
Programmation avancée en C++



Plan

Concepts et mécanismes fondamentaux de C++

- Présentation,
- Histoire et caractéristiques de C++,
- Objets et types,
- La classe C++ et ses nombreuses interprétations,
- Création, suppression et initialisation d'objets,
- Copie d'objets.

Les exceptions

- Mécanisme,
- Exceptions dans un constructeur,
- Listes d'exceptions levées,
- Organisation d'exceptions dans une hiérarchie de classes,
- Sécurité des exceptions (l'idiome de programmation "Resource Acquisition Is Initialization").

Run Time Type Information (RTTI)

- Motivation (problématique du 'down cast', l'opérateur 'dynamic_cast'),
- La classe 'typeid' et l'opérateur 'typeid',
- Utiliser RTTI correctement,
- Les opérateurs 'cast' en C++ (les cast 'static', 'reinterpret' et 'dynamic').

Héritage multiple en C++

- Héritage multiple régulier,
- Héritage multiple virtuel,
- Construction des classes de base virtuelles,
- Conversions en cas d'héritage multiple.

La STL

- Introduction
- La classe string
- Conteneurs et itérateurs de la STL
- les itérateurs sur les conteneurs
- Utiliser ses objets dans les conteneurs de la STL
- Algorithmes et, functors et predicats
- Algorithmes, prédicats, functors et binders
- STL et la performance

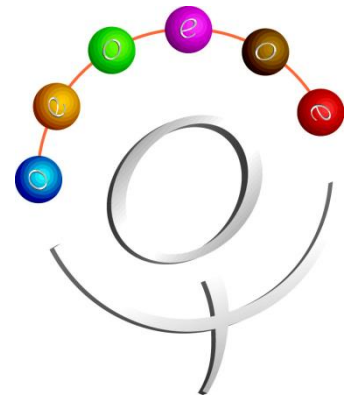
Multi threading

- Problématique Illustration sous Unix
- La programmation concurrente
- Implémentations en C++
- Utiliser la librairie Boost threads

Annexes

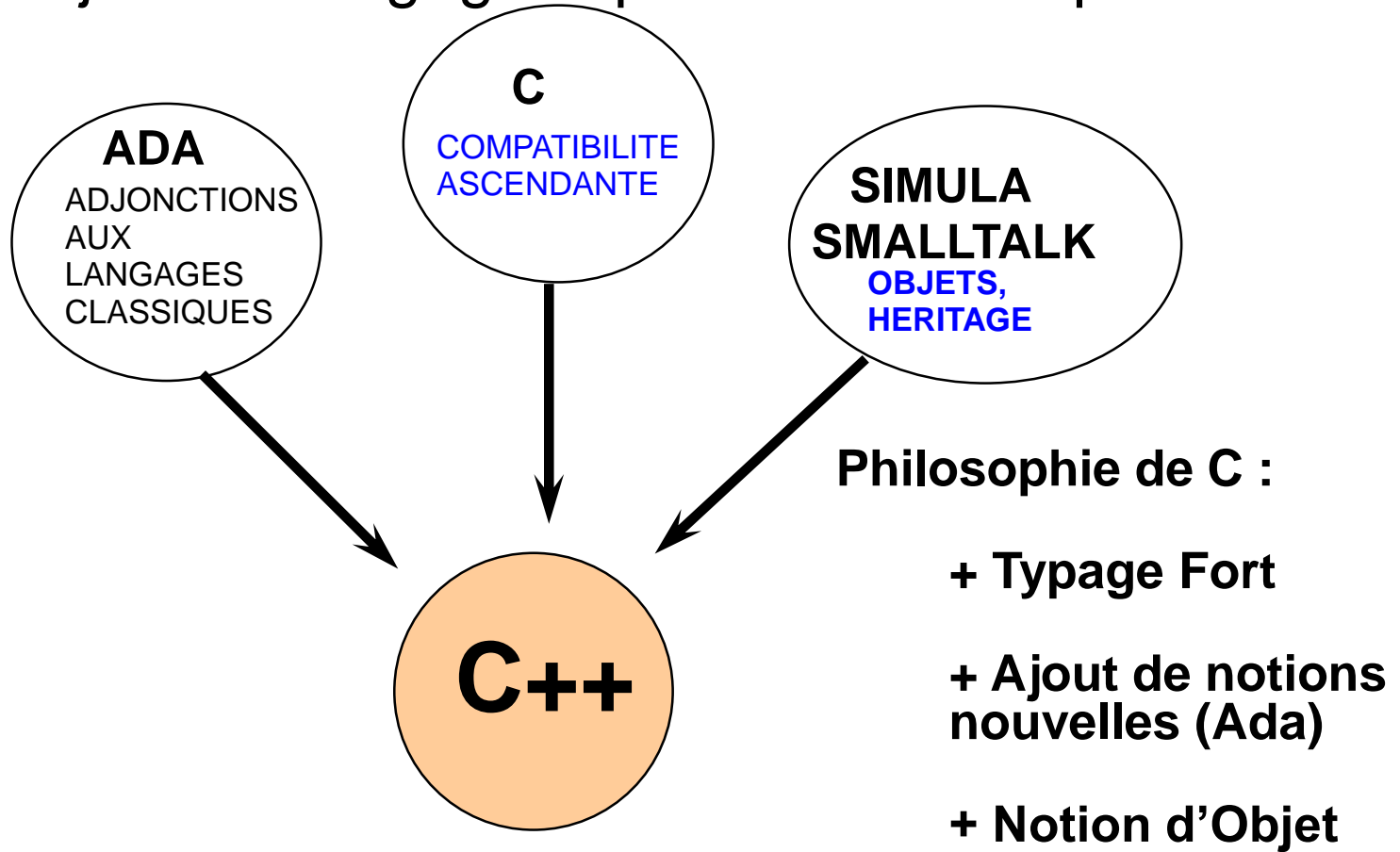
- Le projet Boost
- Sérialisation en C++
- C++ 0x en question
- Pointeurs et autres

Concepts et mécanismes fondamentaux de C++



Présentation

- Mise à jour du langage C par B. Stroustrup

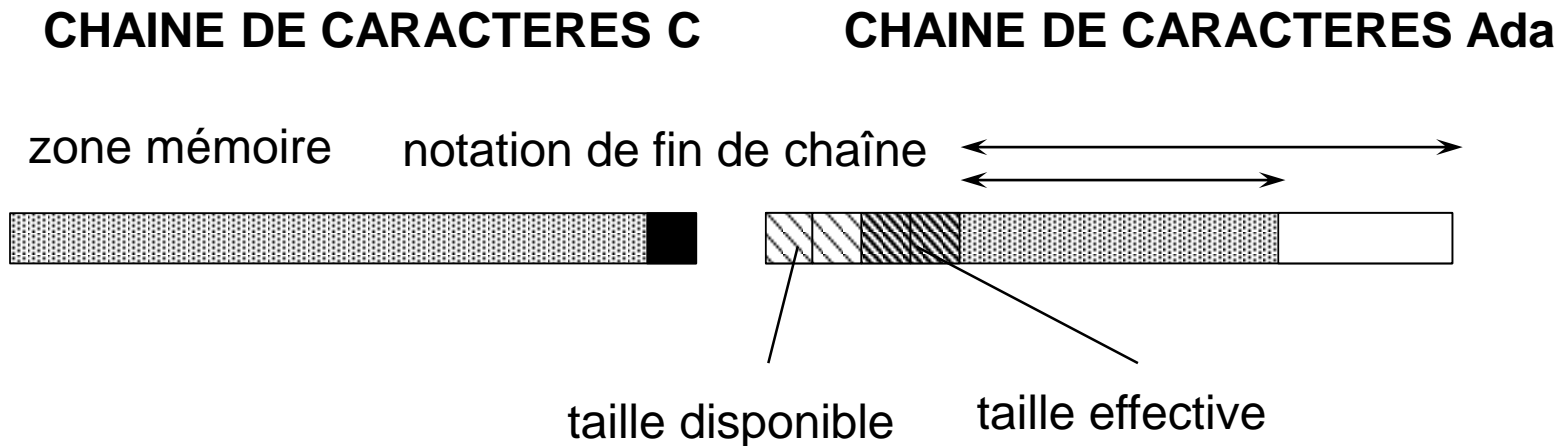


Philosophie Générale du C (1)

- Langage sous-jacent de C++
 - Structure (haut niveau)
 - Proximité de la machine (Bas niveau)
 - Efficacité
 - Pas de philosophie de programmation imposée
 - Puissance de la programmation
 - Concision
 - Capacités du langage

Philosophie Générale du C (2)

- Accent mis sur l'efficacité sur tous les plans
 - Contrôles de programmation relégués au second plan
 - Laxisme du compilateur
 - Absence de contrôle dynamique



C++ : Un Compromis ...

- Programmer objet en C++ nécessite une discipline du programmeur
- Langage objet compilé
- Langage objet efficace
- Syntaxe compatible C



**C++ NE CONTRAINT PAS A
PROGRAMMER PAR OBJETS**



IL FAUT APPRENDRE A OUBLIER C

Positionnement Du C++

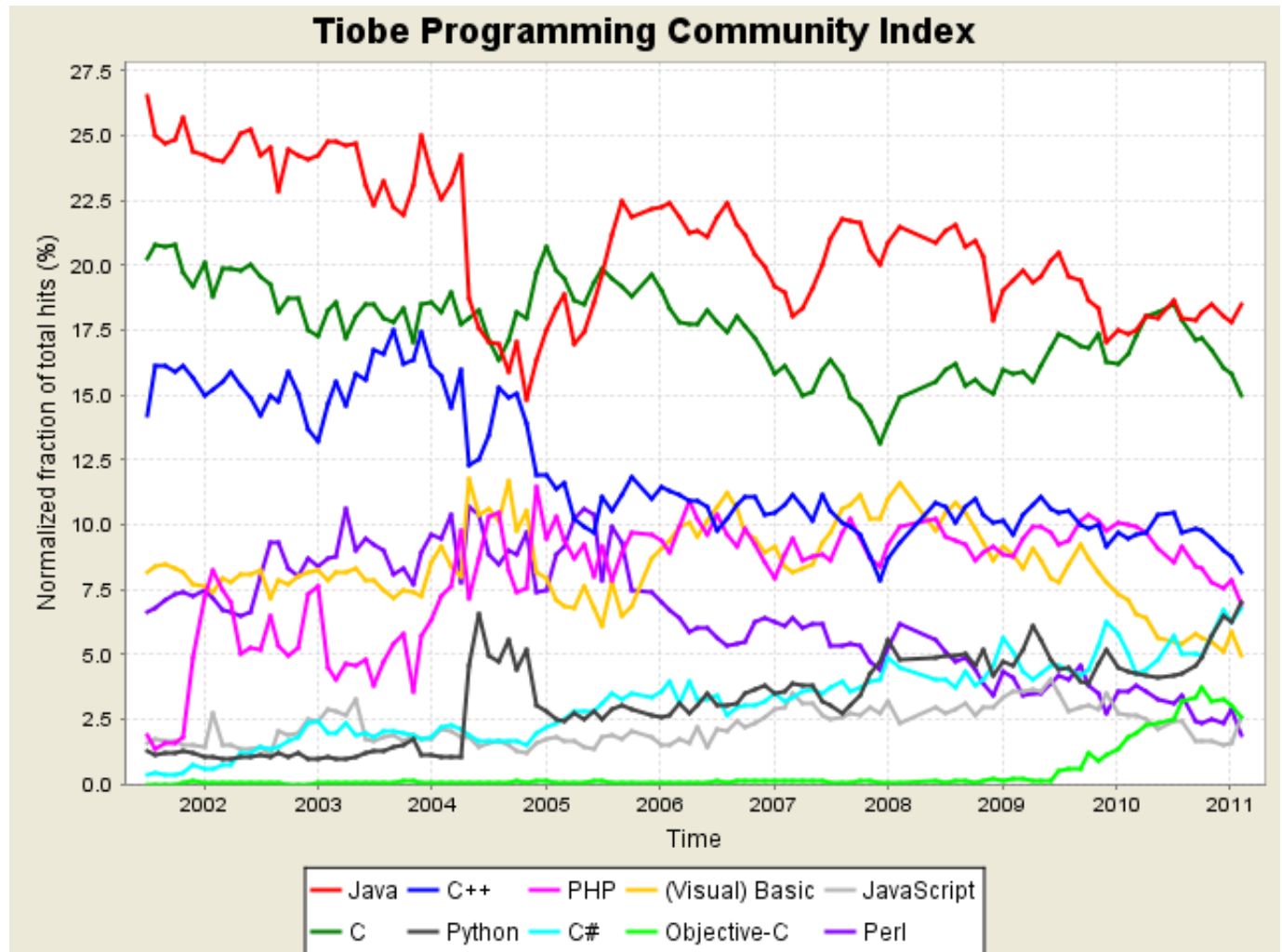
- C++ : Remplaçant de C
- Compatibilité ascendante
 - .C
 - .C ANSI
- Appui AT&T



**C++ est un compromis
C / Langages objet**

Langages Concurrents Du C++

- C++ occupe une part prépondérante du marché

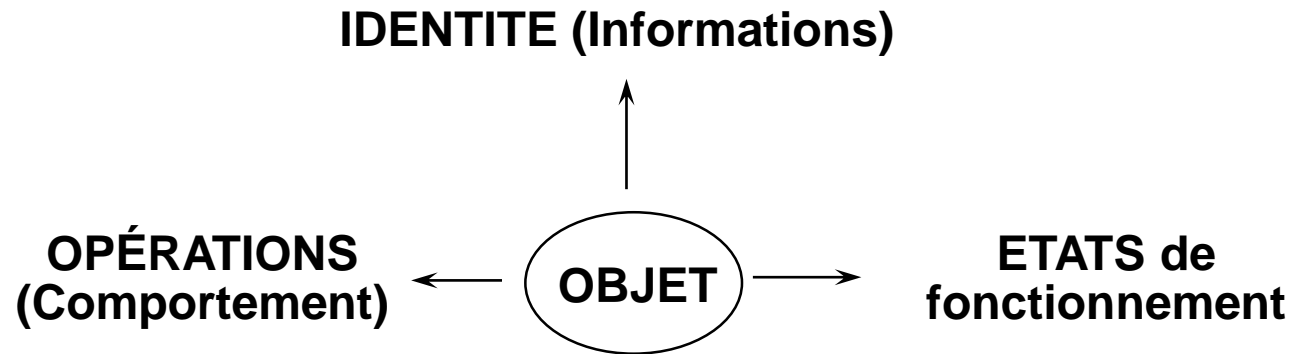


Concepts et mécanismes fondamentaux de C++

**La classe C++ et ses nombreuses
interprétations**

Objet

- Nature des objets :
 - Élément visible ou tangible
 - Compréhensible intellectuellement

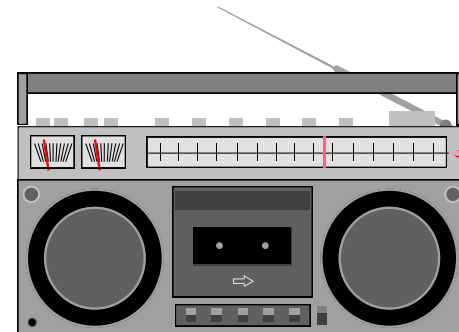
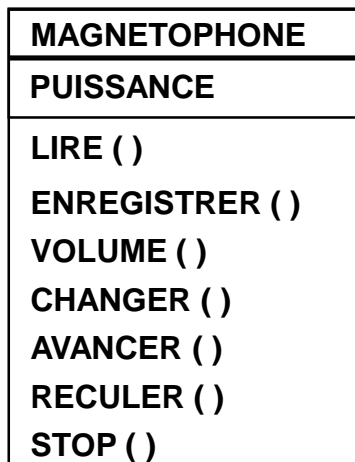


Abstraction (1)

- Définition :
 - « Démarche de l'esprit qui consiste, au cours d'un raisonnement, à éliminer les aspects les moins pertinents de la réflexion pour ne considérer que ceux qui sont essentiels. »
- Consiste à identifier les caractéristiques essentielles d'un objet :
 - Propriétés similaires entre les objets
 - Actions et comportements similaires entre les objets
- Résultat de l'abstraction : Le concept ou la classe d'objets

Abstraction (2)

- Exemple du magnétophone
 - Caractéristiques définies :
 - Puissance, etc ...
 - Actions possibles :
 - LIRE(),
 - ENREGISTRER(),
 - CHANGER(),
 - AVANCER(),
 - RECULER(),
 - STOP()

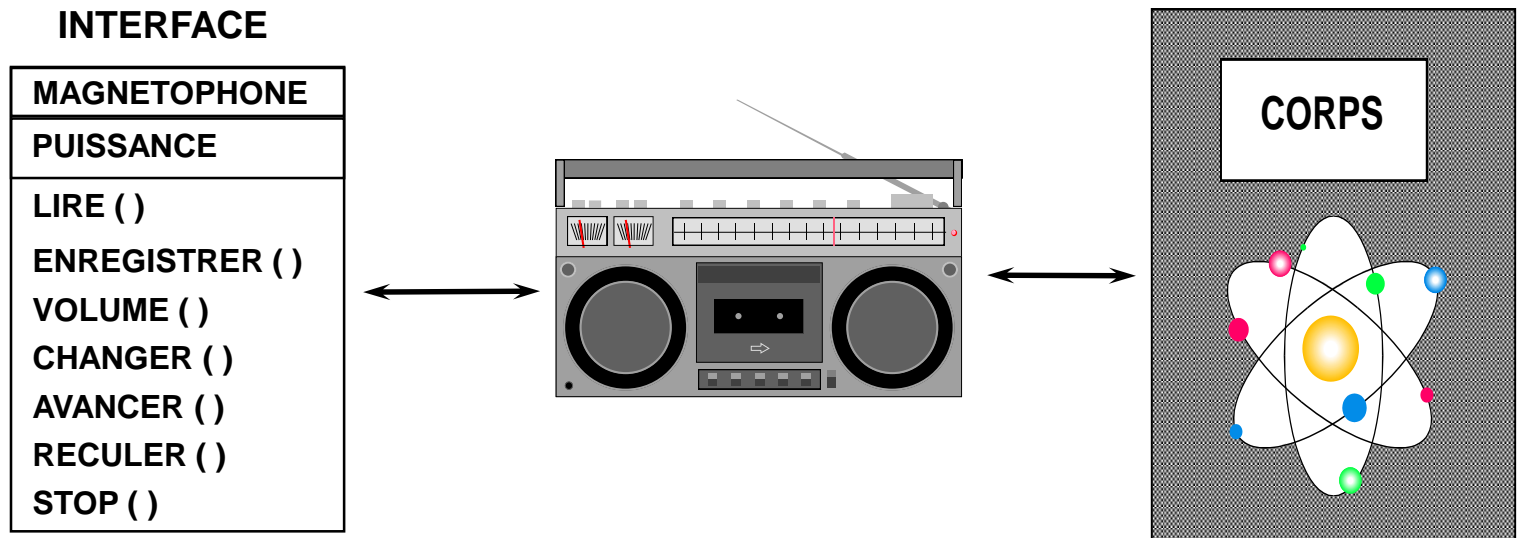


Apports de l'Abstraction

- Diminue le nombre d'éléments concernés
 - Gérer UN ensemble d'objets plutôt que les N objets
- Simplifie le problème
 - Seules les propriétés importantes dans le contexte sont traitées
- Permet de définir des classes d'objets
- Facilite la réutilisation
- Impose une vraie notion de TYPE

Encapsulation

- Encapsulation de la vue interne de l'objet
 - Interface : vue externe (abstraction)
 - Corps : implémentation des comportements

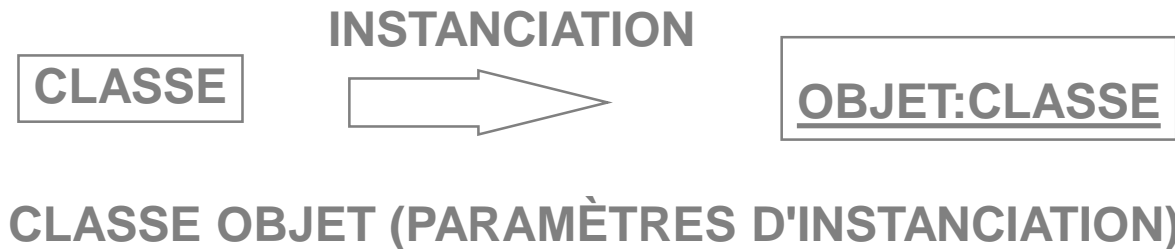


Apports de l'Encapsulation

- Localisation des responsabilités
- Clarification des A.P.I.
- Testabilité
- Tout n'est pas nouveau, mais certains aspects sont renforcés.

Instanciation, objets

- Les objets sont créés par instanciation de la classe



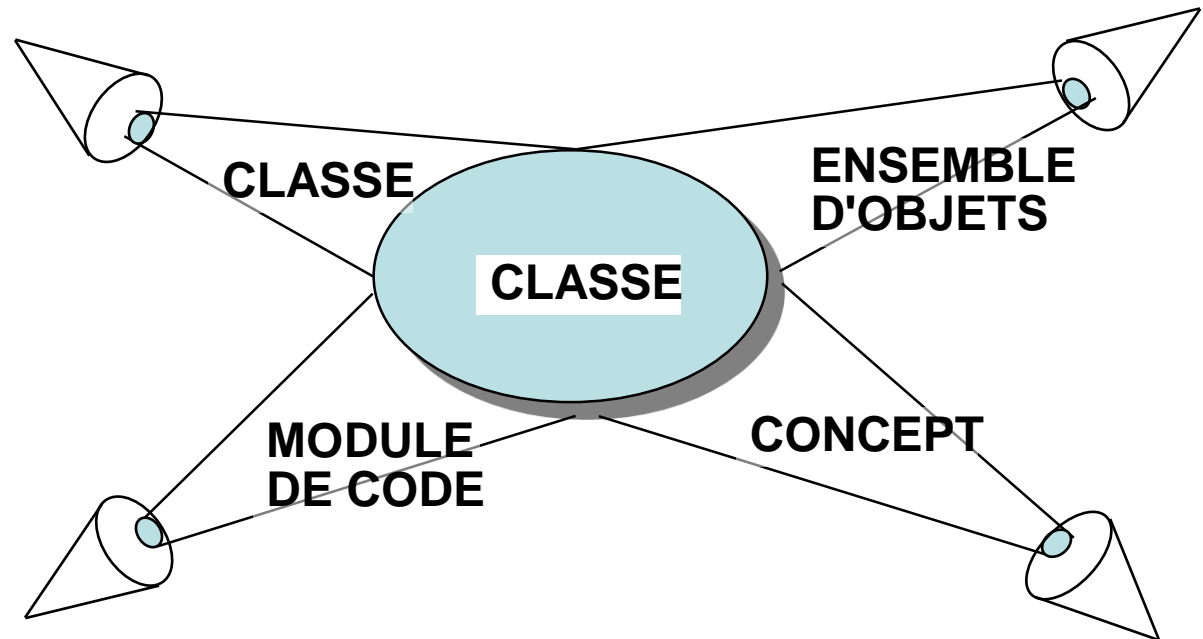
- Exemple :



**L'OBJET POSSÈDE TOUTES LES PROPRIÉTÉS ET TOUS
LES COMPORTEMENTS DÉFINIS AU NIVEAU DE SA
CLASSE**

Les 4 Vues d'une Classe

- Une classe offre différentes vues, utiles pour :
 - Les spécifications => Concept
 - La conception => Structure informatique (Moule à objets)
 - La définition ensembliste
=> Ensemble des représentants possibles
 - Le codage



Classe : Concept

- Un concept : Application aux spécifications
 - Nom commun
 - Une voiture, un match de foot, un triangle, un ensemble, ...
 - Attributs = propriétés
 - Modèle, joueur, base, taille, ...
 - Opérations = manipulations possibles
 - Démarrer, accélérer, remplacer, calculer l'hypoténuse, ...
 - Prototype : un exemplaire de la classe

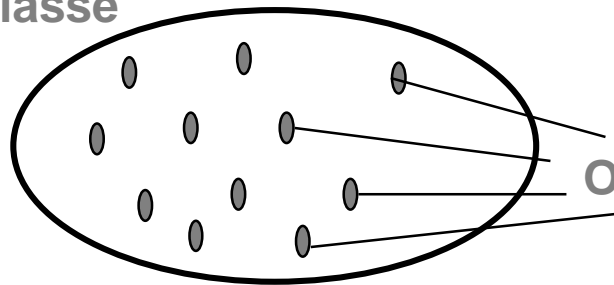
Classe : Ensemble

- L'ensemble des représentants : Définition ensembliste

VOITURE = { RENAULT5 921 MA 92, JAGUAR 769 KZ 75,
PLYMOUTH CAL ZZ 444, ... }

HOMME RICHE = { $x \in \text{HOMME} / \text{fortune}(x) > 100.000.000.000 \text{ €}$ }

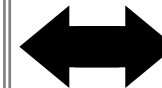
Classe



Objets représentés

Règle : pour tout objet de la classe, chaque opération est applicable, et chaque attribut a un sens.

