



UNIVERSITÀ DEGLI STUDI  
DI MODENA E REGGIO EMILIA

# Dispense del corso di Fondamenti di Informatica 1

## Il linguaggio C - Puntatori (seconda parte)

Aggiornamento del 05/12/2019

## Ancora sul typedef

- Abbiamo visto che in C è possibile definire variabili con tipi di dato composti di diverse parti, ad esempio:

```
unsigned char *p;
```

- La variabile p è un puntatore a intero a 8 bit senza segno.
- Possiamo però definire un nuovo tipo «anni» che diventi un sinonimo di unsigned char:

```
typedef unsigned char anni;
```

- La definizione precedente diventa allora:

```
anni *p;
```

- Si possono creare tipi che includono qualsiasi modificatore o specifica, quindi anche un tipo puntatore:

```
typedef unsigned char *ptranni;
```

- La definizione qui sopra fa sì che ptranni sia il tipo «puntatore a unsigned char». Avremmo potuto anche scrivere:

```
typedef unsigned char anni;
```

```
typedef anni *ptranni;
```

# La standard library del C

- Il C, oltre a tutti i suoi operatori, mette a disposizione una collezione di funzioni, macro e definizioni di tipi che permettono di semplificare il lavoro del programmatore che non deve reimplementare da solo funzioni basilari come quelle matematiche, quelle per l'input/output, la manipolazione di stringhe, ecc.
- Tutto ciò che viene messo a disposizione dalla standard library del C è accessibile attraverso l'inclusione di opportuni file .h nei file del proprio programma:

```
#include <...>
```

- In questo caso utilizzeremo < e > in quanto i file da includere non si troveranno nella cartella corrente ma tra le librerie del compilatore.
- Al momento la libreria standard mette a disposizione 29 file .h
- Di seguito vedremo alcuni dei più importanti

## La libreria stdint.h

- Contiene le definizioni dei tipi interi a larghezza fissa, signed e unsigned e i rispettivi valori minimi e massimi.
- I tipi in essa contenuti esprimono chiaramente la dimensione in bit di una variabile e la presenza del segno, ad esempio:

`uintN_t a;` dichiara un intero senza segno grande esattamente N bit.  
`intN_t a;` dichiara un intero con segno grande esattamente N bit.

- Vengono definite anche le macro che vengono sostituite dal preprocessore con i valori massimi e minimi per ogni tipo:

`INTN_MIN` → il minimo valore rappresentabile con N bit con segno  
`UINTN_MAX` → il massimo valore rappresentabile con N bit senza segno

- I valori di N per cui sono stati definiti i tipi e le relative macro sono 8, 16, 32 e 64.

## La libreria `stdlib.h`

- La libreria `stdlib.h` contiene varie funzioni, macro e definizioni tra cui funzioni per la gestione della memoria, per la generazione di numeri casuali e alcune utilities. Per ora limitiamoci a vedere solo due degli elementi contenuti in questa libreria:
- `NULL`: è una macro che viene utilizzata per rappresentare il valore di un puntatore nullo, cioè un puntatore che non sta puntando a un indirizzo di memoria valido.
- È molto utile sapere con certezza quando l'indirizzo di un puntatore non sta puntando a un indirizzo valido perché ci permette di evitare errori che il programma potrebbe incontrare a runtime.
- Dereferenziare per sbaglio un puntatore `NULL` causerebbe infatti un crash del programma.
- Tipicamente `NULL` è uno 0.

# La libreria `stdlib.h`

- `stdlib.h` contiene anche la definizione di un tipo di dato particolare chiamato `size_t`
- `size_t` è un tipo di dato intero senza segno utilizzato per rappresentare le dimensioni di un oggetto nell'implementazione che si sta usando. Essendo «*implementation defined*» la sua dimensione in memoria può variare ma è di almeno 16 bit. È il tipo che viene comunemente utilizzato quando si deve dichiarare una variabile che contiene la dimensione di un oggetto.
- Con *implementation defined* si intende che lo standard specifica solamente «cosa» fa una funzione o un tipo, lasciando il compito di decidere «come» farlo a chi realizzerà il compilatore per una specifica architettura.
- In Visual Studio quando si compila a 32 bit (default) `size_t` è un unsigned int, quando si compila a 64 bit `size_t` è un unsigned long long.

