Université de Lille 1 SCIENCES ET TECHNOLOGIES

Licence d'informatique
2e année
Module de Pratique du C

Introdution à la « pratique du langage C »

Gilles GRIMAUD Philippe MARQUET

Révision majeure, janvier 2018

Ce sujet est disponible à `http://cristal.univ-lille.fr/~marquet/ens/pdc/tp1`.

Table des matières

Cartes de référence ***...indispensable...***

Une carte de référence, *refcard* en anglais, est un condensé de notes sur un sujet donné, un logiciel, etc. Des cartes de références pour les outils que nous allons utiliser sont disponibles. Des copies de ces cartes vous sont distribuées.
Durant ces travaux pratiques, les cartes de références suivantes vous seront particulièrement utiles :

- *Carte de référence de GNU Emacs* à
  `https://www.gnu.org/software/emacs/refcards/`
- *Carte de référence Unix* (shell `bash`) à
  `http://www.ai.univ-paris8.fr/~djedi/poo/unix-refcard.pdf`
- *C Reference Card (ANSI)* à
  `http://www.math.brown.edu/~jhs/`

Gardez ces cartes avec vous !

C'est à partir du langage C que sont dérivés de nombreux langages tels le C++, C# développé par Microsoft, Objective-C utilisé dans iOS d'Apple, Java maintenant développé par Oracle, et bien d'autres.

Le langage C pourrait donc être étudié en tant qu'ancêtre commun de nombreux langages.

Cependant, si l'enseignement du langage C reste un passage obligé d'un cursus informatique, c'est avant tout parce qu'il permet de contrôler finement le fonctionnement de l'ordinateur. Chaque instruction élémentaire du langage est directement traduisible en une action sur le matériel : le microprocesseur.

On dit que le langage C est un « assembleur portable ». Si il n'est pas nécessaire de connaître de langage machine pour apprendre le langage C, une bonne compréhension générale du modèle von Neumann et du fonctionnement d'un ordinateur est un prérequis indispensable [1] .

Le langage C repose sur un nombre trés limité de concepts qui seront abordés au cours de cet enseignement de « Pratique du C ».

Nous débutons avec la découverte de bonnes pratiques pour créer et organiser des projets en C.

1 Préalables - des outils

Nous allons utiliser quelques outils de base pour pouvoir écrire des programmes en langage C, les compiler, et les exécuter.

1.1 Un shell

Pour éditer, compiler et tester vos programmes, un shell qui s'exécute dans un terminal texte est très pratique. Nous utilisons le shell `bash`.

Rappelez-vous que :

- pour compléter une ligne, on utilise la touche <→|> (*tabulation*) ;
- pour reexécuter une ligne, on utilise la touche <↑> (*flèche haut*) ;
- pour récupérer le code de retour d'un programme, on utilise la commande`echo $?`.

La recheche de documentation est une des opérations essentielles que vous devez savoir faire avec un shell. Vous avez été aguerri à l'utilisation d'un shell dans le cadre du *Stage Unix* dispensé au semestre précédent.

Pour mémoire, vous pouvez consulter les pages du stage :

`http://www.fil.univ-lille1.fr/~sedoglav/SHELL/`

Consultez aussi la *refcard* `bash` pour prendre en main votre shell. □

1. Pour mémoire, le modèle von Neumann a été étudié en cours d'Architecture élémentaire des ordinateurs, voir `http://www.fil.univ-lille1.fr/~meftali/archi/Cours7.pdf`, accessible depuis le portail `http://portail.fil.univ-lille1.fr/ls3/archi`.

Exercice 1 (Trouver la documentation avec votre shell)
Quelle commande pouvez-vous utiliser pour consulter la documentation du shell ? Comment l'utiliser pour consulter l'aide de `echo` ? □

Exercice 2 (Souffleur)
Comme vous le savez déjà la touche *tabulation* <→|> peut vous aidez à souffler une commande dont vous avez saisi les premiers caractères. Cependant, il est aussi possible de l'utiliser pour souffler des paramètres pour la plupart des commandes standard. Combien de commandes documentées par le manuel commence par `e` ? □

Exercice 3 (Manuel de la librairie standard)
Dans la suite de l'énoncé, nous utiliserons une fonction de la librairie standard du langage C. Il s'agit de la fonction `putchar()`. Les fonctions C de la librairie standard sont documentées dans le manuel du shell. Comment pouvez-vous consulter la documentation ? Et comment réutiliser la ligne de commande saisie lors de l'exercice précédent ? □

1.2 Un éditeur, Emacs

Nous vous conseillons d'utiliser `emacs` comme éditeur de fichier texte. Cet éditeur vous permettra d'écrire vos programmes source en langage C. En pratique n'importe quel éditeur de fichiers textes convient. Cependant `emacs` prend en charge les fichiers sources écrits en langage C et rend leur lecture et leur édition plus agréables.

Il s'agit de vous assurer que vous disposez bien de `emacs`. En fait, il faut que vous disposiez d'une version au moins égale à `Emacs 25`.

Exercice 4 (Recherche dans le manuel en ligne)
Comment trouver dans l'aide en ligne l'option de la commande `emacs` qui permet de connaître le numéro de version de votre logiciel ? Plus particulièrement, comment chercher « version » dans les 417 lignes du manuel, relatif à `emacs` ? □

Exercice 5 (Emacs)
À l'aide de ce que vous avez trouvé dans le manuel, déterminez le numéro de version de votre `emacs` ? □

Consultez la carte de référence `emacs` pour prendre en main cet éditeur.

Exercice 6 (C-et M-)
Que signifient les notations <C-> et <M-> utilisées dans cette carte de référence ? □

1.3 Une forge, GitLab

Tout au long du semestre, vous allez rendre vos travaux via GitLab. GitLab est une *forge* qui permet de gérer des dépôts Git.

Git est un logiciel de gestion de versions. Un dépôt Git permet de stocker un ensemble de fichiers en conservant l'historique de toutes les modifications qui ont été effectuées sur ces fichiers.

Il permet de plus de partager cette gestion entre plusieurs utilisateurs et depuis différentes machines (chez vous et à la fac par exemple).

Vous avez accès à un dépôt Git sur GitLab à l'adresse

```
https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/intro
```

ce dépôt contient les éléments nécessaires à la réalisation du présent TP de PdC.

Vous ne pouvez pas directement travailler sur ce dépôt, vous allez devoir réaliser un *fork*. Un fork vous permet de copier ce dépôt, et d'obtenir votre propre copie sous forme d'un dépôt GitLab sur lequel vous pourrez travailler.

Exercice 7 (Forker un dépôt GitLab)
Authentifiez vous sur GitLab avec votre nom d'utilisateur et mot de passe du FIL, et accédez au dépôt

`https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/intro-g<i>-y<yy>`

`<i>` étant votre numéro de groupe et `<yy>` l'année en cours, par exemple `intro-g7-y42` pour le groupe 7 en 2042.

Réalisez un fork de ce dépôt. □

Il est nécessaire d'inviter les personnes qui vont devoir travailler sur ce dépôt via la fonction *Members* du menu *Settings* (se trouvant dans la barre de navigation de gauche sur la page du dépôt). Quand vous ajoutez une personne dans le dépôt, vous pouvez lui donner un rôle (par exemple *développeur*).

Exercice 8 (Ajouter des personnes sur un dépôt)
Ajoutez votre binôme et votre enseignant dans votre dépôt, en tant que développeurs. □

Votre dépôt sur GitLab est maintenant prêt. Toutefois, ce dépôt est sur le serveur GitLab, c'est donc ce qu'on appelle un *dépôt distant*. Pour commencer à travailler, vous devez créer une copie locale (c'est-à-dire sur votre machine de TP) de ce dépôt distant. Cette action s'appelle *cloner* un dépôt.

Quelle est la différence entre un *fork* et un *clone* ?

— le fork permet de créer une copie d'un dépôt GitLab, sous forme d'un autre dépôt GitLab. Les deux dépôts sont donc des dépôts distants, et l'opération se fait via l'interface web de GitLab ;
— le *clone* est réalisé par une commande *git clone* à taper dans votre terminal, et produit une copie local du dépôt sur laquelle il va être possible de travailler.

