



Langage C – Cours 3

Lélia Blin

lelia.blin@irif.fr

2023 - 2024

- Rappel
- Assert
- Pointeurs
- Arithmétique de pointeurs

Retour sur les types

```
//parcours.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void affiche_inv(int[],size_t);

int main(){
    int t[4]={0,1,2,3};
    affiche_inv(t,4);
    return EXIT_SUCCESS;
}

void affiche_inv(int tab[],size_t taille){
    for(size_t i=taille-1;i>=0;--i){
        printf("Valeur indice [%zu] : %d\n",i,tab[i]);
    }
}
```

Quel est le problème avec ce code?

Quel est le problème

- Le problème avec ce code réside dans la boucle `for` de la fonction `affiche_inv`.
- La condition de la boucle est `i >= 0`, ce qui semble correct pour parcourir les indices du tableau de `taille - 1` à 0 inclus.
- Cependant, cela entraîne un problème lors de l'exécution du programme.
 - la condition `i >= 0` restera toujours vraie tant que `i` sera supérieur ou égal à zéro.
 - Cela signifie que la boucle continuera à s'exécuter même lorsque `i` atteindra la valeur zéro, ce qui provoquera un débordement de mémoire car `tab[i]` tentera d'accéder à un élément en dehors des limites du tableau.

```
\\code corriger -> parcouresbis.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

void affiche_inv(int[],size_t);

int main(){
    int t[4]={0,1,2,3};
    affiche_inv(t,4);
    return EXIT_SUCCESS;
}

void affiche_inv(int tab[],size_t taille){
    for(size_t i=0;i<taille;++i){
        printf("Valeur indice [%zu] : %d\n",i,tab[taille-1-i]);
    }
}
```

assert

- On peut utiliser une instruction C qui fait arrêter le programme si une condition n'est pas vérifiée
- Syntaxe (dans <assert.h>) `assert(condition) ;`
- Si la condition est vérifiée, rien ne se passe
- Sinon, le programme s'arrête en affichant un message d'erreur à l'utilisateur
- Cette fonctionnalité est utile pour débbuger votre code

assert

- Par exemple, pour garantir qu'une fonction n'est jamais utilisé avec des mauvaises arguments
- On peut « éteindre » cette fonctionnalité en utilisant (avec gcc) l'option
- `NDNDEBUG=1`
- `gcc -Wall -NDNDEBUG=1 -o fichier fichier.c`


```
//test-assert.c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>

unsigned moyenne(int[],size_t);

int main(){
    int t[4]={0,1,2,3};
    int t2[0]={};
    printf("Moyenne entiere de t %u\n",moyenne(t,4));
    printf("Moyenne entiere de t2 %u\n",moyenne(t2,0));
    return EXIT_SUCCESS;
}

unsigned moyenne(int tab[],size_t taille){
    assert(taille>0);
    unsigned s=0;
    for(size_t i=0;i<taille;++i){
        s+=tab[i];
    }
    return s/taille;
}
```



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

unsigned pgcd(unsigned a,unsigned b);
unsigned pgcd2(unsigned a,unsigned b);

int main(){
    unsigned x=1236;
    unsigned y=544;
    printf("PGCD de %u et %u : %u\n",x,y,pgcd(y,x));
    return EXIT_SUCCESS;
}

unsigned pgcd(unsigned a,unsigned b){
    if(a<=b){
        return pgcd2(a,b);
    }else{
        return pgcd2(b,a);
    }
}

unsigned pgcd2(unsigned a,unsigned b){ // Algorithme d'Euclide
    assert(a<=b);
    if(!a){// cas ou a vaut 0
        return b;
    }
    return pgcd2(b%a,a);
}
```

pgcd.c

Les pointeurs

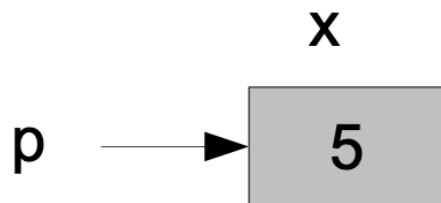
- Le langage C permet d'accéder aux 'adresses' (dans la mémoire virtuelle) des objets stockés dans sa mémoire (virtuelle)
- Pour cela, il utilise des **pointeurs**
- Leur compréhension et manipulation est au centre de la programmation en C
- Un pointeur en C est une variable qui contient l'adresse mémoire d'une autre variable. Contrairement aux variables normales qui contiennent des valeurs directes, un pointeur contient l'adresse de la mémoire où la valeur est stockée.

Récupérer un pointeur

- Pour récupérer, un pointeur vers une donnée stockée dans une variable x , on utiliser `&x` .
 - Le type du pointeur sera alors `type *` si `type` est le type de x .

