Programmation Orientée Objet: C++ - Cours 7

M1 IM/MF-MPA - Année 2023-2024

Planning POO

Mardi 05/12 (auj.)	Dernier cours C++
Lundi 11/12	TP11 (6ème TP C++)
Lundi 18/12	TP12 (TP noté C++ - 30% de la note finale)

Héritage (suite)

Conversion

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit
       a, b" << endl;
  A a(5);
  B b(6,7);

  cout << "On affiche a,
       b" << endl;
  a.affiche();
  b.affiche();
  cout << "on dit a = b"
       << endl;
  a = b;
  cout << "On affiche a,
       b" << endl;
  a.affiche();
  b.affiche();
}
```

Listing 3: main_AB4.cpp

```
./a.out
On construit a, b
Constr A
Constr A
Constr B
On affiche a, b
A : 5
B : A : 6
7
on dit a = b
On affiche a, b
A : 6
B : A : 6
7
```

Néanmoins, comme le montre l'affichage de notre programme, lorsque la variable *b* est convertie, elle perd une partie de ses données membres - celles de *B* - pour ne garder que les données membres héritées: celles de *A*. Ce qui est normal car *a* est de type *A*.

Comme on le voit, quand on réaffiche la variable *a*, sa donnée privée a bien pris la valeur de la partie *A* de *b*.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1 : AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2 : AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit b" <<
       endl;
  A* pa;
  B b(6,7);

  pa = &b;
  cout << "On affiche pa, b"
       << endl;
  pa->affiche();
  b.affiche();
  cout << "ra: reference sur
       b" << endl;
  A &ra = b;
  cout << "On affiche ra, b"
       << endl;
  ra.affiche();
  b.affiche();
}
```

Listing 4 : main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr B
On affiche pa, b
A : 6
B : A : 6
7
ra: reference
    sur b
On affiche ra, b
A : 6
B : A : 6
7
```

Le mécanisme avec les pointeurs et les références reste similaire.

Si on définit un pointeur sur A, on peut l'initialiser avec une adresse de type pointeur sur B.

De même, une référence sur A peut prendre l'adresse d'un objet de type B.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1 : AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2 : AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit b" <<
       endl;
  A* pa;
  B b(6,7);

  pa = &b;
  cout << "On affiche pa, b"
       << endl;
  pa->affiche();
  b.affiche();
  cout << "ra: reference sur
       b" << endl;
  A &ra = b;
  cout << "On affiche ra, b"
       << endl;
  ra.affiche();
  b.affiche();
}
```

Listing 4 : main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr B
On affiche pa, b
A : 6
B : A : 6
7
ra: reference
    sur b
On affiche ra, b
A : 6
B : A : 6
7
```

On commence à entrevoir une chose intéressante: un pointeur ou une référence peuvent pointer vers des objets qui ne sont pas de leur type, mais d'un type dérivé.

C'est plus intéressant que pour les objets car le type d'un objet ne varie pas, mais si on initialise le pointeur avec un autre type, les valeurs supplémentaires sont perdues lors de la conversion.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1 : AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2 : AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit b" <<
       endl;
  A* pa;
  B b(6,7);

  pa = &b;
  cout << "On affiche pa, b"
       << endl;
  pa->affiche();
  b.affiche();
  cout << "ra: reference sur
       b" << endl;
  A &ra = b;
  cout << "On affiche ra, b"
       << endl;
  ra.affiche();
  b.affiche();
}
```

Listing 4 : main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr B
On affiche pa, b
A : 6
B : A : 6
7
ra: reference
    sur b
On affiche ra, b
A : 6
B : A : 6
7
```

Alors qu'un pointeur (ou une référence) n'est qu'une adresse qui pointe vers un type en vérité plus "grand" que le type réel du pointeur.

Limitations

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1 : AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2 : AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* a = new B(5,6);

  a->affiche();

  return 0;
}
```

Listing 5 : main_AB43.cpp

```
./a.out
Constr A
Constr B
A : 5
```

On reprend les classes A et B de l'exemple précédent.

Dans le *main*, on déclare un pointeur sur A, qu'on initialise avec un objet de type B. C'est donc le constructeur de B qui est appelé. C'est valide comme on l'a vu précédemment.

