

Introduction au C++

Jacques-Olivier Moussafir

msfr@ceremade.dauphine.fr

1 Introduction

Ce document contient quelques programmes illustrant les aspects principaux du C++. La présentation des notions est assez rapide et doit permettre – en principe – de ce faire une idée de ce que ce langage apporte.

Pour un exposé détaillé, le document électronique de Bruce Eckel [1] disponible à l'adresse : www.mindview.net est une très bonne référence. Pour débiter il vaut sans doute mieux commencer par le livre de Satir et Brown [2]. Une liste de bibliothèques scientifiques C++ disponibles sur : www.oonumerics.org

Premier programme

Pour créer votre premier programme C++, il faut :

1. Éditer un fichier texte `main.cpp` qui contient le corps du programme.
2. Compiler `main.cpp`.

Pour l'édition il faut utiliser son éditeur de texte préféré. Pour la compilation, c'est un peu plus compliqué : ça dépend du système. Sous Linux, il suffit d'entrer dans un shell (le `#` est le prompt du shell) :

```
# g++ main.cpp
```

Avec SunOS

```
# CC main.cpp
```

La compilation – quand elle se déroule bien – produit un exécutable `a.out`.

Pour l'exécuter entrer

```
# a.out
```

Pour que l'exécutable soit nommé `prog` plutôt que `a.out`, il faut compiler avec la commande

```
# g++ main.cpp -o prog
```

Un exemple de fichier `main.cpp`.

```
#include <iostream>
int main()
{
    cout << "Hello World" << endl;
}
```

On peut aussi compiler en utilisant la commande `make`. Il faut créer un fichier `makefile` et y mettre ce qui suit :

```
CPP = g++
DEBUG = -g
OBJ = main.o
LIBS =
prog: $(OBJ)
      $(CPP) $(OBJ) $(LIBS) -o $@
main.o: main.cc
      $(CPP) -c main.cpp -o $@
```

Pour compiler taper

```
# make
```

L'utilité de la commande `make` apparaît clairement pour les gros programmes.

2 Entrées/Sorties

Pour écrire les exemples on a presque toujours besoin de lire ou d'écrire dans des fichiers textes ou sur l'entrée et la sortie standard. La syntaxe peut sembler un peu mystérieuse. Elle paraîtra plus claire après l'examen de la notion de classe et d'opérateur surchargé. Il faut inclure comme en C :

```
#include<iostream>
```

Il y a trois variables (objets en fait) prédéfinies: `cout`, `cin` et `cerr`. Elles correspondent à `stdout`, `stdin` et `stderr` qu'on utilise en C. Pour écrire :

```
float x = 3.1415;  
cout << x << endl;
```

Pour lire :

```
float x;  
cin >> x;
```

Voici un exemple complet :

```
#include<iostream>  
int main()  
{  
    // Afficher des chaînes de caractères  
    // des int, float...  
    int i=2; // Definit une variable de type int initialisée à 2  
    cout << "i = " << i << endl;  
    float x=3.14; // Définit une variable de type réel  
    cout << "x = " << x << endl;  
    int j;  
    cout << "j = "; // Ecriture d'une chaine de caractères  
    cin >> j;      // Lecture de la valeur de j  
    cout << "j = " << j << endl; // Ecriture de la valeur de j  
}
```

Pour écrire et lire dans un fichier, il faut inclure dans l'en-tête

```
#include<fstream>
```

Le code suivant illustre l'usage de `cout` et `cin`, ainsi que la lecture et l'écriture dans un fichier.

```
#include<iostream>
#include<fstream>

int main()
{
    const int MAX_LENGTH_FILENAME=127;

    // outfile est le nom du fichier dans lequel on écrit

    char outfile[MAX_LENGTH_FILENAME] = "./resutat";
    ofstream ostream(outfile, ios::out);

    // ostream est un objet analogue à cout
    // au lieu d'écrire sur la sortie
    // standard, ostream << écrit dans le fichier
    // dont le nom est outfile

    for(int k=1; k<=100; k++) ostream << k*k << endl;

    // Ouvrir un fichier en lecture
    // La démarche est analogue, on
    // crée un objet analogue à cin

    char infile[MAX_LENGTH_FILENAME] = "./resutat";
    ifstream instream(infile, ios::in);

    int l;
    for(int k=1; k<=200; k++)
    {
        instream >> l;
        cout << l << endl;
    }
}
```

3 Signatures

Le C ANSI impose que les fonctions soient déclarées avec le type des paramètres :

```
int max(int a,int b);
```

Cela empêche de définir dans le même programme deux fonctions

```
int max(int a,int b);
```

```
float max(float a,float b);
```

et conduit à une multiplication de noms de fonctions du type `max_float` et `max_int`. Le C++ résoud ce problème très simplement : il est possible de définir dans le même programme

```
int max(int a,int b);
```

```
float max(float a,float b);
```

Le compilateur détermine la fonction à appeler en fonction du type des paramètres :

```
int a,b,c;
```

```
c=max(a,b); // appel de int max(int a, int b)
```

```
float x,y,z;
```

```
z=max(x,y); // appel de float max(float a,float b)
```

On illustre cette possibilité dans le programme suivant. On définit un type `complex` et deux fonction `mult` : l'une prend pour argument des `float` et l'autre des `complex`. Cet exemple sert aussi à illustrer la notion de `struct`.

```
#include<iostream.h>
```

```
// En C++ les struct sont des types
```

```
struct complex
```

```
{
```

```
    float re;
```

```
    float im;
```

```
};
```

```
complex mult(complex a, complex b)
```

```
{
```

```
    complex c;
```

```
    c.re = a.re * b.re - a.im * b.im;
```

```

    c.im = a.re * b.im + a.im * b.re;
    return c;
}

float mult(float a, float b)
{return a*b;}

int main()
{
    float a, b;

    cout << "a = ";cin >> a; cout << "b = ";cin >> b;
    cout << "a b = " << mult(a,b) << endl;

    complex u,v,w;
    cout << "u.re = ";cin >> u.re; cout << "u.im = ";cin >> u.im;
    cout << "v.re = ";cin >> v.re; cout << "v.im = ";cin >> v.im;

    w = mult(u,v);
    cout << "u v = " << w.re << " + i" << w.im << endl;
}

```

L'exemple qui précède fait aussi ressortir quelques lourdeurs du C: on aimerait pouvoir écrire

```

complex u,v,w;
w = u*v;
Et aussi
float x;
complex y,z;
z = x*y;

```

C'est à dire qu'on aimerait bien pouvoir définir une conversion automatique du type `float` vers le type `complex`, et redéfinir les opérateurs `*` et `+` selon le type des paramètres placés de part et d'autre du symbole. C'est ce que permettent les constructeurs et la surcharge d'opérateurs ...

4 Conversions

Une conversion de type est parfois une opération très naturelle dont le programmeur ne prend pas conscience et qui est prise en charge de manière automatique par le compilateur :

```
int i=2;
float x;
x = i;
```

Pour pouvoir ranger le contenu de la variable `i` dans la variable `x`, il faut interpréter le contenu de `i` en `float`. Mais cette conversion est tellement naturelle que le compilateur `C` accepte cette assignation : la valeur 2 est codée dans le format `float` et rangée dans l'espace mémoire correspondant à `x`.

