AP11 - Enoncé du Contrôle Continu d'Algorithmique
Lundi 17 octobre 2005

Document autorisé : le polycopié distribué en cours
Veuillez préciser votre groupe à côté de vos nom et prénom

Soit un nombre noté *nombre* (*nombre* $\geq$ 10) de *nbChiffres* chiffres. Pour déterminer si *nombre* est un nombre de *Keith*, on procéde de la manière suivante :
On remplit un tableau de *nbChiffres* éléments avec les chiffres de *nombre*. On calcule la somme des éléments de ce tableau. Cette valeur est ensuite « injectée » en fin de tableau en décalant les autres éléments et en perdant la première valeur (correspondant au chiffre de poids fort du nombre à tester à la première itération).
On réitère ce processus, les sommes calculées formant une suite monotone croissante. Si une somme calculée est égale au nombre à tester, ce nombre est de *Keith*. Sinon elle deviendra strictement supérieure au nombre à tester et on s'arrêtera.
Pour vous aider à comprendre cette méthode, voici son déroulement pour *nombre* = 197 ; elle nécessite un tableau de trois naturels :

étape 1	1	9	7	somme = 17
étape 2	9	7	17	somme = 33
étape 3	7	17	33	somme = 57
étape 4	17	33	57	somme = 107
étape 5	33	57	107	somme = 197

197 est un nombre de *Keith*.
Les nombres de *Keith* à deux chiffres : 14, 19, 28, 47, 61, 75
Les nombres de *Keith* à trois chiffres : 197, 742
Les nombres de *Keith* à quatre chiffres : 1104, 1537, 2208, 2580, 3684, 4788, 7385, 7647, 7909
Voici le déroulement de la méthode pour *nombre* = 1537 ; elle nécessite un tableau de quatre naturels :

étape 1	1	5	3	7	somme = 16
étape 2	5	3	7	16	somme = 31
étape 3	3	7	16	31	somme = 57
étape 4	7	16	31	57	somme = 111
étape 5	16	31	57	111	somme = 215
étape 6	31	57	111	215	somme = 414
étape 7	57	111	215	414	somme = 797
étape 8	111	215	414	797	somme = 1537

Par contre 117 n'est pas un nombre de *Keith*

étape 1	1	1	7	somme = 9
étape 2	1	7	9	somme = 17
étape 3	7	9	17	somme = 33
étape 4	9	17	33	somme = 59
étape 5	17	33	59	somme = 109
étape 6	33	59	109	somme = 201

car les valeurs successives de la somme ont « dépassé » 117 sans qu'il y ait eu égalité.

Vous devez définir, **dans notre langage algorithmique**, la fonction *estNombreDeKeith* qui prend un paramètre *nombre* de type Naturel et qui renvoie un booléen égal à VRAI si *nombre* est un nombre de *Keith*, à FAUX sinon.
Dans cette fonction, il faudra définir un tableau pour stocker les différentes valeurs manipulées. Mais comme en langage algorithmique, un tableau est de longueur fixée à l'avance, on définira une constante *NB_MAX_CHIFFRES* égale à neuf. Le nombre à tester devra être strictement inférieur à un milliard.

constante NB_MAX_CHIFFRES = 9 // Le nombre à tester doit être inférieur à 1 000 000 000

tabNombres : **tableau** [1..NB_MAX_CHIFFRES] **de** Naturel

De ce fait, il faudra définir dans la fonction *estNombreDeKeith* une variable *nbChiffres* de type Naturel qui contiendra le nombre de chiffres de la valeur à tester (c'est à vous de le calculer). Vous n'utiliserez dans le tableau *tabNombres* que les cases comprises entre les indices 1 et *nbChiffres* inclus.

Adoptez une démarche descendante ; ce problème s'y prête très bien. Vous pouvez, mais ce n'est qu'un conseil, définir :
- une fonction pour calculer le nombre de chiffres de la valeur à tester
- une procédure pour initialiser le tableau *tabNombres* entre les indices 1 et *nbChiffres*
- une fonction pour calculer la somme des éléments « utiles » du tableau *tabNombres*
- une procédure pour modifier le tableau *tabNombres* entre les indices 1 et *nbChiffres*

Définissez également la fonction *principale* en prenant soin de vérifier que la valeur à tester saisie par « l'utilisateur » est bien strictement inférieure à un milliard et supérieure ou égale à dix.

AP11 - Corrigé du Contrôle Continu d'Algorithmique

Soit un nombre noté *nombre* (*nombre* $\geq$ 10) de *nbChiffres* chiffres. Pour déterminer si *nombre* est un nombre de *Keith*, on procéde de la manière suivante :
On remplit un tableau de *nbChiffres* éléments avec les chiffres de *nombre*. On calcule la somme des éléments de ce tableau. Cette valeur est ensuite « injectée » en fin de tableau en décalant les autres éléments et en perdant la première valeur (correspondant au chiffre de poids fort du nombre à tester à la première itération).
On réitère ce processus, les sommes calculées formant une suite monotone croissante. Si une somme calculée est égale au nombre à tester, ce nombre est de *Keith*. Sinon elle deviendra strictement supérieure au nombre à tester et on s'arrêtera.
Pour vous aider à comprendre cette méthode, voici son déroulement pour *nombre* = 197 ; elle nécessite un tableau de trois naturels :

étape 1	1	9	7	somme = 17
étape 2	9	7	17	somme = 33
étape 3	7	17	33	somme = 57
étape 4	17	33	57	somme = 107
étape 5	33	57	107	somme = 197

197 est un nombre de *Keith*.
Les nombres de *Keith* à deux chiffres : 14, 19, 28, 47, 61, 75
Les nombres de *Keith* à trois chiffres : 197, 742
Les nombres de *Keith* à quatre chiffres : 1104, 1537, 2208, 2580, 3684, 4788, 7385, 7647, 7909
Voici le déroulement de la méthode pour *nombre* = 1537 ; elle nécessite un tableau de quatre naturels :

étape 1	1	5	3	7	somme = 16
étape 2	5	3	7	16	somme = 31
étape 3	3	7	16	31	somme = 57
étape 4	7	16	31	57	somme = 111
étape 5	16	31	57	111	somme = 215
étape 6	31	57	111	215	somme = 414
étape 7	57	111	215	414	somme = 797
étape 8	111	215	414	797	somme = 1537

Par contre 117 n'est pas un nombre de *Keith*

étape 1	1	1	7	somme = 9
étape 2	1	7	9	somme = 17
étape 3	7	9	17	somme = 33
étape 4	9	17	33	somme = 59
étape 5	17	33	59	somme = 109
étape 6	33	59	109	somme = 201

car les valeurs successives de la somme ont « dépassé » 117 sans qu'il y ait eu égalité.

