**Enseignement : GI 62 & GI 63 - C++ -** Programmation orienté objet (C++)

**Objectif :** *Ce module se situe dans la continuité des modules (ii12) et UEL RAN Informatique et permet d'acquérir les bases de la modélisation et de la programmation orientées objet. Le langage C++ est pris comme exemple.*

**Avertissement :** Ce cours suppose la connaissance et la maîtrise du langage C

# CONTENU DU COURS

.

BIBLIOGRAPHIE ........ 1
HISTORIQUE ........ 2
A.INTRODUCTION A LA POO ........ 2
B. LANGAGE C++ ........ 3
1.INCOMPATIBILITES C/C++ ........ 4
2. E/S BASEES SUR LA NOTION DE FLOT ........ 4
3. SPECIFICITES NON OBJET DU C++ ........ 6
5. FONCTION MEMBRE ........ 17
6. FONCTION AMIE ........ 22
7. HERITAGE SIMPLE ........ 24
8. HERITAGE MULTIPLE ........ 25
9. CONTENU DES TDs ........ 31
10. CONTENU DES TPs ........ 35
12. CARACTERISTIQUES DE JAVA ........ 41
13. LES DIFFERENCES JAVA - C++ ........ 42

# BIBLIOGRAPHIE

(disponible à la BU de l'université d'Evry)

| | Titre | Auteur | Année |
|---|---|---|---|
| 1. | Comment programmer en C++ | Deitel, Harvey M. | 1999 |
| 2. | Le C++ efficace | Meyers, Scott | 1999 |
| 3. | Visual C++ 6. | Williams, Mickey | 1999 |
| 4. | UML et C++ :guide pratique | Lee, Richard C. | 1998 |
| 5. | L'essentiel du C++ | Lippman, Stanley B. | 1999 |
| 6. | Le langage C++ | Stroustrup, Bjarne | 1998 |
| 7. | C++ facile | Vaiser, Didier | 1992 |
| 8. | C++ FAQs | Cline, Marshall | 1999 |
| 9. | The C++ Programming Language | Stroustrup, Bjarne | 1997 |
| 10. | Programmer en langage C++ | Delannoy, Claude | 2000 |

# HISTORIQUE

| | |
|---|---|
| • Langage C, 1ère version , Kernighan & Ritchie, Bell Labs AT&T | 1972 |
| • C ANSI/ISO | 1990 |
| • Langage C++, 1ère version, Bjarne Stroustrup, Bell Labs AT&T | 1982 |
| • C++ ANSI/ISO | 1998 |

# A.INTRODUCTION A LA POO

**A.1 Programmation Structurée & Programmation Orientée Objet**

Programmation Structurée

Equation de Wirth :

**Algorithme + Données = Programme**

Programmation Orientée Objet :

Equation de la POO :
**Méthodes + Données = Objet**

**A.2 Programmation Orientée Objet**

- **Méthode**
  - Formuler une stratégie de résolution du problème
  - Isoler les objets faisant partie de la solution et identifier les opérations sur les objets
  - Regrouper avec chaque objet les opérations qui lui sont associées
  - En déduire la spécification des modules
- **Concepts Orientés-Objet**
  - Encapsulation
  - Abstraction
  - Modularité
  - Classification
  - Typage
  - Hiérarchie – Héritage

**A.3 Démarche d'une méthode orientée objet**

- Définir le problème
- Développer une stratégie informelle de résolution
- Définir les objets et leur attributs
- Identifier les opérations sur les objets
- Etablir les interfaces
- Implémenter les objets

Répéter ce processus récursivement si nécessaire

# B. LANGAGE C++

**Introduction**

**C++ = C + C + POO + POO**

C : Langage C

C : Différences/ Langage C
Définitions de fonctions
Compatibilités entre pointeurs

POO : Particularités de C++ non Objet
- Commentaires
- Surdéfinition de fonctions
- Définition de fonction « inline »
- Emplacements quelconques des déclarations
- Nouvelles fonctions de gestion mémoire

POO : Spécificités de C++ Objet
- Classe/objet
- Héritage
- Fonctions amies (originalité)
- Conversions explicites/implicites de classes
- Encapsulation des données
- Constructeurs de classe
- Surdéfinition d'opérateurs
- E/S basées sur la notion de flot (originalité)

**B.2 Terminologie de base :**

**Programme**={ fonction, bloc, variable, constante, type, instruction, opérateur}
**Fonction** : correspond à un emplacement mémoire(adresse de début d'un sous programme),
**Bloc** : ensemble de définitions de variables et d'instructions délimité par {…},
**Variable** : correspond à un emplacement mémoire, elle est statique(globale) ou dynamique(locale),
**Constante** : " qui est le code machine du programme,
**Type** : définit les caractéristiques d'un ou plusieurs **noms.**
**Nom** : est un identificateur qui représente une fonction, ou une variable, ou une constante,
**Définition** : présentation des caractéristiques d'un ou plusieurs noms et réservation de la place mémoire à ceux-ci.
**Déclaration** (prototype) : présentation des caractéristiques d'un ou plusieurs noms,
**Classe** : type qui regroupe des données et des fonctions (méthodes) et qui précise les droits d'accès à ces derniers,
**Objet** : instance de la classe (variable).

# 1.INCOMPATIBILITES C/C++

### 1.1 Définitions de fonction

Compatibles avec la norme ANSI

### 1.2 Prototypes en C et en C++ de fonctions

Def. : déclaration de l'entête d'une fonction dans un autre fichier source que celui où elle est définie.

En C : Les prototypes ne sont pas obligatoires
En C++ : Ils le sont car le C++ permet la surdéfinition de fonction
Dans le cas ou la fonction n'a pas de paramètre ou n'en renvoie pas il faut utiliser le mot clé void .
Ex : void f(void)

### 1.3 Qualificatif "const"

Const : permet de rendre une variable à lecture seule. Ce mécanisme est fort utile pour le passage de paramètre par adresse à une fonction

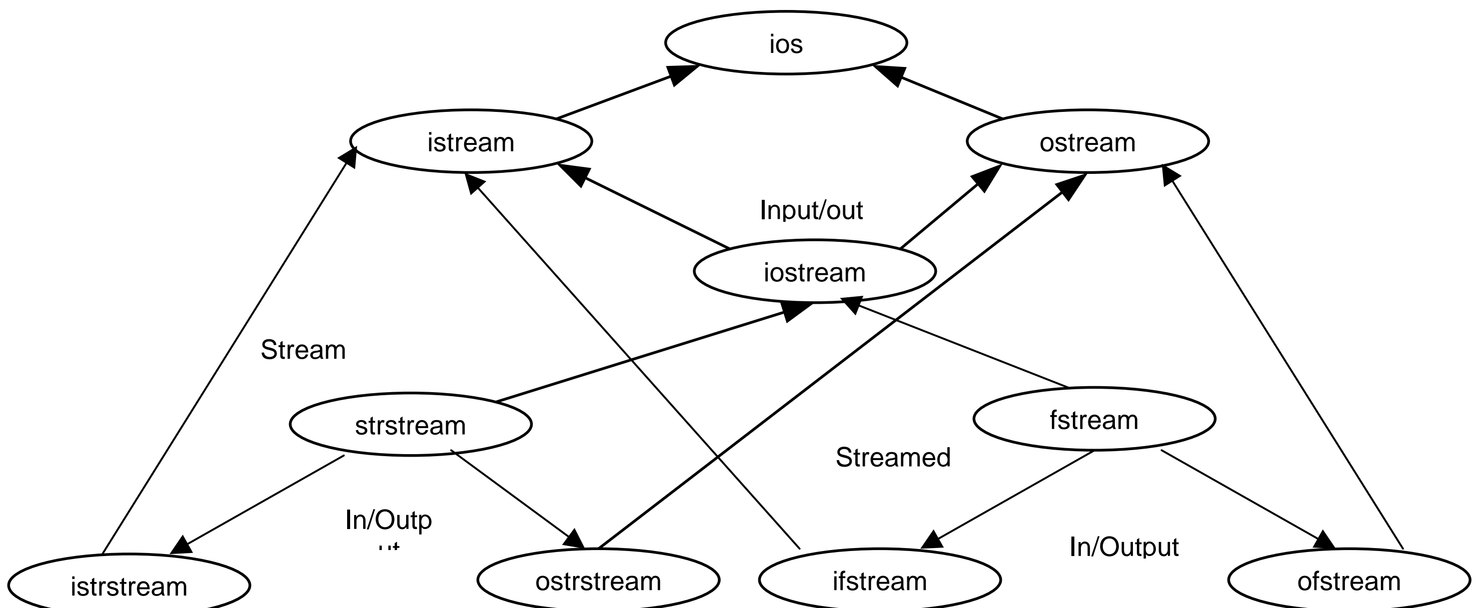
# 2. E/S BASEES SUR LA NOTION DE FLOT

**simple** : permettent de s'affranchir du formatage
**extensible** : de nouveaux types peuvent être définis sous forme de classe.