Certaines conversions en revanche ne sont pas naturelles du tout. On peut souhaiter par exemple convertir une chaîne de caractères "237" dans l'entier correspondant. Dans ce cas il faut faire appel à des fonctions de conversion spéciales.

L'opérateur `cast` est un élément essentiel du `C` : il permet d'assigner un type de variable à un espace mémoire préalablement réservé. Sur certaines machines, les `float` sont stockés sur 4 octets, et occupent donc l'espace de 4 caractères. On peut par exemple écrire

```
float x=3.14159;
cout << (char*)&x << endl;
```

Le compilateur interprète `(char*)&x` comme une chaîne de caractères et produit sur la sortie standard les 4 caractères correspondants.

Cet exemple est destiné à montrer que la mémoire n'est pas altérée par le `cast`, mais seulement le type des variables, et donc leur interprétation et leur comportement.

Voici un autre exemple : on crée deux structures occupant le même espace en mémoire : `structA` et `structB`. On déclare une variable de type `structA`, puis un pointeur sur `structA` casté en `structB` :

```
struct structA
{
    float xA;
    float yA;
};
```

```

struct structB
{
    float xB;
    float yB;
};

```

```

structA a;
a.xA = 0;
a.yA = 1;

```

```

structB *pb;
pb = (structB*)&a;
pb->xB=2; // Autorisé
cout << a.xA << endl;

```

En C on utilise souvent le `cast` au moment de l'allocation mémoire. L'exemple qui suit est typique :

```

// cast courant
struct complex
{
    float re;
    float im;
};

```

```

complex *pc;
{
    float re;
    float im;
};

```

```

complex *pc;
pc = (complex*)malloc(sizeof(complex));
pc->re = 1; pc->im = 2;

```

La fonction `malloc()` retourne un `void*` casté en `complex*`.

Remarque – Il faut noter que certains casts sont permis et d'autres non : on peut caster de `int` vers `float`, de `void*` vers `structA*`, mais pas de `structA` vers `structB`, ni de `complex*` vers `structA*`.

5 const

Le C++ définit deux nouveaux mots clef `const` et `*const`. Ils permettent respectivement de définir des variables et des pointeurs constants. Ils remplacent et généralisent l'usage des macros `#define`. Ils permettent aussi d'éviter les erreurs habituelles dues à la substitution brutale du texte qui suit les `#define`.

Pour définir une variable constante, on fait précéder la déclaration du mot clef `const`

```
const int i=3;
```

Il faut attribuer une valeur à la variable au moment de sa déclaration : après c'est impossible. Si une variable est déclarée `const`, les assignations ultérieures sont rejetées par le compilateur les instructions:

```
const int i=3;
i=2;
```

donne une erreur de compilation.

Un pointeur constant est un pointeur qui pointe toujours sur la même adresse, mais dont le contenu peut changer. Pour définir un pointeur constant, on place le mot clef `*const` devant le nom du pointeur :

```
int i=2;
int *const pi= &i;
```

Le pointeur `pi` pointe sur `i` et une instruction du type

```
int j;
pi=&j;
```

sera rejetée par le compilateur. On peut obtenir des effets un peu compliqués en combinant les usages de `const` et `*const`. On peut par exemple déclarer `i` une variable `const`, et `pi` un poiteur sur `i`, et modifier la valeur de `i` par l'intermédiaire de `*pi`. Des exemples de ce type sont illustrés par le programme qui suit. Le lecteur est invité à faire quelques tests lui-même.

```
#include<iostream>
```

```
int main()
{
    const int i=2;
    // i = 3; -> Erreur à la compilation

    int j=2;
```

```
const int *pj;
pj = &j;

// *pj =3; -> Erreur à la compilation :
// la variable pointée *pj est constante,
// mais pas j.

j=3;
cout << *pj << endl;

// L'opérateur *const définit un pointeur
// constant. La variable pointée peut changer
// de valeur

int k=2; j=3;
int *const pk = &k;
cout << *pk << endl;

// pk = &j; -> Erreur : pk est constant
}
```

6 Références

Lors d'un appel de fonction C, certains paramètres sont dits passés par valeur : la fonction appelée récupère une copie des variables passées :

```
int max(int a, int b);
int i, j=2, k=3;
i = max(j, k);
```

Il y a deux problèmes : la copie de grosses variables est une opération lourde, et la fonction appelée ne peut pas modifier les paramètres passés. Généralement, en C, on résout ces problèmes en passant des pointeurs :

```
int max(int* a, int* b);
int i, j=2, k=3;
i = max(&j, &k);
```

Le C++ introduit la notion de référence : définir une référence sur une variable revient à lui donner deux noms :

```
int i=2;
int &j = i;
```

Ici `i` est `int` et `j` une référence sur `int`. La variable `j` se manipule exactement comme `i`, et si `j` est affectée, `i` l'est aussi.

```
j=3;
cout << i << endl; // 3 sur la sortie standard
```

Syntaxiquement, rien ne distingue un `int` et une référence sur `int`. Mais en fait du point de vue de l'occupation de la mémoire, une référence est plutôt un pointeur.

Une fonction qui modifie ses paramètres s'écrit en C++ :

```
void incremente(int &i)
{i++;}
```

```
int main()
{
    int i=2;
    incremente(i);
    cout << i << endl; // 3 sur la sortie standard
}
```

On peut empêcher qu'une variable passée par référence soit modifiée en utilisant `const`.

```
void incremente(const int &i)
{
    i++; // Erreur de compilation
}
```

Comme les références occupent peu d'espace mémoire et que leur manipulation est identique à celle des variables, le C++ en fait un grand usage. Il y a même quelques cas où leur utilisation est obligatoire.

7 Classes

Une classe est en première approximation comme une structure `C` comprenant des variables et des fonctions. Celles-ci constituent ce qu'on appelle l'interface de la classe : on manipule les classes en utilisant les fonctions de l'interface. **Exemple** – Une pile d'entier : c'est un objet qui contient une liste finie et ordonnée d'entiers : (a_1, \dots, a_n) . On peut insérer un nouvel entier `a` dans la pile qui se transforme en (a_1, \dots, a_n, a) , ou enlever la valeur qui est au sommet de la pile, ici `a`. La pile se transforme à nouveau en (a_1, \dots, a_n) .

L'utilisateur de la pile n'a pas besoin de savoir comment elle est programmée ni quelles structures de données lui correspondent, tableau, liste chaînée, etc. Il a juste besoin d'une fonction qui met une valeur au sommet de la pile et d'une autre qui enlève un entier de la pile. On peut éventuellement ajouter une fonction qui donne le nombre d'éléments de la pile, et une autre qui l'imprime.

Pour résumer, on a besoin des fonction `push()`, `pop()`, `getsize()` et `print()` qui réalisent ces trois opérations. Le code suivant montre une implémentation.

```
#include<iostream>
const int STACK_MAX_SIZE=100;

class stack
{
public :
inline stack()
{
size=0;
for(int i=0; i<=STACK_MAX_SIZE - 1; i++) elems[i]=0;
}
inline ~stack(){ }
void push(int a);
int pop();
int getsize();
void print();
```

```

    private :
    int elems[STACK_MAX_SIZE];
    int size;
};

void stack::push(int a)
{
    elems[size]=a;
    size++;
}

int stack::pop()
{
    int resultat=0;
    if(size>=1) elems[size-1];
    elems[size-1] = 0;
    if(size>0) size--;
    return resultat;
}

int stack::getsize(){return size;}

void stack::print()
{
    cout << "Taille de la liste :" << size << endl;
    for(int i=size-1; i>=0; i--)
        cout << "stack[" << i << "] = " << elems[i] << endl;
}

int main()
{
    stack s;
    s.push(1); s.push(10); s.push(1000);
    s.print();
    s.pop();
    s.pop();
    s.print();
}

```

Ce code illustre plusieurs particularités du C++.