On appelle alors la fonction *affiche* de notre pointeur et $\cdots$ déception, c'est la fonction *affiche* d'un objet de type A qui est appelée, et non pas celle d'un objet de type B, même si c'est bien un objet de ce type qui a été créé.

Limitations

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* a = new B(5,6);

  a->affiche();

  return 0;
}
```

Listing 5: main_AB43.cpp

```
./a.out
Constr A
Constr B
A : 5
```

Cela vient du fait que les fonctions appelées sont "figées" lors de la compilation. Et pour le compilateur, un pointeur sur un objet correspond au type de cet objet, et ce sont donc les fonctions de la classe de cet objet qui seront appelées. Même si lors de l'exécution, l'objet pointé est en réalité "plus grand".

On verra dans la suite un mécanisme qui permet de passer outre cette difficulté.

Polymorphisme

Définition

Nous avons vu qu'un pointeur sur une classe parente pouvait recevoir l'adresse de n'importe quel objet dérivant la classe parente.

Il y avait néanmoins une contrainte: lorsque l'on appelait une fonction de l'objet pointé, c'était la fonction de la classe parente qui était appelée et pas la fonction de l'objet réellement pointé.

Cela provient du fait qu'à la compilation, le compilateur ne connaît pas le type de l'objet réellement pointé, et se base uniquement sur le type du pointeur. Il inclura, dans le code compilé, les appels aux fonctions de ce type, qui correspond au type de la classe parente. Il s'agit d'un typage statique.

Définition

C++ sait faire mieux que cela et permet un typage dynamique de ces objets.

Lors de la compilation, il sera alors mis en place un mécanisme permettant de choisir au moment de l'exécution la fonction qui sera appelée.

Il s'agit du Polymorphisme.

Des objets de types différents peuvent être pointés par le même pointeur et l'exécution du code se fait de manière cohérente avec les types réellement pointés.

Pour cela, nous allons voir un nouveau type de fonctions membres: les fonctions virtuelles.

Fonctions virtuelles

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 1 : AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 2 : AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* pa;
  B b(6,7);

  pa = &b;
  pa->affiche();
}
```

Listing 6 : main_AB44.cpp

```
./a.out

Constr A
Constr B
A : 6
```

Dans cet exemple, nous rappelons le problème.

Dans le *main*, on crée un pointeur sur un objet de type A. Mais celui-ci pointe en réalité sur un objet de type *B* par le jeu des conversions implicites.

Lorsqu'on appelle la fonction *affiche*, c'est celle de *A* qui est appelée et non celle de B.

Fonctions virtuelles

```
#ifndef __AB7__HH__
#define __AB7__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual void affiche();
        // fonction
  // virtuelle
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche(); //
       définie pour B
};
#endif
```

Listing 7: AB_7.hh

```
#include "AB_7.hh"

A::A(int a)
{
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

void B::affiche()
{
  A::affiche();
  cout << "B : ";
  cout << _b << endl;
}
```

Listing 8: AB_7.cpp

```
#include "AB_7.hh"

int main()
{
  A* pa;
  B b(6,7);

  pa = &b;
  pa->affiche();
}
```

Listing 9: main_AB7.cpp

```
./a.out

A : 6
B : 7
```

On se contente ensuite d'ajouter le mot clé *"virtual"* à la déclaration de la fonction *affiche*.

Lors de l'appel dans *main*, c'est maintenant la fonction de *B* qui est appelée!

Que s'est-il passé?

Fonctions virtuelles

Le mot clé "virtual" placé dans la déclaration d'une fonction permet de rendre une fonction "virtuelle".

Même si visuellement, il semblerait que peu de chose ait changé dans le code, en vérité un système relativement complexe a été mis en place par le compilateur pour obtenir un comportement cohérent dans le cas du polymorphisme : c'est en effet la fonction membre du type réel de l'objet qui est appelée et plus celle du type du pointeur.

