Programmazione I

Il Linguaggio C

Le Funzioni

Daniel Riccio

Università di Napoli, Federico II

22 novembre 2021

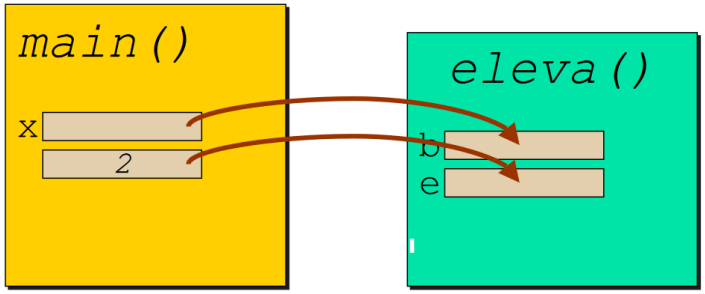
Sommario

- Argomenti
  - Le funzioni
  - Chiamata di una funzione e ritorno da una funzione
  - Parametri formali e parametri attuali
  - Passaggio diparametri

Passaggio dei parametri

I dati possono essere passati ad una funzione esclusivamente per valore (by value):

alla chiamata della funzione vengono create nuove variabili con i nomi di ciascuno dei parametri formali e in esse viene copiato il valore del corrispondente parametro attuale



Come per le assegnazioni, se il **parametro attuale** e il corrispondente **formale** sono di tipo diverso c'è una conversione automatica al tipo del **parametro formale** (se è di tipo meno capiente può essere generato un Warning)

Poiché in memoria i parametri formali e quelli attuali sono completamente **distinti** e **indipendenti**, cambiare il valore di un parametro formale non modifica il parametro attuale corrispondente, neppure se questo è una semplice variabile (è ovviamente impossibile modificare una costante o il risultato di un'espressione): nell'esempio visto **la modifica di b non si ripercuote su x**

Variabili locali static

Le variabili locali hanno **classe di allocazione automatica**:
vengono create ogni volta che si esegue la funzione ed eliminate ogni volta che questa termina (perdendone il valore)

Le variabili locali di **classe di allocazione statica** invece non vengono mai rimosse dalla memoria per cui non perdono il loro valore quando la funzione termina (resta quello che aveva al termine della chiamata precedente)

Le variabili statiche non richiedono la ri-allocazione della memoria ad ogni chiamata della funzione, quindi il programma può essere più veloce

Variabili locali static

Si richiede una classe di allocazione statica e non automatica mediante lo specificatore di classe di allocazione **static**:

```
static int cont = 0;
```

Se non inizializzate esplicitamente, vengono inizializzate automaticamente a 0 (che nel caso dei puntatori viene automaticamente convertito in NULL)

L'inizializzazione delle variabili locali **static** avviene idealmente solo la prima volta che si esegue la funzione (in realtà i valori vengono inizializzati dal compilatore)

Variabili locali static

Possono essere inizializzate dal compilatore solo con espressioni costanti:

- numeri e #define
- indirizzi di memoria di variabili statiche

Non possono essere inizializzate con:

- valori const
- variabili e risultati di funzioni
- indirizzi di memoria di variabili automatiche

Variabili locali static

```
int conta(void)
{
    static int cont = 0;

    return ++cont;
}
```

Ogni volta che conta viene chiamata, essa incrementa il contatore **cont** e ne restituisce il valore, se non fosse statica ma automatica restituirebbe sempre il valore 1 perché **cont** verrebbe re-inizializzata ogni volta a **0**

Variabili locali static

```
char *nomeMese(int n)
{
    static char *nome[] = {"inesistente", "gennaio", "febbraio",
                           ecc...};

    if (n<1 || n>12)
        return nome[0];
    else
        return nome[n];
}
```

La **stringa** di cui viene restituito il puntatore può essere utilizzata dal chiamante in quanto essendo **statica** non viene rimossa dalla memoria

