Retour sur les pointeurs en C

Come back to the pointers

L2 SPI, Bruno Jacob

1 Mécanisme des pointeurs

NB : objet est ici à prendre au sens large. C'est ce qui peut être atteint par un pointeur. Ce peut être une variable, une zone mémoire, une structure...

1.1 Définition d'un pointeur

- pointeur = 1 variable (comme les autres) qui contient 1 adresse d'un objet
- la capacité de la variable dépend de la taille de la mémoire
  Exemple : si pointeurs sur 4 octets (4×8 bits) alors on peut adresser 2^{32} adresses, c'est à dire 4 Giga objets

1.2 Spécificité d'un pointeur

Accès aux objets pointés par adressage indirect.

1.2.1 Déclaration d'un pointeur

Il n'y pas de type pointeur mais une *déclaration* de pointeur sur un type d'objet

Déclaration :

```
type_t * p ;
```

On a déclaré une variable `p` qui est un pointeur sur une variable de type `type_t`

Exemples :

```
int * p ; /* pointeur sur un int */
int * p[N] ; /* tableau de N pointeurs sur des int */
int (*p)[N] ; /* un pointeur sur 1 tableau de N entiers */
```

1.2.2 Principe de l'adressage indirect

On peut accéder au contenu de l'objet par adressage indirect (technique fréquente en assembleur).

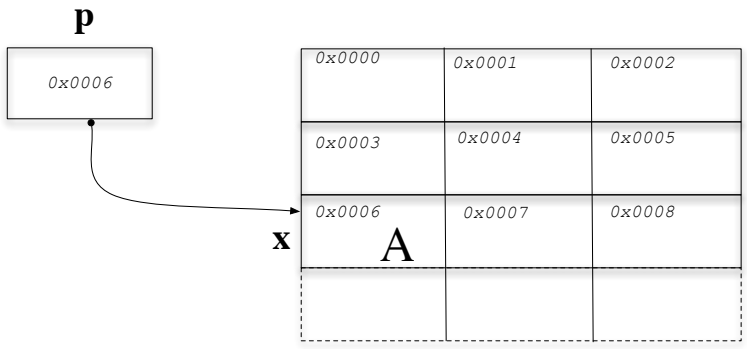
Opérateurs en C

`*` v : contenu d'une variable `v`

`&` v : adresse d'une variable `v`

Principe : supposons que

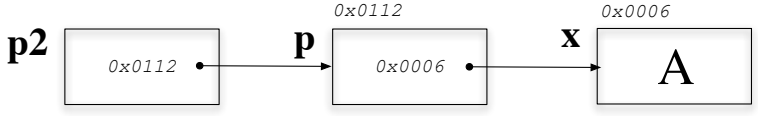
```
char x = 'A' ;
char * p = &x ;
```



— `p` = 6 = adresse de `x`
— `*p` ou `(*p)` = contenu de `x` = A
— `&x` = adresse de `x` = `0x0006` = valeur de `p`

Mais `p` est une variable en mémoire et possède donc une adresse. Cette adresse peut être stockée dans une autre variable

```
char ** p2 = &p ;
```



Donc `p2` est un pointeur qui contient l'adresse d'un pointeur = pointeur de pointeur

— `&p` = `0x0112`
— `*p2` = contenu de `p` = adresse de `x`
— `**p2` = contenu de `x`

`adressage_indirect.c`

1.3 Arithmétique des pointeurs

1.3.1 Addition p+i sur objets homogènes

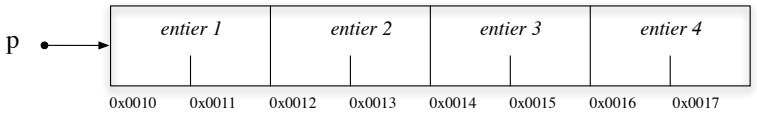
Syntaxe ;

```
type_t * p ;
int i ;
...
p=p+i ;
```

Si $i = 1$ alors $p = p + 1 \Leftrightarrow p + +$

Cette expression est transformée en : $p = p + taille(type\_t) \times i$

Exemple avec des entiers :
Supposons que la taille d'un `int` est de 2 octets (2ø). Si nous avons un pointeur sur une zone mémoire contenant 4 entiers



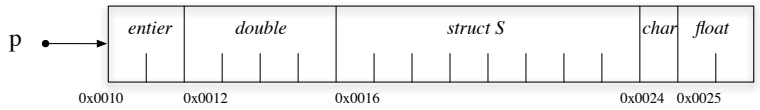
alors
$(*p) \rightarrow entier\ 1$
$p + + \Rightarrow p = p + (taille(int)) = 0x0010 + 2 = 0x0012$
$(*p) \rightarrow entier\ 2$
$p = p + 2 \Rightarrow p = p + (taille(int)) = 0x0012 + (2ø) = 0x0016$
$(*p) \rightarrow entier\ 4$

`arithm_pointeur_1.c`

1.3.2 Addition p+i sur objets hétérogènes

On est obligé d'utiliser des "forceurs" (cast) pour faire

— connaitre la taille des objets au compilateur
— donc de calculer le déplacement correctement



Au début p pointe sur le 1^{er} élément (l'entier)
$p \rightarrow entier$
pour le 2^{ieme} élément (le double)
$p = p + (taille(int)) \rightarrow (int*)p = (int*)p + 1$
pour le 3^{ieme} élément (la structure)
$p = p + (taille(double)) \rightarrow (double*)p = (double*)p + 1$
pour le 4^{ieme} élément (le char)
$p = p + (taille(structS)) \rightarrow (structS*)p = (structS*)p + 1$
pour le 5^{ieme} élément (le float)
$p = p + (taille(char)) \rightarrow (char*)p = (char*)p + 1$

`arithm_pointeur_2.c`

1.3.3 Autres opérateurs

Soustraction p-i : idem p+i

Soustractions de 2 pointeurs p1-p2 :

— Soustraction de 2 pointeurs d'objets homogènes : donne le nombre d'éléments
— Soustraction de 2 pointeurs d'objets hétérogènes : donne n'importe quoi

Comparaison de 2 pointeurs p1==p2 : comparaison d'adresses.

