ENSTA Paris
2023-2024

Résolution de problèmes algorithmiques – IN103

Alexandre Chapoutot

Feuille d'exercices 1

Objectif(s)

- Comprendre la compilation séparée
- Écrire un Makefile

Exercice 1 – Compilation séparée

Allez dans le répertoire `exo1`. Vous y trouverez plusieurs fichiers
- `bubble_sort.c`
- `my_prog.c`
- `simulation.c`

Question 1

Donnez la ligne de commande pour compiler le fichier `my_prog.c` pour générer un exécutable nommé `my_prog.x`.

Solution:
La ligne de commande est : `gcc -Wall -o my_prog.x my_prog.c`

Question 2

Compilez le fichier `my_prog.c`, que se passe-t-il ? Essayez d'expliquer l'erreur.

Solution:
Un avertissement et une erreur de compilation apparaissent

```
my_prog.c: In function 'main':
my_prog.c:16:3: warning: implicit declaration of function 'bubble_sort' [-Wimplicit-function-declaration]
   16 |   bubble_sort (array, size);
      |   ^~~~~~~~~~~
Undefined symbols for architecture x86_64:
  "_bubble_sort", referenced from:
      _main in ccgEB0qU.o
ld: symbol(s) not found for architecture x86_64
collect2: error: ld returned 1 exit status
```

Explication(s):

- le code source `my_prog.c` utilise une fonction `bubble_sort` dont le prototype n'est pas connu, le compilateur essaie de déduire une information suivant la manière dont est utilisée la fonction. Cette étape n'est pas bloquante mais peut conduire à des comportements erronés par la suite ;
- De plus, le compilateur indique qu'il n'a pas accès au code de la fonction `bubble_sort` ce qui l'empêche de générer un fichier binaire `my_prog.x`. Dans cette étape le compilateur ne peut pas inventer un code associé à cette fonction d'où l'arrêt du processus de compilation.

Question 3

Compilez maintenant en utilisant les deux fichiers `bubble_sort.c` et `my_prog.c`. Essayez d'expliquer ce qui se passe.

Solution:

À partir de la ligne de compilation

```
gcc -Wall -o my_prog.x my_prog.c bubble_sort.c
```

seul un avertissement est levé par le compilateur mais le fichier binaire est créé.

```
my_prog.c: In function 'main':
my_prog.c:17:3: warning: implicit declaration of function 'bubble_sort' [-Wimplicit-function-declaration]
   17 |   bubble_sort (array, size);
      |   ^~~~~~~~~~~
```

Explication(s):

- Lors de l'étape de compilation le fichier `bubble_sort.c` est transformé en code binaire donnant accès au code de la fonction `bubble_sort` dans le fichier `my_prog.c` qui l'utilise. Cependant lors de la compilation du fichier `my_prog.c` l'existence de la fonction n'est pas encore connue d'où le message d'avertissement (cf correction question précédente). La création du programme `my_prog.x` utilise un programme, nommé l'éditeur de liens, qui fait la mise en correspondance entre les différents codes binaires générés. En particulier, il fait le lien entre le besoin du code d'une fonction `bubble_sort` dans le code du fichier `my_prog.c` et le code de la fonction `bubble_sort` issu du fichier `bubble_sort.c`

Question 4

Supprimez le fichier exécutable `my_prog.x` et recompilez le programme comme précédemment en ajoutant l'option `-Werror`. Que se passe-t-il ?

Solution:

En utilisant

```
gcc -Wall -Werror -o my_prog.x my_prog.c bubble_sort.c
```

Explication(s):

- Tous les avertissements sont considérés comme des erreurs et la compilation est donc stoppée. En conséquence, le fichier exécutable `my_prog.x` n'est pas créé.

Question 5

En utilisant la directive du pré-processeur `#include` qui inclue le contenu d'un fichier dans un autre, utilisez cette directive pour inclure le fichier `bubble_sort.c` dans le fichier `my_prog.c`. Recompilez comme précédemment en utilisant les deux fichiers `bubble_sort.c` et `my_prog.c`. Que se passe-t-il ? Essayez d'expliquer ce qui se passe.

