Programmazione

Giovanni Da San Martino

Dipartimento of Matematica, Università degli Studi di Padova
giovanni.dasanmartino@unipd.it
A.A. 2021-2022

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Esercitazione 1

- Correttezza: prestate attenzione ai casi particolari (accesso ad elementi di un vettore oltre la loro dimensione) oppure l'utilizzo di una chiave negativa
- Efficienza: a parte dettagli non erano possibili algoritmi significativamente più veloci di altri
- Organizzazione del codice: poiché la chiave di cifratura può essere negativa, è conveniente utilizzare una sola funzione per la codifica e la decodifica (è sufficiente invocare la funzione di codifica con -1*k per decodificare)
- Stile: I commenti (e le PRE/POST) hanno lo scopo minimizzare il tempo per capire il vostro codice. Se mettete troppi commenti, si impiegherà più tempo per leggere il vostro codice

Variabili Globali

- Variabili definite fuori da ogni funzione sono dette globali, sono visibili in tutto il file da dove sono definite in poi (a meno che non si definisca una variabile con lo stesso nome in una funzione – in questo caso la variabile globale non è visibile nella funzione).
- Le variabili globali sono da in generale da evitare (ogni funzione dovrebbe modificare solamente variabili locali e parametri). Si usano nel caso una variabile debba essere usata in molte funzioni e si debba mantenerne il valore tra ogni chiamata

```
#include <stdio.h>
int x=10;


int main(void) {
        //int x = 8; //questa dichiarazione rende invisibile la variabile globale nel main
        printf("%d\n", x); //se tolgo il commento all'istruzione precedente stampa 8
}
```

Variabili Globali in più File

- Se vogliamo utilizzare una variabile globale definita in un altro file, dobbiamo ridichiarare la variabile con la parola chiave extern davanti alla dichiarazione
- Se non esiste la variabile counter in file1.h non si ottiene un errore compilando main.c (diventa una variabile diversa)

file1.h

```
#ifndef FILE1_H
#define FILE1_H

int counter;

#endif
```

main.c

```
#include “file1.h”
extern int counter;

int main(void) {
        counter = 5;
        printf(“%d\n”, counter);
}
```

Variabili Statiche

Variabili Globali

- create all'inizio dell'esecuzione
- mantengono il loro valore per tutta l'esecuzione
- sono accessibili per tutta l'esecuzione*

Variabili Locali

- create quando il blocco dove sono definite viene eseguito
- mantengono il loro valore finché si esegue il blocco
- sono accessibili all'interno dello stesso blocco*

Variabili Statiche

- create all'inizio dell'esecuzione
- mantengono il loro valore per tutta l'esecuzione
- sono accessibili all'interno del blocco dove sono definite*

Variabili Statiche

- le variabili dichiarate statiche mantengono il valore tra due invocazioni della funzione dove sono definite
- L'inizializzazione della variabile viene fatta solamente una volta
- il programma alla destra stampa

1

2

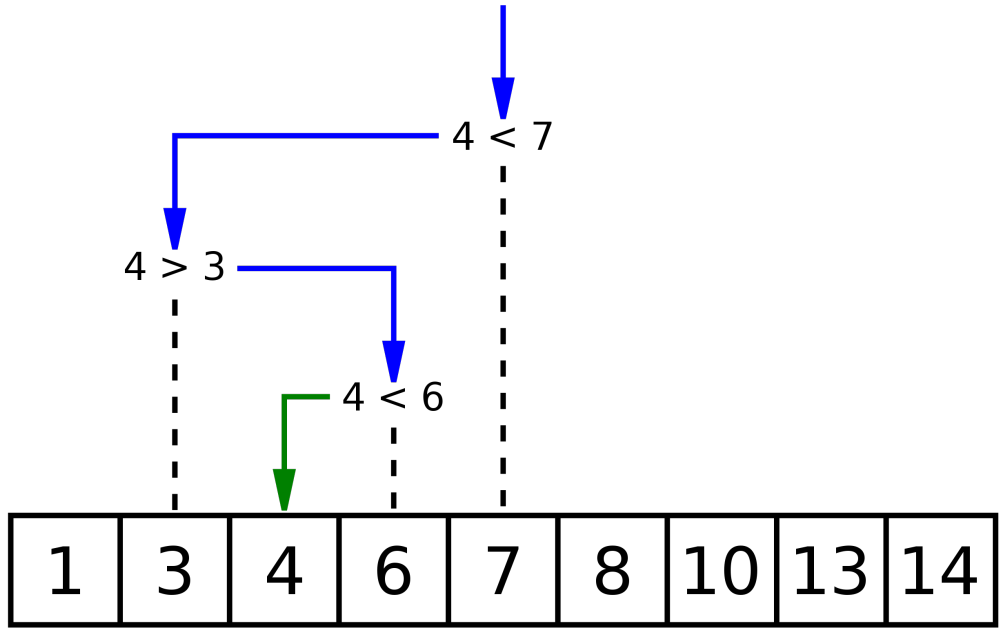
```
int contatore(void) {
    static int c = 0; //eseguita una volta sola
    c += 1;
    return c;
}

int main (void) {
    printf("%d\n", contatore());
    printf("%d\n", contatore());
}
```

