Lezione 6

Struct e qsort

Rossano Venturini

rossano.venturini@unipi.it

Pagina web del corso

http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start

Struct

Struct

Fino ad ora abbiamo utilizzato tipi semplici (char, int, float, ...), puntatori (char *, int *, float *, ...) o array di tipi semplici

Struct

Fino ad ora abbiamo utilizzato tipi semplici (char, int, float, ...), puntatori (char *, int *, float *, ...) o array di tipi semplici

Spesso è utile avere tipi complessi: tipi ottenuti combinando tipi semplici

Struct

Fino ad ora abbiamo utilizzato tipi semplici (char, int, float, ...),
puntatori (char *, int *, float *, ...) o array di tipi semplici

Spesso è utile avere tipi complessi: tipi ottenuti combinando tipi semplici

Esempi:

- coordinate di un punto nel piano (coordinate x e y);
- nodo di un albero (chiave e puntatori ai figli);
- elementi di una lista (chiave e puntatori al predecessore e successore).

Struct

Fino ad ora abbiamo utilizzato tipi semplici (char, int, float, ...),
puntatori (char *, int *, float *, ...) o array di tipi semplici

Spesso è utile avere tipi complessi: tipi ottenuti combinando tipi semplici

Esempi:
- coordinate di un punto nel piano (coordinate x e y);
- nodo di un albero (chiave e puntatori ai figli);
- elementi di una lista (chiave e puntatori al predecessore e successore).

Le struct consentono la creazione di nuovi tipi.
Ad esempio, un punto è un tipo avente due campi: x e y.

Struct

Fino ad ora abbiamo utilizzato tipi semplici (char, int, float, ...),
puntatori (char *, int *, float *, ...) o array di tipi semplici

Spesso è utile avere tipi complessi: tipi ottenuti combinando tipi semplici

Esempi:
- coordinate di un punto nel piano (coordinate x e y);
- nodo di un albero (chiave e puntatori ai figli);
- elementi di una lista (chiave e puntatori al predecessore e successore).

Le struct consentono la creazione di nuovi tipi.
Ad esempio, un punto è un tipo avente due campi: x e y.

```
struct point {
    int x;
    int y;
};
```

Struct

Fino ad ora abbiamo utilizzato tipi semplici (char, int, float, ...),
puntatori (char *, int *, float *, ...) o array di tipi semplici

Spesso è utile avere tipi complessi: tipi ottenuti combinando tipi semplici

Esempi:
- coordinate di un punto nel piano (coordinate x e y);
- nodo di un albero (chiave e puntatori ai figli);
- elementi di una lista (chiave e puntatori al predecessore e successore).

Le struct consentono la creazione di nuovi tipi.
Ad esempio, un punto è un tipo avente due campi: x e y.

```
struct point {
    int x;
    int y;
};
```

NB. una struct può avere campi di tipi diversi, anche altre struct.

Struct

```
struct point {
    int x;
    int y;
};

int main() {
    struct point p;  //dichiarazione
    p.x = 10; //accesso ad un campo con "."
    p.y = 14;
    printf("Posizione x %d, y %d", p.x, p.y);
    return 0;
}
```

NB. una struct può avere campi di tipi diversi, anche altre struct.

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

```
struct point {
    int x;
    int y;
};

int main() {
    struct point *p; // dichiarazione
```

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

```
struct point {
    int x;
    int y;
};

int main() {
    struct point *p; // dichiarazione
    p = (struct point *) malloc(sizeof(struct point)); // allocazione
```

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

```
struct point {
    int x;
    int y;
};

int main() {
    struct point *p; // dichiarazione
    p = (struct point *) malloc(sizeof(struct point)); // allocazione
    (*p).x = 10; // assegnamento CORRETTO
```

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

```
struct point {
    int x;
    int y;
};

int main() {
    struct point *p; // dichiarazione
    p = (struct point *) malloc(sizeof(struct point)); // allocazione
    (*p).x = 10; // assegnamento CORRETTO
    p->x = 10;   // equivalente al precedente
```

