Lezione 1: Introduzione alla Programmazione

Algoritmi e Programmi

Problema → Algoritmo → Programma

Definizione del problema:

- Dati iniziali (input);
- Dati intermedi, se necessari;
- Dati finali (output);
- Definizione di un insieme di passi logici che trasformano i dati iniziali in dati finali

Il risultato è un algoritmo, un insieme finito di istruzioni che, se eseguite ordinatamente, sono in grado di risolvere il problema di partenza.

Le proprietà di un algoritmo sono le seguenti:

- **Non ambiguità** delle istruzioni;
- **Eseguibilità** delle istruzioni;
- **Finitezza**;

Algoritmo del caffè

Dati Iniziali:
Acqua, Caffè, Fornello, Gas, Macchinetta.

Procedura:

1. Riempire la caldaia della Macchinetta con l'Acqua;
2. Riempire il filtro della Macchinetta con il Caffè;
3. Montare i dispositivi della Macchinetta del Caffè;
4. Posizionare la Macchinetta sul Fornello e accendere il Gas;
5. Attendere l'ebollizione dell'Acqua;
6. Spegnere il Gas quando il Caffè è disponibile in forma liquida nella Macchinetta;

Algoritmo del Tè

Dati Iniziali:

Acqua, Tè, Fornello, Gas, Pentolino, Tazza

Procedura:

1. Mettere il pentolino sul fuoco;
2. Finché l'acqua non bolle;
3. Attendere l'ebollizione dell'Acqua;
4. Versare in una tazza
5. Aggiungere il tè

{Istruzioni 2 e 3: ciclo

Entrambi gli algoritmi però presentano delle istruzioni ambigue in quanto non specificano le quantità dei vari ingredienti.

Esempio (1)

Descrizione ad alto livello:
Analizzare i risultati dell'esame di 10 studenti e decidere se debbano essere aumentate le tasse scolastiche

Primo raffinamento:
Inizializzare le variabili;
Prendere in input 10 valutazioni della prova e contare le promozioni e le bocciature;
Visualizzare un sommario dei risultati dell'esame e decidere se le tasse scolastiche debbano essere aumentate

Secondo raffinamento:
Inizializzare le variabili Inizializzare le promozioni a zero
Inizializzare le bocciature a zero
Inizializzare gli studenti a uno

Terzo raffinamento:
Prendere in input 10 valutazioni della prova e contare le promozioni e le bocciature
Finché il contatore degli studenti è inferiore o uguale a dieci prendere in input il prossimo risultato d'esame
Se lo studente è stato promosso
    Aggiungere uno ai promossi
Altrimenti
    Aggiungere uno ai bocciati
Aggiungere uno al contatore degli studenti

Quarto raffinamento:
Visualizzare un sommario dei risultati dell'esame e decidere se le tasse scolastiche debbano essere aumentate
Visualizzare il numero delle promozioni
Visualizzare il numero delle bocciature
Se più di otto studenti sono stati promossi
    Visualizzare il messaggio "aumentare le tasse."

Esempio (2)

Il Fattoriale di *n*: n! =n*(n-1) *(n-2) *…*3*2*1
Prendere in input il numero *n* e considerare una variabile *fact* inizializzata ad 1
Se *n*=0 or *n*=1 allora il risultato è *fact*
Finché *n* non risulta uguale a 1effettuare le seguenti operazioni:
    Moltiplicare *n* e *fact* e memorizzare il risultato in *fact* (*fact*=*n***fact*)
    Decrementare *n* di un'unità (*n*=*n*-1)
Quando *n* risulta uguale ad 1 il risultato è *fact.*

Linguaggio di programmazione

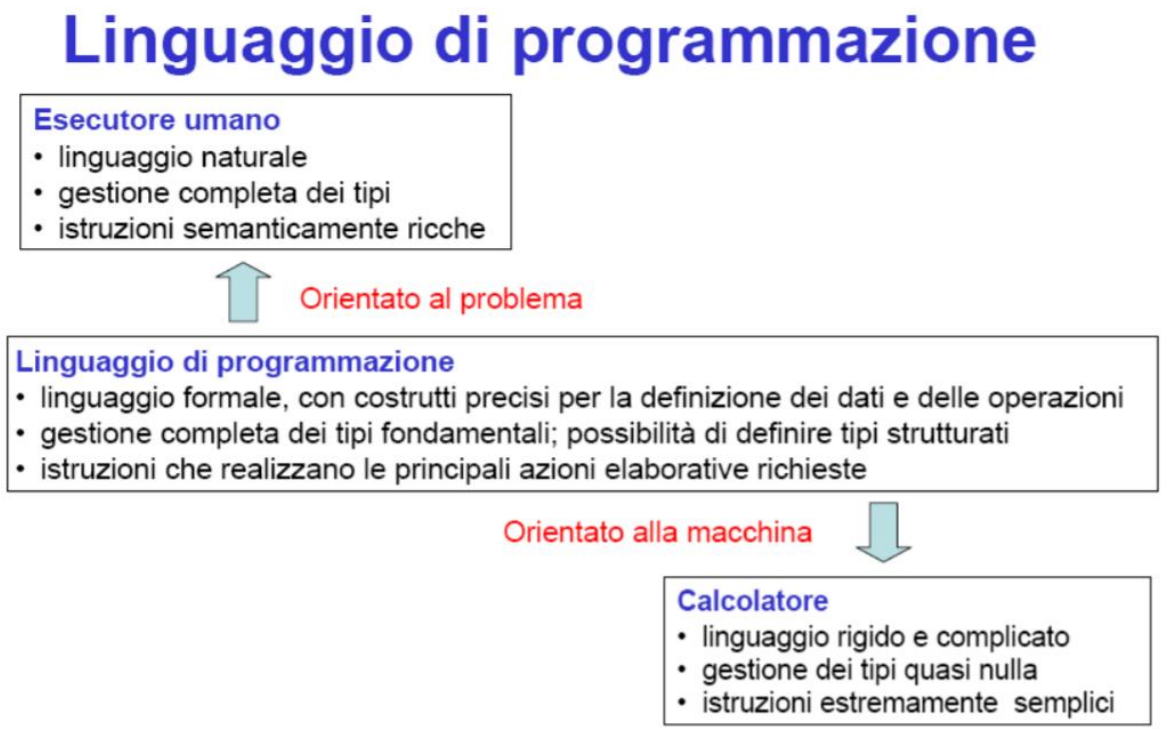
L'algoritmo dovrà essere specificato in un linguaggio che l'elaboratore è in grado di interpretare in modo corretto e contenere istruzioni che possono essere eseguite dal computer stesso.

Un programma può quindi essere definito come un insieme di istruzioni espresse in un linguaggio formale (sintassi e semantica) chiamato linguaggio di programmazione.

I computer però non sono in grado di capire nemmeno il linguaggio programmazione in quanto il microprocessore sa elaborare solo in linguaggio binario: ecco, pertanto, che i programmi dovranno essere ulteriormente tradotti da apposite applicazioni quali interpreti o compilatori.

Un calcolatore basato sul modello di von Neumann permette l'esecuzione di un programma, cioè una sequenza di istruzione descritte nel linguaggio interpretabile dal calcolatore che realizzano un particolare algoritmo. Le caratteristiche ti tale linguaggio sono le seguenti:

- È codificato tramite sequenze di bit
- Accede ai dati tramite gli indirizzi di memoria o i registri interni della CPU
- Ogni istruzione può compiere solo azioni molto semplici
- Non gestisce direttamente i tipi di dati di interesse
- È strettamente legato alla particolare macchina su cui è definito (per questo viene definito *linguaggio macchina*)



Sintassi e Semantica

I linguaggi di programmazione, così come tutti i linguaggi usati per la descrizione formale di fenomeni di qualunque tipo, coinvolgono due aspetti distinti, che è importante distinguere: la sintassi e la semantica. La sintassi ha a che fare con la struttura (o la forma) dei programmi esprimibili in un dato linguaggio di programmazione. La semantica, invece, ha a che fare con il significato dei programmi esprimibili in un dato linguaggio. Prendiamo come esempio la data che ha una sintassi comune ovvero XX/XX/XXXX, ma in base alla regione in cui ci troviamo potremmo leggere le prime due X come giorno, come in Italia. Se invece ci troviamo in America le prime due X rappresenteranno il mese.

Esempio:

Sintassi <*data*> = CC/CC/CCCC dove C rappresenta una cifra

Semantica: 01/02/2003

In Italia → 1 febbraio 2003

In USA → 2 gennaio 2003

Interpreti e compilatori

L'*interprete* è un programma che traduce, istruzione per istruzione, il programma sorgente. Ogni istruzione è sottomessa alla CPU appena è stata tradotta. Anche se una se istruzione è contenuta 10 volte in un programma verrà tradotta ogni volta che si presenterà al traduttore.

Il *compilatore*, invece, traduce tutto il programma in una sola volta e poi lo sottomette alla CPU. Se una stessa istruzione compare 10 volte, viene tradotta solo la prima volta e poi memorizzata in maniera tale che possa essere ripresa dal compilatore e inserita nelle restanti nove righe di programma.

È possibile così riassumere le diverse fasi che portano dall'analisi del problema all'ottenimento di una soluzione:

1. *Analisi*: Viene analizzato il problema in tutti i suoi aspetti e si cercano i fattori sui quali fare leva per risolverlo. Il risultato è una soluzione informale.
2. *Formalizzazione*: La soluzione trovata nel passo precedente è riformulata in maniera da utilizzare una sintassi e una semantica corrette, per produrre un algoritmo di risoluzione (Le relazioni tra costrutti del linguaggio ed azioni sono univocamente definite). Il linguaggio utilizzato sarà di tipo formale, ma necessiterà di ulteriori passi per essere reso utilizzabile dalla macchina.
3. *Programmazione*: Il risultato di questa fase è un programma di alto livello che utilizza un linguaggio di programmazione costituito da segni matematici e parole chiave e non da una successione indecifrabile di 0 e 1. Questa fase però non produce ancora un programma che possa essere compreso dalla macchina: si necessita della fase seguente.
4. *Traduzione*: In questa fase il programma di alto livello viene tradotto da appositi software in linguaggio macchina.
5. *Esecuzione*: Il programma viene sottoposto al microprocessore che lo esegue e fornisce la soluzione al problema.
6. *Verifica*: Tramite un test pratico di funzionamento ed un'analisi teorica del programma dovrebbe si verifica che il software realizzato corrisponda alle sue aspettative e svolga le funzioni per cui è stato elaborato.

A questo punto è necessaria la formazione degli utenti, per impartire loro le istruzioni che occorrono per servirsi dei nuovo software: essa può avvenire secondo diverse modalità. Nella fase di implementazione, infine, tutti gli utenti interessati possono servirsi del programma elaborato (versione beta del programma). Gli aggiornamenti successivi sono detti "versioni", o release, e sono identificati da un numero progressivo.

Il Software

Il Software può essere diviso in tre tipologie:

- Software di base (Sistema Operativo)
- Software applicativo (Office Automation)
- Software di sviluppo (Ambiente di Programmazione)

Il software di base racchiude in sé sia il software di sistema, necessario a far funzionare l'elaboratore, sia quello utilizzato dagli sviluppatosi di programmi per facilitare il loro lavoro. Inoltre, interpreta ed esegue comandi elementari, traduce i comandi applicativi in operazioni della macchina, organizza la struttura della memoria di massa e ripartisce le risorse del sistema tra gli utenti (multitasking – multiuser).

Il software applicativo comprende invece tutti quei programmi utilizzati dagli utenti per gestire, per esempio, la posta, la contabilità di casa, per redigere una lettera, creare una presentazione, telefonare via Internet, ecc., oppure applicativi creati ad hoc per risolvere un determinato problema.

L'ambiente di Programmazione

L'ambiente di Programmazione si compone di quattro elementi fondamentali:

- *Editor*: Ci permette di scrivere fisicamente il programma.
- *Compilatore*: Traduce i programmi in codice oggetto, eseguibile dal calcolatore.
- *Linker*: Collega diversi moduli di un programma e le cosiddette librerie, producendo il vero e proprio eseguibile.
- *Debugger*: permette di interrompere l'esecuzione del programma dopo ogni istruzione per correggere eventuali errori (bug).

Il processo che viene fatto dall'ambiente di programmazione per eseguire il nostro programma C si divide invece nelle seguenti fasi:

1. *Editing*: Il programma è creato nell'editor e memorizzato su disco.
2. *Preprocessing*: Il processore elabora il codice del programma.
3. *Compilazione*: Il compilatore crea il codice oggetto e lo memorizza su disco.
4. *Linking*: Il Linker collega il codice oggetto con le librerie, crea un a.ont e lo memorizza su disco.
5. *Caricamento*: Il Loader pone il programma in memoria.
6. *Esecuzione*: La CPU prende ogni istruzione e la esegue, possibilmente memorizzando i nuovi valori dei dati durante l'esecuzione del programma.

Lezione 2: Introduzione al C

Breve Storia del C

Linguaggio general purpose progettato da Dennis Ritchie (Bell Laboratories) nel 1972 su elaboratore PDP11, inizialmente fu utilizzato per la programmazione del sistema operativo UNIX.
Negli anni '80 fu utilizzato anche nelle scuole e nelle università per la didattica (C tradizionale). Alla fine degli anni '80 nasce ANSI C o C standard che supera alcuni limiti del C tradizionale, la standardizzazione ha reso il C indipendente dal processore (hardware).

