1 Classe de de polynômes

Le but est d'implémenter une classe `Polynome` pour faire du calcul formel sur les polynômes (d'une variable). Voici par exemple ce que l'on aimerait faire avec la classe polynôme :

```
#include <iostream>
#include "polynome.hpp"

using namespace std;
int main(void) {
    int coeff[] = { 2, -1, 4, 0, 2 };
    unsigned degre[] = { 0, 3, 2, 5, 6 };
    Polynome P(coeff, degre, 5);
    cout << "P:\t" << P << endl;
    Polynome Q(4, 2);
    cout << "Q:\t" << Q << endl;
    cout << "Q+1:\t" << Q+1 << endl;
    cout << "P+Q:\t" << P+Q << endl;
    cout << "P-Q:\t" << P-Q << endl;
    cout << "P*Q:\t" << P*Q << endl;
    return 0;
}
```

Ce programme compilé et executé doit donner le résultat suivant :

```
    P:  2 + 4x^2 + -1x^3 + 2x^6
    Q:  4x^2
    Q+1:    1 + 4x^2
    P+Q:    2 + 8x^2 + -1x^3 + 2x^6
    P-Q:    2 + -1x^3 + 2x^6
    P*Q:    8x^2 + 16x^4 + -4x^5 + 8x^8
```

Dans une première étape on code une classe `Monome` (qui n'est pas destiné à être utilisé par l'utilisateur). Puis on codera ensuite une classe `Polynome` qui sera amie (`friend`) avec la classe `Monome` pour l'utiliser plus simplement.

Préliminaire : classe `Monome`

Dans la suite, on considère un monôme d'une variable X de la forme cX^d avec $c \in \mathbb{Z}$ et $d \in \mathbb{N}$.

```
class Monome {
    friend class Polynome;
    public:
        Monome(int coeff = 0, unsigned degre = 0)________
        double operator()_________
        Monome operator-()_________
    private:
        int coeff;
        unsigned degre;
};
```

1.1. *Ecrire et compléter dans un fichier* `monome.hpp` *la déclaration de la classe* `Monome` *qui contient deux variables privées : un entier non signé* `degre` *codant le degré d du monôme, et un entier* `coeff` *codant c.*

1.2. *Ajouter dans cette classe la décalaration des fonctions amies permettant de surcharger les opérateurs suivants :*

comparaisons `==`, `<` *(l'ordre est donné par le degré)*

arithmétiques `+`, `-`, `*` *(avec message d'erreur si nécessaire)*

injection `<<` *(qui produit un affichage similaire à l'exemple)*

Ecrire la définition de ces fonctions dans le fichier `monome.cpp`.

1.3. *Compiler et tester rapidement votre classe en écrivant un petit fichier* `test_monome.cpp`. *Il faut que ce petit programme compile sans erreurs.*

Classe `Polynome` utilisant le conteneur `vector`

On rappelle que le conteneur `vector` du `C++` est défini dans le fichier d'en-tête `vector.h`. C'est un conteneur générique et on va utiliser dans la suite la spécification `vector<Monome>` pour représenter un polynôme comme un ensemble ordonné de monôme. En interne, ce vecteur devra toujours être trié (suivant l'ordre donné par l'opérateur `<` de la classe `Monome`).

1.4. *Définir dans le fichier d'en-tête* `polynome.hpp` *les types suivants pour facilier l'écriture des programmes.*

```
typedef std::vector<Monome> MonoVec;
typedef std::vector<Monome>::iterator MonoVec_It;
typedef std::vector<Monome>::const_iterator MonoVec_CIt;
```

On rappelle qu'un `const_iterator` *est un itérateur sur valeur constante qui est utilisé lorsqu'on ne doit/veut pas modifier l'objet auquel appartient l'itérateur.*

1.5. *Ecrire la déclaration de la classe* `Polynome` *qui doit posséder les 3 constructeurs suivants :*

```
Polynome(int coeff = 0, unsigned degre = 0);
Polynome(int coeff[], unsigned degre[], int n);
Polynome(const MonoVec &data0);
```

Ces constructeurs doivent initialiser la variable privée `data` *de type* `MonoVec`*. Ce vecteur doit être trié en utilisant la fonction* `sort` *définie dans* `algorithm.h` *de prototype :*

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

1.6. *Ecrire une fonction membre* `simplifie` *qui simplifie le vecteur* `data` *de la façon suivante :*

- *on retire tout monôme de coefficient 0*
- *on regroupe tous les monômes de même degré en un seul (en additionnant les coefficients).*

Vous pouvez utiliser la méthode `erase` *de la classe* `vector` *dont le prototype est :*

```
iterator erase (iterator position);
```

qui retire l'élément à la position donné par l'itérateur `position` *et qui renvoie l'itérateur sur le nouvel élément courant).*

1.7. *Surcharger les mêmes opérateurs que ceux de la classe* `Monome`*. En particulier les opérateurs arithmétiques doivent renvoyer un* `Polyome` *dont le vecteur* `data` *est trié et simplifié.*

1.8. *Executer le fichier test donné au début de la feuille.*

Classe `Polynome` utilisant le conteneur `list`

Modifier la classe `Polynome` pour utiliser le conteneur `list` à la place du conteneur `vector` (si on doit utiliser cette classe avec des polynômes dont le nombre de monômes est important il est préférable d'utiliser le conteneur `list`).

2 Vecteurs et programmation dynamique

Ce que l'on veut

Le but est de programmer deux classes génériques `Vect<T>` et `SubVect<T>` qui permettent la compilation du code suivant :

```
#include <iostream>
#include "vect_dyn.hpp"

using namespace std;
int main() {
    Vect<double> v(10);
    for (int i = 1; i <= 10; ++i)
        v(i) = i;
    cout << "v:\t" << v << endl;
    SubVect<double> w(v, 1, 5, 2);
    cout << "w:\t" << w << endl;
    SubVect<double> z(v, 2, 3, 3);
    cout << "z:\t" << z << endl;
    SubVect<double> u(w, 1, 3, 2);
    cout << "u:\t" << u << endl;
    u.init(44);
    cout << "u:\t" << u << endl;
    cout << "v:\t" << v << endl;
    cout << "norme de v:\t" << v.norme() << endl;
    return 0;
}
```

Il est important de noter que l'accès au `i`-ème élément d'un vecteur se fait ici par l'opérateur `()` avec un indice `i` (allant de 1 à la taille du vecteur).
L'execution de ce code doit donner le résultat suivant :

```
v:  1   2   3   4   5   6   7   8   9   10
w:  1   3   5   7   9
z:  2   5   8
u:  1   5   9
u:  44  44  44
v:  44  2   3   4   44  6   7   8   44  10
norme de v: 78.0128
```

Dans un premier temps, pour simplifier l'écriture et le test des classes on suppose que `T = double`, c'est à dire que les classes `Vect` et `SubVect` ne sont pas génériques. Les classes `Vect` et `SubVect` seront des classes dérivées d'une classe abstraite `AbsVect`.

La classe abstraite `AbsVect`

On rappelle qu'une classe abstraite est une classe contenant au moins une fonction virtuelle pure. Une fonction virtuelle pure doit obligatoirement être définie dans les classes filles (ou classes héritées). La syntaxe pour déclarer une fonction virtuelle pure est la suivante : `virtual` *prototype_fonction* `= 0`.

