POINTEURS DE FONCTIONS

INSPIRÉ DU COURS DE G. BIANCHI, UNIVERSITÉ DE BORDEAUX

Vincent Aranega

PROBLÈMES

- Comment faire pour éviter la "copie" de comportement ?
- Comment faire pour choisir entre plusieurs alternatives en limitant les if/switchs ?

PROBLÈMES: EXEMPLE 1

```c
void tri_croissant(int *tab, int length)
{
    int i, j;
    for (i = length - 1; i > 0; i++) {
        for (j = 0; j < i - 1; j++) {
            if (tab[j] > tab[j + 1]) {
                int temp = tab[j];
                tab[j] = tab[j + 1];
                tab[j + 1] = temp;
            }
        }
    }
}
```

Si tri décroissant ?

PROBLÈMES: EXEMPLE 2

```c
void tri_decroissant(int *tab, int length)
{
    int i, j;
    for (i = length - 1; i > 0; i++) {
        for (j = 0; j < i - 1; j++) {
            if (tab[j] < tab[j + 1]) {  // Seule modification...
                int temp = tab[j];
                tab[j] = tab[j + 1];
                tab[j + 1] = temp;
            }
        }
    }
}
```

PROBLÈMES: EXEMPLE 2

```c
void a(){ printf("You choose 0\n"); }
void b(){ printf("You choose 1\n"); }
// c(), d() ...etc

int main(void) {
    int choice = -1;
    scanf("%d", &choice);
    switch (choice) {
      case 0:
        a();
        break;
      case 1:
        b();
        break;
// ..etc
    }
}
```

POINTEUR DE FONCTION

- Permet de référencer directement une fonction
- Permet de passer une fonction en paramètre d’une autre
- Permet d’associer une fonction à une structure

POINTEUR DE FONCTION

- comment récupérer l'@ d’une fonction ?
- Comment déclarer un pointeur de fonction ?
- Comment appeler une fonction référencée ?
- Peut-on définir un tableau de pointeurs de fonction ? Comment ?
- Comment accepter un pointeur de fonction en paramètre ?

RAPPELS SUR LES POINTEURS

- Toute variable est stockée soit dans la pile, soit dans le segment de données (.data ou .bss)
- Ce stockage lui confère une adresse propre qui peut être stockée dans une variable de type pointeur
- Un pointeur = adresse mémoire + type cible
  - `int * p_int`
  - `float * p_float`
  - `char * p_char`

RAPPELS

- Le code du programme est stocké dans une zone mémoire appelée segment de code (.text)
- Toute fonction, comme tout autre objet du programme, a donc également une adresse mémoire
- Pour la manipuler, il faut connaître son "type", comme pour une variable

RÉCUPÉRATION DE L'ADRESSE D'UNE FONCTION

- utilisation temporaire du pointeur générique `void *`

```
void foo(void) {
    print("foo\n");
}

int main(void) {
    void * ptr_foo = &(foo);
    void * ptr_foo2 = foo;
    printf("@foo:%p - %p\n", ptr_foo, ptr_foo2);
    return 0;
}
```

- le nom de la fonction correspond à un pointeur statique vers celle-ci
- l'utilisation simple du nom de la fonction conduit à son évaluation, donc & ( ) est optionel, mais recommandé (lisibilité)

RÉCUPÉRATION DE L'ADRESSE D'UN FONCTION

- Soucis pointeur générique
  - bien que gérée par certain compilateur, n'est pas portable
  - il n'est pas possible de déréférencer un pointeur générique car sans type cible, il ne connaît pas la taille mémoire à récupérer
  - on ne peut donc pas non plus appliquer d'arithmétique sur ce type de pointeur (pas un soucis ici)

TYPE D'UN POINTEUR DE FONCTION

- Un pointeur de fonction a un type précis dépendant du prototype de la fonction pointée
- Ce type est défini à l'aide du type de retour et au nombre ainsi qu'aux types des arguments de la fonction
- La déclaration d'un pointeur de fonction s'effectue ainsi:

```
type_retour (* nom_variable)(l_types_fct);
// l_types_fct est la liste des types des paramètres de la
// fonction vers laquelle on veut pointer
```