# La libreria stdbool.h

- Il C dalla sua revisione C99 supporta il tipo fondamentale booleano `_Bool`, una variabile di questo tipo è in grado di memorizzare valori interi 0 e 1. In una variabile di tipo `_Bool` viene memorizzato il valore 0 quando le si assegna uno 0, altrimenti quando le si assegna qualsiasi altro valore nella variabile viene memorizzato un 1.
- La libreria `stdbool.h` definisce principalmente tre macro:
  - `bool`
  - `true`
  - `false`
- Esse vengono espansive rispettivamente a:
  - `_Bool`
  - `1`
  - `0`
- Con esse è possibile dichiarare e utilizzare variabili di tipo booleano come se fosse un tipo fondamentale, ad esempio scrivendo:

```
bool var;
```

```
bool var = true;
```

# La libreria ctype.h

- La libreria ctype.h mette a disposizione funzioni utili per determinare cosa è contenuto in un singolo char (anche se i parametri sono tutti int).
- Alcuni esempi di funzioni sono:
  - `int islower(int ch)`: verifica se il carattere ch è minuscolo
  - `int isupper(int ch)`: verifica se il carattere ch è maiuscolo
  - `int isalpha(int ch)`: verifica se il carattere ch è un carattere alfabetico
  - `int isprint(int ch)`: verifica se il carattere ch è stampabile
  - `int isdigit(int ch)`: verifica se il carattere ch è una cifra
- Tutte queste funzioni ritornano un valore diverso da 0 quando la condizione è verificata, 0 altrimenti.
- In ctype.h sono presenti anche due funzioni utili per convertire un carattere da minuscolo a maiuscolo o viceversa:
  - `int tolower(int ch)`: converte il carattere ch in minuscolo
  - `int toupper(int ch)`: converte il carattere ch in maiuscolo
- Entrambe queste funzioni ritornano il risultato della conversione o lo stesso carattere se non è stato possibile convertirlo.



# L'operatore sizeof

- L'operatore `sizeof` ritorna la dimensione (in char) di un tipo o di un'espressione in una variabile di tipo `size_t`:

- `sizeof( type )`
- `sizeof expression`

- Ad esempio:

```
size_t res;
float a = 3.2f, b = 2.1f;
char c;
res = sizeof a;           // res = 4
res = sizeof c;           // res = 1
res = sizeof 'c';         // res = 4
res = sizeof(255 + 10);   // res = 4
res = sizeof(a * b);      // res = 4
res = sizeof(3.4 + 21);   // res = 8
res = sizeof(&res);       // res = 4 (compilando a 32 bit)
res = sizeof(char);       // res = 1
res = sizeof(short);      // res = 2
res = sizeof(int);        // res = 4
res = sizeof(double);     // res = 8
```

- **ATTENZIONE:** l'operatore `sizeof` ha precedenza più alta rispetto agli operatori aritmetici, quindi:

```
res = sizeof 2u + 3u;     // res = 7!!
```

## Accesso in memoria con i puntatori

- In ADE8 abbiamo introdotto i puntatori per consentire ai programmi di accedere a più dati consecutivi in memoria.
- Per ora invece i puntatori sono stati usati unicamente per l'accesso indiretto a singole variabili, con lo scopo di modificare da una funzione una variabile esterna.
- Quello che dobbiamo fare è ora utilizzare operazioni aritmetiche sui puntatori (incremento e decremento) e vedere come definire più variabili in memoria a indirizzi consecutivi.
- A differenza di quanto visto in ADE8 però le variabili non sono tutte grandi uguali e quindi «andare all'elemento successivo» non è necessariamente «incrementare di 1 il puntatore».

