Algorithmique et Programmation 1
IMAC 1ere année

TP 8

types structurés

Dans cette séance de travaux dirigés, on présentera la décomposition de code en modules et on travaillera sur les structures.

Pendant le TP, vous écrirez vos codes dans des fichiers source. Ayez le réflexe d'enregistrer régulièrement vos sources. À la fin de la séance, mettez-les dans une archive (commande `tar -zcvf nom_archive fichiers_source`) et rendez-le suivant les instructions données sur e-learning. N'oubliez pas de commenter votre code.

Le sujet est long, ne cherchez pas forcément à le finir, mais plutôt à bien comprendre les premiers exercices. Pensez à consulter les transparents de cours sur elearning.

Exercice 1. (Compilation en modules) – temps prévu : < 30 mins

Dans cet exercice, on montrera comment découper le code source en plusieurs fichiers. L'intérêt de cela devrait être évident (lisibilité, réutilisabilité du code, meilleure gestion des grands projets, ...).

1. Saisir le fichier `code.c` fourni, le regarder, essayer de compiler et de tester le résultat. On va se servir de ce code dans l'exercice.
2. Créer deux nouveaux fichiers : déplacer les définitions des fonctions `puissance` et `partie_decimale` de `code.c` dans `fonctions.c` ; déplacer le reste (donc la ligne de `include...` et la fonction `main`) dans `main.c`.
3. Tenter de compiler naïvement avec `gcc main.c fonctions.c`, observer le résultat.

Celui dépendra de votre compilateur et son réglage, mais il est possible que la compilation réussira sans d'avertissement, sauf que le calcul de la partie décimale ne marchera pas correctement. Récompiler avec l'option `-Wall` et lire l'avertissement (`implicit declaration...`). Le problème est que les deux fonctions qu'on utilise dans `main.c` n'y sont pas déclarées. Le compilateur utilise alors certaines déclarations implicites qui ne servent qu'à obscurcir l'erreur. Il faut ajouter les *prototypes* des fonctions `puissance` et `partie_decimale` dans `main.c`. On pourrait le faire directement, mais il ne serait pas possible de maintenir un tel code en cas de modifications. On fait donc autrement :

4. Créer un fichier en-tête `fonctions.h` : un fichier texte qui contiendra les *prototypes* (pas les définitions !) des fonctions `puissance` et `partie_decimale`. Ajouter une ligne `#include "fonctions.h"` au début de `main.c` et de `fonctions.c`. Recompiler avec `gcc -Wall main.c fonctions.c`.

Cette fois-là, tout devrait marcher. Mais cette commande recompile chaque fois tous les fichiers source. Ce n'est pas nécessaire, on peut procéder comme suit :

5. On compile d'abord séparement le `main.c` avec `gcc -Wall -c main.c` et le `fonctions.c` avec `gcc -Wall -c fonctions.c`. On obtient alors deux fichiers objet, `*.o`, qu'il suffit de relier en appelant `gcc main.o fonctions.o`.
6. Modifier la fonction `puissance` pour qu'elle renvoie toujours la valeur 5. Comme le contenu de `main.c` n'a pas changé, il n'est pas nécessaire de le recompiler. Il suffit de recompiler `fonctions.c` et de refaire l'édition des liens. Tester.
7. Modifier la fonction `puissance` pour qu'elle, en plus, affiche la ligne `"Je refuse de calculer"`. Encore cette fois, pas besoin de recompiler `main.c`. Par contre, il faudra ajouter `#include <stdio.h>` dans fonctions.c pour y pouvoir utiliser la fonction `printf`.

On a donc arrivé à isoler les définitions des fonctions dans un fichier qui peut être compilé séparement et qui peut être reutilisé pour d'autres projets. Dans ce cas, il n'est pas pratique de continuer à diviser le code davantage. Mais avec un projet plus grand, on travaillera avec beaucoup plus de fichiers organisés en plusieurs couches auxiliaires.

Exercice 2. (Nombres complexes)

Dans cet exercice, on utilise les structures pour représenter les nombres complexes en C.

