TD/TP de Programmation – Planche 1
Dates pour les calendriers julien et grégorien

On souhaite programmer quelques fonctions utilitaires sur les dates via la structure suivante :

```
typedef struct Date {
  int day, month, year;
  bool cal_is_julian;
} Date;
```

Les champs entiers `day`, `month`, `year` sont respectivement les numéro du jour, du mois et de l'année de la date repésentée par la structure. Le champ booléen `cal_is_julian` indique si le calendrier de référence est julien ou grégorien.

Dans le calendrier julien, toutes les années multiples de 4 sont bissextiles et ont donc un 29 Février. Dans le calendrier grégorien, un correctif est apporté : les années multiples de 100 ne sont bissextiles que si elles sont multiples de 400. Par exemple, les années 1600, 1700, 1800, 1900 et 2000 sont toutes bissextiles dans le calendrier julien, alors que seules 1600 et 2000 le sont dans le calendrier grégorien.

Par ailleurs, l'Epoch Unix est le Jeudi 01 Janvier 1970 dans le calendrier grégorien, ce qui correspond au Jeudi 19 Décembre 1969 dans le calendrier julien.

1 Fonctions de base

Écrire la fonction `Date_Make()` qui fabrique et retourne une date à partir de ses arguments. (On ne vérifie pas la validité des valeurs numériques).

```
Date Date_Make (int day, int month, int year, bool cal_is_julian);
```

```
int main (void) {
  Date d= Date_Make (25, 12, 2000, false);
  printf ("%d/%d/%d\n", d.day, d.month, d.year);
  printf ("%s␣calendar\n", d.cal_is_julian ? "julian" : "gregorian");
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
25/12/2000
gregorian calendar
```

En déduire `Date_Epoch()` qui retourne l'Epoch Unix pour un calendrier donné :

```
Date Date_Epoch (bool cal_is_julian);
```

```
int main (void) {
  Date d= Date_Epoch (true);
  printf ("date␣is␣%d␣%d␣%d\n", d.day, d.month, d.year);
  printf ("calendar␣is␣%s\n", d.cal_is_julian ? "julian" : "gregorian");
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
date is 19 12 1969
calendar is julian
```

Écrire le prédicat `Date_HasSameCalAs()` qui teste si la date `d1` utilise le même calendrier que la date `d2`.

```
bool Date_HasSameCal (Date d1, Date d2);
```

```
int main (void) {
  Date d1= Date_Epoch (true), d2= Date_Epoch (false);
  bool has_same_cal= Date_HasSameCal (d1, d2);
  printf ("dates␣have␣%s␣calendars\n", has_same_cal ? "same" : "different");
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
dates have different calendars
```

Écrire la fonction `Date_CompareForSameCal()` qui compare deux dates `d1` et `d2` supposées appartenir à un même calendrier. On rejette par assertion les dates de calendriers différents. Comme il est d'usage pour les fonctions de comparaison, on retourne un entier `comp` se comparant à zéro de la même manière que `d1` se compare à `d2`, c'est-à-dire :

`comp < 0` ssi `d1 < d2`, `comp == 0` ssi `d1 == d2`, et `comp > 0` ssi `d1 > d2`.

```
int Date_CompareForSameCal (Date d1, Date d2);
```

```
int main (void) {
  Date d1= Date_Make (25, 12, 2000, false);
  Date d2= Date_Make (10, 12, 2000, false);
  int comp= Date_CompareForSameCal (d1, d2);
  char comp_symbol= (comp < 0) ? '<' : (comp > 0) ? '>' : '=';
  printf ("d1␣%c␣d2\n", comp_symbol);
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
d1 > d2
```

2 Fonctions de conversion entre dates et chaînes

La fonction `sscanf(text, format, ...)` lit des informations depuis la chaîne `text` de la même façon que `scanf(format, ...)` les saisirait depuis l'entrée standard. Comme cette dernière, elle retourne le nombre de variables qui ont pu être assignées à une valeur avant un éventuel échec de conversion.

