Exercice 2.7 Construire une liste à partir d'une source de données

• *Création depuis un tableau d'entiers*

Spécification de l'algorithme

```
FONCTION newListFromArray(content : entier[], longueur : entier) : list
{
VAR listenouv, courant : list ; i : entier
DEBUT
  SI (longueur < 1) ALORS // la liste sera vide
    RETOURNER NULL
  FINSI
  listenouv ← newSingletonList(content[0]) // construction de la tête
  courant ← listenouv
  // ajout des autres places à la suite
  POUR i DE 1 A longueur-1 FAIRE
    courant→succ ← newSingletonList(content[i])
    courant ← courant→succ    // passage au suivant
  FAIT
  RETOURNER listenouv // retour de la tête, qui représente la liste
FIN
```

Implantation C

```
list newListFromArray(int* content, int length)
{
  if (length < 1) return emptyList(); // cas de la liste vide
  // sinon : initialisation de la chaîne avec la création de la tête
  list l = newSingletonList(content[0]);
  // déclaration et initialisation d'un variable d'itération de liste
  list pp = l;
  // itération de la 1ère place à la dernière ([longueur-1] intervalles)
  int i;
  for (i = 1; i < length; i++)
  {
    pp->succ = newSingletonList(content[i]);
    //pp->succ->prev = pp;        // chaînage arrière (opt.)
    pp = pp->succ;                // itération de la liste
  }
  return l;// retour du pointeur sur la tête, lequel représente la liste
}
```

Il est possible mais risqué, par l'astuce suivante, d'éviter le passage du paramètre qui indique la longueur du tableau source : `size_t longueur = sizeof(contenu) / sizeof(int);`

- *Duplication d'une liste existante*

Spécification de l'algorithme

```
FONCTION duplicate(l : list) : list
VAR tête, copie, copie_tête : list
DEBUT
  SI (l = NULL) ALORS
    RETOURNER NULL
  FINSI
  tête ← l
  copie ← newSingletonList(l→content)
  copie_tête ← copie
  TANTQUE (l→succ ≠ NULL) ET (l→succ ≠ tête) FAIRE
    l ← l→succ;    // parcours de la liste d'origine
    // création d'une place à partir de la place d'origine
    copie→succ ← newSingletonList(l→content)
    // insertion de la nouvelle place dans la copie
    SI (l→prev ≠ NULL) ALORS
      copie→succ→prev ← copie
    FINSI
    copie ← copie→succ
  FAIT
  SI (l→succ ≠ NULL) ALORS
    copie→succ ← copie_tête
  FINSI
  RETOURNER copie_tête
FIN
```

Implantation C

```
list duplicate(list l)
{
  if (l == NULL) return NULL;
  list head = l;
  list dupl = newSingletonList(l->content);
  list dupl_head = dupl;
  while (l->succ != NULL && l->succ != head)
  {
    l = l->succ;    // itération de la liste de référence
    dupl->succ = newSingletonList(l->content);
    if (l->prev != NULL) dupl->succ->prev = dupl;
    dupl = dupl->succ; // itération de la liste copiée
  }
```

```
  if (l->succ != NULL) dupl->succ = dupl_head;
  return dupl_head;
}
```

Exercice 2.8 Refermer une liste sur elle-même

Spécification

```
Donnée modifiée : la liste transformée
FONCTION refermer_sur_elle_meme(*l: liste)
VAR p: place
DEBUT
  SI estvide(*l) ALORS
    RETOURNER
  FINSI
  p ⟵ tête(*l)
  TANTQUE ¬ dernier(p) FAIRE
    p ⟵ succ(p)
  FAIT
  succ(p) ⟵ tête(*l)
FIN
```

Réalisation en C

```
Donnée modifiée : la liste transformée
void lin2circ(plist pl)
{
  if (pl == NULL) return;
  list l = *pl;
  if (l == NULL) return;
  while (l->succ != NULL) l = l->succ;  // itération de la liste
  l->succ = *pl;
}
```

Exercice 2.9 Effectuer et retourner deux calculs sur une liste

Étude

Il s'agit de parcourir la liste et de faire le produit d'une part les entiers positifs et d'autre part celui des entiers négatifs.

