Correction
Pointeurs et références

Exercice 1 : tranches maximales

```cpp
#include <iostream>
#include <vector>
#include <limits>
using namespace std;

typedef int Valeur;
typedef vector<Valeur> Liste;

typedef const Valeur* Position;

struct Somme_sous_liste {
  Position debut;
  Position fin;
  Valeur somme;
};

// PROTOTYPES
Somme_sous_liste tranche_max_1(Liste const& a);
Somme_sous_liste tranche_max_2(Liste const& a);
Somme_sous_liste tranche_max_3(Liste const& a);
void affiche(Somme_sous_liste const& s);

// ----------------------------------------------------------------------
int main()
{
  vector<Liste> seq({
      { 11, 13, -4, 3, -26, 7, -13, 25, -2, 17, 5, -8, 1 },
      {},
      { -3, -4, -1, -2, -3 },
      { -1, -4, -3, -2, -3 },
      { 3, 4, 1, 2, -3 },
      { 3, 4, 1, 2, 3 },
      { 3, -1, -1, -1, 5 },
      { -5, -4, 1, 1, 1 }
  });

 for (size_t i(0); i < seq.size(); ++i) {
   cout << "Test " << i+1 << " :" << endl;
   affiche(tranche_max_1(seq[i]));
   affiche(tranche_max_2(seq[i]));
   affiche(tranche_max_3(seq[i]));
   cout << endl;
 }

  return 0;
}

/* ======================================================================
 * Recherche la sous-liste de somme maximale dans une liste.
 *   L'algorithme utilisé ici est l'algorithe naïf de complexité cubique.
 * Entree : la liste où chercher.
 * Sortie : la sous-liste de somme maximale.
 * ====================================================================== */
Somme_sous_liste tranche_max_1(Liste const& a)
{
  Somme_sous_liste rep = { nullptr, nullptr, numeric_limits<Valeur>::min() };
  if (not a.empty()) {
    for (Position debut(&a.front()); debut <= &a.back(); ++debut) {
      for (Position fin(debut); fin <= &a.back(); ++fin) {
```

```
        Valeur somme(0);
        for (Position p(debut); p <= fin; ++p) {
          somme += *p;
        }
        if (somme > rep.somme) {
          rep.debut = debut;
          rep.fin   = fin;
          rep.somme = somme;
        }
      }
    }
  }
  return rep;
}

/* ========================================================================
 * Recherche la sous-liste de somme maximale dans une liste.
 *   L'algorithme utilisé ici est l'algorithe naïf de complexité quadratique.
 * Entree : la liste où chercher.
 * Sortie : la sous-liste de somme maximale.
 * ======================================================================== */
Somme_sous_liste tranche_max_2(Liste const& a)
{
  Somme_sous_liste rep = { nullptr, nullptr, numeric_limits<Valeur>::min() };
  if (not a.empty()) {
    for (Position debut(&a.front()); debut <= &a.back(); ++debut) {
      Valeur somme(0);
      for (Position fin(debut); fin <= &a.back(); ++fin) {
        somme += *fin;
        if (somme > rep.somme) {
          rep.debut = debut;
          rep.fin   = fin;
          rep.somme = somme;
        }
      }
    }
  }
  return rep;
}

/* ========================================================================
 * Recherche la sous-liste de somme maximale dans une liste.
 *   L'algorithme utilisé ici est l'algorithe linéaire.
 * Entree : la liste où chercher.
 * Sortie : la sous-liste de somme maximale.
 * ======================================================================== */
Somme_sous_liste tranche_max_3(Liste const& a)
{
  Somme_sous_liste rep = { nullptr, nullptr, numeric_limits<Valeur>::min() };
  if (not a.empty()) {
    Valeur somme(0);
    Position debut(&a.front());
    for (Position fin(&a.front()); fin <= &a.back(); ++fin) {
      somme += *fin;

      if (somme > rep.somme) {
        rep.debut = debut;
        rep.fin   = fin;
        rep.somme = somme;
      }
      if (somme <= 0) {
        debut = fin + 1;
        somme = 0;
      }
    }
  }
  return rep;
}
```

```
// ---------------------------------------------------------------------
void affiche(Somme_sous_liste const& s)
{
  cout << "La tranche maximale est ";
  if ((s.debut == nullptr) or (s.fin == nullptr) or (s.fin < s.debut)) {
    cout << "vide";
  } else {
    for (Position p(s.debut);  p <= s.fin; ++p)
      cout << *p << ", ";
    cout << endl << "de somme " << s.somme;
  }
  cout << endl;
}
```

Exercice 2 : dérivées

```
#include <iostream>
using namespace std;

struct Arbre_;
typedef Arbre_* Arbre;

struct Arbre_ {
  char  valeur;
  Arbre gauche;
  Arbre droite;

  /* Nécessaire si pointages multiples possibles, i.e. si le même sous-arbre
     est utilisé par différents noeuds pères, par exemple dans des arbres
     différents.
