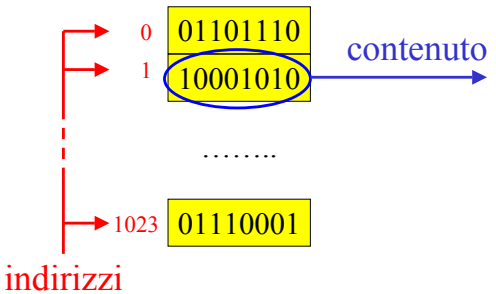
ROADMAP

Funzioni e struttura di un programma	Tipi, operatori, espressioni	Strutture di controllo	Input/Output	Strutture dati
	Livello 1			
	Livello 2			
	Livello 3			

TIPI, OPERATORI ED ESPRESSIONI

- Le variabili puntatore

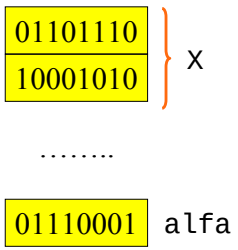
RICHIAMI SULL'ORGANIZZAZIONE DELLA MEMORIA



Ciascuna cella di memoria e` caratterizzata da:

- **un indirizzo fisico**: la sua posizione all'interno della memoria
- **il contenuto**: l'informazione

RICHIAMI SULL'ORGANIZZAZIONE DELLA MEMORIA



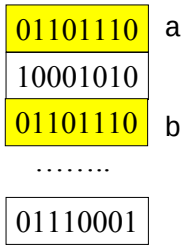
Il concetto di **variabile** rappresenta un'astrazione della nozione di celle di memoria (caratterizzata da un nome e da un valore)

Lo spazio di memoria necessario per rappresentare una variabile dipende dal **tipo** della variabile

MODALITA' DI ACCESSO ALLE VARIABILI

- Considerata l'organizzazione della memoria, esistono due modi per poter accedere ad una variabile:
  - attraverso il suo **nome** (identificatore simbolico)
  - attraverso il suo **indirizzo**

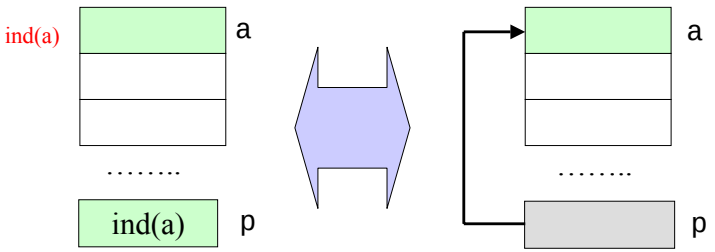
ACCESSO PER NOME



`b = a` significa:

"preleva il valore contenuto nella cella il cui nome e' `a` *e memorizzalo nella cella di nome* `b`*"*

ACCESSO PER INDIRIZZO



"la cella di nome `p` *contiene l'indirizzo della variabile* `a`*"*

`p` e` una variabile particolare (**puntatore**) che contiene l'indirizzo di un'altra variabile

OPERATORI PER LA GESTIONE DELLA MEMORIA

- Il "C" permette una gestione trasparente della memoria che rispecchia l'organizzazione **fisica** della memoria della macchina astratta

- L'operatore **&** fornisce l'indirizzo di un oggetto
  - l'istruzione `p=&c;`
    assegna l'indirizzo di `c` alla variabile `p`
  - si dice che `p` "punta a" `c`

OPERATORI PER LA GESTIONE DELLA MEMORIA

- L'operatore ***** e` l'operatore di **indirezione** o **deferenziazione**
  - l'istruzione `c=*p;`

    assegna il contenuto della cella puntata da `p` alla variabile `c`
  - l'espressione `*p` rappresenta la cella di memoria il cui indirizzo e` contenuto in `p`
  - `p` e` una **variabile puntatore**

DICHIARAZIONE DI PUNTATORI

*Tipo-Dato *Nome-Var-Puntatore;*

- la dichiarazione di una variabile puntatore avviene *implicitamente* attraverso la dichiarazione del tipo della variabile "puntata"
- va considerata come una forma abbreviata che unifica una dichiarazione di tipo e una dichiarazione di variabile

DICHIARAZIONE DI PUNTATORI

- Non e` significativo specificare il tipo della variabile puntatore
  - il suo valore e` infatti sempre un indirizzo, quindi memorizzato, per una data architettura, sempre con lo stesso numero di byte
- E' invece indispensabile specificare il tipo della variabile puntata
  - un puntatore punta alla prima delle celle di memoria associate alla variabile
  - e` necessario conoscere il tipo della variabile puntata per poter manipolare l'intero gruppo di celle consecutive che la rappresentano

DICHIARAZIONE DI TIPO PUNTATORE

- Una dichiarazione piu` esplicita di una variabile puntatore si ottiene con il costrutto **`typedef`**

  `typedef` *`TipoDato *TipoPunt;`*
  - definisce il tipo denominato *`TipoPunt`* come un puntatore a una cella contenente un valore *`TipoDato`*
  - la dichiarazione di una variabile di *`TipoPunt`* si puo` scrivere:

