Pratique du C
Introduction aux pointeurs

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2013-2014

Notion de pointeurs :

- la mémoire physique est vue comme une suite finie d'octets ;
- un pointeur est une variable contenant l'adresse d'une autre variable ;

```
int i = 43 ; int *p_i ; p_i = &i ;
```

- une valeur de type pointeur est une adresse mémoire ;
- un pointeur est donc un espace mémoire pouvant contenir une adresse :

label	...		i				p_i			
adresse	0	...	x	x+1	x+2	x+3	x+4	x+5	x+6	x+7
octet	...		43				x			

Déclaration d'un pointeur

- pointeur : caractérisé par le type de la variable pointée ;
- déclaration : *type_pointé* ∗*identificateur_pointeur* ; ;
- *type_pointé* : peut être d'un type quelconque ;
- la classe d'allocation de la variable pointée peut être tout sauf *register* i.e. la variable pointée peut être externe, statique, automatique (voir cours correspondant) ;
- exemples :

```
int foo ;
int *p_foo ;

short int bar  ;
short int *p_bar ;

foo=44 ; p_foo=&foo;
bar=44 ; p_bar=&bar;
```

```
        .data
         .globl foo    .size foo,4
foo:     .long   44
         .globl p_foo .size p_foo,4
p_foo:   .long   foo
         .globl bar    .size bar,2
bar:     .value  44
         .globl p_bar .size p_bar,4
p_bar:   .long   bar
```

L'opérateur &

Cet opérateur retourne l'adresse d'un objet : opérateur "adresse de"

- il est unaire `&` : adresse mémoire d'un objet ;
- il ne s'applique qu'à des objets en mémoire : variables, éléments de tableaux, fonctions ;
- Exemple : `int i, *p; p = &i;`

On utilise une constante dénomée "pointeur nul" :

```
#define NULL 0
```

et définie dans stddef.h (qui est inclus par le biais de stdio.h). C'est la convention pour une adresse invalide (lorsqu'un pointeur n'est pas initialisé par exemple).

L'opérateur *

Il s'agit du déréférencement ou encore de l'opérateur d'indirection.

- c'est un opérateur unaire * qui retourne la valeur de l'objet pointé ;
- il s'applique à un pointeur de manière préfixée ;
- l'expression de retour est de même type que la valeur pointée ;
- il peut aussi apparaître en partie gauche d'affectation (*lvalue*) ;
- Exemple :

```
int i,j, *p = &i ;
*p = 4 ; j = *p + 1 ;
```

Exemple de déréférencement et d'utilisation d'un pointeur :

```
/* avec les d\'efinitions
   introduites dans les
   transparents pr\'ec\'edents
*/


int
main
(void)
{
   bar = *p_bar * 2 ;
   *p_foo += 4 ;
   return 0 ;
}
```

```
.text
.globl main
.type   main,@function
main:
        ......
movl    p_bar, %eax
movswl  (%eax),%eax
addl    %eax, %eax
movw    %ax, bar
movl    p_foo, %edx
movl    p_foo, %eax
movl    (%eax), %eax
addl    $4, %eax
movl    %eax, (%edx)
    ......
```

Attention

Les instructions :

```
int foo ;
int *p_foo = &foo ;
```

ne sont pas équivalentes aux instructions :

```
int foo ;
int *p_foo ;
*p_foo = &foo ;
```

Il ne faut savoir sur quoi pointent vos pointeurs ; les instructions :

```
int *ptr  ;
*ptr = 0 ;
```

provoqueront probablement une erreur de segmentation car `ptr` n'est pas initialisé (on ne sait pas sur quoi il pointe).

Pointeur de type void

Pointeur vers le type `void` :

- pointeur vers un type *quelconque* ;
- le déréférencement ne s'y applique pas ;
- utiliser l'opérateur de conversion explicite de type.

Exemple :

```
int foo ;

void * p_qlcq = &foo  ;

int
main
(void)
{
        foo =  * (int *) p_qlcq ;
        /*  "foo = *p_qlcq ; " est impossible */
        return 0 ;
}
```

Affectation à un pointeur

Les pointeurs peuvent s'utiliser en valeur gauche (affectation) à condition que :

- les pointeurs soient de même type i.e. même type d'objet pointé ;
- on affecte l'adresse d'une variable du type pointé ;
- l'expression de retour est un pointeur sur le type pointé.

