



# PROGRAMMATION ORIENTEE OBJET EN C++





# Objectifs

Les objectifs de ce cours :

- Découverte de la POO
- Apprentissage du langage C++

Organisation :

- 7 séances de cours / TP
- 1 évaluation pratique





# Le langage C++ - sommaire

1. Intro
2. Exemple introductif
3. Classes
4. Surcharge opérateurs
5. Structures de données
6. Héritage
7. Exceptions / Boost
8. Gestion Mémoire





# Le langage C++

- Développé au début des années 1980 par Bjarne Stroustrup
- Au départ, extension du Langage C
- Programmation Procédurale et Orientée Objet
- Langage compilé
- Normalisé en septembre 2011 => C++11  
( ISO/IEC 14882:2011 )





# Le langage C++



Position Nov 2010	Position Nov 2009	Delta in Position	Programming Language	Ratings Nov 2010	Delta Nov 2009	Status
1	1	=	Java	18.509%	+0.14%	A
2	2	=	C	16.717%	-0.60%	A
3	4	↑	C++	9.497%	-0.50%	A
4	3	↓	PHP	7.813%	-2.36%	A
5	6	↑	C#	5.706%	+0.36%	A
6	7	↑	Python	5.679%	+1.01%	A
7	5	↓↓	(Visual) Basic	5.470%	-2.70%	A
8	13	↑↑↑↑↑	Objective-C	3.191%	+2.30%	A
9	8	↓	Perl	2.472%	-1.02%	A
10	10	=	Ruby	1.907%	-0.50%	A
11	9	↓↓	JavaScript	1.664%	-1.25%	A
12	11	↓	Delphi	1.638%	-0.49%	A
13	17	↑↑↑↑	Lisp	1.087%	+0.47%	A
14	23	↑↑↑↑↑↑↑	Transact-SQL	0.793%	+0.38%	A
15	15	=	Pascal	0.784%	+0.13%	A
16	29	↑↑↑↑↑↑↑↑	Ada	0.695%	+0.39%	B
17	36	↑↑↑↑↑↑↑↑	NXT-G	0.682%	+0.45%	B
18	14	↓↓↓	SAS	0.669%	-0.15%	B
19	30	↑↑↑↑↑↑↑↑	RPG (OS/400)	0.656%	+0.37%	B
20	12	↓↓↓↓↓	PL/SQL	0.655%	-0.25%	B





# Le langage C++



Position Aug 2013	Position Aug 2012	Delta in Position	Programming Language	Ratings Aug 2013	Delta Aug 2012	Status
1	2	↑	Java	15.978%	-0.37%	A
2	1	↓	C	15.974%	-2.96%	A
3	4	↑	C++	9.371%	+0.04%	A
4	3	↓	Objective-C	8.082%	-1.46%	A
5	6	↑	PHP	6.694%	+1.17%	A
6	5	↓	C#	6.117%	-0.47%	A
7	7	=	(Visual) Basic	3.873%	-1.46%	A
8	8	=	Python	3.603%	-0.27%	A
9	11	↑↑	JavaScript	2.093%	+0.73%	A
10	10	=	Ruby	2.067%	+0.38%	A
11	9	↓↓	Perl	2.041%	-0.23%	A
12	15	↑↑↑	Transact-SQL	1.393%	+0.54%	A
13	14	↑	Visual Basic .NET	1.320%	+0.44%	A
14	12	↓↓	Delphi/Object Pascal	0.918%	-0.09%	A--
15	20	↑↑↑↑	MATLAB	0.841%	+0.31%	A--
16	13	↓↓↓	Lisp	0.752%	-0.22%	A
17	19	↑↑	PL/SQL	0.751%	+0.14%	A
18	16	↓↓	Pascal	0.620%	-0.17%	A-
19	23	↑↑↑↑	Assembly	0.616%	+0.11%	B
20	22	↑↑	SAS	0.580%	+0.06%	B





# Webographie

- <http://www.cppreference.com/wiki/>
- <http://cpp.developpez.com/>
- <http://casteyde.christian.free.fr/cpp/cours>





# Les Bibliothèques Graphiques

- <http://www.qt.io>
- <https://www.wxwidgets.org/>
- <http://www.gtkmm.org>







# Exemple introductif : Equation

```
#include <iostream>    // pour l'affichage
#include <cmath>        // pour la ,racine carrée

int main()
{ double a,b,c ;      // déclaration coeffs
  a=1 ;               // initialisation coeffs
  b=5 ;
  c=6 ;
  double delta = b*b - 4*a*c ; // calcul discriminant
  double x1 = (-b - sqrt(delta))/ (2*a) ; // calcul solutions
  double x2 = (-b + sqrt(delta))/ (2*a) ;
  std::cout<< « racine1 : »<<x1<< « \n racine2 : »<<x2 ;
}
```