HOMME
Taille Salaire
Manger() ParlerRusse()



Classe : Programmation

- Une structure physique : Programmation

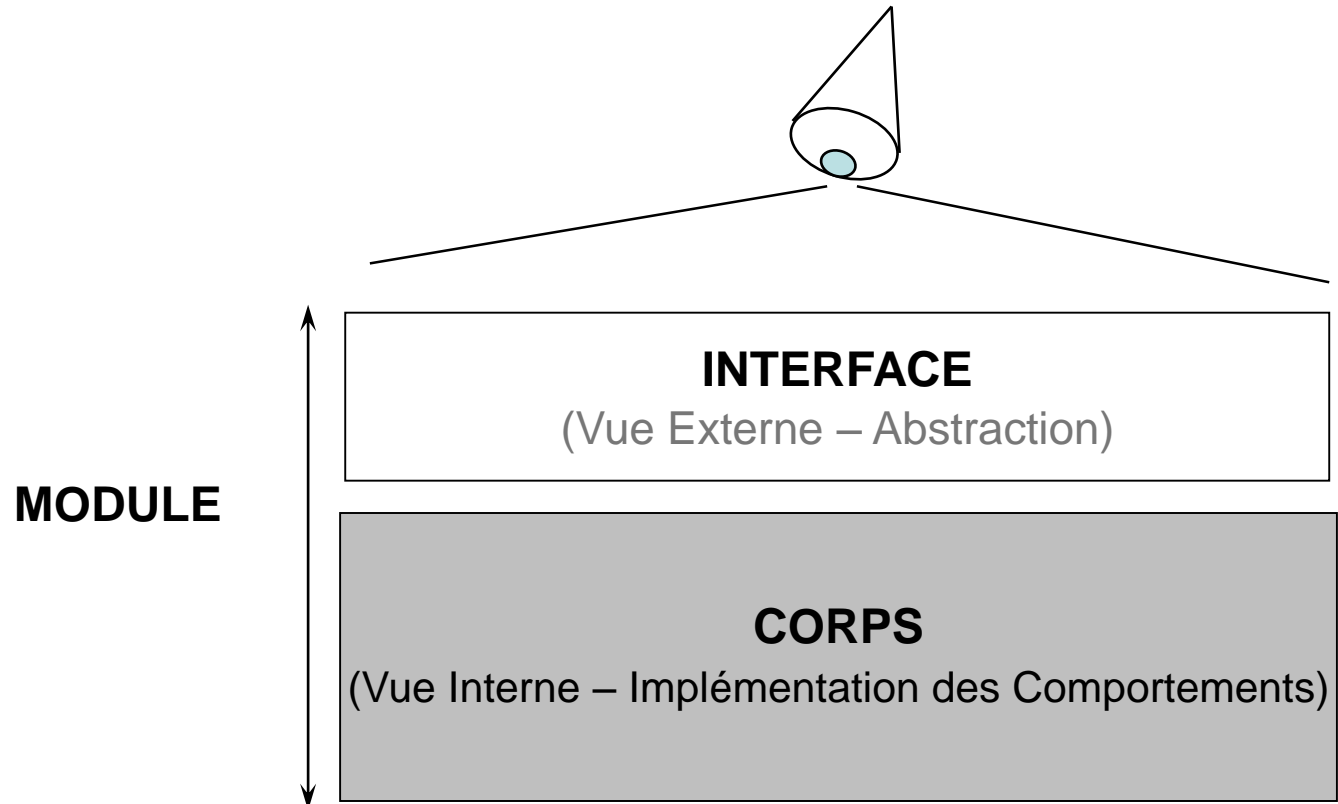
```
class Voiture
{
    // Déclarations des opérations
    void demarrer ();
    void debrayer ();
    void embrayer ();
    void contact ();
    void changervitesse (nouvellevitesse);
    void accélérer ();

    // Déclaration des attributs
    Couleur couleur;
    int NumeroVitesse;
};
```

```
void Voiture::demarrer ()
{
    contact ( );
    debrayer ( );
    changervitesse (1 );
    accélérer ( );
    embrayer ( );
}
```

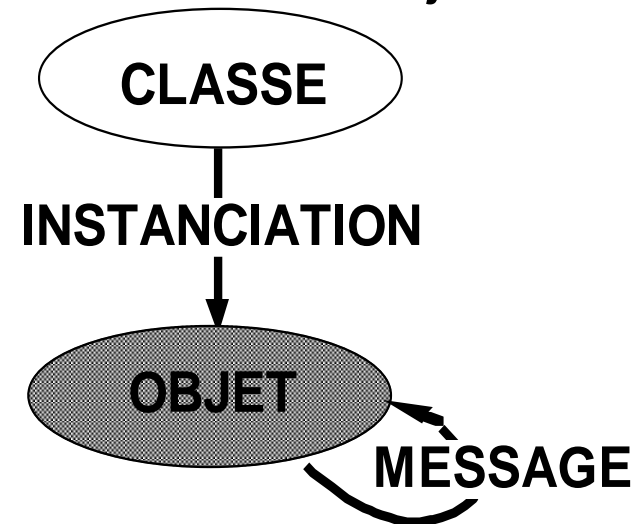
Classe : Modules

- Structuration des classes en module
 - Module = Interface + Corps
 - Module = Données + Traitements



Synthèse

- Une classe d'objets
 - Est la représentation informatique d'un concept
 - Se caractérise par ses attributs et ses méthodes
- Un objet est un élément autonome, représentant d'une classe, qui réagit à des messages
- Mécanismes de base de la programmation objet
 - L'héritage
 - L'instanciation
 - L'envoi de messages

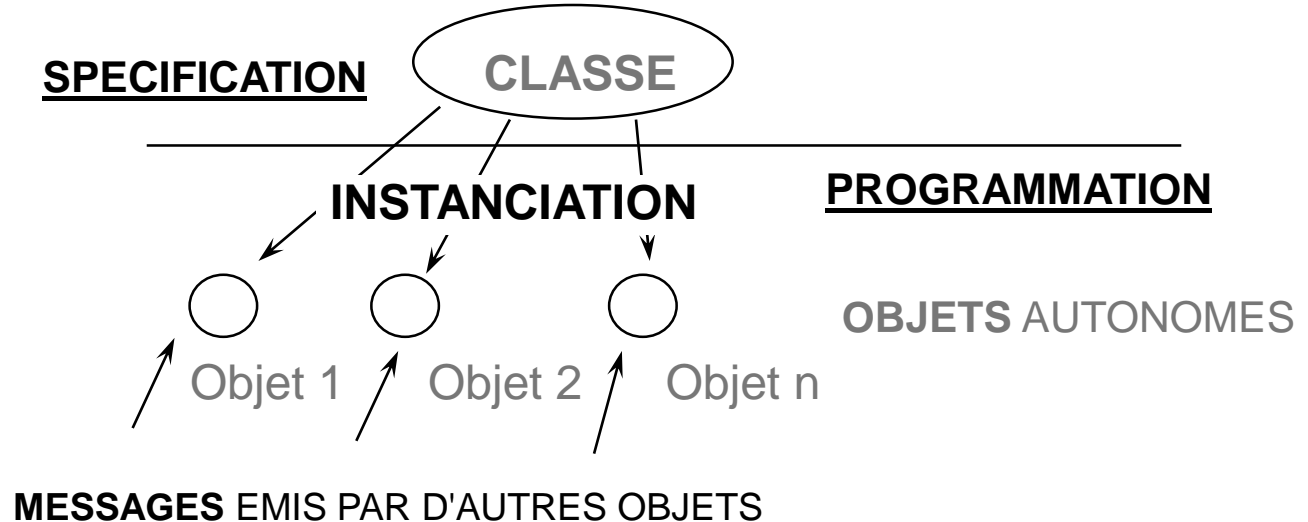


Concepts et mécanismes fondamentaux de C++

**Création, suppression et
initialisation d'objets**

Instanciación

- Consiste en la creación d'objets à partir d'une classe
- Les représentants créés sont des Objets ou Instances
- Un objet est un élément autonome qui réagit à des messages



- Une application est un réseau d'objets interagissant entre eux

Instanciación et Objets (1)

- Chaque objet a ses comportements propres et gère son espace de données

```
struct Personne  
{  
    int age;  
};
```

```
struct Personne  
    anInstance;
```

...

anInstance

```
anInstance.age = 4;
```

4

anInstance

```
class Personne  
{  
    private:  
        int age;  
    public:  
        void setAge(int);  
};
```

```
Personne anInstance;
```

...

anInstance

```
anInstance.setAge(4);
```

4

anInstance

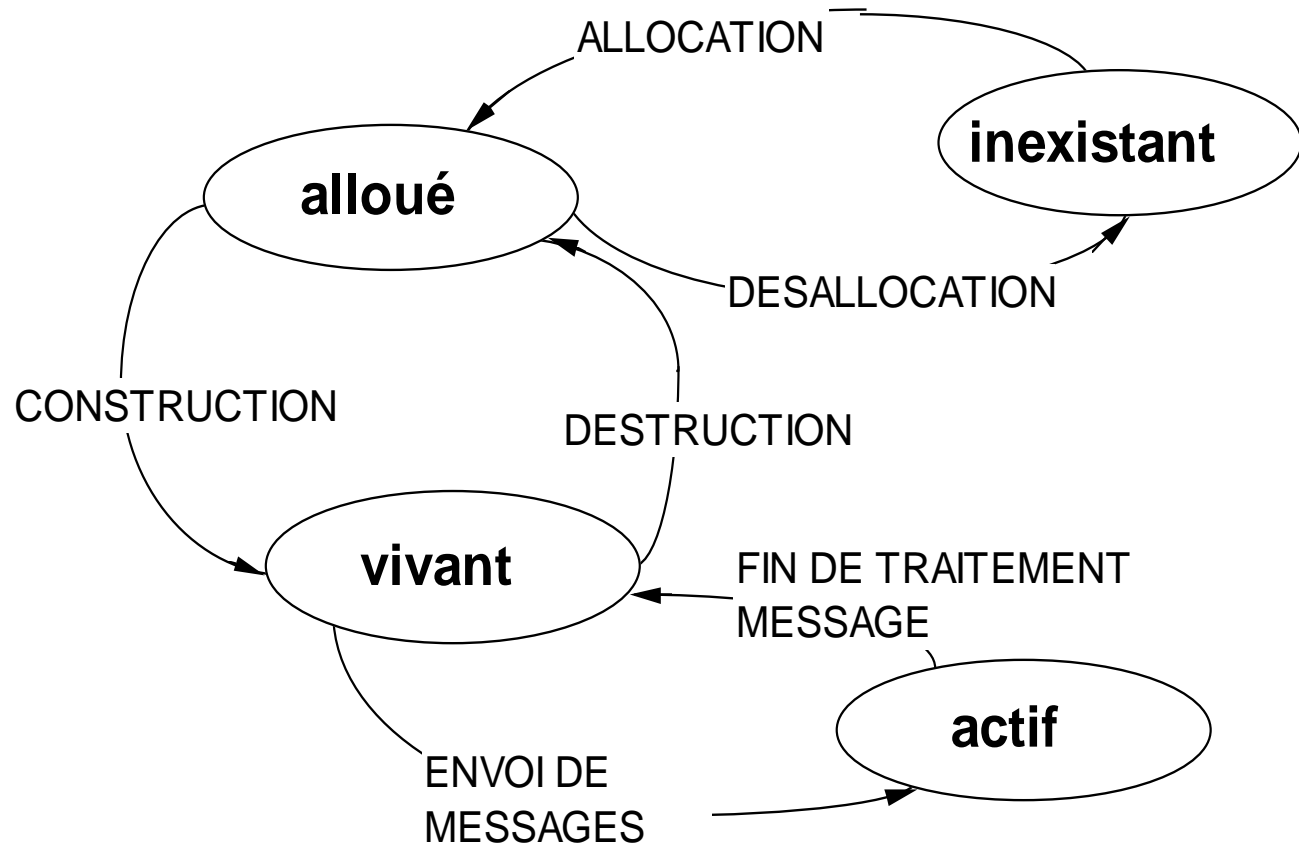
Instanciation et Objets (2)

- L'instanciation se déroule en plusieurs étapes
 - Déclarer un représentant d'une classe
+
Créer ce représentant (par déclenchement du constructeur)
 - Donner existence à un élément (Objet) selon un modèle
+
Particulariser cet élément
- En C++ :

Voiture ma_voiture(mauvaise_tenue_route, demarrage_aleatoire);

classe **objet** **particularités**

Cycle de Vie des Objets (1)



Cycle de Vie des Objets (2)

- Les grandes étapes de manipulation des objets :

1) Création : appel implicite au constructeur

```
Voiture ma_voiture (  
    mauvaise_tenue_route, demarrage_aleatoire);
```

2) Construction (appel implicite du constructeur)

```
Voiture::Voiture (  
    tenue_mauvaise_tenue_route, depart  
    demarrage_aleatoire);
```

3) Activation : objet + ('.' ou '->') + méthode et paramètres

```
Ma_voiture.Tourner(Direction);
```

4) Destruction : appel implicite au destructeur

```
Voiture::~~Voiture ();
```

Durée de Vie des Objets (1)

- Diffère selon les objets

- Objets Statiques (Modulaires)

- `nom_classe nom_objet (paramètres construction);`

- Construction automatique au début de l'application
 - Destruction automatique à la fin de l'application

- Objets Automatiques

- Construction automatique au début de la méthode
 - Destruction automatique à la fin de la méthode

- Objets Dynamiques

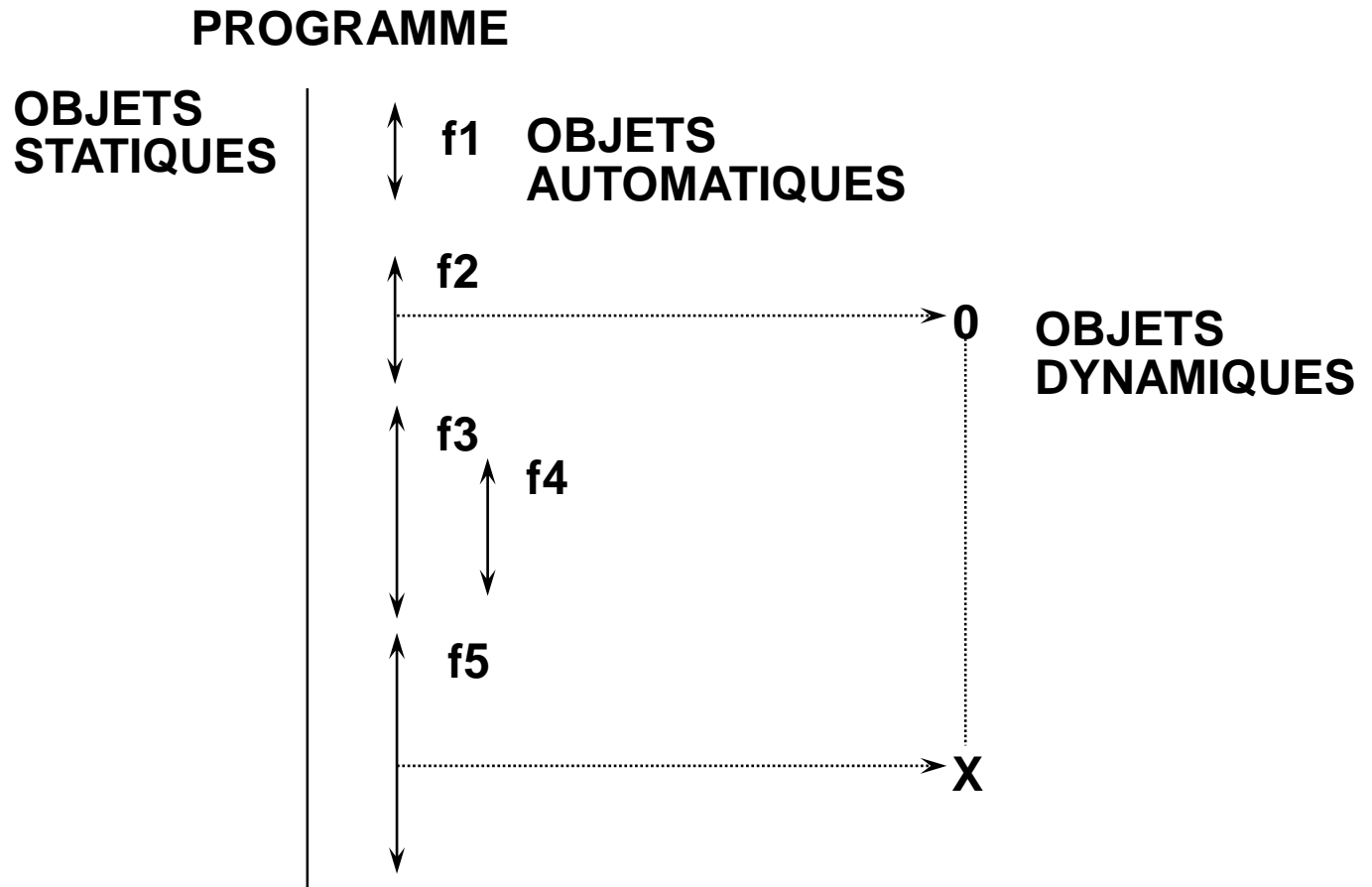
- `nom_classe * nom_objet;`

- `nom_objet = new nom_classe (paramètres construction);`

- `delete nom_objet;`

Durée de Vie des Objets (2)

- Emploi des objets statiques, automatiques ou dynamiques



Constructeur & Destructeur

- Méthodes spécifiques, appelées implicitement dont le nom se déduit de celui de la classe
- Constructeur (Instanciation)
 - Nom de la classe : `Nom_Classe(...);`
 - Peut prendre des paramètres de construction
 - Pas de type de retour (explicite)
 - Place l'objet dans sa condition initiale
 - Fournit les valeurs par défaut
- Destructeur
 - "~" + Nom de la classe : `~Nom_Classe();`
 - Pas de type de retour, Ni de paramètre
 - Un seul destructeur par classe
 - Supprime les éléments dépendants (libère la mémoire utilisée)
 - Place le système dans un état cohérent après la disparition de l'objet

Constructeurs Spécifiques

- Constructeur par défaut
 - Ne prend pas d'argument
 - Implicite, si aucun constructeur n'est défini
 - Ne fait rien (les attributs membres ne sont pas initialisés)
 - *A utiliser avec parcimonie*
 - Constructeur sans argument
 - Permet d'initialiser les attributs membres à des valeurs par défaut
 - *Utilisation recommandée*
- Constructeur de copie
 - Appelé pour toute initialisation d'un objet à partir d'un autre objet de même type
 - Prend en paramètre une référence sur une instance de la classe
`Nom_Classe(const Nom_Classe &);`

Allocation Mémoire

- Deux modes d'allocation mémoire en C++ :
 - Allocation dynamique (dans le tas)
 - Allocation statique (dans la pile)
- Allocation dynamique :
 - **new** & **delete**
 - malloc & free : obsolètes, utilisation déconseillée
- Allocation statique
 - Variables locales ou statiques

Allocation Mémoire & Durée de Vie l'Objet

- La durée de vie de l'objet dépend du mode d'allocation
 - Allocation dynamique **new** \Rightarrow **delete**
 - Allocation statique **variable locale** \Rightarrow **sortie de la fonction**
 - Allocation statique **variable statique** \Rightarrow **sortie du programme**

Opérateurs new & delete

- Agissent en 2 temps
 - new
 - Réserve la place mémoire nécessaire à l'objet dans le tas ;
 - Puis appelle un constructeur.
 - delete
 - Inversement, appelle d'abord le destructeur;
 - Puis libère la place mémoire.
- La syntaxe de l'opérateur "new" permet de préciser (par le type et le nombre de paramètres) le constructeur à utiliser pour l'initialisation de l'objet

```
exemple *pe1 = new exemple(1, 2);  
*pa = new autre(1); // appel de autre::autre(1)  
classexmpl *c1 = new classexmpl; // constructeur par défaut  
classexmpl *c2 = new classexmpl(*c1); // constructeur de copie
```

Exemple d'utilisation

```
1  #include <iostream>
2  using namespace std;
3  class point{
4      private:
5          int x,y;
6      public:
7          point(int coordx,int coordy):x(coordx),y(coordy){}
8          void plot(){
9              cout<<"Dessine point (x="<<x<<" ,y="<<y<<" )"<<endl;}
10         ~point(){cout<<"destructeur de point"<<endl;}
11     };
12     int main(int argc,char** argv){
13         point p(1,2);
14         point* p2= new point(2,4);
15         p.plot();
16         p2->plot();
17         delete p2;
18     }
```

Allocation Mémoire & Tableaux (1)

- Allocation statique de tableaux d'objets de même type
 - Taille définie à la déclaration

```
class CFoo
{
    long m_Val;
public:
    CFoo (long Val = 0): m_Val(Val) {};
    ~CFoo (){};
};

void f()
{
    CFoo Tab[3] = { 5, 6 };
    // Tab[0].m_Val = 5; Tab[1].m_Val = 6; Tab[2].m_Val = 0
    ...
} // Détruit les 3 CFoo avec 3 appels au destructeur
```

Allocation Mémoire & Tableaux

- Allocation dynamique de tableaux d'objets de même type
 - Permet de spécifier dynamiquement la taille du tableau
 - Repose sur les opérateurs "new[]" et "delete[]"
 - Utilise le constructeur par défaut

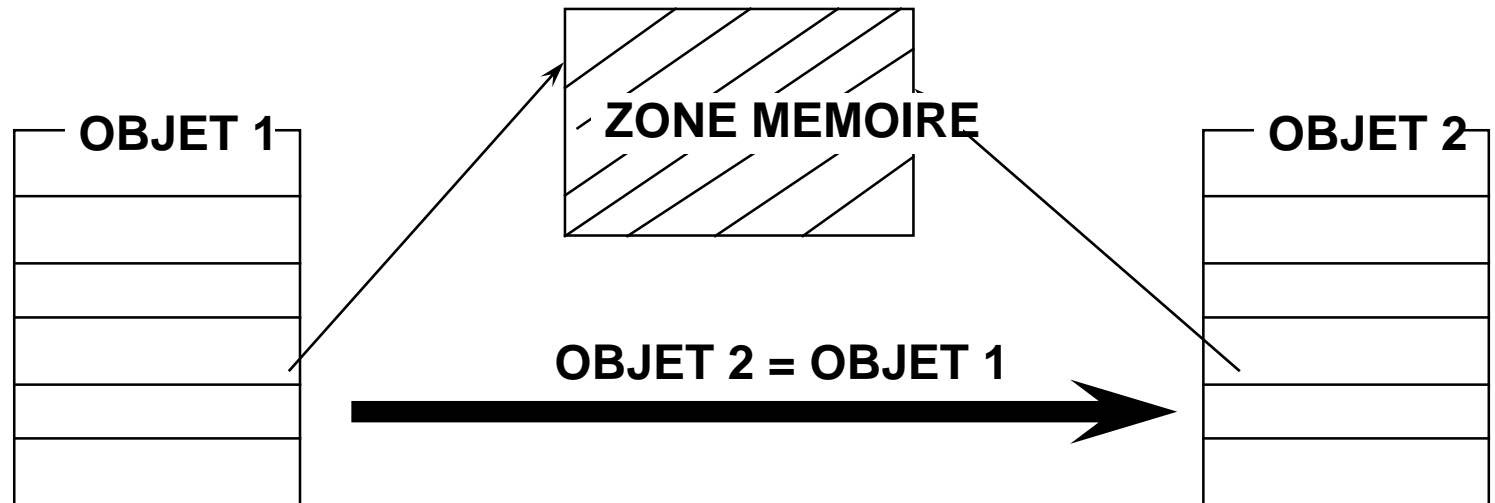
```
int *tab = new int[dim];  
delete []tab;
```

Concepts et mécanismes fondamentaux de C++

Copie d'objets

Affectation d'Objets

- C++ 1.2 : Affectation Bit à Bit
C++ 2.0 : Affectation Membre à Membre
- Surcharger l'opérateur d'affectation = Éviter les effets de bord



Affectation Membre A Membre

- La surcharge de l'opérateur d'affectation est prise en compte par les classes englobantes

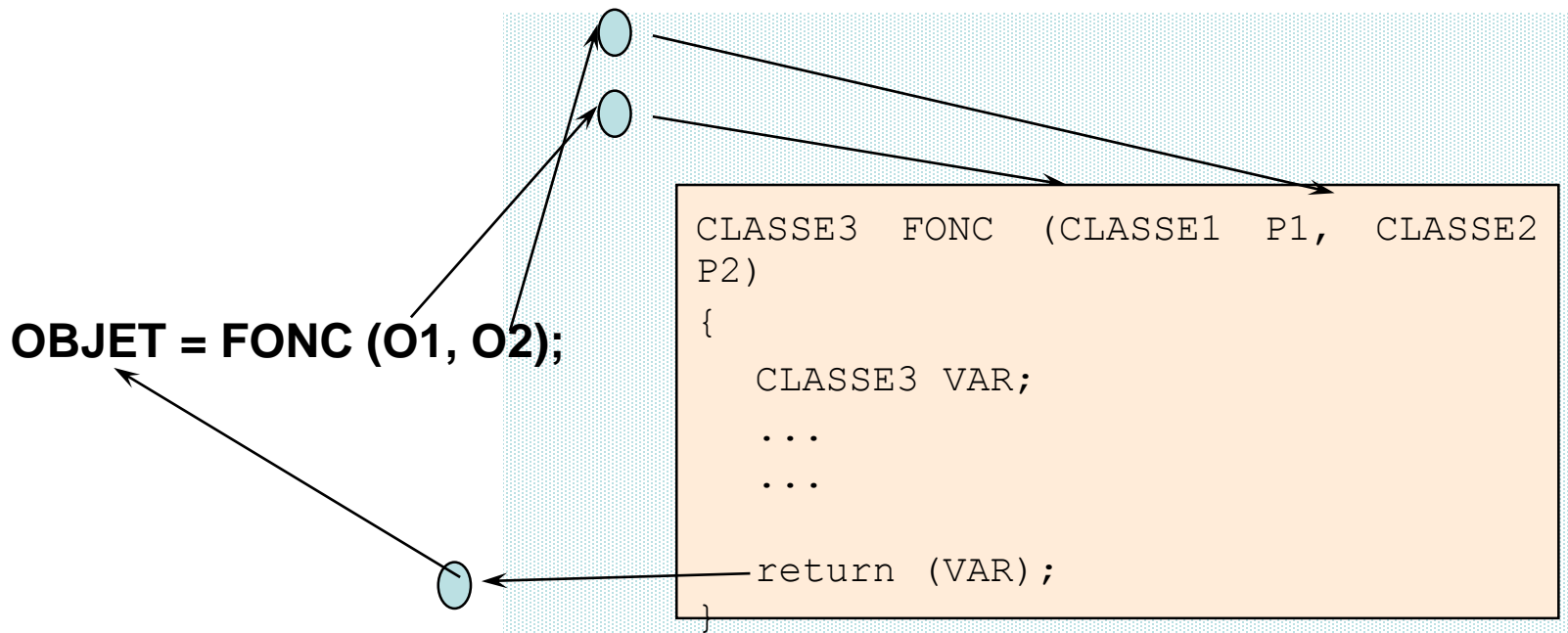
```
class Classe1
{
    operator = (Classe1);
    ...
};

class Classe2
{
    Classe1 m_ClasseContenue;
    ...
};

Classe2  emetteur, receveur;
receveur = emetteur;
```

Objets Temporaires

- Le fonctionnement d'un programme C++ impose la création fréquente d'objets temporaires



Objets Temporaires

- Il est possible d'indiquer au compilateur le mode de construction des objets temporaires

```
class Classe1
{
    ...
    Classe1 operator+ (const Classe1&);
    ...
};

Classe1 obj1, obj2;
FONC (&(obj1 + obj2)); // ??
```

Objets temporaires : initialisation

- Déclaration d'un constructeur spécifique

```
1) Classe1 obj1 = obj2;  
2) FONC (Classe1 param);  
   FONC (obj1);  
3) Classe1 FONC ()  
   {  
       ...  
       return (unObjetDeClasse1);  
   };  
obj2 = FONC ();
```

- Pas d'appel aux constructeurs de "Classe1"
- SAUF si "Classe1::Classe1 (const Classe1 &)"
- La classe est alors responsable des instanciations

Objets temporaires : T&&

- C++ 11 offre un nouveau type de référence: la rvalue reference avec l'opérateur &&

```
/** C++ 2003 */  
// appelle std::string()  
std::string s;  
// appelle std::string(const std::string&)  
std::string another(s);  
// appelle deux fois std::string(const std::string&)  
std::string more(std::string(s));  
  
/** C++ 2011 */  
// appelle d'abord std::string(const std::string&)  
// puis std::string(std::string&&);  
std::string more(std::string(s));
```

- Evite l'allocation d'un objet temporaire

Tableaux d'Objets : Mode d'Initialisation

```
class Chaine
{
    Chaine(char* chaine_c);
    Chaine(int taille);
    Chaine(const Chaine& chaine_modele);
    Chaine();
};

Chaine chaineParticuliere;

Chaine lesChaines [20] =
    {"bonjour", 10, Chaine(), chaineParticuliere};

Chaine *lesChaines2 = new Chaine [20];
delete [] lesChaines2;
```

Classe Élémentaire

- Deux familles de classes existent :
 - Classe Élémentaire
 - Classe Non Élémentaire

```
int i;  
int j;  
int k;           //Variable Intermédiaire  
  
k = j;  
j = i;  
i = k;
```

OK

```
Homme martin;  
Homme dupond;  
Homme durand;    //Variable Intermédiaire  
  
durand = dupond;  
dupond = martin;  
martin = durand;
```

?