Fonctions virtuelles - Limitations

Les fonctions virtuelles ont néanmoins quelques règles à respecter:

- Seule une fonction membre peut être virtuelle. Les fonctions "ordinaires" ou amies sont exclues de ce mécanisme.
- Un constructeur ne peut pas être virtuel. En effet, un constructeur est appelé pour construire une classe. Cela n'aurait pas trop de sens qu'en réalité, il construise une autre classe $\cdots$
- En revanche, un destructeur peut être virtuel.

Fonctions virtuelles - Destructeur

```
#ifndef __AB8__HH__
#define __AB8__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  ~A();
};

class B: public A
{
private:
  int _b;
public:
  B(int, int);
  ~B();
};
#endif
```

Listing 10: AB_8.hh

```
#include "AB_8.hh"

A::A(int a)
{
  _a = a;
}

A::~A()
{
  cout << "Destr A" << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

B::~B()
{
  cout << "Destr B" << endl;
}
```

Listing 11: AB_8.cpp

```
#include "AB_8.hh"

int main()
{
  A* pa = new B(2,3);

  delete pa;
}
```

Listing 12: main_AB8.cpp

```
./a.out

Destr A
```

Que se passe-t-il lorsqu'un destructeur n'est pas virtuel dans cet exemple?

On construit un objet de type *B* avec *new*. Il faudra donc le détruire.

Mais son adresse est stockée dans un pointeur de type A*.

C'est donc le destructeur de *A* qui est appelé! Et l'objet *B* n'est pas entièrement détruit ...

Fonctions virtuelles - Destructeur

```hh
#ifndef __AB9__HH__
#define __AB9__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual ~A();
};

class B: public A
{
private:
  int _b;
public:
  B(int, int);
  ~B();
};
#endif
```

Listing 13: AB_9.hh

```cpp
#include "AB_9.hh"

A::A(int a)
{
  _a = a;
}

A::~A()
{
  cout << "Destr A" << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

B::~B()
{
  cout << "Destr B" << endl;
}
```

Listing 14: AB_9.cpp

```cpp
#include "AB_9.hh"

int main()
{
  A* pa = new B(2,3);

  delete pa;
}
```

Listing 15: main_AB9.cpp

```
./a.out

Destr B
Destr A
```

On déclare maintenant le destructeur de A comme étant virtuel.

Lors de l'exécution, c'est donc bien le destructeur de B qui est appelé.

Intérêt

```
#ifndef __AB7__HH__
#define __AB7__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual void affiche(); //
      fonction
  // virtuelle
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche(); // définie
      pour B
};
#endif
```

Listing 7 : AB_7.hh

```
#include "AB_7.hh"

int main()
{
  A* tab[2]; // tableau de 2
      pointeurs sur A
  tab[0] = new A(5);
  tab[0]->affiche();

  tab[1] = new B(7,9);
  tab[1]->affiche();

  delete tab[0];
  delete tab[1];
}
```

Listing 16 : main_AB72.cpp

Un des intérêts du polymorphisme est de pouvoir créer des tableaux de pointeurs sur une classe.

Ici, à l'intérieur du tableau *tab*, on peut avoir une adresse d'un objet *A* ou d'un objet B, ce qui permet de manipuler différents types de données.

On n'oublie pas le *delete* ici (sans [])!

```
#include "AB_7.hh"

A::A(int a)
{
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

void B::affiche()
{
  A::affiche();
  cout << "B : ";
  cout << _b << endl;
}
```

Listing 8 : AB_7.cpp

```
./a.out

A : 5
A : 7
B : 9
```

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour B
};
#endif
```

Listing 17: AB_11.hh

Les classes abstraites en POO sont des classes qui n'ont pas pour but d'être instanciées directement.

Il s'agira alors pour l'utilisateur de la classe de créer une classe et d'hériter de celle-ci en créant le code supplémentaire si besoin.

Pour cela, le C++ introduit des fonctions virtuelles dites "pures", c'est-à-dire qu'on ne donne pas de définition à cette fonction et c'est la classe fille qui devra définir cette fonction.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour
       B
};
#endif
```

Listing 18 : AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* a = new A(); // ko!
  a->affiche();

