (chiamata a funzione) (vedi Nota 1) Parentesi quadre (array) selezione di membri di oggetti dal nome come sopra, ma dal puntatore incrementi/decrementi postfissi (vedi Nota 2)	da sx a dx
++ -- + - ! ~ (*type*) * & sizeof	incrementi/decrementi prefissi più/meno unari negazione logia/complemento bit a bit cast (conversione di tipi) deferenza indirizzo (di operandi) dimensioni in byte (dipendente dalla piattaforma)	da dx a sx
* / %	moltiplicazione/divisione/modulo	da sx a dx
+ -	Aaddizione/sottrazione	da sx a dx
<< >>	shift a sinistra/destra bit a bit	da sx a dx
< <= > >=	operatori relazionali minore di/minore o uguale a operatori relazionali maggiore di/maggiore o uguale a	da sx a dx
== !=	operatore relazionale uguale a/ non uguale a	da sx a dx
&	AND bit a bit	da sx a dx
^	OR esclusivo bit a bit	da sx a dx
|	OR bit a bit	da sx a dx
&&	AND logico	da sx a dx
| |	OR logico	da sx a dx
? :	condizionale ternario	da dx a sx

Precedenza degli operatori e associatività

Operatore	Descrizione	Associatività
= += -= *= /= %= &= ^= |= <<= >>=	assegnazione assegnazione con addizione/sottrazione assegnazione con moltiplicazione/divisione assegnazione con modulo/AND bit a bit assegnazione con OR esclusivo bit a bit/Or bit a bit assegnazione con shift a sinistra/shift a destra bit abit	da dx a sx
,	virgola (separatore di espressioni)	da sx a dx

Esercizio: addizionale

Scrivere un programma che chieda all'utente di inserire un importo in dollari e centesimi e successivamente lo stampi con un addizionale del 5% di tasse:

```
input: Enter an amount: 100.00
output: With tax added: $105.00
```

```
#include <stdio.h>

int main(void)
{
  float original_amount, amount_with_tax;

  printf("Enter an amount: ");
  scanf("%f", &original_amount);
  amount_with_tax = original_amount * 1.05f;
  printf("With tax added: $%.2f\n", amount_with_tax);

  return 0;
}
```

Se il valore calcolato non lo vogliamo memorizzare in una variabile *amount_with_tax* non è necessaria:

```
#include <stdio.h>

int main(void)
{
  float original_amount;

  printf("Enter an amount: ");
  scanf("%f", &original_amount);
  printf("With tax added: $%.2f\n", original_amount * 1.05f);

  return 0;
}
```

Esercizio: calcolare carattere controllo codici a barre

Numero a 12 cifre che identifica sia il produttore che il prodotto:

- Tipologia di prodotto (1 cifra)
- Produttore (5 cifre)
- Prodotto (5 cifre)
- Cifra di controllo (1 cifra)

Algoritmo per il calcolo della cifra di controllo:

1. Sommare la prima, terza, quinta, settima, nona e undicesima cifra
2. Sommare seconda, quarta, sesta, ottava e decima cifra
3. Moltiplicare la prima somma per 3 e sommarla alla seconda somma
4. Sottrarre 1 dal totale
5. Calcolare il resto del totale diviso per 10
6. Sottrarre il resto dal numero 9

Esercizio: calcolare carattere controllo codici a barre

```
/* upc.c  */
/* Computes a Universal Product Code check digit */

#include <stdio.h>
int main(void)
{
  int d, i1, i2, i3, i4, i5, j1, j2, j3, j4, j5,
      first_sum, second_sum, total;

  printf("Enter the first (single) digit: ");                    //immissione cifra a sinistra
  scanf("%1d", &d);                                              // lettura intero su singola cifra
  printf("Enter first group of five digits: ");                  //immissione primo gruppo 5 cifre
  scanf("%1d%1d%1d%1d%1d", &i1, &i2, &i3, &i4, &i5);             // lettura 5 interi su singola cifra
  printf("Enter second group of five digits: ");                 //immissione secondo gruppo 5 cifre
  scanf("%1d%1d%1d%1d%1d", &j1, &j2, &j3, &j4, &j5);             // lettura 5 interi su singola cifra

  first_sum = d + i2 + i4 + j1 + j3 + j5;
  second_sum = i1 + i3 + i5 + j2 + j4;
  total = 3 * first_sum + second_sum;

  printf("Check digit: %d\n", 9 - ((total - 1) % 10));

  return 0;
}
```

per ora non ci vogliamo preoccupare della dimensione del numero a 5 cifre

Operatori: assegnamento

Per memorizzare il valore di un'espressione dopo che è stata calcolata all'interno di una variabile e poterlo riutilizzare si usa l'operatore **=**

lvalue=expr;

Valuta l'espressione *expr* e assegna a lvalue il valore così ottenuto

L'espressione *expr* può essere una semplice costante o essere una combinazione di variabili, operatori, funzioni e costanti.