La Standardizzazione

Esistevano molte sottili differenze tra le diverse implementazioni del C.
Ciò dava problemi di incompatibilità, quindi fu creato un comitato che si preoccupò di fornire una definizione "non ambigua e indipendente dalla macchina" e così fu creato uno Standard nel 1989, aggiornato nel 1999.

Perché ANSI C (d'ora in poi solo C)

Rispetto ad altri linguaggi (ad esempio il Pascal) ha meno parole chiave ma è più potente per la presenza di strutture di controllo e tipi di dati più adeguati.
È il linguaggio nativo di UNIX, un sistema operativo interattivo molto utilizzato su server, mainframe e workstation, inoltre molti programmi per la gestione database e per applicazioni grafiche sono scritti in C.
Ha operatori che consentono di accedere alla macchina a livello di bit, e che sono in grado in modo pulito, elegante e compatto di realizzare operazioni complesse.

La libreria standard del C

I programmi C consistono di pezzi/moduli detti funzioni e un programmatore può creare le sue proprie funzioni

Vantaggio: il programmatore sa esattamente come lavora.
Svantaggio: si spreca tempo.

I programmatori spesso usano le funzioni di libreria del C usandole come building blocks. Se esiste una funzione precostituita, è generalmente meglio usarla che scrivere la propria in quanto le funzioni di libreria sono scritte attentamente, sono efficienti e portabili.

Semplici programmi in C

```
1   /* Stampa una riga di testo ver. 1*/
2   #include <stdio.h>
3   int main(void)
4   {
5       printf("Welcome to C!\n");
6       return 0;
7   }
```

Commenti

- Il testo racchiuso tra /* e */ è ignorato dal compilatore
- In blu le parole chiave
- **#include <stdio.h>**
  Sono direttive per il preprocessore: carica il contenuto di un certo file **<stdio.h>** che consente operazioni di input/output (ne parleremo nella prossima lezione)
- **int main(void)**
  I programmi C contengono uno o più funzioni, esattamente una di queste deve essere **main.** La parentesi è usata per indicare una funzione
- **int** significa che il main "restituisce" un valore intero
- **void** significa che il main non prende input
- Le parentesi graffe { } indicano un blocco. I corpi di tutte le funzioni devono essere contenuti in parentesi graffe.
- **printf("Welcome to C!\n");**
  La funzione **printf()** è inclusa nella libreria *stdio.h* e istruisce il computer ad eseguire la stampa della stringa di caratteri all'interno delle virgolette. L'intera linea è detta una istruzione (statement). Tutte le istruzioni devono terminare con un punto e virgola.
- \ Carattere di escape (fuga) e toglie l'usuale significato al carattere che lo segue
- **\n** è una sequenza di escape, ed indica il carattere di *newline*
- **return 0:** Un modo di uscire da una funzione in questo caso, significa che il programma termina normalmente
- } Parentesi graffe chiuse indica che la fine del **main** è stata raggiunta

Le parole chiave

Alcune parole sono dette **parole chiave** e non possono essere utilizzate dal programmatore come nomi di variabili.

Rispetto ad altri linguaggi di programmazione il C possiede un numero ridotto di parole chiave.

Keywords PAROLE CHIAVI			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Alcune sequenze di escape

Le sequenze di escape sono delle combinazioni di caratteri che possono essere utilizzate all'interno delle printf, queste ci permetto di compiere determinate azioni:

- **\n** = Newline (nuova linea). Posiziona il cursore all'inizio della riga successiva
- **\t** = Tabulazione orizzontale. Posiziona il cursore alla tabulazione successiva
- **\r** = Ritorno carrello. Posiziona il cursore all'inizio della riga corrente
- **\a** = Allarme. Fa riprodurre un allarme acustico al sistema
- **** = Backslash. Visualizza un carattere backslash
- **\'** = Apice singolo. Visualizza un carattere apice singolo
- **\"** = Virgolette. Visualizza un carattere virgolette

Dichiarazioni, Espressioni e Assegnamento

Oggetti manipolati da un programma:
Costanti: Interi (intervallo tipico [-2^{31}, $2^{31}-1$]), Reali (virgola mobile), Stringhe

Variabili (devono essere dichiarate prima di essere utilizzate) e sono composte da:

- **Nome** (identificatore: No parole chiave)
- **Tipo**
- **Valore**

Utilità della dichiarazione di variabili: Definizione dello spazio di memoria da parte del compilatore e scelta delle istruzioni macchina appropriate per l'esecuzione.

Espressione

Combinazione di costanti, variabili, operatori e chiamate a funzione. Esempi:

- tot=n1+n2;
- user=GetLine();
- tot=GetInteger()+1;

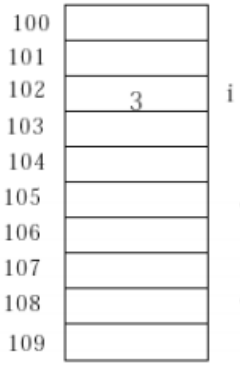
Assegnamento

Operatore di Assegnamento **=**
Associatività: da destra a sinistra

Esempio:
tot=138;
tot=n1+n2

Variabili e Memoria

- La memoria è un insieme di locazioni

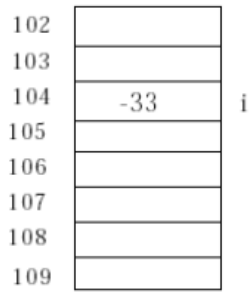



```
int i; /* dichiarazione */

i = 3; /* inizializzazione */
```

- Ogni variabile ha un *nome*, un *tipo*, un *valore* ed una *locazione*.
  - -I nomi di variabili corrispondono alle locazioni nella memoria del computer.
- Nell'esempio:
  - Il nome della variabile è **i**
  - La variabile **i** è di tipo **int**
  - La variabile **i** ha valore **3**

La funzione scanf

```
/* Prendere in input un valore
   e stamparlo a video */

#include <stdio.h>
int main(void)
{
    int i; /* dichiarazione */
    printf("Inserisci un intero:"); /* prompt */
    scanf("%d", &i); /* legge un intero */
    printf("\nIl valore inserito è %d\n", i); /*stampa */
    return 0;
}
```

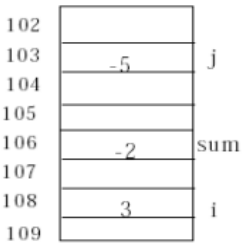


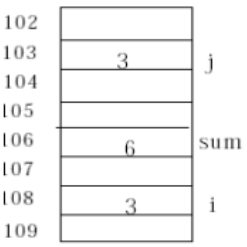
Inserisci un intero:-33
Il valore inserito è -33

La funzione scanf("%d", &i); ottiene un valore dall'utente
scanf usa l'input standard (di solito la tastiera), questa scanf ha due argomenti:

- **%d** - la specifica di conversione indica che il dato dovrebbe essere un intero decimale
- **&i** - indica la locazione in memoria per memorizzare la variabile

```
/* lettura e scrittura di variabili */
#include <stdio.h>
int main(void)
{
    int i, j, sum;
    i = 3;
    printf("Inserisci un intero:");
    scanf("%d", &j);
    sum = i + j;
    printf("\ni = %d\tj = %d\tsomma = %d\n", i, j, sum);
    j = i;
    sum = i +j;
    printf("i = %d\tj = %d\tsomma = %d\n", i, j, sum);
    return 0;
}
```






```
Inserisci un intero: -5
i=3     j=-5    somma=-2
i=3     j=3     somma=6
```

La lettura delle variabili non cambia il loro valore, invece la scrittura è distruttiva: Ogni volta che un valore è sistemato in una locazione di memoria esso si sostituisce al valore contenuto in precedenza in quella locazione.

Operatori e precedenze

Operatori

Caratteri speciali a cui è assegnato un particolare significato:

- **+** addizione
- **-** sottrazione
- ***** moltiplicazione
- **/** divisione
- **%** modulo (a%b equivale al resto di a/b)

Operatori di Incremento e Decremento: ++ e - -. (++i e i++ equivalgono a i=i+1)

Operatori di Assegnamento

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

variable op = expression → variable = variable op (expression)

Esempio:

j***=** k+3 → j=j*(k+3)

d **-=** 4 → d = d - 4

e ***=** 5 → e = e * 5

f **/=** 3 → f = f / 3

g **%=** 9 → g = g % 9

Lezione 3: Tipi di Dato

Potete espressivo di un linguaggio di programmazione:

- Tipi di dati che nel linguaggio di possono esprimere e manipolare
- Operazioni esprimibili direttamente o per composizione (Set istruzioni, funzioni definibili dall'utente)

Un tipo di dato T astratto è definito da:

- Un dominio D di valori
- Un insieme finito F di operatori op_1 op_2 … op_k

- Manipolazione dei valori di D:

  Opi: DxD → D

  Opi: D → D

  - Esempio

    Tipo int:

    D = Z dove Z è l'insieme degli interi nell'intervallo [-∞; +∞]

    F = {+, -, /, *, <, >, ==}

Tipologia di Tipi di Dati

- Tipi di Dati Primitivi

  Basati direttamente sulla rappresentazione Hardware (int, char, real, boolean)

  Supportano i Tipi di Dati del Linguaggio di programmazione
- Tipi di Dati del Linguaggio di programmazione

  Supportati dal linguaggio di programmazione

  Tipi strutturati (Array)
- Livello Utente

  Definiti dall'utente usando le specifiche del Linguaggio di programmazione

- Record, Grafi, Liste

Classificazione dei Tipi di Dato

- Elementari

  Dominio di elementi atomici ed omogenei

- Interi, Caratteri, Reali, Booleani
- Coincidono con i Tipi di Dati primitivi
- Strutturati

  Combinazione logica di Tipi di Dati elementari e strutturati

- Dominio di elementi non atomici ed eterogenei
- Array, Record, Grafi, Liste

Astrazione dei dati

Indipendenza dalla rappresentazione:

- La specifica astratta definisce il comportamento di un oggetto indipendentemente dalla sua implementazione
- La rappresentazione concreta definisce l'implementazione dell'oggetto

Esempio:

- Specifica astratta: Sequenza dei numeri primi nell'intervallo [7, 19]
- Specifica concreta: int primo[5] = {7, 11, 13, 17, 19}

Un programma dovrebbe essere progettato in modo tale che sia possibile modificare la rappresentazione di un oggetto senza modificare il resto del programma.

- Se l'oggetto è un Tipo di Dati: astrazione dati
- Se l'oggetto è una funzione (procedura): Astrazione funzionale

Dichiarazione del Tipo di Dati

Con la dichiarazione di crea un'associazione permanente tra il Tipo di Dati, l'identificatore e l'indirizzo assegnato durante l'esecuzione del codice, ciò che cambia è il contenuto della locazione in sintonia con il concetto di variabile.

Overloading

Si definisce Overloading(sovraccarico) se ad uno stesso operatore è assegnata nell'ambito del Linguaggio di programmazione una semantica differente a seconda del tipo a cui è applicato. Il C consente l'overloading.
Esempio con l'operatore somma (+):

Se applicato ad un int: + :ZxZ → Z

Se applicato ad un float: + :RxR → R

Compatibilità

Siano T_1 e T_2 due Tipi di Dato:

- T_1 ha operatori op_i: $D_1xD_1 \rightarrow D_1$
- T_2 ha operatori op_i: $D_2xD_2 \rightarrow D_2$

T_1 è compatibile con T_2 se $D_1 \subseteq D_2$

Se op_i è definito per il tipo T_2 ossia op_i: $D_2xD_2 \rightarrow D_2$ allora le seguenti notazioni sono corrette:

- op_i:$D_1xD_2 \rightarrow D_2$
- op_i:$D_2xD_1 \rightarrow D_2$

Coercion (Forzatura)

- Le variabili di tipo T_1 sono considerate automaticamente di tipo T_2

Se entrambi gli argomenti sono di tipo T_1:

- L'operatore è definito solo per T_2

  $op_i : D_1xD_1 \rightarrow D_2$

L'operatore è definito sia per T_1 che per T_2 (overloading)

- $op_i : D_1xD_1 \rightarrow D_1$

Type Checking

Il Controllo dei tipi garantisce che le operazioni di un programma siano applicate in modo proprio, ciò aiuta anche nella gestione degli errori (un float gestito come un int determina un errore di tipo)
Un programma si dice safe se è sicuro rispetto ai tipi ottenendo così un'esecuzione senza errori di tipo

Il controllo dei tipi può essere:

- Statico: Il controllo avviene durante la traduzione del codice sorgente, questo porta a una compilazione rallentata.
- Dinamico: Il controllo viene effettuato durante l'esecuzione aggiungendo del codice nell'eseguibile che consuma spazio e tempo (esecuzione meno efficiente) Un eventuale problema di tipo si manifesta solo durante l'esecuzione e poiché possono esserci parti di codice usate solo di rado se il problema è presente in tali sezioni non si avrà un responso.

Lezione 4: Tipi di Dati, Istruzioni di Controllo

Il C ha 5 tipi di dati principali:

- caratteri, **char** (solitamente un carattere in un byte)
- numeri interi, **int** (solitamente un intero in 2 byte)
- numeri in virgola mobile, **float**
- numeri in virgola mobile doppi, **double**
- non valori, **void**

Tutti gli altri tipi in C sono basati su questi 5 tipi.