- On fait appel aux fonctions de la classe comme aux membres d'un `struct`. On parle d'ailleurs de fonctions membres.
- Le mot-clé `public` définit les champs, données ou fonctions, accessibles de l'extérieur, c'est-à-dire depuis une fonction qui n'est pas membre de la classe. Le mot-clé `private` définit au contraire ce qui n'est pas accessible de l'extérieur : dans `main()`, une instruction

```
s.size=2;
```

provoquerait une erreur. Les fonctions membres de la classe accèdent librement aux données `private`. L'ensemble des données et fonctions `public` forme ce qu'on nomme l'interface de la classe.

- Il se pourrait que plusieurs classes définissent une fonction `print`. On utilise une déclaration du type

```
void stack::print(){...}
```

pour préciser de quelle fonction on parle.

- On a écrit une fonction spéciale `stack()` : le constructeur de la classe. Elle est appelée à chaque fois qu'une variable `stack` est créée. Elle initialise les différents constituants de la classe.
- On a aussi déclaré une fonction `~stack()` : c'est le destructeur de la classe. Le code correspondant est exécuté dès que la variable n'est plus accessible. En général les fonctions de ce type effectuent les appels de libération mémoire. Ici, il n'y a rien à faire.
- Le mot-clé `inline` précède la déclaration de `stack()`. Il permet à certains compilateurs de faire des appels de fonction par substitution de code.
- Dans le code correspondant à la classe `stack`, un problème d'implémentation subsiste : si le nombre d'entiers insérés dans la pile est plus grand que `STACK_MAX_SIZE`, on risque de provoquer des `segmentation fault` (erreur qui survient quand un programme cherche à écrire des données à un emplacement qui n'a pas été réservé). Cela suggère de modifier le code de la classe `stack`, sans que l'interface de la classe ne soit modifiée. Cette notion n'est pas vraiment spécifique du C++, mais sa mise en œuvre y est assez facile.

La terminologie usuelle désigne sous le terme de classe abstraite le modèle de la classe, ses données, ses fonctions. Une variable de type `stack` dans `main()`, s'appelle une instance de la classe.

8 Allocation mémoire

En C, l'allocation d'espace mémoire se fait de façon assez directe : on appelle une fonction qui réserve un bloc mémoire et retourne un pointeur sur le début de la région allouée.

La libération de l'espace se fait de manière analogue, par un appel à une fonction :

```
free(p);
```

où `p` est le pointeur obtenu après la demande d'allocation.

Le C++ introduit deux opérateurs : `new` et `delete` qui réalisent les mêmes fonctions. Supposons qu'une classe `stack` soit définie comme dans la section précédente. Dans l'exemple qui suit

- on déclare un pointeur vers un objet de type `stack`
- on alloue un espace mémoire de la taille de `stack`
- on fait pointer `p` vers cet espace
- on libère l'espace pointé par `p`.

```
// Déclaration du pointeur
// l'espace mémoire n'est pas alloué

stack *p;

// On alloue l'espace mémoire avec new
// et on fait pointer p dessus

p = new stack;

// Et on désalloue avec delete

delete p;
```

La syntaxe de `new` et `delete` est un peu plus simple que celle qu'on utilise en C. Nous verrons qu'il y a d'autres avantages.

Une erreur fréquente en C ou en C++ consiste à chercher à écrire dans une zone non allouée. La compilation se passe en général sans problème, et lorsque le programme cherche à écrire dans l'espace non alloué, l'erreur se produit. On obtient alors les messages célèbres du type :

```
Erreur de segmentation
```

Un exercice amusant consiste à écrire un programme qui produit une erreur de ce type.

9 Constructeurs et destructeurs

La classe `stack` donne l'exemple d'un constructeur `stack()` et d'un destructeur `~stack()`. Le constructeur est appelé à chaque fois que le programme crée une nouvelle instance de la classe. C'est par exemple le cas quand on déclare une nouvelle variable de type `stack`.

```
int main()
{
    stack s;
}
```

On peut détecter les appels au constructeur en introduisant dans son code une instruction qui sort un message sur la sortie standard :

```
stack()
{
    size=0;
    for(int i=0; i<=STACK_MAX_SIZE - 1; i++) elems[i]=0;
    cout << "hello" << endl;
}
```

Le mécanisme C++ de signatures permet de définir plusieurs constructeurs pour une même classe. On peut en particulier écrire un constructeur qui prend pour paramètre une instance de la classe `stack` et qui en crée une copie :

```
stack(stack &s)
{
    size = s.getsize();
    for(int i=0; i<=STACK_MAX_SIZE - 1; i++) elems[i]=0;
    for(int i=0; i<=size-1; i++) elems[i]=s.elems[i];
}
```

Ce constructeur porte un nom spécial : c'est le constructeur par copie de la classe `stack`. Quand une variable `stack` est déclarée, comme dans :

```
int main()
{
    stack s;
}
```

c'est le constructeur `stack()` qui est appelé. Quand une fonction passe par valeur des paramètres de type `stack`, c'est le constructeur par copie qui est appelé. On peut le vérifier en insérant une ligne :

```
cout << "coucou" << endl;
```

dans le code de `stack(stack &s)` : on définit une fonction qui ne fait rien, `blob` et on passe par valeur un argument de type `stack`. Le constructeur par copie est appelé, et "coucou" s'affiche sur la sortie standard.

```
void blob(stack s)
{
}
```

```
int main()
{
    stack s;
    blob(s);
}
```

Si la fonction `blob` retournerait une valeur de type `stack`, il y aurait un appel supplémentaire au constructeur par copie :

```
stack blob(stack s)
{
    return s;
}
```

```
int main()
{
    stack s,t; // 2 appels à stack()
    t=blob(s); // 2 appels à stack(stack &s)
}
```

Le résultat de `blob` est copié dans une variable temporaire avant d'être assigné à `t`. Cette copie est créée en utilisant `stack(stack &s)`.

Remarque – La fonction `stack(stack &s)` prend comme paramètre une référence, si elle prenait un paramètre par valeur, on obtiendrait une infinité d'appels au constructeur par copie. La déclaration du constructeur par copie `stack(stack s);`

provoque donc une erreur.