Exercice 9 (Cloner votre dépôt)
Clonez votre dépôt GitLab vers le dossier de votre choix.

Attention : clonez bien votre dépôt (celui résultant du fork) et non pas le dépôt GitLab original. Depuis l'interface web de GitLab, vous pouvez lister vos dépôts via l'action de menu *Projets* → *Your Projects*. □

Une fois votre dépôt cloné, vous pouvez travailler dans le répertoire local en modifiant ou ajoutant des fichiers. Ensuite, pour valider ces modifications, vous devez réaliser un *commit*. Le commit permet d'enregistrer des modifications ou ajouts de fichiers dans le dépôt, de manière versionnée. Avant chaque commit, il faut indiquer à Git l'ensemble des fichiers modifiés (ou ajoutés).

Exercice 10 (Modifier le dépôt local)
1. Modifier le fichier `Readme.md` pour ajouter votre nom et numéro de groupe.
2. Ajoutez un fichier texte `exo-shell.txt` dans lequel vous donnez les réponses aux exercices 1 et 2. □

Il s'agit maintenant de valider ces modifications et de les enregistrer dans le dépôt local.

Exercice 11 (Enregistrer les modifications locales)
1. Exécutez la commande

   ```
   bash$ git status
   ```

   qui affiche l'état, modifié, ajouté, ou supprimé de vos fichiers par rapport au dépôt local.
2. Identifiez ensuite les fichiers dont vous désirez enregister les modifications dans votre dépôt local via la commande `git add` (que le fichier ait été modifié ou ajouté).
3. Exécutez à nouveau la commande

```
bash$ git status
```

qui affiche à nouveau l'état de votre dépôt local.

4. Précisez la nature des modifications apportées à l'ensemble de ces fichiers par un `commit` :

```
bash$ git commit -m "solution premiers exercices"
```

5. Affichez une nouvelle fois l'état de votre dépôt local. □

Votre dépôt local a étét modifié grâce à votre commit, mais cette modification n'est pas encore répercutée sur le dépôt distant (sur GitLab). L'envoi des commits d'un dépôt local vers un dépôt distant s'appelle un *push*.

Exercice 12 (Pousser une modification)
Réalisez un push de votre commit sur votre dépôt distant GitLab. □

L'opération réciproque du push s'appelle le *pull*, elle permet de mettre à jour vôtre dépot local, en récupérant les commits depuis le dépôt distant.

Exercice 13 (Garder le dépôt local à jour)
Demandez à votre binôme de travailler sur sa session pour

— réaliser une copie locale de votre dépôt GitLab ;
— ajouter son nom dans le fichier `Readme.md` et ajouter un fichier `exo-emacs.txt` avec la réponse à l'exercice 6 ;
— enregister ces modifications dans son dépôt local par `git add` et `git commit` ;
— pousser ces modifications sur le dépôt du serveur GitLab.

Cela fait, mettez à jour votre dépôt local à l'aide de la commande

```
bash$ git pull
```

et vérifiez que vous voyez bien les modifications réalisées par votre binôme.

1.4 Un compilateur

Nous vous conseillons d'utiliser `gcc` pour *compiler* vos programmes. Un compilateur de langage C est un programme qui lit un fichier texte contenant le *code source* contenant un programme écrit en C, et qui le traduit en un *code exécutable* qu'un microprocesseur peut exécuter. Le code exécutable par le microprocesseur est composé d'une suite d'octets qui doivent être compris comme autant d'instructions et de données à l'attention du microprocesseur. Il ne constitue donc pas un texte et n'est pas lisible par un humain.

Les microprocesseurs qui équipent vos machines ne savent exécuter qu'un seul type de code, que l'on appelle souvent *code exécutable*, *code machine* ou encore, de manière impropre *code binaire*. Le compilateur fait *une fois pour toute* l'opération de traduction. D'autres langages tel que Python font cette traduction à chaque exécution, on parle alors d'*interpréteurs*, et de *langage interprété*. Notez encore qu'il existe des langages hybrides qui tel que Java, qui recourrent à un compilateur pour transformer le source dans une représentation intermédiaire aussi appelée *bytecode*. C'est alors une *machine virtuelle* qui termine, lors de chaque exécution, la traduction.

Exercice 14 (Un compilateur, `gcc`)
La commande `gcc` transforme donc un *code source* en *code exécutable*. En considérant que le code source est dans le fichier `prog1.c` et que vous souhaitez produire un programme executable dans `prog1`, quelles sont les options qu'il faut saisir pour compiler votre code source dans votre code exécutable ? (Vous pouvez consulter le manuel en ligne !) □

Compilateurs C ***...en savoir plus...***

Dans l'UE Pratique du C, nous utilisons `gcc` mais il existe bien d'autres compilateurs de *code source* en langage C vers du *code exécutable*. Au nombre des plus connus, on trouve : `clang`, `icc` (d'Intel) ou encore `cl` (de microsoft).
En général le compilateur produit un code exécutable pour le microprocesseur de l'ordinateur sur lequel le compilateur est exécuté. Cependant il est possible d'utiliser un compilateur pour produire un `code exécutable` destiné à un autre microprocesseur. Par exemple, on peut compiler un code C sur un PC/Linux (processeur intel) avec un compilateur `arm-gcc` qui produit un *code exécutable* pour processeur ARM (qui équipent notamment les smartphone). Dans ce cas le *code exécutable* produit par `arm-gcc` ne pourra pas être exécuté sur la machine qui l'a compilé, mais seulement sur le smartphone visé. On parle de *cross-compilation*.

2 Premiers pas en C

Il est d'usage d'écrire un programme « hello world ». En voici une version un peu différente.

Exercice 15 (Édition de code)
Avec `emacs`, ouvrez le fichier `prog1.c` présent dans votre dépôt git et dont les premières lignes sont

```
/* Ceci est un commentaire */

/**
 * Nous allons utiliser la fonction "putchar()" de la librairie
 * standard du langage C pour sortir le caractère de code ASCII
 * "c" sur le terminal. Cette fonction retourne -1 en cas d'ereur.
 * Voici la déclaration de cette fonction :
 **/
extern int putchar (int c);

/**
 * Nous définissons maintenant la fonction "main()" : nous allons
 * écrire le corps de cette fonction.
 * "main()" est la fonction qui sera exécutée au démarrage de
 * notre programme.
 * Cette fonction ne prend pas de paramètre et retourne 0 si le
 * programme termine correctement (une autre valeur en cas d'erreur).
 **/
int
main()
  {
 putchar(72);
 ...
```

Notez que ce code choisit délibérément une indentation fantaisiste. Cette écriture peut sembler particulièrement dissonante pour des informaticiens habitués à programmer en Python. Cependant contrairement à Python le langage C n'impose pas d'indentation *a priori*. Ceci étant dit, une indentation fantaisiste complique la lecture d'un programme, aussi avec le temps, diverses stratégies d'indentations ont été proposés, les styles : K&R, BSD KNF, GNU... Consultez éventuellement la page `https://en.wikipedia.org/wiki/Indentation_style`.

Les éditeurs de *code source* proposent souvent des options de normaliser l'indentation d'un fichier ou d'une sélection de texte, automatiquement.

Exercice 16 (Indentation)
Trouvez dans la carte de référence `emacs` un moyen de reprendre l'indentation du code C que vous avez saisi. Reformatez ainsi tout le code source du fichier `prog1.c`.

Ensuite, réalisez un *commit* pour enregistrer cette modification dans votre dépôt Git. Ce commit devra être accompagné d'un message bref et pertinent, décrivant la modification réalisée (exemple : "correction de l'indentation").

De manière générale, vous devrez réaliser un *commit* pour chaque exercice en TP de PdC (même si l'énoncé de l'exercice ne le demande pas explicitement). Pensez également à faire un *push* à la fin de la séance.

□

Exercice 17 (Mise en commentaires)
Sachant que sous `emacs` la commande <M- ;> est relative à la mise en commentaire, commentez la fonction `main()`, puis décommentez la. □

La fonction `putchar()` prend en paramètre une valeur appelée `c` qui est déclarée de type `int`. Ce type `int` en C ne signifie pas exactement « entier ». En fonction des caractéristiques du microprocesseur, une valeur N est choisie par le compilateur C [2], et `int` signifie « mot mémoire de la machine pour manipuler des entiers dans l'intervalle $[-2^N..2^{N-1}]$ ».

