Chapite 1 : C++

1.1.1 Introduction

- programmation fonctionnelle (LISP, CAML, …)
- programmation logique (PROLOG)
- programmation par contraintes : constraint satisfaction problem
- programmation procedurale (Pascal, C, …)
- programmation orientée objet (C++, …)

Intérêts de la POO

- concepts intéressant pour la modélisation : héritage
- polymorphisme
- encapsulation (private, public, …)

N. Wirth (Langage Pascal) : Un programme est une structure de données + un algorithme.

-> Un programme est un algorithme qui agit sur une structure de données.

prog = structdedonnées.algo(paramètre)

1.1.2 Historique

C++ développé par Bjarne Stroustrup (1980-1983)
En 1985 : le C++ programming language (suivi de normalisation)
En 1987 : ANSI
Le C++ est un meilleur C

1.2 Différences avec le C

1.2.1 Fichiers (.cpp ou .cc et .h ou .hpp)

compilateur g++ pour GNU et icpc pour intel

1.2.2 espaces de nommage (namespace) : <nom> { … }

`using namespace <nom>` ou `<nom>::Classe`

exemple : pour la Standard Template Library -> using std; ou std::...;

1.2.3 qualificateur const

- Définition de constantes : `const int MAXI = 100;` à la place de `#define MAXI 100;`
- Sous programmes : un paramètre est non modifiable et les attributs de la classe sont non modifiables.

1.2.4 qualificateur enum

- environ une définition de type + constantes.

ex :

```
enum {zero, un, deux, trois, quinze = 15, par defaut 0, 1, 2 3 seize};
      Jour j = lundi; equivalent à int j = -1;
      unsigned int k = lundi; // 4.296.967.295
```

1.2.5 new et delete

```
      new -> delete
```

```
malloc -> free
int * ptr;
ptr = new int;
delete ptr;
ptr = new int[20];
delete [] ptr;
```

1.2.6 opérateurs de référence &

remplacer `->` (des pointeurs)
par `.`

1.2.7 paramètres par défaut des sous programmes

```
void f(int n, int m=1) { }

f(1,5)
f(2) équivaut à f(2,1)
```

1.2.8 entrées / sorties

`cin >> / cerr << / cout <<`
utiliser `getline(char *b, int max, char delim)`

1.2.9 typage

introduction de nouveaux operateurs
`int a = (int)` …

`_static_cast` : type1 → type2
_const_cast : ajouter ou supprimer un caractère constant (const)
ex : `const int a = 1;`
`int b = const_cast<int>(a);`
`_dynamic_cast` : héritage
`_reinterpret_cast` : pour des types différents. ex : `int → float`

1.3 La classe

1.3.1 Déclaration

1.3.2 Constructeur

```
class A {
      int a;

      A(int n) { a=n;}
      // OU
      A(int n) : a(n) {}
};
```

1.3.3 Destructeur

On libère les ressources allouées

1.3.4 Constructeur par recopie

```
class Vector {
      int size;
      int * tab;
      Vector(int s) {
            size = s;
            tab = new int[s];
      }
      Vector(Vector& v) {
            size = v.size
            tab = new int [size]
            for (int i=0; i<size; i++){
                  tab[i]=v.tab[i]
            }
      }
};
```

1.3.5 recopie d'objet, redefinition de l'opérateur d'affection

```
Vector v1(10);
Vector v2(100);
v2 = v1; // le = à un comportement par défaut

Vector& operator=(Vector& v) {
      if(this != &v) {
            delete [] tab;
            size = v.size
            tab = new int [size]
            for (int i=0; i<size; i++){
                  tab[i]=v.tab[i]
            }
      }
      return *this;
};
```

1.3.6 tableaux d'objets

```
Vector * tab;
tab = new Vector[10];// marche uniquement si dans la classe Vector il y a un
```

constructeur par défaut

1.3.7 Attributs statiques

commun a toutes les instances

a.h
```
class A {
      static int a
};
```

a.cpp
```
#include “a.h”
int A::a = ∅
```

1.3.8 fonction amie

```
class A;

class B {
      friend class A;
};
```
La classe A peut accéder aux attributs de B

