ESERCIZIO: Lettura/Scrittura Array

Non è possibile leggere/scrivere un intero vettore con un'unica operazione (a parte il caso particolare delle ***stringhe***); occorre leggere/scrivere ognuno dei suoi componenti

```
int main() {
   int i,frequenza[25];
   for (i=0; i<25; i++)
   {   scanf("%d",&frequenza[i]);
       frequenza[i]=frequenza[i]+1;
   }   /*    legge a terminale le componenti del
             vettore frequenza e le incrementa
       */
}
```

ESERCIZIO: Assegnamento

Anche se due variabili vettore sono dello ***stesso tipo***, NON è possibile l'assegnamento diretto:

```
int F[25], frequenza[25];
F = frequenza;          /* NO */
```

ma ***occorre copiare componente per componente***:

```
for (i=0; i<25; i++)
     F[i] = frequenza[i];
```

ESERCIZIO: MAX e min di un vettore

```
#define N 15   /* è noto a tutti che la dimensione
                      del vettore è N */

int minimo (int vet[]);
int massimo (int vet[]);

int main ()
{int i, a[N];
 printf ("Scrivi %d numeri interi\n", N);
 for (i=0; i<N; i++)
         scanf ("%d", &a[i]);
 printf ("L'insieme dei numeri è: ");
 for (i=0; i<N; i++)
       printf(" %d",a[i]);
 printf ("Il minimo vale %d e il
          massimo è %d\n", minimo(a), massimo(a));
}
```

ESERCIZIO: MAX e min di un vettore

```
int minimo (int vet[])
{int i, min;
 min = vet[0];
 for  (i = 1; i<N; i++)
   if (vet[i]<min) min = vet[i];
 return min;
}


int massimo (int vet[])
{int i, max;
 max = vet[0];
 for (i = 1; i<N; i++)
    if (vet[i]>max) max = vet[i];
 return max;
}
```

Ricerca in array

- Se l'array non è ordinato → ricerca lineare
- Se l'array è ordinato → ricerca binaria o dicotomica

ESERCIZIO: Ricerca di un elemento

```
#include <stdio.h>
#define N 15

int ricerca (int vet[], int el);
int main ()
{int i;
 int a[N];
 printf ("Scrivi %d numeri interi\n", N);
 for (i = 0; i<N; i++)
     scanf ("%d", &a[i]);
 printf ("Valore da cercare: ");
 scanf ("%d",&i);
 if (ricerca(a,i)) printf("\nTrovato\n");
     else printf("\nNon trovato\n");
}
```

ESERCIZIO: Ricerca di un elemento

```
int ricerca (int vet[], int el)
{int  i=0;
 int  T=0;
 while ((i<N) && (T==0))
  { if (el==vet[i]) T=1;
    i++;}
 return T;
}
```

Proposta di esercizio ulteriore: ricercare ***se e quali*** elementi di un vettore ***V1*** di float sono contenuti in un altro vettore ***V2*** di float. Le dimensioni dei due vettori possono essere diverse

ESERCIZIO: Ricerca di un elemento

Sapendo che il vettore è ***ordinato*** (esiste una relazione d'ordine totale sul dominio degli elementi), la ricerca può essere ottimizzata

- ***Vettore ordinato in senso non decrescente:***

2	3	5	5	7	8	10	11

`se i<j` si ha `V[i]<=V[j]`

- ***Vettore ordinato in senso crescente:***

2	3	5	6	7	8	10	11

`se i<j` si ha `V[i]<V[j]`

In modo analogo si definiscono l'ordinamento in senso ***non crescente*** e ***decrescente***

ESERCIZIO: RICERCA BINARIA

Ricerca binaria di un elemento in un vettore ordinato in senso non decrescente in cui il primo elemento è `first` e l'ultimo `last`