Dans cet exemple de programme, observez que la fonction `putchar()` assume que la valeur de `c` sera un « code ASCII », donc une valeur comprise entre 0 et 255 qui est associée à un caractère alphanumérique. C'est bien ce caractère, et non la valeur entière utilisée pour le coder, qui sera affichée sur le terminal [3] .

Dans la fonction `main()` figurent différentes manières d'exprimer une valeur littérale entière en C :

1. `72` correspond à la valeur décimale 72;
2. `0x69` représente la valeur hexadécimale 69, ce qui est équivalent à la valeur décimale 105 ($6 \times 16 + 9 = 105$);
3. `'!'` représente la valeur du code ASCII du caractère « ! », soit la valeur décimale 33. Consultez éventuellement la page de manuel `man ascii` pour vous en convaincre.
4. `'\n'` représente la valeur du code ASCII du caractère associé à *line feed*, retour à la ligne, ce qui est équivalent à la valeur décimale 10.

Exercice 18 (Valeurs littérales et code ASCII)
Quelle succession de quatre caractères l'exécution du programme principal va-t-elle afficher ? □

Exercice 19 (Compilation)
Listez les fichiers présents dans votre répertoire courant avec la commande `ls`. Assurez vous que votre fichier source est bien présent.

Pour compiler ce programme `prog1.c`, il suffit de lancer la commande suivante :

```
bash$ gcc prog1.c
bash$
```

Dans votre répertoire courant, lister à nouveau les fichiers présents. Quel fichier l'exécution de `gcc` a créé ? □

Exercice 20 (Exécution)
Lancer l'exécution du fichier créé. Ajouter éventuellement `./` devant son nom, pour que `bash` le trouve.

Comparez l'affichage à la réponse que vous aviez fournis à l'execice 18. Que pouvez-vous en dire ? □

2. Selon les standards C, N doit être au moins égal à 16. En pratique, pour des raisons de compatibilité ascendantes, `gcc`, tel qu'il est utilisé dans Linux impose $N = 32$ même lorsque le microprocesseur permettrait $N = 64$.

3. Pour mémoire, la représentation des nombres dans les différentes bases et le codage ASCII ont été discutés dans le cours de codage. Voir `http://www.fil.univ-lille1.fr/~salson/codage/Poly/poly.pdf` accessible depuis le portail `http://portail.fil.univ-lille1.fr/ls3/codage`.

Nous allons maintenant implémenter une nouvelle fonction : `newline()`.

Cette fonction imprime simplement un *line feed* pour forcer l'affichage à passer à la ligne. Comme `putchar()`, elle retourne -1 si elle n'a pas pu produire l'octet en sortie, ou une autre valeur sinon.

Exercice 21 (Une fonction)
Pour définir cette fonction saisissez le code suivant *à la fin* de votre fichier :

```
int
newline()
{
    return putchar('\n');
}
```

Puis appelez la fonction `newline()` à la place de `putchar('\n')` dans le programme principal.

Compilez votre programme et exécutez le.

Qu'observez-vous ?

Rappel : n'oubliez pas de réaliser le commit correspondant à cet exercice. Ceci ne sera plus rappelé sur les exercices suivants. □

La compilation d'un source C est une opération « en flux ». C'est-à-dire que le compilateur traite une ligne de source avant de passer à la suivante. Lorsque un source utilise une fonction avant que le corps de cette fonction n'ait été traité par le compilateur, ce dernier déclare qu'elle n'existe pas. Dans le doute `gcc` « suppose » l'existence d'une fonction implémentée plus tard, mais il génère une alerte (*warning*) : il signale que, même s'il a fait au mieux, il y a un problème dans le *code source*.

Pour éviter ce *warning*, vous pouvez bien sur déplacer le code de la fonction `newline()` avant l'implémentation de la fonction `main()`.

Cependant il peut parfois être nécessaire ou simplement souhaitable de ne pas imposer au développeur un ordre particulier dans l'implémentation des fonctions d'un programme.

Aussi il est possible de *déclarer* dans un premier temps l'existence d'une fonction, sans en donner le code. Si une telle déclaration est donnée avant que le compilateur ne traite la fonction `main()`, lorsqu'il compilera cette dernière il ne génèrera pas de *warning*.

Exercice 22 (Déclaration de fonction)
En tête de votre fichier source, donnez la déclaration de la fonction `newline()` mais pas son implémentation. De la même manière que nous avons donné la déclaration de la fonction `putchar()`. En C on appelle prototypes ces déclarations de fonctions qui ne donnent pas le corps.

Recompilez votre programme et assurez vous que le *warning* a disparu. □

3 Différents sens pour les mêmes nombres

Comme vous l'avez vu en cours de codage, les systèmes informatiques ne traitent que des nombres, mais ces nombres (binaires) peuvent prendre des sens très différents selon le rôle que l'on veut leur faire jouer. La fonction de la librairie standard `putchar()` prend un `int` pour produire un caractère sur la sortie standard. Cependant d'autres affichage de cette même valeur « entière » peuvent être utiles. On peut par exemlpe vouloir d'afficher la suite de caractères entre '0' et '9' qui représentent le nombre en décimal, ou en hexadécimal...

Dans la série d'exercices qui suit, il vous est demandé d'implémenter une série de fonctions. Implémentez ces fonctions dans un fichier `numbers.c`. Dans ce même fichier, fournissez une fonction `int main()` dans laquelle vous ferez des appels aux fonctions implémentées en vue d'en tester le bon fonctionnement.

Les entiers en C ***...on fait le point...***

Notez que comme `int` n'est pas un entier quelconque mais un *mot mémoire* `d` ne peut pas être ≪ arbitrairement grand ≫. Sur votre machine, N est défini à 32 donc `int` peut représenter 4.294.967.296 codes distincts. Les microprocesseurs utilisent le codage des nombres signés en ≪ complément à deux ≫ (cf. Cours de Codage, chapitre 1). Ce codage permet de représenter des nombres entre -2.147.483.648 et 2.147.483.647.
Cette asymétrie entre le plus petit et le plus grand `int` a notamment comme conséquence qu'il n'est pas possible de calculer la valeur absolue du plus petit `int`.

3.1 Afficher des caractères

Exercice 23 (Afficher un chiffre (décimal))
Proposez une implémentation de la fonction de prototype

```
int put_digit(int d);
```

qui affiche le chiffre (*digit*) `d`. Ainsi :
- si `d` vaut 0, c'est le caractère `'0'` (de code ASCII 48) qui est produit sur la sortie standard ;
- si `d` vaut 1, c'est le caractère `'1'` (de code ASCII 49)...
- si `d` est inférieur à 0 ou supérieur à 9, la fonction retourne une erreur (-1, de même que si le `putchar` échoué...).

□

Exercice 24 (Afficher un chiffre hexadécimal)
De même proposez une fonction `int put_hdigit(int h);` qui affiche un nombre `h` sous la forme d'un caractère hexadécimal. `h` est donc une valeur entre 0 et 15 et le code ASCII produit est donc :
- entre `'0'` et `'9'` si $0 \leq d \leq 9$; ou
- entre `'A'` et `'F'` si $10 \leq d \leq 15$.

□

Vous pouvez maintenant afficher des caractères ASCII et des chiffres hexadécimaux.

Il est parfois utile de pouvoir consulter la table des codes ASCII. Un simple programme peut afficher cette table :

```
--  -0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -A -B -C -D -E -F
0-  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
1-  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
2-      !  "  #  $  %  &  '  (  )  *  +  ,  -  .  /
3-   0  1  2  3  4  5  6  7  8  9  :  ;  <  =  >  ?
4-   @  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O
...
E-  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
F-  .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
```

Quand le caractère ASCII n'est pas affichable, les caractères `..` sont produits (les caractères ASCII affichables ont une valeur comprise entre 32 et 126, consultez par exemple la page de manuel `man ascii`).

Exercice 25 (Table des codes ASCII)
Proposez une fonction `void print_ascii_table()` qui affiche la table des codes ASCII, conformément à l'exemple donné ci-dessus. Cette fonction fera exclusivement appel à `putchar()` et à nos fonctions `newline()` et `put_hdigit()`.