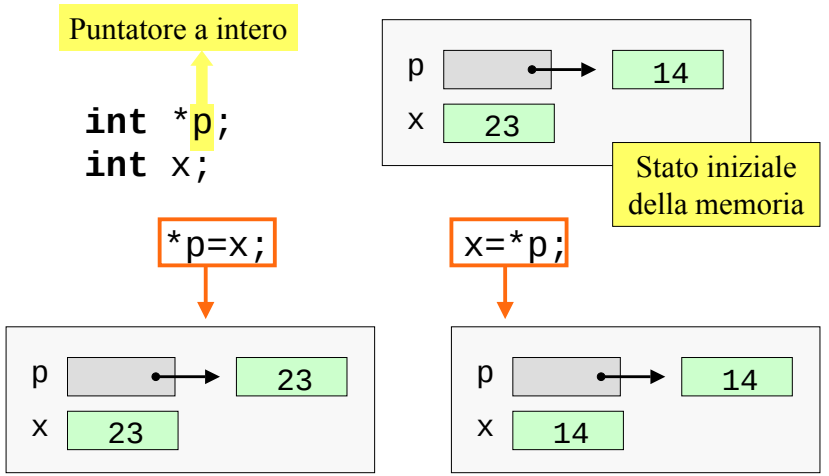
```
TipoPunt p;
```

che equivale a

```
TipoDato *p;
```

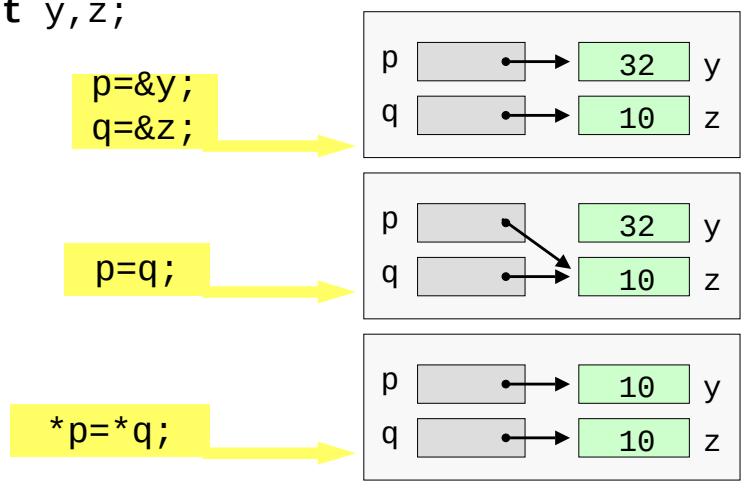
Esempio: **`typedef int`** `*Punt_A_int;`

ESEMPIO 15



ESEMPIO 16

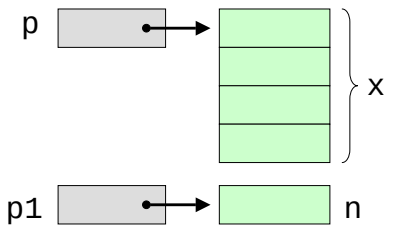
```
int *p,*q;
int y,z;
```

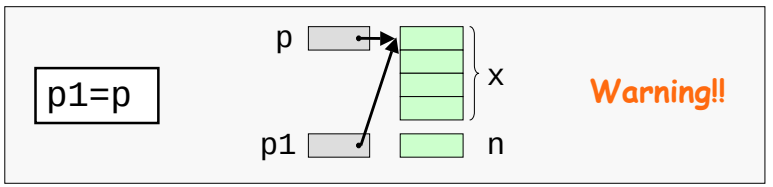


ERRORI

Supponiamo:

```
float *p, x;
int *p1, n;
```

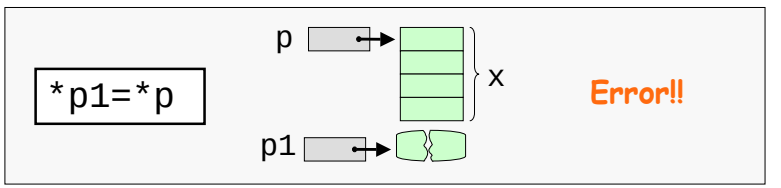




Un successivo assegnamento, ad es. `n=*p1`, provocherebbe infatti un risultato non voluto

ERRORI

Nelle stesse ipotesi,



Non avviene infatti la conversione di tipo!!

Il compilatore rileva il tentativo di assegnare ad una locazione di memoria per un **`int`** un valore **`float`**

OPERAZIONI APPLICABILI A VARIABILI PUNTATORE

- Assegnamento dell'indirizzo di una variabile tramite l'operatore unario **&** seguito dal nome della variabile
- assegnamento del valore di un altro puntatore
- assegnamento di indirizzi di memoria a seguito di operazioni di allocazione esplicita di memoria
- operazione di deferenziazione, indicata dall'operatore *****
- il confronto (==, ! =,>,<,<=,>=)
- operazioni aritmetiche

NOTE

- Il valore speciale **`NULL`**
  - Se una variabile puntatore ha valore **`NULL`**, `*p` e` indefinito

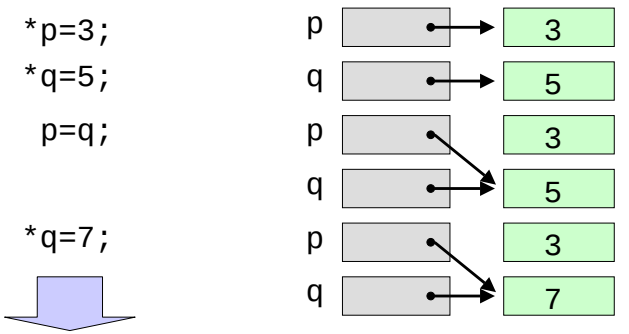
    `p=NULL` significa che `p` non punta ad alcuna informazione significativa

- Il tipo speciale **`void`**
  - Un puntatore a **`void`** viene usato per supportare qualsiasi tipo di puntatore, ma non puo` essere referenziato

NOTE

- Il "C" risulta particolarmente caratterizzato dall'uso ampio e flessibile dei puntatori
  - talvolta costituiscono l'unico modo possibile di esprimere un calcolo
  - generalmente consentono di ottenere un codice particolarmente compatto ed efficiente
- **Attenzione al loro utilizzo!!!**
  - In generale, si raccomanda di limitare l'uso dei puntatori alle loro applicazioni essenziali

RISCHI DELLA PROGRAMMAZIONE MEDIANTE PUNTATORI



(!!) Subdolamente anche `*p=7` (!!)