* fn amies
      class Vecteur;
      class Matrice;

```
class Vecteur
      friend Vecteur * prod(Vecteur *v, Matrice *m); //déclaration

class Matrice
      friend Vecteur * prod(Vecteur *v, Matrice *m); //déclaration

Vecteur *prod(Vecteur *v, Matrice *m) {
…. // implementation
}
```

1.3.9 Les opérateurs

```
      class String {...}
      String s, t, u;
      u = s + t; // redéfinition de l’opérateur + (pour une concaténation)
```

      2 possibilités : soit surcharge d’opérateur
                       soit fonction amie

```
class Integer {
      int value:
            // surcharge
      Integer operator+(const Integer & a) {
            Integer sun(value+a.value);
            return sum
      }
}; // ← Ne pas oublier ce putin de ‘;’
```

// Fonction amie

```
friend Integer operator+(Integer &a, Integer&b){
en dehors de la classe
      Integer sum(a.value+b.value)
      return sum;
}

Integer a(1), b(2), c;
      c = a+b;
```

surcharge : `c = a.+b;`

fonction amie : `c =+ (a,b)`

*autres opérateurs

```
      ++c; // pré incrémentation
      c++; // post incrémentation
```

// pré incrémentation

```
Integer& Integer::operator++() {
      ++value;
      return this;
}
```

//post incrémentation

```
Integer Integer::operator(int) {
      return ++(*this);
}
```

1.3.10 Surcharge []

```
      class Vector {
            int& operator[](int n) {
                  return tab[n];
            }
      };
```

cout << v[3]; mais pas v[3] = 50; sauf si ajout de &

1.3.11 Passage d'objet en argument de fonction

Ecrire une fonction qui fait la somme de 2 vecteurs

v, w ⇒ v = v + w

```
int somme(Vector v, Vector w) {
      int s = 0;
      for (i=0; i<v.getsize(); ++i) {
       v[i] = v[i] + w[i];
       s+= v[i];
      }
      return s; // ATTENTION EN SORTIE RIEN N'EST MODIFIE (passage par
                                                          valeur)
}
```

/!\ c'est du passage par valeur rien n'est modifié voir les deux solutions ci-dessous /!\

2 solutions :

- passer par pointeurs

```
somme(Vector *v, Vector *w) {
      v → getsize()
      (*v)[i] ...
}
```

ex d'appel : somme(&v1, &v2)

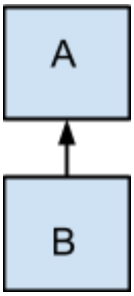
- passage par reference

```
somme(Vector &v, Vector &w){
      v.getsize();
      v[i] ...
}
```

ex d'appel : somme(v1, v2)

1.4 L'Héritage

Hiérarchie de classe.



(La classe fille, ou sous classe, B hérite de la classe mère, ou super classe, A)

A est une classe générale (généralisation)

B est une classe spécialisée (spécialisation)

(c1)
Héritage en diamant étroit -> B et C hérite de A et D hérite de B et C.

1.4.3 Fonctions virtuelles
(c2)
tableau/liste formes
Forme*tab[100]

(COURS 2 / 30 septembre 2013)
1.5 Généricité (template)
Définition : Paramétrer une structure, une fonction avec un type qui sera instancié par la suite.

1.5.1 Fonctions génériques

```
int min(int a, int b) {
        return (a<b) ? a : b;
}
```

****

```
template<class T>
T min(T a, T b) {
        return (a<b) ? a : b;
}
```

****

```
int c = min(3,4); // utilise la fn min
float a = min(3.14,0.66); // utilise le template avec float ) la place de T
```

1.5.2 Classes génériques

```
template<int maxSize, class T>
class Vector {
        protected :
                int size_max;
                int size;
                T * tab;
        public :
                Vector(){
                        tab = new T[size_max = maxSize];
                        size = 0
                }
};
```

exemple si déclaré dans .cpp
template<.....,.....>

```
Vector<maxSize, T>::Vector(){
…
}
Vector<200, int> v1;
Vector <200, Vector<50,int> >V2 ;
```