La présence éventuelle du 0 qui selon le statut qu'on lui donne est un positif ou non (positifs strictement ou non) peut poser un problème d'interprétation de l'énoncé. Le minimum est d'en parler. Disons, pour ce corrigé qu'il n'est ni positif ni négatif et qu'il doit donc ne pas intervenir dans aucun produit qu'il annulerait.

Les cas spéciaux sont les suivants :

- Liste vide ou ne contenant que des zéros : le résultat est indéterminé.
- Absence de positifs ou (exclusif) absence de négatifs : le résultat est partiellement indéterminé.

La difficulté de cet exercice tient également au problème du double résultat qu'il s'agit de retourner. Plusieurs solutions sont possibles, la plus simple consistant à utiliser deux entrées modifiées.

On utilisera un code de statut pour rendre compte des résultats spéciaux vs. standard dont les valeurs signifient les configurations suivantes :

- 4 : au moins un positif et au moins un négatif.
- 2 : au moins un positif mais aucun négatif.
- 1 : au moins un négatif mais aucun positif.
- 0 : ni négatif, ni positif : liste vide ou de zéros.

Spécification abstraite

```
Entrée : la liste d'entiers.
Entrée modifiée : le produit des positifs et le produit des négatifs.
Sortie : code de statut qui peut prendre l'un des 4 états TOUT (4),
UNIQUEMENT_POSITIFS (2), UNIQUEMENT_NEGATIFS (1), RIEN (0).
Algorithme :
FONCTION produits(l: liste<entier>, pos, neg: entier): statut
VAR pos_ok, neg_ok: booléen; p: place;
DEBUT
  pos ← 1
  pos_ok ← faux
  neg ← 1
  neg_ok ← faux
  SI ¬ estvide(l) ALORS
    p ← tête(l)
    SI contenu(p) > 0 ALORS
      pos ← contenu(p)
      pos_ok ← vrai
    FINSI
    SI contenu(p) < 0 ALORS
      neg ← contenu(p)
      neg_ok ← vrai
    FINSI
    TANTQUE ¬ dernier(p) FAIRE
      p ← succ(p)
      SI contenu(p) > 0 ALORS
        pos ← pos * contenu(p)
        pos_ok ← vrai
      FINSI
      SI contenu(p) < 0 ALORS
        neg ← neg * contenu(p)
        neg_ok ← vrai
      FINSI
    FAIT
```

```
  FINSI
  SI pos_ok ALORS
     SI neg_ok ALORS RETOURNER TOUT
    SINON RETOURNER UNIQUEMENT_POSITIFS
    FINSI
  SINON
    SI neg_ok ALORS RETOURNER UNIQUEMENT_NEGATIFS
    SINON RETOURNER RIEN
    FINSI
  FINSI
FIN
```

Implantation C

```
// status code : 4 : tout, 2 : positifs, 1 : négatifs, 0 : rien
int produits(list l, int *pos, int *neg)
{
  if (l == NULL) return 0;
  int pos_prod = 1;
  int pos_prod_ok = 0;
  int neg_prod = 1;
  int neg_prod_ok = 0;
  int content = l->content;  // pas indispensable mais économique
  if (content > 0)
  {
    pos_prod = content;
    pos_prod_ok = 1;
  }
  if (content < 0)
  {
    neg_prod = content;
    neg_prod_ok = 1;
  }
  while (l->succ != NULL)
  {
    l = l->succ;
    content = l->content;
    if (contenu > 0)
    {
      pos_prod *= content;
      pos_prod_ok = 1;
    }
    if (contenu < 0)
    {
```

```
      neg_prod *= content;
      neg_prod_ok = 1;
    }
  }
  if (pos_prod_ok) *pos = pos_prod;
  if (neg_prod_ok) *neg = neg_prod;
  if (pos_prod_ok && neg_prod_ok) return 4;
  if (pos_prod_ok && ! neg_prod_ok) return 2;
  if (! pos_prod_ok && neg_prod_ok) return 1;
  return 0;
}
```