2.1. *Ecrire une classe* `AbsVect` *qui contient*
- *un champ protégé* `n` *dans lequel sera stocké la taille du vecteur*
- *un constructeur initialisant le champ* `n`
- *une fonction membre* `taille` *renvoyant la taille du vecteur*
- *2 fonctions virtuelles pures* `operator()` *déclarées de la façon suivante :*

```
virtual double operator()(unsigned i) const = 0;
virtual double & operator()(unsigned i) = 0;
```

2.2. *Ajouter à cette classe les 2 fonctions membres suivantes que vous définirez en dehors de la classe (mais dans le même fichier* `vec_dyn.hpp`*) :*
- `remplit` *fonction qui initialise*

— `norme` *qui renvoie la norme quadratique du vecteur.*

Ces deux fonctions doivent *utiliser un appel explicite à la fonction* `operator()` *pour accéder à un élément (pensez à utiliser le pointeur* `this` *qui pointe sur l'objet courant).*

2.3. *Surcharger l'opérateur d'injection* `<<` *pour cette classe.*

Les classes filles `Vect` et `SubVect`

2.4. *Ecrire la classe* `Vect` *dérivée de* `AbsVect` *qui contient en champ privé un pointeur* `data` *sur* `double`*. Ce pointeur sera utilisée pour stocker les données du vecteur.*
Quelles fonctions membres doit-on obligatoirement définir ?
Définir les 2 fonctions `operator()`*.*

2.5. *Ecrire la classe* `SubVect` *dérivée de* `AbsVect` *qui contient comme membres privés*
— *une référence vers un objet* `AbsVect`
— *deux entiers* `debut` *et* `saut` *utilisés dans les fonctions d'accès* `operator()`
et dont le constructeur est compatible avec l'exemple donné en introduction.

2.6. *Tester vos classes en écrivant un programme de test similaire à celui donné en introduction.*

Rendre les classes génériques

2.7. *(facultatif) Modifier vos fonctions et vos classes de façon à obtenir les classes génériques* `Vect<T>` *et* `SubVect<T>`*.*

3 Matrices et programmation dynamique

Ce que l'on veut

Le but est de programmer 3 classes génériques `Matrice<T>`, `Ligne<T>` et `Colonne<T>` qui permettent la compilation du code suivant :

```
#include <iostream>
#include "matrice.hpp"

using namespace std;
int main() {
    Matrice<int> M(3, 3);
    for (int i = 1; i <= M.nb_lignes(); ++i)
        for (int j = 1; j <= M.nb_cols(); ++j)
            M(i, j) = i*j + i;
    cout << "Matrice M:\n" << M << endl;

    cout << "Ligne 1:\t" << Ligne<int>(M, 1) << endl;
    cout << "Colonne 3:\t" << Colonne<int>(M, 3) << endl;
    cout << endl;

    Matrice<int> M2(3, 3);
    for (int i = 1; i <= M.nb_lignes(); ++i)
        for (int j = 1; j <= M.nb_cols(); ++j)
            M2(i, j) = produit_scalaire(Ligne<int>(M,i), Colonne<int>(M,j));
    cout << "Matrice M2:\n" << M2 << endl;

    return 0;
}
```

Il est important de noter que l'accès à l'élément de la `i`-ème ligne et la `j`-ème colonne d'une matrice se fait ici par l'opérateur `()` avec des indices `i` et `j` plus grand que 1.
L'execution de ce code doit donner le résultat suivant :

```
Matrice M:
2   3   4
4   6   8
6   9   12

Ligne 1:    2   3   4
Colonne 3:  4   8   12

Matrice M2:
40  60  80
80  120 160
120 180 240
```

Dans un premier temps, pour simplifier l'écriture et le test des classes on suppose que `T = int`, c'est à dire que les classes ne sont pas génériques.

La classe `Matrice`

3.1. *Ecrire une classe* `Matrice` *qui contient*
- *3 champs privés : 2 entiers pour la taille de la matrice et un pointeur sur le contenu (on stocke le contenu de la matrice $n \times m$ dans un tableau dynamique de nm cases mémoires).*
- *2 constructeurs : 1 compatible avec l'exemple et le constructeur de clonage*
- *un opérateur d'affectation*

Faut-il définir le destructeur ? Si oui, faites-le.

3.2. *Définir les deux versions (lecture/écriture) de l'opérateur* `()` *compatible avec l'exemple. Définir les fonctions membres d'accès aux dimensions de la matrice.*

3.3. *Surcharger l'opérateur d'injection* `<<` *pour cette classe.*

Les classes `Ligne` et `Colonne` dérivée de `AbsVect`

On rappelle qu'une classe abstraite est une classe contenant au moins une fonction virtuelle pure. Une fonction virtuelle pure doit obligatoirement être définie dans les classes filles (ou classes héritées). La syntaxe pour déclarer une fonction virtuelle pure est la suivante : `virtual` *prototype_fonction*`= 0`.

3.4. *Définir une classe abstraite* `AbsVect` *qui contient 2 fonctions virtuelles pures* `operator()` *déclarées de la façon suivante :*

```
    virtual T operator()(int i) const = 0;
2   virtual T & operator()(int i) = 0;
```

et un constructeur qui initialise les 3 champs protégés : 2 entiers (l'un stockant un indice et un autre une taille de vecteur) et 1 référence (non constante) sur un objet `Matrice`*.*

3.5. *Ecrire les classes* `Ligne` *et* `Colonne` *comme des classes dérivées de la classe abstraite* `AbsVect`*. Le comportement des opérateur* `operator()` *doit etre compatible avec celui donné en exemple.*

3.6. *Surcharger l'opérateur d'injection* `<<` *pour des objets d'adresse compatible avec l'adresse d'un* `AbsVect`*.*
Ecrire la fonction `produit_scalaire`*.*

3.7. *Tester vos classes en écrivant un programme de test similaire à celui donné en introduction.*

Tout rendre générique

3.8. *(facultatif) Modifier vos fonctions et vos classes de façon à obtenir les classes génériques demandées.*

4 Matrices diverses et programmation dynamique

Ce que l'on veut

Le but est de programmer 3 classes génériques `Matrice<T>`, `MatriceDiag<T>` et `MatriceTriangInf<T>` qui permettent la compilation du code suivant :

```
#include <iostream>
#include "mat.hpp"

using namespace std;
int main() {
    Matrice<int> B(2, 5);
    B(2, 3) = 9;
    cout << "Matrice B:\n" << B << endl;

    int tab[] = { 1, 4, 5, 2, 6, 7, 3, 1, 8 };
    Matrice<int> M(3, 3, tab);
    cout << "Matrice M:\n" << M << endl;

    MatriceDiag<int> D(3, tab);
    cout << "Matrice D:\n" << D << endl;

    MatriceTriangInf<int> T(3, tab);
    cout << "Matrice T:\n" << T << endl;

    return 0;
}
```

Il est important de noter que l'accès à l'élément de la `i`-ème ligne et la `j`-ème colonne d'une matrice se fait ici par l'opérateur `()` avec des indices `i` et `j` plus grand que 1.
L'execution de ce code doit donner le résultat suivant :

```
    Matrice B:
    0   0   0   0   0
    0   0   9   0   0

    Matrice M:
    1   2   3
    4   6   1
    5   7   8

    Matrice D:
    1   0   0
    0   4   0
    0   0   5

    Matrice T:
    1   0   0
    4   2   0
    5   6   7
```

Dans un premier temps, pour simplifier l'écriture et le test des classes on suppose que `T = int`, c'est à dire que les classes ne sont pas génériques.

La classe `Matrice`

4.1. *Ecrire une classe* `Matrice` *qui contient*
- *4 champs protégés : 2 entiers pour les dimensions de la matrice, 1 entier pour la taille du contenu, et un pointeur sur le contenu (on stocke le contenu de la matrice $n \times m$ dans un tableau dynamique de nm cases mémoires).*
- *2 constructeurs compatibles avec l'exemple*
- *2 versions (lecture/écriture) de l'opérateur* `()` *compatible avec l'exemple*
- *les fonctions membres d'accès aux dimensions de la matrice*

4.2. *Définir le constructeur de clonage et l'opérateur d'affectation en utilisant l'opérateur* `()` *(on rappelle que le pointeur* `this` *pointe sur l'objet courant).*
Faut-il définit le destructeur ? Si oui, faites-le.