  A* b = new B();
  b->affiche();
}
```

Listing 19 : main_AB12.cpp

```
  g++ AB_7.cpp main_AB12.cpp -o main_AB12
main_AB12.cpp: In function 'int main()':
main_AB12.cpp:5:16: error: invalid
new-expression of abstract class type 'A'
    5 |   A* a = new A();
      |                ^
In file included from main_AB12.cpp:1:
AB_12.hh:7:7: note:
because the following virtual functions
are pure within 'A':
    7 | class A
      |       ^
      AB_12.hh:12:16: note:
      'virtual void A::affiche()'
      12 |   virtual void affiche() = 0;
      // fonction virtuelle pure
```

Dans cet exemple, dans la fonction *main*, on essaie d'instancier un objet de type A.

Le compilateur refuse car on essaie d'instancier une classe abstraite.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour
       B
};
#endif
```

Listing 18: AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* a = new A(); // ko!
  a->affiche();

  A* b = new B();
  b->affiche();
}
```

Listing 19: main_AB12.cpp

```
  g++ AB_7.cpp main_AB12.cpp -o main_AB12
main_AB12.cpp: In function 'int main()':
main_AB12.cpp:5:16: error: invalid
new-expression of abstract class type 'A'
    5 |   A* a = new A();
      |                ^
In file included from main_AB12.cpp:1:
AB_12.hh:7:7: note:
because the following virtual functions
are pure within 'A':
    7 | class A
      |       ^
      AB_12.hh:12:16: note:
      'virtual void A::affiche()'
      12 |   virtual void affiche() = 0;
      // fonction virtuelle pure
```

Une classe abstraite est une classe contenant au moins une fonction virtuelle pure.

On ne peut pas instancier cette classe directement.

La classe *B* hérite publiquement de *A* et redéfinit la fonction *affiche*. On pourra alors instancier un objet de type *B*.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; // fonction
        virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour B
};
#endif
```

Listing 18: AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* b = new B();
  b->affiche();
}
```

Listing 20: main_AB12ok.cpp

```
  ./a.out

B : 0
```

STANDARD TEMPLATE LIBRARY

LA STL

Le C++, tout comme le C et de nombreux langages, possède une librairie disponible dés l'installation, connue sous le nom de librairie standard.

On connaît déjà quelques éléments de cette librairie notamment par les *#include <cmath> <iostream> <string>* etc.

Une partie de cette librairie concerne des versions stables, optimisées et testées de conteneurs, d'itérateurs et d'algorithmes sur ces conteneurs. Cette sous-librairie est connue sous le nom de *Standard Template Library* ou *STL*.

C'est une librairie puissante conçue à base de classes templates, dont nous allons maintenant étudier quelques éléments.

LA STL

LA CLASSE VECTOR

```
#include <iostream>
#include <vector>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::vector<double> v(tab, tab + 5);
  v.push_back(10);
  std::cout << v[0] << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

Nous avons déjà plus ou moins construit une classe *vector* telle qu'implémentée dans la STL.

On en trouve la documentation complète sur *cplusplus.com*

http://www.cplusplus.com/reference/vector/vector/

Ci-contre, un exemple d'utilisation.

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

La classe *std::list* s'apparente en terme d'utilisation à la classe *std::vector*.

Il y a néanmoins des différences d'implémentation entre un *vector* et une *list* qui font qu'on préfèrera l'une ou l'autre classe en fonction de l'utilisation de celle-ci.

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

La classe *list* telle qu'implémentée dans la STL est ce qu'on appelle une liste **doublement chaînée**.

On remarque qu'il n'y a pas de surcharge de l'opérateur [].

En effet, une liste a des avantages par rapports aux vecteurs/tableaux/espaces mémoire contigus : si cet ensemble d' éléments est appelé à grandir/diminuer ou si on veut insérer un élément dans le tableau.

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

Par contre, pour accéder au n-ième élément d'une liste, il faut la parcourir du début, élément par élément.

Le temps d'accès est donc linéaire par rapport à la taille de la liste.

En notation de Landau : O(n)

Dans un tableau, le temps d'accès est constant. O(1)

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

Pour agrandir une liste, par contre, le temps est constant, alors qu'il est difficile d'augmenter le nombre d' éléments d'un tableau…

On doit copier les éléments de l'ancien tableau dans le nouveau avant de libérer l'ancien espace mémoire etc.

