TD 02 Programmer en C (2)

Fait TD 2.

Table des matières

Prévu

- passage de paramètres (valeur/pointeur) -> adresses et pointeurs
- printf/scanf
- manipulation des chaines de caractères -> tableau
- structures et énumérations, typedef
- préprocesseur
- allocation dynamique : malloc/free
- valgrind

1 Notion de pointeur

1.1 Adresses en paramètre, un exemple

1.1.1 Objectif, solution directe

On veut réaliser le programme suivant (algo)

```
demander combien vaut a
demander combien vaut b
afficher "la somme vaut" a+b
```

Une solution directe serait d'écrire

```
int a, b;
printf("Combien vaut a ?");
scanf("%d", & a);
printf("Combien vaut b ?");
scanf("%d", & b);
printf("la somme vaut %dn, a+b);
```

Rappel : on transmet, comme second paramètre de `scanf`, une adresse qui indique **l'adresse** de l'emplacement où `scanf` doit placer le résultat.

Pour ce qui suit : on imagine qu'on aura beaucoup de questions à poser par ailleurs, il est naturel de vouloir faire une fonction `demander_entier` qui regroupe `printf`/`scanf`, et éventuellement y rajoute des contrôles (vérifier que c'est un nombre valide), repose la question sinon, etc.

1.1.2 Pourquoi on doit transmettre une adresse

Pour la même raison que pour `scanf`, l'appel à cette fonction devra indiquer, en paramètre, **l'adresse** à laquelle le résultat devra être livré.

```
int a, b;
demander_entier("Combien vaut a ?", & a);   // & a = adresse de a
demander_entier("Combien vaut b ?", & b);
printf("la somme vaut %dn, a+b);
```

Remarque typographique : je mets un espace après & pour qu'il se détache visuellement.

La raison : le langage C ne connait qu'un seul type de passage de paramètre : le **passage par valeur**.

Par conséquent, pour réaliser ce qui, du point de vue conceptuel, est un passage de paramètre "en sortie" pour récupérer un résultat d'une action, dans un programme C on **transmet** la valeur de l'adresse où il faudra placer ce résultat.

Note : en C++ que vous étudierez par ailleurs, il existe un "passage par référence".

1.1.3 Que fait-on d'une adresse ? Notion de pointeur.

Donc la fonction `demander_entier` reçoit comme paramètres

- une chaine de caractères (pour la question)
- l'adresse d'un entier.

Son prototype s'écrit

```
void demander_entier(char question[],  int * adr_reponse);
```

Nous verrons les chaines et tableaux de caractères plus tard. Le point important ici est la présence de l'étoile dans la déclaration du second paramètre

La variable `adr_reponse` contiendra donc l'adresse de la réponse, pas sa valeur. On dit aussi que c'est un **pointeur d'entier**.

Dans le code de la fonction, on trouvera

```
void demander_entier(char question[],  int * adr_reponse)
{
   printf("%sn", question);
   scanf("%d",    adr_reponse);
}
```

Question Pourquoi n'y a t'il pas de "&" ?

Réponse parce qu'on donne à `scanf` l'adresse où ira la valeur lue, et que `adr_reponse` en est une.

1.2 Déréférencement d'un pointeur

Une variante : une fonction `demander_entier_positif`, qui repose la question tant qu'on rentre un nombre négatif.

```
void demander_entier_positif(char question[],  int *adr_reponse)
```

Il suffit d'enrober les deux instructions dans un `do/while`

```
do {
   ...
} while ( ...... );              // tant que réponse négative
```

mais la condition `(adr_reponse < 0)` ne va pas faire l'affaire. Ce que nous voulons comparer à 0 ce n'est pas l'adresse, mais la valeur pointée par l'adresse. On écrit :

```
do {
   printf("%sn", question);
   scanf("%d",    adr_reponse);
} while ( *adr_reponse < 0 );
```

- `adr_reponse` est un pointeur, c'est-à-dire une variable qui contient une adresse
- `*adr_reponse` est l'emplacement de la donnée "référencée" par le pointeur.

L'étoile sert à indiquer qu'on "déréférence un pointeur" : on accède à la donnée qu'il désigne.