1.6 Les exceptions

Exemple STL

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

class Integer {

        class DivideByZero:public exception {
                public :
                        virtual const char*whar(void) const throw() {
                                return « Divisionpar 0 » ;
                        }
        } ;
        void divide (int n){
                if (n==0){
                        throw DivideByZero() ;
                }
                value /= n ;
        }

        throw(){
                try{
                        Integer i(10) ;
                        i.divide(null) ;
                }catch(Integer::DivideByZero &e){
                        cout << e.what() << endl ;
                }
        }
};
```

1.7) STL

Standard Template Library

Alexander Stepanov & Meng Lee 1992 Hewlett Packard Lab

#include <_____>

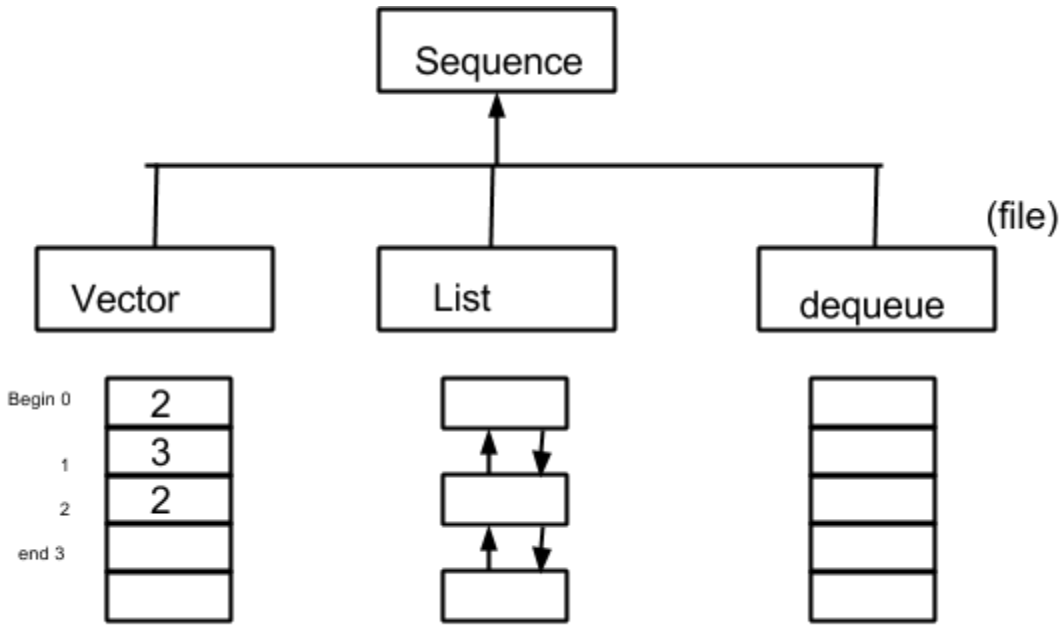
basée sur la générécité

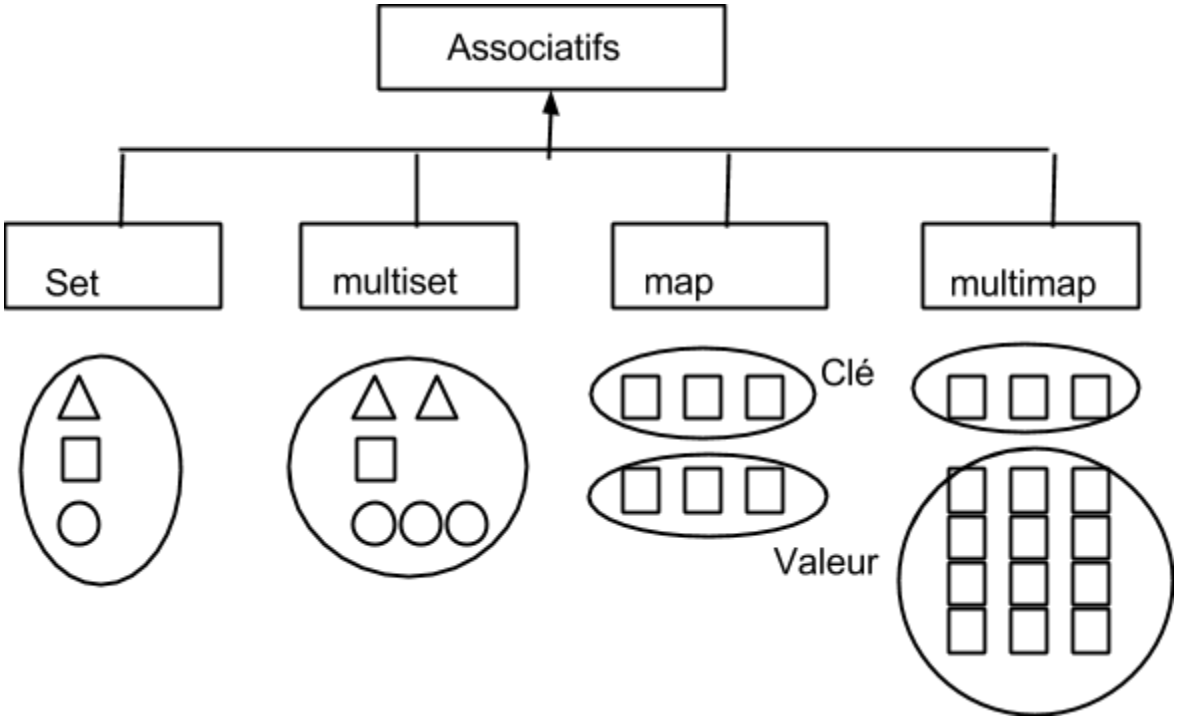
5 composant principaux

- les containers (list, vector,...)
- les algorithmes
- les itérateurs parcours de containers
- les objets fonctions manipulation/traitements
- les adaptateurs, composant qui offre des interfaces

1.7.1) String
implementation = vector de caractères
redéfinition des opérateurs + : concaténation

1.7.2) Containers
- sequence containers
    stockage linéaire (tableau, liste)
- associative containers
    accès pas clef (Hash, mapping)





set permet de stocker des élément différents (pas deux fois le même)
multiset : même que set sauf que l'on peut avoir au moins deux objets différents.
Mapping : ensemble clef → valeur
multi-map : ensemble clef → valeur. Mais ici la clef peut envoyer un Vector , un tableau …