LA STL

LA CLASSE MAP

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  std::map<std::string, double> m;
  m["Charles"] = 5.0;
  m["Toto"] = 10;
  m["Sally"] = 15;

  std::cout << m.size() << std::endl;
  std::cout << m["Sally"] << std::endl;

}
```

La classe *std::map* est un autre conteneur qui permet d'associer une clé à une valeur.

On parle de tableaux associatifs ou de dictionnaires.

Comme c'est une classe template, le type des clés/valeurs peut être attribué lors de l'écriture du code.

L'opérateur [] est cette fois-ci surchargé pour accéder à la valeur d'une clé.

LA STL

Il existe d'autres conteneurs dans la STL et nous ne les verrons pas tous.

On pourra néanmoins se reporter à la référence sur *cplusplus.com* pour connaître les autres types en fonction des besoins.

LA STL

LES ITERATEURS

Pour homogénéiser les actions possibles sur les différents conteneurs, il est apparu la notion d'itérateurs. Un itérateur, qui peut être vu comme une généralisation de la notion de pointeur, permet de parcourir un conteneur.

Il possède ces propriétés :

- À chaque instant, un itérateur pointe vers une valeur qui désigne un élément donné du conteneur.
- Un itérateur peut être incrémenté par l'opérateur ++.
- Il peut être déréférencé, c'est-à-dire que l'utilisation de l'opérateur * permet (comme sur un pointeur) d'accéder à la valeur courante de l'itérateur.
- Deux itérateurs sur un même conteneur peuvent être comparés.

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
        iv != v.end();
        ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
        iv != v.end();
        ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Un itérateur (sur vecteur) se déclare de la façon suivante :

std::vector<double>::iterator iv;

La première partie est un vector de doubles.

Que signifie la deuxième partie ?

Les objets itérateurs sont définis à l'intérieur du conteneur qu'ils itèrent.

On dit qu'il s'agit de classes imbriquées, ou « nested class ». En effet, on peut déclarer une classe à l'intérieur d'une autre classe. Pour s'adresser à cette classe imbriquée, on va faire appel à l'opérateur « :: ».

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Tous les conteneurs fournissent deux valeurs particulières, sous forme d'itérateur, permettant de fournir un début et une fin.

Ainsi, on initialise l'itérateur avec comme valeur l'itérateur pointant sur le début du vecteur.

Comme on peut comparer l'égalité ou l'inégalité de deux itérateurs, on arrête la boucle *for* quand l'itérateur est égal à l'itérateur pointant sur la fin du vecteur.

Attention : *end()* est un itérateur particulier qui ne pointe pas sur le dernier élément, mais un cran plus loin. De sorte que pour un conteneur vide, *begin()* sont égaux.

et end()

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Pour passer à l'élément suivant, l'itérateur peut être incrémenté grâce à l'opérateur « ++ ».

~~**Attention :** cela ne veut pas dire qu'il existe un opérateur « -- ».~~

On remarquera aussi que l'on écrit ++iv et non iv++.

En effet, la notation postfixée entraîne plus d'opérations que la notation préfixée.

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Pour accéder à la valeur pointée par l'itérateur, on le déréférence grâce à l'opérateur « * ».

On peut ainsi modifier ou accéder à cette valeur.

Pour les vecteurs, il existe aussi une surcharge de l'opérateur « [] » permettant d'accéder immédiatement à un élément donné.

Cela entraîne la possibilité d'une arithmétique des itérateurs sur vecteur.

Ainsi, dans l'exemple ci contre,

*(iv+i) est équivalent à v[i]

LA STL

ITERATEUR SUR LIST

```
#include <list>
#include <iostream>

int main()
{
  std::list<int> lis;
  lis.push_back(2);
  lis.push_back(3);
  std::list<int>::iterator il;
  for (il = lis.begin(); il != lis.end(); ++il)
     std::cout << *il << std::endl;
  // NE COMPILE PAS
  //std::cout << *(lis.begin() + 1) << std::endl;
}
```

Les itérateurs sur list se déclarent de la même façon que les itérateurs sur vector.

De même, une liste possède deux fonctions particulières begin() et end() qui permettent d'initialiser et de comparer un itérateur sur list.

De même, l'opérateur de déréférencement permet d'accéder à la valeur pointée par l'itérateur.

LA STL

ITERATEUR SUR LIST

