

Programmation Orientée Objet – C++



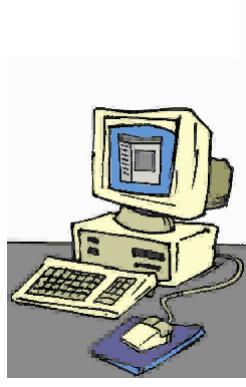
« There are two ways to write error-free programs;
only the third one works »

« You never finish a program,
you just stop working on it »

« L'expérience est une lanterne que l'on porte
sur le dos et qui n'éclaire jamais que le chemin
parcouru »

« La nature nous a dotés de deux oreilles et
d'une seule bouche...
Il convient donc d'écouter deux fois plus que
de parler ! »

« Combien faut-il de programmeurs C++ pour
changer une ampoule ?
Réponse : 6
Un pour la changer et 5 autres, six mois plus
tard, pour comprendre comment il a fait. »




© MM - Reproduction interdite sans l'autorisation de l'auteur.

INSA INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

Mathieu Maranzana
Mathieu.Maranzana@insa-lyon.fr





© MM - Reproduction interdite sans l'autorisation de l'auteur.

Programmation Orientée Objet – C++





Naufrage

PLAN du COURS

- Méthodes et fonctions en ligne
- Classes, méthodes et fonctions amies
- Surcharge
- Conversion de Types
- Généricité
- Entrées / Sorties : `iostream`
- Standard Template Library



Divin ?



Frisson



Bricolage

INSA INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



2

© MM - Reproduction interdite sans l'autorisation de l'auteur.

Programmation Orientée Objet – C++

PLAN du COURS



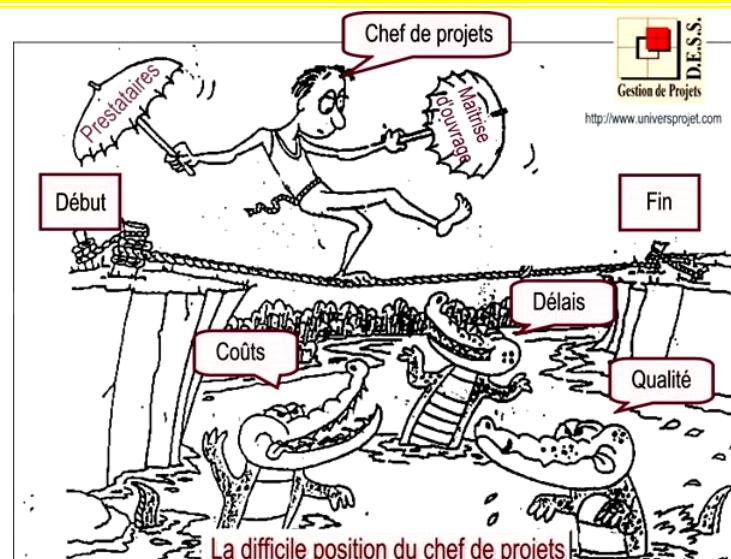



INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

- Méthodes et Fonctions *inline*
 - ◆ Fonction **inline** vs **#define**
 - ◆ Mise en Œuvre
 - ◆ Avantages / Inconvénients
- Classes, Méthodes et Fonctions Amies
- Surcharge
- Conversion de Types
- Généricité
- Entrées / Sorties : **iostream**
- ...

© MM - Reproduction interdite sans l'autorisation de l'auteur.

Méthodes et Fonctions en Ligne



Chef de projets

Début Fin

Prestataires

Mainteneur d'ouvrage

Délais

Coûts

Qualité

D.E.S.S. Gestion de Projets
http://www.universprojet.com

La difficile position du chef de projets

http://www.projetsinformatiques.com

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Fonction inline vs #define

■ Similitude

- ◆ Remplacement du code au point d'appel de la fonction / *macro*
 - Opération effectuée par le préprocesseur pour la macro (`#define`)
 - Opération effectuée par le compilateur pour la fonction `inline`

■ Différences

- ◆ À la substitution, vérification des types (paramètres + retour), comme pour une fonction classique
- ◆ Évaluation unique des expressions passées en paramètre d'une fonction `inline`
Rappel : évaluation multiple possible avec une *macro* (`#define`)
(exemple de *macros* à effets de bord cf. IFA-3-S1-EC-POO1)
- ◆ Même syntaxe (et même sémantique) que pour une fonction classique (simple ajout du mot-clé `inline` sur la déclaration / définition)



Mise en Œuvre

■ Contexte

- ◆ Directive pour le compilateur qui lui **suggère mais n'impose pas** un comportement (substitution du corps de la fonction à chaque appel)
- ◆ Application aux fonctions (très) courtes (peu de lignes de code)
- ◆ Application aux fonctions ((très) courtes) utilisées souvent

■ Comment rendre une fonction ordinaire `inline`...

- ◆ Utiliser le mot-clé C++ `inline` à la définition de la fonction (très souvent) placée dans l'interface (compilation séparée)

■ Comment rendre une fonction membre (méthode) `inline`...

- ◆ Écrire la définition de la méthode au niveau de sa déclaration dans la définition de la classe (mot-clé `inline` facultatif)
- ◆ Écrire la définition de la méthode à la suite de la définition de la classe en utilisant le mot-clé `inline` (la déclaration de la méthode reste inchangée dans la définition de la classe)



Mise en Œuvre inline vs #define

■ Exemple

```
// Macro qui renvoie la valeur minimale entre a et b
#define minARisque(a,b) ( (a) < (b) ? (a) : (b) )

// Fonction inline qui renvoie la valeur minimale entre a et b
// En principe, la définition de la fonction minSur est dans une interface (éditions des liens)
inline int minSur ( int a, int b )
{
    return a < b ? a : b;
}

int f () ;

void test ( int x, int y )
{ int min;
    min = minARisque( ++x, y ); // Danger ! => double incrémentation de x
    min = minARisque( f(), y ); // Danger ! => double appel de f() possible
    min = minSur ( ++x, y ); // Correct ! => une seule incrémentation de x
    min = minSur ( f(), y ); // Correct ! => un seul appel de f()
    ...
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur

⇒ Attention double incrémentation de **x**
si **++x** est la valeur minimale

⇒ Attention double appel de **f** si la valeur
de retour de **f** est la valeur minimale

min = minARisque(**++x**, y); // Danger ! => double incrémentation de **x** possible

min = minARisque(**f()**, y); // Danger ! => double appel de **f()** possible

min = minSur (**++x**, y); // Correct ! => une seule incrémentation de **x**

min = minSur (**f()**, y); // Correct ! => un seul appel de **f()**



Mise en Œuvre de la Méthode *inline*

```
----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined ( POINT_H )
#define POINT_H
class Point
{
public :
    //-----
    void Deplacer ( const Point & delta )
    {
        x += delta.x;
        y += delta.y;
    }

    void Afficher ( const char *message = "" ) const;
    // Méthode inline : définition de la méthode dans l'interface à la fin de la définition de la classe
    // ...

protected :
    //-
    int x; // Les attributs x et y du point sont protégés
    int y;
};

#endif // ! defined ( POINT_H )

inline void Point::Afficher ( const char *message ) const
// Affiche les coordonnées du point précédées par un message optionnel (inutile)
{
    cout << message << x << "," << y << endl;
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur

⇒ Définition de la méthode (bloc) au niveau de
la déclaration dans la définition de la classe
⇒ **Deplacer** est une méthode *inline*

Méthodes publiques

⇒ Attention définition d'une méthode dans une interface
(contraire au principe de base de la compilation séparée)

Attributs protégés

⇒ Rappel : également possible de définir une
fonction ordinaire *inline* dans une interface



Avantages / Inconvénients

■ Avantages

- ◆ Accélération du temps d'exécution...
 - En évitant les surcoûts liés aux appels de fonctions / méthodes
 - En réduisant les *push* / *pop* sur la pile au moment des appels
 - En réduisant les coûts des instructions *return*
- ◆ En marquant une fonction / méthode *inline*, il est possible de placer sa **définition** dans un fichier interface .h (inclusion possible dans différentes réalisations .cpp, sans erreur à l'édition des liens)

■ Inconvénients

- ◆ Accroissement de la taille de l'exécutable (répétition du code)
- ◆ Réduction du temps d'exécution... à cause de l'accroissement de la taille du code ⇒ défauts de pages plus fréquents (*thrashing*)
- ◆ Gestion des *inline* à la compilation ⇒ en cas de modification d'une *inline* de nombreuses compilations peuvent en découler
- ◆ Détails de réalisation dans les interfaces ⇒ peu utiles à l'utilisateur
- ◆ Non applicable dans les systèmes embarqués (mémoire réduite)

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Programmation Orientée Objet – C++

PLAN du COURS

| | |
|---|---|
|  | <ul style="list-style-type: none"> ■ Méthodes et Fonctions <i>inline</i> |
|  | <ul style="list-style-type: none"> ■ Fonctions, Méthodes et Classes Amies <ul style="list-style-type: none"> ◆ Relation d'Amitié ◆ Mise en Œuvre ◆ Limites de la Relation d'Amitié ■ Surcharge ■ Conversion de Types ■ Généricité ■ Entrées / Sorties : <i>iostream</i> ■ ... |
|  |  |
| INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON | <small>© MM - Reproduction interdite sans l'autorisation de l'auteur.</small> |

Fonctions, Méthodes et Classes Amies

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

11 © MM - Reproduction interdite sans l'autorisation de l'auteur.

Relation d'Amitié

- Encapsulation
 - ◆ Accès interdit aux membres privés (interdiction totale) et protégés (interdiction partielle à cause de l'héritage) à tout client de la classe
- Limite de l'encapsulation
 - ◆ Performance réduite (utilisation obligatoire des services de la classe pour accéder aux membres privés)
- Mécanisme de l'amitié
 - ◆ Utilisation du mot-clé C++ **friend**...
 - Pour une fonction ordinaire
 - Pour une méthode d'une autre classe
 - Pour une classe complète
 - ◆ Accès sans restriction pour un élément non membre de la classe (mêmes droits d'accès qu'un membre de la classe)
 - ◆ Ne pas abuser de la relation d'amitié ⇒ rupture de l'encapsulation

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

12 © MM - Reproduction interdite sans l'autorisation de l'auteur.

Mise en Œuvre de la Relation d'Amitié

```

//----- Interface de la classe <C1> (fichier C1.h) -----
#ifndef ! defined ( C1_H )
#define C1_H
class C2; // Déclaration de la classe C2
class C1
{
public :
    // ❶ La fonction non-membre operator << pourra accéder aux membres privés (x) de la classe C1
    friend std::ostream & operator << ( std::ostream & out, const C1 & o );
    // ❷ La méthode m de la classe C2 pourra accéder aux membres privés (x) de la classe C1
    friend const char * C2::m ( const C1 & o );
    // Les membres d'autres classes peuvent être des amies de C1
    // ❸ Les constructeurs et destructeurs d'une classe peuvent également être également des amies de C1
    friend C2::C2 ( const char * ); // Possible mais plus rare
    friend C2::~C2 (); // Possible mais plus rare
    friend class C3; // ❹ Une classe complète (C3) peut être amie de la classe C1
private :
    int x;
};

#endif // ! defined ( C1_H )

```

© MM - Reproduction interdite sans l'autorisation de l'auteur

Annotations:

- operator << : cas classique d'une fonction non membre (ordinaire), déclarée amie d'une classe C1
- Cas particulier d'une méthode unique (m) d'une classe C2, déclarée amie d'une classe C1
- ❶ La fonction non-membre operator << pourra accéder aux membres privés (x) de la classe C1
- ❷ La méthode m de la classe C2 pourra accéder aux membres privés (x) de la classe C1
- ❸ La déclaration d'amitié peut s'appliquer à des méthodes particulières d'une classe C2 (constructeur et destructeur)
- ❹ Définition de la fonction non membre operator << (amie de la classe C1) dans un fichier de réalisation (bloc)
- Non membre de la classe C1 (fonction ordinaire) ⇒ pas d'utilisation de l'opérateur de portée ::

Limites de la Relation d'Amitié

- Pas de symétrie**
 - Si C1 est une classe amie de C2, les fonctions membres de C2 ne pourront pas accéder aux membres privés de C1
 - ⇒ il faut une déclaration d'amitié explicite
- Pas de transitivité**
 - « Les amis des amis ne sont pas des amis ».
 - Une classe C1 amie d'une classe C2, elle-même amie d'une classe C3, n'est pas amie de la classe C3 par défaut
 - ⇒ il faut une déclaration d'amitié explicite

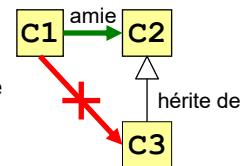
INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

14

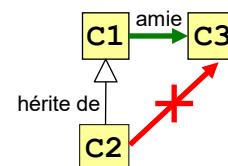
Limites de la Relation d'Amitié

■ Pas d'héritage

- ◆ Si une classe **c1** est amie d'une classe de base **c2**, et que la classe **c3** hérite de la classe de base **c2**, alors **c1** n'est pas amie de la classe dérivée **c3** par défaut
⇒ il faut une déclaration d'amitié explicite



- ◆ Si une classe de base **c1** est amie d'une classe **c3**, et que la classe **c2** hérite de la classe de base **c1**, alors **c2** n'est pas amie de la classe **c3** par défaut
⇒ il faut une déclaration d'amitié explicite



Limites de la Relation d'Amitié

■ Pour une fonction non membre...

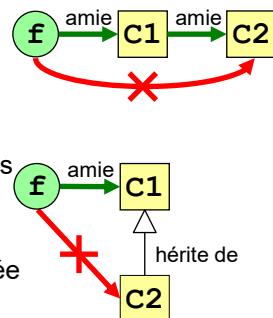
- ◆ Une fonction `f`, amie d'une classe `c1` qui est amie d'une classe `c2`, n'est pas amie de la classe `c2`

⇒ il faut une déclaration d'amitié explicite

- ◆ Une fonction **f**, amie d'une classe **c1** n'est pas amie des classes dérivées de **c1**

En effet, une fonction amie est par définition externe à une classe et ne peut pas être héritée par les classes dérivées

⇒ il faut une déclaration d'amitié explicite



- Réserver la relation d'amitié aux classes très interdépendantes (fortement couplées)



Programmation Orientée Objet – C++

PLAN du COURS

Naufrage

- Méthodes et Fonctions *inline*
- Fonctions, Méthodes et Classes Amies
- Surcharge
 - ◆ Généralités et Subtilités
 - ◆ Résolution de la Surcharge
 - ◆ Surcharge des Opérateurs
 - ◆ Surcharge et Fonctions Amies
- Conversion de Types
- Généricité
- ...

Frisson

Divin ?

Bricolage

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Surcharge



MOTOR DESIGN

La Direction a changé d'avis !
Il faut au minimum une 12 places!

Gestion de Projets D.E.S.S.

<http://www.universprojet.com>

Mieux vaut bien définir son besoin!

<http://www.projetsinformatiques.com>

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Surcharge

■ Caractéristiques

- ◆ La surcharge permet de définir en C++, **sans erreur de compilation**, plusieurs fonctions / méthodes avec le **même nom**
- ◆ N'existe pas en langage C
- ◆ S'appuie sur la notion de **signature** d'une fonction / méthode...
 - 2 composantes dans la signature d'une fonction / méthode...
 - Le nom de la fonction / méthode
 - Le nombre, l'ordre et le type de ses paramètres formels (pas les noms)
 - En C : le compilateur utilise uniquement le nom des fonctions pour les différencier ⇒ **erreur** avec des fonctions de **même nom**
 - En C++ : le compilateur utilise la signature complète des fonctions / méthodes ⇒ plus d'erreur pour des fonctions / méthodes de même nom mais avec des paramètres différents (en nombre, ordre et / ou type)
- **Attention** : Le type de retour ne fait pas partie de la signature
 - Le résultat peut ne pas être utilisé

```
int f ( int n, int d );
double f ( int n, int d );
```

⇒ Ces 2 fonctions **f** ont la même signature
⇒ **erreur de compilation** – pas de surcharge



Surcharge et Paramètres par Défaut

■ Paramètres par défaut

```
int f ( int n = 0, int d = 1 );
```

- ◆ **f** peut être appelée avec 0, 1 ou 2 paramètres
- ◆ Donc **f** a **3 signatures** différentes de fonction
 - Signature 1 : **f** (**int, int**)
 - Signature 2 : **f** (**int**)
 - Signature 3 : **f** ()

■ Erreur de surcharge

```
int f ( int n = 0, int d = 1 );
int f ( ); // erreur de surcharge à l'appel
```

- ◆ La signature de la deuxième déclaration de **f** est identique à l'une des 3 précédentes ⇒ **ambiguïté** et **erreur de compilation**





Subtilités de la Surcharge

```

#include <iostream>
using namespace std;

struct test {
    int x;
};

void f ( test t )
{   cout << t.x << endl;
}

void f ( const test t )      // ❶ Erreur de compilation : redéfinition de void f ( test );
{   cout << t.x << endl;     // ⇒ le const ne permet pas de faire la différence entre les 2 f
}

void f ( test * pt )
{   pt->x = 99;             // La valeur de la structure pointée par pt est modifiée dans l'appelant
    return;                  // ⇒ possible, parce que la structure pointée par pt est non const
}

void f ( const test * pt ) // ❷ Cette redéfinition de f est correcte
{                           // ⇒ le const permet de faire la différence entre les 2 fonctions f
    cout << pt->x << endl; // Cette utilisation de la structure en read-only est correcte
    pt->x = 99;            // Cette modification de la structure const pointée par pt est interdite
    return;                  // Cela met en évidence la différence avec la définition précédente de f
}

void f ( test & t );        // ❸ Surcharge correcte de f ⇒ même comportement avec un
void f ( const test & t ); // passage par référence qu'avec un passage par valeur d'un pointeur

```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Surcharge et Constructeurs

```

----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined ( POINT_H )
#define POINT_H

class Point
{
public :
    ----- Méthodes publiques
    void Deplacer ( const Point & delta );
    // ...

    void Afficher ( const char *message = "" ) const;
    // ...

    ----- Constructeurs - destructeur
    Point ( const Point & unPoint );
    // Constructeur de copie

    Point ( int abs = 0, int ord = 0 );
    // Constructeur par défaut
}

protected :
    ----- Attributs protégés
    int x; // Les attributs x et y du point sont protégés
    int y;
};

#endif // ! defined ( POINT_H )

```

↳ Surcharge des constructeurs de la classe Point
 ⇒ Même nom avec une liste de paramètres différente

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Résolution de la Surcharge

- Choix de la fonction / méthode par le compilateur...
 - ◆ **Règle 1** : concordance parfaite
 - Appel de la fonction / méthode dont la liste des paramètres correspond exactement en nombre, ordre et type
 - ◆ **Règle 2** : utilisation de la conversion de type
 - Appel de la fonction / méthode dont la liste des paramètres correspond exactement en nombre, ordre et type en appliquant sur certains paramètres des conversions de type (implicites ou explicites, c'est-à-dire définies par l'utilisateur)
 - Quelques conversions de type implicites...
 - **int** vers **float**
 - **char** vers **int**
 - **float** vers **int**
 - La **Règle 2** est très pratique mais source de nombreux problèmes
 - ◆ Précédence de la **Règle 1**



Surcharge des Opérateurs

- Généralités
 - ◆ Mécanisme qui permet de remplacer une notation fonctionnelle par une notation algébrique

```
// Notation fonctionnelle
Heure debut, fin, delta;
delta.Affecter ( fin.Moins ( debut ) );
// Affecter et Moins sont des méthodes d'une classe Heure

Heure debut, fin, delta;
delta.operator = ( fin.operator - ( debut ) );
// operator = et operator - sont des surcharges d'opérateur d'une classe Heure

// Notation algébrique
Heure debut, fin, delta;
delta = fin - debut;
// Puissance, élégance et concision de la notation utilisant la surcharge d'opérateurs
```



Surcharge des Opérateurs

■ Généralités

- ◆ Opérateurs ne pouvant être surchargés
 - `::` (portée) `.` (accès à un membre) `.*` (accès à un membre au travers d'un pointeur sur membre) `?:` (expression conditionnelle)
 - Plus marginalement : `sizeof`, `alignof` (C++11), `typeid`, `dynamic_cast`, `const_cast`, `reinterpret_cast` et `noexcept` (C++11)
- ◆ Arité (unaire, binaire, ternaire), priorité et associativité sont les mêmes que pour les opérateurs d'origine
- ◆ Lisibilité du code accrue mais attention aux chutes de performances
 - Problème du grand nombre de copies d'objets qui peut découler d'un usage intensif de la notation algébrique
- ◆ 2 façons de surcharger les opérateurs pour les classes définies par l'utilisateur (conservation de l'esprit de l'opérateur)
 - Définition d'une fonction membre
 - Définition d'une fonction ordinaire en dehors de la classe (`friend`)



© MM - Reproduction interdite sans l'autorisation de l'auteur.

25



Surcharge des Opérateurs : Fonction Membre

■ Syntaxe (opérateur interne – fonction membre)

```
type operator Op ( paramètres );
```

- ◆ Nom de la méthode : mot-clé `operator` suivi de l'opérateur `Op` à surcharger (+, -, *, /, %, <, >, =, ++, +=, ...)
- ◆ Premier opérande : objet invoquant la fonction membre
- ◆ `paramètres` : deuxième opérande et les suivants éventuellement, en fonction de l'arité
- ◆ Retour
 - Très souvent, l'objet invoquant la surcharge (pour respecter la sémantique générale de l'opérateur : par exemple pour l'affectation multiple `o1=o2=o3 ;`)
 - Utilisation du pointeur `this`
- ◆ Notation équivalente : `x Op y` \Leftrightarrow `x.operator Op (y)`
- ◆ Opérateur interne – fonction membre bien adapté pour les opérateurs qui modifient l'objet sur lequel ils travaillent



© MM - Reproduction interdite sans l'autorisation de l'auteur.

26



Surcharge de l'Affectation

■ Surcharge particulière : `operator =`

- ◆ Si aucune surcharge de l'affectation n'est définie, le compilateur en fournit une par défaut ⇒ à éviter !
 - Fonctionnement de cette surcharge par défaut : **affectation attribut par attribut**
 - Ne convient pas forcément pour les attributs *pointeur* ⇒ copie *en surface* (copie des pointeurs avec un partage de la même zone de mémoire pointée) au lieu d'une copie *en profondeur* (duplication de la zone de mémoire pointée)

```
// Classe Vecteur du cours IFA-3-S1-EC-POO1
// Rappel : 2 attributs : int tailleMax et int *elements (pointeur !)
Vecteur v1 ( 5 );
Vecteur v2 ( 9 );
v1 = v2; // La surcharge de l'affectation par défaut ne convient pas à cause du pointeur elements
```

- Récursif : si un attribut est un objet, instance d'une classe c...
 - Appel de la surcharge de l'affectation de la classe c, si elle existe
 - Appel de la surcharge de l'affectation par défaut, fournie par le compilateur, dans le cas contraire



© MM - Reproduction interdite sans l'autorisation de l'auteur

27



Surcharge de l'Affectation : classe Vecteur

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

class Vecteur
{ public :
    Vecteur ( int taille = 10 ); // Constructeur de la classe
    ~Vecteur ( ); // Destructeur de la classe
    Vecteur & operator = ( const Vecteur & v ); // Surcharge de l'affectation

protected :
    int nbElements; // Nombre d'éléments courants dans le vecteur
    int tailleMax; // Nombre d'éléments maximum dans le vecteur (0 ≤ nbElements ≤ tailleMax)
    int *elements; // Eléments de type entier qui composent le vecteur (pointeur sur int)
};

-----
```

⇒ Surcharge de l'affectation obligatoire à cause du pointeur `elements` et du `new` associé

⇒ Le code de la surcharge de l'opérateur d'affectation ressemble souvent à la concaténation du code du destructeur et du code du constructeur de copie
⇒ possibilité d'utiliser des fonctions membres privées pour factoriser le code

```
Vecteur & Vecteur::operator = ( const Vecteur & v )
{
    if ( this != &v ) // Traitement du cas particulier v1 = v1; à l'appel
    {
        delete [ ] elements; // Libération de la zone réservée à l'ancien vecteur
        nbElements = v.nbElements;
        elements = new int [ tailleMax = v.tailleMax ]; // Allocation de la nouvelle zone
        for ( int i = 0 ; i < nbElements ; elements[i] = v.elements[i], i++ );
    } // Boucle for = copie profonde du vecteur v dans le vecteur courant
    return *this; // Permet d'écrire v1=v2=v3; à l'appel (sémantique classique de operator =)
} //---- Fin de operator = (surcharge de l'affectation)
```



Opérateur d'Indexation : classe Vecteur

```

using namespace std;
#include <iostream>

class Vecteur
{ public :
    Vecteur ( int taille = 10 ); // Constructeur
    ~Vecteur ( ); // Destructeur de la classe
    Vecteur & operator = ( const Vecteur & v ); // Surcharge de l'affectation
    // int & operator [] (int index); // Opérateur d'indexation pour des vecteurs non constants
    const int & operator [] (int index) const; // Opérateur [] pour des vecteurs constants

protected :
    int nbElements; // Nombre d'éléments courants dans le vecteur
    int tailleMax; // Nombre d'éléments maximum dans le vecteur (0 ≤ nbElements ≤ tailleMax)
    int *elements; // Eléments de type entier qui composent le vecteur (pointeur sur int)
};

//-----
const int & Vecteur::operator [] ( int index ) const
{
    if ( ( index < 0 ) || ( index >= nbElements ) )
    {   // Gestion de l'erreur... (exception)
    }
    return elements [ index ];
} //----- Fin de operator [] const (surcharge de l'indexation)

```

© MM - Reproduction interdite sans l'autorisation de l'auteur

v2 [5] = 55;
 // écriture licite si le nombre d'éléments présent dans le vecteur
 // est supérieur ou égal à 2 (c'est-à-dire nbElements ≥ 1)

v2 [5] = 55;
 // erreur de compilation : il n'est pas possible de modifier
 // l'objet courant (pas de left-value)

⇒ La surcharge ne peut plus modifier l'objet
 ⇒ ne peut pas apparaître en left-value



Surcharge des Opérateurs : Fonction Ordinaire

- Syntaxe (opérateur externe – fonction ordinaire)

type operator Op (paramètres);

- ◆ Définition de la surcharge de l'opérateur en dehors de la classe
 ⇒ Surcharge d'un opérateur de l'espace de nommage global
- ◆ Nom de la fonction ordinaire : syntaxe classique
- ◆ **paramètres** : liste complète de tous les opérandes selon l'arité
 - Plus de paramètre **this** ⇒ pas d'objet invoquant la méthode !
- ◆ Retour par valeur ou par référence d'un objet (**type** ≡ classe)
- ◆ Utilisation fréquente de la relation d'amitié (**friend**) pour pouvoir accéder aux attributs privés d'une classe dans la fonction ordinaire
- ◆ Opérateur totalement symétrique avec possibilité d'utiliser un transtypage sur n'importe lequel des opérandes, si nécessaire
- ◆ Notation équivalente : **x Op y** ⇔ **operator Op (x, y)**


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


30

MM - Département Informatique C++ Avancé – 15

Surcharge des Opérateurs

```

//----- Interface de la classe <Point>
#ifndef ! defined ( POINT_H )
#define POINT_H
class Point
{
public :
    friend Point operator + ( const Point & p1, const Point & p2 );
    //----- Méthodes publiques
    void Afficher ( const char *message = "" ) const;

    //----- Constructeurs - destructeur
    Point ( const Point & unPoint ); // Constructeur de copie
    Point ( int abs = 0, int ord = 0 ); // Constructeur par défaut

protected :
    //----- Attributs protégés
    int x; int y; // Les attributs x et y du point sont protégés
};

#endif // ! defined ( POINT_H )
//----- Le mot clé friend n'apparaît plus à la définition de la surcharge de l'opérateur (fichier de réalisation)
Point operator + ( const Point & p1, const Point & p2 )
{
    Point p ( p1 ); // Utilisation du constructeur de copie
    p.x += p2.x;    p.y += p2.y;
    return p; // Utilisation du constructeur de copie pour le retour par valeur d'un point
} //----- Fin de operator + (surcharge de l'addition)

```

Point p1 (1, 1);
Point p2 (2, 2);
Point p3;
p3 = 3 + p1;
p3.Afficher ("3 + p1 = ");
p3 = p2 + 5;
p3.Afficher ("p2 + 5 = ");

// Résultat de l'exécution...
3 + p1 = 4,1 // Conversion de l'opérande 3 (cf. Conversion de type)
p2 + 5 = 7,2 // Conversion de l'opérande 5 (cf. Conversion de type)

Note :
Cette symétrie de l'operator + n'est plus possible avec une surcharge de l'opérateur, fonction membre de la classe Point :
Point operator + (const Point & p);
L'écriture p3 = 3 + p1; générera une erreur...

© MM - Reproduction interdite sans l'autorisation de l'auteur.

Surcharge des Opérateurs d'E/S

- Saisie (`operator >>`) ou affichage (`operator <<`) des données selon les règles standard du langage C++
- Déclaration de l'opérateur d'extraction d'un flux `>>`