```
unsigned x = 5 ;  
unsigned *p = &x ;
```

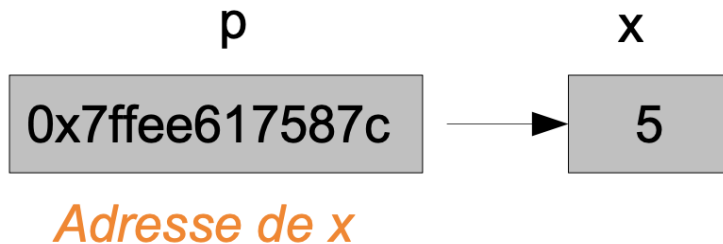
- On dit que **p pointe vers x** et **x est pointée par p** !



Les pointeurs comme adresse

- Le type d'un pointeur est extrêmement important pour la manipulation des pointeurs.
- Si on voit un pointeur p comme une adresse :
 - si p est un pointeur vers un unsigned stocké par exemple sur 4 octets
 - en fait p contient l'adresse du premier octet dans la mémoire virtuelle

```
unsigned x = 5 ;  
unsigned *p = &x ;
```



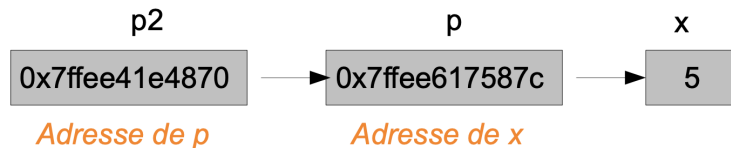
Remarque

- Sur la plupart de ces machines, l'adresse sera codée sur 64bits,
 - c'est-à-dire 8 octets,
 - donc en notation hexadécimale cela donnera 16 chiffres (ayant pour valeur 0,1,...,9,a,b,c,d,e,f)
- En comprenant cela, vous pouvez mieux interpréter et manipuler les adresses mémoire dans les programmes en langage C.
- Cette compréhension est importante pour des tâches telles que la gestion de la mémoire, le débogage et l'optimisation des performances.

Les pointeurs de pointeurs

- On peut faire un pointeur de pointeur

```
unsigned x = 5 ;  
unsigned *p = &x ;  
unsigned **p2=&p ;
```



- Dans ce cas, p2 pointe vers p qui pointer vers x
- Pour afficher la valeur d'un pointeur avec printf, on utilise le format %p !



```
//test-pointeur1.c
#include <stdio.h>
#include <stdlib.h>

int main(){
    unsigned x=5;
    unsigned *p=&x;
    unsigned **p2=&p;
    printf("%p \n",p);
    printf("%p \n",p2);
}
```


Déréférencement

- Le déréférencement en C fait référence à l'action d'accéder à la valeur ou à l'objet référencé par un pointeur.
- Lorsque vous déréférenciez un pointeur, vous utilisez l'opérateur d'indirection `*` pour accéder à la valeur stockée à l'adresse mémoire pointée par ce pointeur.
- On utilise l'opérateur `*` (que l'on fait précéder d'un pointeur) :

```
unsigned x = 5 ;  
unsigned *p = &x ;  
unsigned y=*p ;  
*p=10 ;
```

- Avoir accès à la valeur pointée
- Modifier la valeur pointée

Déréférencement

```
unsigned x = 5 ;//<-----  
unsigned *p = &x ;  
unsigned y=*p ;  
*p=10 ;
```

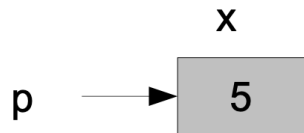
x

5

`unsigned x = 5;` : Déclare une variable non signée `x` et lui attribue la valeur 5.

Déréférencement

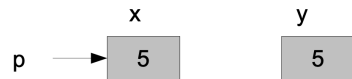
```
unsigned x = 5 ;  
unsigned *p = &x ; //<-----  
unsigned y=*p ;  
*p=10 ;
```



`unsigned *p = &x;` : Déclare un pointeur `p` vers un entier non signé et initialise `p` avec l'adresse de `x`. Maintenant, `p` pointe vers `x`.

Déréférencement

```
unsigned x = 5 ;  
unsigned *p = &x ;  
unsigned y=*p ;//<-----  
*p=10 ;
```



`unsigned y = *p;` : Déclare une variable non signée `y` et lui attribue la valeur de `x` .
Cela est réalisé en déréférençant le pointeur `p` , ce qui signifie que la valeur de `x` à l'adresse pointée par `p` est copiée dans `y` .

Déréférencement