# Aritmetica dei puntatori

- Consideriamo la variabile:

```
int a = 3;
```

```
int *pa = &a;
```

- In questo caso `pa` è un puntatore a una variabile di tipo `int`.  
Supponiamo che il suo valore sia `0x0092f7e0`. Proviamo ad eseguire:

```
pa = pa + 2;
```

- Quale sarà il valore di `pa` dopo questo statement?
- La risposta è `0x0092f7e8`. Il valore è aumentato di 8! Perché?
- In C esiste l'**aritmetica dei puntatori**. Gli operatori «+» e «-», quando applicati tra un puntatore e un'espressione di tipo intero hanno un comportamento diverso da quello che hanno normalmente in espressioni di tipo aritmetico. Nel comando precedente il valore da assegnare a `pa` viene valutato in realtà come:

`<valore di pa> + 2 * <dimensione di int>`

# Aritmetica dei puntatori

- In generale, data la variabile:

`<tipo> *pa;`

- Sono valide le seguenti espressioni:

`pa + <espressione intera>`

`pa - <espressione intera>`

- Le quali, rispettivamente, aggiungono o sottraggono implicitamente al valore del puntatore il valore di:

`<dimensione di tipo> * <espressione intera>`

- Il tipo di queste espressioni sarà sempre `<tipo>*`, quindi potranno essere assegnate solamente a variabili di tipo puntatore a `tipo`!

## Aritmetica dei puntatori

- Esiste anche un altro tipo di espressione valida, la sottrazione tra due puntatori dello stesso tipo:

```
<tipo> *pa, *pb;
```

```
ptrdiff_t n = pa - pb;
```

- Il risultato è memorizzato in una variabile di tipo `ptrdiff_t`, un tipo intero con segno definito nella standard library del C che serve proprio a rappresentare il risultato di differenze tra puntatori.
- In questo caso il valore memorizzato in `n` sarà:

$$\frac{(\text{valore di } pa - \text{valore di } pb)}{\text{dimensione di tipo}}$$

- Questo serve per ottenere la «distanza» tra due puntatori in numero di elementi e non in char (quindi di solito byte).

# Chiedere memoria alla libreria standard

- Come facciamo a definire più variabili dello stesso tipo?
- Una possibile soluzione è quella di chiedere alla libreria standard di fornirci lo spazio necessario: questo è chiamato *allocazione di memoria*.
- A runtime (cioè quando il programma viene eseguito) è possibile chiedere alla libreria standard di allocare una certa quantità di memoria utilizzando la funzione **malloc()**:

```
extern void *malloc(size_t size);
```

- Il parametro `size` della `malloc()` deve contenere il numero di **byte (char)** che si vogliono allocare.
- La funzione ritorna un puntatore al primo byte della zona di memoria che è stata appena riservata, o `NULL` in caso di errore.
- La memoria appena allocata **NON** è inizializzata e quindi conterrà valori a caso!

# Puntatori a void

```
extern void *malloc(size_t size);
```

- La malloc ritorna un *puntatore a void*... che cosa significa?
- Un puntatore a `void` indica un puntatore senza un tipo associato, ovvero un indirizzo a cui manca l'informazione sul tipo di dato a cui si sta puntando.
- Questo è un puntatore speciale, nel senso che non è direttamente utilizzabile:
  - non si può dereferenziare (quanti byte è grande il dato puntato?)
  - l'aritmetica dei puntatori non è ammessa su di esso (di quanto incremento?)
- Come posso fare quindi a utilizzarlo per accedere alla memoria appena allocata?
- Prima di poter essere utilizzato **deve** essere convertito in un puntatore a un tipo specifico.
- In C, un puntatore a `void` può essere assegnato a qualsiasi puntatore e, viceversa, qualsiasi puntatore può essere assegnato ad una variabile di tipo `void*`.

## Puntatori a void

- La conversione avviene implicitamente quando un puntatore a void viene assegnato a un puntatore a un altro tipo, ad esempio:

```
void *vp = malloc(12);
```

```
int *ip = vp;
```

- Si potrà quindi anche scrivere direttamente:

```
int *ip = malloc(12);
```

- La conversione può anche essere resa esplicita attraverso un cast al tipo che dobbiamo ottenere:

```
int *ip = (int*)malloc(12);
```

anche se questo **non è mai necessario**. Non scrivetelo.

- La malloc() utilizza `void*` come tipo di ritorno perché deve essere in grado di allocare memoria per qualsiasi tipo di dato.
- L'unica cosa che deve sapere per allocare memoria è il numero di byte necessari per contenere un certo tipo di dato.



