A+

Ces deux variables sont identiques : une seule suffit.

```c
#include <stdio.h>
extern char **environ;

int nvar(){
   int cpt = 0;
   int i = 0;
   while(*(environ + i) != (char*)0){
       i++;
       cpt++;
   }
   return cpt;
}

int main(void){
    int res = nvar();
    printf("%d\n", res);
    return 0;
}
```

```c
#include <stdio.h>

extern char **environ;

int nvar(){
   int cpt = 0;
   int i = 0;
   while(*(environ + i) != (char*)0){
       i++;
       cpt++;
   }
   return cpt;
}

void printenv(int argc, char *argv[]){
    if(argc > 1){
        int nbVar = nvar();
        for(int i = 1; i < argc; i++){
            for(int j = 0; j < nbVar; environ++, j++){
                char * ptr = *environ;
                int k = 0;
                while(*(ptr + k) != '=' && argv[i][k] != '\0' &&
                    *(ptr + k) == argv[i][k]){
                    k++;
                }

                if(*(ptr + k) == '='){
                    printf("%s", (ptr + k + 1));
                    putchar('\n');
                }

            }
            environ = environ - nbVar;
        }
    }
    else{
        for (environ; *environ != NULL; ++environ){
            printf ("%s\n", *environ);
        }
    }
}

int main(int argc, char *argv[]){
    printenv(argc, argv);
    return 0;
}
```

Utiliser plutôt char ** argv car c'est un pointeur, pas un tableau.

inutile d'utiliser j si vous itérez sur le pointeur. Un itérateur suffit.

&& argv[i][k] == '\0' → On n'a peut-être pas parcouru toute la chaîne.

Style : utilisez plutôt un pointeur local que vous initialisez à environ en début de boucle.

environ est une variable globale : évitez de la modifier pour parcourir le tableau. Utilisez plutôt un pointeur local pour itérer.

✓

Apr 14, 21 21:30 **printenv.c** Page 1/1

```c
#include <stdio.h>
extern char **environ;

void printenv(){
    for (environ; *environ != NULL; ++environ)
        printf ("%s\n", *environ);
}

int main(void){
    printenv();
    return 0;
}
```

1 ✓

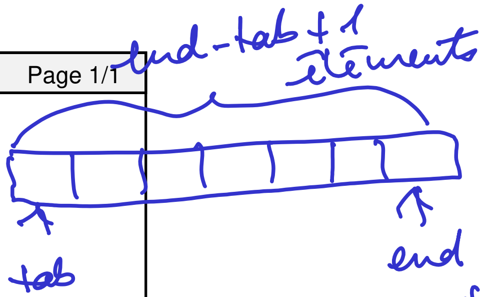
```
#include <stdio.h>

float * search_interval(float v, float *tab, float *end){
    float * res = NULL;
    int size = end - tab;
    int middle = (size/2);
    if(size < 1){
        res = NULL;
    }
    else if(tab[middle] == v){
        res = &tab[middle];
    }
    else if(tab[middle] > v){
        res = search_interval(v, tab, (tab+ middle));
    }
    else if(tab[middle] < v){
        res = search_interval(v, (tab + middle + 1), end);
    }
    return res;
}

int main(void){

    float tab[10] = {1, 15.53, 20.89, 27, 38, 42.2 ,63 ,64.9, 78,80};
    float * p = &tab[0];
    float * end = &tab[9];
    float * res = NULL;
    float x;
    scanf("%f", &x);
    res = search_interval(x, p, end);
    if(res == NULL){
        printf("Po lo\n");
    }
    else {
        printf("l’Ã©lement %.2f est lÃ , merci Ã  lui\n", *res);
    }
    return 0;
}
```

+1 →



|✓

|✓

-1 → *(tab+middle) a déjà été testé.

|✓

&tab[0]; → ou simplement : tab

tab

↳ Vérifiez qu'il n'y a ~~pas~~ eu d'erreur d'entrée en vérifiant la valeur de retour de scanf().

```c
#include <stdio.h>

#define SIZE 10

typedef int(func_t)(int);

int filter_int(func_t *f, const int *from, int *to, unsigned int size){
    int i, j;
    for (j = i = 0; i < size; i++){
        if (f(from[i])){
            to[j++] = from[i];
        }
    }
    return j;
}
```

