Exercices à faire
1, 2, 3, 5, 6, 7, 11, 13, 15, 16
3 12 14 16

ENSEIRB 2006-2007 - Première année
PG101 - Programmation en C

TD 5 : Pointeurs et structures
(commentaires)

1 Pointeurs

Exercice 1:

1. pour que la fonction `echange` puisse échanger la valeur de ses deux paramètres, elle a besoin d'une variable locale qui stocke temporairement la valeur de la première variable écrasée lors de l'échange. Ceci nous donne le programme :

```
#include <stdio.h>

void echange(int a, int b) {
  int x;

  x = a;
  a = b;
  b = x;
}

int main() {
  int a=2, b=3;

  printf("avant : a=%d, b=%d\n", a, b);
  echange(a, b);
  printf("apres : a=%d, b=%d\n", a, b);

  return 0;
}
```

2. À l'exécution, nous obtenons l'affichage suivant :

```
avant : a=2, b=3
apres : a=2, b=3
```

   manifestement, les valeurs de `a` et de `b` ne sont pas échangées par la fonction `echange`

3. Afin de mettre en évidence le problème, nous instrumentons le code pour qu'il affiche les *adresses* des variables `a` et `b` locales à `main` et celles des paramètres de `echange`. Ceci donne le programme :

```
#include <stdio.h>

void echange(int a, int b) {
  int x;

  printf("echange : &a=%p, &b=%p\n", &a, &b);

  x = a;
  a = b;
  b = x;
}

int main() {
  int a=2, b=3;

  printf("main : &a=%p, &b=%p\n", &a, &b);

  printf("avant : a=%d, b=%d\n", a, b);
  echange(a, b);
  printf("apres : a=%d, b=%d\n", a, b);

  return 0;
}
```

Remarque : `%p` est le format utilisé pour l'affichage des valeurs des pointeurs et `&` est l'opérateur qui renvoie l'adresse de la variable qui suit

À l'exécution de ce programme, nous obtenons l'affichage (les adresses peuvent être différentes) :

```
main : &a=ffbffac4, &b=ffbffac0
avant : a=2, b=3
echange : &a=ffbffaa4, &b=ffbffaa8
apres : a=2, b=3
```

Nous voyons ici le problème : la fonction `echange` modifie la valeur de ses paramètres `a` et `b` qui ne partagent pas la même zone mémoire que les variables locales de même nom dans `main`

4. La solution consiste donc à fournir à la fonction `echange` non pas les valeurs des variables `a` et `b` locales à `main`, mais leurs adresses. Ainsi, cette fonction pourra modifier les bonnes zones mémoires. Nous obtenons donc le programme :

```
#include <stdio.h>

void echange(int *a, int *b) {
  int x;

  x = *a;
  *a = *b;
  *b = x;
```

```
}

int main() {
  int a=2, b=3;

  printf("avant : a=%d, b=%d\n", a, b);
  echange(&a, &b);
  printf("apres : a=%d, b=%d\n", a, b);

  return 0;
}
```

Remarque : la déclaration `int *a` signifie que `a` est un pointeur sur une zone mémoire de type `int`. L'instruction `*a` (sans type) renvoie la valeur de la zone mémoire pointée par `a`

On constate que l'échange est maintenant correctement réalisé.

5. Afin de mettre en évidence les modifications, nous instrumentons à nouveau le code :

```
#include <stdio.h>

void echange(int *a, int *b) {
  int x;

  printf("echange : a=%p, b=%p\n", a, b);
  printf("echange : *a=%d, *b=%d\n", *a, *b);

  x = *a;
  *a = *b;
  *b = x;
}

int main() {
  int a=2, b=3;

  printf("main : &a=%p, &b=%p\n", &a, &b);

  printf("avant : a=%d, b=%d\n", a, b);
  echange(&a, &b);
  printf("apres : a=%d, b=%d\n", a, b);

  return 0;
}
```

Voici le résultat de l'exécution (les valeurs des pointeurs peuvent changer) :

```
main : &a=ffbffac4, &b=ffbffac0
avant : a=2, b=3
echange : a=ffbffac4, b=ffbffac0
echange : *a=2, *b=3
```

```
apres : a=3, b=2
```

Nous voyons que la valeur des paramètres `a` et `b` de la fonction `echange` correspond bien aux adresses des variables `a` et `b` locales à `main`. La fonction `echange` possédant ces adresses est donc à même d'y lire et d'y écrire des informations, ce qui permet de réaliser l'échange des valeurs

Exercice 2:

Comme vient de le montrer l'exercice précédent, pour que `scanf` puisse modifier la valeur de la variable `c` (invisible dans `scanf`), il faut lui fournir l'adresse de `c` qui identifie la zone mémoire correspondant à `c`. Ainsi, `scanf` peut écrire dans cette zone mémoire, donc dans `c`. C'est donc pour cela que l'appel à `scanf` s'écrit `scanf("%c", &c)`, plutôt que `scanf("%c", c)`.

Exercice 3:

Voici la sortie obtenue lors de l'exécution de ce programme :

```
Taille de c=4, *c=1
Taille de s=4, *s=2
Taille de n=4, *n=4
Taille de l=4, *l=4
Taille de f=4, *f=4
Taille de v=4, *v=1
```

Nous constatons que tous les pointeurs occupent le même espace mémoire quel que soit le type pointé. Ceci est logique puisqu'un pointeur est une variable dont la valeur est une adresse.