Il est possible d'affecter une constante pointeur `NULL`.

```
int  a, *p_a =&a ;
char b, *p_b =&b ;
char c, *p_c =&c ;

   *p_a = *p_b ; /* conversion implicite */
   p_b  = p_c  ; /*   valide             */
/* p_a  = p_b  ;     invalide            */
```

Addition d'un pointeur et d'un entier :

- si `p` est un pointeur vers des objets de type `T` ;
- et `n` est un entier ;
- alors `p + n` est une expression
  - de type pointeur vers des objets de type `T`,
  - et de valeur l'adresse du `n`*ième* objet suivant celui pointé par `p` ;
- l'addition prend en compte la taille de l'objet.

```
int foo = 20 ;
int *p_foo = &foo ;

int
main
(void)
{
  p_foo += 3 ;
  return 0 ;
}
```

```
        .data
        .globl foo    .size   foo,4
foo:    .long   20
        .globl p_foo .size   p_foo,4
p_foo:  .long   foo
        .text
        .globl main
main:    .........
         addl    $12, p_foo
        .........
```

Spécificité du type void * — Soustraction

On ne peut faire de l'arithmétique de pointeurs de type void.

Soustraction d'un pointeur et d'un entier (identique à l'addition) :

- la valeur étant l'adresse du `n`*ième* objet précédent celui pointé par `p`.

Différence de pointeurs :

- si `p` et `q` sont des pointeurs de même type ;
- alors `p - q` est une expression :
  - de type entier,
  - dont la valeur est le nombre d'objets situés entre `p` et `q` inclus.

Autres opérations

Comparaison de pointeurs :

- si `p` et `q` sont des pointeurs de même type ;
- tous les opérateurs de comparaison sont utilisables ;
- `p == q` : même objet pointé (adresse identique) ;
- `p < q` : `p` pointe un objet précédent celui pointé par `q` ;
- comparaison à `NULL` possible.

Calcul sur les pointeurs cohérent :

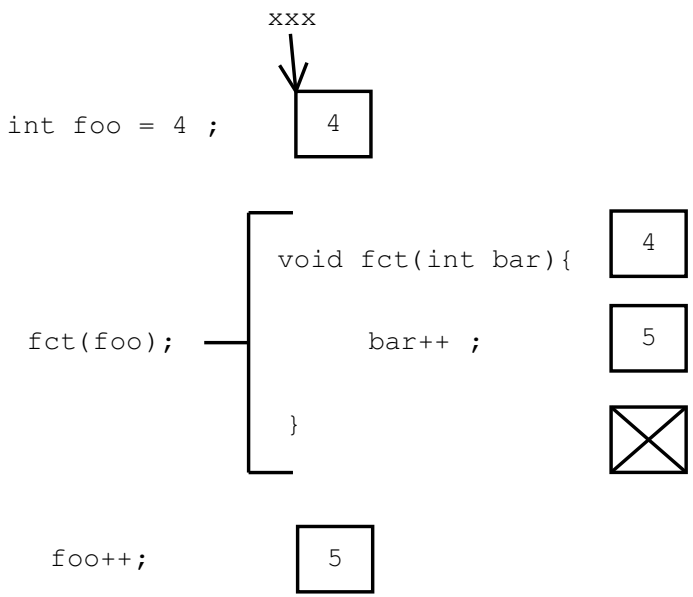
- prend en compte la taille des objets pointés ;
- `char *p; p=p+1;` : fait "avancer" `p` de 1 octet ;
- `int *p; p=p+1;` : fait "avancer" `p` de 4 octets ;
- arithmétique basée sur la taille des objets pointés (`sizeof`).

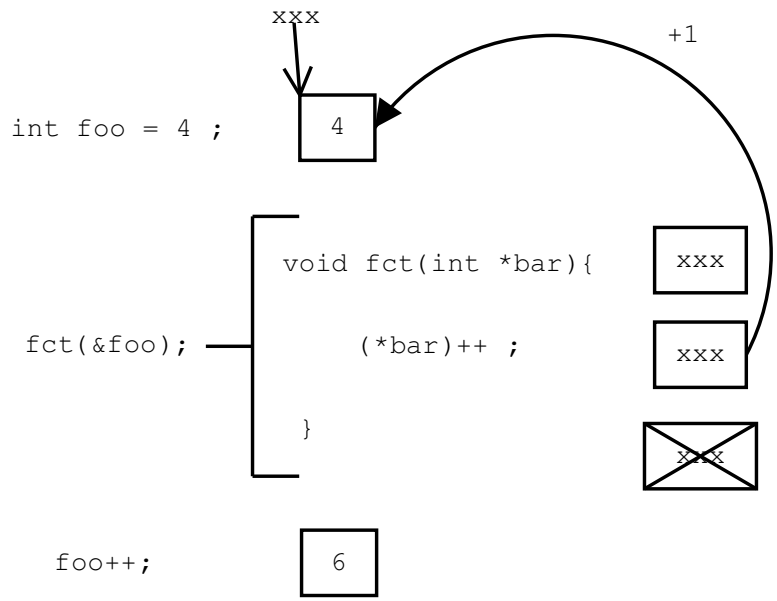
Tout autre calcul sur les pointeurs est *illicite*.