2 pointeurs sont comparables ssi **ils ont le même type**

— Egalité : $p1 == p2$ mais attention :

```
type1_t * p1 ;
type2_t * p2 ;
...
p1 == p2 ; /* Attention !!! */
```

Egalité vraie mais rien ne permet d'affirmer que $p1 + 1 == p2 + 1$ car on ne connaît pas les tailles de `type1_t` et `type2_t`

— Autres : $!=, <, >, <=, >=$ possibles mais n'ont d'intérêt que pour les éléments d'un même tableau car ils sont en principe rangés en mémoire

`arithm_pointeur3.c`

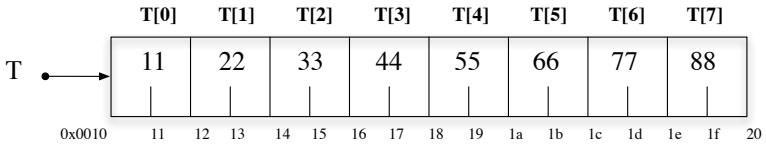
1.3.4 Utilisation arithmétique pointeur : indexations de tableaux

Déjà vue dans les cours précédents donc on passe vite dessus.

Tableau à 1 dimension

Principe :
Si `type_t T[N]` alors `T[i]` ⇔ `*(T+i)`
Le compilateur le transforme en `*(T+i×taille(type_t))`

Exemple :



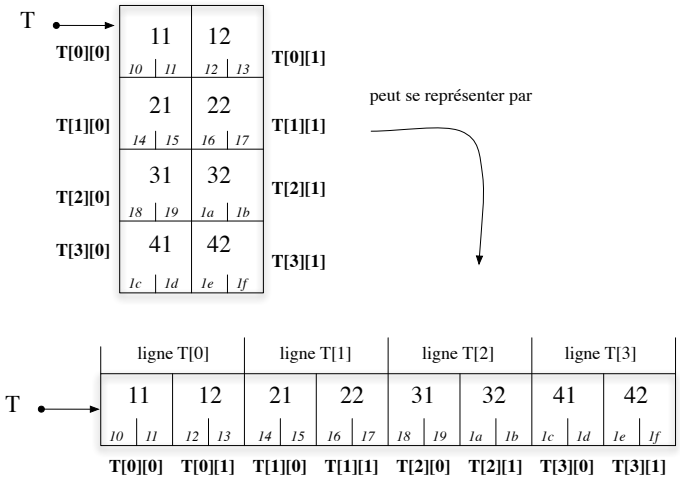

```
int T[8] ;
...
printf(..., T[5] ) ;
```

Si la taille d'un `int` est de 2ø alors `T[5]` ⇔ `*(T+5)`
Compilateur le transforme en `*(T+5×2ø)`
= `*(T+10)` = `*(0x0010+10)` = `*(0x001a)` = 66

Tableau à 2 dimensions

Principe :
Si on a déclaré un tableau `type_t T[L][C]` alors
`T[i][j]` ⇔ `*(T[i]+j)` avec
— `T[i]` = i tableaux de C éléments de taille `type_t` et
— `j` = 1 élément de taille `type_t`
⇔ `*(T + i×C + j)`
Compilateur ⇒ `*(T + i×C×taille(type_t) + j×taille(type_t))`

Exemple :




```
int T[4][2] ;
...
printf(..., T[2][1] ) ;
```

La taille d'une ligne est de 2 éléments (nombre de colonnes)
`T[2][1]` ⇔ `*(T+ 2×2 + 1)`

Si la taille d'un `int` est de 2ø alors
Compilateur ⇒ *(T+ 2×2×2ø+ 1×2ø)
= `*(T+8+2)` = `*(0x0010+0x000a)` = `*(0x001a)` = 32

1.4 Optimisation

L'arithmétique des pointeurs permet d'améliorer le parcours des tableaux en utilisant le fait qu'une expression du type *"indirection"* `*p` (adressage indirect simple) est réputée (légèrement) plus rapide qu'une expression du type *"indexation"* `*(p+i)` ou `p[i]` (arithmétique pointeur + adressage indirect).

Exemple :

La fonction `strcpy(cible,source)` qui copie la chaîne de caractères `source` dans l'espace pointé par `cible` sera un peu moins efficace dans cette première forme

```
char * strcpy( char cible[] , char source[] )
{
  register int i = 0 ;
  while( ( cible[i] = source[i] ) != '\0' )
    i++ ;

  return(cible) ;
}
```

que dans cette deuxième forme

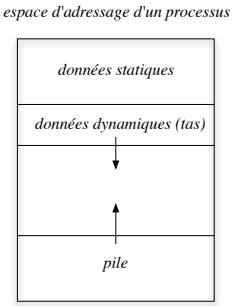
```
char * strcpy( char * cible , char * source )
{
  register char * c = cible ;
  register char * s = source ;
  while( ( *c++ = *s++ ) != '\0' ) ;

  return(cible) ;
}
```

car on a supprimé l'indexation.