Exercice 2.10 Couper une liste en deux

Étude

Dans les cinq cas de figure, il s'agit de vérifier l'existence d'une position de coupe dans la chaîne, et le cas échéant d'effectuer celle-ci, puis de savoir retourner les deux chaînes.
Le cas à évacuer en début de fonction est celui de la liste vide.
Ensuite, nous devons, respectivement pour les cinq cas :

- A – vérifier que le pointeur désigne bien un maillon de la liste ;
- B – vérifier que la liste contient bien la donnée x ;
- C – vérifier que la liste contient au moins k éléments ;
- D – identifier la valeur minimale de la liste ;
- E – mesurer la longueur de la liste.

Les cas A, B et C sont très similaires et doivent intégrer les cas de tête et de queue de liste.
Les cas D, E nécessitent un premier parcours complet de la liste.
Tous les cas sauf le cas E doivent intégrer la possibilité d'une coupure devant la tête de liste.
Le cas E revient quasiment au cas B après l'identification du minimum de la liste.
Dans tous les cas, il est pertinent de retourner un statut d'exécution qui indique si l'opération a été effective ou bien transparente (auquel cas, il n'y a pas eu production d'une seconde liste).
Il peut y avoir erreur si l'un ou l'autre des deux pointeurs sur les listes à modifier est NULL ou bien dans le cas particulier du A (B et C sont discutables) si le pointeur donné pointe sur une autre liste.
Comme nous sommes scrupuleux, nous indiquons également par le statut d'exécution si la seconde liste n'était pas initialisée à NULL avant l'opération effective (mise en garde : écrasement d'une ancienne valeur).

Implantation C

Schéma commun

```
Données modifiées : l1, la liste d'entrée coupée en deux qui devient la liste
amont, et l2 la liste aval éventuellement produite en sortie.
Résulat : un entier comme statut d'exécution
  -1 = ERR :ERREUR,
```

```
   0       : opération blanche,
   1       : une coupe a été effectuée,
   2       : statut 1 + l2 n'était pas nulle.
int cut<Version>(plist pl1, plist pl2, ...)
{
  if (pl1 == NULL || pl2 == NULL) return ERR;
  list l1 = *pl1;
  if (l1 == NULL) return 0;
  // on fixe le statut par défaut selon que *pl2 est NULL ou non
  int status = *pl2 == NULL ? 1 : 2;
  /// CODE SPECIFIQUE <Version> ///
  // on coupe
  *pl2 = l1->succ;
  l1->succ = NULL;
  return status;
}
```

Cas du sélecteur « pointeur »

Donnée : cutPoint, le pointeur sur la cellule suivant le point de coupe

```
int cutA(plist pl1, plist pl2, list cutPoint)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // si cutPoint est NULL, autant arrêter ici
  if (cutPoint == NULL) return 0;
  // cas de la coupure en tête de liste :
  if (cutPoint == l1)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  // ou jusqu'à la queue de liste (cutPoint pointe sur une autre liste)
  while (l1->succ != NULL && l1->succ != cutPoint) l1 = l1->succ;
  // si nous avons atteint la fin de liste, c'est un cas d'erreur
  if (l1->succ == NULL) return ERR;
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « élément »

Donnée : x, l'élément de la cellule suivant le point de coupe

```
int cutB(plist pl1, plist pl2, int x)
{
```

```
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // cas de la coupure en tête de liste :
  if (x == l1->content)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  // ou jusqu'à la queue de liste (pas d'élément x dans la liste)
  while (l1->succ != NULL && l1->succ->content != x) l1 = l1->succ;
  // si nous avons atteint la fin de liste, c'est une opération blanche
  if (l1->succ == NULL) return 0;
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « position »