# Exemple introductif : Equation



Dans une optique de réutilisabilité et de maintenabilité de votre code

=> programmation modulaire

- Spécification dans equation.h
- Implémentation dans equation.cpp
- Utilisation dans test\_equation.cpp





# Fichier en-tête : equation.h

```
#ifndef EQUATION__H
#define EQUATION__H
#include <cmath>
#include <iostream>

/* Définition du type Equation */
typedef struct Equation {
    double coeffA, coeffB, coeffC; /* coefficients de l'équation */
    double x1, x2; /* racines de l'équation */
};
/* Déterminer les racines de l'équation du second degré.*/
void resoudre(Equation *eq);
void afficher (Equation eq) ;

#endif
```





# Fichier implémentation : equation.cpp



```
#include "equation.h"

void resoudre(Equation *eq)
{
    /* variable locale à la fonction resoudre */
    double delta =
        eq->coeffB * eq->coeffB - 4 * eq->coeffA * eq->coeffC;

    eq->x1 = (- eq->coeffB + sqrt(delta)) / (2 * eq->coeffA);
    eq->x2 = (- eq->coeffB - sqrt(delta)) / (2 * eq->coeffA);
}

void affiche ( Equation eq)
{
    std::cout<< « racine 1 : »<< eq.x1 ;    // affichage des résultats
    std::cout<< « \nracine 2 : »<<eq.x2 ;
```





# Fichier utilisateur : test\_equation.cpp



```
#include "equation.h"
```

```
Int main()
```

```
{
```

```
    Equation uneEquation ;          // declaration
```

```
    UneEquation.coeffA = 1 ; // initialisation des coeffs
```

```
    UneEquation.coeffB = 5 ;
```

```
    UneEquation.coeffC = 6 ;
```

```
    Resoudre ( &uneEquation) ; // calculer les racines
```

```
    Affiche(uneEquation) ;
```

```
    Return 0 ;
```

```
}
```





# Fichier en-tête : Equation.h

```
#ifndef EQUATION__H
#define EQUATION__H
#include <cmath>
#include <iostream>

class Equation {
public :
    double coeffA, coeffB, coeffC; /* coefficients de l'équation */
    double x1, x2; /* racines de l'équation */

    void resoudre();
    void afficher();
};
#endif
```





# Fichier implémentation : Equation.cpp



```
#include "Equation.h"
```

```
void Equation::resoudre()
```

```
{
```

```
/* variable locale à la fonction resoudre */
```

```
double delta = coeffB * coeffB - 4 * coeffA * coeffC;
```

```
x1 = (- coeffB + sqrt(delta)) / (2 * coeffA);
```

```
x2 = (- coeffB - sqrt(delta)) / (2 * coeffA);
```

```
}
```

```
Void Equation::afficher()
```

```
{
```

```
Std::cout<< « Racine 1 : »<< x1 ;    // affichage des résultats
```

```
Std::cout<< « \nRacine 2 : »<<x2 ;
```

```
}
```





# Fichier utilisateur : testEquation.cpp



```
#include "Equation.h"
```

```
Int main()
```

```
{
```

```
    Equation uneEquation ;          // declaration
```

```
    uneEquation.coeffa = 1 ; // initialisation des coeffs
```

```
    uneEquation.coeffb = 5 ;
```

```
    uneEquation.coeffc = 6 ;
```

```
    uneEquation.resoudre() ; // calculer les racines
```

```
    uneEquation.afficher() ; //affichage des résultats
```

```
    return 0 ;
```

```
}
```







# Approche Objet



«Les systèmes logiciels sont caractérisés au premier chef par les objets qu'ils manipulent, non par la fonction qu'ils assurent.

Ne demandez pas CE QUE FAIT LE SYSTÈME.

Demandez À QUI IL LE FAIT ! »

Bertrand MEYER





# POO



En Programmation Orientée Objet, 2 concepts-clés :

- Modularité  
=> encapsulation et masquage d'information (notion de classe)
- Extensibilité  
=> relation d'héritage





# POO



En Programmation Orientée Objet, on regroupera au sein d'une CLASSE des *attributs* et des *méthodes*.

Les *attributs* : l'état d' un objet

Ex : les coeffs, les solutions

Les *méthodes* : les actions pouvant être réalisées sur un objet de type Equation (unités de calcul...)

