

```

1  /*
2  -----
3  Nom du fichier   : main.cpp
4  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
5  Date creation   : 10.03.2022
6  Description      : Fichier principal du programme permettant de tester le bon
7                     fonctionnement de la classe Matrix.
8  Compilateur     : Mingw-w64 g++ 8.1.0
9  -----
10 */
11 #include <iostream>
12 #include "matrix.hpp"
13
14 void test();
15
16 int main(int argc, char *argv[]) {
17     // Décommentez la ligne ci-dessous pour run les tests
18     // test();
19
20     // Partie où l'on test que les arguments passés en lignes de commandes sont
21     // cohérents
22     if(argc != 6){
23         throw std::runtime_error("Pas le bon nombre d'argument.");
24     }
25
26     // Si l'un des arguments ne peut pas être transformé en int, le programme crash
27     int nbRow1 = std::stoi(argv[1]);
28     int nbCol1 = std::stoi(argv[2]);
29     int nbRow2 = std::stoi(argv[3]);
30     int nbCol2 = std::stoi(argv[4]);
31     int modulus = std::stoi(argv[5]);
32
33     if(nbRow1 < 0 || nbRow2 < 0 || nbCol1 < 0 || nbCol2 < 0){
34         throw std::runtime_error("Les nombres de lignes et de colonnes doivent etre "
35                                   "des entiers positifs ou nuls.");
36     }
37     if(modulus < 1){
38         throw std::runtime_error("Le modulo doit etre un entier plus grand que 0.");
39     }
40     // Nous avons cette instruction afin d'éviter que le random soit toujours le
41     // même dans la création des valeurs des matrices.
42     srand ((unsigned int) time(NULL));
43     Matrix matrix1((unsigned int) nbRow1, (unsigned int) nbCol1,
44                   (unsigned int) modulus);
45     Matrix matrix2((unsigned int) nbRow2, (unsigned int) nbCol2,
46                   (unsigned int) modulus);
47     std::cout << "The modulus is " << modulus << std::endl;
48
49     std::cout << "One" << std::endl;
50     std::cout << matrix1 << std::endl;
51
52     std::cout << "Two" << std::endl;
53     std::cout << matrix2 << std::endl;
54
55     std::cout << "One + Two" << std::endl;
56     std::cout << matrix1.addStaticNew(matrix2) << std::endl;
57
58     std::cout << "One - Two" << std::endl;
59     std::cout << matrix1.subStaticNew(matrix2) << std::endl;
60
61     std::cout << "One * Two" << std::endl;
62     std::cout << matrix1.multStaticNew(matrix2) << std::endl;
63
64     test();
65     return 0;
66 }
67
68
69

```

```

70  /**
71  * Fonction permettant de tester tous les fonctionnalités d'une Matrix
72  * Il est important de noter que certains test sont commentés car il font crash le
73  * programme. N'hésitez pas à les décommenter si vous souhaitez les tester.
