Pointeurs de Fonctions
et
Unions

Pointeurs de fonctions

- On peut faire des pointeurs
  pour des variables : En général pour mettre en mémoire.
- On peut manipuler des pointeurs de fonctions
  - Objectif : apporter de la généricité dans le code.

Exemple d’utilisation

- On dispose d’une fonction qui parcourt un tableau (d’entiers ou autre) et qui calcule le maximum.
- Si on veut une fonction qui calcule le minimum.
  Ou le minimum parmi les nombre pair il faudra réécrire l’intégralité du code.
- Alors que la partie du parcours est commune.
- On souhaiterai pouvoir changer le critère de sélection sans avoir à tout ré-écrire.

- Dans les langages Objets (C++, Java, python, …) d’autres mécanismes plus sûrs existent
- Ce mécanisme est nécessaire/indispensable pour faire de la généricité en C.

Syntaxe

- Pour déclarer un pointeur de fonction :

```
int (* ma_fonction)(int, int) ;
```

- Le `*` de `(* ma_fonction)` indique que c'est un pointeur de fonction.
- La fonction prendra en paramètres 2 entiers.
- La déclaration peut être faite à l'intérieur à l'extérieur d'une fonction

Utilisation d’un pointeur de fonction

- Une fois déclaré, on peut faire pointer la fonction vers une fonction existante :
- On a une fonction `max` qui renvoie le maximum entre 2 entiers.
- On peut l’appeler directement :

```
printf(‘’Max(1,2):%d\n’’, max(1,2)) ;
```

- Ou :

```
ma_fonction = &max ;
printf(‘’%d\n’’, ma_fonction(1,2)) ;
```

Remarques

- Un pointeur de fonction n’est pas une définition de fonction.
- La déclaration du pointeur de fonction définit :
  - Son type de retour.
  - Le type et le nombre des paramètres.
- Toute fonction avec les même propriété peut être pointé par notre pointeur de fonction.
- Le paramètre d’une fonction peut être un pointeur de fonction.

Exemple

- Fonction de comparaison :
  - `int cmp( int a, int b)`
  - Qui renvoie un nombre négatif si a <b.
  - Qui renvoie 0 si a == b.
  - Qui renvoie un nombre positif si b < a.

Exemple d’utilisation

```
int extract(int * T,  int n, int (*comp)
(int,int)){
  int i, m=0 ;
  for(i=1;i<n;i++){
   if(comp(T[m],T[i]) > 0){
    m=i ;
  }
  }
  return m ;
}
```

Exemple d’utilisation (suite)

```
int Max(int a, int b) {
if(a<b){return +1 ;}else{return -1;}
}

int Min(int a,int b){
if(a>b){return +1 ;}else{return -1;}
}
```

Exemple

```
int A = extract(T,n,&max) ;
// va extraire l'indice du maximum.

Int B= extract(T,n,&min)
// va extraire l'indice du minimum.
```

Fonction `qsort` de stdlib

- La librairie stdlib fournit une fonction de tri générique qui implémente *Quicksort*
- Signature :

```
void qsort(void * Base, size_t Nmemb,
size_t Size, int (* compar)(const void
*,const void *))
```

- Permet de trier n'importe quel tableau.

qsort

- `Base` est le pointeur du tableau (`void*` car on peut mettre n'importe quel type)
- `Nmemb` : le nombre d'éléments dans le tableau
- `Size` : la taille de chacun des éléments
- Une fonction de comparaison.

Utilisation de `qsort`

- ```
  int MaxBis(int * a , int * b){
  if(*a<*b){return +1 ;}else{return -
  1;}
  }
  ```
- ```
  T=qsort(T,n,sizeof(int),&MaxBis) ;
  ```

- Exemple fPointeur.c

Unions

Unions

- Une `union` est comme un structure.
- Tous les membres de l’union partagent la même zone mémoire.
- Cela permet de faciliter l’accès à des données de bas niveau.