ESEMPIO 17

- Nel seguente frammento di programma si presuppone che sia disponibile in memoria un array contenente 100 numeri interi
- Il codice assegna un valore a due variabili che puntano all'elemento dell'array contenente, rispettivamente, il valore piu` basso e il valore piu` alto

ESEMPIO 17 (cont.)

```
#define LunghezzaArray 100
main()                                          {
int i, ArrayDiInt[LunghezzaArray];
int *PuntaAMinore, *PuntaAMaggiore;
. . .
PuntaAMinore = &ArrayDiInt[0];
PuntaAMaggiore = &ArrayDiInt[0];
i=1;
while (i<LunghezzaArray)                        {
  if (ArrayDiInt[i] < *PuntaAMinore)
      PuntaAMinore = &ArrayDiInt[i];
  if (ArrayDiInt[i] > *PuntaAMaggiore)
      PuntaAMaggiore = &ArrayDiInt[i];
  i++;                                          }
                                                    }
```


PUNTATORI E STRINGHE

- Le **stringhe** possono essere dichiarate mediante:
  - vettori di caratteri

```
char Nome[50];
char message[]="Warning message!!";
```

Dimensione fissata a priori o al momento dell'inizializzazione

  - puntatori a vettori di caratteri

```
char *pmessage;
```

Dimensione indefinita sino al momento dell'assegnazione o dell'inizializzazione

PUNTATORI E STRINGHE

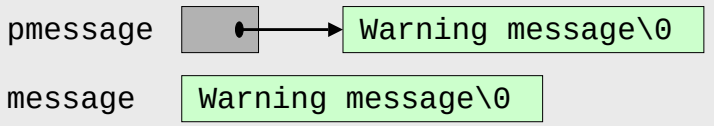
- Esempio:

```
pmessage="Warning message!!";
```

  - assegna a `pmessage` un puntatore al vettore di caratteri
  - *non* viene fatta alcuna copia della stringa

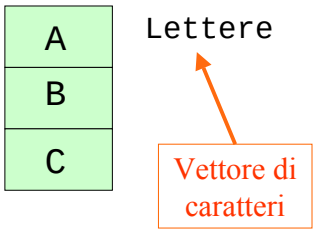
NOTE

- L'accesso ad una stringa di caratteri avviene tramite un puntatore al suo primo elemento
- Il C non fornisce alcun operatore che consenta di trattare una stringa come un'entità unica
- Nella rappresentazione interna, il vettore che rappresenta la stringa e` terminato dal carattere nullo **'\0'**, in modo che il programma possa trovarne la fine



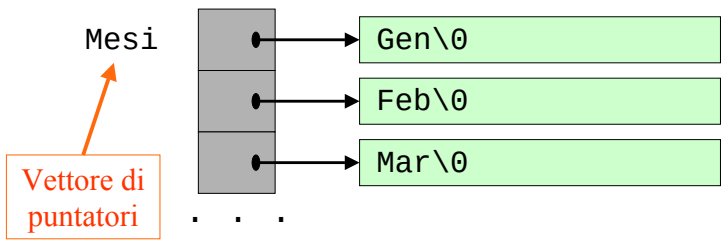
PUNTATORI E STRINGHE

```
char  Lettere[]={'A', 'B', 'C'};
char  Lettere[]={"ABC"};
char  Lettere[3];
Lettere[0]='A';
Lettere[1]='B';
Lettere[2]='C';
```



PUNTATORI E STRINGHE

```
char *Mesi[12];
  Mesi[0]="Gen";
  Mesi[1]="Feb";
  Mesi[2]="Mar";
     . . . . .
char *Mesi[]={"Gen", "Feb", "Mar",. . .};
```



FUNZIONI E STRUTTURA DI UN PROGRAMMA

- Passaggio di parametri "per indirizzo"

RICHIAMI SUI SOTTOPROGRAMMI

- La comunicazione tra programma chiamante e sottoprogramma (funzione) puo` avvenire attraverso un passaggio di parametri
- Al momento della chiamata di funzione vengono "passati" a quest'ultima i **parametri attuali**
  - ovvero i valori degli argomenti rispetto ai quali la funzione deve essere calcolata

Programma chiamante

```
main()
{
funz(parametri attuali);
}
```

Funzione

```
void funz(parametri
          formali);

{ . . . }
```


PASSAGGIO DI PARAMETRI

- Poiche' il "C" passa alle funzioni i parametri per valore, la funzione chiamata **non ha un modo diretto per alterare una variabile nella funzione chiamante**

In generale, vengono modificati i parametri formali e non i corrispondenti parametri attuali

ESEMPIO 18

- Supponiamo di voler costruire una procedura `AppendiElemento (carattere, stringa)` che ci consenta di appendere un carattere ad una stringa di caratteri specificata (*).