74  */
75  void test(){
76      std::cout << "Creation d'une matrice vide" << std::endl;
77      Matrix emptyMatrix(0, 0 ,5);
78      std::cout << "Matrice vide :" << std::endl << emptyMatrix << std::endl;
79
80      Matrix noRowMatrix(0, 2 ,5);
81      std::cout << "Matrice sans ligne :" << std::endl << noRowMatrix << std::endl;
82
83      Matrix noColMatrix(0, 0 ,5);
84      std::cout << "Matrice sans colonne :" << std::endl << noColMatrix << std::endl;
85
86      std::cout << "Creation d'une matrice avec un modulo valant 0" << std::endl;
87
88      try{
89          Matrix zeroModuloMat(2, 3 ,0);
90      } catch(const std::runtime_error& e){
91          std::cout << e.what() << std::endl << std::endl;
92      }
93
94      std::cout << "Creation de la matrice \"une\" de taille 2x3 avec 5 comme "
95          "modulo" << std::endl;
96      Matrix one(2, 3 ,5);
97      std::cout << "Matrice une :" << std::endl << one << std::endl;
98
99      std::cout << "Creation de la matrice \"deux\" de taille 3x3 avec 5 comme "
100          "modulo" << std::endl;
101      Matrix two(3, 3 ,5);
102      std::cout << "Matrice deux :" << std::endl << two << std::endl;
103
104      std::cout << "Creation de la matrice \"trois\" de taille 2x3 avec 7 comme "
105          "modulo" << std::endl;
106      Matrix three(2, 3 ,7);
107      std::cout << "Matrice trois :" << std::endl << three << std::endl;
108
109      std::cout << "Creation de la matrice \"quatre\" copiant la matrice \"une\"<<
110          std::endl;
111      Matrix four(one);
112      std::cout << "Matrice quatre :" << std::endl << four << std::endl;
113
114      std::cout << "Addition des matrices \"une\" et \"deux\"<< std::endl;
115      std::cout << "Addition retournant par valeurs" << std::endl <<
116      one.addStaticNew(two) << std::endl;
117
118      Matrix* oneAddDynamic = one.addDynamicNew(two);
119      std::cout << "Addition retournant par pointeur" << std::endl <<
120      *oneAddDynamic << std::endl;
121      std::cout << "Addition modifiant la premiere matrice" << std::endl;
122      one.addItself(two);
123      std::cout << one << std::endl << std::endl;
124
125      std::cout << "Re-initialisation de la matrice \"une\" de taille 2x3 avec 5 comme "
126          "modulo" << std::endl;
127      one = Matrix(2, 3 ,5);
128      std::cout << "Matrice une :" << std::endl << one << std::endl;
129      std::cout << "Matrice deux :" << std::endl << two << std::endl;
130
131      std::cout << "Soustraction des matrices \"une\" et \"deux\" (une - deux)" <<
132      std::endl;
133      std::cout << "Soustraction retournant par valeurs" << std::endl <<
134      one.subStaticNew(two) << std::endl;
135
136
137
138

```

```

139 Matrix* oneSubDynamic = one.subDynamicNew(two);
140 std::cout << "Soustraction retournant par pointeur" << std::endl <<
141     *oneSubDynamic << std::endl;
142 std::cout << "Soustraction modifiant la premiere matrice" << std::endl;
143 one.subItself(two);
144 std::cout << one << std::endl << std::endl;
145
146 std::cout << "Re-initialisation de la matrice \"une\" de taille 2x3 avec 5 comme "
147     "modulo" << std::endl;
148 one = Matrix(2, 3, 5);
149 std::cout << "Matrice une :" << std::endl << one << std::endl;
150 std::cout << "Matrice deux :" << std::endl << two << std::endl;
151
152 std::cout << "Multiplication des matrices \"une\" et \"deux\" (une * deux)" <<
153     std::endl;
154 std::cout << "Multiplication retournant par valeurs" << std::endl <<
155     one.multStaticNew(two) << std::endl;
156
157 Matrix* oneMultDynamic = one.multDynamicNew(two);
158 std::cout << "Multiplication retournant par pointeur" << std::endl <<
159     *oneMultDynamic << std::endl;
160 std::cout << "Multiplication modifiant la premiere matrice" << std::endl;
161 one.multItself(two);
162 std::cout << one << std::endl << std::endl;
163
164 std::cout << "Re-initialisation de la matrice \"une\" de taille 2x3 avec 5 comme "
165     "modulo" << std::endl;
166 one = Matrix(2, 3, 5);
167 std::cout << "Matrice une :" << std::endl << one << std::endl;
168
169 std::cout << "Operation sur des matrices ayant differents modulus" << std::endl;
170 std::cout << "\"une\" + \"trois\"" << std::endl;
171 try{
172     one.addStaticNew(three);
173 } catch(const std::invalid_argument& e){
174     std::cout << e.what() << std::endl;
175 }
176 std::cout << "\"une\" - \"trois\"" << std::endl;
177 try{
178     one.subDynamicNew(three);
179 } catch(const std::invalid_argument& e){
180     std::cout << e.what() << std::endl;
181 }
182 std::cout << "\"une\" * \"trois\"" << std::endl;
183 try{
184     one.multItself(three);
185 } catch(const std::invalid_argument& e){
186     std::cout << e.what() << std::endl;
187 }
188 delete oneSubDynamic;
189 delete oneAddDynamic;
190 delete oneMultDynamic;
191 }

```

```