`strcpy_1.c vs strcpy_2.c`

1.5 Pointeurs sur des objets statiques

Un processus comporte 3 zones principales :



La place mémoire de l'objet est réservée à la compilation :

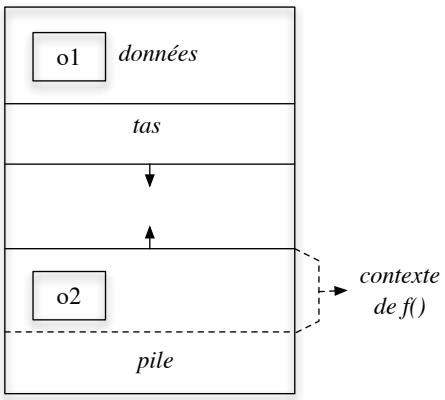
— placée dans la zone statique si c'est une variable globale

— placée dans la pile si c'est une variable d'une fonction à l'appel de cette fonction

Exemple :

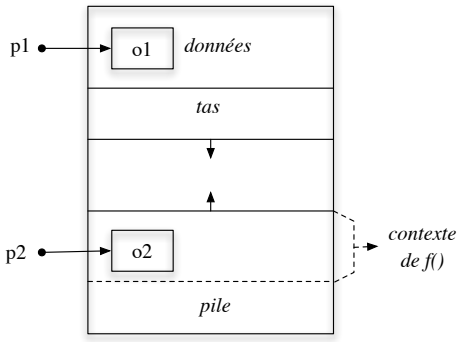
```
type1_t o1 ;
....
f()
{
        type2_t o2 ;
}

main()
{
        f() ;
}
```

Les objets créés seront semblables à :



Avec des pointeurs sur ces objets :

```
type1_t o1 ;
....
f()
{
        type2_t o2 ;
        type2_t * p2 = &o2 ;
}

main()
{
        type1_t * p1 = &o1 ;
        f() ;
}
```



1.5.1 Usage standard

Déclaration

```
typedef struct S_s
{
        char c1 ;
        double c2 ;
} S_t ;

int * p1 ;
S_t * p2 ;
```

Initialisation

```
int v1 ;
S_t v2 ;
p1 = &v1 ;
p2 = &v2 ;
```

Utilisation

```
(*p1) = 99 ;
(*p2).c1 = 'a' ;
(*p2).c2 = 33333 ;
```

Simplification : `(*p).c` $\Leftrightarrow$ `p->c` . Donc le listing peut s'écrire :

```
(*p1) = 99 ;
p2->c1 = 'a' ;
p2->c2 = 33333 ;
```

Attention

```
v2.c1 = 'a' ; /* OK */ v2->c1 = 'a' ; /* KO */
p2.c1 = 'a' ; /* KO */ p2->c1 = 'a' ; /* OK */
```

Suppression

- On ne peut pas détruire (libérer la place mémoire prise par) un objet statique
- on peut en revanche détruire le pointeur sur celui ci (`NULL`)

```
p1 = NULL ;
p2 = NULL ;
```

1.5.2 Avantages

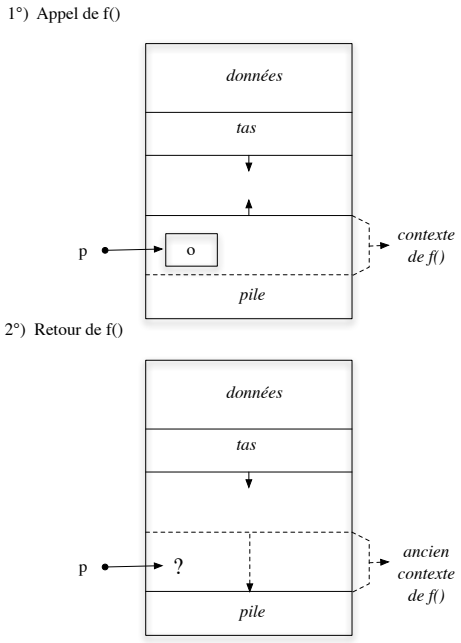
- pas de fuite mémoire

1.5.3 Pièges

- risque de pointeurs fous : quand on se situe au point d'arrêt 2 de l'exemple précédent
- quand la fonction renvoie un pointeur sur une de ses variables locales

Exemple :

```
type_t * f()
{
        type_t o ;
        type_t * p =&o ;
}
main()
{
        type_t * p2 = NULL ;
        p2 = f() ; /* KO */
}
```



`objets_statiques_pointeur_fou.c`

1.6 Pointeurs sur des objets dynamiques

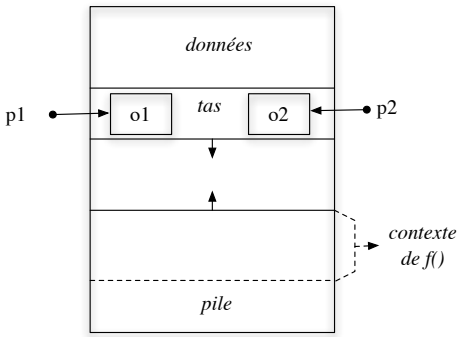
La place mémoire de l'objet est réservée à l'exécution. Repose sur les fonctions d'allocation de mémoire