Passaggio dei parametri

Il passaggio di parametri nella modalità **per riferimento** (**by reference**) prevede che la modifica del parametro formale si ripercuota identica sul corrispondente **parametro attuale** (deve essere una variabile)

In C non esiste il passaggio per riferimento, ma questo può essere **simulato passando per valore alla funzione il puntatore al dato** da passare (che deve essere una variabile, non può essere il risultato di un calcolo)

Nella **scanf** le variabili scalari sono precedute da **&** perché devono essere modificate dalla funzione e quindi se ne passa l'indirizzo

Passaggio dei parametri

```
#include <stdio.h>

void fz(int *);          prototipo

int main()
{
  int x=2;

  fz(&x);
  printf("%d\n", x);

  return EXIT_SUCCESS;
}

void fz(int *p)
{
  *p = 12;
}
```

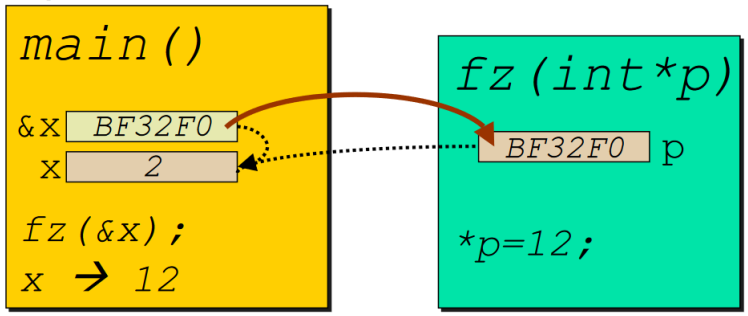
Nell'esempio:

il main alloca **x**, gli assegna il valore **2** e chiama **fz** passandole l'indirizzo di **x** (**&x**) presente in una variabile temporanea ("senza nome")

alla chiamata di **fz**, l'indirizzo di **x** (che è **BF32F0**) viene copiato by-value in **p**

fz accede a **x** come ***p**, modificandola in **12**

quando **fz** termina, **x** vale **12**



Passaggio dei parametri

```
void swap(int *, int *);
```

prototipo

```
int main()
{
  int x=12, y=24;

  swap(&x, &y);
}

void swap(int *a, int *b)
{
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

Passaggio di vettori

Per passare un vettore come argomento, si indica il suo nome senza parentesi
x = **media**(vettore);

Il parametro formale corrispondente definirà un vettore dello stesso tipo (in genere senza dimensione in quanto ininfluente)
float media(**int** v[]);

La funzione deve conoscere in qualche modo la dimensione del vettore (indicarlo tra le parentesi quadre non serve a nulla):

- viene passato come argomento
  float media(**int** v[], **int** lung);
- è noto a priori (es. una **#define**)
- si usano variabili esterne

Passaggio di vettori

Possono essere passati vettori con dimensioni diverse, ma dello stesso tipo **T**

Quando si passa un **vettore-di-T**, poiché si indica il nome del vettore, in realtà si passa l'indirizzo di memoria del suo primo elemento (e questa non dà alcuna informazione né restrizione sulla lunghezza del vettore)

Il parametro formale è quindi in realtà un **puntatore-a-T**, la forma **v[]** viene convertita automaticamente in ***v**, si possono usare le due definzioni indifferentemente

```
float media(int *v);
```

Passaggio di matrici

Per passare una matrice come argomento, si indica il suo nome senza parentesi

x = **media**(**matrice**);

Il parametro formale corrispondente dichiara una matrice dello stesso tipo (in genere senza la prima dimensione in quanto ininfluente)
void **funz**(**int** **matrice**[][10])

La funzione deve conoscere in qualche modo le dimensioni della matrice (indicarle tra le parentesi quadre non serve a questo scopo)

Passaggio di matrici

Possono essere passate matrici con **diverso numero di righe**, ma devono avere lo **stesso numero di colonne** e gli elementi devono essere dello **stesso tipo T**

Poiché una matrice è un vettore-di-vettori, quando essa viene passata ad una funzione, viene in realtà passato l'indirizzo del primo elemento del vettore-di-vettori

Il tipo del parametro formale corrispondente è quindi un **puntatore-a-vettore** e **NON un puntatore-a-puntatore**

Const per puntatori a puntatori