✓

```c
int even(int nb){
    return (nb % 2 == 0) ? 1 : 0;
}
```

inutile : return (nb%2 ==0);

```c
int main(){
    int from[SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int to[SIZE];

    int x = filter_int(&even, from, to, SIZE);

    for (int i = 0; i < x; i++){
        printf("%d\n", to[i]);
    }

    return 0;
}
```

✓

```c
#include <stdlib.h>             /* pour random() */
#include <string.h>             /* pour memcmp() */
#include <assert.h>             /* pour assert() */
#include <stdio.h>

#define SIZE  1021


void memswap(void *to, const void *from, unsigned int size){
    char *cfrom = (char *)from;
    char *cto = (char *)to;
    char interm;
    char t[size], f[size];

    for(int i = 0; i < size; i ++){
        t[i] = cto[i];
        f[i] = cfrom[i];
    }

    for (int i = 0; i < size; i++){
        interm = t[i];
        cto[i] = f[i];
        cfrom[i] = ~~interm~~;
                   t[i]
    }
}
```

| un tableau intermédiaire suffit .

← variable inutile car t[i] n'est pas modifié .

```c
void test_memswap(){

    char tc_orig[SIZE], tc_dest[SIZE], c_orig[SIZE], c_dest[SIZE];
    long int ti_orig[SIZE], ti_dest[SIZE], i_orig[SIZE], i_dest[SIZE];
    int i;


    /* initialisation */
    for (i = 0; i < SIZE; i++){
        tc_orig[i] = random() % 256;
        c_orig[i] = tc_orig[i];
    }
    for (i = 0; i < SIZE; i++){
        tc_dest[i] = random() % 256;
        c_dest[i] = tc_dest[i];
    }
    for (i = 0; i < SIZE; i++){
        ti_orig[i] = random();
        i_orig[i] = ti_orig[i];
    }
    for (i = 0; i < SIZE; i++){
        ti_dest[i] = random();
        i_dest[i] = ti_dest[i];
    }


    /* swapie */
    memswap(tc_dest, tc_orig, SIZE);
    memswap(ti_dest, ti_orig, SIZE * sizeof(long int));

    /* vÃ©rification */
    assert(memcmp(tc_orig, c_dest, SIZE) == 0);
    assert(memcmp(tc_dest, c_orig, SIZE) == 0);
    assert(memcmp(ti_orig, i_dest, SIZE * sizeof(long int)) == 0);
    assert(memcmp(ti_dest, i_orig, SIZE * sizeof(long int)) == 0);
}


int main(){
    test_memswap();
    return 0;
}
```

✓

```c
#include <stdlib.h>             /* pour random() */
#include <string.h>             /* pour memcmp() */
#include <assert.h>             /* pour assert() */

#define SIZE 1021


void mmemcpy(void *to, const void *from, unsigned int size){
    char *cfrom = (char *)from;
    char *cto = (char *)to;

    for (int i = 0; i < size; i++)
        cto[i] = cfrom[i];
}

void test_mmemcpy()
{
    char tc_orig[SIZE], tc_dest[SIZE];
    long int ti_orig[SIZE], ti_dest[SIZE];
    int i;

    /* initialisation */
    for(i=0 ; i<SIZE ; i++) {
        tc_orig[i] = random() % 256;
        tc_dest[i] = random() % 256;
        ti_orig[i] = random();
        ti_dest[i] = random();
    }

    /* copie */
    mmemcpy(tc_dest, tc_orig, SIZE);
    mmemcpy(ti_dest, ti_orig, SIZE * sizeof(long int));

    /* vÃ©rification */
    assert(memcmp(tc_orig, tc_dest, SIZE) == 0);
    assert(memcmp(ti_orig, ti_dest, SIZE * sizeof(long int)) == 0);
}


int main(){
    test_mmemcpy();
    return 0;
}
```