```
unsigned x = 5 ;  
unsigned *p = &x ;  
unsigned y=*p ;  
*p=10 ;//<-----
```



- `*p = 10;` : Déréférence le pointeur `p` et lui attribue la valeur 10. Cela modifie la valeur de `x` à l'adresse pointée par `p` pour qu'elle devienne 10.
- `*p` permet de déréférencer la valeur pointée par `p`

ne pas confondre l'étoile dans la définition de p comme pointeur (`unsigned *p`) et dans le déréférencement (`y=*p` et `*p=10`)

- Lorsque vous déclarez un pointeur en C, vous utilisez l'opérateur `*` pour indiquer que la variable est un pointeur.
 - Par exemple, dans la déclaration `unsigned *p`, l'étoile (`*`) indique que `p` est un pointeur vers un entier non signé.
- Cependant, lorsque vous utilisez l'opérateur `*` dans une expression pour déréférencer un pointeur, cela signifie que vous accédez à la valeur ou à l'objet référencé par le pointeur.
 - Par exemple, dans les expressions `y = *p` et `*p = 10`, l'étoile (`*`) est utilisée pour déréférencer le pointeur `p` et accéder à la valeur de l'objet pointé.

Pourquoi des pointeurs ?

- Déjà dans certaines cas, ils permettent de 'contourner' le passage par valeur dans les fonctions
- Par exemple, une fonction `void echange(unsigned *p1,unsigned *p2)`
 - Elle va recevoir lors de son appel deux pointeurs vers des données
 - Avec le déréférencement, elle va pouvoir lire/modifier les données pointées
 - Une fois, son appel terminé, les valeurs pointées auront été changées

- La fonction suivante échange les valeur des données pointées par les arguments

```
void echange(unsigned *p1,unsigned *p2){  
    unsigned tmp = *p1;  
    *p1 = *p2;  
    *p2 = tmp;  
  
}
```

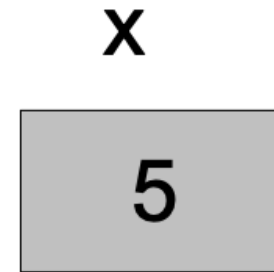


```
//echange.c
#include <stdio.h>
#include <stdlib.h>

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;//<--
    unsigned y=10;
    echange(&x,&y);
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;
    *p1=*p2;
    *p2=tmp;
}
```



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

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;
    unsigned y=10; //<--
    echange(&x,&y);
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;
    *p1=*p2;
    *p2=tmp;
}
```

x



y

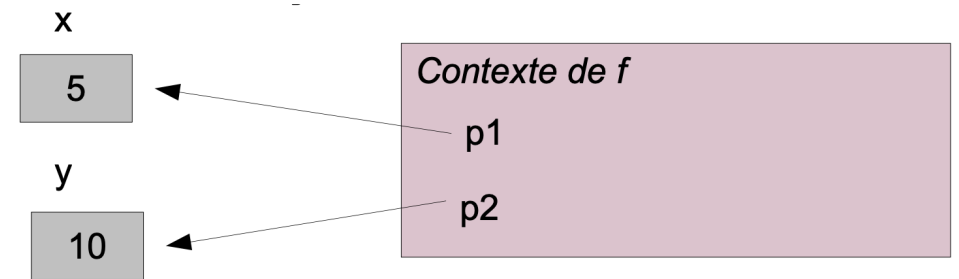


```
//echange.c
#include <stdio.h>
#include <stdlib.h>

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;
    unsigned y=10;
    echange(&x,&y);//<--
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;
    *p1=*p2;
    *p2=tmp;
}
```

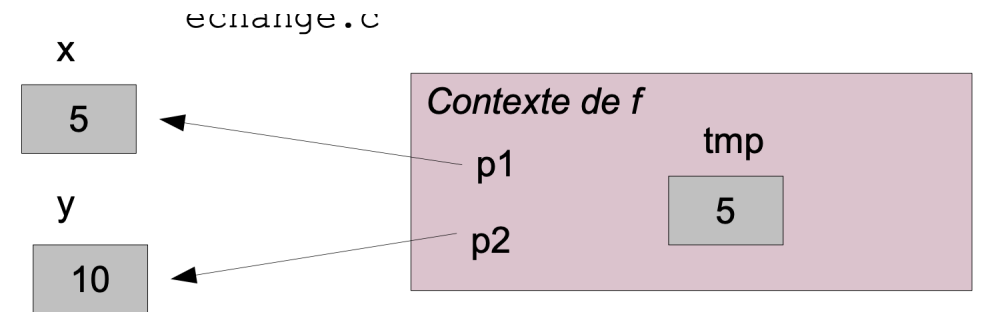


```
//echange.c
#include <stdio.h>
#include <stdlib.h>

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;
    unsigned y=10;
    echange(&x,&y);//<--
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;//<--
    *p1=*p2;
    *p2=tmp;
}
```