Déclaration

- Une `union` se déclare comme un `struct`.
- L'usage est différent.
- Exemple

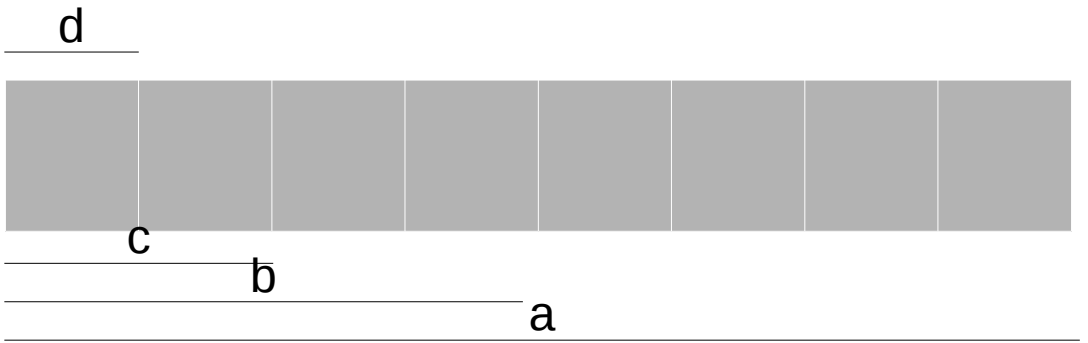
```
union Simple{
   int x ;
   float y ;
}  ;
```

Instanciation

union simple a ;
`a.x=259` ;

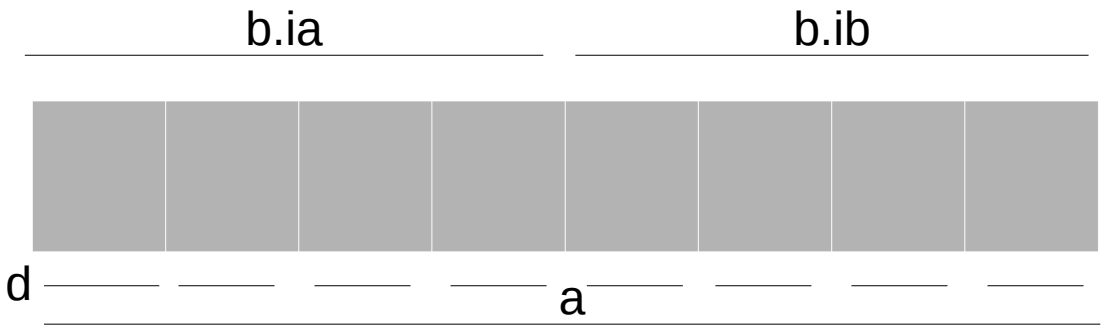
- La valeur `a.y` est automatiquement modifiée car c’est exactement la même zone mémoire.
- Ici un entier et un flottant occupent tous deux 4 octets. Donc la taille de cette union est de 4 octets.

```
union complexe {
double a ; // 8 octets
int b ;//4 octets
short int c ;//2 octets
char d ;// 1 octet
} ;
```



- La taille de l’union est celle de **l’attribut de taille maximum**.
- Pour l’`union complexe`, la taille est de 8 octets à cause de `double a`.
- l’entier `b` permet d’accéder aux 4 premiers octets ...