Solution:

La ligne de compilation

```
gcc -Wall -Werror -o my_prog.x my_prog.c bubble_sort.c
```

génère une erreur qui empêche la production du fichier binaire `my_prog.x`

```
duplicate symbol '_bubble_sort' in:
    /var/folders/tj/fxt8btk175vgg9rb875jfxdh0000gp/T//cc9OpC4r.o
    /var/folders/tj/fxt8btk175vgg9rb875jfxdh0000gp/T//ccxLPyM9.o
ld: 1 duplicate symbol for architecture x86_64
collect2: error: ld returned 1 exit status
```

Explication(s):

- Dans l'environnement de travail du compilateur `gcc` deux fonctions `bubble_sort` sont compilées. La première vient de l'inclusion du fichier `bubble_sort.c` dans le fichier `my_porg.c`. Le seconde vient de la compilation du fichier `bubble_sort.c` lui-même. L'éditeur de lien ne sait pas quelle fonction `bubble_sort` choisir pour générer le fichier exécutable `my_prog.x` d'où le message d'erreur.
- Compilez avec la ligne de commande

  ```
  gcc -Wall -Werror -o my_prog.x my_prog.c
  ```

  On constate dans ce cas que la compilation se déroule sans difficulté car une seule occurrence de la fonction `bubble_sort` existe.

Question 6

Les précédentes questions permettent de mettre en avant les concepts de

— **déclaration** de fonctions, l'étape consistant à donner le *prototype* (aussi appelé *signature*) d'une fonction.
— **définition** de fonctions, l'étape consistante à donner le *corps* (c'est-à-dire le code) de la fonction.

En langage C, les déclarations de fonctions sont stockés dans des fichiers `.h` et les définitions de fonctions dans des fichiers `.c`.

Définissez un fichier `bubble_sort.h` contenant la déclaration de fonction `bubble_sort` et l'inclure dans le fichier `my_prog.c`. Utilisez la ligne de compilation avec les deux fichiers `bubble_sort.c` et `my_prog.c`. Que se passe-t-il ?

Solution:
Le fichier `bubble_sort.h` contient

```
int bubble_sort (int* array, unsigned int size);
```

Explication(s):

- ✿ Pour compiler, il faut retourner à l'utilisation des deux fichiers `my_prog.c` et `bubble_sort.c`
  `gcc -Wall -Werror -o my_prog.x my_prog.c bubble_sort.c`
- ✿ En synthèse, il faut insister que la seule bonne façon de faire est une inclusion de fichiers `.h` et pas l'inclusion de fichier `.c` pour faire de la compilation séparée.

Exercice 2 – Deboggage de programmes

Allez dans le répertoire `exo2`, dans lequel plusieurs fichiers sont présents. Ces fichiers permettent de calculer une base orthornormale de vecteurs à l'aide de l'algorithme de Gram-Schmidt [1].

Plusieurs erreurs ont été introduites dans les fichiers ce qui empêchent la compilation. Après une tentative de compilation, suivez les messages d'erreur pour les corriger.

Question 1

Donnez la ligne de commande pour compiler ces fichiers en un programme nommé `my_prog.x`

Solution:
La ligne de commande est :
`gcc -Wall -Werror -o my_prog.x my_prog.c blas.c gs.c`

Question 2

Quelle est la première erreur qui apparaît dans le processus de compilation ? Comment pouvons-nous la corriger ?

Solution:

```
In file included from my_prog.c:4:
./gs.h:3:9: error: unknown type name 'vector_t'
typedef vector_t* matrix_t;
```

Explication(s):

- ✿ Première étape, pensez à utiliser la commande Linux `grep` pour trouver où est définit le type `vector_t`.
- ✿ Le fichier `gs.h` n'inclut pas le fichier `blas.h` ce qui a pour conséquence que la définition du type `vector_t` n'est pas connue. Il faut donc ajouter la ligne `#include "blas.h"`

Question 3

Après avoir corrigé l'erreur précédente, relancer le processus de compilation. Quelle est la nouvelle (première) erreur qui survient ? Comment pouvons nous la corriger ?