Ex : résoudre, afficher





# Les Classes



Une classe définit :

Un Module => ce sera donc une unité d'encapsulation et un  
« espace de nommage »

Un Type => qui permet de créer des objets statiques ou  
dynamiques





# Objets, Classes, Types



Quelques affirmations :

- Un objet est une instance d'une classe
- Un objet est une instance d'une et une seule classe
- Un objet a pour type le nom de sa classe mais peut avoir d'autres types (héritage)





# Classe (déclaration)

La déclaration d'une classe se fait dans un fichier d'en-tête (Matiere.h) :

```
class Matiere {  
    public :  
        int nbNotes;  
        float notes[10];  
        string nom ;  
        void saisirNotes();  
        void saisir();  
        void affiche();  
}; // fin de classe
```





# Classe (définition)

Le corps des fonctions sera définie dans un autre fichier (Matiere.cpp) :

```
#include « Matiere.h »  
void Matiere::saisir() {  
    ...  
    cout<<« nombre notes ? » <<endl;  
    cin>>nbNotes ;  
    saisirNotes();  
}  
void Matiere::saisirNotes(){  
    ...  
}
```





# Classe (utilisation)

```
#include « Matiere.h »  
  
int main(){  
  
    Matiere m1 ;  
  
    m1.saisir();  
    m1.affiche();  
  
    return 0;  
  
}
```







# Les fonctions

## Surcharge de fonctions :

- Possibilité d'avoir deux fonctions portant le même nom si les paramètres sont différents

=> signatures différentes

Point\* creer\_point(float x);

Point\* creer\_point(float x, float y );

Point\* creer\_point(float x, float y, float z);

- Le type de retour n'est pas pris en compte pour identifier la fonction





# Les constructeurs

Appelé automatiquement lors de la déclaration de l'objet :

- Initialisent les champs de l'objet
- Même identificateur que celui de la classe
- Pas de type de retour
- Accès public
- Peuvent être surchargés
- Si aucun constructeur dans la classe, le compilateur synthétise un constructeur par défaut(qui ne fait rien...)





# Les constructeurs

```
#include <ctime>
```

```
class Date {  
    public :
```

```
        int jour, mois, annee;
```

```
        // Constructeur sans paramètre  
        Date() ;
```

```
        //les attributs sont initialisés par la date courante du SE
```

```
        //Constructeur avec paramètres
```

```
        Date (int j, int m,int a) ;
```

```
        //les attributs sont initialisés avec les paramètres j, m, a
```

```
};
```





# Les constructeurs

```
#include « Date.h »
```

```
Date ::Date(){  
    time_t t;  
    time(&t);  
    tm date = *localtime( &t );  
    jour=date.tm_mday;  
    mois=date.tm_mon;    // valeur entre 0 et 11  
    annee=1900+date.tm_year;  
}
```

```
Date::Date(int j, int m, int a):jour(j),mois(m),annee(a) {}
```

```
...
```

```
int main()  
{
```

```
    Date date_actuelle;    // constructeur Date() appelé
```

```
    Date fete_nat(14,7,1789); // constructeur Date(14,7,1789)
```

appelé





# Les constructeurs



Peuvent posséder une liste d'initialisation :

- L'ordre d'initialisation de la liste doit suivre l'ordre de déclaration des attributs
- La liste d'initialisation est obligatoire lorsque les champs sont des constantes
- A privilégier si certains champs sont des objets





# Le destructeur

- Fonction membre du même nom que la classe précédée du caractère ~
- Pas de paramètre ni de type de retour
- Appelé à la destruction de l'objet, juste avant que la mémoire occupée par l'objet soit récupérée par le système
- Doit libérer la mémoire allouée dans la classe

```
ex : ~Segment(){  
    delete x1;  
    delete x2;}
```





# Les entrées-sorties



En C, utilisation des fonctions de <stdio.h> :

- saisie : scanf, fscanf, fgets, getchar,...

=> Flux d'entrée standard : stdin

- affichage : printf, fprintf, puts, putchar,...=> Flux de sortie standard : stdout

=> Flux d'erreurs standard: stderr





# Les entrées-sorties

En C++, utilisation des objets de `<iostream>` :

- saisie : opérateur d'injection `>>`  
=> Flux d'entrée standard : `std::cin` (type `istream`)
- affichage : opérateur d'extraction `<<`  
=> Flux de sortie standard : `std::cout`  
=> Flux d'erreurs standard: `std::cerr` (type `ostream`)  
`std::clog`







# Les entrées-sorties

Flot de sortie `std::cout` :

```
int i=123;
```

```
float f=3.14;
```

```
string s =« coucou »;
```

```
Point p ;
```

```
std::cout<<i;
```

```
std::cout <<"\n f= "<<f ;
```

```
std::cout<<s<<std::endl<<p<<std::endl ;
```





# Les entrées-sorties

Flot d'entrée `std::cin` :

```
int i;
```

```
float f;
```

```
string chaine;
```

```
std::cout<<" tapez la valeur de i et f:"<<std::endl ;
```

```
std::cin >> i >> f; // l'espace servira de séparateur
```

```
getline(cin, chaine, '\n') ;
```

```
// saisie d'une chaîne jusqu'au retour chariot ou longueur précisée
```

```
std::cin.get(); // saisie d'un caractère
```





# Espace de nommage



Zones de déclaration permettant de délimiter la recherche des noms des identificateurs par le compilateur

=> regrouper les identificateurs logiquement et éviter les conflits de noms entre plusieurs parties d'un même projet.