4.3. *Surcharger l'opérateur d'injection* `<<` *pour cette classe.*

Les classes `MatriceDiag` et `MatriceTriangInf` dérivées de `Matrice`

On rappelle que le polymorphisme dynamique est possible grâce à l'héritage et aux fonctions virtuelles. On se propose d'écrire les classes `MatriceDiag` et `MatriceTriangInf` comme des classes filles de la classe `Matrice`. Pour cela, on modifie légèrement le code de la classe `Matrice` en définissant les opérateurs d'accès `operator()` comme virtuels (mot-clé `virtual` devant ces fonctions).

4.4. *Déclarer les fonctions* `operator()` *de la classe* `Matrice` *comme virtuelles.*
Ajouter les 2 constructeurs suivants (dont le contenu est à définir) à la classe `Matrice` *:*

```
Matrice(unsigned n, unsigned m, unsigned taille_data);
Matrice(unsigned n, unsigned m, unsigned taille_data, T const tab[]);
```

Ces constructeurs permettent de donner une taille `taille_data` *au contenu différente de nm (lire la suite pour comprendre l'intérêt).*

La classe `MatriceDiag` doit dériver de la classe `Matrice`, et pour des soucis d'efficacité et d'optimisation, on souhaite que seulement n cases mémoires soient utilisées pour stocker une matrice diagonale (donc carrée) de dimension $n \times n$.

4.5. *Ecrire la classe* `MatriceDiag` *comme classe fille de la classe* `Matrice` *et définir ses constructeurs et ses opérateurs d'accès* `operator()`*.*

De même, la classe `MatriceTriangInf` doit dériver de la classe `Matrice` et son stockage doit se faire en utilisant uniquement $n(n+1)/2$ cases mémoires.

4.6. *Ecrire la classe* `MatriceTriangInf` *comme classe fille de la classe* `Matrice` *et définir ses constructeurs et ses opérateurs d'accès* `operator()`*.*

4.7. *Tester vos classes en écrivant un programme de test similaire à celui donné en introduction.*

Tout rendre générique

4.8. *(facultatif) Modifier vos fonctions et vos classes de façon à obtenir les classes génériques demandées.*

5 Suites entières et génériques

Classe abstraite `SuiteN`

On considère la classe abstraite `SuiteN` qui permettra une programmation aisée des suites à valeurs entières. Le code suivant doit être défini dans le fichier `suite.hpp` :

```
class SuiteN {
    public:
        SuiteN(int etat_initial) : etat(etat_initial), n_iter(0) {};
        SuiteN & operator++() {
            change_etat();
            ++n_iter;
            return *this;
        };
        virtual void change_etat() = 0;
    protected:
        int n_iter;
        int etat;
};
```

Le champ `n_iter`, initialisé à 0 par le constructeur, représente le numéro de l'itération et le champ (générique) `etat` représente la valeur de la suite à l'itération `n_iter`. L'opérateur d'incrément surchargé ici est le pré-incrément (le post-incrément n'est pas possible car il s'agit d'une classe abstraite).
La méthode virtuelle pure `change_etat()` doit être modifiée dans toutes les classes dérivées de `SuiteN`. La relation de récurrence de la suite doit être codée dans cette méthode.

5.1. *Ecrire cette classe dans le fichier* `suite.hpp` *et ajouter un accesseur* `get_etat()` *qui renvoie la valeur du champ* `etat` *et un accesseur* `get_n()` *qui renvoie la valeur du champ* `n_iter`*.*

5.2. *Ecrire une méthode* `reinit()` *qui réinitialise la suite à sa valeur initiale et remet* `n_iter` *à 0. Vous pouvez ajouter des champs à la classe* `SuiteN`*.*

5.3. *Surcharger l'opérateur d'*affectation `+=` *qui prend pour argument un entier* `k` *et qui effectue* `k` *itérations de la suite.*

5.4. *Surcharger l'opérateur d'*injection `<<` *qui permet d'envoyer la valeur des champs* `n_iter` *et* `etat` *vers un flux de sortie. L'affichage doit être de la forme*

```
n_iter:    etat
```

Suite linéaire

On se propose de tester cette classe abstraite en écrivant une classe `Lineaire` dérivée de `SuiteN` permenant de manipuler les suites récurrentes linéaires de la forme

$$u_{n+1} = au_n + b, \quad u_0 \in \mathbb{N}$$

où a et b sont des paramètres entiers positifs fixés.
On considère l'extrait de code suivant qui affiche les 5 premières itérations de la suite linéaire de condition initiale $u_0 = 3$ et de paramètres $a = 2$ et $b = 1$.

```
Lineaire L(2,1,3);
for (int i = 0; i <= 4; ++i) {
    cout << L << endl;
    ++L;
}
L.reinit();
cout << (L += 3) << endl;
```

Après compilation et execution on doit obtenir le résultat suivant :

```
0:  3
1:  7
2:  15
3:  31
4:  63
3:  31
```

5.5. *Ecrire la classe* `Lineaire` *qui permet de reproduire cet exemple. Il faut donc :*
- *Ecrire un constructeur qui appelle explicitement le constructeur de la classe mère. On rappelle que cela doit se faire au début de la liste d'initialisation.*
- *Définir la fonction* `change_etat()` *(qui devra être qualifiée de* `private`*).*

Compiler et tester cette classe. Tester aussi l'opérateur `+=`*.*

Suite de Fibonacci

On considère maintenant la suite de Fibonacci $(v_n)_{n\geq -1}$ définie par la récurrence suivante :

$$\forall n \geq 0, \quad v_{n+1} = v_n + v_{n-1}, \quad v_0 = 0, \text{ et } v_{-1} = 1.$$

La formule de récurrence fait donc intervenir les deux termes précédents de la suite : *chaque terme de cette suite est la somme des deux termes précédents.*

5.6. *Ecrire une classe* `Fibonacci` *dérivée de* `SuiteN` *qui permet d'implémenter cette suite. Vous pourrez utiliser le champ* `etat` *de la classe mère et un champ* `etat_precedent` *qui sera qualifié de* `private` *dans la classe* `Fibonacci`*.*

5.7. *La méthode* `reinit()` *de la classe mère est-elle toujours valide dans la classe fille* `Fibonacci` *? Proposer une solution.*

5.8. *Ecrire une classe* `FSuiteN` *dérivée de* `SuiteN` *qui permette de manipuler des suites de la forme* $(f(x_n)))_{n\geq 0}$ *où* f *est une fonction de* $\mathbb{N}$ *dans* $\mathbb{N}$ *et* $(x_n)_{n\geq 0}$ *une suite à valeurs entières. Le constructeur prendra donc 2 arguments :*
- *une référence sur un objet* `SuiteN` *(qui représente* $(x_n)_{n\geq 0}$*),*
- *un pointeur de fonction (pour* f*), ou mieux une référence sur objet fonctionnel (à déterminer).*

Passage au générique

On souhaite maintenant étendre la classe abstraite `SuiteN` à une classe abstraite générique `Suite<T>` où `T` est un paramètre template. Les suites considérées pourront alors être à valeurs dans un espace général dont les objets sont codés par le type `T` du template.