```
int ** x;
```
x è …

una variabile di tipo puntatore a un puntatore variabile a un oggetto variabile di tipo `int`

```
const int ** x;
```
x è …

una variabile di tipo puntatore a un puntatore variabile a un oggetto costante di tipo `int`

```
int ** const x;
```
x è …

una costante di tipo puntatore a un puntatore variabile a un oggetto variabile di tipo `int`

```
int * const * x
```
x è …

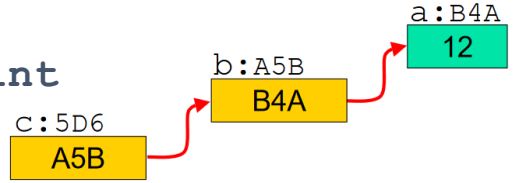
una variabile di tipo puntatore a un puntatore costante a un oggetto variabile di tipo `int`

```
const int * const * x
```
x è …

una variabile di tipo puntatore a un puntatore costante a un oggetto costante di tipo `int`

```
const int * const * const x
```
x è …

una costante di tipo puntatore a un puntatore costante a un oggetto costante di tipo `int`



Passaggio di matrici

Nel caso dell'esempio, il parametro formale è un puntatore-a-vettore-di-10-int:

```
int (*matrice)[10]
```

La forma **matrice**[][10] viene convertita automaticamente in (***matrice**)[10], si possono usare le due definzioni indifferentemente

```
void funz(int (*matrice)[10])
```

E' quindi un errore scrivere:

```
void funz(int **matrice)
```

oltre all'errore di tipo, si perde la dimensione delle colonne e quindi non si può determinare la posizione degli elementi della matrice

Passaggio di matrici e vettori multidimensionali

In memoria l'elemento **Mx**[i][j] viene determinato con il calcolo **indirizzo_di_Mx**+**NC***i+j dove **NC** è il numero delle colonne, come si vede il numero delle righe non serve

Per passare ad una funzione una matrice con qualsiasi numero di righe e qualsiasi numero di colonne si fa ricorso all'allocazione dinamica

Quanto visto per le matrici può essere esteso ai **vettori multidimensionali**. In particolare:

- nel parametro formale si può tralasciare la dimensione del solo primo parametro
- il parametro formale è un puntatore ad un vettore di X vettori di Y vettori di Z vettori di .... di tipo T
- la funzione deve conoscere i valori di tutte le dimensioni
- possono essere passate matrici con la sola prima dimensione diversa

Parametri const

Il modificatore **const** applicato ai **parametri formali** impedisce che all'interno della funzione si possa modificarne il valore

int funzione(**const int** v)

Permette di proteggere i parametri da una successiva incauta modifica (per prevenire errori di programmazione)

Ad esempio, questo richiede al compilatore di segnalare se un **puntatore-a-dato-costante** viene assegnato ad un **puntatore-a-dato-variabile** (ad esempio passandolo come parametro), cosa che by-passerebbe la restrizione

Parametri const

Puntatore variabile a dati variabili

```
int f(int *p)
{
  *p = 12;
   p++; }
```

➡ OK, dato variabile

➡ OK, puntatore variabile

Si può passare un **int***

Non si può passare un **const int***
(non c'è conversione automatica di tipo perché dentro la funzione nulla vieterebbe di poter cambiare il valore alla variabile puntata):

```
int x=12;
const int *y=&x;
f(y);
```

➡ **ERRORE**

Parametri const

Puntatore variabile a dati costanti