I modificatori ricevono la dichiarazione di tipo, altera il significato del tipo base per adattarlo con più precisione alle varie situazioni: signed, unsigned, long, short
I modificatori signed, unsigned, long, short sono applicabili a char e int, long può essere applicato anche a double.

TIPO	Dimensioni approssimative in bit	intervallo minimo
char	8	da –127 a 127
unsigned char	8	da 0 a 255
signed char	8	da –127 a 127
int	16	da –32.767 a 32.767
unsigned int	16	da 0 a 65.535
signed int	16	come int
short int	16	come int
unsigned short int	16	come unsigned int
signed short int	16	come int
long int	32	da –2.147.483.647 a 2.147.483.647
signed long int	32	come long int
unsigned long int	32	da 0 a 4.294.967.295
float	32	6 cifre di precisione
double	64	10 cifre di precisione
long double	80	14 cifre di precisione

Regole di Promozione

Le regole di promozione si applicano automaticamente alle espressioni che contengono dei valori di due o più tipi di dato, dette anche espressioni di tipo misto.
Ogni valore in un'espressione di tipo misto sarà promosso automaticamente a quello più alto dell'espressione (in realtà verrà creata una copia temporanea di ognuno dei valori, lasciando gli originali invariati). Ovviamente convertire una tipo di dato in un tipo di dato più basso produce perdita di informazioni.

Istruzioni di Controllo

L'esecuzione predefinita in C è l'esecuzione sequenziale dove le istruzioni sono eseguite l'una dopo l'altra nell'ordine in cui sono scritte.

Le istruzioni di Controllo

- Condizionali: If, if/else, switch
- Iterative: while, do/while, for
- Salti: break, continue, goto, return

Tipi di dato	Specifiche di conversione per printf	Specifiche di conversione per scanf
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c

Flowchart

Il Flowchart è una rappresentazione grafica di un algoritmo. Viene disegnato usando dei simboli speciali connessi da frecce dette linee di flusso (flowlines).

- Simbolo rettangolo (simbolo di azione): indica qualsiasi tipo di azione.
- Simbolo ovale: denota l'inizio o la fine di un programma, o di una sezione di codice (in questo caso si usano anche simboli cerchietto, detti simboli di connessione).
- Simbolo rombo (di decisione): indica che dovrà essere eseguita una scelta

Istruzione Condizionale

Usata per scegliere tra sequenze di azioni alternative:

Rappresentazione logica: Se il voto dello studente è maggiore o uguale a 60; Visualizza "Promosso".

Se la condizione è vera: viene eseguita l'istruzione di visualizzazione e il programma va alla prossima istruzione, invece se la condizione è falsa: l'istruzione di visualizzazione viene ignorata e il programma va all'istruzione successiva.

```
if (voto >= 60)
        printf("Promosso\n");
```

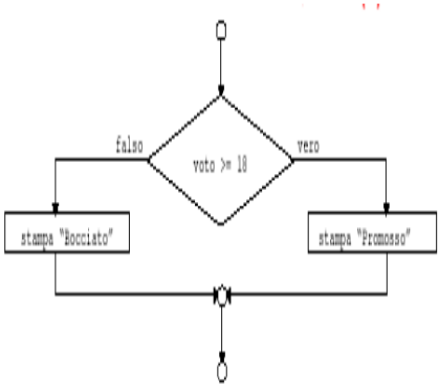
Istruzione condizionale If/Else

La differenza principale tra l'istruzione if e quella if/else è la seguente:

- If: Esegue un'azione solo quando la condizione è vera.
- If/Else: Un'azione differente quando l'azione è vera rispetto a quando è falsa

Rappresentazione logica:

Se il voto dello studente è maggiore o uguale a 18
    Visualizza "Promosso";
altrimenti
    Visualizza "Bocciato";



In C:

```
if (voto >= 18)
    printf("Promosso");
else
    printf("Bocciato);
```

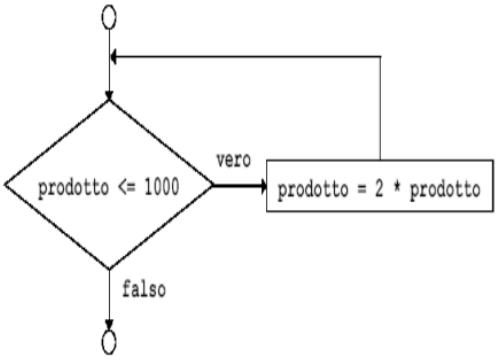
Lezione 5: Istruzioni Iterative

La struttura di iterazione While

Struttura di iterazione: Il programmatore specifica un'azione che deve essere eseguita finché una certa condizione è vera. Il ciclo While viene ripetuto finché la condizione non diventa falsa.

Esempio:
(calcola la prima potenza di 2 maggiore di 1000):

```
int prodotto=2;
while ( prodotto <= 1000 )
    prodotto = 2 * prodotto;
printf("la prima potenza di due maggiore di
1000 è %d", prodotto);
```



Problema

Un college ha una lista di risultati di test (1 = promosso, 2 = bocciato) per 10 studenti. Scrivere un programma che analizza i risultati. Se più di 8 studenti sono stati promossi, visualizzare "Aumentare le tasse"

- Il programma deve elaborare 10 risultati di test
- Sarà usato un ciclo controllato da contatore
- Due contatori possono essere usati (Uno per il numero di promossi, uno per il numero di bocciati)
- Ogni risultato di test è un numero (1 o 2) (Se il numero non è un 1, assumiamo che è un 2)

```
#include <stdio.h>
int main(void)
{
/* inizializza le variabili nelle dichiarazioni */
int promossi = 0, bocciati = 0, studenti = 1, risultato;
/* elabora 10 studenti; ciclo controllato da un contatore */
while ( studenti <= 10 )
{
     printf( "\ninserisci risultato ( 1=promosso,2=bocciato ): " );
     scanf( "%d", &risultato );
            if ( risultato == 1 ) {     /* if/else nidificato nel while */
            promossi ++; }
            else {
            bocciati ++;
            }
      studenti ++;
}
printf( "Promossi %d\n", promossi ); /*prima istruzione eseguita dopo la violazione del test loop*/
printf( "Bocciati %d\n", bocciati );
if ( promossi > 8 )
printf( "Aumenta le tasse\n" );
return 0;   /* chiusura con successo */
}
```

Il problema può essere generalizzato:

Sviluppare il programma per il calcolo della media di una classe, che elaborerà un numero arbitrario di votazioni ogni volta che il programma sarà eseguito.

- Numero di studenti non noto in anticipo (iterazioni indefinite)
- Come saprà il programma quando terminare l'immissione delle votazioni e calcolare e visualizzare la media della classe?

Uso di un valore sentinella

Detto anche valore di segnalazione o valore dummy (fittizio) o valore flag (bandiera), esso indica "la fine dell'immissione dei dati". Il ciclo termina quando viene immesso il valore sentinella, quindi esso deve essere scelto in modo da non confondersi con un valore di input accettabile (come -1 in questo caso).

```
#include <stdio.h>
int main()
{
float media;          /* nuovo tipo di dato*/
int contatore, voto, totale;
/* fase di inizializzazione */

totale = 0;
contatatore = 0;

/* fase di elaborazione */
printf( "Inserisci voto, -1 per finire: " );
scanf( "%d", &voto );
```

```
while ( voto != -1 )
{
totale = totale + voto;
contatore = contatore + 1;
printf( " Inserisci voto, -1 per finire : " );
scanf( "%d", &voto );
}
/* fase di terminazione */

if    ( contatore != 0 ) {
      media = ( float ) totale / contatore;
      printf( "la media della classe è %.2f",
      media );
      }
      else
      printf( "Non è stato inserito alcun
              voto\n" );

      return 0; /* indica che il programma è
      terminato con successo*/

}
```

Loop

- Insieme di istruzioni che il computer esegue ripetutamente finché una condizione risulta vera

Iterazione controllata da un contatore

- Iterazione definita: è esplicitato quante volte eseguire il loop
- Utilizzo di variabile di controllo per contare le iterazioni

Iterazione controllata da una variabile di controllo

- Iterazione Indefinita
- Usata quando non è noto il numero di ripetizioni da eseguire
- Il valore della variabile di controllo indica il test di fine iterazione

L'iterazione controllata da un contatore richiede:

- nome di una variabile di controllo (o loop counter)
- valore iniziale della variabile di controllo
- condizione che verifica il valore finale della variabile di controllo (cioè se il loop deve continuare)
- incremento (o decremento) con il quale la variabile di controllo è modificata ad ogni interazione del loop.

Iterazione: l'istruzione for

- **Formato del loop for**

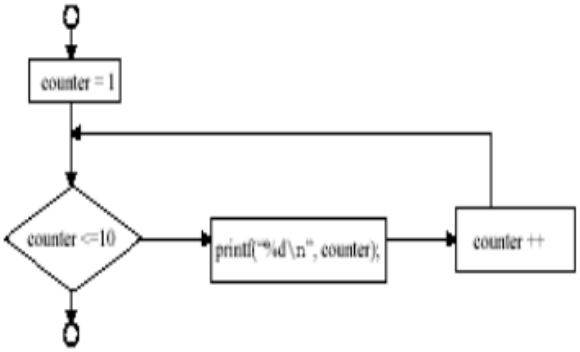
for (*inizializzazione*; *TestLoop*; *incremento*)
Istruzione;

Esempio:

Variabile di controllo

Incremento della variabile di controllo

```
for(counter = 1; counter <= 10; counter++ )
printf( "%d\n", counter );
```



– Stampa gli interi da 1 a 10

- Il loop **for** può essere realizzato in termini dell'istruzione **while** :

```
inizializzazione;
while ( TestLoop)
{
istruzione;
incremento;
}
```

for (*inizializzazione*; *TestLoop*; *incremento*)
Istruzione;

Adesso possiamo scrivere un programma che somma i primi n numeri pari utilizzando il ciclo for:

```
int main(void)
{
int sum = 0, n, counter;
printf("quanti numeri pari desideri sommare?:");
scanf("%d", &n);
for(counter = 1; counter <= n; counter ++)
sum += 2*counter;
printf("la somma dei primi %d numeri pari è %d", n, sum);
return 0;
}
```

Iterazione do/While

Do/While è un'altra istruzione di iterazione simile alla struttura While, ma qui la condizione di ripetizione è effettuata dopo che il body del ciclo è stato eseguito, quindi il corpo del ciclo viene eseguito almeno una volta

Il ciclo è così formato:

```
do{
    Statement
} While (condizione);
```

- Esempio (sia **`counter = 1`**)

```
do {
printf( "%d  ", counter );
}
while (++counter <= 10);
```

– Stampa gli interi da1 a 10

Le istruzioni break e continue

Il **break** causa un'uscita immediata da una struttura While, for, do/while o switch, l'esecuzione del programma continua dalla prima istruzione successiva dopo la struttura.

Di solito il break si usa per uscire da un loop o per saltare la sezione restante di una struttura switch.

Il **continue** invece salta le istruzioni che restano in un corpo di un While, for, do/while, eseguendo la successiva iterazione del loop. Nel While e do/while il test di continuazione viene valutato immediatamente dopo l'esecuzione del continue, nel for invece viene eseguito l'incremento, e successivamente il test di continuazione.

```
#include <stdio.h>
int main()
{
int x;
for ( x = 1; x <= 10; x++ )
{
     if ( x == 5 )
     break;  /* interrompe il ciclo solo se x == 5 */
      printf( "%d ", x );
}
printf( "\nSi è usato break per interrompere il ciclo\n" );
return 0;
}
```

```
1 2 3 4
Si è usato break per interrompere il ciclo
```

```
#include <stdio.h>
int main()
{
int x;
for ( x = 1; x <= 10; x++ )
{
    if ( x == 5 )
    continue;  /* salta il codice restante nel ciclo solo se x == 5 */
    printf( "%d ", x );
 }
printf( "\nSi è usato continue per saltare la stampa di 5\n" );
return 0;
}
```

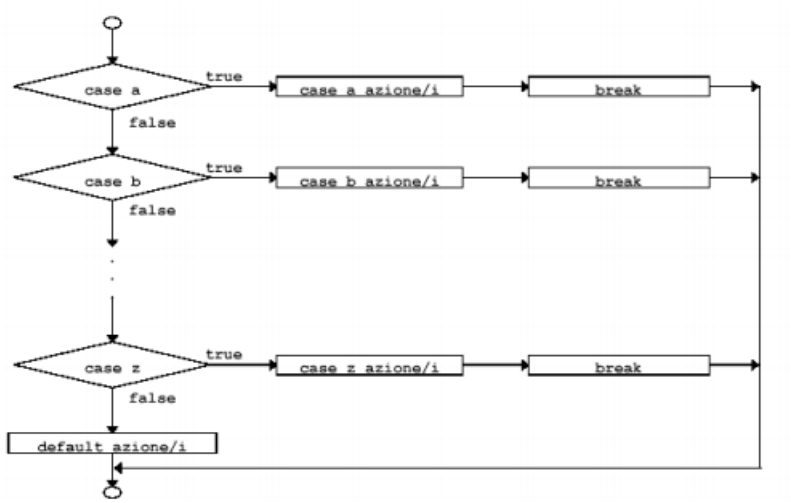
```
1 2 3 4 6 7 8 9 10
Si è usato continue per saltare la stampa di 5
```

Operazione di selezione multipla: switch

Lo **switch** risulta particolarmente utile quando una variabile o espressione è testata per tutti i valori che può assumere e quando diverse azioni possono essere eseguite. Può essere utilizzato soltanto per controllare un'espressione intera costante (ogni combinazione di costanti di caratteri e interi che possa essere valutata con un valore intero.