```

istream & operator >> ( istream & flux, T & variable );
// istream désigne la classe flux en entrée

```

- Déclaration de l'opération d'insertion dans un flux `<<`

```

ostream & operator << ( ostream & flux, const T & variable );
// ostream désigne la classe flux en sortie

```

- Utilisation d'une relation d'amitié (`friend`) pour accéder aux attributs privés (de la classe `T`)
- Retour par référence du flux manipulé pour les enchaînements

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

32

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Surcharge des Opérateurs d'E/S

```

//----- Interface de la classe <Point> (fichier Point.h)
#ifndef ! defined ( POINT_H )
#define POINT_H

class Point
{
public :
    friend Point operator + ( const Point & p1, const Point & p2 );
    friend istream & operator >> ( istream & in, Point & p );
    friend ostream & operator << ( ostream & out, const Point & p );
    ...
protected :
    //----- Attributs protégés
    int x; int y; // Les attributs x et y du point sont protégés
};

#endif // ! defined ( POINT_H )

//-----
istream & operator >> ( istream & in, Point & p ) // Fonction ordinaire
{
    cout << "Coordonnée x du point ? "; in >> p.x;
    cout << "Coordonnée y du point ? "; in >> p.y;
    return in; // Enchaînement possible des opérations d'entrée (cin >> p1 >> p2;)
} //----- Fin de operator >> (surcharge de l'opérateur d'entrée)

ostream & operator << ( ostream & out, const Point & p ) // Fonction ordinaire
{
    out << "(" << p.x << "," << p.y << ")" << endl;
    return out; // Enchaînement possible des opérations de sortie (cout << p1 << p2;)
} //----- Fin de operator << (surcharge de l'opérateur de sortie)

```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Surcharge des Opérateurs ++ et --

- La même notation ++ (ou --) représente 2 opérateurs...
 - ◆ Post incrémentation : l'opérateur est après son opérande
 - ◆ Pré incrémentation : l'opérateur est avant son opérande
- Comment faire la différence à la surcharge de l'opérateur ?
 - ◆ Pas de paramètre puisque la surcharge travaille sur l'objet courant
⇒ une seule signature possible
- Solution retenue...
 - ◆ Ajout d'un paramètre fictif **int** (non utilisé) dans le cas de la post-incrémantion ⇒ 2 signatures possibles pour la surcharge
- Retour...
 - ◆ De la surcharge pour une **pré** incrémentation : **référence** de l'objet
 - ◆ De la surcharge pour une **post** incrémentation : **valeur** de l'objet
 - ◆ **Rappel** : le retour ne fait pas partie de la signature

© MM - Reproduction interdite sans l'autorisation de l'auteur.


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


34



Surcharge des Opérateurs ++ et --

```

//----- Interface de la classe <Point> (fichier Point.h)
#ifndef _POINT_H_
#define _POINT_H_

class Point
{
public :
    Point & operator ++ ( void ); // Opérateur ++ préfixé (pas de paramètres)
    Point operator ++ ( int inutile ); // Opérateur ++ suffixé (un paramètre fictif int)
    ...
protected :
    //-----
    int x; int y; // Les attributs x et y du point sont protégés
};

#endif // ! defined ( _POINT_H_ )
//-----

Point & Point::operator ++ ( void ) // Opérateur ++ préfixé (pas de paramètres)
{   ++x; ++y; // Incrémentation de la valeur du point courant
    return *this; // Retour par référence du nouveau point
} //---- Fin de operator ++ (surcharge de l'opérateur ++ préfixé)

Point Point::operator ++ ( int inutile ) // Opérateur ++ suffixé (un paramètre fictif int)
{   Point tmp ( x, y ); // Construction d'un nouveau point à partir du point courant (attributs x et y)
    ++x; ++y; // Incrémentation de la valeur du point courant ou utilisation de ++ préfixé : ++(*this)
    return tmp; // Retour par valeur du point sauvegardé avant son incrémentation (copie)
} //---- Fin de operator ++ (surcharge de l'opérateur ++ postfixé)

```

// Cas de la post incrémentation...
 Point p (5, 6);
 cout << "1. " << p << endl;
 cout << "2. " << p++ << endl;
 cout << "3. " << p << endl;

// Résultat de l'exécution...
 1. (5,6)
 2. (5,6)
 3. (6,7)

// Cas de la pré incrémentation...
 Point p (-3, -9);
 cout << "1. " << p << endl;
 cout << "2. " << ++p << endl;
 cout << "3. " << p << endl;

// Résultat de l'exécution...
 1. (-3,-9)
 2. (-2,-8)
 3. (-2,-8)

© MM - Reproduction interdite sans l'autorisation de l'auteur



Programmation Orientée Objet – C++

PLAN du COURS



Naufrage

- Surchage
- Conversions de Type
 - ◆ Généralités
 - ◆ Promotion et Autres Conversions
 - ◆ Conversion et Constructeur d'Objet
 - ◆ Opérateur de Conversion
 - ◆ Transtypage
- Généricité
- Entrées / Sorties : `iostream`



Divin ?



Bricolage

© MM - Reproduction interdite sans l'autorisation de l'auteur

36

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
 

Conversion de Type

DESSS.
Gestion de Projets
<http://www.universprojet.com>

Savez vous communiquer dans vos projets ?
<http://www.projetsinformatiques.com>

INSA

© MM - Reproduction interdite sans l'autorisation de l'auteur.

Conversion de Type

- Transformation d'un opérande qui n'a pas exactement le bon type...
 - ◆ Transformation implicite : à l'initiative du compilateur
 - ◆ Transformation explicite : à l'initiative du programmeur
 - ◆ De nombreuses possibilités de transformations...
 - Des promotions (sans modification de valeurs)...
 - Entières : par exemple de `char` vers `int` ou `short` vers `int`...
 - Virgules-flottantes : de `float` vers `double`
 - Des conversions *connues* (conversions arithmétiques usuelles)
 - Des conversions explicites...
 - Constructeurs d'objets sans utilisation du qualificatif `explicit`
 - Opérateurs de conversion de type : surcharge de l'opérateur `Type`
 - Surcharge de l'opérateur d'affectation éventuellement
 - Des transtypages...
 - Un transtypepage est une demande au compilateur de considérer que le type de l'objet est en réalité un autre type (pour les pointeurs)
 - `dynamic_cast`, `static_cast`, `reinterpret_cast` et `const_cast`

INSA

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Conversion de Type

■ Promotion... cas particulier de la conversion de type

- ◆ Promotion entière : d'un type **x** vers **int**, si **int** peut représenter toutes les valeurs du type source et sinon vers **unsigned int**
 - **x ≡ char, unsigned char, signed char, short int, unsigned short int**, constante nommée d'un type énuméré, **bool** (**false** vaut 0 et **true** vaut 1)
- ◆ Promotion virgule-flottante : de **float** vers **double**
- ◆ Avec une promotion, on récupère dans le type destination exactement la même valeur que dans le type source

■ Autres conversions de type : modification de la valeur après conversion (perte de précision, d'information...)

```
// Promotion de type de short vers int
short i = 99; int j;
j = i; // promotion de short vers int sans l'utilisation explicite d'un opérateur

// Conversion de type
int centimes = 99;
float euro = centimes / 100.0; // 100.0 par défaut de type double
```

⇒ Conversion de **double** vers **float**
 ⇒ Conversion de **int** vers **double**


© MM - Reproduction interdite sans l'autorisation de l'auteur
39


Conversion et Constructeur d'Objet

```
----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined ( POINT_H )
#define POINT_H

class Point
{
public :
    friend ostream operator << ( ostream & out, const Point & p );
    //----- Méthodes publiques
    ...

    //----- Constructeurs - destructeur
    Point ( const Point & unPoint ); // Constructeur de copie
    explicit Point ( int abs = 0, int ord = 0 ); // Constructeur par défaut

protected :
    //----- Attributs protégés
    int x; int y; // Les attributs x et y du point sont protégés
};

#endif // ! defined ( POINT_H )
```

⇒ Il n'existe plus de conversion implicite de l'entier **i** vers l'objet **p** de la classe **Point**, à cause de l'utilisation du mot clé **explicit** à la déclaration du constructeur par défaut de la classe **Point**
 ⇒ Évite les conversions à notre insu !



```
// Conversion (implicite ?) avec le
// constructeur par défaut de la classe...
int i = 3;
Point p = i;
cout << p << endl;
// Résultat de l'exécution...
Aucune ! Erreur de compilation...
```

```
// Conversion implicite avec le constructeur par
// défaut de la classe... (absence de explicit)
int i = 3;
Point p = i;
cout << p << endl;
// Résultat de l'exécution...
( 3,0 )
```


© MM - Reproduction interdite sans l'autorisation de l'auteur
40



Opérateur de Conversion de Type

■ Syntaxe

```
operator Type ( ) const;
```

- ◆ Aucun paramètre à cet opérateur...
 - S'applique évidemment à l'objet qui invoque la méthode !
- ◆ Pas de type retour à cet opérateur...
 - Opérateur de conversion de type, son type de retour est défini par **Type**

```
#if ! defined ( POINT_H )
#define POINT_H

class Point
{ public :
    operator int ( ) const; // Utilisation d'un objet Point partout où un int est attendu
    // ...
protected :
    int x; int y;
};

#endif // ! defined ( POINT_H )

//-----
Point::operator int ( ) const // Définition inline possible
{ return x; } // Choix pour la conversion de type Point vers int : rendre l'attribut x
```

// Conversion implicite avec l'opérateur de
// conversion de type de la classe Point...
Point p (5, 3);
cout << "p = " << p << endl;
int i = p;
cout << "i = " << i << endl;

// Résultat de l'exécution...
p = (5, 3)
i = 5

© MM - Reproduction interdite sans l'autorisation de l'auteur.


Conversion Explicite (Type Casting)

■ 2 Syntaxes...

- ◆ Notation C-like : (**Type**) **expression**
- ◆ Notation fonctionnelle : **Type** (**expression**)
- ◆ Applicable aux classes et aux pointeurs...
 - Construction possible de programmes syntaxiquement corrects...
 - Mais en échec à l'exécution (*run-time errors*) à cause des conversions de types explicites

```
// Conversion explicite avec un type casting...
Point *p;
int i = -99;
p = ( Point * ) ( &i );
cout << "*p = " << *p << endl;

// Résultat de l'exécution...
*p = ( -99,1263587 )
```

⇒ Pas d'erreur à l'exécution, mais la valeur du **Point** pointée par **p** est fausse : **y n'existe pas** ⇒ **débordement !**



Opérateur de Transtypage

■ Transtypage dynamique...

```
dynamic_cast <Type> ( expression )
```

- ◆ **Type** : type cible du transtypage
 - Pointeur, éventuellement sur **void**
 - Référence
 - Pas de types de base du langage C++
- ◆ **expression** : expression à transtyper
 - Doit être un pointeur si **Type** est un pointeur
 - Doit être une *left-value* si **Type** est une référence
- ◆ Retour
 - **Nul**, en cas d'échec de transtypage pour un pointeur
 - Exception **bad_cast**, en cas d'échec de transtypage pour une référence
- ◆ Intérêt du transtypage dynamique, par rapport à un *type casting* classique : vérification de sa validité



© MM - Reproduction interdite sans l'autorisation de l'auteur.

43



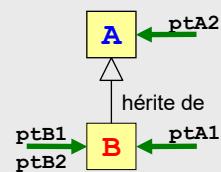
Opérateur de Transtypage

■ Transtypage dynamique...

- ◆ *Pointer upcast* : conversion d'un pointeur d'une classe dérivée vers une classe de base (conversion implicite pour un type polymorphique)
- ◆ *Pointer downcast* : conversion d'un pointeur d'une classe de base vers une classe dérivée si et seulement si l'objet pointé est valide
 - ⇒ Utilisation obligatoire d'une information dynamique de type, connue à l'exécution

```
// Pointer downcast : déplace le pointeur vers le bas dans la hiérarchie de classes
// La classe B hérite de la classe A...
A *ptA1 = new B;
A *ptA2 = new A;

B *ptB1 = dynamic_cast <B *> ( ptA1 );
// OK : ptA1 pointe actuellement vers un B
// ptB1 pointe actuellement vers un B
// ptA2 pointe actuellement vers un A et non pas vers un B
// ⇒ test à l'exécution pour vérifier si expression == ptA2 pointe actuellement
// sur un objet de type B
// ⇒ ici ptB2 est égal à nullptr (à cause de l'échec du transtypage)
```



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Opérateur de Transtypage

- Transtypage dynamique avec le type **void ***...
 - ◆ Si **Type** est **void *** alors un test à l'exécution est effectué pour déterminer le type actuel de l'**expression**
 - Résultat du transtypage = pointeur sur l'objet pointé par l'**expression**

```
// Soit A et B, 2 classes
A *ptA = new A;
B *ptB = new B;
void *pt;

pt = dynamic_cast<void *>( ptA );
// pt pointe maintenant sur un objet de type A
// => test à l'exécution pour vérifier le type exact de expression == ptA

pt = dynamic_cast<void *>( ptB );
// pt pointe maintenant sur un objet de type B
// => test à l'exécution pour vérifier le type exact de expression == ptB
```



Opérateur de Transtypage

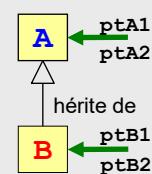
- Transtypage statique...

```
static_cast<Type>( expression )
```

- ◆ **Type** : type cible du transtypage (n'importe quel type)
- ◆ **expression** : expression à transtyper
- ◆ Défaut du transtypage statique : aucune vérification des types dynamiques (héritage) lors du transtypage
 - => Transtypage non sûr dans certains cas

```
// La classe B hérite de la classe A...
void f( A *ptA1, B *ptB1 )
{ B *ptB2 = static_cast<B *>( ptA1 );
  // Transtypage à risque (downcast possible) !
  // L'objet pointé par ptA1 n'est pas forcément de type (dynamique) B
  // => Utilisation à risque de ptB2, par exemple...
  //   Appel d'une fonction membre de la classe B mais pas de la classe A

  A *ptA2 = static_cast<A *>( ptB1 );
  // Transtypage sûr (upcast)
}
```





Opérateur de Transtypage

■ Transtypage de constance et volatilité...

```
const_cast <Type> ( expression )
```

- ◆ Permet de supprimer la caractéristique `const` d'un élément
- ◆ **Type** : type cible du transtypage (suppression de la caractéristique `const`)
- ◆ **expression** : expression à transtyper (pointeur ou référence)
- ◆ Opérateur à éviter...

```
#include <iostream>
static void affiche ( char *msg )
{   msg [ 0 ] = 'b';
    std::cout << msg << std::endl;
}

int main ( )
{   const char *TXT = "Bonjour !";
    affiche ( const_cast <char *> ( TXT ) );
    return 0;
}
```

⇒ Aucune erreur de compilation...
⇒ Mais une *Erreur de segmentation* à l'exécution !



© MM - Reproduction interdite sans l'autorisation de l'auteur

47



Opérateur de Transtypage

■ Réinterprétation des données...

```
reinterpret_cast <Type> ( expression )
```

- ◆ Permet de réinterpréter les données d'un type en un autre type !
 - Opérateur symétrique : réinterprétation de `T1` en `T2` puis du résultat `T2` en `T1` doit redonner l'élément initial
- ◆ Aucune vérification de la validité de l'opération...
- ◆ Opérateur à éviter, le plus dangereux des opérateurs de transtypage

```
// Soit A et B, 2 classes sans relation d'héritage
class A { /* ... */ };
class B { /* ... */ };

A *ptA = new A;
B *ptB = reinterpret_cast <B *> ( ptA );
// Code correct, sans erreur de compilation, mais qui n'a pas beaucoup de sens...
// ⇒ l'utilisation du pointeur ptB risque d'être explosive !
//     ptB pointe dorénavant un objet de la classe A, sans rapport et probablement
//     incompatible avec la classe B
```



© MM - Reproduction interdite sans l'autorisation de l'auteur

48

Programmation Orientée Objet – C++

PLAN du COURS



Naufrage ...

- Conversion de Type
- Généricité
 - ◆ Généralités
 - ◆ Fonction Générique
 - ◆ Classe Générique
 - ◆ Amitié et Classe Générique
 - ◆ Paramétrage et Classe Générique
- Entrées / Sorties : `iostream`
- Standard Template Library



Divin ?



Bricolage

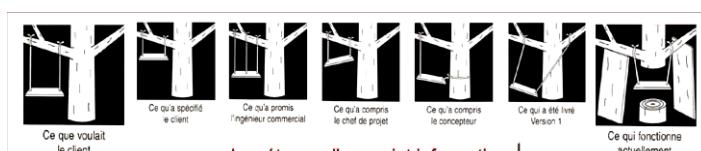
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur.

49



Généricité



Les étapes d'un projet informatique
<http://www.projetsinformatiques.com>

© MM - Reproduction interdite sans l'autorisation de l'auteur.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON





Généralités

- Un *template* est une entité C++ qui peut définir...
 - ◆ Un ensemble de classes (*class template*)
 - ◆ Un ensemble de fonctions (*function template*) (fonctions membres possibles)
 - ◆ Un alias vers un ensemble de types (*alias template*) (depuis C++11)
 - ◆ Un ensemble de variables (*variable template*) (depuis C++14)
 - ◆ Un concept (*constraints and concepts*) (depuis C++20)
- Intérêt
 - ◆ Programmation générique ⇒ réduction du code, factorisation
 - ◆ Limitation à une version générique...
 - Pile d'entiers, Pile de chaînes, Pile de tableaux devient Pile de <T>
 - ◆ Facilité de maintenance, d'évolution et de réutilisation
- Vocabulaire
 - ◆ Classe paramétrée, classe générique, modèle, pattern...

© MM - Reproduction interdite sans l'autorisation de l'auteur.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

51



Fonction Générique

- Syntaxe d'une déclaration

⇒ Attention : pas de point-virgule ici !

```
template < paramètres_template >
type nomFonction ( paramètres_fonction );
```

 - ◆ **paramètres_template** : liste des paramètres *template* (type)
 - ◆ **paramètres_fonction** : paramètres de la fonction utilisant les éléments de **paramètres_template**
 - ◆ **type** : valeur de retour de la fonction (utilisation possible des **paramètres_template**)

```
template < typename T > // 1 seul paramètre template : typename T
T Minimum ( const T & x, const T & y )
{
    return x < y ? x : y;
}
// Exemple d'utilisation de la fonction générique Minimum
double prix ( 14.99 ); // version double
cout << Minimum < double > ( prix, 7.43 ) << endl;
char c1 ( 'm' ), c2 ( 'x' ); // version char
cout << Minimum < char > ( c1, c2 ) << endl;
```

⇒ Note : le passage de paramètre se fait par référence constante et non pas par valeur ⇒ cela évite une copie coûteuse en temps et en mémoire (pile) quand x représentera un objet de grande taille

© MM - Reproduction interdite sans l'autorisation de l'auteur.

52

Fonction Générique

// Déduction automatique du type de la fonction générique par le compilateur,
// à partir du type des paramètres
double prix (14.99);
cout << Minimum (prix, 7.43) << endl; // version double
// Cas ambigu ou l'indication de type est obligatoire
cout << Minimum (prix, 5) << endl; // erreur de compilation
// Résolution de l'ambiguïté avec une conversion de type
cout << Minimum (int (prix), 5) << endl; // version int

- Compléments...
 - ◆ Indication facultative du type entre les `<...>`, à l'utilisation de la fonction générique, s'il n'y a pas d'ambiguïté
 - ◆ Fonction générique et compilation séparée
 - Déclaration et définition sont **obligatoirement** dans l'interface (`.h`)
 - ◆ Limite des fonctions génériques...
 - **Existence** des opérations effectuées dans la fonction générique pour les types passés en paramètres (surcharge des opérateurs, constructeur de copie... pour les types `T` qui sont des classes)

```
// Utilisation de l'opérateur < dans la fonction générique Minimum
template < typename T > // 1 seul paramètre template : typename T
T Minimum ( const T & x, const T & y )
{
    return x < y ? x : y;
}

Point p1 ( 1, 1 );
Point p2 ( 2, 2 );
cout << Minimum < Point > ( p1, p2 ) << endl; // version Point
```

⇒ Attention : cet appel impose l'existence de la surcharge de `operator <` dans la classe `Point` ⇒ En cas d'*absence*, **erreur de compilation** !
 ⇒ Corrections possibles...

- Définition de la surcharge de `operator <` dans la classe `Point`
- Spécialisation de la fonction générique `Minimum` (à venir)

© MM - Reproduction interdite sans l'autorisation de l'auteur

Fonction Générique

- Spécialisation d'une fonction générique
 - ◆ Organisation particulière du fichier interface en cas de spécialisation
 - D'abord la fonction générique, puis...
 - Les différentes fonctions spécialisées (spécialisation partielle ou totale)

```
// Spécialisation totale de la fonction générique Minimum pour la classe Point
template < > // Tous les paramètres génériques de Minimum sont déterminés
Point Minimum < Point > ( const Point & p1, const Point & p2 )
{ // Calcul d'une distance par rapport à ( 0, 0 ) pour rechercher le minimum entre 2 points
    return ((p1.x*p1.x+p1.y*p1.y) < (p2.x*p2.x+p2.y*p2.y)) ? p1 : p2;
}
```

⇒ Attention : absence de type entre `<...>`

⇒ Attention : problème d'accès aux attributs protégés `x` et `y` de la classe `Point` dans la fonction générique spécialisée `Minimum`
 ⇒ Une déclaration d'amitié (`friend`) de la fonction générique `Minimum` est obligatoire dans la classe `Point`

⇒ La définition de la fonction générique `Minimum` utilise directement le type souhaité par la spécialisation totale (ici `Point`) à la place du type générique `typename T`


 INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


 54

MM - Département Informatique

C++ Avancé – 27



Fonction Générique

■ paramètres_template multiples

```
template < typename T, typename R >
R moyenne ( const T tableau[ ], int nbElements )
{
    T somme ( 0 );
    for ( int i ( 0 ) ; i < nbElements ; i++ )
    {
        somme += tableau [ i ];
    }
    return somme / nbElements;
}
```

⇒ Note : tous les arguments d'une fonction générique ne sont pas forcément des templates (nbElements est un int classique)

⇒ Quel que soit le type **T** des éléments du tableau, la fonction **Moyenne** pourrait avoir une signification...
 • Pour les types primitifs (par exemple **int**), il n'y a pas de soucis...
 • Pour les classes, il faut disposer de la surcharge des opérateurs **+=** et **/** pour que la fonction ait un sens
 ⇒ Définition d'une fonction générique avec éventuellement des spécialisations (notamment pour les classes)



© MMI - Reproduction interdite sans l'autorisation de l'auteur



Classe Générique

■ Syntaxe de la définition (sur un exemple)

```
#if ! defined ( PILE_H )
#define PILE_H

template < typename T >
class Pile
{
public :
    // Utilisation possible du type générique T à la déclaration des méthodes
    // ...

protected :
    // Utilisation possible du type générique T lors de la mise en place des attributs
    // ...
};

#endif // ! defined ( PILE_H )
```

⇒ Création d'une classe unique générique pour représenter une pile de n'importe quel type
 ⇒ Spécialisation de cette classe pour un type particulier par instanciation...
 • Pile < int > p1;
 • Pile < Point > p2;
 ⇒ Code ad hoc généré par le compilateur en fonction de l'instanciation

⇒ Note : Types multiples possibles à la déclaration des paramètres_template

⇒ Par exemple : template <typename T1, typename T2>



© MMI - Reproduction interdite sans l'autorisation de l'auteur

De la Classe à la Classe Générique

```

//----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined ( POINT_H )
#define POINT_H

template < typename T >
class Point
{
public :
    //----- Méthodes publiques
    void Deplacer ( const Point < T > & delta );
    // Déplace le point courant en utilisant le point générique delta

    void Afficher ( const char *message = "" ) const;
    // Affiche les coordonnées du point précédées d'un message optionnel

    //----- Constructeurs - destructeur
    Point ( const Point < T > & unPoint ); // Construit un point par copie d'un point existant
    Point ( T abs = T ( ), T ord = T ( ) );
    // Construit un point à partir de 2 paramètres de type générique T
    ~Point ( ); // Destructeur de la classe

protected :
    //----- Attributs protégés
    T x; // Les attributs x et y du point sont protégés et de type générique T
    T y;
};

// Définition des méthodes génériques de la classe Point...
#endif // ! defined ( POINT_H )

```

⇒ Attention : x et y sont désormais de type générique T
 ⇒ À la construction, il faut que les paramètres soient de type T, et les valeurs par défaut doivent être obtenues par le constructeur par défaut (sans paramètre) de T

Méthodes publiques

Constructeurs - destructeur

Attributs protégés

Il reste à donner la définition des fonctions membres génériques

© MM - Reproduction interdite sans l'autorisation de l'auteur

De la Classe à la Classe Générique

```

// Réalisation des méthodes génériques de la classe Point (fichier Point.h)
using namespace std;
#include <iostream>
template < typename T >
void Point < T >::Deplacer ( const Point < T > & delta )
{
    x += delta.x;
    y += delta.y;
}
template < typename T >
void Point < T >::Afficher ( const char *message ) const
{
    cout << message << x << "," << y << endl;
}
template < typename T >
Point < T >::Point ( const Point < T > & unPoint )
{
    x = unPoint.x;
    y = unPoint.y;
}
template < typename T >
Point < T >::Point ( T abs, T ord )
{
    x = abs;
    y = ord;
}
template < typename T >
Point < T >::~Point ( )
{
    cout << "Destructeur de la classe <Point>" << endl;
} // Compilation conditionnelle à MAP

```

⇒ Définition du type template inutile, si la méthode est définie inline, à la suite de sa déclaration

⇒ Utilisation du type template dans le nom de la classe (Point < T >) parce que cette méthode sera instanciée de manière différente pour chaque T

© MM - Reproduction interdite sans l'autorisation de l'auteur



Fonction Amie et Classe Générique

```

//----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined ( POINT_H )
#define POINT_H

template < typename T >
class Point
{
public :
    friend ostream & operator << ( ostream & out, const Point & p )
    {   out << "(" << p.x << "," << p.y << ")";
        return out;
    }

    //----- Méthodes publiques
    void Deplacer ( const Point < T > & delta );
    // Déplace le point courant en utilisant le point générique delta

    //----- Constructeurs - destructeur
    Point ( const Point < T > & unPoint ); // Construit un point par copie d'un point existant
    Point ( T abs = T ( ), T ord = T ( ) );
    // Construit un point à partir de 2 paramètres de type générique T
    ~Point ( ); // Destructeur de la classe

protected :
    //----- Attributs protégés
    T x, y; // Les attributs x et y du point sont protégés et de type générique T
};

// Définition des méthodes génériques de la classe Point...
#endif // ! defined ( POINT_H )

```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Fonction Amie et Classe Générique

```

//----- Interface de la classe <Point> (fichier Point.h) -----
#ifndef ! defined ( POINT_H )
#define POINT_H
#include <iostream>
using namespace std;

template < typename T >
class Point;

template < typename T >
ostream & operator << ( ostream & out, const Point < T > & p );

template < typename T >
class Point
{
public :
    friend ostream & operator << << > ( ostream & out, const Point < T > & p );
    //...

protected :
    T x, y;
};

template < typename T >
ostream & operator << ( ostream & out, const Point < T > & p )
{   out << "(" << p.x << "," << p.y << ")";
    return out;
}
#endif // ! defined ( POINT_H )

```

Diagramme d'annotation pour l'interface de la classe Point :

- Surcharge d'un opérateur (`operator <<`) non membre d'une classe générique (`Point<T>`) et amie de cette classe (`friend`)
- Une solution simple : définir la surcharge de l'opérateur dans la définition de la classe : le compilateur générera une fonction ordinaire (non générique) différente, amie de la classe, pour chaque `T`
- Déclaration de la classe générique `Point` pour résoudre les problèmes de référence avant
- Déclaration de la surcharge générique de `operator <<` (fonction ordinaire générique)
- Spécialisation de la surcharge générique de `operator <<` pour le type `T`
- Définition de la surcharge générique de `operator <<` (fonction ordinaire générique)

© MM - Reproduction interdite sans l'autorisation de l'auteur.

Classe Générique et Paramétrage

```

#ifndef ! defined ( VECTEUR_H )
#define VECTEUR_H

#include <iostream>
using namespace std;

template < typename T, int tailleMax >
class Vecteur;

template < typename T, int tailleMax >
ostream & operator << ( ostream & out, const Vecteur < T, tailleMax > & v );

template < typename T, int tailleMax >
class Vecteur // Classe gérant un tableau d'éléments de type T, de taille maximale variable
{
public :
    friend ostream & operator << < < T, tailleMax > ( ostream & out, const Vecteur & v );
    T Prendre ( int index );
    bool Ajouter ( T unElement ); // true si l'élément a pu être ajouté
    Vecteur ( );
    ~Vecteur ( );
    // ...

protected :
    T elements [ tailleMax ];
    int nbElements;
};

// ... Définition des méthodes de la classe générique Vecteur
#endif // ! Defined ( VECTEUR_H )

```

⇒ Définition de la classe générique **Vecteur** avec 2 paramètres génériques (un type **T** et une taille **tailleMax**)

⇒ Déclaration classique d'une fonction ordinaire (non-membre) générique, amie de la classe générique **Vecteur**

⇒ Utilisation des 2 paramètres génériques (un type **T** et une taille **tailleMax**) pour définir les attributs protégés

© MM - Reproduction interdite sans l'autorisation de l'auteur

Classe Générique et Paramétrage

```

#ifndef ! defined ( VECTEUR_H )
#define VECTEUR_H

// ... Définition des méthodes de la classe générique Vecteur
template < typename T, int tailleMax >
ostream & operator << ( ostream & out, const Vecteur < T, tailleMax > & v )
{
    out << "Taille Maximale = " << tailleMax << endl; // Trace de mise au point
    out << "Nombre d'Eléments = " << v.nbElements << endl; // Trace de mise au point
    for ( int i ( 0 ) ; i < v.nbElements ; i++ )
    {
        out << "[ " << i << " ] = " << v.elements [ i ] << endl;
    }
    return out;
}

template < typename T, int tailleMax >
bool Vecteur < T, tailleMax >::Ajouter
    ( T unElement )
{
    if ( nbElements >= tailleMax )
    {
        return false;
    }
    elements [ nbElements++ ] = unElement;
    return true;
}

template < typename T, int tailleMax >
Vecteur < T, tailleMax >::Vecteur ( ) : nbElements ( 0 )
{
}
// ...