Donnée : k, la distance de la tête de liste au point de coupe

```
int cutC(plist pl1, plist pl2, unsigned k)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // cas de la coupure en tête de liste :
  if (k == 0)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  // ou jusqu'à la queue de liste (cutPoint pointe sur une autre liste)
  while (l1->succ != NULL && --k > 0) l1 = l1->succ;
  // si nous avons atteint la fin de liste, c'est une erreur
  if (l1->succ == NULL) return ERR;
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « élément minimal »

```
int cutD(plist pl1, plist pl2)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // on réutilise la fonction minOf de mesure de longueur de liste
  int min = minOf(l1);  // sur vos copies, il faut la réécrire
  // cas de la coupure en tête de liste :
```

```
  if (min == l1->content)
  {
    *pl1 = NULL;
    *pl2 = l1;
    return status;
  }
  // sinon, on parcourt la liste jusqu'au prédécesseur du point de coupe
  while (l1->succ != NULL && l1->succ->content != min) l1 = l1->succ;
  // avoir atteint la queue de liste ne peut logiquement pas arriver..
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Cas du sélecteur « milieu de liste »

```
int cutE(plist pl1, plist pl2)
{
  /// CODE GENERIQUE COMMUN D'INITIALISATION ///
  // on réutilise la fonction len de mesure de longueur de liste
  int n = len(l1);  // sur vos copies, il faut la réécrire
  // on calcule la distance au point de coupe (note: distance >= 1)
  int cutPoint = (n % 2 == 0) ? n / 2 : n / 2 + 1;
  int status = 1;
  // on modifie le statut si *pl2 n'est pas NULL
  if (*pl2 != NULL) status = 2;
  // on effectue le parcours jusqu'au prédécesseur du point de coupe
  while (--cutPoint > 0) l1 = l1->succ;
  // avoir atteint la queue de liste ne peut logiquement pas arriver..
  /// CODE GENERIQUE COMMUN DE FINALISATION ///
}
```

Spécification abstraite

Une fois n'est pas coutume (petit mensonge d'ingénieur), nous procédons à une rétro conception du principe algorithmique général à partir des cinq cas étudiés et réalisés précédemment.

```
Données modifiées : l1, la liste d'entrée coupée en deux qui devient la liste
amont, et l2 la liste aval éventuellement produite en sortie.
Donnée : c, un paramètre du critère pour repérer la place qui suit
directement le point de coupe et qui devient donc la tête de l2
Résultat : un statut d'exécution
  -1 = ERR  :ERREUR,
   0 = ID : opération blanche,
   1 = OK : une coupe a été effectuée,
FONCTION scinder(modif l1, l2 = ø: liste; c: critère): statut
VAR p: place
```

```
DEBUT
  c ⟵ mettre_à_jour(c, l1)
  SI ¬ estvalide(c) ALORS
    RETOURNER ERR
  FINSI
  SI estvide(l1) ALORS
    RETOURNER ID
  FINSI
  p ⟵ tête(l1)
  SI est_successeur_point_de_coupe(p) ALORS
    l2 ⟵ l1
    l1 ⟵ ø
    RETOURNER OK
  FINSI
  TANTQUE ¬ dernier(p) ∧ ¬ est_successeur_point_de_coupe(succ(p)) FAIRE
    p ⟵ succ(p)
    c ⟵ mettre_éventuellement_à_jour(c)
  FAIT
  SI dernier(p) ALORS
    RETOURNER ID
  SINON
    tête(l2) ⟵ succ(p)
    succ(p) ⟵ ø
    RETOURNER OK
  FINSI
FIN
```

Exercice 2.11 Supprimer un sous-ensemble d'une liste

Étude

Cet exercice avec ses cinq cas de figure est une extension du principe algorithmique vu dans l'exercice 2.5 (supprimer des éléments). La façon de définir le critère de sélection des nœuds à supprimer change selon les cas, mais le schéma général ne change pas.