Vous ajouterez cette fonction au fichier `numbers.c` et y ferez appel dans le `main()` afin de la tester. □

3.2 Afficher des nombres

Les fonctions `putchar()`, `put_digit()` et `put_hdigit()` affichent un caractère correspond à la valeur `int` reçue en paramètre. La fonction `putchar()` permet d'afficher un caractère en donnant un `int` correspondant à son code ASCII (ou UTF-8), la fonction `put_digit()` affiche le chiffre (dont un caractère) correspondant à la valeur de type `int` donnée, etc.

Nous allons maintenant définir des fonctions qui considèrent le *mot mémoire* correspondant à un `int` (cf l'encart *Les entiers en C*) comme une valeur entière qu'il conviendra d'écrire en décimal, hexadécimal, binaire, etc. Nous allons donc afficher des séries de caractères qui représentent les chiffres qui composent le nombre, éventuellement précédés d'un premier caractère pour indiquer le signe du nombre.

Exercice 26 (Afficher un nombre - décimal)
Donnez la définition de la fonction

```
int putdec(int d);
```

qui affiche les chiffres de l'écriture décimale de la valeur `d`, au besoin, précédé d'un signe négatif, et sans afficher de '0' inutiles à gauche du nombre.

Cette fonction retourne normalement 0, -1 en cas d'erreur. □

Nous avons vu que les opérations arithmétiques du langage C sur des valeurs du type `int` sont traduites en opérations élémentaires du microprocesseur. De plus, le microprocesseur travaille en « complément à deux ». Enfin, le type `int` correspond à un mot mémoire de 32 bits. Ces éléments sont a prendre en compte pour comprendre comment est calculé l'opposé de -2.147.483.648.

Exercice 27 (L'opposé de -2.147.483.648)
Modifiez la fonction `main()` de `numbers.c` et saisissez le code suivant :

```
int main()
{
    int i=-2147483648;
    putdec(-i);
    return 0;
}
```

Qu'affiche maintennant l'exécution de ce programme ? Et surtout... pourquoi ? □

Exercice 28 (Afficher un nombre binaire)
Proposez une fonction `putbin(int b)` qui affiche un nombre `b` en paramètre sous la forme d'une série de 0 et de 1.

Notez qu'il est possible de déterminer la valeur du bit de poids faible (ou de plusieurs bit) d'un nombre en utilisant l'opérateur `&`. Utilisez les opérateurs >> pour décaler les bits d'un nombre vers la droite...

Comme pour la fonction `putdec()`, il est inutile, d'afficher les 0 à gauche du nombre. Et comme pour `putdec()`, la fonction retourne 0 si le nombre a été correctement affiché sur la sortie standard, -1 sinon. □

Exercice 29 (Affichage hexadécimal)
Proposez une fonction `int puthex(int h)` qui prend un nombre `h` en paramètre et qui produit sur la sortie standard une série de code ASCII représentant ce nombre `h` sous forme hexadécimale.

Pour vous aider, observez qu'un mot mémoire de 32 bits est composé de 8 quartets de 4 bits. Chaque quartet est un chiffre hexadécimal du nombre. Utilisez ici encore, avantageusement les opérateurs >> et `&`. □

Réalisez maintenant une nouvelle version de la fonction `main()` de `numbers.c` :

```
int put_test_line(int n)
{
    putchar('t');
    putchar('e');
    putchar('s');
    putchar('t');
    putchar(' ');
    putchar('#');
    putdec(n);
    putchar(':');

    return 0;
}

int main()
{
    int i=-2147483648;
    put_test_line(1); putdec(214); newline();
    put_test_line(2); putdec(-74); newline();
    put_test_line(3); putdec(1); newline();
    put_test_line(4); putdec(-1); newline();
    put_test_line(5); putdec(0); newline();
    put_test_line(6); putdec(2147483647); newline();
    put_test_line(7); putdec(-2147483648); newline();
    put_test_line(8); putdec(-(-2147483648)); newline();
    put_test_line(9); puthex(0); newline();
    put_test_line(10); puthex(10); newline();
    put_test_line(11); puthex(16); newline();
    put_test_line(12); puthex(2147483647); newline();
    put_test_line(13); puthex(-2147483648); newline();
    put_test_line(14); puthex(0xCAFEBABE); newline();
    put_test_line(15); puthex(-1); newline();
    put_test_line(16); putbin(0); newline();
    put_test_line(17); putbin(1); newline();
    put_test_line(18); putbin(-1); newline();
    put_test_line(19); putbin(2147483647); newline();
    put_test_line(20); putbin(-2147483648); newline();

    return 0;
}
```

Vous trouverez ce code dans le fichier `numbers-test.c` du dépôt Git.

Exercice 30 (Tester vos programmes)
Pour valider le bon fonctionnement de ce programme, nous vous donnons un fichier de trace correct que nos avons produit. Voyez `numbers-out.txt` du dépôt Git.

Vous aller utiliser la commande `diff` pour comparer la trace attendue par le programme précédent.

Si votre programme n'a pas le comportement attendu, c'est-à-dire si ce qu'il affiche diffère de la trace proposée... corrigez le ! □

4 Le préprocesseur

Par commodité le langage C exploite un préprocesseur. Un préprocesseur de texte est un logiciel qui parcourt un fichier source pour le transformer en un autre source, avant de donner ce nouveau source au véritable compilateur. A priori, le préprocesseur utilisé par le langage C est destiné au sources écrites en C, mais aussi en C++ et en Objective C. Cependant il peut être utilisé sur d'autres langages pourvu que ces derniers utilisent des conventions de commentaires et de chaînes de caractères équivalentes à celles du langage C. Il pourrait même être utilisé sur des sources Java, même si ce n'est pas l'usage.

4.1 Substitution de texte, les macros du préprocesseur

Le préprocesseur permet donc de remplacer des symboles d'un *fichier source* par « autre chose », c'est-à-dire par une chaîne de caractères quelconque que le compilateur traitera en lieu et place du symbole.

Considérez le code source suivant disponible dans le fichier `macro1.c` (on travaille pour cetet section sur le processeur dans le sous-répertoire `prepro/` du dépôt Git) :

```
extern int putchar(int c);

int main()
{
    int u=68;
    putchar(u);                 /* premier char */
    putchar(T);                 /* deuxième char */
}
```

Exercice 31 (Ceci n'est pas défini)
Pourquoi ce programme ne peut-il pas fonctionner ? □

Il est possible d'indiquer au compilateur que le symbole `T` doit être transformé en autre chose.

Ces symboles qui sont simplement, syntaxiquement, remplacés par autre chose sont appelés des *macros*.

Il existe deux manières de définir une macro.

Exercice 32 (Directive de préprocesseur en ligne de commande)
La première solution consiste à indiquer en ligne de commande que l'on souhaite que le préprocesseur remplace un symbole par autre chose :

```
bash$ gcc macro1.c -D T=65@
```

Sur cet exemple, on indique au préprocesseur du compilateur C que les symboles `T` doivent être remplacés par des 65.

Expliquez ce qu'affiche le programme ainsi compilé ? □

La seconde solution consiste à utiliser la directive `#define`. dans le fichier source. À l'instar de la solution précédente, cette directive permet d'indiquer au préprocesseur qu'il doit remplacer un symbole par autre chose. Ajoutez la ligne suivante en tête du fichier source :

```
#define T (65+1)
```

Notez qu'il ne s'agit pas d'une instruction du langage C. Elle ne se termine pas par un `;`. Elle ne correspond pas à une affectation, il n'y a pas de =... Ici on dit simplement au préprocesseur du compilateur, qu'à partir de maintenant, à chaque fois qu'il rencontre `T`, il doit le remplacer par `(65+1)`.

Exercice 33 (Directive `#define` de préprocesseur)
Expliquez ce qu'affiche le programme ainsi compilé ? □

Il est possible de demander un compilateur de n'exécuter que le traitement préprocesseur et de produire le fichier résultant de ce traitement sur la sortie standard (ou dans un fichier). Il s'agit de l'option `-E`.