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

```
struct point {
    int x;
    int y;
};

int main() {
    struct point *p; // dichiarazione
    p = (struct point *) malloc(sizeof(struct point)); // allocazione
    (*p).x = 10; // assegnamento CORRETTO
    p->x = 10;   // equivalente al precedente
    p[0].x = 10;  // equivalente ai precedenti
```

Struct: puntatori e allocazione

Un puntatore a struct è dichiarato come per le altre variabili.

```
struct point {
    int x;
    int y;
};

int main() {
    struct point *p; // dichiarazione
    p = (struct point *) malloc(sizeof(struct point)); // allocazione
    (*p).x = 10; // assegnamento CORRETTO
    p->x = 10;   // equivalente al precedente
    p[0].x = 10;  // equivalente ai precedenti
    *p.x = 10;   // Errato! . ha la precedenza su *
```

Struct e funzioni (1)

Una funzione può restituire una struct, esattamente come con gli altri tipi.

```
struct point {
    int x;
    int y;
};

struct point createPoint(int x, int y) {
    struct point p;
    p.x = x;
    p.y = y;
    return p;
}
```

Struct e funzioni (2)

Una struct può essere passata (per valore) come parametro ad una funzione

```
struct point {
    int x;
    int y;
};

struct point sumPoint(struct point p1, struct point p2) {
    p1.x = p1.x + p2.x; // si opera direttamente su p1
    p1.y = p1.y + p2.y; // ma il chiamante vede le modifiche?
    return p1;
}
```

Struct e funzioni (3)

Struct e funzioni (3)

In molti casi è necessario o preferibile passare il puntatore ad una struct anziché la struct stessa:

Struct e funzioni (3)

In molti casi è necessario o preferibile passare il puntatore ad una struct anziché la struct stessa:

- quando vogliamo modificare i campi della struct (necessario);

Struct e funzioni (3)

In molti casi è necessario o preferibile passare il puntatore ad una struct anziché la struct stessa:

- quando vogliamo modificare i campi della struct (necessario);
- quando la dimensione della struct è grande e la copia dei suoi campi sarebbe troppo costosa (preferibile).

Struct e funzioni (3)

In molti casi è necessario o preferibile passare il puntatore ad una struct anziché la struct stessa:

- quando vogliamo modificare i campi della struct (necessario);
- quando la dimensione della struct è grande e la copia dei suoi campi sarebbe troppo costosa (preferibile).

```
void swapPoint(struct point* p){
    int tmp=p->x;
    p->x=p->y;
    p->y=tmp;
}

int main(){
    struct point* p = (struct point *) malloc(sizeof(struct_point));
    p->x = 5;
    p->y = 2;
    swapPoint(p);
    printf("x %d, y %d", p->x, p->y);
}
```

Typedef

Typedef

In C è possibile assegnare nuovi nomi ai tipi di dato.

```
typedef tipo nuovo_nome;
```

Non viene creato un nuovo tipo ma solo un alias.

Typedef

In C è possibile assegnare nuovi nomi ai tipi di dato.

```
typedef tipo nuovo_nome;
```

Non viene creato un nuovo tipo ma solo un alias.

Esempi

```
typedef char* string;
```

Ora string può essere usato al posto di char *

Typedef

In C è possibile assegnare nuovi nomi ai tipi di dato.

```
typedef tipo nuovo_nome;
```

Non viene creato un nuovo tipo ma solo un alias.

Esempi

```
typedef char* string;
```

Ora string può essere usato al posto di char *

```
typedef struct _punto {
    int x;
    int y;
} punto;
```

Ora punto può essere usato al posto di struct _punto

Struct ricorsive

Una struct può avere come campo un puntatore alla struct stessa.

Esempio

```
typedef struct _node {
    int key;
    struct _node *next;
} node;
```

Qsort

La libreria standard (includere stdlib.h) ha una implementazione del QuickSort pensata per ordinare qualunque tipo di dato.

