Wieslaw Zielonka
zielonka@irif.fr

pointeurs de fonctions

pointeurs de fonctions

A quoi servent les pointeurs de fonctions ?

On implemente un algorithme de tri (par exemple le tri par insertion).

C'est le même algorithme pour trier :

- un tableau de int
- un tableau de double,
- le même tableau de double mais dans l'ordre décroissant
- un tableau de pointeurs vers des chaînes de caractères à trier dans l'ordre lexicographique (ou son inverse),
- un tableau de structures etc. etc.

Ce qui change chaque fois :

1. le type de données à trier et
2. la relation d'ordre sur les données

A quoi servent les pointeur de fonctions ?

Une solution pour ne pas réécrire la reimplémenter la fonction de tri :

A. implémenter l'algorithme de tri une seule fois et

B. faire en sorte que la fonction qui compare deux éléments a, b pour déterminer lequel est plus petit que l'autre devient un paramètre de l'algorithme

A quoi servent les pointeur de fonctions ?

Comment on résout ce problème en java ?

Comment on évite de reimplementer 100 fois le même algorithme de tri ?

Dans la classe `Arrays` :

```
static <T> void sort(T[] a, Comparator<? super T> c)
```

`Comparator` : l'interface qui contient la méthode

```
int compare(T o1, T o2)
```

il suffit d'implémenter `Comparator` avec la méthode `compare()` adaptée pour le problème et une seule implémentation de l'algorithme `sort()` suffit pour trier le vecteur d'éléments de n'importe quel type.

Solution en C : faire passer une fonction qui compare les éléments comme un paramètre de la fonction de tri

L'algo de tri implémenté une fois, mais la fonction qui définit l'ordre d'éléments change et est différente pour chaque type de données et pour chaque relation d'ordre.

Comment est implémenté un appel de fonction ?

Compilation de fichier `*.c` :

`gcc -S file.c`

produit un fichier assembler `file.s` :

```
/* file.c */

int f(int a){ ....

}

int g(int b){  ......

int x = f( y* y );

}
```

```
_f :



_g :



callq  _f
```

Chaque fonction devient en bloc de code étiqueté par le nom de la fonction (avec _ ajouté en préfixe). L'appel à la fonction f() devient l'instruction `callq _f,` c'est un saut à l'adresse de la première instruction de f() (un saut un peu spécial - il sauvegarde l'adresse de retour dans un registre spécial). La préparation de paramètres de f() précède `callq _f.` Donc pour faire l'appel à la fonction `f()` il suffit

- savoir préparer les paramètres de f() et
- connaître l'adresse de la première instruction de `f().`

pointeur de fonctions

Intuitivement, un pointeur de fonction c'est l'adresse de la première instruction de la fonction.

Définir une variable de type pointeur de fonction :

```
type_ret (*pf)(type1, type2, typeN);
```

`pf :` une variable (non-initialisée) de type pointeur de fonction. Comme pour d'autres pointeurs, ce pointeur a un type fixe. Une fonction peut être utilisée comme valeur de ce pointeur si elle a les paramètres de types `type1, type2, typeN` et si elle retourne une valeur de type `type_ret`.

Notez la position du nom `de la variable pf` dans la définition.

pointeurs de fonctions

```
int (*pf)(int, int);
```

`pf :` la variable de type pointeur de fonction qui prend deux valeur int et retourne int.

Notez la différence avec

```
int *fun(int, int);
```

`fun :` est un prototype de fonction qui prend deux `int` et retourne `int *`

`fun` n'est pas une variable mais un nom d'une fonction existante,

`pf` est une variable (non-initialisée).

pointeur de fonctions

```
int add(int a, int b){                  // add, mult deux fonction
  return a+b;                           // (int, int)   ——→ int

}

int mult(int a, int b){
  return a*b;
}

int main(void){
  int (*pf)(int, int);  /* pf : variable pointeur
                         * de fonction  de (int, int)  -->  int
                         * non-initialisée
                         */

pf = add;              /* pf reçoit l'adresse de la fonction add */

int k = pf(5,6);     /* l'appel de la fonction dont l'adresse
                         * se trouve dans pf, donc on appelle add()  */

pf = mult ;

int l = pf(5,6); /* ici un appel à mult() */

printf("k=%d l=%d\n", k, l);  // k == 11 l == 30
```

pointeur de fonctions

Les seules opérations possibles sur les pointeurs de fonction :

- changer la valeur du pointeur en y mettant NULL ou l'adresse d'une fonction
- faire appel à la fonction dont l'adresse est stockée dans le pointeur.