Exercice 34 (Sortie du préprocesseur)
Observez la sortie du préprocesseur. Les lignes qui commencent par un `#` sont des informations que le préprocesseur transmet au compilateur. Pour cette question, ne les prenez pas en compte. Vous pouvez retirez ces lignes en filtrant la sortie de `gcc` avec

```
... | grep -v "#"
```

qui ne retiendra que les lignes qui ne contiennent pas un `#`. Comment le préprocesseur a modifié le source ? □

Dans les exemples précédents la substitution opère sur ce qui semble être une simple variable. Il n'en est rien. Il s'agit en fait d'une substitution syntaxique, qui n'a aucune valeur sémantique : `t` n'est pas une variable, elle n'a d'ailleyrs pas étét déclarée, mais un symbole qui a été remplacé par `(65+1)`.

Il est donc possible de remplacer n'importe quel symbole par autre chose, pour vu que ce soit par quelque chose que le compilateur C définisse, comme une fonction, une instruction élémentaire, ou une séquence d'instructions.

Considérez le programme `elif.c` suivant :

```
extern int putchar(int c);
extern int getchar(void);

#define DIGIT 'D'
#define LOWER 'L'
#define UPPER 'U'
#define OTHER 'O'

int
main()
{
    int c;
    for (;;) {
        c = getchar();
        if ('0'<= c && c<='9')
            putchar(DIGIT);
        else if ('a'<=c && c<='z')
            putchar(LOWER);
        else if ('A'<=c && c<='Z')
            putchar(UPPER);
        else
            putchar(OTHER);
    }
    return 0;
}
```

Exercice 35 (elif)
Le langage Python, que vous avez étudié depuis la première année de Licence, dispose d'un mot clef du langage : `elif` qui pourrait être utilisé dans l'exemple précédent. Définissez une macro, c'est-à-dire une directive `#define`, de telle sorte qu'il soit possible de remplacer les `else if` du programme précédent pour un `elif` tel que le langage Python le propose. □

Il existe un certain nombre de macros qui sont prédéfinies. elles qui permettent d'instrumenter le code source. Les plus utiles sont les macros implicites suivantes :

__LINE__ donne le numéro de la ligne courante au moment de la compilation ;

__FILE__ donne le nom du fichier courant au moment de la compilation ;

__DATE__ donne la date courante, au moment de la compilation ;

__TIME__ donne l'heure minute seconde courante, au moment de la compilation.

Exercice 36 (Macros prédéfinies)
En réutilisant votre fonction `putdec()`, afficher la ligne courante, au début de la fonction `main()`. Déclarez aussi une variable globale `ln` qui sera nitialisée par

```
int ln=__LINE__;
```

et affichez la valeur de `ln` dans la fonction `main()` avant, et après le `putdec()`. Qu'observez vous ? □

Le système de substitution de symbole permet de définir des paramètres de substitution. On parle alors de macros paramétrées. L'exemple suivant illustre cette possibilité (fichier `macrop.c`) :

```
extern int putchar(int c);

#define bit(i,j) (i>>j)&1

int main() {
    int i=16;
    int i0=bit(i,0);
    int i4=bit(i,4);
    putchar('0'+i0);
    putchar('0'+i4);
    putchar('\n');
    putchar('0'+bit(i,0));
    putchar('0'+bit(i,4));
    putchar('\n');
}
```

Dans ce source `macrop.c`, la macro `bit` est paramétrée par deux informations, `i` et `j`. Dans une lecture maladroite, on pourrait percevoir cette macro comme une fonction qui prend 2 paramètres (`i` et `j`) et qui renvoie la valeur du `j`-ième bit de `i` (0 ou 1).

Cependant ce n'est pas le sens de cette ligne de C. Cette ligne doit être comprise comme une indication au préprocesseur pour qu'il remplace, partout dans le code source, les occurrences de `bit(`*x*`,`*y*`)` par des `(`*x*`>>`*y*`)&1`.

Exercice 37 (Macros paramétriques)
Compilez et exécuter ce programme `macrop.c`. Expliquez l'affichage obtenu.

Pour vous aider à mieux comprendre ce qui ce passe exactement, vous pouvez consulter le code produit par le préprocesseur, option `-E` du compilateur. □

4.2 Suppression de lignes de code, la compilation conditionnelle

Le préprocesseur permet aussi de supprimer du fichier source des lignes de codes, selon des conditions qui seront appréciées au moment de la compilation. Il est par exemple possible que le même fichier source soit compiler avec, ou sans, certaines fonctions, selon que des macros soient définies ou pas dans la ligne de commande lors de la compilation.

Le préprocesseur permet de conditionner le fait que certaines portions de codes seront (ou pas) données au compilateur. Le développeur peut par exemple conditionner la compilation d'une portion du code source, en fonction de la valeur d'une macro particulière.

Considérez le programme suivant

```
extern int putchar(int c);

#if NO_LOG==1

int logchar(int c) {
    return 0;
}

#else

int logchar(int c) {
    return putchar(c);
}

#endif

int main() {
    int i=1;
    i=3*i;
    logchar('0'+i);
    return i;
}
```

disponible dans le fichier `compcond.c` (compilation conditionnelle).

Exercice 38 (Un *fichier source* pour deux programmes)
Donnez la ligne de compilation pour que la macro `NO_LOG` produise un code qui va afficher le resultat avant de le retourner. Utilisez l'option `-E` pour vous assurer que vous avec compiler le code voulu. □

Notez que l'expression évaluée par le préprocesseur ne peut porter que sur des constantes et sur des macros. Le préprocesseur ne peut pas apprécier la valeur d'une variable C par exemple.

Simplement le préprocesseur ne « comprends rien » au langage C.

En général nous n'avons pas besoin de tester une valeur particulière mais simple de tester un « oui ou non ». Pour cela l'usage consiste à tester l'existence ou non d'une macro et non sa valeur. Les primitives `#ifdef MACRO` sont alors utilisées.

`#ifdef MACRO` est validée par le préprocesseur si la macro `MACRO` est définie, quelque soit sa valeur (et en particulier, même si elle n'a aucune valeur, ce que l'on obtiendrait avec un `#define MACRO` sans rien de plus...).

Exercice 39 (Tester l'existence plutôt que la valeur)
Remplacez la compilation conditionnelle du programme précédent de tel sorte que l'affichage n'ai pas lieu lorsque la macro `NO_LOG` est définie. Vérifiez que votre implémentation fonctionne en faisant :

```
bash$ gcc compcond.c -D NO_LOG
```

Notez qu'il n'est plus utile de donner une valeur à la macro `NO_LOG` puisque seule son existance est testée.

Comparez le résultat avec celui de la compilation obtenue par :

```
bash$ gcc compcond.c
```

□

Considérez maintenant le programme suivant (`compchk.c` :

```
#ifndef SIZE
#error "définissez SIZE avec l'option -D SIZE=n"
#define SIZE 0
#endif
#if SIZE & (SIZE-1)
#warning "SIZE devrait être une puissance de 2."
#endif

int main(void)
{
    return SIZE;
}
```

Exercice 40 (Erreur de compilation programmée)
Compilez ce programme. Que ce passe-t'il si vous ne définissez pas `SIZE` sur la ligne de compilation ? □

Exercice 41 (Avertissement de compilation programmé)
Compilez maintenant le programme en définissant une valeur pour la macro `SIZE`.

Choisissez une valeur qui n'est pas une puissance de 2. Que ce passe-t-il ? Pourquoi ? □

Avant de pourusivre avec l'étude des fonctionnalités du préprocesseur, assurez-vous de bien comprendre les quatre premières recommandations de l'encart *Bon usage du préprocesseur*.