Dans tous les cas, il s'agit de traiter spécialement la chaîne préfixe (de tête) constitué d'une succession de nœuds à éliminer, puis s'il reste au moins un élément, d'éliminer tous les éléments répondant au critère et situés au-delà de la tête.

Le cas A donne lieu à une version simplifiée de l'algorithme général, laquelle profite à plein de la régularité des intervalles de suppression.

- *Schéma général pour les cas B à E*

La méthode générale de résolution reste la même, seul le critère de choix des éléments à supprimer change. Pour illustrer ce fait, dans la partie ***Spécification de l'algorithme***, le second paramètre de la fonction `removeAll` est en réalité une fonction nommée `critère`. Cette fonction `critère` a pour paramètre un `entier` (la valeur extraite de la liste) et pour sortie un `booléen` indiquant si la valeur

répond au critère de suppression. Cette fonction `critère` peut avoir d'autres paramètres, notamment un `entier` indiquant le seuil à partir duquel effectuer la suppression pour les cas D et E.

Spécification de l'algorithme

```
FONCTION removeAll(*pl:list, critère(entier,...) : booléen ): entier
VAR l, suppr : list
DEBUT
  SI (pl = NULL) ALORS
    RETOURNER -1 // -1 signale une erreur
  FINSI
  l ← *pl
  SI (l = NULL) ALORS
    RETOURNER 0 // 0 indique qu'aucune suppression n'est faite
  FINSI
  // suppression à partir de la tête de liste
  TANTQUE (l ≠ NULL) ET (critère(l→content,...)=vrai) FAIRE
    suppr ← l
    l ← suppr→succ
    LIBERER(suppr)
  FAIT
  *pl ← l
  // suite de la liste
  TANTQUE (l ≠ NULL) FAIRE
    TANTQUE (l→succ ≠ NULL) ET (critère(l→succ→content,...)=vrai) FAIRE
      suppr ← l→succ
      l→succ ← suppr →succ
      LIBERER(suppr)
    FAIT
    l ← l→succ
  FAIRE
  RETOURNER 1 // indique que tout s'est bien passé
FIN
```

Implantation C

```
Données modifiées : la liste dont on supprime des éléments.
Résultat : un entier comme statut d'exécution
  -1 = ERR : erreur,
   0 = ID  : opération blanche,
   1 = OK  : au moins une suppression a été effectuée.
int removeAll<V>(plist pl[, param<V>])
{
  if (pl == NULL) return ERR;  /// cas d'erreur
  list l = *pl;
```

```
  if (l == NULL) return ID;   /// cas d'opération blanche
  list killed;
  /// cas de suppression de toute la chaîne préfixe d'éléments impairs
  while (l != NULL && tosuppr<V>(l->content, param<V>))
  {
    killed = l;
    l = killed->succ;
    free(killed);
  }
  *pl = l;
  /// à partir de ce point : chaîne vide, soit tête non supprimée
  /// supression des éléments cf. critère et au-delà de la tête
  while (l != NULL)
  {
    while (l->succ != NULL && tosuppr<V>(l->succ->content, param<V>))
    {
      killed = l->succ;
      l->succ = killed->succ;
      free(killed);
    }
    /// on passe soit au conservé suivant, soit à NULL (fin de chaîne)
    l = l->succ;
  }
  return OK;
}
```

- *Adaptations pour les cas B à E*

Les correspondances suivantes permettent de spécialiser le schéma général pour répondre au quatre cas de figure B, C, D, E :

Spécification de l'algorithme

```
FONCTION critère_estPair(n : entier) : booléen // casB
VAR pair : booléen
DEBUT
  SI n % 2 = 0 ALORS
    pair ← vrai
  SINON
    pair ← faux
  FINSI
  RETOURNER pair
FIN
FONCTION critère_estImpair(n : entier) : booléen // casC
DEBUT
  RETOURNER ¬ critère_estPair(n)
```

```
FIN
FONCTION critère_estSup(n : entier, seuil : entier) : booléen // casD
VAR estSup : booléen
DEBUT
  SI n > seuil ALORS
    estSup ← vrai
  SINON
    estSup ← faux
  FINSI
  RETOURNER estSup
FIN
FONCTION critère_estInf(n : entier, seuil : entier) : booléen // casE
VAR estInf : booléen
DEBUT
  estInf ← faux
  SI (n < seuil) ALORS
    estInf ← vrai
  FINSI
  RETOURNER estInf
FIN
```