EXEMPLES

- Déclaration de pointeurs de fonctions ne retournant rien

```c
void foo_1(void) {...}
void foo_2(int i){...}
void foo_3(int i, char * s, size_t l){...}

int main(void) {
    void (*ptr_1)(void);
    void (*ptr_2)(int);
    void (*ptr_3)(int, char *, size_t);
    ptr_1=&(foo_1);
    ptr_2=&(foo_2);
    ptr_3=&(foo_3);
    printf("%p - %p - %p\n",ptr_1,ptr_2,ptr_3);
    return 0;
}
```

EXEMPLES

- Déclaration de pointeurs de fonctions ne retournant un résultat

```
int foo_1(void) {...}
char * foo_2(int i){...}
float foo_3(int i, char * s, size_t l){...}

int main(void) {
    int (*ptr_1)(void);
    char * (*ptr_2)(int);
    float (*ptr_3)(int, char *, size_t);
    ptr_1=&(foo_1);
    ptr_2=&(foo_2);
    ptr_3=&(foo_3);
    printf("%p - %p - %p\n",ptr_1,ptr_2,ptr_3);
    return 0;
}
```

RAPPEL: APPEL DE FONCTION

- L'appel d'une fonction s'effectue simplement en utilisant son nom suivi entre parenthèses des arguments
- Le compilateur va remplacer cet appel par la demande de l'exécution du code correspondant à la fonction (à l'aide de son adresse) et en positionnant sur la pile l'évaluation des arguments et variables locales

RAPPEL: APPEL DE FONCTION

```
int foo(int i){
    return i*2;
}
int main(void){
    int j=1;
    int k=0;
    k=foo(j);  // Le compilateur remplace ça par un "call"
               // de l'@ de foo en asm
    return 0;
}
```

APPEL DE FONCTION

- Il est possible de procéder de manière similaire à l'aide d'un pointeur de fonction

```
(* nom_ptr)(arg0, arg1, …);
```

```c
void foo_1(void){ printf("foo_1\n"); }
void foo_2(int i){ printf("foo_2:%d\n",i); }
int foo_3(int i, char * s, size_t l){
    printf("foo_3:%d,%s,%d\n",i,s,l);
    return 1;
}
int main(void){
    void (*ptr_1)(void)=&(foo_1);
    void (*ptr_2)(int)=&(foo_2);
    int (*ptr_3)(int, char *, size_t)=&(foo_3);
    int return_value;
    (*ptr_1)();
    (*ptr_2)(2);
    return_value = (*ptr_3)(2,"Hello",5);
    printf("Return value: %d\n", return_value);
    return 0;
}
```

TABLEAU DE POINTEURS DE FONCTIONS

- Comme un pointeur de fonction est une adresse, il est possible d'en stocker plusieurs dans un tableau
- A l'instar de la déclaration d'un tableau d'un type "simple" (e.g., `int tab[TAILLE]`), la déclaration d'un tableau de pointeurs de fonctions se fait généralement sous la forme suivante :

```
type_retour (* tab[TAILLE])(l_types_fct);
```

- L'écriture est très similaire à la déclaration d'un pointeur de fonction, c'est l'ajout des [] et de la taille qui fait toute la différence

TABLEAU DE POINTEURS DE FONCTIONS

- Ces tableaux sont soumis aux mêmes règles d'utilisation que les tableaux "standards":
  - indice commence à `0`
  - un tableau ne connait pas sa taille
  - Tous les éléments contenus (ici des pointeurs) doivent être de même type – les fonctions dont on stockera les adresses devront avoir un prototype commun (i.e. même type de retour et mêmes types et ordre des arguments)

RETOURNER UN POINTEUR DE FONCTION

- Il peut être souhaitable de retourner un pointeur de fonctions
- Les prototypes de la fonction renvoyant le pointeur (disons `foo`) et du type de fonctions pointées s'entremêlent ainsi :

```
type_ret_fct (* foo(l_args_foo))
             (l_types_fct);
```

- où la fonction `foo` prend en paramètre une liste d'arguments (i.e. `l_args_foo`) et retourne un pointeur vers des fonctions dont le prototype est `type_ret_fct fct(l_types_fct);`

EXEMPLE

```
void foo_1(void){ printf(", World!\n"); }
int foo_2(char * s){ return printf("%s", s); }
void (* foo_3(int i))(void){
    printf("%d\n", i);
    return &(foo_1);
}
int (* foo_4(int i))(char *){
    printf("%d\n", i);
    return &(foo_2);
}
int main(void){
    void (*ptr_1)(void)=foo_3(1);
    void (*ptr_2)(char *)=foo_4(2);
    (*ptr_2)(”Hello”);
    (*ptr_1)();
    return 0;
}
```

Que ce code affiche-il ?