1.3 Exercices

1. Écrire un programme qui

— lit deux entiers a, b
— les affiche
— appelle une fonction `echanger_entiers` qui les échange
— les affiche

2. Écrire un programme qui

— lit deux entiers a, b
— appelle une fonction `ordonner_entiers` qui les ordonne (le plus petit va dans a, le plus grand dans b) en appelant au besoin `echanger_entiers`.

3. **Exercice à la maison :** &crire une fonction qui “normalise” une durée passée dans trois entiers : nombre d’heures, minutes et secondes. Par exemple 123 minutes et 78 secondes, c’est 1 heure, 4 minutes et 18 secondes sous forme normalisée

1.4 Remarque : logique des déclarations

Remarque : Pour déclarer un pointeur `p`, on écrit `int` puis `*` puis `p;` mais la lecture littérale correcte de cette déclaration est

> “`*p` est un `int`”

Un principe fondamental en C (qu’on retrouve en C++) est que “la déclaration se conforme à l’usage”. Si on a un tableau `tab` de pointeurs d’entiers, on utilisera des expressions comme `*tab[n] = *tab[n+1];` et la déclaration de `tab` sera de la forme

```
int *tab[10];
```

De préférence, on colle donc l’étoile à la variable dans la déclaration. Coller l’étoile du côté du type est trompeur. A votre avis, dans

```
int* p, q;
```

quel est le type de la variable `q` ? Que devrait-on écrire à la place ?

2 Structures

2.1 Déclaration du type

En C, comme dans tous les langages modernes, on peut construire des types de données qui servent à agréger des valeurs.

Exemple :

```
struct Personne {
   char nom[100];
   char prenom[100];
   struct Date date_naissance;
}
```

Exercice : devinez la déclaration du type de données “`struct Date`”.

2.2 Déclaration et initialisation des variables

Le mot-clé `struct` fait partie du nom du type (ce n'est pas le cas en C++). Exemple de déclarations :

```
struct Personne mb;             // personne non initialisée

struct Personne jjd = {         // avec des "initialisateurs désignés",
     .prenom = "Jean-Jacques",  // ordre indifférent (depuis C99)
     .nom = "Dupont",
     .date_naissance = {
         .annee = 1923,
         .jour = 1,
         .mois = 4
      }
};

struct Personne jpd = {
     "Durand", "Jean-Paul", { 3, 4, 2001 };   // vous êtes sûr de l'ordre ?
};
```

2.3 Utilisation des champs, notations "point" et "flèche"

La notation "point" permet de désigner un champ d'un agrégat, comme en java : Exemples

```
printf("nom = %sn",   jpd.nom);
printf("annee de naissance = %dn",   jpd.date_naissance.annee);
```

Dans le cas où on part d'un **pointeur sur un agrégat**, il y a une notation commode

```
struct Personne *adr_personne =  & p1;   // adresse d'une structure
printf("nom = %sn",   adr_personne->nom);
```

On peut s'en passer, en déréférençant le pointeur et en accédant au champ, mais au prix de parenthèses

```
printf("nom = %sn",   (*adr_personne).nom);
```

à cause de priorités entre opérateurs "étoile" et "point".

2.4 Exercices agrégats

Pour ces exercices, n'hésitez pas à faire un "`main`" contenant des tests unitaires, avec des assertions

```
#include <assert.h>
...
assert (2 + 2 == 5);
```

le programme s'arrêtera à la première assertion fausse, en imprimant la ligne fautive.

1. Déclarer une structure pour représenter des durées, exprimées en heures, minutes et secondes.
2. Écrire une fonction `void normaliser_duree(struct Duree *adr_duree);` (qui ramène les secondes à un nombre entre 0 et 55, etc).

3. Écrire une fonction `construire_duree(int heures, int minutes, int secondes`, qui retourne une structure construire à partir d'un temps exprimée en heures, minutes et secondes.

```
struct Duree construire_duree(int heure, int minutes, int_secondes);
```

4. Écrire une fonction à 4 paramètres, qui prend une Durée et "l'éclate" en heures, minutes et secondes (paramètres de sorties).
5. Écrire une fonction `void afficher_duree(const struct Duree *adr_duree);` qui fait ce qu'elle dit. Pourquoi préfère-t-on passer une adresse plutôt qu'une structure ? Que signifie l'attribut `const` ?