Ex : `using namespace std ;`  
(on peut utiliser `cout` directement plutôt que `std::cout` )





# Chaines de caractères

Depuis la STL, il existe une classe string équivalente à celle de Java :

```
#include <string>
string s1 =« bonjour », s2=« salut » ;
string s3 = s1+ s2;
if (s1==s2)
    cout<<s1;
if (s1.size() != 0)
    cout<< s1.substr(3) ; // affiche jour
getline(cin , s3 ); //saisie d'une chaine
```

avec espace

[www.enac.fr](http://www.enac.fr)

36





=> TP 1





# Gestion mémoire

En C, utilisation des fonctions malloc / free:

```
Point * p = (Point*)malloc(sizeof(Point));
```

...

```
free(p);
```

En C++, opérateur new et delete

```
Point * p = new Point;
```

```
p -> x = 2.0 ;
```

```
p -> afficher() ;
```

...

```
delete p;
```





# Gestion mémoire



Pour les tableaux :

```
int * pti = new int[3]; // réservation d'un tableau de 3 entiers  
Point * tab = new Point[20]; // réservation de 20 objets Point  
...  
delete[] pti ;      // ne pas oublier les [ ] !!  
delete [] tab ;
```





# Gestion mémoire

En C++,

```
Point p1 ;           //réservation dans la pile
Point *p2 = new Point ;
Point t1[3];         //3 appels successifs au constructeur sans paramètres
```

```
Point t2[2][2] ; // 4 appels ...
Point *t3 = new Point[3];
Point **t4 = new Point*[2];
    t4[0] = new Point[2] ;
    t4[1] = new Point[2] ;
    ...
```







# Référence

Une référence est un alias sur une variable :

- Initialisation obligatoire à la déclaration

Exemple :

```
int i=0;  
int &ieref=i;    // &ieref contient l'adresse de i  
ieref =1;        // idem que i=1;
```

- Permet dans une fonction de passer des arguments par adresse
- En Python, passage des objets en paramètres par référence





# Référence

*Version C++ :*

```
void permuter(int &x, int &y)
{
    int tampon =x;
    x=y;
    y=tampon;
}

Int main()
{
    int n1= 1,n2=2;
    permuter(n1,n2);
    cout<<n1<<endl<<n2;
    return 0;
}
```

*Version C:*

```
void permuter(int *x, int *j)
{
    int tampon = *x;
    *x=*y;
    *y=tampon;
}

Int main()
{
    int n1= 1,n2=2;
    permuter(&n1,&n2);
    printf(«%d %d »,n1,n2);
    return 0;
}
```





# Référence

En C++2011, on peut écrire :

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

int main()
{
    int tab[] = { 1, 2 ,3 ,4};
    for (int& val : tab)
        cout<<val<<endl;
    return 0;
}
```





# Encapsulation

Chaque caractéristique (attributs ou méthodes) d'une classe a un droit d'accès. Il existe 3 niveaux de droit d'accès :

- **private** : le nom du membre n'est connu que des fonctions membres et amies de la classe (par défaut).
- **protected** : idem que private mais ce membre est accessible à toutes les méthodes des classes dérivées.
- **public** : accessible par tous.





# Encapsulation



Intérêt :

Les droits d'accès permettent le masquage d'informations.

Règle :

Un attribut devrait toujours être déclaré private pour permettre à l'auteur de la classe de garantir l'intégrité des objets.

Conséquence :

Définition de méthodes (public) d'accès et de modification  
( getters and setters)





# Encapsulation



```
#ifndef EQUATION__H
#define EQUATION__H
#include <cmath>
#include <iostream>

Classe Equation {
    // attributs private
    double coeffA, coeffB, coeffC; /* coefficients de l'équation */
    double x1, x2; /* racines de l'équation */
public :
    void resoudre();
    void afficher();
};
#endif
```





# Encapsulation