**Descripteurs & opérateurs**

Flot d'entrée : mot clé **cin** avec opérateur **>>** (*stdin*)
Flot de sortie : mot clé **cout** avec opérateur **<<** (*stdout*)
Flot d'erreur : mot clé **cerr** avec opérateur **<<** (*stderr*)

**Hiérarchie simplifiée des classes d'E/S par flots :**
22



**Exemples d'E/S par flots :**

```
// E/S par flots sur un fichier en ecriture
#include <iostream.h>
#include <fstream.h>
main()
{
  fstream fout("f", ios::out);
  fout<<"salut ceci est test\n";
}
/* Résultat
$ more f
salut ceci est un test
*/

//________________________________________________________
// E/S par flots sur un fichier en lecture
#include <iostream.h>
#include <fstream.h>
main()
{
int i;
  fstream fin("f", ios::in);
  fin >> i;
  cout << i;
}
/* resultat
$ ./fin
*/
//________________________________________________________
// E/S par flots en lecture en memoire
#include <iostream.h>
#include <strstream.h>
main()
{
int a;
char  b;
char c[6]="2245a";
  istrstream strin(c, sizeof(c));
  strin >>a >>b;
  cout << sizeof(c) << "\t" << a << "\t" << b;
}
/*Resultat
$./strin
6 2245 a*/
//________________________________________________________
// E/S par flots en ecriture en memoire
#include <iostream.h>
#include <strstream.h>
main()
{
int a=2345;
char  b='d';
char c[6];
  ostrstream strout(c, sizeof(c));
  strout <<a <<b;
  for (int i=0; i<6; i++)cout << c[i];
}
/* Resultat
$ ./strout
2345d*/
```

# 3. SPECIFICITES NON OBJET DU C++

3.1 Commentaires

3.2 Définitions généralisées et notion de portée

3.3. Passage de paramètres à une fonction

3.4 Surdéfinition de fonction

3.5 Opérateurs de gestion dynamique de mémoire

3.6 Fonction "inline"

**3.1 Commentaires**

// : précèdent un commentaire. Celui-ci est valable jusqu'à la fin de la ligne

**3.2 Définitions généralisées et notion de portée d’un nom**

Une définition ou une déclaration peut se faire à tout endroit du programme, néanmoins il est recommandé de les regrouper au début de chaque bloc.

La **portée** d’un nom est la région du programme ou le nom peut être utilisé.

Règle de la portée : un nom est toujours associé à la définition le concernant la + proche syntaxiquement (comme en C).

La portée d’un nom peut être :
**Locale** : noms définis dans un bloc, la portée du nom est le bloc.
**Fonction** : les arguments d’une fonction sont considérés comme des noms locaux à celle-ci. Les étiquettes sont des noms qui ont une portée limitée à la fonction (“goto <etiquette>” n’est possible qu’a l’intérieur d’une fonction – non recommandé.
**Globale** : noms définis à l’extérieur de tout bloc. Leur portée est celle du fichier.

**3.3. Passage de paramètres à une fonction**

En C, les arguments d'une fonction sont passés par adresse

**Exemple 3.3a**:

```
…
int carre_add(int * a) ;
int carre_add(int * a)
{
return ((*a)*(*a)) ;
}
```

En C++, les arguments d'une fonction sont transmis par référence.

**Exemple 3.3.b:**

```
…
int carre_ref(int & a) ;
int carre_ref(int & a)
{
return(a*a) ;
}
```

**3.4 Surdéfinition de fonction**

La surdéfinition ou la surcharge d'une fonction est la définition multiple de celle-ci avec des paramètres au nombre et au type qui diffèrent.

Les opérateurs de base sont surchargés car ils peuvent s'appliquer à des variables de type différent.
Le compilateur applique le schéma de conversion :

**Char, short, long → int → double ← float**

La conversion int → double n'est effectuée que si cela est nécessaire, les autres se font automatiquement.

Principe de la surcharge d'une fonction : le compilateur établi la meilleure correspondance entre les paramètres effectifs (appel de fonction) et les paramètres formels (fonction appelée). Si plusieurs fonctions sont candidates, alors il y a ambiguïté.

La surdéfinition de fonctions est aussi appelé *polymorphisme*.

**Exemple 3.4a – Surdéfinition d'une fonction en tenant compte de la nature des arguments effectifs :**
…
**Exemple 3.4b – Surdéfinition d'une fonction en tenant compte du nombre des arguments:**
…
**Exemple 3.4c – Surdéfinition d'une fonction avec mise en place des conversions implicites :**
……

**3.5 Opérateurs de gestion dynamique de mémoire**

En C : Les fonctions malloc() et free() sont utilisées pour allouer et libérer dynamiquement de la mémoire.
**Exemple 3.5a :**
**…**

En C++ : Ces 2 fonctions sont remplacées par 2 opérateurs **new** et **delete** afin d'optimiser le temps d'exécution des programmes .

Syntaxe de new :
**New type**
**New type[n]**
**New type[n][m]…**

Syntaxe de delete :
**Delete <adresse>**
**Exemple 3.5b :**
**…**

### 3.6 Fonction "inline"

```
inline f(…)
{
…
}

f(…) ;  // f est considérée comme une macro
```

inline permet de transformer une fonction en macro. Elle doit figurer dans le même fichier où est appelée la fonction f. L'avantage est l'optimisation du temps d'exécution, l'inconvénient est que f perd le statut de fonction et ne peut par voie de conséquence être compilée séparément des autres fonctions : Il faut placer sa définition dans un fichier entête (.h) à inclure dans tout fichier où f est utilisée.

# 4. CLASSE ET OBJET

### 4.1 Définition des notions d'objet et de classe

**Objet** : généralisation de la notion de variable composée ou structure. Un objet est constitué de données et de fonctions.
**Classe** :extension de la notion du type *struct* du langage C. Une classe décrit les données et les fonctions(méthodes) d'un objet. Les constituants d'une classe sont appelés membres.

**Exemple 4.1a : struct une classe particulière**

```
// Utilisation d'une structure en C
#include <iostream.h>
struct point {
int x,y;
void init(int, int, point &);
void affiche(point);
};
void init(int a, int b,   point &w)
{
w.x=a; w.y=b;
}
void affiche(  point & w)
{
  cout << w.x << "\t" << w.y << endl;
}
main()
{
  point p;
init(1,2,p);
affiche(p);
  point p2;
init(3,4,p2);
affiche(p2);
}
```

**Exemple 4.1b : Utilisation d’une structure en C++**

```
#include <iostream.h>
// Tous les membres sont publiques
struct point {
int x,y;
  //public :
void init(int, int);
void affiche();
};
// :: est l’opérateur de portée qui permet d’accéder à des noms normalement hors de portée

void point::init(int a, int b)
{
x=a; y=b;
}
void point::affiche()
{
cout << x << "\t" << y << endl;
}
```

**Exemple 4.1c : Utilisation d’une classe**

```
#include <iostream.h>
// Les membres d’une classe peuvent être privée(private) ou publique(public),
// private est pris par défaut
class point {
int x,y;
public :
void init(int, int);
void affiche();
};
void point::init(int a, int b)
{
x=a; y=b;
}
void point::affiche()
{
cout << x << "\t" << y << endl;
}
main()
{
  point p;
p.init(1,2);   // ‘.’ : permet de référencer un membre
p.affiche();
  point q;
q.init(3,4);
q.affiche();
}               // p.x  est interdit car x est une donnée privée , seules les méthodes de la classe point peuvent
//accéder aux données privées
// Exo. : ajouter la fonction membre void deplace(int, int)
```

**4.2 OBJETS**

**4.2.1 Constructeur et destructeur**

**Constructeur :**
- est une fonction membre qui porte le même nom que sa classe,
- est appelé après l'allocation de l'espace mémoire destiné à l'objet,
- ne renvoie pas de valeur (pas même *void* , ne doit figurer devant sa déclaration ou sa définition).