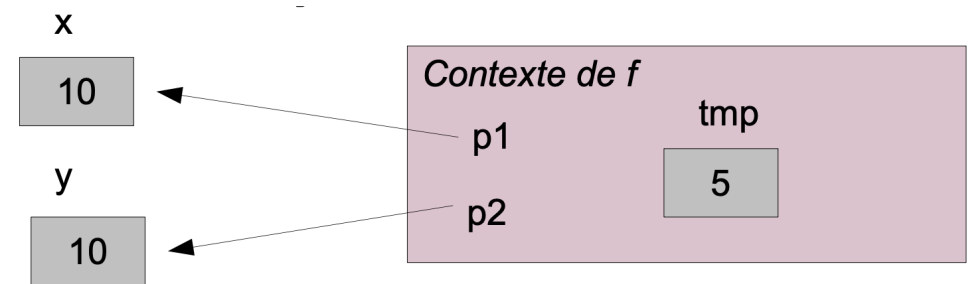


```
//echange.c
#include <stdio.h>
#include <stdlib.h>

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;
    unsigned y=10;
    echange(&x,&y);
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;
    *p1=*p2; //<--
    *p2=tmp;
}
```

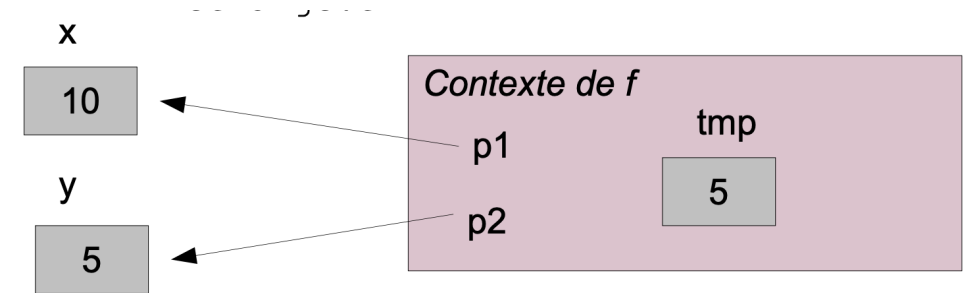


```
//echange.c
#include <stdio.h>
#include <stdlib.h>

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;
    unsigned y=10;
    echange(&x,&y);
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;
    *p1=*p2;
    *p2=tmp;//<--
}
```

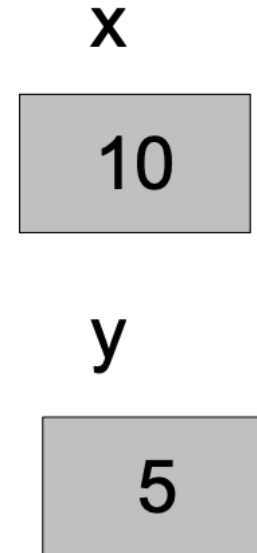


```
//echange.c
#include <stdio.h>
#include <stdlib.h>

void echange(unsigned *p1,unsigned *p2);

int main(){
    unsigned x=5;
    unsigned y=10;
    echange(&x,&y);
    printf("x vaut %u \n",x); // x vaut 10
    printf("y vaut %u \n",y); // y vaut 5
}

void echange(unsigned *p1,unsigned *p2){
    unsigned tmp=*p1;
    *p1=*p2;
    *p2=tmp;//<--
}
```



La valeur de pointeur null

- Il est utile de pouvoir initialiser les pointeurs avec une valeur spécifique
- Cette valeur signifie que l'adresse dans le pointeur est **une adresse ne correspondant à aucun emplacement dans la mémoire**
- Dans la plupart des programmes, on utilise la constante **NULL** (qui se trouve dans `<stdio.h>`)
- **On peut aussi utiliser 0**

Débat

- Il y a **débat** si utiliser 0 ou NULL, mais l'important c'est d'être consistant (soit vous utilisez toujours l'un soit
- **Attention :** si on a un pointeur qui est null, alors si on le dérèfère, on peut obtenir **une erreur de segmentation** (ou un comportement non défini)
- L'erreur de segmentation : segmentation fault survient lorsque l'on essaie d'accéder à une zone de la mémoire non définie

