Programmation Avancée

TP n°1: Programmation modulaire

Simon Forest

14 janvier 2021

Exercice 1 : Échauffement avec `Makefile`

Dans cet exercice, on se donne un exemple simple sur lequel on se forme à utiliser les `Makefile`s.

1. Écrire dans un fichier `main.c` un programme simple qui demande en entrée deux entiers et en affiche la somme.
2. Quelle commande faut-il faire pour compiler ce fichier en une étape ? Quelles commandes faut-il faire pour compiler en passant par les fichiers objets ?
3. Écrire un fichier nommé `Makefile` (avec un `M` majuscule) dans le même dossier que le `main.c` et y écrire les directives permettant de compiler en une étape. Que faut-il rajouter pour que l'invocation de `make` sans arguments suffise à compiler le programme ?
4. Maintenant, écrire un `Makefile` qui permet de compiler en deux étapes (en passant par un fichier objet `.o`). On fera en sorte ici aussi que l'invocation de `make` sans arguments suffise à compiler le programme.

Exercice 2 : Géométrie

Dans cet exercice, on s'intéresse à un exemple un peu plus complexe faisant intervenir plusieurs fichiers.

1. Écrire des fichiers `vecteur.h` et `vecteur.c` permettant de décrire des vecteurs 2-dimensionnels ainsi que des fonctions élémentaires associées. On devra avoir par exemple
   - une structure `vecteur` avec deux attributs flottants `x` et `y` ;
   - une fonction `vecteur_depuis_coordonnees` qui produit le vecteur associé à une paire de flottants passés en arguments ;
   - une fonction `vecteur_somme` qui calcule la somme de deux vecteurs passés en argument ;
   - une fonction `vecteur_difference` qui calcule la différence de deux vecteurs passés en argument ;
   - une fonction `vecteur_affiche` qui affiche proprement une structure de type `vecteur` passée en argument.

   Que faut-il mettre dans `vecteur.h` ? Dans `vecteur.c` ?
2. Écrire des fichiers `tests_vecteur.h` et `tests_vecteur.c` dans lesquels on déclarera et définira des fonctions associées aux fonctions de `vecteur.c` et qui effectueront des tests pour voir si les calculs de vecteurs sont corrects. On écrira
   - une fonction `test_vecteur_somme` qui testera sur quelques exemples que le vecteur calculé par `vecteur_somme` est le bon ;
   - une fonction `test_vecteur_difference` qui testera sur quelques exemples que le vecteur calculé par `vecteur_difference` est le bon.

Quel `#include` faut-il faire pour que `test_vecteur.c` ait accès à la structure `vecteur` ainsi qu'aux fonctions définies dans `vecteur.c` ?

3. Écrire un fichier `test.c` avec un `main` appelant à la suite les tests de `test_vecteur.c`.
4. Comment compiler `vecteur.c`, `test_vecteur.c` et `test.c` en une seule étape ? En plusieurs étapes en passant par les fichiers objets ? Écrire un `Makefile` avec plusieurs directives reflétant la dernière méthode.
5. Écrire des fichiers `geometrie.h` et `geometrie.c` contenant des fonctions faisant divers calculs sur des vecteurs :
   - une fonction `scalaire` qui calcule le produit scalaire de deux vecteurs passés en argument ;
   - une fonction `norme` qui calcule la norme d'un vecteur ;
   - une fonction `aire_triangle` qui calcule l'aire du triangle défini par deux vecteurs passés en arguments ;
   - une fonction `aire_parallelepipede` qui calcule l'aire du parallélépipède rectangle défini par deux vecteurs passés en argument ;
   - une fonction `est_rectangle` qui décide si le parallélépipède défini par deux vecteurs passés en argument est un rectangle ;
   - une fonction `est_carre` qui décide si le parallélépipède défini par deux vecteurs passés en argument est un carré.
6. Écrire des fichiers `test_geometrie.h` et `test_geometrie.c` déclarant et définissant des fonctions permettant, comme pour `test_vecteur.c`, de tester sur plusieurs exemples que les fonctions de `geometrie.c` sont correctes. Adapter le `Makefile` pour pouvoir compiler ces fichiers.
7. Écrire un fichier `main.c` avec une fonction `main` qui permettra à l'utilisateur de calculer l'aire associée à un triangle ou à un parallélépipède à l'utilisateur en proposant une interface par le terminal. On demandera d'abord son choix à l'utilisateur, puis on prendra en entrée une paire de vecteurs définissant la figure choisie et on appellera la fonction adéquate de `geometrie.c`. Adapter le `Makefile` pour que seul le programme associé à `main.c` soit compilé. On créera pour les tests une directive `test` permettant de compiler et de lancer le programme associé à `test.c`.

Exercice 3 : Tableaux

Dans cet exercice, on définit plusieurs fichiers permettant la gestion et le tri de tableaux d'entiers.

1. Écrire des fichiers `tableau.h` et `tableau.c` définissant une structure

```
typedef struct {
  int taille;
  int* contenu;
} tableau;
```

   représentant des tableaux d'entiers alloués dynamiquement, ainsi que plusieurs fonctions permettant de les manipuler :
   - une fonction `tableau_creer` qui prend en argument un `tableau*` et un entier `n` et alloue un nouveau tableau de taille `n` pour cette structure ;
   - une fonction `tableau_liberer` qui prend en argument un `tableau*` et libère le pointeur `contenu` de la structure ;
   - des fonctions `tableau_inserer` et `tableau_supprimer` qui prennent en argument un `tableau*` ainsi que d'autres arguments et permettent d'insérer / supprimer un élément à une certaine position du `tableau` passé en argument.

2. Écrire des fichiers `tri.h` et `tri.c` déclarant et définissant des fonctions de tri standards pour les tableaux d'entiers :
   - une fonction `tri_bulles` pour le tri à bulles [1] ;
   - une fonction `tri_qsort` pour le tri Quick Sort ;
   - une fonction `tri_fusion` pour le tri fusion.
3. Quelles sont les complexités de ces fonctions ?
4. Écrire un fichier `main.c` permettant à l'utilisateur de spécifier un tableau à trier ainsi qu'un algorithme de tri parmi les trois ci-dessus.
5. Écrire un `Makefile` permettant de compiler tous ces fichiers ensemble en faisant une compilation par étapes.

1. `https://fr.wikipedia.org/wiki/Tri_%C3%A0_bulles`