(*) L'esempio e` una versione modificata della procedura presentata nell'Esempio 11

ESEMPIO 18 (cont.) - errato -

```
#include <stdio.h>
typedef struct { char Testo[30];
                 int  PosCorrente;
               } My_String;
main()                                  {
  My_String String1, String2;
  char Carattere1, Carattere2;
                          . . .
  scanf ("%c%c", &Carattere1, &Carattere2);
  AppendiElemento(Carattere1, String1);
  AppendiElemento(Carattere2, String2);  }
void AppendiElemento(char C, My_String S)     {
  S.PosCorrente++;
  S.Testo[S.PosCorrente] = C;          }
```

NOTE

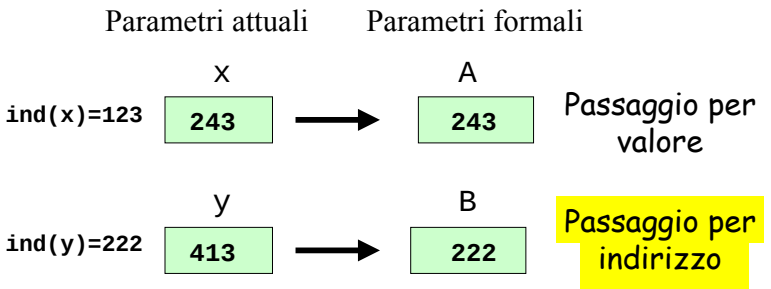
- L'esecuzione della procedura provoca l'inserimento dei caratteri letti in `S`, ma non in `String1` e `String2` che sono invece rimasti esattamente come prima!!
- La dichiarazione di `String1` e `String2` come variabili globali assicurerebbe la possibilita` di modificarne lo stato, ma occorrerebbero due diverse procedure (una per `String1` e una per `String2`)
  - non sarebbe cioe` possibile la "*parametrizzazione*" della procedura

PASSAGGIO PARAMETRI PER INDIRIZZO

- Per ottenere l'effetto desiderato bisogna ricorrere al **passaggio dei parametri per indirizzo:**
  - invece di copiare il **valore** del parametro attuale nella corrispondente cella del parametro formale
  - si copia l'**indirizzo** della cella contenente il parametro attuale in un'opportuna cella del parametro formale

In questo modo si passa alla funzione il riferimento alla variabile da modificare

PASSAGGIO PARAMETRI PER VALORE E PER INDIRIZZO



In corrispondenza del parametro formale, la macchina accede alla cella del parametro attuale tramite l'indirizzo che le e` stato precedentemente passato

PASSAGGIO PARAMETRI PER INDIRIZZO

- E' possibile realizzare l'effetto di un passaggio per indirizzo
  - utilizzando un tipo puntatore per la definizione dei parametri formali della funzione
  - usando l'operatore di deferenziazione di puntatore all'interno del corpo della funzione
  - passando al momento della chiamata della funzione come parametro attuale un indirizzo di variabile (usando eventualmente l'operatore &)

ESEMPIO 18 (cont.) - corretto -

```
#include <stdio.h>
typedef struct { char Testo[30];
                 int  PosCorrente;
               } My_String;
main()                              {
  My_String String1, String2;
  char Carattere1, Carattere2;
                        . . .
  scanf ("%c%c", &Carattere1, &Carattere2);
  AppendiElemento(Carattere1, &String1);
  AppendiElemento(Carattere2, &String2);  }
void AppendiElemento(char C, My_String *S)     {
  (*S).PosCorrente++;
  (*S).Testo[(*S).PosCorrente] = C;          }
```


NOTE

- La funzione `scanf` utilizza un passaggio di parametri "per indirizzo"
  - `scanf` ha come parametro formale una variabile di tipo puntatore
  - la chiamata di `scanf` passa alla funzione l'indirizzo della variabile che si desidera leggere
  - il codice della `scanf` modifica la variabile propria dell'ambiente chiamante e in essa viene inserito il valore letto dal dispositivo di ingresso

In sostanza alla funzione `scanf` viene comunicato **dove** deve essere memorizzata la variabile che ci si appresta a leggere

ISTRUZIONI DI INPUT / OUTPUT

- gestione dei file

I FILE

- sono contenitori di informazione **permanenti** che spesso trascendono la vita di un programma
  - esistono prima della sua esecuzione
  - continuano ad esistere anche dopo la sua fine
- sono **parte del sistema** e sono gestiti dal S.O.
- programmi scritti in linguaggi diversi devono poter manipolare l'informazione contenuta nei file, chiedendo **l'intervento del sistema operativo** per creare/scrivere/leggere o cancellare *file*

GESTIONE DEI FILE

- Il linguaggio 'C' mette a disposizione opportune funzioni di libreria della *Standard Library* che
  - tengono conto del S.O. su cui il programma viene eseguito
  - consentono un'invocazione corretta delle funzioni del S.O. stesso

FILE E DISPOSITIVI

- Il S.O. mostra al programmatore le periferiche di I/O come *file*
  - il programma puo` cosi` realizzare un'operazione di scrittura/lettura sul dispositivo di Standard output/input scrivendo/leggendo un *particolare file* che lo rappresenta

☞ Memorizzazione permanente nella memoria di massa e operazioni di I/O avvengono con la stessa modalita`: scrivere o leggere un file tramite una delle funzioni messe a disposizione dalla `stdlib`

FLUSSI, FILE E PROGRAMMI "C"

- L'uso di un file da parte di un programma richiede l'apertura di un **flusso di comunicazione**
- Per aprire un flusso di comunicazione occorre:
  - dichiarare una variabile di tipo puntatore
  - chiedere l'apertura del flusso mediante una funzione di libreria (**`fopen`**)

`fopen` richiede al S.O. la creazione di un nuovo file o l'apertura di un file esistente