Exemple posant problème

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

int main(){
    unsigned *p=NULL;
    unsigned d=*p;
    printf("Valeur de d : %u\n",d);
    return EXIT_SUCCESS;
}
```

probleme-pointeur.c

Sur ma machine, produit : Segmentation fault !!

Pourquoi des pointeurs ?

- Les pointeurs peuvent aussi servir à modifier le champ d'une structure
- Pour rappel : lorsque l'on passe une structure en arguments d'une fonction, la structure est copiée

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

- Comment faire une fonction qui réinitialise les champs x et y d'une structure à 0 ?
- Il faut écrire une fonction qui prend en paramètre un pointeur vers une structure

```
void initialise(point *p){  
    (*p).x=0;  
    (*p).y=0;  
  
}
```



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

struct point{
    int x;
    int y; };

typedef struct point point;

void initialise(point *p);

int main(){
    point p={.x=5,.y=10};
    printf("Coordonnees du point: %d %d\n",p.x,p.y);
    initialise(&p);
    printf("Coordonnees du point: %d %d\n",p.x,p.y);
    return EXIT_SUCCESS;
}

void initialise(point *p){
    (*p).x=0;
    (*p).y=0;
}
```

initialise-point.c

Pointeur sur des structures

```
void initialise(point *p){  
    (*p).x=0;  
    (*p).y=0;  
}
```

- Première remarque sur ce code, on ne peut pas écrire *p.x sans les parenthèses car cela serait équivalent à *(p.x). **(le . est plus prioritaire que)*
- **L'écriture (p).x est un peu lourde mais peut être simplifiée par p->x*
- Ainsi si p est un pointeur de structure et membre est un champ de cette structure, on peut y accéder en faisant p->membre !

La fonction se réécrit

```
void initialise(point *p){  
    p->x=0;  
    p->y=0; }  

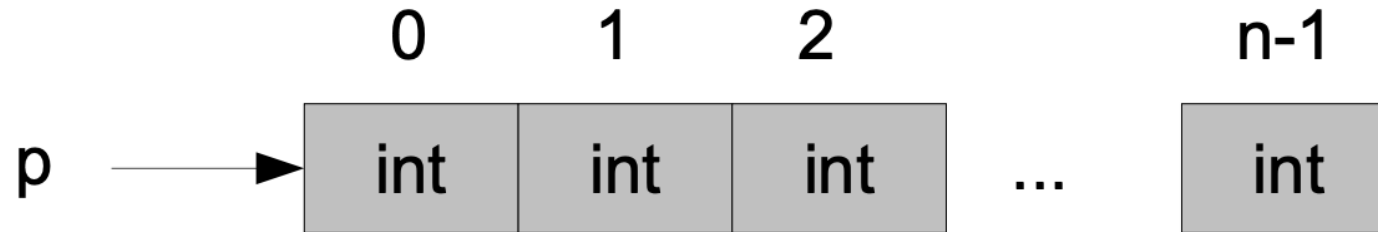
```

Erreur fréquente : confondre l'utilisation de . et de -> sur les structures !!!

Les pointeurs : parcours de tableaux

- En fait, un pointeur pointe sur le premier élément d'un tableau d'éléments du type de pointeur
- `unsigned *p` : p pointe sur un tableau d'entiers non signés
- `int *p` : p pointe sur un tableau d'entiers
- `char *p` : pointe sur un tableau de caractères

- Par exemple, si la zone pointée est un tableau de n entiers (int) :



- **MAIS on ne peut pas connaître la taille de la zone pointée !**

Les pointeurs : zone et déplacement

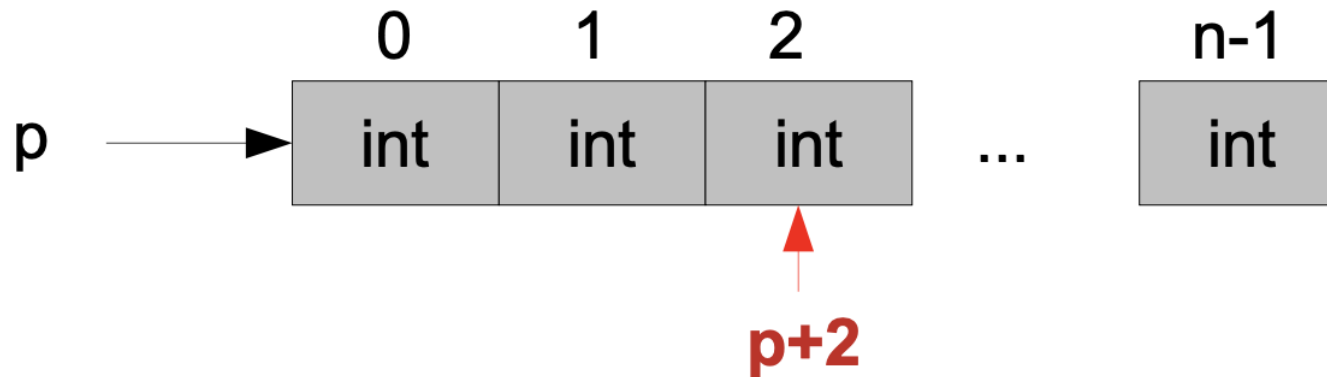
- Quand on fait :

```
unsigned x = 5 ;  
unsigned *p = &x ;
```