208  /*
209  -----
210  Nom du fichier   : matrix.hpp
211  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
212  Date creation    : 03.03.2022
213  Description      : Classe permettant de modéliser des matrices de tailles diverses.
214                    Il est possible de leur appliquer plusieurs opérations:
215                    (L'addition, la soustraction ainsi que la multiplication).
216                    Chaque opération peut se faire de 3 manières. Soit en modifiant
217                    la matrice sur laquelle l'opération est invoquée. Soit en
218                    renvoyant par valeur une nouvelle matrice allouée statiquement.
219                    Soit en retournant un pointeur sur une nouvelle matrice allouée
220                    dynamiquement.
221                    Il est également possible d'afficher une matrice dans la console
222                    via l'opérateur d'écriture de flux "<<".
223  Compileur       : Mingw-w64 g++ 8.1.0
224  -----
225  */
226
227  #ifndef LAB01_MATRIX_HPP
228  #define LAB01_MATRIX_HPP
229
230  #include <iostream>
231  #include <cstdlib>
232  #include <ctime>
233  #include "operation.hpp"
234
235  class Matrix;
236
237  std::ostream &operator<<(std::ostream &os, const Matrix &matrix);
238
239  class Matrix {
240  public:
241      /**
242       * Constructeur de la classe matrix.
243       * Une exception (runtime_error) est lancée si le modulo vaut 0
244       * @param row taille des lignes
245       * @param col taille des colonnes
246       * @param mod modules
247       */
248      Matrix(size_t row, size_t col, unsigned mod);
249
250      /**
251       * Constructeur de copie
252       * @param other matrice à copier
253       */
254      Matrix(const Matrix &other);
255
256      /**
257       * Destructeur de la classe Matrix
258       */
259      ~Matrix();
260
261      /**
262       * Surcharge de l'opérateur d'affectation
263       * @param other Matrice à copier
264       * @return la matrice après l'affectation
265       */
266      Matrix &operator=(const Matrix &other);
267
268      /**
269       * Surcharge de l'opérateur de flux afin de pouvoir afficher une matrice
270       * @param os opérateur de flux
271       * @param matrix matrice à afficher
272       * @return opérateur de flux utilisé
273       */
274      friend std::ostream &operator<<(std::ostream &os, const Matrix &matrix);
275
276

```

```

277     /**
278     * Addition entre deux matrices, modifie la matrice depuis laquelle la fonction
279     * est appelée.
280     * Cette méthode peut throw une exception invalid_argument.
281     * @param matrix matrice à additionner
282     * @return une référence sur la matrice modifiée après l'addition
283     */
284     Matrix &addItself(const Matrix &matrix);
285
286     /**
287     * Addition entre deux matrices, crée une nouvelle matrice résultant du calcul.
288     * Cette méthode peut throw une exception invalid_argument.
289     * @param matrix matrice à additionner
290     * @return la matrice modifiée après l'addition
291     */
292     Matrix addStaticNew(const Matrix &matrix) const;
293
294     /**
295     * Addition entre deux matrices, crée une nouvelle matrice résultant du calcul.
296     * Cette méthode peut throw une exception invalid_argument.
297     * @param matrix matrice à additionner
298     * @return un pointeur sur la matrice modifiée après l'addition
299     */
300     Matrix *addDynamicNew(const Matrix &matrix) const;
301
302     /**
303     * Soustraction entre deux matrices, modifie la matrice depuis laquelle la
304     * fonction est appelée.