Lo switch è formato da una serie di label case ed uso facoltativo di default:

```
switch( value ){
        case '1':
        azione/i
        case '2':
        azione/i
        default:
        azione/i
}
```



- break; per uscire dalla struttura

```
/*conta_lettere.c*/

#include <stdio.h>
int main()
{
int grade; int aCount = 0, bCount = 0, cCount = 0,
      dCount = 0, fCount = 0;
printf(  "Inserire le votazioni in lettere.\n"  ); printf(
      "Inserire il carattere G per terminare l'input.\n");
while ( ( grade = getchar() ) != 'G' )
{
        switch ( grade ) { /* switch annidato nel while
            */
            case 'A': case 'a':  /* la votazione è stata
                A */ ++aCount;        /* oppure a */
                break;
            case 'B': case 'b':  /* la votazione è stata
                B */ ++bCount;        /* oppure b */
                break;
            case 'C': case 'c':  /* la vo;tazione è stata
                C */ ++cCount;        /* oppure c */
                break;
            case 'D': case 'd':  /* la votazione è stata
                D */ ++dCount;        /* oppure d */
                break;
            case 'F':  case 'f':  /*la votazione è stata
                 F */ ++fCount;        /*oppure f */
            break;
            case '\n': case' ': /*ignorare questi caratteri
            nell'input*/
                    break;
            default: /* cattura tutti gli altri caratteri*/
            printf( "La votazione inserita è incorretta." );

            break;
            }
printf( "\nInserire una nuova votazione.\n" );

}

printf( "\nI totali per ciascuna votazione sono:\n" );

printf( "A: %d\n",aCount          );

printf( "B: %d\n", bCount         );

printf( "C: %d\n", cCount         );

printf( "D: %d\n",dCount          );

printf( "F: %d\n", fCount         );


return 0;

}
```

Lezione 6: Astrazione Funzionale/Funzioni

Divide et impera è strategia di sviluppo che consiste nel costruire programmi a partire da piccoli pezzi o componenti dove ciascun pezzo è meglio gestibile rispetto al programma iniziale.

Le funzioni in C permettono di modulare un programma combinando funzioni definite dall'utente con funzioni di libreria. Le librerie standard del C hanno una grande varietà di funzioni che facilitano il lavoro del programmatore e permettono di estendere il set di operatori del C

I vantaggi delle funzioni in C permettono uno sviluppo più facile del programma (grazie al divide et impera), inoltre permettono una riusabilità del software usando funzioni esistenti come building blocks per nuovi programmi. (Astrazione: si nascondono i dettagli interni tramite le funzioni di libreria). Inoltre se dobbiamo svolgere le stesse operazioni in due momenti diversi possiamo utilizzare nuovamente la funzione evitando la ripetizione del codice.

Astrazione Funzionale

Innanzitutto di procede con la creazione di unità di programma (nome assegnato ad un gruppo di istruzioni che in sequenza logica implementano un'operazione non presente nel Linguaggio di Programmazione), le unità di programma possono essere Funzioni o Procedure.

Funzione: Astrazione della nozione di operatore su un TD primitivo o definito dall'utente. Può essere attivata durante la valutazione di una qualsiasi espressione. Restituisce un valore:

<tipo del risultato> <nome> (<lista di argomenti>)

Procedura: Astrazione della nozione di istruzione su un TD primitivo o definito dall'utente. Una procedura è di fatto un'istruzione complessa e può essere attivata ovunque. Non restituisce alcunché:

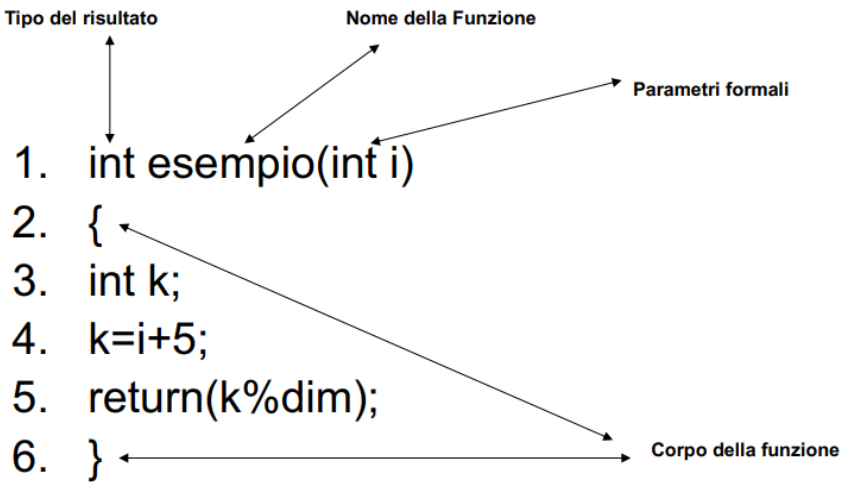
void <nome> (<lista di argomenti>)

Funzione e Procedure sono analoghe dal punto di vista sintattico e per ciò che concerne le regole che governano la loro valutazione:

- Scopo di una Procedura è di modificare il contenuto di locazioni di memoria.
- Scopo di una Funzione è restituire un valore all'interno dell'espressione dove è stata invocata

Dichiarazione della funzione (o procedura)

- Il nome
- I parametri formali
- Il corpo, costituito da dichiarazioni locali e da una sequenza di istruzioni
- Un tipo per il risultato



Binding

La dichiarazione di una variabile è definita **binding (legame)**, all'atto della dichiarazione il binding associa al TD tutte le informazioni necessarie per il suo utilizzo. Il ciclo di vita di queste informazioni legate alle variabili nella dichiarazione è equivalente al tempo di attivazione del programma (funzione o procedura) in cui compare la dichiarazione.

```
1. int esempio(int i)
2. {
3. int k;
4. k=i+5;
5. pippo(k);
6. return(k%dim);
7. }
```

- Il *ciclo di vita* di k inizierà ogni volta che ***esempio*** sarà invocato e terminerà nel momento in cui la funzione restituisce il valore (step 5).
- I valori che k assume durante le varie chiamate sono indipendenti tra di loro

Tutte le occorrenze del nome di un binding sono dette essere nell'ambito del binding. Il binding di un nome X è visibile in un punto del programma se un'occorenza di X in quel punto ricadrebbe nel binding medesimo.

- Nome globale
  - Visibile a tutte le funzioni (o procedure)
- Nome locale
  - Visibile solo nella funzione in cui è stato dichiarato

```
1. int esempio(int i)
2. {
3. int k;
4. k=i+5;
5. return(k%dim);
6. }
```

- dim→globale
- i,k→locale

Il legame tra parametri si realizza durante la chiamata di una funzione.

- I parametri formali
  - int exp(int base, int espon);
    - base ed espon sono parametri formali
    - Un *parametro formale* indica che un valore deve essere passato alla funzione e corrisponde ad una variabile *locale* alla funzione
- I parametri attuali

```
for (x = 1; x <= 10; x++)
        printf("%d ", exp(2, x));
```

  - 2 e x sono *parametri attuali*
  - *Un parametro attuale* è il valore effettivamente passato alla funzione. Questo valore viene assegnato al corrispondente parametro formale *all'atto della chiamata della funzione*

Classificazione Legami

Vi sono due tipi di legami**, legame per valore**, dove si crea una copia della variabile trasmessa e non si altera il valore originale al termine dell'esecuzione, avendo così una gestione locale. Il **legame per riferimento** invece trasmette l'indirizzo della variabile, quindi ogni modifica ha effetto sulla variabile stessa.

tipo-del-valore-restituito nome_funzione **(** *lista-parametri* **)**
{
    dichiarazioni e istruzioni
}

- Nome-funzione : qualsiasi identificatore valido
- Tipo-del-valore-restituito: tipo del risultato
  - **void** – la funzione non restituisce nulla
- Lista-parametri : lista separata da virgole, dichiara il tipo e il numero dei parametri
  - **void** – la funzione non riceve nulla

```c
/*
   Trovare il massimo tra tre interi */
#include <stdio.h>

int massimo( int, int, int );
int pippo ( int, float, int, int );
int main()
{
   int a, b, c;

   printf( "Inserisci tre interi: " );
   scanf( "%d%d%d", &a, &b, &c );
   printf( "Il massimo è: %d\n", massimo( a, b, c ) );/*Chiamata*/

   return 0;
}

/* definizione della funzione massimo */
int massimo( int x, int y, int z )
{
   int max = x;

   if ( y > max )
      max = y;

   if ( z > max )
      max = z;

   return max;
}
```

```
Inserisci tre interi: 22 85 17
Il massimo è: 85
```

Visibilità delle variabili

Ogni variabile è 'locale' alla funzione in cui è dichiarata infatti funzioni diverse possono dichiarare variabili distinte con lo stesso nome. L'allocazione e la deallocazione di queste variabili è automatica.

Risolvere problemi con la decomposizione top-down

La **decomposizione top-down** analizza il problema scomponendo in sottoproblemi, ripetendo la decomposizione in maniera gerarchica fino ad arrivare a sottoproblemi di dimensione gestibile e risolvendoli progettando funzioni specifiche.

I vantaggi della decomposizione top-down:

- Programmi concettualmente ordinati e meglio documentati
- Ciascun sottoproblema può essere risolto in maniera indipendente (Modularità)
- È più facile intervenire su parti del programma in maniera circoscritta (Manutenibilità)

Lezione 7: Strutturare i Dati: gli Array

È importante poter aggregare i dati per trattarli più comodamente in modo collettivo (Esempio: descrivere come operare sulla singola componente e reiterare il procedimento su tutte). Le strutture dati permettono una astrazione di dati, ovvero costruzione di nuovi tipi di dati (dominio dei valori e relativi operatori) a partire da quelli di base. Il C offre la possibilità di definire **array**, **record**, **tipi enumerati**, **file**, **strutture a puntatori**, usando tipi già esistenti.

Finora abbiamo usato solo **variabili scalari** che possono contenere un solo elemento di dati.

Il C permette di usare anche **variabili aggregate**, che possono contenere insiemi di valori. Esistono due tipi di variabili aggregate: i **vettori** (arrays) e le **strutture** (structures).

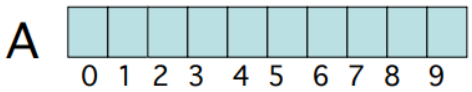
In C un Array è una collezione di variabili omogenee accessibili attraverso un indice. Un array è caratterizzato da:

- Tipo delle componenti (omogenee)
- Taglia (numero fissato di elementi)

– **Esempio**:

double A[10];

dichiara un array di double con 10 componenti



Array monodimensionali

Un **array** è una struttura dati che contiene un certo numero di elementi di dati dello stesso tipo. Questi valori, detti elementi dell'array, possono essere selezionati individualmente usando la loro posizione nell'array.

Gli elementi di un array monodimensionale "a" sono organizzati uno dopo l'altro in quella che possiamo pensare una riga (o una colonna)

Per dichiarare un array dobbiamo specificare il tipo e il numero degli elementi:

int a[10];

Il tipo può essere un tipo qualsiasi mentre il numero può essere specificato da una espressione intera costante. Usare una macro per definire la lunghezza di un array è una buona pratica:

#defina N 10

…

int a[N];

Per accedere ad un singolo elemento dell'array si può usare l'indice dell'elemento. Gli elementi di un array di lunghezza n sono indicizzati dai numeri da 0 ad n – 1, quindi se a è un array di lunghezza 10, i suoi elementi sono a[0], a[1], …, a[9]:

Espressioni del tipo a[i] sono ***lvalues***, quindi possono essere usate come normali variabili:

a[0] = 1;
printf("%d\n", a[5]);
++a[i];

In generale, se un array contiene elementi di tipo T, allora ogni elemento dell'array viene trattato come una variabile di tipo T.

Molti programmi usano dei cicli for nel cui corpo viene eseguita un'operazione su un elemento dell'array: ad ogni ciclo di accede ad un elemento.

Esempi con un array a di lunghezza N:

```
for (i = 0; i < N; i++)
  a[i] = 0;           /* clears a */

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);   /* reads data into a */

for (i = 0; i < N; i++)
  sum += a[i];        /* sums the elements of a */
```

Il compilatore non fa nessun controllo sugli indici utilizzati; se usiamo un indice che va al di fuori dei limiti il comportamento è impredicibile.