5.9. *Toujours dans le fichier* `suite.hpp` *écrire une classe générique* `Suite<T>` *qui a les mêmes fonctionnalités (méthodes et champs) que la classe* `SuiteN`*. Si vous définissez l'opérateur d'injection* `<<` *comme méthode amie, il faut que cette fonction soit déclarer comme générique avec un nouveau paramètre template (différent de* `T`*).*

5.10. *Définir une nouvelle classe* `Fibonacci2` *qui se comporte comme la classe* `Fibonacci` *mais qui est héritée de la classe abstraite générique* `Suite<T>` *instanciée avec le type* `std::vector<int>`*, c'est à dire* `Suite< std::vector<int> >`*. Un état de cette suite sera donc un vecteur de taille 2 représentant les 2 dernières valeurs de la suite de Fibonacci. Compiler et tester cette classe.*

5.11. *Ecrire une classe générique* `FSuite<T>` *dérivée de* `Suite<T>` *qui permette de manipuler des suites de la forme* $(F(x_n)))_{n\geq 0}$ *où* $(x_n)_{n\geq 0}$ *une suite à valeurs dans un espace général* $\mathbb{T}$ *(dont les objets sont codés par un type générique) et* F *est une fonction à valeurs dans un espace général* $\mathbb{S}$ *(qui peut être différent de* $\mathbb{T}$*).*
Le constructeur prendra donc 2 arguments :
- *une référence sur un objet* `Suite<T>` *(qui représente* $(x_n)_{n\geq 0}$*),*
- *une référence sur objet fonctionnel.*

6 Suites aléatoires

Préambule : variales aléatoires

Soit l'objet fonctionnel (ou foncteur) `Bernoulli` suivant

```
struct Bernoulli : public VarAleaZ {
    Bernoulli(int a = -1, int b = 1, double p = 0.5) : a(a), b(b), p(p) {};
    int operator()() { return etat = (random() < p*RAND_MAX) ? a : b; };
    private:
        int a, b;
        double p;
};
```

Ce foncteur `Bernoulli` dérivé d'une classe abstraite `VarAleaZ` renvoie une réalisation de la loi de Bernoulli généralisée $\mathcal{B}(\{a,b\},p)$ c'est à dire prenant ses valeurs dans $\{a,b\}$ (a et b entiers) et de paramètre $p \in]0,1[$. Voici un exemple d'utilisation :

```
srandom(time(NULL));
Bernoulli B(-1,1,0.5);
for (int i = 0; i < 6; ++i) {
    cout << B() << "\t";
}
cout << endl << B.derniere_realisation() << endl;
```

Un résultat sera par exemple

```
1    -1    -1    1    -1
-1
```

6.1. *D'après l'exemple précédent, écrire la classe abstraite* `VarAleaZ` *dont dérive* `Bernoulli`*. La méthode* `operator()` *devra être virtuelle pure.*

De même, on considère la loi Binomiale généralisée $\mathfrak{B}(n,\{a,b\},p)$. Une variable aléatoire X de loi $\mathfrak{B}(n,\{a,b\},p)$ peut être définie comme la somme de n variables aléatoires indépendantes de loi $\mathfrak{B}(\{a,b\},p)$.

6.2. *Ecrire une classe* `Binomiale` *héritée de* `VarAleaz` *qui a deux champs* `private` *: un objet* `B` *de la classe* `Bernoulli` *et un entier* `n`*. Spécialiser la méthode* `operator()` *pour cette classe.*

Classe abstraite `ChaineMarkovZ`

On considère la classe abstraite `ChaineMarkovZ` suivante qui permettra une implémentation des suites aléatoires de la forme

$$U_{n+1} = f(U_n, X_{n+1}), \quad U_0 \in \mathbb{Z} \tag{*}$$

où $(X_n)_{n+1}$ est une suite de variables aléatoires indépendantes et identiquement distribuées, à valeurs dans $\mathbb{Z}$.

```
class ChaineMarkovZ {
    public:
        ChaineMarkovZ(int valeur_init, VarAleaZ & X)
            : valeur(valeur_init), n_iter(0), X(X) {};
        ChaineMarkovZ & operator++() {
            change_etat(X());
            ++n_iter;
            return *this;
        };
    protected:
        virtual void change_etat(int) = 0;
        int valeur;
        int n_iter;
        VarAleaZ & X;
};
```

Les champs protégés sont les suivants :

- `valeur` : code la valeur de la suite à l'instant n i.e. U_n
- `n_iter` : code la valeur de l'instant i.e n

— `X` : est une référence sur un objet fonctionnel d'une classe héritée de `VarAleaZ`, l'appel de `X()` renvoie une réalisation de X_{n+1}.

L'opérateur d'incrément surchargé ici est le pré-incrément (le post-incrément n'est pas possible car il s'agit d'une classe abstraite). La méthode virtuelle pure `change_etat(int Xnp1)` doit être modifiée dans toutes les classes dérivées de `ChaineMarkovZ`. La relation (*) qui définit la suite aléatoire doit être codée dans cette méthode.

6.3. *Ecrire cette classe dans le fichier* `chaine.hpp` *et ajouter un accesseur* `etat()` *qui renvoie la valeur du champ* `valeur`*, un accesseur* `n()` *qui renvoie la valeur du champ* `n_iter`*, et un accesseur* `last_alea()` *qui renvoie la valeur de la réalisation de* X_{n+1}*.*

6.4. *Ecrire une méthode* `reinit()` *qui réinitialise la suite à sa valeur initiale et remet* `n_iter` *à 0. Vous pouvez ajouter des champs à la classe* `ChaineMarkovZ`*.*

6.5. *Surcharger l'opérateur d'*injection `<<` *qui permet d'envoyer la valeur des champs* `n_iter` *et* `valeur` *vers un flux de sortie. L'affichage doit être de la forme*

```
n_iter:    etat
```

Exemple, processus AR(1)

On se propose de tester cette classe abstraite en écrivant une classe `Autoregressif` dérivée de `ChaineMarkovZ` permenant de manipuler les processus autorégressifs d'ordre 1 sur $\mathbb{Z}$ i.e. les suites de la forme

$$U_{n+1} = \mu + \lambda U_n + X_{n+1}, \quad U_0 \in \mathbb{Z},$$

où $(\mu, \lambda) \in \mathbb{Z}^2$ et $(X_n)_{n+1}$ est une suite de variables aléatoires indépendantes et identiquement distribuées, à valeurs dans $\mathbb{Z}$.
On considère l'extrait de code suivant qui affiche les 10 premières itérations du processus autorégressif de paramètres $\mu = 0$, $\lambda = 1$ et dont la loi de X_{n+1} est $\mathfrak{B}(5, \{-1, 1\}, 0.5)$ et valeur initiale $U_0 = 0$.

```
Binomiale B(5, -1, 1, 0.5);
Autoregressif X(0, 1, B);
for (int i = 0; i < 10; ++i) {
    cout << X << endl;
    ++X;
}
X.reinit();
```

6.6. *Ecrire la classe* `Autoregressif` *qui permet de reproduire cet exemple. Il faut donc :*
— *Ecrire un constructeur qui appelle explicitement le constructeur de la classe mère. On rappelle que cela doit se faire au début de la liste d'initialisation.*
— *Définir la fonction* `change_etat()` *(qui devra être qualifiée de* `private`*).*
Compiler et tester cette classe.

Passage au générique

On souhaite maintenant étendre les classes abstraites `VarAleaZ` et `ChaineMarkovZ` en des classes abstraites génériques `VarAlea<T>` et `ChaineMarkov<T>` où `T` est un paramètre template. Les suites aléatoires considérées pourront alors être à valeurs dans un espace général dont les objets sont codés par le type `T` du template.

6.7. *Dans le fichier* `chaine.hpp` *écrire la classe* `VarAlea<T>` *qui dépend d'un paramètre template* `T`*. Les classes* `Bernoulli` *et* `Binomiale` *doivent maintenant être héritées de* `VarAlea<int>`*.*

6.8. *Ecrire une classe générique* `ChaineMarkov<T>` *qui a les mêmes fonctionnalités (méthodes et champs) que la classe* `ChaineMarkovZ`*. Si vous définissez l'opérateur d'injection* `<<` *comme méthode amie, il faut que cette fonction soit déclarer comme générique avec un nouveau paramètre template (différent de* `T`*).*

Une marche aléatoire sur $\mathbb{Z}^2$ est définie par $Z_{n+1} = Z_n + X_{n+1}$, $(Z_0 = 0)$ où Z_n et X_{n+1} sont des vecteurs de $\mathbb{Z}^2$, $X_{n+1} = (X^1_{n+1}, X^2_{n+1})$ avec X^1_{n+1} et X^2_{n+1} indépendantes et de même loi $\mathcal{B}(\{-1, 1\}, 0.5)$. La suite $(X_n)_{n\geq 1}$ est une suite de variables aléatoires indépendantes.