Implantation C

```
int removeAllB(plist pl)
  param<B> ≡ ø
  tosuppr<B>(element) ≡ element % 2 == 0
int removeAllC(plist pl)
  param<C> ≡ ø
  tosuppr<C>(element) ≡ element % 2 == 1
int removeAllD(plist pl, int threshold)
  param<D> ≡ int threshold
  tosuppr<D>(element, threshold) ≡ element > threshold
int removeAllE(plist pl, int threshold)
  param<E> ≡ int threshold
  tosuppr<E>(element, threshold) ≡ element < threshold
```

• *Adaptations pour le cas A*

Le cas de figure A, du fait de la régularité des intervalles de suppression conduit à une version simplifiée du schéma général précédent. On élimine notamment un niveau d'imbrication de boucles *tantque*, traduction du fait qu'il n'existe pas de chaîne de plusieurs éléments directement successifs à supprimer.

Spécification de l'algorithme

```
FONCTION removeAllA(*pl : list) : entier
VAR l, suppr : list
DEBUT
  SI (pl = NULL) ALORS
    RETOURNER -1
  FINSI
  l ← *pl
  SI (l = NULL) ALORS
    RETOURNER 0
  FINSI
  *pl ← l→succ
  LIBERER(l)
  l ← *pl
  TANTQUE (l ≠ NULL) ET (l→succ ≠ NULL) FAIRE
    suppr ← l→succ
    l→succ ← suppr →succ
    LIBERER(suppr)
    l ← l→succ
  FAIT
  RETOURNER 1
FIN
```

Implantation C

```
int removeAllA(plist pl)
{
  if (pl == NULL) return ERR;  /// cas d'erreur
  list l = *pl;
  if (l == NULL) return ID;   /// cas d'opération blanche
  /// suppression de la tête
  *pl = l->succ;
  free(l);
  l = *pl;
  /// supression à raison d'un nœud sur deux
  list killed;
  while (l != NULL && l->succ != NULL)
  {
    killed = l->succ;
    l->succ = killed->succ;
    free(killed);
    l = l->succ;
```

```
    }
    return OK;
}
```

Corrigé du problème

Problème 2.1 Saisir, enregistrer puis évaluer un polynôme

Étude

Pour le module de saisie, on s'inspirera de la fonctionnalité de construction de liste à partir de la saisie sur l'entrée standard, dont on fera une adaptation.
Pour l'évaluation, il s'agit d'un calcul itératif.
Deux fonctionnalités utilitaires pourront constituer une amélioration de la version de base :

- Une fonction d'affichage de la formule polynôme.
- Une fonction (laissée en exercice non corrigé) de normalisation du polynôme après sa saisie et avant son emploi pour des évaluations (optimisation).
- *La structure de donnée*

Pseudo code formel

```
STRUCTURE terme
  coeff   : réel
  exposant : entier
  *suivant : terme
TYPE *polynome : terme
```

Implantation C

```
typedef struct term
{
  long int coeff;   // intervalle : -2 147 483 648 à 2 147 483 647
  long int expon;
  struct term* nextTerm;
} term;
typedef term *polynomial;
```

