CRO TD1

First C program, compilation tools (`gcc, make`)
(1 séance sur machine), 17 juillet 2024

En complément de ce sujet se trouvent sur Moodle, les transparents présentant les notions vues dans ce TD. Pensez à les télécharger et les parcourir avant de commencer.

1 Premier programme C

Nous allons commencer par mettre en place l'environnement qui vous permettra de programmer en C. Cet environnement consiste simplement à avoir :

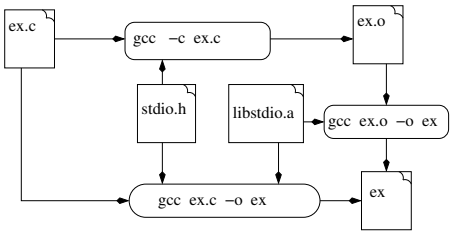
- Un éditeur pour la programmation (`emacs` ou `vi` pour les plus vieux), éviter gedit.
- Connaître la commande de compilation : `gcc`
- Connaître les bases de l'outil `make`

On rappelle ci-dessous le processus de compilation d'un programme C, par exemple le programme `ex.c` illustré à gauche :

```
#include <stdio.h>

int main()
{
  printf("hello World\n");

  return(0);
}
```



QUESTION 1 ▶ Tapez le programme C ci-dessus dans un fichier texte que vous nommerez `hello.c` (si vous ne comprenez pas ce qu'est censé faire ce programme, appelez l'enseignant).

QUESTION 2 ▶ Compilez ce programme avec la commande :
`gcc hello.c -o hello`
Débuggez s'il y a un message d'erreur. Vérifiez bien que vous avez un fichier exécutable `hello` qui a été créé en tapant :
`file hello`
Quel est le format du binaire ?

QUESTION 3 ▶ Exécutez ce programme en tapant :
`./hello`

2 Mon premier Makefile

Nous allons apprendre à utiliser systématiquement un outils d'aide à la compilation, l'outil `make`
QUESTION 4 ▶ Créez un fichier intitulé `Makefile` dans lequel vous taperez les lignes suivante :

```
all: hello

hello: hello.c
	gcc hello.c -o hello

clean:
	rm -f hello
```

Attention, les ligne 4 et 7 doivent commencer par un caractère de tabulation (et non pas 8 espace), c'est indispensable pour la syntaxe de `make`
Ce fichier `Makefile`, traduit en français veut dire :

- Lorsqu'on m'appelle (i.e. commande `make`), je dois réaliser la *cible* `hello` (ça c'est la première ligne).
- Pour réaliser la cible `hello`, je dois avoir la cible `hello.c` (ligne 3, on dit : "`hello` dépend de `hello.c`).
- De plus (ligne 4), si cette cible `hello.c` est plus récente que le `hello` que j'ai, je dois exécuter la commande indiqué à la ligne suivante (après le caractère de tabulation) : `gcc hello.c -o hello`.
- Enfin, pour réaliser la cible `clean`, je doit faire la commande `rm -f hello`.

QUESTION 5 ▶ Exécuter la commande :
`make`
Comprenez vous ce qu'il se passe ? Exécutez la commande : `make clean`, puis `make` à nouveau.

Principe de la commande `make`

Le fichier `Makefile` est le fichier de configuration par défaut de la commande `make` (on peut changer le nom du fichier de configuration avec l'option `-f` de `make`). La commande `make` va chercher à réaliser la première cible du `Makefile`. En général on met la cible `all` en premier et donc en général la commande `make` est équivalente à `make all`, c'est à dire qu'elle essaye de réaliser la cible `all`. Pour réaliser une cible il faut réaliser ses dépendances (ici `hello`). Les cibles peuvent être des fichiers (comme `hello` par exemple), ou pas (comme `clean` par exemple). La cible `hello` *dépend* de la cible `hello.c`.

L'outil `make` vérifie que la date de création de `hello.c` **est postérieure** à la date de création de `hello` (et oui, sinon il n'y a pas besoin de recompiler, on l'a déjà fait), c'est le grand intérêt d'un tel outil (en dehors du fait de ne pas être obligé de taper la commande `gcc` à chaque fois) : il n'exécute que les commandes de compilation nécessaires après un changement dans les fichiers sources, ce qui fait gagner beaucoup de temps par rapport à une recompilation de tous les fichiers.
Pour plus de détail sur `make`, lire les slides sur Moodle associés à ce TD.