FLUSSI, FILE E PROGRAMMI "C"

- L'apertura del flusso di comunicazione provoca l'assegnamento della variabile puntatore

Il valore assunto dalla variabile puntatore serve al programma per far riferimento al file corrispondente al particolare flusso di comunicazione aperto

- Una funzione **`fclose`** permette la chiusura del flusso di comunicazione

`fclose` impedisce ulteriori riferimenti al file precedentemente utilizzato

FLUSSI DI COMUNICAZIONE

- Possono essere aperti 2 tipi di flussi:
  - **binario** → una sequenza di byte
  - **di tipo testo** → una sequenza di caratteri (generalmente suddivisa in linee, terminate da un carattere di *newline*)

OSS_1: Il file corrispondente al flusso di comunicazione viene aperto e trattato nelle operazioni successive in modo dipendente dalla modalita' di apertura

OSS_2: In un flusso di testo alcuni caratteri possono essere tradotti da parte dell'ambiente;
In un flusso binario nessun byte viene tradotto

FILE E PROGRAMMI "C"

- L'accesso ad un file e` possibile in un programma attraverso una variabile puntatore
- Ogni variabile che punta a un file deve essere definita come segue:

```
FILE *nome-variabile;
```

- La variabile cosi` definita "punta" ad un oggetto di tipo **`FILE`** capace di registrare tutte le informazioni necessarie a controllare un flusso

FILE E PROGRAMMI "C"

- **`FILE`** e` un tipo strutturato che puo` contenere:
  - un campo con la specifica della **modalita` di utilizzo** (lettura, scrittura o lettura e scrittura)
  - un campo con la **posizione corrente** sul file (che punta al prossimo byte da leggere o scrivere sul file)
  - un campo contenente un **indicatore di errore** che registra il presentarsi di un errore di lettura/scrittura
  - un campo contenente un **indicatore di end-of-file** (`EOF`) che registra se e` stata raggiunta la fine del file

FILE E PROGRAMMI "C"

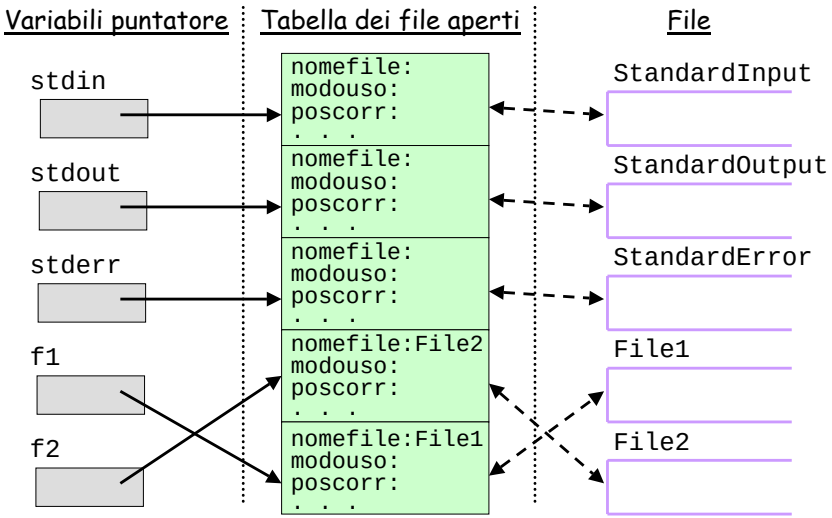
- E` il sistema operativo a manipolare i campi della struttura di tipo **`FILE`**
  - l'utente ne provoca la **manipolazione indiretta** invocando funzioni della *Standard Library*
- Il sistema operativo gestisce una tabella che tiene traccia del numero di file aperti e del loro stato
  - le funzioni di libreria interagiscono con tale tabella

FLUSSI DI COMUNICAZIONE STD

- Vengono automaticamente aperti quando inizia l'esecuzione di un programma
  - `stdin` (associato al file che rappresenta la tastiera)
  - `stdout`, `stderr` (associati al file che rappresenta il video)
- `stdout`, `stderr`, `stdin` sono le variabili puntatore ai descrittori di tali file
  - il loro valore viene assegnato quando inizia l'esecuzione di un programma

`printf` e `scanf` utilizzano questi flussi standard

FILE, S.O. E PROGRAMMI "C"



OPERAZIONI SU FILE

- Attraverso le principali funzioni fornite dalla *Standard Library* (*):
  - operazioni di gestione dei file
  - operazioni di gestione degli errori
  - operazioni di lettura e scrittura

(*) Per una trattazione completa delle funzioni disponibile nella Standard Library per gestire le operazioni sui file si rimanda ad un manuale di Linguaggio "C"

OPERAZIONI DI GESTIONE DEI FILE

```
FILE *fopen(nomefile,modalita`);
```

- apre un file (eventualmente creandolo) e vi associa un flusso

Modalita`:

- `"r"` → lettura, posizionamento all'inizio del file
- `"w"` → scrittura, posizionamento all'inizio del file
- `"a"` → scrittura, posizionamento alla fine del file

- Restituisce `NULL` se l'operazione fallisce

OPERAZIONI DI GESTIONE DEI FILE

```
int fclose(FILE *fp);
```

– chiude il file cui fa riferimento il puntatore `fp`

☞ assegnamento del valore `NULL` a `fp`

☞ il rilascio del descrittore di tipo **`FILE`**

- Se l'operazione di chiusura viene eseguita correttamente restituisce valore uguale a `0`, altrimenti restituisce il valore `EOF`