Classe Élémentaire & Règles d'Utilisation

- La nature de l'objet, de classe élémentaire ou non, détermine son mode d'utilisation
- Il est possible de copier et d'avoir des doublons d'objets de classe élémentaire
 - Objets temporaires
 - Passage par valeur
 - Attributs
- Il est peu recommandé de dupliquer les objets de classe Non élémentaire
 - Jamais en passage par valeur ou attribut
 - Préférer le passage par référence
 - Emploi systématique de "relations" (pointeurs)

Exceptions

Exemple introductif

```
1  #include <iostream>
2  using namespace std;
3  float divide_by_int(int a,int b){
4      return a/b;
5  }
6  int main(int argc,char** argv){
7      int a;
8      int b;
9      float result;
10     cout<<"calcul d'une division entre 2 entiers"<<endl;
11     cout<<"Saisie du numerateur(a)";
12     cin>>a;
13     cout<<"\nSaisie du denominateur(b)";
14     cin>>b;
15     result = a/b;
16     cout<<"\nResultat de la division : " << result << endl;
17 }
```

Commentaires sur l'exemple précédent

- Que se passe t'il si le dénominateur (b) est nul ?

```
Fichier Éditer Affichage Terminal Aller Aide
[1] 6277
xen-server% kbuildsysoca running...
Reusing existing ksycoca
Launched ok, pid = 6310

xen-server% cd work/cpp_training/exceptions/
xen-server% ls
a.out introductory.cpp
xen-server% ./a.out
calcul d'une division entre 2 entiers
Saisie du numerateur(a)12

Saisie du denominateur(b)3

Resultat de la division : 4
xen-server% kbuildsysoca running...
Reusing existing ksycoca

xen-server% ./a.out
calcul d'une division entre 2 entiers
Saisie du numerateur(a)12

Saisie du denominateur(b)0
zsh: floating point exception ./a.out
xen-server%
```

Améliorations possibles

- Il est envisageable de mieux gérer ce cas particulier en :
 - Renvoyant un code d'erreur ?
 - Quel type float ou bool ?
 - bool serait plus correct...
 - Renvoyant un message ?
 - Comment s'interfacer avec notre programme depuis une GUI ?
 - Comment rendre possible les appels du type : $(a/(b/c))$?

Rôle d'une exception ?

- Modéliser un cas d'erreur induisant une impossibilité de traitement lors d'un appel à une méthode (fonction membre)
- Éviter la gestion de codes d'erreurs à la mode C
 - Peu portables (mêmes codes pour erreurs différentes suivant les systèmes)
- Donner un aspect objet à ce qui est un cas d'utilisation de votre code (objet)...

Exemples d'exceptions

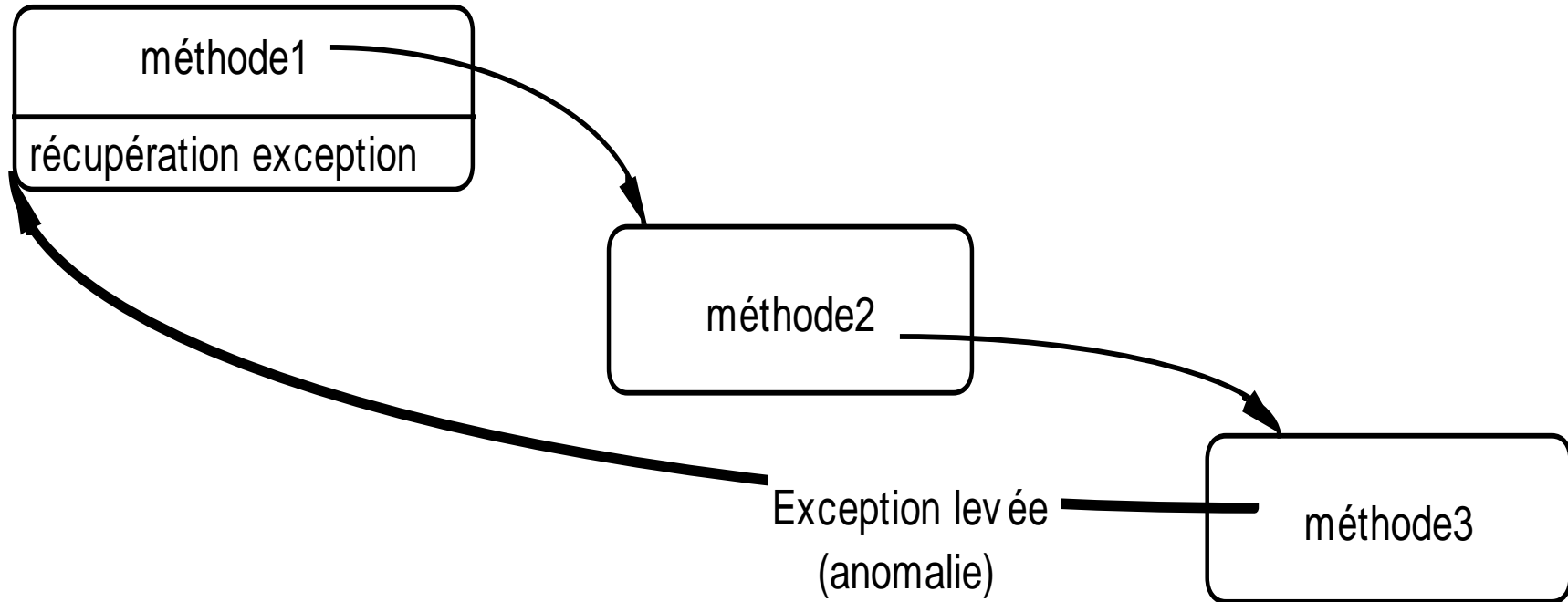
- Exceptions techniques
 - Division par zéro
 - Traitement de racines carrées négatives dans le cas standard des réels (pas des complexes)
 - Écriture d'un fichier avec des droits insuffisants
 - Contacter un serveur indisponible
- Exceptions fonctionnelles
 - Solde bancaire insuffisant lors d'une transaction

C++: Gestion des exceptions

- Syntaxe de base
 - Lancer et récupérer les exceptions
 - `try`,
 - `catch`,
 - `throw`
 - Relancer une exception
- Principes avancés
 - Traitement par défaut et fonction `std::terminate`
 - Exceptions autorisées pour une fonction, fonction `std::unexpected`
 - Hiérarchie d'exceptions
 - Exception des constructeurs
- Exceptions standard

Principe

- Remonter une anomalie dans le graphe d'appel
- Traitement obligatoire sous peine d'arrêt du programme



Exception – Flot de contrôle

- Le lancement d'une exception déroute le flot de contrôle.
 - il est impossible de reprendre l'exécution là ou elle a été interrompue
- Bloc try détermine la portée des blocs catch
- Les blocs catch doivent être correctement ordonnés
- Possibilité de capturer n'importe quel type d'exception en utilisant les "ellipses" :
 - catch (...)
 - On peut relancer avec un simple throw

Exception – Flot de contrôle

- Si pas de catch activés :
 - Utilisation de la fonction `std::terminate`
 - Correspond à un `abort()`
- Possibilité de modifier `std::terminate` :
 - Utilisation de `std::set_terminate()`
 - Paramètre : pointeur de fonction sans paramètre et retournant `void`

Mots Clés

- "try", "catch" & "throw"

```
class Plus_de_mémoire { ... };  
plus_de_mémoire P;  
f ( ) {  
    try { ...  
    }  
    catch (plus_de_mémoire x) { ...  
    }  
    catch (autre_chose) { ... }  
    catch (...) { ... }  
}  
throw ( P ) ;
```

Exemple : Déclaration Des Classes

```
#include <string>

class ErreurInsertion // La classe d'exception
{
    public:
        ErreurInsertion(const char* message): _message(message){};
        std::string _message ;
};

class MonTableau // Une classe utilisant les exceptions
{
    public:
        MonTableau();
        void addElement(int i) throw (ErreurInsertion);
    private:
        int _tab[10];
};
```

Exemple : Définition Des Classes

```
MonTableau::MonTableau() {
    for(int i = 0; i < 10 ; i++) _tab[i] = -1 ;
}

void MonTableau::addElement(int element) // throw (ErreurInsertion)
{
    int i = 0;
    bool inserted = false ;
    while((!inserted) && (i < 10)) {
        if (_tab[i] == -1 ) {
            _tab[i] = element;
            inserted = true;
        }
        ++i;
    }
    if (inserted == false) {
        ErreurInsertion e("Impossible d'ajouter element");
        throw(e);
    }
}
```

Exemple : Appel Des Méthodes

```
int main(int argc, char* argv[])
{
    MonTableau mt ;
    for(int i=0 ; i< 11 ; i++) {
        try {
            mt.addElement(i);
        }
        catch(ErreurInsertion e) {
            cout << e._message.c_str() << endl ;
        }
    }
}
```

Exception & Type d'allocation

- Instances d'exceptions à lancer peuvent être créées
 - dans le tas : `throw new Exception(...);`
 - Attention à la mémoire
 - dans la pile (objet local) : `throw Exception(...);`
 - le bloc **catch** va recevoir une copie de l'exception, à un emplacement garanti accessible.
- Catch par valeur ou par référence ?
 - `catch (Exception const & e)` ou `catch (Exception e)` ?
- Règles :
 - Ne pas stocker dans les exceptions, des références vers des objets susceptibles de ne plus exister
 - « Throw by value, catch by reference »

Définition d'un gestionnaire d'exception

```
#include <iostream>
#include <exception>

using namespace std;

void mon_gestionnaire(void)
{
    cout << "Exception non gérée reçue !" << endl;
    cout << "Je termine le programme proprement..."
        << endl;
    exit(-1);
}

int lance_exception(void)
{
    throw 2;
}

int main(void)
{
    set_terminate(&mon_gestionnaire);
    try
    {
        lance_exception();
    }
    catch (double d)
    {
        cout << "Exception de type double reçue : " <<
            d << endl;
    }
    return 0;
}
```

Exceptions

Exceptions dans un constructeur

Exception & Constructeur

- Le corps d'une méthode peut-être un bloc try catch
 - Nécessaire pour attraper les exceptions lancées par les évaluations des expressions des listes d'initialisations de constructeurs.

```
MonTableau::MonTableau() // constructeur
try
{
: maVariable(expr1), monObject(expr2), ...
    {
        // Corps constructeur
    }
}
catch (...)
{
}
```

Attention !

- Cas d'une exception survenant lors de la construction d'un objet :
 - objet non créé donc jamais détruit ! ! !
 - désallocation explicite à faire dans le bloc catch

Exemple

```
// Constructeur susceptible de lancer une exception :  
A::A() throw (int)  
{  
    try  
    {  
        pBuffer = NULL;  
        pData = NULL;  
        cout << "Début du constructeur" << endl;  
        pBuffer = new char[256];  
        cout << "Lancement de l'exception" << endl;  
        throw 2;  
        // Code inaccessible :  
        pData = new int;  
    }  
    catch (int)  
    {  
        cout << "Je fais le ménage..." << endl;  
        delete[] pBuffer;  
        delete pData;  
    }  
}
```

Exception & Destructeur

- Règle simple :
 - NE JAMAIS LANCER D'EXCEPTION DANS UN DESTRUCTEUR

Exceptions

Listes d'exceptions levées

Liste des exceptions autorisées pour une fonction

- Spécifier les exceptions qui peuvent être lancées par une fonction :

```
int fonction_sensible(void)
throw (int, double, erreur)
{
    ...
}
```

- Fonction_sensible n'a le droit de lancer que des exceptions du type **int**, **double** ou **erreur**.
- Si une exception d'un autre type est lancée, il se produit encore une fois une erreur à l'exécution.
 - En fait, la fonction `std::unexpected` est appelée.

std::unexpected

- Identique à `std::terminate` (par défaut)
- Modifiable par appel de : `std::set_unexpected`
 - Paramètre : pointeur de fonction « void » sans paramètre
 - Retourne la fonction utilisée précédemment en tant que `std::unexpected`.
- Possibilité de relancer une autre exception à l'intérieur du code de traitement d'erreur
 - Si exception connue (« catchée ») : traitement habituel
- Possibilité de lancer l'exception `std::bad_exception`

Exceptions

**Organisation d'exceptions dans
une hiérarchie de classes**

Exception & Héritage

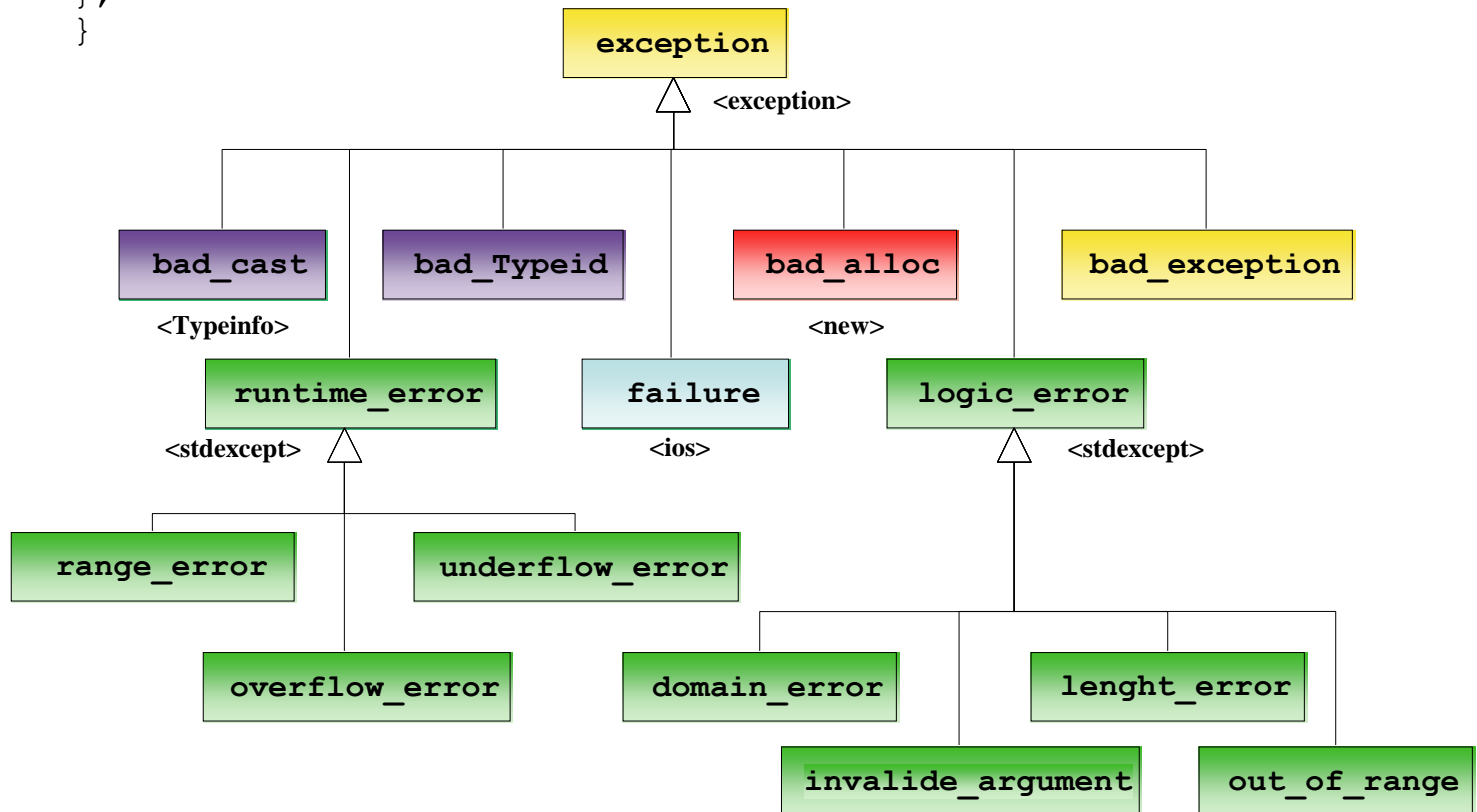
- Les exceptions sont généralement regroupées en hiérarchie de classes
- L'héritage se combine avec les exceptions

```
class PB { } ;  
class plus_de_mémoire : public PB { } ;  
class File_System_Full : public PB { } ;  
  
f ( ) {  
    try { ... }  
    catch (PB) { ... }  
}
```

- **STL** propose une hiérarchie
 - il est préconisé de l'utiliser

Exceptions Standards

```
// header exception
namespace std {
class exception
{
public:
    exception() throw();
    exception(const exception &) throw();
    exception &operator=(const exception &) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
}
```



Exceptions

Sécurité des exceptions

(l'idiome de programmation "Resource Acquisition Is Initialization")

Définition « Resource Acquisition Is Initialization »

- **RAII** est l'acronyme anglais
 - « l'Acquisition d'une Ressource est une Initialisation »
- Technique de programmation créée par B. Stroustrup afin de sécuriser l'utilisation d'une ressource
- Utilisation d'une classe spécifique « sécurisée »
 - Constructeur = acquisition de la ressource
 - Destructeur = libération de la ressource
- Si exception = appel du destructeur
- Pour des raisons de respect du Principe de Responsabilité Unique (SRP), une classe RAII ne doit rien faire d'autre que de gérer une ressource!

Exemple

```
class FichierSecurise {  
    Fichier fichier_interne;  
public:  
    FichierSecurise(Fichier f) : fichier_interne(f) {  
        fichier_interne.open();  
    }  
    ~FichierSecurise() {  
        fichier_interne.close();  
    }  
};
```

Le RAI et la STL

- En pratique, écrire une classe RAI demande un peu plus de code. Il faut en effet :
 - respecter la sémantique de valeur,
 - ajouter les constructeurs par copie,
 - par déplacement,
 - les opérateurs d'affectation,
 - etc.
- Peut-être également utiliser des abstractions un peu plus évoluées (template) pour avoir un code plus générique.

Le RAII et la STL

- Heureusement, une grande partie du travail peut être simplifiée, puisque la bibliothèque standard (STL) utilise le RAII et propose de nombreuses fonctionnalités pour la gestion des ressources.
- Ainsi, en C++, on évitera d'utiliser les syntaxes héritées du C, mais on utilisera (et abusera) des classes de la STL.
Par exemple :

Fonctionnalité	En C	En C++
Créer une chaîne de caractères	<code>char* s;</code>	<code>std::string s;</code>
Créer un tableau de données	<code>Type* v;</code>	<code>std::vector<Type> v;</code>
Créer un fichier	<code>File f;</code>	<code>std::ifstream f;</code>
Créer un objet sur le Tas	<code>Object* o;</code>	<code>std::unique_ptr<Object> p;</code>
		<code>std::shared_ptr<Object> p;</code>
Verrouiller un mutex	?	<code>std::lock_guard<std::mutex> l;</code>

Smart Pointers

« delete c'est mal ! »

Smart pointeur

- Problématique des pointeurs
 - Allocation et surtout destruction des pointeurs lorsqu'ils ne sont plus utilisés
 - Source de bug, de fuite mémoire
- Smart pointeur:
 - Propose des fonctionnalités de libération automatique de la mémoire
 - Respecte RAII
 - Adapté aux exceptions

auto_ptr (C++03)

- Classe de gestion automatique de la mémoire
- Pointeur dynamiquement alloué par l'opérateur "new"
- Permet d'accéder aux méthodes de l'objet pointé grâce à la surcharge de l'opérateur "->"
- A sa destruction un auto_ptr propriétaire appelle l'opérateur "delete" sur l'objet pointé
- Au cours d'une copie la propriété de l'auto_ptr source est transférée à celui de destination (que ce soit par copie ou initialisation)

```
auto_ptr<Animal> p1 ( new Chien ("Lassie") );  
auto_ptr<Animal> p2 = p1;  
  
p1->afficher();  
// à la fin, p2 (le propriétaire) détruit l'objet
```

auto_ptr C++03

Exemple:

```
{  
// Alloue dynamiquement un objet :  
auto_ptr<A> p(new A);  
// Lance une exception, en laissant au pointeur  
// automatique le soin de détruire l'objet alloué :  
throw 2;  
}
```

Pièges à éviter

- éviter l'utilisation en paramètre de fonction
- ne pas initialiser avec l'adresse d'un objet qui n'a pas été alloué dynamiquement
- ne pas initialiser avec un pointeur obtenu par new[] ou malloc()
- Deprecated en C++11 au profit de `unique_ptr<T>`

unique_ptr (C++11 ou Boost)

- Semblable à auto_ptr:
 - Classe de gestion automatique de la mémoire
 - Pointeur dynamiquement alloué par l'opérateur "new"
 - Permet d'accéder aux méthodes de l'objet pointé grâce à la surcharge de l'opérateur "->"
 - A sa destruction appelle l'opérateur "delete" sur l'objet pointé
- Transfert explicite de la propriété de l'objet pointé, via std::move()

```
unique_ptr<Product> p1 ( new Product("Velo") );  
  
unique_ptr<Product> p2 = p1; // erreur  
  
// Transfert l'ownership à p3  
unique_ptr<Product> p3 = std::move(p1);  
// maintenant p1 est nullptr
```

unique_ptr (C++11 ou Boost)

- Overhead négligeable
 - Fonctionne aussi avec new []
 - Appelle delete [] à la destruction
 - Fournit l'opérateur [] au lieu de -> et de *
 - Remplace avantageusement scoped_ptr et auto_ptr
 - A utiliser sans parcimonie !
- ➔ Sémantique : un unique propriétaire de l'objet pointé (le propriétaire est le responsable de la destruction)

- Principe
 - Définit un pointeur sur un objet donné
 - Gère un compteur de référence
 - A sa destruction le compteur est décrémenté et si le compteur passe à 0 l'objet pointé est détruit
 - Lors de la copie ou de l'affectation le compteur est incrémenté
 - La responsabilité de la destruction est partagé par tous les shared_ptr
- ➔ Sémantique : plusieurs propriétaires d'un objet

shared_ptr - Exemple

```
void test() {  
    shared_ptr<int> a(new int (42));  
    cout << a.use_count() << endl;  
    shared_ptr<int> b = a;  
    cout << a.use_count() << endl;  
    { shared_ptr<int> c;  
      shared_ptr<int> d (new int (314));  
      cout << a.use_count() << endl;  
      c = a;  
      cout << a.use_count() << endl;  
      d = c;  
      cout << a.use_count() << endl;  
    }  
    cout << a.use_count() << endl;  
    b.reset();  
    cout << a.use_count() << endl;  
}
```

shared_ptr - inconvénients

- Références circulaires : jamais libérés
 - Il faut manuellement briser la chaîne
- Plus coûteux que unique_ptr :
 - Espace mémoire supplémentaire pour le compteur
 - Chaque copie (même temporaire) incrémente/décrémente le compteur
 - Le compteur est thread-safe
 - Utilisation d'un mutex ou d'instructions CPU atomiques → ralentit l'exécution

weak_ptr

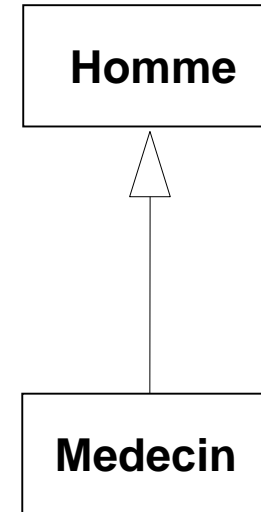
- Utilisation conjointe avec les shared_ptr dans le cadre de cycle
- N'impacte pas le compteur de référence
- Permet de savoir si l'objet pointé est toujours vivant avant d'y accéder
 - Appel à lock(), qui retourne un shared_ptr ou nullptr

Run Time Type Information (RTTI)

C++: RTTI

- Problématique:

```
class Homme {};  
class Medecin : public Homme {};  
  
Homme* unHomme= new Medecin ();
```



- Comment connaître lors de l'exécution le type dynamique du pointeur **unHomme** ?

```
Homme* quidam = new Medecin;  
(Medecin*) quidam->soigner(); // ??
```

Run Time Type Information (RTTI)

Les opérateurs 'cast' en C++

RTTI - Opérateurs de cast

- Basés sur les templates
- Offrent plus de contrôles que les casts C
- 4 opérateurs:
 - **Transtypage dynamique (dynamic_cast)**
 - **Transtypage statique (static_cast)**
 - **const_cast**
 - **reinterpret_cast**
- Syntaxe générale
 - xxxxx_cast< TargetType >(pointeur ou référence)
 - $a = (A^*)\ b;$ \Rightarrow $a = \text{static_cast}\langle A^* \rangle(b);$

dynamic_cast : Exemple

```
class Homme {...};  
class Medecin : public Homme {  
    void soigner();  
};  
  
Homme* h = new Medecin() ;  
  
...  
h->soigner(); // refusé par le compilateur  
  
Medecin* m = dynamic_cast <Medecin*> (h);  
if (m !=0)    // le cast renvoie 0 s'il est impossible  
    m->soigner();
```


RTTI (2)

- **operator dynamic_cast**
 - Vérification dynamique des types à l'exécution
 - En cas d'erreur:
 - Les conversions de pointeurs retournent **NULL**
 - Les conversions de références lancent une exception **std::bad_cast**
- **operator static_cast**
 - Vérification à la compilation
 - attention, pas de vérification dynamique des types
 - les attributs de constance et de volatilité sont conservés
 - `static_cast<type> (expression)`
- **operator const_cast**
 - Supprime, à la compilation, le caractère **const** ou **volatile** des expressions
- **operator reinterpret_cast**
 - réinterprète les données d'un type en un autre type
 - aucune vérification à la compilation, aucune vérification à l'exécution
 - **reinterpret_cast< Medecin >(ordinateur)**

Run Time Type Information (RTTI)

Opérateur typeid()

Identification Dynamique des Types (RTTI)

- Class `type_info`

```
namespace std {  
    class type_info {  
    public:  
        int  operator==( const type_info&  rhs ) const;  
        int  operator!=( const type_info&  rhs ) const;  
        // nom  
        const char *  name() const;  
        // (aucune indication hiérarchique !)  
        int  before( const type_info&  rhs ) const;  
    };  
}
```

- opérateur `typeid`

`const type_info & typeid(expression);`

```
#include <typeinfo>  
  
class Homme {};  
  
class Medecin : public Homme {};
```

```
Homme* monPointeur= new Medecin ();  
const type_info & monType =  
    typeid(*monPointeur);  
cout << monType.name();
```

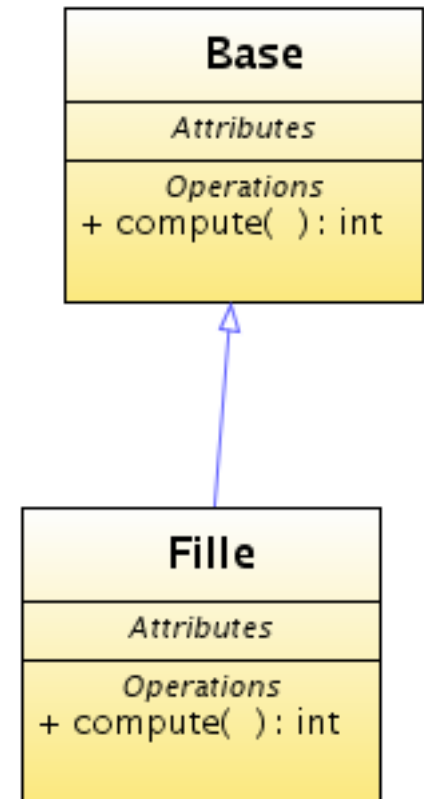
Mise en œuvre

```
1  #include <typeinfo>
2  #include <iostream>
3  using namespace std;
4  class Base{
5      public:
6          Base(){
7              cout<<"base.."<<endl;
8          }
9          virtual int compute(){
10             cout<<"Base::compute()"<<endl;
11         }
12     };

