



Initiation au langage

Objet C++

TABLE DES MATIERES

1) Le Monde objet

1.1) Les objets sous Windows page N°4
1.2) Constitution d'un objet page N°5

2) Les Classes

2.1) Constitution d'une classe page N°6
2.2) Données privées ou publiques page N°7
2.3) Représentation d'une classe en UML page N°8
2.4) Définition d'une méthode page N°8
2.5) Création d'un objet page N°8
Fiche TD N°1 : les classes, membres et méthodes page N°9

3) Les constructeurs

3.1) Définition d'un constructeur page N°10
3.2) Appel d'un constructeur page N°11
Fiche :TD N°2 (Les classes : constructeurs et appel) page N°12
3.3) Construction d'un objet membre page N°15

4) Destruction d'un objet page N°16

Fiche :TD N°3 (Les classes :destructeur) page N°17

5) Les fonctions et classes amies page N°19

5.1) Les classes amies page N°19
5.2) Les fonctions amies page N°20
Fiche :TD N°4 (Les classes amies) page N°21

6) L'héritage page N°24

6.1) L'héritage simple page N°25
6.2) L'héritage multiple page N°29

7) Les exceptions page N°31

7.1) Introduction page N°31
7.2) Mise en situation – « Un problème bien ennuyeux » page N°31
7.3) La gestion des exceptions page N°34

8) Rappels - Flux standards les plus communs

8.1) Les entrées - sorties page N°37
8.2) Les fonctions d'attente page N°39

9) Les allocations dynamiques

9.1) Introduction page N°40
9.2) Allocation dynamique d'une variable page N°40
9.3) Allocation dynamique d'un tableau page N°42

10) Les fonctions virtuelles et classes abstraites

10.1) Les fonctions virtuelles page N°44
10.2) Les classes abstraites page N°45

11) Annexes

11.1) Les classes abstraites page N°49
11.2) Le pointeur This page N°50
11.3) Diagramme de classes page N°52
11.4) Les threads ou processus léger page N°53

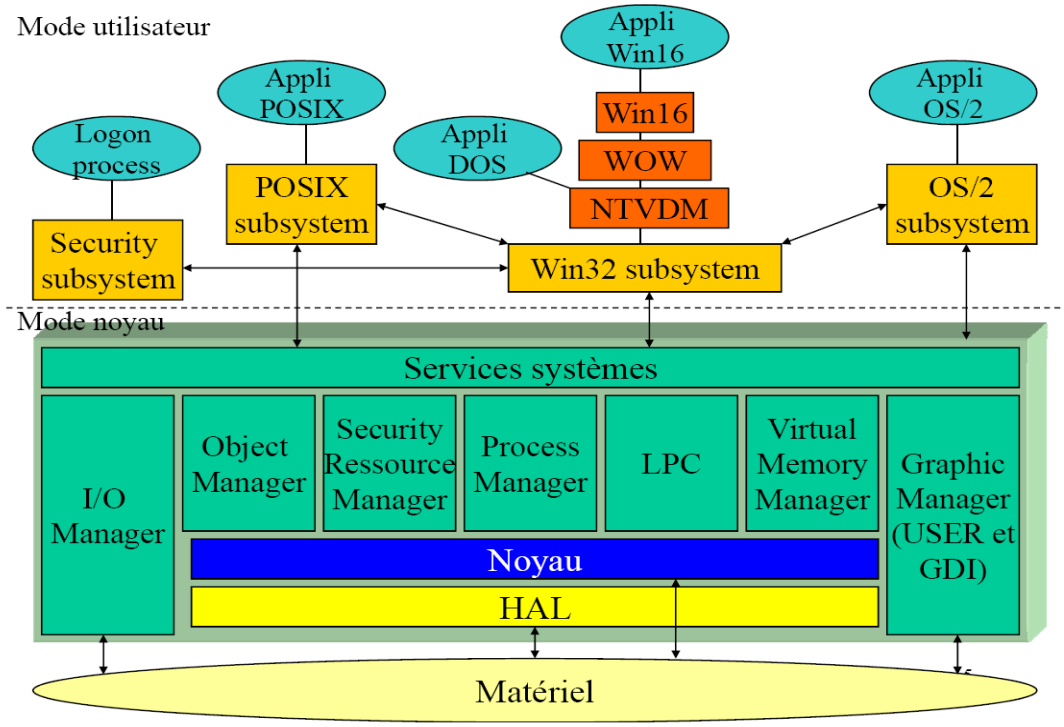
Initiation au langage objet C++

1) Le monde objet

1.1) Les objets sous Windows

Sous windows , les ressources de l'OS sont représentés par des objets :
Les objets de noyau (processus, tache, fichier …).
Les objets graphiques (GDI Graphic interface development).
Les objets utilisateurs.

Un gestionnaire d'objet permet de gérer ces objets.



Il assure de nombreuses fonctionnalités :

- vérification des droits d'accès des tâches et processus sur les objets et réalisation de leur protection.
- création et suppression des objets; surveillance des objets temporaires pour les supprimer dès qu'ils ne sont plus utilisés; suivi des objets en utilisant leur en-tête.
- centralisation des opérations de contrôle des ressources.
- attribution des noms aux objets (chaque nom est global sur une machine, mais pas dans le réseau) et implantation des objets nommés dans une mémoire globale gérée dynamiquement.