Vous devez définir, **dans notre langage algorithmique**, la fonction *estNombreDeKeith* qui prend un paramètre *nombre* de type Naturel et qui renvoie un booléen égal à VRAI si *nombre* est un nombre de *Keith*, à FAUX sinon.

```
constante NB_MAX_CHIFFRES = 9 // Le nombre à tester doit être inférieur à 1 000 000 000

fonction estNombreDeKeith ( nombre : Naturel ) : Booléen

    nbChiffres = nombreChiffres ( nombre ) : Naturel
    sommeNombres : Naturel
    tabNombres : tableau [ 1..NB_MAX_CHIFFRES ] de Naturel

    initialiserTabNombres ( nombre , nbChiffres, tabNombres ) // version directe
    // version inverse : initialiserTabNombres ( nombre , tabNombres )

    faire
        sommeNombres ← sommeTabNombres ( nbChiffres , tabNombres )
        modifierTabNombres ( nbChiffres , sommeNombres , tabNombres )

    tant que sommeNombres < nombre

    retourner ( sommeNombres = nombre )

ffct estNombreDeKeith
```

Définition de la fonction *nombreChiffres*

```
fonction nombreChiffres ( nombre : Naturel ) : Naturel

    nb = 0 : Naturel // valeur à retourner

    tant que nombre ≠ 0 faire
        nb ← nb + 1
        nombre ← nombre div 10
    ftq

    retourner nb

ffct nombreChiffres
```

Définition de la procédure *initialiserTabNombres*

```
procédure initialiserTabNombres ( donnée nombre, nbChiffres : Naturel
                                  résultat tNombres = tableau [ 1..NB_MAX_CHIFFRES ] de Naturel )

    i = nbChiffres : Naturel // indice le tableau de nombres

    tant que nombre ≠ 0 faire // ou tant que i > 0 faire
        tNombres [ i ] ← nombre mod 10
        nombre ← nombre div 10
        i ← i - 1
    ftq


fproc initialiserTabNombres
```

Définition de la fonction *sommeTabNombres*

```
fonction sommeTabNombres ( nbChiffres : Naturel,
                           tNombres = tableau [ 1..NB_MAX_CHIFFRES ] de Naturel ) : Naturel

    somme = 0 : Naturel // valeur à retourner

    pour i = 1 : Naturel à nbChiffres
        somme ← somme + tNombres [ i ]
    fpour i

    retourner somme

ffct sommeTabNombres
```

Définition de la procédure *modifierTabNombres*

```
procédure modifierTabNombres ( donnée nbChiffres, somme : Naturel
                               donnée-résultat tNombres = tableau [ 1..NB_MAX_CHIFFRES ] de Naturel )

    pour i = 1 : Naturel à nbChiffres - 1
        tNombres [ i ] ← tNombres [ i + 1 ]
    fpour i
    tNombres [ nbChiffres ] ← somme

fproc modifierTabNombres
```

Définition de la fonction *principale*

```
fonction principale ( ) : CodeRetour

    nombreAtester : Naturel

    faire
        afficher ( "saisir un nombre ≥ 10 et < 1 000 000 000" )
        saisir ( nombreAtester )
    tant que nombreAtester < 10 ou nombreAtester ≥ 1 000 000 000

    afficher ( nombreAtester )
    si estNombreDeKeith ( nombreAtester ) alors
        afficher ( " est " )
    sinon
        afficher ( " n'est pas " )
    fsi
    afficher ( "un nombre de Keith." )
    allerAlaLigne ( )

    retourner OK

ffct principale
```

AP11 - Enoncé du Contrôle Continu d'Algorithmique
Vendredi 27 octobre 2006

Durée : 1 heure
Document autorisé : le polycopié distribué en cours
Veuillez préciser votre groupe à côté de vos nom et prénom

Calculer la racine carrée d'un réel strictement positif noté *a* revient à résoudre l'équation $x^2 = a$. On en déduit alors :

$$x^2 = a \Rightarrow 2x^2 = x^2 + a \Rightarrow x = \frac{x^2+a}{2x} \Rightarrow x = \frac{1}{2}(x + \frac{a}{x})$$

Si l'on indice la variable *x* selon qu'elle se trouve dans le membre droit ou le membre gauche de la dernière équation ci-dessus, on obtient la formule de récurrence suivante :

$$x_{i+1} = \frac{1}{2}(x_i + \frac{a}{x_i})$$

En posant $x_0 = a$, on retrouve l'algorithme attribué à *Héron d'Alexandrie* (1er siècle ap. J.-C.) mais les babyloniens connaissaient déjà cette méthode. En fait, on peut faire démarrer l'algorithme avec n'importe quelle valeur strictement positive. Mais le nombre de calculs pour obtenir la racine carrée d'un nombre avec une précision convenable est dépendant de ce nombre.
On vous **fournit** donc une fonction *valeurInitiale* qui prend un paramètre *a* de type Réel (le nombre dont on cherche la racine carrée) et qui renvoie un réel. Celui-ci vous servira de valeur initiale x_0. Voici la signature de cette fonction :

fonction valeurInitiale (a : Réel) : Réel

Encore une fois, **vous n'avez pas à définir cette fonction**. Vous trouverez sa définition dans le corrigé. Sachez simplement qu'avec cette valeur initiale, seulement **trois calculs successifs** permettront d'obtenir la racine carrée d'un nombre avec une précision convenable, et cela, quelque soit ce nombre.
On peut donc définir une constante *NB_CALCULS_MAX* que vous utiliserez dans les fonctions que vous avez à définir.

constante NB_CALCULS_MAX = 3

En utilisant cette méthode, définissez en **langage algorithmique** :

- la fonction **itérative** *racineCarréeItérative* qui prend deux paramètres *a* et *valInit* de type Réel et qui renvoie un réel : la racine carrée du nombre *a*. Voici la signature de cette fonction :

  fonction racineCarréeItérative (a, valInit : Réel) : Réel

- la fonction **récursive** *racineCarréeRécursive* qui prend deux paramètres *a* et *valInit* de type Réel,un paramètre *nbCalculs* de type NatureINonNul et qui renvoie un réel : la racine carrée du nombre *a*. Voici la signature de cette fonction :

  fonction racineCarréeRécursive (a, valInit : Réel, nbCalculs : NatureINonNul) : Réel

Définissez également la fonction *principale* qui permettra à l'utilisateur de saisir un nombre réel strictement positif. Elle vérifiera que la valeur saisie est correcte et appellera la fonction *valeurInitiale*. Finalement, elle affichera d'une manière lisible les valeurs retournées par les fonctions *racineCarréeItérative* et *racineCarréeRécursive*.