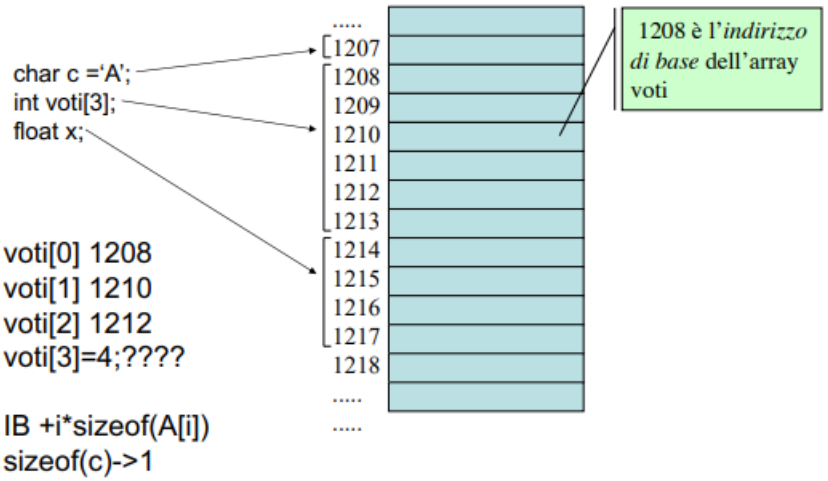
```
#include <stdio.h>
#define NUMEL  250

int  main (void)
{
     double valori[NUMEL];
     int i;
     for (i=0; i<NUMEL; i++)
          scanf("%lf", &valori[i]);
     for (i=NUMEL - 1; i>=0; i--)
          printf("Valore %d: %f\n", i, valori[i]);
     return 0;
}
```

La *taglia*: dell'array deve essere una costante intera

valori[i] è un *selettore*: identifica l'elemento di indice *i*

Rappresentazione interna dei dati

La memoria è organizzata in bytes, l'aggregato minimo di bit accessibile tramite un indirizzo. Quando una variabile viene allocata in memoria, un adeguato numero di bytes viene riservato ad essa, a partire da un certo indirizzo.



L'operatore sizeof()

sizeof(TIPO) ritorna il numero di bytes assegnati a TIPO.

sizeof(ESPRESSIONE) ritorna il numero di bytes assegnati al tipo del valore di ESPRESSIONE.

Esempi: sizeof(int) = 2 (nel nostro esempio)
sizeof(x/2.4 + 1) = 4

L'indirizzo di una componente di array A[i] si calcola con la formula: IB + i * sizeof(A[i]) dove IB è l'indirizzo di base di A.

Inizializzazione

Un **array**, come una qualsiasi altra variabile, può essere inizializzato quando lo si dichiara. La forma più comune per una inizializzatore di array è una lista di esperessioni costanti fra parantesi graffe e separate dalla virgola:

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Se l'inizializzatore è più piccolo dell'array, ai rimanenti elementi verrà assegnato il valore 0:

int a[10] = {1, 2, 3, 4, 5, 6};

/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */

Possiamo sfruttare questo questa caratteristica per inizializzare a 0 l'intero array scrivendo un solo 0:

int a[10] = {0};

Non è permesso usare un inizializzatore più lungo all'array.

Se c'è l'inizializzatore la lunghezza può essere omessa, ovviamente il compilatore considererà la lunghezza dell'inizializzatore come lunghezza dell'Array.

Int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Passaggio di array come parametri

```
#include <stdio.h>
#define NUMEL  10

int  main (void)
{
     int lista [NUMEL];
     GetLista(lista);
     Reverse(lista);
     PrintLista(lista);
     return 0;
}
```

```
void GetLista(int pippo [ ])
{
     int i;
     for (i=0; i<NUMEL; i++)
          scanf("%d", &pippo[i]);
}

void PrintLista(int lista[ ])
{
     int i;
     printf("\nLa lista inversa è:\n");
     for (i=0; i<NUMEL; i++)
          printf("%d", lista[i]);
}
```

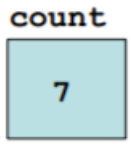
```
void Reverse(int lista[ ])
{
     int i, temp;
     for (i=0; i<NUMEL/2; i++)
     {
          temp = lista[i];
          lista[i] = lista[NUMEL-1- i];
          lista[NUMEL-1- i] = temp;
     }
}
```

Dato che in C il passaggio dei parametri avviene per valore, come mai il programma dell'esempio funzione? Perché ciò che viene passato è l'indirizzo di base del vettore, ogni modifica effettuata all'interno della funzione chiamata ha effetto sull'argomento della chiamata.

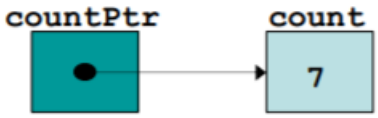
Lezione 8: I puntatori

I **puntatori** sono una delle caratteristiche più potenti del C, ma sono difficili da padroneggiare. Si usano per simulare la chiamata per riferimento e hanno una stretta correlazione con vettori e stringhe.

Le variabili puntatore contengono indirizzi come valori, a differenza delle variabili normali che contengono uno specifico valore (riferimento diretto)



I puntatori contengono l'indirizzo di una variabile che ha uno specifico valore (riferimento indiretto).



Il **deferimento** è invece il riferimento a un valore per mezzo di un puntatore.

Dichiarazione e inizializzazione di una variabile puntatore

Il tipo puntatore è un classico esempio di tipo derivato: infatti non ha senso di parlare di un tipo puntatore generale, ma occorre sempre specificare a quale tipo esso punta.

Dichiarazione di una variabile riferimento o puntatore:

tipo_base *var_punt;

dove tipo_base è uno dei tipi fondamentali già introdotti: char, int, float e double.

var_punt è definita come una variabile di tipo puntatore a tipo_base.

var_punt è creata per mantenere l'indirizzo di variabili di tipo tipo_base.

Dichiarazioni di puntatori

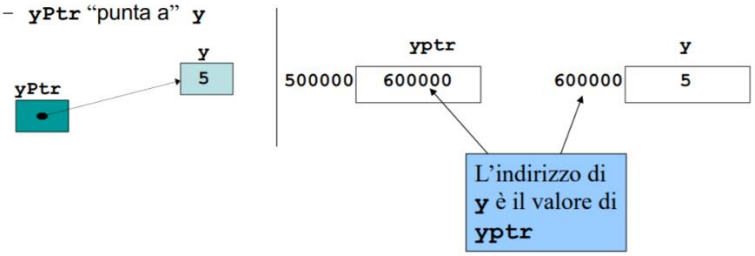
- * usato con variabili puntatore. (int *myPtr;)
- Dichiara un puntatore a un valore di tipo int. (Si legge “myPtr è un puntatore a tipo int”)
- Puntatori multipli richiedono * multipli. (int * myPtr1, *myPtr2”).
- È possibile dichiarare puntatori a qualsiasi tipo di dato.
- I puntatori possono essere inizializzati a 0, a NULL (costante simbolica definita in molti file di intestazione) oppure a un indirizzo. 0 e NULL rappresentano un riferimento a nessun dato, e tra i due si preferisce usare **NULL**.

Inizializzazione puntatori

& (operatore di indirizzo), restituisce l’indirizzo dell’operando. L’operando dell’operatore di indirizzo deve essere necessariamente una variabile, l’operatore di indirizzo non può essere applicato a costanti, a espressioni o a variabili dichiarate con la specifica di classe di memoria *register*.

```
int y = 5;
int *yptr;
yptr = &y; //a yPtr viene assegnato l'indirizzo di y
```

– `yPtr` “punta a” `y`



Operatori sui Puntatori

Il simbolo *** è l’operatore di deriferimento** o di risoluzione del riferimento. Il deriferimento restituisce il valore dell’oggetto puntato dal suo operando, che deve essere un puntatore.

```
int y = 5, x;
int *yPtr;
yPtr = &y;
x = *yptr; /*x=y;*/
```

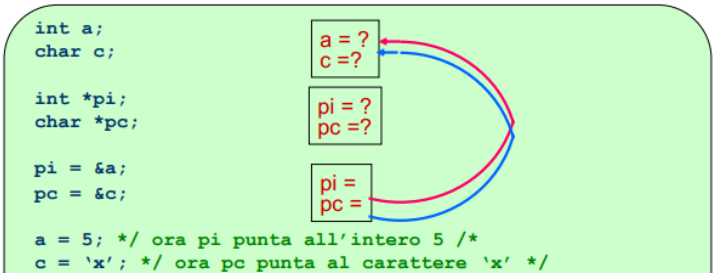
`*yptr` restituisce il valore di `y` (perché `yptr` punta a `y`)

Questo operatore può essere usato anche per assegnare valori alle variabili puntate.

```
*yptr = 7; // modifica y in 7
```

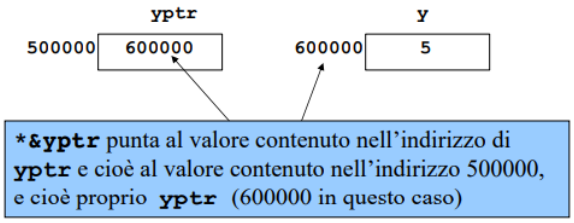
Esempio di utilizzo dei puntatori

```
int a;
char c;
int *pi; /*pi è una variabile di tipo puntatore a int */
char *pc; /* pc è una variabile di tipo puntatore a char */
pi = &a; /* pi è inizializzata con l'indirizzo di a */
pc = &c; /* pc è inizializzata con l'indirizzo di c */
a = 5;
c = 'x';
```



& e * sono l'uno l'inverso dell'altro, quindi si annullano tra di loro.

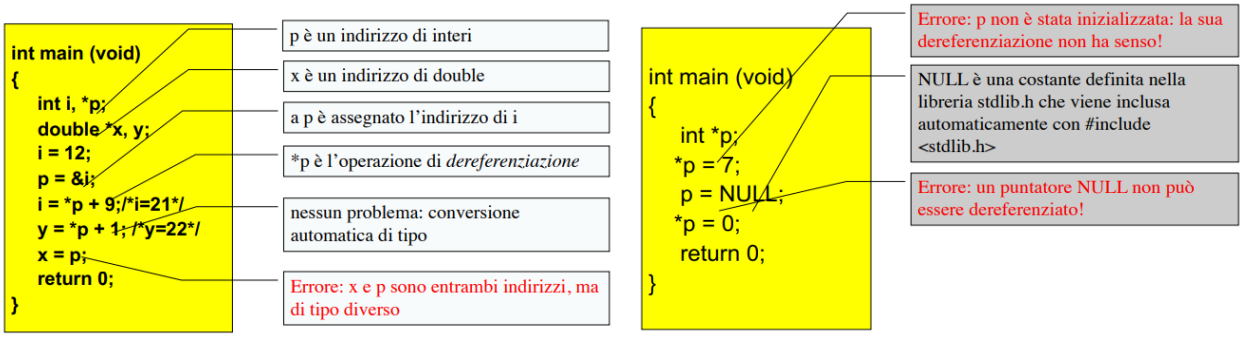
```
*&yptr  ->  * (&yptr) -> * (indirizzo di yptr)->
  restituisce l'alias di ciò a cui punta l'operando
  -> yptr
```



Analogamente:

```
&*yptr -> &(*yptr) -> &(y) -> restituisce
  l'indirizzo di y, che è yptr -> yptr
```

Gli indirizzi come dati



Specifica di conversione %p

La specifica di conversione **%p** restituisce la locazione di memoria cui si riferisce, convertendola in un formato definito dall'implementazione (molto spesso un intero esadecimale).

```
      int *ptr;
      int x = 1123;
      ptr = &x;
      printf("l'indirizzo  di  x  è  %p\n",
&x);
      printf("il  valore  di  ptr  è  %p\n",
ptr);
```

```
l'indirizzo di x è 001F2BB4
il valore di ptr è 001F2BB4
```

La chiamata per riferimento delle funzioni

Nella chiamata per riferimento con puntatori come argomenti, si passa l'indirizzo dell'argomento usando l'operatore &, ciò consente di fare modifiche alla reale locazione di memoria che contiene il valore passato dalla funzione chiamante. Gli array vengono passati senza & poiché il nome dell'array è già un puntatore.

L'operatore * è usato come soprannome per la variabile all'interno della funzione.

```
void double(int *number) /* number conterrà un indirizzo */
 {
  *number = 2 * (*number); /*  *number contiene un intero */
 }
```

Supponiamo che la variabile x di tipo int debba essere passata per riferimento ad una funzione foo:

- Nella chiamata alla funzione foo si passa l'indirizzo di c usando l'operatore &: foo(&x)
- Nella definizione della funzione:
  - Il parametro deve essere un puntatore ptr: Es. void foo (int *ptr)
  - Nel corpo si utilizza l'operatore di deferimento *: Es. *ptr = 5;

Se dobbiamo invece passare un array a ad una funzione foo dobbiamo ricordarci che:

- a è equivalente a &a[0]
- Cioè a è un puntatore
- int *a è equivalente di int [a]

Quindi nella chiamata alla funzione foo non si utilizza & poiché il nome dell'array è già un indirizzo: foo(a).

Nella definizione della funzione:

- Il parametro deve essere un array void foo (int array[]) oppure void foo (int *array)
- Nel corpo non si utilizza l'operatore di deriferimento *. Es. array[3] = 5;

Espressioni con i puntatori e aritmetica dei puntatori

Sui puntatori è possibile eseguire **operazioni aritmetiche** come incremento/decremento del puntatore (++ o --), aggiungere o sottrarre un intero a un puntatore (+, +=, -, -=) e inoltre si può sottrarre un puntatore da un altro puntatore.

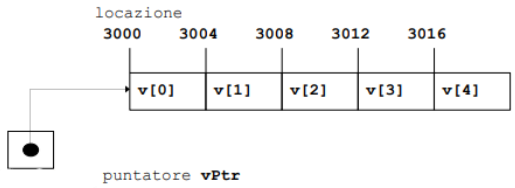
Le operazioni aritmetiche sui puntatori hanno significato solo se eseguite su un vettore, infatti solo in quel caso si può essere sicuri di operare su oggetti immagazzinati in locazioni di memoria consecutive.

