Langage C
Les pointeurs de fonctions

1 Introduction

Il nous reste à aborder un élément essentiel du langage C. Supposons que nous souhaitions implémenter un algorithme de tri, par exemple le tri à bulles. Une implémentation de prototype `void tri_bulles_int(size_t len, int tab[])` nous permettra par exemple de trier un vecteur d'entiers `tab` de longueur `len`.
Mais si nous souhaitons pouvoir aussi trier un vecteur de strings `char *`, il nous faudra une seconde implémentation `void tri_bulles_strings(size_t len, char *tab[])`. La même chose se répètera pour chaque nouveau type de données : nous devrons réimplementer le même algorithme, encore et encore.
Les pointeurs de fonctions permettent d'éviter ce type d'écritures multiples – il est possible d'écrire *une seule* implémentation du tri à bulles, en ajoutant un paramètre lui permettant de s'adapter à plusieurs types de données distincts.

2 Définition de pointeur de fonctions

Rappelons que lorsque nous écrivons :

```
1 int fun(int,int);
```

cette ligne est interprétée par le compilateur comme la déclaration (le prototype) d'une fonction `fun` prenant en arguments deux `int` et renvoyant une valeur de type `int`. Ceci implique que quelque part dans le code-source se trouve une définition de la fonction `fun`.

La définition d'une variable de type « pointeur de fonctions » est à première vue un peu déroutante. Elle ressemble beaucoup à un prototype, mais avec deux éléments supplémentaires : une étoile, et des parenthèses.

type_de_retour `(*`*nom_de_variable*`)(` *liste_de_paramètres*`);`

Par exemple

```
1 int (*pfun)(int,int);
```

définit une nouvelle *variable* `pfun`. L'étoile indique que `pfun` est un *pointeur*. La forme de cette déclaration indique que `pfun` est susceptible de pointer vers *toute fonction prenant en arguments deux* `int` *et renvoyant un* `int` – toute fonction de type `int (int, int)`. Le type de `pfun` est donc : *pointeur vers* les fonctions à deux arguments en `int` et renvoyant un `int`, ou encore, pointeur vers `int (int, int)`.
Les parenthèses qui entourent `(*pfun)` sont très importantes : sans parenthèses, la ligne ci-dessus serait le *prototype* d'une fonction à deux arguments en `int` et renvoyant un `int *`. Avec les parenthèses `pfun` est bien une *variable* qui pour l'instant, n'a pas été initialisée.
Que pouvons-nous faire avec les pointeurs de fonctions ? L'exemple suivant illustre quelques-une des possibilités :

```
1  int add(int a, int b){  // une premiere fonction en int(int, int)
2    return a+b;
3  }
4
5  int mult(int a, int b){ // une seconde fonction en int(int, int)
6    return a*b;
7  }
8
9  int main(void){
10   //definition d'un pointeur vers les fonctions en int(int,int)
11   int (*pfun)(int, int);
12
13   pfun = add;           // pfun pointe a present vers add
14   int k = pfun(6, 7);   // appel de add via pfun
15
16   pfun = mult;          // pfun pointe a present vers mult
17   int l = pfun(5, 6);   // appel de mult via pfun
18 }
```

Aux lignes 13 et 16, la variable `pfun` est (ré)affectée avec à droite du symbole d'affectation (`=`) un nom de fonction. Après l'affectation, on dit que `pfun` *pointe* vers cette fonction. Le prototype de la fonction doit correspondre au type de `pfun` : il s'agit dans chaque cas d'une fonction prenant en arguments deux `int` et retournant un `int`.
La deuxième chose que l'on peut faire, c'est *appeler* la fonction pointée par `pfun` via ce pointeur. L'appel a la même forme que s'il s'agissait de la fonction elle-même, et aura le même effet. À la ligne 14, `pfun` pointe vers `add`, et `k` reçoit la valeur de `add(6, 7)`. À la ligne 17, `pfun` pointe vers `mult`, et `l` reçoit la valeur de `mult(5, 6)`.

