Gentille Initiation au langage C

6 février 2024

1 Préambule

Ce document propose un tutoriel pour vous initier au langage C sur 4 séances de TD. Il propose une série d'exercices couvrant une bonne partie du langage C. Ce document est accompagné d'un autre document en-ligne qui fournit les explications nécessaires à la réalisation de ces exercices. Le mot DOC dans ce tutoriel fera toujours référence au document suivant :

`https://homepages.laas.fr/ypencole/ens/poo/langage_c.html`

Pour cette initiation, vous aurez besoin d'un éditeur de texte, d'un terminal sous Linux. Nous vous recommandons l'usage de l'éditeur `code`, ce logiciel contient un éditeur de texte intelligent (avec coloration syntaxique) et embarque un terminal pour lancer les compilations et exécuter vos programmes. Mais vous pouvez utiliser tout autre logiciel.

À lire: Pour commencer, lire **DOC:section 1 Généralités**.

2 À la découverte d'un programme écrit en C

2.1 Analyse d'un code écrit en C

À faire: Télécharger l'archive `programme.zip` présente sur moodle et décompresser cette archive dans un répertoire que vous aurez créé auparavant.

Cette archive contient trois fichiers `programme.c`, `mesfonctions.c` et `mesfonctions.h`. Ces trois fichiers sont des *fichiers sources*, ils contiennent du code écrit en C. Les fichiers de suffixes `.c` contiennent la *définition* d'un ensemble de fonctions (ils décrivent le déroulement de chaque fonction). Les fichiers de suffixes `.h` (h pour *header*) sont des fichiers d'entête qui contiennent la *déclaration* d'un ensemble de fonctions, fonctions qui sont définies dans les fichiers de suffixes `.c`. Les fichiers d'entête permettent d'utiliser des fonctions dans la définition d'autres fonctions qui se trouvent dans différents fichiers sources. Un programme C est décrit comme un ensemble de fonctions et contient nécessairement une *fonction principale* qui s'appelle toujours `main`. La fonction `main` est le point d'entrée du programme, elle décrit comment débute le programme et généralement comment il se termine.

À faire:

— À l'aide de votre éditeur de texte (par ex `code`), ouvrir les trois fichiers.
— Où se trouve la fonction principale de ce programme ? Dans quel fichier ?
— Combien y'a-t-il de fonctions définies ? déclarées ?
— Le langage C exploite une bibliothèque standard de fonctions prédéfinies, pouvez-vous lister l'ensemble des fonctions prédéfinies qui sont utilisées dans ce programme ?
— Le programme décrit dans cette archive est complet et fonctionnel. Sans rentrer dans les détails techniques de C, pouvez-vous globalement décrire le déroulement de ce programme ? Chaque fonction appelle d'autres fonctions, quel est l'enchaînement de ces appels dans le déroulement de ce programme ?

2.2 Compilation et exécution du programme

Après une lecture rapide du code, vous devez globalement avoir compris ce que fait ce programme même si vous ne maitrisez pas du tout les détails techniques de C. Le problème est que le code C que vous comprenez n'est absolument pas compris par l'ordinateur. Un ordinateur ne comprend que des programmes écrits en binaire (succession de 0 et de 1), programme qui est illisible à nos yeux. Ainsi, afin de pouvoir lancer ce programme, nous devons d'abord *traduire* le programme C en un programme binaire. Cette phase de traduction est la *compilation*. Le logiciel en charge de compiler est un *compilateur*. Le compilateur que nous allons utiliser est `gcc`.

À faire: Dans un terminal, placez-vous dans le répertoire où se trouvent les fichiers C (par exemple à l'aide de la commande de terminal `cd`) puis lancez la commande de compilation suivante :

```
gcc -o programme programme.c mesfonctions.c
```

La commande précédente fait beaucoup de choses :

1. Compilation de `programme.c` dans un fichier binaire temporaire `programme.o`.
2. Compilation de `mesfonctions.c` dans un fichier binaire temporaire `mesfonctions.o`.
3. Assemblage/Editions de liens entre `programme.o` et `mesfonctions.o` pour produire le programme binaire complet dont le nom est `programme` (nom donné par l'option `-o programme`).
4. Suppression des fichiers intermédiaires `programme.o` et `mesfonctions.o`

Le programme binaire est prêt à être lancé, il se nomme `programme`.