6.9. *En utilisant le conteneur* `std::vector<int>`*, écrire une classe générique* `MarcheAleaZ2` *dérivée de* `ChaineMarkov<T>` *instanciée avec le type* `std::vector<int>`*.*
Compiler et tester.

7 Rationnels et représentation Stern-Brocot

Le but est d'implémenter une classe `Rationnel` pour faire du calcul formel sur les fractions. Voici une utilisation possible de cette classe :

```
#include <iostream>
#include "rationnel.hpp"

using namespace std;
int main(void) {
    Rationnel a(24, 126);
    cout << "a:\t" << a << endl;
    cout << "valeur:\t" << a() << endl;
    cout << "a-1:\t" << a-1 << endl;
    Rationnel b = 2;
    cout << "b:\t" << b << endl;
    cout << "a*b:\t" << a*b << endl;
    cout << "a/b:\t" << a/b << endl;
    cout << "b/a:\t" << b/a << endl;
    return 0;
}
```

Ce programme compilé et executé doit donner le résultat suivant :

```
    a:  4/21
    valeur: 0.190476
    a-1:    -17/21
    b:  2
    a*b:    8/21
    a/b:    2/21
    b/a:    21/2
```

Classe `Rationnel`

Dans la suite, on considère une fraction $\frac{p}{q}$ avec $(p, q) \in \mathbb{Z}^2$. On autorise l'écriture $\frac{p}{0}$ qui représente l'infini.

```
class Rationnel {
    public:
        Rationnel(int p = 0, int q = 1)________
        Rationnel operator-() const { return Rationnel(-p, q); };
        double operator()()_________
        _________
    private:
        int p, q;
};
```

7.1. *Ecrire et compléter dans un fichier* `rationnel.hpp` *la déclaration de la classe* `Rationnel` *qui contient deux variables privées : un entier* `p` *codant le numérateur du rationnel et un entier* `q` *codant le dénominateur. L'opérateur fonctionnel* `operator()` *doit renvoyer la valeur numérique du rationnel.*

7.2. *Ecrire une fonction membre* `inv` *qui a deux comportements différents en fonction de sa signature : si elle est déclarée constante elle renvoie un objet* `Rationnel` *égal à l'inverse de l'objet courant, sinon elle modifie l'objet courant en l'inversant (au sens mathématique* i.e. $\frac{q}{p}$*).*

7.3. *Ajouter dans cette classe la déclaration des fonctions amies permettant de surcharger les opérateurs suivants :*

comparaisons *égalité, différent, inférieur strict*

arithmétiques `+`, `-`, `*`, `/`

injection `<<` *(qui produit un affichage similaire à l'exemple)*

Ecrire la définition de ces fonctions dans le fichier `rationnel.cpp`*.*

On souhaite dans la suite travailler uniquement avec des fractions irréductibles (les fonctions membres et les opérateurs de la classe `Rationnel` doivent renvoyer des objets avec `p` et q premiers entre eux).

7.4. *Ecrire une méthode privée* `simplifie` *qui simplifie la fraction codée dans l'objet courant et dont le prototype est :* `Rationnel simplifie()`.
Modifier le constructeur pour qu'il utilise cette méthode. De même pour toutes les méthodes et fonctions amies de la classe si nécessaire.

7.5. *Compiler et tester votre classe en écrivant un fichier* `test.cpp`. *Il faut que ce petit programme compile sans erreurs.*

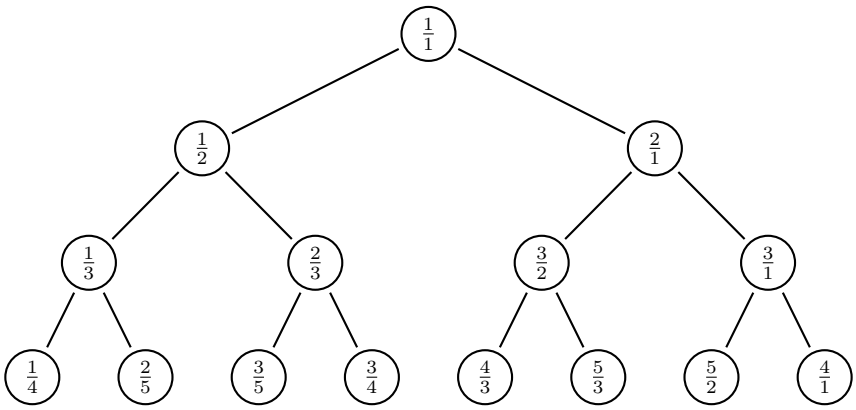
Représentation de Stern-Brocot

On définit la fraction *médiante* $\frac{m}{n}$ de 2 fractions $\frac{p}{q}$ et $\frac{p'}{q'}$ par $\frac{m}{n} = \frac{p+p'}{q+q'}$.

7.6. *Ecrire une méthode* `mediante` *de la classe* `Rationnel` *qui renvoie la fraction médiante de l'objet courant et d'un autre objet* `Rationnel`.

On définit l'arbre binaire de Stern-Brocot (cf. ci-contre) itérativement en considérant la fraction médiante de deux fractions. En partant des fractions $\frac{0}{1}$ et $\frac{1}{0}$ on construit la fraction $\frac{1}{1}$, puis on construit $\frac{1}{2}$ à partir de $\frac{0}{1}$ et $\frac{1}{1}$, $\frac{2}{1}$ à partir de $\frac{1}{1}$ et $\frac{1}{0}$, etc.

On peut prouver que toute fraction *irréductible positive* se trouve dans cet arbre binaire.



Cet arbre donne une représentation binaire de toute fraction irréductible positive. En effet, pour un rationnel a il existe un chemin de longueur k (atteignant a) dans l'arbre qui peut être représenté comme une suite de droite / gauche, ou bien une suite de vrai / faux.
L'algorithme pour trouver un rationnel a dans cet arbre consiste à comparer a avec le rationnel r du nœud courant : si $a == r$ c'est fini, si $a < r$ on va à gauche (valeur `false`) et si $a > r$ on va à droite (valeur `true`). Par exemple, la fraction $\frac{1}{1}$ est le trajet vide (point de départ) et la fraction $\frac{3}{5}$ est le trajet : gauche, droite, gauche que l'on codera `010`.
On rappelle que le conteneur `vector` du `C++` est défini dans le fichier d'en-tête `vector.h`. C'est un conteneur générique et on va utiliser dans la suite la spécialisation `vector<bool>` pour représenter un chemin de l'arbre de Stern-Brocot. On utilisera par exemple la méthode `v.push_back(b)` pour ajouter le `bool b` à l'objet `v` de la classe `vector<bool>`.

7.7. *Ecrire une fonction* `stern_brocot` *qui à une fraction renvoie son chemin dans l'arbre binaire.*
Tester sur différentes fractions (positives) et afficher la représentation binaire.

7.8. *Surcharger la fonction* `stern_brocot` *pour effectuer l'opération inverse : retrouver la fraction d'après un chemin.*

Application à l'approximation d'un réel par une fraction

7.9. *En vous inspirant de l'algorithme précédent, programmer une fonction qui, pour un réel positif x donné, renvoie le premier rationnel q trouvé dans l'arbre de Stern-Brocot vérifiant $|x - q| < \epsilon$ pour un paramètre ϵ (petit) donné.*

8 Algorithmes type Newton

Préambule : fonctions

Soit l'objet fonctionnel (ou foncteur) abstrait `FctR` suivant

```
struct FctR {
    FctR(bool b) : existe_der(b) {};
    virtual double operator()(double) const = 0;
    virtual double derivee(double) const { return 0; };
    bool existe_derivee() const { return existe_der; };
    protected:
        bool const existe_der;
};
```

Le champ `existe_der` est constant et doit être initialisé à la valeur `true` pour une classe dérivée qui contient une redéfinition de la méthode `derivee(double)` et à `false` sinon.