```
#include "Equation.h"
```

```
Int main()
```

```
{
```

```
    Equation uneEquation ;          // declaration
```

```
    uneEquation.coeffa = 1 ; // droits d'accès insuffisants
```

```
    uneEquation.coeffb = 5 ; // => erreur à la compilation
```

```
    uneEquation.coeffc = 6 ;
```

```
    uneEquation.resoudre() ; // calculer les racines
```

```
    uneEquation.afficher() ; //affichage des résultats
```

```
    return 0 ;
```

```
}
```





# Encapsulation



```
#ifndef EQUATION__H
#define EQUATION__H
#include <cmath>
#include <iostream>

Classe Equation {
    // attributs private
    double coeffA, coeffB, coeffC; /* coefficients de l'équation */
    double x1, x2; /* racines de l'équation */
public :
    void resoudre();
    void afficher();
    void init(double a, double b, double c) ;
};
#endif
```







# Encapsulation



```
#include "Equation.h"
```

```
Int main()
```

```
{
```

```
    Equation uneEquation ;           // declaration
```

```
    uneEquation.init(1,5,6) ;
```

```
    uneEquation.resoudre() ; // calculer les racines
```

```
    uneEquation.afficher() ; //affichage des résultats
```

```
    return 0 ;
```

```
}
```





# Amitié



## Fonction amie :

- Fonction non-membre de la classe
- Accès à tous les membres (publics ou privés)
- Déclaration explicite dans la classe précédée du mot réservé *friend*
- L'amitié ne s'hérite pas :
  - ⇒ Si une classe B hérite de A, les amies de la classe A ne sont pas amies de la classe B





# Amitié

## Fonction amie :

```
class C {  
    int a;  
    friend void fonc_amie(C , int); // fonction amie  
    public :  
        void fonc_membre(int);  
};  
// fonction amie indépendante : pas de résolution de portée  
void fonc_amie(C c,int i) { c.a=i; }  
// fonction membre de la classe C : résolution de portée  
void C::fonc_membre(int j) { a=j; }
```





# Amitié

## Fonction amie :

```
class D {  
    float f;  
    friend void C::fonc_membre(int); // fonction amie  
                                     de classe  
    friend void fonc_amie(int); // fonction amie indépendante  
    ...  
};  
  
...  
void C::fonc_membre(int i) {  
    D d1;  
    .... d1.f .. // autorisé  
}
```





# Amitié



## Classe amie :

```
class C1; // déclaration incomplète mais nécessaire pour le compilateur
class C2 {
    friend C1; // toutes les fonctions membres de C1
               // sont amies de C2
    int i;
    ...
};

class C1 {
    void fonc1( C2 ...); // accès à i de C2
    void fonc2(C2 ...); // accès à i de C2
    ...
};
```





# Les membres statiques

Comme en java, on peut utiliser des attributs et des méthodes de classes :

- partagés par tous les objets de la classe
- un seul exemplaire par classe





# Les membres statiques

## Attributs statiques :

```
//déclaration ds Point.h  
class Point{  
    public:  
        static int nbPoints;  
    ...  
};
```

```
//initialisation ds Point.cpp  
int Point::nbPoints = 0;
```

```
// ds TestPoint.cpp  
Point p1;  
cout << p1.nbPoints;  
cout << Point::nbPoints ;
```





# Les membres statiques

## Fonctions membres statiques :

```
//déclaration ds Point.h  
class Point{  
    public:  
    static int combien() ;  
    ...  
};
```

```
//initialisation ds Point.cpp  
int Point::combien() {  
    static int nbPoints = 0;  
    return nbPoints++;  
}
```

```
// ds TestPoint.cpp  
Point p1;  
cout << p1.combien();  
cout << Point::combien() ;
```







# Const

- Pour rendre une variable constante
- Situé après la déclaration d'une méthode :
  - Modification des attributs de l'objet interdite
  - Fait partie de sa signature
  - Il est conseillé de qualifier const toute méthode qui peut l'être, cela élargit son champ d'application





# Const



```
int main()
{
    A x ;
    const A y ;
    x.f1() ; //ok
    x.f2() ; //ok
    y.f1() ; // interdit
    y.f2() ; //ok
```

```
class A {
    int a ;
    public :
        void f1() ;
        void f2() const ;
};
...
void A::f2() const {
    a = 0 ; // interdit
}
```





=> TP 2





# This

Pointeur sur l'objet sur lequel la fonction membre est invoqué

Déclaré par défaut dans toute classe :

**Matiere \*const this;**

En Java, this représente l'objet !!!





# Compilation conditionnelle

Pour éviter les inclusions multiples, concevoir ses propres fichiers d'en-tête de la façon suivante :

Dans « matiere.h » :

```
#ifndef _MATIERE_H  
#define _MATIERE_H  
    ... // contenu du fichier « matiere.h »  
#endif
```





# Les fonctions

Valeurs par défaut à certains paramètres :

- Spécifiées uniquement dans le prototype, pas dans la définition de la fonction

```
Point * creer_point(float x, float y=0, float z =0);
```

...

