Licence informatique 2ème année
Université de La Rochelle

Programmation C - Contrôle continu  *

Christophe Demko <christophe.demko@univ-lr.fr>

2017-2018_3

Le travail est à faire en **binôme ou en monôme** sur une durée de **3 heures**. La consultation des ressources sur moodle et internet est autorisée (**à l'exclusion des moyens de communications avec des tiers**). Vous avez aussi accès à vos comptes personnels.

L'archive (**format *zip***) est à déposer sur moodle sur les comptes de chaque partie du binôme avant l'heure limite (**aucune dérogation ne sera accordée**). Vous la nommerez en utilisant **le nom et le prénom des deux parties** du binôme (le cas échéant).

Lisez l'énoncé en entier avant de commencer à programmer. Ne restez pas bloqué sur un exercice qui vous semble difficile. **Il peut être plus facile d'accumuler des points sur certains exercices** (un barème indicatif se trouve en fin d'énoncé).

Vous devez absolument utiliser le système de compilation fourni par l'utilitaire *cmake*. Après configuration de votre projet à l'aide de *cmake*, il devra être possible de l'installer relativement à la variable `CMAKE_INSTALL_PREFIX`. Vous devez également respecter les noms des types et des fonctions demandées. **Le non-respect de ces instructions entrainera une non-correction de votre travail**.

Vous prendrez soin d'indenter correctement votre code et d'utiliser les conventions de programmation vues en cours. Tachez d'utilisez une programmation *DRY* [1]

1. https://fr.wikipedia.org/wiki/Ne_vous_répétez_pas

Sujet

La décomposition d'un entier naturel en produit de facteurs premiers [2] consiste à l'écrire comme le produit unique de nombres premiers.

Par exemple, la décomposition du nombre 504 est donnée par $2\times2\times2\times3\times3\times7 = 2^3\times3^2\times7^1$. Les cas de la décomposition des nombres 0 et 1 feront l'objet d'un traitement particulier.

Dans cet ensemble d'exercice, la *décomposition en produit de facteurs premiers* sera représentée par un tableau de puissances.

Exemple : la décomposition du nombre 504 sera représentée par le tableau d'entiers

`[3,2,0,1]` :

- le 3 du tableau représente la puissance du nombre premier 2 (nombre premier n°1)
- le 2 du tableau représente la puissance du nombre premier 3 (nombre premier n°2)
- le 0 du tableau représente la puissance du nombre premier 5 (nombre premier n°3)
- le 1 du tableau représente la puissance du nombre premier 7 (nombre premier n°4)

Exemple : la décomposition du nombre 911525 est donnée par $5 \times 5 \times 19 \times 19 \times 101 = 5^2 \times 19^2 \times 101^1$ et sera représentée par le tableau d'entier

`[0,0,2,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1]` :

- le 0 représente la puissance du nombre premier 2
- le 0 représente la puissance du nombre premier 3
- le 2 représente la puissance du nombre premier 5
- le 0 représente la puissance du nombre premier 7
- le 0 représente la puissance du nombre premier 11
- le 0 représente la puissance du nombre premier 13
- le 0 représente la puissance du nombre premier 17
- le 2 représente la puissance du nombre premier 19
- les dix-sept 0 suivants représentent les puissances des nombres premiers 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97
- le 1 représente la puissance du nombre premier 101

En langage `C`, ce tableau sera représenté par une structure à 2 champs (voir figure 1) :

- `count` le nombre de nombre premiers utilisés (ici 26)
- `powers` un tableau alloué **dynamiquement** (un pointeur donc) (contenant ici les 26 puissances des premiers nombres premiers).

> Dans cet ensemble d'exercices, la librairie devra se nommer *prime-factors* (*libprime-factors.so* sous linux ou son équivalent *dll* sous windows) et l'entête *prime-factors.h*. Le code suivant dans un fichier `CMakeLists.txt` permettra d'utiliser la librairie installée :
>
> ```
> find_package(PrimeFactors REQUIRED)
> ```
>
> à l'aide des variables
>
> - `PRIME_FACTORS_INCLUDE_DIRS` (dossiers où retrouver le fichier d'entête du projet