```

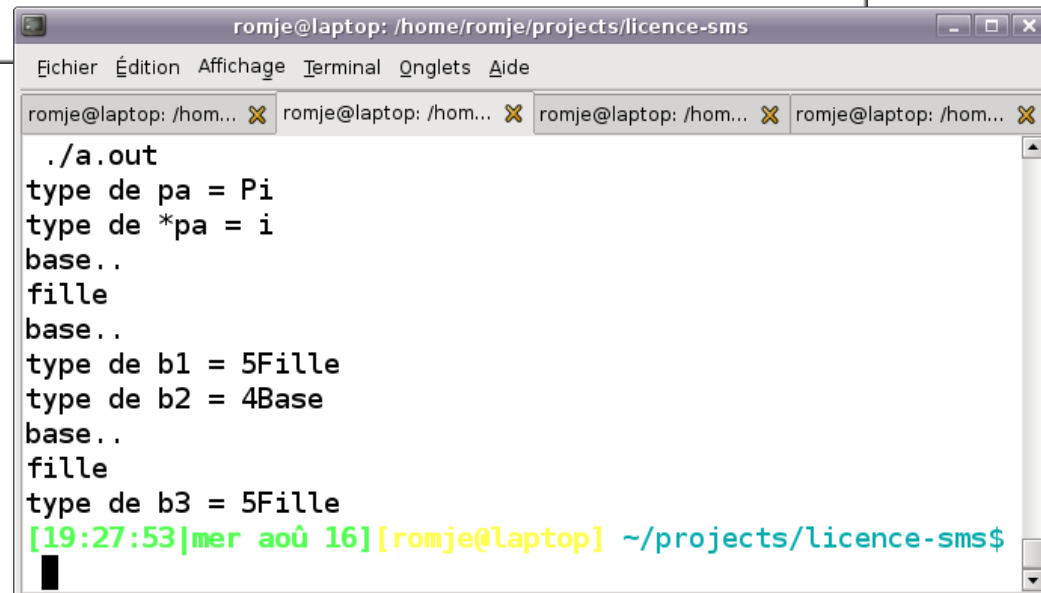
```
1  class Fille:public Base{
2      public:
3          Fille(){
4              cout<<"fille"<<endl;
5          }
6          int compute(){
7              cout<<"Fille::compute()" <<endl;
8          }
9      };

```



Mise en œuvre

```
1  int main(){
2      int a=2;
3      int* pa=&a;
4      float f=0.12f;
5      cout<<"type de pa = " <<typeid(pa).name()<<endl;
6      cout<<"type de *pa = " << typeid(*pa).name() << endl;
7      Base* b1=new Fille();
8      Base* b2=new Base();
9      cout<<"type de b1 = " << typeid(*b1).name() << endl;
10     cout<<"type de b2 = " << typeid(*b2).name() << endl;
11     Base *b3=new Fille();
12     cout<<"type de b3 = " << typeid(*b3).name() << endl;
13 }
```



The screenshot shows a terminal window titled "romje@laptop: /home/romje/projects/licence-sms". The window has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". Below the menu bar, there are four tabs, each showing "romje@laptop: /hom...". The main content area of the terminal displays the output of the program, which is the same as the code in the first block, but with the variable names replaced by their actual types: "Pi" for "int", "i" for "int*", "base.." for "Base", "fille" for "Fille", "5Fille" for "Fille*", and "4Base" for "Base*". The output is as follows:

```
./a.out
type de pa = Pi
type de *pa = i
base..
fille
base..
type de b1 = 5Fille
type de b2 = 4Base
base..
fille
type de b3 = 5Fille
[19:27:53][mer août 16][romje@laptop] ~/projects/licence-sms$
```

Règles

- Attention :
 - L'utilisation de typeid sur des classes non polymorphes (sans fonctions virtuelles) pourrait renvoyer le type statique
- Nécessite de rajouter des options de compilation
 - Forcer le compilateur à ajouter les informations de type dynamique
 - Visual C++ : /GR ou Projects|Settings|C/C++|C/C++ Language|Enable RTTI
- Penser à déréférencer les pointeurs pour obtenir les informations sur le type de l'objet pointé et pas sur le type du pointeur!
- Si ptr est 0 ou NULL, typeid(*ptr) lance l'exception std::bad_typeid

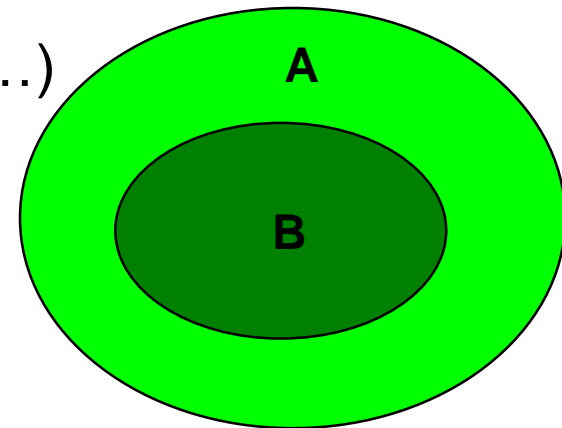
Héritage multiple en C++

Héritage multiple en C++

Héritage multiple régulier

Héritage Simple : Rappel

- B hérite de A, si B est un cas particulier de A
- Tout représentant de B peut être considéré comme un représentant de A
- A est une classe définie par :
 - Ses attributs (Nom, Type)
 - Ses méthodes (Nom, Paramètres , ...)
- B hérite de A si :
 - B a toutes les propriétés de A
 - B est un cas particulier de A
 - Tout ce qui est vrai pour A, l'est pour B



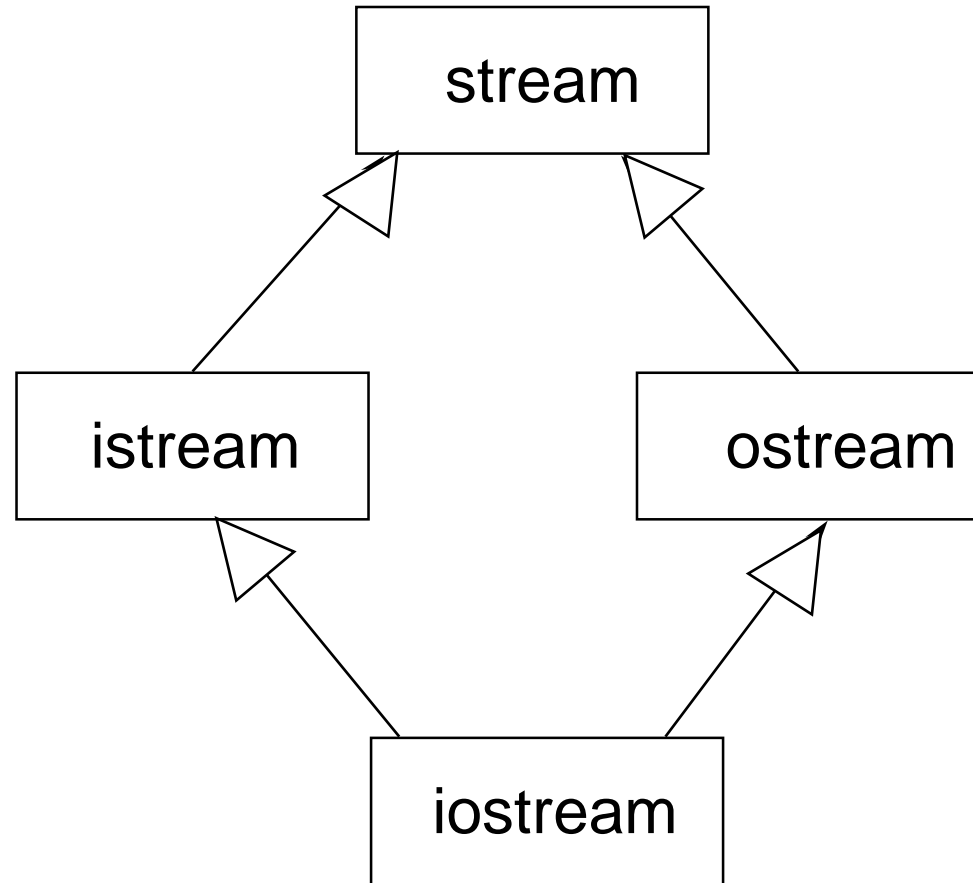
Règles d'Héritage (2)

- Constructeurs
 - Jamais hérités
- Méthodes
 - Héritées
 - Peuvent être *redéfinies* (overriding)
 - La nouvelle méthode *remplace* celle de la classe mère

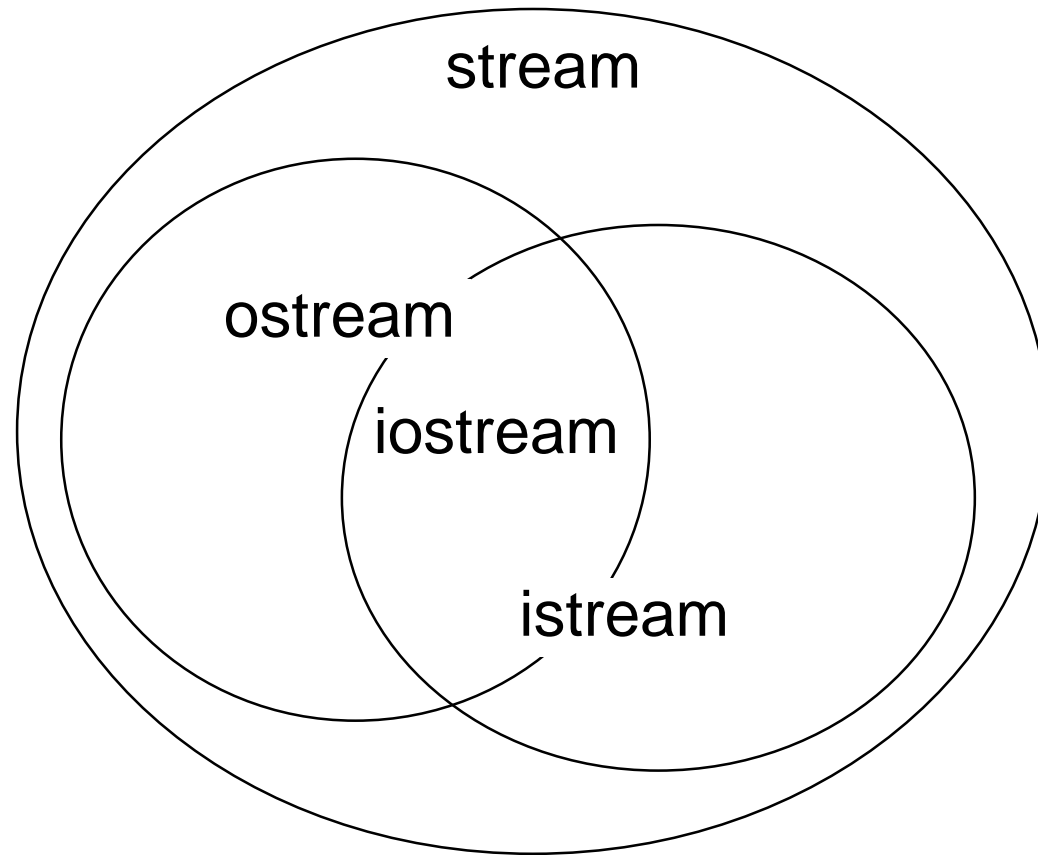
Ne pas confondre surcharge et redéfinition !
- Variables
 - Héritées
 - Peuvent être *précisées* (shadowing)
 - La nouvelle variable *cache* celle de la classe mère

Héritage Multiple

- Héritage de plusieurs classes parentes



Vue Ensembliste



Héritage Multiple

- Constitue une extension logique de l'héritage simple

```
class istream
{
    istream (char* name);
};

class ostream
{
    ostream (char* name, boolean append = false);
};

class iostream : public istream, public ostream
    {iostream (char* name);};



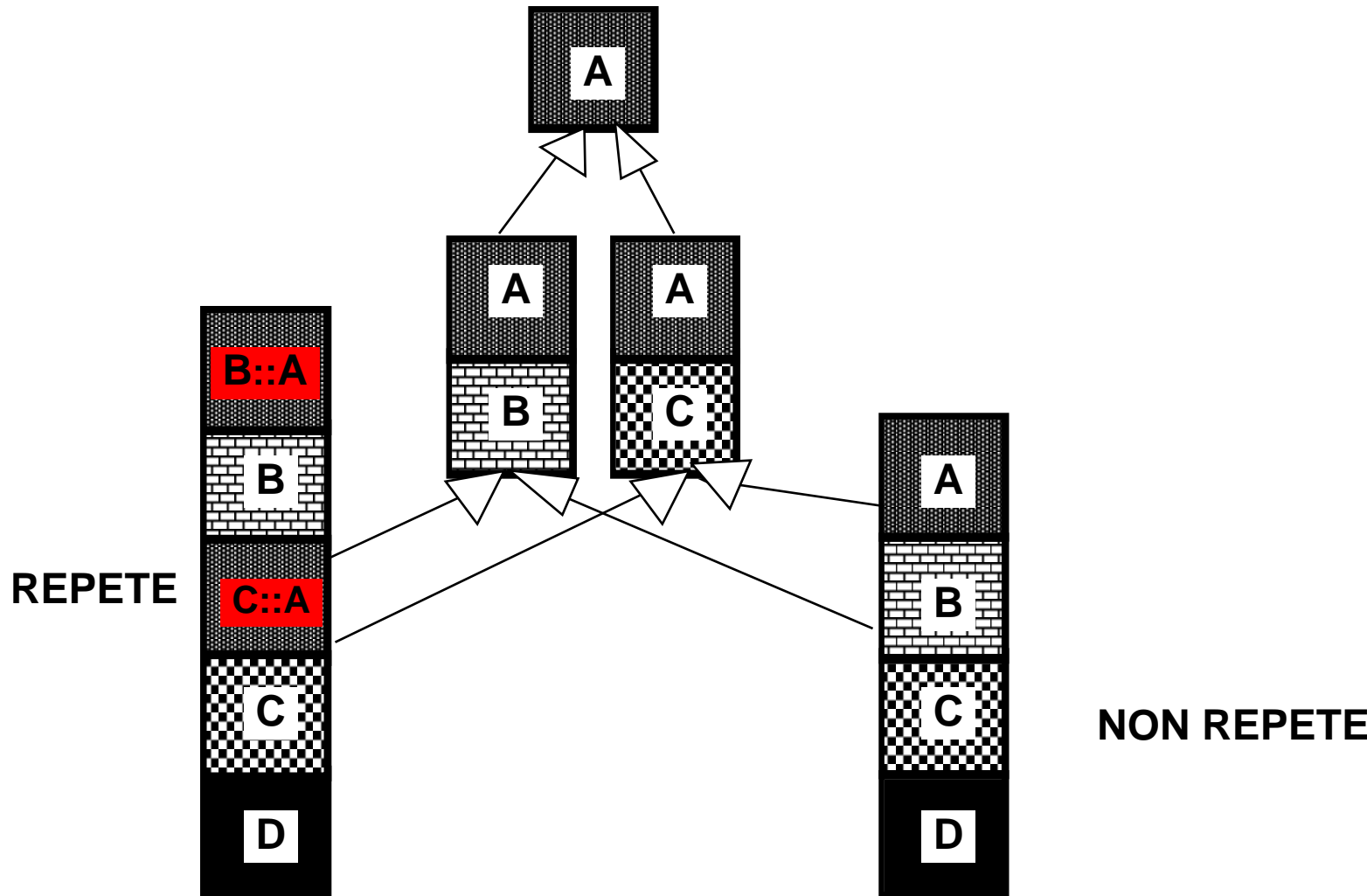
---



iostream:: iostream (char* name): istream (name), ostream (name, true)
    { ... }
```

Héritage Répété (1)

- Effets de bord négatifs de l'héritage répété



Héritage Répété (2)

- Ambiguïté sur l'accès à un attribut

```
class stream {
    stream (char* name, mode open_mode);
    char* file_name;
    mode open_mode;
};

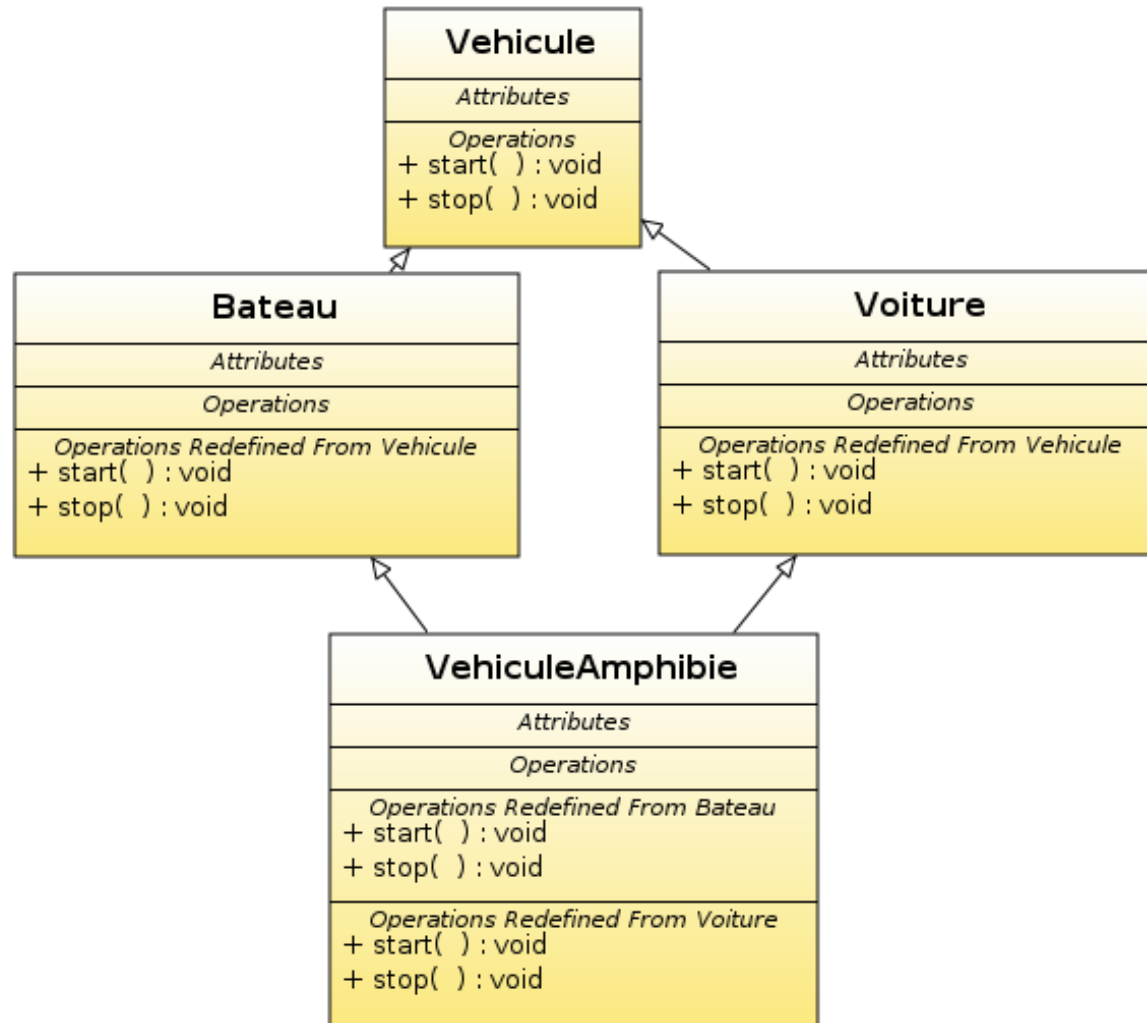
class istream : public stream {...}
class ostream : public stream {...}
class iostream : public istream, public ostream
    { iostream (char* name);};

iostream::iostream (char* name) : ...
{
    ...
    istream::file_name = ....
    ostream::file_name = ....
}
```

Héritage multiple en C++

Héritage multiple virtuel

Héritage multiple (UML)



Une solution avec l'héritage virtuel ?

- C++ propose une solution avec l'héritage virtuel.
- Celle-ci est destinée à lever les soucis dus notamment aux appels à répétition des constructeurs de la super classe.
 - préciser `virtual public` comme type d'héritage entre Bateau et Véhicule,
 - préciser `virtual public` comme type d'héritage entre Voiture et Véhicule,
 - ajouter explicitement la clause `virtual public Vehicule` sur la définition de l'héritage de Amphibie en premier terme (avant les déclarations d'héritage de Bateau et Voiture) .

Une solution avec l'héritage virtuel ?

```
1  #include <iostream>
2  using namespace std;
3  class vehicule{
4      public:
5          vehicule(){
6              cout<<"constructeur vehicule" <<endl;
7          }
8          virtual void start()=0;
9  };
10 class voiture:virtual public vehicule{
11     public:
12         voiture():vehicule(){
13             cout<<"constructeur de voiture" <<endl;
14         }
15
16         virtual void start(){
17             cout<<"voiture demarre"<<endl;
18         }
19     };
```

Une solution avec l'héritage virtuel ?

```
1  class bateau:virtual public vehicule{
2      public:
3          bateau():vehicule(){
4              cout<<"bateau construit"<<endl;
5          }
6          virtual void start(){
7              cout<<"bateau demarre.."<<endl;
8          }
9  };
10 class amphibie:virtual public vehicule,public voiture,public bateau{
11     public:
12         amphibie():vehicule(),voiture(),bateau(){
13             cout<<"constructeur amphibie"<<endl;
14         }
15
16         void start(){
17             cout<<"start() amphibie" <<endl;
18         }
19     };
```

Une solution avec l'héritage virtuel ?

```
1  int main(int argc, char** argv){
2      cout<<"debut du test heritage virtuel"<<endl;
3      vehicule* amph= new amphibie();
4
5      amph->start();
6
7      vehicule* b = new bateau();
8      b->start();
9      cout<<"fin du test héritage virtuel" << endl;
10 }
```

Une solution avec l'héritage virtuel ?

```
Fichier Éditer Affichage Terminal Aller Aide
└─(romje@xen-server:pts/10)───(~/work/cpp_training/inheritance\ )-
└─ \ └─(\ \ 17:51\ :%)─\ -\      ──\ (lun,mai04)─┐
debut du test heritage virtuel
constructeur vehicule
constructeur de voiture
bateau construit
constructeur amphibie
start() amphibie
constructeur vehicule
bateau construit
bateau demarre..
fin du test héritage virtuel
└─(romje@xen-server:pts/10)───(~/work/cpp_training/inheritance\ )-
└─ \ └─(\ \ 17:51\ :%)─\ -\    □      ──\ (lun,mai04)─┐
```

Héritage multiple en C++

**Construction des classes de
base virtuelles**

Construction & Héritage Multiple

- L'héritage virtuel supprime toute transmission de paramètres entre les constructeurs des classes dérivées et celui de la classe mère
- La classe dérivée ("iostream") peut transmettre directement des paramètres de construction à la classe mère ("stream")
- Que se passe t'il si :
 - class istream : public stream ...{.....}
 - ET
 - class ostream : public virtual stream ...{.....}
- Que se passe t'il si :
 - class istream : public virtual stream ...{.....}
 - ET
 - class ostream : private virtual stream ...{.....}

Ordre De Construction

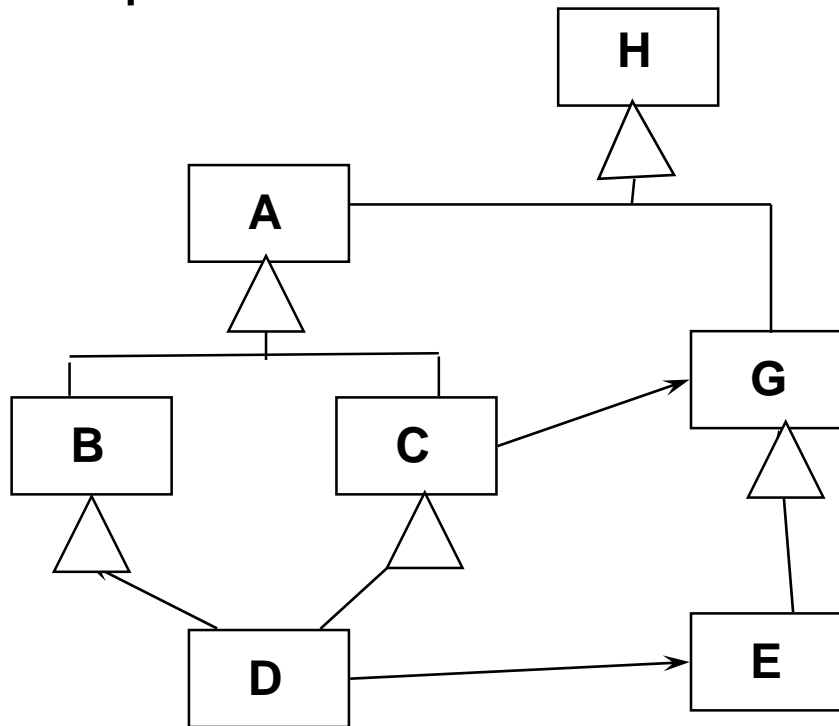
- L'ordre de construction est totalement défini

```
MaClasse::MaClasse(P1, P2, P3, ... Pn) :  
    ClasseMere1(Pk, f(Pj)), ...,  
    ClasseImbriquee1(g(Pz), ...), ...  
{  
    ...  
}
```

1. Construction des classes héritées dans l'ordre fourni par la déclaration de classe (application récursive de la construction à ces classes)
2. Construction des classes imbriquées dans l'ordre de leur déclaration dans la classe les contenant
3. Construction de la classe finale

Ordre De Construction : Exercice

- Dans quel ordre sont construits les représentants des classes pour une instantiation de D ?



```
class D : public B, public C
{
    E attribut1;
};

class C : public virtual A
{
    G attribut2;
};

class B : public virtual A
{...}

class A : public H {...}
class E : public G {...}
class G : public H { ...}
```

