UNIVERSITÉ DE ROUEN	ANNÉE 2005-2006
FACULTÉ DES SCIENCES	SYSTÈME D'EXPLOITATION
LICENCE INFORMATIQUE 3ème ANNÉE	FICHE DE TD N° 1

Autour du C

On se propose de réaliser un programme permettant d'afficher l'heure locale. On propose, dans un premier temps, le code C suivant :

```
/* fichier main.c */

#include <time.h>
#include <stdio.h>

void affiche_heure() {
  time_t heure;
  struct tm *h;

  heure=time(NULL);
  h=localtime(&heure);

  if (h!=NULL)
    printf("%ih %imin %is\n", h->tm_hour, h->tm_min, h->tm_sec);
}

int main() {
  printf("Il est ");affiche_heure();
  exit(0);
}
```

Exercice 1 `time.h` est un fichier d'en-tête déclarant des fonctions de la bibliothèque C standard. En particulier, la structure `struct tm` y est décrite. Elle contient, entre autres, les champs :

```
int tm_sec
int tm_min
int tm_hour
```

De plus on y trouve les fonctions

`time_t time(time_t *);` et
`struct tm *localtime(const time_t *);`

La fonction `time_t time(time_t *);` renvoie une valeur numérique codant l'heure et la date. La fonction `struct tm *localtime(const time_t *);` renvoie un pointeur sur une structure `struct tm` plus facile à manier.

1) Ce programme vous semble-t-il correct ? Répond-il à notre objectif ?

On rappelle que `gcc` est un ensemble de compilateurs GNU (`gcc` pour «GNU Compiler Collection»). En ce qui nous concerne, nos fichiers porteront l'extension «.c» ; ils seront donc compilés en tant que sources C.

Les arguments qui nous intéresserons pour `gcc` sont (vous pourrez regarder en TP les pages du manuel afin d'avoir un aperçu de toutes les options disponibles) :

```
gcc [-c] [-ansi] [-Wall] [-Idir...]
    [-Ldir...] [-llibrary] [-shared]
    [-o outfile] infile...
```

2) Quelles sont les arguments que vous reconnaissez ?

3) Donnez la ligne de commande permettant de compiler le programme avec `gcc`.

4) Quelles sont les commandes permettant de dissocier l'étape de compilation de celle de l'édition des liens ?

La fonction `main` peut recevoir deux arguments :

```
int main(int argc, char argv[][]);
```

`argv` est un tableau de chaînes de caractères.

`argv[0]` contient le nom du programme. `argv[1]`...`argv[n]` contiennent les n arguments transmis au programme.

`argc` contiendra alors le nombre d'éléments du tableau `argv` (c'est donc le nombre d'arguments plus 1).

Ainsi la commande

```
prg arg1 arg2 arg3
```

lancera l'exécutable `prg` et `argc` vaudra 4, `argv[0]` vaudra «`prg\0`», `argv[1]` vaudra «`arg1\0`», `argv[2]` vaudra «`arg2\0`» et `argv[3]` vaudra «`arg3\0`». Enfin `argv[4]` vaudra `NULL`.

5) Modifiez le programme de façon à ce qu'on puisse indiquer en paramètre si on veut l'heure locale ou l'heure GMT (il faudra alors utiliser la fonction `gmtime` au lieu de `localtime`).

Exercice 2 On préfère généralement réaliser des programmes modulaires. On isole alors chaque partie du programme dans des fichiers séparés.

On peut, par exemple, réaliser le programme suivant :

```
/* fichier f.c */
int f(int n) { return n*n;}

/* fichier f.h */
int f(int n);

/* fichier main.c */
#include "f.h"
int main() {
  int n;
  scanf("%d", &n);printf("Le carre de %i est %i\n", n,f(n));
}
```

1) Modifiez le programme précédent de façon à isoler la fonction `affiche_heure` dans un fichier séparé `heure.c`.

2) Donnez les lignes de commande permettant de compiler ce programme avec `gcc`.

3) On suppose maintenant que la fonction `affiche_heure` se trouve déjà dans une bibliothèque «`heure`». Donnez les lignes de commande permettant de compiler le nouveau programme avec `gcc`.