Ritorna *expr*

E' associativo da destra a sinistra

lvalue: oggetto conservato nella memoria del computer, non una costante o il risultato di un calcolo

```
i=5;        //adesso i vale 5
j=i;        //adesso j vale 5
k=10*i+j;   //adesso k vale 55
i=j=k=0;    //equivalente a i=(j=(k=0)) cioè asegno 0 a k, poi a j, poi a i
```

```
12=i;       //SBAGLIATO
i+j=0;      // SBAGLIATO
-i=j;       // SBAGLIATO
```

Operatori: assegnamento composto

Gli assegnamenti che usano il vecchio valore di una variabile per calcolare quello nuovo possono essere scritti in forma più compatta:

```
a=a+2;   //si può anche scrivere
a+=2;
```

L'operatore += si comporta come un operatore di assegnamento e quindi è associativo a destra. Es.:

```
i += j += k;        //significa
i += (j += k);
```

Quindi:

```
v +=e;              //somma v a e, memorizza il risultato in v
v -=e;              //sottrae e da v, memorizza il risultato in v
v *= e;             //moltiplica v per e, memorizza il risultato in v
v /= e;             //divide v per e, memorizza il risultato in v
v %= e;             //resto divisione v per e, memorizza il risultato in v
```

Analogamente:

-=, *=, /=, %=, &=, ^=, |=, <<=, >>=

Operatori: incremento e decremento

Notazione prefissa:

++a; incrementa e ritorna il nuovo valore a+1

--a; decrementa e ritorna il nuovo valore a-1

Notazione **postfissa**:

a++; ritorna a e, dopo, lo incrementa

a--; ritorna a e, dopo, lo decrementa

Possono essere applicati a tutti i tipi interi (e ai puntatori)

Con i moderni compilatori, utilizzare ++ e – non renderà il progeamma nè più piccolo nè più veloce: la loro popolarità deriva dalla brevità e comodità di utilizzo (in origine il compilatore del linguaggio B generava una traduzione più compatta di ++i rispetto a i=i+1;) .

Incremento e decremento: esempi

Ricordiamo che gli operatori ++ e – possiedono dei **side effect**, ovvero modificano il valore dei loro operandi (come fa anche l'operatore di assegnamento). Esempio:

```
int i =1;
printf("i vale %d\n", ++i);          // stampa "i vale 2"
printf("i vale %d\n", i);            // stampa "i vale 2"
```

```
int i =1;
printf("i vale %d\n", i++);          // stampa "i vale 1"
printf("i vale %d\n", i);            // stampa "i vale 2"
```

Incremento e decremento: esempi

```
int i, j, k = 5;
i = ++k;                /* i vale 6, k vale 6 */
i = k++;                /* i vale 5, k vale 6 */


int i=4, j, k = 5;
j = i + k++;            /* j vale 9, k vale 6 */
```

Side effects

Gli operatori di assegnamento, di incremento e decremento provocano **effetti collaterali**, ovvero modificano il valore di una variabile, oltre a produrre un valore come risultato di un'espressione

L'ordine con cui si manifestano gli effetti collaterali **non è predicibile**

Esempio : x=j*j++;

il linguaggio C non definisce l'ordine di valutazione degli operatori moltiplicativi: compilatori diversi potrebbero valutare secondo ordini differenti i due operandi, producendo risultati diversi; se j=5, ad esempio, x=25 valutando prima l'operando di sinistra, 30 valutando prima il destro

Gli effetti collaterali

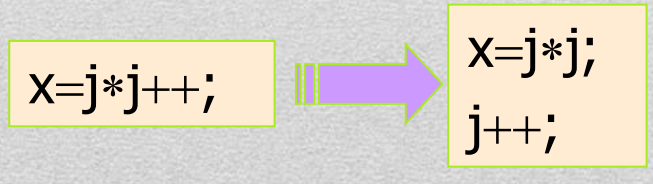
Istruzioni che provocano effetti collaterali imprevedibili non sono portabili e vanno evitate

Il problema degli effetti collaterali si manifesta anche nelle chiamate di funzione, perché lo standard C non definisce l'ordine con cui vengono valutati gli argomenti