1. Définir la structure

```
typedef struct Complexe{
  float reel;
  float img;
} Complexe;
```

2. Écrire une fonction `initComplexe(Complexe *c, double reel, double img)` initialisant le complexe pointé par `c` grâce à `reel` et `img`. Pourquoi n'a-t-on pas passé directement un complexe plutôt que son adresse en argument de la fonction ?
3. Définir une fonction `afficheComplexe(Complexe c)` affichant `c` sous la forme c.reel + c.img*i.
4. Définir un fonction `Complexe sommeComplexe(Complexe cUn, Complexe CDeux)` retournant la somme des deux complexes passés en arguments.
5. Définir une fonction `Complexe produitComplexe(Complexe cUn, Complexe cDeux)` retournant le produit des deux nombres complexes passés en arguments. On rappelle que le produit de $z = x + iy$ avec $z' = x' + iy'$ est $zz' = (xx' - yy') + (xy' + x'y)i$.
6. (*) Définir une fonction `Complexe conjugue(Complexe c)` retournant le conjugué du nombre complexe passé en argument. On rappelle que le conjugué d'un nombre complexe $z = x + iy$ est égal à $\bar{z} = x - iy$.
7. (*) Définir une fonction `double moduleComplexe(Complexe c)` calculant le module du nombre complexe passé en argument. On rappelle que le module d'un nombre complexe $z = x + iy$ est égal à $\mid z \mid = \sqrt{x^2 + y^2}$. De plus, la fonction racine carrée en C s'appelle `sqrt` et se trouve dans la bibliothèque `math.h`.
8. (*) On rappelle que l'inverse d'un nombre complexe non nul z est égale à $\bar{z} / \mid z \mid^2$. Définir une fonction `inverseComplexe` qui calcule, lorsque c'est possible, l'inverse d'un nombre complexe passé en argument. Vous devrez traiter le cas où la division n'est pas possible.

9. (*) Si vous ne l'avez pas fait, testez vos fonctions dans un `main`. Vous penserez bien à traiter le cas où on essaie de calculer l'inverse d'un nombre complexe qui peut être nul !

Exercice 3. (Fiches d'étudiants)

1. Définir un type structuré `Date` contenant des champs `int jour`, `char mois[10]` et `int annee`.
2. Définir une fonction `void ecrireDate(Date date)` affichant une date sous le format `jour mois annee`.
3. Définir une fonction `void lireDate(Date *date)` initialisant les champs d'une date en scannant leurs valeurs. On ne vous demande pas de tester si la date entrée au clavier est correcte.
4. Définir un type structuré `Fiche` possédant des champs `char nom[20]`, `char prenom[20]`, `Date dateDeNaissance`, `float notes[MAXNOTES]` et `int nbNotes`. L'entier `MAXNOTES` est une constante définie en début de programme, par exemple 4.
5. Définir la fonction `lireFiche(Fiche *fiche)` : les champs correspondant au nom et au prénom son initialisés en scannant leurs valeurs, la date de naissance est initialisée par un appel à la fonction `lireDate` et `nbNotes` est initialisé à `0`.
6. Définir une fonction `ajoutNote` ajoutant une note à la fiche passée en argument. Vous penserez à traiter le cas où le tableau de notes est déjà plein.
7. Définir la fonction `void ecrireFiche(Fiche fiche)` affichant les informations sur la fiche sous forme d'items, c'est-à-dire, quelque chose de la forme :

```
1       - Nom : fiche.nom.
2       - Prenom : fiche.prenom.
3       - Date de naissance : fiche.dateDeNaissance.
4       - Notes : /*lister les notes*/
```

   S'il n'y a aucune note, la dernière ligne affichée sera `- Notes : aucune note.`
8. Écrire une fonction `moyenne` qui calcule, lorsque c'est possible, la moyenne de la fiche passée en argument. Vous devez traiter le cas où la fiche ne contient aucune note.
9. Si vous ne l'avez pas déjà fait, testez vos fonctions dans un `main`. Vous penserez bien à traiter les cas mentionnés plus haut : lorsqu'on chercher à ajouter une note alors que le tableau est déjà plein, lorsqu'on n'a aucune note à afficher au moment de l'écriture de la fiche et lorsqu'on cherche à calculer une moyenne alors qu'aucune note n'a été entrée.