La déclaration d'un pointeur en langage C implique de déclarer le type référencé. Par exemple `int *n` déclare un pointeur `n` sur un `int` qui est ici le type référencé. Notons que le type `void *` permet de déclarer des pointeurs sans spécifier le type référencé. Ceci est particulièrement utile lorsqu'on ne connaît pas le type référencé à la compilation, mais seulement lors de l'exécution (par exemple pour l'implantation d'un type de données)

Nous verrons à l'exercice «Arithmétique de pointeurs» pourquoi le type référencé par un pointeur est déclaré en langage C contrairement à d'autres langages.

Remarque : les valeurs issues de cette exécution peuvent varier en fonction de la machine utilisée (32 bits comme ici ou 64 bits), mais aussi en fonction du compilateur. En effet, le type `void` a parfois la taille 0 et non pas 1 (`sizeof(*v)`)

Exercice 4:

Voici l'affichage obtenu lors de l'exécution de ce programme :

```
*** main
Adresse de a : ffbffac4
Adresse de b : ffbffab0
Adresse de g : 20ab8
*** f
```

```
Adresse de x : ffbffa94
Adresse de y : ffbffa98
Adresse de z : ffbffa9c
Adresse de w : ffbffa3c
Adresse de g : 20ab8
```

Cet exercice veut mettre en évidence l'existence et la fonctionnement de la pile d'appel des fonctions. À chaque appel d'une fonction, ses paramètres et ses variables locales sont empilées. Lors de la terminaisons de l'exécution de cette fonction, les paramètres sont dépilés et le résultat (s'il y en a un) est empilé à son retour pour être récupéré par la fonction appelante.

Ce programme a été exécuté sur la machine `tanit` qui est un serveur SUN d'architecture SPARC qui exécute le système d'exploitation Solaris. Il faut savoir que sur cette architecture, la pile croît dans le sens décroissant des adresses.

Commençons tout d'abord par l'affichage issu de la fonction `main`. Par définition `a` et `b`, les deux variables locales de `main` sont sur la piles. Nous voyons qu'elle s'y trouvent dans l'ordre de leur définition. La pile est alors dans l'état suivant :

```
            | ... |
            |-----|
            |     |
            |     | a    (4 octets)
ffbffac4    |     |
            |-----|
ffbffac0    | ... |      (4 octets)
            |-----|
            |     |
            |     | b[1] (4 octets)
            |     |
ffbffabc    |     |
            |-----|
            |     |
            |     | b[0] (4 octets)
            |     |
ffbffab8    |     | <----
            |-----|
            | ... |
```

Comme nous le voyons, les variables sont empilées de haut vers le bas (dans le sens décroissant des variables). Les tableaux vont dans le sens des adresses croissantes (voir l'exercice «Arithmétique de pointeurs»). Les 4 octets entre `b` et `a` sont ajoutés pour des raisons internes au système. La flèche représente l'emplacement du *pointeur de pile*, c'est à dire l'octet à partir duquel le contenu de la pile est exploité par le programme.

Par contre, l'adresse de `g`, `20b10`, montre que cette variable globale n'est pas stockée sur la pile du programme. Nous savons en effet que les variables globales sont stockées dans une zone mémoire séparée et réservée à cet effet.

Nous considérons maintenant l'affichage produit par l'exécution de la fonction `f`. Les adresses affichées montre le contenu de pile suivant :

```
            | ... |
            |-----|
            |     |
            |     | a    (4 octets)
ffbffac4    |     |
            |-----|
ffbffac0    | ... |      (4 octets)
            |-----|
            |     |
            |     | b[1] (4 octets)
            |     |
ffbffabc    |     |
            |-----|
            |     |
            |     | b[0] (4 octets)
            |     |
ffbffab8    |     |
            |-----|
            | ... |      (16 octets)
            |-----|
            |     |
            |     | z    (4 octets)
            |     |
ffbffaa4    |     |
            |-----|
            |     |
            |     | y    (4 octets)
            |     |
ffbffaa0    |     |
            |-----|
            |     |
            |     | x    (4 octets)
            |     |
ffbffa9c    |     |
            |-----|
            | ... |      (84 octets)
            |-----|
            |     |
            |     | w    (4 octets)
            |     |
ffbffa44    |     | <----
            |-----|
            | ... |
```

Nous voyons que les paramètres de `f` sont empilés à la suite sur la pile, dans l'ordre inverse de leur déclaration (ceci est lié à cette architecture : les paramètres sont alors lisibles dans l'ordre où ils sont déclarés). Nous voyons ensuite que la variable locale `w` apparaît à son tour

au dessus des autres sur la pile. Des zones mémoires sont réservées pour le système entre les différents blocs, en particulier pour gérer le retour de l'appel de la fonction `f`. Le pointeur de pile a été déplacé en conséquence. Enfin, la variable `g` a bien entendu toujours la même adresse, hors de la pile.

On remarque que le paramètre `y` ne consomme que 4 octets en mémoire, alors qu'on a passé à `f` le tableau `b`, de taile 8 octets, en paramètre. Ceci est du au fait que c'est l'adresse de `b` qui est passée à `f`, il n'y a pas recopie du tableau.