1. https://fr.wikipedia.org/wiki/Algorithme_de_Gram-Schmidt

Solution:

```
blas.c:40:10: error: implicitly declaring library function 'sqrt' with type 'double␣(double)' [-Werror,-Wimp
  return sqrt(result);
         ^
blas.c:40:10: note: include the header <math.h> or explicitly provide a declaration for 'sqrt'
1 error generated.
```

Explication(s):

- ✿ Il faut inclure le fichier `math.h` (comme ici le suggère `gcc`) qui contient la définition des fonctions mathématiques qui sont utilisées dans le fichier `blas.c`.
- ✿ Un autre effet de cette inclusion est qu'il faut ajouter le drapeau `-lm` dans la ligne de compilation pour faire le liens avec la bibliothèque mathématique de C. A noter qu'il semble que les dernières versions de `gcc` ou `clang` font ce lien automatiquement. Il est cependant préférable d'indiquer aux étudiants de positionner ce drapeau.

Question 4 – Pour aller plus loin (Optionnel)

Note erreur qui n'est pas générée sous Linux mais uniquement sous Mac OS. Après avoir corrigé l'erreur précédente, relancer le processus de compilation en ajoutant l'option `-std=c99 -pedantic` (révision du langage C datant de 1999 et une application stricte de la norme). Quelle la nouvelle (première) erreur survient ? Comment pouvons nous la corriger ?

Solution:
En utilisant la ligne de commande `gcc -Wall -Werror -std=c99 -pedantic -o my_prog.x blas.c gs.c my_prog.c -lm` on obtient

```
In file included from my_prog.c:5:
./blas.h:3:17: error: redefinition of typedef 'vector_t' is a C11 feature [-Werror,-Wtypedef-rede
typedef double* vector_t;
```

A noter que cette erreur disparaît avec l'option `-std=c11` (révision du langage C datant de 2011) qui est le comportement par défaut de `gcc`.

Explication(s):

- ✿ Cette erreur signifie qu'il y a plusieurs définitions du type `vector_t`. Il faut donc protéger contre les inclusions multiples le fichier `blas.h`.
- ✿ Dans cette question, on ne donne le mécanisme du pré-processeur pour protéger les inclusions multiples (soit `#pragama once` ou avec les `include guards` cf code en solution). Il faut comprendre que la directive `#include` est un traitement très bête.
  Remarque : dans cette question on donne la réponse car les étudiants ne pourront pas la trouver seuls. Cette question est présente juste pour expliquer ce que les étudiants trouveront dans les fichiers de la bibliothèque C que nous utiliserons dans la suite du cours.
- ✿ Au final on le fichier suivant.

```c
#ifndef __BLAS_H__
#define __BLAS_H__

#include <stdlib.h>

typedef double* vector_t;

/* Compute alpha * x + beta * y */
/* Assume that x and y have the same size */
vector_t axby (int size, vector_t x, vector_t y, double alpha, double beta);

double dot (int size, vector_t x, vector_t y);

double norm2 (int size, vector_t x);

vector_t normalize (int size, vector_t x);

void printV (int size, vector_t x);

#endif
```

Remarque : on a definit une constante du pré-processeur avec la directive `#define` sans donner de valeur (c'est une différence par rapport aux cours précédents). Il faut indiquer que nous souhaitons juste vérifier la présence (la définition) d'une constante ou pas en se fichant de la valeur qu'on lui donne. Il ne faut pas utiliser cette constante pour autre chose qu'empêcher les inclusions multiples !

Exercice 3 – Makefile

Allez dans le répertoire `exo3` dans lequel vous trouverez un ensemble de fichiers permettant de calculer les racines d'un polynôme du second degré à cœfficients réels. Plus précisément, nous avons les fichiers suivants :

— `complex.h` et `complex.c` qui contiennent respectivement les déclarations et les définitions des fonctions manipulant des nombres complexes ;
— `solve.h` et `solve.c` qui contiennent respectivement les déclarations et les définitions des fonctions qui permettent de résoudre une équation du second degré ;
— `my_prog.c` qui contient la fonction principale.

L'objectif de cet exercice est d'écrire un fichier `Makefile` qui permet d'automatiser la compilation. Différentes versions seront écrites pour introduire progressivement les constructions des fichiers `Makefile`.

Question 1

Avant d'écrire le `Makefile` donnez les lignes de compilations pour générer
— le fichier `complex.o` ;

— le fichier `solve.o` ;
— le fichier `my_prog.o` ;
— le programme final `my_prog.x` qui nécessite l'utilisation des fichiers précédents.