(`EOF` e` una costante definita in `stdio.h`)

OPERAZIONI DI GESTIONE ERRORI

- Nella struttura che descrive lo stato di un file aperto esistono due campi destinati a registrare
  – una generica situazione di errore
  – il verificarsi della circostanza di *"raggiungimento della posizione di fine file"*
- Il programmatore puo` testare il contenuto di tali campi invocando le due funzioni **`feof`** o **`ferror`**, messe a disposizione dalla *Standard Library*

OPERAZIONI DI GESTIONE ERRORI

```
int ferror(FILE *fp);
```

– controlla se e` stato commesso un errore nella precedente operazione di lettura o scrittura
  - restituisce `0` in caso di **assenza di errore**
  - restituisce un valore diverso da `0` in caso di **errore**

```
int feof(FILE *fp);
```

– controlla se e` stata raggiunta la fine del file nella precedente operazione di lettura o scrittura
  - restituisce `0` se non e` stata raggiunta la fine del file
  - restituisce un valore diverso da `0` in caso contrario

OPERAZIONI LETTURA E SCRITTURA

- Possono essere effettuate in 4 modi diversi:
  – precisando il **formato** dei dati in ingresso e in uscita
  – accedendo ai dati **carattere per carattere**
  – accedendo ai dati **linea per linea**
  – accedendo ai dati **blocco per blocco**

LETTURA /SCRITTURA FORMATTATA

- Sui dispositivi di *StdInput* e *StdOutput*:

```
int printf(stringa_controllo, elementi);
int scanf(stringa_controllo, indirizzo_elementi);
```

- Su file precisati dall'utente tramite il puntatore `fp`

```
int fprintf(FILE *fp, string_contr, elementi);
int fscanf(FILE *fp, string_contr, indirizzo_elem);
```


LETTURA /SCRITTURA DI CARATTERI

```
int getchar(void);
```

Legge da *stdin* il prossimo carattere restituendolo come intero

```
int putchar(int c);
```

Scrive come prossimo carattere sul file di *stdout* il carattere che riceve come parametro restituendo il carattere scritto

```
int getc(FILE *fp);     int fgetc(FILE *fp);
```

Leggono il prossimo carattere del file specificato tra i parametri di ingresso restituendolo come intero

LETTURA /SCRITTURA DI CARATTERI

```
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
```

Scrivono come prossimo carattere del file il carattere specificato tra i parametri di ingresso restituendolo come intero

- Tutte le funzioni restituiscono `EOF` in caso di errore.
- Per verificare se si tratta di un caso di fine file o di altro errore bisogna utilizzare `feof` o `ferror`

ESEMPIO 19

```
#include <stdio.h>
#include <stddef.h>
main()
{
  FILE *fp;
  char c;
  if ((fp = fopen("filechar","r")) != NULL)
    {
      while ((c=fgetc(fp)) != EOF)
        putchar(c);
      fclose(fp);
    }
  else
     printf("Il file non puo` essere aperto\n");
}
```

Contiene la definizione di EOF, del tipo FILE e le testate delle funzioni che operano su file

Contiene la definizione NULL

Il file viene aperto in lettura con modalità testo

Viene letto e stampato a video un carattere per volta sino a fine file

LETTURA/SCRITTURA DI STRINGHE

```
char *gets(char *s);
```

- Legge caratteri dal file *stdin* fino a quando non raggiunge un carattere di *newline*
- `s` punta al primo elemento del vettore in cui vengono registrati i dati letti
- Non inserisce nel vettore il carattere di *newline* eventualmente incontrato bensi` il terminatore di stringa `\0`
- Restituisce il primo parametro se l'operazione e` stata correttamente eseguita, `NULL` in caso contrario

LETTURA/SCRITTURA DI STRINGHE

```
int puts(char *s);
```

- Scrive su *stdout* il contenuto della stringa puntata da `s`, seguita da un carattere di *newline*
- il carattere `\0` terminatore di stringa non viene scritto
- Restituisce `0` se l'operazione e` stata eseguita correttamente, un valore diverso da `0` in caso contrario

LETTURA/SCRITTURA DI STRINGHE

```
char *fgets(char *s, int n, FILE *fp);
```

- Legge caratteri dal file puntato da `fp` fino a quando non ha letto n-1 elementi, non raggiunge un carattere di newline o non raggiunge la fine del file
- `s` punta al primo elemento del vettore in cui vengono messi i dati letti
- Inserisce nel vettore il carattere di newline eventualmente incontrato e inserisce nel vettore il terminatore di stringa `\0`
- Restituisce il primo parametro se l'operazione e` stata correttamente eseguita, `NULL` in caso contrario

LETTURA/SCRITTURA DI STRINGHE

```
int fputs(char *s, FILE *fp);
```

- Scrive sul file cui fa riferimento `fp` il contenuto della stringa puntata da `s`
- il carattere `\0` terminatore di stringa non viene scritto
- Restituisce `0` se l'operazione e` stata eseguita correttamente, un valore diverso da `0` in caso contrario

LETTURA / SCRITTURA PER BLOCCHI

- E` possibile accedere ai dati di un file leggendo o scrivendo un **intero blocco** di dati binari
- Attraverso due funzioni di libreria

  `fread` e **`fwrite`**

  che consentono di precisare:
  - *l'indirizzo di un vettore* destinato a ricevere gli elementi che compongono il blocco (`fread`) o che contiene gli elementi che si desidera scrivere (`fwrite`)
  - il *numero di elementi* da leggere o scrivere
  - le *dimensioni* del singolo elemento