À l'issue de l'exécution de `f`, lors du retour dans la fonction `main`, la pile reprend l'aspect qu'elle avait auparavant. C'est à dire que le pointeur de pile est déplacé dans la position qu'il occupait avant l'appel (le contenu de la pile n'est à priori pas nettoyé, ceci n'étant pas utile)

Exercice 5:

Voici le programme de calcul récursif de la factoriel, instrumenté pour observer la pile :

```
#include <stdio.h>

unsigned int fact(unsigned int n) {
  unsigned int f;

  printf("Adresse de n : %p, Valeur de n : %u\n", &n, n);
  if (n == 0)
    return 1;
  else {
    f = fact(n-1);
printf("Adresse de n : %p, Valeur de n : %u\n", &n, n);
return (n*f);
}
}

int main() {
  unsigned int n;

  printf("n: ");
  scanf("%u", &n);
  printf("La factorielle de %u vaut : %u\n", n, fact(n));

  return 0;
}
```

Son exécution produit l'affichage suivant :

```
n: 5
Adresse de n : ffbffab4, Valeur de n : 5
Adresse de n : ffbffa3c, Valeur de n : 4
Adresse de n : ffbff9c4, Valeur de n : 3
Adresse de n : ffbff94c, Valeur de n : 2
Adresse de n : ffbff8d4, Valeur de n : 1
```

```
Adresse de n : ffbff85c, Valeur de n : 0
Adresse de n : ffbff8d4, Valeur de n : 1
Adresse de n : ffbff94c, Valeur de n : 2
Adresse de n : ffbff9c4, Valeur de n : 3
Adresse de n : ffbffa3c, Valeur de n : 4
Adresse de n : ffbffab4, Valeur de n : 5
La factorielle de 5 vaut : 120
```

Nous pouvons observer deux choses :

- les valeurs de `n` sont successivement empilées à chaque appel récursif faisant croître la pile de 120 octets à chaque fois. Ces octets stockent la valeur de `n`, mais aussi divers éléments permettant l'exécution de la fonction `fact` et le retour dans la fonction appelante
- ceci est illustré dans les lignes où la valeur de `n` croît. En effet, nous voyons qu'une fois l'appel à `fact(0)` terminé, nous revenons dans `fact(1)` qui avait appelé `fact(0)`. Nous voyons alors que le contexte d'appel de `fact(1)` est à nouveau disponible pour poursuivre le calcul de la factorielle (calculer la multiplication par `n`)

Exercice 6:

1. Lors de l'exécution du programme, nous observons l'affichage suivant :

   ```
   Valeur de t=ffbffac8
   Valeur de p=0
   ```

   nous constatons que la variable `t`, qui est un tableau, stocke en fait une adresse. Il s'agit de l'adresse du premier octet de `t`.

   La valeur `NULL` (constante symbolique) représente l'adresse nulle. Il s'agit d'une adresse inexistante (généralement, 0) servant souvent de butée pour des files de pointeurs, ou pour des initialisations

2. Après modification du programme nous obtenons l'affichage suivant :

   ```
   Valeur de t=ffbffac8
   Valeur de p=0
   *** Modification de p
   Valeur de p=ffbffac8
   t[1]=2
   Modification de p[1]
   t[1]=77
   ```

   Comme nous pouvons nous y attendre, l'instruction `p = t` affecte la valeur de `t` à `p`. Par conséquent, `p` pointe alors la même zone mémoire que `t` après l'exécution de cette instruction, comme le montre la nouvelle valeur de `p` égale à `ffbffac8` comme `t`.

   Dès lors, la modification de `p[1]` affecte la zone mémoire commune à `p` et `t`. En conséquence de quoi, la valeur de `t[1]` a bel et bien changé après l'instruction `p[1] = 77`

3. Nous obtenons le message suivant du compilateur :

   ```
   prog9.c: In function 'main':
   prog9.c:20: error: incompatible types in assignment
   ```

L'instruction `t = p` est illégale : les types de `t` et `p` ne la permettent pas. Nous ne pouvons en effet pas modifier la valeur de `t` (i.e. l'adresse du premier octet de `t`)

Il y a plusieurs différences entre les tableaux et les pointeurs :

- lors de la définition d'un tableau, une zone mémoire est réservée pour le stockage de ses éléments, contrairement à un pointeur. Par exemple, la définition `int t[5]` définit le tableau `t` et réserve $5 \times$ `sizeof`(*int*) octets (20 octets dans notre cas). Par contre, la définition `int *p` ne réserve que l'espace nécessaire au stockage de la valeur de `p`. Ainsi, l'instruction `t[1]=2` est valide après la définition de `t` puisque `t[1]` est une zone mémoire réservée, mais l'instruction `p[1]=2` est invalide (bien que syntaxiquement correcte, donc non détectable par le compilateur) car `p[1]` correspond (sauf hasard autement improbable qui ferait coïncider `p` et `t`) à une zone mémoire non réservée
- autre différence essentielle : la valeur de `p` peut librement être modifiée, alors qu'il est syntaxiquement incorrect de vouloir modifier celle de `t`. Ceci est justement dû au fait qu'une zone mémoire a été réservée pour `t`, contrairement à `p`. Ainsi `p = t` est une instruction valide, alors que `t = p` est invalide.
  ATTENTION DONC, L'OPÉRATEUR D'AFFECTATION NE PEUT PAS ÊTRE UTILISÉ SUR UN TABLEAU. POUR COPIER LE CONTENU D'UN TABLEAU DANS UN AUTRE, IL FAUT LE FAIRE ÉLÉMENT PAR ÉLÉMENT, COMME NOUS L'AVONS FAIT POUR LES CHAÎNES DE CARACTÈRES

Exercice 7:

1. Voici l'affichage observé lors de l'exécution de ce programme :

```
Valeur de s1=ffbffac8
Affichage de la chaine s1 : Bonjour!
Valeur de s2=10790
Affichage de la chaine s2 : Bonjour!
```