305     * Cette méthode peut throw une exception invalid_argument.
306     * @param matrix matrice à soustraire
307     * @return une référence sur la matrice modifiée après la soustraction
308     */
309     Matrix &subItself(const Matrix &matrix);
310
311     /**
312     * Soustraction entre deux matrices, crée une nouvelle matrice résultant du
313     * calcul.
314     * Cette méthode peut throw une exception invalid_argument.
315     * @param matrix matrice à soustraire
316     * @return la matrice modifiée après la soustraction
317     */
318     Matrix subStaticNew(const Matrix &matrix) const;
319
320     /**
321     * Soustraction entre deux matrices, crée une nouvelle matrice résultant du
322     * calcul.
323     * Cette méthode peut throw une exception invalid_argument.
324     * @param matrix matrice à soustraire
325     * @return un pointeur sur la matrice modifiée après la soustraction
326     */
327     Matrix *subDynamicNew(const Matrix &matrix) const;
328
329     /**
330     * Multiplication entre deux matrices, modifie la matrice depuis laquelle la
331     * fonction est appelée.
332     * Cette méthode peut throw une exception invalid_argument.
333     * @param matrix matrice à soustraire
334     * @return une référence sur la matrice modifiée après la soustraction
335     */
336     Matrix &multItself(const Matrix &matrix);
337
338     /**
339     * Multiplication entre deux matrices, crée une nouvelle matrice résultant du
340     * calcul.
341     * Cette méthode peut throw une exception invalid_argument.
342     * @param matrix matrice à multiplier
343     * @return la matrice modifiée après la multiplication
344     */
345     Matrix multStaticNew(const Matrix &matrix) const;

```

```

346
347 /**
348  * Multiplication entre deux matrices, crée une nouvelle matrice résultant du
349  * calcul.
350  * Cette méthode peut throw une exception invalid_argument.
351  * @param matrix matrice à multiplier
352  * @return un pointeur sur la matrice modifiée après la multiplication
353  */
354 Matrix *multDynamicNew(const Matrix &matrix) const;
355
356 private:
357 /**
358  * Génère une matrice possédant des valeurs aléatoires
359  */
360 void generateMatrix();
361
362
363 /**
364  * Retourne un élément de la matrice, renvoi 0 si les index sont en dehors de
365  * la matrice
366  * @param row le numéro de la ligne
367  * @param col le numéro de la colonne
368  * @return l'élément se trouvant aux index passés en paramètre
369  */
370 unsigned getVal(size_t row, size_t col) const;
371
372 /**
373  * Change la taille de la matrice
374  * @param row nouveau nombre de ligne
375  * @param col nouveau nombre de colonne
376  */
377 void changeSizeValues(size_t row, size_t col);
378
379 /**
380  * Désalloue et détruit le tableau de tableau possédant les valeurs de la matrice
381  */
382 void deleteValues();
383
384 /**
385  * Applique une opération passée en paramètre à tous les éléments de deux
386  * matrices.
387  * @param matrix
388  * @param op
389  * @return
390  */
391 Matrix *applyOperator(const Matrix &matrix, Operation *op);
392
393 /**
394  * Remplace les valeurs de la matrice actuelle par les valeurs de la matrice
395  * passée en paramètre.
396  * @param other
397  */
398 void replaceValues(const Matrix &other);
399
400 size_t row, col;
401 unsigned int mod;
402 unsigned **values;
403 };
404
405 #endif //LAB01_MATRIX_HPP
406
407
408
409
410
411
412
413
414

```

```