- *Saisie du polynôme*

Implantation C

```
// Fonction dérivée de la fonction
// constr::newListFromStandardInput(list* pl) : int
void recordPolynomial(polynomial* pp)
{
  *pp = NULL;
  int size = 1;
  char coeffStr[11];
```

```
  char exponStr[11];
  printf("Saisissez une serie de coefficient et exposants,\n");
  printf("puis terminez votre saisie par le coeff 'fin' :\n");
  printf("coeff. %d> ", size);
  scanf("%s", coeffStr);
  if (strcmp("fin", coeffStr) == 0) return;
  printf("expon. %d> ", size);
  scanf("%s", exponStr);
  size++;
  *pp = malloc(sizeof(term));  // allocation et rattachement tête
  polynomial p = *pp;
  p->coeff = atol(coeffStr);
  p->expon = atol(exponStr);
  p->nextTerm = NULL;
  printf("coeff. %d> ", size);
  scanf("%s", coeffStr);
  if (strcmp("fin", coeffStr) == 0) return;
  polynomial prev_p;
  while (strcmp("fin", coeffStr) != 0)
  {
    prev_p = p;
    p = p->nextTerm;         // itération
    printf("expon. %d> ", size);
    scanf("%s", exponStr);
    size++;
    p = malloc(sizeof(term));    // allocation
    p->coeff = atol(coeffStr);
    p->expon = atol(exponStr);
    // important, car la valeur par défaut n'est pas forcément NULL !!!
    prev_p->nextTerm = p;
    printf("coeff. %d> ", size);
    scanf("%s", coeffStr);
  }
  p->nextTerm = NULL;
}
```

- *Normalisation du polynôme saisi*

Spécification de l'algorithme

```
FONCTION normalizePolynomial(*pp : polynome)
VAR ???
DEBUT
     ???
FIN
```

Implantation C

```
void normalizePolynomial(polynomial* pp)
{
     A VOUS DE JOUER !!
}
```

- *Évaluation du polynôme (instanciation)*

Spécification de l'algorithme

```
FONCTION evalPolynomial(p : polynome, x : réel) : réel
VAR résultat, valeur : réel
DEBUT
  résultat ← 0
  SI (p = NULL) ALORS
    RETOURNER résultat
  FINSI
  résultat ← résultat + (p⟶coeff) * (puissance(x, p⟶exposant))
  TANTQUE (p⟶suivant ≠ NULL) FAIRE
    p ← p⟶suivant
    valeur ← (p⟶coeff) * (puissance(x, p⟶exposant))
    résultat ← résultat+valeur
  FAIT
  RETOURNER résultat
FIN
```

Implantation C

```
long evalPolynomial(polynomial p, long x)
{
  long res = 0;
  if (p == NULL) return res;
  res += (p->coeff) * (powl(x, p->expon));
  printf("first term eval : %ld\n", res);
  while (p->nextTerm != NULL)
  {
    p = p->nextTerm;
    long termVal = (p->coeff) * (powl(x, p->expon));
    printf("next term eval : %ld\n", termVal);
    res += termVal;
  }
  return res;
}
```

- *Affichage du polynôme*

Spécification de l'algorithme

```
FONCTION printPolynomial(p : polynome)
DEBUT
  SI (p = NULL) ALORS
    RETOURNER
  FINSI
  printTerm(p)
  TANTQUE (p⟶suivant ≠ NULL) FAIRE
    p ⟵  p⟶suivant
    printTerm(p)
  FAIT
FIN
```

Implantation C

```
void printPolynomial(polynomial p)
{
  if (p == NULL) return;
  printf("[");
  printTerm(p);
  while (p->nextTerm != NULL)
  {
    p = p->nextTerm;
    printTerm(p);
  }
  printf("]");
}
```