En C, on réserve la mémoire en déclarant une variable comme

```
int i;
```

ou en faisant appel à des fonctions de réservation d'espace mémoire comme `malloc`, `calloc`

```
void *p;  
p=(int*)malloc(sizeof(int));
```

L'espace réservé selon la première technique est libéré dès que la variable `i` n'est plus accessible. En revanche, l'espace réservé par `malloc` reste réservé tant que le programme s'exécute et que le programmeur n'a pas explicitement utilisé les fonctions de désallocation. Il arrive donc souvent qu'une fonction écrite en C présente l'aspect suivant :

```
/* Allocation mémoire */  
void p;  
p=(int*)malloc(sizeof(int));  
...  
/* Fin des allocation */  
  
/* Corps du programme */  
(*p)+=1;  
  
/* Desallocation */  
free(p);
```

Les opérations de gestion mémoire qui précèdent et suivent l'algorithme sont souvent longues et fastidieuses. Une erreur d'allocation produit un plantage immédiat, et une erreur dans la désallocation peut facilement conduire le programme à réserver toute la mémoire du système, et provoquer des catastrophes sérieuses. Mais en plus, le programme devient illisible car rempli d'instructions qui n'ont en fait pas grand chose à voir avec l'algorithme implémenté.

En C++ les constructeurs se chargent en général de l'allocation. Pour la désallocation, on fait appel aux destructeurs. Dès qu'une variable cesse d'être accessible, le destructeur de sa classe est appelé. De cette manière, le code correspondant à l'allocation et la désallocation est réduit à presque rien.

On illustre le rôle des constructeurs et des destructeurs, en donnant une nouvelle implémentation de la classe `stack`. On crée en fait deux classes : une classe `elem`, qui est une boîte contenant un entier et un pointeur, et une classe `stack`, qui implémente la notion de pile. Les éléments de la pile sont maintenant de type `elem`.

```

#include<iostream>;
const int STACK_MAX_SIZE=100;

class elem
{
    public :
    inline elem(){ value=0; next=0; }
    inline elem(const int i){ value=i; next=0; }
    inline void setvalue(const int i){ value=i; next=0; }
    inline int getvalue(){ return value; }
    void setnext(elem &e){ next=&e; }
    elem* getnext(){ return (*this).next; }
    private :
    int value;
    elem *next;
};

class stack
{
    public :
    inline stack(){ size=0; first=0; }
    inline ~stack(){}
    void push(int a);
    int pop();
    int getsize();
    void print();
    private :
    elem *first;
    int size;
};

// Push(a) pose un élément sur la pile
// Le code est caractéristique de la manipulation de
// pointeurs dans les listes chaînées. Dans le code du
// programme principal, on utilise seulement des appels
// s.push(i) et s.pop(), et on ne manipule plus les pointeurs.

```

```

void stack::push(int a)
{
// on réserve un espace de la taille de la variable elem,
// et on déclare un pointeur pe qui pointe sur cette case.

    elem *pe = new elem;
    pe->setvalue(a);
    pe->setnext(*first);
    first=pe;
    size++;
}

int stack::pop()
{
    int return_value=0;
    if(size>=1)
        {
            elem *first_bak=first;
            return_value = first->getvalue();
            first = first->getnext();
            size--;
            delete first_bak;
        }
    else return 0;
}

int stack::getsize(){ return size; }

void stack::print()
{
    elem *stack_ptr=first;
    cout << "Taille de la pile: " << getsize() << endl;
    for(int i=1; i<=size; i++)
        {
            cout << stack_ptr->getvalue() << endl;
            stack_ptr=stack_ptr->getnext();
        }
}

```

```

// Programme principal
// On définit une fonction f qui crée une liste et la
// remplit d'entiers.

int f()
{
    stack s;
    for(int i=1; i<=1000; i++) s.push(i);
    ...
}

int main()
{
    stack s;
    f();
    ...
}

```

Après l'appel de `f`, l'espace réservé pour `s` n'est pas libéré: la variable `s` elle-même n'est plus accessible, mais l'espace réservé derrière `s.first` n'est pas désalloué. Il faut compléter le corps du destructeur `~stack()`: pour cela, on parcourt la liste en désallouant l'un après l'autre les variables de type `elem` qui ont été allouées par la fonction `stack`:

```

stack::~~stack()
{
    elem *p1, *p2=first;
    while(p2->getnext())
    {
        p1=p2;
        p2=p2->getnext();
        delete p1;
    }
}

```

10 Assignement

Nous avons vus dans les définition de la classe `stack`, que le C++ introduit une fonction spéciale appelée constructeur de la classe. Ce langage définit aussi par défaut un opérateur `=` pour les classes définies par l'utilisateur. Considérons par exemple une classe `complex` :

```
#include <iostream>

class complex
{
public :
    float re;
    float im;
    void print()
    {
        cout << re << "+ i " << im << endl;
    }
};

int main()
{
    complex a,b;
    a.re = 1.0;
    a.im = 1.0;

    // Utilisation de l'opérateur d'assignation

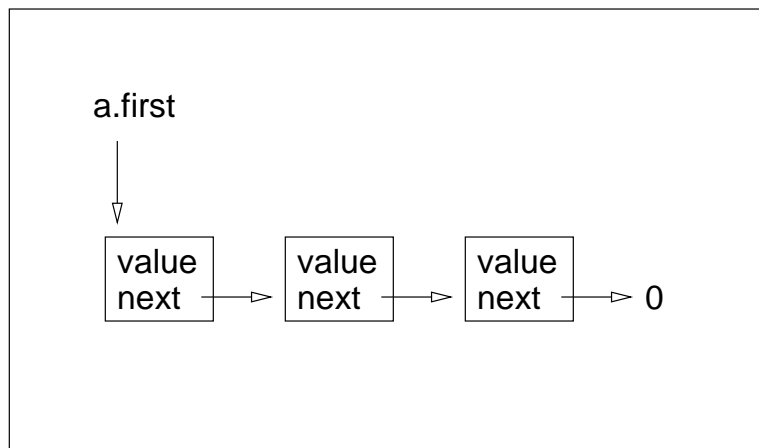
    b=a;

    a.print();
    b.print();
}
```

Ce petit exemple montre que l'opérateur `=`, fait exactement ce à quoi on peut s'attendre, c'est-à-dire une copie membre à membre des données : `a.re` est copié dans `b.re`, et `a.im` dans `b.im`;

Le plus souvent, c'est en effet ce que l'on souhaite faire. Mais pas toujours. Si on considère par exemple une variable de type `stack` implémentée à l'aide de liste chaînée, la copie membre à membre pose un problème. On a représenté

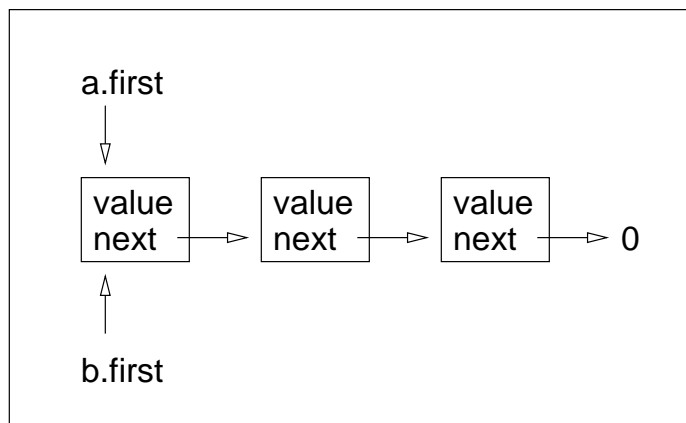
sur la figure suivante une variable `stack` qui contient 3 éléments. Les flèches sont censées représenter les pointeurs. La variable `stack` ne contient, en tant que données membres, qu'un pointeur `first` vers la première case de la liste chaînée et un entier `size` qui donne sa taille.