```
int f(const int*p)  /* int const */
{*p = 12;          ➡  NO, dato costante
p++;}              ➡  OK, puntatore variabile
```

Si può passare un **int*** (c'è conversione di tipo automatica in quanto si passa ad un tipo più restrittivo)

Si può passare un **const int ***

Note:

Tipicamente utilizzato per passare un puntatore ad una **struct** o ad un **vettore** impedendo che possano essere modificati

Non si può passare senza cast un **Tipo**** dove è richiesto un **const Tipo****

Parametri const

Puntatore costante a dati variabili

```
int f(int *const p)
{*p = 12;
p++;}
```

→ OK, dato variabile

→ NO, puntatore costante

Si può passare un **int ***

Puntatore costante a dati costanti

```
int f(const int * const p)
{*p = 12;
p++;}
```

→ NO, dato costante

→ NO, puntatore costante

Si può passare un **int ***

Variabili esterne

Vengono definite (riservando memoria) esternamente al corpo delle funzioni:

- in testa al file, tipicamente dopo le direttive **#include** e **#define**
- oppure tra una funzione e un'altra

Sono **visibili** e **condivisibili** da tutte le funzioni che nello stesso file seguono la definizione

Possono essere utilizzate come metodo alternativo per comunicare dati ad una funzione e per riceverne

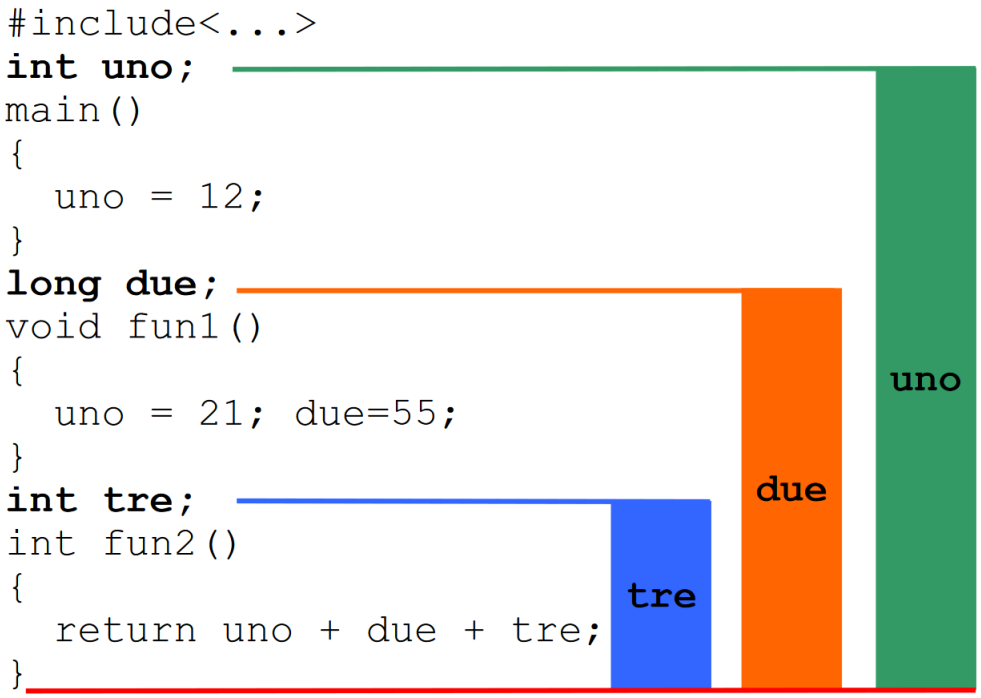
A questo scopo si usino con parsimonia:
rendono poco evidente il flusso dei dati all'interno del programma

Variabili esterne

Hanno classe di allocazione statica:

- non vengono mai rimosse dalla memoria e, salvo inizializzazione esplicita, vengono inizializzate automaticamente a 0 (i puntatori a NULL)

Una variabile locale (interna ad una funzione) con lo stesso nome di una esterna copre la visibilità di quella esterna alla quale quindi non può accedere con quel nome (non è buona pratica di programmazione)



Variabili esterne

Lo specificatore di classe di allocazione **extern** permette di estendere la visibilità delle variabili esterne

La clausola **extern** viene premessa ad una definizione di variabile per trasformarla in dichiarazione (non riserva memoria)