- En fait, p pointe vers une zone de taille 1 contenant un entier non signé
- D'où viennent du coup les zones plus grandes :
 - des tableaux
 - des **zones allouées dynamiquement** (cf prochain cours)

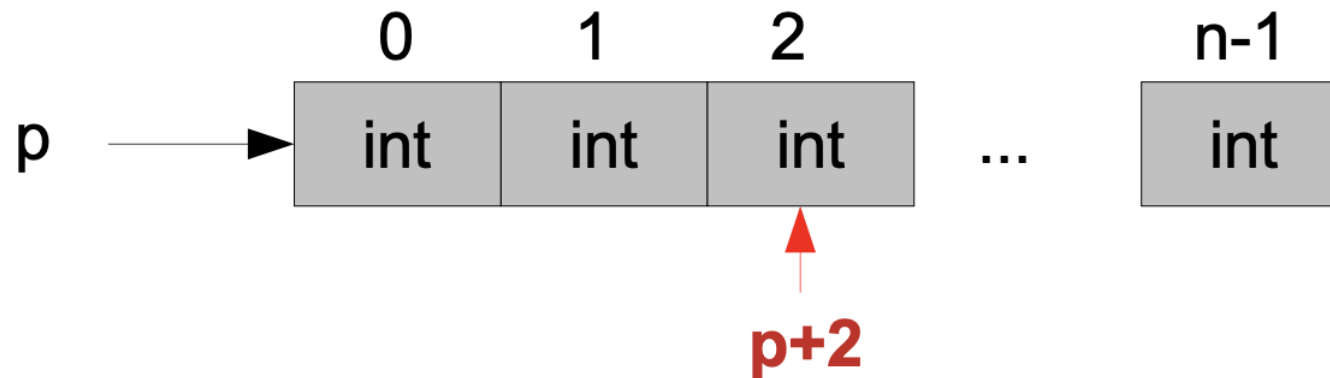
Déplacement du pointeur

- On peut déplacer le pointeur dans cette zone en ajoutant ou retirant des offsets
 - Si p est un pointeur on peut faire $p+k$ ou $p-k$ où k est un entier (de préférence de type `size_t`)