1.7.3) Les itérateurs

iterator begin() // premier élément d'un containers.
iterator end() // Ce qui se trouve après le dernier élément.

const_iterator begin()
const_iterator end()

redéfinition de ++ (passer au suivant), * (contenu), …

1.7.4) La liste

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

typedef list<int> listInt;
int zero() {return 0; }
void cube(int n) { cout << (n*n*n*) << endl; }
```

```
int main() {
	listInt l;
	unsigned int i;

	for(i=0; i<10 ; ++i) {
		l.push_back(rand()%100);
	}
	l.insert(l.end(), -1); // ajoute la valeur -1 à la fin de la liste

	listInt::iterator iter;
	for (iter=l.begin() ; iter != l.end() ; ++iter) {
		cout << (*iter) << endl; //(*iter) → élément/valeur pointé par iter
	}

	for_each(l.begin(), l.end(), cube)t;

	copy(l.begin(), l.end(), ostream_iterator<int>(cout, " ")); // Même chose que le premier for
}
```

1.7.7 Exemples

* convertir en majuscule

```
	string s = ".....";
	transform(s.begin(), s.end(), s.begin(), (int(*)(int)) toupper);
```

*somme des éléments d'un tableau

```
	int tab[] = {1,2,3,4,5,...};
	int sum = accumulate(&tab[0], &tab[4], 0);
```

*tokenizer

```
	string s = "cou cou toto";
	string word;
	istringstream stream(s);
	while(stream >> word) {
		cout <<word << endl;
	}
```

*mapping : compter le nombre d'occurrences de chaque mot dans une chaîne

```
map<string,int> occ;
while(stream >> word) {
        if(occ.find(word) == occ.end()) {
                occ[word] = ∅;
        }
        ++occ[word];
}

map<string, int>::iterator iter;
for(iter = occ.begin() ; iter != occ.end() ; ++iter) {
        cout << (*iter).first << " apparait "
        << (*iter).second << "fois"
        << endl;
}
```