   Nous voyons tout d'abord que `s1` et `s2` contiennent toutes les deux la chaîne `Bonjour !`
   Cependant, les valeurs de `s1` et `s2` indiquent que les deux chaînes de caractères ne sont manifestement pas stockées dans la même zone de la mémoire. C'est effectivement le cas :

   - la définition `char s1[] = "Bonjour !"` réserve un tableau de taille 9 (8 caractères plus la butée `'\0'`) et y copie les caractères `'B'`, `'o'`, `n`, etc. Cette définition est exactement équivalente à :
     `char s1[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '!', '\0'};`
     que nous avons déjà rencontrée pour les tableaux. Elle a pour but de simplifier l'initialisation des chaînes de caractères en ajoutant automatiquement la butée `'\0'`
   - par contre, la définition `char *s2 = "Bonjour !"` fait tout autre chose. Comme nous l'avons dit a l'exercice précédent, la définition d'un pointeur ne donne lieu à aucune réservation d'espace mémoire. Pour stocker un la chaîne de caractères, un espace mémoire est donc réservé par le compilateur. Mais celui-ci ne peut pas être pris sur la pile (voir l'exercice «Pile et adresses») car la valeur de `s2` peut être modifier. L'adresse de la chaîne sur la pile serait alors perdue, rendant impossible la sortie de la fonction. Cette mémoire est donc réservée dans un espace spécial (d'où la différence d'adresses entre `s1` et `s2`) et `s2` contient l'adresse du premier octet de cette zone

2. Après modification du programme, nous obtenons à l'exécution l'affichage suivant :

```
Valeur de s1=ffbffac8
Affichage de la chaine s1 : Bonjour!
Valeur de s2=107f0
Affichage de la chaine s2 : Bonjour!
*** Modification de s1[0]
Affichage de la chaine s1 : $onjour!
*** Modification de s2[0]
Segmentation Fault (core dumped)
```

   Nous constatons que la modification de `s1` ne pose pas de problème, comme nous pouvons nous y attendre, par contre, celle de `s2` semble impossible.

   C'est en effet le cas. La zone mémoire spéciale réservée pour stocker la chaîne pointée par `s2` est protégée en écriture : il n'est donc pas permis de modifier cette chaîne

3. Après modification du programme, celui-ci affiche :

```
Valeur de s1=ffbffac8
Affichage de la chaine s1 : Bonjour!
Valeur de s2=107d0
Affichage de la chaine s2 : Bonjour!
*** Modification de s2
Valeur de s2=10868
Affichage de la chaine s2 : Bonsoir!
```

   Ceci est valide, puisque l'instruction `s2 = "Bonsoir !"` ne fait que modifier la valeur du pointeur `s2`. Une zone mémoire protégée a été réservée pour la chaîne `Bonsoir !` et après l'affectation, `s2` pointe sur le premier octet de cette chaîne

4. Après modification du programme, nous obtenons le message suivant du compilateur :

```
prog12.c: In function 'main':
prog12.c:13: error: incompatible types in assignment
```

   L'instruction `s1 = "Bonsoir !"`. En effet, comme nous l'avons vu, elle correspond à une initialisation de tableau qui ne peut survenir que lors de la déclaration de celui-ci. L'opérateur d'affectation ne peut pas être utilisé sur les tableaux, donc sur les chaînes de caractères non plus (hors initialisation bien entendu)

Les chaînes de caractères étant des tableaux, il existe en premier lieu les mêmes différences entre les tableaux et les pointeurs, qu'entre les chaînes de caractères et les pointeurs. Mais nous avons observé une différence supplémentaire lors de l'initialisation des chaînes de caractères

Exercice 8:

Notons tout d'abord que les opérateurs arithmétiques peuvent être utilisés sur les pointeurs : l'expression `p+i` consiste à ajouter à la valeur de `p` (un pointeur) celle de `i` (un `int`). Il y a cependant des restrictions qui sont expliquées dans le cours.

1. À l'exécution du programme, nous observons l'affichage suivant :

```
0
4
8
c
10
```

Nous en déduisons donc que `p+0` vaut `0`, ce qui est normal puisque `p` est initialisée à `0`. Par contre, `p+1` vaut `4`, `p+2` vaut `8`, `p+3` vaut `c` et `p+4` vaut `10`. Pourquoi donc cette valeur augmente de 4 en 4 alors que celle de `i` augmente de 1 en 1 ?

Le langage C a été crée de façon à rendre la programmation plus simple que si nous utilisions un langage d'assemblage, sans pour autant restreindre la capacité du programmeur. Il est donc possible d'accéder à la mémoire, comme nous le ferions avec un langage d'assemblage. Par contre, les créateurs du langage C ont tiré parti des types des données pour rendre plus aisée la navigation dans la mémoire. Ainsi, l'instruction `p+i` n'ajoute pas `i`, mais `i` fois la taile du type pointé par `p` (c'est à dire `sizeof(*p)`) à la valeur de `p`. Ici, le type pointé par `p` est `int`, qui occupe 4 octets en mémoire sur notre machine, d'où l'incrément de 4 en 4.

Nous voyons donc l'utilité de déclarer le type pointé afin de pouvoir utiliser l'arithmétique de pointeur pour se déplacer plus simplement en mémoire. Ceci est illustré dans l'exercice suivant.