Écrire la fonction `Date_FromString()` qui retourne une date correspondant à une description donnée au format `"jour/mois/année:cal"`. Si la description n'est pas constituée de 3 entiers suivis d'un spécificateur valide (`'j'` ou `'g'`), on retourne la date bidon `Date_Dummy()` correspondant à `"0/0/0:g"`. En revanche, on ne se préoccupe pas de la validité des valeurs numériques obtenues.

```
Date Date_Dummy (void);
Date Date_FromString (char const description[]);
```

```
int main (void) {
  Date d= Date_FromString ("3/2/2018:g");
  printf ("date␣is␣%d␣%d␣%d\n", d.day, d.month, d.year);
  printf ("calendar␣is␣%s\n", d.cal_is_julian ? "julian" : "gregorian");
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
date is 3 2 2018
calendar is gregorian
```

La fonction `sprintf(text, format, ...)` remplit la chaîne `text` avec ce qu'afficherait la fonction `printf(format, ...)` sur la sortie standard avec les mêmes arguments. Comme cette dernière, elle retourne le nombre de caractères écrits (zéro terminal exclu). La capacité du tableau `text` doit être strictement supérieure à ce nombre, afin de pouvoir contenir la chaîne, zéro terminal inclus. La variante `snprintf(text, capacity, format, ...)` permet de préciser la capacité de `text` et de se prémunir d'un débordement de tableau.

Écrire la fonction `Date_ToString()` qui pour une date `d`, remplit la chaîne `text` avec sa représentation au format `"jour/mois/année:cal"` où le spécificateur `cal` vaut `'j'` ou `'g'` selon que le calendrier est julien ou grégorien.

```
void Date_ToString (Date d, char text[], int capacity);
```

```
int main (void) {
  Date date= Date_Epoch (true);
  int const CAPACITY= 100;
  char text [CAPACITY];
  Date_ToString (date, text, CAPACITY);
  printf ("date␣is␣%s\n", text);
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
date is 19/12/1969:j
```

Remarque : En C, utiliser l'identificateur d'un tableau sans crochets donne l'adresse de ce tableau (qui est aussi l'adresse de sa première case). En particulier, retourner l'identificateur d'une variable tableau dans une fonction ne retourne pas une copie de la variable, mais retourne son adresse. Si ce tableau est local à la fonction, il n'existe pas en dehors l'exécution de la fonction, et il est alors interdit de retourner son adresse. Si la variable est déclarée `static` à l'intérieur de la fonction, le tableau devient une variable globale (dont le nom est caché) disposant d'une durée de vie égale à celle du programme. Il est alors possible de retourner son adresse :

```
char * // address of a char
Date_ToStaticString (Date d) {
# define STATIC_TEXT_CAPACITY 100
  static char static_text [STATIC_TEXT_CAPACITY];
  Date_ToString (d, static_text, STATIC_TEXT_CAPACITY);
  return static_text; // evaluates to & static_text[0], the address of static_text[]
}
```

La variable globale étant unique, c'est le même tableau qui utilisé à chaque appel de la fonction, et donc chaque nouvel appel écrase le contenu écrit par l'appel précédent :

```
int main (void) {
  Date d1= Date_Epoch (true), d2= Date_Epoch (false);
  printf ("d1␣is␣%s\n", Date_ToStaticString (d1));
  printf ("d2␣is␣%s\n", Date_ToStaticString (d2));
  return 0;
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date
$ ./tp-date
d1 is 19/12/1969:j
d2 is 1/1/1970:g
```

On peut obtenir les arguments fournis sur la ligne de commande avec une variante de la fonction `main()` :

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

L'entier `argc` est le compteur d'arguments et `argv[]` est le vecteur d'arguments. Le nom de la commande lui-même compte pour un argument et se trouve dans `argv[0]`. S'il y a 4 arguments derrière le nom de la commandes, il se trouvent dans `argv[1], ..., argv[4]`, et alors `argc` vaut 5.

```
int main (int argc, char * argv []) {
  printf ("argc : %d\n", argc);
  for (int k= 0; k < argc; k++) {
    printf ("argv[%d] : <%s>\n", k, argv [k]);
  }
}
```