Règle d'Utilisation de l'Héritage Multiple

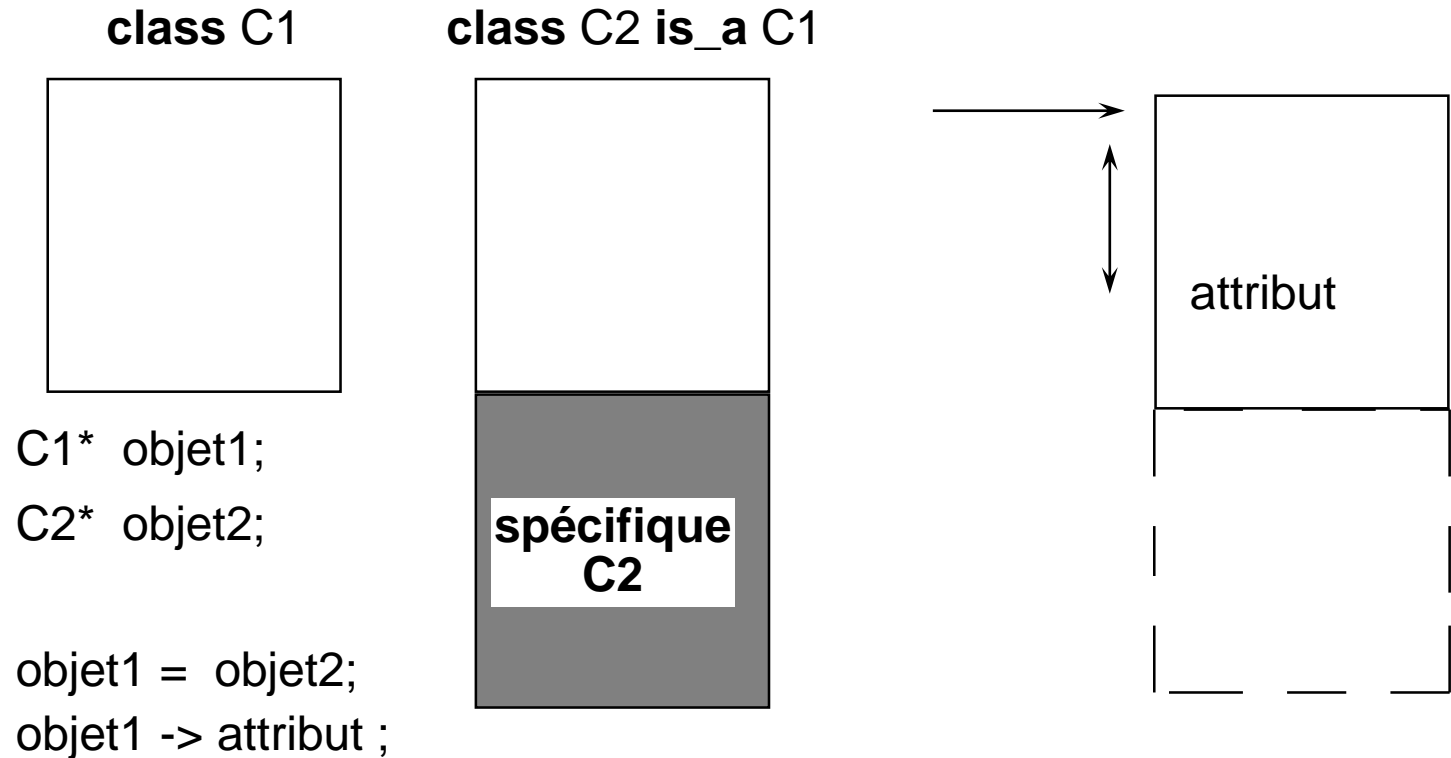
- Éviter autant que possible l'héritage multiple
 - Très souvent l'emploi de l'héritage multiple correspond à une mauvaise modélisation
 - Ne jamais utiliser l'héritage répété !
- Utiliser l'opérateur de résolution de scope « :: » en cas de conflits de noms dus à l'héritage multiple

Héritage multiple en C++

Conversions en cas d'héritage multiple

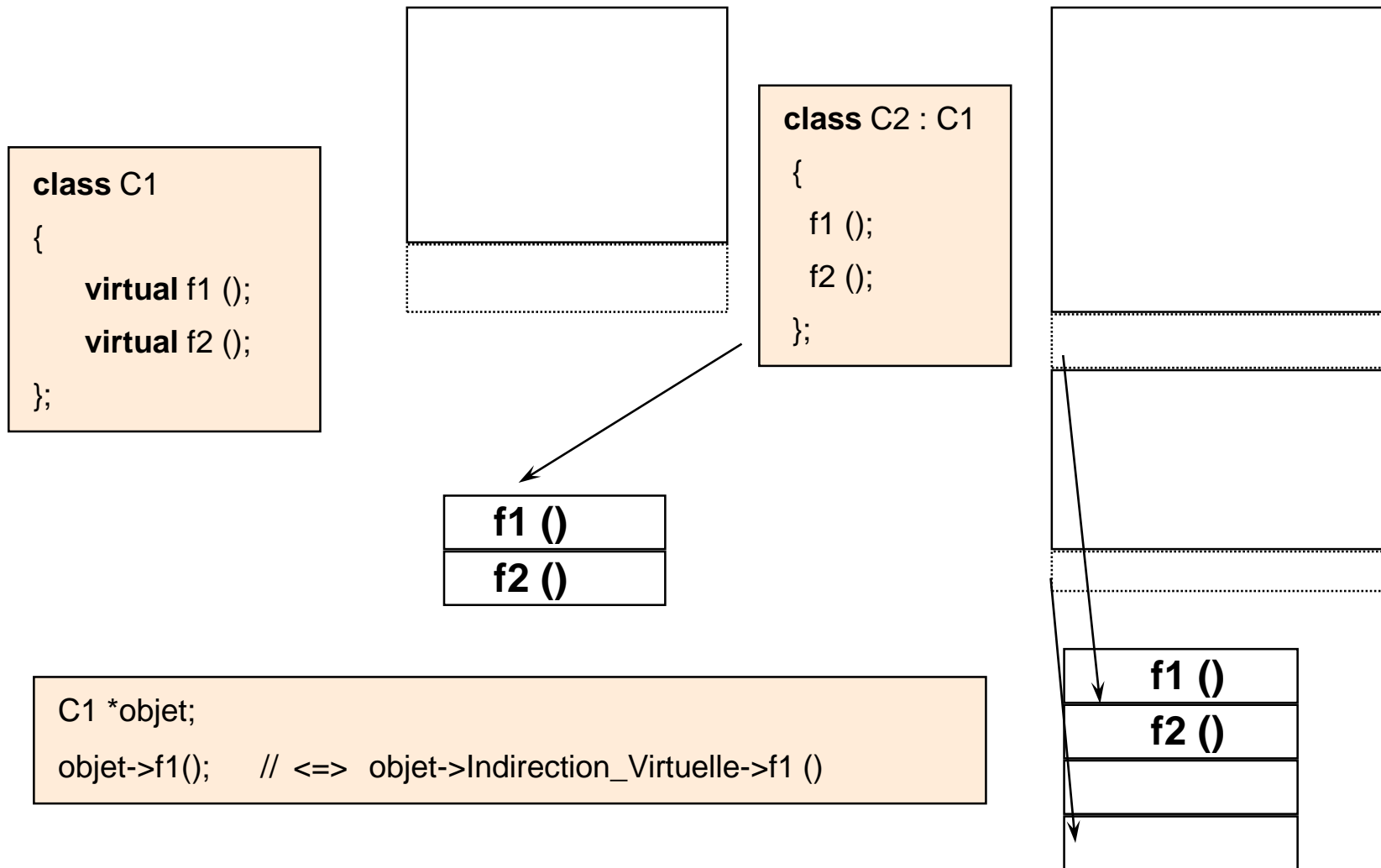
Implémentation de l'héritage

- Agrégation des nouveaux attributs



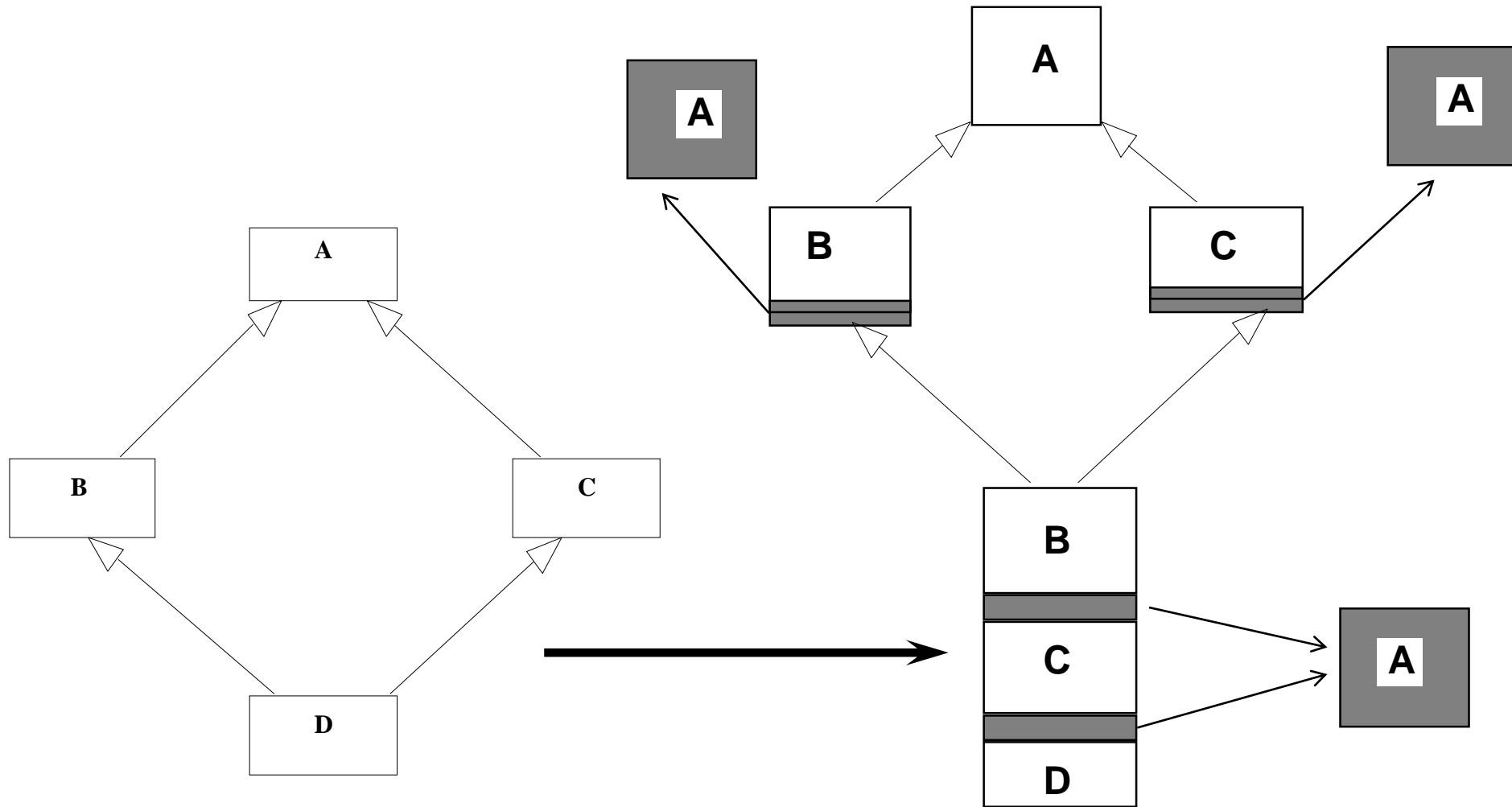
Implantation C++ & Méthodes Virtuelles

- Utilisation de pointeurs de fonctions



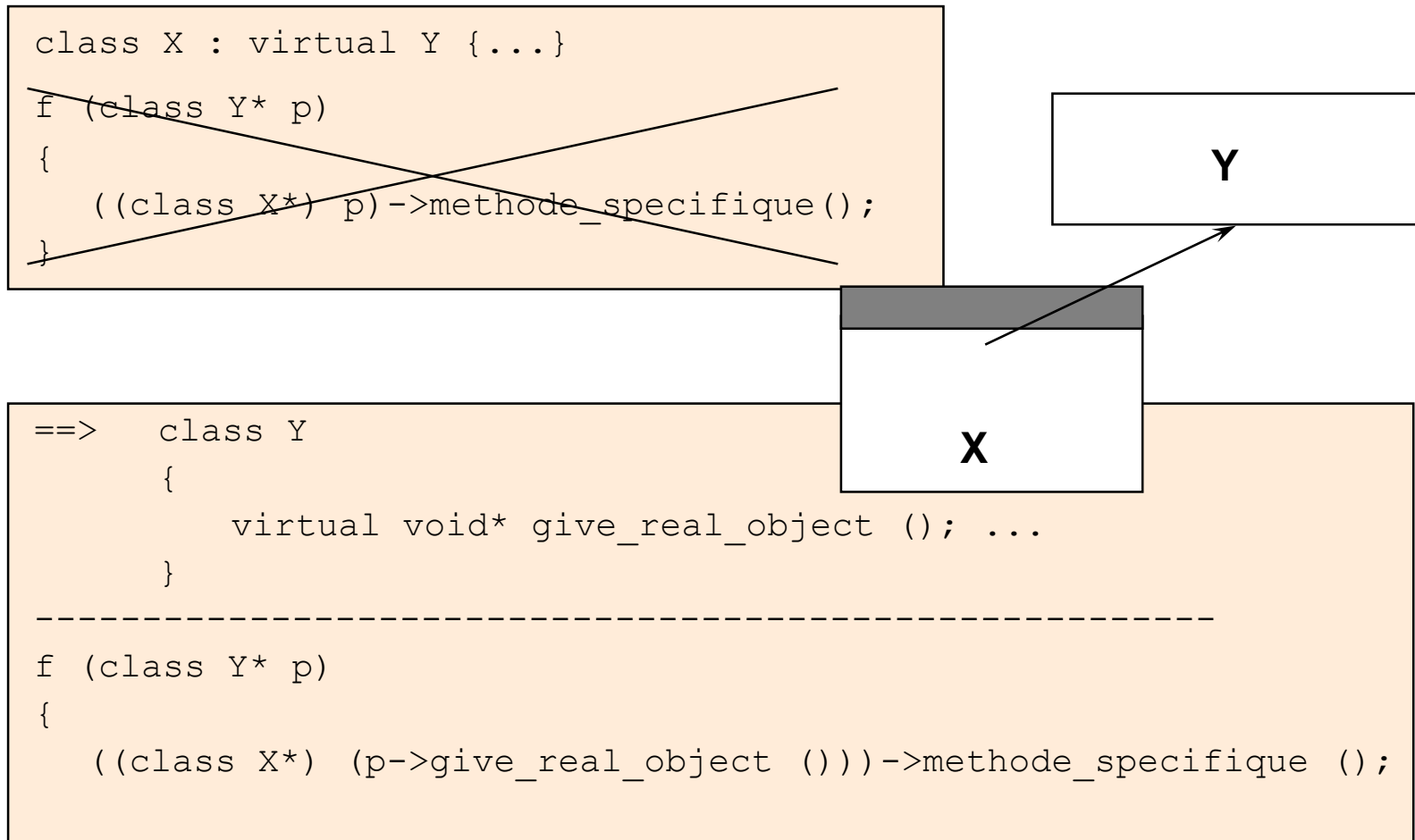
Implantation C++ & Héritage Multiple

- Accès indirect aux attributs



Casting & Héritage Virtuel

- L'héritage virtuel rend complexe le "Down-Casting"



La STL

(Standard Template Library)

- Standard Template Library
- Propose des implémentations génériques (template) de la plupart des conteneurs
 - vecteur, liste simple, file, pile, ...
- Fournit également itérateurs et algorithmes génériques pour la manipulation des données des conteneurs
- Les itérateurs : "iterator"
 - Proposent une interface commune pour le parcours des conteneurs
 - Manipulés comme un pointeur parcourant un tableau contigu

La STL

La classe string

String

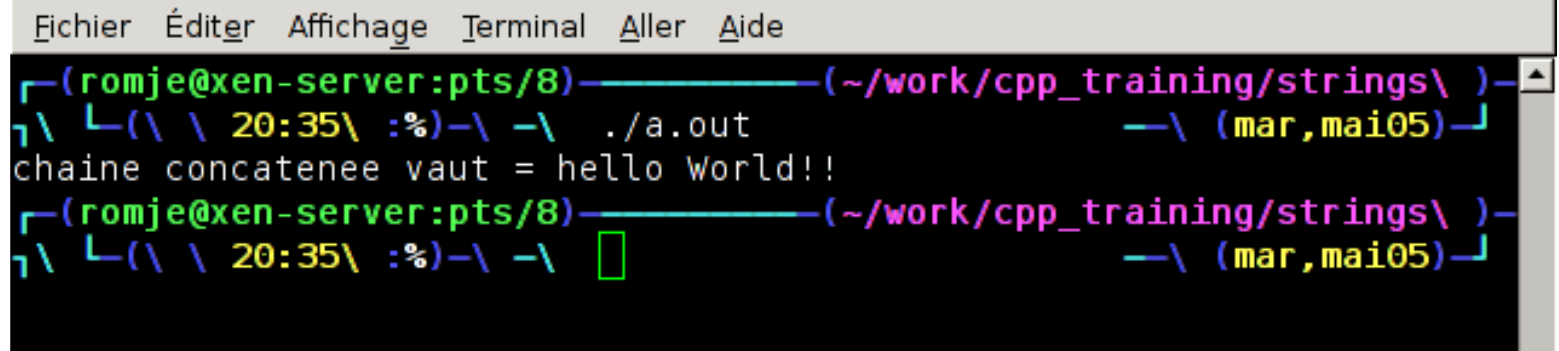
- Classe `basic_string<charT, char_traits<charT>>`
 - `typedef basic_string <char> string`
 - `typedef basic_string<wchar_t> wstring;`
- Opérations Arithmétiques
 - `+`, `+=`, `=`
 - `insert`, `append`, `replace`, `copy`, `substring`
- Opérations de comparaisons
 - `==`, `!=`, `>`, `<`, `>=`, `<=`
- Opérations de recherches
 - `find`, `rfind`, `find_first_of`, `find_last_of`, ...
- Opérations d'accès
 - `[]`
 - `c_str()`, `data`

String

- Manipulation aisée de chaînes de caractères :
 - Concaténation avec + (opérateur redéfini)
 - Compatibilité avec le C pour le code prenant des `const char *`
 - Possibilité d'appliquer un certain nombre d'algorithmes
- Nombreuses fonctionnalités :
 - Méthodes de recherche, remplacement
 - `find()`
 - `rfind()`
 - `replace()`
 - Comparaison intuitive avec l'opérateur `>`

String : Exemples

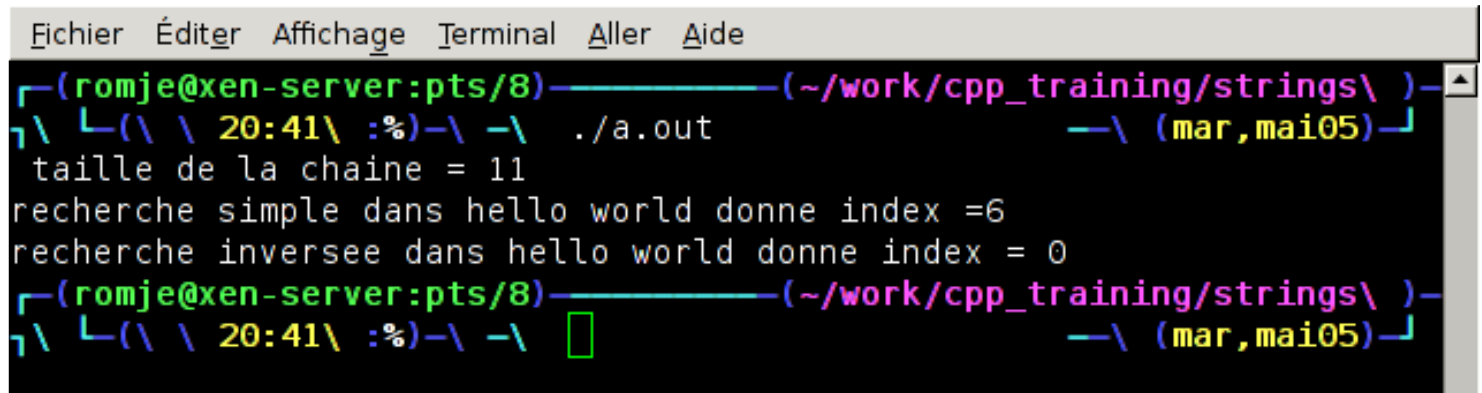
```
1  #include <iostream>
2  #include <string> //a ne pas oublier!!
3  using namespace std;
4  int main(int argc, char** argv){
5      string s1;
6      string s2("hello");
7      string s3(" World!!");
8      string concat=s2+s3;
9      cout<<"chaine concatenee vaut = " << concat << endl;
10 }
```



```
Fichier Éditer Affichage Terminal Aller Aide
(romje@xen-server:pts/8)~/work/cpp_training/strings\ )-
└─(\ \ 20:35\ :%)-\ -\ ./a.out ─\ (mar,mai05)┐
chaine concatenee vaut = hello World!!
(romje@xen-server:pts/8)~/work/cpp_training/strings\ )-
└─(\ \ 20:35\ :%)-\ -\ ┐ ─\ (mar,mai05)┐
```

String : Exemples

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main(int argc, char** argv){
5      string s1("hello world");
6      int from_begin = s1.find("world");
7      int from_end=s1.rfind("hello");
8      cout<<" taille de la chaine = " << s1.size() << endl;
9      cout<<"recherche simple dans " << s1 << " donne index =" << from_begin << endl;
10     cout<<"recherche inversee dans " << s1 << " donne index = " << from_end << endl;
11 }
```



```
Fichier Éditer Affichage Terminal Aller Aide
(romje@xen-server:pts/8)~/.work/cpp_training/strings\ )-
└─\ (mar,mai05)┐
└─\ L(\ \ 20:41\ :%)─\ ─\ ./a.out
taille de la chaine = 11
recherche simple dans hello world donne index = 6
recherche inversee dans hello world donne index = 0
(romje@xen-server:pts/8)~/.work/cpp_training/strings\ )-
└─\ L(\ \ 20:41\ :%)─\ ─\ □
```

String : Exemples

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main(int argc, char** argv){
5      string s1("abc");
6      string s2("ABC");
7      cout << " s1> s2 ? " << (s1>s2) << endl;
8      // remplacement d'un caractere via le tableau
9      s1[0]='*';
10     cout<<"nlle version de la chaine =" << s1 << endl;
11
12     // remplacement via la methode find
13     s1.replace(s1.find("*"),1,"toto");
14     cout<<"chaine alteree vaut = " << s1 << endl;
15     return 0;
16 }
```

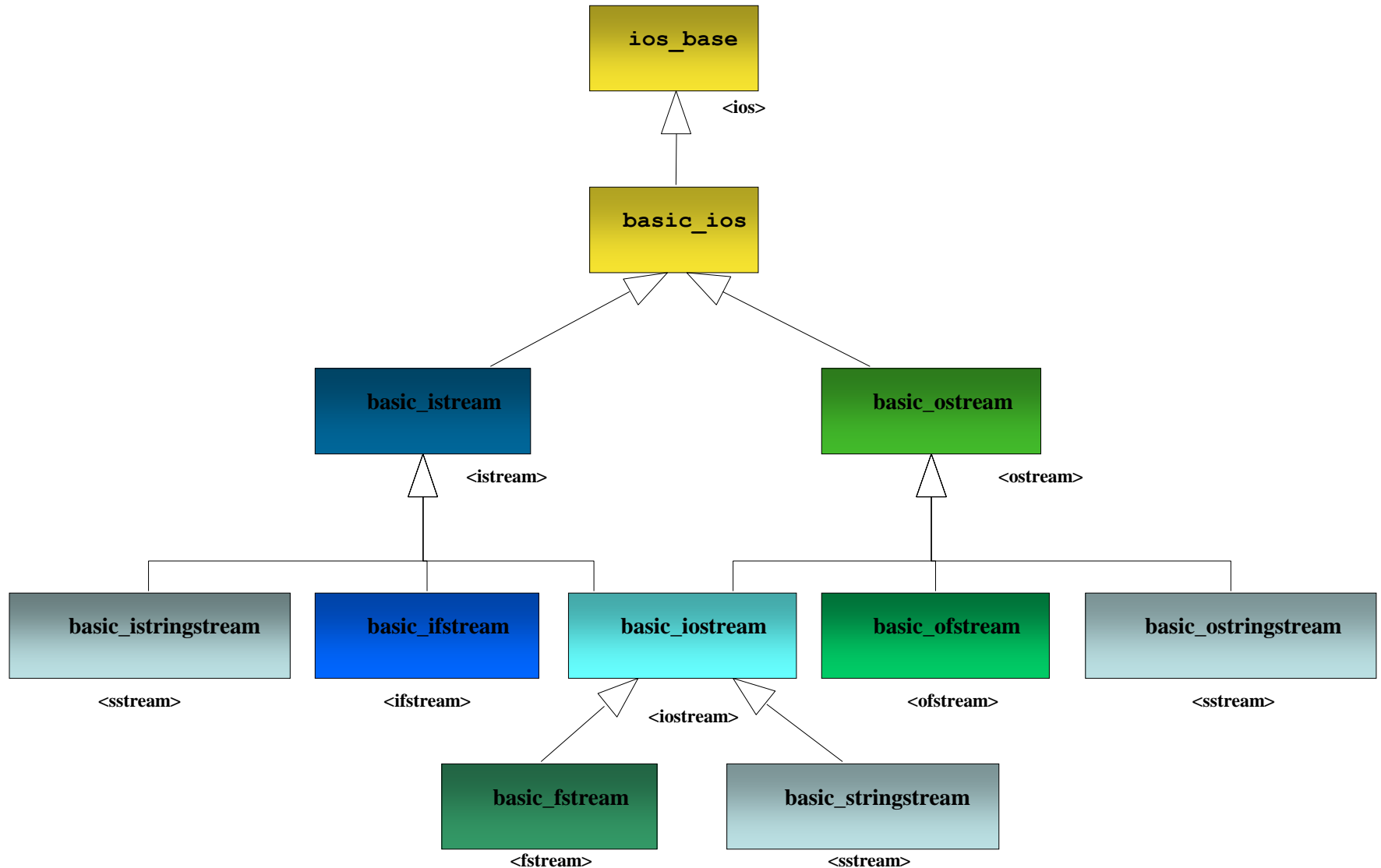
Fichier Éditer Affichage Terminal Aller Aide

```
(romje@xen-server:pts/8)----- (~/.work/cpp_training/strings\ )-
└─┐ └─┐ \ 09:09\ :%)─\ ─\ ./a.out ─\ (mer,mai06)└─┐
  s1> s2 ? 1
nlle version de la chaine =*bc
chaine alteree vaut = totobc
(romje@xen-server:pts/8)----- (~/.work/cpp_training/strings\ )-
└─┐ └─┐ \ 09:09\ :%)─\ ─\  ─\ (mer,mai06)└─┐
```

C++: STL : Flux

Flux d'entrée/sortie de la STL

Les Flux / STL



flux standard d'entrée-sortie

- Flux d'entrée-sortie:
 - `std::cout` : sortie standard
 - `std::cerr` : sortie des erreurs
 - `std::cin` : entrée standard (utiliser `>>` au lieu de `<<`)
- *Opération de Lecture / Écriture :*
 - Opérateur `>>` : écriture
 - Opérateur `<<` : lecture

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

int main() {
    int radius=0;
    cin >> radius;
    Circle* c = new Circle(radius);

    cout << "rayon= " << c->getRadius()
         << « aire= "    << c->getArea() << endl;
    cerr << "c = " << c    << endl;
}
```

Les Flux : La classe ios_base (2)

- Modes d'ouverture des fichiers
 - Champ static const openmode de ios_base
 - in
 - out
 - binary
 - trunc
 - app
 - ate

```
ofstream f("monFichier.txt", ios_base::out | ios_base::trunc);
```

Les Flux : la classe ios_base (3)

- Etats des flux
 - static const iostate
 - goodbit
 - eofbit;
 - failbit;
 - badbit;

- Formatage des flux
 - boolalpha
 - hex.
 - oct
 - dec.
 - fixed
 - scientific
 - left
 - right
 - internal
 - showbase
 - showpoint
 - showpos
 - uppercase
 - unitbuf
 - skipws

Les Flux : La classe `basic_istream`

- Opérations de lecture formatées
 - `operator>>(type &);`
 - `bool`, `char`, `short`, `int`, `long`, `float`, `double`, `string`
 - Bref tous les types de base...
- Opérations de lecture non formatées
 - `int_type get();`
 - `get(char_type &c), ...`
 - `int_type peek();`
 - `read(char_type *s, streamsize n);`
 - `getline(char_type *s, streamsize n);`
- Opérations de gestion du buffer
 - `basic_ostream<charT, traits> &flush();`
 - `pos_type tellp();`
 - `seekp(pos_type);`

Les Flux : La classe `basic_ostream`

- Opérations d'écritures formatées
 - `opérateur<<(type);`
 - `bool`, `char`, `short`, `int`, `long`, `float`, `double`, `string`
 - Bref tous les types de base...
- Opérations d'écritures non formatées
 - `put(char_type);`
 - `write(const char_type *p, streamsize taille);`
- Opérations de gestion du buffer
 - `flush();`
 - `pos_type tellp();`
 - `seekp(pos_type);`
- Opération de gestion des tampons
 - `rdbuf` `r/w`
- Opération de gestion des exceptions
 - `exceptions` `r/w` (`goodbit` + `eofbi` + `failbit` + `badbit`).

Les Flux sur Chaînes de Caractères

- Classes travaillant avec la `basic_string`
 - `istringstream`, `ostringstream` et `stringstream`
- Opérations communes
 - `rdbuf()` retourne un pointeur sur le string buffer du flot;
 - `str()` retourne un string avec le contenu du flot;
 - `str(chaine)` fixe le contenu du flot.