#endif // ! Defined ( VECTEUR_H )

```

```

// Programme principal de test de la classe générique Vecteur
int main ( )
{
    Vecteur < int, 10 > v1;
    v1.Ajouter ( -1 ); v1.Ajouter ( 99 );
    cout << v1 << endl;

    double x = 1.5;
    Vecteur < double, 2 > v2;
    v2.Ajouter ( -5 ); v2.Ajouter ( x );
    v2.Ajouter ( 9.9 ); // Ajout en échec : dimension du vecteur = 2
    cout << v2 << endl;

    return 0;
}

```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe Générique

■ Exemple : interface d'une classe **Pile** générique

```
#if ! defined ( PILE_H )
#define PILE_H
template < typename T >
class Pile // Pile est un patron de classe
{
public: // Quelques méthodes de la classe Pile – Définition de la forme canonique de la classe
    void Empiler ( const T & unElement );
    T Depiler ( void );
    bool EstVide ( void ) const;
    void Vider ( void );
    Pile < T > & operator = ( const Pile < T > & unePile );
    Pile ( void );
    Pile ( const Pile < T > & unePile );
    // La classe est référencée en indiquant son type entre < et > ("Pile < T >")
    ~Pile ( void );

protected :
    typedef struct unElement           // Diverses structures de données possibles :
    {                                    // Définition d'un maillon (structure unElement)
        T element;                    // avec un champ element de type T
        struct unElement *suivant;   // et un pointeur suivant sur le prochain maillon
    } tElement;

    tElement *tete;                  // Pointeur sur le sommet de la pile
};

#endif // ! defined ( PILE_H )
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Forme Canonique d'une Classe

```
----- Interface de la classe <A> (fichier A.h) -----
#ifndef ! defined ( A_H )           // Permet de manipuler un objet d'une classe de façon similaire à une variable d'un
#define A_H                         // type de base

class A                           // En cas de non-définition explicite de ces méthodes, le compilateur fournit une version
{                                 // minimalistique et souvent insuffisante en présence d'attributs dynamiques
    public :
        //----- Méthodes publiques
        //----- Surcharge d'opérateurs
        A & operator = ( const A & unA ); // Surcharge de l'affectation

        //----- Constructeurs - destructeur
        A ( ); // Constructeur par défaut

        A ( const A & unA ); // Constructeur de copie

        virtual ~A ( ); // Destructeur // Attention : si ces méthodes ne sont pas définies et si elles ne doivent pas être utilisées, il faut que leur appel provoque une erreur
                            // Déclaration de la méthode sans définition associée

    protected :
        //----- Méthodes protégées
        //----- Attributs protégés
        // Attributs accessibles à A et à ses descendants

    private :
        //----- Méthodes privées
        //----- Attributs privés
        // Attributs privés à A => accès interdit même aux descendants

};

#endif // ! defined ( A_H )
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.

Programmation Orientée Objet – C++

PLAN du COURS



Naufrage

- Généricité
- Entrées / Sorties : **iostream**
 - ◆ Hiérarchie de Classes - Organisation
 - ◆ État d'un Flux
 - ◆ Formatage
 - ◆ Manipulateurs
 - ◆ Classe **ostream / istream**
 - ◆ Classe **ofstream / ifstream / fstream**
 - ◆ Classe **stringstream**
- Standard Template Library



Divin ?

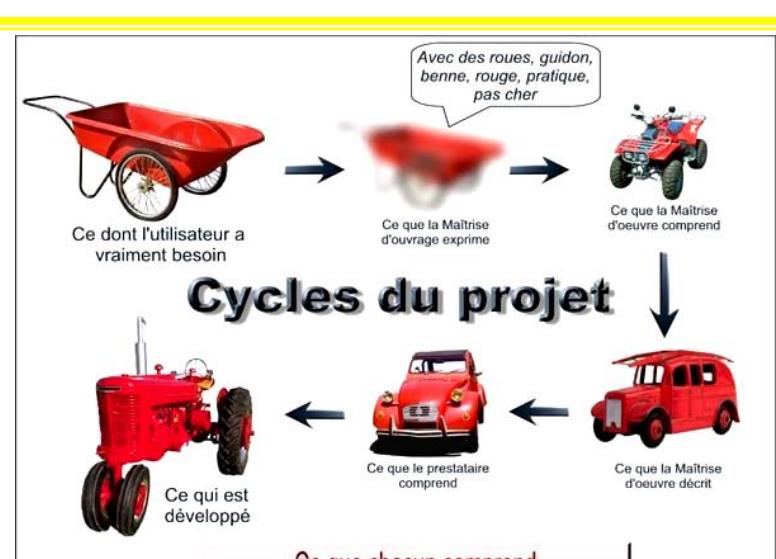


Bricolage

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur

Programmation Orientée Objet – C++



Cycles du projet

Ce dont l'utilisateur a vraiment besoin

Avec des roues, guidon, benne, rouge, pratique, pas cher

Ce que la Maîtrise d'œuvre exprime

Ce que la Maîtrise d'œuvre comprend

Ce qui est développé

Ce que le prestataire comprend

Ce que la Maîtrise d'œuvre décrit

Ce que chacun comprend...

<http://www.projetsinformatiques.com>

© MM - Reproduction interdite sans l'autorisation de l'auteur



Librairie d'Entrées / Sorties en C++

■ Généralités

- ◆ Librairie standard offrant des services d'entrées / sorties manipulant des flux (*stream* en C++)
- Flux ou *stream* : abstraction sur laquelle on effectue des opérations d'entrée et/ou de sortie (produit ou consomme des informations)
- Généralement associée à un périphérique physique, source ou destination de caractères : fichier disque, **clavier**, **écran**...
- ◆ Définition grâce à une hiérarchie de classes génériques
- Préfixe des classes : **basic_...** (**basic_fstream...** sauf **ios_base**)
- Définition des services de la librairie de manière indépendante du type
- ◆ 2 instanciations spécifiques essentielles...
 - Une pour la manipulation d'éléments de type **char** (*narrow-oriented*)
 - Correspond à la composante la plus utilisée de la librairie standard d'entrées / sorties : les classes **ios**, **istream** et **ofstream** sont *narrow-oriented*
 - Une pour la manipulation d'éléments de type **wchar_t** (*wide-oriented*)

— Même nomenclature avec un '**w**' en préfixe (**wios**, **wistream**...)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



67



Librairie d'Entrées / Sorties en C++

■ Décomposition des classes de la librairie en 2 catégories...

- ◆ Classes d'abstraction
 - Définissent une interface capable de fonctionner avec n'importe quel type de flux
 - Ne nécessitent pas de connaître la localisation exacte de la donnée (fichier, mémoire, socket...) qui est lue ou écrite
- ◆ Classes d'implémentation
 - Héritent des classes d'abstraction et définissent une implémentation spécifique pour un type précis de source de données
 - Disponibles dans la librairie standard...
 - Une implémentation pour les *streams* basés sur des fichiers
 - Une implémentation pour les *streams* basés sur des tampons mémoire (*memory buffer*)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



68

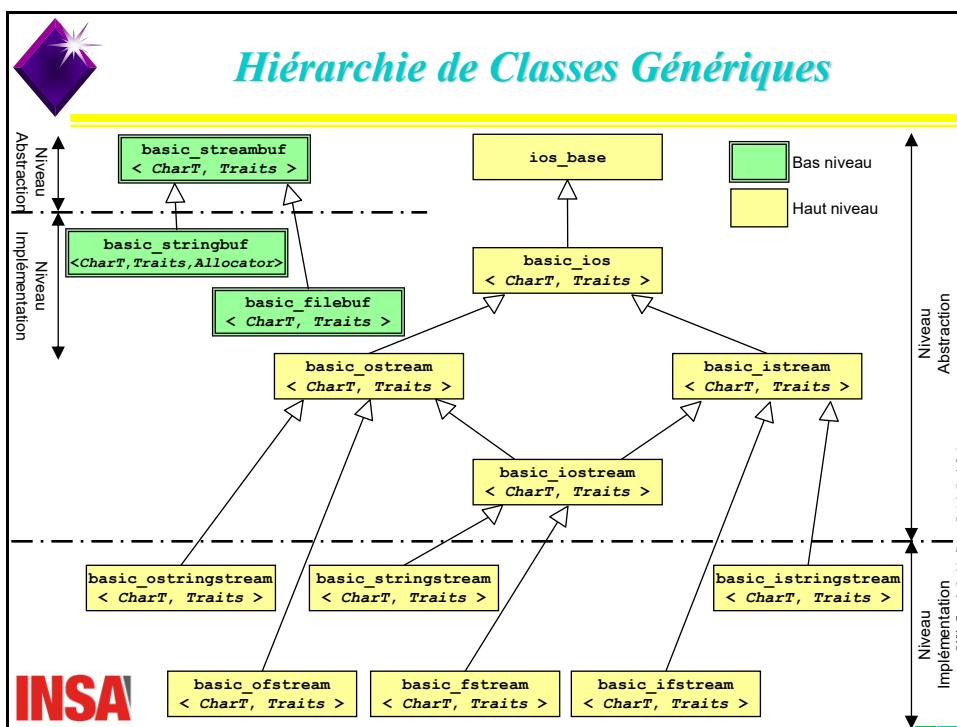


Librairie d'Entrées / Sorties en C++

- Décomposition des classes de la librairie en 2 groupes...
 - ◆ Pour les opérations de bas niveau
 - Les classes *stream buffers*
 - Elles agissent sur des caractères sans fournir aucune possibilité de formatage
 - Elles sont rarement utilisées directement
 - ◆ Pour les opérations de haut niveau
 - Les classes *stream*
 - Elles fournissent de nombreuses possibilités de formatage
 - Elles s'appuient sur les classes *stream buffers*

© MM - Reproduction interdite sans l'autorisation de l'auteur

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
 69

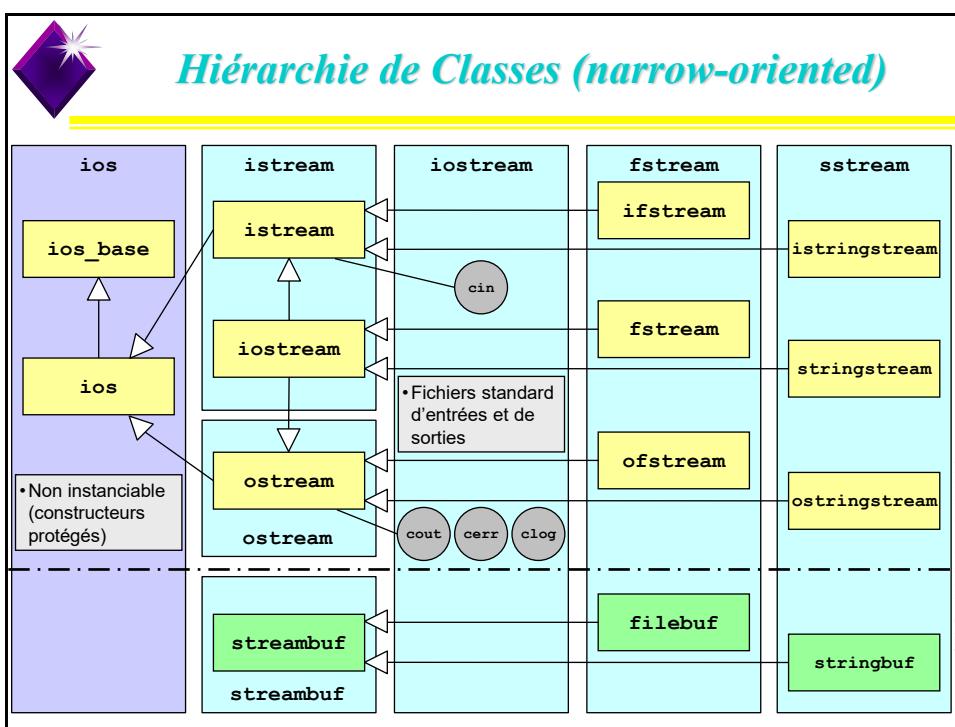




Hiérarchie de Classes Génériques

- Abstraction d'un flux...
 - Définie dans l'interface `<streambuf>`
 - `basic_streambuf` : définit une abstraction d'un périphérique *raw* (classe template)
 - Définie dans l'interface `<iostream>`
 - `ios_base` : gère les indicateurs de formatage et les exceptions d'entrées/sorties (classe)
 - `basic_ios` : gère un tampon de *stream* quelconque (classe template)
 - Définie dans l'interface `<ostream>`
 - `basic_ostream` : encapsule un périphérique abstrait donné (`std::basic_streambuf`) et fournit une interface de sortie de haut niveau avec des possibilités de formatage (classe template)
 - Définie dans l'interface `<istream>`
 - `basic_istream` : encapsule un périphérique abstrait donné (`std::basic_streambuf`) et fournit une interface d'entrée de haut niveau avec des possibilités de formatage (classe template)
 - `basic_iostream` : encapsule un périphérique abstrait donné (`std::basic_streambuf`) et fournit une interface d'entrée et de sortie de haut niveau avec des possibilités de formatage (classe template)

© MM - Reproduction interdite sans l'autorisation de l'auteur





Hierarchie de Classes (narrow-oriented) (instanciation char)

■ Les différentes classes...

- ◆ **ios_base** : classe de base pour les *streams*
- ◆ **ios** : classe de base pour les *streams* (type-dependent components)
- ◆ **istream** : classe *input stream*
- ◆ **ostream** : classe *output stream*
- ◆ **iostream** : classe *input / output stream*
- ◆ **ifstream** : classe *input file stream*
- ◆ **ofstream** : classe *output file stream*
- ◆ **fstream** : classe *input / output file stream*
- ◆ **istringstream** : classe *input string stream*
- ◆ **ostringstream** : classe *output string stream*
- ◆ **stringstream** : classe *input / output string stream*
- ◆ **streambuf** : classe de base pour les *streams buffer*
- ◆ **filebuf** : classe *file stream buffer*
- ◆ **stringbuf** : classe *string stream buffer*



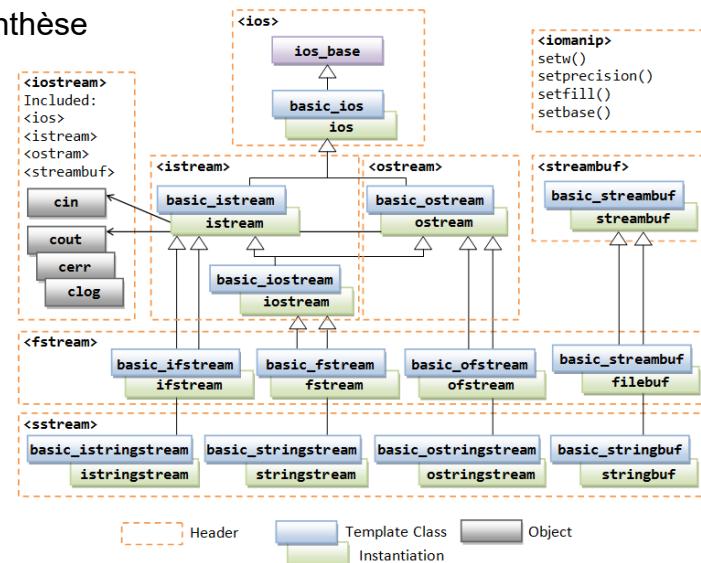
© MM - Reproduction interdite sans l'autorisation de l'auteur.

73



Classes, Instanciation, Objets et Interface

■ Synthèse



© MM - Reproduction interdite sans l'autorisation de l'auteur.

74

Issue de : https://personal.ntu.edu.sg/ehchua/programming/cpp/cp10_IO.html



Organisation de la Librairie

■ Les différents fichiers d'en-tête...

- <iostream>, <istream>, <ostream>, <streambuf> et <iosfwd>
 - Généralement, non inclus directement dans la plupart des codes C++
 - Description des classes de base de la hiérarchie et automatiquement inclus par d'autres fichiers d'en-tête de la librairie (classes dérivées)
- <iostream>
 - Déclaration des objets utilisés pour communiquer via l'entrée et la sortie standard, y compris `cin`, `cout` et `cerr`
- <fstream>
 - Définition des classes utilisées pour manipuler des fichiers à l'aide de flux ainsi que les classes des objets *stream buffer* associés à ces fichiers
- <sstream>
 - Définition des classes utilisées pour manipuler des objets chaînes de caractères comme s'ils s'agissaient de flux
- <iomanip>
 - Déclaration des manipulateurs standard à utiliser avec les opérateurs d'extraction (`operator >>`) et d'insertion (`operator <<`) pour modifier les indicateurs et les options de formatage



© MM - Reproduction interdite sans l'autorisation de l'auteur



Entrées / Sorties Basées sur les streams

■ 2 étapes dans une opération d'entrée / sortie en C++

- ◆ Formatage des données...
 - À la charge d'une classe *stream*
- ◆ Communication de la donnée vers un périphérique externe...
 - À la charge d'une classe *stream buffer*, représentation interne du *stream* dans laquelle on lit et/ou on écrit
 - Toutes les classes gérant un flux (*stream*) possèdent un objet de la classe de la classe *streambuf*
 - Instanciation de `basic_streambuf` avec les paramètres suivants :
 - `CharT` devient `char`
 - `Traits` devient `char_traits < char >`
 - Récupération possible du pointeur sur le *stream buffer* courant du flux à l'aide de la méthode `ios::rdbuf()`
 - Modification possible de ce pointeur (redirection)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur



Redirection du Flux de Sortie

```
// Opération d'E/S - Formatage de la donnée : classe stream
// Opération d'E/S - Communication de la donnée vers le périphérique externe : classe stream buffer
#include <iostream>
#include <fstream>
using namespace std;

int main ( )
{
    // Affichage formaté à l'aide de l'objet cout sur la console
    cout << "Sur la sortie standard" << endl;

    // Instanciation d'un objet fic et donc ouverture du fichier redirection.txt
    ofstream fic ("redirection.txt");

    // Redirection du stream buffer associé à l'objet cout sur le stream buffer associé à l'objet fic
    // en conservant l'adresse de l'ancien stream buffer de cout
    streambuf *oldCoutBuffer = cout.rdbuf ( fic.rdbuf ( ) );

    // Affichage formaté avec cout, redirigé dans redirection.txt à cause du changement de stream buffer
    cout << "Dans le fichier <fic=redirection.txt>" << endl;

    // Restauration du stream buffer par défaut pour cout, c'est-à-dire la console
    cout.rdbuf ( oldCoutBuffer );

    // Affichage formaté à l'aide de l'objet cout, à nouveau sur la console
    cout << "À nouveau sur la sortie standard" << endl;

    // Fermeture du fichier redirection.txt
    fic.close ( );
    return 0;
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



État d'un Flux

■ Généralités

- ◆ Chaque flux possède un vecteur de bits définissant ses indicateurs d'erreur
 - **goodbit** : activé, si l'état est correct
 - **eofbit** : activé, si la fin de fichier a été atteinte sur le flux d'entrée (lors de la dernière opération effectuée sur le flux)
 - **failbit** : activé, si la dernière opération d'entrée / sortie a échoué
 - **badbit** : activé, si la dernière opération d'entrée / sortie est invalide
- ◆ Positionné automatiquement par les opérations d'entrée / sortie

■ Manipulation...

- ◆ Globale grâce à 3 méthodes publiques de la classe **basic_ios**
 - **rdstate**, **setstate** et **clear**
- ◆ Individuelle grâce à 4 méthodes publiques de la classe **basic_ios**
 - **good**, **eof**, **fail** et **bad**



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



78



État d'un Flux

■ Exemple de manipulation du statut d'erreur

```
// Manipulation du statut d'erreur d'un flux avec la méthode publique rdstate
#include <iostream> // cerr
#include <fstream> // ifstream
using namespace std;

int main ()
{
    ifstream fic; // input stream
    fic.open ( "test.txt" ); // tentative d'ouverture du fichier test.txt
    if ( ( fic.rdstate ( ) & ifstream::failbit ) != 0 )
    {
        cerr << "Erreur d'ouverture de <test.txt>" << endl;
    }
    return 0;
}
```

⇒ Et bit à bit pour tester le vecteur de bits du flux



État d'un Flux

■ Utilisation de `operator bool`

```
explicit operator bool ( ) const;
```

- ◆ Permet de vérifier s'il n'y a pas d'erreur sur un flux
- ◆ Renvoie `true` s'il n'y a pas d'erreur sur le flux et qu'il est prêt pour des opérations d'entrée / sortie
- ◆ Ne renvoie pas la même chose que la méthode `good()` mais correspond à `!fail()`
- ◆ Utilisation possible d'un flux (ou d'une méthode qui renvoie une référence sur un flux) dans une condition ou une boucle

```
...
ifstream fic ( "test.txt" ); // ouverture du fichier
char carLu;
// prototype de la méthode utilisée dans la condition de la boucle
// istream & get ( char & c ); (retour = référence sur un flux)
while ( fic.get ( carLu ) ) // boucle de lecture d'un caractère
    cout << carLu;
...
```



État d'un Flux

- Exemple de manipulation du statut d'erreur d'un flux en utilisant **operator bool**

```
// Manipulation du statut d'erreur d'un flux en utilisant operator bool
#include <iostream> // cerr
#include <fstream> // ifstream
using namespace std;

int main ()
{
    ifstream fic;
    fic.open ( "test.txt" );
    if ( fic ) // Utilisation d'un flux dans une condition
    {
        // lecture du fichier <fic=test.txt>
    }
    else
    {
        cerr << "Erreur d'ouverture de <test.txt>" << endl;
    }
    return 0;
}
```



© MM - Reproduction interdite sans l'autorisation de l'auteur.

81



État d'un Flux

- Utilisation de **operator !** :

```
bool operator ! ( ) const;
```

- ◆ Permet de vérifier s'il n'y a pas d'erreur sur un flux
- ◆ Renvoie **true** si **failbit** ou **badbit** est positionné et **false** sinon
- ◆ Appel équivalent possible avec la méthode publique **fail**

```
if ( ! fic ) ⇔ if ( fic.fail ( ) )
où fic correspond à un objet stream
```

```
#include <iostream> // std::cerr
#include <fstream> // std::ifstream

int main ( )
{
    std::ifstream fic;
    fic.open ( "test.txt" );
    if ( ! fic )
        std::cerr << "Erreur d'ouverture de <test.txt>" << std::endl;
    return 0;
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



État d'un Flux

■ Table des combinaisons possibles

| Indicateurs <code>ios_base::iostate</code> | | | Méthodes publiques de <code>basic_ios</code> | | | | | | |
|---|----------------------|---------------------|--|----------------------|---------------------|---------------------|----------------------------|-------------------------|--|
| <code>eofbit</code> | <code>failbit</code> | <code>badbit</code> | <code>good()</code> | <code>fail()</code> | <code>bad()</code> | <code>eof()</code> | <code>operator bool</code> | <code>operator !</code> | |
| false | false | false | true | false | false | false | true | false | |
| false | false | true | false | true | true | false | false | true | |
| false | true | False | false | true | false | false | false | true | |
| false | true | true | false | true | true | false | false | true | |
| true | false | false | false | false | false | true | true | false | |
| true | false | true | false | true | true | true | false | true | |
| true | true | false | false | true | false | true | false | true | |
| true | true | true | false | true | true | true | false | true | |



Formatage avec le type `fmtflags`

■ Les indicateurs de format

- ◆ Définition avec un type `ios_base::fmtflags`
- ◆ Champ de bits (*bitmask*) pour représenter les différents indicateurs
- ◆ Utilisation de constantes nommées publiques de la classe `ios_base`
- ◆ Manipulation possible à l'aide de fonctions membres...
 - `fmtflags flags () const; // get`
 - Renvoie le formatage (indicateurs de format) actuellement actif sur le flux
 - `fmtflags flags (fmtflags fmtfl); // set`
 - Positionne le formatage (indicateurs de format définis par `fmtfl`) sur le flux
 - Les indicateurs non définis par `fmtfl` sont réinitialisés (valeurs par défaut)
 - Renvoie le formatage actif avant la modification sur le flux (restauration)
 - `void unsetf (fmtflags mask);`
 - Réinitialise les indicateurs de format du flux définis par le paramètre `mask`
 - Définition possible de `mask` avec une combinaison `ou` (opérateur ou bit à bit) d'indicateurs de format, par exemple : `ios::showpos | ios::showbase`



Formatage avec le type `fmtflags`

■ Les indicateurs de format (suite)...

- `fmtflags setf (fmtflags fmtfl); // set`
 - Positionne le formatage (indicateurs de format définis par `fmtfl`) sur le flux en laissant inchangés les autres indicateurs (pas de réinitialisation)
 - Renvoie le formatage actif avant la modification sur le flux (restauration)
 - `fmtflags setf (fmtflags fmtfl, fmtflags mask);`
 - Positionne les indicateurs de format dont les bits sont activés à la fois dans `fmtfl` et dans `mask` et réinitialise les indicateurs de format dont les bits sont définis dans `mask` mais pas dans `fmtfl`
 - Utilisation possible de `basefield`, `floatfield` ou `adjustfield` pour le paramètre `mask` avec les définitions suivantes...
 - `basefield = dec | oct | hex | 0` // choix de la base
 - `floatfield = scientific | fixed | (scientific | fixed) | 0`
 - `adjustfield = left | right | internal` // choix de l'alignement
 - Renvoie le formatage actif avant la modification sur le flux (restauration)
- ◆ **Attention :** Ne pas confondre les valeurs du type `ios_base::fmtflags` avec les manipulateurs (*voir plus loin*)
 - `ios_base::skipws` // valeur constante de type `ios_base::fmtflags`
 - `skipws` // manipulateur (fonction globale)

INSA

© MM - Reproduction interdite sans l'autorisation de l'auteur



Formatage avec le type `fmtflags`

| Champ | Membre constant | Signification quand l'indicateur (<i>flag - bit</i>) est positionné |
|--|-------------------------|---|
| Indicateurs divers | <code>boolalpha</code> | Lire et écrire les valeurs booléennes comme des chaînes (<code>true</code> et <code>false</code>) |
| | <code>showbase</code> | Écrire les valeurs entières précédées de leur indication de base (<code>0x</code> ou <code>0</code>) |
| | <code>showpoint</code> | Écrire les valeurs à virgule-flottante en incluant toujours la virgule |
| | <code>showpos</code> | Écrire les valeurs numériques non-négatives précédées d'un signe plus (+) |
| | <code>skipws</code> | Ignorer les séparateurs de tête sur certaines opérations d'entrée |
| | <code>unitbuf</code> | Vider la sortie après chaque opération d'insertion |
| | <code>uppercase</code> | Écrire des majuscules en remplacement de minuscules dans certaines opérations d'insertion (par exemple pour l'indication de base <code>hex</code>) |
| Base (<code>basefield</code>) | <code>dec</code> | Lire et écrire des valeurs entières en utilisant le format décimal (base 10) |
| | <code>hex</code> | Lire et écrire des valeurs entières en utilisant le format hexadécimal (base 16) |
| | <code>oct</code> | Lire et écrire des valeurs entières en utilisant le format octal (base 8) |
| Format <code>float</code> (<code>floatfield</code>) | <code>fixed</code> | Écrire les valeurs à virgule flottante en utilisant une notation à virgule fixe |
| | <code>scientific</code> | Écrire les valeurs à virgule flottante en utilisant une notation scientifique. |
| Alignment (<code>adjustfield</code>) | <code>internal</code> | Aligner la sortie en complétant avec le caractère de remplissage après le signe ou la base (point interne spécifié) |
| | <code>left</code> | Aligner la sortie à gauche en complétant à droite (à la fin) avec le caractère de remplissage (pour atteindre la largeur du champ) |
| | <code>right</code> | Aligner la sortie à droite en complétant à gauche (au début) avec le caractère de remplissage (pour atteindre la largeur du champ) |

© MM - Reproduction interdite sans l'autorisation de l'auteur



Formatage avec la Largeur du Champ

■ La largeur du champ de sortie : fonction membre `width`

- ◆ Manipulation possible à l'aide de fonctions membres...
 - `streamsize width () const; // get`
 - Renvoie la valeur courante de la largeur du champ de sortie
 - `streamsize width (streamsize wide); // set`
 - Définit une nouvelle largeur (`wide`) du champ de sortie pour le flux
 - Renvoie la valeur actuelle de la largeur du champ de sortie
- ◆ Détermine le nombre minimum de caractères à écrire...
 - Si l'information à écrire est plus courte que la largeur du champ, elle est ajustée avec des caractères de remplissage (fonction membre `fill`), en fonction de l'indicateur `adjustfield(left, right ou internal)`