Bon usage du préprocesseur ***...indispensable...***

Le préprocesseur permet d'écrire de transformer le code source, et, dans une certaine mesure, de faire évoluer la syntaxe du langage C, de façon assez radicale. Si cet usage a connu son heure de gloire dans les années 1990, il est aujourd'hui largement déprécié. En effet, l'usage intensif du préprocesseur, pour créer de nouveaux ≪ pseudo mots clefs ≫, pose deux problèmes principaux :

— le code source ainsi traité devient difficile à lire par un tier qui ne connaît pas les macros mises en place ;

— le code source devient difficile à débugger. Considérez

```
#define MAX 3;
int i;
if (i < MAX) {
...
```

Pour mettre en évidence les segments de codes qui seront transformés par le préprocesseur, dans un source C, un certain nombre d'usages se sont progressivement imposés :

1. les symboles du préprocesseur sont écrits en majuscule : `#define MA_MACRO 17`
2. les expressions évaluées sont sur-parenthèsées : `#define MA_MACRO (3+i*2)` plutôt que `3+i*2`...
3. les macros paramétrées sont dépréciées au profit de fonctions `inline` du type `inline int max(int i,int j) { return (i<j)?i:j; }`
4. les notices précisent toujours les fonctions susceptibles d'être implémentées avec des macros : voir par exemple `$ man putc`
5. les `#include` portent toujours sur des fichiers de prototypes `.h` et pas sur des fichiers `.c` ;
6. les fichiers de prototypes ne devraient jamais contenir d'implémentation mais seulement des déclarations (à l'exception des fonctions `inline`) ;
7. il est conseillé d'utiliser des *include guard* : consulter `https://fr.wikipedia.org/wiki/Include_guard`.

4.3 Inclure des fichiers sources, les uns dans les autres

Le préprocesseur du langage C permet d'inclure un fichier dans un autre. C'est le sens de la directive `#include "un_fichier"`.

Lorsque le préprocesseur rencontre cette commande il effectue l'équivalent d'un ≪ copié-collé ≫ de *un_fichier* en lieu est place du `#include "un_fichier"`. Il est ainsi possible d'inclure un autre fichier source dans un fichier source C.

Exercice 42 (L'inclusion de fichiers)
Éditez un fichier `include_file.c` comme suit :

```
int f(int x)
{
    return 2*x;
}
```

Puis un fichier `main_file.c`[4]

```
#include "include_file.c"

int main()
{
    return f(3);
}
```

4. Comme précisé dans l'encart *Bon usage du préprocesseur,* les directives `#include` doivent toujours porter sur des fichiers `.h`. L'usage fait ici de fichier `.c` est uniquement à visée pédagogique.

Exercice 43 (Un premier `#include`)
Observez le résultat du traitement du fichier `main_file.c` par le préprocesseur en utilisant les mêmes options que précédemment. Qu'a fait le préprocesseur ? À quoi correspondent les lignes qui commencent par un `#` ? □

En utilisant les `#include` il devient possible de mettre des fonctions dans les fichiers que l'on réutilise d'un programme à l'autre. (Cependant le langage C nous invite à ne pas faire usage du préprocesseur en ce sens, mais à plutôt privilégier la création de *fichiers objets* que nous expérimenterons dans un exercice ultérieur.)

L'inclusion de fichiers est trés utilisées pour éviter d'avoir à redéclarer les fonctions externes que nous avons l'intention d'utiliser.

Précédemment, nous avons de nombreuses fois déclaré la fonction `int putchar(int c);` qui est implémentée dans une librairie de fonctions standard du langage C. Cette fonction, comme toutes celles de la librarie standard sont très fréquemment utilisées. Aussi il peut être fastidieux, de déclarer chacune d'elles au début de chaque nouveau programme. C'est pourquoi un fichier de déclaration de ces fonctions a été standardisé. Notez bien que ce fichier ne contient pas l'implémentation mais seulement la *déclaration* des fonctions (c'est-à-dire la ligne *e.g.* `int putchar(c) ;` sans le corps de la fonction).

Exercice 44 (Les fichiers `.h`)
Trouver le nom du fichier dans lequel la fonction `putchar()` est déclarée en interrogeant le manuel. □

Notez que dans le manuel le fichier a pour extension `.h` et pas `.c`. Un fichier `.h` est un source en langage C au même titre qu'un fichier `.c`. Cependant l'extension `.h` a pour vocation d'indiquer que le fichier contient des déclarations, mais pas d'implementation. On parle de *fichiers de prototypes*. Ils sont donc destinés à être inclus dans d'autres fichiers C mais pas à être compilés pour eux-mêmes.

Vous pouvez observer que votre répertoire courant (celui de vos fichiers sources) ne contient pas le fichier `.h` mentionné par le manuel. Vous pouvez aussi remarquer que dans le manuel le nom du fichier à inclure n'est pas donné entre guillemets `" ... "`, mais pas entre chevrons `< ... >`.

Lorsqu'un fichier est donné entre guillemets, c'est que le préprocesseur doit le trouver dans le repertoire courant (ou qu'il doit éventuellement être trouvé dans des repertoires annexes définis *explicitement* au compilateur par l'option `-I dir`).

Lorsqu'un fichier à inclure est donné entre chevrons, c'est qu'il doit être trouvé dans des répertoires annexes définis *implicitement* par le compilateur. Pour connaitre la liste de ces répertoires définit par `gcc` vous pouvez faire :

```
bash$ echo | gcc -E -Wp,-v -
```

Exercice 45 (Les `#include` par défaut)
Modifier le programme précédent pour qu'il affiche le résultat de `f(3)` sur l'écran avant de sortir. Pour cela utilisez la fonction `putchar()`, mais au lieu de la déclarer en début de fichier, demandez au préprocesseur d'inclure le fichier indiqué par le manuel.

Vérifiez que votre nouveau programme fonctionne.

Observez le fichier source produit par le préprocesseur. Combien de ligne contient le fichier produit par le préprocesseur ? Comment est déclarée la fonction `putchar()` ? □

Lorsque les programmes deviennent plus complexes, il peut être tentant de mettre des directives `#include` dans des fichiers `.h`. Dans ce cas, par défaut, le préprocesseur va inclure, récursivement les fichiers référencés. La norme du langage C précise qu'un compilateur doit pouvoir gérer jusqu'à 16 niveaux d'inclusion (d'un fichier, dans un autre, dans un autre...). `gcc` en gère jusqu'à 200.

Considérez le petit exemple suivant composé de trois fichiers :

Un fichier `abs.h` :

```
inline int abs(int x) { return (x<0)?-x:x; }
```

Un fichier `minmax.h` :

```
#include "abs.h"

inline int min(int x,int y) { return (x+y-abs(x-y))/2; }
inline int max(int x,int y) { return (x+y+abs(x-y))/2; }
```

Et enfin un dernier fichier `guard.c` :

```
#include "minmax.h"
#include "abs.h"

int putchar(int c);

int main() {
    putchar('0' + min(3, 4));
    putchar('0' + max(3, 4));
    putchar('0' + abs(-2));
    return 0;
}
```

Dans cet exemple, on imagine que les deux fichiers `.h` ont été conçus par Alice. Bob désire utiliser les fonctions proposées par Alice a écrit le fichier `.c` indépendamment.

La compilation de `guard.c` échoue, le compilateur rapporte que la fonction `abs()` est définie plusieurs fois.

Exercice 46 (Règle de la simple définition)
Observez le résultat du traitement du préprocesseur sur le fichier `guard.c`.

Pourquoi Bob transgresse la règle de simple définition ? □

Bob pourrait corriger son fichier `guard.c`. Cependant sa première version est raisonnable et ne devrait pas poser de problème de compilation.

C'est à Alice de fournir, en tant que le développeuse expérimentée, de livrer des fichiers sources qui puissent toujours être utilisés.

Exercice 47 (*#include guard*)
Proposez une modifcation des fichiers d'Alice pour que Bob ne rencontre plus de problème de compilation, sans toucher à son programme.

Notez aussi que Alice ne souhaite pas fusionner les deux fichiers `.h` et qu'elle souhaite bien inclure `abs.h` dans `minmax.h` (pour que `min()` et `max()` profitent des prochaines évolutions de `abs()`). □

5 Compilation modulaire

Avec la notion de préprocesseur, la compilation modulaire est un mécanisme fondamental du langage C. Le but de la compilation modulaire et de concevoir un programme comme l'assemblage de plusieurs « modules ».

Chaque module est le fruit de la compilation séparée d'un fichier source (fichier `.c`) différent. Les modules sont autant de *fichiers objets* (fichiers `.o`) différents. Pour produire un fichiers objets à partir d'un fichier source il faut utiliser l'option `-c` du compilateur `gcc`.

L'assemblage des différents modules est appelé *linkage*, *liaison*, ou encore *édition de liens*. Lors de cette opération de liaison les différents fichiers objets sont réunis dans un seul fichier

exécutable. Pour que cette opération aboutisse il faut cependant qu'un et un seul des fichiers liés contienne la fonction `main()` qui sera lepont d'entrée de l'exécution du programme produit.