QUESTION 6 ▶ Refaites la commande `make`, il ne devrait rien se passer. Modifiez le fichier `hello.c` (rajoutez des espaces, ou mieux : un commentaire du style `/* mon premier commentaire C */`) et retentez un `make`.

QUESTION 7 ▶ Allez lire les transparents associé au TD1 sur le moodle du cours CRO, ils expliquent un peu mieux la commande `make` (transparents 11 à 15 au moins, les suivants sont pour une utilisation avancée de `make`). Demander à l'enseignant ce que vous ne comprenez pas.

3 Compilation et édition de lien

Comme vous l'avez vu sur le schéma de compilation vu en cours (et redonné au début de ce TD), on a deux chemins pour passer de `hello.c` à `hello` :

- le chemin direct comme on a fait en section **??** : `hello.c`→`hello`.
- le chemin en deux étapes, d'abord on produit un fichier **objet** (`hello.o`), puis un fichier **exécutable** (`hello`)

QUESTION 8 ▶ Tapez les commandes correspondantes de compilation (l'option `-c` de gcc permet de spécifier que l'on produit un fichier objet et non pas un executable : on ne fait pas l'édition de lien) :

```
gcc -c hello.c -o hello.o
gcc hello.o -o hello
```
Comprenez bien ce qu'il se passe, demandez à l'enseignant si vous avez un doute

QUESTION 9 ▶ Modifiez votre `Makefile` pour que la compilation se fasse en deux étapes (attention, il faut que vous rajoutiez une cible pour cela).
La compilation en deux étapes, ou *compilation séparée*, est primordiale pour réduire le temps de compilation des gros logiciels. On ne fait que la compilation (la première étape s'appelle aussi la *compilation*, c'est un peu déroutant) du fichier que l'on a modifié et on fait l'**édition de lien** (c'est le nom de la deuxième passe) avec tous les fichiers objets.

QUESTION 10 ▶ Sachant que l'édition de lien prend beaucoup moins de temps que la compilation, quel est, selon vous, l'intérêt de faire de la compilation séparée (on parle *compilation séparée* parce que l'on sépare la compilation des différents fichiers sources et l'édition de lien).

☞ **Validez avec un enseignant.**

4 Variables et Types en C

Les variables en C

La notion de variable est très importante en programmation. Du point de vue *sémantique*, une variable est une entité qui contient plusieurs information :

- Une variable possède un **nom**, on parle d'*identifiant*
- Une variable possède une **valeur** qui change au cours de l'exécution du programme
- Une variable possède un **type** qui caractérise la nature des valeurs qu'elle peut prendre.

Du point de vue pratique, une variable est **un moyen mnémotechnique pour désigner une partie de la mémoire**.
En C les noms des variables sont composés de la manière suivante :
une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres (sauf en début de nom),
- le "blanc souligné" (_).

Un programme C peut avoir des variables globales ou des variables locales à une fonction. Il se présente de la façon suivante :

directives au préprocesseur
déclarations de variables globales
fonctions secondaires

int main()
{ *déclarations de variables internes*
instructions
}

```
#include <stdio.h>
//pas de variable globale ici

int main()
{
  int year;

  year=2018;
  printf("hello %d World\n",year);
  return(0);
}
```

QUESTION 11 ▶ Que fait le programme C ci-dessus ? Modifiez votre programme précédent pour qu'il devienne celui-ci et compilez le avec `make`.

Les types de base en C

En C, les variables doivent être *déclarées* avant d'être utilisées, on dit que C est un **langage *typé***. Les types de base en C sont désignés par des *spécificateurs de type* qui sont des mots clefs du langages, il y en a essentiellement 3 :

— les caractères (`char`),

— les entiers (`int, short, unsigned long`)

— les flottants (nombres réels, `float, double`).

