TP 5 - C

Les Tps se font sous un environnement linux, vous pouvez utiliser l'éditeur de texte que vous preferez.

Si vous souhaitez avoir un exemple d'usage de structures, enums et unions, vous pouvez consulter la première partie du fichier `vehicule.c` fourni avec ce tp.

Bases

1. Vecteur

Reprenez le code de l'exercice "Vecteur" du tp précédent et adaptez le pour qu'il utilise une structure. La structure contiendra le tableau ainsi que le nombre d'éléments. Sauvegardez votre code pour un futur exercice.

```
struct Vecteur {
    int* tableau;
    int taille;
};
```

2. Gestion d'erreurs

Le fichier `tp_5.c` contient le code d'une application qui contient trois comptes en banques et permet aux utilisateurs de retirer de l'argent.

Lisez ce code pour vous familiariser avec son fonctionnement, compilez le et lancez le.

La fonction `retrait_argent` renvoie −1 si l'opération de retrait a échouée mais elle n'indique pas l'erreur (mauvais nom ou solde insuffisant), le but de cet exercice est d'implémenter une gestion d'erreur robuste et efficace.

1. Créez une structure `struct Result` qui contiendra un numéro de compte ainsi qu'un `enum ResultCode` qui permettra de savoir si l'opération est réussie et sinon quelle erreur est survenue.

   Le code de resultat contiendra donc trois variantes: ok, compte_non_trouve, solde_insuffisant.

2. Modifiez le code pour que les deux fonctions `retrait_argent` et `trouver_compte` renvoient une `struct Result` plutôt qu'un `int`.

3. Créez une fonction `affiche_erreur` qui accepte en argument une `struct Result` et affiche l'erreur dans la console.

4. Pour l'instant, lorsque le solde est insuffisant, le programme n'affiche pas le solde courant. Modifiez la `struct Result` en ajoutant un attribut de type `union ErreurInfo` qui contiendra les informations nécéssaires pour afficher l'erreur. Les informations seront :
   - Pour l'erreur "compte non trouvé": le nom du compte recherché.
   - Pour l'erreur "solde inssufisant": le solde courant du compte.

Sauvegardez votre code pour l'exercice suivant

3. Création de comptes

1. Modifiez le code pour utiliser un tableau de `struct User` au lieu d'utiliser trois tableau. Les utilisateurs auront toujours un nom, un identifiant et un solde.
2. Inspirez vous du code de l'exercice Vecteur pour permettre d'ajouter des comptes à la volée:
   - Lorsque quelqu'un demande à créer un compte, il devra renseigner son nom ainsi que le solde qu'il y dépose.
   - Le numéro de compte devra être généré automatiquement par l'application de façon à être unique pour chaque compte.
3. Ajoutez un plafond maximum à ne pas dépasser sur les comptes et assurez vous que les erreurs sont correctement traitées (vous pouvez modifier les structures, enums et unions créées dans l'exercice précédent).

4. Transferts d'argent

1. Ajoutez la possibilité pour un utilisateur d'effectuer un transfert d'argent:
    - Pour effectuer un transfert, l'utilisateur renseignera deux numéros de comptes.
    - Le transfert ne sera effectué que si le solde du compte émetteur est suffisant et le solde du compte destinataire ne dépasse pas son plafond.
2. Modifiez le code écrit à l'exercice 2 pour que la fonction transfert puisse renvoyer les bonnes erreurs et qu'elle affiche correctement les informations du transfert demandé et pourquoi il a été refusé.

Conservez votre code pour la partie suivante

5. Annulation d'opérations

L'objectif de cet exercice est de donner à l'utilisateur la possibilité d'annuler la précédente opération.

1. Créez une `struct Operation` qui contiendra un `enum OperationType` et une `union OperationInfo` qui permettront d'enregistrer les trois types d'opérations:
    - Retrait.
    - Dépôt.
    - Transfert.
    - Création de compte.
2. Créez une fonction `demande_operation` qui demande à l'utilisateur quelle opération il veut effectuer et renvoie une `struct Operation`. Modifiez votre code pour utiliser cette fonction.
3. Donnez à l'utilisateur la possibilité d'annuler la précédente opération.
4. Ajoutez la possibilité de supprimer un compte.

6. Pointeur vers une fonction

1. Créez un programme qui contiendra deux fonctions `int plus_deux(int)` et `int fois_trois(int)` et qui demandera à l'utilisateur la quelle utiliser.

   Enregistrez alors un pointeur vers la fonction choisie dans une variable et appliquez la cinq fois d'affilée à la valeur 5.