LETTURA / SCRITTURA PER BLOCCHI

```
int fread(void *ptr, dim_el, num_el, FILE *fp);
```

- Legge un blocco di dati binari dal file cui fa riferimento `fp` e li memorizza nel vettore puntato da `ptr`
- Termina correttamente se legge il numero di byte richiesti (`dim_el*num_el`); termina anche se incontra la fine del file o se si verifica un errore di lettura
- Restituisce il numero di elementi effettivamente letti: se tale numero e` inferiore rispetto al numero richiesto e` necessario usare `feof` o `ferror` per capire i motivi del (mal)funzionamento ottenuto

LETTURA / SCRITTURA PER BLOCCHI

```
int fwrite(void *ptr, dim_el, num_el, FILE *fp);
```

- Scrive un blocco di dati binari sul file cui fa riferimento `fp` prelevandoli dal vettore puntato da `ptr`
- Termina correttamente se scrive il numero di byte richiesti (`dim_el*num_el`); termina anche se incontra la fine del file o se si verifica un errore di scrittura
- Restituisce il numero di elementi effettivamente scritti: se tale numero e` inferiore rispetto al numero richiesto e` necessario usare `feof` o `ferror` per capire i motivi del (mal)funzionamento ottenuto

STRUTTURE DATI

- relazioni tra vettori e puntatori
- allocazione dinamica di vettori

VETTORI E PUNTATORI

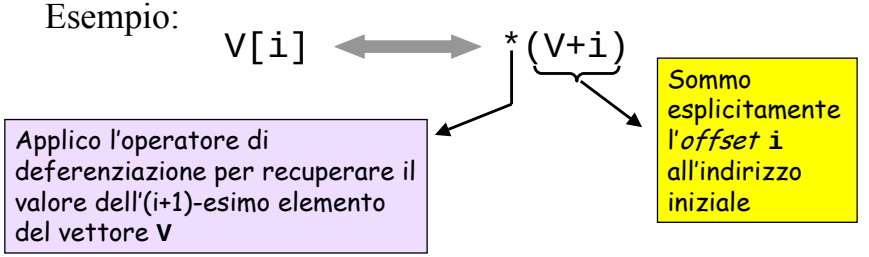
- Il nome di una variabile che identifica un vettore viene considerata dal "C" come l'indirizzo del primo elemento della variabile vettore
  - Si puo` dunque affermare che essa "punta" ad una parola di memoria esattamente come fa una variabile puntatore
- Esempio: `TipoDato V[5];`

  e` un **puntatore fisso** al primo elemento dell'array di `TipoDato`
  - non possiamo assegnare a `V` l'indirizzo di un'altra cella di memoria

VETTORI E PUNTATORI

- E` possibile accedere agli elementi del vettore attraverso un **indirizzamento esplicito della memoria**
  - usando l'operatore di deferenziazione
  - sfruttando la possibilita` di eseguire operazioni di somma e sottrazione tra interi e puntatori

Esempio:



PUNTATORI E VETTORI

- Analogamente, e` possibile **indicizzare** un puntatore a una variabile
- Esempio: `TipoDato *P;`

  Supposta `i` come variabile `int`

`*(P+i)` ⟷ `P[i]`

- `P+i` indica l'indirizzo dell'i-esimo oggetto che segue quello attualmente puntato da `P`.
  - indipendentemente dal tipo di oggetto a cui punta `P`
  - **l'*offset* che corrisponde a `i` viene "dimensionato" in base alla dimensione degli oggetti puntati da `P`**

ALLOCAZIONE STATICA DI MEMORIA

- Le dimensioni fisiche di ogni dato sono note prima dell'esecuzione del programma e non possono essere modificate in fase di esecuzione
  - l'allocazione avviene al tempo della compilazione
- Svantaggi:
  - spesso costituisce uno spreco considerevole di memoria fisica (Es. si dimensiona un vettore in base alla massima dimensione prevista)
  - non esclude del tutto il rischio di *overflow* (dovuto ad una sottostima della memoria necessaria)

ALLOCAZIONE DINAMICA DI MEMORIA

- Il "C" permette l'allocazione di memoria durante l'esecuzione di un programma:
  - nuove variabili possono venire "create" dal programmatore
- L'allocazione e il rilascio dinamico della memoria e` possibile grazie ad alcune funzioni disponibili nella *Standard Library*.

  Le principali sono:

  `malloc()` e **`free()`**

ALLOCAZIONE DINAMICA DI MEMORIA

- L'uso delle funzioni `malloc` e `free` richiede l'inclusione del file header **`<stdlib.h>`** tramite l'istruzione:

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

- I puntatori forniscono il **supporto** necessario per accedere e gestire le celle di memoria create "dinamicamente"
  - le variabili create dinamicamente sono anonime e **possono solo essere puntate**

LA FUNZIONE malloc( )

- La chiamata di funzione

```
malloc(sizeof(TipoDato));
```

  - crea in memoria una variabile di tipo *`TipoDato`*
  - restituisce come risultato l'indirizzo della variabile creata (piu` precisamente, l'indirizzo del primo byte riservato alla variabile)
- Esempio:

```
TipoDato *P;
P=malloc(sizeof(TipoDato));
```

☞ **P "punta" alla nuova variabile**

LA FUNZIONE malloc( )

- Il prototipo della funzione `malloc` e`:

```
void *malloc(int Dim);
```

  - restituisce un puntatore allo spazio per un oggetto di ampiezza pari a `Dim`, oppure `NULL` se la richiesta di allocazione non puo` essere soddisfatta. **Lo spazio non e` inizializzato!!**
- Esempi:

```
V=malloc(256);
X=malloc(sizeof(double));
m=malloc(DIM*sizeof(unsigned char));
y=(float *) malloc(sizeof(float));
```