2. https://fr.wikipedia.org/wiki/Décomposition_en_produit_de_facteurs_premiers

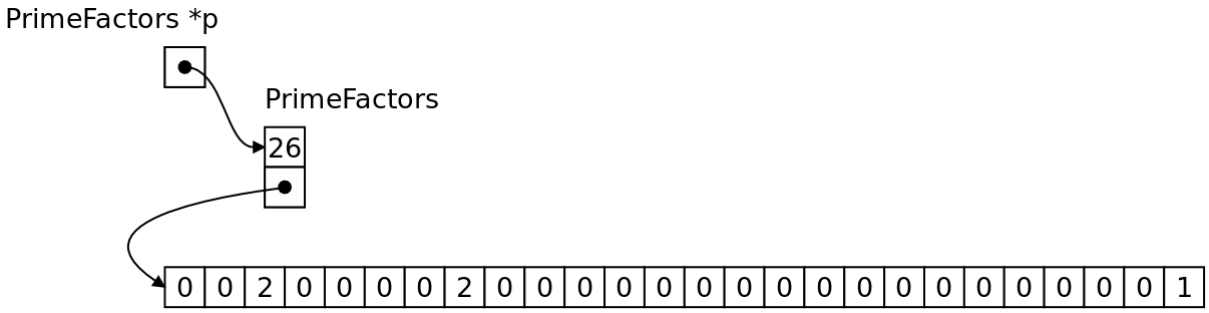

FIGURE 1 – Représentation du nombre 911525

`PrimeFactors`)
- `PRIME_FACTORS_LIB_DIRS` (dossiers où retrouver les librairies du projet `PrimeFactors`)
- `PRIME_FACTORS_LIBRARIES` (les librairies du projet `PrimeFactors`)

Il est conseillé, dans un premier temps, de ne pas vérifier le retour des fonctions d'allocations et de se limiter aux 25 premiers nombres premiers pour effectuer la décomposition.

Le fichier *prime-factors.c* fourni définit des fonctions statiques qui peuvent vous être utiles pour l'écriture de vos fonctions. Vous n'êtes pas obligés de les utiliser.

Exercice 1 *(Définition du type `PrimeFactors`)*

Définissez le type `PrimeFactors`. Vous prendrez soin de faire en sorte que la structure ne soit pas connu à l'extérieur de la librairie (voir l'exercice des *Champs protégés* du Cahier 1).

Exercice 2 *(Fonctions d'initialisation et de finalisation des facteurs premiers)*

1. Écrivez deux fonctions permettant d'initialiser (resp. de finaliser) la librairie gérant la *décomposition en produit de facteurs premiers*.

   ```
   extern bool prime_factors_init(void);
   extern bool prime_factors_finish(void);
   ```

   À son premier appel, la fonction d'initialisation doit initialiser deux variables **statiques** au sein du fichier `prime-factors.c` :
   - `primes` qui est initialement un tableau **dynamique** des 25 premiers nombres premiers.
   - `primes_size` qui est le nombre de nombres premiers du tableau `primes` (25 initialement)

   À son dernier appel, la fonction de finalisation doit
   - libérer le tableau **dynamique** `primes` ;
   - mettre le tableau **dynamique** `primes` à `NULL` ;
   - mettre la variable `primes_size` à `0`.

Les fonctions doivent partager un compteur comme cela a été le cas pour les fonctions `fraction_init` et `fraction_finish` du cahier 1.

2. Écrivez un programme de test `test-init-finish` permettant de valider les 2 fonctions écrites.

Exercice 3 *(Fonctions de création et de destruction)*

1. Écrivez une fonction permettant de créer par allocation une nouvelle *décomposition en produit de facteurs premiers*.

```
extern PrimeFactors *prime_factors_create_full(
  unsigned long number
);
```

où :

— `number` représente le nombre à décomposer.

Vous utiliserez l'algorithme de factorisation naïf[3].

Par exemple, l'appel `prime_factors_create_full(911525)` retournera le pointeur `p` de la figure 1.

— si `number` est égal à `0`, la fonction retourne `NULL`

— si `number` est égal à `1`, la fonction retourne une décomposition vide (une structure dont `count` vaut 0 et dont `powers` vaut `NULL`)

Vous aurez peut-être besoin d'étendre le tableau `primes` préalablement rempli dans la fonction `prime_factors_init` si il ne contient pas assez de nombres premiers (cela doit être le cas si l'on demande la décomposition du nombre `911525` ; il est effectivement un produit du $26^{\text{ème}}$ nombre premier : `101`)

2. Écrivez une fonction permettant de créer par allocation une nouvelle *décomposition en produit de facteurs premiers* vide.

```
extern PrimeFactors *prime_factors_create_default(void);
```

L'appel

```
prime_factors_create_default();
```

doit être équivalent à

```
prime_factors_create_full(1);
```

3. Écrivez une fonction permettant de libérer toute la mémoire utilisée par une décomposition en produit de facteurs premiers.

```
extern void prime_factors_destroy(PrimeFactors *factors);
```

4. Écrivez un programme de test `test-create-destroy` permettant de valider les fonctions écrites.

Conseil : la fonction `prime_factors_create_full(unsigned long number)` est certainement la plus difficile à écrire. Il peut être judicieux de créer les factorisations *à la main* dans les programmes de test qui ne teste pas cette fonction ; i.e. pour tous les autres exercices.

Exercice 4 *(Fonctions d'accès aux données)*

1. Écrivez une fonction permettant de récupérer le nombre initial d'une *décomposition en produit de facteurs premiers*.

   ```
   extern unsigned long prime_factors_get_number(
     const PrimeFactors *factors
   );
   ```

   où :
   — `factors` est une *décomposition en produit de facteurs premiers*.
   La fonction retourne le nombre dont `factors` est la décomposition.
2. Écrivez un programme de test `test-get` permettant de valider la fonction écrite.