```
#include <iostream>
int main ()
{ std::cout << 123 << std::endl;
  std::cout.width ( 8 );
  std::cout << 123 << std::endl;
  std::cout.fill ( '*' ); std::cout.width ( 12 );
  std::cout << std::left << 123 << std::endl;
  return 0;
}
```

// Affichage
123
 123
123*****



© MM - Reproduction interdite sans l'autorisation de l'auteur

87



Formatage avec la Précision

■ La précision d'affichage des valeurs réelles (6 par défaut)

- ◆ Manipulation possible à l'aide de fonctions membres...
 - `streamsize precision () const; // get`
 - Renvoie la valeur courante de la précision (édition en virgule flottante)
 - `streamsize precision (streamsize prec); // set`
 - Définit une nouvelle précision (`prec`) pour l'affichage des réels (`float` ou `double`) pour le flux de sortie
 - Renvoie la valeur courante de la précision du format d'affichage des réels
- ◆ Détermine le nombre maximum de chiffres à indiquer lors d'une opération d'insertion, mais l'interprétation dépend de `floatfield`

```
int main ( )
{ double pi = 3.14159;
  std::cout.unsetf ( std::ios::floatfield );
  std::cout.precision ( 5 ); // interprétation par défaut
  std::cout << pi << std::endl; // ni scientific, ni fixed
  std::cout.precision ( 10 );
  std::cout << pi << std::endl; // ni scientific, ni fixed
  std::cout.setf( std::ios::fixed, std::ios::floatfield );
  std::cout << pi << std::endl; // interprétation fixed
  return 0;
}
```

// Affichage
3.1416
3.14159
3.1415900000



© MM - Reproduction interdite sans l'autorisation de l'auteur

88



Formatage avec le Caractère de Remplissage

■ Le caractère de remplissage : fonction membre `fill`

- ◆ Manipulation possible à l'aide de fonctions membres...

- `char fill () const; // get`

- Renvoie la valeur actuelle du caractère de remplissage pour les alignements

- `char fill (char fillch); // set`

- Définit le nouveau caractère (`fillch`) de remplissage pour le flux

- Renvoie la valeur du caractère de remplissage avant l'appel (restauration)

```
#include <iostream>

int main ( )
{
    char old;
    std::cout.width ( 8 );
    std::cout << 123 << std::endl;

    old = std::cout.fill ( '#' );
    std::cout.width ( 8 );
    std::cout << 123 << std::endl;

    std::cout.fill ( old );
    return 0;
}
```

// Affichage
123
#####123



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur.

89



Manipulateurs

■ Généralités (`#include <iomanip>`)

- ◆ Fonctions globales à actions locales
- ◆ À utiliser avec les opérations d'insertion (`<<`) ou d'extraction (`>>`) sur des objets *streams iostream*
- ◆ Modifient le comportement, les propriétés et les caractéristiques de formatage des *streams*

■ Syntaxe d'utilisation

- ◆ Appel de fonction

```
nomManipulateur ( nomFlux );
```

- Par exemple: `endl (cout);`

- ◆ Utilisation de la surcharge `operator <<` ou `operator >>`

```
nomFlux << nomManipulateur;
```

- Par exemple: `cout << endl;`



© MM - Reproduction interdite sans l'autorisation de l'auteur.

90



Manipulateurs

- Liste des manipulateurs de la librairie standard...

- ◆ **`ios_base & boolalpha`** (`ios_base & str`);
 - Utilise les valeurs chaîne de caractères "true" et "false" pour l'affichage de valeurs booléennes (fonction dans `ios`)
- ◆ **`ios_base & dec`** (`ios_base & str`);
 - Utilise la base décimale pour les valeurs entières (fonction dans `ios`)
- ◆ **`ostream & endl`** (`ostream & os`);
 - Insère une nouvelle ligne ('\n') et vide le *stream buffer* (fonction dans `ostream`)
- ◆ **`ostream & ends`** (`ostream & os`);
 - Insère un caractère nul ('\0') dans le flux (utilisation avec `std::ostrstream`) (fonction dans `ostream`)
- ◆ **`ios_base & fixed`** (`ios_base & str`);
 - Utilise une notation virgule fixe (fonction dans `ios`)
- ◆ **`ostream & flush`** (`ostream & os`);
 - vide le *stream buffer* (fonction dans `ostream`)

INSA



© MM - Reproduction interdite sans l'autorisation de l'auteur

91



Manipulateurs

- Liste des manipulateurs de la librairie standard (suite)...

- ◆ **`ios_base & hex`** (`ios_base & str`);
 - Utilise la base hexadécimale pour les valeurs entières (fonction dans `ios`)
- ◆ **`ios_base & internal`** (`ios_base & str`);
 - Aligne la sortie en complétant avec le caractère de remplissage (à définir avec le manipulateur `setfill`) après le signe ou la base. Par exemple, pour un champ de largeur 8 caractères et un caractère de remplissage '#' : -###9.87 ou 0x####3F (fonction dans `ios`)
- ◆ **`ios_base & left`** (`ios_base & str`);
 - Aligne la sortie à gauche en complétant à droite avec le caractère de remplissage (à définir avec le manipulateur `setfill`). Par exemple, pour un champ de largeur 6 caractères et un caractère de remplissage '*' : -1.23* ou 0xAB** (fonction dans `ios`)
- ◆ **`ios_base & noboolalpha`** (`ios_base & str`);
 - Utilise les valeurs entières 1 et 0 pour l'affichage de valeurs booléennes (fonction dans `ios`)

INSA



© MM - Reproduction interdite sans l'autorisation de l'auteur

92



Manipulateurs

- Liste des manipulateurs de la librairie standard (suite)...

- ◆ **`ios_base & noshowbase`** (`ios_base & str`) ;
 - N'affiche pas le préfixe identifiant la base (0x pour une valeur hexadécimale et 0 pour une valeur octale) (fonction dans `ios`)
- ◆ **`ios_base & noshowpoint`** (`ios_base & str`) ;
 - Affiche la virgule, uniquement si cela est nécessaire, pour les valeurs *virgule flottante* (fonction dans `ios`)
- ◆ **`ios_base & noshowpos`** (`ios_base & str`) ;
 - N'affiche pas le signe '+' devant les valeurs numériques positives (fonction dans `ios`)
- ◆ **`ios_base & noskipws`** (`ios_base & str`) ;
 - Toutes les opérations sur le *stream* considèrent les séparateurs comme des caractères valides qu'il faut extraire (fonction dans `ios`)
- ◆ **`ios_base & nounitbuf`** (`ios_base & str`) ;
 - Ne force pas le vidage du tampon après chaque opération d'insertion dans le *stream* (fonction dans `ios`)



© MMI - Reproduction interdite sans l'autorisation de l'auteur



Manipulateurs

- Liste des manipulateurs de la librairie standard (suite)...

- ◆ **`ios_base & nouppercase`** (`ios_base & str`) ;
 - Ne génère pas de lettres majuscules, par exemple en préfixe d'une valeur hexadécimale (fonction dans `ios`)
- ◆ **`ios_base & oct`** (`ios_base & str`) ;
 - Utilise la base octale pour les valeurs entières (fonction dans `ios`)
- ◆ **`/* unspecified */ resetiosflags (ios_base::fmtflags mask)`** ;
 - Réinitialise les indicateurs de formatage selon la valeur du paramètre `mask` (opérateur | (ou bit à bit) possible pour la construction de la valeur du paramètre) (fonction dans `iomanip`)
- ◆ **`ios_base & right`** (`ios_base & str`) ;
 - Aligne la sortie à droite en complétant à gauche avec le caractère de remplissage (à définir avec le manipulateur `setfill`) Par exemple, pour un champ de largeur 6 caractères et avec le caractère de remplissage '\$' : \$\$5.55 ou \$\$0xFF (fonction dans `ios`)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MMI - Reproduction interdite sans l'autorisation de l'auteur





Manipulateurs

- Liste des manipulateurs de la librairie standard (suite)...

- ◆ **`ios_base & scientific`** (`ios_base & str`) ;
 - Utilise une notation scientifique (fonction dans `ios`)
- ◆ **`/* unspecified */ setbase`** (`int base`) ;
 - Définit la base pour les valeurs entières à partir de son paramètre `base` (fonction dans `iomanip`)
- ◆ **`/* unspecified */ setfill`** (`char_type c`) ;
 - Définit le caractère `c` comme caractère de remplissage (fonction dans `iomanip`)
 - `char_type` = type du caractère utilisé par le *stream*, c'est-à-dire le premier paramètre template *CharT* de la classe
- ◆ **`/* unspecified */ setiosflags`** (`ios_base::fmtflags mask`) ;
 - Positionne les indicateurs de formatage selon la valeur du paramètre `mask` (opérateur `|` (ou bit à bit) possible pour la construction de la valeur du paramètre) (fonction dans `iomanip`)



Manipulateurs

- Liste des manipulateurs de la librairie standard (suite)...

- ◆ **`/* unspecified */ setprecision`** (`int n`) ;
 - Définit, à l'aide de son paramètre `n`, la précision lors de l'affichage de valeurs en virgule flottante (fonction dans `iomanip`)
- ◆ **`/* undefined */ setw`** (`int n`) ;
 - Définit la largeur du champs de sortie (valeur du paramètre `n`) à utiliser lors des opérations de sortie (fonction dans `iomanip`)
- ◆ **`ios_base & showbase`** (`ios_base & str`) ;
 - Affiche le préfixe identifiant la base (`0x` pour une valeur hexadécimale et `0` pour une valeur octale). Par défaut, cet indicateur n'est pas positionné pour les *streams* standard (fonction dans `ios`)
- ◆ **`ios_base & showpoint`** (`ios_base & str`) ;
 - Affiche toujours la virgule pour les valeurs en *virgule flottante* avec un possible ajout de `0` pour respecter la précision (fonction dans `ios`)
- ◆ **`ios_base & showpos`** (`ios_base & str`) ;
 - Insère un signe `+` devant chaque valeur numérique non négative, y compris la valeur `0` (fonction dans `ios`)



Manipulateurs

- Liste des manipulateurs de la librairie standard (suite)...

- ◆ **ios_base & skipws** (**ios_base & str**);
 - Ignore les caractères séparateurs (*tabulation, retour chariot et espace*) lors d'une opération d'entrée (fonction dans **ios**)
- ◆ **ios_base & unitbuf** (**ios_base & str**);
 - Force le vidage du tampon après chaque opération d'insertion dans le *stream* (fonction dans **ios**)
- ◆ **ios_base & uppercase** (**ios_base & str**);
 - Utilise, dans les opérations de sortie, des majuscules (à la place des minuscules) quand cela est nécessaire (par exemple, en préfixe d'une valeur hexadécimale) (fonction dans **ios**)
- ◆ **istream & ws** (**istream & is**);
 - Extrait (en les rejetant) les séparateurs du *stream* d'entrée, à partir de la position courante
 - S'arrête au premier non-séparateur (fonction dans **istream**)



fmtflags et/ou Manipulateurs

```
// Utilisation de ios_base::fmtflags (type et constantes publiques) et des manipulateurs
#include <iostream>

int main ()
{
    // Utilisation de fmtflags comme type de constantes publiques, membres de la classe ios_base
    std::cout.setf ( std::ios_base::hex, std::ios_base::basefield );
    std::cout.setf ( std::ios_base::showbase );
    std::cout << 123 << std::endl;

    // Utilisation de fmtflags comme type de constantes publiques, membres hérités de la classe ios
    std::cout.setf ( std::ios::hex, std::ios::basefield );
    std::cout.setf ( std::ios::showbase );
    std::cout << 123 << std::endl;

    // Utilisation de fmtflags comme type de constantes publiques manipulées par l'objet cout
    std::cout.setf ( std::cout.hex, std::cout.basefield );
    std::cout.setf ( std::cout.showbase );
    std::cout << 123 << std::endl;

    // Utilisation de fmtflags comme un type (variable flags)
    std::ios_base::fmtflags flags;
    flags = std::cout.flags(); // Récupération des indicateurs actuellement actifs sur le flux cout (bitmask)
    flags &= ~std::cout.basefield; // Réinitialisation des bits basefield (dec, oct et hex) (opération & et ~)
    flags |= std::cout.hex; // Mise en place du bit hex (opération bit à bit |)
    flags |= std::cout.showbase; // Mise en place du bit showbase (opération bit à bit |)
    std::cout.flags ( flags ); // Mise en place des nouveaux indicateurs sur le flux cout
    std::cout << 123 << std::endl;

    // Utilisation des manipulateurs sans utiliser fmtflags
    std::cout << std::hex << std::showbase << 123 << std::endl;
}

return 0;
}
```

⇒ Écriture plus compacte et plus lisible...



Écriture de son Propre Manipulateur

■ Définition classique d'un manipulateur sans paramètre...

```
ostream & flush ( ostream & os )
{   return os.flush ( );
}

ostream & endl ( ostream & os )
{   return os << '\n' << flush;
}
```

- ◆ Usage classique (par exemple pour `endl`)

```
cout << endl;
```

- ◆ Surcharge obligatoire de `operator <<` dans la classe `ostream`

```
ostream & ostream::operator << ( ostream & (*pf) ( ostream & ) )
{
    // pf : pointeur vers une fonction qui admet un ostream en paramètre par référence
    // et qui renvoie une référence sur un ostream
    (*pf) ( *this );
    return *this;
}
```



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Écriture de son Propre Manipulateur

```
// Écriture et utilisation de ses propres manipulateurs sans paramètre
#include <iostream>
using namespace std;

const char ROUGE [ ] = { 033,['','3','1','m' } ;
const char RESET [ ] = { 033,['','m',017 } ;

ostream & rouge ( ostream & os )
{
    os.write ( ROUGE, sizeof (ROUGE) );
    return os;
}

ostream & reset ( ostream & os )
{
    os.write ( RESET, sizeof (RESET) );
    return os;
}

int main ( )
{
    cout << rouge << "ATTENTION !" << reset << endl; // Affichage en rouge
    cout << "Plus de danger..." << endl; // Affichage en noir
    return 0;
}
```

// Rappel : Pour réaliser son propre manipulateur, il faut
// utiliser la surcharge de `operator <<` dans la classe
// `ostream` (cf. Classe `ostream::operator <<`)

```
ostream & ostream::operator <<
        ( ostream & (*pf) ( ostream & ) )
{
    (*pf) ( *this );
    return *this;
}
```

...> manipulateur.
ATTENTION !
 Plus de danger...
 ...>





Écriture d'un Manipulateur Paramétré

■ Manipulateur...

- ◆ Fonction passée en paramètre aux opérateurs d'insertion et d'extraction dans un flux
- ◆ Fonction appelée par ces opérateurs

■ Création d'une classe correspondant au manipulateur...

- ◆ **Constructeur avec paramètres** : correspondant aux paramètres du manipulateur
- ◆ Stockage des valeurs des paramètres du constructeur dans des attributs privées de la classe
- ◆ Définition d'une **fonction amie** à la classe, correspondant à la surcharge de l'**operator <<** (ou **operator >>**) pour les objets de la classe du manipulateur



Écriture d'un Manipulateur Paramétré

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

enum IndexCouleur { NOIR, ROUGE, VERT, JAUNE, BLEU, MAGENTA, CYAN, BLANC, RESET };
static const char * const COULEUR [ ] = {
    "\033[30m", /* Noir */ "\033[31m", /* Rouge */ "\033[32m", /* Vert */
    "\033[33m", /* Jaune */ "\033[34m", /* Bleu */ "\033[35m", /* Magenta */
    "\033[36m", /* Cyan */ "\033[37m", /* Blanc */ "\033[m\017" /* Reset */
};

class CouleurTTY
{ public :
    explicit CouleurTTY ( IndexCouleur i ) : index ( i ) { }
    inline friend ostream & operator << ( ostream & os, const CouleurTTY & manip )
    { os.write ( COULEUR [ manip.index ], strlen ( COULEUR [ manip.index ] ) );
        return os.flush ( );
    }
    private :
    IndexCouleur index;
};

int main ( )
{ cout << CouleurTTY ( BLEU ) << "BLEU " << CouleurTTY ( ROUGE ) << "ROUGE "
    << CouleurTTY ( VERT ) << "VERT" << CouleurTTY ( RESET ) << endl;
    return 0;
}
```

...> manipulateur,>
BLEU ROUGE VERT
...>

⇒ Constructeur paramétré de la classe CouleurTTY

⇒ Surcharge de **operator <<** pour des objets de la classe CouleurTTY, correspondant au manipulateur paramétré

⇒ Sauvegarde du paramètre du manipulateur avant l'appel par l'opérateur d'insertion (<<) dans un flux



Écriture d'un Manipulateur Paramétré

```
// Construction d'un manipulateur paramétré ignoreLigne pour ignorer nb lignes
#include <iostream>
#include <limits>
using namespace std;

class ignoreLigne
{
public :
    explicit ignoreLigne ( unsigned int nb = 1 ) : nbLignes ( nb ) { }

    template < typename CharT, typename Traits >
    friend basic_istream < CharT, Traits > & operator >>
        ( basic_istream < CharT, Traits > & is, const ignoreLigne & ignLig )
    {
        // On ignore les caractères jusqu'à la fin de la ligne et on répète cela nbLignes fois
        for ( unsigned int i = 0 ; i < ignLig.nbLignes ; i++ )
        {
            is.ignore ( numeric_limits < streamsize >::max ( ), '\n' );
        }
        // On renvoie le flux pour pouvoir enchaîner les opérations
        return is;
    }

private :
    unsigned int nbLignes;
};


```

⇒ Manipulateur paramétré ⇒ construction d'une classe

⇒ Sauvegarde du paramètre du manipulateur avant l'appel par l'opérateur d'insertion (<<) dans un flux

⇒ Surcharge de `operator <<` pour des objets de la classe `ignoreLigne`, correspondant au manipulateur paramétré

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe ostream

- Généralités (`#include <ostream>`)

```
ios_base --> ios --> ostream --> iostream
                                         |           |
                                         |           v
                                         |           ofstream
                                         |
                                         v
                                         ostringstream
```

- ◆ Objets standard `cout`, `cerr` et `clog` instances de cette classe
- ◆ Méthodes publiques – Présentation rapide...
 - Sortie formatée
 - `operator <<` : opérateur d'insertion formatée dans un flux de sortie
 - Sortie non formatée
 - `put` : insère un caractère dans un flux
 - `write` : insère un bloc de données (séquence de caractères) dans un flux
 - Positionnement
 - `tellp` : retourne la position courante dans un flux
 - `seekp` : positionne le pointeur courant dans un flux
 - Synchronisation
 - `flush` : vide le *stream buffer*

104

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe `ostream` : Méthodes Publiques

- Description détaillée des méthodes publiques...
 - ◆ `ostream & flush () ;`
 - vide le *stream buffer* (indicateurs `failbit` et `badbit`)
 - Un manipulateur de même nom existe, avec le même comportement
 - ◆ `ostream & put (char c) ;`
 - Insère le caractère `c` dans le flux de sortie (positionnement des indicateurs `failbit` et `badbit`, si nécessaire)
 - ◆ `streampos tellp () ;`
 - Retourne la position du caractère courant dans le flux de sortie (positionnement des indicateurs `failbit` et `badbit`, si nécessaire)
 - Fonctionne même si `eofbit` est activé
 - `streampos` est un type `pos`, convertible vers un type entier
 - ◆ `ostream & write (const char * s, streamsize n) ;`
 - Insère les premiers `n` caractères de la chaîne pointée par `s` dans le flux sans aucune vérification ⇒ copie possible de caractères nuls sans arrêt de l'écriture (positionnement des indicateurs `failbit` et `badbit`, si nécessaire)

INSA

© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe `ostream` : Méthodes Publiques

- Description détaillée des méthodes publiques (suite)...
 - ◆ `ostream & seekp (streampos p) ;`
 - Positionne le pointeur dans le flux de sortie à l'endroit où le prochain caractère sera inséré (indicateurs `failbit` et `badbit`)
 - `p` définit la nouvelle position en absolu dans le flux de sortie (à partir du début du flux)
 - `streampos` est un type `pos`, convertible vers un type entier
 - ◆ `ostream & seekp (streamoff o, ios_base::seekdir s) ;`
 - Positionne le pointeur dans le flux de sortie à l'endroit où le prochain caractère sera inséré (indicateurs `failbit` et `badbit`)
 - `o` donne la valeur de l'offset (déplacement) en fonction du point de départ défini par `s`
 - `streamoff` est un type `offset`, généralement un type entier signé
 - `s` définit le point de départ du déplacement
 - `ios_base::beg` : à partir du début du flux de sortie
 - `ios_base::cur` : à partir de la position courante dans le flux de sortie
 - `ios_base::end` : à partir de la fin du flux de sortie

INSA



© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe ostream : Méthodes Publiques

```
// Petit exemple de synthèse pour les méthodes publiques de la classe ostream
#include <fstream>

int main ()
{
    // Instanciation d'un objet fic
    std::ofstream fic;

    // Ouverture du fichier test.txt
    fic.open ( "test.txt" );

    // Écriture des 7 premiers caractères de la chaîne dans le flux fic
    fic.write ( "Bonjour Monsieur", 7 );

    // Définition de la position courante dans le flux de sortie fic
    long position = fic.tellp ( );
    // Déplacement en absolu, dans le flux de sortie fic, à la position position - 4
    fic.seekp ( position - 4 );

    // Écriture des 4 premiers caractères de la chaîne dans le flux fic, à partir de la position courante
    fic.write ( "soir", 4 );

    // Fermeture du fichier test.txt
    fic.close ( );

    return 0;
}
```

⇒ Autres possibilités...
 // -4 à partir de la fin du fichier fic
 fic.seekp (-4, std::ios_base::end);
 // -4 depuis la position courante dans fic
 // position courante = fin du fichier
 fic.seekp (-4, std::ios_base::cur);
 // +3 à partir du début du fichier fic
 fic.seekp (3, std::ios_base::beg);

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe ostream : operator <<

- Surcharge pour les types arithmétiques (**fonctions membres** de la classe ostream)

```
ostream & operator<< ( bool v );           ostream & operator<< ( long long v );
ostream & operator<< ( short v );          ostream & operator<< ( unsigned long long v );
ostream & operator<< ( unsigned short v ); ostream & operator<< ( float v );
ostream & operator<< ( int v );             ostream & operator<< ( double v );
ostream & operator<< ( unsigned int v );   ostream & operator<< ( long double v );
ostream & operator<< ( long v );            ostream & operator<< ( void * p );
ostream & operator<< ( unsigned long v );
```

- Surcharge pour les *streams buffer* (**fonctions membres** de la classe ostream)

```
ostream & operator<< ( streambuf * sb );
```

- Surcharge pour les manipulateurs (**fonctions membres** de la classe ostream)

```
ostream & operator<< ( ostream & (*pf) ( ostream & ) );
ostream & operator<< ( ios & (*pf) ( ios & ) );
ostream & operator<< ( ios_base & (*pf) ( ios_base & ) );
```



Classe ostream : operator <<

- Surcharge pour les caractères (fonctions globales de la classe iostream)

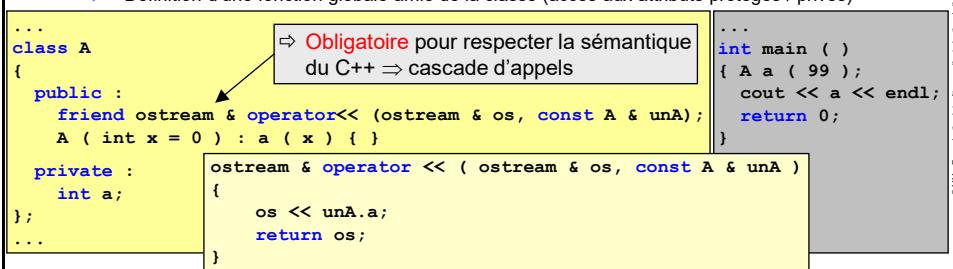
```
ostream & operator << ( ostream & os, char c );
ostream & operator << ( ostream & os, signed char c );
ostream & operator << ( ostream & os, unsigned char c );
```

- Surcharge pour les chaînes (fonctions globales de la classe iostream)

```
ostream & operator << ( ostream & os, const char * s );
ostream & operator << ( ostream & os, const signed char * s );
ostream & operator << ( ostream & os, const unsigned char * s );
```

- Surcharge souhaitable (obligatoire) pour les types (classes) de l'utilisateur

- Définition d'une fonction globale amie de la classe (accès aux attributs protégés / privés)

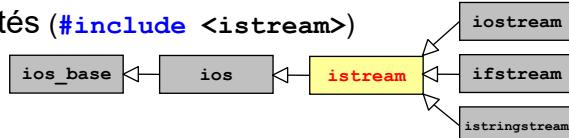


© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe istream

- Généralités (#include <iostream>)



- Objets standard cin instance de cette classe
- Instanciation de basic_istream avec les paramètres suivants :
 - CharT ≡ char et Traits ≡ char_traits < char >
- Méthodes publiques – Présentation rapide...
 - Entrée formatée
 - operator >> : opérateur d'extraction formatée dans un flux d'entrée
 - Positionnement
 - tellg : récupère la position courante dans la séquence d'entrée
 - seekg : définit la position courante dans la séquence d'entrée
 - Synchronisation
 - sync : synchronise le buffer d'entrée



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON





Classe istream

■ Généralités (`#include <iostream>`)

- ◆ Méthodes publiques – Présentation rapide (suite)...
 - Entrée non formatée
 - `gcount` : récupère le nombre de caractères lus
 - `get` : récupère un caractère dans le flux d'entrée
 - `getline` : récupère une ligne de caractères dans le flux d'entrée
 - `ignore` : extrait et omet des caractères du flux d'entrée
 - `peek` : récupère le caractère suivant sans le détruire dans le flux d'entrée
 - `read` : récupère un bloc de données (séquence de caractères) dans le flux
 - `readsome` : récupère un bloc de données (séquence de caractères) à partir d'un flux asynchrone
 - `putback` : remet un caractère dans le flux d'entrée
 - `unget` : remet un caractère dans le flux d'entrée (similaire à `putback`)



© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe istream : Méthodes Publiques

■ Description détaillée des méthodes publiques...