- `void * malloc(size_t size);` : réservation d'une zone
- `void * realloc(void *ptr, size_t size);` : réutilisation d'une zone
- `void * calloc(size_t count, size_t size);` : réservation et mise à zéro d'une zone
- `void free(void *ptr);` : libération d'une zone

Exemple :

```
type1_t * o1 ;
....
f()
{
 type2_t * o2 = malloc(sizeof(type2_t));
}
main()
{
 o1 = malloc(sizeof(type1_t)) ;
 f() ; /* <-- point d'arret 1 */
 /* <-- point d'arret 2 */
}
```

Que l'on situe au point d'arrêt 1 ou 2 le schéma de la mémoire sera le suivant :



1.6.1 Usage standard

Déclaration (idem)

```
typedef struct S_s
{
        char c1 ;
        double c2 ;
} S_t ;

int * p1 ; /* bien */ int * p1 = NULL ; /* mieux */
S_t * p2 ; /* bien */ S_t * p2 = NULL ; /* mieux */
```

Initialisation

```
p1 = malloc(sizeof(int)) ;
p2 = malloc(sizeof(S_t)) ;
```

Utilisation (idem)

```
(*p1) = 99 ;
p2->c1 = 'a' ;
p2->c2 = 33333 ;
```

Suppression

- Il faut libérer la place mémoire prise par l'objet (sinon risque de fuite mémoire)
- on peut aussi indiquer que le pointeur ne pointe plus sur rien (`NULL`)

```
free(p1) ;
(*p1) = NULL ;
free(p2) ;
(*p2) = NULL ;
```

1.6.2 Avantages

- les objets ne sont pas volatiles

1.6.3 Pièges

Possibilités de

- fuites mémoire : si nombre de `malloc` $\neq$ nombre de `free` sur les mêmes objets (compteur de références sur les objets)
- pointeurs fous : si utilisation d'un pointeur après un `free`

1.7 Pointeurs sur les fonctions

Le nom d'une fonction = adresse du début de la fonction
(idem int T[N] → T = @ début de T)
Exemple :

```
double F()
{
 ...code diabolique...
}
```

F = adresse du début de F, pointeur sur F
⇔ F → début des instructions de F, pour l'exécuter il faut mettre les ()
⇔ F() → exécution, appel de F

On a le droit de manipuler F comme un pointeur

```
F2 = F ;
F2() ; /* exécute F */
```

Déclaration Syntaxe :

```
type_t (*pt_fonction)(...paramètres...) ;
```

Pointeur sur une fonction
- renvoyant un résultat de type `type_t`
- de nom `pt_fonction`
- ayant les paramètres entre les parenthèses. Attention : si pas de paramètre alors mettre `void` sinon pris pour un appel de fonction

Exemple sans paramètre

Dans l'exemple précédent

```
double (*F2)(void) ;
```

Pointeur sur une fonction :
- renvoyant un double
- n'ayant pas de paramètre

Exemple avec paramètres

```
int (*F3)( int , S_t * ) ;
```

Pointeur sur une fonction :
- renvoyant un `int`
- ayant deux paramètres : un `int` et un pointeur sur un objet `S_t`

`pointeurs_fonctions.c`

1.8 Salade de pointeurs

- `type_t *p` : pointeur sur un objet de type_t
- `type_t *p[10]` : tableau de 10 pointeurs sur des type_t
- `type_t *p(void)` : fonction qui renvoie un pointeur sur un type_t
- `type_t (*p)(void)` : pointeur sur une fonction qui renvoie un type_t et n'ayant pas de paramètre
- `type_t *(*p)(void)` : pointeur sur une fonction qui renvoie un pointeur sur un type_t et n'ayant pas de paramètre
- `type_t (*p[10])(void)` : p est un tableau de 10 pointeurs sur des fonctions qui renvoient un type_t
- `type_t *(*p[10])(void)` : p est un tableau de 10 pointeurs sur des fonctions qui renvoient un pointeur sur un type_t
- `type_t (*p2)(void) p(int , type_t (*p2)(void) f)` : p est une fonction
  - qui renvoie un pointeur de fonction.
  - qui a 2 paramètres
    - un int
    - un pointeur de fonction

  Les 2 pointeurs de fonctions sont du même type : c'est un pointeur sur une fonction qui renvoie un `type_t` et qui n'a pas de paramètre

Exemple réel : La fonction

```
void (*signal(int sig, void (*func)(int)))(int);
```

permet de définir un gestionnaire de signal. La signature de cette fonction étant un peu complexe, on peut la simplifier en utilisant un `typedef` :

```
typedef void (*t_handler)(int);
t_handler signal(int sig, t_handler func);
```

1.9 Pointeurs et constantes

Utilisation du mot clé `const` pour indiquer qu'une variable est une constante

- compilation/exécution plus efficace
- donne des garanties à l'utilisateur sur vos fonctions

Exemples

- `char *p` : pointeur sur une variable. La valeur du `char` et le pointeur peuvent changer
- `const char * p` : pointeur sur une constante. Le pointeur peut changer mais pas la valeur du `char`
- `char * const p` : pointeur constant sur une variable. Le pointeur ne peut changer mais la valeur du `char` si.
- `const char * const p` : pointeur constant sur une constante. La valeur du `char` et le pointeur ne peuvent pas changer

Attention

```
/* Declaration d'une constante qui ne devrait pas changer */
const int x = 45 ;
/* Pointeur sur cette constante */
const int * p = &x  ;
printf( "constante x AVANT = %d\n" , x ) ;
/* Or ceci passe sur certains compilateurs */
(*(int *)p) = 67 ;
printf( "constante x APRES = %d\n" , x ) ;
```

Le "forceur" (ou le "cast") `(int *)` "casse" le `const int *` et la modification de la constante devient possible.