8.1. *Ecrire une classe* `Cubique` *héritée de* `FctR` *qui code les fonctions de la forme*

$$f(x) = (ax + b)^3, \quad a \in \mathbb{R}, \quad b \in \mathbb{R}.$$

Pour un argument x donné, l'opérateur `operator()` *doit renvoyer $f(x)$ et la méthode* `derivee` *doit renvoyer $f'(x)$. Les paramètres a et b seront des champs privés de la classe.*

Classe abstraite `AlgorithmeR`

On considère la classe abstraite `AlgorithmeR` suivante qui permettra une implémentation des algorithmes numériques de la forme $x_{n+1} = g(n, x_n)$ où g est une fonction codant l'algorithme dépendant d'une fonction f donnée.

```
class AlgorithmeR {
    public:
        AlgorithmeR(double valeur_init, FctR & f)
            : x_n(valeur_init), n_iter(0), f(f) {};
        AlgorithmeR & operator++() {
            calcul_iter();
            ++n_iter;
            return *this;
        };
    protected:
        virtual void calcul_iter() = 0;
        double x_n;
        int n_iter;
        FctR & f;
};
```

Les champs protégés sont les suivants :

- `x_n` : code la valeur de l'algorithme à l'instant n
- `n_iter` : code la valeur de l'instant i.e n
- `f` : est une référence sur un objet fonctionnel d'une classe héritée de `FctR`,
- `calcul_iter()` : méthode virtuelle pure qui devra être redéfinie dans les classes filles et qui code l'algorithme, i.e. la fonction g.

8.2. *Ecrire cette classe dans le fichier* `algorithme.hpp` *et ajouter un accesseur* `etat()` *qui renvoie la valeur du champ* `x_n` *et un accesseur* `n()` *qui renvoie la valeur du champ* `n_iter`*.*

8.3. *Ecrire une méthode* `reinit(double x_0)` *qui réinitialise l'algorithme en remettant sa valeur (son état) à* `x_0` *et* `n_iter` *à 0.*

8.4. *Surcharger l'opérateur d'*injection `<<` *qui permet d'envoyer la valeur des champs* `n_iter` *et* `x_n` *vers un flux de sortie. L'affichage doit être de la forme*

```
n_iter:    x_n
```

Algorithme de Newton et de la Secante

On se propose de tester cette classe abstraite en écrivant une classe `Newton` dérivée de `AlgorithmeR` qui code l'algorithme de Newton pour la recherche d'un zéro d'une fonction (réelle) i.e.

$$\forall n \geq 0, \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad x_0 \in \mathbb{R}.$$

On considère l'extrait de code suivant qui affiche les 30 premières itérations de l'algorithme de Newton appliqué à la fonction $f(x) = (-3.5x + 5.5)^3$ avec la condition initiale $x_0 = 5.1$.

```
Cubique F(-3.5, 5.5); // ie a = -3.5 et b = 5.5
Newton N(5.1, F);     // ie x_0 = 5.1
for (int i = 0; i <= 30; ++i) {
    cout << N << endl;
    ++N;
}
```

8.5. *Ecrire la classe* `Newton` *qui permet de reproduire cet exemple. Il faut donc :*
- *Ecrire un constructeur qui appelle explicitement le constructeur de la classe mère. On rappelle que cela doit se faire au début de la liste d'initialisation. Le constructeur doit vérifier que l'objet fonctionnel* `f` *à une méthode* `derivee()` *définie et afficher un message d'erreur si ce n'est pas le cas.*
- *Définir la fonction* `calcul_iter()` *qui devra être qualifiée de* `private` *et qui ne fait rien si la méthode* `derivee()` *n'est pas définie.*

Compiler et tester cette classe.

Si on remplace la dérivée de la fonction dans la méthode de Newton avec une différence finie on obtient la méthode de la sécante. Elle utilise la relation récurrente

$$\forall n \geq 0, \quad x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n), \quad x_0 \in \mathbb{R}$$

où on pose $x_{-1} = x_0 - 1$.

8.6. *Ecrire une classe* `Secante` *dérivée de* `AlgorithmeR` *qui permet d'implémenter cet algorithme. Vous pourrez utiliser le champ* `x_n` *de la classe mère et un champ* `x_nm1` *qui sera qualifié de* `private` *dans la classe* `Secante`.

8.7. *La méthode* `reinit()` *de la classe mère est-elle toujours valide dans la classe fille* `Secante` *? Proposer une solution.*

Passage au générique

On souhaite maintenant étendre les classes abstraites `FctR` et `AlgorithmeR` en des classes abstraites génériques `Fct<S>` et `Algorithme<S>` où `S` est un paramètre template. Les fonctions et algorithmes considérés pourront alors être à valeurs dans un espace général dont les objets sont codés par le type `S` du template.

8.8. *Dans le fichier* `algorithme.hpp` *écrire la classe* `Fct<S>` *qui dépend d'un paramètre template* `S`. *La classe* `Cubique` *doit maintenant être héritée de* `Fct<double>`. *Tester votre code.*

8.9. *Ecrire une classe générique* `Algorithme<S>` *qui a les mêmes fonctionnalités (méthodes et champs) que la classe* `AlgorithmeR`. *Si vous définissez l'opérateur d'injection* `<<` *comme méthode amie, il faut que cette fonction soit déclarer comme générique avec un nouveau paramètre template (différent de* `S`*).*

8.10. *Définir une nouvelle classe* `Secante2` *qui se comporte comme la classe* `Secante` *mais qui est héritée de la classe abstraite générique* `Algorithme<S>` *instanciée avec le type* `std::vector<double>`. *Un état de l'algorithme sera donc un vecteur de taille 2 représentant les 2 dernières valeurs de l'algorithme de la sécante.*
Compiler et tester cette classe.

8.11. *En utilisant le type* `std::complex<double>` *(charger* `complex.hpp`*) ou votre propre classe complexe écrire un code qui recherche le zéro de la fonction complexe* $f(z) = z^3 - 1$ *par les deux algorithmes Newton et Secante.*

9 Interpolation de Newton-Lagrange

Le but est d'implémenter une classe permettant de construire le polynôme d'interpolation de Lagrange de n points $(x_1, y_1), \ldots, (x_n, y_n)$ et de l'évaluer facilement en tout point réel. Dans toute la suite on suppose $x_1 < x_2 < \cdots < x_n$. Le polynôme de Lagrangeest le polynôme P de degré $n-1$ defini par

$$P(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{(x - x_j)}{(x_i - x_j)} \tag{1}$$

vérifiant en particulier $P(x_i) = y_i$.

Classe abstraite `Interpol`

Dans un premier temps, on code une classe abstraite `Interpol`. Cette classe a pour but d'avoir des classes filles qui réutilisent les champs et les méthodes de cette classe abstraite. Voici la définition partielle de la classe `Interpol`.

```
class Interpol {
    public:
        Interpol(std::string nom_fichier, int n);
        ----------
        virtual double operator()(double x) = 0;
    protected:
        int n;
        double * points_x, * points_y, * poids;
};
```

On définit le constructeur de la façon suivante :

```
Interpol::Interpol(std::string nom_fichier, int n) : n(n) {
    std::ifstream file(nom_fichier.c_str());
    points_x = new double[n];
    points_y = new double[n];
    poids = new double[n];
    for (int i = 0; i < n; ++i) {
        file >> points_x[i] >> points_y[i];
    }
    file.close();
};
```

Le constructeur défini ci-dessus alloue les différentes zones mémoires utilisées par la classe (plus précisément un objet d'une classe fille non abstraite). Pour simplifier, on allouera `n double` pour chacun des champs `points_x`, `points_y`, `poids`. L'initialisation des champs `points_x` et `points_y` se fait à partir des `n` premières lignes d'un fichier texte qui contient 2 colonnes : la première codant les $(x_k)_{k=1,\ldots,n}$ et la seconde les $(y_k)_{k=1,\ldots,n}$.