**Destructeur** :
- est une fonction membre portant le même nom que sa classe, précédé du symbole tilde (~),
- est appelé avant la libération de l'espace mémoire associé à l'objet
- ne peut pas comporter d'arguments et il ne renvoie pas de valeur (aucune indication de type ne doit être prévue).

**4.2.1.1 Constructeur/ destructeur statique**

```
// Exemple 4.2.1.1a : utilisation d'une classe avec un constructeur usuel
// statique avec surdéfinition de ce dernier
#include <iostream.h>
class point {
int x,y;
public :
point();
point(int);
point(int , int );
void affiche();
};

point::point()
{x=y=0;}
point::point(int a)
{x=y=a;}
point::point(int a, int b)
{x=a; y=b;}
void point::affiche()
{cout << x << "\t" << y << endl;}

main()
{point p;
p.affiche();
  point q(1);
q.affiche();
point r(1,2);
r.affiche();
}
/* Resultat d'execution
[root@localhost 4.2const]# ./point
0       0
1       1
1       2*/
```

```
// Exemple 4.2.1.1b: utilisation d'une classe avec un constructeur usuel
// statique en utilisant des paramètres par défaut
#include <iostream.h>
class point {
int x,y;
static int ctr ;   // compteur d'objets
public :
point(int =0, int =0); // déclaration des paramètres par défaut
void affiche();
~point();
};

int point : : ctr = 0;  // init. A 0 du nb d'objets
point ::point(int a, int b)
{x=a; y=b; ctr++;}

void point::affiche()
{cout << x << "\t" << y << endl;}

point :: ~point()
{ctr-- ;
cout << " Nombre d'objets restants : " << ctr << endl ;}
main()
{
  point p;
p.affiche();
  point q(1);
q.affiche();
point r(1,2);
r.affiche();
} // le destructeur est appelé automatiquement
/* Résultat d'exécution
[root@gsc3 TEST]# ./a.out
0       0
1       0
1       2
 Nombre d'objets restants : 2
 Nombre d'objets restants : 1
 Nombre d'objets restants : 0
*/
```

**4.2.1.2 Constructeur/ destructeur dynamique**

```
// Exemple 4.2.1.2a :utilisation d'une classe avec un constructeur usuel
// dynamique avec surdéfinition de ce dernier + comptage du nbre d'objets - application du
destructeur
#include <iostream.h>
#include <stdlib.h>
#define NMAX 4
class entier {
  int nbval;
  int *val;
  static int ctr;  //compteur d'objets, variable statique(globale)
public :
entier();
entier(int);
~entier();
void affiche();
static void compte();};  // fonction statique
```

```
int entier :: ctr = 0;  // init. A 0 du nb d'objets

entier::entier()
{val = new int [nbval=NMAX];
for(int i=0; i<NMAX; i++) val[i]=abs((char)rand());
ctr+=nbval;}

entier::entier(int n)
{val = new int [nbval=n];
for(int i=0; i<n; i++)val[i]=abs((char)rand());
ctr+=nbval;}
entier :: ~entier()
{ ctr-=nbval;
 cout << "dest :Nb entiers restants = " << ctr << endl;
delete val;}

void entier::affiche()
{for(int i=0; i<nbval; i++)cout << val[i]<< "\t";
cout << endl;}

void entier :: compte()
{  cout << "Nb. de entiers crees : " << ctr << endl;}

main()
{
  entier :: compte();

    entier p(3);
    p.affiche();
    entier :: compte();

    entier :: compte();
    entier q;
    q.affiche();
    entier r;
    r.affiche();
    entier :: compte();
}
/* Résultat
 [root@localhost 4.2const]# ./p
Nb. De entiers crées : 0
103     58      105
Nb. De entiers crées : 3
dest :Nb entiers restants = 0
Nb. De entiers crées : 0
115     81      1       74
20      41      51      70
Nb. De entiers crées : 8
dest :Nb entiers restants = 4
dest :Nb entiers restants = 0
*/
```

```
// Exemple 4.2.1.2b:utilisation d'une classe avec un constructeur usuel
// dynamique avec surdéfinition de ce dernier + comptage du nbre d'objets - application du
destructeur
// Utilisation du constructeur par recopie par défaut
#include <iostream.h>
#include <stdlib.h>
#define NMAX 4
class entier {
  int nbval;
  int *val;
  static int ctr;  //compteur d'objets
public :
entier();
entier(int);
~entier();
void affiche();
static void compte();
};

int entier :: ctr = 0;  // init. à 0 du nb d'objets
entier::entier()
{
val = new int [nbval=NMAX];
for(int i=0; i<NMAX; i++)val[i]=abs((char)rand());
ctr+=nbval;
}
```

```
// Exemple 4.2.1.2c: utilisation d'une classe avec un constructeur usuel
// dynamique avec surdefinition de ce dernier + comptage du nbre d'objets - application du
destructeur
// Utilisation du constructeur par recopie explicite
#include <iostream.h>
#include <stdlib.h>
#define NMAX 4
class entier {
  int nbval;
  int *val;
  static int ctr;  //compteur d'objets
public :
entier();
entier(int);
entier(entier &);
~entier();
void affiche();
static void compte();
};
int entier :: ctr = 0;  // init. à 0 du nb d'objets
entier::entier()
{
val = new int [nbval=NMAX];
for(int i=0; i<NMAX; i++)val[i]=abs((char)rand());
ctr+=nbval;
}
entier::entier(int n)
{
val = new int [nbval=n];
for(int i=0; i<n; i++)val[i]=abs((char)rand());
ctr+=nbval;
```

```
}
entier::entier(entier &p)
{
val = new int [nbval=p.nbval];
for(int i=0; i<nbval; i++)val[i]=p.val[i];
ctr+=nbval;
}
entier :: ~entier()
{
ctr-=nbval;
 cout << "dest :Nb entiers restants = " << ctr << endl;
delete val;
}
void entier::affiche()
{
for(int i=0; i<nbval; i++)cout << val[i]<< "\t";
cout << endl;
}
void entier :: compte()
{
  cout << "Nb. de entiers crees : " << ctr << endl;
}
main()
{
  entier :: compte();
  entier p(3);
p.affiche();
  entier :: compte();
  entier q(p);
q.affiche();
entier r;
r.affiche();²
  entier :: compte();
}
/* Resultat
[ens-linux]# ./p
Nb. De entiers crees : 0
103     58      105
Nb. De entiers crees : 3
103     58      105
115     81      1       74
Nb. De entiers crees : 10
dest :Nb entiers restants = 6
dest :Nb entiers restants = 3
dest :Nb entiers restants = 0
*/
```

**4.2.1.3. Constructeur par recopie**

Le constructeur par recopie permet de créer un nouveau objet et d'initialiser son contenu avec celui d'un objet existant et passé en paramètre au constructeur.
Il existe un constructeur par recopie par défaut. Néanmoins, il est indispensable d'en prévoir dans le cas de gestion dynamique de la mémoire.

**4.2.2 Catégories d'objets**
**4.2.2.1 Objet automatique/statique**

Les règles de portée des variables automatiques/statiques s'appliquent aux objets de mêmes noms. Pour tout objet, son constructeur est appelé lors la définition de l'objet. Le destructeur est appelé pour un objet automatique à la fin du bloc où il est défini, et à la fin du programme pour un objet statique.

**4.2.2.2 objet temporaire**

Ce sont des objets, qui ne sont pas définis dans le programme, mais que le compilateur crée pour pouvoir compiler certaines syntaxes se référant aux objets. Ce sont donc des objets transparents pour le programmeur, le constructeur est alors appelé explicitement.
**Exemple 4.2.2.2:**
**…**

**4.2.2.3 Objet dynamique**

Ce sont des objets dont les constructeurs et destructeurs font appel aux opérateurs de gestion dynamiques de la mémoire (cf. § 3.5). La gestion de la mémoire dynamiquement permet d'optimiser l'utilisation de celle-ci et d'être maître des instants d'allocation et de libération de la mémoire.

**Exemple 4.2.2.3:**
**…**

**4.2.2.4 Tableau d'objets**

Les objets d'une classe peuvent être regroupés sous forme de tableau. Dans ce cas le constructeur d'objet sera appelé successivement pour tous les objets du tableau du 1er au dernier. Le destructeur d'un tableau d'objets les élimine du dernier crée au premier.