Après un assignement du type

```
b=a;
```

On a `b.first = a.first`, et `b.size = a.size`. Autrement dit on se retrouve avec une liste chaînée et deux pointeurs sur le premier élément.



Ce n'est sans doute pas ce qu'on l'on souhaitait obtenir. Pour créer deux copies identiques de la liste, il faut définir une nouvelle fonction `=` pour les

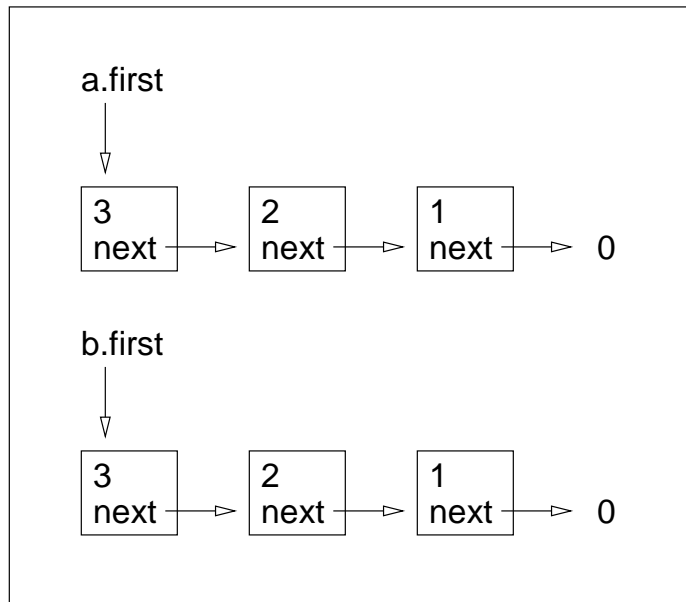
variables de type `stack`. On dit qu'on surcharge l'opérateur `=`. Cela se fait en créant une fonction membre dans la classe `stack::operator=`.

```
void stack::operator=(const stack &s)
{
    elem *ps=s.first;
    elem *p, *last;

    if(ps)
    {
        p=new elem;
        first=p;
        last=p;
        p->setvalue(ps->getvalue());
        size=1;
        ps = ps->getnext();

        while(ps)
        {
            p = new elem;
            last->setnext(*p);
            last=p;
            p->setvalue(ps->getvalue());
            size++;
            ps=ps->getnext();
        }
    }
    else
    {
        size=0;
        first=0;
    }
}
```

On obtient deux listes chaînées distinctes contenant les mêmes valeurs :



Le mécanisme de redéfinition des opérateurs est un élément important du C++ et permet de produire un code très lisible :

```
int main()
{
    stack s,t;
    for(int i=1; i<=3; i++)
        s.push(i);

    // Une seule instruction pour copier dans t
    // les valeurs contenues dans s

    t=s;
    s.print();
    t.print();
}
```

11 Surcharge d'opérateurs

Le fait de redéfinir l'opérateur `*` constitue ce qu'on appelle une surcharge d'opérateurs. Cette technique permet d'écrire des programmes très lisibles. Considérons par exemple la classe `complex`:

```
#include <iostream>

class complex
{
public :
    complex(){ re=0; im=0; }
    complex(const float x, const float y){ re=x; im=y; }
    complex(const complex &z){ re=z.re; im=z.im; }

    float re;
    float im;

    void print()
    {
        cout << re << "+ i " << im << endl;
    }

    complex operator*(const complex a)
    {
        complex c;
        c.re = re * a.re - im * a.im;
        c.im = im * a.re + re * a.im;
        return c;
    }
};

int main()
{
    complex a(1,1),b(0,1),c;
    c=a*b;
    a.print(); b.print(); c.print();
}
```

Le fait de pouvoir écrire `c=a*b` simplifie beaucoup la transcription d'algorithmes et permet une relecture facile. Pour compléter la classe `complex`, il faut encore surcharger les opérateurs `+` et `-`. On peut enfin définir une constante `i` de telle sorte qu'une instruction du type

```
z = a + i b;
```

ait la signification habituelle. On peut enfin utiliser les signatures et surcharger `*` de telle sorte que `x*y` ait un sens pour `x` et `y` de type `float` ou `complex`.

12 Composition et Dérivation

Deux mécanismes permettent la création de nouvelles classes à partir des classes déjà créées : la composition et la dérivation. On dit qu'on utilise la composition quand on crée une classe dont les données membres sont des instances d'autres classes.

```
class A
{
    public:
    int i;
    void blob(){ cout << " Hello " << endl;}
};

class B
{
    public:
    A a;
};

int main()
{
    B b;
    b.a.i=1;
    b.a.blob();
}
```

Dans cet exemple, on déclare dans la classe B une donnée membre qui est une instance de la classe A : `a`. Si `b` est une instance de B, on accède aux membres de `b.a` de façon naturelle : `b.a.i` fait référence à l'entier `i` qui est membre de la classe `a`, membre de `b`. On appelle de la même façon les fonctions membres : l'instruction

```
b.a.blob();
```

imprime " Hello " sur la sortie standard.

Remarque – Quand on déclare une instance de `b`, le compilateur appelle un constructeur. Mais comme la classe B contient une instance de A, un appel au constructeur de A est aussi généré.

Le mécanisme de dérivation permet de créer des classes spécialisées à partir de classes générales.

```
class Base
{
    public:
    int i;
    void f(){ cout << "Appel à Base::f()" << endl;}
    void g(){ cout << "Appel à Base::g()" << endl;}
};

class Derive : public Base
{
    public:
    int j;
    void g(){ cout << "Appel à Derive::g()" << endl;};
};

int main()
{
    Base b;
    Derive d;

    b.g();

    // Que produisent ces appels ?

    d.i = 10;
    d.f();
    d.g();
    d.Base::g();
}
```

Le code qui précède appelle quelques remarques.

- Il faut penser à la classe **Derive** comme à une version particulière de la classe **Base**. On a déclaré dans **Base** une donnée membre de type **int** : **i**. La classe dérivée possède donc aussi une donnée membre **i**.
- Ce qui est vrai des données membres l'est également des fonctions membres. On dit que la classe **Derive** hérite de la classe **Base**.

- Il est possible de redéfinir dans `Derive` la fonction `g`. Cette nouvelle définition annule et remplace celle qui est donnée dans `Base`. Par défaut l'instruction `d.g()` appelle la fonction `g()` définie dans le corps de `Derive`. Pour appeler la fonction `g()` définie dans le corps de `Base`, il faut utiliser l'instruction `d.Base::g()`

On fournit un exemple de dérivation en implémentant de pile triée d'entiers. On s'appuie sur la classe `stack` déjà définie. Il faut réécrire la fonction `push()` : on insère un entier dans la pile triée de telle sorte que les éléments de la pile soient dans l'ordre.

```
void sorted_stack::push(const int i)
{
    switch (size)
    {
        case 0 :
        {
            stack::push(i);
            break;
        }
        case 1 :
        {
            if(i>=first->getvalue())
            {
                elem *q = new elem;
                q->setvalue(i);
                first->setnext(*q);
                size+=1;
            }
            else stack::push(i);
            break;
        }
        default :
        {
            elem *p1, *p2;
            p1=first; p2=p1->getnext();
            while(p1 && p2 &&
                (p1->getvalue() <= i) &&
                (p2->getvalue()<=i))
```

```

        {
            p1=p2; p2=p1->getnext();
        }
        elem *p = new elem;
        p->setvalue(i);
        p1->setnext(*p);
        p->setnext(*p2);
        size+=1;
        break;
    }
}
}

```

```

int main()
{
    sorted_stack s;
    s.push(12);
    s.push(3);
    s.push(125);
    s.push(17);
    s.print();
}

```

Remarque – La variable `size` est déclarée `private` dans la classe `stack`. Pour pouvoir y accéder dans les classes dérivées, il faut changer l'attribut `private` en `protected`.

```

class stack
{
public :
    inline stack(){ size=0; first=0; }
    stack(const stack &s);
    ~stack();
    void push(int a);
    int pop();
    int getsize();
    void print();
}

```

```
protected :  
    elem *first;  
    int size;  
};
```

avec ces déclarations, **size** et **first** ne sont accessibles que par les fonctions membres de **Base** et les fonctions membres de ses classes dérivées.