Solution:
— `gcc -Wall -Werror -o complex.o -c complex.c`
— `gcc -Wall -Werror -o solve.o -c solve.c`
— `gcc -Wall -Werror -o my_prog.o -c my_prog.c`
— `gcc -Wall -Werror -o my_prog.x complex.o solve.o my_prog.o`

Explication(s):

✿ L'idée est de montrer à l'étudiant qu'un Makefile est un ensemble de règles qui effectuent ce genre de travail avec en plus une gestion des dépendances pour recompiler que ce qui est nécessaire. Cependant, rien n'est magique les dépendances sont données par le concepteur du `Makefile`.

Question 2

Pour rappel un fichier `Makefile` est constitué de règles de la forme :

```
cible: dépendances
      actions
```

Créez un fichier `Makefile` avec plusieurs règles pour générer les différents fichiers `complex.o`, `solve.o`, `my_prog.o` et `my_prog.x`

Exécutez le Makefile en lançant la commande `make`. Que se passe-t-il ?

Solution:

```
complex.o: complex.c
        gcc -Wall -Werror -c complex.c -o complex.o

solve.o: solve.c
        gcc -Wall -Werror -c solve.c -o solve.o

my_prog.o: my_prog.c
        gcc -Wall -Werror -c my_prog.c -o my_prog.o

my_prog.x: complex.o solve.o my_prog.o
        gcc -o my_prog.x complex.o solve.o my_prog.o -lm
```

Explication(s):

✿ l'ordre des règles est importante. Par défaut `make` exécute la première directive du fichier `Makefile`. En conséquence, seule la directive liée à `complex.o` est exécutée (Hypothèse : que les étudiants on écrit les règles en suivant le schéma de compilation de la question précédente) ;

✿ Par convention, on ajoute une première règle nommée `all` qui permet d'exécuter la règle "principale" dans notre cas, il faut ajouter la règle suivante (sans action) en début de fichier pour compiler le programme `all: my_prog.x`.
On peut aussi changer l'ordre des directives pour mettre celle associée à `my_prog.x` en première position. Cependant, si nous voulons par la suite ajouter d'autres règles de production, nous serons embêtés. La règle `all` permet de gérer ce cas.

Question 3

Dans le fichier `Makefile` ajoutez une règle `clean` qui supprime les fichiers `*.o` générés par la compilation. Ajoutez également une règle nommée `realclean` qui dépend de la règle `clean` et qui supprime le fichier `my_prog.x`

Solution:

```
all: my_prog.x

my_prog.x: complex.o solve.o my_prog.o
	gcc -o my_prog.x complex.o solve.o my_prog.o -lm

complex.o: complex.c
	gcc -Wall -Werror -c complex.c -o complex.o

solve.o: solve.c
	gcc -Wall -Werror -c solve.c -o solve.o

my_prog.o: my_prog.c
	gcc -Wall -Werror -c my_prog.c -o my_prog.o

clean:
	rm -f my_prog.o complex.o solve.o

realclean: clean
	rm -f my_prog.x
```

Explication(s):

- ✿ Il y a normalement pas de difficulté
- ✿ Pour exécuter ces règles il suffit d'appeler une des commandes suivantes :
  `make clean` ou `make realclean`
- ✿ On peut donc appeler la commande `make` suivi du nom du règle (on peut aussi faire le test avec `make all`).

Question 4

Un fichier `Makefile` autorise l'utilisation de variables. Des variables spécifiques `CC` ou `CFGLAS` permettent de renseigner quel commande à utiliser pour compiler les programmes écrits en langage C et quelles options sont à utiliser lors de la compilation. Ces variables permettent également de factoriser les règles pour ne pas dupliquer l'information.

Modifiez les directives du fichier `Makefile` pour utiliser les variables `CC` et `CFLAGS`.