À faire: Lancez-le dans le terminal par exemple avec la commande suivante :

```
./programme
```

En modifiant la ligne de commande précédente, essayez de faire dérouler le programme de façon différente.

3 Programmation de base en C

3.1 Affichage à l'écran en C, Lecture au clavier

Pour des raisons pratiques, la première étape de votre apprentissage va consister à savoir écrire des messages sur l'écran et à récupérer des informations écrites au clavier au sein d'un programme C.

À lire: DOC:Section 2 Afficher un message à l'écran printf.

À faire: Ecrire un programme dans un fichier `message.c` qui écrit `Bonjour tout le monde!` à l'écran. Il sera constitué de la seule fonction `main` sans paramètre. Dans le terminal, compilez le programme `gcc -Wall -o message message.c` et lancez-le : `./message`.

À lire: DOC:Section 3 La variable et **DOC:Section 4 La chaîne de caractères**.

Nous allons maintenant utiliser `printf` pour faire un message formaté. L'objectif de notre deuxième programme est d'afficher une ligne qui ressemble à celle-ci `Je m'appelle Kevin et j'ai 40 ans.` où `Kevin` aura été préalablement stocké dans une variable de type chaîne de caractères et 40 aura été préalablement stocké dans une variable de type entier (`unsigned`).

À faire: Ecrire un programme dans un fichier `message_format.c` : ce programme déclare dans `main` une variable entière `age` et l'initialise à 40, il déclare également une chaine de caractères `prenom` initialisé avec `Kevin` et affiche `Je m'appelle Kevin et j'ai 40 ans.` en s'appuyant sur les variables `prenom` et `age`. Dans le terminal, compilez le programme `gcc -Wall -o message_format message_format.c` et lancez-le : `./message_format`.

Ce dernier programme n'est pas encore très interactif. On voudrait maintenant que le programme nous propose de donner notre prénom et notre âge au clavier pour l'afficher.

À lire: DOC:Section 5 Gestion du clavier scanf.

À faire: Ecrire un programme dans un fichier `prenom_age.c` : ce programme déclare dans `main` une variable entière `age` et un tableau de caractères `prenom` de taille 256. Avec `printf/scanf` demander son prénom à l'utilisateur et stocker dans la variable `prenom`. Demander son âge et stocker-le dans `age`. Afficher enfin à l'écran, les prénom et âge effectivement stockés dans les deux variables. Exécuter votre programme.

Compétences à acquérir : savoir utiliser basiquement `printf` et `scanf`

3.2 Structure de contrôle conditionel

L'objectif de cet exercice est d'utiliser une première structure de contrôle du langage C : le contrôle conditionnel.

À lire: DOC:Section 5.1 Structure de contrôle conditionnel if/else.

À faire: Reprendre le programme `prenom_age.c` et copiez-le dans `verifier_age.c`. On veut modifier l'affichage final du programme. Si l'âge de la personne est moins de 50 ans alors il faut afficher `Bonjour <prenom>, vous êtes plus jeune que le langage C`. Sinon, il faut afficher `Bonjour <prenom>, vous êtes plus vieux que le langage C`.

3.3 Variables et Itérations

L'objectif de cette section est de vous familiariser avec les itérations en C.

À lire: DOC:Section 6.3 L'iteration avec for.

Nous allons écrire maintenant écrire un programme de conversion de degrés (fahrenheit vers celsius). On considère que l'on dispose de deux variables entières signées `celsius` et `fahr` (`signed`), la formule de conversion entre degrés fahrenheit et celsius est la suivante : `celsius = 5 * (fahr - 32) / 9.` On souhaite que le programme affecte à la variable `fahr` successivement les valeurs `0,20,40,..,220`, calcule la conversion en degrés celsius et affiche le résultat suivant à l'écran :

```
0 -17
20 -6
40 4
60 15
80 26
100 37
...
220 104
```

À faire: Ecrire ce programme `conversion_degre.c` à l'aide d'une boucle `for` et le tester.