# Allocare memoria

- Abbiamo visto che possiamo allocare spazio per *un po' di int* in questo modo:

```
int *ip = malloc(12);
```

- Quanti int possiamo mettere in 12 byte?
- Quanti byte ci servono per contenere 379 int?
- È meglio lasciare che questi calcoli li faccia il compilatore per noi, sfruttando l'operatore `sizeof`:

```
int *ip = malloc(3 * sizeof(int));
```

- In questo modo evitiamo lo sforzo di fare i conti e non è necessario specificare la dimensione che ha un int.
- È anche possibile, ma non sempre chiaro, scrivere:

```
int *ip = malloc(3 * sizeof *ip);
```

- In questo modo il risultato è analogo e se cambiamo il tipo di `ip`, non dobbiamo cambiare il tipo all'interno della `malloc`. Se però non vi è chiaro, evitatelo!

# Liberare la memoria

- La memoria che viene allocata in questo modo deve essere liberata quando non serve più. Questo viene fatto chiamando la funzione **free()**:  
`void free(void* ptr);`
- La funzione `free` comunica alla libreria standard che la memoria puntata da `ptr` è da deallocare. Se `ptr` è `NULL`, la funzione `free` non fa nulla.
- Bisogna fare molta attenzione a non utilizzare più i puntatori la cui memoria puntata è stata liberata con una `free`, l'accesso alla memoria puntata da un puntatore dopo che esso è stato passato alla `free` genera un *undefined behavior*.
- Un puntatore che non si riferisce più a un indirizzo valido è detto *dangling pointer*, in italiano «puntatore penzolante», ma nessuno lo chiama così.
- Un *undefined behavior* viene generato anche in altri due casi:
  - Quando alla `free` viene passato un puntatore che è già stato liberato in precedenza (*double free*)
  - Quando alla `free` viene passato un puntatore che non si riferisce a un'area di memoria allocata dinamicamente in precedenza.

# Liberare la memoria

- Quando si utilizza memoria allocata dinamicamente è molto importante ricordarsi di liberarla quando non è più necessaria, altrimenti si può incorrere in un *memory leak*.

- Consideriamo la seguente funzione:

```
int func(void)
{
    int *p = malloc(1000 * sizeof(int));
    // operazioni sulle celle puntate da p
    return *p; // ritorno il primo int puntato da p
}
```

- In questo caso i 4000 byte allocati non vengono liberati e quando la funzione ritorna al chiamante la memoria che conteneva il puntatore p viene liberata e con essa anche ogni possibilità di poter puntare nuovamente a quei 4000 byte. Ogni riferimento ad essi è perso per sempre. Quindi non posso nemmeno farne la `free()`.
- Se immaginiamo che la funzione `func` venga chiamata 100.000 volte durante l'esecuzione del programma finiremmo per avere 400.000.000 byte (circa 400MB) allocati per il nostro programma senza avere più la possibilità di liberarli.

# Accesso alle variabili appena allocate

- Una volta che è stata chiesta alla libreria standard della memoria, possiamo accedere ad essa usando i puntatori e la loro aritmetica.
- Supponiamo di avere allocato memoria per contenere 3 interi.

```
int *ip = malloc(3 * sizeof(int));
```

- Dato che le variabili si trovano a indirizzi consecutivi possiamo usare il puntatore `ip` per ricavare dei puntatori a tutte le altre variabili:

```
int *p1 = ip;           → punterà al primo intero.
```

```
int *p2 = ip + 1;       → punterà al secondo
```

```
int *p3 = ip + 2;       → punterà al terzo.
```

- Sfruttando poi l'operatore `*` per dereferenziare i puntatori appena ricavati potremo accedere ai valori puntati, ad esempio assegnando ad essi dei valori:

```
*p3 = 12;
```

- Per rendere tutto più compatto potremmo scrivere direttamente:

```
*(ip + 2) = 12;
```

# Esempio

- Il fatto che tutte le variabili si trovino a indirizzi di memoria consecutivi rende molto facile l'utilizzo dei cicli per l'accesso ad esse.
- Ad esempio, dato che la memoria allocata attraverso la funzione `malloc()` non è inizializzata, possiamo scrivere il codice per inizializzare un numero arbitrario di variabili di tipo `int` appena allocate:

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

```
int main(void)  
{
```

```
    size_t n = 10;
```

```
    int *p = malloc(n * sizeof(int));
```

```
    for (size_t i = 0; i < n; ++i) {  
        *(p + i) = 0;  
    }
```

```
    free(p);
```

```
    return 0;
```

```
}
```

Stabilisco il numero di variabili da allocare.