Solution:

```
CC=gcc
CFLAGS=-Wall -Werror

all: my_prog.x

my_prog.x: complex.o solve.o my_prog.o
	$(CC) -o my_prog.x complex.o solve.o my_prog.o -lm

complex.o: complex.c
	$(CC) $(CFLAGS) -c complex.c -o complex.o

solve.o: solve.c
	$(CC) $(CFLAGS) -c solve.c -o solve.o

my_prog.o: my_prog.c
	$(CC) $(CFLAGS) -c my_prog.c -o my_prog.o

clean:
	rm -f my_prog.o complex.o solve.o

realclean: clean
	rm -f my_prog.x
```

Question 5 – Pour allez plus loin (Optionnel)

La génération de fichiers `*.o` à partir de fichiers `*.c` suivent un même schéma. Dans le fichier `Makefile` il est possible de définir des "pattern" de règles. En particulier, le caractère `%` est un joker pour représenter le nom d'un fichier. Par exemple, `%.o: %.c` indique une règle dont le produit (un fichier .o) dépend d'un fichier .c qui a le même nom (mais pas la même extension).

Une conséquence d'utilisation des ces patterns de règles de compilation est la difficulté de nommer les fichiers qui sont impliqués dans la production d'une règle implicite. Il y a plusieurs variables qui existent pour cela :

— `$@` correspond à l'élément associé à une cible d'une règle ;
— `$<` correspond au premier élément de la liste des dépendances d'une règle ;
— `$^` correspond à la liste des dépendances d'une règle.

Modifiez le fichier `Makefile` pour rendre générique la production de fichiers `.o`

Solution:

```
CC=gcc
CFLAGS=-Wall -Werror

all: my_prog.x

my_prog.x: complex.o solve.o my_prog.o
	$(CC) -o $@ $^ -lm

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f my_prog.o complex.o solve.o

realclean: clean
	rm -f my_prog.x
```

Question 6 – Pour allez plus loin (Optionnel)

Pour continuer l'utilisation des variables dans les fichiers `Makefile` il est possible de transformer une liste de noms de fichier. Par exemple, il est possible de transformer une liste liste de fichiers `.c` en une liste de fichier `.o`.

La construction à utiliser est la fonction `patsubst` qui à la signature suivante : `\$(patsubst pattern,replacement,text)`. Son fonctionnement est (traduction de la documentation [2])

> Trouve les mots séparés par des espaces dans la variable `text` qui correspondent au modèle donné par `pattern` et les remplace par le contenu de `replacement`. Ici, le motif peut contenir un '%' qui agit comme un caractère générique, correspondant à n'importe quel nombre de caractères dans un mot. Si le remplacement contient également un '%', le '%' est remplacé par le texte qui correspond au '%' du modèle donné par `pattern`.

Modifiez le fichier `Makefile` en créant une variable `SRC` qui contient la liste des fichiers `.c` et en créant une variable `OBJ` qui est générée à partir de la variable `SRC` qui correspond aux fichiers `.o`

Solution:

```
SRC=complex.c solve.c my_prog.c
OBJ=$(patsubst %.c, %.o, $(SRC))

CC=gcc
CFLAGS=-Wall -Werror -std=c99 -pedantic -O2

all: my_prog.x

my_prog.x: $(OBJ)
	$(CC) -o $@ $^ -lm

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f $(OBJ)

realclean: clean
	rm -f my_prog.x
```

2. https://www.gnu.org/software/make/manual/html_node/Text-Functions.html

POUR S'ENTRAÎNER À LA MAISON

Exercice 1 – Rendre un programme modulaire

Allez dans le répertoire `exo1`, vous y trouverez un fichier `simulation.c`.

Question 1

Transformez le fichier `simulation.c` en plusieurs fichiers : `heun-euler.h`, `heun-euler.c` et garder seulement dans le fichier `simulation.c` la fonction `main`.

Solution:
TODO

APPROFONDISSEMENT

Exercice 1 – Masquer des fonctions de la libc – version 1

Dans cet exercice, nous allons masquer des définitions de fonctions la bibliothèque standard. Attention toutes les fonctions de la bibliothèque standard peuvent être aussi définies comme des macros à paramètres. Il convient donc de prendre quelques précautions d'usage. Nous travaillons dans le répertoire `exo4` pour cet exercice.

Question 1 – Préliminaires

Dans un fichier `plus.c`, écrivez une macro à paramètre nommée `my_plus(a,b)` qui s'évalue en `a+b`. Essayer ensuite d'y définir une fonction `int my_plus(int a, int b)`. Que se passe-t-il à la compilation ?