1.2) Constitution d'un objet

Un objet comporte deux parties :

- un en-tête contrôlé par le gestionnaire d 'objets. Les services génériques (fermer un handle, dupliquer un handle, interroger l'objet, interroger la sécurité, changer la sécurité, attendre un objet) accèdent aux données de l'en-tête.

- un corps contrôlé par le sous-ensemble du SE qui a créé ce type d'objet. Le corps comprend plusieurs champs : type d'objet, attributs de l'objet (données), services de l'objet (méthodes).

Définition : on appelle handle d'objet un index dans une table de pointeurs sur les objets spécifique à un processus. Cette table comprend des pointeurs vers tous les objets pour lesquels le processus a ouvert un handle. Un handle est donc un pointeur indirect vers un objet. Seul le gestionnaire d'objets a le droit de créer un handle.

Les avantages de la programmation objet :

- Facilité de conception et de réutilisation et facilité de compréhension.
- Amélioration de fiabilité.
- Abstraction et encapsulation accrue.
- Masquage et amélioration des informations.

Afin de créer et de manipuler des objets nous nous familiariserons avec le langage objet C++. La suite logicielle Borland Builder C++ nous initiera au langage C++. Cette suite nous permettra à terme de créer des applicatifs utilisateur sous l'environnement graphique de windows.

2) Les classes

2.1) Constitution d'une classe

Un objet est constitué par l'association d'une certaine quantité de mémoire, organisée en champs (**membres**), avec des **données**, et d'un ensemble de fonctions (**méthodes**), destinées principalement à la consultation et la modification des valeurs de ces champs.

Une classe décrit la structure interne d'un objet.

La définition d'un type objet s'appelle une **classe**. D'un point de vue syntaxique, cela ressemble beaucoup à une définition de structure, sauf que

{ le mot réservée ***class*** remplace le mot ***struct***,

{ certains champs de la classe sont des fonctions.

Une classe comme une structure, contient des *membres* (champs). **Les membres d'une classe contiennent des informations (« données ») et des fonctions de traitement de ces données, les « méthodes ».**

Le nom de chaque classe est unique.

```
class Point {
public:
    void afficher() {
        cout << '(' << x << ',' << y << ')';
    }
    void placer(int a, int b) {
        validation des valeurs de a et b;
        x = a; y = b;
    }
    double distance(Point autrePoint) {
        int dx = x - autrePoint.x;
        int dy = y - autrePoint.y;
        return sqrt(dx * dx + dy * dy);
    }
private:
    int x, y;
};
```

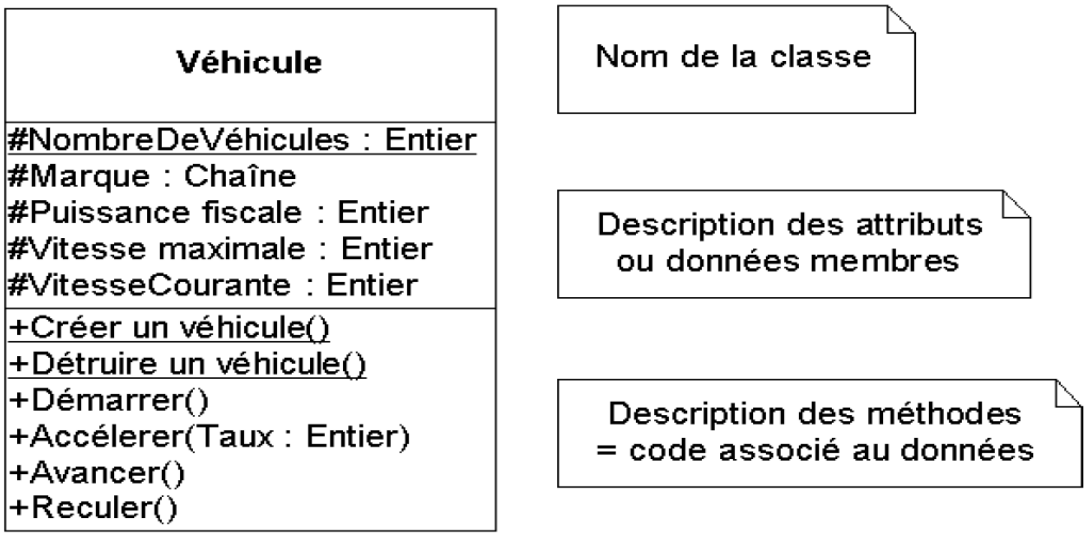
2.2) Données privées et publiques

Le contrôle de l'accès aux membres d'une classe permet de réduire les risques d'erreurs d'utilisation de ces membres par le programme. Ce contrôle consiste à privatiser ces membres. En fait les membres d'une classe sont *privées* par défaut et l'étiquette *public* permet de marquer ceux dont l'accès direct doit être autorisé au reste du programme.

Les membres d'une classe peuvent être ***public*** ou ***private***. Un programme peut **accéder directement aux membres *publics*** à l'aide de l'opérateur point. Un programme **ne peut accéder à un membre *private* que par l'intermédiaire des méthodes de sa classe**.
A défaut d'autres indication, les membres d'une classe sont considérés comme privés.