13 Polymorphisme

On a vu qu'une classe dérivée pouvait modifier le comportement de fonctions définies dans la classe de base. C'est typiquement ce qu'on souhaite pouvoir faire avec les fonctions d'impression: la classe de base définit une fonction membre `print()` qui imprime sur la sortie standard ses données membres. La fonction `print()` est redéfinie dans la classe dérivée pour permettre l'impression des données membres de la classe de base et de la classe dérivée.

```
class A
{
public:
    A(){ a=0; }
    A(const int i){ a = i; }
    int a;
    void print(){ cout << a << endl; }
};

class B : public A
{
public:
    B(){ b=0; }
    B(const int i, const int j){ a=i; b=j; }
    int b;
    void print()
    {
        cout << a << "--" << b << endl;
    }
};

int main()
{
    A a(12);
    B b(15, 37);

    a.print();
    b.print();
}
```

On pourrait s'en tenir là. Mais il y a une difficulté. On a dit qu'une classe dérivée implémente en général une notion plus particulière que celle implémentée dans la classe de base. Dans cette optique, le C++ autorise une déclaration du type

```
int main()
{
    A a(12);
    B b(15, 37);

    A *pa;
    pa = &b;
}
```

Le pointeur `pa` est déclaré comme un pointeur sur un objet de type `A`, mais pointe sur un objet de type `B`. Dans le même esprit, on peut vérifier que le programme suivant ne pose pas de problèmes de compilation :

```
int main()
{
    A a;
    B b(15, 37);

    a=b;
    a.print(); // Appel de A::print();
}
```

Que fait donc le programme suivant ?

```
int main()
{
    A a(12);
    B b(15, 37);

    A *pa;
    pa = &b;
    pa->print();
}
```

Si le compilateur considère le type de `pa`, c'est sans doute `A::print()` qui sera appelée. Mais il est possible que le compilateur examine la nature de l'objet sur lequel `pa` pointe et appelle `B::print()`. En fait, c'est `A::print()`

qui est appelée. Pour permettre l'appel à `B::print()`, il faut déclarer la fonction `print` avec le mot clef `virtual`.

```
class A
{
    public:
    A(){ a=0; }
    A(const int i){ a = i; }
    int a;
    virtual void print(){ cout << a << endl; }
};

class B : public A
{
    public:
    B(){ b=0; }
    B(const int i, const int j){ a=i; b=j; }
    int b;
    void print()
    {
        cout << a << "--" << b << endl;
    }
};

int main()
{
    A a;
    A *pa;
    B b(15, 37);

    a=b; a.print();          // Appel de A::print();
    pa = &b; pa->print();    // Appel de B::print();
}
```

Le compilateur tient compte du type dérivé sur lequel `pa` pointe, pour résoudre l'appel à `print()`. Une fonction précédée du mot-clef `virtual` est dite virtuelle. Cette technique s'appelle le polymorphisme.

Il arrive enfin que la classe de base ne définisse pas la fonction `print()`, mais que celle-ci soit seulement définie dans les classes dérivées. La syntaxe

correspondante est la suivante :

```
class A
{
public:
    A(){ }
    virtual void print()=0;
};

class B : public A
{
public:
    B(){ b=0; }
    B(const int i){ b=i; }
    int b;
    void print(){ cout << "Appel de B::print(): " << b << endl;}
};

class C : public A
{
public:
    C(){ c=0; }
    C(const int i){ c=i; }
    int c;
    void print(){ cout << "Appel de C::print(): " << c << endl;}
};

int main()
{
    A *pa;
    B b(2);
    C c(5);

    pa = &b; pa->print();
    pa = &c; pa->print();
}
```

La fonction `print()` est dite virtuelle pure. En fait la classe `A` est elle-même virtuelle pure: il n'est pas possible d'en créer une instance – on peut seule-

ment créer des pointeurs sur A. Son existence n'a d'intérêt que par les classes qu'on peut en dériver.

14 Template

La classe `stack` dont on a discuté dans les chapitres précédents permet d'implémenter la notion de pile d'entiers. Si on voulait implémenter une pile de `float` il faudrait tout réécrire.

Le C++ permet de créer une pile d'objets dont le type n'est pas défini : c'est la technique des templates. En gros, il faut reprendre le code des classes `elem` et `stack` et remplacer `int` par un type indéfini `T`. On fait précéder la définition de la classe d'une instruction

```
template <class T>
pour exprimer qu'une variable de type T est du type indéfini.
#include <iostream>

template<class T> class elem
{
    public :
    inline elem(const T &e){ value=e; next=0; }
    inline void setvalue(const T e){ value=e; next=0; }
    inline T getvalue()
    {
        return value;
    }
    void setnext(elem &e){ next=&e; }
    elem* getnext()
    {
        return (*this).next;
    }
    void print()
    {
        cout << value << endl;
    }
    private :
    T value;
    elem<T> *next;
};
```

```

template<class T> class stack
{
    public :
        inline stack(){ size=0; first=0; }
        stack(const stack &s);
        ~stack();
        void push(const T a);
        T pop();
        int getsize();
        void print();
        protected :
            elem<T> *first;
            int size;
};

template<class T> stack<T>::~~stack()
{
    elem<T> *p1, *p2=first;

    while(p2->getnext())
    {
        p1=p2;
        p2=p2->getnext();
        delete p1;
    }
}

template<class T> void stack<T>::push(const T a)
{
    elem<T> *pe = new elem<T>(a);
    pe->setnext(*first);
    first=pe;
    size++;
}

```

```

template<class T> T stack<T>::pop()
{
    int return_value=0;
    if(size>=1)
    {
        elem<T> *first_bak=first;
        return_value = first->getvalue();
        first = first->getnext();
        size--;
        delete first_bak;
    }
    else return 0;
}

template<class T> int stack<T>::getsize(){ return size; }

template<class T> void stack<T>::print()
{
    elem<T> *stack_ptr=first;

    cout << "Taille de la pile: " << getsize() << endl;
    for(int i=1; i<=size; i++)
    {
        cout << stack_ptr->getvalue() << endl;
        stack_ptr=stack_ptr->getnext();
    }
}

int main()
{
    stack<float> sf;
    stack<int> si;

    sf.push(3.2); sf.push(6.5); sf.print();

    si.push(2); si.push(12); si.print();
}

```

L'implémentation de ces classes appelle quelques remarques. La première est qu'on a utilisé `<<` dans la fonction `print()`. L'expression

```
cout << x << endl;
```

a un sens pour `x`, `float` ou `int`, mais pas pour `x` instance d'une classe quelconque, `complex` par exemple. Autrement dit, le code

```
int main()
{
    stack<complex> sc;
    sc.print();
}
```

provoque une erreur à la compilation. Pour résoudre ce problème il faut donner un sens à

```
cout << x << endl;
```

quand `x` est du type `complex`: il faut surcharger l'opérateur `<<`.