.

5.1 les fichiers objets

Vous avez réalisé un fichier `numbers.c` qui compile toute une série de fonctions d'affichage des entiers sous leurs formes décimale, hexadécimale ou binaire. Cet ensemble de fonctions est utile dans de nombreux programmes. Selon les canons de la compilation modulaire il est donc pertinent de compiler une fois cet ensemble de fonctions dans un fichier `.o` puis de réutiliser ces fonctions dans tous les programmes que vous produirez, sans les recompiler.

Vous allez travailler dans le répertoire `module/` de votre dépôt.

Exercice 48 (Un module `put_numbers.c`)
Donnez un fichier `put_numbers.c` qui propose une série de fonctions permettant d'afficher des nombres sur la sortie standard. Ce fichier ne contiendra pa la fonction `main()`.

Produisez un module `put_numbers.o` qui contiendra l'ensemble des fonctions que vous avez réalisées. Comment réaliser cette ligne de compilation ? □

Exercice 49 (Utiliser un module)
Produisez maintenant un fichier `test_numbers.c` qui implémente la fonction `main()` de test de l'ensemble des fonctions de votre module `put_numbers.o`.

Ce fichier ne doit pas contenir les fonctions testées, `putdec()`, `puthex()`...). Produisez un module `test_numbers.o` à partir de votre programme. Comment compiler ce programme ? La compilation provoque des *warning*. Pourquoi ? Que faut-il ajouter pour éviter ces *warning* de compilation ? □

Exercice 50 (Fichier de prototype d'un module)
Les déclarations nécessaires pourraient être placées dans le fichier `test_number.c` lui-même. Cependant ces mêmes modifications seraient à reporter dans chaque fichier source qui utiliserait des fonctions du module `put_number.c`.

Proposez une solution générique qui repose sur l'usage du préprocesseur et des fichiers de prototypes vus précédemment. □

Exercice 51 (Liaison de code)
Comment lier ce module `test_numbers.o` maintenant compilé sans *warning* avec le module `put_numbers.o` pour produire un programme executable `test_numbers` ? □

5.2 Compilation globale du projet

Finalement, votre programme `test_numbers` est composé de 3 fichiers sources et 3 fichiers contenant du code machine :

`put_numbers.c` contient le code source des fonctions d'affichage des nombres ;

`put_numbers.h` contient les prototypes des fonctions implémentées par `put_numbers.c` ;

`test_numbers.c` contient le code source du programme de test ;

`put_numbers.o` contient le code machine des fonctions d'affichage des nombres compilé avec `gcc` ;

`test_numbers.o` contient le code machine du programme principal de test ;

`test_numbers` contient le programme exécutable qui teste les fonctions d'affichage des nombres.

La compilation modulaire nous a permis *(i)* de structurer notre programme en différents fichiers servant des objectifs différents et *(i)* de produire une librairie de code `put_numbers.o` que l'on pourra réutiliser dans d'autres programmes grace au fichier `put_numbers.h` (en liant le `.o` avec les programmes qui l'utilise).

Cependant lorsque vous modifiez le fichier `put_numbers.c` la recompilation de l'exécutable `test_numbers` devient fastidieuse...

Exercice 52 (Un premier script de compilation)
Proposez un script `bash` que vous appellerez `compile_test_numbers.sh` qui exécute, en cascade l'ensemble des directives de compilation pour produire, à partir des fichiers sources, les fichiers `.o` et le fichier exécutable. □

Ce script effectue systématiquement toutes les phases de compilation pour tous les fichiers. L'un des intérêts de la compilation modulaire est justement de pouvoir ne recompiler que ce qui est nécessaire. Voyons comment améliorer cette première version du script.

Dans un script `bash`, il est possible de tester si un fichier a été modifié avant un autre fichier, en utilisant l'opérateur `-nt`, *newer than* :

```
if [ "file1" -nt "file2" ]; then
  echo "file1 has been changed after file2"
else
  echo "file1 has been changed before file2"
fi
```

Exercice 53 (Un script de compilation amélioré)
Modifiez votre fichier de script `bash` pour ne recompiler les fichiers `.o` et l'exécutable final, que lorsque cela est nécessaire :

— un fichier `.o` ne doit être recompilé que si le fichier `.c` correspondant a été modifié depuis la dernière compilation ;
— quand un fichier `.c` contient un `#include`, si ce fichier `.h` a été modifié, une recompilation du `.c` est nécessaire ;
— l'exécutable doit être lié à nouveau si l'un des `.o` qui le compose a été modifié (recompilé) ;

Une fois votre script mis au point, assurez-vous d'avoir tester ces différents cas de figures de recompilation. □

5.3 Compilation modulaire avec `make`

L'utilisation de fichiers de script permet de produire efficacement un programme (ou un ensemble programmes) en ne recompilant que les choses nécessaires. La mise au point de tels scripts est néanmoins délicate. De plus lorsque le projet évolue, il faut modifier le script en conséquence en intégrant les bonnes lignes de compilations aux bons endroits et en faisant évoluer les conditions de compilation.

Finalement, ce qu'exprime ce script de compilation, c'est que le fichier `test_numbers` « dépend » des fichiers `test_numbers.o` et `put_numbers.o`. Si l'un de ces deux fichiers a changé depuis la dernière production de `test_numbers`, il faut simplement exécuter la commande `gcc put_numbers.o test_numbers.o -o test_numbers`.

Dans un fichier `Makefile` on exprime cela par une *règle* comme suit :

```
test_numbers: test_numbers.o put_numbers.o
        gcc test_numbers.o put_numbers.o -o test_numbers
```

Le fichier `Makefile` peut contenir un série de *règles*. Chaque règle débute par une *ligne de dépendances* qui pour une `cible` donnée (`test_numbers`), liste ses *prérequis* (`test_numbers.o put_numbers.o`). Cette ligne de dépendances est suivie de lignes de commandes qui seront exécutée si un des prérequis est plus récent que la cible. Ces lignes de commandes shell doivent être précédées d'une tabulation en début de ligne.

Pour déclencher la recompilation conditionnelle d'un fichier, il suffit ensuite de lancer la commande `make`. Cette commande va essayer de reconstruire, si nécessaire, la cible de la

première règle trouvée dans le fichier `Makefile`. Si les prérequis de cette règle sont eux-mêmes l'objet d'autres règles de compilation, ils seront préalableent reconstruits si nécessaire, et ce récursivement.

Ainsi un fichier `Makefile` permet d'exprimer beaucoup plus simplement qu'un fichier de script ce qui doit être fait pour recompiler un programme.

Exercice 54 (Les `Makefile`)
Consultez le manuel en ligne `man` pour la commande `make` .

Réalisez un fichier `Makefile` qui réalise la même tâche que votre fichier de script `compile_test_numbers.sh`.

Testez votre `Makefile` de la même manière que vous aviez testé votre script de compilation. □

Notez que dans ce `Makefile`, contrairement au script shell, l'ordre dans lequel les règles sont données n'a pas d'importance. Tout au plus, vous pouvez notez que lorsque vous lancez la commande `make` sans paramètre, c'est la première règle trouvée dans le fichier qui est considérée.

L'usage (ou une bonne pratique) consiste à s'assurer que la première règle, la règle évaluée par défaut, assure la compilation de l'ensemble du projet. Pour assurer cela, on place généralement, en première règle du `Makefile` règle blanche (sans commande associée) qui s'écrit :

```
all: ...
```

Les `...` sont remplacés par la liste des règles qui doivent être évaluées lors du lancement du `make`. Donc votre cas, `make` doit produire une chose : `test_numbers`.

Il est aussi d'usage qu'un `Makefile` propose une règle `clean` qui supprime tous les fichiers compilés (`.o` et exécutables). Cette règle ne dépendant de rien, et elle consiste en l'exécution d'une commande `rm` avec les paramètres appropriés.

Enfin, une bonne pratique consiste à définir un certains nombre de variables au début du `Makefile` qui pourront être changées, au besoin et qui définissent :

`CC` pour le compilateur C à utiliser (c);

`SRC_DIR` pour le répertoire ou les fichiers sources sont placés;

`BLD_DIR` pour le répertoire ou les fichiers compilés doivent être placés.