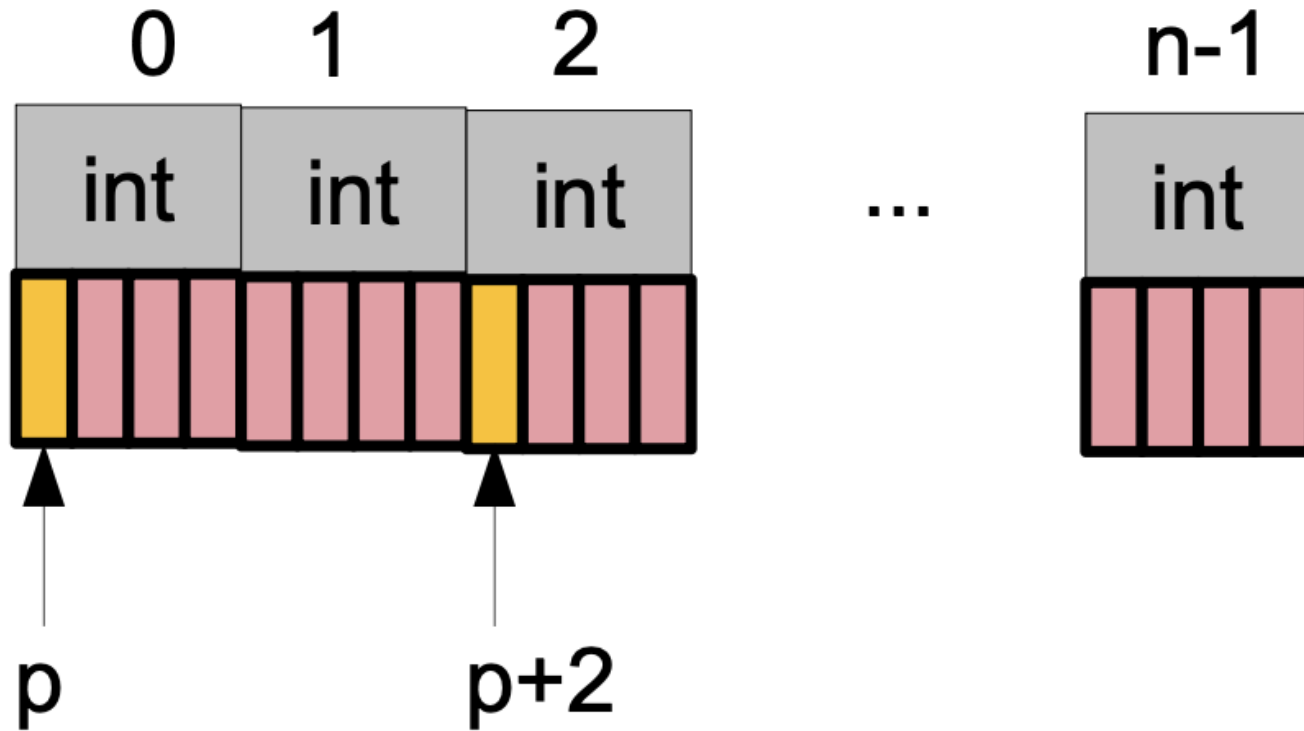
Les pointeurs : zone et déplacement

- Comment se passe le déplacement : mettons que p soit de type `int *`



- On sait que p pointe sur le premier octet de la première case du tableau
- Quand on fait $p+2$, si mettons que `int` est stockée sur 4 octets, alors $p+2$ va en fait déplacer p de $2*4$ octets !!!
- $p+2$ pointera alors sur le premier octet de la troisième case de la zone

- Pour cette opération déplacement on utilise le fait que p est de type int^* (le déplacement ne serait pas le même pour char^* par exemple)



Les pointeurs et tableaux

- Sur les pointeurs et sur les tableaux (si A est un pointeur ou un tableau) :
 - **A[i] est la même chose que (A+i)*
- En fait, si A est un tableau, quand on l'évalue (i.e. si il est dans une expression), alors il devient :
 - &A[0] : le pointeur sur le premier élément du tableau !!!
- Quand on passe un tableau en argument d'une fonction, celui-ci est transformé en pointer vers le premier élément du tableau
- **Rappel : on ne peut pas connaître la taille de la zone pointée !**
 - D'où la nécessité de donner en argument la taille de la zone pointée

Les pointeurs et tableaux

```
// Fonction qui calcule la somme des éléments d'un tableau d'entiers
int somme_element(int t[], size_t n){
    int r=0;
    for(size_t i=0; i<n; ++i){
        r+=t[i];
    }
    return r; }
```

```
//- Peut donc être réécrite sous la forme
int somme_element(int *t, size_t n){
    int r=0;
    for(size_t i=0; i<n; ++i){
        r+=*(t+i);
    }
    return r; }
```



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

int somme_element(int *,size_t);

int main(){
    int t[4]={0,1,2,3};
    int s=somme_element(t,4);
    printf("La somme vaut %d\n",s);
    return EXIT_SUCCESS;
}

int somme_element(int *t,size_t n){
    int r=0;

    for(size_t i=0;i<n;++i){
        r+=*(t+i);
    }

    return r; }
```

somme-tab.c

Les pointeurs et tableaux

- **Les tableaux ne sont pas des pointeurs**
- **Pourquoi ?**
 - Pour connaître la taille d'un tableau (non vide) `t`, on peut faire `size(t)/size(t[0])` (**à ne pas utiliser**), pas sur un pointeur
 - Si `tab1` et `tab2` sont des tableaux d'entiers, on ne peut pas faire d'affectation `tab1=tab2`.
 - Sur les pointeurs, on peut !
- On ne peut pas comparer deux tableaux.
 - On peut comparer deux pointeurs pour voir si ils indiquent le même emplacement

Arithmétique de pointeurs

- Comme nous l'avons vu, si p est un pointeur :
 - On peut faire $p+k$, le résultat est un pointeur de même type
 - Cela déplace p de k éléments 'en avant' dans la zone pointée (en prenant en compte le type de p)
 - On peut faire $p-k$
 - Cela déplace p de k éléments 'en arrière' dans la zone pointée (en prenant en compte le type de p)
- On peut aussi faire $p-p_2$ où p_2 est un pointeur de même type. Cela donne une valeur entière de type **ptrdiff_t** (qui se trouve dans **<stddef.h>**)
 - Cette valeur peut être négative