AP11 - Corrigé du Contrôle Continu d'Algorithmique

constante NB_CALCULS_MAX = 3

Définition de la fonction *principale* :

fonction principale ( ) : CodeRetour

    nombre : Réel

    afficher ("Saisir un réel > 0 : ")
    saisir (nombre)

    si nombre $\leq$ 0 **alors**
        afficher ("Erreur de saisie")
        allerAlaLigne ( )
        retourner KO
    sinon

        valInit = valeurInitiale (nombre) : Réel

        afficher ("Racine carrée de ", nombre, " est égale à ", racineCarréeItérative (nombre, valInit))
        allerAlaLigne ( )
        afficher ("Racine carrée de ", nombre, " est égale à ", racineCarréeRécursive (nombre, valInit, 1))
        allerAlaLigne ( )
    fsi

    retourner OK

ffct principale

Définition de la fonction *racineCarréeItérative* :

fonction racineCarréeItérative (a, valInit : Réel) : Réel

    x = valInit : Réel

    pour i = 1 : Naturel **à** NB_CALCULS_MAX
        x $\leftarrow$ (x + a / x) / 2
    pour i

    retourner x

ffct racineCarréeItérative

Définition de la fonction *racineCarréeRécursive* :

fonction racineCarréeRécursive (a, valInit : Réel, nbCalculs : NaturelNonNul) : Réel

si nbCalculs > NB_CALCULS_MAX **alors**
retourner valInit
sinon
x = racineCarréeRécursive (a, valInit, nbCalculs +1) : Réel
retourner (x + a / x) / 2
fsi i

ffct racineCarréeRécursive

Définition de la fonction *valeurInitiale* :

Cette fonction utilise une fonction *puissance*, supposée prédéfinie, qui prend deux paramètres *n* et *m* de type Entier et qui renvoie un réel égal à n^m (*n* et *m* pouvant être négatifs). Voici la signature de cette fonction :

fonction puissance (n, m : Entier) : Réel

On va mettre le nombre *a* sous la forme $a = m \times 4^p, 1/4 \leq m \leq 1$, et donc $\sqrt{a} = \sqrt{m} \times 2^p$. La courbe $\sqrt{x}$ et la droite d'équation $y = \frac{1+2x}{3}$ passent toutes les deux par les points ($\frac{1}{4}$,$\frac{1}{2}$) et (1,1) et sont assez proches sur l'intervalle $]1/4, 1[$. Ce qui nous permet d'approximer $\sqrt{m}$ par $\frac{1+2m}{3}$ sur l' intervalle $[1/4, 1]$.

fonction valeurInitiale (a : Réel) : Réel

p : Entier
m : Réel

si a < 1/4 **alors**
p ← 0
faire
p ← p - 1
m ← a / puissance (4, p)
tant que m < 1/4
sinon
p ← -1
faire
p ← p + 1
m ← a / puissance (4, p)
tant que m > 1
fsi

retourner ((1 + 2 x m) / 3) x puissance (2, p)

ffct valeurInitiale

AP11
Enoncé du Contrôle Continu de Langage C
Vendredi 6 janvier 2006

Document autorisé : le polycopié distribué en cours **Durée** : 1 heure

Vous devez impérativement utiliser la feuille jointe avec l'énoncé pour y inscrire les réponses aux questions posées à chacun des trois exercices.

- **Exercice 1** (sur 7 points) : Analysez ce premier programme C

monProgramme.c

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

int main ( int argc, char **argv ) {

  if ( argc != 2 ) {
    fprintf ( stderr, "usage : %s nombre\n", argv[0] );
    exit ( EXIT_FAILURE );
  }

  int n = atoi ( argv[1] ), i = 0, j, s = 0;

  for ( ; i < n ; s += ++i );
  printf ( "%d\n", s );
  return "EXIT_SUCCESS";
}
```

Vous devez répondre aux questions suivantes :

1. comment compiler ce fichier source dans le but d'obtenir le fichier exécutable *monProgramme* en demandant au compilateur d'afficher toutes les erreurs
2. y a-t-il une erreur *fatale* (qui stoppe le processus de compilation)
3. y a-t-il des erreurs de type *« warning »* (qui ne stoppent pas le processus de compilation)
4. en supposant que les éventuelles erreurs soient corrigées, indiquez ce qui est affiché dans la fenêtre *shell* pour l'exécution de :
   - **$ monProgramme**
   - **$ monProgramme 3**
   - **$ monProgramme 5**
5. que calcule et affiche ce programme

– **Exercice 2** (sur 6 points) : Voici un second programme C

appels.c

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

// Prototypes des fonctions

void f1 ( int a, int *b );
void f2 ( int *b, int c );
void f3 ( int *a, int c );
void f4 ( int **b, int a );

int main () {

  int x = 5, y = 7, z = 9;

  f1 ( x, &y );
  printf ( "x = %d, y = %d, z = %d\n", x, y, z );
  f2 ( &x, y );
  printf ( "x = %d, y = %d, z = %d\n", x, y, z );
  f3 ( &y, z );
  printf ( "x = %d, y = %d, z = %d\n", x, y, z );
  return 0;
}

// Définition des fonctions

void f1 ( int a, int *b ) {
  a = *b;
}

void f2 ( int *b, int c ) {
  *b = c;
}

void f3 ( int *a, int c ) {
  f4 ( &a, c );
}

void f4 ( int **b, int a ) {
  **b = a;
}
```

Ce programme compile normalement avec l'option *-Wall*. Il n'y a aucune erreur à l'exécution. Vous devez simplement écrire sur votre feuille de réponses ce que ce programme afficherait dans la fenêtre *shell* si on l'exécutait.

– **Exercice 3** (sur 7 points) : QCM à renseigner directement sur la feuille de réponses fournie
On rappelle qu'il y a au moins une réponse vraie pour chaque question et que :
  – toute case non-cochée correspondant à une réponse vraie entraîne une pénalité
  – toute case cochée correspondant à une réponse fausse entraîne également une pénalité

1. La valeur retournée par la fonction *main* peut être affichée sous le *shell* par la commande :
   - ☐ echo $STATUS
   - ☐ echo $?
2. La séquence d'instructions suivante provoque comme affichage :

   ```
   int n = 5 ;
   while ( n > 0 )  {
      n -= 2 ;
      printf ( " %d   ", n ) ; }
   ```

   - ☐ 5 3 1
   - ☐ 3 1 -1
   - ☐ 3 1
3. Quelle valeur contient la variable *n* après la séquence d'instructions suivante :

   ```
   int n, tab [ ] = { 2, 3, 4, 2 } ;
   n = * ( tab + tab [ 0 ] ) ;
   ```

   - ☐ 2
   - ☐ 3
   - ☐ 4
   - ☐ aucune car l'expression affectée à la variable *n* est syntaxiquement incorrecte
4. La déclaration **int (*f [ ]) (int, int) ;** considère *f* comme :
   - ☐ un tableau de fonctions de deux entiers retournant un pointeur d'entier
   - ☐ un tableau de pointeur sur fonctions de deux entiers retournant un entier
   - ☐ un pointeur sur un tableau de fonctions de deux entiers retournant un entier
5. Quand on veut lire à partir du clavier une séquence de caractères pouvant contenir des espaces, il faut utiliser :
   - ☐ la fonction *scanf*
   - ☐ la fonction *gets*
   - ☐ la fonction *fgets* car elle est sécurisée
6. Le fait de tester la valeur de retour de la fonction *fopen* :
   - ☐ évite de bloquer le processus de compilation
   - ☐ permet d'éviter le risque d'une « erreur de segmentation » à l'exécution
   - ☐ n'a aucune incidence
7. Quand on utilise correctement la fonction *fgets*, si la longueur d'une ligne lue à partir d'un fichier texte est supérieure au nombre de caractères demandé :
   - ☐ les caractères en nombre exédentaire ne sont pas lus
   - ☐ les caractères en nombre exédentaire sont perdus
   - ☐ les caractères en nombre exédentaire seront lus à l'appel suivant de *fgets*
8. Quand après en avoir lu une partie, on veut se repositionner en début d'un fichier ouvert en lecture seule, il vaut mieux :
   - ☐ utiliser la fonction *rewind*
   - ☐ fermer et réouvrir le fichier
   - ☐ utiliser la fonction *fseek* avec **0** et **SEEK_SET** comme 2ème et 3ème paramètres
9. Dans un fichier d'en-tête (**.h**), on peut trouver :
   - ☐ des définitions de types
   - ☐ des définitions de fonctions
   - ☐ des prototypes de fonctions
10. Si dans un fichier *makefile* on trouve la ligne suivante :

    monFichier.o : monFichier.o monFichier.h

    - ☐ ça ne pose aucun problème
    - ☐ cela produit une « dépendance circulaire »