```
#include <list>
#include <iostream>

int main()
{
  std::list<int> lis;
  lis.push_back(2);
  lis.push_back(3);
  std::list<int>::iterator il;
  for (il = lis.begin(); il != lis.end(); ++il)
     std::cout << *il << std::endl;
  // NE COMPILE PAS
  //std::cout << *(lis.begin() + 1) << std::endl;
}
```

Par contre, il n'y a pas d'opérateur [] sur une liste.

Une liste ne s'utilise pas comme un vector.

Accéder directement à un élément n'est pas souhaitable car c'est une procédure lente.

Il n'y a donc pas d'arithmétique sur les itérateurs de list : on garde une cohérence entre le conteneur et son itérateur.

LA STL

ALGORITHMES – FOR_EACH

```
#include <vector>
#include <algorithm>
#include <iostream>

void affiche(int i){
    std::cout << i << ";";
}

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> v(myints, myints+8);
    std::for_each(v.begin(), v.end() - 2, affiche);
}
```

La STL contient également des algorithmes. Ces fonctions template travaillent sur des conteneurs et des itérateurs.

On présente ici la fonction *for_each* qui exécute une fonction (ou un foncteur) passée en paramètre, entre deux bornes d'un conteneur – c'est-à-dire entre deux valeurs d'itérateurs.

```
$ ./a.out
32;71;12;45;26;80;
```

LA STL

ALGORITHMES – COUNT/COUNT_IF

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

bool est_pair(int i){
    return i % 2 == 0;
}

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33,32};
    vector<int> v(myints, myints+9);
    int c = count_if(v.begin(), v.end(),  est_pair);
    cout << "Nbr pair :" << c << endl;
    cout << "32 :"<< count(v.begin(), v.end(), 32)
            << endl;
}
```

La fonction ***count*** permet de compter le nombres d'occurrences d'un élément entre deux itérateurs d'un conteneur.

La fonction ***count_if*** permet de compter le nombre d'éléments satisfaisant une condition.

Cette condition prend la forme d'une fonction (ou d'un foncteur) - renvoyant un bool et prenant en paramètre un élément du conteneur - passée en paramètre.

```
$ ./a.out
Nbr pair :5
32 :2
```

LA STL

ALGORITHMES – SORT

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


int main()
{
    int myints[] = {32,71,12,45,26,80,53,33,32};
    vector<int> v(myints, myints+9);
    sort(v.begin(), v.end());
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << "-";
    cout << endl;
}
```

La fonction *sort* permet d'appliquer une fonction de tri sur un conteneur entre deux valeurs d'itérateurs.

Le troisième argument - optionnel - est une fonction renvoyant vrai si deux éléments du tableau sont dans la bonne position.

```
$ ./a.out
12-26-32-32-33-45-53-71-80-
```

LA STL

STL ET POLYMORPHISME

```
#ifndef _AB_HH_
#define _AB_HH_
#include<iostream>
class A
{
private:
   int _a;
public:
   virtual void affiche() ;
};

class B : public A
{
private:
   int _b;
public:
   void affiche();
};

#endif
```

Fonction virtuelle

AB.hh

```
#include "AB.hh"
#include <vector>
using namespace std;

int main()
{
   vector<A*> tab;  // tableau de  A*
  tab.push_back(new A() );
  tab[0]->affiche();

  tab.push_back(new B( ));
  tab[1]->affiche();

delete tab[0];
delete tab[1];
}
```

main.cpp

```
#include "AB.hh"
using namespace std;

void A::affiche() {
cout << "classe A: " << _a << endl;  }

void B::affiche() {
cout << "classe B: "  << _b << endl; }
```

AB.cpp

Ici on utilise un *vector* pour stocker des pointeurs sur A.

Comme vu précédemment, à l'intérieur du tableau *tab*, on peut avoir une adresse d'un objet A ou d'un objet B, ce qui permet de manipuler différents types de données.