Exercice 5 *(Opérations)*

1. Écrivez une fonction permettant de calculer le plus grand commun diviseur de deux *décompositions en produit de facteurs premiers*.

   ```
   extern PrimeFactors *prime_factors_gcd(
     PrimeFactors *result,
     const PrimeFactors *a,
     const PrimeFactors *b
   );
   ```

   où
   — `result` (déjà créé) contiendra le plus grand commun diviseur de `a` et `b` ;
   — `a` est une *décomposition en produit de facteurs premiers* ;
   — `b` est une *décomposition en produit de facteurs premiers*.
   L'algorithme consiste à calculer les minima des puissances de `a` et `b`.
   Exemple :
   Le plus grand commun diviseur de la décomposition de 32 (`[5]`) et de la décomposition de 56 (`[3,0,0,1]`) sera 8 (`[3]`)
2. Écrivez une fonction permettant de calculer le plus petit commun multiple de deux *décompositions en produit de facteurs premiers*.

   ```
   extern PrimeFactors *prime_factors_lcm(
     PrimeFactors *result,
     const PrimeFactors *a,
   ```

3. https://fr.wikipedia.org/wiki/Décomposition_en_produit_de_facteurs_premiers#Algorithmes_de_factorisation

```
  const PrimeFactors *b
);
```

où :

— `result` (déjà créé) contiendra le plus petit commun multiple de `a` et `b` ;
— `a` est une *décomposition en produit de facteurs premiers* ;
— `b` est une *décomposition en produit de facteurs premiers*.

L'algorithme consiste à calculer les maxima des puissances de `a` et `b`.

Exemple :

Le plus petit commun multiple de la décomposition de 32 (`[5]`) et de la décomposition de 56 (`[3,0,0,1]`) sera 224 (`[5,0,0,1]`)

3. Écrivez un programme de test `test-gcd-lcm` permettant de valider les fonctions écrites.

Exercice 6 *(Conversion en chaîne de caractères)*

1. Écrivez une fonction permettant de convertir une *décomposition en produit de facteurs premiers* en chaîne de caractères.

```
extern const char *prime_factors_to_string(
  const PrimeFactors *factors
);
```

où :

— `factors` est une *décomposition en produit de facteurs premiers*

La conversion en chaîne de caractères de la décomposition de 911525 devrait être :

```
5^2x19^2x101
```

Aucune fuite de mémoire ne doit avoir lieu si un appel tel que :

```
printf("%s\n", prime_factors_to_string(factors));
```

est réalisé.

2. Écrivez un programme de test `test-to-string` permettant de valider la fonction écrite.

Exercice 7 *(Écriture et lecture sur fichier)*

1. Écrivez deux fonctions permettant d'écrire et de lire une *décomposition en produit de facteurs premiers* sur un fichier déjà ouvert.

```
extern PrimeFactors *prime_factors_fwrite(
  const PrimeFactors * factors,
  FILE * stream
);
extern PrimeFactors *prime_factors_fread(
```

```
  PrimeFactors * factors,
  FILE * stream
);
```

où :

— `factors` est une *décomposition en produit de facteurs premiers* ;

— `stream` est un flux ouvert en écriture pour `prime_factors_fwrite` et en lecture pour `prime_factors_fread`.

— `prime_factors_fwrite` permet d'écrire une *décomposition en produit de facteurs premiers* sur un fichier. Elle renvoie la *décomposition en produit de facteurs premiers* passée en paramètre s'il n'y a pas d'erreur ou `NULL` en cas d'erreur ;

— `prime_factors_fread` permet de lire une *décomposition en produit de facteurs premiers* sur un fichier. Elle renvoie la *décomposition en produit de facteurs premiers* ou `NULL` en cas d'erreur.

2. Écrivez un programme de test `test-file` permettant de valider les 2 fonctions écrites.

Barème indicatif

— compilation sans erreurs : **1 point**
— programmation *DRY* : **1 point**
— optimisation de la vitesse d'exécution : **1 point**
— exercice 1 : **2 points**
— exercice 2 : **2 points**
— exercice 3 : **4 points**
— exercice 4 : **1 point**
— exercice 5 : **3 points**
— exercice 6 : **3 points**
— exercice 7 : **2 points**

Historique des modifications

2017-2018_1 *Vendredi 18 mai 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Version initiale.

2017-2018_2 *Vendredi 18 mai 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Renommage *fonction de terminaison* en *fonction de finalisation* (Exercice 2 (Fonctions d'initialisation et de finalisation des facteurs premiers)) ;
— Correction de l'exemple de l'Exercice 5 (Opérations) (56 est représenté par `[3,0,0,1]` et 224 par `[5,0,0,1]`).

2017-2018_3 *Dimanche 25 novembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation des nouvelles normes de programmation.