Qsort

La libreria standard (includere stdlib.h) ha una implementazione del QuickSort pensata per ordinare qualunque tipo di dato.

```
void qsort( void *buf, size_t num, size_t size,
                                int (*compare) (const void *, const void *) );
```

Qsort

La libreria standard (includere stdlib.h) ha una implementazione del QuickSort pensata per ordinare qualunque tipo di dato.

```
void qsort( void *buf, size_t num, size_t size,
                              int (*compare) (const void *, const void *) );
```

void *buf: puntatore al primo elemento dell'array da ordinare;

Qsort

La libreria standard (includere stdlib.h) ha una implementazione del QuickSort pensata per ordinare qualunque tipo di dato.

```
void qsort( void *buf, size_t num, size_t size,
                              int (*compare) (const void *, const void *) );
```

void *buf: puntatore al primo elemento dell'array da ordinare;

size_t num: numero di elementi da ordinare;

Qsort

La libreria standard (includere stdlib.h) ha una implementazione del QuickSort pensata per ordinare qualunque tipo di dato.

```
void qsort( void *buf, size_t num, size_t size,
                              int (*compare) (const void *, const void *) );
```

void *buf: puntatore al primo elemento dell'array da ordinare;

size_t num: numero di elementi da ordinare;

size_t size: la dimensione in byte di ciascun elemento;

Qsort

La libreria standard (includere stdlib.h) ha una implementazione del QuickSort pensata per ordinare qualunque tipo di dato.

```
void qsort( void *buf, size_t num, size_t size,
                              int (*compare) (const void *, const void *) );
```

void *buf: puntatore al primo elemento dell'array da ordinare;

size_t num: numero di elementi da ordinare;

size_t size: la dimensione in byte di ciascun elemento;

int (*compare) (const void *, const *void):
(il puntatore ad) una funzione che specifichi come confrontare due elementi dell'array. Attraverso questa funzione si fornisce il criterio per l'ordinamento.

Funzione di confronto

Il qsort necessita di una funzione per confrontare gli elementi.

```
int compare (const void *, const void *);
```

che dati i puntatori a due elementi restituisca

Funzione di confronto

Il qsort necessita di una funzione per confrontare gli elementi.

```
int compare (const void *, const void *);
```

che dati i puntatori a due elementi restituisca

< 0: se il primo elemento deve precedere il secondo nell'ordinamento;

Funzione di confronto

Il qsort necessita di una funzione per confrontare gli elementi.

```
int compare (const void *, const void *);
```

che dati i puntatori a due elementi restituisca

< 0: se il primo elemento deve precedere il secondo nell'ordinamento;

> 0: se il secondo elemento deve precedere il primo nell'ordinamento;

Funzione di confronto

Il qsort necessita di una funzione per confrontare gli elementi.

```
int compare (const void *, const void *);
```

che dati i puntatori a due elementi restituisca

< 0: se il primo elemento deve precedere il secondo nell'ordinamento;

> 0: se il secondo elemento deve precedere il primo nell'ordinamento;

$= 0$: altrimenti.

Funzione di confronto

Il qsort necessita di una funzione per confrontare gli elementi.

```
int compare (const void *, const void *);
```

che dati i puntatori a due elementi restituisca

< 0: se il primo elemento deve precedere il secondo nell'ordinamento;

> 0: se il secondo elemento deve precedere il primo nell'ordinamento;

$= 0$: altrimenti.

Questo tipo di funzione (con nome arbitrario) deve essere implementata a seconda del tipo da ordinare e dell'ordinamento desiderato.

