Licence 1 - section B
TD 6 d'éléments d'informatique

Catherine RECANATI – Département d'Informatique – Institut Galilée

Semaine du 21 novembre au 25 novembre 2016

1 Passage de variable par valeur

Exercice 1.1 Trace d'un programme avec appel fonctionnel de type f(x)

Soit le programme suivant :

```
1. /* déclaration de fonctionnalités supplémentaires */
2. #include <stdlib.h> /* pour EXIT_SUCCESS */
3. #include <stdio.h> /* pour printf */
4.
5. /* déclaration de constantes et types utilisateurs */
6.
7.        /* déclaration de fonctions utilisateurs */
8. int add1(int p) ;     /* la fonction add1 ajoute 1 a son argument */
9.        /* fonction principale */
10. int main()
11. {
12.     /* déclaration des variables locales x et y */
13.    int x ;
14.    int y ;
15.
16.    x = 4;
17.    y = add1(x);
18.
19.    return EXIT_SUCCESS;
20. }
21.
22.     /* definition de la fonction add1 */
23. int add1(int p) {
24.    return (p + 1);
25. }
```

Donner la trace d'une exécution de ce programme sous forme de tableau (en colonne : le numéro de ligne du programme, les valeurs des variables `x` et `y` , la valeur du paramètre formel `p` de la fonction `add1`, et sa valeur de retour `return`). Quelle est la valeur de la variable `y` en fin d'exécution ?

2 Pointeur et passage de variable par adresse

Les pointeurs sont des variables dont les contenus sont des adresses d'emplacement (ou case) mémoire.
Si on initialise un pointeur `p` avec l'adresse mémoire d'une variable `x`, on dit que `p` pointe sur `x`, car le pointeur permet alors d'accéder au contenu de la variable nommée `x`, comme le ferait une flèche pointant sur cette case. Syntaxiquement, le contenu pointé par p est alors désigné par `*p`.
Pour déclarer un pointeur `p` sur une case mémoire de type `int`, on déclare que `p` a le type `int*`, c'est-à-dire le type "pointeur sur un `int`" :

```
int*  p;
```

Petite astuce (pour assimiler la syntaxe) : pour se souvenir que la valeur pointée par `p` est un entier, on peut coller plutôt l'étoile au `p` comme ceci

```
int  *p;
```

En effet les espaces blancs sont ignorés du compilateur (le préprocesseur les supprime), et on a donc le choix de les mettre où on veut et même de les supprimer. On pourrait donc aussi écrire `int*p;` sans aucun espace. Mais si on colle l'étoile au `p` on se souviendra que la valeur pointée par `p` se note `*p` et que c'est une valeur de type `int`.

Mais cette déclaration de `p` n'est qu'une déclaration de type, et la variable `p` ne peut pas être utilisée avant d'avoir été définie. Pour la définir, il faut lui affecter une première valeur, c'est-à-dire l'initialiser. Pour l'initialiser on ne peut pas lui affecter n'importe quel entier, car un entier quelconque ne constitue pas nécessairement une adresse mémoire accessible. On initialise donc toujours les pointeurs par une adresse accessible depuis le programme, et donc a priori par l'adresse d'une variable.

Pour obtenir l'adresse d'une variable, on dispose de l'opérateur &. Cet opérateur, appliqué à une variable retourne son adresse. Il est noté & qu'on prononcera "et commercial" ou "esperluette". Ainsi, si `x` est une variable, `&x` est l'adresse de cette variable, et l'opérateur `*` est l'opérateur inverse de l'opérateur d'adresse. Dès que p sera initialisé par `&x`, on aura `*p == x` car `*p` égale `*(&x)`, et `*(&x)` égale `x`.

Exercice 2.1 Passage de variable par adresse : appel `f(&x)`

1. Préliminaire : qu'imprime le `main` suivant ?

```
5. int main() {
6.      int x;
7.      int  *p; /* p est un pointeur sur une variable de type int */
8.
9.      p = &x; /* p pointe maintenant sur la variable x */
10.     x = 3;
11.
12.     if (( *p == 3 ) && ( *p == x))
13.          printf("*p = 3 ET *p = x");
14.     return EXIT_SUCCESS;
15. }
```

2. Modifier le code de la fonction `add1` du premier exercice pour que son argument `p` soit de type pointeur sur un entier. Modifier aussi la fonction `main`, pour que l'appel à la fonction `add1` soit consistant avec le nouveau type de son paramètre formel.
   Vocabulaire : quand on passe l'adresse d'une variable (ou un pointeur sur une variable) en paramètre d'appel d'une fonction, on dit que *la variable est passée par référence* (par opposition au passage par valeur, illustré dans le premier exercice). On peut dire aussi que *la variable est passée par adresse*.

Exercice 2.2 Echanger les valeurs (ou contenus) de 2 variables

Soit le programme suivant :

```
1. /* déclaration de fonctionnalités supplémentaires */
2. #include <stdlib.h> /* pour EXIT_SUCCESS */
3. #include <stdio.h> /* pour printf */
4.
5. /* déclaration de constantes et types utilisateurs */
6.
7. /* déclaration de fonctions utilisateurs */
8. void echange(int a, int b) ;
9. /* fonction principale */
10. int main()
11. {
12.     /* déclaration et initialisation des variables x et y */
13.     int x = 4;
14.     int y = 0;
15.
16.     echange(x,y);
17.
18.     return EXIT_SUCCESS;
19. }
```

```
20.
21.  /* definition de la fonction echange */
22. void echange(int a, int b) {
24.    int sauve;
25.
26.    sauve = a;
27.    a = b;
28.    b = sauve;
29. }
```

1. Faire la trace du programme. Est-ce que la fonction `echange` a bien échangé les valeurs des paramètres formels `a` et `b` ? Et qu'en est-il de celles de `x` et `y` ?

2. On modifie le type des arguments formels de la fonction `echange` en la déclarant

   ```
   void echange(int *a, int *b);
   ```

   pour permettre le passage des variables `x` et `y` par référence, c'est-à-dire par leurs adresses. Modifier le corps de définition de la fonction `echange` pour tenir compte de cette nouvelle déclaration du type de ses paramètres formels. Modifier aussi le corps de la fonction `main` si nécessaire.

3. Faire la trace d'une exécution du programme ainsi modifié. Est-ce que la fonction `echange` a bien échangé les valeurs pointées par les paramètres formels `a` et `b` ? Et qu'en est-il de l'échange des valeurs initiales des variables `x` et `y` ?