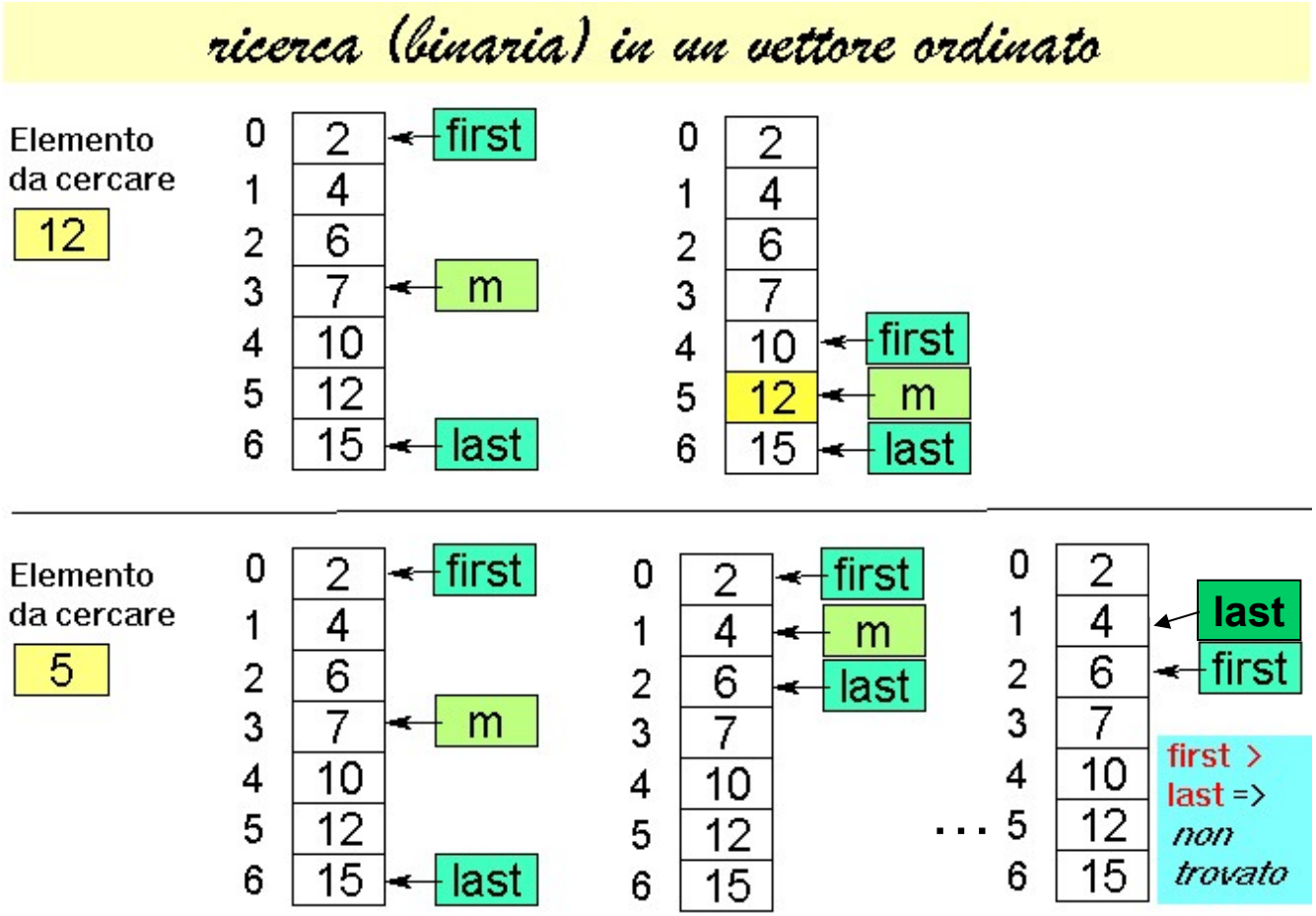
La tecnica di ***ricerca binaria***, rispetto alla ricerca esaustiva, consente di ***eliminare ad ogni passo metà degli elementi del vettore***

ESERCIZIO: RICERCA BINARIA

- Si confronta l'elemento cercato **`el`** con quello mediano del vettore, **`V[med]`**
- Se **`el==V[med]`**, fine della ricerca (**`trovato=true`**)
- Altrimenti, se il vettore ha almeno una componente (**`first<=last`**):
  - se **`el<V[med]`**, ripeti la ricerca nella prima metà del vettore (indici da **`first`** a **`med-1`**)
  - se **`el>V[med]`**, ripeti la ricerca nella seconda metà del vettore (indici da **`med+1`** a **`last`**)

ESERCIZIO: RICERCA BINARIA

Esempio



ESERCIZIO: RICERCA BINARIA

```
int ricerca_bin (int vet[], int el)
{int first=0, last=N-1, med=(first+last)/2;
 int T=0;
 while ((first<=last)&&(T==0))
  { if (el==vet[med])
          T=1;
    else
       if (el < vet[med])  last=med-1;
       else   first=med+1;
    med = (first + last) / 2;
  }
 return  T;
}
```

ESERCIZIO: Ricerca binaria di un elemento

```
#include <stdio.h>
#define N 15

int ricerca_bin (int vet[], int el);
int main ()
{int i;
 int a[N];
 printf ("Scrivi %d numeri interi ordinati\n", N);
 for (i = 0; i<N; i++)
     scanf ("%d", &a[i]);
 printf ("Valore da cercare: ");
 scanf ("%d",&i);
 if (ricerca_bin(a,i)) printf("\nTrovato\n");
     else printf("\nNon trovato\n");
}
```

OSSERVAZIONI

Si noti che la ricerca binaria può essere definita facilmente in ***modo ricorsivo***