Compléments – pour mieux comprendre

Exercice 4. (Fichiers en-tête)

Il peut y avoir de problèmes si un fichier en-tête est inclus plusiers fois pendant la compilation. On peut assurer que cela n'arrive pas avec les ≪ include guards ≫ :

1. Ajouter tout au début de fichier `fonctions.h` les deux lignes

```
#ifndef __FONCTIONS_H
#define __FONCTIONS_H
```

   et à sa fin la ligne

```
#endif  /* __FONCTIONS_H */
```

Cela suffit. Le compilateur (préprocesseur) va alors traiter le contenu de ce fichier `.h` seulement la première fois qu'il y accède (définissant une constante macro unique vide '__FONCTIONS_H' au chemin). À chaque passage suivant, le test dans la première ligne détectera que '__FONCTIONS_H' est déjà définie et saute directement à la fin du fichier.

Exercice 5. (Fichiers en-tête et structures)

Diviser le code d'exercice 2 en suivant le modèle d'exercice 1. Il faut réfléchir sur l'emplacement de la définition de structure 'Complexe'. Comme on en a probablement besoin pour définir les fonctions ainsi que pour les traiter dans le programme principal, on peut la placer directement dans le fichier en-tête (qui sera inclus dans les deux fichiers .c).

Exercice 6. (Makefile)

Deux fichiers, `makefile1` et `makefile2` sont fournis pour complémenter l'exercice 1. Il s'agit des fichiers makefile : des fichiers texte contenant de règles permettant d'automatiser la compilation.

1. Stocker `makefile1` dans le dossier contenant les fichiers source correspondants à l'exercice 1. Le renommer en `makefile`. Appeler dans le terminal, étant placé dans le dossier correspondant, la commande `make`. La première fois, elle devrait compiler le projet. Appelé la deuxième fois, elle annoncera que tout est à jour. La commande `make clean` effacera les fichiers `.o`.

   Analyser le contenu de ce fichier makefile pour comprendre son fonctionnement.
2. Répéter pour `makefile2`.

Exercice 7. (Règles implicites de make)

Dans certains cas il n'est pas nécessaire de tout spécifier explicitement. En cas d'absence d'instructions détaillées, `make` tentera de suivre certaines règles implicites.

1. Écrire un simple programme (genre Hello world) et le sauvegarder dans `hello.c`. Créer dans le même répertoire un `makefile` contenant une seule ligne :

```
all: hello
```

   On demande alors make de créer un fichier hello, sans lui dire comment. Appeler `make`. Il se rendra lui-même compte qu'il y a un fichier `hello.c` dans le dossier et qu'il est peut-être une bonne idée d'essayer de le compiler en tant que programme en C. Il le fera et produit alors un executable `hello` qui fonctionne (au moins sur ma machine).

2. Certaines variables de makefile ont une signification particulière. Ajouter la ligne suivante dans le makefile et observer que la compilation s'effectue cette fois avec l'option `-Wall` additionnelle (déclarer une nouvelle variable dans `hello.c` sans jamais la utiliser pour le mettre en valeur) :

```
CFLAGS = -Wall
```

On est prêt à régarder le `makefile` fourni avec les exemples de la bibliothèque MLV. On y spécifie les programmes à créer (leurs noms correspondent aux noms des fichiers source `.c`). On définit les variables `CFLAGS` (qui réprésente par défaut les options de compilateur pendant compilation) et `LDFLAGS` (qui concerne les bibliothèques utilisés pour l'édition des liens) pour permettre d'utiliser les fonctions de la bibliothèque MLV. Le reste est assuré par les règles implicites.

3. Compiler un programme exemple de MLV, comme `02_shapes.c`, utilisant le makefile fourni avec la bibliothèque graphique MLV. Observer que la commande affiché en terminal après l'appel de `make`,

```
cc -g -O2 -Wall -Werror `pkg-config --cflags MLV` 02_shapes.c
          `pkg-config --libs MLV` -o 02_shapes ,
```

est consistante avec nos observations sur le makefile.

4. Pour séparer les étapes de compilation et d'édition des liens, tester les deux commandes suivantes :

```
gcc -c -O2 -Wall -Werror `pkg-config --cflags MLV` 02_shapes.c
     gcc `pkg-config --libs MLV` -o 02_shapes 02_shapes.o
```

REMARQUE : L'idée de cet exercice est de donner de l'intuition sur l'emploi de `make`. Il n'a pas pour ambition de présenter cet outil de manière exhaustive et formelle. Pour plus d'information et notamment toute utilisation sérieuse consulter d'autres sources, en particulier directement le manuel de GNU make.