Il est ensuite possible d'utiliser ces variables dans le `Makefile` lors de la description des commandes shell à lancer en écrivant `$(VAR)` la ou elles doivent être utilisées.

De la même façon il est possible de faire référence au fichier cible de la règle courante avec `$@`, à l'ensemble des fichiers sources de la règle courante avec `$<` et à l'ensemble des fichiers sources avec `$^`.

Exercice 55 (Règle pour fabriquer la cible)
Considérez la règle suivante :

```
$(BLD_DIR)/test_numbers: $(BLD_DIR)/test_numbers.o $(BLD_DIR)/put_numbers.o
        $(CC) $^ -o $@
```

Quel est le sens de cette règle ?

Quelle sera la commande exécutée si cette règle est considérée ?

Modifiez votre `Makefile` pour qu'il utilise cette règle. Ré-écrivez les autres règles sur le même principe. Proposez aussi une règle par défaut `all` et une règle de nettoyage `clean`. □

Dans un répertoire de projet, il est d'usage de placer les fichiers sources dans un répertoire `src/` les fichiers compilés dans un répertoire `build/`, et de proposé à la racine un fichier `Makefile` et un fichier `README.md`.

Exercice 56 (Mise en oeuvre)
Faites évoluer votre répertoire de travail de la sorte. Assurez-vous que l'on puisse compiler l'ensemble du projet en lançant simplement une commande depuis le répertoire racine `make`. □

Bravo ! Vous savez créer et organiser un projet en langage C !

6 Mes commandes Unix

Pour terminer ces travaux pratiques, nous allons implanter notre propre version de quelques commandes Unix bien connues.

Nous travallerons dans le répertoire `mcu/`, *mes commandes Unix*, de notre dépôt.

Un filtre est une commande qui lit un texte à traiter sur son entre standard (`stdin`), produit son résultat sur la sortie standard (`stdout`), et produit éventuellement des messages d'erreur — dans l'idéal sur la sortie d'erreur (`stderr`). De plus, un filtre retourne 0 s'il se termine correctement et un code d'erreur dans le cas contraire.

Pour faire simple, nous n'allons implanter que des filtres ne prenant aucune option et qui traitent l'entrée standard ligne par ligne. On convient que nos versions basiques de ces commandes considérent que les lignes font au plus 80 caractéres. Si l'entrée standard fournie contient une ligne de plus de 80 caractères, le filtre se termine sur un échec et le code d'erreur retourné est 1.

Nous allons implanter des versions simplifiées des commandes UNIX suivantes :

— `mcu_wc` qui retourne sur la sortie standard le nombre de caractères — octets — composant le fichier ;

— `mcu_wl` qui retourne sur la sortie standard le nombre de lignes composant le fichier (une ligne se termine par le caractère '
`n');`

— `mcu_ww` qui retourne sur la sortie standard le nombre de mots composant le fichier (deux mots sont séparés par un caractère de ponctuation *e.g.* espace, virgule, etc) ;

— `mcu_rev` qui renverse l'ordre des caractères pour chaque ligne. Par exemple, on a :

```
bash$ echo "la vie est belle" | mcu_rev
elleb tse eiv al
bash$
```

— `mcu_uniq` reproduit sur la sortie standard chaque ligne lues depuis l'entrée standard sauf si elle fait doublon avec la ligne précédente. Par exemple, on a :

```
bash$ echo "Hello" > essai
bash$ for((i=0;i<5;i++));do echo "Hello World" >> essai ;done;
bash$ mcu_uniq < essai
Hello
Hello World
bash$
```

— `mcu_xxd` retourne sur la sortie standard une représentation formatée de l'entrée standard. Chaque ligne de sortie correspond à 16 caractères en entrée et on a le formattage en colonne suivant :

  — la première colonne de 7 caractères représente l'offset du premier caractère du paquet de 16 caractères considérés du fichier ;

  — la seconde colonne comporte 32 caractères groupés par paquet de 4. Elle indique le codage hexadécimal des caractères considérés ;

  — la dernière colonnes de 16 caractères correspond à l'affichage de ces derniers (les caractères non imprimables — e.g. retour charriot — sont remplacés par un point).

  Par exemple, on a :

```
bash$ echo "Hello Unix World !" | ./mcu_xxd
000: 4865 6c6c 6f20 556e 6978 2057 6f72 6c64  Hello Unix World
010: 2021 0a                                   !.
bash$
```

La compilation et la mise à jour des exécutables en fonction des évolutions du code se fera en utilisant la commande `make`. Vous devez donc construire le `makefile` correspondant au fur et à mesure des exercices.

6.1 Fonctions partagées

Nous allons commencer par coder des fonctions qui vont être utilisées dans plusieurs exécutables que nous devons implanter. Pour décrire ces fonctions, nous indiquons ci-dessous une partie des fichiers d'entêtes contenant leurs prototypes et des macros utiles :

— `mcu_macros.h:Sources/mcu_macros.h`
— `mcu_fatal.h:Sources/mcu_fatal.h`
— `mcu_readl.h:Sources/mcu_readl.h`
— `mcu_affiche_entier.h:Sources/mcu_affiche_entier.h`

Exercice 57 (Implantation des fonctions partagées)
Implanter ces fonctions et assurez vous de leurs bon fonctionnement. Prenez soin à ce que chaque fonction soit dans un fichier source différent et que votre code fonctionne avec le fichier de test suivant : `Sources/mcui_test.c` pour produire le résultat suivant :

```
bash$ ./mcu_test < mcu_test.c ; echo "le code de retour est $?"
#include <stdio.h>
18
1==1 is true
le code de retour est 2
bash$
```

après la compilation :

```
bash$ make mcu_test
gcc  -c mcu_affiche_entier.c
gcc  -c mcu_fatal.c
gcc  -c mcu_readl.c
gcc  -c test.c
gcc  -o test mcu_affiche_entier.o mcu_fatal.o mcu_readl.o test.o
bash$
```

□

6.2 les commandes UNIX proprement dites

Exercice 58 (commandes de comptage)
Les filtres `mcu_wc` et `mcu_ww` doivent fournir les résultats suivants sur ces exemples :

```
bash$ echo "Hello Unix World!" | ./mcu_wc
18
bash$ echo "Hello unix World!" | ./mcu_ww
3
bash$
```

Le filtre `mcu_wc` est très simple alors que le filtre `mcu_ww` nécessite plus de technique comme l'utilisation d'un automate simple.

Pensez à vérifier votre code. Par exemple :

```
bash$ echo "`wc -c < mcu_wc.c` - `./mcu_wc < mcu_wc.c`" | bc
0
```

Implanter les filtres `mcu_wc` et `mcu_ww`. □

Exercice 59 (commandes `mcu_wl` et `mcu_rev`)
Nous allons maintemant utiliser la fonction `readl` de la librairie standard pour implanter les filtres `mcu_wl` et `mcu_rev`. Vous devriez obtenir ce genre de résultat.

```
bash$ echo "Hello World," > essai
bash$ echo "Hello Unix World!" >> essai
bash$ ./mcu_wl < essai
2
bash$ ./mcu_rev < essai
,dlrow olleH
!dlroW xinU olleH
bash$
```

Implanter les filtres `mcu_wl` puis `mcu_rec`. □

Exercice 60 (commande `uniq`)
Implanter le filtre `mcu_uniq` décrit précédemment. □

Exercice 61 (commande `mcu_xxd`)
L'exécution de la commande `mcu_xxd` doit vous permettre d'obtenir ce genre de résultat (en utilisant le fichier créer dans l'exercice précédent).

```
bash$ ./mcu_xxd < essai
00000000: 4865 6c6c 6f20 776f 726c 642c 0a48 656c  Hello world,.Hel
00000010: 6c6f 2055 6e69 7820 576f 726c 6421 0a    lo Unix World!.
bash$
```

Implanter le filtre `mcu_xxd` décrit précédemmment. □

Indication. Pour faire simple, vous pouvez lire les caractères et remplir un tampon (i.e. un tableau de caractères) en ajoutant au fur et à mesure les champs nécessaires. Lorsque ce tampon est plein, vous devez retourner ce tampon sur la sortie standard.

Fin du sujet