```
creer_point(3);
```

```
creer_point(3,5);
```

- Les paramètres facultatifs doivent forcément se trouver à droite





# Fonctions *inline*

Optimisation du code :

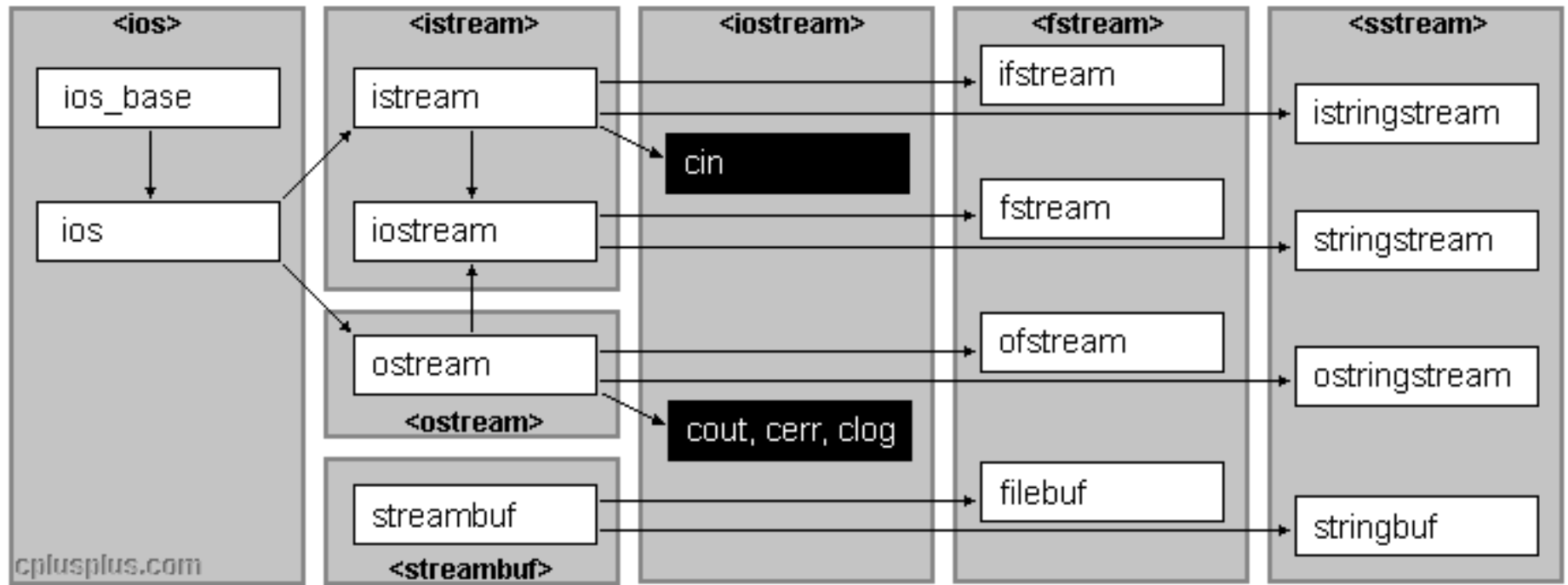
- Pas de rupture de séquence, le compilateur remplace l'appel de la fonction par le code de celle-ci
- A écrire dans les fichiers d'en-tête

```
inline int abs (int x) {  
    return x >= 0 ? x : -x ;  
}
```





# Gestion fichiers







# Gestion Fichiers



Lecture en utilisant *ifstream* :

```
ifstream ( const char * nom, openmode mode=in)
```

```
Ex :ifstream fic (« monfic »);  
    int a;  
    if (fic.is_open())  
{  
    fic >> a; // lire un entier  
    cout << "A = " << a;  
    fic.close() ;  
}
```





# Gestion Fichiers



Ecriture en utilisant *ofstream* :

`ofstream ( const char * nom, openmode mode=out|trunc)`

```
Ex : ofstream fic (« monfic »,ios_base::app);  
      int a = 5;  
      if (fic.is_open())  
      {  
        fic<<a<<endl; // ecriture de a en fin de fichier  
        fic.close() ;  
      }
```





# Surcharge des opérateurs

En C++, on peut redéfinir la plupart des opérateurs  
(sauf `.` `.*` `::` `?` `:` `sizeof` )

- soit pour les étendre à des objets

Complexe c1,c2,c3;

...

c3 = c2 + c1 ;

- soit pour changer l'effet d'opérateurs prédéfinis sur des objets

Segment s1,s2;

...

s1 = s2 ;





# Surcharge des opérateurs



Par une fonction membre :

*Complexe.h*

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

*main.cpp*

```
Complexe c1,c2,c3;  
...  
c3 = c2 + c1 ;//compris comme c3 = c2.operator+(c1)
```





# Surcharge des opérateurs



Par une fonction membre :

*Complexe.cpp*