```
int main(){ //arithmetique1.c
    int t[5]={0,1,2,3,4};
    printf("Size of int : %zu\n",sizeof(int));
    int *ptr=t;
    printf("Adresse premiere case : %p\n",ptr);
    ++ptr;
    printf("Adresse deuxieme case : %p\n",ptr);
    ++ptr;
    printf("Adresse troisieme case : %p\n",ptr);
}
```

```
>gcc -Wall -o arithmetique1 arithmetique1.c
>./arithmetique1
Size of int : 4
Adresse premiere case : 0x7ffeecacd840
Adresse deuxieme case : 0x7ffeecacd844
Adresse troisieme case : 0x7ffeecacd848
```

```
int main(){//arithmetique2.c
    char tab[5]={'a','b','c','d'};
    printf("Size of char : %zu\n",sizeof(char));
    char *ptr=tab;
    printf("Adresse premiere case : %p\n",ptr);
    ++ptr;
    printf("Adresse deuxieme case : %p\n",ptr);
    ++ptr;
    printf("Adresse troisieme case : %p\n",ptr);
}
```

```
>./arithmetique2
Size of char : 1
Adresse premiere case : 0x7ffedfd3d85b
Adresse deuxieme case : 0x7ffedfd3d85c
Adresse troisieme case : 0x7ffedfd3d85d
```

```
//parcours3.c
void affiche_inv(int *,size_t);

int main(){
    int t[4]={0,1,2,3};
    affiche_inv(t,4);
    return EXIT_SUCCESS;
}

void affiche_inv(int *tab,size_t taille){
    for(int *ptr=tab+taille-1;ptr>=tab;--ptr){

        printf("%d ",*ptr);
    }

    printf("\n");
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
//renverse.c

void renverse(int *,size_t);

int main(){
    int t[8]={0,1,2,3,4,5,6,7};
    renverse(t,8);
    for(int *p=t;p<t+8;++p){

        printf("%d ",*p);
    }

    printf("\n");

    return EXIT_SUCCESS;
}

void renverse(int *tab,size_t taille){
    for(size_t i=0;i<taille/2;++i){

        int tmp=*(tab+i);
        *(tab+i)=*(tab+(taille-1-i));
        *(tab+taille-1-i)=tmp;
    }
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
//renverse2.c
void renverse(int *,size_t);

int main(){
    int t[7]={0,1,2,3,4,5,6};
    renverse(t,7);
    for(int *p=t;p<t+7;++p){

        printf("%d ",*p);
    }

    printf("\n");

    return EXIT_SUCCESS;
}

void renverse(int *tab,size_t taille){
    int *ptr1=tab;
    int *ptr2=tab+taille-1;
    while(ptr2>ptr1){

        int tmp=*ptr1;
        *ptr1=*ptr2;
        *ptr2=tmp;
        ++ptr1;
        --ptr2;
    } }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
//renverse3.c
void renverse(int *,size_t);

int main(){
    int t[7]={0,1,2,3,4,5,6};
    renverse(t,7);
    for(int *p=t;p<t+7;++p){
        printf("%d ",*p);
    }
    printf("\n")
    return EXIT_SUCCESS;
}

void renverse(int *tab,size_t taille){
    int *ptr1=tab;
    int *ptr2=tab+taille-1;
    ptrdiff_t pdiff=ptr2-ptr1;
    while(pdiff>0){
        int tmp=*ptr1;
        *ptr1=*ptr2;
        *ptr2=tmp;
        ++ptr1;
        --ptr2;
        pdiff=ptr2-ptr1;
    } }
```


Code de bonne conduite

- Ayez toujours bien en tête où sont vos pointeurs (quitte à rajouter des tests)
 - L'erreur de segmentation deviendra bientôt votre pire ennemi
- Ne jamais déplacer un pointeur hors d'une zone valide ou alors dans la case juste après ou juste avant

Code de bonne conduite

- Évitez de faire des cast sur des pointeurs
 - Comme par exemple : `int *p = &d` ; puis `unsigned *q=(unsigned *) p` ;
- Faites des différence de pointeurs uniquement entre pointeurs d'une même zone
- Évitez de mettre des pointeurs dans des typedef (pour bien savoir si vous manipulez un pointeur ou non)
 - Évitez des choses comme `typedef int *inptr;`