Séance 2 : Types et structures de contrôle avancés

L2 Informatique – Université Côte d'Azur

Remarque 1 : lors du premier TP nous avons travaillé dans le répertoire `~/progC/tp1` ; pour ce TP, créez dans le dossier `~/progC` un répertoire `tp2`. Chaque exercice sera contenu dans un fichier (exemple : `exercice3.c`) et on utilisera un Makefile pour le compiler (toujours dans cet exemple vers : `exercice3`).

Remarque 2 : dans ce TP, vous devez écrire de nombreuses fonctions. Il est fondamentale de les tester, même lorsque ce n'est pas explicitement demandé dans l'énoncé. À n'importe quel moment, et à la demande de votre enseignant, vous devez pouvoir lancer l'exécutable des exercices déjà finis qui afficheront de manière lisible les tests effectuées. À retenir : *« Un code non testé est un code faux »*

Exercice 1 — Lire et ranger dans un tableau

1. Écrire une fonction `void affiche_tableau(int tab[], int n)` qui affiche les `n` premiers éléments d'un tableau `tab` d'entiers (`n` devant être inférieur à la taille du tableau). Écrire une fonction `question1` pour tester votre fonction.
2. Récupérer la fonction lire `lire_entier` du TP précédent (exercice 3, question 1).
3. Écrire un procédure `int saisie_tab(int tableau[])` qui lit sur l'entrée standard des entiers et les range dans un tableau. On s'arrêtera lorsque l'utilisateur rentrera une valeur non entière. La fonction renverra le nombre de valeurs lues et stockées dans le tableau.

stdin/stdout

```
entier> 12
entier> 23
entier> 0
entier> Bonjour ! Ça va ?
Le tableau lu est [ 12 , 23 , 0 ]
```

4. Que se passe-t-il si on saisit plus d'entiers que le tableau peut contenir d'éléments ? Et si on essaie d'afficher plus d'éléments que ne contient le tableau ? Expliquez.

Exercice 2 — Maîtriser la structure du temps

1. Définissez une structure de type `struct horaire_struct` contenant trois champs de type `short` pour les heures, minutes et secondes. Avec `typedef`, renommer ce type en `horaire`.
2. Créez une fonction `horaire nouvel_horaire(short h, short m, short s)` renvoyant un nouvel horaire à partir des trois valeurs données en paramètre.
3. Créez une fonction `void afficher_horaire(horaire h)` qui affiche proprement l'horaire sur une ligne de la forme : `12h 23m 15s`.
4. Écrivez une fonction `horaire ajout(horaire h1, horaire h2)` qui renvoie un nouvel horaire résultat de la somme des deux paramètres. On rappelle que 1h20m35s + 22h40m0s = 0h0h35s.
5. Écrivez une fonction `horaire secondes_vers_horaire(int s)` qui à partir du nombre de secondes écoulées depuis 0h0m0s renvoie l'horaire correspondant.
6. Écrivez une fonction `int horaire_vers_secondes(horaire h)` faisant l'inverse de la précédente.

Exercice 3 — Polymorphisme numérique

Partie I : création des types

1. Créez un nouveau type `enum` contenant deux valeurs : `entier` et `decimal`. Renommer ce nouveau type `typage` avec `typedef`.
2. Créez une `union` contenant deux champs : `ent` de type `int` et `dec` de type `float`. Ce type s'appellera `valeur`.
3. Enfin, créez une structure contenant deux champs : le premier de type `typage` et le second de type `valeur`. Le type ainsi créé sera dénommé `nombre`.
4. Écrire maintenant les cinq fonctions suivantes (on utilisera des `switch` lorsqu'on testera la valeur du champs `typage`).
   - `nombre nombre_entier(int i)`
   - `nombre nombre_decimal(float f)`
   - `int est_entier(nombre n) /* Renvoie un booléens*/`
   - `int est_decimal(nombre n) /* Renvoie un booléens*/`
   - `void afficher_nombre(nombre n) /* 2 chiffres après la virgule si décimal */`

Partie II : calculs sur les nombres

1. Écrire une fonction `nombre somme(nombre a, nombre b)` calculant la somme de deux nombres. On part du principe que la somme de deux entiers est un entier, la somme de deux décimaux est un décimal et que la somme d'un entier avec un décimal est un décimal.
2. Écrire une fonction `nombre division(nombre a, nombre b)` calculant le quotient de deux nombres. Si les deux nombres sont de typage entier et que `a` est un multiple de `b`, alors le résultat sera de typage entier ; dans tous les autres cas, le résultat sera décimal.
3. Écrire une fonction `nombre moyenne(nombre notes[], int taille)` calculant la moyenne du tableau. On s'assurera que les trois notes « 10, 5 et 15 » renvoie bien une moyenne de typage entier et que les cinq notes « 10, 5, 15, 12, 9 » renvoie un moyenne décimale.

Exercice 4 — Travailler avec des chaînes de caractères

Rappel : dans cet exercice et les autres, chaque fonction devra être testée dans une fonction dédiée (`question1`, `question2`, etc). Ces tests doivent s'afficher lorsque l'enseignant vous demande de lancer le programme.

1. Écrivez la fonction `int chercher_caractere(char chaine[], char symbole)` : cette fonction renvoie vrai (sous forme d'entier) si un caractère `symbole` apparaît dans une chaîne `chaine`, faux sinon.
2. Écrivez la fonction `int egale(char chaine1[], char chaine2[])` testant l'égalité de deux chaînes de caractères.
3. Écrivez la fonction `void miroir(char chaine [])` qui modifie la chaine en argument pour la remplacer par son miroir.
4. Écrivez la fonction `int inclue(char chaine[], char sous_chaine[])` qui teste si le second paramètre de la fonction est une sous-chaîne du premier.