La deuxième remarque concerne le constructeur `elem()`. Dans l'implémentation de la pile d'entiers, le code de `elem()` était

```
elem(){ value=0; next=0; }
```

Le fait d'assigner 0 à `value` ne peut se faire que parce qu'on sait que `value` est du type entier. L'assignation `value=0`, n'a pas de sens si `value` est `complex`. Il faut donc supprimer cette instruction du constructeur `elem()`. Du coup celui-ci perd un peu d'intérêt, et nous l'avons supprimé de la classe template.

La troisième remarque concerne la séparation en fichiers. Jusqu'à présent, on a écrit tout le code dans un seul fichier `main.cpp`. Mais il est d'usage de créer un fichier pour chaque classe, ou famille de classe. J'aurais par exemple volontiers placé le code de `elem` et le code de `stack` dans un fichier `stack.cpp`, crée un fichier `stack.h` contenant les déclarations des fonctions de la classe, puis placé au début de `main.cpp` une instruction

```
#include "stack.h"
```

Avec les template, on crée un seul fichier `stack.h` qui contient tout le code, et on place dans l'en-tête la même instruction.

Remarquons enfin que modulo ces quelques détails, la classe template se manipule de façon naturelle. On crée une pile d'entiers avec

```
stack<int> si;
```

on pose une valeur au sommet de la pile avec

```
si.push(128);
```

Il existe maintenant une librairie générale : STL qui implémente sous forme de templates les structures fréquemment utilisées en informatique : listes, ensembles, vecteurs, piles, ensembles non triés, etc. et les algorithmes de tri, de recherche usuels. Dès qu'on a affaire à des notions de ce type il faut l'utiliser. Voici par exemple comment on crée et qu'on trie une liste d'entiers.

```
#include <iostream>
#include <list>      // Inclusion des classes STL
#include <algorithm> // Pour utiliser l'algorithme de tri

int main()
{
    list<int> l;
    l.push_back(6); l.push_back(1); l.push_back(8);

    l.sort(); // L'algorithme de tri des listes
              // est déjà écrit dans STL.

    list<int>::iterator l_iter=l.begin();
    while(l_iter!=l.end())
    {
        cout << *l_iter << endl;
        l_iter++;
    }
}
```

15 Exceptions

La gestion des exceptions en C++ se fait de la manière suivante : quand une fonction rencontre une instruction de nature à générer une erreur, elle renvoie à la fonction appelante un objet que celle-ci peut exploiter pour résoudre le problème.

Dans le code qui suit, la fonction `printseq(int upto)` écrit des entiers de 1 à `upto` mais s'arrête et génère une exception si l'entier à imprimer dépasse `MAX`. Cet entier est alors renvoyé à la fonction appelante, ici `main()` qui exploite cet entier dans un bloc `catch`.

La gestion des exceptions se fait donc en trois étapes : la fonction appelante crée un bloc `try` dans lequel les fonctions susceptibles de générer des exceptions sont appelées. Dans ces fonctions, quand une erreur est détectée, un objet est renvoyé vers la fonction appelante avec l'instruction `throw`. Le bloc `catch` exécute le code correspondant à la résolution du problème.

```
void printseq(int upto) throw(int)
{
    const int MAX=10;
    int i=1;
    while (i<=upto)
    {
        if (i >= MAX)
            throw i;          // L'exception renvoyée est i
        cout << i << endl;
        i++;
    }
}

int main(void)
{
    try { printseq(120); }
    catch(int exept)
    {
        cout << "Exception : " << exept << endl;
    }
    return 0;
}
```

Remarque – Le prototype de la fonction est modifié, on fait suivre le nom de la fonction de `throw(int)`.

```
void printseq(int upto) throw(int)
```

L'utilisateur de la fonction est averti de l'existence d'exceptions et de leurs types.

16 Exercices et problèmes

Exercices

1. Ecrire le programme “Hello world!”.
2. Ecrire une fonction qui permute deux entiers.
3. Ecrire un programme qui lit un réel x sur l’entrée standard et qui écrit x^2 sur la sortie standard.
4. Ecrire un programme qui lit un fichier contenant des reels sur une colonne: x_1, x_2, \dots, x_n , et qui écrit $x_1^2, x_2^2, \dots, x_n^2$ dans un fichier.
5. Ecrire un programme qui lit n entiers sur une colonne x_1, \dots, x_n , et qui écrit x_n, \dots, x_1 .
6. Créer une structure `complex` qui implémente la notion de nombre complexe, et une structure `quaternion` qui implémente la notion de quaternion. Ecrire une fonction `print` qui prend comme argument un `complex` ou un `quaternion` et qui imprime sur la sortie standard, le complexe ou le quaternion sous la forme

$$a + ib \quad \text{ou} \quad a + ib + jc + kd$$

selon le type du paramètre.

7. Vérifier l’effet d’une assertion `w=z` quand `w` et `z` sont du type `complex`.
8. Ecrire une classe `vecteur` de dimension 3, et une fonction qui calcule le produit scalaire de deux vecteurs. On passera les arguments par référence.
9. Vérifier que si la variable `i` est de type `const int`, alors elle n’est pas modifiable directement. Essayer de modifier `i` en utilisant des pointeurs sur `i`.
10. Essayer de modifier une variable du type `*const`.
11. Ecrire une classe `complex` qui implémente la multiplication, l’addition des nombres complexes. Utiliser les exceptions pour traiter le problème des divisions par zéro.
12. Ecrire une classe qui implémente la notion d’ensemble fifo: c’est un ensemble ordonné de n entiers: a_1, a_2, \dots, a_n . On peut insérer un nouvel élément `a`. La liste devient a, a_1, a_2, \dots, a_n , ou enlever le dernier élément: a_n .

13. Ecrire une classe qui implémente la notion de polygone. Implémenter les fonctions membres donnant les sommets du polygone, l'équation des côtés.

Problèmes

1. Ecrire une classe qui implémente la notion de champs de vecteur dans un domaine du plan. Ecrire une classe qui implémente la notion de courbe paramétrée. Ecrire un schéma d'Euler pour une équation autonome du type :

$$\dot{x} = \varphi(x).$$

Reprendre ce problème sur la sphère S^2 et dans un domaine de \mathbb{R}^3 .

2. Implémenter en C++ la notion de 1-forme différentielle dans \mathbb{R}^2 . Implémenter la notion de courbe paramétrée. Programmer une fonction qui donne

$$\int_{\gamma} f d\alpha$$

pour α 1-forme différentielle, et γ courbe paramétrée.