2 Utilisation des pointeurs

2.1 Référence sur des objets

Au lieu de manipuler des objets complexes dans leur ensemble, on préfère manipuler des références/pointeurs sur eux.

2.1.1 Structures "standard"

Pointeurs dans des structures qui pointent sur d'autres structures.

```
typedef struct brosse_a_dent_s
{
        char * couleur ;
        int longueur ;
} brosse_a_dent_t ;
```

```
typedef struct individu_s
{
        char * prenom ;
        char * nom ;
        brosse_a_dent_t * brosse ;
} individu_t ;
```

NB : `brosse_a_dent_t` est à déclarer avant `individu_t`

2.1.2 Structures récursives

Structures ayant des pointeurs sur elles-mêmes. Exemples courants : les listes chaînées, les arbres...

```
typedef struct noeud_s
{
 char * etiquette ;
 struct noeud_s * fils_gauche ;
 struct noeud_s * fils_droit ;
} noeud_t ;
```

ou

```
typedef struct noeud_s noeud_t ;
struct noeud_s
{
 char * etiquette ;
 noeud_t * fils_gauche ;
 noeud_t * fils_droit ;
} ;
```

2.1.3 Structures mutuellement récursives

Par exemple, si on a 2 structures S1 et S2, on dit qu'elles sont mutuellement récursives si S1 a des pointeurs sur S2 et réciproquement S2 a des pointeurs sur S1.

```
typedef struct famille_s
{
 char * nom ;
 struct individu_s * membres ;
 struct famille_s * suivante ;
} famille_t ;
```

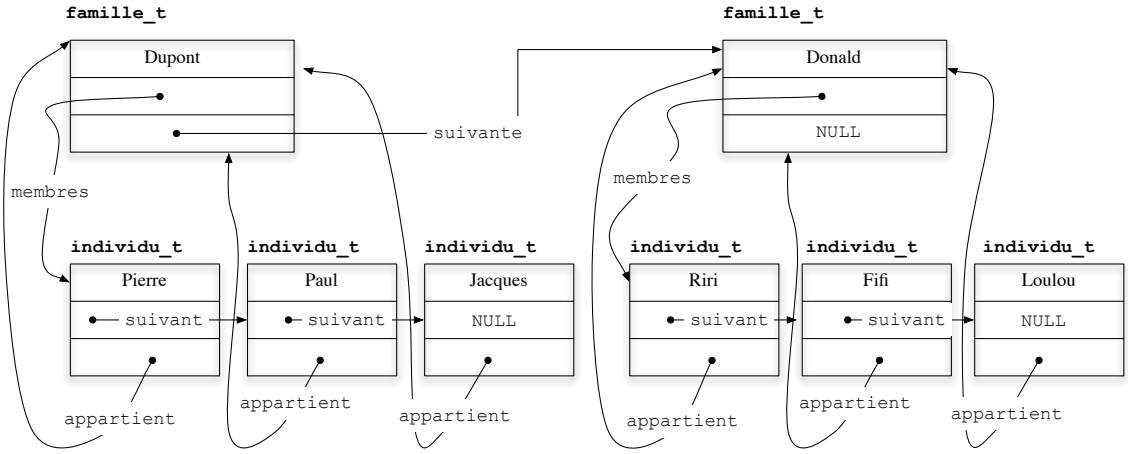
```
typedef struct individu_s
{
 char * prenom ;
 struct individu_s * suivant ;
 struct famille_s * appartient ;
} individu_t ;
```

ou

```
typedef struct famille_s famille_t ;
typedef struct individu_s individu_t ;
```

```
struct famille_s
{
 char * nom ;
 individu_t * membres ;
 famille_t * suivante ;
} ;
```

```
struct individu_s
{
 char * prenom ;
 individu_t * suivant ;
 famille_t * appartient ;
} ;
```



2.2 Partage d'objets

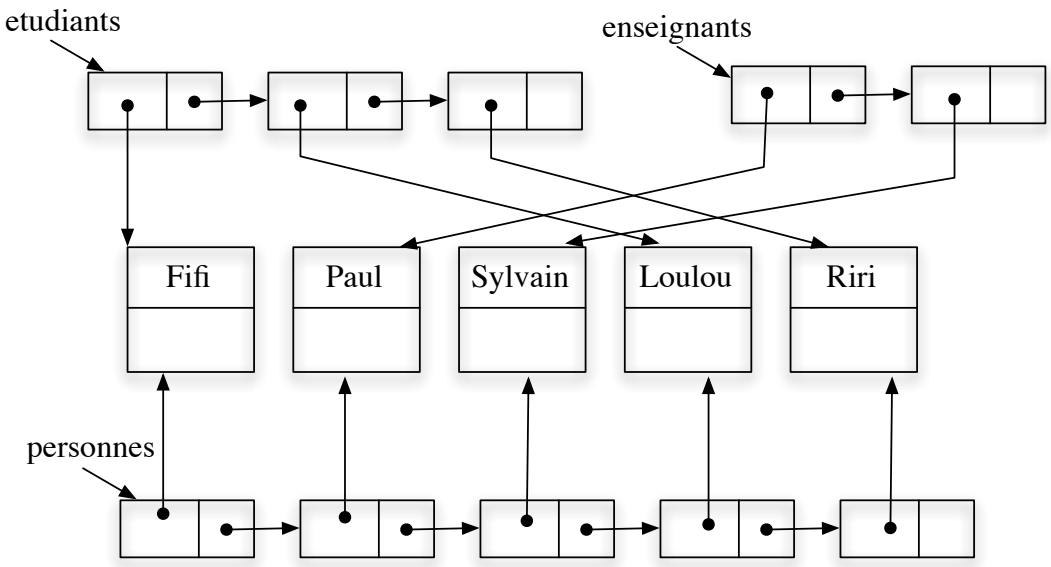
Quand on a plusieurs références sur un même objet.

