TD 01 Programmer en C

Fait au TD01.

Contents

1 Généralités, exemple

1.1 Pourquoi programmer en C ?

Ce cours concerne la **programmation système**.

Distinction **programmation d'application** (applications qui rendent service à des utilisateurs) et **programmation système** (outils et bibliothèques destinés aux programmeurs).

Les bibliothèques mettent à disposition une **interface de programmation** (Application Programming Interface), une liste de fonctions que l'on peut appeler.

Le système d'exploitation UNIX est écrit en C, qui est un langage de programmation de bas niveau, et son API est utilisable en langage C. C'est une bonne raison.

Autre raison : la connaissance du langage C est incontournable dans la formation d'un développeur décent.

1.2 Historique

Langage développé conjointement avec UNIX à partir de 1972 par Dennis Ritchie et Ken Thompson. Influencé par BCPL et B. Popularisé par le livre "The C Programming Language" de Brian Kernighan, et Dennis Ritchie (1978). Normalisation en 1989 (C "ANSI"), 1990 (ISO), évolution C99, puis C11.

1.3 Un exemple

```c
// fichier hello.c, premier exemple

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Hello World!n");
    return EXIT_SUCCESS;
}
```

En gros :

- le programme affiche un message et saute à la ligne
- `printf` : affichage formatté - on verra plus loin - sur la sortie standard.
- il s'arrête

1.4 Explications détaillées

1. un programme C contient une fonction `main`, qui est exécutée au lancement.
2. le code contient des **appels** à des fonctions dont la déclaration se trouve dans des "fichiers d'entetes" (headers) avec suffixe .h.
3. C'est le cas de `printf`, qui fait partie de la bibliothèque des entrées-sorties C (standard input output), et qui est déclarée dans `stdio.h`
4. Il faut inclure ces fichiers d'entête : `#include <....>`. Notation "chevrons" pour les bibliothèques systeme.
5. La fonction `main` retourne un code qui indique au système le succès ou l'échec du programmation en fin d'exécution.
6. La constante `EXIT_SUCCESS` est définie dans la bibliothèque `stdlib.h` (bibliothèque standard)

2 Prise en main de VS-code

Vue d'ensemble :

- on va entrer le source dans un fichier `hello.c` avec un **éditeur de textes** (Visual Studio Code)
- avec le **compilateur** `gcc` , on va construire - si tout se passe bien - un exécutable nommé `hello`
- puis on va lancer la commande `hello`

2.1 Préalable : configurer Visual Studio Code

Les commandes pour configurer Code :

```
# C/C++ for Visual Studio Code (IntelliSense, code browsing, debugging, ...)
code --install-extension ms-vscode.cpptools

# Intellisense for GNU C/C++
code --install-extension austin.code-gnu-global
```

2.2 Prise en main

- lancer `code`
- taper le fichier `hello.c`
- ouvrir un teminal
- compiler par `gcc hello.c`
- exécuter `./a.out`
- Avec les bonnes options :

```
gcc --std=c11 -Wall -Wextra -pedantic hello.c -o hello
./hello
```

3 Utilisation d'un Makefile

Pour éviter de retaper cette longue commande à chaque fois. Et mieux gérer les compilations.

3.1 Contenu

Dans le même répertoire, on ajoute un fichier `Makefile` contenant les lignes

```
CFLAGS = --std=c11 -Wall -Wextra -pedantic

hello: hello.o
```

3.2 Utilisation, explications

Taper `make`.

Explications :

- `Makefile`, fichier décrit ce qu'il faut faire pour compiler un projet
- celui-ci contient une définition de variable `CFLAGS`, et une cible `hello`
- la commande `make` demande la fabrication de la première cible

- la ligne dit explicitement que `hello` est fabriqué à partir de `hello.o`
- la commande `make` en déduit que `hello` est obtenu par "édition des liens" de `hello.o` avec la bibliothèque standard.
- comme il existe un fichier `hello.c`, elle induit que `hello.o` est obtenu par compilation du programme C
- appel du compilateur C (`cc` alias `gcc`) en utilisant les options figurant dans `CFLAGS`

Remarques :

- si relance la commande `make`, celle-ci constate que le fichier `hello` est présent n'a pas besoin d'être recompilé
- déduit à partir des **dépendances** connues : `hello` fabriqué à partir de `hello.o` obtenue à partir de `hello.c`, qui n'a pas changé
- constatation faite à partir des dates de dernière modification.

4 Compilation séparée

Un programme d'une certaine taille va être composé de plusieurs "modules", qu'on va compiler séparément (et qui serviront éventuellement dans plusieurs exécutables).

4.1 Un exemple

Principe : pour cet exemple, on "sort" le `printf` dans une autre fonction, qui sera définie dans une autre ficheir source.