2.1 Remarque sur les notations

Dans la plupart des implémentations, après l'affectation `pfun = add;`, la variable `pf` contient *l'adresse* en mémoire de la fonction `add`. Si l'on devait suivre les mêmes principes d'écriture que pour les pointeurs usuels, il faudrait plutôt écrire :

```
1 pf = &add;
2 int k = (*pf)(6,7);
```

puisque l'opérateur `&` retourne l'adresse d'un objet, et donc `&add` vaut l'adresse en mémoire de la fonction `add`. De même, pour accéder à l'objet pointé par un pointeur, il faut utiliser l'opérateur `*` (star), donc `(*pf)` est bien la fonction pointée par `pf`. En fait, les *deux* notations sont autorisées en C, et leurs effets sont strictement équivalents. La première, celle de notre premier exemple, à l'avantage de la simplicité, mais il s'agit d'un simple raccourci d'écriture : le sens *réel* des instructions est en fait celui de la seconde notation.

2.2 Vecteurs de pointeurs de fonctions, pointeurs de fonctions comme paramètre de fonctions

Un pointeur de fonctions peut être argument d'une fonction (c'est en fait son utilisation principale), et une fonction peut retourner un pointeur de fonction. Il est aussi possible

de définir des vecteurs de pointeurs de fonctions. Les structures peuvent aussi avoir des champs de type pointeur de fonctions.
Vu la notation assez particulière pour les pointeurs de fonctions, définir d'autres objets composés de pointeurs de fonctions peut donner lieu à des définitions difficiles à déchiffrer. Un exemple (déjà difficilement lisibile), celui d'un fonction prenant en argument un pointeur de fonctions et un `int`, et retournant un pointeur de fonctions :

```
int (*f (int (*pfun) (int, int), int n)) (int, int);
```

Imaginez ce que peut être le prototype d'une fonction prenant en argument un *vecteur de* pointeurs de fonctions et un `int`, et retournant un pointeur de fonctions ...

Simplification des notations par `typedef` . On peut simplifier les déclarations utilisant des pointeurs de fonctions en définissant des alias pour leurs types, à l'aide de `typedef`. Dans l'exemple ci-dessous, `bin_fun` devient un alias pour le type "pointeur vers les fonctions en `int (int, int)`".

```
typedef int (*bin_fun)(int, int) ;

/* tableau de pointeurs de fonctions initialisé avec deux éléments*/
bin_fun tab[] = {add, mult};

// tab[0] pointe vers add
// tab[1] pointe vers mult

int k = tab[0](3, 4); // equivalent a : int k = add(3, 4)
int l = tab[1](5, 6)/ // equivalent a : int l = mult(5, 6)
```

Sans le `typedef`, il faudrait écrire :

```
int (*tab[])(int,int) = {mult, add};
```

ce qui est à la fois difficile à déchiffrer et à comprendre.

3 Les fonctions prédéfinies `qsort` et `bsearch`

La bibliothèque standard C contient deux fonctions `qsort` et `bsearch` qui prennent en paramètres des pointeurs de fonctions. La fonction `qsort` implémente l'algorithme du *tri rapide* (quick sort) pour trier un vecteur. La fonction `bsearch` implémente la recherche binaire (dichotomique) dans un vecteur trié.

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));

void *bsearch(const void *key, const void *base, size_t nel, size_t width,
              int (*compar) (const void *, const void *));
```

Les paramètres de `qsort` sont les suivants :

- `base` : un pointeur vers le premier éléments du vecteur à trier. On utilise le type de pointeur générique `void *`, car `qsort` doit être capable de faire le tri d'un vecteur dont les éléments sont de type quelconque, inconnu d'avance,

- `nel` : le nombre d'éléments du vecteur pointé par `base`,
- `width` : la taille en octets d'un élément du vecteur pointé par `base` ,
- `compar` : un pointeur de fonction qui définit *l'ordre* choisi sur les éléments de vecteur `base`, le tri séffectuant selon cet ordre.

Pour faire le tri d'un vecteur, la fonction doit pouvoir déterminer quand un élément doit être considéré comme inférieur, égale ou supérieur à un autre. C'est le rôle de la fonction de comparaison dont l'adresse sera fournie par le dernier paramètre `compar`.
La fonction `compar` prend comme arguments deux pointeurs génériques – ils ne sont pas nommés dans le prototype de `qsort` mais pour simplifier, appelons-les `x` et `y`. La fonction `compar` doit permettre de comparer les valeurs du vecteur en respectant la convention suivante, elle doit renvoyer, selon l'ordre choisi :

- une valeur < 0 si `*x` est strictement inférieur à `*y`,
- la valeur 0 en cas d'égalité,
- une valeur > 0 si `*x` est strictement supérieur à `*y`,

La fonction `bsearch` a les mêmes paramètres que `qsort` sauf le premier. Elle retourne un pointeur vers un élément du vecteur égal (selon la fonction de comparaison) à la valeur pointée par `key`, ou `NULL` si un tel élément est introuvable.

3.1 Tri d'un vecteur de `int` par `qsort`

Pour illustrer l'usage de `qsearch`, commençons par trier un vecteur de `int` dans par ordre croissant, suivant l'ordre naturel pour les entiers. On commence par écrire la fonction de comparaison, qui doit avoir la même signature que le pointeur de fonction `compar` de `qsearch`.