Exemple Supposons que nous ayons des listes des enseignants et des étudiants de l'IC2, qui sont toutes des personnes.

```
typedef struct individu_s
{
 char * prenom ;
 char * nom ;
} individu_t ;
```

```
typedef struct liste_s
{
 individu_t * individu ;
 struct liste_s * suivant ;
} liste_t ;
```

```
liste_t * personnes , * enseignants , * eleves ;
```



Intérêts

Plutôt qu'une duplication des données, nous avons
— un gain de mémoire
— mais surtout des mises à jours plus sûres et plus rapides

Précautions

Attention à la suppression des listes. Pour éviter les pointeurs fous et/ou les fuites de mémoire il faut par exemple :

1. Supprimer les listes étudiants et enseignants → que les cellules pas les individus : `free(chainon)`
2. Supprimer la liste des personnes → les cellules et les individus : `free(chainon)` + `free(chainon->individu)`

2.3 Passage de résultats en paramètres des fonctions

En C le passage des paramètres se fait *par valeur*.

— Si une variable V est passée en paramètre d'une fonction F alors c'est une **copie** de V qui est transmise (une variable de même type avec le même contenu). Ce qui est modifié dans F c'est la copie de V

— on doit donc passer l'adresse de V : `&V` et la manipuler comme un pointeur si on veut stocker les modifications dans V

Mauvais exemple

Définition de la fonction

```
F( int x )
{
   x = 10 ;
}
```

Appel

```
main()
{
   i = 5 ;
   F(i) ;
}
```

Exemple correct

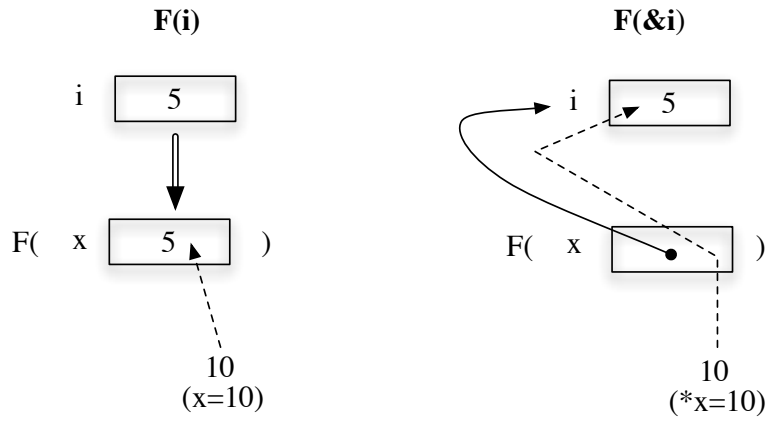
Définition de la fonction

```
F( int * x ) /* pointeur sur un résultat */
{
   (*x) = 10 ;
}
```

Appel

```
main(
{
   i = 5 ;
   F(&i) ; /* @ du resultat */
}
```

Différences entre les 2 exemples



`parametres_resultat.c`

2.4 Passage de paramètres complexes à des fonctions

On peut consédérer comme complexe des variables comme tableaux, structures, chaînes de caractères, fonctions (tout ce qui n'est pas un type "simple" comme les `int` , `float` ...). Si il y a un tel paramètre, même s'il n'est pas modifié, alors on ne passe pas sa valeur mais un pointeur sur celui-ci.

Argument	*Déclaration paramètre*	*Passage paramètre*
`int T[10000]`	`F( int * T )`	`F(T)`
`noeud_t * N1` `noeud_t N2`	`F( noeud_t * N )`	`F(N1)` `F(&N2)`
`char * s1 = "abc"` `char s2[4] = "abc"`	`F( char * s )`	`F(s1)` `F(s2)`
`void F2(int i)`	`F( void (*f)(int) )`	`F(F2)`

2.5 Pointeurs de fonctions dans les structures

Assimilable à la programmation objet : des méthodes dans des objets
- les types des objets sont les structures
- les attributs sont les champs de ces structures
- les méthodes sont des pointeurs de fonctions dans ces structures

Exemple d'affichage d'objets
- principe simplifié

`affichage_objets.c`

- pour se rapporcher de la technique des objets : envoie de messages aux objets avec une fonction générique d'aiguillage

`affichage_objets_2.c`

3 Généricité en C

3.1 Pointeurs génériques

Un pointeur pointe sur une adresse qui peut en principe contenir n'importe quel objet → objet générique. Pour déclarer un tel pointeur

```
void * p ;
```

`void` ≈ "bon à tout, bon à rien". Un pointeur `void *` pointe sur n'importe quoi → c'est un pointeur générique

Voici les spécificités des pointeurs génériques :

3.1.1 Compatible avec n'importe quel pointeur

Affectation avec n'importe quel pointeur sans `cast` (`int *` , `float *` , `double *` , `struct *` , `union *` ...)

```
void * p ;
p = autre pointeur ;
et
autre pointeur = p ;
```