```
string s, si;  
int ii;  
double di;  
  
cout << "Entrez une chaine _ , un entier _, un double" << endl;  
getline(cin, s);  
istringstream is(s);  
is >> si >> ii >> di;  
ostringstream os;  
os << "chaine:" << si << " entier:" << ii << " double:" << di << endl;  
cout << os.str();
```

Les Flux – Les manipulateurs

- Opérateur << pour accepter des fonctions de type `ostream& (*)(ostream&)`
 - lorsqu'elles sont appliquées elles modifient le flot courant.
Ce sont des manipulateurs.
- Manipulateur prédéfinis:
 - `endl` Retour à la ligne + vide le flot
 - `ends` Fin de ligne
 - `flush` Vide le flot
 - `left` / `right` alignement
 - `boolalpha` / `noboolalpha` forme des booléens (true ou 1)
 - `hex` / `oct` / `dec` base des entiers
 - `uppercase` / `nouppercase`
 - ...

Flux fichier

- 3 types de flux:
 - Flux d'entrée : `ifstream`
 - Flux de sortie : `ofstream`
 - Bidirectionnel : `fstream` Opérations communes
 - `bool is_open()`
 - `void open(const char* filename, openmode mode)`
 - `void close()`
- Mode de positionnement des fichiers
 - Champ `static const seekdir;`
 - `beg`
 - `cur`
 - `end`
- Même utilisation que les flux standards

Les Flux sur fichiers - Exemple

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream f (« monFichier.txt");
    f << uppercase << hex << 255 << endl;
    f.close();

    fstream fic;
    fic.open("monFichier.txt"), ios_base::in| ios_base::out| ios_base::trunc);
    if (fic.is_open())
    {
        // écriture
        fic << "une chaine" << " " << 24 << " " << 3.14 << endl;
        fic.seekg(0);
        // lecture
        string si ;
        int ii;
        double di;
        fic >> si >> ii >> di;
        cout << si << " " << ii << " " << di << endl;
        fic.close();
    }
}
```

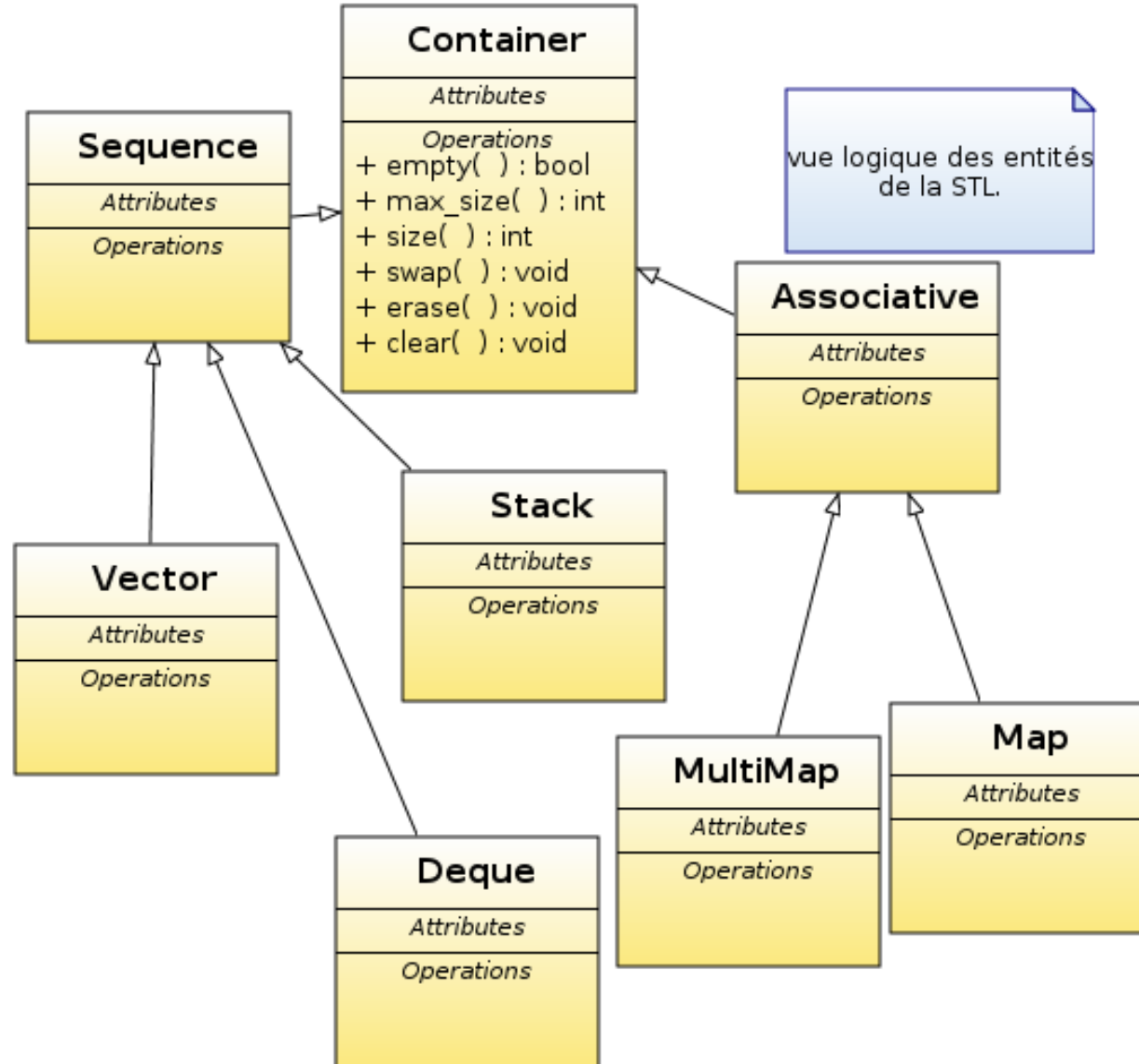
La STL

Conteneurs et itérateurs de la STL

Généralités

- Conteneurs génériques de données similaires à des tableaux de tailles variables. Implémentés à l'aide de classes templates.
- Classification suivant :
 - le type d'accès :
 - Séquentiel
 - Indirect via un hash d'une clé (similaire à un accès base de données).
 - L'implémentation interne : Conditionne les cas d'utilisation typiques et les anti patterns pour chacune des implémentations

Vue logique des conteneurs de la STL



- Il est bon de noter que tous les conteneurs de la STL adoptent la forme dite de Coplien.

Forme de Coplien

Permet de désigner une classe dotée de :

- constructeur sans paramètres
- destructeur virtuel
- constructeur par copie

Utilisation de la liste

```
1  #include <iostream>
2  #include <list>
3  using namespace std;
4  class test_list{
5  private:
6      list<int> l;
7      void displayList(){
8          cout<<"-----"<<endl;
9          cout<<"list vide ? " << l.empty()<< " taille liste = " << l.size() << endl;
10         cout<<"Element de tete = " << l.front() << " de queue = " << l.back() << endl;
11     }
12
13 public:
14     test_list(){
15         cout<<"init de test_list"<<endl;
16     }
17     void runTest();
18 };

```

Utilisation de la liste

```
1 void test_list::runTest(){
2     displayList();
3     l.push_front(1); //contenu{1}
4     l.push_front(2); //contenu{2,1}
5     displayList();
6     l.push_back(3); //contenu(2,1,3}
7     l.pop_front(); //contenu{1,3}
8     displayList();
9     l.pop_back(); //contenu{1}
10    displayList();
11 }
12 int main(int argc, char** argv){
13     test_list test;
14     test.runTest();
15 }
```

```
Fichier Éditer Affichage Terminal Aller Aide
\ \ 12:21 \ :%) -\ -\ ./a.out
init de test_list
-----
list vide ? 1 taille liste = 0
Element de tete = 4199792 de queue = 4199792
-----
list vide ? 0 taille liste = 2
Element de tete = 2 de queue = 1
-----
list vide ? 0 taille liste = 2
Element de tete = 1 de queue = 3
-----
list vide ? 0 taille liste = 1
Element de tete = 1 de queue = 1
(romje@xen-server:pts/8) - (~/.wor
\ \ 12:21 \ :%) -\ -\ □
```

Il est intéressant de remarquer les valeurs incohérentes de la tête et de la queue juste après l'initialisation de la structure de données.

Conteneurs : Concepts

- Container
 - Détient d'autres objets
 - Les éléments sont rangés dans un ordre indéterminé
 - Associé à un itérateur pour parcourir les éléments détenus
- Forward Container
 - Les éléments sont rangés dans un ordre déterminé
 - Associé à un itérateur de Type Forward
- Reversible Container
 - Les éléments sont rangés dans un ordre déterminé
 - Associé à un itérateur de Type Bidirectional
- Random Access Container
 - Les éléments sont rangés dans un ordre déterminé
 - Associé à un itérateur de Type Random

Conteneurs : Sequences

- Sequence
 - Conteneur de taille variable
 - Insertion / Suppression d'élément à une position donnés
 - Accès à un l'élément à une position donnés
- Front Insertion Sequence
 - Conteneur de taille variable
 - Insertion / Suppression d'élément en tête du conteneur
 - Accès à l'élément de tête
- Back Insertion Sequence
 - Conteneur de taille variable
 - Insertion / Suppression d'élément en queue du conteneur
 - Accès à l'élément de queue

Classes Conteneurs - 2 types (1)

- Sequences

- vector<Type, Alloc>
 - valarray<Type, Alloc>
 - deque<Type , Alloc >
 - list<Type, Alloc >
 - slist<Type, Alloc > *
 - bit_vector

- Conteneur associatifs

- set <Key, Compare, Alloc >
 - map <Key,Type, Compare , Alloc >
 - multiset <Key , Compare , Alloc >
 - multimap <Key,Type , Compare , Alloc >
 - hash_set <Key, HashFcn, EqualKey , Alloc > *
 - hash_map <Key, Type, HashFcn, EqualKey , Alloc > *
 - hash_multiset <Key, HashFcn, EqualKey , Alloc > *
 - hash_multimap < Key, Type, HashFcn, EqualKey , Alloc > *

Autres classes

- String package
- Container adaptors
 - `stack<Type, Sequence>`
 - `queue<Type, Sequence>`

Conteneurs : Opérations Communes (1)

- Constructeurs
 - `Cont<...> c; // size ==0`
 - `Cont<...> c2(c); // copie collection homogène`
 - `Cont<...> c(begin, end); // copie à partir d'un intervalle`
- Opérateurs membres
 - `operator= c1=c1; // copie collection homogène`
 - `operator== c1==c2 // vrai pour une taille égale et éléments égaux`
 - `operator< c1 < c2 // comparaison lexicale`
- Opérateurs non membres(fonctions)
 - `operator> c1 > c2 // comparaison lexicale`
 - `operator>= c1 >= c2 // comparaison lexicale`
 - `operator<= c1 <= c2 // comparaison lexicale`
 - `operator!= c1 != c2 // vrai pour taille ou éléments différents`
- Taille
 - `c.size(); // nombre d'éléments contenus`
 - `c.empty(); // vrai si vide`
 - `c.max_size(); // taille max allouée`

Conteneurs : Opérations Communes (2)

- Suppression

- `c.erase(index);`
 - supprime un élément
- `c.erase(begin, end);`
 - supprime de begin à end-1
- `c.clear();`
 - équivalent `c.erase(c.begin(), c.end());`

- Echange

- `c.swap(c2);` // échange le contenu

- Iterateurs

- `C<...>::iterator it = c.begin();`
 - renvoie un itérateur pointant sur le premier élément
- `C<...>::iterator it c.end();`
 - renvoie un itérateur pointant sur le dernier élément

- Accès aux éléments

- `c.front()`

- retourne la référence du premier élément

- `c.back()`

- retourne la référence du dernier élément

- Insertion

- `c.insert(index, data);`

- ajoute un élément à une position donnée

- `c.insert(index, n, data);`

- ajoute n éléments à partir d'une position donnée

- `c.insert(index, begin, end);`

- ajoute les éléments entre begin et end à partir d'une position donnée

- `c.push_back(data);`

- équivalent `c.insert(c.end(), data);`

- `c.push_front();`

- équivalent `c.insert(c.begin(), data);`

- Suppression

- `c.erase(key);`
- `c.pop_back();`
 - équivalent `c.erase(c.end());`
- `c.pop_front();`
 - équivalent `c.erase(c.begin());`

- Iterateurs

- `C<...>::reverse_iterator it = c.rbegin();`
 - renvoie un itérateur pointant sur le premier élément
- `C<...>::reverse_iterator it c.rend();`
 - renvoie un itérateur pointant sur le dernier élément

Vector - Exemple

```
private :  
    vector<int> vecteurTest;  
void exemple::testVector() {  
  
    cout << "Remplissage: \n " << endl;  
    for(int i = 0; i<10; i++){  
        vecteurTest.push_back(i);  
    }  
  
    cout << "suppression: \n " << endl;  
    while(vecteurTest.size()>0) {  
        v2.pop_back();  
    }  
}
```

Conteneurs : Conteneurs Associatifs

- Propriétés commune
 - Utilise une « clé » pour trouver l'élément au sein du conteneur
- Map, set, unordered_map, unordered_set
 - Chaque clé est unique dans le conteneur, pas de doublons
- Multimap et multiset
 - Plusieurs entrées peuvent avoir la même clé (doublons)

- **Type : notion de clé / valeur**

- `typedef pair<const Key, Type> value_Type`

- `map`, `multimap`

- `typedef Key value_Type`

- `set`, `multiset`

- **Accès aux éléments**

- `operator[key]`

- retourne la référence de l'élément

- **Recherche**

- `C<Key, Type>::iterator It = c.find(key);`

- renvoie la position sur l'élément ou sur `c.end`

- `size_Type ix = c.count(key);`

- renvoie le nombre d'éléments répondants à `key`

- `C<Key, Type>::iterator It = c.lower_bound(key);`

- `C<Key, Type>::iterator It = c.upper_bound(key);`

- Insertion
 - c. `insert(value)`; //ajoute un élément de Type `value_Type`
 - c. `insert(index, value)`; //ajoute un élément à partir d'une position donnée
- Suppression
 - c. `erase(key)`;
 - c. `pop_front()`;

map - Example

```
private :  
    map<int,string> mapTest;  
void exemple::testVector() {  
    mapTest[0] = "Jean";  
    mapTest[1] = "Anne-Laure";  
    mapTest[2] = "Yves";  
    mapTest.insert(std::pair<int,string>(5,"Marie"));  
    mapTest.insert(map<int,string>::value_type (7,"Olivier"));  
    mapTest.insert(std::make_pair(8,"Claire"));  
}
```

unordered_map, unordered_set

- Préfixé par `boost::` ou `std::` (C++11)
- Même usage que `map` et `set`, mais
 - Utilise une fonction de hash au lieu de `Compare`
 - Nécessite de spécifier `std::hash<T>` si `T` est un type utilisateur
 - Déjà définie pour les types primitifs, `unique_ptr` et `shared_ptr`
 - Temps d'accès constant en moyenne $O(1)$, contrairement au `set` et `map` en temps d'accès logarithmique $O(\log n)$

La STL

les itérateurs sur les conteneurs

Abstraction de pointeurs : Itérateur (1)

- Un itérateur permet d'accéder à tous les objets d'un conteneur donné

- Itérateur constant : Input Iterator

- Lecture de valeurs (`val = *it`)
- Un seul sens de déplacement (`++`)
- Simple passe sur les éléments

- Output Iterator

- Affectation de valeurs (`*it = val`)
- Un seul sens de déplacement (`++`)
- Simple passe sur les éléments

- Forward Iterator

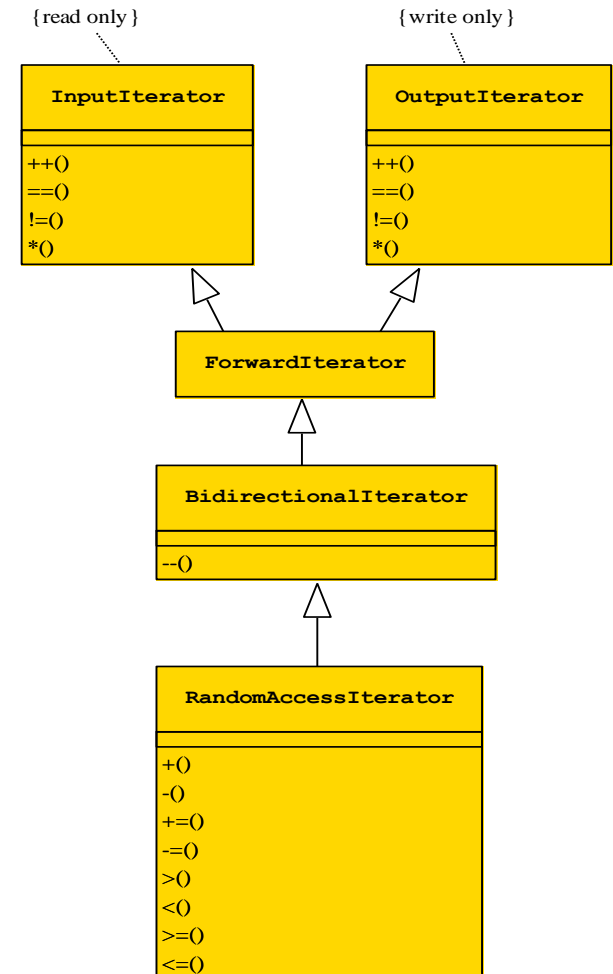
- Affectation et lecture de valeurs
- Un seul sens de déplacement (`++`)
- Multiple passe sur les éléments

- Bidirectional Iterator

- Affectation et lecture de valeurs
- Deux sens de déplacement (`++`, `--`)
- Multiple passe sur les éléments

- Random Access Iterator

- Affectation et lecture de valeurs
- Deux sens de déplacement (`++`, `--`)
- Déplacement par indexation (`+`, `-`)
- Multiple passe sur les éléments



Abstraction de pointeurs : Itérateur (2)

- Tous les itérateurs dérivent de la classe suivante :

```
// header iterator
template <class Category, class T, class Distance = ptrdiff_t,
class Pointer = T*, class Reference = T &>
struct iterator
{
    typedef T value_type;
    typedef Distance difference_type;
    typedef Pointer pointer;
    typedef Reference reference;
    typedef Category iterator_category;
};
```

Itérateur et conteneur

- L'interface des conteneurs permet d'obtenir différents itérateurs :
 - Parcours « normal »
 - Méthode: begin
 - Méthode: end
 - Parcours « inversé »
 - rbegin
 - Rend
- L'itérateur peut être constant

Iterateur constant Vector - Exemple

```
private :  
    vector<int> vecteurTest;  
void exemple::listeVector() {  
  
    cout << "lecture: \n " << endl;  
    vector<int>::const_iterator iter;  
    for (iter = vecteurTest.begin(); iter!=values.end(); ++iter) {  
        cout<<" " << *iter;  
    }  
}  
}
```

La STL

Utiliser ses objets dans les conteneurs de
la STL

Règle d'or

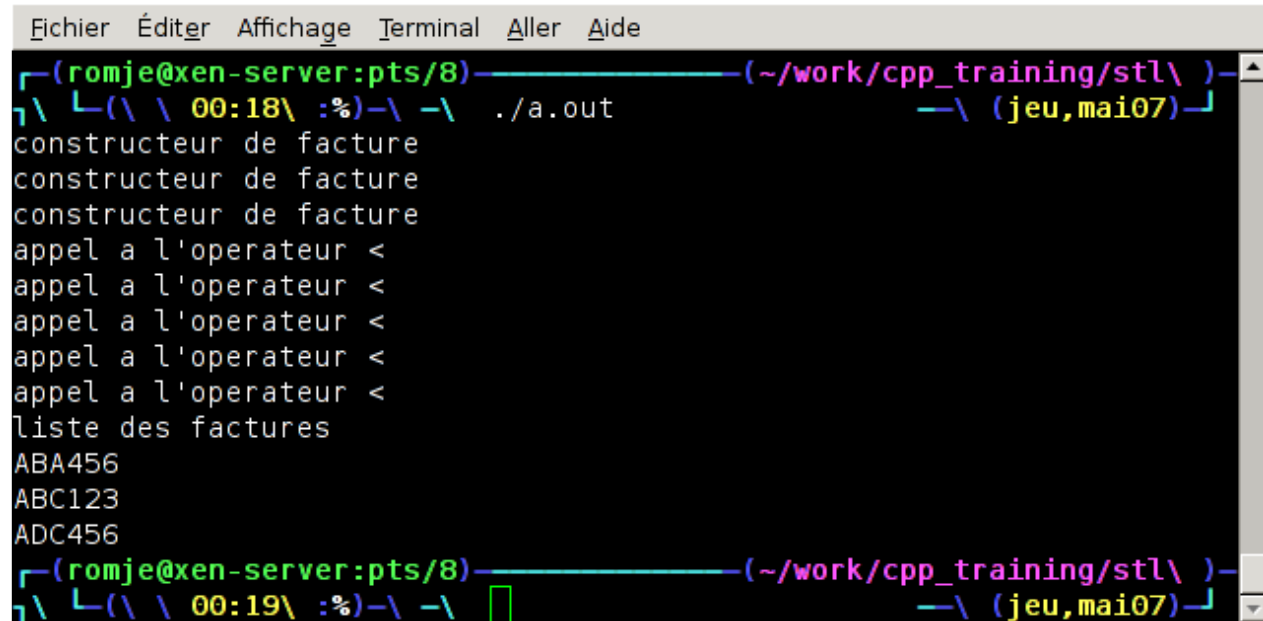
- Comment utiliser ses objets dans des sets ou dans des maps ?
 - réclame la fourniture d'une relation d'ordre,
 - au sein de la STL, fournir une implémentation de l'opérateur < suffit

Exemple d'utilisation

```
1  #include <iostream>
2  #include <set>
3  #include <string>
4  using namespace std;
5  class facture{
6  private:
7      string __date;
8      string __facNumber;
9      float __montant;
10 public:
11     facture(string number,float howmuch){
12         cout<<"constructeur de facture"<<endl;
13         __date="today";//naive implementation
14         __montant=howmuch;
15         __facNumber=number; }
16     bool operator < (const facture& f) const{
17         cout<<"appel a l'opérateur "<< endl;
18         return __facNumber<f.facNumber(); }
19     string facNumber() const{
20         return __facNumber; }
21 };
```


Exemple d'utilisation

```
1  int main(int argc, char** argv){
2      set<facture> factures_list;
3      facture f1("ABC123",12.5), f2("ADC456",2.78), f3("ABA456",789.00);
4      factures_list.insert(f1);
5      factures_list.insert(f2);
6      factures_list.insert(f3);
7      cout<<"liste des factures"<<endl;
8      set<facture>::const_iterator iter;
9      for(iter=factures_list.begin();iter!=factures_list.end();++iter){
10         cout << (*iter).facNumber() << endl;
11     }
12 }
```



```
Fichier  Éditer  Affichage  Terminal  Aller  Aide
romje@xen-server:pts/8 ~(/work/cpp_training/stl)
./a.out
constructeur de facture
constructeur de facture
constructeur de facture
appel a l'operateur <
appel a l'operateur <
appel a l'operateur <
appel a l'operateur <
appel a l'operateur <
liste des factures
ABA456
ABC123
ADC456
romje@xen-server:pts/8 ~(/work/cpp_training/stl)
```

La STL

Algorithmes et, functors et predicats

Généralités

- Les algorithmes :
 - sont des fonctions templates permettant de manipuler la donnée d'un conteneur,
 - sont indépendants du type de conteneur manipulé,
 - fournisse un service clair du type :
 - génération de données dans un conteneur à partir d'une fonction avec `std::generate`
 - remplissage d'un conteneur avec `std::fill`
 - recherche dans un conteneur `std::find`

Généralités

- Les algorithmes permettent d'appliquer sur une série de données balayées par un itérateur une fonction ou prédicat.
- Le code suivant extrait de la documentation de la STL montre le header d'un des ces algorithmes :

```
1  template<class InputIterator, class T, class Predicate> inline
2      InputIterator find_if(
3          InputIterator First,
4          InputIterator Last,
5          Predicate Predicate
6      )
```

Prédicat

Définition

- Un prédicat est une fonction membre renvoyant un booléen.
- Elle sera utilisée par des algorithmes de manière à valider une donnée au sens de l'algorithme.

Algorithmes standards 1/3

- Liste des algorithmes ne modifiant pas la collection :
 - **for_each** : applique une fonction sur une collection
 - **find** : trouve une occurrence d'une valeur spécifiée
 - **find_if** : trouve une occurrence par le biais d'un prédicat,
 - **count** : trouve le nombre d'occurrences d'une valeur dans une séquence
 - **count_if** : trouve le nombre d'occurrences dans une séquence par le biais d'un prédicat
 - **Accumulate** : somme sur une collection
 - **max_element** : trouve l'élément max d'une séquence
 - **min_element** : trouve l'élément minimal d'une séquence
 - **Equal** : compare 2 séquences entre elles.

Algorithmes standards 2/3

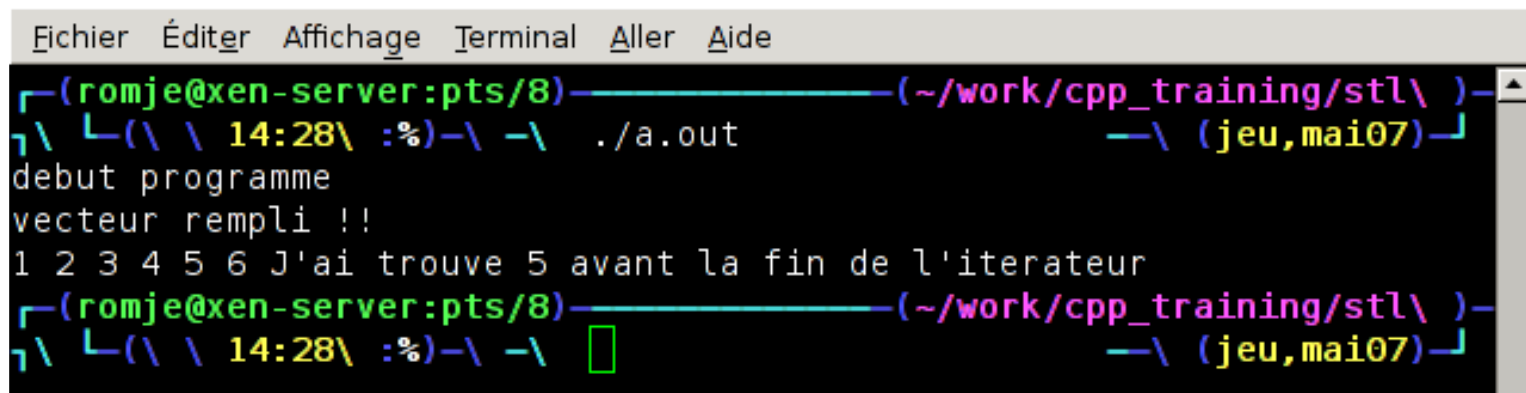
- Liste des algorithmes modifiant la collection :
 - **transform** : applique une transformation
 - **copy** : copie d'une séquence dans une autre
 - **replace** : remplacement dans une séquence
 - **replace_if** : remplacement conditionné
 - **remove** : suppression
 - **remove_if** : suppression conditionnée
 - **reverse** : basculement des index des objets
 - **random_shuffle** : repositionnement aléatoire des éléments
 - **fill** : remplissage d'une séquence
 - **generate** : remplissage par application d'une fonction .