Si noti infatti che si effettua un ***confronto dell'elemento cercato `el` con l'elemento di posizione media del vettore `V[med]`***

- Se l'elemento cercato è uguale si termina (caso base)
- Altrimenti se **`el < V[med]`** si effettua una ricerca binaria sulla prima metà del vettore
- Altrimenti (**`se el > V[med]`**) si effettua una ricerca binaria sulla seconda metà del vettore

Esercizio: si scriva procedura per ***ricerca binaria ricorsiva***

Ricerca binaria

- Definizione
  - Sia **`dim`** la dimensione dell'array
  - Se l'elemento mediano (posizione **`med`**) dell'array è l'elemento da cercare → ***elemento trovato***
  - Se l'elemento mediano dell'array è maggiore dell'elemento da cercare → ***cercare nella prima metà dell'array*** (dalla posizione "0" alla posizione **`med-1`**)
  - Se l'elemento mediano dell'array è minore dell'elemento da cercare → ***cercare nella seconda metà dell'array*** (dalla posizione **`med+1`** alla posizione "finale")
- La definizione è evidentemente ricorsiva...

Ricerca binaria ricorsiva

- Parametri in ingresso:
  - Array in cui cercare
  - Indice first da cui partire
  - Indice last a cui fermarsi nella ricerca
  - Elemento da cercare

- Valori in uscita:
  - Successo della ricerca
  - Posizione dell'elemento nell'array

  → I due valori sono "condensabili"?

  →La posizione in un array è sempre maggiore o uguale a zero

  →Un numero negativo può essere considerato un insuccesso nella ricerca...

RICERCA BINARIA RICORSIVA

```
int ricerca_bin (int vet[], int first, int last, int el) {

   int med;

   if (first > last)
        return -1;
   else {
        med = (first + last) / 2;
        if (el == vet[med])
               return med;
        else
               if (el > vet[med])
                      return ricerca_bin(vet, med+1, last, el);
               else
                      return ricerca_bin(vet, first, med-1, el);
   }
}
```

RICERCA BINARIA RICORSIVA
Versione “compatta”

```
int binarySearch(int vet[], int dim, int el) {
   int startPos,result;
   int med = dim / 2;

   if (vet[med] == el)
        return med;
   if (med == 0)
        return -1;
   if (el < vet[med])
   {
        return binarySearch(vet, med, el);
   }
   else {
        startPos = med + 1;
        result = binarySearch(&vet[startPos], dim - startPos, el);
        return ( result == -1) ? -1 : startPos + result;
    }
}
```


18

Ricerca binaria: note

- **`&vet[startPos]`**

  → Indirizzo dell'elemento di posizione **`startPos`**

  → Sotto-array parzialmente sovrapposto all'array di partenza (**`vet`**) i cui elementi sono quelli compresi fra **`startPos`** (compreso) e la fine dell'array

- **`startPos + binarySearch(&vet[startPos],`**
  `dim - startPos, el);`

  → La ricerca riparte dal sotto-array che inizia da **`startPos`**

  → occorre sommare la posizione di partenza al risultato della sottoricerca

  → la dimensione del sotto-array è **`dim - startPos`**

  → l'elemento da cercare è sempre lo stesso...

Ricerca binaria – estensione

- E se cambia il tipo di dato? Come permettere il riutilizzo di codice *(solo se necessario…)*?

Il tipo di dato DEVE essere dotato di una opportuna operazione di confronto:

- **`int compare(TYPE d1, TYPE d2);`**
- Il risultato è:
  - Positivo per **`d1`** maggiore **`d2`**
  - Nullo per **`d1`** uguale **`d2`**
  - Negativo per **`d1`** minore **`d2`**

Domande a cui sapere assolutamente rispondere in sede d'esame ☺

- Perché nell' interfaccia di una funzione che prevede il passaggio di array è possibile omettere la dimensione dell'array stesso?
- Quali sono le (piccole) differenze fra array e puntatori in C?
- È possibile cambiare *upper* e *lower bound* di un array?

Perché nella *intefaccia* di una funzione…

Nella ***definizione di un array*** la ***dimensione*** serve per ***allocazione della memoria***. A runtime, all'atto della chiamata di funzione, non viene effettuato alcun *bound checking* (attenzione, non c'è quindi alcun controllo!) → alla macchina runtime servono solo:

- indirizzo del primo elemento dell'array
- dimensione del tipo di dato contenuto dall'array

Definizione:

```
int myArray[53];
```

Passaggio:

```
void myProcedure(int anArray[])
{
        anArray[3] = 10;
}
```

Array e puntatori?