Alloco la memoria per `n` int usando la `malloc()`.

Con un ciclo `for` assegno a ogni variabile `int` il valore 0 usando l'operatore `*` e l'aritmetica dei puntatori.

Libero la memoria puntata da `p` quando non ne ho più bisogno passando alla `free()` il puntatore alla memoria da deallocare.

## L'operatore [ ]

- Scrivere:  $\ast (\underline{p} + i)$  ogni volta che si vuole accedere all' $i$ -esimo elemento della zona di memoria puntata da  $\underline{p}$  può risultare prolisso, per questo in C è stato introdotto un metodo più veloce per ottenere lo stesso risultato, ossia l'operatore [ ].
- La sintassi per utilizzare l'operatore [ ] è la seguente:

$\langle \text{exp1} \rangle \text{ [ } \langle \text{exp2} \rangle \text{ ]}$

- Per definizione usare dell'operatore [ ] è equivalente a scrivere:

$\ast (\text{exp1} + \text{exp2})$

- Dove  $\text{exp1}$  e  $\text{exp2}$  dovranno sempre essere due espressioni, una di un tipo intero e l'altra di tipo puntatore a qualcosa.
- Non potranno mai essere entrambe espressioni di tipo intero o di tipo puntatore.

## L'operatore [ ]

- Ad esempio, supponendo che  $p$  sia un puntatore valido, avremo che le due definizioni:

```
int a = p[4];
```

e:

```
int a = *(p + 4);
```

produrranno lo stesso risultato.

- Ma riguardando meglio la sintassi dell'operatore [ ] si deduce che è valido anche scrivere:

```
int a = 4[p];
```

anche in questo caso ottenendo gli stessi risultati.

- Anche se valido, questo modo di scrivere è estremamente poco chiaro!
- Durante il corso scriveremo **sempre** prima il puntatore poi, all'interno delle parentesi quadre, l'espressione intera.

# L'operatore [ ]

- Rivediamo ora l'esempio visto poco fa per azzerare tutte le variabili di tipo `int` appena allocate:

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

```
int main(void)
```

```
{  
    size_t n = 10;  
    int *p = malloc(n * sizeof(int));  
  
    for (size_t i = 0; i < n; ++i) {  
        p[i] = 0;  
    }  
  
    free(p);  
  
    return 0;  
}
```

Il codice è rimasto sostanzialmente invariato. L'utilizzo dell'operatore [ ] lo ha reso però più leggibile e chiaro.



## Funzioni alternative per l'allocazione

- Esistono anche altre due funzioni che svolgono un lavoro simile a quello della `malloc()` con alcune piccole differenze, sono le funzioni **`calloc()`** e **`realloc()`**.
- La funzione `calloc()` funziona esattamente come la `malloc()` con la differenza che la memoria appena allocata viene anche azzerata, la sua dichiarazione è:

```
void* calloc(size_t num, size_t size);
```

- I parametri sono diversi da quelli della `malloc()` ma hanno lo stesso significato, il parametro `num` indica il numero di oggetti da allocare mentre il parametro `size` indica la dimensione di un singolo oggetto.
- La memoria totale allocata sarà quindi di `num * size` bytes, e anche in questo caso il puntatore ritornato punterà al primo byte allocato.

# Esempio con calloc()

- Rivediamo ancora l'esempio di prima:

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

```
int main(void)
```

```
{
```

```
    size_t n = 10;
```

```
    int *p = calloc(n, sizeof(int));
```


```
    // ...
```

```
    free(p);
```

```
    return 0;
```

```
}
```

L'introduzione della calloc() ha permesso di evitare un ciclo esplicito per l'azzeramento.



# Cambiare la dimensione della memoria allocata

- Supponiamo di voler creare un vettore di numeri e inizialmente di aver bisogno solo del numero 27:

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

```
int main(void)
```

```
{
```

```
    int *v;
```

```
    size_t n = 1;
```

```
    v = malloc(n * sizeof(int));
```

```
    v[0] = 27;
```

```
    // ... Qui vogliamo aggiungere un elemento al vettore
```

```
    free(v);
```

```
    return 0;
```

```
}
```

- Adesso però come facciamo ad aggiungere il numero 137 per esempio?