Solution:

```c
#define my_plus(a,b) (a+b)

int my_plus (int a, int b) {
  return a + b;
}
```

.

Explication(s):

- ✿ Il y a une erreur à la compilation
- ✿ Une macro à paramètre doit forcément être suivie d'une parenthèse ouvrante pour être évaluée par le pré-processeur. Il est donc possible de l'inhiber en utilisant la déclaration suivante :
  `int (my_plus)(int a, int b)`.

Question 2

Définissez dans un fichier `my_malloc.c` une fonction `void* malloc(size_t)` qui affiche la valeur du paramètre de type `size_t` et retourne la valeur `NULL`.

Solution:

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

void* malloc(size_t t)
{
  printf("%zu\n", t);
  return NULL;
}
```

Explication(s):

- ✿ Il faut penser à mettre les parenthèses autour du nom de la fonction parce que l'implémentation est libre de définir tous les identificateurs de la bibliothèque standard sous forme de *function-like macros* qui ne sont pas masquables.

Question 3
Définissez, dans un fichier `my_main.c`, une fonction `main` qui fait un appel à la fonction `malloc`.

Solution:

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

int main(void)
{
  int* toto = malloc(10);
  return 0;
}
```

Question 4
Transformez les deux fichiers `my_main.c` et `my_malloc.c` en fichiers objets (`my_main.o` et `my_malloc.o`).

Solution:

```
% make my_main.o my_malloc.o
gcc     -c -o my_main.o my_main.c
gcc     -c -o my_malloc.o my_malloc.c
```

Question 5
Créez un programme `my_main` en liant uniquement `my_main.o`.

Solution:

```
% make my_main
gcc -Wall my_main.o -o my_main
```

Question 6
Créez un programme `my_main2` en liant `my_main.o` et `my_malloc.o`.

Solution:

```
% gcc -o my_main2 my_main.o my_malloc.o
```

Question 7
Comparez les deux exécutions de ces deux programmes, comment les expliquez-vous ?

Solution:

```
% ./my_main
% ./my_main2
10
```

Explication(s):

- ✿ Dans un cas rien ne s'affiche mais de la mémoire est allouée, c'est le `malloc()` de la libc qui est utilisé. Dans l'autre cas on voit s'afficher un entier, c'est notre `malloc()` qui a été utilisé.

Exercice 2 – Masquer des fonctions de la libc – version 2
Nous travaillons dans le répertoire `exo5` pour cet exercice.

Dans l'exercice précedent, nous avons vu comment, lors de l'édition des liens d'un programme, le masquage des fonctions de la bibliothèque standard C était possible en donnant une définition propre des fonctions. Cependant, la technique utilisée ne peut s'appliquer qu'aux programmes que nous liions nous-mêmes.

Une autre technique permet de masquer les fonctions utilisées par un programme lors du chargement de bibliothèques dynamiques, sans avoir à modifier le programme.

Dans cet exercice, une modification de la commande `date` va être réalisée pour lui faire indiquer une autre date que la date courante, par exemple, le 1er janvier 2030, ce qui donnerait

```
% date
mar jan  1 00:00:00 CET 2030
% date -I
2030-01-01
```

Pour réaliser cette modification, il est nécessaire de

— trouver la fonction utilisée par la commande `date` retournant la date du système
— écrire notre propre version de cette fonction retournant la date choisie
— créer une bibliothèque dynamique contenant cette fonction
— pré-charger cette bibliothèque lors de l'exécution de la commande `date` afin que notre fonction remplace celle du système.

Question 1 – Inspection

Plusieurs outils de mise au point sont accessibles sous linux pour étudier les programmes :

— `strace` : intercepte et enregistre les appels système lancés par un processus
— `ltrace` : intercepte et enregistre les appels à des bibliothèques dynamiques qui sont appelées par un processus
— `objdump` : affiche des informations provenant de fichiers objets

Utilisez ces fonctions pour découvrir quelle fonction est utilisée par la commande `date` pour récupérer la date du système. Pensez à utiliser la commande `grep` sur la sortie de ces commandes (le mot-clé `time` semble être une bonne piste)

Solution:

— La commande strace est inutile dans ce cas

```
% strace date 2>&1 | grep -i time
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
```