- ◆ `streamsize gcount () const;`
 - Renvoie le nombre de caractères extrait lors de la dernière opération d'entrée non formatée (`get`, `getline`, `ignore`, `read` et `readsome`)
 - Pour `peek`, `putback` et `unget`, renvoie est 0 (pas d'extraction)
- ◆ `int get (); OU istream & get (char & c);`
 - Extrait un unique caractère du flux
- ◆ `istream & get (char * s, streamsize n);`
`istream & get (char * s, streamsize n, char delim);`
 - Extrait des caractères du flux et les range à l'adresse `s`, jusqu'au moment où...
 - Soit `n-1` caractères ont été extraits
 - Soit le caractère de délimitation ('`\n`' ou `delim`) est rencontré

Attention : le caractère de délimitation n'est jamais extrait
 - Un caractère nul ('`\0`') est automatiquement ajouté à `s` (chaîne C++) si `n > 0`, même si une chaîne vide est extraite



© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe `istream` : Méthodes Publiques

■ Description détaillée des méthodes publiques (suite)...

- ◆ `istream & get (streambuf & sb) ;`
`istream & get (streambuf & sb, char delim) ;`
 - Extrait des caractères du flux d'entrée et les range dans la séquence de sortie contrôlée par l'objet *stream buffer* `sb`, jusqu'au moment où...
 - Soit, l'insertion dans `sb` est en échec
 - Soit, le caractère de délimitation ('`\n`' ou `delim`) est rencontré (non extrait)
 - **Attention** : seuls les caractères insérés avec succès dans `sb` sont effectivement extraits du flux
- ◆ `istream& getline (char *s, streamsize n) ;`
`istream& getline (char *s, streamsize n, char delim) ;`
 - Extrait des caractères du flux d'entrée (entrée non formatée) et les range à l'adresse `s` (*c-string*), jusqu'au moment où...
 - Soit `n-1` caractères ont été extraits (indicateur `failbit`)
 - Soit le caractère de délimitation ('`\n`' ou `delim`) est rencontré
 - **Attention** : le caractère de délimitation est extrait mais pas rajouté à `s`
- Un caractère nul ('`\0`') est automatiquement ajouté à `s`, si `n > 0`, même si une chaîne vide est extraite

© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe `istream` : Méthodes Publiques

■ Description détaillée des méthodes publiques (suite)...

- ◆ `istream & ignore (streamsize n=1, int delim=EOF) ;`
 - Extrait des caractères du flux d'entrée et les ignore, jusqu'au moment où...
 - Soit, `n` caractères ont été extraits
 - Soit, l'un des caractères est égal à `delim` (ce caractère est extrait)
- ◆ `int peek () ;`
 - Renvoie le prochain caractère du flux d'entrée mais sans l'extraire
- ◆ `istream & putback (char c) ;`
 - Tente de remettre le caractère `c` dans le flux d'entrée, pour le rendre à nouveau accessible à une prochaine opération d'entrée
 - Si `eofbit` est positionné, l'opération échoue (positionne `failbit`)
- ◆ `istream & read (char * s, streamsize n) ;`
 - Extrait `n` caractères du flux d'entrée et les range à l'adresse `s` (sans aucune vérification des caractères extraits, par exemple '`\0`')
 - Positionne `eofbit` et `failbit`, si `EOF` avant l'extraction des `n` caractères





Classe `istream` : Méthodes Publiques

- Description détaillée des méthodes publiques (suite)...
 - ◆ `istream & seekg (streampos pos) ;`
 - Définit la position (paramètre `pos`) du prochain caractère à extraire du flux d'entrée (la position est définie par rapport au début du flux)
 - Si `eofbit` est positionné, l'opération échoue (positionne `failbit`)
 - ◆ `istream & seekg (streamoff off, ios_base::seekdir way) ;`
 - Positionne le pointeur dans le flux d'entrée à l'endroit où le prochain caractère sera extrait
 - `off` donne la valeur de l'offset (déplacement) en fonction du point de départ défini par `way`
 - `streamoff` est un type `offset`, généralement un type entier signé
 - `way` définit le point de départ du déplacement
 - `ios_base::beg` : à partir du début du flux d'entrée
 - `ios_base::cur` : à partir de la position courante dans le flux d'entrée
 - `ios_base::end` : à partir de la fin du flux d'entrée
 - Si `eofbit` est positionné, l'opération échoue (positionne `failbit`)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

115



Classe `istream` : Méthodes Publiques

- Description détaillée des méthodes publiques (suite)...
 - ◆ `int sync () ;`
 - Synchronise le stream buffer avec le flux d'entrée
 - Pour `cin`, cela peut conduire au vidage du flux d'entrée
 - ◆ `streampos tellg () ;`
 - Retourne la position du caractère courant dans le flux d'entrée
 - Fonctionne même si `eofbit` est activé
 - `streampos` est un type `pos`, convertible vers un type entier
 - ◆ `istream & unget () ;`
 - Essaie de diminuer la position actuelle dans le flux d'entrée d'un caractère pour rendre à nouveau accessible, à la prochaine opération d'entrée, le dernier caractère extrait
 - Si `eofbit` est positionné, l'opération échoue (positionne `failbit`)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

116



Classe istream : operator >>

- Surcharge pour les types arithmétiques (**fonctions membres** de la classe **istream**)

```
istream & operator>> ( bool & v );           istream & operator>> ( long long & v );
istream & operator>> ( short & v );          istream& operator>> (unsigned long long& v );
istream & operator>> ( unsigned short& v ); istream & operator>> ( float & v );
istream & operator>> ( int & v );            istream & operator>> ( double & v );
istream & operator>> ( unsigned int & v ); istream & operator>> ( long double & v );
istream & operator>> ( long & v );           istream & operator>> ( void * & p );
istream & operator>> ( unsigned long& v );
```

- Surcharge pour les *streams buffer* (**fonctions membres** de la classe **istream**)

```
istream & operator>> ( streambuf * sb );
```

- Surcharge pour les manipulateurs (**fonctions membres** de la classe **istream**)

```
istream & operator>> ( istream & (*pf) ( istream & ) );
istream & operator>> ( ios & (*pf) ( ios & ) );
istream & operator>> ( ios_base & (*pf) ( ios_base & ) );
```



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe istream : operator >>

- Surcharge pour les caractères (**fonctions globales** de la classe **iostream**)

```
istream & operator >> ( istream & is, char & c );
istream & operator >> ( istream & is, signed char & c );
istream & operator >> ( istream & is, unsigned char & c );
```

- Surcharge pour les chaînes (**fonctions globales** de la classe **iostream**)

```
istream & operator >> ( istream & is, char * s );           // absence de const sur s
istream & operator >> ( istream & is, signed char * s ); // absence de const sur s
istream & operator >> ( istream & is, unsigned char * s ); // absence de const sur s
```

- Surcharge souhaitable (obligatoire) pour les types (classes) de l'utilisateur

- Définition d'une fonction globale amie de la classe (accès aux attributs protégés / privés)

```
...
class A
{
public :
    friend istream & operator >> ( istream & is, A & unA );
    A ( int x = 0 ) : a ( x ) { }

private :
    istream & operator >> ( istream & is, A & unA )
    {
        int a;
        is >> unA.a;
    }
    ...
}
```

⇒ Obligatoire pour respecter la sémantique
du C++ ⇒ cascade d'appels

⇒ Attention : absence de const
⇒ modification de l'objet unA

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

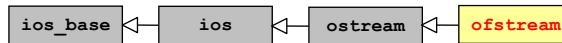
int main ( )
{
    A a ( 99 );
    cin >> a;
    cout << a << endl;
    return 0;
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe ofstream

■ Généralités (`#include <fstream>`)



- ◆ La classe `ofstream` (*output file stream*) permet de manipuler des fichiers en sortie
- ◆ Les instances de cette classe s'appuient sur un objet `filebuf` pour la gestion de leur *stream buffer* (bas niveau)
- ◆ Ouverture du fichier...
 - Par construction d'un objet (instanciation)
 - Par appel à la méthode `open`
- ◆ Méthodes publiques – Présentation rapide...
 - `open` : ouvre le fichier
 - `is_open` : vérifie si le fichier est ouvert
 - `close` : ferme le fichier
 - `rdbuf` : récupère le *stream buffer* associé au fichier
 - `operator =` : non présentée dans ce cours (C++11)
 - `swap` : non présentée dans ce cours (C++11)



© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe ofstream : Méthodes Publiques

■ Description détaillée des méthodes publiques...

- ◆ `void close ();`
 - Ferme le fichier associé à l'objet `ofstream`, en vidant le tampon associé
 - **Note** : Fichier automatiquement fermé à la destruction de l'objet
- ◆ `bool is_open ();`
 - Renvoie `true` si le fichier est ouvert et associé à un objet *stream buffer*
 - Ouverture : par construction d'un objet (instanciation) ou par appel à la méthode `open`
- ◆ `filebuf * rdbuf () const;`
 - Renvoie un pointeur sur l'objet `filebuf` associé au fichier



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe `ofstream` : Méthodes Publiques

■ Description détaillée des méthodes publiques (suite)...

◆ `void open (const char * fileName,
ios_base::openmode mode=ios_base::out);`

- Ouverture du fichier identifié par `fileName` (association à un *stream buffer* pour la gestion des opérations d'entrée / sortie)
- Le paramètre `mode` donne le mode d'ouverture...
 - `in` (`input`) : fichier ouvert en lecture (valeur par défaut pour les objets `ifstream`)
 - `out` (`output`) : fichier ouvert en écriture (valeur par défaut pour les objets `ofstream`)
 - `binary` (`binary`) : opérations de lecture ou d'écriture en binaire plutôt qu'en mode texte
 - `ate` (`at end`) : positionnement à la fin du fichier à l'ouverture en lecture ou en écriture
 - `app` (`append`) : ajout des données à la fin du fichier (à la suite du contenu déjà existant)
 - `trunc` (`truncate`) : effacement du contenu actuel du fichier et ouverture avec une longueur nulle



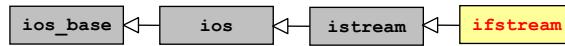
- Combinaison possible des indicateurs avec l'opérateur `|` (ou bit à bit)

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe `ifstream`

■ Généralités (`#include <fstream>`)



- ◆ La classe `ifstream` (*input stream*) permet de manipuler des fichiers en entrée
- ◆ Les instances de cette classe s'appuient sur un objet `filebuf` pour la gestion de leur *stream buffer* (bas niveau)
- ◆ Ouverture du fichier...
 - Par construction d'un objet (instanciation)
 - Par appel à la méthode `open`
- ◆ Méthodes publiques – Présentation rapide (cf. Classe `ofstream`)...
 - `open` : ouvre le fichier
 - `is_open` : vérifie si le fichier est ouvert
 - `close` : ferme le fichier
 - `rdbuf` : récupère le *stream buffer* associé au fichier
 - `operator =` : non présentée dans ce cours (C++11)
 - `swap` : non présentée dans ce cours (C++11)

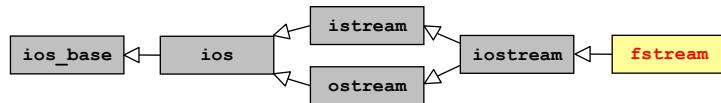


© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe `fstream`

■ Généralités (`#include <fstream>`)



- ◆ La classe `fstream` (*input / output file stream*) permet de manipuler des fichiers en entrée / sortie
- ◆ Les instances de cette classe s'appuient sur un objet `filebuf` pour la gestion de leur *stream buffer* (bas niveau)
- ◆ Ouverture du fichier...
 - Par construction d'un objet (instanciation)
 - Par appel à la méthode `open`
- ◆ Méthodes publiques – Présentation rapide (cf. Classe `ofstream`)...
 - `open` : ouvre le fichier
 - `is_open` : vérifie si le fichier est ouvert
 - `close` : ferme le fichier
 - `rdbuf` : récupère le *stream buffer* associé au fichier
 - `operator =` et `swap` : non présentée dans ce cours (C++11)



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Classe `fstream` : Exemple

```

// Copie d'un fichier source input.txt vers un fichier destination output.txt
// Comme on utilise operator >> à la lecture, cette copie subit les règles de formatage
// par défaut sur les flux
#include <fstream>

int main ()
{
    std::ifstream is;
    std::ofstream os;
    char c;

    is.open ( "input.txt" );
    os.open ( "output.txt" );

    // is.setf ( std::ios::skipws );
    // Attention : comportement par défaut
    while ( is >> c )
    {
        os.put ( c );
        // autre écriture os << c;
    }
    is.close ( );
    os.close ( );

    return 0;
}
  
```

...> cat input.txt
Ceci est un petit texte
pour illustrer les E/S en C++
utilisant une lecture formatée
sur le flux d'entrée (`operator >>`)

...> ./fstream
...> cat output.txt
Ceci est un petit texte pour illustrer les E/S en C++ utilisant une lecture formatée sur le flux d'entrée (`operator >>`)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Paramètres du main et Classes `fstream`

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

int main ( int nbArg, const char *listArg [ ] )
{
    if ( nbArg != 3 )
        std::cerr << "Appel : " << listArg [ 0 ] << " <IN> <OUT>" << std::endl;
        exit ( 1 );
    }

    std::ifstream ifs ( listArg [ 1 ] );
    if ( ifs.fail ( ) )
        std::cerr << "Erreur d'ouverture de IN <" << listArg [ 1 ] << ">" << std::endl;
        exit ( 1 );
    }

    std::ofstream ofs;
    ofs.open ( listArg [ 2 ], std::ofstream::out | std::ofstream::app );
    if ( !ofs.is_open ( ) )
        std::cerr << "Erreur d'ouverture de OUT <" << listArg [ 2 ] << ">" << std::endl;
        exit ( 1 );
    }

    char c;
    while ( ifs.get ( c ) )
        { ofs.put ( c );
    }

    ifs.close ( );
    ofs.close ( );
    return 0;
}
```

// variante au niveau de la boucle de copie
`#define MAX 5`
`char s [MAX];`
`while (ifs.get (s, MAX, EOF))`
`{`
 `ofs.write (s, MAX);`
`}`

© MM. Reproduction interdite sans l'autorisation de l'auteur.



Manipulation des streams : copie d'un fichier

```
// Copie simple d'un fichier source src vers un nouveau fichier dest
#include <fstream>
using namespace std;

int main ( )
{
    // Instanciation et donc ouverture des fichiers src et dest
    ifstream src ( "test.txt", ifstream::binary );
    ofstream dest ( "copieTest.txt", ofstream::binary );

    // Calcul de la taille du fichier src
    src.seekg ( 0, src.end ); // Placement à la fin du fichier
    long taille = src.tellg (); // Position courante dans le flux
    src.seekg ( 0 ); // Placement au début du fichier

    // Allocation mémoire d'un buffer capable de contenir tout le fichier src
    char * buffer = new char [ taille ];

    // Lecture du contenu du fichier src (une seule lecture)
    src.read ( buffer, taille );

    // Écriture du contenu du buffer pour construire le fichier dest (une seule écriture)
    dest.write ( buffer, taille );

    // Libération de la zone mémoire réservée pour le buffer
    delete [ ] buffer;

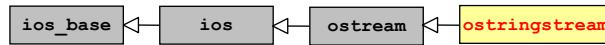
    dest.close ( );
    src.close ( );
    return 0;
}
```

© MM. Reproduction interdite sans l'autorisation de l'auteur.



Classe ostringstream

■ Généralités (`#include <sstream>`)



- ◆ La classe `ostringstream` (*output string stream*) permet de manipuler des chaînes de caractères
- ◆ Les instances de cette classe s'appuient sur un objet `stringbuf` qui gère la séquence de caractères (chaîne)
 - Accessible directement comme un objet `string` à l'aide de la méthode `str`
- ◆ Manipulable comme un *output stream* classique ⇒ opérations sur les *streams* valables pour les *stringstreams*
- ◆ Méthodes publiques – Présentation rapide...
 - `str` : récupère ou définit le contenu de l'objet *string stream*
 - `rdbuf` : récupère le *string buffer* associé à l'objet *string stream*
 - `operator =` : non présentée dans ce cours (C++11)
 - `swap` : non présentée dans ce cours (C++11)

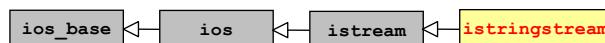


© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe istream

■ Généralités (`#include <sstream>`)



- ◆ La classe `istringstream` (*input string stream*) permet de manipuler des chaînes de caractères
- ◆ Les instances de cette classe s'appuient sur un objet `stringbuf` qui gère la séquence de caractères (chaîne)
 - Accessible directement comme un objet `string` à l'aide de la méthode `str`
- ◆ Manipulable comme un *input stream* classique ⇒ opérations sur les *streams* valables pour les *stringstreams*
- ◆ Méthodes publiques – Présentation rapide...
 - `str` : récupère ou définit le contenu de l'objet *string stream*
 - `rdbuf` : récupère le *string buffer* associé à l'objet *string stream*
 - `operator =` : non présentée dans ce cours (C++11)
 - `swap` : non présentée dans ce cours (C++11)



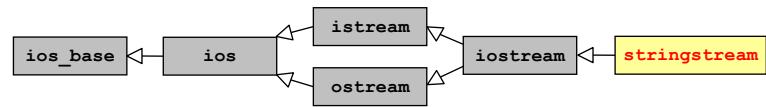
© MM - Reproduction interdite sans l'autorisation de l'auteur





Classe stringstream

■ Généralités (`#include <sstream>`)



- ◆ La classe **stringstream** (*input / output string stream*) permet de manipuler des chaînes de caractères
- ◆ Les instances de cette classe s'appuient sur un objet **stringbuf** qui gère la séquence de caractères (chaîne)
- ◆ Insertion ou extraction de caractères du *stream* à l'aide des opérations classiques sur les *streams* en *input* ou *output*
- ◆ Méthodes publiques – Présentation rapide...
 - **str** : récupère ou définit le contenu de l'objet *string stream*
 - **rdbuf** : récupère le *string buffer* associé à l'objet *string stream*
 - **operator =** : non présentée dans ce cours (C++11)
 - **swap** : non présentée dans ce cours (C++11)



© MM - Reproduction interdite sans l'autorisation de l'auteur



Classe istream : Exemple

```

// Manipulation d'un objet string stream comme un input stream
// Découpage d'une phrase en mots
#include <string> // std::string
#include <iostream> // std::cout
#include <sstream> // std::istringstream

int main ()
{
    std::istringstream iss;
    std::string texte = "Un petit texte de test";
    string mot;
    ...> ./istringstream
    Un
    petit
    texte
    de
    test
    while ( iss >> mot )
    {
        std::cout << mot << std::endl; Fin du découpage du texte : Un petit
        ...
        texte de test
    }
    std::cout << "Fin du découpage du texte : ";
    std::cout << iss.str () << std::endl;

    return 0;
}

```



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur



Programmation Orientée Objet – C++

PLAN du COURS

Naufrage

- ...
- Généricité
- Entrées / Sorties : `iostream`
- Standard Template Library
 - ◆ Généralités
 - ◆ Itérateur
 - ◆ Conteneur
 - ◆ Algorithme

Frisson

Divin ?

Bricolage

© MM - Reproduction interdite sans l'autorisation de l'auteur.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Standard Template Library

Réalisation d'un projet informatique en SSII

| | | | | |
|-----------------------------------|---------------------------------------|-----------------------------------|---|---|
| Comment le client l'a décrit | Comment le chef de projet l'a compris | Comment l'analyste l'a schématisé | Comment le programmeur l'a écrit | Comment le business consultant l'a décrit |
| Comment le projet a été documenté | Ce qui a été installé chez le client | Comment le client a été facturé | Comment le support technique est effectué | Ce dont le client avait réellement besoin |

© MM - Reproduction interdite sans l'autorisation de l'auteur.

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Standard Template Library (STL)

■ Généralités

- ◆ Bibliothèque C++, normalisée par l'ISO
- ◆ Nombreux composants efficaces et réutilisables
- ◆ Disponible dans divers environnements de développement

■ Contenu

- ◆ Un ensemble de classes conteneurs (*containers library*)
- ◆ Une abstraction des pointeurs : itérateurs (*iterators library*)
- ◆ Des algorithmes génériques : insertion, suppression, tri, recherche (*algorithms library*)...
- ◆ D'autres composantes...
 - Une classe **string**
 - Un mécanisme de bas-niveau pour l'allocation et la libération de la mémoire (*memory management library*)
 - Pour la concurrence (*concurrency support library*)...



© MM - Reproduction interdite sans l'autorisation de l'auteur

133



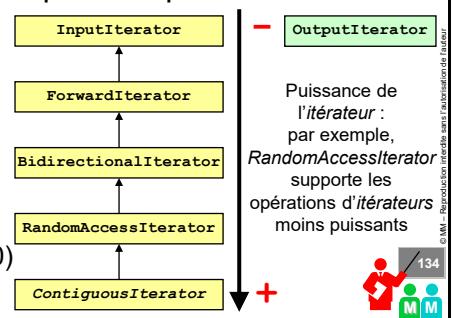
Itérateur

■ Caractéristiques

- ◆ Objet utilisé pour parcourir les éléments d'un conteneur
- ◆ Objet utilisé par les algorithmes (liaison algorithme / conteneur)
- ◆ S'appuie sur un ensemble d'opérateurs, avec au minimum...
 - L'incrémentation `++`
 - Le déréférencement `*`

■ 6 catégories d'*itérateurs* définies par les opérations supportées

- ◆ **InputIterator**
- ◆ **OutputIterator**
- ◆ **ForwardIterator**
- ◆ **BidirectionalIterator**
- ◆ **RandomAccessIterator**
- ◆ **ContiguousIterator (C++20)**



© MM - Reproduction interdite sans l'autorisation de l'auteur

134



Itérateur

- Détails...
 - ◆ **InputIterator** et **OutputIterator**
 - Les plus limités : opérations d'entrée et de sortie séquentielles dans un seul sens
 - ◆ **ForwardIterator**
 - Mêmes opérations qu'un *itérateur input* et s'il n'est pas constant, il possède les mêmes possibilités qu'un *itérateur output*
 - Déplacement du début vers la fin (dans un seul sens)
 - Tous les conteneurs supportent cet *itérateur*
 - ◆ **BidirectionalIterator**
 - Mêmes fonctionnalités qu'un *itérateur forward* mais avec des possibilités de déplacement dans les 2 sens (*forward* et *backward*)
 - ◆ **RandomAccessIterator**
 - Fonctionnalités similaires à un pointeur qui est d'ailleurs un *itérateur* de cette catégorie
 - Accès direct à un élément possible sans itérer à travers tous les éléments

© MM - Reproduction interdite sans l'autorisation de l'auteur.



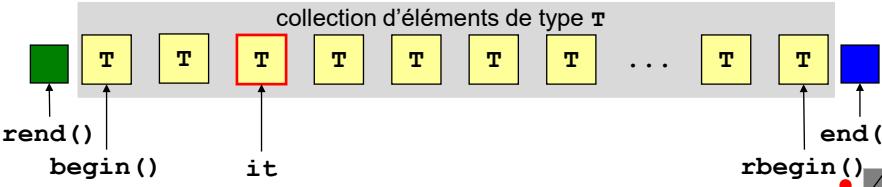

135



Itérateur

- Détails...
 - ◆ Un *itérateur* est associé à une séquence d'éléments
 - ◆ Un *itérateur* désigne une position dans cette séquence
 - ◆ Par convention, on délimite une séquence d'éléments par 2 *itérateurs* désignant respectivement...
 - La position du premier élément de la séquence
 - La position après le dernier élément (*past-the-end element*)
 - ◆ Utilisation des méthodes **begin()** (ou **rbegin()**) et **end()** (ou **rend()**) pour obtenir des valeurs d'*itérateurs* pour les conteneurs

collection d'éléments de type **T**



rend()

begin()

it

end()

rbegin()

© MM - Reproduction interdite sans l'autorisation de l'auteur.

© MM - Reproduction interdite sans l'autorisation de l'auteur.




136



Itérateur

| Catégorie d'itérateurs | | | | Propriétés | Expressions valides... |
|------------------------|---------------|---------|--------|--|--|
| Toutes les catégories | | | | Construction par copie, affectation et destruction | <code>x i2 (i1); i2 = i1;</code> |
| Random Access | Bidirectional | Forward | Input | Incrémentation ++ | <code>++i1 i1++</code> |
| | | | | Égalité == Différence != | <code>i1 == i2 i1 != i2</code> |
| | | | | Déréférancement comme <code>rvalue</code> | <code>*i1 i1->m</code> |
| | | | Output | Déréférancement comme <code>lvalue</code> | <code>*i1 = t *i1++ = t</code> |
| | | | | Construction par défaut | <code>X i1; X ()</code> |
| | | | | Multi-passes | |
| | | | | Décrémentation -- | <code>--i1 et i1-- *i1--</code> |
| | | | | Opérateurs arithmétiques + et - | <code>i1 + n et n + i1 i1 - n et n - i1 i1 - i2</code> |
| | | | | Comparaison <, >, <= et >= entre itérateurs | <code>i1 < i2 et i1 > i2 i1 <= i2 et i1 >= i2</code> |
| | | | | Opérateurs += et -= | <code>i1 += n i1 -= n</code> |
| | | | | Opérateur [] | <code>i1 [n]</code> |

↳ `x` est un type d'itérateur
 ↳ `i1` et `i2` sont 2 objets de type `x` (type d'itérateur)
 ↳ `t` est un objet du type pointé par l'itérateur
 ↳ `n` est une valeur entière

© MM - Reproduction interdite sans l'autorisation de l'auteur



Itérateur d'Insertion

Fonctionnement...

- ◆ Permet d'insérer un élément dans un conteneur...
- À une position donnée : `insert_iterator`
 - Si `ii` est un `insert_iterator` alors `*ii = x` réalise l'insertion de l'élément `x` dans le conteneur `c` à la position de l'itérateur
 - Cela correspond à `c.insert (p, x)`
- En début : `front_insert_iterator`
 - Il faut que le conteneur accepte l'insertion en tête (par exemple `vector <T>`, `list <T>`, `deque <T>`, `array <T>`)
- En fin : `back_insert_iterator`
 - Il faut que le conteneur accepte l'insertion en queue
- ◆ Peu d'opérations disponibles
- ◆ Les `itérateurs inserter`, `front_inserter` et `back_inserter` construisent automatiquement un `itérateur d'insertion` à partir de ses arguments (inférence de type)

© MM - Reproduction interdite sans l'autorisation de l'auteur


138



Itérateur d'Insertion : `insert_iterator`

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

int main ( void )
{ vector < int > v { 1, 2, 3, 4, 5 };
list < int > l { -1, -2, -3 };

copy ( v.begin ( ), v.end ( ), // peut être simplifié avec std:: inserter
       insert_iterator < list < int > > ( l, l.begin ( ) ) );
for ( list < int >::iterator it = l.begin ( ) ; it != l.end ( ) ; ++it )
{ cout << " " << *it;
}
cout << endl;
insert_iterator < list < int > > ii ( l, l.begin ( ) );
copy ( v.begin ( ), v.end ( ), inserter ( l, l.begin ( ) ) );
ii = 99; // * et ++ sont inutiles sur un insert_iterator (sans effet)
ii = 88;
for ( list < int >::iterator it = l.begin ( ) ; it != l.end ( ) ; ++it )
{ cout << " " << *it;
}
cout << endl;
return 0;
}
```

```
mmaranzana@if501-219-13:~/Cours/C++/ITERATOR$ ./mm
-1 -2 -3 1 2 3 4 5
99 88 -1 -2 -3 1 2 3 4 5 1 2 3 4 5
mmaranzana@if501-219-13:~/Cours/C++/ITERATOR$
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Itérateur d'Insertion : `insert_iterator`

```
#include <list>
#include <iostream>
#include <iterator>
#include <algorithm>
using namespace std;

int main ( void )
{
list < int > l { -1, -2, -3 };

front_insert_iterator < list < int > > fii ( l );
fii = -99; fii = -77; fii = -88; // insertion toujours en tête

cout << "l (après front...) =" ;
for ( list < int >::iterator it = l.begin ( ) ; it != l.end ( ) ; ++it )
{ cout << " " << *it;
}
cout << endl;

back_insert_iterator < list < int > > bii ( l );
bii = 7; bii = 9; bii = 8; // insertion toujours en queue
cout << "l (après back...) =" ;
for ( list < int >::iterator it = l.begin ( ) ; it != l.end ( ) ; ++it )
{ cout << " " << *it;
}
cout << endl;

return 0;
}
```

⇒ Attention : déréférencement inutile

front_insert_iterator < list < int > > fii (l);

fii = -99; fii = -77; fii = -88; // insertion toujours en tête

cout << "l (après front...) =" ;

for (list < int >::iterator it = l.begin () ; it != l.end () ; ++it)

{ cout << " " << *it;

}

cout << endl;

back_insert_iterator < list < int > > bii (l);

bii = 7; bii = 9; bii = 8; // insertion toujours en queue

cout << "l (après back...) =" ;

for (list < int >::iterator it = l.begin () ; it != l.end () ; ++it)

{ cout << " " << *it;

}

cout << endl;

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Itérateur Inversé

■ Fonctionnement...

- ◆ Les *itérateurs inversés* sont des sortes d'*itérateur*
- ◆ Permet de parcourir un conteneur en sens inverse
 - Inverse la direction de parcours dans laquelle un *itérateur bidirectionnel* ou à accès aléatoire parcourt une plage d'éléments

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

int main ( void )
{
    vector < int > v { 1, 2, 3, 4, 5 };

    vector < int >::reverse_iterator ri;
    for ( ri = v.rbegin ( ) ; ri != v.rend ( ) ; ++ri )
    {
        cout << " " << *ri;
    }
    cout << endl;

    return 0;
}
```



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur

■ Caractéristiques

- ◆ Objet support qui stocke une collection d'autres objets (ses éléments)
- ◆ Implémentation sous forme de modèles de classe (*template*)
- ◆ Gestion complète de l'espace mémoire alloué aux éléments
- ◆ Manipulation possible des éléments à partir des services (méthodes)

■ Différents conteneurs...