- Si l’on souhaite pouvoir accéder à
  tous les octets de plusieurs manière on peut utiliser des tableaux et/ou des structures :
- ```
  union X {
    double a ;
    struct {int ia ; int ib;} b ; //structure anonyme
    char d[8] ;
  } ;
  union X Instance ;
  Instance.a = 256000 ;
  Instance.b.ia = 999 ;
  Instance.d[7]=89 ;
  ```


  ```

- Les unions sont d’un usage limité.
- Utiles pour des opérations très bas niveau (modification de bit de manière individuelle).
- Il ne faut pas confondre avec les `struct` !!!

Alignements des données/*padding*

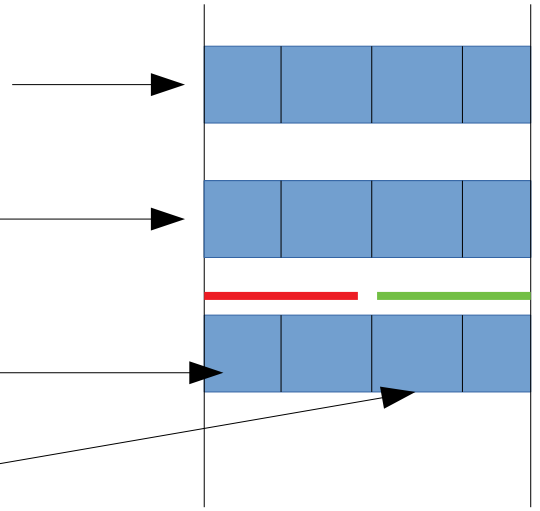
Exemple

- Hypothèse (raisonnable) :
  taille d'un `short int` : 2 octets
  taille d'un `int` : 4 octets
- Soit s une structure qui contient
  deux entiers `a` et `b` et deux entiers courts `c` et `d`
- Quelle est la taille de cette structure.

- Exemple 1 : Structure.c

```
struct s1{
   int a ;
   int b ;
   short int c ;
   short int d ;
} ;
```

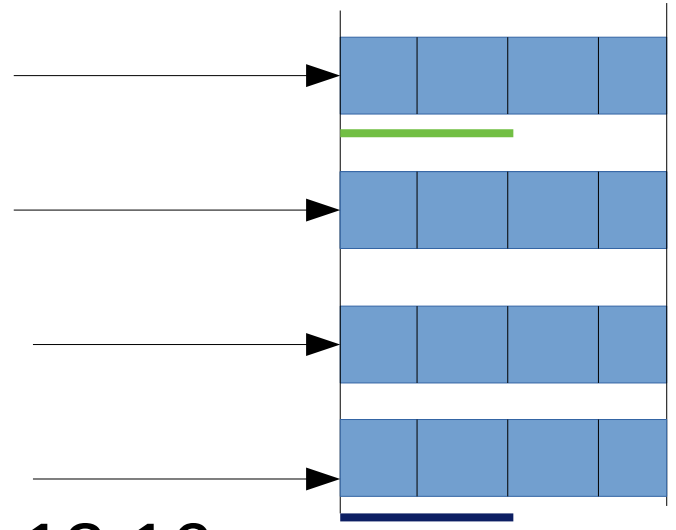
- a : 4
- b : 4
- c : 2
- d :2
- Total : 12 octets



Scénario alternatif

```
struct s2 {
   short int c ;
   int a;
   int b ;
   short int d ;
} ;
```

- c : 2
- a: 4
- b: 4
- d: 2
- Total : ~~12~~ 16
- 2+2 octets perdus



- Les variables de type `int` ou `long int`

  doivent être « alignés » sur des adresses qui correspondent à des registres. Des multiples de 4.
- L’ordre de déclarations des variables peut avoir une importance.
- Le compilateur peut possiblement changer l’ordre des attributs.

- L’ordre des attributs dans une structure peut influer sur la taille de cette dernière.
- Le problème peut être amplifié si on a besoin d’allouer un tableau de cette structures.
- Si on fait un tableau de 1000 éléments de la structure s1 : 12000 octets seront nécessaires Pas de perte
- Si on fait un tableau de 1000 éléments de la structure s2 : 16000 octets seront nécessaires 4000 octets réservés pour rien.

Problème de sérialisation

- La sérialisation d’un objet consiste à l’écrire en mémoire.
- On peut directement écrire la structure dans un fichier.

```
struct s2 instance ;
//…

fwrite(&instance, sizeof(struct s2), 1,
fd) ;
```

- Exemple 2 : Structure 3.c