2.3) Représentation d'une classe en UML (Unified Modelisation Langage)

Les classes se représentent de la façon suivante :



public	+	Elément non encapsulé visible par tous
protégé	#	Element encapsulé visible dans les sous classes de classe
privé	-	Element encapsulé visible seulement dans la classe

2.4) Définition d'une méthode

La définition d'une fonction de classe ou méthode peut être incluse ou non dans la définition de la classe. Si elle est définie en dehors de la définition de sa classe, le nom de la classe dont elle relève doit précéder le sien, séparé par *l'opérateur global de résolution* sous la forme ***classe::fonction***

```
Double  point ::distance(point autrepoint)
{
}
```

```
Point q ;
Point P ;
```

Lors d'un appel tel que ***p.distance(q)***, l'objet ***p*** accède aux membres privées (via les méthodes) ***x*** et ***y*** de l'objet ***q***. On dit que *C++ pratique l'encapsulation au niveau de la classe*, non au niveau de l'objet.

2.5) Création d'un objet (l'instanciation d'une classe)

Une fois une classe définie, il est possible de déclarer des objets de cette classe en utilisant le nom de la classe comme type.
Les programmes accèdent aux membres (publics) d'une classe (données ou fonctions) à l'aide de l'opérateur point ou la flèche si c'est un pointeur.

```
Point a, b, *pt;          // deux points et un pointeur de point
```

Dans un contexte où le droit de faire un tel accès est acquis (cf. section 2.2.3) l'accès aux membres du point `a` s'écrit :

```
a.x = 0;                  // un accès bien écrit au membre x du point a
d = a.distance(b);        // un appel bien écrit de la fonction distance de l'objet a
```

Si on suppose que le pointeur `pt` a été initialisé, par exemple par une expression telle que

```
pt = new Point;           // allocation dynamique d'un point
```

alors des accès analogues aux précédents s'écrivent :

```
pt->x = 0;                // un accès bien écrit au membre x du point pointé par pt
d = pt->distance(b);      // un appel de la fonction distance de l'objet pointé par pt
```

Fiche TD N°1

(Les classes : membres et méthodes)

Le programme :

```
class identite
{
    public:
            char nom[64];
            int annee;
            void affecter_nom(char* n)
            {
                    strcpy(nom,n);
            }
            void affiche( )
            {
                    cout <<"son nom est"<< nom;
            }
} ;

int main( )
{
    identite client1 ;
    client1.annee = 1976 ;
    client1.affecter_nom(" durand");
    client1.affiche( );
    cout <<"\n L annee est : "<< client1.annee;

    system("PAUSE") ;
    return 0;
}
```

Questions :

1) Tapez le programme suivant sous DEVC++ et testez le.

2) On souhaite déclarer la méthode affecter_nom à l'extérieure de la classe identité, proposez une modification du programme et testez ce nouveau programme.

3) On souhaite rendre private le membre annee de la classe identite, modifiez la déclaration de la classe et testez le programme. Que remarquez vous et pourquoi ?. Proposez une solution.

3) Les constructeurs

3.1) définition d'un constructeur

Un constructeur d'une classe est une fonction membre spéciale qui :

- **A le même nom que la classe,**
- **n'indique pas de type de retour,**
- **ne contient pas d'instruction return.**

Le rôle d'un constructeur est d'initialiser un objet, notamment en donnant des valeurs à ses données membres.
Le constructeur n'a pas à s'occuper de trouver l'espace pour l'objet ; il est appelé (immédiatement) après que cet espace ait été obtenu, et cela quelle que soit la sorte d'allocation qui a été faite : statique, automatique ou dynamique, cela ne regarde pas le constructeur

```
class Point {
public:
    Point(int a, int b) {
        validation des valeurs de a et b
        x = a; y = b;
    }
    ... autres fonctions membres ...
private:
    int x, y;
};
```

C'est le couple ***définition de la variable + appel du constructeur*** **qui constitue la réalisation en C++ du concept création d'un objet.**

L'intérêt pour le programmeur est évident : garantir que, dès leur introduction dans un programme, tous les objets sont garnis et cohérents, c'est-à-dire éviter les variables indéfinies, au contenu incertain.

Une classe peut posséder plusieurs constructeurs, qui doivent alors avoir des signatures différentes (au niveau des paramètres) :

```
class Point {
public:
    Point(int a, int b) {
        validation de a et b
        x = a; y = b;
    }
    Point(int a) {
        validation de a
        x = a; y = 0;
    }
    Point() {
        x = y = 0;
    }
    ...
private:
    int x, y;
};
```

3.2) Appel des constructeurs

Un constructeur est *toujours* appelé lorsqu'un objet est créée, soit explicitement, soit implicitement. Les appels explicites peuvent être écrits sous deux formes :

```
Point a(3, 4);
Point b = Point(5, 6);
Point c= Point() // constructeur par défaut
   Point *pt;
   ...
   pt = new Point(1, 2);
```

Le constructeur par défaut est un constructeur qui peut être appelé sans paramètres : ou bien il n'en a pas, ou bien tous ses paramètres ont des valeurs par défaut. Il joue un rôle remarquable, car *il est appelé chaque fois qu'un objet est créé sans qu'il y ait appel explicite d'un constructeur*, soit que le programmeur ne le juge pas utile, soit qu'il n'en a pas la possibilité.