```
#include "Complexe.h"

Complexe Complexe :: operator+(const Complexe& c)
{
    return Complexe (this->x+c.x, this->y+c.y);
}
```





# Surcharge des opérateurs



Par une fonction indépendante :

*Complexe.h*

```
class Complexe{  
    ...  
};  
Complexe operator+(const Complexe& a,const Complexe& b) ;
```

*main.cpp*

```
Complexe c1,c2,c3;  
...  
c3 = c2 + c1 ;//compris comme c3 = operator+(c2,c1)
```





# Surcharge des opérateurs

Par une fonction indépendante :

Complexe.cpp

```
#include "Complexe.h"

Complexe operator+(const Complexe& a,const Complexe& b)    {
    return Complexe (a.x+b.x, a.y+b.y);
}
```





# Surcharge d'opérateurs

- Si les deux possibilités existent, le compilateur soit privilégie la fonction membre, soit génère une erreur estimant l'expression ambiguë
- Du fait des conversions implicites, il est préférable de privilégier la fonction non-membre:

Complexe c1,c2;

c1 = c2 + 1;      //cast implicite de 1 en  
Complexe( 1,0)

c1 = 1 + c2;      // erreur 1 n'est pas un objet







# Surcharge d'opérateurs



Certains opérateurs ne peuvent être surchargés que par fonction membre :

ex :

=> opérateur d'affectation =

=> opérateur d'indexation [ ]

=> opérateur de pré/post incrémentation





# Surcharge d'opérateurs

*TabInt.h*

```
class TabInt{  
    int  taille; // dimension du tableau  
    int *tab, error;  
    public :  
        TabInt(int s=10) ;  
        ~TabInt() ;  
        int & operator[] (int i) ;  
};
```

*Main.cpp*

```
TabInt t(5);  
t[0] = 10;  
cout<<t[0] ;
```





# Surcharge d'opérateurs

*TabInt.cpp*

```
#include "TabInt.h"
```

```
TabInt :: TabInt(int s=10): taille(s), tab(new int[taille]) {}
```

```
TabInt :: ~TabInt() { delete [] tab; }
```

```
int & TabInt :: operator[](int i) {}  
{  
    if (i < taille)  
        return tab[i];  
    else  
        cerr << "indice trop grand " << endl;  
        return error;  
}
```





# Surcharge d'opérateurs

## Cas des opérateurs ++ et -- :

On définira l'opérateur :

`a.operator++()` pour une préincrémentation.

`a.operator++(int)` pour la postincrémentation.

Les appels resteront : `++a` pour `a.operator++()`

`a++` pour `a.operator++(0)`





# Surcharge d'opérateurs

*Complexe.h*

```
class Complexe{  
    ...  
    Complexe& operator ++() ;  
};
```

*Main.cpp*

```
Complexe c1, c2 ,c3 ;  
c3 = c2 + (++c1) ;
```





# Surcharge d'opérateurs

*Complexe.cpp*

```
Complexe& Complexe ::operator ++(){  
    x++ ;  
    y++ ;  
    return *this ;  
}
```





# Surcharge d'opérateurs



La surcharge des opérateurs d'injection << et d'extraction de données >> dans les flux ne peut se faire que par fonction indépendante :

*Point.h*

```
class Point {  
    double x,y ;  
    public :  
        double getX() const ;  
        double getY() const ;  
    friend istream& operator>>(istream& i, Point& p) ;  
...}; // fin de la classe  
ostream & operator<<(ostream& o,const Point& p) ;
```





# Surcharge d'opérateurs

*Point.cpp*

```
ostream& operator<<(ostream& o,const Point& p){  
    o<< p.getX()<<« , »<<p.getY()<<endl;  
    return o;  
}  
  
istream& operator>>(istream& i, Point& p){  
    i>>p.x>>p.y ;  
    return i ;  
}
```

*Main.cpp*

```
Point p1, p2, p3 ;  
cin>>p1>>p2>>p3 ;  
cout<<p1<<p2<<p3;
```







=> TP 3





# La Standard Template Library (STL)

La classe string

Des conteneurs :

=> Implémentations de grande qualité pour les structures de données les plus courantes

Des itérateurs :

=> Variables qui repèrent la position d'un élément dans un conteneur

Des algorithmes

=> Fonctions génériques qui permettent de faire des opérations sur des containers (tri, ...)





# Conteneurs

Structures de données appelées à contenir d'autres objets.

Classes *template* paramétrées par un type de données :

```
std::vector<int> t1 ; std::stack<Complex> t2 ;  
std::map<string,int> t3 ;
```

Inclusion du header correspondant :

```
#include <vector>  
#include <map>  
...
```





# Conteneurs

Classés en 2 catégories :

- Les séquences :

=> Ordre bien défini

Ex : vector , list , stack , deque

- Les conteneurs associatifs :

=> Chaque élément possède une clef

Ex : set , multiset, map , multimap





# Conteneurs

...