Un array di 5 elementi **int** in una macchina che implementa gli interi con 4 byte

- **vPtr** punta al primo elemento **v[0]** alla locazione **3000**. (**vPtr = 3000**) Potrà essere inizializzato con una delle seguenti istruzioni

```
vPtr = &v[0];
vPtr = v;
```

- **vPtr +=2;** pone **vPtr** a **3008**
  - **vPtr** punta a **v[2]** (incrementato di 2), ma la macchina ha **int** di 4 byte.



Sottrazione tra puntatori

- Restituisce il numero di elementi compresi tra l'uno e l'altro.

```
vPtr2 = &v[2];
vPtr = &v[0];
vPtr2 - vPtr = 2.
```

Confronto tra puntatori (<, ==, >)

- Per esempio per vedere quale di due puntatori punta all'elemento del vettore con l'indice più alto.
- Anche per vedere se un puntatore è **NULL**

È possibile assegnare un puntatore a un altro purché dello stesso tipo. Se non sono dello stesso tipo si può applicare un operatore di conversione per convertire il puntatore a destra dell'assegnamento nel tipo di quello a sinistra.

Eccezione: **puntatore a void** (tipo void *)

Il puntatore a void è detto puntatore generico, che rappresenta qualsiasi tipo, a una variabile puntatore a void può essere assegnato a qualsiasi tipo di puntatore, nessuna conversione è necessaria per convertire un puntatore in un puntatore a void. Viceversa, un puntatore a void può essere assegnato a tutti gli altri tipi di puntatori senza bisogno di conversioni. Non è possibile risolvere il riferimento di un puntatore a void.

La relazione tra puntatori e vettori

I vettori e i puntatori sono strettamente **correlati**, il nome di un vettore può essere infatti considerato come un puntatore costante e i puntatori possono essere utilizzati per svolgere qualsiasi operazione che coinvolga gli indici di un vettore.

Supponiamo che sia stato dichiarato un vettore b[5] e un puntatore bPtr.

bPtr = b;
Il nome del vettore è in realtà l'indirizzo del primo elemento

OPPURE

bPtr = &b[0];
Assegna esplicitamente a bPtr l'indirizzo del primo elemento.

Prendiamo per esempio l'elemento b[n], vi si può accedere tramite espressione con puntatore *(bPtr + n). Il vettore stesso può usare l'aritmetica dei puntatori. b[3] equivale a *(b + 3) e a *(bPtr + 3). Anche l'indirizzo di un elemento di un vettore si può ottenere con la stessa notazione: &b[3] equivale a bPtr + 3.

In generale tutte le espressioni con gli indici potranno essere convertite in quelle con puntatore e offset.

I puntatori possono essere usati in combinazione con gli indici esattamente come i vettori (notazione con puntatore e indice): bPtr[3] punterà all'elemento b[3].

Due modi per inizializzare un array

Primo modo

```
char tab [100];
int i;
for (i=0; i < 100; i++)
tab[i] = 'k';
```

Secondo modo

```
char tab [100];
char * s;
int i;
s = tab;
for (i=0; i < 100; i++)
*s++ = 'k';
```

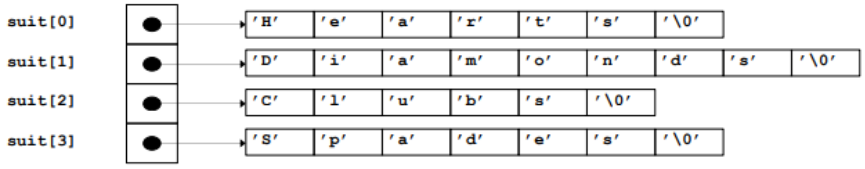
L'istruzione `*s++ = 'k';` opera nel seguente modo:
- copia k nella locazione di memoria puntata da s;
- poi incrementa s di un elemento.

Vettori di puntatori

I vettori possono contenere dei puntatori (Es. Vettore di Stringhe).

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades" };
```

- Stringa: puntatore al primo carattere della stringa
- **char *** - ogni elemento di **suit** è un puntatore a un **char**
- Le stringhe non sono realmente nel vettore - soltanto i *puntatori* alle stringhe sono nel vettore



- Il vettore **suit** ha una dimensione fissa, ma le stringhe possono essere di qualsiasi lunghezza.

Lezione 9: Stringhe e Puntatori, Array di Puntatori (cenni)

Operazioni sulle stringhe letterali

Possiamo usare una **stringa letterale** dovunque sia permesso usare un char*:

char *p;
p = “abc”;

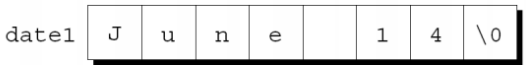
Questo assegnamento fa puntare p al primo carattere della stringa.

Inizializzazione di una stringa

Una stringa variabile può essere inizializzata al momento della dichiarazione:

chat date1[8] = “June 14”;

Il compilatore 8 byte per l’array ed inserirà il carattere di fine stringa.

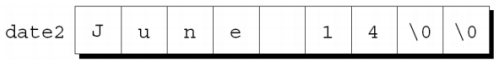


In questo contesto, “June 14” non è una stringa letterale ma un’abbreviazione dell’inizializzatore per l’array.

Se l’inizializzatore ha meno caratteri della grandezza dell’array, il compilatore riempie il resto con caratteri nulli:

char date2[9] = “June 14”;

La stringa date2 sarà quindi:



La dichiarazione char date[] = “June 14”; dichiara date come un **array**.

La dicharazione char *date = “June 14”; dichiara date come un **puntatore**.

Entrambe le variabili possono essere usate come stringhe.

Non è possibile però modificare una stringa letterale:

char *p = “abc”;

*p = ‘d’; */*** WRONG ***/*

Un programma che cerca di cambiare una stringa letteralmente può andare in crash o avere comportamenti non definiti.

Tuttavia ci sono differenze importanti fra le due versioni della variabile date.

- Nella versione **“array”**, i caratteri memorizzati nella variabile possono essere modificati. Nella versione **“puntatore”**, date punta ad una stringa letterale che non può essere modificata.
- Nella versione **“array”**, date è il nome dell’array. Nella versione **“puntatore”**, date è una variabile che punta ad una stringa.

La dichiarazione char *p; non alloca la memoria. Prima di poter usare p come stringa, dobbiamo farla puntare ad un array.

Una possibilità è quella di farla puntare ad una stringa variabile:

```
char str[STR_LEN+1], *p;
p = str;
```

Un’altra possibilità è quella di **allocare dinamicamente** la memoria necessaria.

Usare un puntatore non inizializzate può avere conseguenze disastrose, ecco un tentativo maldestro per la costruzione della stringa “abc”:

```
char* p;
p[0] = ‘a’;        /*** WRONG ***/
p[1] = ‘b’;        /*** WRONG ***/
p[2] = ‘c’;        /*** WRONG ***/
p[3] = ‘\0’;       /*** WRONG ***/
```

Poiché la variabile p non è stata inizializzata il comportamento del programma non è definito.

La specifica di conversione %s permette di usare una stringa con printf:

```
char str[] = “Are we having fun yet?”;
printf(“%s\n”, str);
```

L’output sarà

Are we having fun yet?

printf scrive i caratteri di una stringa una alla volta fino a che non trova il carattere nullo. Per stampare una parte di una stringa si può usare la specifica di **conversione %.ps** dove p è il numero di caratteri da visualizzare.

L’istruzione printf(“%.6s\n”, str); stamperà Are we.

Esistono vari modi per memorizzare una stringa in ogni riga

```
Char planets[][8] = {“Mercury”, “Venus”, “Earth”, “Mars”, “Jupiter”, Saturn”, “Uranus”, “Neptune”,
                     “Pluto”};
```

Il numero di righe nell'array può essere omesso, ma occorre specificare il numero di colonne.

L'array planets contiene però vari byte non utilizzati:

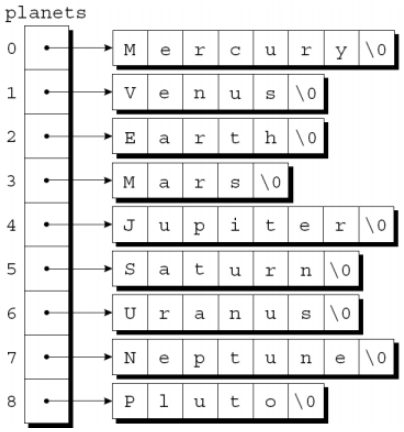
	0	1	2	3	4	5	6	7
0	M	e	r	c	u	r	y	\0
1	V	e	n	u	s	\0	\0	\0
2	E	a	r	t	h	\0	\0	\0
3	M	a	r	s	\0	\0	\0	\0
4	J	u	p	i	t	e	r	\0
5	S	a	t	u	r	n	\0	\0
6	U	r	a	n	u	s	\0	\0
7	N	e	p	t	u	n	e	\0
8	P	l	u	t	o	\0	\0	\0

Spesso quando dobbiamo gestire molte stringhe ci troveremo in questa situazione, con alcune stringhe più lunghe e altre più corte.

Potremmo usare un **array irregolare (ragged array)** in cui le righe possono avere lunghezze diverse. Possiamo "simulare" tale array utilizzando un array alle stringhe:

char *planets[] = {"Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto"};

Ecco come verranno memorizzate le stringhe planets:



Per accedere ad una delle stringhe dobbiamo usare l'indice per selezionare il puntatore dell'array planets, dopo di che possiamo accedere ai caratteri della stringa come si fa con un normale array bidimensionale. Ecco un ciclo che cerca le stringhe che iniziano per M:

```
for(i = 0; i < 9; i++)
    if (planets[i][0] = 'M')
        printf("%s begins with M\n", planets[i]);
```

Lezione 10: Allocazione Dinamica della Memoria, Array di Puntatori

Allocazione dinamica della memoria

calloc() e malloc() sono due funzioni presenti nella libreria stdlib.h. calloc() = allocazione contigua e malloc() = allocazione della memoria, consentono di allocare dinamicamente array(sia calloc() che malloc()) e record(malloc()).

*calloc() void *calloc (size_t n, size_t el_size);*

Riceve due parametri di tipo size_t (che rappresenta un tipo senza segno) e alloca uno spazio contiguo di n elementi (1° parametro) ciascuno di dimensioni el_size (2° parametro) inizializzato a 0, restituendo un puntatore void *.

```
int *vect; /*si dichiara un puntatore che poi sarà usato come vettore*/
int n;
…….
printf("Inserisci la dimensione del vettore: \n")
scanf("%d",&n);
vect=calloc(n,sizeof(int)); /*calloc(n,el_size) crea lo spazio per vect*/
……
free(vect); /*calloc() e malloc() non restituiscono lo spazio occupato*/
```

*malloc() void *malloc(size_t size);*

malloc() riceve un parametro di tipo size_t e alloca un blocco di memoria di size byte non inizializzato, restituendo un puntatore a void *. `vect=malloc(n*sizeof(int));equivale a vect=calloc(n,sizeof(int));`

I Record in C: le strutture.

Le strutture sono collezioni di variabili correlate (aggregati) da un unico nome, le strutture possono contenere variabili di tipo differente. Sono comunemente usate per definire record da memorizzare nei file e se combinate con puntatori, possono servire a creare tipi di dati strutturati come liste a puntatori, pile, code e alberi.

Le strutture sono utili in quanto spesso si devono rappresentare entità che sono collezioni di oggetti diversi.

Titolo
Autore
Editore
Anno
Prezzo

Variabili di tipo diverso

Titolo	Autore	Editore	Anno	Prezzo

Una struttura è una collezione di una o più variabili normalmente di tipo diverso raggruppate sotto lo stesso nome.

Specificatore del tipo di dato

Etichetta della struttura

```
typedef struct Libro {
    char    titolo[40];
    char    autore[20];
    char    editore[20];
    int     anno;
    float   prezzo;
} Libro;
```

Campi della Struttura

Definire un tipo di record

Sintassi della dichiarazione:

```
typedef struct NOMETIPO {
      DICHIARAZIONE DI CAMPI
} NOMETIPO;
```

- **struct** introduce la definizione della struttura **corpoCeleste**
- **corpoCeleste** è il *nome della struttura* (l'etichetta) ed è usata per dichiarare variabili del *tipo della struttura*
- **corpoCeleste** contiene un membro di tipo **char[ ]** e tre di tipo **float**

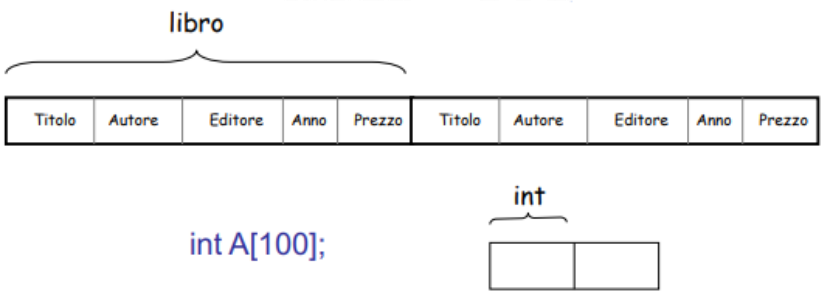
```
typedef struct corpoCeleste {
    char nome[16];
    float diametro;
    float giorno;
    float anno;
} corpoCeleste;
```