Fiche :TD N°2 (Les classes : constructeurs et appels)

Premier programme :

```
class identite
{
    public:
            char nom[64];
            int annee;
            void affecter_nom(char* n)
            {
                    strcpy(nom,n);
            }
            void affiche( )
            {
                    cout <<"son nom est"<< nom;
            }
} ;

int main( )
{
    identite client1 ;
    client1.annee = 1976 ;
    client1.affecter_nom(" durand");
    client1.affiche( );
    cout <<"\n L annee est : "<< client1.annee ;

    system("PAUSE") ;
    return 0;
}
```

Questions :

1) Créez les 3 constructeurs suivants sous REDHAT9 (voir guide d'utilisation LINUX):

identite(char* n)	/* qui permet d'affecter le membre ***nom*** de l'objet créé lors de l'appel du constructeur avec la chaîne de caractères pointée par ***n*** */
identite()	/* qui permet d'affecter par défaut un *nom* nommé ***" dupont"*** de l'objet créé lors de l'appel du constructeur */
identite(identite *i)	/* qui permet de copier le *nom* de l'objet pointé par ***i*** avec le *nom* de l'objet crée lors de l'appel du constructeur */

2) Créez 3 objets différents de classe identité permettant de faire appel à ces 3 constructeurs. L'objet n°1 sera passé en paramètre à l'objet n°3.

3) Affichez avec la console le nom de ces 3 objets.

Deuxième programme :

```
#include <vcl.h>
#include <iostream.h>
#include <conio.h>
#include <string.h>

class point
{
    public:
        point (float a, float b);
        void afficher_point( );
    private:
        float x, y;
} ;

/* ******************Programme Principal ************** */
point::point(float a, float b)
{
        x=a; y=b;
}

void point::afficher_point( )
{
        cout <<"cordonnees x="<< x <<"\n"<<"cordonnees y="<< y <<"\n";
}
……
```

Questions :

1) Sous CENTOS 6.2, dans la fonction main(),initialisez un premier objet point nommé cible1 de coordonnées x, y est de valeur respective x=2 et y=3.

2) Proposez une méthode de la classe point permettant de calculer la distance entre 2 points. Cette méthode renverra la distance sous forme d'un double.

double point::distance(point p)

3) Complétez la fonction main suivante et testez votre programme

```
int main()
{
```

```
        double res;                                  // distance résultante entre les 2 points
        double a,b;                                  // abscisse et ordonnée du deuxième point
        cout <<"Entrer l abscisse du deuxieme point: ";          //saisie de a et b
        cin >>a;
        cout <<"Entrer l ordonnee du deuxieme point: ";
         cin >>b;
        _____________________________________;       // déclaration des 2 objets
        _____________________________________;       //affichage du point cible 1
        res=_____________________________;           // calcul de la distance entre la
                                                     //cible 1 et la cible 2
        cout<<"\n\t-La distance resultante est:"_____________;   //affichage de la distance
        system("PAUSE") ;
        return 0;
};
```

3.3) Construction des objets membres

Lorsque des membres d'une classe sont à leur tour d'un type *classe*, on dit que la classe a des *objets membres*.

L'initialisation d'un objet de la classe nécessite alors l'initialisation de ces objets membres.

```
NomDeLaClasse(paramètres)
        : membre(paramètres) , ... membre(paramètres) {
    corps du constructeur
}
```

```
class Segment {
    Point origine, extremite;
    int epaisseur;
public:
    Segment(int ox, int oy, int ex, int ey, int ep)
            : origine(ox, oy), extremite(ex, ey) {
        epaisseur = ep;
    }
    ...
};
```

Initialisation d'un objet

```
Segment objet(2, 3,3,4,6) ;     //ox=2, oy=3, ex=3, ey=4, ep=6 lancement du
                                //constructeur point de l'objet origine puis constructeur point
                                //de l'objet extremite et pour finir le constructeur de la classe.
```

4) Destruction d'un objet (les destructeurs)

Le *constructeur* et le *destructeur* sont deux méthodes particulières qui sont appelées respectivement à la création et à la destruction d'un objet. Toute classe a un constructeur et un destructeur par défaut, fournis par le compilateur.

Dès qu'un constructeur ou un destructeur a été défini par l'utilisateur, le compilateur ne définit plus automatiquement le constructeur ou le destructeur par défaut correspondant.

Un destructeur doit obéir à 3 règles :

- Elle doit porter le même nom que la classe ;
- Elle ne doit avoir aucun type, *pas même le type void*.
- Son nom sera toujours précédé du signe tilde ('~').

- Pour les ***objets statiques***, le constructeur et le destructeur sont appelés automatiquement.