AP11
Corrigé du Contrôle Continu de Langage C

– **Exercice 1** :

1. commande de compilation : **$ gcc -Wall monProgramme.c -o monProgramme**

2. y a-t-il une erreur *fatale* : **oui** □ **non** ■

3. y a-t-il des erreurs de type *« warning »* : **oui** ■ **non** □ , si **oui** lesquelles :

   – « return transforme en entier un pointeur sans transtypage » : *EXIT_SUCCESS* est une constante symbolique et non une chaîne de caractères
   – « variable inutilisée j »

4. en supposant que les éventuelles erreurs soient corrigées, indiquez ce qui est affiché dans la fenêtre *shell* pour l'exécution de :

   – **$ monProgramme**
   usage : monProgramme nombre

   – **$ monProgramme 3**
   6

   – **$ monProgramme 5**
   15

5. que calcule et affiche ce programme :

   – ce programme calcule et affiche la somme des *nombre* premiers naturels non nuls ; *nombre* étant l'argument de la commande *monProgramme*

– **Exercice 2** :

– **x = 5, y = 7, z = 9**

– **x = 7, y = 7, z = 9**

– **x = 7, y = 9, z = 9**

– **Exercice 3** : QCM

1. La valeur retournée par la fonction *main* peut être affichée sous le *shell* par la commande :
   □ echo $STATUS
   ■ echo $?
2. La séquence d'instructions suivante provoque comme affichage :

```
int n = 5 ;
while ( n > 0 )  {
   n -= 2 ;
   printf ( " %d   ", n ) ; }
```

- □ 5 3 1
- ■ 3 1 -1
- □ 3 1

3. Quelle valeur contient la variable *n* après la séquence d'instructions suivante :

   int n, tab [ ] = { 2, 3, 4, 2 } ;
   n = * (tab + tab [0]) ;

   - □ 2
   - □ 3
   - ■ 4
   - □ aucune car l'expression affectée à la variable *n* est syntaxiquement incorrecte

4. La déclaration **int (*f [ ]) (int, int) ;** considère *f* comme :
   - □ un tableau de fonctions de deux entiers retournant un pointeur d'entier
   - ■ un tableau de pointeur sur fonctions de deux entiers retournant un entier
   - □ un pointeur sur un tableau de fonctions de deux entiers retournant un entier
5. Quand on veut lire à partir du clavier une séquence de caractères pouvant contenir des espaces, il faut utiliser :
   - □ la fonction *scanf*
   - □ la fonction *gets*
   - ■ la fonction *fgets* car elle est sécurisée
6. Le fait de tester la valeur de retour de la fonction *fopen* :
   - □ évite de bloquer le processus de compilation
   - ■ permet d'éviter le risque d'une « erreur de segmentation » à l'exécution
   - □ n'a aucune incidence
7. Quand on utilise correctement la fonction *fgets*, si la longueur d'une ligne lue à partir d'un fichier texte est supérieure au nombre de caractères demandé :
   - ■ les caractères en nombre exédentaire ne sont pas lus
   - □ les caractères en nombre exédentaire sont perdus
   - ■ les caractères en nombre exédentaire seront lus à l'appel suivant de *fgets*
8. Quand après en avoir lu une partie, on veut se repositionner en début d'un fichier ouvert en lecture seule, il vaut mieux :
   - ■ utiliser la fonction *rewind*
   - □ fermer et réouvrir le fichier
   - ■ utiliser la fonction *fseek* avec **0** et **SEEK_SET** comme 2ème et 3ème paramètres
9. Dans un fichier d'en-tête (**.h**), on peut trouver :
   - ■ des définitions de types
   - □ des définitions de fonctions
   - ■ des prototypes de fonctions
10. Si dans un fichier *makefile* on trouve la ligne suivante :

    monFichier.o : monFichier.o monFichier.h

    - □ ça ne pose aucun problème
    - ■ cela produit une « dépendance circulaire »

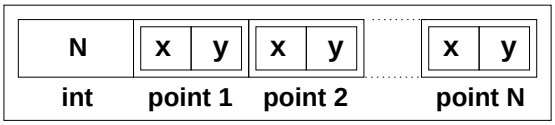
AP11 - Enoncé du Contrôle Continu de Langage C
Mercredi 10 Janvier 2007

Durée : 1 heure
Document autorisé : le polycopié distribué en cours
Vous devez impérativement utiliser la feuille jointe avec l'énoncé pour y inscrire les réponses aux questions posées à chacun des deux exercices.

Exercice 1 (sur 10 points)
La fonction ci-dessous permet de lire un fichier binaire de points. Le type *Point* est défini dans le fichier *point.h* comme une structure à deux champs de type *int* :

```
typedef struct { int x;
                 int y; } Point;
```

La structure du fichier est très simple ; le nombre de points codé sur un *int* suivi des différents points :



Voici la définition de la fonction C *lireFichier*, déclarée dans le fichier *io.h*, qui permet de lire le nombre de points et l'ensemble de points à partir du fichier de points :

```
#include <stdlib.h>
#include <stdio.h>
#include "point.h"
#include "io.h"

Point *lireFichier ( FILE *f, int *pNb ) {

  Point *pPoint;

  // lecture du nombre de points
  fread ( pNb, sizeof(int), 1, f );

  // allocation dynamique du tableau de points
  if ( ( pPoint = malloc ( *pNb * sizeof(Point) ) ) == NULL ) {
    fprintf ( stderr, "Problème allocation mémoire\n" );
    exit ( EXIT_FAILURE );
  }
  // lecture de l'ensemble de points
  if ( fread ( pPoint, sizeof(Point), *pNb, f ) != *pNb ) {
    fprintf ( stderr, "Problème lecture fichier\n" );
    exit ( EXIT_FAILURE );
  }
  return pPoint;
}
```

Et voici la fonction *main* qui appelle la fonction C *lireFichier* :

```
#include <stdlib.h>
#include <stdio.h>
#include "point.h"
#include "io.h"
```