2. Se déplacer d'octet en octet revient à se déplacer de 1 en 1. Il suffit donc de modifier la définition de `p` pour qu'elle pointe sur un type de taille 1 : `char`. La définition de `p` devient donc :

```
char *p = 0;
```

Exercice 9:

Ce programme consiste à déplacer caractère par caractère dans une chaîne et à les affficher un à un. Pour cela, nous utilisons un pointeur `p` défini par : `char *p`. Voici l'analyse du traitement de file (on rappelle qu'une chaîne est file dont la butée n'est pas traitée) :

- `<obtenir le 1er élément>` consiste à lire le caractère pointé par `p` par l'instruction `*p`
- `<obtenir l'élément suivant>` consiste à incrémenter `p` pour qu'il pointe le caractère suivant dans la chaîne : `p++` puis à lire la la valeur comme précédemment par `*p`
- `<élément != butée>` il suffit de tester que le caractère courant est différent de la butée de fin de chaîne : `*p != '\0'`
- `<traiter l'élément courant>` consiste à l'afficher : `printf("%c", *p)`
- `<initialisations>` il faut que `p` pointe sur le premier caractère de `s` avant de débuter le traitement de file, ce qui s'obtient par : `p = s`
- `<présentation du résultat>` on affiche un saut de ligne pour l'esthétique

Nous obtenons le programme suivant :

```
#include <stdio.h>

int main() {
  char s[] = "Bonjour!";
```

```
  char *p = s;

  while (*p != '\0') {
    printf("%c", *p);
    p++;
  }
  printf("\n");

  return 0;
}
```

Nous déduisons de cet exercice et du précédent que `t[i]` est équivalent à `*(t+i)`, c'est à dire la valeur obtenue en se déplaçant de `i` éléments dans `t`

Exercice 10:

Nous mettons en évidence la structure de `t` en affichant `t[0]`, `t[1]`, etc, par le programme suivant :

```
#include <stdio.h>

int main() {
  int t[5][5], i;

  for (i=0; i<5; i++)
    printf("Adresse de t[%d] : %p\n", i, t[i]);

  return 0;
}
```

Nous obtenons à l'exécution l'affichage suivant :

```
Adresse de t[0] : ffbffa70
Adresse de t[1] : ffbffa84
Adresse de t[2] : ffbffa98
Adresse de t[3] : ffbffaac
Adresse de t[4] : ffbffac0
```

Nous constatons que `t[0]`, `t[1]`, etc, se suivent en mémoire, à 20 octets d'intervalle (rappel : en héxadécimal, 20 s'écrit 14). Nous voyons donc que `t[0]` est bien un tableau de 5 `int`, donc de taille $5 \times 4 = 20$ octets, de même pour `t[1]`, et ainsi de suite, et que ces sous-tableaux sont bien rangés les uns à la suite des autres, de façon contigüe.

Exercice 11:

Voici ce que nous obtenons à l'exécution de ce programme :

```
a=3, *b=3
a=3, *b=-12895704
```

(les valeurs peuvent être légèrement différentes).

Pourquoi la valeur de `y` est-elle modifiée bien qu'aucune instruction du programme ne le fasse explicitement ? C'est donc qu'une instruction le fait implicitement. Travaillons sur une vue abstraite de la pile où les adresses sont fictives et les tailles des variables sont abstraites. Voici l'état de la pile avant l'appel à `copie_int` :

```
        | ... |
        |-----|
10      |  3  | a
        |-----|
 9      |  ?  | b <----
        |-----|
        | ... |
```

La flèche représente là encore le pointeur de pile. Lors de l'appel à la fonction `copie_int`, le paramètre `x` est ajouté sur la pile, tout comme la variable locale `y`. Enfin, la valeur de `a` est copiée dans `x` à l'appel, et le pointeur de pile est actualisé :

```
        | ... |
        |-----|
10      |  3  | a
        |-----|
 9      |  ?  | b
        |-----|
 8      |  3  | x
        |-----|
 7      |  ?  | y <----
        |-----|
        | ... |
```

La fonction `copie_int` copie la valeur de `x` dans `y` :

```
        | ... |
        |-----|
10      |  3  | a
        |-----|
 9      |  ?  | b
        |-----|
 8      |  3  | x
        |-----|
 7      |  3  | y <----
        |-----|
        | ... |
```

Puis elle retourne l'adresse de `y` qui est copiée dans `b` au retour de l'appel de la fonction. Au retour dans la fonction principale, la pile a donc l'aspect suivant :

```
        | ... |
        |-----|
```

```
10      |  3  | a
        |-----|
 9      |  7  | b <----
        |-----|
 8      |  3  | (x)
        |-----|
 7      |  3  | (y)
        |-----|
        | ... |
```

Notez bien l'actualisation de la valeur de `b` qui contient l'adresse de l'ancienne variable locale `y`, et le déplacement du pointeur de pile. Les zones mémoires auparavant réservées à `x` et `y` sont maintenant hors du champs utilisé de la pile (d'où leur nommage entre parenthèses).

Ensuite, la fonction `printf` est appelée. Avant l'appel, ses paramètres `a` et `*b` sont évalués, et remplacés par les valeurs `3` (stockée dans la case de nom `a`) et `3` (stockée dans la case d'adresse 7, la valeur de `b`) respectivement. L'appel à `printf` procède, comme tout appel de fonction, à l'empilement de ses paramètres :

```
        | ... |
        |-----|
10      |  3  | a
        |-----|
 9      |  7  | b
        |-----|
 8      |  "  |
        |-----|
 7      |  a  |
        |-----|
 6      |  =  |
        |-----|
 5      |  %  |
        |-----|
        | ... |
        |-----|
        |  3  |
        |-----|
        |  3  | <----
        |-----|
        | ... |
```