- Pour les ***objets alloués dynamiquement***, le constructeur et le destructeur sont appelés automatiquement par les expressions qui utilisent les opérateurs ***new***, ***new[ ], delete et delete[ ]***. C'est pour cela qu'il est recommandé de les utiliser à la place des fonctions ***malloc*** et ***free*** du C pour créer dynamiquement des objets.

```
class chaine                        // Implémente une chaîne de caractères.
{
        char * s;                   // Le pointeur sur la chaîne de caractères.

        public:
            chaine(void);           // Le constructeur par défaut.
            chaine(unsigned int);   // Le constructeur défini par l'utilisateur.
            ~chaine(void);          // Le destructeur.
} ;

chaine :: chaine(void)
{
        s=NULL;                     // La chaîne est initialisée avec le pointeur null.
}

chaine :: chaine(unsigned int Taille)
{
        s = new char[Taille+1];     // Alloue de la mémoire pour la chaîne.
        s[0]='\0';                  // Initialise la chaîne à "".
}
```

```
chaine :: ~chaine(void)
{
        if (s!=NULL) delete[ ] s;            // Restitue la mémoire utilisée si nécessaire
}

/* *****************Programme Principal ************** */

int main()
{
        chaine* obj_ch = new chaine(3);
        delete obj_ch ;                                          //obj_ch->~chaine();

        system("PAUSE");
        return 0;
}
```

Fiche :TD N°3 (Les classes :destructeur)

Programme :

```
#define DIM 16

class identite
{
   private:
           char nom[DIM];
   public:
           identite(char* n)
           {
                   strcpy(nom,n);
           }
           identite( )
           {
                   strcpy(nom,"dupont");
           }
           identite(identite *i)
           {
                   strcpy(nom,i->nom);
           }
           void affiche( );
} ;

/* *****************Programme Principal ************** */

void identite :: affiche( )
{
   cout <<"\nSon nom est "<< nom <<"\n";
```

```
}


int main()
{
    identite *client4 = new identite("jean");        //Déclaration et init Client4
    identite client1("durand"), client3;             //Déclaration et init Client1 et Client 3
    identite client2(&client1);                      //Déclaration et init Client2
    client1.affiche( );
    client2.affiche( );
    client3.affiche( );
    client4->affiche( );

    system("PAUSE") ;
    return 0;
}
```

Questions :

1) Sous Fedora 15 en utilisant le logiciel Qtcreator, proposez un destructeur pour la *classe* ***identite***. Le destructeur n'affiche simplement que le message *"destruction de l'objet".*

2) Testez le nouveau programme et commentez les résultats obtenus.

3) Détruire dans la fonction *main( )* le **client4**.

4) Testez à nouveau le programme et conclure ?

Conclusion :

Un objet alloué dynamiquement comme ***client 4*** est "contrôlé par le programmeur". En particulier, celui-ci a la responsabilité de le détruire.
Cette opération n'est pas automatiquement faite par le compilateur.
La destruction explicite de l'objet par appel de l'opérateur ***delete*** appelle automatiquement le destructeur de l'objet. L'exemple ci-dessus montre une utilisation typique de cet opérateur.

5) Les fonctions et classes amies

5.1) Les classes amies

Une classe **amie** d'une classe **Hote** est une classe qui a le droit d'accéder à tous les membres de Hote. Une telle classe doit être déclarée dans la classe Hote (la classe qui accorde le droit d'accès), précédée du mot réservée ***friend***, indifféremment parmi les membres privées ou parmi les membres publics de Hote.

```
class Hote
{
     friend class Amie;                    // Toutes les méthodes de Amie sont amies.
     int nb;                               // Donnée privée de la classe Hote.
     public:
          Hote(void)
          {
                nb=5;
                return;
          }
};

class Amie
{
    Hote h;                                // Déclaration d'un objet de type "classe Hote"
    public:
         void print_hote(void)
         {
             cout <<"Donnee prive nb de Hote: "<< h.nb<<endl;        // Accède à la donnée privée de h: nb.
             return;
         }
         void ecrit_nb(int i)
         {
             h.nb=i;                       // Ecrit dans la donnée privée de h: nb.
             cout <<"Nouvelle valeur de nb: "<< h.nb<<endl;
             return;
         }
};

/* ******************Programme Principal ************** */
int main()
{
   Amie a;
   a.print_hote( );
   a.ecrit_nb(12);

   cout <<"\n\n";
   system("PAUSE");
   return 0;
}
```

```
Donnee prive nb de Hote: 5
Nouvelle valeur de nb: 12


Appuyez sur une touche pour continuer...
```

5.2) Les fonctions amies

Pour limiter à certaines méthodes l'accès à ses données privées, une classe peut ne déclarer comme ***friend*** que certaines méthodes d'une autre classe. Sa définition introduira chaque fonction à l'aide du mot clé ***friend*** suivi de son prototype.

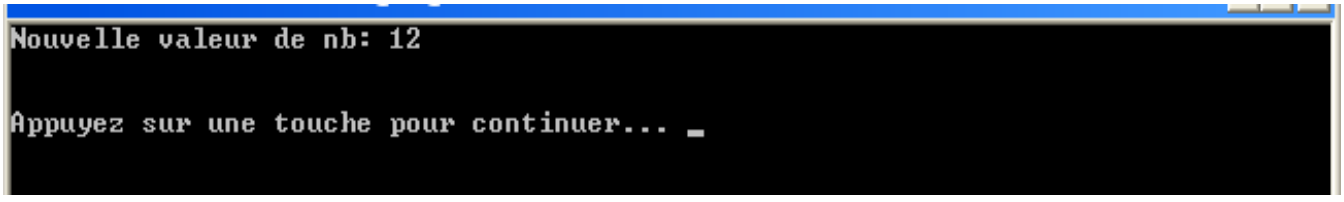
```
class Hote
{
        int nb;                                 // Donnée privée de la classe Hote.
        friend void ecrit_nb(int i);            // Une fonction amie.
};


/* ******************Programme Principal ************** */
Hote h;                                         // Déclaration d'un objet de type "classe Hote"

void ecrit_nb(int i)
{
   h.nb=i;                                  // Ecrit dans la donnée privée de h: nb.
   cout <<"Nouvelle valeur de nb: \n"<< h.nb);
   return;
}


int main()
{
   ecrit_nb(12);                            // Appel à la fonction "ecrit_nb", AMIE à la classe Hote

   cout <<"\n\n";
   system("PAUSE");
   return 0;
}
```



Pour une mise en forme dans un seul fichier source .cpp :

La déclaration de fonction ***friend*** peut amener des erreurs de syntaxe si la classe de ces fonctions est définie après la classe qui les déclare comme ***friend***.