- ◆ Séquence (*sequence containers*)
 - `array` (C++11), `vector`, `deque`, `forward_list` (C++11), `list`
- ◆ Adaptateur (*container adaptors*) : modifie l'interface d'un conteneur
 - `stack` (LIFO - `deque`), `queue` (FIFO - `deque`)
 - `priority_queue` (`vector`)
- ◆ Associatif (*associative containers*)
 - `set`, `multiset`, `map`, `multimap`
- ◆ Associatif non ordonné (*unordered associative containers*) (C++11)
 - `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : Séquence

| En-tête – Interface | | <code><array></code> | <code><vector></code> | <code><deque></code> | <code><forward_list></code> | <code>list</code> |
|---------------------|-------------------------|----------------------------|-----------------------------|----------------------------|---|-------------------------|
| Fonctions membres | | C++11 | C++98 | C++98 | C++11 | C++98 |
| constructeur | constructeur | <i>implicit</i> | <code>vector</code> | <code>deque</code> | <code>forward_list</code> | <code>list</code> |
| | destructeur | <i>implicit</i> | <code>~vector</code> | <code>~deque</code> | <code>~forward_list</code> | <code>~list</code> |
| | <code>operator =</code> | <i>implicit</i> | <code>operator =</code> | <code>operator =</code> | <code>operator =</code> | <code>operator =</code> |
| | affectation | | <code>assign</code> | <code>assign</code> | <code>assign</code> | <code>assign</code> |
| itérateurs | <code>begin</code> | <code>begin</code> | <code>begin</code> | <code>begin</code> | <code>begin</code> <code>before_begin</code> | <code>begin</code> |
| | <code>end</code> | <code>end</code> | <code>end</code> | <code>end</code> | <code>end</code> | <code>end</code> |
| | <code>rbegin</code> | <code>rbegin</code> | <code>rbegin</code> | <code>rbegin</code> | | <code>rbegin</code> |
| | <code>rend</code> | <code>rend</code> | <code>rend</code> | <code>rend</code> | | <code>rend</code> |
| const itérateurs | <code>cbegin</code> | <code>cbegin</code> | <code>cbegin</code> | <code>cbegin</code> | <code>cbegin</code> <code>cbefore_begin</code> | <code>cbegin</code> |
| | <code>cend</code> | <code>cend</code> | <code>cend</code> | <code>cend</code> | <code>cend</code> | <code>cend</code> |
| | <code>crbegin</code> | <code>crbegin</code> | <code>crbegin</code> | <code>crbegin</code> | | <code>crbegin</code> |
| | <code>crend</code> | <code>crend</code> | <code>crend</code> | <code>crend</code> | | <code>crend</code> |



Conteneur : Séquence (suite)

| En-tête – Interface | | <code><array></code> | <code><vector></code> | <code><deque></code> | <code><forward_list></code> | <code>list</code> |
|---------------------|----------------------------|----------------------------|-----------------------------|----------------------------|-----------------------------------|-----------------------|
| Fonctions membres | | C++11 | C++98 | C++98 | C++11 | C++11 |
| capacité taille | <code>size</code> | <code>size</code> | <code>size</code> | <code>size</code> | | <code>size</code> |
| | <code>max_size</code> | <code>max_size</code> | <code>max_size</code> | <code>max_size</code> | <code>max_size</code> | <code>max_size</code> |
| | <code>empty</code> | <code>empty</code> | <code>empty</code> | <code>empty</code> | <code>empty</code> | <code>empty</code> |
| | <code>resize</code> | | <code>resize</code> | <code>resize</code> | <code>resize</code> | <code>resize</code> |
| | <code>shrink_to_fit</code> | | <code>shrink_to_fit</code> | <code>shrink_to_fit</code> | | |
| | <code>capacity</code> | | <code>capacity</code> | | | |
| | <code>reserve</code> | | <code>reserve</code> | | | |
| accès aux éléments | <code>front</code> | <code>front</code> | <code>front</code> | <code>front</code> | <code>front</code> | <code>front</code> |
| | <code>back</code> | <code>back</code> | <code>back</code> | <code>back</code> | | <code>back</code> |
| | <code>operator []</code> | <code>operator []</code> | <code>operator []</code> | <code>operator []</code> | | |
| | <code>at</code> | <code>at</code> | <code>at</code> | <code>at</code> | | |

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

■ Disponible depuis C++98
■ Disponible depuis C++11


144



Conteneur : Séquence (suite)

| En-tête – Interface | | <code><array></code> | <code><vector></code> | <code><deque></code> | <code><forward_list></code> | <code>list</code> |
|---------------------|----------------------------|----------------------------|-----------------------------|----------------------------|-----------------------------------|----------------------------|
| Fonctions membres | | C++11 | C++98 | C++98 | C++11 | C++98 |
| modification | <code>assign</code> | | <code>assign</code> | <code>assign</code> | <code>assign</code> | <code>assign</code> |
| | <code>emplace</code> | | <code>emplace</code> | <code>emplace</code> | <code>emplace_after</code> | <code>emplace</code> |
| | <code>insert</code> | | <code>insert</code> | <code>insert</code> | <code>insert_after</code> | <code>insert</code> |
| | <code>erase</code> | | <code>erase</code> | <code>erase</code> | <code>erase_after</code> | <code>erase</code> |
| | <code>emplace_back</code> | | <code>emplace_back</code> | <code>emplace_back</code> | | <code>emplace_back</code> |
| | <code>push_back</code> | | <code>push_back</code> | <code>push_back</code> | | <code>push_back</code> |
| | <code>pop_back</code> | | <code>pop_back</code> | <code>pop_back</code> | | <code>pop_back</code> |
| | <code>emplace_front</code> | | | <code>emplace_front</code> | <code>emplace_front</code> | <code>emplace_front</code> |
| | <code>push_front</code> | | | <code>push_front</code> | <code>push_front</code> | <code>push_front</code> |
| | <code>pop_front</code> | | | <code>pop_front</code> | <code>pop_front</code> | <code>pop_front</code> |
| <code>clear</code> | | | <code>clear</code> | <code>clear</code> | <code>clear</code> | <code>clear</code> |
| <code>swap</code> | | <code>swap</code> | <code>swap</code> | <code>swap</code> | <code>swap</code> | <code>swap</code> |

Disponible depuis C++98 Disponible depuis C++11



Conteneur : Séquence (fin)

| En-tête – Interface | | <code><array></code> | <code><vector></code> | <code><deque></code> | <code><forward_list></code> | <code>list</code> |
|---------------------|----------------------------|----------------------------|-----------------------------|----------------------------|-----------------------------------|----------------------------|
| Fonctions membres | | C++11 | C++98 | C++98 | C++11 | C++98 |
| opérations liste | <code>splice</code> | | | | <code>splice_after</code> | <code>splice</code> |
| | <code>remove</code> | | | | <code>remove</code> | <code>remove</code> |
| | <code>remove_if</code> | | | | <code>remove_if</code> | <code>remove_if</code> |
| | <code>unique</code> | | | | <code>unique</code> | <code>unique</code> |
| | <code>merge</code> | | | | <code>merge</code> | <code>merge</code> |
| | <code>sort</code> | | | | <code>sort</code> | <code>sort</code> |
| | <code>reverse</code> | | | | <code>reverse</code> | <code>reverse</code> |
| observers | <code>get_allocator</code> | | <code>get_allocator</code> | <code>get_allocator</code> | <code>get_allocator</code> | <code>get_allocator</code> |
| | <code>data</code> | <code>data</code> | <code>data</code> | | | |

Disponible depuis C++98 Disponible depuis C++11


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON
146



Conteneur : Associatif

| En-tête – Interface | | <set> | | <map> | |
|---------------------|--------------|------------|------------|------------|------------|
| Fonctions membres | | set | multiset | map | multimap |
| Fonctions membres | constructeur | set | multiset | map | multimap |
| | destructeur | ~set | ~multiset | ~map | ~multimap |
| | affectation | operator = | operator = | operator = | operator = |
| Itérateurs | begin | begin | begin | begin | begin |
| | end | end | end | end | end |
| | rbegin | rbegin | rbegin | rbegin | rbegin |
| | rend | rend | rend | rend | rend |
| const itérateurs | cbegin | cbegin | cbegin | cbegin | cbegin |
| | cend | cend | cend | cend | cend |
| | crbegin | crbegin | crbegin | crbegin | crbegin |
| | crend | crend | crend | crend | crend |

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

■ Disponible depuis C++98
■ Disponible depuis C++11


147



Conteneur : Associatif (suite)

| En-tête – Interface | | <set> | | <map> | |
|---------------------|--------------|--------------|--------------|--------------|--------------|
| Fonctions membres | | set | multiset | map | multimap |
| Fonctions membres | size | size | size | size | size |
| | max_size | max_size | max_size | max_size | max_size |
| | empty | empty | empty | empty | empty |
| Accès aux éléments | operator [] | | | operator [] | |
| | at | | | at | |
| Modification | emplace | emplace | emplace | emplace | emplace |
| | emplace_hint | emplace_hint | emplace_hint | emplace_hint | emplace_hint |
| | insert | insert | insert | insert | insert |
| | erase | erase | erase | erase | erase |
| | clear | clear | clear | clear | clear |
| | swap | swap | swap | swap | swap |

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

■ Disponible depuis C++98
■ Disponible depuis C++11


148

| En-tête – Interface | | <set> | | <map> | |
|---------------------|---------------|---------------|---------------|---------------|---------------|
| Fonctions membres | | set | multiset | map | multimap |
| opérations | count | count | count | count | count |
| | find | find | find | find | find |
| | equal_range | equal_range | equal_range | equal_range | equal_range |
| | lower_bound | lower_bound | lower_bound | lower_bound | lower_bound |
| | upper_bound | upper_bound | upper_bound | upper_bound | upper_bound |
| observers | get_allocator | get_allocator | get_allocator | get_allocator | get_allocator |
| | key_comp | key_comp | key_comp | key_comp | key_comp |
| | value_comp | value_comp | value_comp | value_comp | value_comp |
| | key_eq | | | | |
| | hash_function | | | | |

 INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

Disponible depuis C++98
Disponible depuis C++11


| En-tête – Interface | | <unordered_set> | | <unordered_map> | |
|---------------------|--------------|-----------------|---------------------|-----------------|---------------------|
| Fonctions membres | | unordered_set | unordered_multiset | unordered_map | unordered_multimap |
| constructeur | constructeur | unordered_set | unordered_multiset | unordered_map | unordered_multimap |
| | destructeur | ~unordered_set | ~unordered_multiset | ~unordered_map | ~unordered_multimap |
| | affectation | operator = | operator = | operator = | operator = |
| itérateurs | begin | begin | begin | begin | begin |
| | end | end | end | end | end |
| | rbegin | | | | |
| | rend | | | | |
| const itérateurs | cbegin | cbegin | cbegin | cbegin | cbegin |
| | cend | cend | cend | cend | cend |
| | crbegin | | | | |
| | crend | | | | |

 INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

Disponible depuis C++98
Disponible depuis C++11




Conteneur : Associatif Non Ordonné (suite)

| En-tête – Interface | | <unordered_set> | | <unordered_map> | |
|---------------------|--------------|-----------------|--------------------|-----------------|--------------------|
| Fonctions membres | | unordered_set | unordered_multiset | unordered_map | unordered_multimap |
| capacité taille | size | size | size | size | size |
| | max_size | max_size | max_size | max_size | max_size |
| | empty | empty | empty | empty | empty |
| | reserve | reserve | reserve | reserve | reserve |
| accès aux éléments | operator [] | | | operator [] | |
| | at | | | at | |
| modification | emplace | emplace | emplace | emplace | emplace |
| | emplace_hint | emplace_hint | emplace_hint | emplace_hint | emplace_hint |
| | insert | insert | insert | insert | insert |
| | erase | erase | erase | erase | erase |
| | clear | clear | clear | clear | clear |
| | swap | swap | swap | swap | swap |

Disponible depuis C++98 Disponible depuis C++11



Conteneur : Associatif Non Ordonné (suite)

| En-tête – Interface | | <unordered_set> | | <unordered_map> | |
|---------------------|---------------|-----------------|--------------------|-----------------|--------------------|
| Fonctions membres | | unordered_set | unordered_multiset | unordered_map | unordered_multimap |
| opérations | count | count | count | count | count |
| | find | find | find | find | find |
| | equal_range | equal_range | equal_range | equal_range | equal_range |
| | lower_bound | | | | |
| | upper_bound | | | | |
| observers | get_allocator | get_allocator | get_allocator | get_allocator | get_allocator |
| | key_comp | | | | |
| | value_comp | | | | |
| | key_eq | key_eq | key_eq | key_eq | key_eq |
| | hash_function | hash_function | hash_function | hash_function | hash_function |

Disponible depuis C++98 Disponible depuis C++11


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


152



Conteneur : Associatif Non Ordonné (fin)

| En-tête – Interface | <unordered_set> | | <unordered_map> | |
|--------------------------|------------------|--------------------|------------------|--------------------|
| Fonctions membres | unordered_set | unordered_multiset | unordered_map | unordered_multimap |
| <i>buckets</i> | bucket | bucket | bucket | bucket |
| | bucket_count | bucket_count | bucket_count | bucket_count |
| | bucket_size | bucket_size | bucket_size | bucket_size |
| | max_bucket_count | max_bucket_count | max_bucket_count | max_bucket_count |
| <i>politique de hash</i> | rehash | rehash | rehash | rehash |
| | load_factor | load_factor | load_factor | load_factor |
| | max_load_factor | max_load_factor | max_load_factor | max_load_factor |

■ Disponible depuis C++98
■ Disponible depuis C++11


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


153



Conteneur : vector

- Caractéristiques
 - ◆ Conteneur **séquentiel** qui encapsule les tableaux dynamiques
 - ◆ **Stockage contigu** ⇒ accès par *itérateur* ou pointeur ou `[]`
 - ◆ Augmentation / réduction **automatique** de la taille du tableau
 - Opération coûteuse
 - Méthodes spécifiques : `size`, `capacity` et `shrink_to_fit`
- Complexité des opérations...
 - ◆ Accès aléatoire (par index `[]`) : $O(1)$
 - ◆ Insertion ou suppression d'éléments à la fin : $O(1)$
 - ◆ Insertion ou suppression d'éléments au milieu : $O(n)$ (linéaire en fonction du nombre d'éléments jusqu'à la fin du vecteur)
 - ◆ Recherche d'un élément : trié $O(\log n)$ sinon $O(n)$

■ Disponible depuis C++98
■ Disponible depuis C++11


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


154



Conteneur : vector

■ Quelques méthodes (non exhaustif)...

◆ Constructeurs

- `vector <T> v1` : construit un vecteur `v1` vide (sans élément)
- `vector <T> v2 (n, val)` : construit un vecteur `v2` avec `n` éléments (copie de la valeur `val`)
- `vector <T> v3 (v2.begin(), v2.end())` : construit un vecteur `v3` en itérant au travers d'un autre vecteur `v2`
- `vector <T> v4 (v3)` : construit un vecteur `v4` par copie d'un vecteur `v3` déjà existant

◆ Accès aux éléments (en plus de `operator []`)

- `at(x)` : renvoie une **référence** sur l'élément du vecteur à la position `x`, en vérifiant les bornes (exception `std::out_of_range`)
- `front()` : renvoie une **référence** sur le **premier** élément du conteneur (le conteneur ne doit pas être vide)
- `back()` : renvoie une **référence** sur le **dernier** élément du conteneur (le conteneur ne doit pas être vide)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

155



Conteneur : vector

■ Quelques méthodes (suite)...

◆ Itérateurs

- `begin()` : renvoie un **itérateur** sur le **premier élément** du vecteur. Si le vecteur est vide, l'**itérateur** est égal à `end()`
- `end()` : renvoie un **itérateur** sur l'**élément suivant le dernier** élément du vecteur (*past-the-end element*)
- Une version inverse existe pour ces 2 **itérateurs** : `rbegin()` et `rend()`
- Une version **const** existe pour ces 4 **itérateurs** : `cbegin()` / `crbegin()` et `cend()` / `crend()`

◆ Capacité et taille

- `empty()` : vérifie si le vecteur est vide
- `size()` : renvoie le nombre d'éléments du vecteur, c'est-à-dire `std::distance(begin(), end())`
- `capacity()` : renvoie le nombre d'éléments pouvant être conservés dans le vecteur actuellement alloué



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur

156



Conteneur : vector

- Quelques méthodes (suite)...
 - ◆ Modification
 - `clear()` : supprime tous les éléments du vecteur sans toucher à la capacité du vecteur (`shrink_to_fit`)
 - `push_back(x)` : ajoute un nouvel élément `x` (passage par référence) à la fin du conteneur (ajustement de la taille, si nécessaire)
 - `pop_back()` : supprime le dernier élément du conteneur (le conteneur ne doit pas être vide)
 - `insert(pos, x)` : insère l'élément `x` avant la position `pos` dans le conteneur
- Quelques fonctions non-membres...
 - ◆ Comparaison de 2 vecteurs
 - `operator ==, operator !=, operator <, operator <=,`
`operator > et operator >=`
 - Compare le nombre d'éléments (`size()`) des 2 vecteurs
 - Compare, position après position, les éléments des 2 vecteurs

© MM - Reproduction interdite sans l'autorisation de l'auteur

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON




Conteneur : vector

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

static void afficher ( const vector < int > & v )
{ cout << "Taille = " << v.size ( ) << endl;
  cout << "Capacité = " << v.capacity ( ) << endl;
  vector < int >::const_iterator iv;
  cout << "Contenu = ";
  for ( iv = v.cbegin ( ) ; iv != v.cend ( ) ; ++iv )
  { cout << " " << *iv;
  }
  cout << endl;
}

int main ( void )
{ vector < int > v1;
  v1.push_back ( 99 );
  v1.push_back ( -55 );
  v1.insert ( v1.begin ( ), 22 );
  afficher ( v1 );
  vector < int > v2 ( 5, 3 );
  afficher ( v2 );
  vector < int >::iterator iv = v2.end ( );
  vector < int > v3 ( v2.begin ( ), iv - 2 );
  afficher ( v3 );
  return 0;
}
```

```
mmaranzana@if501-219-13:~/mm
Taille = 3
Capacité = 4
Contenu = 22 99 -55
Taille = 5
Capacité = 5
Contenu = 3 3 3 3 3
Taille = 3
Capacité = 3
Contenu = 3 3 3
mmaranzana@if501-219-13:~$
```

⇒ Parcours du vecteur **constant** `v` (passage par référence) à l'aide d'un **itérateur constant** (type `const_iterator`)

⇒ Utilisation des **itérateurs cbegin()** et **cend()**

⇒ Construction d'un vecteur vide `v1`

⇒ Ajout d'éléments dans le vecteur `v1` : sa taille et sa capacité s'adaptent automatiquement

⇒ Construction d'un vecteur `v2` de taille 5 et initialisé par la valeur 3 (5 éléments de valeur 3)

⇒ Construction d'un vecteur `v3` en parcourant un vecteur `v2` existant sur la plage `begin()` / `end() - 2` (utilisation d'un **itérateur iv**)

© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : list

■ Caractéristiques

- ◆ Liste doublement chaînée qui favorise l'insertion et la suppression
- ◆ Parcours de la liste possible dans les 2 sens au prix d'une augmentation de l'occupation mémoire
- ◆ Différence essentielle avec le conteneur `forward_list` : le chaînage
- ◆ Meilleure performance que `array`, `vector` et `deque` pour l'insertion, la suppression et le déplacement (très utile dans les tris)
- ◆ Inconvénient majeur : pas d'accès direct à un élément quelconque

■ Complexité des opérations...

- ◆ Accès rapide au début et à la fin de la liste : $O(1)$
- ◆ Accès aléatoire à une position : $O(n)$
- ◆ Insertion / suppression d'éléments à n'importe quelle position : $O(1)$
- ◆ Recherche d'un élément : $O(n)$



Conteneur : list

■ Quelques méthodes (non exhaustif)...

- ◆ Constructeurs
 - `list <T> 11` : construit une liste 11 vide (sans élément)
 - `list <T> 12 (n, val)` : construit une liste 12 avec n éléments (copie de la valeur val)
 - `list <T> 13 (12.begin(), 12.end())` : construit une liste 13 en itérant au travers d'une autre liste 12
 - `list <T> 14 (13)` : construit une liste 14 par copie d'une liste 13 déjà existante
- ◆ Accès aux éléments
 - `front()` : renvoie une référence sur le premier élément du conteneur (le conteneur ne doit pas être vide)
 - `back()` : renvoie une référence sur le dernier élément du conteneur (le conteneur ne doit pas être vide)



Conteneur : list

■ Quelques méthodes (suite)...

◆ Itérateurs

- `begin()` : renvoie un *itérateur* sur le **premier élément** de la liste. Si la liste est vide, il ne faut pas déréférencer l'*itérateur*
- `end()` : renvoie un *itérateur* (à ne pas déréférencer) sur l'**élément suivant le dernier** élément de la liste (*past-the-last element*)
 - Si la liste est vide, la valeur rendue est identique à `begin()`
- Une version inverse et une version `const` existent pour ces *itérateurs*

◆ Capacité et taille

- `empty()` : vérifie si la liste est vide
- `size()` : renvoie le nombre d'éléments de la liste
- `max_size()` : renvoie le nombre maximum possible d'éléments que pourra mémoriser une liste (en fonction des restrictions *système* ou d'implémentation de la librairie)



Conteneur : list

■ Quelques méthodes (suite)...

◆ Modification

- `clear()` : supprime tous les éléments de la liste (`size() = 0`)
- `erase(pos)` : supprime l'élément à la position `pos`
- `push_back(x)` : ajoute un nouvel élément `x` (passage par référence) à la fin du conteneur (`push_front(x)` au début du conteneur)
- `pop_back()` : supprime le dernier élément du conteneur (`pop_front()` le premier élément) (le conteneur ne doit pas être vide)
- `insert(pos, x)` : insère l'élément `x` avant la position `pos` dans le conteneur (plusieurs syntaxes possibles...)

◆ Opérations diverses...

- `splice(pos, 1)` : déplace tous les éléments de la liste `1` avant la position `pos` dans la liste courante
 - Déplacement ⇒ pas de construction / destruction d'éléments
 - Plusieurs syntaxes possibles : transfert de tous les éléments, d'un seul élément (celui pointé par l'*itérateur*) ou d'une séquence d'éléments



Conteneur : list

■ Quelques méthodes (suite)...

◆ Opérations diverses (suite)...

- **remove(x)** : supprime tous les éléments de la liste valant **x** (appel au destructeur)
- **remove_if(cond)** : supprime tous les éléments pour lesquels la condition **cond** renvoie **true** (appel au destructeur)
- **unique()** : supprime les doublons consécutifs (utile pour les listes triées)
- **merge(l)** : fusionne la liste **l** avec la liste courante en transférant les éléments de la liste **l** vers la liste courante et en respectant la position des éléments (les 2 listes doivent être triées)
- **sort()** : trie les éléments de la liste courante utilisant **operator <**
 - Possibilité d'utiliser sa propre fonction de comparaison
- **reverse()** : inverse l'ordre des éléments dans la liste courante

```
mmaranzana@if501-219-13:~$ ./mm
liste 1 = 5 3 9
liste 1 triée = 3 5 9
liste 2 (sans doublons ?) = 8 2 7 3 7 9
liste 2 triée = 2 3 7 7 8 9
Fusion = 2 3 3 5 7 7 8 9 9
Fusion (sans doublon) = 2 3 5 7 8 9
mmaranzana@if501-219-13:~$
```

Conteneur : list

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

ostream & operator << ( ostream &os, const list < int > & l )
{ for ( list < int >::const_iterator il = l.cbegin() ; il != l.cend() ; ++il )
    { os << " " << *il;
    }
    return os;
}

int main ()
{ list < int > list1;
list1.push_back ( 5 ); list1.push_back ( 9 );
list1.insert ( ++list1.begin(), 3 );
cout << "liste 1 = " << list1 << endl;
list1.sort ();
cout << "liste 1 triée = " << list1 << endl;
list < int > list2 = { 8, 2, 7, 3, 7, 9 };
list2.unique ();
cout << "liste 2 (sans doublons ?) = " << list2 << endl;
list2.sort ();
cout << "liste 2 triée = " << list2 << endl;
list1.merge ( list2 );
cout << "Fusion = " << list1 << endl;
list1.unique ();
cout << "Fusion (sans doublon) = " << list1 << endl;
return 0;
}
```

⇒ Parcours de la liste **constante** **l** (passage par référence) à l'aide d'un **itérateur constant** (type **const_iterator**)

⇒ Utilisation des **itérateurs cbegin()** et **cend()**

⇒ Construction d'une liste vide **list1**

⇒ Ajout d'éléments (5, 9 et 3 en position 2) dans la liste **list1**

⇒ Comme la liste **list2** n'est pas triée, les 7 ne sont pas côté à côté ⇒ **unique()** ne peut pas supprimer les doublons

⇒ Suppression des doublons dans la liste **list1** (fusion de **list1** et **list2**), les 2 listes étaient triées



Conteneur : deque

■ Caractéristiques

- ◆ Prononciation « *deck* », acronyme de ***double-ended queue***
- ◆ Conteneur séquentiel avec un ajustement possible aux 2 bouts
- ◆ Similaire au **vector** mais avec des performances accrues à l'insertion et à la suppression d'éléments en début de séquence
- ◆ **Attention** : éléments **pas forcément contigus** dans le conteneur
- ◆ Performance plus faible que le conteneur **list** ou **forward_list** en cas d'insertion / suppression d'éléments à des positions autres que le début ou la fin

■ Complexité des opérations...

- ◆ Accès aléatoire (par index `[]`) : $O(1)$
- ◆ Insertion ou suppression d'éléments au début ou à la fin : $O(1)$
- ◆ Insertion ou suppression d'éléments au milieu : $O(n)$ (linéaire en fonction du nombre d'éléments dans le conteneur)
- ◆ Recherche d'un élément : trié $O(\log n)$ sinon $O(n)$



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Conteneur : deque

■ Quelques méthodes (non exhaustif)...

- ◆ Constructeurs
 - `deque <T> d1` : construit un conteneur **deque** `d1` vide (sans élément)
 - `deque <T> d2 (n, val)` : construit un conteneur **deque** `d2` avec `n` éléments (copie de la valeur `val`)
 - `deque <T> d3 (d2.begin(), d2.end())` : construit un conteneur **deque** `d3` en itérant au travers d'un conteneur **deque** `d2`
 - `deque <T> d4 (d3)` : construit un conteneur **deque** `d4` par copie d'un conteneur **deque** `d3` déjà existant
- ◆ Accès aux éléments (en plus de **operator []**)
 - `at(x)` : renvoie une **référence** sur l'élément du conteneur à la position `x`, en vérifiant les bornes (exception `std::out_of_range`)
 - `front()` : renvoie une **référence** sur le **premier** élément du conteneur (le conteneur ne doit pas être vide)
 - `back()` : renvoie une **référence** sur le **dernier** élément du conteneur (le conteneur ne doit pas être vide)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Conteneur : deque

■ Quelques méthodes (suite)...