On veut également afficher l'état de l'eau en fonction de la température comme suit :

```
0 -17 solide
20 -6 solide
40 4 liquide
60 15 liquide
80 26 liquide
100 37 liquide
...
220 104 gaz
```

> **À faire:** Reprendre le programme `conversion_degre.c` ci-dessus pour qu'il affiche le résultat ci-dessus. Vous utiliserez la structure de contrôle `if else`.

Compétences à acquérir :
— savoir déclarer et initialiser des variables locales de type élémentaire
— savoir manipuler la boucle `for`
Si vous avez le temps, vous pouvez reprendre cet exercice en exploitant les deux autres types de boucles en vous basant sur **DOC:Sections 6.4, 6.5**

3.4 Fonctions, Programmation modulaire

Le programme précédent est fonctionnel mais du point de vue ingénierie logiciel, il n'est pas satisfaisant. En effet, il est composé d'une seule fonction, la fonction principale `main`. Dans cette fonction, on ne distingue pas les aspects calcul des aspects entrées/sorties vers le terminal.

3.4.1 Création de fonctions intermédiaires

Cette section a pour objectif d'améliorer la qualité du code en écrivant plusieurs fonctions simples qui permettront de rendre le code plus lisible et reexploitable.

> **À lire: DOC:Section 7 la fonction (7.1, 7.2 et 7.3.1)**

La première étape va consister à écrire une fonction qui ne fait que la conversion de degrés celsius en fahrenheit et uniquement cela.

> **À faire:** Créer le fichier `conversion_degre_fonctions.c` et y définir la fonction `fahrenheit_vers_celsius`. Cette fonction a un seul paramètre (la température en degré fahrenheit) et son type de retour est un entier (`signed`). Son objectif est de retourner la température en celsius issue de la conversion de la température en fahrenheit donné en paramètre.

Le fichier `conversion_degre_fonctions.c` ne constitue pas encore un programme complet (pas de fonction `main`) donc vous ne pouvez pas le compiler pour en faire un programme binaire et le tester. Néanmoins, vous pouvez tester si vous avez bien écrit la fonction `fahrenheit_vers_celsius` avec le compilateur avec la commande suivante : `gcc -c conversion_degre_fonctions.c`

Une fois que cette fonction a été écrite, nous allons en écrire une deuxième qui va se charger d'écrire à l'écran la température en celsius, puis en fahrenheit, puis l'état solide/liquide/gazeux.

À faire: À la suite de `fahrenheit_vers_celsius` dans le fichier `conversion_degre_fonctions.c`, écrire la fonction `affiche_conversion`. Cette fonction a un paramètre (la température en degrés celsius). Son type de retour est vide (`void`). Cette fonction affiche la valeur du paramètre, sa conversion en fahrenheit et l'état solide/liquide/gazeux associé pour cette valeur. La fonction `affiche_conversion` doit appeller la fonction `fahrenheit_vers_celsius`.

Pour tester la syntaxe cette nouvelle fonction : `gcc -c conversion_degre_fonctions.c`

Enfin nous pouvons compléter le programme en introduisant la fonction `main`. Cette fonction reprend le même principe que dans le fichier `conversion_degre.c` mais le bloc itératif ne fait qu'appeler la fonction `affiche_conversion`.

À faire: À la suite de `affiche_conversion`, completer le programme en y ajoutant la fonction `main`. Tester votre programme.

Compétences à acquérir
- Savoir définir une fonction quelconque et l'utiliser dans une autre.
- Savoir décomposer une fonction en fonctions plus simples.

3.4.2 Programmation modulaire (si vous avez le temps)

À lire: DOC:Section 7.4 Déclaration d'une fonction, programmation modulaire

On désire reprendre le programme précédent et le rendre plus modulaire. La fonction principale serait dans un fichier `conversion_degre_modulaire.c` et les deux fonctions `affiche_conversion` et `fahrenheit_vers_celsius` serait dans un fichier `conversion_degre_fonctions.c`.

À faire: En vous appuyant sur le contenu de la section 7.4 (déclaration de fonctions), rendre ce programme modulaire.