**Exemple 4.2.2.4 :**
**…**

# 5. FONCTION MEMBRE

**syntaxe** :
type_retour classe_appartenance : : nom_fonction ( param1, param2, …)
{
<instructions>
}

**5.1 arguments par défaut**
La déclaration d'une fonction peut contenir pour un ou plusieurs paramètres (obligatoirement les derniers de la liste) des valeurs par défaut. Ce mécanisme permet d'appeler la fonction avec un nombre variable de paramètres, donc de la surdéfinir.

**Exemple 5.1:**
…
Voir TD 1 Exo2.

**5.2 Fonction membre « inline »**

La fonction est transformée en macro. Il existe 2 façons de faire :

**Exemple 5.2a :**

```
Class point
{
…
public :
inline point (int =0 , int =0) ; // le paramètre est nul s'il est absent
};
inline point : : point (int abs=0, int ord=0){x=abs ; y=ord ;}
```

**Exemple 5.2b :**

```
Class point
{
…
public :
point (int abs=0 , int ord=0){x=abs ; y=ord ;}
} ;
```

**5.3 surdéfinition**

**a – surdéfinition d'une fonction**

La sur définition d'une fonction est la re déclaration de celle-ci avec des paramètres de nombre et de type différents. Cette surcharge permet un paramétrage qui permet d'obtenir une hiérarchie de fonctions de même nom.

**Exemple 5.3a:**

…

Voir les exemples 3.4.a , 3.4.b et 3.4.c

**b- surdéfinition d'un opérateur**

**Objectif :** Faire en sorte que vos classes soient plus faciles à utiliser.

La surcharge d'opérateur permet de donner aux opérateurs C/C++ un comportement spécifique quand ils sont appliqués à des types spécifiques (des classes). Les opérateurs surchargés remplacent avantageusement les *appels de fonction.*

La surcharge d'un opérateur est réalisée à l'aide d'une **fonction** (fonction membre ou non) spécialisée en utilisant le mot clé **operator**

*Type **operator** l'opérateur_à_surcharger (… ) {…}*

Les opérateurs suivants peuvent être surchargés

```
   +        -        *        /        %        ^        &
   |        ~        !        =        <        >        +=
  -=       *=       /=       %=       ^=       &=       |=
  <<       >>      >>=      <<=       ==       !=       <=
  >=       &&       ||       ++       --      ->*       ,
  ->       []       ()      new    delete   type()
```

Soit la classe Complexe suivante :

```
class Complexe
{
        double reel, imaginaire;
 public :
    Complexe(double, double);
    Complexe &operator =(const Complexe &);
    Complexe operator +(const Complexe &);
    friend ostream & operator <<(ostream &, const Complexe &);   //AMIE
 };
 Complexe :: Complexe (double r=0.0, double im=0.0)
 {   reel= r; imaginaire =im; }
```

**Exemple 5.3b1 : surcharge de l'opérateur d'affectation '='**

```
Complexe &Complexe::operator=(const Complexe &source)
{
    reel = source.reel;
    imaginaire = source.imaginaire;
    return *this; // adresse de l'objet
}
```

**Exemple 5.3b2 : surcharge d'opérateurs de sortie << du flot cout.**

```
ostream &operator<<(ostream &sortie, Type_classe objet)
        {
         // envoi sur le flot de sortie des membres de l'objet
         // sortie << ...
         ...
         return sortie;
       }

ostream& operator <<(ostream& f, const Complexe &C)
  {
  return f << "("<< C.reel<<","<< C.imaginaire<<")";
 }
```

**Exemple 5.3b3 : surcharge d'opérateurs d'entrée >> du flot cin :**

```
istream &operator>>(istream &entree, Type_classe objet)
        {
         // lecture des membres de l'objet
         // entree >> ...
         ...
         return entree;
        }
```

**Exemple 5.3b4 : surcharge d'un opérateur arithmétique :**

**…**

```
Complexe Complexe :: operator +(const Complexe & comp1)
 {  Complexe c;
    c.reel = reel +comp1.reel;
   c.imaginaire = imaginaire +comp1.imaginaire;
   return c;
 }

int main(void)
 { Complexe C1(1.5, 2.5), C2(3.5, 4.5), C3, C4;
  C3=C1 + C2; C4=C3 ;
  cout << C3; cout << C4;
  return 0 ;
  }
```

**5.4. Passage et retour de paramètres**

**5.4.1 Objet transmis en argument d'une fonction membre**

Une fonction membre peut avoir comme argument un objet de la même classe à laquelle elle appartient.
Par défaut une fonction membre reçoit l'adresse de l'objet l'ayant appelé dont elle a accès directement aux données privées. L'accès aux données privés d'un objet, de la même classe, passé en argument est obtenu en passant par l'objet.
En revanche, la fonction membre n'a accès qu'aux membres publiques de l'objet passé en paramètre s'il appartient à une classe différente de celle de la fonction membre.

**Exemple 5.4.1 : fonction membre avec un objet de même classe en argument**

```
#include <iostream.h>
class Point
{
      int x;
      int y;

public :
      Point(int a=0, int b=0){ x=a; y=b; }
      int Coincide(Point &);
};
int Point::Coincide(Point & p)
{
      if( (p.x==x) && (p.y==y) )
        return 1;
      else
        return 0;
}

void main()
{
      Point p(2,0);
      Point pp(2);
      if( p.Coincide(pp) )
           cout << "p et pp coincident !" << endl;
      if( pp.Coincide(p) )
           cout << "pp et p coincident !" << endl;
}
```

**5.4.2 Mode de transmission des objets**

Il existe 3 modes de passage de d'objets en paramètre vers et à partir d'une fonction :
- par valeur : une copie de l'objet est passé en paramètre.
- Par adresse : l'adresse de l'objet est passé en paramètre.
- Par référence : la fonction appelée accède directement à l'objet référencé.

**Exemple 5.4.2 :**

```
int Point::Coincide(Point p){ if( (p.x==x) && (p.y==y) ) return 1;
                               else return 0; }
… a.coincide(b); // l'appel
int Point::Coincide(Point & p) { if( (p.x==x) && (p.y==y) )return 1;
                               else return 0; }
… a.coincide(b); // l'appel
int Point::Coincide(Point * p) { if( (p->x==x) && (p->y==y) )return 1;
                               else return 0; }
… a.coincide(&b); // l'appel
```

**5.4.3 Auto référence à un objet**
. Un objet peut s'auto référencer en utilisant ***this***.
. Toute classe en C++ possède un membre donnée privé nommé ***this***.
. ***this*** est utilisable uniquement au sein d'une fonction membre, désigne un pointeur sur l'objet l'ayant appelé.

**Exemple 5.4.3:**

```
int point::coincide (point pt)
{ if ((this -> x == pt.x)) && (this -> y == pt.y)) return 1 ;
else return 0 ;
}
```

**5.4.4 qualificatif *const***

Permet de rendre accessible en lecture seule une variable. *const* est notamment utilisé lors du passage de paramètres par référence .

***const*** ➔ l'objet est constant ➔ ne peut pas être modifié par le constructeur.

**Exemple 5.4.4. :**

```
Class Point { …
        Point(const Point &) ;  …} ;
Point :: Point(const Point &P) { x=P.x ;  y=P.y ;}
```

Voir Exemple 5.3b1, etc..

**5.4.5 fonction membre « static »**
C'est une fonction dont la déclaration est précédée par static et est appelée directement sans passer par un objet de la classe à laquelle elle appartient. Dans ce cas, cette fonction agit sur des données « static ».

**Exemple 5.4.5: (cf. ex. 4.2.1.2a)**

**5.4.6. Pointeur sur une fonction membre**

L'appel d'une fonction membre peut se faire en passant par un pointeur contenant l'adresse de cette fonction. Ce mécanisme est utile pour paramétrer les appels de plusieurs fonctions en passant par le même pointeur.

**Exemple 5.4.6:**

Si la classe **point** comporte deux fonctions membres de prototype :
void deplacer_hor (int)
void deplacer_ver (int)

➔ la déclaration ***void (point :: *adf) (int ) ;*** signifie que ***adf*** est un pointeur sur une fonction membre de la classe **point** recevant un argument de type ***int*** et ne r'envoie aucune valeur.