9.1. *Etant donné que les champs* `points_x`, `points_y` *et* `poids` *sont des adresses vers des zones mémoires en dehors de la classe, quelles sont les 3 méthodes à redéfinir ?*
Dans un fichier `interpol.hpp`*, définissez la classe* `Interpol` *et ses 4 méthodes (indication : 2 constructeurs, un opérateur et une autre méthode spéciale à déterminer).*

9.2. *Ajouter une méthode* `min()` *à cette classe qui renvoie le plus petit point x_k pour $k \in \{1, \ldots, n\}$.*
Définir de la même façon une méthode `max()`*.*

Interpolation linéaire

Pour tester cette classe abstraite, on définit une classe `InterpolLin` dérivée (publiquement) de `Interpol` qui permet d'implémenter l'interpolation linéaire entre les points $((x_k, y_k)_{k=1,\ldots,n})$. Il s'agit de la fonction l définie par

$$\forall x \in [x_k, x_{k+1}], \quad l(x) = y_k + a_k(x - x_k),$$

$l(x) = y_1$ si $x \leq x_1$ et $l(x) = y_n$ si $x \geq x_n$. Les réels a_k sont les poids de l'interpolation.
Un code pour tester cette classe serait par exemple :

```
InterpolLin h("points.dat", 7);
for (double x = h.min()-1; x <= h.max()+1; x += 0.2)
    cout << x << "\t" << h(x) << endl;
```

9.3. *Définir la classe* `InterpolLin` *dans le fichier* `interpol.hpp` *ainsi que le constructeur appelé dans cet exemple : le constructeur devra appeler celui de la classe mère et définir notamment les points* $(a_k)_{k=1,\dots,n}$ *utilisés pour l'interpolation linéaire.*

9.4. *Définir l'opérateur fonctionnel* `operator()` *(qui redéfinit alors celui de la classe mère* `Interpol`*).*

Avant de passer à la suite, il faut compiler et tester vos 2 classes.

Interpolation polynômiale

Le polynôme de Lagrange P défini en (1) peut se réécrire(sous forme de Newton) de la façon suivante

$$\forall x, \quad P(x) = y_1 + \sum_{k=2}^{n} a_{1,\dots,k}(x - x_1)\dots(x - x_{k-1}), \tag{2}$$

où pour tout $i \in \{1, \dots, n\}$, et j tel que $i + j \in \{1, \dots, n\}$, le coefficient $a_{i,\dots,j}$ est défini récursivement par

$$\begin{cases} a_{i,\dots,i+j} = \frac{a_{i+1,\dots,i+j} - a_{i,\dots,i+j-1}}{x_{i+j} - x_i}, \\ a_{k,k} = y_k, \quad \forall k \in \{1, \dots, n\}. \end{cases}$$

9.5. *De façon similaire à la classe* `InterpolLin`*, on définit une classe* `InterpolPoly` *dérivée de* `Interpol`*. Cette classe a le même constructeur que la classe* `InterpolLin` *dont le but est d'appeler celui de la classe mère et d'initilialiser les poids* $a_{1,..,k}$ *pour* $k = 2, \dots, n$*.*

9.6. *Définir l'opérateur* `()` *et écrire un programme de test complet.*

10 Fonctions par morceaux

Classe mère `StepFunction`

On considère la classe mère `StepFunction` qui permettra une manipulation aisée des fonctions $f : \mathbf{R}_+ \to \mathbf{R}_+^*$ définies par morceaux sur les intervalles de la forme $[k, k+1[$, pour $k \geq 0$. Une fonction par morceaux dans la suite du sujet sera strictement positive sur $\mathbf{R}_+^*$ (positive en 0) et définie à partir de sa valeur $x_k \in \mathbf{R}_+^*$ au point k. On considèrera dans la suite des fonctions constantes par morceaux ou linéaires par morceaux. Voici la définition de la classe `StepFunction`

```
class StepFunction {
    public:
        StepFunction(double const * xk, unsigned size);
    protected:
        std::vector<double> points;
};
```

Le constructeur de cette classe doit initialiser le champ `points` avec les valeurs x_k. Voici un exemple d'appel :

```
double xk[] = { 0.2, 0.36, 0.5, 1.2 };
StepFunction f(xk, 4);
```

La fonction f prend la valeur 0.2 en 0, 0.5 en 1, 0.36 en 2 et 1.2 en 3. On considèrera qu'elle n'est pas définie après 4 et qu'elle prend dans ce cas la dernière valeur (ici 1.2).

10.1. *Ecrire cette classe dans le fichier* `StepFunction.hpp` *et définir le constructeur dans le fichier* `StepFunction.cpp`.

10.2. *Définir (dans le fichier* `StepFunction.cpp`*) l'opérateur d'injection* `<<` *pour un objet de cette classe. Vous afficherez* k *et* x_k.

Classes filles `CstStepFunction` et `LinearStepFunction`

On définit maintenant deux classes filles dérivées publiquement de `StepFunction` : la classe `CstStepFunction` pour manipuler les fonctions en escalier (constantes par morceaux) et le classe `LinearStepFunction` pour les fonctions linéaires par morceaux croissantes et continues (interpolation linéaire entre les points (k, x_k) et $(k+1, x_{k+1})$. Le principal ajout est l'opérateur fonctionnel `operator()` pour obtenir le comportement suivant

```
double xk[] = { 0.2, 0.36, 0.5, 1.2 };
CstStepFunction f(xk, 4);
LinearStepFunction g(xk, 4);
double t = 1.5;
std::cout << t << "\t" << f(t) << "\t" << g(t) << std::endl;
```

Le résultat doit être l'affichage

```
1.5     0.36    0.43
```

10.3. *Définir les deux classes* `CstStepFunction` *et* `LinearStepFunction` *dans le fichier* `StepFunction.hpp` *ainsi que l'opérateur fonctionnel qui sera défini dans la classe pour être* `inline`.

10.4. *Définir l'opérateur arithmétique* `+` *pour ces deux classes.*
A-t-on besoin de redéfinir l'opérateur d'affectation `=` *? Si oui faites-le.*

10.5. *Définir deux opérateurs de multiplication* `*` *pour chacunes de ces classes qui permettent la multiplication d'une fonction par un scalaire (positif, le test n'est pas nécessaire). Ces opérateurs permettront les appels suivants :*

```
CstStepFunction h = f*2;
LinearStepFunction k = 2*g;
```

On rappelle qu'un objet de la classe `LinearStepFunction` code une fonction strictement croissante.

10.6. *Ajouter une méthode* `inverse` *dans la classe* `LinearStepFunction` *qui code l'inverse pour la composition (la fonction réciproque) de l'objet courant. Par exemple* `g.inverse(0.43)` *doit renvoyer* `1.5`.
Vous pourrez faire une simple recherche linéaire, ou bien utiliser la fonction `find` *dans* `algorithm` *ou la fonction (plus efficace)* `binary_search` *(cf. polycopié page 187).*

Fonctions globales associées

10.7. *Ecrire la fonction* `integrate` *de prototype*

```
LinearStepFunction integrate(CstStepFunction const & f);
```

Cette fonction doit renvoyer la fonction linéaire par morceaux continue, primitive (nulle en 0) de la fonction f constante par morceaux.