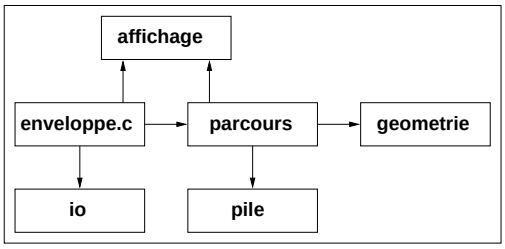
```
int main ( int argc, char **argv ) {

  Point *tabPoints = NULL; // ne pointe sur rien de valide pour l'instant
  int nbPoints;
  FILE *f;

  if ( argc != 2 ) {
    fprintf ( stderr, "usage : %s nomFichier\n", argv[0] );
    exit ( EXIT_FAILURE );
  }
  if( ( f = fopen ( argv[1], "rb" ) ) == NULL ) {
    fprintf ( stderr, "problème ouverture fichier\n" );
    exit ( EXIT_FAILURE );
  }
  // appel de la fonction lireFichier
  tabPoints = lireFichier ( f, &nbPoints );
  // la suite du programme
  return EXIT_SUCCESS;
}
```

Vous remarquerez que la fonction C *lireFichier* n'est pas typée *void* et qu'elle prend néanmoins un paramètre en résultat : *pNb*. Elle implémente un « mélange » de fonction et de procédure du langage algorithmique.
Si l'on voulait supprimer cette « anomalie », qui n'en est pas une pour le langage C, il faudrait typer *void* la fonction C *lireFichier* et lui rajouter un autre paramètre en résultat. C'est ce que vous avez à faire sur la feuille de réponses fournie.

Exercice 2 (sur 10 points)
Voici, représentée graphiquement, une application qui détermine l'enveloppe convexe d'un ensemble de points dans le plan.



Cette application est constituée d'un programme principale *enveloppe.c* (qui contient la définition de la fonction *main*) et de cinq modules. Chaque module est constitué d'un fichier *.h* (l'interface du module) et d'un fichier *.c* (sa réalisation). Ainsi la fonction *lireFichier* de l'exercice 1 est déclarée dans le fichier *io.h* et définie dans le fichier *io.c*.
Une flèche qui part d'un bloc 1 (programme principal ou module) vers un bloc 2 (module) signifie, qu'au moins, une fonction définie dans le bloc 1 appelle une fonction définie dans le bloc 2.
D'autre part, le programme principal et tous les modules ont besoin de connaître le type *Point* défini dans le fichier *point.h* (non représenté sur le graphique).
Enfin, le module *affichage* utilise la bibliothèque *XWindow*. Il faudra prendre en compte les options supplémentaires de la commande *gcc* pour compiler le fichier *affichage.c* et réaliser l'édition des liens (comme dans l'application *lissage*).
Ecrivez, sur la feuille de réponses fournie, le fichier *makefile* pour produire l'application *enveloppe*.

AP11
Corrigé du Contrôle Continu de Langage C

Exercice 1

io.c (extrait)

```
#include <stdlib.h>
#include <stdio.h>
#include "point.h"
#include "io.h"

void lireFichier ( FILE *f, int *pNb, Point **pTabPoints ) {

  // il faut supprimer la variable locale pPoints

  // lecture du nombre de points

  fread ( pNb, sizeof(int), 1, f );

  // allocation dynamique du tableau de points

  if ( ( *pTabPoints = malloc ( *pNb * sizeof(Point) ) ) == NULL ) {
    fprintf ( stderr, "Problème allocation mémoire\n" );
    exit ( EXIT_FAILURE );
  }
  // lecture de l'ensemble de points

  if ( fread ( *pTabPoints, sizeof(Point), *pNb, f ) != *pNb ) {
    fprintf ( stderr, "Problème lecture fichier\n" );
    exit ( EXIT_FAILURE );
  }
  // il faut supprimer l'instruction return
}
```

enveloppe.c

```
#include <stdlib.h>
#include <stdio.h>
#include "point.h"
#include "parcours.h"
#include "affichage.h"
#include "io.h"

int main ( int argc, char **argv ) {

  Point *tabPoints;
  int nbPoints;
  FILE *f;
```

enveloppe.c (suite)

```
  if ( argc != 2 ) {
    fprintf ( stderr, "usage : %s nomFichier\n", argv[0] );
    exit ( EXIT_FAILURE );
  }
  if( ( f = fopen ( argv[1], "rb" ) ) == NULL ) {
    fprintf ( stderr, "problème ouverture fichier\n" );
    exit ( EXIT_FAILURE );
  }
  // appel de la fonction lireFichier

  lireFichier ( f, &nbPoints, &tabPoints );

  // la suite du programme
  return EXIT_SUCCESS;
}
```

Exercice 2

makefile

```
OBJETS = enveloppe.o parcours.o pile.o geometrie.o io.o affichage.o

enveloppe : $(OBJETS)
        gcc -Wall -L/usr/X11R6/lib -lX11 $(OBJETS) -o enveloppe

enveloppe.o : parcours.h io.h affichage.h point.h enveloppe.c
        gcc -Wall -c enveloppe.c

parcours.o : parcours.h affichage.h pile.h geometrie.h point.h parcours.c
        gcc -Wall -c parcours.c

pile.o : pile.h point.h pile.c
        gcc -Wall -c pile.c

geometrie.o : geometrie.h point.h geometrie.c
        gcc -Wall -c geometrie.c

io.o : io.h point.h io.c
        gcc -Wall -c io.c

affichage.o : affichage.h point.h affichage.c
        gcc -Wall -I/usr/include/X11 -c affichage.c
```

AP11
Enoncé du Contrôle Final n°1
Vendredi 3 février 2006

Document autorisé : le polycopié distribué en cours **Durée** : 1h30

Exercice d'Algorithmique (sur 10 points)

Soient *n* et *d* deux naturels non nuls. Si *d* est un diviseur de *n* et s'il est premier avec *n* /*d*, alors *d* est un diviseur unitaire de *n*. Un nombre est unitairement parfait s'il est égal à la somme de ses diviseurs unitaires autres que lui-même.
On ne connaît que cinq nombres unitairement parfaits : 6, 60, 90, 87360 et $1{,}4582\ldots10^{23}$; on conjecture qu'ils sont en nombre fini.

Définissez en **langage algorithmique** la fonction *estNombreUnitairementParfait* qui prend un paramètre *nombre* de type naturel non nul et qui renvoie un booléen égal à VRAI si *nombre* est unitairement parfait, à FAUX sinon.
Adoptez impérativement une démarche descendante. Définissez également la fonction *principale* qui permettra de saisir un nombre, d'appeler la fonction *estNombreUnitairementParfait* et d'afficher de manière lisible le résultat.