➔ Les affectations suivantes sont possibles :
***adf = point ::deplacer_hor ; // ou adf = & point ::deplacer_hor ;***
***adf = point ::deplacer_ver ; // ou adf = & point ::deplacer_ver ;***

➔ Si a est un objet de type point alors on peut écrire :

***(a.*adf) (3) ;***

pour transmettre la valeur 3 en argument de la fonction dont l'adresse est contenue dans ***adf.***

# 6. FONCTION AMIE

C'est une fonction indépendante ou appartenant à une classe B et qui peut accéder aux données privées d'une classe A parce que dans cette elle a été déclarée comme amie.

Il existe plusieurs situations d'amitié :

**6.1 Fonction indépendante f amie d'une classe A**

```
class A
{
…
friend …f(A)
…
} ;

… f(A a){…} ;  //définition de f , dans l'objet a de classe A est utilisé
```

**6.2 Fonction indépendante f amie de plusieurs classes (A et B)**

```
class A ;  // utile pour compiler class B

class B
{
…
friend …f(A,B)
…
} ;

class A
{
…
friend …f(A,B) ;
…
} ;
```

**6.3 Fonction membre f d'une classe B, amie d'une classe A**

```
class A ;  // utile pour compiler class B

class B
{
…
…f(A)
…
} ;

class A
{
…
friend …B :: f(A) ;
…
} ;
```

**6.4 Fonctions d'une classe B amies d'une classe A**

```
class A ;
class B
{
…
…f1(A) ;
…f2(A) ;
…
} ;

class A
{
…
friend class B ; //Toutes les fns membre de B sont amies de A
…
} ;
```

# 7. HERITAGE SIMPLE

**7.1 Definition**
L'héritage est l'accès de tout ou partie des membres d'une classe de base par une classe dérivée.

Syntaxe :

```
Class Base
{
…
} ;

class Derivee : type_heritage Base
{
…
} ;
```

**7.2 Qualificatifs des membres d'une classe**

**public** : accessible aux fonctions membre de la classe, aux fonctions amies et à l'objet utilisant les membres de la classe .
**private** : accessible aux fonctions membres de la classe et aux fonctions amies. Inaccessible aux fonctions membres d'une classe dérivée.
**Protected** : accessible aux fonctions membres de la classe, aux fonctions amies et aux fonctions membres d'une classe dérivée.

**7.3 Relations entre le type d'héritage et les qualificatifs des membres d'une classe**

| Classe<br>Type_héritage | Base | Derivee | Accès aux fonctions membres et amies | Accès à un objet utilisant la classe |
|---|---|---|---|---|
| **Public** | Private<br>Protected<br>Public | Private<br>Protected<br>Public | Non<br>Oui<br>Oui | Non<br>Non<br>Oui |
| **Protected** | Private<br>Protected<br>Public | Private<br>Protected<br>Protected | Non<br>Oui<br>Oui | Non<br>Non<br>Non |
| **Private** | Private<br>Protected<br>Public | Private<br>Private<br>Private | Non<br>Oui<br>Oui | Non<br>Non<br>Non |

**7.4 Héritage des fonctions amies**

Une fonction amie d'une classe dérivée accède aux membres protégés (protected) de la classe de base. (un ami peut partager un héritage !)

Une fonction amie d'une classe de base ne peut être héritée par une classe dérivée. (l'amitié ne s'hérite pas !)

**7.5 Héritage des constructeurs**

**a – constructeur usuel**

Syntaxe :
B : :B(int x, int y, int z) : A(x,y) ; // la classe B hérite le constructeur usuel de la classe A

**b – constructeur par recopie**

Syntaxe :
B : :B(B &b) : A(b) ; // la classe B hérite le constructeur par recopie de la classe A

# 8. HERITAGE MULTIPLE

**8.1 Définition**
Une classe peut hériter simultanément de plusieurs autres classes. L'héritage de chacune de ces classes peut être public/protected/private. Les mêmes règles de l'héritage simple s'y appliquent.

Syntaxe :
Class D : private C, protected B, public A ;
{
…
} ;

S'il ya redéfinition d'un membre des classes A, B ou C dans D, l'accès se fait par défaut au membre de D. Si le membre visé est celui des classes A,B ou C, il y accéder en utilisant l'opérateur de portée : :

**8.2 Héritage multiple des constructeurs**

Syntaxe :
D(int a, int b, int c, int d) : C(c), B(b), A(a) ;

L'appel des constructeurs de C, B et A se fait dans l'ordre mentionné . Les destructeurs sont appelés dans l'ordre inverse.

**8.3 Exemple d'heritage simple publique/protege**

```
// Heritage simple publique/protege : la classe disque herite de cercle qui herite de point.
#include <iostream.h>
class point {
protected :
int x,y;
public :
point(int =0, int =0);
void affiche();
};
point::point(int a, int b)
{
x=a; y=b;
}
void point::affiche()
{
  cout << "point : " << x << "\t" << y << endl;
}
class cercle : public point
{
protected :
  int r; //rayon
public :
 cercle(int, int, int);
void affiche();
};
cercle :: cercle(int a, int b, int c) : point(a,b)
{r=c;}
void cercle :: affiche()
{cout << "cercle de centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << endl;}
class disque : public cercle
{
  double s; //surface
public :
 disque(int, int, int);
void affiche();
};
disque :: disque(int a, int b, int c) : cercle(a,b,c)
{
s=3.14*r*r;
}
void disque :: affiche()
{
cout << "disque de centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << "\t et de surface : " << s
<<endl;
}
main()
{  point p;   // centre de coord. (0,0)
p.affiche();
  cercle c(0,0,1);  // rayon
c.affiche();
  disque d(0,0,1);
d.affiche();
}
/* Resultat d'execution
point : 0       0
cercle ce centre : 0    0       et de rayon : 1
disque de centre : 0    0       et de rayon : 1  et de surface : 3.14
*/
```

**8.4 Exemple d'héritage simple privée**

**// Heritage simple privée : la classe disque herite de cercle qui herite de point.La classe disque ne peut pas heriter les donnees de la classe point car celles-ci deviennent privees.**

```
#include <iostream.h>
class point {
protected :
int x,y;
public :
point(int =0, int =0);
void affiche();
};
point::point(int a, int b)
{
x=a; y=b;
}
void point::affiche()
{
  cout << "point : " << x << "\t" << y << endl;
}
class cercle : private point
{
protected :
  int r; //rayon
public :
 cercle(int, int, int);
void affiche();
};
cercle :: cercle(int a, int b, int c) : point(a,b)
{
r=c;
}
void cercle :: affiche()
{
  cout << "cercle ce centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << endl;
}
class disque : private cercle
{
  double s; //surface
public :
 disque(int, int, int);
void affiche();
};
disque :: disque(int a, int b, int c) : cercle(a,b,c)
{
s=3.14*r*r;
}
void disque :: affiche()
{
cout << "disque de centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << "\t et de surface : " << s
<<endl;
}
main()
{
  point p;   // centre de coord. (0,0)
p.affiche();
  cercle c(0,0,1);  // rayon
c.affiche();
  disque d(0,0,1);
```

```
d.affiche();
}
/* Resultat de la compilation
$ g++ p.cpp -o p
p.cpp: In method `void disque::affiche()':
p.cpp:56: member `x' is from private base class
p.cpp:56: member `y' is from private base class */
```

**8.5 Exemple 1 d'héritage multiple de +sieurs familles**

**// Heritage multiple protégé : la classe disque herite de cercle qui herite de point.La classe disque hérite également de « materiau ». Le constructeur par défaut de cette classe est appelé implicitement.**

```
#include <iostream.h>
class point {
protected :
int x,y;
public :
point(int =0, int =0);
void affiche();
};
point::point(int a, int b)
{
x=a; y=b;
}
void point::affiche()
{
  cout << "point : " << x << "\t" << y << endl;
}
class cercle : protected point
{
protected :
  int r; //rayon
public :
 cercle(int, int, int);
void affiche();
};
cercle :: cercle(int a, int b, int c) : point (a,b)
{
r=c;
}
void cercle :: affiche()
{
  cout << "cercle ce centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << endl;
}
class materiau
{
protected :
  int m; //materiau
public :
 materiau(int=12);
void affiche();
};
materiau :: materiau(int a)
{
m=a;
}
void materiau :: affiche()
{
  cout << "materiau : " << m << endl;
}
```

```
class disque : protected cercle, protected materiau
{
  double s; //surface
public :
 disque(int, int, int);
void affiche();
};
disque::disque(int a,int b,int c):cercle(a,b,c)
{
s=3.14*r*r;
}
void disque :: affiche()
{
  cout << "disque de centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << "\t et de surface : " << s
<< "\et de materiau : " << m <<endl;
}
main()
{
  materiau m(45);
m.affiche();
  point p;   // centre de coord. (0,0)
p.affiche();
  cercle c(0,0,2);  // rayon
c.affiche();
  disque d(0,0,2);
d.affiche();
}
/* Resultat de la compilation
[root@localhost 4.7hermul]# ./p
materiau : 45
point : 0       0
cercle ce centre : 0    0       et de rayon : 2
disque de centre : 0    0       et de rayon : 2  et de surface : 12.56 de materiau : 12
*/
```