Il n'y a pas de type booleen, ils sont codés par des `int` (`False` étant 0, toutes les autres valeurs étant `True`)
Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs éventuellement initialisés séparés par une virgule est une déclaration. Par exemple :

```
int a;     //declaration d'une variable a entiere, sans initialisation
int b = 1, c;  //declaration des variables b et c. b est initialise a 1
double x = 2.38e4; //declaration et initialisation de x de type double
char message[80];  // declaration d'une variable message
                   //  de type 'tableau de 80 caracteres' (cf next TD)
```

5 Fichiers `.h` et `.c`

La notion de *bibliothèques* (*library* en anglais, si bien que l'on fait souvent l'angliscisme de les appeler *librairies*) est très importante : on réutilise des fonctions programmées par d'autres personnes.
En C, contrairement à d'autres langages comme Python, il y a assez peu de bibliothèque *standard* que l'on trouvera sur tous les systèmes. On les rencontrera au fur est à mesure. Les plus importantes pour l'instant sont la **librairie C** (`libc`) qui permet de faire les opérations élémentaires sur le système et la **librairie d'entrées/sortie** (`libstdio`) qui permet de faire des entrées/sortie dans les programmes.
Assez rapidement, nous allons nous habituer à programmer nous-mêmes nos programmes en utilisant plusieurs fichiers source. Pour cela on aura besoin d'un nouveau type de fichier : **les fichiers .h** qui contiennent les en-têtes de fonction ainsi que les définitions de type.

QUESTION 12 ▶ Modifiez le nom de la fonction `main` dans le fichier `hello.c`, appelez là `hello` et créez les nouveaux fichiers `hello.h`, `Makefile` et `main.c` qui sont les suivants :

Fichier `hello.h`

```
int hello();
```

Fichier `Makefile`

```
all: main

main: main.o hello.o
        gcc main.o hello.o -o main

hello.o: hello.c hello.h
        gcc -c hello.c -o hello.o

main.o: main.c hello.h
        gcc -c main.c -o main.o

clean:
        rm -f main *.o
```

Fichier `main.c`

```
#include <stdio.h>
#include "hello.h"

int main()
{
  int erreur;

  erreur=hello();

  if (erreur!=0)
    printf("Error in hello\n");

  return(0);
}
```

QUESTION 13 ▶ Vérifiez d'abord que la compilation se passe bien, puis étudiez bien chaque fichier, vous devez être capable de répondre aux questions suivantes :

— À quoi sert l'include de `hello.h` dans `main.c` ?
— Pourquoi certains includes sont entre des chevrons (< et >) et d'autres entre des guillemets ?
— Faut-il mettre `#include "hello.h"` au début du fichier `hello.c` ?
— Que faut-il changer si on rajoute une nouvelle fonction `goodbye()` dans le fichier `hello.c` ?
— Pourquoi y-a-t'il des `hello.h` après les deux points sur les règles de `hello.o` et `main.o` ?
— À quoi sert la cible `clean` ?
— Qu'est ce qu'il se passe si je renomme mon fichier `Makefile` en `Makefile.hello` ?
— Quel est le format du fichier `hello.o`, du fichier `main`, du fichier `main.c` ?

5.1 On met tout ensemble : calcul du PGCD

L'algorithme d'Euclide pour calculer le plus grand commun diviseur de deux entiers naturels a et b (au moins un des deux nombres n'est pas nul) utilise la propriété suivante (*mod* est l'opération modulo) :

$$\text{si } a > b \text{ alors } \begin{cases} \text{si } a \bmod b = 0, PGCD(a,b) = b \\ \text{sinon } PGCD(a,b) = PGCD(b, a \bmod b). \end{cases}$$

L'algorithme que vous allez implémenter est donc le suivant :

```
// PGCD(a,b) renvoi le PGCD de deux entiers
ENTIER PRODECURE PGCD(ENTIER a, ENTIER b)
    VARIABLE ENTIER res
DEBUT
    SI (a>b) ALORS
        SI (a mod b == 0) ALORS
            res ← b
        SINON
            res ← PGCD(b, a mod b)
        FINSI
    SINON
        SI (b mod a == 0) ALORS
            res ← a
        SINON
            res ← PGCD(a, b mod a)
        FINSI
    FINSI
    RETOUR(res)
FIN
```

```
//squelette de la fonction
//pgcd(int a, int b)
#include <stdio.h>
#include "pgcd.h"

int pgcd(int a, int b)
{
  int res;
  //ici mettre le calcul
  //du pgcd
   [...]
   return res;
}
```

QUESTION 14 ▶ Créez un repertoire, `pgcd` et implémenter une fonction `pgcd` en C, dans un fichier `pgcd.c` en utilisant l'algorithme proposé. On pourra utiliser l'opérateur modulo (`%`). Vous ne savez pas encore comment faire une fonction en C, mais ce n'est pas très compliqué, on verra cela en détail au prochain TD. En attendant utilisez le squelette proposé ci-dessus à droite. On mettra en place une fonction `main()` dans un fichier `main.c`. La fonction `main()` appelera cette fonction `pgcd()` et on écrira le Makefile avec compilation séparée permettant de compiler l'ensemble.