LA FUNZIONE free( )

- La chiamata di funzione
  `free(`*`Puntatore_A_TipoDato`*`);`
  – rilascia lo spazio di memoria "puntato" dalla variabile *`Puntatore_A_TipoDato`*
- Esempio: `free(P);`

☞ **la corrispondente memoria fisica puntata da P risulta nuovamente disponibile**

- La funzione `free` deve ricevere, come parametro d'ingresso, un puntatore al quale era stato assegnato l'indirizzo restituito da una funzione `malloc`

ESEMPIO 20

```
. . .
#include <stdlib.h>
main()                                          {
  float *V;
  int Dim, n;
  . . .
  Dim=espressione1;
  V=(float *) malloc(Dim*sizeof(float));
  for(n=0;n<Dim;n++)
    {
      V[n] = espressione2;
            . . .
    }
  free(V);
  . . .                                         }
```

Inclusione dell'header file

Dichiarazione di una variabile puntatore

Allocazione della memoria

Indicizzazione del puntatore

Rilascio della memoria

CONSIDERAZIONI SULLA GESTIONE DELLA MEMORIA

Allocazione statica

- meno flessibile
- piu` efficiente

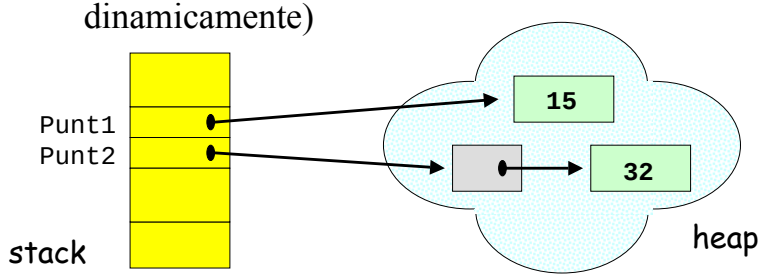
Allocazione dinamica

- piu` flessibile
- meno efficiente

La minor efficienza nel caso di allocazione dinamica dipende dal fatto che l'allocazione e il rilascio sono **controllate direttamente dal programmatore**

CONSIDERAZIONI SULLA GESTIONE DELLA MEMORIA

- La memoria e` partizionata in 2 porzioni distinte:
  – **Stack** (contiene solo le celle corrispondenti alle variabili dichiarate - allocate staticamente -)
  – **Heap** (contiene solo le variabili create dinamicamente)



ESERCIZIO

Si scriva un programma che

- *legga una matrice di numeri (A) contenuta in un file*
- *effettui una trasformazione sui valori della matrice (A) ottenendo una matrice (B)*
- *salvi la matrice B in un nuovo file*

La dimensione della matrice deve poter essere specificata dall'utente

RISOLUZIONE

- Passi logici:
  - leggo dimensioni della matrice
  - leggo nome file Input
  - leggo nome file Output
  - alloco lo spazio in memoria per le due matrici
  - predispongo la lettura del file Input
  - leggo da file i valori della matrice A
  - trasformo la matrice A nella matrice B
  - predispongo la scrittura del file Output
  - salvo su file i valori della matrice risultante B
  - chiudo i file aperti

RISOLUZIONE (cont.)

- Variabili necessarie:
  - nome del file Input (`In`)
  - nome del file Output (`Out`)
  - dimensioni della matrice (`DimX`, `DimY`)
  - le due matrici (`A`, `B`)
  - variabili per la referenziazione dei file (`fpin`, `fpout`)
  - variabili "di conteggio" per poter accedere agli elementi delle matrici (`i`)
  - eventuali variabili temporanee necessarie al processo di trasformazione A → B

RISOLUZIONE (cont.)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
main()
{
  char In[30], Out[30];
  FILE *fpin, *fpout;
  int i,DimX, DimY;
  float **A, **B;
  void Converto(float **a, float **b);

  printf("Inserire dimensioni [dimX] [dimY]");
  scanf("%d %d", &DimX, &DimY);
  printf("\nNome file Input --> ");
  scanf("%s", In); . . .
```

RISOLUZIONE (cont.)

```
. . .
  printf("\nNome file Output --> ");
   scanf("%s", Out);
  A=(float **)malloc(DimY*sizeof(float *));
  for(i=0;i<DimY;i++)
     A[i]=(float *)malloc(DimX*sizeof(float));
  B=(float **)malloc(DimY*sizeof(float *));
  for(i=0;i<DimY;i++)
     B[i]=(float *)malloc(DimX*sizeof(float));
  fpin=fopen(In,"r");
  if (fpin==NULL) { printf("Inpossibile aprire
  file %s\n",In);  exit(0);}
. . .
```


RISOLUZIONE (cont.)

```
  . . .
  for(i=0;i<DimY;i++)
    fread(A[i],sizeof(float),DimX,fpin);
  fclose(fpin);

  Converto(A,B,DimX,DimY);

  fpout=fopen(Out,"w");
  for(i=0;i<DimY;i++)
  fwrite(B[i],sizeof(float),DimX,fpout);
  fclose(fpout);
}
. . .
```


RISOLUZIONE (cont.)

```
. . .
void Converto(float **a, float **b, int dimX,
              int dimY)
{
 int i,j;

 for(i=0;i<dimX;i++)
 for(j=0;j<dimY;j++)
  {
   b[i][j]=sqrt(a[i][j]);
   . . .
  }
}
```