```
int cmp_int(const void *pa, const void *pb){
 int a = *(int *)pa;
 int b = *(int *)pb;
 if(a < b)
   return -1;
 if(a == b)
   return 0;
 return 1;
```

Les pointeurs `pa` et `pb` pointent vers les données à comparer. Dans notre exemple, ces paramètres sont les adresses de deux `int` donc, pour retrouver le type correct de ces deux pointeurs, il faut effectuer les conversions explicites `(int *)pa` et `(int *)pb`. De plus, pour retrouver les valeurs `int` pointées, il faut encore ajouter à gauche des pointeurs convertis l'opérateur `*`. Autrement dit, `*(int *)pa` et `*(int *)pb` sont les deux `int` à comparer [1].
Voici la manière dont on peut trier un vecteur d'`int` :

```
int t[] = {4, -8, 33, 43, -22, 7, 8, -11, 99, -123, -32 };
size_t nlem = sizeof t/sizeof t[0];
qsort(t, nlem, sizeof t[0], cmp_int);
```

1. Noter que la seconde comparaison `(a == b)` ne sera atteinte que si la première n'est pas vérifiée (sinon, le premier `return` aurait été atteint et l'on serait déjà sorti de la fonction). De même, le troisième `return` ne sera atteint que si `(a < b)` est vérifié : il est donc inutile de rajouter des `else` après les `if`.

Pour trier le vecteur dans l'ordre décroissant il suffit de remplacer dans l'appel de `qsort` la fonction `cmp_int` par la fonction :

```
1 int cmp_int_inv(const void *pa, const void *pb){
2   return -cmp_int(pa,pb);
3 }
```

qui inverse le signe de la valeur renvoyée par `cmp_int`

3.2 Tri d'un vecteur de chaînes de caractères par `qsort`

Le titre de cette section est trompeur. Selon la terminologie adoptée dans le poly sur les chaînes de caractères (strings) il faudrait plutôt parler du tri d'un vecteur de *pointeurs vers* des chaînes de caractères.
Considérons un vecteur à trier contenant des éléments de type `char *`, chacun pointant vers le début d'une des chaînes à trier. Pour écrire la fonction de comparaison, reprenons la fonction `cmp_int` de la section précédente. Le type `int` des lignes 2 et 3 sera remplacé par le type des éléments du vecteur, c'est-à-dire `char *` (ceci implique que `int *` sera remplacé par `char **`).
Une fois cette modification effectuée, on peut entièrement déléguer la comparaison des chaînes à la fonction `strcmp` de la bibliothèque standard :

```
1 int cmp_string(const void *mota, const void *motb ){
2   char *a = *(char **)mota;
3   char *b = *(char **)motb;
4   return strcmp( a, b );
5 }
```

Un exemple de tri :

```
1 char *mots[] = { "the", "program", "illustrates", "in", "which",
2                  "recursive", "mutex", "necessary" };
3 nlem = sizeof mots / sizeof mots[0];
4 qsort(mots, nlem, sizeof(mots[0]), cmp_string);
```

3.3 Recherche de valeur dans un vecteur trié avec `bsearch`

Soit `t` un vecteur de `int` trié suivant l'ordre défini par la fonction `cmp_int`. Pour chercher l'adresse de la valeur 7 dans ce vecteur, on peut écrire :