3. Implémenter en C++ la notion de 1-forme différentielle dans \mathbb{R}^2 . Implémenter la notion de courbe paramétrée. Programmer une fonction qui détermine pour une forme différentielle

$$p(x,y)dx + q(x,y)dy$$

les courbes paramétrées $\gamma(t) = (x(t), y(t))$ vérifiant pour tout t

$$\int_{\gamma(0)}^{\gamma(t)} p(x(t), y(t)) \dot{x}(t) + q(x(t), y(t)) \dot{y}(t) dt = 0.$$

4. Implémenter en classes C++ les notions de vecteur et de matrice.
 - (a) Surcharger $+$, $*$, $==$, $!=$.
 - (b) Surcharger l'opérateur $[\]$, de telle sorte que si v est un vecteur, $v[i]$ désigne la i -ème composante de v , et si m est une matrice, $m[i][j]$ désigne le coefficient de m qui se trouve à la ligne i et à la colonne j .

(c) Programmer l'exponentielle de matrice.

5. Implémenter en classes C++ les notions de vecteur et de matrice.

(a) Surcharger $+$, $*$, $==$, $!=$.

(b) Surcharger l'opérateur $[]$, de telle sorte que si \mathbf{v} est un vecteur, $\mathbf{v}[\mathbf{i}]$ désigne la \mathbf{i} -ème composante de \mathbf{v} , et si \mathbf{m} est une matrice, $\mathbf{m}[\mathbf{i}][\mathbf{j}]$ désigne le coefficient de \mathbf{m} qui se trouve à la ligne \mathbf{i} et à la colonne \mathbf{j} .

(c) Résoudre le problème d'optimisation quadratique

$$\min \mathbf{q}(\mathbf{x}, \mathbf{x})$$

$$\langle \mathbf{a}, \mathbf{x} \rangle = \mathbf{b}$$

6. Ecrire une classe qui implémente la notion de loi de probabilité d'une variable aléatoire réelle – on peut supposer que la loi a une densité ou non.

(a) Implémenter les méthodes qui donnent l'espérance et la variance.

(b) Ecrire une fonction qui prend comme paramètre une loi \mathbf{d} et un entier \mathbf{n} , et qui retourne un échantillon

$$(X_1(\omega), X_2(\omega), \dots, X_n(\omega))$$

où X_1, \dots, X_n sont indépendantes et de loi \mathbf{d} .

7. Ecrire une classe `signal` qui implémente la notion de suite indexée dans \mathbb{Z} , et dont tous les termes sauf un nombre fini sont nuls.

(a) Surcharger les opérateurs $+$, $*$, $==$, $!=$, $= \dots$

(b) Surcharger l'opérateur $[]$ de telle sorte que si \mathbf{s} est de type `signal` et $\mathbf{i} \in \mathbb{Z}$, $\mathbf{s}[\mathbf{i}]$ désigne le \mathbf{i} -ème terme de \mathbf{s} .

(c) Implémenter la notion de filtre linéaire.

(d) Ecrire une méthode de la classe `signal` qui prend comme paramètre un filtre linéaire, et retourne le signal filtré.

8. On considère D le disque de centre $(0,0)$ de rayon 1, et f une fonction continue définie sur D . Utiliser la librairie de générateurs de nombres aléatoires disponible sur www.robertnz.net pour résoudre le problème de Dirichlet

$$\begin{aligned} \Delta u &= -f && \text{sur } D \\ u &= 0 && \text{sur } \partial D \end{aligned}$$

Remarque – On rappelle que la solution du problème précédent vérifie

$$u(x) = \mathbb{E} \left[\int_0^{T_D^x} f(x + B_t) dt \right]$$

où x désigne un point de D , $(B_t)_{t \geq 0}$ le mouvement brownien standard, et T_D^x , le temps de sortie de D .

On peut donc utiliser la loi des grands nombres et considérer pour chaque point $x \in D$ un grand nombre N de trajectoires issues de x . Calculer pour chacune le temps de sortie T_D^x , et

$$\int_0^{T_D^x} f(x + B_t) dt.$$

La moyenne de ces N valeurs fournit une approximation de $u(x)$.

9. Utiliser la librairie de générateurs de nombres aléatoires disponible sur www.robertnz.net pour simuler un processus X_n tel que :
- $X_0 = 0$
 - les différences $Y_{n+1} = X_{n+1} - X_n$ soient indépendantes gaussiennes centrées réduites.
- Soit $a > 0$ et τ_a le temps de sortie du segment $[-a, a]$. Evaluer $\mathbb{E}[\tau_a]$.

10. Ecrire une classe qui implémente la notion de fonction définie sur un segment. Mettre en œuvre une méthode de type Monte-Carlo pour l'évaluation de

$$\int_a^b f(x) dx$$

en utilisant la librairie de générateurs de nombres aléatoires disponible sur www.robertnz.net.

11. Ecrire une classe qui implémente la notion d'arbre dont les noeuds et les feuilles contiennent des valeurs de type `int`. On donnera des méthodes permettant d'agrandir l'arbre et de le parcourir.

12. Utiliser l'interface `C++` de la librairie disponible sur swox.com/gmp/ pour calculer $1000!$. Ecrire le même programme sans utiliser l'interface `C++`.

13. Une transformation de Möbius de \mathbb{C} est une transformation du type

$$w = \frac{az + b}{cz + d} \quad \text{avec } a, b, c, d \in \mathbb{C}, ad - bc \neq 0.$$

(a) Programmer une classe qui implémente la notion de nombre complexe.

(b) Programmer une classe qui implémente la notion de transformation de Möbius.

On considère m une transformation de Möbius et $G(m)$ le groupe engendré par m . A quelle conditions sur a, b, c, d les orbites sous l'action de $G(m)$ sont elles finies? Vérifier!

14. Ecrire un programme `C++` qui calcule l'enveloppe convexe de n points du plan. Quelles classes faut-il définir?

15. Programmer une classe qui implémente la notion de polygone dans \mathbb{R}^2 . Programmer les méthodes de construction de telle sorte que l'utilisateur de la classe puisse accéder aux points extrémaux, au cône asymptotique et aux formes linéaires définissant le polygone.

16. Programmer une classe qui implémente la notion de polygone compact dans \mathbb{R}^2 . Ecrire une fonction qui retourne le projeté d'un point $x \in \mathbb{R}^2$ sur un polygone P .

Sections

1	Introduction	1
2	Entrées/Sorties	3
3	Signatures	5
4	Conversions	7
5	const	9
6	Références	11
7	Classes	13
8	Allocation mémoire	16
9	Constructeurs et destructeurs	18
10	Assignement	24
11	Surcharge d'opérateurs	28
12	Composition et Dérivation	30
13	Polymorphisme	35
14	Template	40
15	Exceptions	45
16	Exercices et problèmes	47
	Sections	52
	Index	54
	Bibliographie	54

Index

cerr, 3
cin, 3
classe, 13
 constructeur, 15
 destructeur, 15
 instance, 15
 interface, 15
 virtuelle, 37
composition, 30
cout, 3

delete, 16
derivation, 30

fstream, 3

heritage, 31

inline, 15
iostream, 3

make, 2

new, 16

polymorphisme, 37
private, 15
protected, 33
public, 15

signature, 5
struct, 5
surcharge, 26, 28

template, 40

virtual, 37

Références

- [1] B. Eckel. *Thinking in C++*, volume 1 and 2. Prentice Hall, 2000.
- [2] G. Satir and D. Brown. *C++ The Core Language*. O'Reilly & Associates, 1995.
- [3] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Pub. Co., 2000.