Etapes

1. On remplace la fonction `main` par un appel à une fonction

```
// hello.c
...
int main(void)
{
   dire_bonjour();                  <----
   return EXIT_SUCCESS;
}
```

2. La fonction `dire_bonjour` est définie dans un autre fichier `salutations.c`

```
// salutations.c
#include <stdio.h>
...
void dire_bonjour()
{
   printf("Bonjour !n");
}
```

Elle fait référence à `printf`, ce qui explique l'inclusion de `stdio.h`.

3. La fonction `main` fait référence à `dire_bonjour`, il fait donc qu'elle en "importe" la déclaration.

Pour cela on crée un fichier d'entête contenant le **prototype** de la fonction (sans le corps)

```
// salutations.h
...

void  dire_bonjour();
```

et le fichier `hello.c` “importe” cette déclaration

```
// hello.c

#include <stdlib.h>
#include "salutations.h"            <----

int main()
{
  ...
}
```

4. Les informations concernant les paramètres de la fonction (ici : aucun) et le type de retour (void = aucun) sont présentes à deux endroits: `salutations.c` et `salutations.h`. Pour que le compilateur vérifie la cohérence, dans `salutations.c`, on inclut `salutations.h`

```
// salutations.c

#include "salutations.h"            <----

void  dire_bonjour()
{
   printf("Bonjour !n");
}
```

4.2 Le Makefile

Le Makefile évolue un peu.

1. L'exécutable `hello` est obtenu en réunissant deux fichiers .o et la bibliothèque

```
hello: hello.o salutations.o            <--- ajout
```

On peut déja constater en lançant `make` que l'exécutable est reconstruit correctement en trois commandes compilation des deux fichiers .c, et édition des liens. Si on modifie un des fichiers, l'autre n'est pas recompilé.

2. il se trouve que `salutations.h` est inclus par les deux sources C. Si on modifie ce fichier d'entête (qui pourrait contenir des constantes utiles), il faut reconstruire les fichiers .o. Le `Makefile` ne peut pas le deviner : il est nécessaire d'ajouter des dépendances explicites

```
hello.o       : salutations.h    # lignes à ajouter
salutations.o : salutations.h
```

On peut tester : si on modifie `salutations.h` par exemple pour y ajouter une autre déclaration, ou un commentaire, la commande `make` va relancer les compilations.

4.3 Complément : éviter les inclusions multiples

Une précaution standard dans les fichiers d'entête éviter les problèmes d'inclusion multiple.

La situation qui pose problème, c'est si un fichier `A.c` fait un `include` de `B.h` et `C.h`, qui incluent eux-même `D.h`.

On verra les détails plus tard.

```c
// fichier salutations.h

#ifndef SALUTATIONS_H
#define SALUTATIONS_H

void dire_bonjour();

#endif
```

On utilise ici une variable du préprocesseur dont le nom `SALUTATIONS_H` dérive, par convention, du nom de fichier.

5 Entrées sorties, adresses.

5.1 Exemple

Un programme qui lit des infos, et affiche le résultat d'un calcul

```c
// demande le nom
char nom[50];
...
// demande l'année de naissance
int annee;
...
// affiche le nom et l'age

printf("Bonjour %s, vous avez %s ansn", nom, 2018 - annee);
```

- `nom` est un tableau de 50 caractères, dans lequel on mettra une chaine. On verra sa représentation plus tard.
- `annee` est un entier.

5.2 L'affichage

L'instruction `printf` fait de l'affichage formatté

- son premier paramêtre est une chaine qui indique le **format** : allure générale de ce qu'il faut afficher, avec des "trous" %s pour les chaines, et %d pour les nombres en décimal.
- les autres paramètres sont les **valeurs** à afficher

Pour le format `%s`, le paramètre attendu est l'emplacement de la chaine, qui est ICI l' **adresse du début** du tableau (pas sa valeur). **A savoir** : Le nom d'une variable *tableau* représente l'adresse de son premier octet.

Pour %d, on fournit la valeur d'une expression (qui peut être réduite à une variable).

5.3 La lecture

Faire une lecture, c'est analyser de qui a été tapé, et placer les valeurs correspondantes quelque part.

La fonction `scanf` fait la lecture, avec un format comme premier paramètre, et ensuite l'**adresse** de la variable (il peut y en avoir plusieurs) où il faut placer la valeur lue. Cette adresse désigne un emplacement en mémoire.

5.3.1 Entier

```
int annee;
scanf("%d", & annee);
```

l'opérateur `&` (address of) est utilisé pour la fournir à `scanf`(fonction de lecture) l'adresse de la variable `annee`,

5.3.2 Chaine

Le principe est le même pour lire une chaine

```
char nom[50];
scanf("%s", nom);
```

sachant que `nom` **est** une adresse. Donc il n'y a pas à mettre de "address-of".