```
1 int m = 7;
2 int *found = bsearch( &m, t, nlem, sizeof( int ),  cmp_int );
3 if(found){
4   printf("%d est à l'indice %td\n", m, found - t );
5 }else{
6   printf("%d not found\n", m);
7 }
```

Rappelons que la différence des deux pointeurs `found - t` donne le nombre d'objets `int` entre les deux adresses, donc l'indice de l'élément retrouvé.

4 Exemple : un sélecteur d'éléments

Notre but est ici d'écrire une fonction `selector` permettant de sélectionner certains éléments d'un vecteur `tab`. L'idée est de parcourir le vecteur en appliquant à chaque élément une fonction de sélection `sel` qui retourne 1 si l'élément doit être sélectionné et 0 sinon. En s'inspirant de `bsearch` nous voulons que :

1. le type de données stockées dans `tab` ne soit pas fixé d'avance. La fonction doit pouvoir effectuer une sélection sur tout type de vecteur (vecteur d' `int`, de `char *`, de structures etc.).
2. la fonction `sel` indiquant les éléments à sélectionner soit l'un des paramètres de la fonction `selector`.

Ces conditions nous mènent de manière naturelle à un prototype de la forme :

```
void **selector(size_t nb, size_t len, void *tab,
                int (*sel)(const void *));
```

où les valeurs de paramètres attendues sont :

- `tab` : un vecteur – ou plus exactement, un pointeur vers le premier élément d'un vecteur,
- `nb` : le nombre d'éléments de `tab`,
- `len` : la taille en octets d'un élément de `tab`,
- `sel` : un pointeur vers la fonction de sélection.

La fonction de sélection pointée par `sel` a un seul paramètre : un pointeur vers une donnée (implicitement, un élément de `tab`) (rappelons que la fonction doit renvoyer 1 si l'élément doit être sélectionné, et 0 sinon)
Il reste à définir la forme du résultat de `selector`. La fonction construira un vecteur de pointeurs – appelons-le `res` – chaque pointeur pointant vers un élément sélectionné. Le tout dernier pointeur dans le vecteur construit sera le pointeur `(void *) 0` qui marquera la fin du vecteur[2]. Le vecteur `res` sera un vecteur de pointeurs génériques en `void *`
La fonction `selector` retournera un pointeur vers le premier élément du vecteur `res`. Ce pointeur sera de type `void **`, d'où le type de retour `selector`.

Voici un exemple concret d'utilisation de `selector`. Prenons comme fonction de sélection la fonction suivante :

```
int estPositif( const void *elem ){   return *(int *)elem > 0 ; }
```

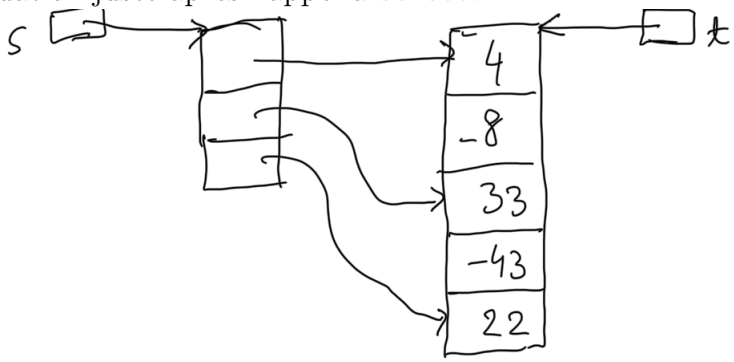
La fonction `estPositif` permet de sélectionner dans un vecteur tous les `int` strictement supérieurs à 0 (la fonction est censée prendre en paramètre un pointeur vers un `int`).
La fonction `selector` peut être appelée avec cette fonction de sélection de la manière suivante :