Algorithmes standards 3/3

- Liste des algorithmes relatifs au tri :
 - **sort** : tri.
 - **stable_sort** : tri en préservant l'ordre relatif des éléments avec des valeurs équivalentes.
 - **binary_search** : vérifie l'existence d'un élément sur un ensemble d'éléments obligatoirement triés
 - **nth_element** : partitionne un ensemble d'éléments, en positionnant le nième, de telle sorte que tous les éléments devant le nième sont inférieurs ou égaux à lui et que tous les éléments qui le suivent sont supérieurs ou égaux à lui.

Exemples de mise en œuvre des algorithmes

```
1  #include <iostream>
2  #include <vector>
3  #include <string>
4  #include <algorithm>
5  #include <iterator>
6  using namespace std;
7  int main(int argc, char** argv){
8      const int TAILLE=6;
9      int values[TAILLE]={1,2,3,4,5,6};
10     vector<int> v(values, values+TAILLE); //initi du vecteur a partir du tableau...
11     cout<<"vecteur rempli !! " << endl;
12     ostream_iterator<int> sortie(cout, " "); // cree un iterator sur un flux pointant sur la
        console
13     copy(v.begin(), v.end(), sortie); // copie le contenu du vecteur via l'iterateur vers la
        console
14     cout<<endl; //pour faire joli!!
15     vector<int>::iterator iter = find(v.begin(), v.end(), 5);
16     if(iter!=v.end())
17         cout<<"J'ai trouve 5 avant la fin de l'iterateur"<< endl;
18 }
```

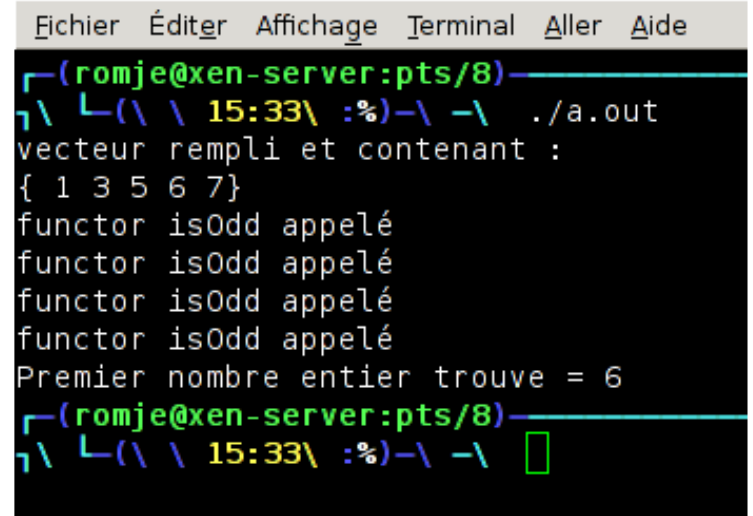


```
Fichier Éditer Affichage Terminal Aller Aide
romje@xen-server:pts/8) (~/.work/cpp_training/stl)
└─\ 14:28\ :%)─\ ─\ ./a.out ─\ (jeu, mai07)─┐
debut programme
vecteur rempli !!
1 2 3 4 5 6 J'ai trouve 5 avant la fin de l'iterateur
romje@xen-server:pts/8) (~/.work/cpp_training/stl)
└─\ 14:28\ :%)─\ ─\ ┐ ─\ (jeu, mai07)─┐
```

Illustration

- Le code suivant va chercher le premier entier pair dans un conteneur (vector).

```
1  #include <iostream> #include <vector> #include <string>
2  #include <algorithm>
3  using namespace std;
4  bool isOdd(int value){
5      cout<<"functor isOdd appelé"<<endl;return (value%2==0);}
6  int main(int argc,char** argv){
7      vector<int> values;
8      values.push_back(1);
9      values.push_back(3);
10     values.push_back(5);
11     values.push_back(6);
12     values.push_back(7);
13     cout<<"vecteur rempli et contenant :\n" << "{";
14     vector<int>::const_iterator iter;
15     for(iter=values.begin();iter!=values.end();++iter)
16     cout<<" " << *iter;
17     cout<<"}"<<endl;
18     iter= find_if( values.begin() , values.end() ,isOdd );
19     if(iter!=values.end())
20     cout<<"Premier nombre entier trouve = " << *iter << endl;}
```



```
Fichier Éditer Affichage Terminal Aller Aide
(romje@xen-server:pts/8)
(romje@xen-server:pts/8) 15:33 ./a.out
vecteur rempli et contenant :
{ 1 3 5 6 7}
functor isOdd appelé
functor isOdd appelé
functor isOdd appelé
functor isOdd appelé
Premier nombre entier trouve = 6
(romje@xen-server:pts/8)
(romje@xen-server:pts/8) 15:33
```

La STL

Algorithmes, prédicats, functors et binders

Définition

- La **STL** est très puissante de par la variété de ses conteneurs et du nombre d'algorithmes fournis. Elle peut l'être plus encore avec l'introduction de la notion de **binders**.
- Définition :
 - Les binders permettent d'utiliser les algorithmes standards de la **STL** (à 2 arguments) en tant que fonctions à un argument.
 - Ils suivent le design pattern **Adapter**.

Exemple introductif

- Comment trouver les éléments d'une bibliothèque par le biais de leur ISBN ? Ici cherchons les éléments étranges (sans ISBN)

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <string>
5  class book{
6  private:
7      string isbn,author,collection,editor,title;
8  public:
9      book(string _number,string _title):title(_title),
10         isbn(_number){}
11      book(string _title):title(_title){}
12      string getIsbn() const{return isbn;}
13      string getTitle() const {return title;}
14 };
15 class emptyisbnpredobj {
16 public:
17     bool operator () ( const book &book ) {
18         return book.getIsbn() == "";
19     } };
```

Exemple introductif

```
1  bool emptyisbnpredfunction( const book &book ) {
2      return book.getIsbn() == "";
3  }
4  int main(int argc, char** argv){
5      book b1("autant en emporte le vent");
6      book b2("Guerre et Paix", "A302B13");
7      book b3("Cahiers du Programmeur J2EE");
8      vector<book> biblio;
9      biblio.push_back(b1);
10     biblio.push_back(b2);
11     biblio.push_back(b3);
12     // 2 façons de faire la recherche par une classe template ou via le predicat
13     std::vector<book>::iterator found = std::find_if( biblio.begin(), biblio.end(),
14         emptyisbnpredfunction );
15     std::vector<book>::iterator found2 = std::find_if(biblio.begin(), biblio.end(), emptyisbnpredobj
16         ());
17     if(found!=biblio.end())
18         std::cout << "Livre trouve = " << (*found).getTitle() << endl;
19 }
```

Fichier Éditer Affichage Terminal Aller Aide

```
(romje@xen-server:pts/4)----- (~/.work/cpp_training/binders\ )-
^L (\ \ 15:54\ :%) -\ -\ ./intro ----- (dim, mai 24) ^L
Livre trouve = autant en emporte le vent
(romje@xen-server:pts/4)----- (~/.work/cpp_training/binders\ )-
^L (\ \ 15:54\ :%) -\ -\ [ ] ----- (dim, mai 24) ^L
```

Exemple introductif

- Le code précédent montre la puissance des algorithmes de la **STL**.
- il utilise 2 solutions pour résoudre un même problème
 - la première par le biais d'un **Functor** , (une classe simplement dotée d'un opérateur () redéfini et renvoyant un **bool**),
 - la seconde utilisant une fonction membre renvoyant elle aussi un **bool**
- Ce code est séduisant mais va favoriser une programmation où le développeur codera beaucoup de portions de code inutiles et lourdes à maintenir.

Utilisation de binders

- Les algorithmes sont donc prévus pour fonctionner sur une collection d'objets.
- Si le code précédent marche bien , il n'est guère évolutif (recherche hardcodée) et surement redondant vis à vis de la STL !
- Comment faire en sorte de se dispenser de l'écrire ?

```
int num_values = count_if (values.begin(), values.end(),  
                           bind2nd (greater<int>(), 5));
```

l'expression se lit :

lions le deuxième paramètre `greater<int>` à 5

Utilisation

```
1  int main (){
2      typedef std::vector<int>          Vector;
3      typedef std::equal_to<Vector::value_type> EqualTo;
4      const Vector::value_type arr [] = { 1, 2, 3, 4, 5 };
5      const Vector v1 (arr, arr + sizeof arr / sizeof *arr);
6      const Vector::value_type x (3);
7      const std::binder1st<EqualTo> equal_to_3 =
8          std::bind1st (EqualTo (), x);
9      const Vector::const_iterator it1 =
10         std::find_if (v1.begin (), v1.end (), equal_to_3);
11      const Vector::const_iterator it2 =
12         std::find_if (v1.begin (), v1.end (), std::bind1st (EqualTo (), x));
13      const Vector::const_iterator it3 =
14         std::find_if (v1.begin (), v1.end (), std::bind2nd (EqualTo (), x));
15      const Vector::size_type n =
16         std::count_if (v1.begin (), v1.end (), std::bind2nd (EqualTo (), x));
17      std::ostream_iterator<Vector::value_type> out (std::cout, " ");
18      std::cout << "The vector { ";
19      std::copy (v1.begin (), v1.end (), out);
20      std::cout << "} contains " << n << " element equal to "
21          << x << " at offset " <<
22          it1 - v1.begin () << ".\n";
23      const bool success = 1 == n && it1 == it2 && it1 == it2 && *it1 == x;
24      return success ? 0 : 1;
25 }
```

Ce qu'il faut retenir

- outil puissant,
- réduction du volume de code à écrire,
- permet de bénéficier de la puissance des algorithmes de la STL,
- syntaxe déroutante à priori,
- idée généralisée par la librairie **Boost** : **:Bind**.

La STL

STL et la performance

Généralités

- Il est bon de dire quelques mots des bonnes pratiques de la STL liées à la performance. A ce titre les conseils suivant constituent le BA.BA :
 - préférer la méthode **empty()** à un test de nullité de la méthode **size()**.
 - ne choisissez pas une implémentation au hasard ceci peut avoir des effets très pervers sur votre application,
 - le choix de l'implémentation ne peut se faire sans une connaissance de l'implémentation utilisée et donc de la structure de données sous jacente,
 - il n'y a aucune structure de données miracle mais simplement un choix de types bien adaptés à des contextes et peu efficaces dans d'autres.

Bien utiliser les vecteurs

De part leur vocation de tableau à taille dynamique il est :

- bon de les utiliser pour des accès direct (par l'index),
- bon d'utiliser la méthode **reserve()** de manière à éviter les retailages mémoire,
- préférable de ne pas trop itérer sur ce type de collections.
- préférable de ne pas faire des ajouts/suppressions au milieu du conteneur.

Bien utiliser les listes

leur double chaînage permet aux listes de :

- proposer des temps d'accès constants,
- permet des insertions/suppressions rapides,
- sont bien adaptées à des parcours séquentiels.
- mal adaptées à l'accès direct.

Bien utiliser les deque

Implémentées en tant que listes de tableaux les **deque** sont donc caractérisées par :

- des temps d'accès rapides
- une insertion/suppression au début/fin rapide
- une insertion/suppression au milieu : lente

Ce qu'il faut retenir

La STL offre des composants puissants et performants mais :

- pose le problème du choix des conteneurs,
- s'avère peu souple en cas de changement d'implémentation d'un conteneur,
- réclame une implication du programmeur quant au choix du conteneur.

Boost

Le projet Boost

La librairie Boost:présentation

- Librairie Open Source
- Issue d'un travail communautaire né des carences de la version ANSI/ISO 98
- Plébiscité par ses utilisateurs et soutenu par divers organismes (facultés, sociétés privées)
- Collection de projets permettant de répondre à un besoin ponctuel et précis
- Code de très haut niveau influençant le langage et en partie intégré dans les versions du standard C++.

- Thèmes abordés:
 - Smart pointers
 - Librairie de graphes
 - Sérialisation d'objets
 - Manipulation de flux, filesystems...
 - Gestion des paramètres passés en ligne de commande,
 - Expressions régulières,
 - Threading....

Boost

Outils utiles

Boost::lexical_cast

- Conversion rapide et safe de string vers un type et vice versa

```
void log_message(const std::string &);  
  
void log_errno(int yoko) {  
    log_message("Error " +  
        boost::lexical_cast<std::string>(yoko) + ": " +  
    strerror(yoko));  
}
```

- Plusieurs syntaxes pratiques
- Plus performante que stringstream

Boost::regex

- Manipulation de chaîne à l'aide d'expression régulière
 - Matching
 - Extraction/remplacement de sous-chaînes

```
bool validate_card_format(const std::string& s) {  
    static const boost::regex e("(\\d{4}[- ]){3}\\d{4}");  
    return regex_match(s, e);  
}
```

- Supporte l'unicode

Boost::filesystem

- Construction, manipulation de path
- Portable
- Query du filesystem (existence, tailles, attributs, ...)
- Erreurs via des exceptions

```
boost::filesystem::path p("/tmp");  
try {  
    if (boost::filesystem::exists(p)) {  
        // do something on the filesystem  
    }  
} catch (const filesystem_error& ex) {  
    cout << ex.what() << endl;  
}
```

Boost's string algorithms

- Fonctions classiques de manipulation de string
 - Trim
 - Join
 - Replace
 - Split
 - Uppercase/lowercase
 - Ends_with()
 - ...
- `#include <boost/algorithm/string.hpp>`

Boost::format

- Formatage de chaîne à la manière de `printf()`
 - Sans risque de débordement mémoire
- Erreurs via des exceptions
- Syntaxe particulière :

```
For(int i=0; i < names.size(); ++i) {  
    cout << format("%1%, %2%, %|30t|%3%\n") % names[i] %  
    surname[i] % tel[i];  
}
```

```
// Marc-François Michel, Durand, +33 (0) 123 456 789  
// Jean, de Lattre de Tassigny, +33 (0) 987 654 321
```

- `#include <boost/format.hpp>`

Multi threading

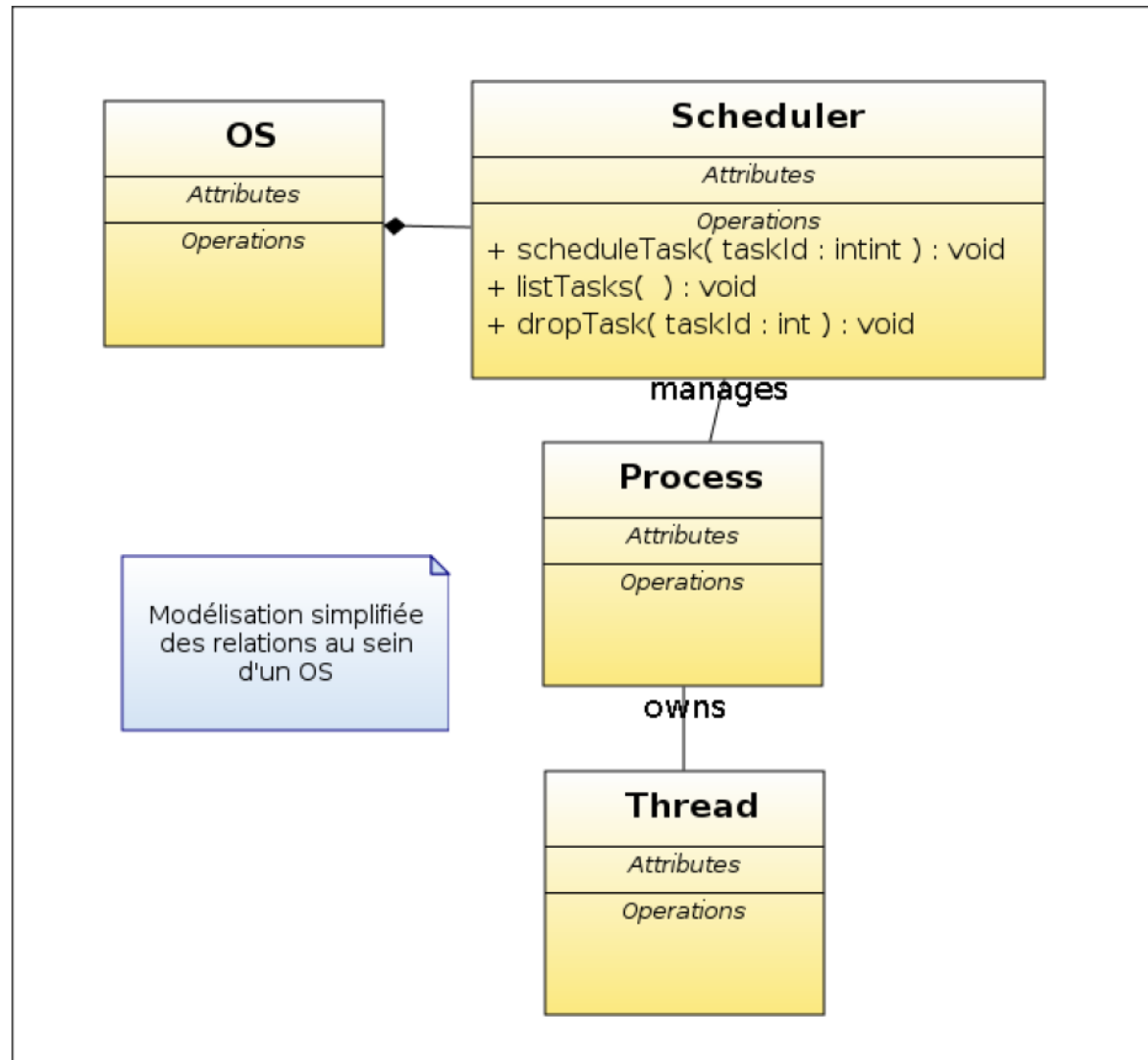
Concepts

- Parallélisation de tâches au sein d'un programme de manière à apporter un gain en confort ou performance.
 - pouvoir continuer à lire ses mails pendant que l'on rapatrie les
 - nouveaux messages en tâche de fond,
 - gérer diverses requêtes clientes en parallèle pour un serveur HTTP,
 - etc. . .
- Notion proche de la notion de processus on appelle parfois les threads : processus légers

Le besoin

- Pourquoi le besoin de cette notion de threads ?
 - les processus sont lourds à créer même sous Unix
 - le nombre de processus gérables par une machine est limité,
 - la communication entre processus est assez complexe IPC.

Représentation d'un thread



Concepts de base

- En tant que sous processus au sein d'un processus les threads partagent donc des ressources communes ce qui se traduit dans le code par un accès dit concurrent aux attributs de vos classes.
- Ceci donne lieu à des soucis en cas de non précaution :
 - lecture de valeurs incohérentes,
 - doubles écritures,
 - etc. . .

Patterns et idiomes de programmation

- Dans le monde de la programmation concurrente on se tourne donc vers des solutions standards vis à vis de ces problèmes :
 - le design pattern Rendez Vous,
 - certaines structures de données comme les sémaphores, barrières et autres mutex

Idées reçues et anti patterns

- Il est bon de couper court à certaines rumeurs :
 - utiliser un thread ralentit un programme mais permet de faire autre chose en cas d'attente (entrée sortie),
 - les problèmes de threads sont très durs à diagnostiquer et nécessitent un outillage idoine,
 - les threads ne sont pas complexes à mettre en place mais complexes à déboguer et à concevoir (plus d'aspect séquentiel).
 - attention au caractère non déterministe de certains bugs
 - l'usage d'un débogueur standard peut camoufler des bugs liés au threading,
 - attention aux priorités attribuées par le scheduler, il est extrêmement dangereux de dépendre de celles-ci pour obtenir un bon fonctionnement.

Multi threading

Utiliser la librairie threads

Stratégie de codage d'un thread

- Utiliser des threads revient principalement à :
 1. identifier et isoler les tâches devant être accomplies en tant que threads,
 2. les encapsuler dans des *functors* (fonctions sans arguments)
 3. instancier des threads instances de la classe **boost : :thread** ou **std::thread**
 4. identifier les ressources critiques accédées par divers threads et les protéger

Example

```
1  #include <boost/thread/thread.hpp>
2  #include <iostream>
3  void hello()
4  {
5      std::cout << "Hello world, I'm a thread!" << std::endl;
6  }
7  int main(int argc, char* argv[])
8  {
9      boost::thread thrd(&hello);
10     thrd.join();
11     return 0;
12 }
```

Commentaires sur l'exemple

- L'exemple réclame quelques précisions :
 - remarquer les fichiers d'en-tête nécessaires,
 - remarquer la signature de la méthode **hello** qui est un *functor*.
 - remarquer le constructeur de la classe **boost : :thread** prenant en paramètre l'adresse du *functor* **hello**.
 - la méthode d'instance **join()** de la classe **boost : :thread** permet d'attendre la complétion de la tâche à exécuter avant de rendre la main.

Compilation et lancement de l'exemple

```
Fichier Éditer Affichage Terminal Aller Aide
└─(romje@xen-server:pts/1)───(~/work/cpp_training/threads\ )─┐
└─(ven,mai22)─┘
└─(romje@xen-server:pts/1)───(~/work/cpp_training/threads\ )─┐
└─(ven,mai22)─┘
g++ sample1.cc -lboost_thread -lpthread -o sample1
└─(romje@xen-server:pts/1)───(~/work/cpp_training/threads\ )─┐
└─(ven,mai22)─┘
./sample1
Hello world, I'm a thread!
└─(romje@xen-server:pts/1)───(~/work/cpp_training/threads\ )─┐
└─(ven,mai22)─┘
```

Les mutex

- Le code suivant illustre comment protéger une ressource critique à l'aide d'un mutex.

```
1  #include <boost/thread/mutex.hpp>
2  #include <boost/thread/thread.hpp>
3  #include <iostream>
4  boost::mutex io_mutex; // The iostreams are not guaranteed to be thread-safe!
5  class counter
6  {
7  public:
8      counter() : count(0) { }
9
10     int increment() {
11         boost::mutex::scoped_lock scoped_lock(mutex);
12         return ++count;
13     }
14 private:
15     boost::mutex mutex;
16     int count;
17 };
18 counter c;
```

... à suivre

Les mutex

- Suite du code

```
1  void change_count()
2  {
3      int i = c.increment();
4      boost::mutex::scoped_lock scoped_lock(io_mutex);
5      std::cout << "count == " << i << std::endl;
6  }
7  int main(int, char*[])
8  {
9      const int num_threads = 4;
10     boost::thread_group thrds;
11     for (int i=0; i < num_threads; ++i)
12         thrds.create_thread(&change_count);
13     thrds.join_all();
14
15     return 0;
16 }
```

Commentaires

- Dans ce programme 1 seule ressource critique (le compteur) donc 1 seul mutex garantissant l'intégrité de celle-ci. 4 threads sont instanciés en prenant en paramètre le *functor* accédant à notre ressource critique.
- Autre ressource critique artificielle, le flux de sortie **cout** non « **thread safe** » garanti lui aussi par un mutex dédié.

Boost

Tests Unitaires

- Robustesse du code
 - mettre son application à l'épreuve
 - Compte rendu exhaustif de l'application, chose importante avant de la livrer !
 - Eviter les tests manuelles incomplets et gourmands en temps
- Non Regression
 - Je ne comprends pas pourquoi, hier ca marchait!
 - Petite perte de temps à écrire des tests mais ENORME gain en temps de Débuggage !
 - Plus un bug est détecté tôt, moins il faudra d'énergie pour le corriger

- Mauvaises Pratiques :
 - Eviter de mettre dans le même dossier/fichier
 - les tests et l'application Test Triviaux = Perte de temps
 - Eviter les effets de Bords !
 - Ne pas donner d'ordre
- Bonnes Pratiques
 - Une Classe, Une fonction (voir une variable) = Un Test
 - Regrouper les tests suivant leurs caractéristiques...
 - Tester la Réussite, L'échec, et la cohérence.
 - Mettre l'application en difficulté
 - Documenter !

- Utilisation des macros
- `BOOST_AUTO_TEST_CASE`
- `BOOST_CHECK` = continue les autres tests cases
- `BOOST_REQUIRE` = STOP directement

```
#define BOOST_TEST_DYN_LINK  
#define BOOST_TEST_MODULE  
Hello  
#include <boost/test/unit_test.hpp>
```

```
int add(int i, int j)  
{  
    return i + j;  
}
```

```
BOOST_AUTO_TEST_CASE(universe  
InOrder)  
{  
    BOOST_CHECK(add(2, 2) == 4);  
}
```

- **Autres macros d'assertion**
- **BOOST_CHECK_MESSAGE**: allows you specify a custom failure message as a second argument. You can pass a string, or any type supporting the << operator.
- **BOOST_CHECK_EQUAL**: checks two arguments for equality using ==. Improves upon the normal check in the above examples by showing the actual values when the assertion fails.
- **BOOST_CHECK_THROW**: checks that an expression causes a specified type of exception to be thrown.

- Suites

- Utilisation des macros :
 - BOOST_AUTO_TEST_SUITE
 - BOOST_AUTO_TEST_CASE
 - BOOST_AUTO_TEST_SUITE_END

```
int add(int i, int j)
```

```
{  
    return i + j;  
}
```

```
BOOST_AUTO_TEST_SUITE(Maths)
```

```
BOOST_AUTO_TEST_CASE(universeInOrder)
```

```
{  
    BOOST_CHECK(add(2, 2) == 4);  
}
```

```
BOOST_AUTO_TEST_SUITE_END()
```

```
BOOST_AUTO_TEST_SUITE(Physics)
```

```
BOOST_AUTO_TEST_CASE(specialTheory)
```

```
{  
    int e = 32;  
    int m = 2;  
    int c = 4;
```

```
    BOOST_CHECK(e == m * c * c);
```

```
}
```

```
BOOST_AUTO_TEST_SUITE_END()
```

- Suites

- Utilisation des macros :
 - BOOST_AUTO_TEST_SUITE
 - BOOST_AUTO_TEST_CASE
 - BOOST_AUTO_TEST_SUITE_END

```
int add(int i, int j)
```

```
{  
    return i + j;  
}
```

```
BOOST_AUTO_TEST_SUITE(Maths)
```

```
BOOST_AUTO_TEST_CASE(universeInOrder)
```

```
{  
    BOOST_CHECK(add(2, 2) == 4);  
}
```

```
BOOST_AUTO_TEST_SUITE_END()
```

```
BOOST_AUTO_TEST_SUITE(Physics)
```

```
BOOST_AUTO_TEST_CASE(specialTheory)
```

```
{  
    int e = 32;  
    int m = 2;  
    int c = 4;
```

```
    BOOST_CHECK(e == m * c * c);
```

```
}
```

```
BOOST_AUTO_TEST_SUITE_END()
```

- **Fixtures**
- Utilisation de FIXTURE dans les macros de suites
- Utilisation d'une structure ou classe:
 - Faire le setup avec le constructeur
 - Faire le teardown avec le destructeur

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Fixtures
#include <boost/test/unit_test.hpp>
struct Massive
{
    int m;
    Massive() : m(2) {
        BOOST_TEST_MESSAGE("setup mass");
    }

    ~Massive() {
        BOOST_TEST_MESSAGE("teardown mass");
    }
};
BOOST_FIXTURE_TEST_SUITE(Physics, Massive)

BOOST_AUTO_TEST_CASE(specialTheory)
{
    int e = 32;
    int c = 4;

    BOOST_CHECK(e == m * c * c);
}
BOOST_AUTO_TEST_CASE(newton2)
{
    int f = 10;
    int a = 5;

    BOOST_CHECK(f == m * a);
}
BOOST_AUTO_TEST_SUITE_END()
```

Boost

Sérialisation

- Problématique :
 - sauver/restaurer l'état d'un objet
 - Sauvegarde sous quel format?
 - Binaire
 - Textuel
 - XML

- Sérialisation:
 - Action d'exporter dans un flux un objet en mémoire
 - Implique action opposée permettant de restituer en mémoire un objet depuis un flux
 - Réclame l'utilisation de données dites sérialisables!!!
 - Certaines entités n'ont aucun sens à être sérialisées!!!
 - Ainsi un descripteur de fichier ou un objet lié à un processus spécifique n'a pas de sens à être restauré par la suite

- Utilisations typiques de la sérialisation (concept)
 - Échanges d'objets entre machines (cas des serveurs d'applications en Java)
 - Stockage du contexte de l'application côté client (préférences utilisateurs par exemple)
- Introduit une nouvelle problématique
 - Versions de classes compatibles!!!