```
extern int x;
```

Indica al compilatore che la variabile è definita (con allocazione di memoria, senza extern) altrove (più avanti nello stesso file o in un altro file)

In seguito il linker ricondurrà tutte le dichiarazioni all'unica definizione

Documentazione delle funzioni

È utile scrivere sempre la **documentazione** relativa allo scopo e all'uso di una funzione come commento iniziale contenente:

Scopo: a cosa serve la funzione
Parametri: tipo e descrizione di ciascuno
Valore restituito:tipo e descrizione

Possibilmente anche:
Pre-condizioni: requisiti particolari sui parametri che devono essere soddisfatti da chi invoca la funzione (es. param > 29)

Post-condizioni: garanzie date dalla funzione sul valore restituito o sullo stato del programma, purché le precondizioni siano state soddisfatte (es. risultato >= 23 && risultato <= 32)

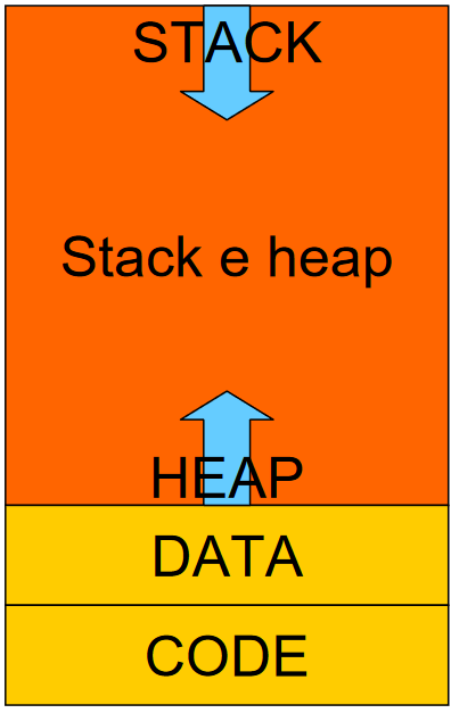
Chiamata di funzioni - dettagli

Il programma compilato è costituito da due parti distinte:

code segment - codice eseguibile
data segment - costanti e variabili note alla compilazione (statiche ed esterne)

Quando il programma viene eseguito,
il **Sistema Operativo** alloca spazio di memoria per:

- il code segment (CS)
- il data segment (DS)
- lo stack e lo heap (condiviso)



Chiamata di funzioni - dettagli

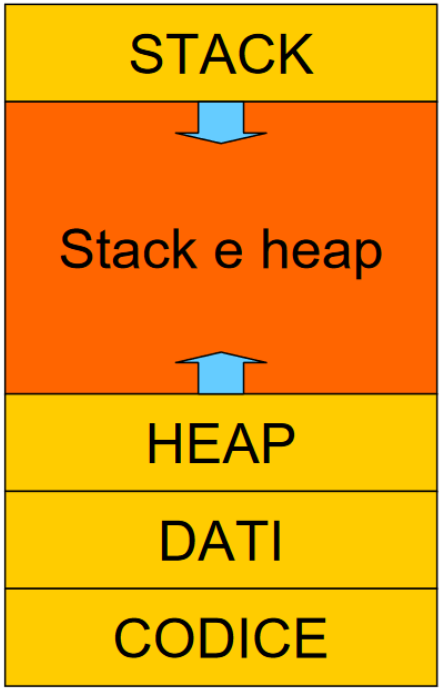
Lo **stack** contiene inizialmente le variabili locali della funzione **main**()

Lo **heap** inizialmente è vuoto e serve per contenere i blocchi di memoria allocati dinamicamente con funzioni **malloc**() (trattate in altre slide)

Stack e **heap** crescono nell'area condivisa nel senso indicato dalle frecce

Quando viene chiamata una funzione, sullo stack vengono prima copiati i valori dei suoi argomenti e poi vi viene allocato un **Activation Record** (o stack frame) per contenere tutte le variabili locali (e altro)

Quando la funzione termina, gli argomenti e l'AR vengono rimossi dallo **stack** che quindi ritorna nello stato precedente la chiamata

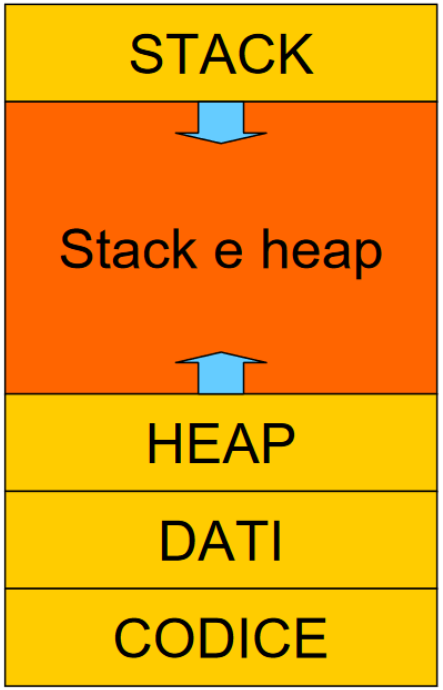


Chiamata di funzioni - dettagli

Nell'**Activation Record** viene anche memorizzato l'indirizzo di ritorno dalla funzione: la locazione di memoria che contiene l'istruzione del chiamante da cui continuare dopo che la funzione è terminata

Queste operazioni di allocazione e deallocazione di spazio sullo stack e in generale il meccanismo di chiamata e ritorno da una funzione richiedono tempo

In casi estremi di necessità di elevate performance si può cercare di limitare il numero delle chiamate a funzione, a costo di ricopiare lo stesso codice in più punti (eventualmente utilizzando macro con argomenti)



Progetti multi-file

È possibile suddividere le funzioni che costituiscono un eseguibile su più file (detti translation unit)

