Université de Lille 1 SCIENCES ET TECHNOLOGIES

Licence d'informatique
2e année
Module de BPC

Bases de la Programmation en C

Gilles GRIMAUD Philippe MARQUET

Révision majeure, janvier 2018
version de janvier 2020

Ce sujet est disponible à `https://gitlab-etu.fil.univ-lille1.fr/ls4-pdc/theme1`.

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 !

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.

Ce support sert à la fois pour les TDs et les TPs. Sauf mention contraire, chaque exercice est à préparer en TD, et ensuite à implémenter, compiler et tester en séance de TP.

Thème 1 : Les expressions, les types entiers, les fonctions

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 recherche 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.

Vous pouvez trouver la documentation de votre shell en tapant la commande `man bash`. De manière générale, vous pouvez trouver la documentation de `commande` en tapant `man commande`.

Exercice 1 (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 2 (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 3 (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 4 (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.

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 BPC.

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 5 (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 2041/42.

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 6 (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 locale du dépôt sur laquelle il va être possible de travailler.

Exercice 7 (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).

Vous avez normalement vu l'utilisation de Git au premier semestre, cependant un bref rappel est peut-être nécessaire : si vous en avez besoin, vous pouvez passer un peu de temps à manipuler votre dépôt. Vous pouvez par exemple modifier et ajouter des fichiers, réaliser un commit et un push, puis demander à votre binome de récuperer vos modifications avec un pull.

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 recourent 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 8 (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 !) □

2 Premiers pas en C

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

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*.

Git ignore ***...en savoir plus...***

Le dépôt ne doit contenir que les seuls fichiers ≪ utiles ≫.
Les fichiers de sauvegarde de votre éditeur, par exemple `prog1.c~`, ou les fichiers exécutables, par exemple `a.out` qui existe dans la copie locale du dépôt n'ont pas à être enregistrés dans le dépôt. Ils n'ont qu'un intérêt temporaires (les `*~`), ou peuvent être reconstruits (les exécutables par exemple).
On peut indiquer à Git d'ignorer ces fichiers en plaçant un fichier `.gitignore` à la racine du dépot.
Chaque ligne identifie un motif de noms de fichiers qui doivent être ignorés, par exemple :

```
bash$ cat .gitignore
a.out
*~
```

Ce fichier lui-même peut être (est) géré par Git !
On viendra compléter ce fichier au fur et à mesure.

Exercice 9 (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 10 (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 11 (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 [1], 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 [2] .

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

Exercice 13 (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$
```

1. 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 $N = 64$.
2. 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`.

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

Exercice 14 (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'exercice 12. Que pouvez-vous en dire ? □

3 Exercices de préparation

Exercice 15 (Les expressions et leurs évaluations)
Ceci est un exercice de TD uniquement.

En supposant que :

$$A = 20 \quad B = 5 \quad C = -10 \quad D = 2 \quad X = 12 \quad Y = 15$$

évaluer les expressions valides parmi les expressions suivantes :

$5 * X + 2 * 3 * B/4$	$A == B = 5$	$A+ = X + 2$	$A! = C* = -D$
$A\% = D + +$	$A\% = + + D$	$(X + +) * A + C$	$A = 6 * D == X?B : Y$
$!(X - D + C)\| \| + +D$	$A\&\&B\|\|!0\&\&C\&\&!D$	$C = 12 - -$	`(1<<(2<<1))==((1<<2)<<1)`

En utilisant le parenthésage, corriger les expressions invalides.

Exercice 16 (Type atomique et conversion implicite)
Ceci est un exercice de TD uniquement.

En supposant faites les déclaration :

```
long int A = 15 ;
char B = 'A' ;
short int C = 10 ;
```

évaluer et donner le type des expressions valides suivantes :

$B + 1$	$B + A$	$B + C$
$3 * B + 2 * B$	$2 * B + (A + 10)/C$	$2 * B + (A + 10.0)/C$
$B = 2 * B + (A + 10.0)/C$	$C = 666$	$B* = 3.14$

Exercice 17 (Que fait ce programme ?)
La fonction `int putchar(int)` prends en argument un code ASCII (un entier), et affiche sur la sortie standard le caractère correspondant à ce code ASCII. De plus, la fonction renvoie le code ASCII du caractère (en cas de succes), ou bien la valeur spéciale EOF en cas d'échec.