UTILISATION D'UN TYPEDEF

- Alléger la syntaxe en déclarant un nouveau type

```
typedef void (*t_ptrf_1)(void);

void foo_1(void){ ... }

t_ptrf_1 foo_2(int i){
    printf("%d\n",i);
    return &(foo_1);
}

int main(void) {
    t_ptrf_1 ptr_foo = foo2(1);
    (*ptr_foo)();
    return 1;
}
```

- Masque la nature réelle du type

POINTEUR DE FONCTION EN PARAMÈTRE

- Pour passer un pointeur de fonctions en paramètre, il suffit de déclarer le paramètre comme une variable classique

```
void foo_1(int i, char * s){
    printf(”%s : %d\n”,s,i);
}

void foo_2(int i, char * s){
    printf(”%s - %d\n”,s,i);
}

void foo_3(int n, void bar(int, char *)){
    bar(n,"Hello");
}

int main(void) {
    foo_3(1, &(foo_1));
    foo_3(2, &(foo_2));
    return 0;
}
```

POINTEUR DE FONCTION EN PARAMÈTRE (AVEC UN TYPEDEF)

```c
typedef void(*t_ptrf)(int, char*);
void foo_1(int i, char * s){
    printf(”%s : %d\n”,s,i);
}

void foo_2(int i, char * s){
    printf(”%s - %d\n”,s,i);
}

void foo_3(int n, t_ptrf bar){
    bar(n,"Hello");
}

int main(void) {
    foo_3(1, &(foo_1));
    foo_3(2, &(foo_2));
    return 0;
}
```

RETOUR SUR L'UTILITÉ

- Nous proposons d'implémenter une simple calculatrice basée sur le cahier des charges simpliste suivant

"" Le programme permet d'effectuer un calcul simple décrit comme un argument du programme. Les opérations admises sont l'addition, la soustraction, la multiplication et la division. Un calcul est composé d'un premier opérande suivi d'une opération et d'un second opérande. Le programme affiche le résultat. ""

RETOUR SUR L'UTILITÉ: EXEMPLE

- Les opérations de calcul sont implémentées comme suit et "stockées" dans un tableau de pointeurs de fonctions

```
double add(double a, double b){
    return a + b;
}
double substract(double a, double b){
    return a - b;
}
double multiply(double a, double b){
    return a * b;
}
double divide(double a, double b){
    return a / b;
}

double (*operations[4])(double,double) = {&(add), &(substract),
    &(multiply), &(divide)};
```

RETOUR SUR L'UTILITÉ: EXEMPLE

- Le choix de l'opération est effectué ainsi

```
double (* selectOperation(char choice))(double,double){
    int i=-1;
    switch(choice){
        case '+': i=0; break;
        case '-': i=1; break;
        case 'x': i=2; break;
        case '/': i=3; break;
        default: return NULL;
    }
    return operations[i];
}
```

RETOUR SUR L'UTILITÉ: EXEMPLE

- Le main (attention des vérifications manquent pour que le code soit complet)

```
int main(int argc, char *argv[]) {
    char * pEnd=NULL;
    double a = strtod (argv[1], &pEnd);
    double b = strtod (argv[3], &pEnd);
    op=selectOperation(argv[2][0]);
    printf("%f\n",(*op)(a,b));
    return 0;
}
```

CONCLUSION

- Les pointeurs de fonctions permettent d'introduire une certaine flexibilité dans le code
- La déclaration de type, variable, argument ou tableau utilisant un pointeur de fonction doit respecter le prototype de fonction correspondante
- Les erreurs sont faciles et définir un pointeur de fonction nécessite une grande vigilance