```
typedef struct corpoCeleste {
    char nome[16];
    float diametro;
    float giorno;
    float anno;
} corpoCeleste;

int main (void)
{
    corpoCeleste  terra;
    ........
}
```

```
int main (void)
{
    corpoCeleste  terra, luna, sole;
    double numOre;
    terra.nome = "Terra";
    terra.diametro = 12800;
    terra.giorno = 23.56;
    terra.anno = 365.24;
    numOre = terra.giorno * terra.anno;
    .....
    .....
}
```

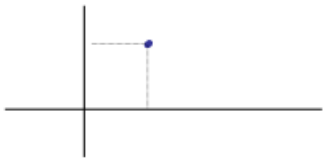
Array di Strutture

Come rappresentare una collezione di libri? Gli elementi di un array possono essere delle strutture.

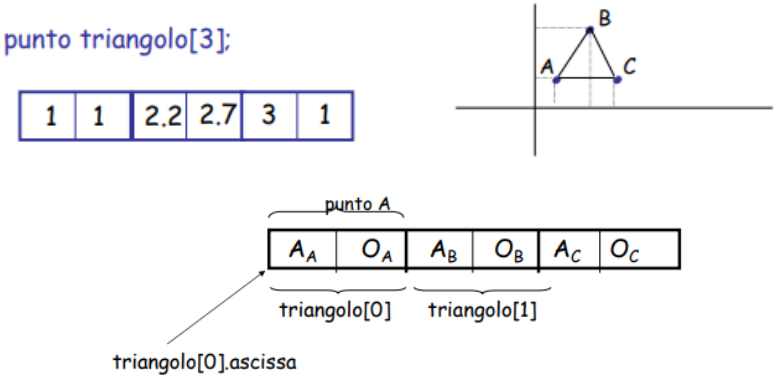


Si consideri una struttura rappresentante un punto:

```
typedef struct punto{
        float ascissa;
        float ordinata;
}punto;
```

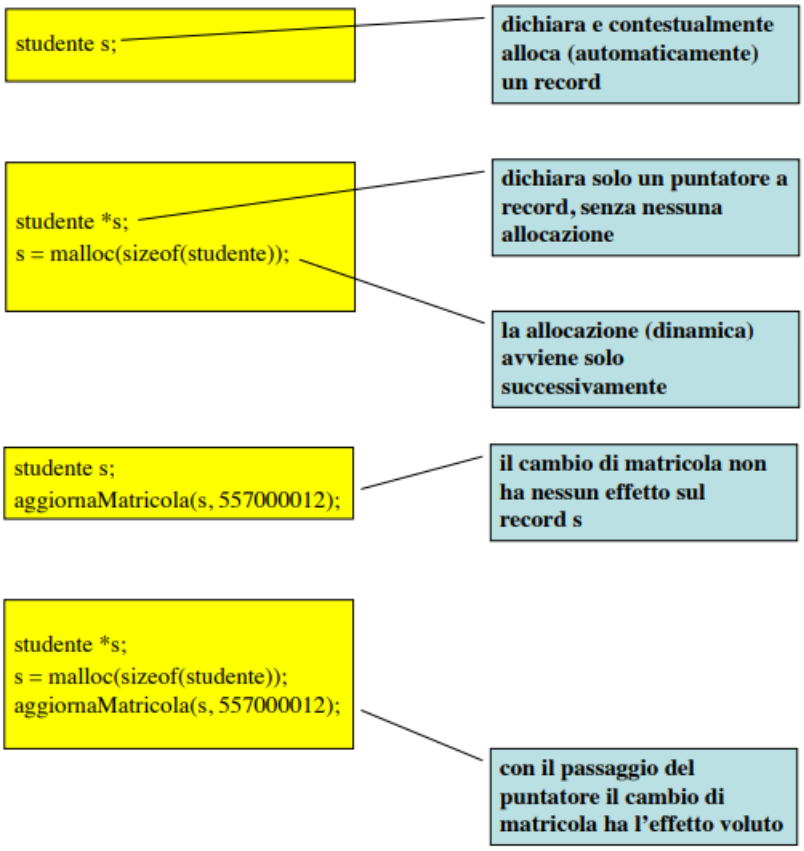


E si rappresenti un triangolo come array di 3 punti (i suoi vertici):



Puntatori a Record

I puntatori a record possono essere usati per allocare record dinamicamente, passare record con call-by-reference e creare strutture a puntatori.



Puntatori nulli

Un esempio di uso di malloc:

```
p = malloc(10000);
if (p==NULL) {
/* allocation failed; take appropriate action */
}
```

NULL è una macro (definita in vari file di intestazione) che rappresenta il valore speciale che indica un puntatore nullo. Spesso si chiama la funzione e si controlla il valore del puntatore in un'unica espressione:

```
if((p=malloc(10000)) == NULL) {
/* allocation failed; take appropriate action */
}
```

Il test sul valore di un puntatore restituisce vero o falso seguendo le stesse regole dei numeri:

- Un puntatore non nullo è "vero"
- Un puntatore nulla è "falso"

Quindi if (p == NULL) equivale a if (!p), e if (p != NULL) equivale a if(p).

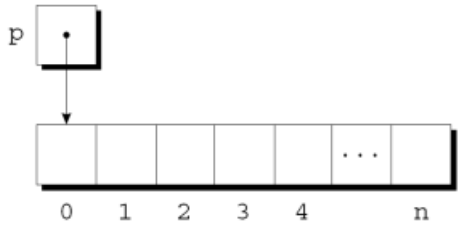
L'allocazione dinamica è spesso utile per le stringhe dato che le stringhe sono array di caratteri e spesso è difficile prevedere quanti caratteri saranno effettivamente utilizzati. Se decidiamo una dimensione fissa in fase di programmazione dobbiamo usare la dimensione massima che si prevede, invece allocando dinamicamente la memoria possiamo posporre la decisione a run-time.

Usare malloc() per una stringa

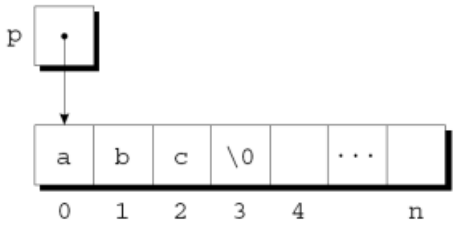
Ecco una chiamata malloc() che alloca memoria per una stringa di n caratteri: p = malloc(n+1); dove p è una variabile char *.

Ogni carattere della stringa richiede un byte di memoria, il bute addizionale serve per il carattere nullo che viene usato per terminare una stringa. Alcuni programmatori preferiscono esplicitare il cast del valore di ritorno di malloc, anche se non è necessario (p = (char *) malloc (n + 1);).

La memoria allocata da malloc non viene inizializzata, quindi p punterà ad una zona di memoria di cui non sappiamo il contenuto:



Chiamare strcpy è un modo per inizializzare la stringa: strcpy(p, "abc"); adesso i primi 4 byte avranno come contenuto a, b, c, e \0.



L'allocazione dinamica permette di scrivere delle funzioni che restituiscono un puntatore a "nuove" stringhe. Supponiamo di dover scrivere una funzione che concatena due stringhe senza alterare le stringhe di input, la funzione misurerà la lunghezza delle stringhe di input e usando malloc allocherà la memoria necessaria a scrivere il risultato.

Funzioni come concat che allocano dinamicamente la memoria devono essere usate con molta cautela. Quando la stringa non serve più è necessario "liberare" la memoria allocata chiamando la funzione free. Se non lo facciamo la memoria occupata sarà persa e un'eccessiva memoria "persa" potrà determinare la mancanza di memoria.

Come fatto per le stringhe, possiamo allocare la memoria necessaria a memorizzare un array dinamicamente, la stretta relazione tra array e puntatori ci permette di usare nello stesso modo un array allocato staticamente e uno allocato dinamicamente.

Sebbene si possa utilizzare malloc, spesso si preferisce calloc in quanto la memoria allocata viene inizializzata (a 0). La funzione realloc invece ci permette di "cambiare" la dimensione dell'array.

Supponiamo di dover usare un vettore di n interi, e che n venga calcolato durante l'esecuzione.

Possiamo dichiarare un puntatore: int *a; quando il valore n è noto il programma può chiamare malloc per allocare la memoria necessaria: a = malloc(n * sizeof(int));

Bisogna sempre usare sizeof per calcolare lo spazio necessario per ogni elemento.

Liberare la memoria allocata dinamicamente

Lo spazio di memoria allocato in maniera automatica viene anche liberato in maniera automatica, all'uscita dalla funzione. Lo spazio allocato dinamicamente non viene deallocato all'uscita dalla funzione ma deve essere liberato manualmente quando non è più necessario.

void free (void *p);

Che succede quando la memoria è esaurita?

Quando l'heap di memoria a disposizione del programma è pieno, malloc() e calloc() ritornano un valore NULL, bisogna quindi controllare sempre il valore restituito prima di procedere.

```
int *   leggiNvalori (int numval)
{
    int *A, i;
    A = (int *) calloc(numval, sizeof(int));
    if (!A)
    {   printf("Spazio di memoria esaurito\n");
        return NULL;
    }
    for (i=0; i<numval; i++)
    {
        printf("Inserire valore (%d): ", i);
        scanf("%d", &A[i]);
    }
    return A;
}
```

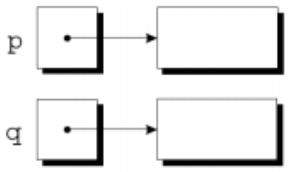
Le funzioni di allocazione della memoria (calloc(), malloc(), realloc()) ottengono la memoria da un blocco detto heap. Chiamare queste funzioni troppo spesso o richiedere blocchi di dimensioni enormi può portare al consumo dell'intero heap, ma può anche succedere di peggio: il programma continua ad allocare memoria ma ne perde traccia (perdendo i puntatori) e quindi di fatto spreca la memoria.

Deallocare la memoria

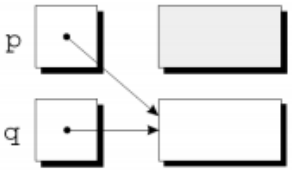
Esempio:

```
p = malloc(…);
q = malloc(…);
p = q;
```

Ecco la situazione dopo l'esecuzione delle prime due istruzioni:



Dopo l'assegnazione di q a p, la situazione diventa:



Non ci sono più puntatori al primo blocco che di fatto non potrà più essere usato e quindi è memoria sprecata (il che contribuisce al consumo dell'heap).

Un blocco di memoria non più utilizzabile è detto garbage (immondizia), un programma che genera garbage ha un memory leak (perdita di memoria). Alcuni linguaggio forniscono dei meccanismi per il recupero della memoria persa (garbage collection) ma non il C dove ogni programma ha la responsabilità di deallocare la memoria che non serve più usando la funzione free.

La funzione free

Prototipo: void free (void* p);

La funzione free ha bisogno di un puntatore ad un blocco di memoria da deallocare:

```
p = malloc(…);
q = malloc(…);
free(p);
p = q;
```

La funzione free libera il blocco di memoria puntato da p restituendolo all'heap.

L'uso di free crea un nuovo problema, quello del puntatore pendente. free(p) dealloca la memoria puntata da p ma non cambia p (ad esempio rendendolo nullo), se proviamo ad usare p dopo la deallocazione, il risultato non è prevedibile. Usare un puntatore pendente è un grave errore.

Individuare i puntatori pendenti può essere molto difficile in quanto più puntatori possono puntare allo stesso blocco di memoria e quando il blocco viene deallocato, tutti i puntatori che puntano al blocco diventano pendenti.

Lezione 11: Allocazione Dinamica della Memoria, Array di Puntatori, I Record

Gli elementi di un array possono essere di tipo puntatore, questo permette di allocare dinamicamente array bi-dimensionali.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

void riempi (int **, int);
void stampa (int **, int);

int main (void)
{
  int *A[MAX], i, n;
  for (i=0; i<MAX; i++)
  {
    printf("lunghezza riga %d: ", i);
    scanf("%d", &n);
    A[i] = calloc(n+1, sizeof(int));
    if (!A[i])
    {  printf("Spazio di memoria esaurito\n");
       return 0;
    }
    A[i][0] = n;
  }
  riempi(A, MAX);
  stampa(A, MAX);
  return 0;
}
```

```
void riempi (int **A, int m)
{
  int i, j;
  for (i=0; i<m; i++)
  {
      printf("\ninserire la riga %d\n", i);
      for (j=1; j<=A[i][0]; j++)
      {
          printf("prossimo valore:  ");
          scanf("%d", &A[i][j]);
      }
  }
}

void stampa (int **A, int m)
{
  int i, j;
  for (i=0; i<m; i++)
  {
      printf("\nriga %d:", i);
      for (j=1; j<=A[i][0]; j++)
          printf("%d\t", A[i][j]);
  }
}
```

Questo oltre un'allocazione dinamica di matrici, permette di passare le matrici come parametri senza specificare il numero di colonne.

```
double determinante(double M[ ][ ], int n);

double determinante(double **M, int n);
```