Exercice 5 — Les bases du binaire

1. Écrivez une fonction `unsigned int lire_binaire(char chaine[])` qui transforme une chaîne représentant un nombre en binaire en entier. On autorisera les espaces (`"1111 1111"` correspond à 255). On renverra `-1` en cas d'erreur.
2. Écrivez une fonction `void afficher_binaire(unsigned int n)` qui affiche un entier en binaire. On pourra utiliser un tableau de taille suffisante pour contenir tous les bits d'un entier de type `int` (astuce : `sizeof(int)` donne le nombre d'octet utilisé par le type `int`). Par soucis de lisibilité, on affichera les bits par groupe de 4 et on n'affichera pas les zéros inutiles à la gauche du nombre. Exemple : `10 0010 1101`.

Exercice 6 — Jouer avec des bits

Dans cet exercice on considère les entiers comme une suite de bits. Le nombre 13 par exemple, s'écrivant en binaire `1101`, a son bit de position 0 égal à `1`, celui de position 1 égal à `0` et ceux de positions 2 et 3 égals à `1` ; tous les autres bits étant nuls.

1. Écrivez le code des fonctions suivantes (`x` est une variable, et `pos` est la position du bit à modifier). Chacune de ses fonctions ne devra contenir qu'une ligne avec un `return` et une unique formule contenant des opérations sur les bits.
   (a) `unsigned int get_bit(unsigned int x, int pos)` : retourne la valeur du bit situé à la position `pos`,
   (b) `unsigned int set_bit(unsigned int x, int pos)` : retourne un entier qui est égal à l'entier `x` pour lequel on a mis le bit à la position `pos` à 1,
   (c) `unsigned int clear_bit(unsigned int x, int pos)` : retourne un entier qui est égal à l'entier `x` pour lequel on a mis à zéro le bit à la position `pos`,
   (d) `unsigned int toggle_bit(unsigned int x, int pos)` : retourne un entier qui est égal à l'entier `x` pour lequel on a inversé le bit à la position `pos`,
   (e) `unsigned int define_bit(unsigned int x, int pos, int bool)` : retourne un entier qui est égal à l'entier `x` pour lequel on a défini le bit à la position `pos` suivant la valeur de `bool`.
2. Écrivez un programme qui effectue le miroir d'un entier non signé sur 32 bits. Par exemple si les entiers sont représentés sur 4 bits : 8 s'écrira 1000 son miroir est 0001 = 1 (question de contrôle terminal en 2013).
3. Écrivez un programme qui isole les 10 bits les plus à gauche, les 10 bits les plus à droite et les 12 bits restants au milieu d'un entier non signé.

Exercice 7 — Polymorphisme numérique, suite

Partie III : ajout du type fraction

1. Créer un type `fraction` à partir d'une structure ayant deux champs entiers : `numerateur` et `denominateur`.
2. Modifier l'exercice 3 pour ajouter un typage `rationelle` dans l'`enum` énumerant les typages. Modifier le type `nombre` en conséquent.
3. Ajouter la fonction `nombre rationnel(int p, int q)`. On s'assurera de simplifier la fraction avant de la sauvegarder dans un `nombre`. Dans la suite de l'exercice on s'assurera que la fraction est toujours sous forme irréductible.
4. Modifier aussi la fonction `afficher_nombre`. Un rationnel sera affiché au format : `p/q`.
5. Ajouter les deux opérations : `soustraction` et `multiplication`. Pour ces deux opérations ainsi que pour l'addition, en cas de typage différents entre deux paramètres, le résultat aura pour typage le plus grand des deux (avec la convention `entier` < `fraction` < `decimal` ; ainsi `fraction + decimal` donne `decimal`).
6. Modifier la fonction `division` pour que le quotient de deux entiers soit une fraction dans le cas de non-divisibilité. Dans tous les autres cas, on utilise la convention de la question précédente.
7. Enfin, écrire une fonction `float vers_flottant(nombre n)`.

Exercice 8 — Les bases sur les bases

On cherche à généraliser l'exercice 5 en travaillant sur des bases allant jusqu'à la base 36 (`0123…9ABCD…YZ`).

1. Écrivez une fonction `void remplir_chiffre(char chaine[])` qui remplit une chaîne de taille suffisante pour qu'elle soit égale à `"0123…9ABCD…YZ"`. Cette fonction permettra d'avoir un tableau contenant tous les chiffres nécessaire à l'utilisation de bases inférieures à 36.
2. Écrivez une fonction `int lire_base(char chaine[], int base)` généralisant la question 1 de l'exercice 5 mais pour une base quelconque (jusqu'à 36).
3. Écrivez une fonction `void afficher_base(int n, int base)` généralisant la question 2 de l'exercice 5 mais, là aussi, pour une base quelconque (jusqu'à 36).

Exercice 9 — Divisibilté par 3

Écrivez un programme, qui contient la fonction `multiple_de_3` : cette fonction renvoie vrai si une chaîne de caractères (passée en paramètre) contenant un nombre positif (supposé valide syntaxiquement) est multiple de trois, faux sinon. Rappel : Pour qu'un nombre soit multiple de trois, il suffit que la somme des chiffres de ce nombre soit un multiple de trois. 12345678 multiple de 3 ? $1+2+3+4+5+6+7+8 = 36$ et $3+6 = 9$, donc multiple de 3. Attention, votre fonction doit fonctionner même si votre entier est trop grand pour être représenté (en tant qu'entier) en C.