Nous voyons donc sur la pile la chaîne de caractères passée à `printf`, suivie des valeurs de `a` et `*b`, ses deux derniers paramètres. L'affichage est alors réalisé, et au retour de l'appel à `printf`, nous avons la pile suivante :

```
        | ... |
        |-----|
10      |  3  | a
        |-----|
```

```
9       |  7  | b <----
        |-----|
8       |  "  |
        |-----|
7       |  a  |
        |-----|
6       |  =  |
        |-----|
5       |  %  |
        |-----|
        | ... |
        |-----|
        |  3  |
        |-----|
        |  3  |
        |-----|
        | ... |
```

Notez bien le déplacement du pointeur de pile. Nous allons maintenant procéder au second appel à `printf`. Comme précédemment, nous évaluons tout d'abord les expressions `a` et `*b` qui sont les paramètres de `printf`. La valeur de `a` est toujours 3, celle de `*b` est celle qui est dans la case d'adresse 7 (la valeur de `b`), c'est à dire `'a'` (le caractère `'a'`). Les paramètres de `printf` sont empilés et l'appel est effectué, ce qui donne la pile suivante :

```
        | ... |
        |-----|
10      |  3  | a
        |-----|
9       |  7  | b
        |-----|
8       |  "  |
        |-----|
7       |  a  |
        |-----|
6       |  =  |
        |-----|
5       |  %  |
        |-----|
        | ... |
        |-----|
        |  3  |
        |-----|
        | 'a' | <----
        |-----|
        | ... |
```

Nous voyons donc le problème : la valeur de `*b` a été indirectement modifiée par l'appel à la fonction `printf`.

Ceci est du au fait que nous avons gardé dans `b` l'adresse d'une zone mémoire qui n'est plus valide une fois que nous avons quitté la fonction `copie_int`. D'ailleurs, le compilateur nous a signalé le problème :

```
prog16.c: In function 'copie_int':
prog16.c:8: warning: function returns address of local variable
```

Remarque : rappelez-vous bien que nous avons fait abstraction des adresses réelles et des tailles de variables sur la pile, si bien qu'en réalité, ce 'est pas le caractère `'a'` qui occupe la zone mémoire pointée par `b`. Cependant, aux valeurs près, le mécanisme mis en œuvre est exactement celui que nous avons décrit.

La modification qui consiste à remplacer `return (&y)` par `return (&x)` produit un problème identique. En effet, `x` est un paramètre de la fonction `copie_int`, ce qui équivaut à une variable locale (sauf que sa valeur est initialisée lors de l'appel de la fonction). Par conséquent, la validité de cette zone mémoire se limite à l'intérieur de la fonction `copie_int`, et il est totalement invalide de l'exploiter dans la fonction `main`.

NOUS EN DÉDUISONS QU'IL NE FAUT EN AUCUN CAS RETOURNER L'ADRESSE D'UNE VARIABLE LOCALE OU D'UN PARAMÈTRE D'UNE FONCTION, CAR CELLE-CI/CELUI-CI NE DOIT PAS ÊTRE EXPLOITÉE EN DEHORS DE LA FONCTION

2 Définitions de nouveaux types

Exercice 12:

Un point dans le plan Euclidien se représente par un couple de nombre réels. Nous définissons donc la structure `point` de la façon suivante :

```
struct point {
  float x, y;
};
```

La manipulation des structures se fait usuellement champ par champ, l'affichage devra donc s'y conformer. Pour accéder à un champ d'une structure, nous utilisons l'opérateur `.`

Nous obtenons alors le programme suivant :

```
#include <stdio.h>

int *copie_int(int x) {
  int y;

  y=x;

  return (&y);
}

int main() {
  int a, *b;
```

```
  a = 3;
  b = copie_int(a);
  printf("a=%d, *b=%d\n", a, *b);
  printf("a=%d, *b=%d\n", a, *b);

  return 0;
}
```

Remarque : les structures peuvent s'initialiser comme dans l'exemple ci-dessus, au moyen d'un n-uplet de valeurs correspondant, en types, en nombres et en ordre aux champs de celle-ci. On pourrait de façon équivalent l'initialiser champs par champs après sa définition :

```
p.x = 2.4;
p.y = 3.17;
```

Exercice 13:

À l'exécution du programme nous obtenons l'affichage suivant :

```
Nom: Dupont
Prenom: Philippe
Numero d'etudiant: 430568
Etablissement: ENSEIRB
Nom: Dupont
Prenom: Philippe
Numero d'etudiant: 430568
Etablissement: ENSEIRB
```

Nous en déduisons que l'opérateur d'affectation effectue une recopie champ à champ des valeurs de la structure, y compris celle des structures complexes sur lesquelles il ne s'applique pas d'habitude, comme les tableaux

Exercice 14:

Une seule petite difficulté : comment obtenir l'adresse qui suit `c` ? D'après les principes d'arithmétique des pointeurs, nous savons qu'il s'agit de `&c+1`. Nous avons donc le programme :

```
#include <stdio.h>

#define MAX 10

struct carte_etd {
  char nom[MAX];
  char prenom[MAX];
  unsigned int num_etd;
  char etablissement[MAX];
};
```

```
void carte_etd__affiche(struct carte_etd c) {
  printf("Nom: %s\n", c.nom);
  printf("Prenom: %s\n", c.prenom);
  printf("Numero d'etudiant: %u\n", c.num_etd);
  printf("Etablissement: %s\n", c.etablissement);
}

int main() {
  struct carte_etd c1 = {"Dupont", "Philippe", 430568, "ENSEIRB"}, c2;

  carte_etd__affiche(c1);
  c2 = c1;
  carte_etd__affiche(c2);

  return 0;
}
```

1. À l'exécution, ce programme produit l'affichage suivant :