Variabili Globali Statiche

- La visibilità di una variabile static è limitata al file dove è definita
- Se il qualificatore static viene usato per una variabile globale, questa non potrà essere riferita in un altro file (tramite extern)
  - se si usa extern, verrà creata una variabile
- si può utilizzare static con una funzione per renderla visibile solamente nel file dove è definita

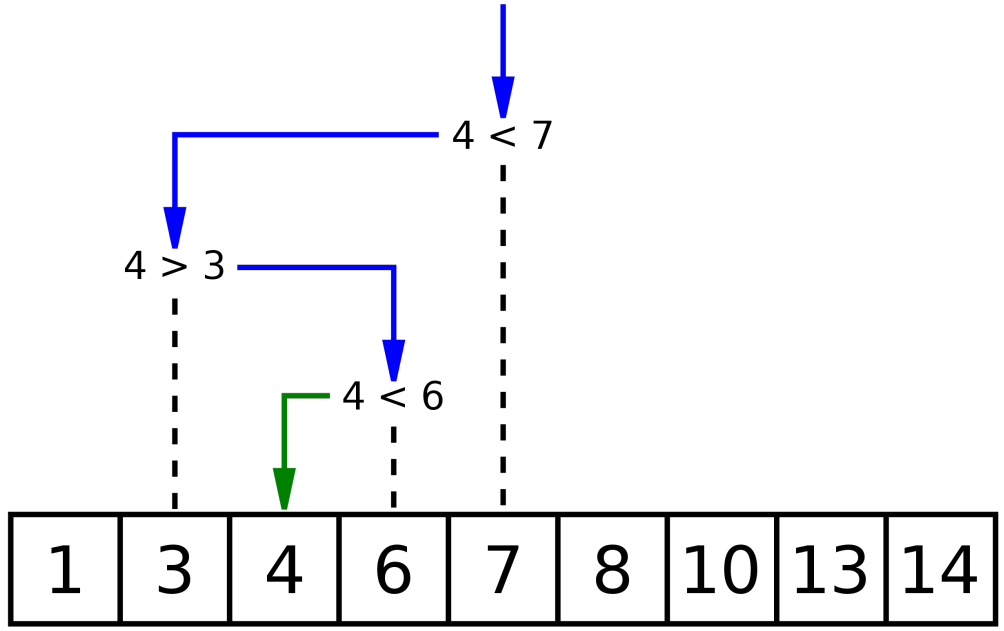
Ricerca Binaria


```
int ricerca_binaria(int *X, int dim, int elem) {
/*
    PRE: X ordinato in modo crescente
    POST: restituisce 1 se elem è in X, 0 altrimenti
*/
/*
confronta elem con l'elemento centrale dell'array; se è minore elem può solo essere nella metà di sinistra (X è ordinato), altrimenti può solo essere a destra. Ripeti l'algoritmo con la metà dell'array prescelta.
*/
}
```



Ricerca Binaria


```
int ricerca_binaria(int *X, int dim, int elem) {
   /* PRE: X ordinato in modo crescente
        POST: restituisce 1 se elem è in X, 0 altrimenti */
   int n = dim/2;
   if (dim<=0)
      return 0;
   if (elem==X[n])
      return 1;

   if (elem<X[n])
      return ricerca_binaria(X, n, elem);
   else
      return ricerca_binaria(X+n+1, dim-n-1, elem);
}
```