Exercice de Langage C et Structures de Données (sur 10 points)

On considère deux listes linéaires triées *liste1* et *liste2*. Elles sont gérées par pointeurs et contiennent des valeurs entières sans doublon. Voici, pour rappel, la définition du type *Liste* en langage C :

```
typedef struct element { int valeur;
                         struct element *lien; } Element;

typedef Element *Liste;
```

Définissez en **langage C** la fonction **itérative** *intersectionListes* qui prend deux paramètres *liste1* et *liste2* de type *Liste* et qui renvoie un objet de type *Liste* : la liste qui contient les valeurs présentes simultanément dans *liste1* et *liste2*. Voici le prototype de cette fonction :

```
  // Intersection des deux listes : liste1 et liste2

Liste intersectionListes ( Liste liste1, Liste liste2 );
```

Pour ce faire, vous disposez des fonctions C dont les prototypes suivent :

```
  // Création d'une nouvelle liste vide

Liste nouvelleListe ( void );

  // Test si la liste est vide

Booleen estListeVide ( Liste liste );

  // Enlister un entier

Liste enlister ( Liste liste, int v );
```

AP11
Corrigé du Contrôle Final n°1

Exercice d'Algorithmique

Définition de la fonction *estNombreUnitairementParfait*

```
fonction estNombreUnitairementParfait ( nombre : NaturelNonNul ) : Booléen

    somme = 0 : Naturel

    pour diviseur = 1 : NaturelNonNul à nombre div 2
        si nombre mod diviseur = 0 et estDiviseurUnitaire ( nombre, diviseur ) alors
            somme ← somme + diviseur
        fsi
    fpour diviseur

    retourner ( somme = nombre )

ffct estNombreUnitairementParfait
```

Définition de la fonction *estDiviseurUnitaire*

```
fonction estDiviseurUnitaire ( n, d : NaturelNonNul ) : Booléen

    si pgcd ( n div d , d ) = 1 alors  // 2 nombres sont premiers entre eux si leur pgcd est égal à 1
        retourner VRAI
    fsi

    retourner FAUX

ffct estDiviseurUnitaire
```

Définition récursive de la fonction *pgcd*

```
fonction pgcd ( a, b : NaturelNonNul ) : NaturelNonNul

    r = a mod b : Naturel

    si r = 0 alors
        retourner b
    fsi

    retourner pgcd ( b, r )

ffct pgcd
```

Définition de la fonction *principale*

fonction principale ( ) : CodeRetour

```
    nombreAtester : NaturelNonNul

    afficher ( "Saisir un nombre ≥ 1 : " )
    saisir ( nombreAtester )

    afficher ( nombreAtester )
    si estNombreUnitairementParfait ( nombreAtester ) alors
        afficher ( " est " )
    sinon
        afficher ( " n'est pas " )
    fsi
    afficher ( "un nombre unitairement parfait" )
    allerAlaLigne ( )

    retourner OK
```

ffct principale

Exercice de Langage C et Structures de Données

```
  // Intersection des deux listes : liste1 et liste2

Liste intersectionListes ( Liste liste1, Liste liste2 ) {

  Liste liste = nouvelleListe ( );
  Liste ptrCourant1 = liste1, ptrCourant2 = liste2;

  while ( ( ptrCourant1 != NULL ) && ( ptrCourant2 != NULL ) ) {

    while ( ( ptrCourant1 != NULL ) && ( ptrCourant2 != NULL ) &&
            ( ptrCourant1->valeur != ptrCourant2->valeur) ) {

      if ( ptrCourant1->valeur < ptrCourant2->valeur )
        ptrCourant1 = ptrCourant1->lien;
      else
        ptrCourant2 = ptrCourant2->lien;
    }

    if ( ptrCourant1 != NULL &&  ptrCourant2 != NULL &&
         ptrCourant1->valeur == ptrCourant2->valeur ) {

      liste = enlister ( liste, ptrCourant1->valeur );
      ptrCourant1 = ptrCourant1->lien;
      ptrCourant2 = ptrCourant2->lien;
    }
  }
  return liste;
}
```

AP11 - Enoncé du Contrôle Final n°1
Lundi 29 Janvier 2007

Durée : 1 heure 30
Document autorisé : le polycopié distribué en cours

Exercice d'algorithmique (sur 10 points)

Soit une matrice d'entiers notée *matrice* définie par :

constante N = 3, M = 4 // par exemple

type Matrice = **tableau**[1 .. N][1 .. M] **de** Entier

matrice : Matrice

Un couple d'indices (*l* , *c*) représente un min-max de cette matrice si l'élément *matrice* [*l*][*c*] est un minimum de la ligne *l* et un maximum de la colonne *c*.
La méthode qui permet de déterminer les couples min-max d'une matrice (s'il en existe) consiste pour chaque ligne *l* à :

- déterminer les minima de cette ligne ;
- pour chacun de ces minima d'indice *c*, vérifier si *matrice* [*l*][*c*] est un maximum pour sa colonne.

Définir **en langage algorithmique** la procédure *afficherMinMax* qui prend en donnée un paramètre *matrice* de type *Matrice* et qui affiche dans la fenêtre *shell* tous les couples min-max de cette matrice. Cet affichage devra être réalisé comme le montre l'exemple suivant :

```
Le couple (2,3) de valeur 7 est un min-max
```

Quelques rappels sur les aspects méthodologiques qui vous ont été présentés durant ces cinq derniers mois :

- l'approche descendante ; votre algorithme doit être constitué de plusieurs fonctions/procédures et cet exercice s'y prête fort bien ;
- tout objet utilisé doit être préalablement défini ;
- le choix judicieux des identificateurs de fonction, procédure et variable permettra d'améliorer la lisibilité de votre algorithme ;
- il en va de même de l'indentation de votre texte.

Exercice de langage C et structures de données (sur 10 points)

La structure de liste linéaire permet de représenter un graphe comme il est indiqué sur le transparent 3.7 du chapitre 11 de votre polycopié. Vous trouverez ci-dessous le même exemple avec une numérotation des sommets du graphe qui commence à zéro (à la C).
Voici les définitions de types, en langage C, qui permettent de représenter un graphe par ses listes de successeurs gérées par pointeurs :

- pour les différents successeurs d'un sommet donné (listes représentées horizontalement sur le dessin). Ces listes peuvent être éventuellement vides (comme pour le sommet de numéro 2 sur l'exemple) ;

```
// définition des types Successeur et ListeSuccesseurs;

typedef struct successeur { int numero;
                            struct successeur *lien; } Successeur;
```

```
typedef Successeur *ListeSuccesseurs;
```

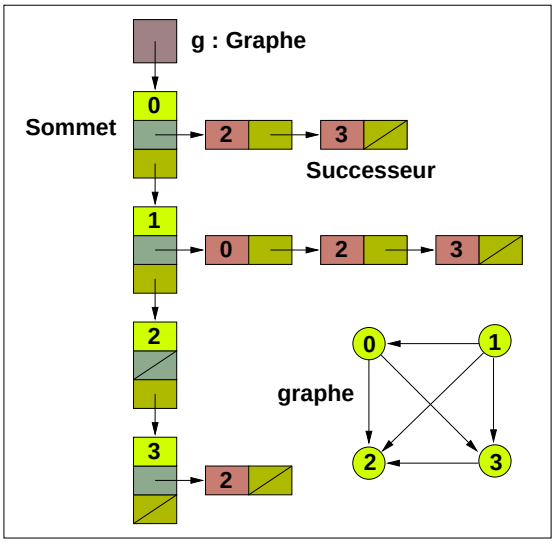
– pour les différents sommets du graphe (liste représentée verticalement sur le dessin).

```
// définition des types Sommet et Graphe

typedef struct sommet { int numero;
                        ListeSuccesseurs successeurs;
                        struct sommet *lien; } Sommet;

typedef Sommet *Graphe;
```

Ces listes sont triées selon les numéros de sommet croissants. Cela permet un parcours partiel de la liste de sommets quand on recherche un sommet donné et de même, un parcours partiel d'une liste de successeurs quand on veut vérifier si un sommet est successeur d'un autre sommet.