415  /*
416  -----
417  Nom du fichier   : matrix.cpp
418  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
419  Date creation   : 03.03.2022
420  Description      : Fichier contenant l'implémentation de la classe Matrix.
421  Compilateur     : Mingw-w64 g++ 8.1.0
422  -----
423  */
424
425  #include "matrix.hpp"
426  #include "add.hpp"
427  #include "subtract.hpp"
428  #include "multiply.hpp"
429  #include "utils.hpp"
430
431  Matrix::Matrix(size_t row, size_t col, unsigned mod) : row(row), col(col), mod(mod) {
432      if(mod == 0) {
433          throw std::runtime_error("Modulo invalide");
434      }
435      generateMatrix();
436  }
437
438  Matrix::Matrix(const Matrix &other) {
439      row = other.row;
440      col = other.col;
441      mod = other.mod;
442      values = new unsigned*[row];
443      for (size_t i = 0; i < row; ++i) {
444          values[i] = new unsigned[col];
445      }
446      replaceValues(other);
447  }
448
449  Matrix::~Matrix() {
450      deleteValues();
451  }
452
453
454
455  void Matrix::deleteValues() {
456      for (size_t i = 0; i < row; ++i) {
457          delete[] values[i];
458      }
459      delete[] values;
460  }
461
462  void Matrix::replaceValues(const Matrix &other) {
463      for (size_t i = 0; i < row; ++i) {
464          for (size_t j = 0; j < col; ++j) {
465              values[i][j] = other.getVal(i,j);
466          }
467      }
468  }
469
470  Matrix &Matrix::operator=(const Matrix &other) {
471      if(this != &other){
472
473          changeSizeValues(other.row,other.col);
474          this->row = other.row;
475          this->col = other.col;
476          this->mod = other.mod;
477          replaceValues(other);
478      }
479      return *this;
480  }
481
482
483

```

```

484     std::ostream &operator<<(std::ostream &os, const Matrix &matrix){
485         for (size_t i = 0; i < matrix.row; ++i) {
486             for (size_t j = 0; j < matrix.col; ++j) {
487                 os << matrix.getVal(i,j) << " ";
488             }
489             os << std::endl;
490         }
491         return os;
492     }
493
494     void Matrix::generateMatrix() {
495         values = new unsigned*[row];
496         for (size_t i = 0; i < row; ++i) {
497             values[i] = new unsigned[col];
498             for (size_t j = 0; j < col; ++j) {
499                 values[i][j] = Utils::randomNumber(mod);
500             }
501         }
502     }
503
504     Matrix& Matrix::addItself(const Matrix &matrix){
505         static Add* add = new Add();
506         applyOperator(matrix,add);
507         return *this;
508     }
509
510     Matrix Matrix::addStaticNew(const Matrix &matrix) const{
511         Matrix m(*this);
512         m.addItself(matrix);
513         return m;
514     }
515
516     Matrix *Matrix::addDynamicNew(const Matrix &matrix) const{
517         Matrix* m = new Matrix(*this);
518         try{
519             m->addItself(matrix);
520         } catch(const std::invalid_argument& e){
521             delete m;
522             throw;
523         }
524         return m;
525     }
526
527     Matrix& Matrix::subItself(const Matrix &matrix) {
528         static Subtract* sub = new Subtract();
529         applyOperator(matrix,sub);
530         return *this;
531     }
532
533     Matrix Matrix::subStaticNew(const Matrix &matrix) const{
534         Matrix m(*this);
535         m.subItself(matrix);
536         return m;
537     }
538
539     Matrix *Matrix::subDynamicNew(const Matrix &matrix) const{
540         Matrix* m = new Matrix(*this);
541         try{
542             m->subItself(matrix);
543         } catch(const std::invalid_argument& e){
544             delete m;
545             throw;
546         }
547         return m;
548     }
549
550     Matrix& Matrix::multItself(const Matrix &matrix){
551         static Multiply* mult = new Multiply();
552         applyOperator(matrix,mult);
553         return *this;
554     }

```



```

553 Matrix Matrix::multStaticNew(const Matrix &matrix) const{