Le préprocesseur permet la macrogénération de code. La directive permettant de définir les macros est `#define`. Elle permet de définir des macros constantes comme `#define true 1` ou des macros paramétrées comme `#define f(n) n*n`. Une expression comme `a==true` sera alors remplacée par `a==1` par le préprocesseur et une instruction comme `a=f(3)` sera alors remplacée par `a=3*3`.

Exercice 3

1) Donnez une macro permettant de calculer le minimum de deux entiers.

2) Réalisez une macro `heure(a)` affichant le contenu de la structure `struct tm` sur la sortie standard sous la forme `tm_hour`h `tm_min`min `tm_sec`s.

3) Réécrivez le fichier `heure.h` de façon à utiliser la macro précédente.

Il peut être usant d'avoir à réécrire plusieurs lignes à chaque fois qu'on veut compiler un programme. Afin d'y remédier on peut écrire un script contenant ces lignes. L'inconvénient est qu'alors on recompile tout les fichiers à chaque fois même si on n'a pas modifié tous les fichiers. `make` est un utilitaire permettant, entre autre, d'automatiser et d'optimiser les étapes de compilation et d'édition des liens. On indique à `make` les commandes à exécuter à travers un fichier de construction (par défaut ces fichiers se nomment `makefile` ou `Makefile`). Un fichier de construction contient des règles indiquant à `make` comment construire les différents fichiers du programme. On peut de plus indiquer à `make` quelles sont les dépendances de ce fichier (par exemple, un fichier exécutable peut dépendre d'un fichier ".c" et d'un fichier ".h"). Une règle est de la forme :

```
cible: dependance ...
[tab]commands
[tab]...
```

`[tab]` indique qu'il s'agit d'une tabulation (attention c'est obligatoire).

On peut avoir, par exemple, la règle :

```
prg: prg.c prg .h
    gcc -o prg prg.c
```

Dans ce cas, si le fichier `prg` n'existe pas ou s'il est plus ancien que `prg.c` ou que `prg .h` (ce qui veut dire que ces derniers fichiers ont étés modifiés depuis la dernière compilation) alors la commande `gcc -o prg prg.c` est exécutée.

Exercice 4

1) Donnez un fichier `Makefile` permettant de simplifier la compilation du programme de l'exercice 2.1.

Afin de simplifier la modification du fichier ainsi que sa lecture, on peut définir des variables. Une variable est définie à travers la ligne :

```
variable=valeur...
```

On récupère la valeur de la variable par l'expression `$(variable)`.

2) Réécrivez le fichier de construction en mettant un maximum de valeurs dans des variables.

Il existe également des variable prédéfinies. Par exemple, `$@` correspond au nom de la cible de la règle en cours. `$*` correspond au nom de la cible sans son extension. `$<` correspond au nom du premier fichier des dépendances. `$^` contient le nom de toutes les dépendances.

3) Reprenez votre fichier `Makefile` et utilisez les variables `$@`, `$<`, `$^` et `$*` à chaque fois que c'est possible.

On peut également créer des règles dont la cible n'est pas un nom de fichier. Cela permet d'avoir des noms standards pour certaines actions. Ainsi `all` permet généralement de compiler le programme et `clean` permet de supprimer tous les fichiers créés lors de la construction du programme.

4) Créez les règles `all` et `clean`.

Pour créer une bibliothèque dynamique à partir d'un fichier «.o», on doit le lier avec l'option `-shared` :

```
gcc -shared -o libnom.so bibli.o
```

On crée une archive pouvant être liée de façon statique par :

```
ar -r libnom.a bibli.o
ranlib libnom.a
```

Une bibliothèque dynamique commence par «lib» et a pour suffixe «.so», ainsi la bibliothèque mathématique «m» a pour fichier `libm.so`. De même, une bibliothèque statique commence par «lib» et a pour suffixe «.a».

5) Modifiez le Makefile afin de générer une bibliothèque incluant les fonctions du fichier `heure.h`. On devra pouvoir choisir entre une bibliothèque statique et une bibliothèque dynamique.