CM 14 - POINTEUR DE FONCTIONS

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

RAPPELS

- ▷ Nous avons vu que 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;`
  - ▷ `double * p_double;`

RAPPELS

- ▷ Il existe une notion de pointeur générique en C noté `void*` compatible avec tous les autres pointeurs (i.e., à même de pointer vers n'importe quel type cible)
- ▷ ⚠Ce n'est pas un pointeur vers un objet de type `void` mais bien un type à part
- ▷ 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

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
- ▷ Afin de manipuler cette dernière, il faudra avoir connaissance du "type" de la fonction (à l'instar des variables)
- ▷ ⚠L'utilisation d'un pointeur générique pour stocker l'adresse d'une fonction, bien que gérée par certain compilateur, n'est pas portable

RÉCUPÉRATION DE L'ADRESSE D'UNE FONCTION

- ▷ De manière similaire aux variables, on peut utiliser l'opérateur `&(nom_fonction)` pour obtenir l'adresse de la fonction `nom_fonction`

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

int main(void){
    void * ptrFoo = &(foo);
    printf("@foo:%x\n",ptrFoo);
    return EXIT_SUCCESS;
}
```

```
$> gcc -std=c99 -m32 foo.c
$> ./a.out
@foo:0x804842b
$> nm a.out | grep foo
0804842b T foo
```

- ▷ GNU `nm` liste les symboles d'un fichier objet en précisant pour chacun sa valeur, son type (ici, `T` précise que le symbole est présent dans le segment `.text`) et son label (c.f. `man nm`)

RÉCUPÉRATION DE L'ADRESSE D'UNE FONCTION

- ▷ Il s'avère que le nom de la fonction correspond déjà à un pointeur statique sur cette dernière
- ▷ L'utilisation simple du nom de la fonction conduit à son évaluation comme un pointeur sur cette dernière et rend l'utilisation de `&()` optionnelle mais très fortement conseillée pour des raisons de lisibilité

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

int main(void){
    void * ptrFoo = &(foo);
    void * ptrFoo2 = foo;
    printf("@foo:%p - %p\n",ptrFoo,ptrFoo2);
    return EXIT_SUCCESS;
}
```

```
$> gcc -std=c99 -m32 foo.c
$> ./a.out
@foo:0x804842b - 0x804842b
```


TYPE CIBLE D'UN POINTEUR DE FONCTION

- ▷ ⚠L'utilisation d'un pointeur générique permet de stocker l'adresse mais pas de l'utiliser de manière portable !
- ▷ 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);
```

- ▷ où `l_types_fct` est la liste des types des paramètres des fonctions que l'on souhaite pouvoir pointer

EXEMPLES

- ▷ Déclaration de pointeurs de fonctions ne retournant rien

```
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 EXIT_SUCCESS;
}
```

```
$> gcc -std=c99 -m32 foo.c
$> ./a.out
0x80483fb - 0x8048400 - 0x8048405
$> nm | grep foo
080483fb T foo_1
08048400 T foo_2
08048405 T foo_3
```

EXEMPLES

▷ Déclaration de pointeurs de fonctions 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 EXIT_SUCCESS;
}
```

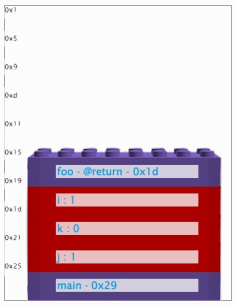
```
$> gcc -std=c99 -m32 foo.c
$> ./a.out
0x80483fb - 0x8048400 - 0x8048405
$> nm | grep foo
080483fb T foo_1
08048400 T foo_2
08048405 T foo_3
```


APPEL DE FONCTIONS

▷ 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 dans la pile l'évaluation des arguments et variables locales

```
int foo(int i){
    return i*2;
}
int main(void){
    int j=1;
    int k=0;
    k=foo(j);
    return EXIT_SUCCESS;
}
```

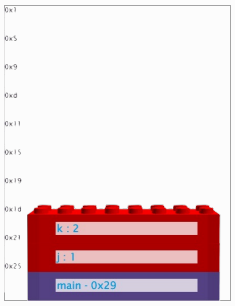


APPEL DE FONCTIONS

▷ 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 dans la pile l'évaluation des arguments et variables locales

```
int foo(int i){
    return i*2;
}
int main(void){
    int j=1;
    int k=0;
    k=foo(j);
    return EXIT_SUCCESS;
}
```



APPEL DE FONCTIONS

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

```
type_ret_fct var = (* nom_ptr)(arg0, arg1, ...);
```

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

```
$> gcc -std=c99 -m32 truc.c
$> ./a.out
foo_1
foo_2:2
foo_3:2,Hello,5
```


TABLEAU DE POINTEURS DE FONCTIONS

▷ 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"

▷ Ils sont indicés à partir de **0**

▷ Ils ne connaissent pas leur 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 FONCTIONS

▷ 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 EXIT_SUCCESS;
}
```

▷ Que ce code affiche-t-il?

UTILISATION D'UN TYPEDEF

▷ L'utilisation de `typedef` avec les pointeurs de fonctions peut permettre d'alléger la syntaxe mais masque la nature réelle du type (i.e., le fait que ce soit un pointeur)

```
typedef void (*t_ptrf_1)(void);
typedef void (*t_ptrf_2)(char *);

t_ptrf_1 foo_3(int i){
    printf("%d\n",i);
    return &(foo_1);
}

t_ptrf_2 foo_4(int i){
    printf("%d\n",i);
    return &(foo_2);
}
```

```
void foo_1(void){ ... }

int foo_2(char * s){ ... }

int main(void){
    t_ptrf_1 ptr_1=foo_3(1);
    t_ptrf_2 ptr_2=foo_4(2);
    (*ptr_1)();
    (*ptr_2)("Hello");
    return EXIT_SUCCESS;
}
```


POINTEUR DE FONCTIONS 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 EXIT_SUCCESS;
}
```

```
$> gcc -std=c99 -m32 foo.c
$> ./a.out
Hello : 1
Hello - 2
```


POINTEUR DE FONCTIONS EN PARAMÈTRE

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

```
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 EXIT_SUCCESS;
}
```

```
$> gcc -std=c99 -m32 foo.c
$> ./a.out
Hello : 1
Hello - 2
```


UTILITÉ DES POINTEURS DE FONCTIONS

▷ Les pointeurs de fonctions permettent d'apporter une flexibilité difficile à obtenir avec des appels de fonctions statiques

▷ 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. ""

LES FONCTIONS DE CALCUL

▷ 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)};
```


LE CHOIX DE L'OPÉRATION

▷ 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];
}
```


LA FONCTION MAIN

```
int main(int argc, char* argv[]){
  double (*op)(double,double);
  if((argc<4)||(strlen(argv[2])>1)){
    usage();
    exit(EXIT_FAILURE);
  }
  char * pEnd=NULL;
  double a = strtod (argv[1], &pEnd);
  if(pEnd==argv[1]){
    usage();
    exit(EXIT_FAILURE);
  }
  double b = strtod (argv[3], &pEnd);
  if(pEnd==argv[3]){
    usage();
    exit(EXIT_FAILURE);
  }
  op=selectOperation(argv[2][0]);
  if(op==NULL){
    usage();
    exit(EXIT_FAILURE);
  }
  printf("%f\n",(*op)(a,b));
  return EXIT_SUCCESS;
}
```

DOGGY BAG

Questions?

TO TAKE AWAY ...

- ▷ Le pointeur générique n'est pas d'une grande aide ici
- ▷ 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