Errore: non è specificato il numero di colonne

problema risolto: passo alla funzione un array di puntatori

Passaggio di record come parametri

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

typedef struct punto {
            double x, y;
} punto;

punto creaPunto (double, double);
double distanza (punto, punto);

int main (void)
{
    double dist;
    punto origine, centro;
    origine = creaPunto(0,0);
    centro = creaPunto(3.1, 4.7);
    dist = distanza(origine, centro);
    return 0;
}

punto creaPunto (double x, double y)
{
   punto p;
   p.x = x;
   p.y = y;
   return p;
}
```

copia di record campo-per-campo

passaggio per valore

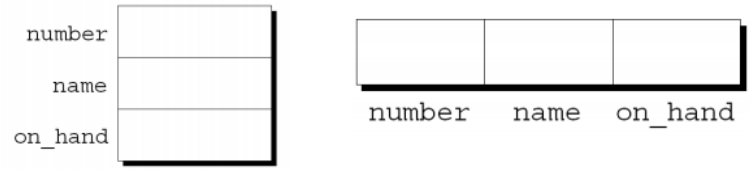
```
double distanza (punto p1, punto p2)
{
   double d, xdiff, ydiff;
   xdiff = p1.x - p2.x;
   ydiff = p1.y - p2.y;
   d = sqrt(xdiff*xdiff + ydiff*ydiff);
   return d;
}
```

Dichiarazione di strutture

Ogni struttura rappresenta un nuovo spazio dei nome (scope), i nomi dichiarati all'interno della struttura non andranno in conflitto con altri nomi (uguali) dichiarati nel programma. Nella terminologia del C, si dice che ogni struttura ha uno spazio dei nomi per i suoi membri.

Ad esempio queste dichiarazioni non creano conflitto di nomi:

```
typedef struct part{
  int number;
  char name[NAME_LEN+1];
  int on_hand;
}part;

typedef struct employee {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee

int number;
```

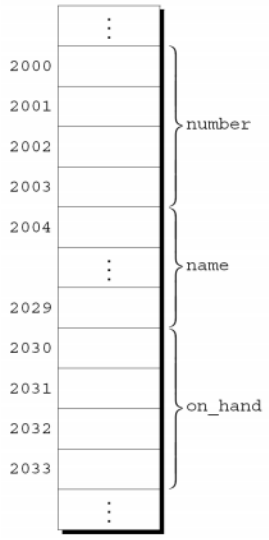


I valori dei membri verranno scritti successivamente nella memoria rappresentata dai riquadri. I membri vengono memorizzati nell'ordine in cui appaiono nella dichiarazione.

Memoria per part1:

Assunzioni:

- part1 inizia alla locazione 2000
- interi occupano 4 byte
- NAME_LEN vale 25
- Non ci sono gap (buchi) fra i vari membri



Per accedere ad un membro di una struttura scriviamo il nome della struttura prima, poi un punto e poi il nome del membro, ecco delle istruzioni che stampano i valori dei membri della struttura part1:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

I membri di una struttura sono lvalues, possono apparire nella parte sinistra di un assegnamento oppure come operando di un incremento o decremento.

```
part1.number = 258;
                /* changes part1's part number */
part1.on_hand++;
                /* increments part1's quantity on hand */
```

Il punto usato per specificare un membro della struttura è in effetti un operatore e ha precedenza su tutti gli altri operatori.

Esempio:

```
scanf("%d", &part1.on_hand);
```

L'operatore . ha precedenza sull'operatore &, quindi & calcola l'indirizzo di part1.on_hand.

L'altra operazione principale sulle strutture è l'assegnamento:

part2 = part1;

L'effetto di questa istruzione è la copia di part1.number in part2.number, di part1.name in part2.name e così via.

Le funzioni possono avere strutture come argomenti e come valori di ritorno, una funzione con una struttura come argomento:

```
void print_part(part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

La chiamata a print_part(part p) è print_part(part1);

Sia passare una struttura ad una funzione che restituire una struttura come valore di ritorno richiedono la copia di tutti i membri della struttura, per evitare troppe operazioni di copia è preferibile passare o restituire un puntatore alla struttura.

Esercizio: funzione che verifica se due triangoli sono uguali, a meno di traslazione, rotazione o ribaltamento.

Consideriamo uguali due triangoli che hanno le lunghezze dei lati uguali, definiamo il tipo punto, rappresentiamo i triangoli con vettori di tre punti. Prendere in input i due triangoli, calcolare le lunghezze dei lati e confrontarle.

```
typedef struct punto {
  double x,y;
} punto;

void inserT (punto []);
int verifT (punto [], punto[]);
double dist(punto, punto);
int confr(double [], double[]);
void ordina (double []);

int main (void)
{
  punto tr1[3], tr2[3];
  printf("coordinate primo triangolo:");
  inserT(tr1);
  printf("coordinate secondo:");
  inserT(tr2);
  if (verifT(tr1, tr2))
    printf("sono uguali");
  else
    printf("sono diversi");
  return 0;
}
```

```
void inserT(punto tr[3])
{
  int i;
  for (i=0; i<3; i++)
  {
    printf ("inserisci ascissa vertice %d: ", i);
    scanf ("%d", &tr[i].x);
    printf ("inserisci ordinata vertice %d: ", i);
    scanf ("%d", &tr[i].y);
  }
}
```

```
void ordina (double A[3])
{
……..
  }
}
```

```
double distanza (punto p1, punto p2)
{
   double d, xdiff, ydiff;
   xdiff = p1.x - p2.x;
   ydiff = p1.y - p2.y;
   d = sqrt(xdiff*xdiff + ydiff*ydiff);
   return d;
}
```

```
int verifT (punto tr1[3], punto tr2[3])
{
  double A[3], B[3];
  int i;
  A[0] = distanza (tr1[0], tr1[1]);
  A[1] = distanza (tr1[2], tr1[1]);
  A[2] = distanza (tr1[0], tr1[2]);
  B[0] = distanza (tr2[0], tr2[1]);
  B[1] = distanza (tr2[2], tr2[1]);
  B[2] = distanza (tr2[0], tr2[2]);
  ordina(A);
  ordina(B);
  for (i=0; i<3; i++)
    if (A[i] != B[i])
      return 0;
  return 1;
}
```

I puntatori a record possono essere usati per allocare record dinamicamente, passare record con call-by-reference e creare strutture a puntatori.

Creare Strutture a Puntatori

Una struct non può contenere una istanza di se stesso, può contenere però un membro che è un puntatore allo stesso tipo di struttura. La definizione della struttura non riserva spazio di memoria e crea un nuovo tipo di dato che è usato per dichiarare variabili di tipo struttura.

```
typedef struct corpoCeleste {
    char nome[16];
    float diametro;
    float giorno;
    float anno;
    corpoCeleste orbita;
} corpoCeleste;
```

Errore: questo tipo di autoreferenziazione non è gestibile

```
typedef struct corpoCeleste *rifer;

typedef struct corpoCeleste {
    char nome[16];
    float diametro;
    float giorno;
    float anno;
    rifer orbita;
} corpoCeleste;
```

uno dei campi della struttura è un riferimento ad un altra struttura dello stesso tipo

Lezione 12: Accesso File in C

Il File: Dichiarazione

In C è definito un tipo FILE

FILE *infile;

FILE *outfile;

Le variabili infile e outfile sono puntatori a file.

La dichiarazione di un file non è associata alla creazione del file, bisognerà utilizzare la funzione fopen:

file pointer variable = fopen(filename, mode);

- file name: stringa che specifica il nome del file;
- mode: stringa che specifica la modalità di trasferimento dati:
  - "r": lettura (il file deve già esistere);
  - "w": scrittura (viene creato un file di nome file name);
  - "a": appending;

Il File: Accesso

Possibili modalità di trasferimento:

- "rw": lettura e scrittura ("r+");
- "wr": scrittura e lettura ("w+");
- "ra": lettura e appending;
- "b": apertura di file binario;

Lettura e scrittura sullo stesso file (o viceversa):

- fflush(FILE *fp); - Svuota il buffer.
- fseek(FILE *fp, long offset, int place); - La posizione è offset byte da place.
- rewind(FILE *fp); - La posizione è all'inizio del File (void fseek(fp, 0L, Seek_Set)).

Vi sono inoltre altre funzioni che consentono di scrivere o leggere i file:

- getc(pointer to file) e putc (char, pointer to file): che agiscono sui singoli file.
- fscanf(pointer, …) e fprintf(pointer, …): che lavorano su tutti i tipi di variabile.

Il File: Chiusura

I file aperti in C devono essere chiusi al termine dell'esecuzione:

- fclose (infile);
- fclose(outfile);

Esempio: Scrivere un programma in C che presi in input due file, denominati fileuno e filedue contenenti caratteri minuscoli, copi il contenuto di filedue in coda a fileuno, e successivamente crei un nuovo file, il cui nome è inserito dall'utente, in cui sia copiato il contenuto di fileuno invertendo le minuscole con le maiuscole.

```
#include <stdio.h>


/*definizione prototipi*/


int controllo_input(int);
void controllo_file(FILE *, char *);
int append_file(FILE *, FILE *);
int converti_maiuscole(char *);
```

```
int main(int argc, char *argv[])
{
    FILE *uno, *due;

    controllo_input(argc);
    uno=fopen(argv[1],"a");
    controllo_file(uno,argv[1]);
    due=fopen(argv[2],"r");
    controllo_file(due,argv[2]);
    if(append_file(uno,due))
    printf("La scrittura è OK\n");
    else
    printf("La scrittura non è
    avvenuta\n");
    if(converti_maiuscole(argv[1])
    printf("La conversione è OK\n");
    return 0;
}
```

```
int controllo_input (int n)

{
    if(n!=3)
    {
    printf("Il programma apre due file denominati
    fileuno e filedue\n");
    printf("preceduti dal nome programma: il
    risultato sarà la concatenazione\n");
    printf("dei due file memorizzata in fileuno.
    Successivamente il contenuto\n");
    printf("di fileuno verrà copiato in un file
    nuovo, il cui nome verrà deciso\n");
    printf("dall'utente, convertendo le minuscole
    in maiuscole\n");
    exit(1);
    }
    else
    return 0;
}
```

```
#include <stdio.h>


/*definizione prototipi*/


int controllo_input(int);
void controllo_file(FILE *, char *);
int append_file(FILE *, FILE *);
int converti_maiuscole(char *);


int main(int argc, char *argv[])
{
        FILE *uno, *due;

        controllo_input(argc);
        uno=fopen(argv[1],"a");
        controllo_file(uno,argv[1]);
        due=fopen(argv[2],"r");
        controllo_file(due,argv[2]);
        if(append_file(uno,due))
        printf("La scrittura è OK\n");
        else
        printf("La scrittura non è avvenuta\n");
        if(converti_maiuscole(argv[1])
        printf("La conversione è OK\n");
        return 0;
}
```

```
void controllo_file(FILE *file, char
    *stringa)

{
if(file!=NULL)
printf("%s è stato aperto
    correttamente\n",
stringa);
else
exit(1);
}
```

```
#include <stdio.h>


/*definizione prototipi*/


int controllo_input(int);
void controllo_file(FILE *, char *);
int append_file(FILE *, FILE *);
int converti_maiuscole(char *);


int main(int argc, char *argv[])
{
        FILE *uno, *due;

        controllo_input(argc);
        uno=fopen(argv[1],"a");
        controllo_file(uno,argv[1]);
        due=fopen(argv[2],"r");
        controllo_file(due,argv[2]);
        if(append_file(uno,due))
        printf("La scrittura è OK\n");
        else
        printf("La scrittura non è avvenuta\n");
        if(converti_maiuscole(argv[1])
        printf("La conversione è OK\n");
        return 0;
}
```

```
int append_file(FILE *f1, FILE *f2)

    {
    int c;
    while((c=getc(f2))!=EOF)
    putc(c,f1);
    fclose(f1);
    fclose(f2);
    return(1);
    }
```

```
#include <stdio.h>


/*definizione prototipi*/


int controllo_input(int);
void controllo_file(FILE *, char *);
int append_file(FILE *, FILE *);
int converti_maiuscole(char *);


int main(int argc, char *argv[])
{
        FILE *uno, *due;

        controllo_input(argc);
        uno=fopen(argv[1],"a");
        controllo_file(uno,argv[1]);
        due=fopen(argv[2],"r");
        controllo_file(due,argv[2]);
        if(append_file(uno,due))
        printf("La scrittura è OK\n");
        else
        printf("La scrittura non è avvenuta\n");
        if(converti_maiuscole(argv[1])
        printf("La conversione è OK\n");
        return 0;
}
```

```
int converti_maiuscole (char *stringa)
{
char nome[10]; int c;
FILE *tre, *f1;
printf("Inserisci il nome file\n");
scanf("%s",nome);
tre=fopen(nome,"wb");
controllo_file(tre,nome);
f1=fopen(stringa,"r");
controllo_file(f1,stringa);
while((c=getc(f1))!=EOF)
if(c!=' ')
        putc(c-32,tre);
else
        putc(c,tre);
fclose(tre);
fclose(f1);
return (1);
}
```

```
#include <stdio.h>


/*definizione prototipi*/


int controllo_input(int);
void controllo_file(FILE *, char *);
int append_file(FILE *, FILE *);
int converti_maiuscole(char *);


int main(int argc, char *argv[])
{
        FILE *uno, *due;

        controllo_input(argc);
        uno=fopen(argv[1],"a");
        controllo_file(uno,argv[1]);
        due=fopen(argv[2],"r");
        controllo_file(due,argv[2]);
        if(append_file(uno,due))
        printf("La scrittura è OK\n");
        else
        printf("La scrittura non è avvenuta\n");
        if(converti_maiuscole(argv[1])
        printf("La conversione è OK\n");
        return 0;
}
```