Esempio: Interi

Ordinare un array di interi in modo non decrescente

Esempio: Interi

Ordinare un array di interi in modo non decrescente

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

Esempio: Interi

Ordinare un array di interi in modo non decrescente

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

int confrontaInt(const void *a, const void *b) {
    return ( *(int*)a - *(int*)b );
}
```

Esempio: Interi

Ordinare un array di interi in modo non decrescente

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

int confrontaInt(const void *a, const void *b) {
    return ( *(int*)a - *(int*)b );
}

int main(){
    int values[] = {10, 21, 1 , 7 , 24 , 9};
    int n = 6;
    qsort(values, n, sizeof(int), confrontaInt);
    ...
```

Esempio: Stringhe

Ordinare un array di stringhe in ordine lessicografico

Esempio: Stringhe

Ordinare un array di stringhe in ordine lessicografico

```
int compare(const void *a, const void *b){
    char **a1= (char **) a;
    char **b1= (char **) b;
    return strcmp(*a1,*b1);
}
```

Esempio: Stringhe

Ordinare un array di stringhe in ordine lessicografico

```
int compare(const void *a, const void *b){
    char **a1= (char **) a;
    char **b1= (char **) b;
    return strcmp(*a1,*b1);
}

int main(){
    int i;
    int n=10;
    char ** values = malloc(n*sizeof(char *));
    for(i=0; i < n; i++) {
        values[i] = malloc(101*sizeof(char));
        scanf("%s", values[i]);
    }
    qsort(values, n, sizeof(char *), compare);
    ...
```

Esercizio 1

Qsort su interi

Scrivere un programma che utilizzi la funzione `qsort` e ordini un vettore di interi (in input), in modo da ottenere il seguente effetto. L'array ordinato deve contenere prima tutti i numeri pari e, a seguire, i numeri dispari. Si assuma che il numero 0 sia pari. I numeri pari devono essere ordinati in modo *crescente* fra di loro. I numeri dispari devono essere ordinati in modo *decrescente* fra di loro.

Esercizio 2

Qsort su stringhe

Scrivere un programma che legga in input un array A di stringhe e che utilizzi la funzione `qsort` per ordinare in ordine alfabetico *decrescente* le stringhe in input. Assumere che la lunghezza massima di una stringa sia 100 caratteri.

Esercizio 3

Qsort e struct

Scrivere un programma che utilizzi la funzione `qsort` per ordinare un vettore di punti del piano cartesiano. Ciascun punto è formato da una coppia di coordinate (x, y).

I punti devono essere ordinati per ascissa crescente. A parità di ascissa, si ordini per ordinata decrescente.

Esercizio 4

Qsort su stringhe e struct

Scrivere un programma che utilizzi la funzione `qsort` per ordinare un array di stringhe. Le stringhe devono essere ordinate per lunghezza crescente. A parità di lunghezza, si utilizzi l'ordinamento lessicografico. Utilizzare una struct che memorizzi una stringa e la sua lunghezza per evitare di calcolare quest'ultima ad ogni confronto.

Puzzle

Elemento maggioritario

Un array A contiene n interi. Uno di essi è detto *elemento maggioritario* se occorre in A *almeno* $\lfloor \frac{n}{2} \rfloor + 1$ volte. Si vuole un algoritmo che identifichi l'elemento maggioritario, se presente, in tempo $O(n)$ utilizzando $O(1)$ spazio aggiuntivo. L'algoritmo deve stampare N se non è presente alcun elemento maggioritario.

Caso particolare. Si assuma che tutti gli elementi, ad esclusione eventualmente del maggioritario, occorrano una sola volta in A.

Caso generale. Gli elementi possono avere un numero arbitrario di occorrenze.

Input

```
5
1 10 22 11 2
```

Output

```
N
```

Input

```
5
22 10 22 11 22
```

Output

```
22
```

Puzzle

Due uova
Interview di Google

In un palazzo di 100 piani si vuole stabilire qual è il piano più alto dal quale è possibile lanciare un uovo senza che esso si rompa. Le uova sono considerate tutte aventi lo stesso grado di resistenza.

Si hanno a disposizione solamente due uova e si vuole individuare tale piano con il minor numero di lanci.

Soluzione banale: si provano tutti i piani sequenzialmente dal primo al novantanovesimo. Dopo ogni lancio si prosegue se e soltanto se l'uovo appena lanciato risulta ancora intatto. Al caso pessimo sono necessari 99 lanci.