Ciascun file viene compilato separatamente e il **linker** li assembla per costituire un unico file eseguibile

Uno solo dei file deve contenere la definizione della funzione principale **main**

L'insieme dei file sorgenti viene spesso chiamato progetto

In ciascun file si collocano funzioni che insieme forniscono una certa funzionalità

Ciascuno dei file si comporta come una libreria di funzioni, salvo che queste vengono compilate (e non solo linkate) con il programmma principale

Un file con funzioni specifiche per fornire una determinata funzionalità può essere facilmente riutilizzato in altri programmi: basta **includerlo nel progetto**

Progetti multi-file

Ciascun file ha bisogno delle sole direttive **#include** e **#define** necessarie al codice di quel file

Per usare in un file una funzione dichiarata in un altro file (non può essere **static**), si deve indicarne il prototipo (extern opzionale)

Spesso si raggruppano tutte le **#define**, le variabili esterne e i prototipi di tutte le funzioni (non può essere **static**) di un progetto in un unico file **.h** e i file **.c** del progetto che ne abbisognano lo includono (con virgolette): **#include** "**mioheader.h**"

Le variabili esterne usate in tutti i file devono essere definite solo in uno dei file, mentre gli altri devono avere la dichiarazione extern corrispondente

Non si può usare **extern** con variabili esterne con specificatore di classe di allocazione **static** in un altro file (sono utilizzabili solo dalle funzioni di quel file)

Linkage di variabili e funzioni

Il **linkage** esprime la corrispondenza di identificatori (variabili o costanti) omonimi presenti in più blocchi e/o in più **translation unit** diverse (linkate insieme) e/o in librerie

Un identificatore con **linkage esterno** è visibile in più translation unit (es. variabili e funzioni esterne non static)

Un identificatore con **linkage interno** è visibile solo nella translation unit dove è definito (es. variabili e funzioni static)

Un identificatore non ha linkage se è **locale al blocco** dove è definito (es. variabili locali, parametri di funzioni, tag, membri, etc.)

Esercizio

Scrivere un programma che legga da input un valore **nmax** e allochi un vettore di interi lungo **nmax**. L'utente deve poter inserire un numero arbitrario di valori ordinati **n**, con **n**<**nmax**. Inoltre, l'utente inserisce un valore val da cercare nel vettore.
Il programma implementa una ricerca **dicotomica** e stampa la posizione in cui è stata trovata la prima occorrenza di val oppure un messaggio "**Valore non trovato**".
Il programma implementa una ricerca **lineare** e stampa la posizione in cui è stata trovata la prima occorrenza di val oppure un messaggio "**Valore non trovato**".
Implementare il programma mediante le seguenti funzioni:

```
int LeggiInput(int *v, int nmax);
int CercaElementoLineare(int *v, int val, int n);
int CercaElementoDicotomica(int *v, int val, int n);
```

```
Inserisci il numero di valori: 5
Inserisci i valori in input (-1) per terminare
Inserisci valore: 1
Inserisci valore: 5
Inserisci valore: 7
Inserisci valore: 9
Inserisci valore: 10
Valori inseriti: 1 5 7 9 10
Valore da cercare: 9
Elemento trovato in posizione 4
Elemento trovato in posizione 4
```

Esempio: prototipi delle funzioni

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

int LeggiInput(int *v, int nmax);
int CercaElementoLineare(int *v, int val, int n);
int CercaElementoDicotomica(int *v, int val, int n);
```

Esempio: funzione main

```
int main()
{
    int i=0;
    int n = 0;
    int ninseriti = 0;
    int posizione = 0;
    int *v = NULL;
    int val=0;


    printf("Inserisci il numero di valori: ");
    scanf("%d", &n);
```

Esempio: funzione main

```
    ninseriti = LeggiInput(v, n);

    printf("Valori inseriti: ");
    for(i=0; i<ninseriti; i++)
       printf("%d ", v[i]);
    printf("\n");

    printf ("Valore da cercare: ");
    scanf ("%d",&val);

   posizione = CercaElementoDicotomica(v, val, ninseriti);

   if(posizione == -1)
      printf("Elemento non trovato\n");
   else
      printf("Elemento trovato in posizione %d\n", posizione);

   posizione = CercaElementoLineare(v, val, ninseriti);

   if(posizione == -1)
      printf("Elemento non trovato\n");
   else
      printf("Elemento trovato in posizione %d\n", posizione);


   free(v);
}
```

Esempio: LeggiInput

```
int LeggiInput(int *v, int nmax)
{
    int n=0;
    int val = 0;

    printf("Inserisci i valori in input (-1) per terminare\n");

    do{
        printf("Inserisci valore: ");
        scanf("%d", &val);

        if(val != -1){
            v[n++] = val;
        }

    }while(val != -1 && n<nmax);

    return n;
}
```

Esempio: CercaElementoDicotomica

```
int CercaElementoDicotomica(int *v, int val, int n)
{
    int Trovato=0;
    int Primo=0;
    int Ultimo=n-1;
    int Medio;

    while ((Primo<=Ultimo) && (Trovato==0)){

        Medio = (Primo+Ultimo)/2;

        if (val==v[Medio])
            Trovato = 1;
        else if(val<v[Medio])
            Ultimo = Medio - 1;
        else
            Primo = Medio + 1;
    }

    if(Trovato){
        return Medio+1;
    } else {
        return -1;
    }
}
```

Esempio: CercaElementoLineare

```
int CercaElementoLineare(int *v, int val, int n)
{
    int i=0;

    for(i=0; i<n; i++)
      if(v[i]==val)
         return i+1;

    return -1;
}
```