Esempio: f(a, a++)

Per prevenire errori dovuti agli effetti collaterali, occorre seguire la regola:

Se in un'espressione si usa un operatore che implica effetti collaterali, la variabile coinvolta non deve essere usata in altro modo nella stessa espressione



Due operatori speciali

sizeof:

L'operatore permette a un programma di determinare quanta memoria viene richiesta per memorizzare un valore di un particolare tipo. Restituisce un intero senza segno che rappresenta il numero di byte; es.:

```
int i;

printf("Dimensione di i: %d\n", sizeof(i));
```

Casting:

(type) expr;

Conversione esplicita di tipo

forza l'espressione *expr* ad essere interpretata come fosse di tipo *type*; es.:

```
float f, frac_part;

frac_part = f - (int) f;        //calcolo della parte frazionaria di un valore float
```

Compatibilità fra tipi

- Ha senso l'assegnamento del valore di una variabile dichiarata come **int** a una variabile dichiarata come **float**?
- Che cosa succede in fase di esecuzione?
- Come viene trattato il risultato ottenuto?

Il C risponde con la tipizzazione forte e precise regole per la conversione implicita ed esplicita tra tipi

Tipizzazione forte

Le regole per la gestione del tipo delle variabili sono tutte verificabili a tempo di compilazione senza richiedere l'esecuzione del programma

- Grazie alla dichiarazione di tipi e variabili è normalmente possibile controllare durante la compilazione se un'istruzione è compatibile con le variabili coinvolte
- Eventuali violazioni alle regole possono essere individuate preliminarmente all'esecuzione del programma, aumentandone la sicurezza e risparmiando sui costi per la sua verifica

Regole di conversione implicita

risultato = x operando y;

Il tipo del risultato dell'espressione viene convertito in quello della variabile a cui è assegnato. Se il tipo della variabile è «grande» almeno quanto quello dell'espressione, il tutto funzionerà senza problemi. Esempio:

```
char c;
int i;
float f;
double d;
i=c;        //c viene convertita al tipo int
f=i;        //i viene convertita al tipo float
d=f;        //f viene convertita al tipo double
```

Regole di conversione implicita

Gli altri casi sono problematici: assegnare un numero a virgola mobile a una variabile intera causa la perdita della parte razionale del numero

```
int i;

i=876.21;          //adesso i vale 876
i=-876.21;         //adesso i vale -876
```

Inoltre, assegnare un valore a una variabile più piccola quando il valore è al di fuori del range di quest'ultima, conduce, quantomeno, a un risultato privo di significato

```
char c;
int i;
float f;
c=10000;           /***SBAGLIATO***/
i=1.0e20;          /***SBAGLIATO***/
f=1.0e100;         /***SBAGLIATO***/
```

Conversione aritmetica

Le normali conversioni aritmetiche vengono applicate agli operandi della maggior parte degli operatori, se non concordi (esempio, valori interi e a virgola mobile).

La strategia è quella di convertire gli operandi nel tipo «più piccolo» che sia in grado di conciliare con sicurezza i valori coinvolti nell'espressione, secondo la gerarchia:

float < double < long double

- se uno dei due operandi è long double, l'altro viene convertito a long double
- se uno dei due operandi è double, l'altro viene convertito a double
- se uno dei due operandi è float, l'altro viene convertito a float

Conversione aritmetica: esempi

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i=i+c;          //c viene convertita al tipo int
i=i+s;          //s viene convertita al tipo int
u=u+i;          //i viene convertita al tipo unsigned int
l=l+u;          //u viene convertita al tipo long int
ul=ul+l;        //l viene convertita al tipo unsigned long int
f=f+ul;         //ul viene convertita al tipo float
d=d+f;          //f viene convertita al tipo double
ld=ld+d;        /d viene convertita al tipo long double
```

Costanti

Le **costanti** sono dati che il programma non può modificare.

Una costante è un valore esplicito, che può essere assegnato ad una variabile, ma al quale non può essere mai assegnato un valore diverso da quello iniziale.

La direttiva *#define*, associa tra loro due sequenze di caratteri in modo tale che, prima della compilazione ed in modo del tutto automatico, ad ogni occorrenza della prima (detta *manifest constant*) è sostituita la seconda. Es.:

```
#define GG_ANNO_FIN 360        //durata in giorni dell'anno finanziario

interesse = importo * ggDeposito * tasso / GG_ANNO_FIN;
```

La direttiva *#define* non crea una variabile, ne' è associata ad un tipo di dato particolare: essa informa semplicemente il preprocessore che la costante manifesta, ogniqualvolta compaia nel sorgente in fase di compilazione, deve essere rimpiazzata con la stringa di sostituzione.

Come tutte le direttive al preprocessore, anche la *#define* non si chiude mai con il punto e virgola

Progetto di programmazione - ISBN

Il codice *International Standard Book Number* contiene 13 cifre suddivise in 5 gruppi. Il **primo** gruppo solitamente è 978 o 979. Il **secondo** specifica la lingua o il paese di origine (0 e 1 per i Paesi anglofoni). Il **terzo** gruppo identifica l'editore. Il **quarto** viene assegnato dall'editore per identificare uno specifico libro. Il **quinto** è una cifra di controllo utilizzata per verificare la correttezza delle cifre precedenti.
Scrivere un programma che suddivida in gruppi il codice ISBN immesso dall'utente.
Esempio:

input: Enter ISBN: 978-3-16-148410-0

output: GS1 prefix: 978
Group identifier: 3
Publisher code: 16
Item number: 148410
Check digit: 0

Nota: il numero di cifre in ogni gruppo può variare; non è corretto assumere che i gruppi abbiano sempre la lunghezza presentata nell'esempio.

Progetto di programmazione - ISBN

```
#include <stdio.h>

int main(void)
{
  int prefix, group, publisher, item, check_digit;

  printf("Enter ISBN: ");
  scanf("%d-%d-%d-%d-%d", &prefix, &group, &publisher, &item, &check_digit);

  printf("GS1 prefix: %d\n", prefix);
  printf("Group identifier: %d\n", group);
  printf("Publisher code: %d\n", publisher);
  printf("Item number: %d\n", item);
  printf("Check digit: %d\n", check_digit);

  return 0;
}
```

Progetto di programmazione – reverse cifre

Scrivere un programma che chieda all'utente di immettere un numero a tre cifre e successivamente stampi il numero con le cifre invertite. Esempio:

input: Enter a three-digit number: 978

output: The reversal is: 879

Suggerimento: se n è un intero allora n%10 indica l'unità di n, (n/10)%10 indica la decina mentre n/100 indica la cifra corrispondente alle centinaia

Progetto di programmazione – reverse cifre

```
#include <stdio.h>

int main(void)
{
  int n;

  printf("Enter a three-digit number: ");
  scanf("%d", &n);
  printf("The reversal is: %d%d%d\n", n % 10, (n / 10) % 10, n / 100);

  return 0;
}
```

Esercizio: visualizzare range dei valori assunti

```
#include <stdio.h>
#include <limits.h>
int main(){

 printf("Codifica del tipo SHORT \n");
 printf("Valore minimo = %d \n",SHRT_MIN);
 printf("Valore massimo = %d \n",SHRT_MAX);

 printf("Codifica del tipo INT \n");
 printf("Valore minimo = %d \n",INT_MIN);
 printf("Valore massimo = %d \n",INT_MAX);

 printf("Codifica del tipo LONG INT \n");
 printf("Valore minimo = %ld \n", LONG_MIN);
 printf("Valore massimo =%ld \n", LONG_MAX);

 printf("Codifica del tipo UNSIGNED SHORT \n");
 printf("Valore minimo = 0 \n");
 printf("Valore massimo = %u \n", USHRT_MAX);

 printf("Codifica del tipo UNSIGNED INT \n");
 printf("Valore minimo = 0 \n");
 printf("Valore massimo = %u \n", UINT_MAX);

 printf("Codifica del tipo UNSIGNED LONG INT \n");
 printf("Valore minimo = 0 \n");
 printf("Valore massimo = %lu \n", ULONG_MAX);
 getchar();
 returno 0;
}
```

Limiti nei tipi float

La libreria **<float.h>** contiene le definizioni delle costanti FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX, LDBL_MIN, LDBL_MAX relative rispettivamente ai valori massimi e minini dei tipi float, double e long double

```
#include <stdio.h>
#include <float.h>
int main(){
 printf("Codifica del tipo FLOAT \n ");
 printf("Valore minimo = %e \n", FLT_MIN);
 printf("Valore massimo = %e \n", FLT_MAX);

 printf("Codifica del tipo DOUBLE \n ");
 printf("Valore minimo = %e \n", DBL_MIN);
 printf("Valore massimo = %e \n", DBL_MAX);

 printf("Codifica del tipo LONG DOUBLE \n ");
 printf("Valore minimo = %Le \n", LDBL_MIN);
 printf("Valore massimo = %Le \n", LDBL_MAX);
 getchar();
 returno 0;
}
```