6 bibliothèque statique (*static library*) en C

Nous allons créer une bibliothèque statique simple, cette bibliothèque sera constituée d'une unique fonction `pgcd(int a, int b)` écrite à la question précédente.

Génération d'un fichier de bibliothèque statique

Un fichier de bibliothèque statique en C est généralement simplement la concaténation d'un certain nombre de fichiers objets que l'on regroupe dans une fichier `.a` grâce à la commande `ar`. Par exemple pour créer une bibliothèque `libpgcd.a` à partir du fichier `pgcd.o`, la commande est simplement la suivante (on ne s'attarde pas sur les options crs, vous pouvez faire `man` pour en apprendre plus) :

```
ar -crs libpgcd.a pgcd.o
```

Les librairie en C sont par convention toutes préfixées par `lib`. Lorsque l'on utilise des librairies dans un programme C, il faut indiquer à l'éditeur de lien *où elles se trouvent*. cela va se faire grâce au flag `-L`. Donc pour utiliser la librairie `libpgcd.a` qui se trouve dans le répertoire `../libpgcd/`, lors de l'édition de lien du fichier `main.o` par exemple, on exécutera la commande :

```
gcc -Wall -L../libpgcd/ main.o -o main -lpgcd
```

Notez bien le `-lpgcd` *en fin de ligne*, si vous mettez ce flag avant, par exemple :

```
gcc -Wall -lpgcd -L../pgcd/ main.o -o main
```

`gcc` ne trouvera pas la fonction `pgcd()`, c'est le seul flag de `gcc` pour lequel la position est importante.

QUESTION 15 ▶ Ajoutez la commande `ar` a votre `Makefile` dans le repertoire `pgcd` comme indiqué ci-dessus pour créer la librairie statique `libpgcd.a` .

QUESTION 16 ▶ Créez un autre répertoire `testpgcd` à coté du repertoire `pgcd`. Dans ce répertoire vous aurez simplement le fichier `main.c` qui utilise la fonction `pgcd()` ainsi que le `Makefile` pour compiler tout ça. Attention, vous avez deux choses à prévoir pour utiliser la librairie `libpgcd` :

1. Mettre `#include "pgcd.h"` au début de votre programme C pour qu'il connaisse le prototype de la fonction `pgcd()`. Pour cela il faudra indiquer, dans le Makefile, avec le flag `-I` où se trouve le fichier `pgcd.h` en question.
2. Mettre `-lpgcd` dans la commande d'édition de lien.

Fichier d'en-tête

En général, dès que l'on fait un projet conséquent en C, les fichiers d'en tête (`.h`) ne sont pas forcément dans le répertoire courant. Soit on a développé certaines fonctions dans d'autres répertoires, soit on a fait le choix de séparer les répertoires ou se trouvent les fichiers sources (`.c`) et les fichiers d'en-tête (`.h`). Le repertoire ou se trouve les fichiers d'en-tête s'appelle souvent `include` (d'ailleurs on dit souvent *les includes* plutôt que *les fichiers d'en-tête*, ça va plus vite...).
Pour indiquer au compilateur (lors de la phase de compilation) où est ce qu'il doit trouver les fichier d'en-tête on utilise le flag `-I`. Par exemple :

```
gcc -Wall hello.c -I../include -c -o hello.o
```