Vous devez définir, **en langage C**, la fonction *afficherSuccesseurs* qui prend un paramètre *g* de type *Graphe* et un paramètre *num* de type *int*. Cette fonction affichera dans la fenêtre *shell* les successeurs du sommet de numéro *num* comme le montre les deux exemples suivants :

– **`Successeur(s) de 0 : 2 3`**
– **`Successeur(s) de 2 :`**

Voici le prototype de cette fonction :

```
void afficherSuccesseurs ( Graphe g, int num );
```

Vous disposez de la fonction C *estGrapheVide* qui prend un paramètre *g* de type *Graphe* et qui renvoie la valeur *VRAI* si *g* contient la valeur *NULL*, *FAUX* sinon. Voici son prototype :

```
Booleen estGrapheVide ( Graphe g );
```

Quelques conseils :

– adoptez une démarche descendante : la fonction C *afficherSuccesseurs* parcourera partiellement la liste de sommets car elle est triée ; une autre fonction C parcourera totalement la liste de successeurs ;
– préférez une implémentation récursive de ces deux fonctions ;
– il y a très peu de C à écrire. Vérifiez soigneusement la syntaxe de votre source.

AP11
Corrigé du Contrôle Final n°1

Exercice 1

```
constante N = 3, M = 4 // par exemple

type Matrice = tableau[1 .. N][1 .. M] de Entier
```

Définition de la procédure *affichierMinMax*

```
procédure afficherMinMax ( donnée matrice : Matrice)

    minLigne : Entier // pour mémoriser la valeur minimum de la ligne

    // balayage des lignes

    pour ligne = 1 : Naturel à N

        // recherche du minimum sur la ligne

        minLigne ← minimumLigne ( matrice, ligne )

        // parcours des minima d'une ligne

        pour colonne = 1 : Naturel à M
            si matrice [ ligne ] [ colonne ] = minLigne alors
                si estMaximumColonne ( matrice, ligne, colonne ) alors

                    afficher ( "Le couple ( ", ligne, " , ", colonne, " ) de valeur " )
                    afficher ( matrice [ ligne ] [ colonne ], " est un min-max" )
                    allerAlaLigne ( )

                fsi
            fsi
        fpour colonne

    fpour ligne

fproc afficherMinMax
```

Définition de la fonction *minimumLigne*

```
fonction minimumLigne ( matrice : Matrice, ligne : Naturel ) : Entier

    minimum = matrice [ ligne ] [ 1 ] : Entier

    pour colonne = 2 : Naturel à M
        si matrice [ ligne ] [ colonne ] < minimum alors
            minimum ← matrice [ ligne ] [ colonne ]
        fsi
    fpour colonne
```

retourner minimum

ffct minimumLigne

Définition de la fonction *estMaximumColonne*

fonction estMaximumColonne (matrice : Matrice, ligne, colonne : Naturel) : Booléen

pour l = 1 : Naturel **à** N
si matrice [l] [colonne] > matrice [ligne] [colonne] **alors**
retourner FAUX
fsi
fpour l

retourner VRAI

ffct minimumLigne

Exercice 2

Définition de la fonction C *afficherSuccesseurs*

```
void afficherSuccesseurs ( Graphe g, int num ) {

  if ( !estGrapheVide ( g ) && g->numero < num )
    afficherSuccesseurs ( g->lien, num );
  else
    if ( !estGrapheVide ( g ) && g->numero == num ) {
      printf ( "Successeur(s) de %d : ", num );
      afficherListe ( g->successeurs );
    }
}
```

Définition de la fonction C, modifiée *static*, *afficherListe*

```
static void afficherListe ( ListeSuccesseurs liste ) {

  if ( liste != NULL ) {
    printf ( "%d ", liste->numero );
    afficherListe ( liste->lien );
  }
  else
    printf ( "\n" );
}
```

```
        colonne ← colonne - 1

      fsi
  ftq
```

fproc remplirCarré

Exercice 2

Définition de la fonction *minimum* appelée par les deux versions de la fonction *minListe*

```
int mininum ( int a, int b ) {

  if ( a < b )
    return a;
  return b;
}
```

Définition de la fonction C itérative *minListe*

```
int minListe ( Liste l ) {

  int min = l->valeur;  // il y a au moins un élément
                        // dans la liste
  l = l->lien;
  while ( l != NULL ) {
    min = minimum ( l->valeur, min );
    l = l->lien;
  }
  return min;
}
```

Définition de la fonction C récursive *minListe*

```
int minListe ( Liste l ) {

  if ( l->lien == NULL) // il y a au moins un élément
    return l->valeur;   // dans la liste
  else
    return minimum ( l->valeur, minListe ( l->lien ) );
}
```

AP11
Enoncé du Contrôle Final n°2
Vendredi 21 avril 2006

Document autorisé : le polycopié distribué en cours **Durée** : 1h30

Exercice d'Algorithmique (sur 10 points)

Pour mémoire, les coefficients binomiaux sont définis par :

$C_n^n = C_n^0 = 1$ et $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ si $0 < p < n$, n étant un naturel non nul et p étant un naturel

Définissez en **langage algorithmique** une procédure *développer* qui prend en donnée un paramètre *n* de type NaturelNonNul et qui affiche le développement de l'expression $(x + y)^n$ dont on rappelle que c'est la somme des termes $C_n^p x^{n-p} y^p$, $0 \leq p \leq n$.
Vous devez obligatoirement adopter une démarche descendante ; i. e. définir une fonction et plusieurs procédures. Prenez comme repère, l'algorithme qui affiche l'histogramme correspondant à un tableau d'entiers. On ne doit pas trouver d'appel de l'action *afficher* dans la définition de la procédure *développer* elle-même.
De plus, soignez particulièrement l'affichage. x^n devra être affiché sous la forme **x^n**. La présence d'un caractère espace peut être représenté par le symbole '␣'. L'algorithme, s'il était implémenté, devrait afficher dans la fenêtre *shell* :

- pour *n* = 1 : **`x + y`**
- pour *n* = 2 : **`x^2 + 2 x y + y^2`**
- pour *n* = 3 : **`x^3 + 3 x^2 y + 3 x y^2 + y^3`**

Enfin, **vous n'avez pas à écrire** la fonction *principale*.