```
int foo(int x) {
  int a = putchar(x);
  return 42;
}

main() {
  int b = foo(65); /* 65 est le code ascii de A */
}
```

Que fait ce programme ? Quelle est la valeur que recoit la variable `a` (en supposant que l'appel de `putchar` est un succes) ? Meme question pour la variable `b`. i En supposant que le source

soit dans le fichier `prog.c`, quelle est la ligne de commande shell à taper pour créer un fichier exécutable correspondant `prog` ?

Quelle est la ligne de commande shell à taper pour lancer le fichier exécutable `prog` ?

Sur vos machines de TP, testez ce programme et vérifiez que le résultat est conforme à votre prédiction.

Exercice 18 (Géométrie)
Créer une fonction `float square(float)` qui permet de renvoyer le carré d'un nombre passé en paramètre.

Créer ensuite une fonction :

```
int is_within_distance(float x1, float y1, float x2, float y2, float dist)
```

qui utilise la fonction `square`, et qui renvoie 1 si la distance entre le point aux coordonnées (x1, y1) et le point aux coordonnées (x2, y2) est inférieur à `dist` (on renvoie 0 dans le cas contraire).

Créer ensuite un programme pour tester ces fonctions.

4 Projet : 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 affichages de cette même valeur « entière » peuvent être utiles. On peut par exemple vouloir 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.

Un `Makefile` vous est fourni, ce qui vous permettra de compiler votre fichier `numbers.c` en tapant simplement la commande `make`.

Le Makefile est un fichier qui vous permet d'automatiser les tâches répétitives du développement (par exemple la compilation). De plus amples détails sur l'utilisation des Makefiles vous seront fournies plus tard.

4.1 Afficher des caractères

Exercice 19 (Afficher un chiffre (décimal))
Proposez une implémentation de la fonction de prototype

```
int put_digit(int digit);
```

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)...

□

On utilisera la fonction de prototype `int putchar(int)` pour réaliser l'affichage.

La fonction `put_digit` renverra 0 en cas de succès, ou -1 en cas d'erreur (par exemple, si l'argument passé n'est pas compris entre 0 et 9, ou bien si l'appel de `putchar` échoue).

Créez ensuite un programme principal (fonction `main`) qui appelle cette fonction avec un chiffre de votre choix, ainsi qu'avec un argument invalide, et qui vérifie que la valeur de retour est bien celle attendue, dans les deux cas.

Exercice 20 (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 :

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`.

— entre `'0'` et `'9'` si $0 \leq h \leq 9$; ou
— entre `'A'` et `'F'` si $10 \leq h \leq 15$.

□

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 21 (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. □

4.2 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 (donc 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 22 (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 23 (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 maintenant l'exécution de ce programme ? Et surtout... pourquoi ? □

Exercice 24 (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 25 (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 26 (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 ! □

Exercice 27 (Test avec le Makefile)
Vous allez modifier votre fichier `Makefile` pour qu'il permette de réaliser automatiquement les tests.

Un Makefile est composé d'un ensemble de règles. Chaque règle définit comment réaliser quelque chose (par exemple : produire un fichier exécutable, ou bien réaliser les tests).

Chaque possède 3 élements :

— La `cible`, c'est un nom pour le "résultat" produit par la règle.
— Les commandes shells utilisées pour produire ce résultat.
— Les fichiers qui doivent être au préalable présents avant de lancer ces commandes (c'est les `dépendances`).

Par exemple, pour produire l'exécutable `toto` à partir du source `toto.c`, il faut taper la commande `gcc -o toto toto.c`. Donc, si on voulait créer une règle de Makefile pour automatiser cela, la cible serait `toto`, la commande serait `gcc -o toto toto.c`, et la dépendance serait `toto.c` (car il y a besoin que `toto.c` soit présent pour lancer le processus de compilation).

Les règles dans le Makefile sont formattées de la sorte :

```
maCible: maDependance1 maDependance2 MaDependance3 [... etc]
        commande shell 1
        commande shell 2
        [... etc]
```

Attention : l'indentation des commandes shell doit impérativement être faite par une tabulation, et non des espaces.

Vous pouvez avoir autant de règles que vous le souhaitez dans le Makefile. Lorsque vous tapez la commande `make`, c'est la première règle qui va être utilisée. Si vous voulez utiliser une autre règle, il faudra taper `make <cible>` (où `<cible>` est le nom de la cible de la règle que vous voulez utiliser).

Vous pouvez éditer le Makefile fourni pour compiler `numbers.c` et examiner la règle pour compiler `numbers.c`

Vous constaterez également qu'il existe une cible `test` dans le Makefile, mais celle ci n'est pas complète. Vous allez la compléter. Pour ceci, il va falloir déterminer les dépendances, et la commande à lancer pour réaliser le test.

Quelles sont les dépendances de la cible `test` ? (En d'autre mots, quels fichiers doivent être présents pour que les tests puissent être lancés ?)

Pourquoi `numbers.c` n'est pas une dépendance de la cible `test` ?

Quelle est la commande shell à lancer pour faire le test ? (indice : vous l'avez déja lancée lors de l'exercice précédent)

Enfin, terminez en modifiant votre Makefile pour mettre en place ce test. Vérifiez que cela fonctionne en tapant `make test`

[Ici, faire quelque chose du style de ce qui était fait en PDC avec les putbin/puthex et compagnie.]

[Fournir un Makefile permettant d'exécuter des tests unitaires, qui devront "passer" avant de rendre le TP.]