7 Bibliothèques dynamiques en C

Les bibliothèques en C

En général en C, le fichier pour une bibliothèque dynamique `XXX` s'appelle `libXXX.so`. On a vu qu'elle s'appelait `libXXX.a` pour la version statique. Les bibliothèques dynamiques sont beaucoup plus utilisées que les bibliothèques statiques : les tailles des binaires générés sont beaucoup plus petites puisqu'ils n'incluent pas le code de la bibliothèque. Comme pour une bibliothèque statique, on utilise pour cela la flag `lXXX` **en fin de ligne** de la commande d'édition de lien pour indiquer au compilateur que notre programme a besoin d'utiliser la librairie `libXXX.so`.
Pour les bibliothèques standard (`https://fr.wikipedia.org/wiki/Bibliothèque_standard_du_C`) Le compilateur se débrouille tout seul pour trouver la librairie `libXXX.so`. Par exemple pour la libraire *mathématique* (`libm.so`) qui contient un certain nombre de fonction utile comme par exemple la fonction `sqrt`. Pour ces bibliothèques il n'y a pas besoin de flag particulier à l'édition de lien. mais on a toujours besoin des **fichiers d'en-tête** pour les fonctions qu'on utilise, c'est la raison du `#include <stdio.h>` que vous retrouvez dans tous vos fichier. Pour la librairie mathématique, on utilise `#include <math.h>`.
Vous pouvez vous même définir une bibliothèque, statique ou dynamique. Si on utilise une bibliothèque statique, le code de la bibliothèque est rajouté dans votre binaire. Si on utilise une bibliothèque dynamique, le code n'est pas inclus dans le binaire, mais il est chargé en mémoire quand on charge le binaire en mémoire, ce qui permet de ne pas du duppliquer le code des bibliothèques utilisées dans plusieurs exécutables.

QUESTION 17 ▶ Au fait savez vous comment trouver le fichier `libm.so` sur votre machine ?

QUESTION 18 ▶ Ecrivez un programme simple (type hello.c) mais affichez la racine carrée de 5 calculée avec la fonction `sqrt`. Utilisez la commande `man` pour trouver le type de la fonction `sqrt`. Vérifiez que vous trouvez aussi l'information qui vous est donnée ci-dessus, à savoir que le fichier d'en-tête dans lequel est déclarée la fonctionsqrtest le fichier `math.h` (Note : sur certaines distribution récentes de Linux, il n'est plus necéssaire d'inclure `math.h` ni de mettre le flag `-lm` pour utiliser `sqrt`)

8 Pour aller plus loin (facultatif)

Dans chaque TP vous trouverez une section intitulée *pour aller plus loin* qui vous propose des exercices à programmer en C. La solution des exercices sera disponible sur Moodle. L'intérêt de ces exercices est, bien sûr, de chercher à les réaliser. Regarder simplement la solution ne vous apprendra pas à programmer.

8.1 pgcdPrint

QUESTION 19 ▶ Afin de visualiser l'emboîtement des appels récursifs, copiez la fonction `pgcd` créée ci-dessus en une nouvelle fonction `pgcdPrint` et modifier cette fonction afin d'afficher :

- En début d'exécution de la fonction, un message indiquant le début du calcul et les valeurs des paramètres a et b ;
- En fin d'exécution de la fonction, un message indiquant la fin du calcul et la valeur retournée par la fonction.

Essayez de mettre une identation différente pour chaque appel récursif pour bien visualiser ce que chaque appel exécute. Exemple d'affichage qui pourrait être produit par votre fonction :

```
 Appel de pgcd(779,665)
     Appel de pgcd(665,114)
         Appel de pgcd(114,95)
             Appel de pgcd(95,19)
             retour de pgcd(95,19)=19
         retour de pgcd(114,95)=19
     retour de pgcd(665,114)=19
 retour de pgcd(779,665)=19
Le pgcd de 779 et 665 est... 19
```

8.2 Carré d'un nombre

Utilisez l'identité $n^2 = (n-1)^2 + 2n - 1$ pour écrire un programme récursif qui calcule le carré d'un entier positif ou nul n. Afficher, comme précédemment, l'argument lors de l'appel de la fonction et le résultat avant de sortir de la fonction. On remarque que le résultat est différent à chaque appel, ce n'est pas une récursion terminale.

8.3 Coefficient binomial

Le coefficient binomial des entiers naturel n et k noté $\begin{pmatrix} n \\ k \end{pmatrix}$ (anciennement C_n^k : combinaison de k parmi n) peut être calculé récursivement avec les informations suivantes :

$$\begin{pmatrix} n \\ k \end{pmatrix} \text{ vaut 1 si } k = n \text{ ou } k = 0$$
$$\begin{pmatrix} n \\ k \end{pmatrix} + \begin{pmatrix} n \\ k+1 \end{pmatrix} = \begin{pmatrix} n+1 \\ k+1 \end{pmatrix} \text{ si 0<k<n}$$

Écrire un programme qui, par une double récursion calcule $\begin{pmatrix} n \\ k \end{pmatrix}$, l'exécutable prendra en paramètres les deux entiers n et k lu sur la ligne de commande.