Compétences à acquérir
- Savoir déclarer des fonctions dans des fichiers d'entête.
- Savoir utiliser la directive `#include` pour utiliser les fichiers d'entête

3.5 Tableaux (à démarrer au plus tard au début du TD3)

Dans cette section, nous allons utiliser des tableaux C dans nos programmes. Dans cet exercice, nous allons écrire un programme `analyse_adn`.

À lire: DOC:Section 8 Le tableau (8.1, 8.2, 8.3)

À faire:
- créer un nouveau fichier `analyse_adn.c` avec une fonction `main()` vide.
- Dans `main()`, définir le tableau de caractères suivant : `'A' 'C' 'G' 'T' 'G' 'C' 'A' 'C' 'C' 'A' 'C' 'T' 'G'`

À l'aide d'un parcours de tableau, afficher son contenu à l'écran.

On souhaite maintenant compter le nombre d'occurrences de chaque lettre dans la séquence précédente. Le principe est de parcourir le tableau pour compter le nombre de A, C, T et G. Pour cela il nous faut 4 compteurs. Ces 4 compteurs seront stockés dans un tableau d'entier. Pour distinguer si un élément du tableau est un A, ou un C, ou un T, ou un G, on pourrait utiliser une succession de `if..else` mais ici il est préférable d'utiliser un `switch` (il y a 4 cas).

À lire: DOC:Section 6.2 Structure de sélection de cas switch

À faire:
- créer un tableau de 4 entiers et initialiser tous les éléments à 0.
- à l'aide d'un parcours de tableau et d'un `switch`, stocker dans le tableau le nombre de A, C, T, G
- afficher le résultat à l'écran :

```
A: 3
C: 5
G: 3
T: 2
```

Compétences à acquérir :
- savoir déclarer, initialiser, parcourir et modifier un tableau.
- savoir utiliser un `switch`

3.6 Structure

Dans cet exercice, nous allons définir et exploiter un premier *type structuré*.

À lire: DOC:Section 9 La structure

Notre objectif ici est tout d'abord de définir le type structuré `nucleotide`. Ce type est constitué de trois membres. Le premier membre s'appelle `lettre` de type caractère. Le deuxième membre s'appelle `nom`, un tableau de 9 caractères, et enfin le dernier membre est un entier non signé nommé `occurrences`.

À faire: Dans un nouveau fichier `adn_structure.c`, définir le type `nucleotide`.

On va maintenant déclarer et initialiser quatre variables de type `nucleotide` dans notre programme.

À faire: Dans `adn_structure.c`, dans la fonction `main()`, déclarer et initialiser 4 variables de type `nucleotide` :

1. variable `adenine` : sa lettre est A, son nom est `Adenine` ;
2. variable `guanine` : sa lettre est G, son nom est `Guanine` ;
3. variable `thymine` : sa lettre est T, son nom est `Thymine` ;
4. variable `cytosine` : sa lettre est C, son nom est `Cytosine` ;
5. les occurrences de chaque variable sont initialement à 0.

Notre objectif maintenant est de reprendre le même principe que dans le programme `analyse_adn.c` mais au lieu de stocker les occurrences dans un tableau d'entiers, l'objectif ici est de compter les occurrences de chaque nucléotide et de mettre à jour le membre `occurrences` de la bonne variable.

À faire: Compléter le programme de `adn_structure.c` pour compter le nombre d'occurrence sur le même tableau de caractères `'A' 'C' 'G' 'T' 'G' 'C' 'A' 'C' 'C' 'A' 'C' 'T' 'G'` que précédemment. En se exploitant tous les membres des variables de nucléotides, le programme doit afficher au final :

```
Adenine (A): 3
Cytosine (C): 5
Guanine (G): 3
Thymine (T): 2
```

Compétences à acquérir :
— savoir définir un type structuré
— savoir déclarer, initialiser, consulter et modifier une variable de type structuré.

Pour aller plus loin (si vous avez le temps, les sections 4-5 suivantes sont plus importantes que cette partie)

Chaque nucléotide a une composition chimique unique faite de carbone, hydrogène, azote et oxygène.