2. Créez une fonction `float* for_each(int nombre_elems, int* elems, float (*func)(int))`.
    - La fonction acceptera en argument un tableau d'entiers, sa longueur ainsi qu'un pointeur vers une fonction qui accepte un int et renvoie un float.
    - La fonction renverra un tableau de flottants de la même taille contenant les flottants obtenus en appelant la fonction func sur le tableau d'entiers.

Conservez votre code précédent dans un fichier à part pour la suite

Exercices avancés

7. Classes et tables virtuelles

Dans certains langages de programmation (souvent pour les langages objets: C++, C#, Java), le support pour le polymorhpisme est effectué en utilisant une "table virtuelle" ou "vtable".

L'objectif de cet exercice est de modifier le code de gestion de la banque pour utiliser une table virtuelle. Pour cela, il vous est donné un exemple d'utilisation dans le fichier `vehicule.c` .

1. En vous inspirant de cet exemple, transformez l'objet `struct Operation` pour qu'il utilise une table virtuelle au lieu d'utiliser une `enum` et une `union`.
2. Faites de même pour la `struct Result`.

8. Polymorphisme statique

Jusqu'à présent, nous n'avons pu utiliser qu'un seul type pour les éléments de notre structure `vecteur`. Dans d'autres langages, il est facile de demander au compilateur de générer le même code pour plusieurs types (en utilisant des templates en C++, ou des génériques en Rust par exemple).

Pour arriver à un résultat similiaire en C, nous devons faire usage du préprocésseur.

Le préprocesseur est l'étape de la compilation qui s'execute en premier et va modifier le code sans se vérifier son sens. Les directives préprocesseur commencent toujours par un '#'. En voici quelques unes:

- La directive `#include` par exemple va recopier le code d'un fichier à l'endroit ou elle est appelée.
- La directive `#define N 10` va associer le texte `10` au texte `N` et remplacera donc toutes les occurences de `N` par `10` dans la suite du code.
- La directive `#undef N` va supprimer la définition précédente de `N`.
- La directive `#ifdef N` suivie plus loin de `#endif` permet d'inclure le code situé entre les deux instructions si et seulement si `N` est définie.
- L'opérateur ## permet de concaténer deux "mots".

Par exemple:

```
#define A(N) valeur_ ## N

int A(2), A(3) = 5;
```

Si l'on compile le code ci-dessus avec le flag -E (`gcc main.c -E`) il nous montre le résultat de l'étape du préprocesseur:

```
int valeur_2, valeur_3 = 5;
```

Ci-dessous un exemple de code qui permet de définir une structure contenant une liste de valeurs ainsi qu'une fonction pour initialiser cette structure.

```
//////////////////////////
///// fichier list_struct.c
//////////////////////////
// ce code contient une structure qui contient un tableau de 3 variables de types A
// il contient aussi une fonction qui crée une liste avec trois fois le même élément

// la variable A doit être définie avant d'inclure ce code
#define CONCAT_DEFER(A, B) A##B
// on utilise la macro CONCAT_DEFER plutot que directement l'opérateur ## pour s'assurer que
le préprocesseur ne concatène pas des noms de variables avant qu'ils aient été remplacés
#define CONCAT(A, B) CONCAT_DEFER(A, B)
#define LIST(A) CONCAT(List_, A)

typedef struct LIST(A){
    A l[3];
} LIST(A);

LIST(A) CONCAT(LIST(A), _creer)(A val){
    LIST(A) liste;
    liste.l[0] = val;
    liste.l[1] = val;
    liste.l[2] = val;
    return liste;
}

//////////////////////
/// fichier main.c
//////////////////////
#include "stdio.h"
#define A int
#include "list_struct.c"
#define A float
#include "list_struct.c"

int main()
{
    List_int a = List_int_creer(1);
```

```
    printf("%d\n", a.l[2]);
    List_float b = List_float_creer(1.2);
    printf("%f\n", b.l[0]);
    return 0;
}
```

1. Adaptez le code de l'exercice "vecteur" pour vous permettre de facilement créer des listes dynamiques de différents éléments.
2. Assurez vous d'avoir les fonctions: créer, ajouter et libérer.
3. Ajoutez une fonction `afficher` qui affichera tous les elements du tableau dans la console.
    - Vous pouvez définir le format à utiliser de la même façon que le type des éléments.
    - Vous pouvez utiliser `#undef` et `#ifdef` pour vous assurer que la fonction affiche n'existe que si un format d'affichage à été défini.
4. Créez une fonction `for_each` qui applique une autre fonction sur un tableau d'éléments.