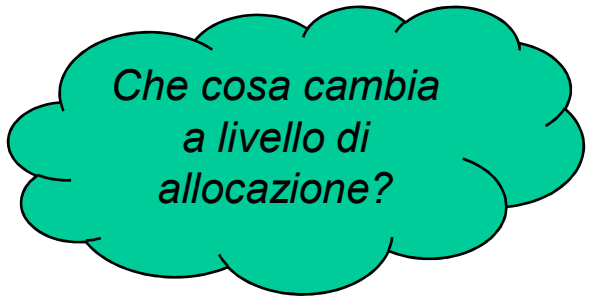
- La variabile che denota un array contiene ***l'indirizzo del primo elemento dell'array***…
- …tale indirizzo può essere ugualmente contenuto in un puntatore!

- Però la variabile che denota l'array è assimilabile a un ***puntatore costante*** (mantenuto in modo simile a quanto fatto in altri linguaggi per variabili con valori non modificabili), mentre un normale puntatore può ovviamente cambiare di valore:

```
int *p, a[5];
p = a; //Ok!
a = p; //Errore!
```

Array e puntatori?

Per il resto, che piaccia o meno, le notazioni di array e puntatori in C sono del tutto simili e possono essere usate in modo mescolato




```
int *p, a[5];
p = a;
p[1] = 4;
*(a+2) = 3;
p = &a[2];
```

. . . *upper* e *lower bound*?

In C

- Il *lower bound* di un array è sempre 0, l'*upper bound* è la dimensione dell'array meno 1
- *Upper* e *lower bound* degli array non vengono verificati:

```
int i, a[4];
i = a[-2];
```

Non genera errore di compilazione ma "solo" eventuale errore a runtime

- Usando le proprietà di array e puntatori è possibile ottenere un "array" dove *upper* e *lower bound* sono diversi dal solito

```
int i, *p, a[5];
p = &a[2];
for (i = -2; i <= 2; i++)
    p[i] = i;
```

Lower bound = -2; Upper bound = 2

Si usa `p` come se fosse un normale array… un po' speciale!

…*upper* e *lower bound*?

- Si supponga di voler fare in modo che il *lower bound* di un array sia 1 → potrebbe aver senso in quanto il primo elemento sarebbe l'elemento di indice 1…

```
int a[5], *p;
p = &a[-1]; // p = a - 1;
p[1]…
```

è il primo elemento; p[0] è l'elemento -1esimo: occhio!

…upper e *lower bound*?

- Attenzione: cambiare le convenzioni è sempre pericoloso
- La cosa deve essere altamente giustificata, per esempio per far aderire meglio il programma al sistema che si sta modellando… ma anche in quel caso…
- Fra le altre cose, cambiare upper e lower bound rende il programma meno leggibile
- Un altro discorso è voler estrarre un sotto-array:

```
int a[5], *sa;
sa = a + 2;
```

Sa = sotto-array che comincia due elementi più avanti → vedi ricerca binaria ricorsiva

C’era una volta un hacker…

Calcolo della lunghezza di una stringa

- Versione 0

```
int lunghezza(char s[])
{
   int lung;
   for (lung=0; s[lung]!='\0'; lung++);
   return lung;
}
```

- Versione 1

```
int lunghezza(char *s)
{
   int lung=0;
   for (lung=0; s[lung]!='\0'; lung++);
   return lung;
}
```

C'era una volta un hacker…

- Versione 2

```
int lunghezza(char *s)
{
   char *s0 = s;
   while (*s) s++;
   return s-s0;
}
```

- Versione 3

```
int lunghezza(char *s)
{
   char *s0 = s;
   while (*s++);
   return s-s0-1;
}
```

1. Viene dereferenziato il puntatore e usato nel test

2. Viene incrementato il puntatore (e non il valore puntato)

*→ Gli operatori unari * e ++ sono equiprioritari e associativi da destra a sinistra!*

Puntatori ed Operatori

Associatività degli operatori

- Dereferencing e incremento/decremento hanno la stessa priorità ed associatività da destra a sinistra.
- *p++ equivale a *(p++) p viene incrementato **dopo il suo utilizzo per accedere alla memoria.**
- *++p equivale a *(++p) p viene incrementato **prima del suo utilizzo per accedere alla memoria.**
- ++*p equivale a ++(*p) p e' utilizzato per l'accesso alla memoria, ma il contenuto della locazione viene incrementato **prima del suo utilizzo.**
- (*p)++ equivale a (*p)++ p e' utilizzato per l'accesso alla memoria; il contenuto della locazione viene incrementato **dopo del suo utilizzo.**

Esempio

```
int *p;
int b[10];
p = b;      /* *p equivale a b[0] */
*p++ = 5;   /* b[0] = 5; *p equivale a
  b[1] */
p = b;
*++p = 5;   /* b[1] = 5; *p equivale a
  b[1] */
 p = b;
*p = 5;         /* b[0] = 5 */
b[1] = ++*p;   /* b[1] = 6; b[0] = 6; *p
  equivale a b[0] */
```