554     Matrix m(*this);
555     m.multItself(matrix);
556     return m;
557 }
558
559 Matrix *Matrix::multDynamicNew(const Matrix &matrix) const{
560     Matrix* m = new Matrix(*this);
561     try{
562         m->multItself(matrix);
563     } catch(const std::invalid_argument& e){
564         delete m;
565         throw;
566     }
567     return m;
568 }
569
570 void Matrix::changeSizeValues(size_t row, size_t col){
571     if(this->row == row && this->col == col) {
572         return;
573     }
574     unsigned** newValues = new unsigned*[row];
575     for (size_t i = 0; i < row; ++i) {
576         newValues[i] = new unsigned[col];
577     }
578     for (size_t i = 0; i < row; ++i) {
579         for (size_t j = 0; j < col; ++j) {
580             newValues[i][j] = this->getVal(i,j);
581         }
582     }
583     deleteValues();
584     this->row = row;
585     this->col = col;
586     values = newValues;
587 }
588
589 Matrix* Matrix::applyOperator(const Matrix &matrix, Operation* op){
590     if(this->mod != matrix.mod){
591         throw std::invalid_argument("Les modules des deux matrices ne sont pas "
592                                     "egaux");
593     }
594     changeSizeValues(std::max(this->row, matrix.row),std::max(this->col,matrix.col));
595
596     for (size_t i = 0; i < this->row; ++i) {
597         for (size_t j = 0; j < this->col; ++j) {
598             values[i][j] = Utils::floorMod(op->apply(this->getVal(i,j), matrix
599             .getVal(i,j)),mod);
600         }
601     }
602     return this;
603 }
604
605 unsigned Matrix::getVal(size_t row, size_t col) const{
606     if(row >= this->row || col >= this->col){
607         return 0;
608     }
609     return values[row][col];
610 }
611
612
613
614
615
616
617
618
619
620
621

```

```

622  /*
623  -----
624  Nom du fichier   : operation.hpp
625  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
626  Date creation    : 03.03.2022
627  Description      : Fichier contenant la déclaration de la classe opération ainsi
628                    que de sa méthode "apply" qui permettra d'effectuer une opération
629                    sur des matrices. La méthode "apply" devra être redéfinie dans
630                    les sous-classes.
631  Compilateur      : Mingw-w64 g++ 8.1.0
632  -----
633  */
634
635  #ifndef LAB01_OPERATION_HPP
636  #define LAB01_OPERATION_HPP
637
638
639  class Operation {
640  public:
641      virtual long long apply(unsigned a, unsigned b) = 0;
642  };
643
644
645  #endif //LAB01_OPERATION_HPP
646
647  /*
648  -----
649  Nom du fichier   : add.hpp
650  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
651  Date creation    : 03.03.2022
652  Description      : Fichier contenant la déclaration de la classe add. Cette
653                    dernière redéfini la méthode "apply" afin de permettre de faire
654                    une addition entre deux matrices.
655  Compilateur      : Mingw-w64 g++ 8.1.0
656  -----
657  */
658
659  #ifndef LAB01_ADD_HPP
660  #define LAB01_ADD_HPP
661
662  #include "operation.hpp"
663
664
665  class Add : public Operation{
666  public:
667      long long apply(unsigned a, unsigned b);
668  };
669
670
671  #endif //LAB01_ADD_HPP
672
673  /*
674  -----
675  Nom du fichier   : add.cpp
676  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
677  Date creation    : 03.03.2022
678  Description      : Fichier contenant l'implémentation de la classe add.
679  Compilateur      : Mingw-w64 g++ 8.1.0
680  -----
681  */
682
683  #include "add.hpp"
684  long long Add::apply(unsigned a, unsigned b){
685      return (long long) a + (long long) b;
686  }
687
688
689
690

```

```

691  /*
692  -----
693  Nom du fichier   : subtract.hpp
694  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
695  Date creation    : 03.03.2022
696  Description      : Fichier contenant la déclaration de la classe subtract. Cette
697                    dernière redéfinit la méthode "apply" afin de permettre de faire
698                    une soustraction entre deux matrices.