`Adenine:` $C_5H_5N_5$

`Cytosine:` $C_4H_5N_3O$

`Guanine:` $C_5H_5N_5O$

`Thymine:` $C_5H_6N_2O_2$

On veut stocker la composition chimique de chaque nucléotide et pouvoir comptabiliser le nombre d'atomes respectifs de carbone, hydrogène, azote et oxygène dans la séquence ci-dessus.

À faire: Étendre le programme précédent comme suit :

1. Définir un type structuré `composition_chimique` contenant 4 membres entiers (un par type d'atome).
2. Étendre le type `nucleotide` avec un membre `composition` de type `composition_chimique`.
3. Compléter l'initialisation des variables `adenine,....` avec leur composition respective.
4. Afficher en plus du nombre d'occurrences de nucleotides, le nombre d'atomes de carbone, hydrogène, azote et oxygène contenu dans la séquence.

```
Carbone: 62, Hydrogène: 70, Azote: 44, Oxygène: 15
```

4 Pointeurs (à démarrer au plus tard au début du TD4)

Les *pointeurs* est ce qui fait la force du langage C mais c'est aussi ce qui rend la maîtrise de ce langage difficile. Pour comprendre la notion de pointeur, il faut comprendre le principe général du fonctionnement de la mémoire d'un ordinateur.

4.1 Adresse mémoire et taille mémoire

À lire:
DOC:Section 10 La mémoire

Chaque variable est associée à une zone mémoire qui est à un emplacement particulier (une adresse) et qui a une taille particulière. Avec l'exercice suivant, on va regarder cela.

À faire: Ecrire un programme `adresse_variable.c` qui déclare des variables de type `char`, `int`, `long`, `long long`, `long double`. Reprendre également le type structuré `nucleotide` et déclarer une variable de ce type. Pour chaque variable, vous afficherez son adresse à l'écran et sa taille avec `printf` comme suit :

```
printf("Adresse variable = %p, taille variable = %zu\n", &variable,
                        sizeof(variable));
```

Analyser les résultats.

Maintenant, on va regarder les tableaux :

À faire: Dans le même programme `adresse_variable.c`, déclarer un tableau de 20 entiers et un tableau de 4 nucléotides. Pour chaque tableau, vous afficherez son adresse à l'écran et sa taille avec `printf` comme suit :

```
printf("Adresse tab = %p, taille tab = %zu\n", tab, sizeof(tab));
```

(Pas de '`&`' avant `tab` ici).
Analyser les résultats. Modifier le code pour qu'il affiche également le nombre d'éléments dans les deux tableaux.

4.2 Pointeur : une variable qui stocke des adresses mémoires

À lire:
DOC:Section 11 Le pointeur (11.1 et 11.2)

Pour se convaincre qu'un pointeur ne fait que stocker une adresse mémoire, on va regarder son contenu.

À faire: Dans un programme `pointeur.c`, dans une fonction `main()`, déclarez deux variables `entier1` et `entier2` de type `int`. Avec un `printf`, afficher leur adresse respective. Déclarez maintenant un pointeur `pentier` et affectez-lui l'adresse de `entier1`. Affichez le contenu de `pentier` et sa taille :
`printf("pentier = %p, taille de pentier= %zu",pentier,sizeof(pentier));`

Affectez-lui ensuite l'adresse de `entier2` et réaffichez `pentier`. Conclusion ?

Maintenant on va utiliser le déréférencement des pointeurs pour accéder et modifier le contenu d'autres variables.

À faire: Dans le programme `pointeur.c`, initialiser `entier1` à 40 et `entier2` à 22. Après avoir affecté à `pentier` l'adresse de `entier1`. Afficher le contenu de `*pentier`. Affecter à `*pentier` la valeur `2 * *pentier`. Ensuite, affecter à `pentier` l'adresse de `entier2` et afficher le contenu de `*pentier`. Affecter à `*pentier` la valeur `2 * *pentier`. Au final que contiennent les variables `entier1` et `entier2` ?

4.3 Pointeur sur un type structuré

À lire:
DOC:Section 11 Le pointeur (11.3)

Pour illustrer la notion de pointeur sur un type structuré, on va reprendre la structure `nucleotide`.