◆ Itérateurs

- `begin()` : renvoie un *itérateur* sur le *premier élément* du conteneur *deque*. Si la liste est vide, il ne faut pas déréférencer l'*itérateur*
- `end()` : renvoie un *itérateur* (à ne pas déréférencer) sur l'*élément suivant le dernier* élément du conteneur *deque* (*past-the-last element*)
 - Si la liste est vide, la valeur rendue est identique à `begin()`
- Une version inverse et une version `const` existent pour ces *itérateurs*

◆ Capacité et taille

- `empty()` : vérifie si le conteneur *deque* est vide
- `size()` : renvoie le nombre d'éléments du conteneur *deque*
- `max_size()` : renvoie le nombre maximum possible d'éléments que pourra mémoriser un conteneur *deque* (en fonction des restrictions *système* ou d'implémentation de la librairie)
- `shrink_to_fit()` : libère la mémoire inutilisée dans le conteneur



Conteneur : deque

■ Quelques méthodes (suite)...

◆ Modification

- `clear()` : supprime tous les éléments du conteneur (`size() == 0`)
- `erase(pos)` : supprime l'élément à la position `pos`
- `push_back(x)` : ajoute un nouvel élément `x` (passage par référence) à la fin du conteneur (`push_front(x)` au début du conteneur)
- `pop_back()` : supprime le dernier élément du conteneur (`pop_front()` le premier élément) (le conteneur ne doit pas être vide)
- `insert(pos, x)` : insère l'élément `x` avant la position `pos` dans le conteneur (plusieurs syntaxes possibles...)
- `emplace(pos, y)` : insère un nouvel élément dans le conteneur *deque* à la position `pos`, après l'avoir construit à partir de `y`
 - S'appuie sur la notion de référence sur *rvalue* (syntaxe `&&` en C++11)



Conteneur : deque

```

#include <iostream>
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

ostream & operator << ( ostream & os, const deque < int > & d )
{
    for ( deque < int >::const_iterator id = d.cbegin() ; id != d.cend() ; ++id )
        os << " " << *id;
    return os;
}

int main ( void )
{
    deque < int > d1;
    d1.push_back ( 3 );
    d1.push_front ( 1 );
    d1.push_front ( 1 );
    d1.insert ( d1.begin ( ) + 2, 2 );
    d1[0] = 0;
    cout << "deque = " << d1 << endl;
    d1.pop_front ( );
    cout << "deque (après pop) = " << d1 << endl;
    copy ( d1.begin ( ), d1.end ( ), ostream_iterator < int > ( cout, " " ) );
    cout << endl;
    return 0;
}

```

mmaranzana@if501-219-13:~\$./mm
deque = 0 1 2 3
deque (après pop) = 1 2 3
mmaranzana@if501-219-13:~\$

© MM - Reproduction interdite sans l'autorisation de l'auteur

⇒ Parcours de la **deque constante** `d` (passage par référence) à l'aide d'un **itérateur constant** (type `const_iterator`)

⇒ Construction d'une **deque vide** `d1`

⇒ Ajout de 2 éléments (1 et 1) en tête de la **deque d1**

⇒ Insertion d'un élément 2 en 3^{ème} position
⇒ entre le deuxième 1 et l'élément 3

⇒ Écrasement de l'élément à la position 0,
c'est-à-dire le premier élément 1
⇒ contenu de la **deque** : 0 1 2 3

⇒ Suppression de l'élément de tête

⇒ Utilisation de l'algorithme `copy` avec un **ostream_iterator** pour écrire séquentiellement les éléments sur le flux de sortie `cout` en utilisant un séparateur " "



Conteneur : stack

- Caractéristiques
 - ◆ Adaptateur de conteneur

```

template < typename T, typename Container = std::deque < T > >
class stack;

```

- ◆ Comportement d'une pile, plus précisément d'une pile FILO (**F**irst **I**n – **L**ast **O**ut)
- ◆ Encapsulation du conteneur sous-jacent (**deque**) en offrant qu'un petit sous ensemble de fonctionnalités...
 - Accès aux éléments : `top`
 - Capacité et taille : `empty` et `size`
 - Modification : `push`, `emplace`, `pop` et `swap`
 - Fonctions **non-membres**...
 - `operator ==`, `operator !=`, `operator <`, `operator <=`, `operator >` et `operator >=`

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

170

© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : stack

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

// Exemple d'implémentation de l'adaptateur de conteneur stack (existe dans la STL)
template < typename T, typename Conteneur = deque< T > >
class maStack
{
public :
    bool empty () const
    { return unePile.empty (); }

    typename Conteneur::size_type size () const
    { return unePile.size (); }

    T & top ()
    { return unePile.back (); }

    void push ( const T & x )
    { unePile.push_back ( x ); }

    void pop ()
    { unePile.pop_back (); }

protected :
    Conteneur unePile;
};

mmaranzana@if501-219-01:~$ ./mm
3 éléments dans la pile
Elément de tête : 0.123
3 éléments dans la pile
2 éléments dans la pile
Elément de tête : 3.14
mmaranzana@if501-219-01:~$
```

⇒ Définition d'un adaptateur de conteneur en s'appuyant sur un type quelconque **T** et le conteneur **deque**

⇒ Définition des services de l'adaptateur de conteneur

// Utilisation de l'adaptateur de conteneur **maStack**

```
int main ( void )
{ maStack < double > p;
p.push ( 1.5 );
p.push ( 3.14 );
p.push ( 0.123 );

cout << p.size () << " éléments dans la pile" << endl;
cout << "Elément de tête : " << p.top () << endl;
// p.top() laisse l'élément sur le sommet de la pile
cout << p.size () << " éléments dans la pile" << endl;
p.pop ();
// p.pop() supprime l'élément sur le sommet de la pile
cout << p.size () << " éléments dans la pile" << endl;
cout << "Elément de tête : " << p.top () << endl;
return 0;
}
```

⇒ Utilisation du nouveau type **Conteneur** pour définir la caractéristique de la classe **maStack**

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Conteneur : queue

- Caractéristiques
 - ◆ Adaptateur de conteneur

```
template < typename T, typename Container = std::deque < T > >
class queue;
```

- ◆ Comportement d'une file d'attente, plus précisément d'une file FIFO (First In – First Out)
- ◆ Encapsulation du conteneur sous-jacent (**deque**) en offrant qu'un petit sous ensemble de fonctionnalités...
 - Accès aux éléments : **front** et **back**
 - Capacité et taille : **empty** et **size**
 - Modification : **push**, **emplace**, **pop** et **swap**
 - Fonctions **non-membres**...
 - **operator ==**, **operator !=**, **operator <**, **operator <=**, **operator >** et **operator >=**

© MM - Reproduction interdite sans l'autorisation de l'auteur.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

172



Conteneur : priority_queue

■ Caractéristiques

- ◆ Adaptateur de conteneur pour définir une file d'attente à priorité

```
template <
    typename T,
    typename Container = std::vector < T >
    typename Compare = std::less < typename Container::value_type >
> class priority_queue;
```

- ◆ Possibilité de redéfinir la fonction de comparaison pour changer les priorités dans la file d'attente
- ◆ Encapsulation du conteneur sous-jacent (*vector*) en offrant qu'un petit sous ensemble de fonctionnalités...
 - Accès aux éléments : `top`
 - Capacité et taille : `empty` et `size`
 - Modification : `push`, `emplace`, `pop` et `swap`



© MMI - Reproduction interdite sans l'autorisation de l'auteur

Conteneur : set

■ Caractéristiques

```
template <
    typename Key,
    typename Compare = std::less < Key >
    typename Allocator = std::allocator < Key > > class set;
```

- ◆ Conteneur **associatif** qui manipule un ensemble de clés uniques (pas de doublons) et triés
- ◆ Utilisation d'une fonction générique **Compare** pour effectuer le tri
- ◆ Implémentation fréquente sous forme d'arbre rouge et noir pour garantir la complexité des opérations (complexité logarithmique)

■ Complexité des opérations...

- ◆ Accès aléatoire (par index []) : $O(1)$
- ◆ Insertion ou suppression d'éléments (peu importe la position dans le *set*) : $O(\log n)$
- ◆ Recherche d'un élément dans le *set* : $O(\log n)$



© MMI - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : set

■ Quelques méthodes (non exhaustif)...

◆ Constructeurs

- `set <T> s1` : construit un `set` `s1` vide (sans élément)
- `set <T> s2 (s1.begin(), s1.end())` : construit un `set` `s2` en itérant au travers d'un autre `set` `s1` (les *itérateurs* peuvent être différents de `begin` / `end`)
- `set <T> s3 (s2)` : construit un `set` `s3` par copie d'un `set` `s2` existant
- `set <T> s4 { e1, e2, e3 }` : construit un `set` `s4` à partir d'une liste d'éléments `ei` de type `T`

◆ Recherche

- `count(k)` : renvoie le nombre d'éléments correspondant à la clé `k`
- `find(k)` : renvoie un *itérateur* sur l'élément correspondant à la clé `k`, s'il existe. Sinon, renvoie `end()` (*past-the-end-element*)
- `lower_bound(k)` : renvoie un *itérateur* sur le premier élément qui n'est pas plus petit que la clé `k`, s'il existe. Sinon, renvoie `end()` (*past-the-end-element*)

- `upper_bound(k)` : idem, mais plus grand que `k`



© MM - Reproduction interdite sans l'autorisation de l'auteur

Conteneur : set

■ Quelques méthodes (suite)...

◆ Itérateurs

- `begin()` : renvoie un *itérateur* sur le **premier élément** du `set`. Si le `set` est vide, l'*itérateur* est égal à `end()`
- `end()` : renvoie un *itérateur* sur l'**élément suivant le dernier** élément du `set` (*past-the-last element*)
- Une version inverse existe pour ces 2 *itérateurs* : `rbegin()` et `rend()`
- Une version `const` existe pour ces 4 *itérateurs* : `cbegin()` / `crbegin()` et `cend()` / `crend()`

◆ Capacité et taille

- `empty()` : vérifie si le `set` est vide
- `size()` : renvoie le nombre d'éléments du `set`, c'est-à-dire `std::distance(begin(), end())`
- `max_size()` : renvoie le nombre maximum possible d'éléments que pourra mémoriser un conteneur `set` (en fonction des restrictions *système* ou d'implémentation de la librairie)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : set

■ Quelques méthodes (suite)...

◆ Modification

- `clear()` : supprime tous les éléments du `set`
- `insert(x)` : insère l'élément `x` dans le `set`, s'il n'existe pas
- `emplace(x)` : insère un nouvel élément dans le `set`, si il n'existe pas, après l'avoir construit à partir de `x`
 - S'appuie sur la notion de référence sur `rvalue` (syntaxe `&&` en C++11)
- `erase()` : supprime un élément du `set` (le `set` ne doit pas être vide)
 - `erase(pos)` : élément à `pos`
 - `erase(it1, it2)` : éléments définis par la séquence `it1` à `it2`
 - `erase(key)` : élément défini par `key` (s'il existe)
- `swap(s)` : permute les contenus du `set s` avec le `set` courant
- `extract()` : C++17 – non présenté
- `merge()` : C++17 – non présenté



Conteneur : set

```
ostream & operator << ( ostream & os, const set < string > & s )
{ os << "{ " << *s.begin ( );
  for ( set <string>::iterator it = ++( s.begin ( ) ) ; it != s.end ( ) ; ++it )
  { os << ", " << *it;
  os << " }" << endl;
  return os;
}

int main ( void )
{ set < string > e1; // (1) Construction par défaut
  e1.insert ( "table" ); e1.insert ( "chaise" ); e1.insert ( "tabouret" );
  e1.insert ( "chaise" ); e1.insert ( "tabouret" );
  cout << "e1 = " << e1 << endl;

  set < string > e2 ( e1.find("table"), e1.end()); // (2) Construction en itérant dans un autre set
  cout << "e2 = " << e2 << endl;

  set < string > e3 ( e1 ); // (3) Construction par copie d'un set existant
  e3.insert( "buffet" );
  cout << "e1 = " << e1;
  cout << "e3 = " << e3 << endl;

  set < string > e4 ( move ( e1 ) ); // (4) Construction par déplacement d'objets d'un autre set
  if ( e1.empty ( ) )
  { cout << "e1 = vide" << endl;
  cout << "e4 = " << e4 << endl;

  set < string > e5 { "un", "deux", "trois", "quatre", "cinq" }; // (5) Initialisation à partir d'une liste
  cout << "e5 = " << e5 << endl;
  return 0;
}
```

⇒ Les doublons ne sont pas autorisés à la création d'un set

⇒ Choix du point de départ de l'itération

⇒ Construction de e3 à partir de e1 puis ajout de l'élément "buffet" (l'ensemble e3 doit rester trié)

⇒ Construction de e4 par déplacement des éléments de e1 (à la fin de la construction de e4, e1 est vide)

⇒ Construction de e5 à partir d'une liste d'initialisation (e5 est trié après la construction)



Conteneur : multiset

■ Caractéristiques

```
template <
    typename Key,
    typename Compare = std::less < Key >
    typename Allocator = std::allocator < Key > > class multiset;
```

- ◆ Conteneur **associatif** qui manipule un ensemble de clés triés
- ◆ Contrairement au **set**, les clés avec des valeurs équivalentes sont autorisées (doublons)
 - Pour le reste, mêmes possibilités qu'un conteneur **set**
- ◆ Utilisation d'une fonction générique **Compare** pour effectuer le tri

■ Complexité des opérations...

- ◆ Accès aléatoire (par index []) : $O(1)$
- ◆ Insertion ou suppression d'éléments (peu importe la position dans le **set**) : $O(\log n)$
- ◆ Recherche d'un élément dans le **set** : $O(\log n)$



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : map

■ Caractéristiques

```
template <
    typename Key,
    typename T,
    typename Compare = std::less < Key >
    typename Allocator = std::allocator <std::pair<const Key, T>>
> class map;
```

- ◆ Conteneur **associatif trié** qui contient des paires (**clé, valeur**) avec des clés uniques
- ◆ Utilisation d'une fonction générique **Compare** pour effectuer le tri
- ◆ Implémentation fréquente sous forme d'arbre rouge et noir pour garantir la complexité des opérations (complexité logarithmique)

■ Complexité des opérations...

- ◆ Accès aléatoire (par index []) : $O(1)$
- ◆ Insertion ou suppression (peu importe la position) : $O(\log n)$
- ◆ Recherche d'un élément dans la **map** : $O(\log n)$



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : map

■ Gestion des paires (`#include <utility>`)

```
template < typename T1,typename T2 > struct pair;
```

- ◆ `std::pair` est une structure *template* qui permet de stocker 2 objets hétérogènes (de type `T1` et `T2`) en une seule composante
- ◆ Construction d'une paire avec...

```
template < typename T1,typename T2 >
constexpr std::pair < V1, V2 > make_pair ( T1 && t, T2 && u );
(syntaxe C++14 - && = value reference)
```

- ◆ Manipulation de la paire à l'aide des attributs...
- `first` : le premier élément de la paire (la clé dans une *map* / *multimap*)
- `second` : le second élément de la paire (la valeur dans une *map* ou *multimap*)



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Conteneur : map

■ Quelques méthodes (non exhaustif)...

- ◆ Constructeurs
 - `map <T, int> m1` : construit une *map* `m1` vide (sans élément)
 - `map <T, int> m2 (m1.begin(), m1.end())` : construit une *map* `m2` en itérant au travers d'une autre *map* `m1` (les *itérateurs* peuvent être différents de `begin` / `end`)
 - `map <T, int> m3 (m2)` : construit une *map* `m3` par copie d'une *map* `m2` existant
 - `map <T, int> m4 { {c1, x1}, {c2, x2}, {c3, x3} }` : construit une *map* `m4` à partir d'une liste de paire `{ci, xi}` (avec `ci` de type `T` et `xi` de type `int`)
- ◆ Accès à l'élément
 - `operator []` :
 - Renvoie une référence sur sa valeur mappée, si `k` (paramètre) correspond à la clé d'un élément dans le conteneur
 - Sinon, insère un nouvel élément nul avec cette clé
 - `at(k)` : renvoie une référence sur l'élément correspondant à la clé `k`, s'il existe. Sinon, lance l'exception `out_of_range`



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Conteneur : map

■ Quelques méthodes (non exhaustif)...

- ◆ Recherche

- `count(k)` : renvoie le nombre d'éléments correspondant à la clé `k`, soit `0` ou `1` puisque ce conteneur ne tolère pas les doublons
- `find(k)` : renvoie un *itérateur* sur l'élément correspondant à la clé `k`, s'il existe. Sinon, renvoie `end()` (*past-the-end-element*)
- `equal_range` : renvoie une paire d'*itérateurs*, définissant une plage incluant tous les éléments dont la valeur est égale à la clé `k` (paramètre), soit **au plus un** élément (absence de doublons)
- `lower_bound(k)` : renvoie un *itérateur* sur le premier élément qui n'est pas plus petit que la clé `k`, s'il existe. Sinon, renvoie `end()` (*past-the-end-element*)
- `upper_bound(k)` : idem, mais plus grand que `k`



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : map

■ Quelques méthodes (suite)...

- ◆ *Itérateurs*

- `begin()` : renvoie un *itérateur* sur le **premier élément** de la *map*. Si la *map* est vide, l'*itérateur* est égal à `end()`
- `end()` : renvoie un *itérateur* sur l'**élément suivant le dernier** élément de la *map* (*past-the-last element*)
- Une version inverse existe pour ces 2 *itérateurs* : `rbegin()` et `rend()`
- Une version `const` existe pour ces 4 *itérateurs* : `cbegin()` / `crbegin()` et `cend()` / `crend()`

- ◆ Capacité et taille

- `empty()` : vérifie si la *map* est vide (c'est-à-dire `begin() == end()`)
- `size()` : renvoie le nombre d'éléments dans le conteneur *map*, c'est-à-dire `std::distance(begin(), end())`
- `max_size()` : renvoie le nombre maximum possible d'éléments que pourra mémoriser un conteneur *map* (en fonction des restrictions *système* ou d'implémentation de la librairie)



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur



Conteneur : map

■ Quelques méthodes (suite)...

◆ Modification

- `clear()` : supprime tous les éléments de la *map*
- `insert(pair<key,T>(c,x))` : insère l'élément (paire *(c,x)*) dans la *map*, si le conteneur ne contient pas déjà un élément avec une clé *c* équivalente (de nombreuses syntaxes existent...)
 - Il faut utiliser `operator[]`, si on souhaite écraser un élément déjà présent dans la *map*
- `insert_or_assign()` : C++17 – non présenté
- `emplace(args)` : insère un nouvel élément dans la *map*, après l'avoir construit à partir de *args*, s'il n'existe pas dans le conteneur *map*, un élément avec la même clé
 - S'appuie sur la notion de référence sur *rvalue* (syntaxe `&&` en C++11) pour définir *args*
- `emplace_hint()` : C++11 – non présenté
- `try_emplace` : C++17 – non présenté



Conteneur : map

■ Quelques méthodes (suite)...

◆ Modification

- `erase()` : supprime un ou plusieurs éléments de la *map* (la *map* ne doit pas être vide)
 - `erase(pos)` : élément à *pos*, *itérateur* valide et déréférençable
 - `erase(it1, it2)` : éléments définis par la séquence (valide) *it1* à *it2*
 - `erase(key)` : élément défini par *key* (s'il existe)
- `swap(m)` : permute les contenus de la *map m* avec la *map* qui invoque la méthode
- `extract()` : C++17 – non présenté
- `merge()` : C++17 – non présenté

Conteneur : map

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

int main ( void )
{
    map < char, int > m1;
    map < char, int >::iterator ilow;
    map < char, int >::iterator iup; // Ajout de paire ( ci, xi ) dans la map m1

    m1 [ 'a' ] = 11;
    m1 [ 'b' ] = 22;
    m1 [ 'c' ] = 33; // Mise à jour de la valeur (11) associée à la clé 'a'
    m1 [ 'd' ] = 44; // ⇒ la nouvelle valeur est 99
    m1 [ 'e' ] = 55;
    m1 [ 'a' ] = 99; // Mise en place des itérateurs ilow et iup dans la map m1

    ilow = m1.lower_bound ( 'b' ); // ilow pointe l'élément de clé 'b'
    iup = m1.upper_bound ( 'd' ); // iup pointe l'élément de clé 'e' (et non 'd')
    m1.erase ( ilow, iup ); // on efface les éléments dans la plage [ ilow, iup ]

    // Affichage du conteneur map m1
    for ( map < char, int >::iterator it = m1.begin ( ) ; it != m1.end ( ) ; ++it )
    {
        cout << " ( " << it->first << " -> " << it->second << " ) " << endl;
    }
    cout << " Taille maximale de la map 'm1' = " << m1.max_size ( ) << endl;
    return 0;
}
```

mmaranzana@if501-219-13:~/Cours/C++/STL/MAP\$./mm
(a -> 99)
(e -> 55)
Taille maximale de la map 'm1' = 461168601842738790
mmaranzana@if501-219-13:~/Cours/C++/STL/MAP\$

© MM - Reproduction interdite sans l'autorisation de l'auteur

Conteneur : map (exemple : insert)

```
#include <map>
#include <iostream>
#include <string>
#include <vector>
#include <utility> // make_pair ( )
using namespace std;

template < typename M >
void affiche ( const string & msg, const M & m, bool rc = true )
{
    cout << msg;
    cout << m.size ( ) << " élément(s) = ";
    // syntaxe C++11
    // for ( const auto & p : m )
    // {
    //     cout << " ( " << p.first << " , " << p.second << " ) ";
    // }

    for ( typename M::const_iterator it = m.cbegin ( ) ; it != m.cend ( ) ; ++it )
    {
        cout << " ( " << it->first << " , " << it->second << " ) ";
    }
    cout << endl;

    if ( rc )
    { cout << endl;
    }
}
```

Procédure d'affichage générique (type M)
⇒ affichage possible d'une map ou d'un vector contenant des paires (cf. le main)

Boucle d'affichage des paires du conteneur selon une syntaxe C++11 avec une utilisation de **auto** (non présentée dans le cours)

Boucle d'affichage des paires du conteneur en utilisant explicitement un **itérateur constant** (la référence **m** est **const**)

© MM - Reproduction interdite sans l'autorisation de l'auteur

Conteneur : map (exemple : insert)

```

int main ( void )
{
    map < int, int > m1;
    m1.insert ( { 1, 10 } ); // appel de insert( const value_type & )
    m1.insert ( make_pair ( 2, 20 ) ); // appel de insert ( ValTy && )

    affiche ( "Contenu de 'm1' sous la forme ( clé, valeur )\n", m1 );

    // tentative d'insertion d'un élément existant
    pair <map < int, int >::iterator, bool> crdu = m1.insert ( make_pair ( 1, 99 ) );
    if ( ! crdu.second )
    {
        pair < int, int > p = *crdu.first;
        cout << "">>>> Echec à l'insertion : l'élément avec la clé = 1 existe déjà" << endl
            << "    Voici l'élément = (" << p.first << ", " << p.second << ")" << endl;
        cout << endl;
    }
    else
    {
        affiche ( "Contenu de 'm1' sous la forme ( clé, valeur )\n", m1 );
    }

    // insertion d'un élément (une paire), avec une suggestion de position (à la fin)
    m1.insert ( m1.end(), make_pair ( 3, 30 ) );
    affiche ( "Contenu de 'm1' sous la forme ( clé, valeur )\n", m1 );
    // ...
}

```

© MM - Reproduction interdite sans l'autorisation de l'auteur

⇒ Tentative d'insertion d'un élément avec une clé existante (clé = 1)
⇒ Échec à l'insertion (cf. `operator []`)

⇒ crdu est une paire : un itérateur sur l'élément inséré (ou celui correspondant à la clé, s'il existe déjà) et un booléen (état de l'insertion)

⇒ Suggestion du point d'insertion de la nouvelle paire

Conteneur : map (exemple : insert)

```

// Insertion dans la map m2 à partir d'un vecteur v de paires sans ordre
map < int, int > m2;
vector < pair < int, int > > v;
v.push_back ( make_pair ( 44, 444 ) );
v.push_back ( make_pair ( 33, 333 ) );
v.push_back ( make_pair ( 55, 555 ) );
v.push_back ( { 22, 222 } ); // syntaxe possible en C++11

affiche ( "Insertion du vecteur 'v' dans la map 'm2'\n", v, false );
m2.insert ( v.begin ( ), v.end ( ) - 1 );
affiche ( "Contenu de 'm2' sous la forme ( clé, valeur )\n", m2 );

map < int, string > m3;
pair < int, string > e1 ( 475, "Pierre" );
pair < int, string > e2 ( 510, "Louis" );

m3.insert ( move ( e1 ) ); // déplacement d'un élément e1 dans la map m3
affiche ( "Contenu de 'm3' après l'insertion par déplacement de e1\n", m3, false );
cout << "">>>> e1.second = " << e1.second << endl;
// insertion par déplacement de e2 avec une suggestion (fausse) de position dans m3
m3.insert ( m3.begin ( ), move ( e2 ) );
affiche ( "Contenu de 'm3' après l'insertion par déplacement de e2\n", m3 );

map < int, int > m4;
// insertion à partir d'une liste d'initialisation de paires
m4.insert ( { { -1, 44 }, { -2, 22 }, { -3, 33 }, { -1, 11 }, { -5, 55 } } );
affiche ( "Contenu de 'm4' après insertion à partir d'une liste d'initialisation\n", m4 );
return 0;
}

```

© MM - Reproduction interdite sans l'autorisation de l'auteur

⇒ Construction de la map m2 à partir des paires de v, sauf la dernière (qui aurait dû être la première de la map)

⇒ Construction par déplacement d'une paire dans la map
Attention : la string de e1 est vide après le déplacement

⇒ Ce n'est qu'une suggestion...

⇒ { -1, 11 } n'est pas inséré



Conteneur : multimap

■ Caractéristiques

```
template <
    typename Key, typename T,
    typename Compare = std::less < Key >
    typename Allocator = std::allocator<std::pair<const Key, T>>
> class multimap;
```

- ◆ Conteneur **associatif trié** qui contient des paires (**clé, valeur**)
- ◆ Clés multiples possibles : l'ordre des paires (**clé, valeur**) dont les clés sont égales est défini par l'ordre d'insertion et ne change pas (depuis C++11)
- ◆ Utilisation d'une fonction générique **Compare** pour effectuer le tri
- ◆ **operator []** n'existe pas pour une *multimap*

■ Complexité des opérations...