C permet deux notations pour désigner ces opérations :

```
int add(int a, int b){}

int (*pf)(int, int);

/* notation version lourde*/

pf = &add;

int k = (*pf)(5,6);
```

```
int add(int a, int b){}

int (*pf)(int, int);

/* notation version légère
(préférable)*/

pf = add;

int k = pf(5,6);
```

La notation à gauche est proche de celle utilisée avec d'autre types de pointeurs, celle à droite est plus facile et plus naturelle.

pointeur de fonctions
désastre pour les déclarations ?

```
int add(int a, int b){ return a+b; }

int mult(int a, int b){ return a*b; }

int divi( int a, int b){ return a/b ;}

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

`tab : vecteur` de pointeurs de fonctions de type `(int,int) -> int`

Question:

comment fabriquer les définitions de types compliquées qui utilisent les pointeurs de fonctions?

pointeur de fonctions
simplifier les définitions avec typedef

```
int add(int a, int b){ return a+b; }

int mult(int a, int b){ return a*b; }

int divi( int a, int b){ return a/b ;}
```

Définir le type : pointeur de fonction (int, int) -> int :

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

Ressemble à la définition de variable de type pointeur de fonction mais précédé par `typedef`.

Tableau d'éléments de type pfun initialisé devient :

```
pfun  tab[] = { add, mult, divi };

// tableau d'éléments de type pfun initialisé avec 3 éléments

int k = tab[1](3,6); /* multiplier 3 et 6*/
```

pointeurs de fonctions
quelques exemples de la bibliothèque standard

L'algorithme quick sort :

```
void qsort(void *base,   /*l'adresse du premier element*/
           size_t nel,   /*nombre d'elements*/
           size_t width, /*taille d'un element*/

           /* pointeur de fonctions qui compare
            * deux éléments */
           int (*compar)(const void*, const void *))
    )
```

Notez que `compar` prend comme paramètres les pointeurs vers les données, pointeurs vers les éléments à comparer.

`compar(a,b)` retournera `< 0` quand `*a < *b`

`0` quand `*a == *b`

`>0` quand `*a > *b`

Exemple : qsort

```
int main(void){
  int t[] = {4, -8, 33, 43, -22, 7, 8, -11, 99, -123, -32 };
  size_t nlem = sizeof t/sizeof t[0]; /*nombre d'elements de t*/
  /* faire le tri de tableau t de int  */
  qsort(t, nlem, sizeof t[0], cmp_int);

  char *mots[] = { "the", "program", "illustrates", "in", "which",
                   "recursive", "mutex", "necessary" };
  nlem = sizeof mots/sizeof mots[0]; /* nombre d'éléments de mots[] */
  /* faire le tri de tableau mots de strings */
  qsort(mots, nlem, sizeof(mots[0]), cmp_string);
```

Il reste à écrire les fonctions cmp_int et cmp_string.

Exemple : qsort

La fonction cmp_int de comparaison pour les int.

Les paramètres sont **des pointeurs vers les données.**

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

pa - pointeur vers les données, c'est-à-dire l'adresse d'un int.

(int *)pa : retypage pour retrouver le bon type de pointeur.

***(int *) pa : pour retrouver int qui est à l'adresse pa**

Exemple : qsort

La fonction `cmp_string` de comparaison pour les strings.

`Est-ce que` cmp_string c'est tout simplement la fonction du C standard

int strcmp(const char *str1, const char *str2)

NON : cmp_string doit prendre en paramètre les pointeurs vers (char *) tandis que strcmp prend comme paramètres deux (char *)

Exemple : qsort

```
int cmp_string(const void *mota, const void *motb )
{

   return strcmp( *(char **)mota, *(char **)motb);

}
```

mota est un pointeur vers les données char *, donc

(char **) mota : donne le "vrai" pointeur vers les données

*(char **) mota : donne le string, i.e. un objet de type char *

Exemple : sort

Et maintenant, si on veut trier le tableau dans l'ordre inverse? Il suffit d'écrire de nouvelles fonctions de comparaison: qui inversent les arguments :

```
int cmp_int_inverse(const void *pa, const void *pb){

 return cmp_int(pb, pa);
}

int cmp_string_inverse(const void *pa, const void *pb){

  return cmp_string(pb,pa);
}
```

Exemple : implémenter l'insertion dans une liste triée générique avec la tête

```
struct elem{
  struct elem *suivant;       /* pointeur vers le suivant */
  void *val;                  /* pointeur vers les donnees */
};
typedef struct elem elem;
typedef struct elem *liste;

liste inserer_ordre(liste l, void *data,
                        int (*cmp)(const void *, const void *)){
  liste l = NULL;
  liste cour = l->suivant;
  /* comparer la valeur de cour avec data */
  while( cour != NULL && cmp( cour -> val, data) < 0 ){
    prec =cour;
    cour = cour -> suivant;
  }

  return inserer_apres( prec, data );     /* ajouter après prec */

}
```

pointeur de fonctions quelques exemples de la bibliothèque standard

```
void *bsearch(void *base, /*l'adresse du premier element*/
              size_t nel,   /*nombre d'elements*/
              size_t width, /*taille d'un element*/
              int (*compar)(const void*, const void *))
      )
```

recherche binaire dans un tableau trié.

Le paramètre `compar de qsort() et bsearch()` prend comme paramètres les pointeurs vers les données (les pointeurs vers les éléments à comparer)

`compar(a,b)` retournera `< 0 quand  *a < *b`

`0   quand  *a == *b`

`>0   quand  *a > *b`