Ricerca Binaria


```
int ricerca_binaria(int *X, int dim, int elem) {
   /* PRE: X ordinato in modo crescente
        POST: restituisce 1 se elem è in X, 0 altrimenti */
   int n = dim/2;
   if (dim<=0)    // *1
      return 0;
   if (elem==X[n])
      return 1;   // *2

   if (elem<X[n])
      return ricerca_binaria(X, n, elem);  //*3
   else
      return ricerca_binaria(X+n+1, dim-n-1, elem); //*4
}
```

*1 l'array vuoto non contiene elem; *2 elem trovato

*3 ricerca_binaria()==1 => elem trovato; se ricerca_binaria()==0 elem non è in X[0],...,X[n-1], ma elem<X[n]<=X[n+1]<=...<=X[dim-1], quindi possiamo concludere che elem non sia in X e restituire 0 (*4 stesso ragionamento di *3)

Notate che stiamo usando l'ipotesi P(n/2)=>P(n) (possiamo usare qualsiasi n'<n)

Algoritmi Divide and Conquer

- Divide and Conquer consiste in

1. Dividere il problema in sottoproblemi più piccoli.
2. Risolvere ricorsivamente i sottoproblemi
3. Combinare le soluzioni dei sottoproblemi per ottenere la soluzione del problema originale

- Notate che la ricorsione è parte integrante della strategia di risoluzione del problema

Mergesort

- Mergesort è un algoritmo che utilizza la strategia divide and conquer per ordinare un array di valori
- Dato un array X di dimensione n,

1. richiama mergesort sulla prima metà dell'array ottenendo X_1
2. richiama mergesort sulla seconda metà dell'array ottenendo X_2
3. combina gli array ordinati X_1, X_2

```
MERGE-SORT(A, p, r) {
        if (p<r) {
                q = (p + r)/2;
                MERGE-SORT(A, p, q);
                MERGE-SORT(A, q + 1, r);
                combina_array(A, p, q, r);
        }
}
```

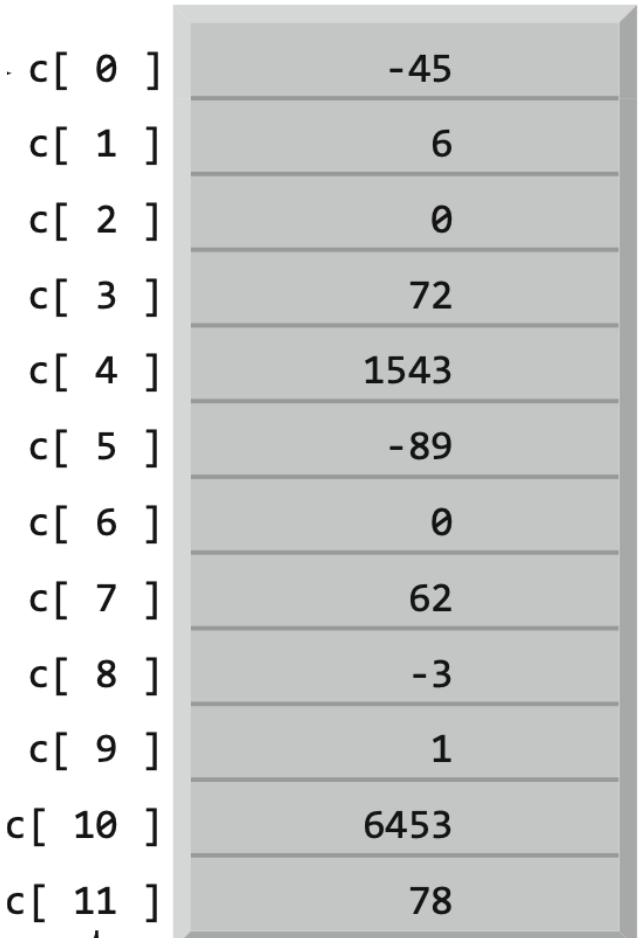
Passaggio di Parametri: Matrici

- int c[12]; definisce 12 variabili in memoria
- c[i] equivale a *(c+i)
- se passiamo c come parametro ad una funzione, il C lo trasforma in un puntatore ad intero
  - int f(int a[]); int f(int *a); sono equivalenti
  - Nel main si invoca con f(c);
- int c[3][4]; definisce una matrice di 3 righe, ciascuna con 4 elementi
- sappiamo che c[2] equivale a *(c+2*4); per eseguire l'operazione il C deve sapere che la matrice c ha 4 colonne, questa informazione deve essere mantenuta nel tipo di c