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date titi tata toto 'bla bla bla'
argc : 5
argv[0] : <./tp-date>
argv[1] : <titi>
argv[2] : <tata>
argv[3] : <toto>
argv[4] : <bla bla bla>
```

Écrire un programme qui prend sur sa ligne de commande exactement un argument derrière le nom de la commande (ou échoue avec un message d'erreur dans le cas contraire). Cet argument est une représentation de date au format `"jour/mois/année:cal"`. Le programme convertit cette chaîne en date, fabrique l'Epoch pour le même calendrier, et affiche le résultat de la comparaison de ces 2 dates.

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date
usage: ./tp-date day/month/year:cal

$ ./tp-date 31/12/1969:g
1/1/1970:g > 31/12/1969:g

$ ./tp-date 31/12/1969:j
19/12/1969:j < 31/12/1969:j

$ ./tp-date toto
1/1/1970:g > 0/0/0:g
```

3 Contrôle des plages numériques

Écrire les prédicats `JulianYear_IsLeap()` et `GregorianYear_IsLeap()` qui testent respectivement si une année est bissextile pour le calendrier julien et le calendrier grégorien.

```
bool JulianYear_IsLeap (int year);
bool GregorianYear_IsLeap (int year);
```

En déduire le prédicat `Year_IsLeap()` qui teste si une année est bissextile pour un calendrier donné, ainsi que la fonction `Year_Length()` qui retourne la longueur d'une année en jours.

```
bool Year_IsLeap (int year, bool cal_is_julian);
int  Year_Length (int year, bool cal_is_julian);
```

Écrire également les wrappers pour ces deux fonctions, paramétrées avec une date :

```
bool Date_YearIsLeap (Date date);
int  Date_YearLength (Date date);
```

Une technique populaire pour identifier les mois longs et les courts, consiste à les identifier aux bosses et aux creux du dos de nos deux poings. Sur le poing gauche, il y a quatre bosses séparés par trois creux, et donc pour les sept premiers mois, les mois impairs sont long et les mois pairs sont courts. Pour les cinq derniers mois, on utilise le poing droit, soit trois bosses séparés par deux creux, et donc à partir du huitième mois, les mois pairs sont long et les mois impairs sont courts.

En utilisant cette technique, écrire le prédicat `Month_Has31Days()` qui teste si un mois est long. En déduire la fonction `Month_Length()` retournant la longueur d'un mois en jours, pour un calendrier donné :

```
bool Month_Has31Days (int month);
int  Month_Length (int month, int year, bool cal_is_julian);
```

En déduire le wrapper `Date_MonthLength()` qui retourne la longueur du mois courant d'une date donnée, ainsi que le prédicat `Date_IsValid()` qui teste si une date est valide, c'est-à-dire, si les entiers du triplet sont compris dans les bonnes plages de valeurs. Par exemple la date `29/2/1800:g` n'est pas une date valide, car 1800 n'est pas bissextile dans le calendrier grégorien :

```
int  Date_MonthLength (Date d);
bool Date_IsValid     (Date d);
```

Écrire un programme de test pour ces fonctions. Le programme prend sur sa ligne de commande une date au format `"jour/mois/année:cal"`. Si la date est valide, le programme indique la longueur de l'année et du mois courants :

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

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date
usage: ./tp-date day/month/year:cal

$ ./tp-date 5/2/1800:j
date: 5/2/1800:j, year length: 366, month length: 29

$ ./tp-date 5/2/1800:g
date: 5/2/1800:g, year length: 365, month length: 28

$ ./tp-date 29/2/1800:g
date: 29/2/1800:g, invalid date

$ ./tp-date toto
date: 0/0/0:g, invalid date
```

4 Lendemain et veille

Écrire les fonctions suivantes : `Date_Add1Day()` qui retourne le lendemain d'une date ; `Date_Sub1Day()` qui retourne la veille d'une date.