Il suffit d'introduire en début de programme un identificateur prévenant le compilateur de la définition ultérieure de cette classe. Cet identificateur sera de la forme ***class nom_de_classe***.

Fiche :TD N°4 (Les classes amies)

Programme :

```
class polaire
{
   double mod, angle;

   public:
      polaire();
      polaire(double a, double b);
      void afficher_polaire( );
} ;

class cible
{
   double x, y;

   public:
      cible(double a, double b);
      void afficher_point();
      double distance(cible p);
      //polaire cartesien_pol();          // A FAIRE - Question 1.
} ;

/* ******************Programme Principal ********************** */

// Méthodes de la classe "Polaire"
polaire :: polaire()
{
   mod=1;
   angle=3.14;
}

polaire :: polaire(double a, double b)
{
   mod=a;
   angle=b;
}

void polaire :: afficher_polaire( )
{
     cout <<"\n\t Module="<< mod <<"\n\t Angle="<< ((360*angle)/(2*3.14)) <<"\n";
}

// Méthodes de la classe "Cible"
cible :: cible(double a, double b)
{
   x=a;
   y=b;
}
```

```
void cible :: afficher_point( )
{
     cout <<"Coordonnees:\n\t x="<< x <<"\n\t y="<< y <<"\n";
}

double cible :: distance(cible p)
{
   double res;
   res = sqrt((pow((x-p.x),2)) + (pow((y-p.y),2)));
   return res;
}


int main( )
{
   system("PAUSE");
   return 0;
}
```

Questions :

1) Sour Borland C++, proposez une méthode ***cartesien_pol( )*** de la classe *cible* qui permet à partir des coordonnées cartésiennes de calculer les cordonnées polaires, cette méthode renverra un objet du type *polaire*.

```
polaire cible :: cartesien_pol( )
{
        polaire q;
        _____________________________________________ ;
        _____________________________________________ ;
        return q;
}
```

2) Que doit on rajouter dans la classe *polaire* pour que les méthodes de la classe *cible* accèdent aux données privées de la classe *polaire*.

3) On donne la déclaration des variables suivantes dans la fonction ***main*** :

```
int main( )
{
        double res;
        double a,b;
        cible cible1(2,3) ;
        cout <<"Entrer vos coordonnees: \n ";
```

```
cout <<"x:";
cin >>a;
cout <<"y:";
cin >>b;
cible cible2(a,b);
. . .
```

3.1) Affichez les coordonnées de la ***cible1***.

3 .2) Déclarez un objet ***P2*** du type *polaire* en initialisant le module à la valeur 2 et l'angle à 3.14

3.3) Convertir les coordonnées cartésiennes de la ***cible2*** en coordonnées polaires. ***P2*** comportera les nouvelles coordonnées polaires de la ***cible2***.

3.4) Affichez les coordonnées de l'objet ***P2***.

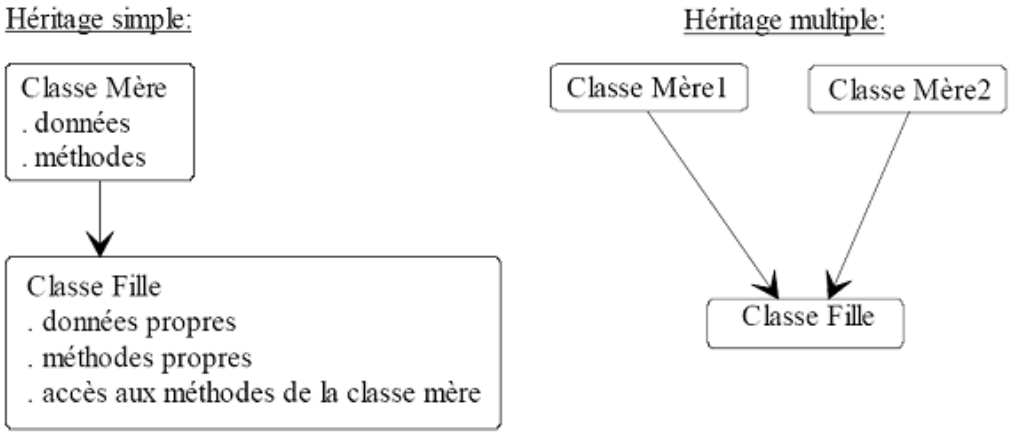
3.5) Calculez la distance qui existe entre la ***cible2*** et la ***cible1*** , cette distance sera mémorisée dans la variable ***res***.

3.6) Affichez cette distance.