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Passaggio di Parametri: Matrici

- c[2] equivale a *(c+2*4); per eseguire l'operazione il C deve sapere che la matrice c ha 4 colonne
- Ricordiamo che 2 viene moltiplicato per la dimensione del tipo delle variabili che compongono l'array, in questo caso ciascuna riga punta ad un vettore di 4 elementi
- il tipo di c (c[3][4]) è int (*p)[4], perché le parentesi?
- poiché [] ha priorità su *, int *p[4] si legge: p è un array di 4 elementi (ciascuno) int *, ovvero un array di 4 puntatori ad intero
- int (*p)[4] si legge: p è un puntatore ad un array di 4 interi



Passaggio di Parametri: Matrici

- c[2][1]: equivale a

= (c[2])[1] =

= (*(c+2*sizeof(int (*) [4]))[1] =

= (*(c+2*4))[1] = // il tipo di c e c[2] è int (*) [4]

// *(c) è int[4]

// se x = (*(c+2*4)) e x è di tipo int[4]

= *(x+1*sizeof(int)) =

= *(x+1)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Esercizio su Scoping

https://www.wooclap.com/GTQNSZ

- Cosa stampa il codice a destra?

```
#include <stdio.h>

int c = 2;

int main(void) {

    int a = 3;
    {
        a += 1;
        int c;
        printf("%d\n", c);
    }
    printf("%d,%d\n", a, c);
}
```

Esercizi

- (Senza eseguirlo al calcolatore), cosa stampa il seguente codice? Perché?

```
int x;
int y=2;
int *p, *q = &y;
int **qq = &p;
**qq = 3;
printf("%d\n", **qq);
```

Esercizi

Definire una funzione ricorsiva che determini se esiste un percorso che permetta di attraversare un campo fiorito, dal basso verso l'alto, senza calpestare alcun fiore.

Il campo è rappresentato da una matrice, i cui valori rappresentano la presenza di un fiore (0) oppure la sua assenza (1).

E' possibile muoversi una casella in alto oppure una casella verso destra.

{0,0,0,1,0},

{0,1,0,1,0},

{1,0,0,1,0},

{1,0,1,1,1},

{1,0,1,0,0}

Esercizi

- L'esercizio su campo fiorito, sarebbe stato possibile risolverlo con una funzione ricorsiva se le mosse possibili fossero state 3: muovi di una casella a sinistra, muovi di una casella in alto, muovi di una casella a destra? Perché?

{0,0,0,1,0},

{0,1,0,1,0},

{1,0,0,1,0},

{1,0,1,1,1},

{1,0,1,0,0}

Esercizio


```
int mossa(int dim_x, int dim_y, int campo[dim_x][dim_y], int pos_x, int pos_y) {

   if ( (pos_x<0) || (pos_y<0) || (pos_y>=dim_y) || (pos_x>=dim_x) ) {
         return 0; //mossa fuori dal percorso
   }
   if (campo[pos_x][pos_y]==0)
      return 0; // calpestato un fiore, percorso non valido
   if (pos_x==0)
      return 1; // sono arrivato nella prima riga, percorso valido
   return (
      (mossa(dim_x, dim_y, campo, pos_x-1, pos_y)) ||
      (mossa(dim_x, dim_y, campo, pos_x, pos_y-1)) ||
      (mossa(dim_x, dim_y, campo, pos_x, pos_y+1))
   );
}
```

Percorso:

{0,0,0,1,0},

{0,1,0,1,0},

{1,0,0,1,0},

{1,0,1,1,1},

{1,0,1,0,0}

mossa(3,3) ha successo, perché?