— La commande ltrace fournie plus d'informations

```
% ltrace date 2>&1 | grep -i time
clock_gettime(0, 0x7fff66114e00, 0, 0x55ad5648a400) = 0
localtime_r(0x7fff66114d30, 0x7fff66114d40, 0x55ad554e2c4d, 13) = 0x7fff66114d40
strftime("␣lundi", 1024, "␣%A", 0x7fff66114d40)  = 6
strftime("␣janvier", 1024, "␣%B", 0x7fff66114d40) = 8
```

— La commande `objdump` permet de connaître la liste des fonctions de la libc qui sont potentiellement utilisées mais pas la fonction précisément appelée

```
% objdump -T /bin/date | grep -i time
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 localtime
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 localtime_r
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.17  clock_gettime
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 gmtime_r
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 gettimeofday
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 time
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 settimeofday
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.17  clock_settime
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 mktime
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 timegm
0000000000000000      DF *UND* 0000000000000000  GLIBC_2.2.5 strftime
```

Explication(s):

✿ La principale astuce est que les commandes `strace` et `ltrace` affichent leur résultat sur la sortie d'erreur standard. L'utilisation d'un pipe nécessite une redirection de flux.

Question 2

Selon les versions installées, la commande `date` peut utiliser la fonction `clock_gettime`, `time` ou `gettimeofday`. En fonction du résultat à la question précédente, écrivez une version de la fonction dans un fichier `mydate.c`. Pour information, la date du 1er janvier 2030 est associée au nombre 1893452400.

Solution:

```
#include <time.h>

int clock_gettime (clockid_t clk_id, struct timespec *tp)
{
  tp->tv_sec = 1893452400; /* 01/01/2030 */
  tp->tv_nsec = 0;
  return 0;
}
```

Explication(s):

- ✿ Voire la page man de la fonction `clock_gettime`

Question 3

Générez une bibliothèque dynamique à partir de votre fonction précédente. Pour cela, il suffit d'ajouter l'option `-shared` au compilateur `gcc` et de générer un fichier avec l'extension `.so`

Solution:

```
% gcc -Wall -shared -o mydate.so mydate.c
```

Question 4 – Préchargement de bibliothèques dynamiques

Cherchez sur la page de manuel de `ld.so` le nom d'une variable d'environnement qui peut servir à forcer la commande `date` à utiliser votre bibliothèque. Utilisez-la pour lancer la commande `date`.

Solution:

```
% export LD_PRELOAD=./mydate.so
% date
mar jan  1 00:00:00 CET 2030
```

ou, mieux :

```
% LD_PRELOAD=./mydate.so date
mar jan  1 00:00:00 CET 2030
```

Une question que peuvent se poser les étudiants curieux est « comment appeler la fonction d'origine depuis celle qui la masque ? » On ne peut bien sûr pas appeler simplement `clock_gettime()` puisque pour le compilateur c'est le nom de la fonction qu'on écrit.
Ce qu'on peut faire, c'est utiliser `dlsym()` pour récupérer un pointeur vers l'ancienne fonction de même nom. Le `dlsym()` de la GNU libc possède une option pour chercher le symbole dans les bibliothèques suivantes dans l'ordre de recherche. (Sans cette option il aurait fallu faire un `dlopen()` sur la `libc.so` et indiquer à `dlsym` où chercher.)
Voici comment ajouter 24h à la date courante :

```
#define _GNU_SOURCE  /* Pour avoir RTLD_NEXT. */
#include <time.h>
#include <dlfcn.h>

int
clock_gettime (clockid_t clk_id, struct timespec *tp)
{
  static int (*orig_clock_gettime) (clockid_t, struct timespec *) = 0;

  if (!orig_clock_gettime)
    orig_clock_gettime = dlsym (RTLD_NEXT, "clock_gettime");

  orig_clock_gettime (clk_id, tp);
  tp->tv_sec += 24 * 60 * 60;
  return 0;
}
```

Il faut penser à compiler avec la `libdl.so` (absolument en fin de ligne de compilation) :

```
% gcc -Wall -shared -o mycg2.so  mycg2.c -ldl
% date; LD_PRELOAD=./mycg2.so date
lundi 22 janvier 2024, 12:31:42 (UTC+0100)
mardi 23 janvier 2024, 12:31:42 (UTC+0100)
```