---

**8.6 Exemple 2 d'héritage multiple de plusieurs familles**

// **Heritage multiple public/protegee : la classe disque herite de cercle qui herite de point.La classe disque herite egalement de la classe 'materiau', le constructeur de cette classe est appele explicitement**

```
#include <iostream.h>
class point {
protected :
int x,y;
public :
point(int =0, int =0);
void affiche();
};
point::point(int a, int b)
{
x=a; y=b;
}
void point::affiche()
{
  cout << "point : " << x << "\t" << y << endl;
}
class cercle : protected point
{
protected :
  int r; //rayon
public :
 cercle(int, int, int);
```

```
void affiche();
};
cercle :: cercle(int a, int b, int c) : point (a,b)
{
r=c;
}
void cercle :: affiche()
{
  cout << "cercle ce centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << endl;
}
class materiau
{
protected :
  int m; //materiau
public :
 materiau(int=12);
void affiche();
};
materiau :: materiau(int a)
{
m=a;
}
void materiau :: affiche()
{
  cout << "materiau : " << m << endl;
}
class disque : public cercle, public materiau
{
  double s; //surface
public :
 disque(int, int, int,int);
void affiche();
};
disque::disque(int a,int b,int c,int d):cercle(a,b,c),materiau(d)
{
s=3.14*r*r;
}
void disque :: affiche()
{
  cout << "disque de centre : " << x << "\t" << y << "\t" << "et de rayon : " << r << "\t et de surface : " << s
<< "\et de materiau : " << m <<endl;
}
main()
{
  materiau m(45);
m.affiche();
  point p;   // centre de coord. (0,0)
p.affiche();
  cercle c(0,0,2);  // rayon
c.affiche();

  disque d(0,0,2,33);
d.affiche();
}
/* Resultat d'execution
[root@localhost 4.7hermul]# ./p
materiau : 45
point : 0       0
cercle ce centre : 0    0       et de rayon : 2
disque de centre : 0    0       et de rayon : 2  et de surface : 12.56 de materiau : 33    */
```

# 9. CONTENU DES TDs

TD 1 : SPECIFICITES NON OBJET DU C++
TD 2 : CLASSE ET OBJET
TD 3 : CONSTRUCTEUR/DESTRUCTEUR
TD 4 : HERITAGE

**Université d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TD1 C++**

**Spécificités non objet du C++ et Introduction à la notion de classe**

**Exercice 1 :** Ecrire un programme qui fait appel à deux fonctions permettant de calculer et renvoyer le carré d'un nombre saisi dans le programme principal. La première fonction utilisera un appel par valeur, la deuxième, un appel par référence . Quelle est la différence.

**Exercice** 2 : Ecrire un programme qui fait appel à une fonction qui permet de calculer le volume d'une boite. Ses trois arguments, à savoir : longueur, largeur et hauteur ont une valeur par défaut égale à 1.
Vous calculerez le volume des trois boites suivantes :
Long=5, larg=1, haut=1 Long=5, larg=3, haut =1 Long=1, larg=5, haut=2

**Exercice** 3 : Ecrire un programme qui fait appel à une fonction qui retourne le quotient de deux nombres qui peuvent être entiers ou réels. Le résultat est réel.

**Exercice** 4 : Ecrire un programme qui permet d'allouer dynamiquement de la mémoire suffisante pour saisir n éléments de type entier et de saisir ces éléments; n étant donné par l'utilisateur.

**Exercice** 5 : Réaliser une structure Point permettant de manipuler un point d'un plan. Ecrire :
- Une fonction "saisie()" recevant en entrée, les coordonnées d'un point
- une fonction *translat()* effectuant une translation définie par ses deux arguments float.
- Deux fonctions membres nommées *abscisse()* et *ordonnee()* retournant respectivement l'abscisse et l'ordonnée d'un point.

**Université d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TD2 C++**

**Classes, Constructeur/Destructeur**

**Exercice 1 :** Convertir la structure Point de l'exercice 5 du TD 1 en classe en lui ajoutant les fonctions membres suivantes :

- *homothetie()* effectuant un déplacement d'un point suivant un coefficient multiplicateur fourni en argument ;
- *rotation()* qui effectue une rotation dont l'angle est fourni en argument ;
- *rho()* et *theta()* qui retournent les coordonnées polaires d'un point.

**Exercice 2 :** Ecrire un petit programme avec une fonction main() permettant de manipuler un point :

Saisir les deux coordonnées du point, lui appliquer une translation (*dx, dy*), une homothétie de coefficient *k*, puis une rotation d'angle *alpha*. Afficher après chaque transformation ponctuelle les coordonnées cartésiennes et polaires du point.

**Exercice 3 :** Modifier la classe précédente pour que les données privées ne soient plus l'abscisse et l'ordonnée mais les coordonnées polaires.
Vous aurez besoin de la fonction *angle()* qui calcule l'angle correspondant aux coordonnées polaires

**Université d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TD3 C++**

**Classes, Constructeur/Destructeur, Héritage**

**Exercice 1 :**
Ajouter à la classe Point, de l'exercice 5 TD1, une fonction *affiche()* permettant d'afficher les coordonnées d'un point ainsi que le nombre de points déjà créés.

**Exercice 2 :**
Reprendre la classe *Point* et transformer la fonction *affiche()* en fonction amie. Faites les modifications que cela induit dans le programme.

**Exercice 3 :**

Ecrire une classe *Rectangle* héritant de la classe *Point.*
Vous devrez :

- Ajouter trois caractéristiques pour la longueur, la largeur et l'angle de sa base par rapport à l'axe des x.
- Ecrire le constructeur d'un rectangle
- Ecrire trois fonctions *longueur(), largeur() et angle()* permettant de connaître la longueur, la largeur et l'inclinaison du rectangle.
- Rajouter la fonction *diagonale()* permettant de calculer la longueur de la diagonale du rectangle.
- Redéfinissez les fonctions *rotation() et homothetie()* pour qu'elles permettent la rotation effective et l'homothétie d'un rectangle.
- Ecrire un programme utilisant des rectangles.

**Universite d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TD4 C++**

**Classes, Héritage**

**Exercice 1 :**
Réaliser une classe *Matériau*. Ecrire un constructeur permettant d'initialiser le type de matériau ainsi que deux fonctions membres :

- une fonction *choix()* qui permet de choisir et renvoyer le type de matériau,
- une fonction *affiche()* qui permet d'afficher le type de matériau choisi.

**Exercice 2 :**
Réaliser une classe *Parallélépipède* héritant de la classe *Rectangle*. Ne pas oublier d'ajouter aux données la hauteur du parallélépipède.

**Exercice** 3
Ecrire une classe *Boite* héritant de la classe *Parallélépipède* et de la classe *Matériau*. Ecrire un programme qui permet de créer une boite.

# 10. CONTENU DES TPs

Programmation en C++ d'une application en plusieurs parties :

**TP 1** : Spécificités non objet du C++
**TP 2** : Notion de classe en C++
**TP 3** : Notion de constructeur et de destructeur d'objet en C++
**TP 4** : Notion de fonction amie en C++ et introduction à l'héritage
**TP 5** : Héritage

**Universite d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TP1 C++**

**Spécificités non objet du C++**

**1.Objectif :**
Le but de ce TP est de vous familiariser avec les bases du C++ par l'écriture de programmes pour implémenter des notions de structure et de passage de paramètres à une fonction.