- *Affichage d'un terme du polynôme*

Spécification de l'algorithme

```
FONCTION printTerm(p : polynome)
DEBUT
  SI (p = NULL) ALORS
    RETOURNER
  FINSI
  SI (p⟶coeff = 0) ALORS
    RETOURNER
  FINSI
  SI (p⟶coeff = 1) ALORS
    SI (p⟶exposant = 0) ALORS
      AFFICHER("+1")
    SINON
```

```
      SI (p⟶exposant = 1) ALORS
        AFFICHER("+x")
      SINON
        AFFICHER("+x^", p⟶exposant)
      FINSI
    FINSI
    RETOURNER
  FINSI
  SI (p⟶coeff = -1) ALORS
    SI (p⟶exposant = 0) ALORS
      AFFICHER("-1")
    SINON
      SI (p⟶exposant = 1) ALORS
        AFFICHER("-x")
      SINON
        AFFICHER("-x^", p⟶exposant)
      FINSI
    RETOURNER
  FINSI
  SI (p⟶coeff < 0) ALORS
    SI (p⟶exposant = 0) ALORS
      AFFICHER(p⟶coeff)
    SINON
      SI (p⟶exposant = 1) ALORS
        AFFICHER(p⟶coeff)
      SINON
        AFFICHER(p⟶coeff, "x^", p⟶exposant)
      FINSI
    FINSI
  SINON
    SI (p⟶exposant = 0) ALORS
      AFFICHER("+", p⟶coeff)
    SINON
      SI (p⟶exposant = 1) ALORS
        AFFICHER("+", p⟶coeff, "x")
      SINON
        AFFICHER("+", p⟶coeff, "x^", p⟶exposant)
      FINSI
    FINSI
  FINSI
FIN
```

Implantation C

```
void printTerm(polynomial p)
{
  if (p == NULL) return;
  if (p->coeff == 0) return;
  if (p->coeff == 1) {
    if (p->expon == 0) printf("+1");
    else if (p->expon == 1) printf("+X");
    else printf("+X^(%ld)", p->expon);
    return;
  }
  if (p->coeff == -1) {
    if (p->expon == 0) printf("-1");
    else if (p->expon == 1) printf("-X");
    else printf("-X^(%ld)", p->expon);
    return;
  }
  if (p->coeff < 0)
  {
    if (p->expon == 0) printf("%ld", p->coeff);
    else if (p->expon == 1) printf("%ldX", p->coeff);
    else printf("%ldX^(%ld)", p->coeff, p->expon);
  }
  else
  {
    if (p->expon == 0) printf("+%ld", p->coeff);
    else if (p->expon == 1) printf("+%ldX", p->coeff);
    else printf("+%ldX^(%ld)", p->coeff, p->expon);
  }
}
```

- *Fonction principale (saisie et évaluations de polynômes)*

```
void testPolynomial()
{
  printf("\n\t\t>> TD 1.10 Saisie et evaluation de polynomes <<\n\n");
  int i = 1;
  char input[10];
  polynomial newPolynomial;
  printf("\nSouhaitez-vous creer un nouveau polynome depuis
          la ligne de commande (OUI/*) ?\n\n");
  printf("newPolynomial %d> ", i);
  scanf("%s", input);
  printf("\n");
```

```
  while (strcmp("OUI", input) == 0)
  {
    recordPolynomial(&newPolynomial);
    printf("Le polynome saisi est : ");
    printPolynomial(newPolynomial);
    printf("\n");
    testEvalPolynomial(newPolynomial);
    printf("\nSouhaitez-vous creer un nouveau polynome depuis
              la ligne de commande (OUI/*) ?\n\n");
    printf("newPolynomial %d> ", i);
    scanf("%s", input);
    printf("\n");
  }
}
```

- *Fonction principale déléguée (évaluations d'un polynôme)*

```
void testEvalPolynomial(polynomial p)
{
  int i = 1;
  char input[10];
  long x;
  printf("\nSouhaitez-vous evaluer le polynome pour une nouvelle
            valeur de x (OUI/*) ?\n\n");
  printf("eval %d> ", i);
  scanf("%s", input);
  printf("\n");
  while (strcmp("OUI", input) == 0)
  {
    printf("x %d> ", i);
    scanf("%s", input);
    printf("\n");
    x = atol(input);
    long result = evalPolynomial(p, x);
    printPolynomial(p);
    printf("(x = %ld) -> [%ld]\n", x, result);
    i++;
    printf("\nSouhaitez-vous evaluer le polynome pour une nouvelle
              valeur de x (OUI/*) ?\n\n");
    printf("eval %d> ", i);
    scanf("%s", input);
    printf("\n");
  }
}
```