6) L'héritage

La P.O.O. permet de définir de nouvelles classes (classes **filles**) dérivées de classes de base (classes **mères**), avec de nouvelles potentialités. Ceci permettra à l'utilisateur, à partir d'une bibliothèque de classes donnée, de développer ses propres classes munies de fonctionnalités propres à l'application.

On dit qu'une classe fille DERIVE d'une ou de plusieurs classes mères.



La classe fille n'a pas accès aux membres privés de la classe mère.

Une classe dérivée a accès aux membres publics de sa classe de base comme à ses propres membres. Elle accède toutefois aux membres privés de la classe que par les fonctions d'interface de cette dernière.

Le mécanisme de l'héritage consiste en la définition d'une classe par réunion des membres d'une ou plusieurs classes préexistantes, dites *classes de base directes*, et d'un ensemble de membres spécifiques de la classe nouvelle, appelée alors *classe dérivée*.

- La syntaxe est :

```
class classe : dérivation classe, dérivation classe, ... dérivation classe
{
    déclarations et définitions des membres spécifiques de la nouvelle classe
};
```

ou ***dérivation*** est un des mots-clés : **private**, **protected** ou **public**.

Quand une variable membre est en ***protected*** dans une classe, elle est pour cette classe considérée comme ***private***, c'est à dire que seules les fonctions membres ou méthodes auront accès a cette variable . Mais les classes dérivées y auront accès comme si elle était en ***public***.

6.1) L'héritage simple

La définition d'une classe dérivée commence par le mot clé ***class***, suivi du nom de la classe, d'un deux points **:** et du nom de la classe de base avec son type de statut (protected, private, public).

```
class Point                          // Classe mère
{
   protected :
          int x,y;
   public :
        void initialiser(int abs, int ord);
        void deplacer(int dx, int dy);
        void afficher() const;
};


class PointCol : public Point        // Classe fille
{
   protected:
          int couleur;
   public:
        void colorer(int c);
};

/* ******************Programme Principal ************** */

// Méthodes de la classe Mère "Point"
void Point::initialiser(int abs, int ord)
{
   x=abs;
   y=ord;
}

void Point::deplacer(int dx, int dy)
{
   x+=dx;
   y+=dy;
}

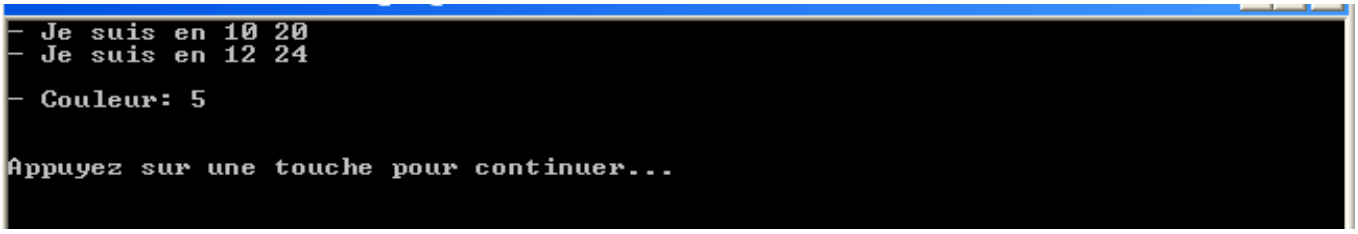
void Point :: afficher() const
```

```
{
   cout <<"- Je suis en "<< x <<" "<< y <<endl;
}

// Méthodes de la classe Fille "PointCol"
void PointCol::colorer(int c)
{
   couleur=c;
   cout <<"\n- Couleur: "<< couleur <<endl;
}

int main()
{
   PointCol p;
   p.initialiser(10,20);
   p.afficher();
   p.deplacer(2,4);
   p.afficher();
   p.colorer(5);

   cout <<"\n\n";
   system("PAUSE");
   return 0;
}
```



La fonction ***constructeur*** d'une classe dérivée doit faire appel à la fonction ***constructeur*** de la classe de base. Son en-tête est donc suivi de deux points puis du nom de la fonction ***constructeur*** de la classe de base avec les paramètres correspondants :

```
PointCol::PointCol(int a, int b, int c)        // Supposons que B dérive de A

    :Point(a, b)          // Ici on passe les paramètres au constructeur de A

  {      ...              // instructions constructeur de B

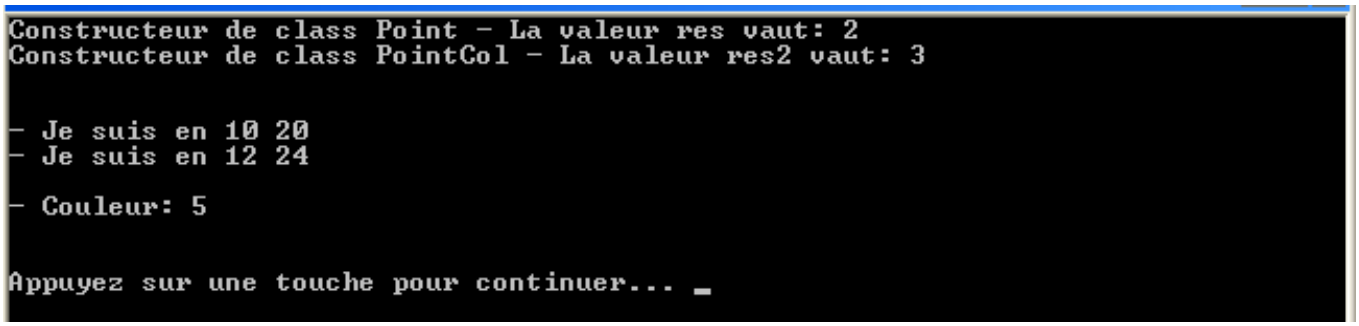
  }
```