**2.Consignes de programmation :**
Il s'agit de partir d'un exemple de programme écrit en C++ et de le modifier ou de creer de nouveaux programmes.
Les fichiers sources exemples ainsi que des exécutables se trouvent dans le répertoire :
[serveur]$**/home/ufrst/mallem/cours/c++/tp1**
La recopie dans votre répertoire de "tp1" sous serveur peut etre obtenue par:
[serveur]$**cp -r /home/ufrst/mallem/cours/c++/tp1 .**
La compilation, du programme "ensemble.cpp" est obtenue par la commande :
[serveur]$**g++ ensemble.cpp -o ensemble**
L'execution de "ensemble" est realisee par la commande :
[serveur]$**./ensemble**
N'oubliez pas, une fois le TP terminé de sauvegarder votre travail sur votre compte(et éventuellement sur disquette) car le résultat de ce TP est utilisé dans les TPs suivants.
(Les commandes de recopie du TP, d'edition et de compilation sous Linux sont indiquées en annexe)

**3. Sujet :**
Q1: commenter le programme "ensemble.cpp" donné comme exemple de base.
Q2: Ecrire une fonction "intersection(ensemble X, ensemble Y, ensemble &Z)" qui place dans Z l'intersection entre X et Y.
Q3: Ecrire une fonction "union(ensemble X, ensemble Y, ensemble &Z)" qui place dans Z l'union de X et Y.
Q4: Ecrire une fonction "difference(ensemble X, ensemble Y, ensemble &Z)" qui place dans Z la différence entre X et Y.
Q5: Revoir la fonction "inclusion(ensemble X, ensemble Y)" de manière à ce que celle-ci fonctionne pour des ensembles contenant des éléments redondants.

**4. Remarques :**
1)Vous prendrez soin de definir les prototypes des fonctions en utilisant des references afin de pouvoir modifier les composantes des structures definissant les ensembles.
2)Vous n'avez pas a realiser une interface avec menu graphique. Vous vous limitez a saisir deux ensembles X, Y et un element x au clavier (comme c'est le cas du programme donne en exemple), puis a calculer tous les cas de figures enumeres ci-haut (affiches simplement comme dans l'exemple).
3) Vous pourrez utiliser, dans le programme deux structures d'ensembles pour les ensembles saisis, puis une troisieme pour stocker le resultat des operations
difference et intersection. Pour l'union, vous pouvez definir une structure contenant un tableau de taille deux fois plus grandes (l'union des deux ensembles aura au plus une cardinalite deux fois plus grandes que l'un des deux ensembles), ainsi qu'une composante cardinal de maniere a ce que la fonction d'affichage puisse traiter cet ensemble aussi .
4)l'exécution du programme ./ensemble2 donne une idee du résultat attendu
P.S. : Si vous avez fini le TP avant la fin de la séance, vous pourrez vous attaquer au complément du TP1 se trouvant dans le fichier sujet1_bis.

**Annexe : Commandes de recopie de fichiers, d'edition, de compilation ... sous Linux**

**1. Connexion sur le serveur Linux**
login : nom
password : mot de passe
[serveur ] $ mkdir c++ (si ce n'est pas fait)
[serveur ] $ cd c++

**2. Recopie du TP sous "serveur"**
[serveur ] $ telnet serveur
[serveur] $ cp -r /home/ufrst/mallem/cours/c++/tpx . (x=1..5)

**3. Edition de fichier sous Linux local**
[serveur ] $ emacs &

**4. Compilation sous Linux local**
[serveur ] $ g++ fichier.cpp -o fichier

**5. Envoi du fichier par e-mail :**
[serveur ] $ mail user@******* <ensemble.cpp

**6. Impression :**
[serveur ] $ lpr ensemble.cpp

**7. Utilisation du lecteur de disquette**
Lancer Windows, procéder par "ftp", sous windows, pour recupérer ou sauver un fichier

**Universite d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TP2 C++** **Notion de classe en C++**

**1. Objectif :**
Le but de ce TP est de vous familiariser avec la notion de classe et d'encapsulation de données et de fonctions membres en C++.

**2. Consignes de programmation :**
Il s'agit de modifier l'exemple "ensemble2.cpp" du tp1 de façon a ce que la structure ensemble devienne une classe.
Les fichiers sources exemples ainsi que des exécutables se trouvent dans le répertoire :
**serveur $ /home/ufrst/mallem/cours/c++/tp2**
La recopie dans votre répertoire de "tp2" sous serveur peut etre obtenue par:
**serveur $ cp -r /home/ufrst/mallem/cours/c++/tp1 .**
La compilation, du programme "ensemble.cpp" est obtenue par la commande :
**$ g++ ensemble.cpp -o ensemble**
L'execution de "ensemble" est realisee par la commande :
**$ ./ensemble** (Les commandes de recopie du TP, d'edition et de compilation …sous Linux sont indiquées en annexe)

**3. Sujet :**
La structure "ensemble" :

```
struct ensemble {
  int cardinal;
  int elements[NMAX];
};
```

devient :

```
class ensemble {
  int cardinal;
  int elements[NMAX];
public :
int appartient(int);
int inclusion(ensemble);
ensemble reunion(ensemble);
ensemble intersection(ensemble);
ensemble  difference(ensemble);
void saisie();
void affichage();
int menu(void);
int Cardinal(); // retourne le cardinal de ensemble
 };
```

Q1: Modifier le programme ensemble.cpp, resultat du TP1, en un programme ensemble2.cpp de façon a ce que "ensemble" devienne une classe comme indiqué ci-dessus.
Q2 : Remplacer la fonction "void saisie(ensemble&)" par un constructeur usuel "ensemble :: ensemble()",qui fait exactement le meme travail que saisie(), et apporter les egalement les modifications necessaires dans le programme principal.
Que remarquez vous lors de l'execution du programme dans ce cas ?
l'exécution du programme ./ensemble2 donne une idee du résultat attendu

---

**Annexe : Commandes de recopie de fichiers, d'edition, de compilation ... sous Linux**

**1. Connexion sur le serveur Linux**
login : nom
password :  mot de passe
[serveur ] $ mkdir c++ (si ce n'est pas fait)
[serveur ] $ cd c++

**2. Recopie du TP sous "serveur"**
[serveur ] $ telnet serveur
[serveur] $ cp -r /home/ufrst/mallem/cours/c++/tpx .  (x=1..5)

**3. Edition de fichier sous Linux local**
[serveur ] $ emacs &

**4. Compilation sous Linux local**
[serveur ] $ g++ fichier.cpp -o fichier

**5. Envoi du fichier par e-mail :**
[serveur ] $ mail user@******* <ensemble.cpp

**6. Impression :**
[serveur ] $ lpr ensemble.cpp

**7. Utilisation du lecteur de disquette**
Lancer Windows, procéder par "ftp", sous windows, pour recupérer ou sauver un fichier

**Universite d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TP3 C++** **Notion de constructeur et de destructeur d'objet en C++**

**1. Objectif :**
Le but de ce TP est de vous familiariser avec la notion de constructeur et de destructeur d'objet en C++.

**2. Consignes de programmation :**

Il s'agit de modifier l'exemple "ensemble2.cpp" resultat du tp2 de façon a ce que la classe ensemble contienne des constructeurs et un destructeur d'objets.

Les fichiers sources exemples ainsi que des exécutables se trouvent dans le répertoire :
serveur $ /home/ufrst/mallem/cours/c++/tp3
(Les commandes de recopie du TP, d'edition et de compilation sous Linux sont indiquées en annexe)

**3. Sujet :**
La classe ensemble devient :

```
//fichier ensemble.h - declaration de la classe  ensemble
 #include <iostream.h>
#include <stdlib.h>
#define XMAX 10   //Taille max de ensemble X
#define YMAX 10   //Taille max de ensemble Y
#define ZMAX 20   //Taille max de ensemble Z
class ensemble {
 int cardinal;
  int *elements;    //pointeur sur les elements de ensemble
public :
ensemble(int); //tp3 - constructeur dynamique usuel
ensemble(const ensemble&); // TP3 - Constructeur dynamique de recopie
~ensemble();   //tp3- destructeur dynamique
int appartient(int);
int inclusion(ensemble);
ensemble reunion(ensemble);
ensemble intersection(ensemble);
ensemble  difference(ensemble);
//interface.h - declaration des prototypes des fonctions d'entree sortie
void affichage();
int menu(void);
int Cardinal(); // retourne le cardinal de ensemble };
```

**Q1** : Ecrire le programme correspondant au constructeur usuel dynamique ensemble(int). Utiliser la fonction "abs((char)rand()/10);" pour initialiser chaque element de l'ensemble cree.