   ```
   Taille de c : 36
   Taille de c.nom : 10
   Taille de c.prenom : 10
   Taille de c.num_etd : 4
   Taille de c.etablissement : 10
   Adresse de c : ffbffab0
   Adresse de c.nom : ffbffab0
   Adresse de c.prenom : ffbffaba
   Adresse de c.num_etd : ffbffac4
   Adresse de c.etablissement : ffbffac8
   Adresse suivant c : ffbffad4
   ```

   Nous pourrions nous attendre à ce que la taille d'une structure soit égale à la somme des taille de ses champs (comme pour les tableaux, en quelque sorte). Or, nous voyons sur cet exemple que `c` occupe 36 octets en mémoire, alors que la somme des tailles de ses champs vaut 34 : il y a 2 octets d'écart. Ceci est dû au fait que les variables doivent être alignés en mémoire (voir le cours de «Structure des ordinateurs»), ce que le compilateur accomplit en ajoutant des octets superflus là où cela est nécessaire

   D'après leurs adresses, nous voyons que les champs sont espacés les uns des autres du nombre d'octets correspondant à leur taille (attention, on compte en base 16 !). D'après l'adresse suivant `c`, nous voyons que ces deux octets supplémentaires sont en fin de structure.

   On manipule généralement (en fait, sauf pour l'affectation), les structures champs à champs car l'alignement mémoire dépend de l'architecture sur laquelle s'exécute un programme. Par conséquent, lors de l'échange d'informations entre deux ordinateurs (via des fichiers ou un réseau), la structure en mémoire d'un même type `struct` peut différer sur les deux ordinateurs, provoquant alors des corruptions de données (si l'échange n'est pas réalisé champs à champs)

2. Nous modifions donc la structure et la fonction `main` en conséquence :

```
...
struct carte_etd {
  char nom[MAX];
  short annee;
  char prenom[MAX];
  unsigned int num_etd;
  char etablissement[MAX];
};

int main() {
  ...
  printf("Taille de c.annee : %lu\n", sizeof(c.annee));
  ...
printf("Adresse de c.annee : %p\n", &(c.annee));
  ...
return 0;
}
```

À l'exécution de ce nouveau programme, nous obtenons l'affichage suivant :

```
Taille de c : 40
Taille de c.nom : 10
Taille de c.annee : 2
Taille de c.prenom : 10
Taille de c.num_etd : 4
Taille de c.etablissement : 10
Adresse de c : ffbffab0
Adresse de c.nom : ffbffab0
Adresse de c.annee : ffbffaba
Adresse de c.prenom : ffbffabc
Adresse de c.num_etd : ffbffac8
Adresse de c.etablissement : ffbffacc
Adresse suivant c : ffbffad8
```

Nous voyons maintenant que la situation a empiré : le compilateur a été obligé d'ajouter 2 octets supplémentaires de bourrage, portant la taille de la structure à 40 octets pour 36 utiles. D'après les adresses des champs de `c`, nous voyons que des octets de bourrage ont été ajoutés entre les champs `prenom` et `num_etd`.

Remarquons que le compilateur n'a pas le droit de stocker les champs dans un ordre différent de leur déclaration : sinon il aurait pu stocker `annee` en fin de structure, résolvant ainsi les problèmes d'alignement mémoire

3. Nous modifions la structure en conséquence, et nous déplaçons les affichages des informations relatives à `annee` de sorte qu'elles soient produites dans l'ordre des champs :

```
...
struct carte_etd {
  char nom[MAX];
  char prenom[MAX];
```

```
  unsigned int num_etd;
  char etablissement[MAX];
  short annee;
};
...
```

Ceci nous conduit à l'affichage suivant :

```
Taille de c : 36
Taille de c.nom : 10
Taille de c.prenom : 10
Taille de c.num_etd : 4
Taille de c.etablissement : 10
Taille de c.annee : 2
Adresse de c : ffbffab0
Adresse de c.nom : ffbffab0
Adresse de c.prenom : ffbffaba
Adresse de c.num_etd : ffbffac4
Adresse de c.etablissement : ffbffac8
Adresse de c.annee : ffbffad2
Adresse suivant c : ffbffad4
```

Nous voyons qu'il n'y a plus désormais de gaspillage de mémoire puisque la taille de la structure est exactement égale à la somme des tailles de ses champs.

Lorsqu'on manipule de gros volumes de données, il est donc nécessaire de faire attention à l'ordre des champs des structure. Sur notre exemple, entre la deuxième et la troisième définition, on économise 4 octets par variable de type `struct carte_etd`. On gaspille donc 10% de la mémoire utilisée avec la deuxième définition de ce type

Exercice 15:

Par la somme de la taille des champs de la structure, on estime que 34 octets (réellement 36 d'après l'exercice précédent) sont ajoutés sur la pile à chaque appel de la fonction `carte_etd__affiche`. Notons que si on augmente la valeur de `MAX`, cela empire.