À faire: Copier le programme `adn_structure.c` dans un nouveau fichier `adn_structure_pointeur.c`. Dans ce nouveau fichier, écrire la fonction `void affiche_nucleotide(nucleotide * nuc)` dont l'objectif est d'afficher le nucléotide (nom, lettre, occurrences) qui est pointé par `nuc`. Remplacer l'ancien affichage issu de `adn_structure.c` par l'utilisation de la fonction `affiche_nucleotide` dans la fonction `main` de `adn_structure_pointeur.c`.

À lire: DOC:Section 11 en entier

5 Allocation dynamique

À lire: DOC:Section 12 dans sa totalité

À faire: Dans un fichier `personne.c`, copiez le programme qui se trouve **DOC:Section 12.5**. Analysez-le et repérez en particulier les instructions liées à des opérations d'allocation/désallocation dynamique de mémoire. Compilez-le et exécutez-le comme d'habitude.

Nous allons maintenant utiliser un nouvel outil : `valgrind`. Cet outil analyse la gestion mémoire d'un programme en cours d'exécution (pratique pour repérer des erreurs dans la gestion mémoire).

À faire: Lancer le programme `personne` que vous venez de compiler avec `valgrind` avec la commande suivante dans le terminal : `valgrind ./personne`. Regarder ce que valgrind affiche. Puis dans `personne.c`, mettre en commentaire la ligne `jacques = liberer_personne(jacques);`. Compilez à nouveau le programme. Y'a-t-il une erreur de compilation ? Exécutez le programme sans valgrind ? Y'a-t-il une erreur d'exécution ? Relancez `valgrind ./personne`. Que remarquez-vous ? Conclusion ?

6 Projet final (si vous avez le temps)

Ce projet pourra s'écrire dans un seul fichier `notes.c` mais vous pouvez le faire de façon plus modulaire et l'améliorer comme bon vous semble. Il fait appel à de nombreuses notions que vous avez vues au cours de cette initiation.

À faire: Ecrire le type structuré `etudiant` composé de deux membres : un prénom de type `char *` et une note de type `int`.

À faire: Ecrire une fonction `etudiant * creer_etudiant(char * pn);` qui alloue dynamiquement de la mémoire pour un étudiant et initialise le membre `prenom` de l'étudiant à `pn`. Le membre `note` devra être initialisé à -1.

À faire: Dans la fonction `main()`, déclarer un tableau `promo` de 7 pointeurs d'étudiants (c'est à dire que les éléments de `promo` sont des pointeurs). À l'aide de la fonction `etudiant * creer_etudiant(char * pn);`, on va créer 7 étudiants dont les prénoms sont les suivants : `Prof, Simplet, Dormeur, Grincheux, Atchoum, Joyeux, Timide`. Le tableau `promo` devra contenir les pointeurs sur les sept étudiants créés.

On doit maintenant mettre en œuvre un moyen interactif de rentrer les notes de POO pour cette promo.

À faire: Ecrire une fonction `void donner_note(etudiant * e)`. Cette fonction demande à l'utilisateur de rentrer la note de POO pour l'étudiant `e` (avec `printf/scanf`) et initialise le membre `note` avec. Dans la fonction `main()`, itérez sur le tableau `promo` pour demander la note à tous les étudiants.

À faire: Ecrire une fonction `char fiche_jury(etudiant * e)` qui renvoie `'A'` si la note de l'étudiant est entre 14 et 20, `'B'` si entre 10 et 14 exclu, `'C'` si entre 7 et 10 exclu, `'D'` si entre 0 et 7 exclu et `'E'` dans tous les autres cas (ce cas code l'absence d'un étudiant à l'examen). Utiliser cette fonction dans le `main` pour afficher les informations officielles pour le jury de cette promo, par exemple :

```
Prof: A
Simplet: D
Dormeur: Absent
...
```

À faire: (difficile) Pour le moment, ce programme est plein de fuite de mémoires (essayez `valgrind` dessus). Il faut y remédier. Pour cela, écrire la fonction `void c_est_la_quille(etudiant ** promo)` qui doit libérer la mémoire associée à tous les membres du tableau `promo` (rappel `promo` est un tableau de pointeurs sur des étudiants, c'est donc un pointeur sur des pointeurs (voir doc section 11.4)).