- Diverses solutions existent
 - Ajout de méthodes exportAs() dans chacune de vos classes à exporter
 - Intrusive!!!
 - impossible avec du code dont vous ne possédez pas les sources!!!
 - AOP (programmation orientée aspects)
 - Peu de solutions existent dans le monde C++ contrairement au monde Java
 - Design
 - Utilisation d'un design Pattern decorateur
 - Pas toujours aisée si pas d'accès au code de toutes les classes!!!

- Solution la plus propre et la plus standard:
 - Se reposer sur la librairie Boost!!!

- Cette librairie repose sur :
 - Concept de classes amies pour accès aux champs privés
 - Programmation générique (templates)
 - Redéfinition d'opérateurs
 - Design Pattern décorateur pour les nouveaux flux permettant d'écrire/lire un objet vers/depuis un fichier ou une socket...
 - Macro (dans le cas de la sérialisation d'objets ayant évolué dans le temps avec des méthodes distinctes save/load)

- Fonctionnalités offertes:
 - Gestion des numéros de version de classes,
 - Sérialisation transparente des collections de la STL ou des tableaux d'objets,

Annexes

C++ 11 en question

C++ 11

- Version courante du langage standardisée par l'ISO/ANSI date de 1998 avec diverses mises à jour mineures
- C++ 11 est le nom de code de la nouvelle version de la norme C++
 - Validée le 12 août 2011
 - Ancien nom C++0x
 - Nom officiel : ISO/IEC 14882:2011

- Les nouveautés :
 - Les énumérations fortement typées
 - Le « bug » des chevrons
 - Les listes d'initialisateurs
 - Le mot-clé auto
 - La boucle basée sur un intervalle
 - Initialisation d'un pointeur
 - Les fonctions anonymes et les fermetures
 - Les tableaux à taille fixe
 - Un nouveau type de conteneur : le tuple
 - Gestion du temps
 - Les initialisateurs d'attributs

- Les nouveautés (suite) :
 - Initialisation des données membres non-statique
 - Alias de templates
 - Constructeurs délégués
 - Les expressions constantes généralisées
 - Littérales définies par l'utilisateur
 - Déclarations étendues de l'amitié
 - Surcharges explicites de la virtualité
 - Les unions sans restrictions

C++ 11 - Les nouveautés

- Les énumérations fortement typées
 - Intérêt : Augmentation de la sécurité
 - Déclaration :

```
enum class Direction { HAUT, DROITE, BAS, GAUCHE };
```

- Utilisation :

```
Direction direction = Direction::HAUT;
```

- Attention :

```
std::cout << direction << std::endl; ERREUR
```

OK

```
std::cout << static_cast<int>(direction) << std::endl;
```

C++ 11 - Les nouveautés

- Les énumérations fortement typées
 - Evite cela :

```
int nombre = 5 + Direction::DROITE;
```

Pas d'opérations mathématiques avec des enum!

C++ 11 - Les nouveautés

- Le « bug » des chevrons
 - Avant :

```
std::vector<std::vector<int> > nombres;
```

**Pensez à mettre
un blanc**

- Sinon : ">>" confondu avec opérateur de flux

- Maintenant :

```
std::vector<std::vector<int>> nombres;
```

C++ 11 - Les nouveautés

- Les listes d'initialisateurs – Côté utilisateur

- Avant :

```
std::vector<int> nombres;  
nombres.push_back(1);  
nombres.push_back(2);  
nombres.push_back(3);  
nombres.push_back(4);  
nombres.push_back(5);
```

- Maintenant :

```
std::vector<int> nombres = { 1, 2, 3, 4, 5 };
```


C++ 11 - Les nouveautés

- Les listes d'initialisateurs – Côté utilisateur
 - Exemple :

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

int main() {
    std::map<int, std::string> nombres =
    {
        { 1, "un" },
        { 2, "deux" },
        { 3, "trois" }
    };
    std::cout << nombres[1] << std::endl;
}
```

C++ 11 - Les nouveautés

- Les listes d'initialisateurs – Côté créateur
 - Fonctions avec liste d'initialisateurs

```
#include <initializer_list>
void afficherListe(std::initializer_list<int> liste) {
    for(std::initializer_list<int>::iterator i(liste.begin()) ;
        i != liste.end() ;
        ++i)
    {
        std::cout << *i << std::endl;
    }
}
```

```
int main() {
    afficherListe({ 1, 2, 3, 4, 5 });
    return 0;
}
```

C++ 11 - Les nouveautés

- Le mot-clé auto
 - Inférence de type
 - Type déterminé à la compilation

```
auto nombre = 5;
```

Nombre vaut 5

- Doit être initialisé

```
auto nombre;
```

Erreur

- decltype pour plusieurs variables

```
auto variable = 5;  
decltype(variable) autreVariable;
```

- Utile pour code générique

C++ 11 - Les nouveautés

- Le mot-clé auto

```
#include <iostream>

template <typename T>
T maximum(const T& a, const T& b) {
    if(a > b) {
        return a;
    }
    else {
        return b;
    }
}
```

```
template <typename T>
T minimum(const T& a, const T& b) {
    if(a < b) {
        return a;
    }
    else {
        return b;
    }
}
```

```
int main() {
    int a(10), b(20);
    auto plusGrand = maximum(a, b);
    decltype(plusGrand) plusPetit = minimum(a, b);
    std::cout << "Le plus grand est : "
                << plusGrand << std::endl;
    std::cout << "Le plus petit est : "
                << plusPetit << std::endl;
    return 0;
}
```



```
int main() {
    // int a(10), b(20);
    double a(10.5), b(20.5);
    auto plusGrand = maximum(a, b);
    decltype(plusGrand) plusPetit = minimum(a, b);
    std::cout << "Le plus grand est : "
                << plusGrand << std::endl;
    std::cout << "Le plus petit est : "
                << plusPetit << std::endl;
    return 0;
}
```

C++ 11 - Les nouveautés

- La boucle basée sur un intervalle
 - Avant :

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

using namespace std;

void afficherElement(int element) {
    cout << element << endl;
}

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for_each(nombres.begin(), nombres.end(), afficherElement);

    return 0;
}
```

C++ 11 - Les nouveautés

- La boucle basée sur un intervalle
 - Maintenant

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

using namespace std;

int main() {
    vector<int> nombres = { 1, 2, 3, 4, 5 };
    for(int &element : nombres) {
        cout << element << endl;
    }

    return 0;
}
```

Exemple
avec une
map

```
map<int, string> nombres = { { 1, "un" },
                             { 2, "deux" }, { 3, "trois" } };
for(auto i : nombres) {
    cout << i.first << " => " << i.second << endl;
}
```

C++ 11 - Les nouveautés

- La boucle basée sur un intervalle

```
vector<double> v;  
for (double d : v) { cout << d << endl; } // syntaxe de base  
for (auto d : v) { cout << d << endl; } // utilisation du mot clé auto  
for (auto& d : v) { ++d; } // référence pour modifier les éléments  
for (auto const& d : v) { + +d; } // passage par référence constante  
  
// liste d'initialisation  
for (auto x : { 1, 2, 3, 5, 8, 13, 21, 34 }) { cout << x << endl; }
```

C++ 11 - Les nouveautés

- Initialisation d'un pointeur

- Avant :

```
int *pointeur = NULL;
```

Ou

```
int *pointeur(0);
```

- Maintenant :

```
int *pointeur(nullptr);
```

```
int nul = NULL;
```

OK

```
int nul = nullptr;
```

Erreur

C++ 11 - Les nouveautés

- Les fonctions anonymes et les fermetures
 - Appelée aussi fonction lambda et ‘closures’
 - Définition :
 - « La fonction lambda est une fonction sans nom, inline, créée directement dans votre code »
 - « La fermeture est une fonction qui capture des variables à l’extérieur de celle-ci. »
 - Utilisation:
 - paramètre de fonctions
 - dans une variable
 - Intérêt :
 - Performance (inline)
 - Protection (pas de publication en header)
 - Concision

C++ 11 - Les nouveautés

- Les fonctions anonymes et les fermetures
 - Syntaxe lambda :

```
int main() {  
    // fonction lambda dans une variable.  
    auto p = [] () {  
        cout << "Je suis une fonction lambda" << endl;  
    };  
    p(); // Appel de la fonction  
    return 0;  
}
```

Ou

```
int main() {  
    int a = 2;  
    auto p = [&a] () { return a*a; };  
    p();  
    return 0;  
}
```

C++ 11 - Les nouveautés

- Les fonctions anonymes et les fermetures
 - Syntaxe générale:

[liste de capture] (paramètres) retour {code}

- liste de capture
 - liste de variables déclarées hors de la lambda et qui seront accessibles dans la lambda ;
- paramètres (optionnel)
 - paramètres qui seront envoyés par l'algorithme ;
- retour (optionnel)
 - type retourné par la lambda ;
- code
 - corps de la fonction lambda.

C++ 11 - Les nouveautés

- Les fonctions anonymes et les fermetures
 - Exemple avec la STL :

```
vector<int> v = {50, -10, 20, -30};  
  
// tri selon la valeur absolue  
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a)<abs(b); });  
// maintenant, v contient { -10, 20, -30, 50 }
```

- 2 paramètres de type 'int' : 'a' et 'b'
- renvoie un booléen
- Type de retour :
 - Généralement automatiquement déterminé ('void' si rien)
 - Peut être explicité :

```
// type de retour fixe  
generate(indices.begin(),indices.end(),[&count]() ->int { return ++count; });  
  
// type de retour déterminé avec decltype  
generate(indices.begin(),indices.end(),[&count]() ->decltype(++count) { return ++count; });
```

C++ 11 - Les nouveautés

- Les fermetures
 - Les paramètres capturés par la lambda :

```
vector<int> indices(v.size());  
int count = 0;  
generate(indices.begin(), indices.end(), [&count]() { return ++count; });
```

- Syntaxe :
 - [] : capture aucune variable ;
 - [&] : capture toutes les variables par référence ;
 - [=] : capture toutes les variables par copie ;
 - [&count] : capture la variable count par référence ;
 - [x, &y] : x capturée par copie, y capturée par référence ;
 - Identique à : [=x, &y]
 - [=count] : capture la variable count par copie.

C++ 11 - Les nouveautés

- Les tableaux à taille fixe

- Avant :

```
int tableauFixe[5];
```

- Maintenant :

```
#include <array>
std::array<int, 5> tableauFixe;
```

- Utilisation identique aux autres conteneurs
 - Performances semblables aux tableaux statiques

Uniquement si
const

```
int const taille = 5;
std::array<int, taille> tableauFixe = { 1, 2, 3, 4, 5 };
```

```
for(int nombre : tableauFixe) {
    std::cout << nombre << std::endl;
}
```

C++ 11 - Les nouveautés

- Les tableaux à taille fixe

- Méthodes :

- size() : taille du tableau
 - [] : accéder à un élément (sans vérification des limites)
 - at() : accéder à un élément (avec vérification des limites)
 - Si hors limite : exception `std::out_of_range`

```
tableauFixe[0] = 10;  
tableauFixe.at(1) = 40;  
std::cout << tableauFixe[1] << std::endl;  
std::cout << tableauFixe.at(0) << std::endl;
```

- front() et back() : premier et dernier élément
 - empty() : indique si le tableau est vide
 - fill() : modifier la valeur de tous les éléments par une autre
 - Possibilité d'utiliser les itérateurs
 - Méthodes `begin()` et `end()` classiques

C++ 11 - Les nouveautés

```
//Création d'un tableau à taille fixe de cinq entiers.
std::array<int, 5> tableauFixe = { 1, 2, 3, 4, 5 };

//Affichage de tous ses éléments.
for(int nombre : tableauFixe) {
    std::cout << nombre << std::endl;
}

//Affichage de sa taille.
std::cout << "Taille : " << tableauFixe.size() << std::endl;

//Modification d'éléments.
tableauFixe[0] = 10;
tableauFixe.at(1) = 40;
//Affichage d'un élément précis sans vérification de limite.
std::cout << tableauFixe[1] << std::endl;
//Même chose sauf qu'il y a une vérification de limite.
std::cout << tableauFixe.at(0) << std::endl;
//Afficher le premier élément.
std::cout << tableauFixe.front() << std::endl;
//Afficher le dernier élément.
std::cout << tableauFixe.back() << std::endl;

if(not tableauFixe.empty()) {
    std::cout << "Le tableau n'est pas vide." << std::endl;
}

tableauFixe.fill(5); //Modifier tous les éléments pour la valeur 5.

for(std::array<int, 5>::iterator i(tableauFixe.begin()) ; i != tableauFixe.end() ; ++i) {
    std::cout << *i << std::endl;
}
return 0;
```


C++ 11 - Les nouveautés

- Un nouveau type de conteneur : le tuple

- Définition :

- Un tuple est une collection de dimension fixe d'objets de types différents.

- En-tête : `#include <tuple>`

- Exemple de déclaration :

```
std::tuple<int, double, std::string> nomTuple;
```

- Syntaxe générale :

```
std::tuple<type1, type2, ...> nomTuple;
```

- Initialisation :

```
std::tuple<int, double, std::string> personne(22, 185.4, "Jack");
```

C++ 11 - Les nouveautés

- Un nouveau type de conteneur : le tuple
 - Accéder aux éléments du tuple :

`std::get()`

```
std::get<0>(personne) = 23;  
std::cout << "Âge : " << std::get<0>(personne) << std::endl;
```

- Vérifier l'égalité des valeurs de 2 tuples :
 - opérateur ==

C++ 11 - Les nouveautés

- Un nouveau type de conteneur : le tuple

```
#include <iostream>
#include <string>
#include <tuple>

int main() {
    //Création d'un tuple.
    std::tuple<int, double, std::string> personne(22, 185.4, "Jack");
    std::get<0>(personne) = 23; //Modification d'un objet d'un tuple.

    //Affichage des objets du tuple.
    std::cout << "Âge : " << std::get<0>(personne) <<
        std::endl << "Taille : " << std::get<1>(personne) <<
        std::endl << "Prénom : " << std::get<2>(personne) <<
        std::endl;

    typedef std::tuple<int, double, std::string> tuple_personne;
    //Affichage de la taille du tuple.
    std::cout << "Taille du tuple : " <<
        std::tuple_size<tuple_personne>::value <<
        std::endl;
    tuple_personne jack(23, 185.4, "Jack");
    if(personne == jack) { //Comparaison de tuples.
        std::cout << "Les tuples sont identiques." << std::endl;
    }
    return 0;
}
```

C++ 11 - Les nouveautés

- Gestion du temps

- En-tête : `#include <chrono>`

- Obtenir le temps actuel :

```
std::chrono::time_point<std::chrono::system_clock> temps =  
    std::chrono::system_clock::now();
```

- Ou plus simplement :

```
auto temps = std::chrono::system_clock::now();
```

- Autres unités :

```
std::chrono::nanoseconds nbrNanoSecondes = (temps2 - temps1);  
std::cout << nbrNanoSecondes.count() << " nanosecondes." << std::endl;
```

C++ 11 - Les nouveautés

- Gestion du temps
 - Autres unités : problème des millisecondes

```
std::chrono::duration<double, std::milli> nbrMilliSecondes = (temps2 - temps1);  
std::cout << nbrMilliSecondes.count() << " millisecondes." << std::endl;
```

- Autres unités, les secondes :

```
std::chrono::duration<double> nbrSecondes = (temps2 - temps1);
```

C++ 11 - Les nouveautés

- Les initialisateurs d'attributs
 - D'une classe ou d'un autre type de donnée
 - Syntaxe :

```
struct Paire {  
    int nombre1;  
    int nombre2;  
};
```

```
Paire paire{ 5, 25 }; //Initialise nombre1 à 5 et nombre2 à 25.
```

- Possibilité de modification :

```
paire = { 3, 9 };
```

- Ou retour :

```
Paire getPaire() {  
    return { 10, 100 }; //Retourne une Paire dont nombre1 = 10 et nombre2 = 100.  
}
```

C++ 11 - Les nouveautés

- Les initialiseurs d'attributs

```
class Paire {  
    public:  
        Paire(int nombre1, int nombre2) : nombre1_(nombre1), nombre2_(nombre2) {}  
  
    private:  
        int nombre1_;  
        int nombre2_;  
};  
  
Paire getPaire() {  
    return { 10, 100 }; //Retourne une Paire dont nombre1_ = 10 et nombre2_ = 100.  
}  
  
int main() {  
    Paire paire{ 5, 25 }; //Initialise nombre1_ à 5 et nombre2_ à 25.  
    paire = { 3, 9 };  
    paire = getPaire();  
    return 0;  
}
```

C++ 11 - Les nouveautés

- Initialisation des données membres non-statique :
 - Avant :

```
class A {  
    static const int m1 = 7; // ok  
    const int m2 = 7; // erreur: non statique  
    static int m3 = 7; // erreur: non constant  
    static const int m4 = var; // erreur: n'est pas une expression constante  
    static const string m5 = "odd"; // erreur: n'est pas un type intégral  
};
```

- Maintenant :

```
struct A {  
    int i = 42;  
} a;  
  
// est équivalent à :  
  
struct A {  
    int i;  
    A() : i(42) {}  
} a;
```


C++ 11 - Les nouveautés

- Initialisation des données membres non-statique :
 - Avant :

```
class A {  
public:  
    A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}  
    A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}  
    A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor run") {}  
  
    int a, b;  
  
private:  
    HashingFunction hash_algorithm;  
    std::string s;  
};
```

- Devient :

```
class A {  
public:  
    A() {}  
    A(int a_val) : a(a_val) {}  
    A(D d) : b(g(d)) {}  
  
    int a = 7;  
    int b = 5;  
  
private:  
    HashingFunction hash_algorithm{"MD5"};  
    std::string s{"Constructor run"};  
};
```

C++ 11 - Les nouveautés

- Alias de templates

- Avant :

```
template<class T> typedef T* Ptr; // illégal
```

```
template<class T> Ptr { typedef T* type; }; // légale
Ptr<int>::type ip; // decltype(ip) = int*
```

- Maintenant :

```
template <class T> using Ptr = T*;
Ptr<int> ip; // decltype(ip) = int*
```

- spécialisations partielles

```
template<typename T>
using MyAllocVec = std::vector<T, MyAllocator>;
MyAllocVec<int> v; // std::vector<int, MyAllocator>

template<std::size_t N>
using StringArray = std::array<std::string, N>;
StringArray<15> sa; // std::array<std::string, 15>
```

C++ 11 - Les nouveautés

- Alias de templates : pas de spécialisation

```
template<typename T>
using MyAllocVec = std::vector<T, MyAllocator>;

template<typename T>
using MyAllocVec = std::vector<T*, MyPtrAllocator>; // erreur
```

- Mais utilisation de 'trait' :

```
template<typename T> // trait de base
struct VecAllocator { typedef MyAllocator type; };

template<typename T> // trait spécialisé
struct VecAllocator<T*> { typedef MyPtrAllocator type; };

template<typename T>
using MyAllocVec = std::vector<T, typename VecAllocator<T>::type>;
```

C++ 11 - Les nouveautés

- Constructeurs délégués
 - Avant :
 - Recopie du même code dans plusieurs constructeurs
 - Maintenant :
 - Possibilité d'appeler un constructeur dans un autre :

```
struct A {  
    A(int);  
    A(): A(42) { } // délègue au constructeur A(int)  
};
```

C++ 11 - Les nouveautés

- Les expressions constantes généralisées
 - Mot clé : 'constexpr'
 - Fonctionne sur :
 - Expression
 - Fonction
 - Constructeur
 - Evaluation lors de la compilation

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

Flags operator& (Flags f1, Flags f2) { return Flags( int(f1) & int(f2) ); }
constexpr Flags operator| (Flags f1, Flags f2) { return Flags( int(f1) | int(f2) ); }

void f(Flags x)
{
    switch (x) {
        case bad: break;
        case fail&eof: break; // erreur, ne peut être évalué à la compilation
        case bad|eof: break; // ok, expression constante
        default: break;
    }
}
```

C++ 11 - Les nouveautés

- Les expressions constantes généralisées
 - Permet des calculs à la compilation :

```
// version expression constante
constexpr unsigned constexpr_pow(unsigned base, unsigned exp)
{
    return (exp==0) ? 1 : (base * constexpr_pow(base, exp-1));
}

// version template
template<unsigned base, unsigned exp> struct template_pow {
    enum { value = base * template_pow<base, exp - 1>::value };
};

template<unsigned base> struct template_pow<base, 0> {
    enum { value = 1 };
};

int main() {
    cout
        << "Avec constexpr: " << constexpr_pow(5, 3) << endl
        << "Avec template: " << template_pow<5, 3>::value << endl;
}
```

C++ 11 - Les nouveautés

- Littérales définies par l'utilisateur:
 - Avant :
 - Uniquement les qualificateurs pré-définis :
 - 1f : float
 - 1L : long
 - Maintenant :
 - Possibilité de définir ses propres qualificateurs
 - Utilisation de : ' operator' "

```
constexpr long double operator"" _degrees (long double d)
    { return d * 0.0175; }
long double pi = 180.0 degrees;
```

C++ 11 - Les nouveautés

- Littérales définies par l'utilisateur:
 - Exemples :

```
std::string operator"" _s (const char* p, size_t n) { return string(p,n); }

template<class T> void f(const T&);
f("Hello"); // utilise un const char*
f("Hello"_s); // utilise un std::string
```

```
Bignum operator"" _x(const char* p) { return Bignum(p); }
void f(Bignum);
f(123456789012345678901234567890_x);
// pas besoin d'écrire "123456789012345678901234567890"_x
```


C++ 11 - Les nouveautés

- Littérales définies par l'utilisateur:
 - Elles acceptent quatre types de paramètre en entrée :
 - les entiers, en utilisant les types unsigned long, long ou const char* ;
 - les nombres réels, en utilisant les types long, double ou const char* ;
 - les chaînes de caractères, en utilisant la paire d'arguments (const char*, size_t) ;
 - un caractère, en utilisant le type char.
 - Seuls les suffixes commençant par "_" sont autorisés. Les autres sont réservés pour un usage ultérieur par la norme.

C++ 11 - Les nouveautés

- Déclarations étendues de l'amitié :
 - Avant :
 - Pas de 'friend' possible dans les typedef et les template
 - Maintenant :

```
class C;
typedef C Ct;

class X1 {
    friend C; // OK : la classe C est amie
};

class X2 {
    friend Ct; // OK : la classe C est amie
    friend class D; // OK : on déclare une nouvelle classe qui sera amie
};

template <typename T>
class R {
    friend T;
};

R<C> rc; // OK : la classe C est amie de la classe R<C>
R<int> Ri; // OK : l'amitié est ignorée pour les types de base
```

C++ 11 - Les nouveautés

- Surcharges explicites de la virtualité :
 - Problème du masquage :

```
struct B {  
    virtual void f();  
    virtual void g() const;  
    virtual void h(char);  
    void k();  
};  
  
struct D : B {  
    void f(); // dérive de B::f()  
    void g(); // ne dérive pas de B::g() (fonction non constante)  
    virtual void h(char); // dérive de B::h(char)  
    void k(); // ne dérive pas de B::k() (B::k() n'est pas virtuelle)  
};
```

C++ 11 - Les nouveautés

- Surcharges explicites de la virtualité :
 - Deux nouveaux mots clé :
 - final : fonction non dérivable
 - override : la fonction doit dériver d'une autre fonction
 - Possibilité sur les classes:
 - final : la classe ne peut pas être dérivée.

```
struct E final { };  
struct F: E { }; // erreur : dérive d'une classe finale
```

C++ 11 - Les nouveautés

- Surcharges explicites de la virtualité :
 - Exemple :

```
struct B {  
    virtual void f() final;  
    virtual void g() const;  
    virtual void h(char);  
    void k();  
};  
  
struct D : B {  
    void f(); // erreur : B::f() ne peut être dérivable  
    void g() override; // erreur : ne dérive pas de B::g()  
    virtual void h(char); // dérive de B::h(char) mais génère une alerte  
    void k() override; // erreur : B::k() n'est pas virtuel  
};
```

C++ 11 - Les nouveautés

- Les unions sans restrictions :

- Avant :

```
union U {  
    int m1;  
    complex<double> m2; // erreur  
    string m3; // erreur  
};
```

- Maintenant :

```
union U1 {  
    int m1;  
    complex<double> m2; // ok  
};  
  
union U2 {  
    int m1;  
    string m3; // ok  
};
```

Attention !
Appel
manuel à
certaines
méthodes
nécessaire

C++ 11 - Les nouveautés

- Les unions sans restrictions :

```
class Widget { // 3 implémentations alternatives représentées par une union
private:
    enum class Tag { point, number, text } type;    // discriminant

    union {    // représentation
        point p;    // point possède un constructeur
        int i;
        string s;    // string possède un constructeur, un destructeur et permet la copie
    };

    widget& operator= (const widget& w) // nécessaire à cause de string
    {
        if (type==Tag::text && w.type==Tag::text) { // si les 2 widgets sont de type string
            s = w.s;    // on réalise une copie classique des string
            return *this;
        }

        if (type==Tag::text) s.~string(); // on détruit explicitement la chaîne

        switch (type==w.type) {
            case Tag::point: p = w.p; break; // copie normale
            case Tag::number: i = w.i; break; // copie normale
            case Tag::text: new(&s)(w.s); break; // placement new
        }
        type = w.type;
        return *this;
    }
};
```

Annexes

Liens

Pointeurs sur la toile

- Librairie Boost
 - <http://www.boost.org>
- Normalisation ANSI/ISO C++0.x
 - <http://www.open-std.org/jtc1/sc22/wg21/>
- Dr Dobbs Journal
 - <http://www.ddj.com>
- Site communautaire français
 - <http://cpp.developpez.com/>

Bibliographie

- Scott Meyers
 - Exceptional C++
 - et titres affiliés
- Herb Sutter
- James Coplien
- Bruce Eckel
 - Thinking in C++ (PDF disponible)
- Deitel & Deitel
 - C++ How to program
- Jacquelin Charbonnel - Dunod
 - Langage C++