10.8. *Ecrire la fonction* `derive` *qui réalise l'opération inverse (la fonction linéaire par morceaux est strictement croissante).*

10.9. *Tester ces différentes classes et fonctions dans un fichier* `test.cpp`*, en codant le petit exemple suivant :*

- *définir la fonction* `rate` *constante par morceaux codant la fonction λ*

$$\lambda(x) = 0.05 \times \mathbf{1}_{[0,1[}(x) + 0.15 \times \mathbf{1}_{[1,2[}(x) + 0.3 \times \mathbf{1}_{[2,3[}(x) + 0.2 \times \mathbf{1}_{[3,4[}(x) + 0.1 \times \mathbf{1}_{[4,+\infty[}(x)$$

- *construire la fonction* `integrated_rate` *codant $\Lambda(t) = \int_0^t \lambda(x)dx$*
- *afficher les fonctions λ et Λ*
- *initialiser un réel* `u` *avec l'instruction* `random()/(1.+RAND_MAX)` *(*`u` *est considéré comme une réalisation d'une variable aléatoire uniforme sur $[0, 1[$)*
- *afficher* `u` *et la transformation* `t=f(u)` *où $f(u) = \Lambda^{-1}(-\log(1-u))$ (*`t` *est une réalisation de la variable aléatoire de densité $\lambda(x)e^{-\Lambda(x)}$).*

Code générique

On généralise maintenant le code pour manipuler toute fonction à valeurs dans un espace E dont un élément est codé par une variable de type `T` générique. On suppose qu'il existe les opérateurs de comparaison `<` et `==` pour des objets de type `T`.

10.10. *Rendre la classe* `StepFunction` *générique, dépendant d'un type générique* `T` *et faîtes hériter* `CstStepFunction` *et* `LinearStepFunction` *de l'instance* `StepFunction<double>`*. Tester avec le fichier* `test.cpp`*.*

10.11. *Rendre les classes* `CstStepFunction` *et* `LinearStepFunction` *génériques qui dépendent du même type générique que l'instance de la classe mère* `StepFunction<T>`*.*
Retester tout votre code avec comme type générique `T` *le type* `double`*.*

10.12. *Ajouter les opérateurs* `<` *et* `==` *à votre classe* `CstStepFunction<T>`*. Modifier votre code pour qu'il compile avec le type générique* `T` *qui serait* `CstStepFunction<double>`*. Ecrire un petit code de test.*

11 Suites cycliques

Classe abstraite `Suite<T>`

On considère la classe abstraite `Suite<T>` qui permettra une programmation aisée des suites récurrentes (à valeurs dans un espace général). Le code suivant doit être défini dans le fichier `suite.hpp` :

```
template <typename T>
class Suite {
    public:
        Suite(T const & etat) : etat_init(etat), etat(etat), n_iter(0) {};
        Suite<T> & operator++() {
            change_etat();
            ++n_iter;
            return *this;
        };
        virtual void change_etat() = 0;
        T get_etat() const { return etat; }
        unsigned get_n() const { return n_iter; }
        void reinit() { etat = etat_init; n_iter = 0; }
    protected:
        T const etat_init;
        T etat;
        unsigned n_iter;
};
```

Le champ `n_iter`, initialisé à 0 par le constructeur, représente le numéro de l'itération et le champ (générique) `etat` représente la valeur de la suite à l'itération `n_iter`. L'opérateur d'incrément surchargé ici est le pré-incrément (le post-incrément n'est pas possible car il s'agit d'une classe abstraite).
La méthode virtuelle pure `change_etat()` doit être modifiée dans toutes les classes dérivées de `suite<T>`. La relation de récurrence de la suite doit être codée dans cette méthode.

11.1. *Ecrire cette classe dans le fichier* `suite.hpp` *et surcharger l'opérateur d'affectation* `+=` *qui prend pour argument un entier* `k` *et qui effectue* `k` *itérations de la suite.*

11.2. *Définir dans le fichier* `suite.hpp` *une classe* `Lucas` [1] *dérivée de l'instance* `Suite< std::pair<double, double> >` *dont le constructeur prend deux paramètres entiers* `P` *et* `Q` *et qui permet de coder la suite récurrente suivante :*

$$\forall n \geq 1, \quad u_{n+1} = Pu_n - Qu_{n-1},$$
$$u_0 = 0, \quad u_1 = 1.$$

Suite Logistique

La suite logistique est une suite récurrente non linéaire définie par la relation suivante

$$\forall n \geq 0, \quad x_{n+1} = \mu x_n(1 - x_n), \quad x_0 \in]0, 1[$$

où $\mu \geq 0$ est un paramètre fixé. Le comportement de cette suite varie en fonction de μ. Par exemple, la suite devient cyclique à partir d'un certain rang pour une valeur de $\mu \in]3; 3, 57[$.
On considère l'extrait de code suivant qui affiche les 5 premières itérations de la suite logistique de condition initiale $x_0 = 0, 5$ et de paramètre $\mu = 3, 47$.

```
Logistique X(3.47, 0.5);
for (int i = 0; i <= 5; ++i) {
    cout << X << endl;
    ++X;
}
```

11.3. *Définir dans le fichier* `suite.hpp` *une classe* `Logistique` *dérivée de l'instance* `Suite<double>` *compatible avec l'exemple précédent. Il faut donc :*

1. généralisation de la suite de Fibonacci

— *Ecrire un constructeur qui appelle explicitement le constructeur de la classe mère instanciée avec le type* `double`*, c'est à dire* `Suite<double>`*. On rappelle que cela doit se faire au début de la liste d'initialisation.*
— *Définir l'opérateur d'injection* `<<` *pour afficher l'itération courante et l'état courant.*
— *Définir la fonction* `change_etat()` *(qui devra être qualifiée de* `private`*).*

Compiler et tester cette classe. Tester aussi l'opérateur `+=`*.*

Pour une suite Logistique donnée (un paramètre μ donné) on veut déterminer la longueur d'un cycle. Voici le code à trou proposé :

```
int cycle(Logistique const & suite) {
    list<double> hist;
    list<double>::iterator it;
    _________
    do {
        hist.push_back( ________ );
        _________
        _________
        it = find( __________ );
    } while (it == hist.end());
    return distance(it, hist.end());
};
```

Le principe est le suivant : sauver dans une variable `hist` toutes les valeurs passées de la suite, puis incrémenter la suite et chercher la nouvelle valeur dans `hist`. La fonction `find` est définie dans `algorithm` et son prototype est donnée page 87 du polycopié.

11.4. *Dans le programme de test, coder la fonction* `cycle` *en complétant le code à trou ou en utilisant votre propre code.*
Tester la fonction avec la suite logistique `X` *(de paramètre $\mu = 3,47$). Vous devez trouver un cycle de longueur 4.*

Solitaire bulgare

On souhaite maintenant coder une suite récurrente modélisant le jeu suivant :
— On dispose d'un paquet de m cartes que l'on sépare en un certain nombre k de tas contenant chacun une ou plusieurs cartes.
— Un mouvement consiste à prendre une carte de chacun des k tas pour former un nouveau tas de k cartes. Les anciens tas ont diminué chacun d'une carte et il y a donc un nouveau tas de k cartes, le nombre de tas est maintenant $k+1$ au plus, mais peut-être moins si certains tas qui ne contenaient qu'une carte se sont totalement vidés.
— La disposition ou l'ordre des tas importe peu.

Ce jeu peut être modélisé par une suite récurrente $(x_n)_{n\geq 0}$ où un état x_n est représenté par une liste d'entiers `std::list<int>` ou un vecteur d'entiers `std::vector<int>` codant chacun la taille d'un tas.
On considère le code suivant

```
Jeu J(6);
for (int i = 0; i <= 5 ; ++i) {
    cout << J << endl;
    ++J;
}
```

qui donne le résulat :

```
0:  1 - 1 - 1 - 1 - 1 - 1
1:  6
2:  5 - 1
3:  4 - 2
4:  3 - 1 - 2
5:  2 - 1 - 3
```

11.5. *Définir la classe* `Jeu`*, héritée d'une spécialisation de* `Suite<T>`*, implémentant ce jeu. La fonction* `change_etat()` *devra être qualifiée de* `private`*. Le constructeur et les opérateurs à surcharger doivent se déduire de l'exemple. Vous pourrez utiliser les méthodes* `erase` *et* `push_back` *du conteneur utilisé.*

11.6. *Généraliser la fonction* `cycle` *à tout objet d'une classe héritée d'une instance de la classe* `Suite<T>`*.*

11.7. *Quelle est la longueur d'un cycle du solitaire bulgare si le nombre de cartes est $m = 8$? et $m = 15$?*