Exercice de Langage C et Structures de Données (sur 10 points)

On considère une liste linéaire *liste* non nécessairement triée, éventuellement avec doublon. Cette liste est gérée par pointeurs et contient des valeurs entières. Voici, pour rappel, la définition du type *Liste* en langage C :

```
typedef struct element { int valeur;
                         struct element *lien; } Element;

typedef Element *Liste;
```

Vous allez vous intéresser au « vidage » de cette liste ; i. e. retirer ses éléments un à un à l'aide d'une désallocation mémoire.
Dans un premier temps, définissez en **langage C** la fonction **itérative** *viderListe* qui prend un paramètre *liste* de type *Liste* et qui renvoie un objet de type *Liste* correspondant à une liste vide. Voici le prototype de cette fonction :

```
Liste viderListe ( Liste liste ); // style fonctionnel
```

Redéfinissez ensuite, toujours en **langage C**, la fonction **récursive** *viderListe* typée *void* qui prend un paramètre *pListe* de type *Liste* *. Voici le prototype de cette fonction :

```
void viderListe ( Liste *pListe ); // style procédurale
```

Vous ne disposez que de la fonction C *estListeVide* dont voici le prototype :

```
Booleen estListeVide ( Liste liste );
```

En aucun cas, n'utilisez la fonction C *delister*. D'ailleurs, vous ne connaissez pas les valeurs stockées dans cette liste.

AP11 - Enoncé du Contrôle Final nº2
Mercredi 4 avril 2007

Durée : 1 heure 30
Document autorisé : le polycopié distribué en cours

Exercice d'algorithmique (sur 10 points)

Un carré magique est un tableau de *N* lignes et *N* colonnes. Chaque case de ce tableau doit contenir un nombre différent allant de 1 à N^2. La somme des nombres de chaque ligne, de chaque colonne et des deux diagonales doit être la même.
Il existe une méthode prouvée pour construire un carré magique d'ordre impair (*N* impair), la voici :

- les nombres sont placés par ordre croissant (de 1 à N^2) ;
- le milieu de la dernière colonne est la première position de placement (le nombre 1) ;
- on place les N^2-1 valeurs restantes comme suit :
  - on cherche à placer la valeur suivante dans la case en dessous à droite de la position courante ;
  - si la position courante est sur la dernière ligne ou/et sur la dernière colonne, on juxtapose le même tableau de telle sorte qu'il existe toujours une case située en dessous à droite de la position courante ;
  - si cette case est vide, la valeur suivante y est placée ;
  - sinon la case à gauche de la position courante est nécessairement vide et on y place la valeur suivante.

Prenons comme exemple la création d'un carré magique d'ordre 5. Voici son contenu après placement des dix premières valeurs et après placement des 25 valeurs :

	10	4		
		6	5	
			7	1
2				8
9	3			

11	10	4	23	17
18	12	6	5	24
25	19	13	7	1
2	21	20	14	8
9	3	22	16	15

Vous pouvez vérifier que la somme des nombres de chaque ligne, de chaque colonne et des deux diagonales est égale à 65.

Voici la définition du type *Carré* :

constante N = 5 // obligatoirement impaire

type Carré = **tableau**[1 .. N][1 .. N] **de** Naturel

Vous devez définir, **en langage algorithmique**, la procédure *construireCarré* qui prend un paramètre en résultat *carréMagique* de type *Carré* et qui insert dans ce carré les valeurs allant de 1 à N^2 selon la méthode décrite ci-dessus.
Une initialisation du carré avec une valeur appropriée peut faciliter l'expression de l'algorithme.

Quelques rappels sur les aspects méthodologiques qui vous ont été présentés durant le module :

- l'approche descendante ;
- la cohérence algorithmique : l'appel d'une fonction/procédure doit être en cohérence avec sa définition ;
- tout objet utilisé doit être préalablement défini ; un objet n'est visible que dans dans l'entité (fonction ou procédure) où il a été défini ;

– le choix judicieux des identificateurs de fonction, procédure et variable permettra d'améliorer la lisibilité de votre algorithme ;
– il en va de même de l'indentation de votre texte.

Exercice de langage C et structures de données (sur 10 points)

On considère une liste linéaire **non triée** d'entiers gérée par pointeurs. Cette liste peut éventuellement contenir des doublons.
Voici un rappel en langage C de la définition du type *Liste* contenue dans le fichier *liste.h* :

```
typedef struct element { int valeur;
                         struct element *lien; } Element;

typedef Element *Liste;
```

Vous devez déterminer un élément de valeur **minimum** dans cette liste. On supposera que la liste n'est pas vide ; elle contient au moins un élément. Cela signifie que dans l'appelant, il y aura un test préalable (utilisant la fonction *estListeVide*) avant l'appel de la fonction *minListe*. Mais vous n'avez pas à écrire cette séquence d'appel.

Vous devez définir, **en langage C**, deux versions de la fonction C *minListe* :

– une version **itérative**
– une version **récursive**

Ces deux versions ont le même prototype :

```
int minListe ( Liste l );
```

La définition d'une fonction C renvoyant le minimum de deux entiers et utilisable par les deux versions de la fonction *minListe* serait la bienvenue.

AP11
Corrigé du Contrôle Final n°2

Exercice 1

constante N = 5 // obligatoirement impaire

type Carré = **tableau**[1 .. N][1 .. N] **de** Naturel

Définition de la procédure *construireCarré*

procédure construireCarré (**résultat** carréMagique : Carré)

```
    initialiserCarré ( carréMagique )
    remplirCarré ( carréMagique )
```

fproc construireCarré

Définition de la procédure *initialiserCarré*

constante VIDE = 0 // par convention pour distinguer les cases vides des autres

procédure initialiserCarré (**résultat** carré : Carré)

```
    pour ligne = 1 : Naturel à N
        pour colonne = 1 : Naturel à N
          carré [ ligne ] [ colonne ] ← VIDE
        fpour colonne
    fpour ligne
```

fproc initialiserCarré

Définition de la la procédure *remplirCarré*

procédure remplirCarré (**donnée-résultat** carré : Carré)

```
    ligne = N div 2 + 1, colonne = N : Naturel     // position courante
    nouvelleLigne, nouvelleColonne : Naturel       // position nouvelle possible
    valeur = 1, valeurMax = N × N : Naturel        // valeurs courante et maximum

    tant que valeur ≤ valeurMax faire              // boucle de placement

        carré [ ligne ] [ colonne ] ← valeur
        valeur ← valeur + 1

        nouvelleLigne ← ( ligne mod N ) + 1
        nouvelleColonne ← ( colonne mod N ) + 1

        si carré [ nouvelleLigne ] [ nouvelleColonne ] = VIDE alors

          ligne ← nouvelleLigne
          colonne ← nouvelleColonne

        sinon
```

```
        colonne ← colonne - 1

      fsi
  ftq
```

fproc remplirCarré

Exercice 2

Définition de la fonction *minimum* appelée par les deux versions de la fonction *minListe*

```
int mininum ( int a, int b ) {

  if ( a < b )
    return a;
  return b;
}
```

Définition de la fonction C itérative *minListe*

```
int minListe ( Liste l ) {

  int min = l->valeur;  // il y a au moins un élément
                        // dans la liste
  l = l->lien;
  while ( l != NULL ) {
    min = minimum ( l->valeur, min );
    l = l->lien;
  }
  return min;
}
```

Définition de la fonction C récursive *minListe*

```
int minListe ( Liste l ) {

  if ( l->lien == NULL) // il y a au moins un élément
    return l->valeur;   // dans la liste
  else
    return minimum ( l->valeur, minListe ( l->lien ) );
}
```