```
Date Date_Add1Day (Date d);
Date Date_Sub1Day (Date d);
```

Écrire un programme de test pour ces fonctions. Le programme prend sur sa ligne de commande une date au format `"jour/mois/année:cal"`. Si la date est valide, le programme affiche la veille et le lendemain.

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

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date 1/1/1800:g
date: 1/1/1800:g, prev: 31/12/1799:g, next: 2/1/1800:g
```

5 Compte des jours écoulés et restants dans l'année

Écrire les fonctions suivantes : `Date_YearElapsedDays()` qui compte le nombre de jours écoulés depuis le début de l'année pour une date donnée ; `Date_YearRemainingDays()` qui compte le nombre de jours jusqu'à la fin de l'année pour une date donnée.

```
int Date_YearElapsedDays   (Date d); // d.day excluded
int Date_YearRemainingDays (Date d); // d.day included
```

Écrire un programme de test pour ces fonctions. Le programme prend sur sa ligne de commande une date au format `"jour/mois/année:cal"`. Si la date est valide, le programme affiche le nombre de jours écoulés et restants dans l'année.

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

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date 1/1/2000:g
date: 1/1/2000:g, elapsed: 0, remaining: 366

$ ./tp-date 31/12/2000:g
date: 31/12/2000:g, elapsed: 365, remaining: 1

$ ./tp-date 5/3/2000:g
date: 5/3/2000:g, elapsed: 64, remaining: 302
```

6 Tampon

On appellera *tampon* (ou *stamp*) d'une date le nombre de jours qui la sépare de l'Epoch Unix (ce nombre est négatif si la date est antérieure à l'Epoch, il est nul si la date est l'Epoch, et il est postif sinon).

Écrire la fonction `Date_DaysBetweenForSameCal()` qui compte le nombre de jours entre deux dates `older` et `newer` d'un même calendrier tel que `older <= newer` (ce nombre est toujours positif ou nul, et il est nul ssi `older == newer`). Puis en déduire `Date_Stamp()` qui retourne le tampon d'une date donnée.

```
long Date_DaysBetweenForSameCal (Date older, Date newer);
long Date_Stamp (Date d);
```

Écrire un programme de test pour ces fonctions. Le programme prend sur sa ligne de commande une date au format `"jour/mois/année:cal"`. Si la date est valide, le programme affiche son tampon.

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

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date 31/1/1971:g
date: 31/1/1971:g, stamp: +395

$ ./tp-date 15/12/1969:g
date: 15/12/1969:g, stamp: -17

$ ./tp-date 15/12/1969:j
date: 15/12/1969:j, stamp: -4
```

7 Différence et comparaison

Écrire la fonction `Date_Diff()` qui retourne le nombre de jours entre deux dates `d1` et `d2`, qui n'ont pas nécéssairement le même calendrier. Le résultat est négatif si `d1 < d2`, nul si `d1 == d2`, et positif si `d1 > d2`. Écrire `Date_Compare()` qui compare deux dates à la manière de `Date_Compare_ForSameCal()`, mais sans l'hypothèse d'un calendrier commun.

```
long Date_Diff    (Date d1, Date d2);
int  Date_Compare (Date d1, Date d2);
```

Écrire un programme de test pour ces fonctions. Le programme prend sur sa ligne de commande deux dates au format `"jour/mois/année:cal"`. Si les deux dates sont valides, le programme affiche la comparaison des deux dates, ainsi que leur différence en nombre de jours.

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

```
$ clang -W -Wall -std=c99 -pedantic tp-date.c -o tp-date

$ ./tp-date 23/1/2013:g 25/12/2012:g
23/1/2013:g (epoch+15728) > 25/12/2012:g (epoch+15699)
diff: +29

$ ./tp-date 1/1/2013:g 1/1/2013:j
1/1/2013:g (epoch+15706) < 1/1/2013:j (epoch+15719)
diff: -13
```