     La version C++11 "pro" utiliserait des shared_ptr.
     Une alternative serait d'utiliser des copies profondes dans creer_arbre au
     lieu de faire pointer tout le monde au même endroit, mais cette
     dernière solution est plus lourde.
  */
  unsigned int nb_acces;
};

Arbre creer_feuille(char op)
{
  return new Arbre_({ op, nullptr, nullptr, 0 });
}

Arbre creer_arbre(char op, Arbre gauche, Arbre droite)
{
  if (gauche != nullptr) ++(gauche->nb_acces);
  if (droite != nullptr) ++(droite->nb_acces);
  return new Arbre_({ op, gauche, droite, 0 });
}

void libere_arbre(Arbre& a)
{
  if (a != nullptr) {
    if (a->nb_acces > 0) { --(a->nb_acces); }
    if (a->nb_acces == 0) {
      libere_arbre(a->gauche);
      libere_arbre(a->droite);
      delete a;
      a = nullptr;
    }
  }
}

void affiche_arbre(Arbre a)
{
```

```
  if (a->gauche != nullptr) {
    cout << '(';
    affiche_arbre(a->gauche);
    cout << ' ';
  }
  cout << a->valeur;
  if (a->droite != nullptr) {
    cout << ' ';
    affiche_arbre(a->droite);
    cout << ')';
  }
}

Arbre derive(Arbre arbre, char variable)
{
  if (arbre == nullptr) return nullptr;

  /* Pre-calcul des dérivées à gauche et à droite */
  Arbre derive_gauche(derive(arbre->gauche, variable));
  Arbre derive_droite(derive(arbre->droite, variable));

  Arbre resultat;
  switch (arbre->valeur) {
  case '+':
  case '-':
    resultat = creer_arbre(arbre->valeur, derive_gauche, derive_droite);
    break;

  case '*':
    resultat =
      creer_arbre('+',
                  creer_arbre('*', derive_gauche, arbre->droite),
                  creer_arbre('*', arbre->gauche, derive_droite)
                  );
    break;

  case '/':
    resultat =
      creer_arbre('/',
                  creer_arbre('-',
                              creer_arbre('*', derive_gauche, arbre->droite),
                              creer_arbre('*', arbre->gauche, derive_droite)),
                  creer_arbre('*', arbre->droite, arbre->droite)
                  );
    break;

  case '^':
    resultat =
      creer_arbre('*',
                  creer_arbre('*', arbre->droite, derive_gauche),
                  creer_arbre('^', arbre->gauche,
                              creer_arbre('-',
                                          arbre->droite,
                                          creer_feuille('1')))
                  );
    libere_arbre(derive_droite); // on ne fait pas de f(x)^g(x) ;-)
    break;

  default:
    if (arbre->valeur == variable)
      resultat = creer_feuille('1');
    else
      resultat = creer_feuille('0');
  }

  return resultat;
}

int main()
```

```
{
  Arbre a(creer_feuille('a'));
  Arbre b(creer_feuille('b'));
  Arbre x(creer_feuille('x'));

  Arbre xpa(creer_arbre('+', x, a));
  Arbre xpb(creer_arbre('+', x, b));
  Arbre x2 (creer_arbre('*', x, x));

  Arbre expr2(creer_arbre('*', xpa, xpb));
  Arbre expr3(creer_arbre('+', creer_arbre('*', x, x2),
                               creer_arbre('*', a, x )));
  Arbre expr4(creer_arbre('/', creer_arbre('^', x, a),
                               creer_arbre('+', x2,
                                           creer_arbre('*', b, x))));
  affiche_arbre(xpa);   cout << endl;
  affiche_arbre(expr2); cout << endl;
  affiche_arbre(expr3); cout << endl;
  affiche_arbre(expr4); cout << endl;

  /* Calcul des derivees */
  Arbre d_expr1(derive(xpa,  'x'));
  Arbre d_expr2(derive(expr2,'x'));
  Arbre d_expr3(derive(expr3,'x'));
  Arbre d_expr4(derive(expr4,'x'));

  /* Afficher les derivees */
  cout << "d( xpa )/dx = "; affiche_arbre(d_expr1); cout << endl;
  cout << "d(expr2)/dx = "; affiche_arbre(d_expr2); cout << endl;
  cout << "d(expr3)/dx = "; affiche_arbre(d_expr3); cout << endl;
  cout << "d(expr4)/dx = "; affiche_arbre(d_expr4); cout << endl;

  /* liberation de la memoire */
  libere_arbre(d_expr1);
  libere_arbre(d_expr2);
  libere_arbre(d_expr3);
  libere_arbre(d_expr4);
  libere_arbre(expr2);
  libere_arbre(expr3);
  libere_arbre(expr4);

  return 0;
}
```