```
int t[] = {4, -8, 33, -43, 22};
size_t nlem = sizeof t/sizeof t[0];

int **s = (int **)selector(nlem, sizeof t[0], t, estPositif );
// le dernier élément de s est le pointeur 0
for(size_t j = 0; s[j] != (void *) 0; j++){
  printf("%d\n", *s[j] );
```

2. Ceci évite d'ajouter encore un paramètre pour récupérer le nombre d'éléments sélectionnés.

Dans la boucle `for` ci-dessus, `s[j]` est le j-ième élément du vecteur dont l'adresse `s` est renvoyée par `selector`. Mais les éléments de ce vecteur sont des pointeurs vers `int` donc il faut appliquer l'opérateur `*` pour accéder aux valeurs `int` sélectionnées.
Le dessin suivant montre la situation juste après l'appel à `selector`.



Pour conclure, voici une implémentation possible de la fonction `selector` :

```
void **selector(size_t nb, size_t len, void *tab,
     int (*sel)(const void *)){
  char *cur = tab;
  size_t i;
  unsigned n = 0;
  /* compter le nombre d'éléments qui passent le test de sélection */
  for(i = 0 ; i < nb ; i++){
    if( sel( cur ) )
      n++;
    cur += len;
  }
  /* res : le vecteur de résultat */
  void **res = malloc( sizeof( void *[n+1] ) );
  if( res == NULL )
    return NULL;
  n = 0;
  /* remplir le vecteur de résultat */
  cur = tab;
  for(i = 0 ; i < nb ; i++){
    if( sel( cur ) )
      res[n++] = cur;
    cur += len;
  }
  /* terminer le vecteur de résultat par le pointeur 0 */
  res[n] = (void *) 0;
  return res;
}
```

5 Priorité des opérateurs

Nous savons que $a + b * c$ s'évalue comme $a + (b * c)$ et non pas comme $(a + b) * c$ parce que $*$ a une priorité plus élevée que $+$.
Pour les opération qui ont la même priorité il faut préciser s'il sont évaluées de gauche à droite ou de droite à gauche. Par exemple est-ce que `a->b->c` c'est `(a->b)->c` (l'évaluation de gauche à droite) ou `a->(b->c)` (l'évaluation de droite à gauche). Bien sûr parfois de gauche à droite ou l'inverse donne le même résultat comme dans `(a+b)-c = a+(b-c)`, mais ce n'est pas forcement le cas pour tous les opérateurs.
Le tableau suivant donne les priorités et l'ordre d'évaluation pour tous les opérateurs du langage C. Les opérateurs dans la même case ont la même priorité et les priorités augmentent vers le haut du tableau.
La colonne **associativité** indique l'ordre d'évaluation (gauche -> droit ou droit -> gauche).
La colonne **arité** donne le nombre d'arguments d'un opérateur.
Notez que `*` apparaît deux fois, comme l'opérateur de multiplication `a*b` avec arité 2 et comme opérateur à un seul argument devant un pointeur, `*p` où `p` un pointeur.

opérateur	associativité	arité
`()` `[]`	GD	
`->` `.`	GD	2
`!` `~` `++` `--` `+` `-`	DG	1
`*` `(type)` `sizeof`	DG	1
`*` `/` `%`	GD	2
`+` `-`	GD	2
`<<` `>>`	GD	2
`<` `>` `>=` `<=`	GD	2
`==` `!=`	GD	2
`&`	GD	2
`|`	GD	2
`&&`	GD	2
`||`	GD	2
`?` `:`	DG	3
`=` `+=` `*=` `/=` `%=`	DG	2
`>>=` `<<=` `&=` `^=` `|=`	DG	2
`,`	GD	2

Exemple.
On déchiffre :

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

Commençons par `(*signal(...))` : la priorité de `( )` est supérieure à celle de `*` et `( )` indique que `signal` est une fonction. Maintenant on applique `*`, donc c'est ne fonction qui retourne un pointeur.
Quel pointeur ? `void (*signal())(int)` donc `signal` renvoie un pointeur de fonction qui prend un `int` et qui renvoie `void`.
Il reste à déchiffrer les paramètres de `signal` : `(int sig, void (*func)(int))` – il en a deux, le premier `int sig` est un `int`, le deuxième paramètre de `signal` est `void (*func)(int)`. Ce paramètre s'appelle `func` est c'est un pointeur : les parenthèses autour de `*func`

indiquent que `*` s'applique premier. Ensuite on applique `(int)`, donc `func` est un pointeur de fonction qui prend un argument `int` et retourne `void`.