Or, nous pouvons nous rendre indépendant de la taille de la structure : il suffit pour cela de passer un pointeur sur celle-ci.

```
#include <stdio.h>

#define MAX 10

struct carte_etd {
  char nom[MAX];
  char prenom[MAX];
  unsigned int num_etd;
  char etablissement[MAX];
  short annee;
};
```

```
int main() {
  struct carte_etd c;

  printf("Taille de c : %u\n", sizeof(c));

  printf("Taille de c.nom : %u\n", sizeof(c.nom));
  printf("Taille de c.prenom : %u\n", sizeof(c.prenom));
  printf("Taille de c.num_etd : %u\n", sizeof(c.num_etd));
  printf("Taille de c.etablissement : %u\n", sizeof(c.etablissement));
  printf("Taille de c.annee : %u\n", sizeof(c.annee));

printf("Adresse de c : %p\n", &c);
printf("Adresse de c.nom : %p\n", &(c.nom));
printf("Adresse de c.prenom : %p\n", &(c.prenom));
printf("Adresse de c.num_etd : %p\n", &(c.num_etd));
printf("Adresse de c.etablissement : %p\n", &(c.etablissement));
printf("Adresse de c.annee : %p\n", &(c.annee));

printf("Adresse suivant c : %p\n", &c+1);

return 0;
}
```

Lorsque nous avons un pointeur sur une structure, comme `c` dans la fonction `carte_etd__affiche`, nous pouvons accéder à ses champs soit par : `(*c).nom`, soit par `c->nom`, l'opérateur `->` étant simplement un raccourci pratique.

Maintenant, seuls `sizeof(struct carte_etd *)` octets, 4 dans notre cas, sont ajoutés sur la pile et cela ne dépend pas de `sizeof(struct carte_etd)`.

Remarque : vous verez souvent des fonctions prenant en paramètre des pointeurs sur des variables qu'elles ne modifient pas, pour économiser de la mémoire et du temps (empiler des valeurs consomme aussi du temps)

Exercice 16:

1. Cette solution présente l'inconvénient de ne fournir aucune vérification de cohérence des données. Pour le champ `inscription`, certaines valeurs auront une signification, par exmeple : 0 pour informatique1, 1 pour telecom1, 2 pour electronique1, etc. Mais certaines valeurs, par exemple 345678, n'auront aucune signification associée, présentant ainsi un risque de corruption de données qu'il faudra contrer par de nombreux tests de cohérence
2. Les types énumérés nous permettent de résoudre ce problème. Nous définissons ici le type énuméré `enum formation` dont les valeurs sont l'ensemble des formations recensées dans notre programme :

   ```
   enum formation {
   I1, I2, I3, T1, T2, T3, E1, E2, E3
   };
   ```

Nous définissons alors la structure par :

```
struct carte_etd {
  char nom[MAX];
  char prenom[MAX];
  unsigned int num_etd;
  char etablissement[MAX];
  enum formation inscription;
};
```

Nous résolvons ainsi nos problèmes de cohérence : le compilateur va s'assurer que les variables de type `enum formation` sont correctement valuées, restreignant automatiquement leur domaine aux valeurs significatives

3. l'affichage d'un type énuméré se fait généralement au moyen de l'instruction `switch`. Nous obtenons le programme suivant :

```
#include <stdio.h>

#define MAX 10

struct carte_etd {
  char nom[MAX];
  char prenom[MAX];
  unsigned int num_etd;
  char etablissement[MAX];
};

void carte_etd__affiche(struct carte_etd *c) {
  printf("Nom: %s\n", c->nom);
  printf("Prenom: %s\n", c->prenom);
  printf("Numero d'etudiant: %u\n", c->num_etd);
  printf("Etablissement: %s\n", c->etablissement);
}

int main() {
  struct carte_etd c1 = {"Dupont", "Philippe", 430568, "ENSEIRB"}, c2;

  carte_etd__affiche(&c1);
  c2 = c1;
  carte_etd__affiche(&c2);

  return 0;
}
```

Exercice 17:

1. Une matrice 10×10 est un tableau de 10 tableaux. Nous choisissons le type `float` pour représenter les nombres flottants. Voici les définitions :

```
typedef float mat10R[10][10];

mat10R m;
```

2. La définition du type `carte_etd_t` est la suivante :

   ```
   typedef struct carte_etd carte_etd_t;
   ```

   Il est déconseillé de masquer un type structuré. En effet, ces types sont particuliers : par exemple, l'opérateur d'affectation s'applique sur une structure. Cela signifie par exemple qu'une structure est intégralement recopiée sur la pile lors de l'appel ou du retour d'une fonction dont elle serait paramètre ou type de retour. On peut être tenté d'utiliser ce type de déclaration pour masquer le type `struct carte_etd`, mais ce n'est pas la bonne façon de faire (on attendra le S2 pour cela)

3. Il y a plusieurs raisons pour ne pas faire cela, en voici une. Supposons que nous ayons deux variables `c1` et `c2` de type `carte_etd_t` :

   ```
   carte_etd_t c1, c2;
   c2 = c1;
   ```

   l'instruction `c2 = c1`, copie la valeur de `c1` dans `c2`. Donc, après l'exécution de l'affectation, `c1` et `c2` pointent la même zone mémoire.

   Cependant, syntaxiquement, rien ne laisse apparaître que ces deux variables sont des pointeurs, et on pourrait s'attendre à ce que l'affectation fasse une copie, comme dans le cas suivant :

   ```
   struct carte_etd c3, c4;
   c3 = c4;
   ```

   Ici, il y a copie champ à champ, `c3` et `c4` ne partagent pas la même zone mémoire.

   Syntaxiquement, rien ne permet de distinguer les deux situations, il y a donc une source potentielle d'erreurs quand on pense avoir deux zones mémoires distinctes (comme dans le cas de `c3` et `c4`) alors qu'elle est en réalité partagée par les deux variables (comme dans le cas de `c1` et `c2`). Voilà pourquoi il est fortement conseillé de ne pas dissimuler un pointeur dans un type.

Nous voyons sur ces deux exemples que la définition d'un type suit la même syntaxe que la définition d'une variable.

Remarque : les définitions de types se font en dehors de toute fonction