Lors de l'appel de fonction, le passage de paramètres est par valeur :

donc, pas d'effet de bord possible sur le paramètre dans l'appelante.

Les pointeurs permettent un effet latéral : c'est le passage par adresse.





Dans un langage "classique" : après la déclaration d'un tableau `t` :

- `t` est une variable ;
- `t` est de type tableau de *quelque chose* ;
- `t` désigne le tableau en son entier.

En C :

- `t` **n'est pas** une variable ;
- `t` **n'est pas** de type tableau de *quelque chose* ;
- `t` **ne désigne pas** le tableau en son entier.

Ainsi, si on a `int t[10];` :

- `t` est une **constante** ;
- `t` est du type **pointeur vers** `int` ;
- valeur de `t` : adresse du premier élément du tableau ;

Si `t` est un tableau, alors `t` $\equiv$ `&t[0]`.

Pointeurs et tableaux partagent abusivement des opérateurs

Il y a équivalence de notation :

Si `t` est un tableau, `t[i]` $\equiv$ `*(t + i)`

- un opérateur d'indexation est inutile dans le langage ;
- mais l'indexation est tout de même applicable à des variables de type pointeur ;

```
int tab[2] = { 1, 2 } ;

int *p = tab ;

int
main
(void)
{
 *(tab+1) = *tab ;
  p[1] = p[0] ;
  return 0 ;
}
```

```
        .data
tab:    .long   1,2
  p:    .long   tab
        .text
main:   .....
        movl    tab, %eax
        movl    %eax, tab+4
        movl    p, %edx
        addl    $4, %edx
        movl    p, %eax
        movl    (%eax), %eax
        movl    %eax, (%edx)
```

Passage de tableaux en paramètre

En conséquence des similitudes entre pointeurs et tableaux :

- un paramètre tableau est l'adresse du premier élément ;
- c'est une *variable* locale à la fonction : donc copie de l'adresse ! ! ! !
  Le passage de paramètre est systématiquement par adresse i.e. le tableau est modifié ! ! !
- deux syntaxes sont possibles :
  - par pointeur :

```
void inc_tab(int *tableau, int size) {
  register int i;
  for (i = 0; i < size; i = i + 1;)
    *(tableau + i) = *(tableau + i) + 1;
}
```

  - par tableau :

```
void inc_tab(int tableau[], int size) {
  register int i;
  for (i = 0; i < size; i = i + 1;)
    tableau[i] = tableau[i] + 1;
}
```

Pointeur et tableau multidimensionel (I)

```
char tab[3][4] = {"123","456","789"} ;
char *foo=NULL;
char bar=0 ;
int main(void){
        foo = *(tab+1) ;
        bar = *(*(tab+2)) ;
        bar = foo[2] ;
        return 0 ;
}

/* la syntaxe des pointeurs
   s'applique aux tableaux
   et celles des tableaux
   aux pointeurs bien qu'il ne
   s'agissent  pas exactement
   des m\^emes types d'objet */
```

```
   .globl tab .data
tab:    .string "123"
        .string "456"
        .string "789"
        .globl foo
foo:    .long 0
.globl bar
bar:    .byte 0
        .text
.globl main
main:  .........
  movl $tab+4, foo
  movb tab+8, %al
  movb %al, bar
  movl foo, %eax
  addl $2, %eax
  movb (%eax), %al
  movb %al, bar
     ............
```

Pointeur et tableau multidimensionel (II)

Les pointeurs permettent un codage des tableaux multidimensionnels par un "arbre" :

```
#include <stdio.h>
char tab[3][4] = {"123","456","789"} ;

char *foo[3] = {NULL,NULL,NULL} ; /* ce sera char **foo ; */
                                  /* lorsque nous saurons faire */
int                               /* de l'allocation dynamique  */
main                              /* (prochain cours).          */
(void)
{
     foo[0] = tab[0] ;
     foo[1] = tab[1] ;
     foo[2] = tab[2] ;
     return (int) foo[1][1] ;
}
```

Pointeurs et types composés

Pointeur sur une structure : usage de l'opérateur de sélection ->

En utilisant un exemple du cours précédent :

```
struct adresse {
  int num;
  char rue[40];
  long int code;
  char ville[20];
};

struct adresse var, *ptr = &var;

ptr->num=39; (*ptr).code = 59000 ;
ptr->rue[0]=N ; (*ptr).rue[1] = i;
```

accès : *(*pointeur_union).membre* ou *pointeur_union->membre*

On peut maintenant jouer avec les pointeurs...

```
enum bool_m {FALSE,TRUE} ;

enum bool_m *p_bool_v, bool_v ;

int
main
(void)
{
        bool_v   = TRUE    ;
        p_bool_v = &bool_v ;
        return *p_bool_v   ;
}
```

tout en ayant une idée claire de ce qui se passe en mémoire.
(au besoin, gdb peut nous aider).