- ◆ Accès aléatoire (par index []) : $O(1)$
- ◆ Insertion ou suppression (peu importe la position) : $O(\log n)$
- ◆ Recherche d'un élément dans la *map* : $O(\log n)$



© MM - Reproduction interdite sans l'autorisation de l'auteur



Algorithme

■ Caractéristiques

- ◆ Collection de fonctions spécialement définies pour être utilisées sur des séquences d'éléments
- ◆ Séquence d'éléments : accessible via un *itérateur* (ou un pointeur)
 - Notamment les tableaux et certains conteneurs (dépend de l'algorithme)

■ Classes d'algorithmes (#include <algorithm>)

- ◆ Algorithmes sur des séquences sans modification
 - **all_of** (C++11), **any_of** (C++11) et **none_of** (C++11) : vérifie si un prédicat est vrai pour tous, n'importe lequel ou aucun élément d'une séquence (une plage)
 - **for_each** : exécute une fonction sur chaque élément d'une séquence
 - **for_each_n** (C++17) : exécute une fonction sur les **n** premiers éléments d'une séquence
 - **count** et **count_if** : renvoie le nombre d'éléments d'une séquence respectant des critères spécifiques
 - **find**, **find_if** et **find_if_not** (C++11) : trouve le premier élément d'une séquence respectant des critères spécifiques



© MM - Reproduction interdite sans l'autorisation de l'auteur



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

◆ Algorithmes sur des séquences sans modification (suite)

- `find_end` : trouve la dernière sous-séquence d'éléments dans une séquence d'éléments
- `find_first_of` : trouve le premier élément concordant dans une séquence
- `adjacent_find` : trouve les deux premiers éléments adjacents qui sont égaux (ou qui respectent à un prédictat donné)
- `equal` : détermine si 2 séquences d'éléments sont identiques
- `mismatch` : détermine la première position où 2 séquences d'éléments diffèrent
- `search` : recherche d'une séquence d'éléments dans une séquence
- `search_n` : recherche un nombre de copies consécutives d'un élément dans une séquence d'éléments



Algorithme : find

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

int main ( void )
{
    int n1 = 3;
    int n2 = 5;
    vector < int > v { 0, 1, 2, 3, 4 };

    vector < int >::iterator resultat1 = find ( begin ( v ), end ( v ), n1 );
    vector < int >::iterator resultat2 = find ( begin ( v ), end ( v ), n2 );

    if ( resultat1 != end ( v ) )
        cout << "v contient l'entier : " << n1 << endl;
    else
        cout << "v ne contient pas l'entier : " << n1 << endl;

    if ( resultat2 != end ( v ) )
        cout << "v contient l'entier : " << n2 << endl;
    else
        cout << "v ne contient pas l'entier : " << n2 << endl;
}

return 0;
}
```

⇒ Utilisation d'un conteneur `vector` d'entiers

⇒ Utilisation d'`itérateurs` (`resultati, begin et end`) sur la séquence définie par le conteneur `v`, `vector` d'entiers

⇒ Délimitation de la plage de recherche (`begin / end`)

⇒ Utilisation de l'algorithme `find`



Algorithme

```
...  
int main ()  
{ // Transformation en majuscules d'une chaîne de caractères  
    string s ( "bonjour" );  
    transform ( s.begin ( ), s.end ( ), s.begin ( ),  
               ( int (*) ( int ) )toupper );  
    cout << s << endl;  
    return 0;  
}
```

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Algorithmes sur des séquences avec modification
 - `copy` et `copy_if` (C++11) : copie une séquence d'éléments (une plage) vers un nouvel emplacement
 - `copy_n` (C++11) : copie un nombre donné d'éléments vers un nouvel emplacement (valeur en paramètre)
 - `copy_backward` : copie une séquence d'éléments dans un ordre inverse
 - `move` (C++11) : déplace une séquence d'éléments vers un nouvel emplacement
 - `move_backward` (C++11) : déplace une séquence d'éléments vers un nouvel emplacement dans un ordre inverse
 - `fill` : attribue une valeur spécifique à une séquence d'éléments
 - `fill_n` : attribue une valeur spécifique à un nombre donné d'éléments
 - `transform` : applique une fonction à une séquence d'éléments

© MM - Reproduction interdite sans l'autorisation de l'auteur.



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

```
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHMS$ ./mm  
BONJOUR  
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHMS$
```


195



Algorithme : transform

```
mmaranzana@if501-219-13:~$ ./mm  
v1 = 11 22 33  
v2 (après x 2) = 22 44 66  
v1 (après plus < int >) = 33 66 99  
mmaranzana@if501-219-13:~$
```

```
#include <iostream>  
#include <algorithm>  
#include <vector>  
#include <functional> // std::plus  
using namespace std;  
  
int fois2 ( int i ) { return i*2; }  
  
static void affiche ( const vector < int > v, const char *msg = "" )  
{ cout << msg;  
  for ( vector < int >::const_iterator iv = v.cbegin ( ) ; iv != v.cend ( ) ; ++iv )  
  { cout << ' ' << *iv;  
  cout << endl;  
}  
int main ( void )  
{ vector < int > v1; vector < int > v2; int n = 11;  
  
  v1.push_back ( n ); // v1 = 11  
  for ( int i = 1 ; i < 3 ; i++ ) // Initialisation du vecteur v1  
  { v1.push_back ( v1.back ( ) + n ); }  
  affiche ( v1, "v1 =" ); // v1 = 11 22 33  
  v2.resize ( v1.size ( ) ); // redimensionnement de v2 à la taille de v1  
  transform ( v1.begin ( ), v1.end ( ), v2.begin ( ), fois2 );  
  affiche ( v2, "v2 (après x 2) =" ); // v2 (après x 2) = 22 44 66  
  // std::plus : additionne ses 2 paramètres formels  
  transform ( v1.begin ( ), v1.end ( ), v2.begin ( ), v1.begin ( ), plus < int > () );  
  affiche ( v1, "v1 (après plus <int>) =" ); // v1 (après plus < int >) = 22 44 66  
  return 0;  
}
```

⇒ Utilisation de l'algorithme `transform`
avec 2 variantes d'appel

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Algorithmes sur des séquences avec modification (suite)
 - `generate` : affecte à chaque élément d'une plage donnée une valeur générée par une fonction (paramètre de type `Generator`)
 - `generate_n` (C++11) : affecte aux `n` premiers éléments d'une plage donnée des valeurs générées par une fonction (paramètre de type `Generator`)
 - `remove` et `remove_if` : supprime tous les éléments dans une plage donnée s'ils respectent un critère donné
 - `remove_copy` et `remove_copy_if` : copie les éléments d'une plage source vers une plage destination en omettant tous les éléments qui respectent un critère donné.
- Note: le recouvrement entre les plages source et destination est interdit
- `replace` et `replace_if` : remplace tous les éléments d'une plage donné qui respectent un critère donné par une nouvelle valeur



Algorithme : replace et replace_if

```
#include <iostream>
#include <algorithm>
#include <array>
#include <functional> // std::placeholders
using namespace std;

static void affiche ( const array < int, 10 > t, const char *msg = "" )
{ cout << msg;
  for ( array < int, 10 >::const_iterator it = t.cbegin() ; it != t.cend() ; ++it )
  { cout << ' ' << *it; }
  cout << endl;
}
int main ( void )
{
  array < int, 10 > t { -3, 1, 2, 8, 66, 8, 1, 77, -11, 5 };
  // Remplace toutes les valeurs égales à 8 par 88 dans l'ensemble du conteneur
  replace ( t.begin ( ), t.end ( ), 8, 88 );
  affiche ( t, "array (après remplacement) = " );
  // Remplace toutes les valeurs inférieures à 5, dans la plage (début+2, fin-2)
  replace_if ( t.begin ( ) + 2, t.end ( ) - 2,
               bind ( less < int > ( ), placeholders::_1, 5 ), 55 );
  affiche ( t, "array (après remplacement conditionnel) = " );
  // Affichage en utilisant des syntaxes C++11
  // for ( int val : t )
  // { cout << val << ' ' ; }
  // cout << endl;
  return 0;
}
```



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Algorithmes sur des séquences avec modification (suite)
 - `replace_copy` et `replace_copy_if` : copie tous les éléments d'une plage source vers une plage destination en remplaçant tous les éléments qui respectent un critère donné par une nouvelle valeur.
Note: le recouvrement entre les plages source et destination est interdit
 - `swap` : échange les valeurs de 2 objets
 - `swap_ranges` : échange les éléments d'une plage source avec une plage destination
 - `iter_swap` : échange les valeurs des éléments pointés par leur *itérateur* respectif
 - `reverse` : inverse l'ordre des éléments dans une plage donnée
 - `reverse_copy` : copie les éléments d'une plage source vers une plage destination en inversant l'ordre
 - `rotate` : effectue une rotation à gauche des éléments d'une plage donnée (le premier élément de la plage devient le dernier)



Algorithme : swap

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

int main ( void )
{
    int a = 5;
    int b = 3;

    // Valeurs avant la permutation
    cout << "a = " << a << " - " << "b = " << b << endl;

    // Effectue le même travail que la fonction Permuter de IFA-3-S1-EC-POO1
    swap ( a, b );

    // Valeurs après la permutation
    cout << "a = " << a << " - " << "b = " << b << endl;

    return 0;
}
```

```
mmaranzana@iff501-219-13:~/Cours/C++/ALGORITHM$ ./mm
a = 5 - b = 3
a = 3 - b = 5
mmaranzana@iff501-219-13:~/Cours/C++/ALGORITHM$
```



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

◆ Algorithmes sur des séquences avec modification (suite)

- `rotate_copy` : copie les éléments d'une plage source vers une plage destination en effectuant une rotation des éléments au moment de la copie, à partir d'un élément de référence
- `random_shuffle` (jusqu'à C++17) et `shuffle` (C++11) : mélange aléatoirement les éléments d'une plage donnée
- `sample` (C++17) : sélectionne aléatoirement `n` éléments dans une plage donnée et les écrit dans l'*itérateur* de sortie (`OutputIterator`)
- `unique` : élimine tous les doublons `consécutifs` dans une plage donnée
- `unique_copy` : copie les éléments d'une plage source vers une plage destination en supprimant les doublons consécutifs



Algorithme : shuffle

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

int main ( void )
{
    vector < int > v = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    copy ( v.begin ( ), v.end ( ) - 1, ostream_iterator < int > ( cout, ", " ) );
    cout << v.back ( ) << endl;

    // Utilisation d'un générateur de nombres pseudo-aléatoires de Mersenne Twister
    random_device rd;
    mt19937 g ( rd ( ) );

    shuffle ( v.begin ( ), v.end ( ), g );

    copy ( v.begin ( ), v.end ( ) - 1, ostream_iterator < int > ( cout, ", " ) );
    cout << v.back ( ) << endl;
}

return 0;
}
```

```
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$ ./mm
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
10, 4, 2, 5, 7, 1, 9, 3, 8, 6
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$
```



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

◆ Algorithmes de tris sur des séquences

- `is_sorted` (C++11) : vérifie si les éléments d'une plage donnée sont triés dans un ordre croissant
- `is_sorted_until` (C++11) : recherche la plus grande plage dans la plage donnée qui possède des éléments triés dans un ordre croissant
- `sort` : trie les éléments d'une plage donnée dans un ordre croissant
- `partial_sort` : effectue un tri partiel des éléments d'une plage donnée en fonction d'un nombre d'éléments à trier
- `partial_sort_copy` : effectue un tri partiel des éléments d'une plage donnée en fonction d'un nombre d'éléments à trier et copie les éléments triés dans une nouvelle plage
- `stable_sort` : trie les éléments d'une plage donnée dans un ordre croissant en conservant l'ordre des éléments égaux
- `nth_element` : réorganise les éléments d'une plage donnée de telle sorte que l'élément à la position `n` serait l'élément à cette position dans une séquence triée (les autres éléments sont laissés sans ordre spécifique, sachant qu'aucun des éléments précédant le `n`-ième n'est supérieur à celui-ci et qu'aucun des éléments le suivant n'est inférieur)

INSA

© MM - Reproduction interdite sans l'autorisation de l'auteur



Algorithme : partial_sort_copy

```
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$ ./mm
Copie des éléments (5) vers le plus petit vecteur (3) (ordre croissant) = a b c
L'itérateur est sur l'élément "past-the-end-element"
Copie des éléments (5) vers le plus grand vecteur (7) (ordre décroissant) =
e d c b a g k
L'itérateur est sur l'élément de valeur = g
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$
```

```
#include <algorithm>
...
using namespace std;

int main ( void )
{ vector < char > v0 { 'd', 'b', 'e', 'a', 'c' };
  vector < char > v1 { 'x', 'y', 'z' };
  vector < char > v2 { 'z', 'p', 'o', 'm', 'r', 'g', 'k' };
  vector < char >::iterator it;

  it = partial_sort_copy ( v0.begin ( ), v0.end ( ), v1.begin ( ), v1.end ( ) );
  cout << "Copie des éléments (5) vers le plus petit vecteur (3) (ordre croissant) = ";
  copy ( v1.begin ( ), v1.end ( ), ostream_iterator < char > ( cout, " " ) );
  cout << endl;
  if ( it == v1.end ( ) )
    cout << "L'itérateur est sur l'élément \"past-the-end-element\" " << endl;
  else
    cout << "L'itérateur est sur l'élément de valeur = " << *it << endl;

  it = partial_sort_copy ( v0.begin(),v0.end(),v2.begin(),v2.end(),greater<char>());
  cout << "Copie des éléments (5) vers le plus grand vecteur (7) (ordre décroissant) = ";
  copy ( v2.begin ( ), v2.end ( ), ostream_iterator < char > ( cout, " " ) );
  cout << endl;
  if ( it == v2.end ( ) )
    cout << "L'itérateur est sur l'élément \"past-the-end-element\" " << endl;
  else
    cout << "L'itérateur est sur l'élément de valeur = " << *it << endl;
  return 0;
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Algorithmes de partitionnement de séquences
 - `is_partitioned` (C++11) : renvoie `true` si tous les éléments de la plage donnée pour lesquels le prédictat (paramètre de l'algorithme) retourne `true`, précèdent ceux pour lesquels il renvoie `false`
 - `partition` : réarrange les éléments de la plage donnée, pour que tous les éléments pour lesquels le prédictat (paramètre de l'algorithme) renvoie `true` précèdent tous ceux pour lesquels il renvoie `false`
L'*itérateur* renvoyé pointe le premier élément du deuxième groupe
 - `partition_copy` (C++11) : copie les éléments de la plage donnée, pour lesquels le prédictat (paramètre) retourne `true`, dans la plage pointée par l'*itérateur résultat_true* (paramètre) et ceux pour lesquels le prédictat renvoie `false`, dans la plage pointée par l'*itérateur résultat_false* (paramètre)



© MM - Reproduction interdite sans l'autorisation de l'auteur



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Algorithmes de partitionnement de séquences
 - `stable_partition` : réarrange les éléments de la plage donnée, pour que tous les éléments pour lesquels le prédictat (paramètre) renvoie `true` précèdent tous ceux pour lesquels il renvoie `false`
Contrairement à l'algorithme `partition`, l'ordre relatif des éléments au sein de chaque groupe est préservé
 - `partition_point` (C++11) : retourne un *itérateur* sur le premier élément de la plage partitionnée pour lequel le prédictat (paramètre) n'est pas `true` (définition du point de partitionnement)

Note : les éléments de la plage doivent déjà être partitionnés à l'aide d'un appel à l'algorithme `partition`, avec les mêmes arguments.



© MM - Reproduction interdite sans l'autorisation de l'auteur



Algorithme : is_partition et stable_partition

```

#include <iostream>
#include <algorithm>
#include <array>
using namespace std;
static const int N = 8;

static bool estImpair ( int nb ) { return nb % 2; }

static void affiche ( const array < int, N > t, const char *msg = "" )
{ // Affichage du contenu du tableau t
    cout << msg;
    for ( int i = 0 ; i < N - 1 ; i++ )
    { cout << t [ i ] << ", " ;
    cout << t [ N - 1 ];
    if ( is_partitioned ( t.begin ( ), t.end ( ), estImpair ) )
    { cout << " (partitionné)" << endl; }
    else
    { cout << " (non partitionné)" << endl; }
}

int main ( void )
{ array < int, N > t { 0, 1, 2, 3, 4, 5, 6, 7 };
// Affichage du contenu du tableau t (avant partition)
affiche ( t, "t (avant partition) = " );
// Partition du tableau t selon le critère de la parité des nombres
stable_partition ( t.begin ( ), t.end ( ), estImpair );
// Affichage du contenu du tableau t
affiche ( t, "t (après partition) = " );
return 0;
}

```

// Résultat de l'exécution du programme (avec `stable_partition`)
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHMS ./mm
t (avant partition) = 0, 1, 2, 3, 4, 5, 6, 7 (non partitionné)
t (après partition) = 1, 3, 5, 7, 0, 2, 4, 6 (partitionné)
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHMS



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Algorithme

- Classes d'algorithmes (`#include <algorithm>`)
 - ◆ Algorithmes de recherche binaire
 - `lower_bound` : renvoie un *itérateur* pointant sur le premier élément de la plage donnée qui est plus petit ou égal à `val` (paramètre)
 - `upper_bound` : renvoie un *itérateur* pointant sur le premier élément de la plage donnée qui est plus grand que `val` (paramètre de l'algorithme)
 - `binary_search` : renvoie `true` si un élément de la plage donnée est égal à `val` (paramètre) et `false` sinon (utilisation de `operator <` ou d'une fonction personnelle `comp` pour la comparaison)

Note : deux éléments `a` et `b` sont considérés comme égaux...
`Si (! (a < b) && ! (b < a))` ou si `(! comp (a, b) && ! comp (b, a))`

Les éléments de la plage doivent déjà être triés selon ce même critère (`operator <` ou `comp`)

- `equal_range` : renvoie une paire d'*itérateurs*, définissant une sous-plage, incluant tous les éléments de la plage donnée dont la valeur est égale à `val` (paramètre)



© MM - Reproduction interdite sans l'autorisation de l'auteur.

INSA INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON

MM - Département Informatique

C++ Avancé – 104



Algorithme : equal_range

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

bool plusGrand ( char c1, char c2 ) { return ( c1 > c2 ); }

int main ( void )
{
    char mesCar [ ] = { 'c', 'h', 'x', 'x', 'h', 'c', 'c', 'h' };
    vector < char > v ( mesCar, mesCar + 8 ); // 'c' 'h' 'x' 'x' 'h' 'c' 'c' 'h'
    pair < vector < char >::iterator, vector < char >::iterator > bounds;

    // En utilisant la comparaison par défaut (operator <)
    sort ( v.begin ( ), v.end ( ) );
    bounds = equal_range ( v.begin(), v.end(), 'h' );
    cout << "Itérateurs début = " << ( bounds.first - v.begin ( ) );
    cout << " et fin = " << ( bounds.second - v.begin ( ) ) << endl;

    // En utilisant une comparaison personnelle comp = "plusGrand"
    sort ( v.begin ( ), v.end ( ), plusGrand );
    bounds = equal_range ( v.begin(), v.end(), 'h', plusGrand );
    cout << "Itérateurs début = " << ( bounds.first - v.begin ( ) );
    cout << " et fin = " << ( bounds.second - v.begin ( ) ) << endl;

    return 0;
}
```

```
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$ ./mm
Itérateurs début = 3 et fin = 6
Itérateurs début = 2 et fin = 5
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$
```

© MM. Reproduction interdite sans l'autorisation de l'auteur.



Algorithme

■ Classes d'algorithmes (#include <algorithm>)

- ◆ Algorithmes sur des ensembles (séquences ordonnées)
 - **merge** : fusionne les éléments issus des 2 plages triées dans une nouvelle plage repérée par l'itérateur **résultat** (paramètre) (tous les éléments sont triés)
 - **inplace_merge** : fusionne deux plages triées consécutives (**premier**, **milieu**) et (**milieu**, **dernier**), en plaçant le résultat dans la plage triée (**premier**, **dernier**)
 - **includes** : renvoie **true** si une plage donnée triée contient tous les éléments d'une autre plage donnée triée
Note : deux éléments **a** et **b** sont considérés comme égaux...
Si **(! (a<b) && !(b<a))** ou si **(!comp(a,b) && !comp(b,a))**
Les éléments de la plage doivent déjà être triés selon ce même critère (**operator <** ou **comp**)
 - **set_difference** : copie les éléments de la plage triée (**premier1**, **dernier1**) qui ne se trouvent pas dans la plage triée (**premier2**, **dernier2**) vers la plage identifiée par l'itérateur **résultat** (paramètre)



© MM. Reproduction interdite sans l'autorisation de l'auteur.



Algorithm : includes et set_difference

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

static void affiche ( const vector < int > v, const char *msg = "" )
{ cout << msg;
  for ( vector < int >::const_iterator iv = v.cbegin ( ) ; iv != v.cend ( ) ; ++iv )
  { cout << " " << *iv;
  cout << endl;
}

int main ( void )
{ vector < int > v1 { 1, 2, 5, 5, 5, 9 }; // déjà trié
  vector < int > v2 { 2, 5, 7 };           // déjà trié
  vector < int > v3 { 2, 9 };             // déjà trié
  vector < int > diff;                 // vecteur résultat
  cout << "v2 est inclus dans v1 = " << boolalpha
    << includes ( v1.begin ( ), v1.end ( ), v2.begin ( ), v2.end ( ) ) << endl;
  cout << "v3 est inclus dans v1 = " << boolalpha
    << includes ( v1.begin ( ), v1.end ( ), v3.begin ( ), v3.end ( ) ) << endl;
  set_difference ( v1.begin ( ), v1.end ( ), v2.begin ( ), v2.end ( ),
    inserter ( diff, diff.begin ( ) ) );
  affiche ( v1, "v1 =" );
  affiche ( v2, "v2 =" );
  affiche ( diff, "diff =" );
  return 0;
}
```

© MM - Reproduction interdite sans l'autorisation de l'auteur.



Algorithm

- Classes d'algorithmes (`#include <algorithm>`)
 - ◆ Algorithmes sur des ensembles (séquences ordonnées)
 - **set_intersection** : construit l'intersection entre 2 plages données triées et la range triée dans la plage repérée par l'*itérateur résultat* (paramètre de l'algorithme)
 - **set_symmetric_difference** : construit à partir de 2 plages sources données triées, une nouvelle plage triée contenant tous les éléments qui ne sont pas présents à la fois dans les 2 plages sources
 - **set_union** : construit la réunion entre 2 plages données triées et la range triée dans la plage repérée par l'*itérateur résultat* (paramètre de l'algorithme)

© MM - Reproduction interdite sans l'autorisation de l'auteur.


INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON


212



Algorithme : set_intersection et set_union

```
#include <algorithm>
#include <iterator>
using namespace std;

int main ( void )
{
    vector < int > v1 { 1, 4, 2, 5, 3, 4, 5, 6, 9, 7, 8, 7 };
    vector < int > v2 { 9, 5, 7, 5, 10 };
    sort ( v1.begin ( ), v1.end ( ) ); // v1 = 1, 2, 3, 4, 4, 5, 5, 6, 7, 7, 8, 9
    sort ( v2.begin ( ), v2.end ( ) ); // v2 = 5, 5, 7, 9, 10

    vector < int > vInter;
    set_intersection ( v1.begin ( ), v1.end ( ), v2.begin ( ), v2.end ( ),
                       back_inserter ( vInter ) );

    cout << "Intersection = ";
    copy ( vInter.begin(), vInter.end(), ostream_iterator <int> ( cout, " " ) );
    cout << endl;

    vector < int > vReunion;
    set_union ( v1.begin ( ), v1.end ( ), v2.begin ( ), v2.end ( ),
                back_inserter ( vReunion ) );

    cout << "Réunion = ";
    copy ( vReunion.begin(), vReunion.end(), ostream_iterator <int> ( cout, " " ) );
    cout << endl;

    return 0;
}
```

```
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$ ./mm
Intersection = 5 5 7 9
Réunion = 1 2 3 4 4 5 5 6 7 7 8 9 10
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM$
```

© MM. Reproduction interdite sans l'autorisation de l'auteur.



Algorithme

- Classes d'algorithmes (`#include <algorithm>`)
 - ◆ Algorithmes sur des min ou max dans une séquence
 - `max` : renvoie la valeur max parmi toutes les valeurs fournies (max entre 2 valeurs `a` et `b` ou max dans une liste d'initialisation)
 - `max_element` : renvoie un *itérateur* sur le premier élément le plus grand dans une plage donnée
 - `min` : renvoie la valeur min parmi toutes les valeurs fournies (min entre 2 valeurs `a` et `b` ou min dans une liste d'initialisation)
 - `min_element` : renvoie un *itérateur* sur le premier élément le plus petit dans une plage donnée
 - `minmax` (C++11) : renvoie une paire de valeurs...
 - les valeurs min et max parmi toutes les valeurs fournies (min et max entre 2 valeurs `a` et `b` ou min et max dans une liste d'initialisation)
 - `minmax_element` (C++11) : renvoie une paire d'*itérateurs*...
 - un *itérateur* sur le premier élément le plus petit et un *itérateur* sur le premier élément le plus grand dans une plage donnée
 - `clamp` (C++17) : non présenté dans le cadre de ce cours

Note : utilisation possible d'une fonction personnelle de comparaison `comp` pour ces différents algorithmes (à la place de `operator <`)


INSA
214



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Algorithmes sur des min ou max dans une séquence (suite)
 - `lexicographical_compare` : renvoie `true` si la première plage donnée est *lexicographiquement* plus petite que la seconde plage
 - `is_permutation` (C++11) : renvoie `true` s'il existe une permutation des éléments de la plage (`premier1, dernier1`) qui rend cette plage égale à la plage (`premier2, dernier2`)
 - `next_permutation` : transforme la plage donnée en utilisant la prochaine permutation de l'ensemble des permutations ordonnées *lexicographiquement* (utilisation de `operator <` ou `comp`)
 - Renvoie `true`, si la permutation existe
 - `prev_permutation` : transforme la plage donnée en utilisant la précédente permutation de l'ensemble des permutations ordonnées *lexicographiquement* (utilisation de `operator <` ou `comp`)
 - Renvoie `true`, si la permutation existe



Algorithme : `next_permutation` et `prev_permutation`

```
#include <algorithm>
#include <string>
#include <iostream>
#include <functional>
using namespace std;

int main ( void )
{
    string s1 = "abb";
    sort ( s1.begin ( ), s1.end ( ) );
    do
    {
        cout << s1 << " ";
    } while ( next_permutation ( s1.begin ( ), s1.end ( ) ) );
    cout << endl;

    string s2 = "xyz";
    sort ( s2.begin ( ), s2.end ( ), greater < char > ( ) );
    do
    {
        cout << s2 << " ";
    } while ( prev_permutation ( s2.begin ( ), s2.end ( ) ) );
    cout << endl;

    return 0;
}
```

mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM\$./mm
abb bab bba
zyx zxy yzx yxz xzy xyz
mmaranzana@if501-219-13:~/Cours/C++/ALGORITHM\$





Algorithme

■ Classes d'algorithmes (#include <algorithm>)

◆ Opérations numériques

- **iota** (C++11) : remplit une plage donnée avec des valeurs successives croissantes, à partir d'une valeur initiale fournie
- **accumulate** : effectue la somme entre une valeur val (passée en paramètre) et les éléments d'une plage donnée (utilisation de **operator +** ou d'une fonction binaire donnée **op**)
- **inner_product** : non présenté dans le cadre de ce cours
- **adjacent_difference** : non présenté dans le cadre de ce cours
- **partial_sum** : calcule les sommes partielles des éléments dans les sous-plages de la plage (**premier**, **dernier**) et les écrit dans une plage commençant à **résultat** (utilisation de **operator +** ou d'une fonction binaire donnée **op**)
- **reduce** (C++17) : non présenté dans le cadre de ce cours



Algorithme : iota et copy

```
// Exemple de manipulation d'algorithmes (copy et iota) sur des conteneurs (vector)
#include <iostream>
#include <algorithm>
#include <vector>
#include <iomanip>
using namespace std;

int main ()
{
    vector < int > vSource ( 10 );
    iota ( vSource.begin ( ), vSource.end ( ), 0 );

    vector < int > vDestination ( vSource.size ( ) );
    copy ( vSource.begin ( ), vSource.end ( ), vDestination.begin ( ) );

    cout << "vDestination contient : ";
    copy ( vDestination.begin ( ), vDestination.end ( ),
           ostream_iterator < int > ( cout, " " ) );
    cout << endl;

    return 0;
}
```

⇒ Algorithme qui remplit la séquence définie par **begin / end** avec des valeurs croissantes séquentielles, en partant de **0** c'est-à-dire, 0, 1, 2, etc.

⇒ Définition du vecteur destination à partir de la taille du vecteur source

⇒ Copie la séquence d'éléments définie par la plage **source begin / end** dans une plage **destination** débutant à **begin**

⇒ Utilisation d'un **itérateur** sur le flux de sortie **cout** (**ostream_iterator**) pour afficher le contenu du vecteur **vDestination** (séquence d'éléments définie par **begin / end**)

⇒ Entre chaque élément du vecteur, il est possible d'insérer un délimiteur (ici " , ") dans le flux de sortie



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Opérations numériques (suite)
 - `exclusive_scan` (C++17) : non présenté dans le cadre de ce cours
 - `inclusive_scan` (C++17) : non présenté dans le cadre de ce cours
 - `transform_reduce` (C++17) : non présenté dans le cadre de ce cours
 - `transform_exclusive_scan` (C++17) : non présenté dans le cadre de ce cours
 - `transform_inclusive_scan` (C++17) : non présenté dans le cadre de ce cours



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur.



Algorithme

■ Classes d'algorithmes (`#include <algorithm>`)

- ◆ Opérations sur des zones de mémoire non initialisées
 - `uninitialized_copy` : non présenté dans le cadre de ce cours
 - `uninitialized_copy_n` (C++11) : non présenté dans le cadre de ce cours
 - `uninitialized_fill` : non présenté dans le cadre de ce cours
 - `uninitialized_fill_n` : non présenté dans le cadre de ce cours
 - `uninitialized_move` (C++17) : non présenté dans le cadre de ce cours
 - `uninitialized_move_n` (C++17) : non présenté dans le cadre de ce cours



INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
LYON



© MM - Reproduction interdite sans l'autorisation de l'auteur.