Exemples :

```
struct individu_s * ind ; /* ou individu_t * ind */
void * p = ind ;
```

comme la fonction `malloc` à la signature suivante :

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

on peut écrire sans faire appel à des "forceurs"

```
ind = malloc(sizeof(individu_t));
```

NB : pour limiter les erreurs, on *ne doit pas* faire de `cast`

3.1.2 Opérations interdites

Le problème est que l'on ne connaît pas la taille des objets pointés, donc :

— pas d'arithmétique de pointeurs tels que

```
p++ , p--, *(p+i) ...
```

— on ne peut pas déférencer les pointeurs / pas d'indirection

```
void * p ;
individu_t * ind ;
p = ind ;
(*p) --> interdit
```

3.1.3 Opérations autorisées

1. Affectation

```
void * p ;
void * q ;
p = q ; --> autorisé
```

2. Passage de paramètres

```
void F( void * ) ;
...
int * p ;
float * q ;
F(p) ; --> OK
F(q) ; --> OK
```

3.2 Fonctions génériques

Fonctions utilisant des paramètres avec des pointeurs génériques.

3.2.1 Paramètres représentant des objets génériques

Quand on veut manipler des objets inconnus

Si on veut faire une fonction qui copie n ø depuis une @ `source` vers une @ `cible` et renvoie un pointeur sur la zone de départ :

```
void * copie( void * cible , void * source , int n)
{
 void * depart = source ;
 while( n-- ) *cible++ = *source++ ;
 return(depart) ;
}
```

Mais cette fonction est incorrecte car `source` et `cible` sont des pointeurs `void *` et donc les instructions `*cible++` et `*source++` sont illégales. Pourtant le but de cette fonction est de faire l'affectation

```
*cible++ = *source++ ;
```

Il faut donc abandonner le type générique `void *` pour un type qui permet de transférer une zone octet par octet → utilisation du type `char` = 1 ø. D'où la version correcte :

```
extern
char  * copie( char * cible , char * source , int n )
{
  char * d = source ;
  while( n-- ) *cible++ = *source++ ;
  return d ;
}
```

Alors quel rapport avec les pointeurs génériques ? Réponse : dans son utilisation.
Supposons que `copie` ait été compilées à part. Pour l'utiliser il faut la déclarer pour qu'elle puisse fonctionner avec n'importe quel type (`int` , `float` ...). D'où l'utilsation du type générique `void *` dans la déclaration de la fonction :

```
extern void  * copie( void * cible , void * source , int n )  ;
```

Utilisation de la fonction :

`test_copie.c`

On peut vérifier que tout est correct :

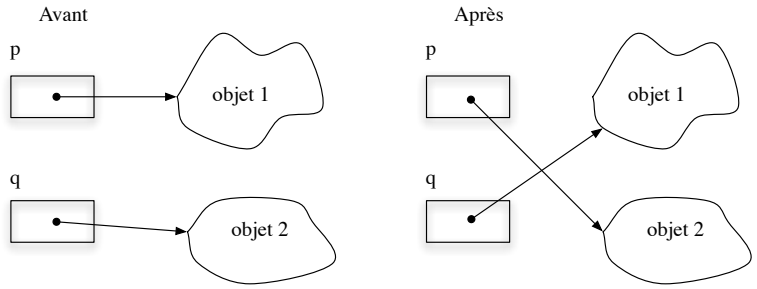
— dans les retours de la fonction : `void *` est compatible à n'importe quel autre pointeur

— dans le passage des paramètres : n'importe quel pointeur est compatible avec `void *` de la signature.
Pour les `++` comme on le fait sur des `char` et que ce type est la plus petite zone mémoire adressable alors ça marche : on dit que le type `char` à la plus petite contrainte d'alignement que les autres types.

3.2.2 Paramètres représentants des pointeurs sur des objets génériques

Quand on veut manipuler des pointeurs sur des objets inconnus

Exemple de la fonction de l'échange



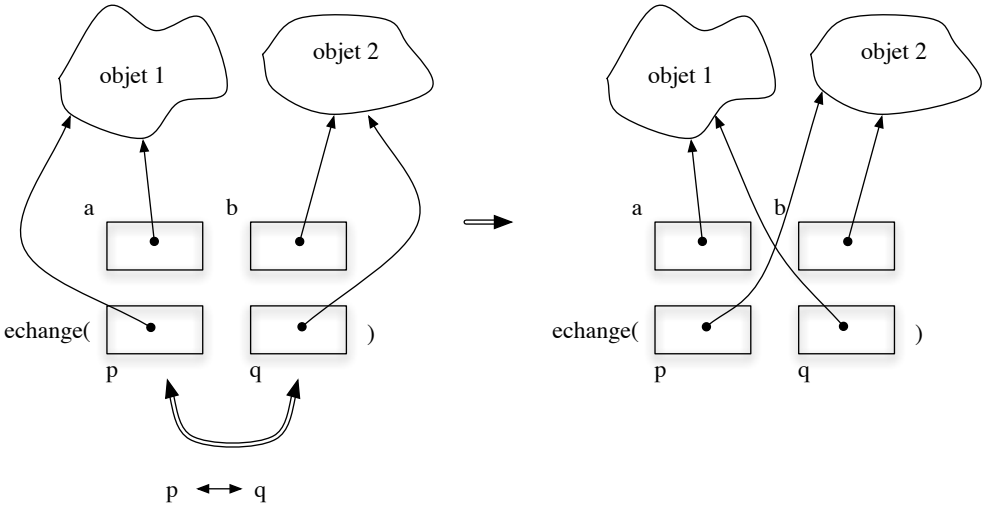
Le principe d'échanger 2 pointeurs est moins coûteux que d'échanger 2 objets
On serait alors tenté de faire pour la fonction `echange`

```
void echange( void * p , void * q )
{
 void * w ;
 w = p ; p = q ; q = w ;
}
```