# Cambiare la dimensione della memoria allocata

- Serve effettuare una serie di operazioni non proprio banali:
  1. creare un nuovo spazio di memoria per contenere gli elementi già presenti nel vettore più quello che vogliamo aggiungere

```
int *tmp = malloc((n + 1) * sizeof(int)); // 1.
```

2. copiare tutti gli elementi vecchi nella nuova zona di memoria

```
for (size_t i = 0; i < n; ++i) // 2.  
    tmp[i] = v[i];
```

3. aggiungere alla fine il nuovo elemento

```
tmp[n] = 137; // 3.
```

4. liberare la memoria allocata precedentemente

```
free(v); // 4.
```

5. far puntare il puntatore alla nuova zona di memoria

```
v = tmp; // 5.
```

6. aumentare la dimensione del vettore

```
++n; // 6.
```

# Cambiare la dimensione della memoria allocata

- Il risultato diventa quindi:

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

```
int main(void)
```

```
{
```

```
    int *v;
```

```
    size_t n = 1;
```

```
    v = malloc(n * sizeof(int));
```

```
    v[0] = 27;
```

```
    int *tmp = malloc((n + 1) * sizeof(int)); // 1.
```

```
    for (size_t i = 0; i < n; ++i) // 2.
```

```
        tmp[i] = v[i];
```

```
    tmp[n] = 137; // 3.
```

```
    free(v); // 4.
```

```
    v = tmp; // 5.
```

```
    ++n; // 6.
```

```
    free(v);
```

```
    return 0;
```

```
}
```

# Cambiare la dimensione della memoria allocata

- Quello che abbiamo fatto, andrebbe certamente inserito in una funzione, vista la complessità della cosa.
- Non esiste una soluzione alternativa? Per fortuna sì. Esiste la funzione:

```
void *realloc(void *ptr, size_t new_size);
```

- La `realloc()` ha due parametri, `ptr` è un puntatore alla zona di memoria che deve essere ridimensionata e `new_size` è la nuova dimensione che le si vuole dare.
- Se la zona di memoria viene ingrandita, la nuova parte non è inizializzata e contiene valori a caso.
- Se la zona viene ridotta, la parte che finisce oltre la nuova dimensione viene deallocata.
- Se `ptr` è `NULL` allora la `realloc()` si comporta esattamente come la `malloc()`.

# Cambiare la dimensione della memoria allocata

- Attenzione che la funzione ritorna un **nuovo puntatore**. Quindi come la si **deve** usare?
- Il puntatore ritornato dalla `realloc()` **deve sempre essere assegnato** ad una variabile, tipicamente quella di prima:

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

```
int main(void)
```

```
{
```

```
    int *v;
```

```
    size_t n = 1;
```

```
    v = malloc(n * sizeof(int));
```

```
    v[0] = 27;
```

```
    v = realloc(v, (n + 1) * sizeof(int)); // 1.2.4.5.
```

```
    v[n] = 137; // 3.
```

```
    ++n; // 6.
```

```
    free(v);
```

```
    return 0;
```

```
}
```

# Altri esempi

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

```
int main(void) {
```

```
    float *p = malloc(5 * sizeof(float));
```

```
    int *q = calloc(3, sizeof(int));
```

```
    *(q+1) = 10;
```

```
    *(p+4) = 12.4f;
```

```
    p = realloc(p, sizeof(float)* 10);
```

```
    *(p+9) = 13.2f;
```

```
    free(p);
```

```
    free(q);
```

```
    return 0;
```

```
}
```

Alloco memoria per 5 float (20 byte)  
non inizializzati

alloco memoria per 3 int inizializzati  
a zero (12 byte)

Assegno al secondo intero puntato  
da q il valore 10

Assegno al quinto float puntato da p  
il valore 12.4

Ridimensiono la zona puntata da p  
per contenere 10 float (40 byte)

Assegno al decimo float puntato da  
p il valore 13.2

dealloco la memoria puntata da p e  
la memoria puntata da q



# Altri esempi

```
#include <stdlib.h>

int main(void) {
    float *p = malloc(5 * sizeof(float));
    int *q = calloc(3, sizeof(int));

    q[1] = 10;
    p[4] = 12.4f;

    p = realloc(p, sizeof(float)* 10);

    p[9] = 13.2f;

    free(p);
    free(q);

    return 0;
}
```