**Q2** : Ecrire le programme correspondant au constructeur de recopie dynamique ensemble(const ensemble&).
Ecrire le programme correspondant au destructeur ~ensemble(). Apporter les modifications necessaires dans le programme principal. L'exécution du programme ./ensemble3_1 donne une idee du résultat attendu lors de l'utilisation d'un constructeur pour une allocation memoire statique.
l'exécution du programme ./ensemble3_2 donne une idee du résultat attendu lors de l'utilisation d'un constructeur dynamique____________

## Annexe : Commandes de recopie de fichiers, d'edition, de compilation ... sous Linux

**1. Connexion sur le serveur Linux**
login : nom
password : mot de passe
[serveur ] $ mkdir c++ (si ce n'est pas fait)
[serveur ] $ cd c++

**2. Recopie du TP sous "serveur"**
[serveur ] $ telnet serveur
[serveur] $ cp -r /home/ufrst/mallem/cours/c++/tpx . (x=1..5)

**3. Edition de fichier sous Linux local**
[serveur ] $ emacs &

**4. Compilation sous Linux local**
[serveur ] $ g++ fichier.cpp -o fichier

**5. Envoi du fichier par e-mail :**
[serveur ] $ mail user@******* <ensemble.cpp

**6. Impression :**
[serveur ] $ lpr ensemble.cpp

**7. Utilisation du lecteur de disquette**
Lancer Windows, procéder par "ftp", sous windows, pour recupérer ou sauver un fichier

**Universite d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TP4 C++** 

`Notion de fonction amie en C++`

**1. Objectif :**
Le but de ce TP est de vous familiariser avec la notion de fonction amie permettant de manipuler des objets d'une classe dont elle n'est pas membre.

**2. Consignes de programmation :**
Il s'agit de modifier votre programme resultat du TP3 de façon a ce que la classe ensemble contienne deux fonctions amies de la classe ensemble.
Les fichiers sources exemples ainsi que des exécutables se trouvent dans le répertoire :
serveur $ /home/ufrst/mallem/cours/c++/tp4

(Les commandes de recopie du TP, d'edition et de compilation sous Linux sont indiquées en annexe)

**3. Sujet :**

Q1 : Ecrire une fonction amie de la classe ensemble appelee "void parite(ensemble)" qui permet de donner la parite (0:pair / 1: impair) de chaque element de l'ensemble passe en parametre a la fonction parite. Cette fonction doit afficher la parite de tous les elements d'un ensemble resultat d'une operation.

Q2 : Ecrire une fonction amie de la classe ensemble appelee "void premier(ensemble)" qui permet d'afficher pour chaque element de l'ensemble resultat d'une operation, passe en parametre a la fonction premier, s'il s'agit d'un nombre premier (0: non premier/ 1: premier).

l'exécution du programme ./ensemble4 donne une idee du résultat attendu

___________

**Annexe : Commandes de recopie de fichiers, d'edition, de compilation ... sous Linux**

**1. Connexion sur le serveur Linux**
login : nom
password : mot de passe
[serveur ] $ mkdir c++ (si ce n'est pas fait)
[serveur ] $ cd c++
**2. Recopie du TP sous "serveur"**
[serveur ] $ telnet serveur
[serveur] $ cp -r /home/ufrst/mallem/cours/c++/tpx . (x=1..5)
**3. Edition de fichier sous Linux local**
[serveur ] $ emacs &
**4. Compilation sous Linux local**
[serveur ] $ g++ fichier.cpp -o fichier
**5. Envoi du fichier par e-mail :**
[serveur ] $ mail user@******* <ensemble.cpp
**6. Impression :**
[serveur ] $ lpr ensemble.cpp
**7. Utilisation du lecteur de disquette**
Lancer Windows, procéder par "ftp", sous windows, pour recupérer ou sauver un fichier

**Universite d'Evry**
**UFR Sces et Technologie**
**IUP-Licence**
**TP5 C++**

`Notion d'heritage simple en C++`

**1. Objectif :**
Le but de ce TP est de vous familiariser avec la notion d'heritage simple en C++.

**2. Consignes de programmation :**
Il s'agit de modifier votre programme resultat du TP4 de façon a ce que la classe ensemble contienne deux fonctions amies de la classe ensemble.
Les fichiers sources exemples ainsi que des exécutables se trouvent dans le répertoire :
serveur $ /home/ufrst/mallem/cours/c++/tp5
(Les commandes de recopie du TP, d'edition et de compilation sous Linux sont indiquées en annexe)

**3. Sujet :**
Il s'agit de rajouter la classe :

```
// classe ensembleplus herite de la classe ensemble
class ensembleplus : public ensemble
{
public :
int *parite;
int *premier;
  ensembleplus(int);  //constructeur
  ensembleplus(const ensemble&); //constructeur par recopie
  void affichageplus();  // affichage de l'ensemble et de ses proprites parite et premier
friend void Parite(ensemble&); //fonction amie independante
friend void Premier(ensemble&); //fonction amie independante
};
```

Q1 : Ecrire les fonctions membres de cette nouvelle classe.
Q2 : Modifier la fonction "void parite(ensemble)" qui devient amie de la classe ensembleplus et ayant pour protoype "void Parite(ensembleplus&)";.
Q3 : Modifier la fonction "void premier(ensemble)" qui devient amie de la classe ensembleplus et ayant pour protoype "void Premier(ensembleplus&)";.

L'exécution du programme ./ensemble5 donne une idee du résultat attendu

---

**Annexe : Commandes de recopie de fichiers, d'edition, de compilation ... sous Linux**
**1. Connexion sur le serveur Linux**
login : nom
password : mot de passe
[serveur ] $ mkdir c++ (si ce n'est pas fait)
[serveur ] $ cd c++
**2. Recopie du TP sous "serveur"**
[serveur ] $ telnet serveur
[serveur] $ cp -r /home/ufrst/mallem/cours/c++/tpx . (x=1..5)
**3. Edition de fichier sous Linux local**
[serveur ] $ emacs &
**4. Compilation sous Linux local**
[serveur ] $ g++ fichier.cpp -o fichier
**5. Envoi du fichier par e-mail :**
[serveur ] $ mail user@******* <ensemble.cpp
**6. Impression :**
[serveur ] $ lpr ensemble.cpp
**7. Utilisation du lecteur de disquette**
Lancer Windows, procéder par "ftp", sous windows, pour recupérer ou sauver un fichier

# 12. CARACTERISTIQUES DE JAVA

**Tout est classe (pas de fonctions) sauf les types primitifs (int, float, double, ...) et les tableaux**

**Toutes les classes dérivent de java.lang.Object**

**Héritage simple pour les classes**

**Héritage multiple pour les interfaces**

**Les objets se manipulent via des références**

**Une API objet standard est fournie**

**La syntaxe est proche de celle de C**

**Le compilateur Java génère du *byte code*.**

**La *Java Virtual Machine* (JVM) est présente sur Unix, Win32, Mac, OS/2, Netscape, IE, ...**

**Le langage a une sémantique très précise.**

**La taille des types primitifs est indépendante de la plate-forme.**

**Java supporte un code source écrit en Unicode.**

**Java est accompagné d'une librairie standard.**

# 13. LES DIFFERENCES JAVA - C++

| | |
|---|---|
| **Pas de préprocesseur (#define, #include)** | **Pas d'héritage multiple(sauf pour les interfaces)** |
| **Pas de struct, union, enum et typedef** | **Pas de *template(patron de classe)*** |
| **Pas de variables globales, ni de fonction** | **Pas de surcharge d'opérateurs** |
| **Pas de fonction à nombre variable d'arguments** | **Pas de passage d'argument par recopie** |
| **Pas de pointeurs (en particulier de fonction) mais des références** | **Pas d'allocation mémoire statique pour les objets (et tableaux)** |

**Sites Web :**

**http://www.javasoft.com :** ***Site officiel Java (JDK et doc.)***

**http://www.javaworld.com :** ***Info sur Java***

**http//www.gamelan.com :** ***applications, applets, packages, ...***

**http://www.jars.com :** ***idem***

**http://www.blackdown.com :** ***Java pour linux***