Cette fonction compile bien mais est incorrecte car le passage des paramètres s'effectue par valeur. Si on est dans le cas d'utilisation

```
int x , y ;
int * a = &x ;
int * b = &y ;
...
echange( a , b ) ;
```

Dans `echange` on va échanger `p` et `q` qui sont respectivement les copies de `a` et `b`.



dans ce cas, `a` et `b` ne sont pas échangés. Il faut donc passer les @ de `a` et `b` . La fonction devient donc :

```
void echange( void ** p , void ** q )
{
        void *w ;
        w = (*p) ; (*p) = (*q) ; (*q) = w ;
}
```

`echange_generique.c`

3.3 Etude de cas : tri d'un tableau

Exemple du tri générique d'un tableau : on veut trier les éléments génériques du tableau sans savoir donc, leur représentation.

3.3.1 Algorithme

Supposons que l'on prenne un tri facile à programmer comme le "tri bulles" avec

T : tableau à trier

n : nombre d'éléments du tableau

```
tri( T , n )
Début
 i <— 0
 TQ i < n−1 FRE
    SI T[i] < T[i+1] ALORS
       T[i] <—> T[i+1]
       i <— 0
    SINON
       i <— i+1
    FSI
 FTQ
Fin
```

3.3.2 Réalisation en C

```
void tri( void * T , int n )
{
 int i = 0 ;
 void * w = NULL ;
 while( i<n−1 )
 {
    if( T[i] < T[i+1] )
    {
      w = T[i] ; T[i] = T[i+1] ; T[i+1] = w ;
      i = 0 ;
    }
    else
      i++ ;
 }
}
```

Problème 1

C'est celui de la comparaison des objets : comme on se sait pas comment ils sont réalisés, on se sait pas comment les comparer. Il nous faut donc une fonction de comparaison que l'on doit passer en paramètre → pointeur de fonction.
Supposons que l'on ait une fonction de comparaison qui ait les caractéristiques suivantes :

- a 2 paramètres : `e1` et `e2`
- renvoie une valeur
  - < 0 si `e1` $<$ `e2`
  - $== 0$ si `e1` $==$ `e2`
  - > 0 si `e1` $>$ `e2`

La fonction devient donc :

```
void tri( void * T , int n , int (*comparer)(void * , void *))
{
 int i = 0 ;
 void * w = NULL ;
 while( i<n-1 )
 {
   if( (comparer(T[i] , T[i+1]) < 0 )
   {
     w = T[i] ; T[i] = T[i+1] ; T[i+1] = w ;
     i = 0 ;
   }
   else
     i++ ;
 }
}
```

Si on a un tableau d'entiers alors on pourrait utiliser :

```
int comparer_int( int e1 , int e2 )
{
        return(e1-e2) ;
}
...
tri( T , n , comparer_int ) ; --> KO
```

mais comme la fonction qui compare dans `tri` a des pointeurs génériques en paramètre, il faut que `compare_int` ait elle aussi des pointeurs sur les objets à comparer, d'où la signature

```
int comparer_int( int * e1 , int * e2 )
{
        return((*e1)-(*e2)) ;
}
...
tri( T , n , comparer_int ) ; --> OK
```

Problème 2

Celui de l'échange des objets : ne connaissant pas leur taille on ne peut les déférencer et faire de l'arithmétique de pointeurs dessus : on ne peut pas écrire `T[i]` car $\Leftrightarrow$ `*(T+i)` , c'est interdit avec les `void *` . Il faut donc utiliser une fonction d'échange générique de 2 zones mémoires ne faisant pas

appel à l'arithmétique des pointeurs
→ on peut utiliser la fonction `void echange( void * z1, void * z2)`
du paragraphe 3.2.2 mais on échange les pointeurs, pas les objets
→ si on veut échanger les objets (comme dans le tri bulle standard) alors il faut utiliser la fonction
`void * copie( void * cible , void * source , int n)`
du paragraphe 3.2.1. Dans ce cas il faut ajouter dans les paramètres de `tri` la taille des éléments à trier. La signature devient donc :

```
void tri( void * T, int n, int (*comparer)(void * , void *), int taille )
```

Solution complète

— solution qui marche mais compilation avec des Warnings

`tri_generique_perso_1.c`

— pour compilation sans Warning faire une fonction d'encapsulation ou de "callback"

`tri_generique_perso_2.c`

3.3.3 Réalisation avec la fonction `qsort`

Même principe mais la bibliothèque standard `stdlib.h` offre des tris plus performants. Donc pas besoin de faire la fonction de tri : elle existe déjà et s'appelle "qsort" (tri rapide) mais il en existe d'autres :

— `heapsort` : le tri par tas
— `mergesort` : le tri fusion

Voir cours de programmation, d'algorithmique précédents
La signature de la fonction `qsort` est la même que celle de notre fonction précédente à part quelques `const` qui garantissent que les éléments comparés n'auront pas leur valeur modifiée.

```
void qsort( void *base, /* @ début tableau */
            size_t nel, /* Nb éléments */
            size_t width, /* Taille des éléments */
            int (*compar)(const void *, const void *));
            /* Fonction de comparaison */
```

`tri_generique_qsort.c`