```
int main() {  
    vector<int> tabentiers;  
    int temp;  
    cout << "donner un entier : " ;  
    cin >> temp;  
    tabentiers.push_back(temp); // insère la valeur de temp à la fin  
    cout << " il y a : " << tabentiers.size() << " éléments dans le tableau " << endl;  
    cout << " premier élément du tableau " << tabentiers.front() << endl;  
    cout << " premier élément du tableau " << tabentiers[0] << endl;  
    cout << " premier élément du tableau " << tabentiers.at(0) << endl;  
    ...  
}
```





# Conteneurs



Un vector utilise en interne un tableau alloué dynamiquement, qui sera donc ré-alloué :

- A chaque rajout
- A chaque insertion
- A chaque suppression

Accès aux éléments d'un vector :

`tabentiers[i]`

`tabentiers.at(i)` => out\_of\_range exception





# Conteneurs

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

int main() {
    list<int> listeentiers;
    int temp;
    cout << "donner un entier : " ;
    cin >> temp;
    listeentiers.push_back(temp);           // insère la valeur de temp à la fin
    cout << " il y a : " << listeentiers.size() << " éléments dans la liste " << endl;
    cout << " premier élément de la liste " << listeentiers.front() << endl;
    ...
}
```





# Conteneurs



A l'insertion dans le conteneur, l'objet est recopié !!!

```
Point p(2,2) ;  
list<Point> poly ;  
poly.push_back(p) ;  
p.setX(0) ;  
cout<<poly.front().getX()<<endl ;  
=> affiche 2
```







# Conteneurs



Si utilisation de pointeurs, seule l'adresse est copiée, pas de duplication de l'objet.

```
Point * p = new Point(2,2) ;  
list<Point*> poly ;  
poly.push_back(p) ;  
p->setX(0) ;  
cout<<poly.front()->getX()<<endl ;  
=> affiche 0
```





# Itérateurs

Les itérateurs sont des objets permettant d'accéder à tous les objets d'un conteneur donné

Ils s'utilisent comme des pointeurs

=> ++ , -- , \*

Chaque conteneur possède son propre type d'itérateur  
(constant ou pas)





# Itérateurs

```
vector<Point> tab(10);  
vector<Point>::iterator it ;  
// vector<Point>::const_iterator it; // pr un objet constant  
// list<Point>::iterator it; // pr une liste
```

```
for (it=tab.begin() ; it != tab.end(); it++){  
    cout << *it<<" "<<endl;  
    cout<<it->getX()<<endl ;
```

...

\*it : accès à la valeur.

it++ : incrémentation d'un itérateur





# Itérateurs C++ 2011

```
vector<Point> tab(10);  
list<Point> traj;
```

```
for (const Point &p : traj) {  
    cout << p << " "<<endl;  
    cout<< p.getX()<<endl ;  
}    ...
```

```
for (Point &p : tab) {  
    p.init(0,0) ;
```





# Itérateurs

AJOUT / SUPPRESSION D'UN ELEMENT :

```
vector<Point> tab(10);
```

```
vector<Point>::iterator it = tab.begin() ;
```

```
Point temp (0,5) ;
```

```
tab.insert(it+2, temp) ;
```

```
for (it=tab.begin() ; it != tab.end(); it++)
```

```
    if(p->getX()<0)
```

```
        it = tab.erase(it) ;    //attention si dernier élément
```

```
    ...
```





# Algorithmes

Fonctions *template* optimisées permettant d'effectuer différentes opérations , en faisant appel aux itérateurs :

- Tri
- Recherche
- Suppression , remplacement, ...





# Algorithmes

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

```
int main() {
    vector<int> tabentiers;
    ...remplissage du tableau

    sort(tabentiers.begin(), tabentiers.end()); // tri du tableau en
                                                utilisant l'opérateur <

    vector<int>::iterator it;
    for (it = tabentiers.begin() ; it != tabentiers.end(); it++)
        cout << *it << " " << endl;
```





# Algorithmes

```
bool decroissant( int a,int b) { return a > b; }
```

```
int main() {  
    vector<int> tabentiers;  
    ...remplissage du tableau
```

```
    // tri du tableau en utilisant la fonction decroissant  
    sort(tabentiers.begin(), tabentiers.end(), decroissant);
```

```
    vector<int>::iterator it;  
    for (it = tabentiers.begin() ; it != tabentiers.end(); it ++)  
        cout << *it << " " << endl;
```

```
    ...
```







=> TP 4