699  Compilateur      : Mingw-w64 g++ 8.1.0
700  -----
701  */
702
703  #ifndef LAB01_SUBTRACT_HPP
704  #define LAB01_SUBTRACT_HPP
705
706  #include "operation.hpp"
707
708  class Subtract : public Operation{
709  public:
710      long long apply(unsigned a, unsigned b);
711  };
712
713
714  #endif //LAB01_SUBTRACT_HPP
715
716
717  /*
718  -----
719  Nom du fichier   : subtract.cpp
720  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
721  Date creation    : 03.03.2022
722  Description      : Fichier contenant l'implémentation de la classe subtract.
723  Compilateur      : Mingw-w64 g++ 8.1.0
724  -----
725  */
726
727  #include "subtract.hpp"
728  long long Subtract::apply(unsigned a, unsigned b){
729      return (long long) a - (long long) b;
730  }
731
732  /*
733  -----
734  Nom du fichier   : multiply.hpp
735  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
736  Date creation    : 03.03.2022
737  Description      : Fichier contenant la déclaration de la classe multiply. Cette
738                    dernière redéfinit la méthode "apply" afin de permettre de faire
739                    une multiplication composante par composante entre deux matrices.
740  Compilateur      : Mingw-w64 g++ 8.1.0
741  -----
742  */
743  #ifndef LAB01_MULTIPLY_HPP
744  #define LAB01_MULTIPLY_HPP
745
746  #include "operation.hpp"
747
748  class Multiply : public Operation{
749  public:
750      long long apply(unsigned a, unsigned b);
751  };
752
753
754
755  #endif //LAB01_MULTIPLY_HPP
756
757
758
759

```

```

760  /*
761  -----
762  Nom du fichier   : multiply.cpp
763  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
764  Date creation   : 03.03.2022
765  Description      : Fichier contenant l'implémentation de la classe multiply.
766  Compilateur     : Mingw-w64 g++ 8.1.0
767  -----
768  */
769
770  #include "multiply.hpp"
771  long long Multiply::apply(unsigned a, unsigned b){
772      return (long long) a * (long long) b;
773  }
774
775  /*
776  -----
777  Nom du fichier   : utils.hpp
778  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
779  Date creation   : 09.03.2022
780  Description      : Fichier contenant la déclaration de fonctions utils. Ce dernier
781                   défini des fonctions telles que floorMod (inspirée du
782                   java) et randomNumber permettant de générer un entier.
783  Compilateur     : Mingw-w64 g++ 8.1.0
784  -----
785  */
786
787  #ifndef LAB01_UTILS_HPP
788  #define LAB01_UTILS_HPP
789
790  #include <cstdlib>
791  class Utils {
792  public:
793      /**
794       * Génère un entier aléatoire modulo la valeur passée en paramètre
795       * @param mod est le modulo à appliquer sur la valeur aléatoire.
796       * @return une valeur entre [0, mod - 1]
797       */
798      unsigned static randomNumber(unsigned mod);
799
800      /**
801       * Effectue un modulo mathématique entre 2 valeurs passées en paramètres. Le
802       * résultat ne peut pas être négatif.
803       * @param a
804       * @param b
805       * @return le modulo des deux paramètres
806       */
807      unsigned static floorMod(long long a, unsigned b);
808  };
809  #endif //LAB01_UTILS_HPP
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828

```

```
829  /*
830  -----
831  Nom du fichier   : utils.cpp
832  Auteur(s)       : Alexandre Jaquier, Jonathan Friedli
833  Date creation   : 09.03.2022
834  Description      : Fichier contenant l'implémentation de fonctions utilitaires.
835  Compilateur     : Mingw-w64 g++ 8.1.0
836  -----
837  */
838
839  #include "utils.hpp"
840
841  unsigned Utils::randomNumber(unsigned mod) {
842      return (unsigned)rand() % mod;
843  }
844
845
846  unsigned Utils::floorMod(long long a, unsigned b) {
847      a %= (long long)b;
848      return unsigned (a < 0 ? a + (long long)b : a);
849  }
850
```