Exemple avec le programme précédent, on insère des constructeurs en plus :

```
//Constructeur de la classe Point
Point::Point(int a, int b)
{
    int res;
    res=a+b;
    cout <<"Constructeur de class Point"<<" - La valeur res vaut: "<< res <<endl;
}

//Constructeur de la classe PointCol
PointCol::PointCol(int a, int b, int c)
:Point(a, b)
{
        int res2;
        res2=a+b+c;
        cout <<"Constructeur de class PointCol"<<" - La valeur res2 vaut: "<< res2 <<"\n\n"<<endl;
}
```

Fenêtre obtenue après compilation et exécution :



Une classe de base déclare ***protected*** les membres privés qu'elle souhaite laisser accéder directement à ses classes dérivés. Ces membres protégés, ***public*** pour les classes dérivées restent privés pour le reste du programme.
En choisissant le mot-clé qui indique la dérivation, parmi ***private***, ***protected*** ou ***public***, le programmeur détermine l'accessibilité dans la classe dérivée des membres de la classe de base.

Statut dans la classe mère	Nouveau statut dans la classe fille	Accès par les méthodes de la classe fille	Accès par des entités externes Méthode main ou winmain
Public	Public	**Oui**	**Oui**
Protected	Protected	**Oui**	**Non**
Private	Private	**Non**	**Non**

Héritage public

Statut dans la classe mère	Nouveau statut dans la classe fille	Accès par les méthodes de la classe fille	Accès par des entités externes Méthode main ou winmain
Public	Protected	**Oui**	**Non**
Protected	Protected	**Oui**	**Non**
Private	Private	**Non**	**Non**

Héritage protégé

Statut dans la classe mère	Nouveau statut dans la classe fille	Accès par les méthodes de la classe fille	Accès par des entités externes Méthode main ou winmain
Public	Private	**Oui**	**Non**
Protected	Private	**Oui**	**Non**
Private	Private	**Non**	**Non**

Héritage privé

6.2) L'héritage multiple

L'héritage multiple offre la possibilité à une classe dérivée d'hériter des caractéristiques d'une ou plusieurs classes de base.

Pour faire dériver une classe de deux ou plusieurs classes, on fait suivre le nom de la nouvelle classe d'un : et des noms des classes de base séparés par des *virgules* sur le modèle.

La fonction constructeur de la classe dérivée doit appeler les fonctions constructeurs de chaque classe de base et leur transmettre les paramètres convenables (Voir exemple ci-dessous).

```
class un                                    // Classe mère 1
{
   public:
      un( );
} ;


class deux                                  // Classe mère 2
{
   public:
      deux( );
} ;

class trois                                 // Classe mère 3
```

```
{
   public:
        trois( );
} ;

class derivee : public un, public deux, public trois        // Classe dérivée
{
   public:
        derivee( );
} ;

/* *****************Programme Principal ************** */
un::un( )                                                   // Constructeur de la classe "un"
{
     cout << "Constructeur de la classe -un- \n";
}

deux::deux( )                                               // Constructeur de la classe "deux"

{
     cout << "Constructeur de la classe -deux- \n";
}

trois::trois( )                                             // Constructeur de la classe "trois"
{
     cout << "Constructeur de la classe -trois- \n";
}

derivee::derivee( )                                         // Constructeur de la classe "derivee"
: un( ), deux( ), trois( )                                  // Appels des constructeurs des 3 classes mères
{
     cout << "Constructeur de la classe -derivee- \n";
}

int main( )
{
   derivee ma_classe;                                       // Déclaration d'un objet de type classe "dérivee"

    cout <<"\n\n" ;
    system("PAUSE") ;
   return 0;
}
```

Affichage du résultat du programme :

```
Constructeur de la classe -un-
Constructeur de la classe -deux-
Constructeur de la classe -trois-
Constructeur de la classe -derivee-


Appuyez sur une touche pour continuer...
```

7) Les exceptions

7.1) Introduction

La gestion des exceptions est un moyen de gérer les erreurs qui pourraient survenir dans votre programme; on peut alors traiter ces erreurs et reprendre l'exécution du programme là où elle s'était arrêtée.

On va voir comment créer des exceptions et comment les traiter.

7.2) Mise en situation : « Un problème bien ennuyeux »

En programmation, quelque soit le langage utilisé, il existe plusieurs types d'erreur qui peuvent survenir. Ces erreurs sont généralement classées dans les 4 catégories suivantes :

1. Les erreurs de **syntaxe**. Elles surviennent si le programme est mal écrit, par exemple si il manque un **;** à la fin d'une ligne. Ces erreurs sont détectées par le compilateur et donc facilement corrigeables.
2. Les erreurs d'**implémentation**. Elles surviennent lors de l'exécution du programme. Elles se caractérisent souvent par un plantage du programme. Ces erreurs surviennent par exemple lors d'une division par zéro ou si l'on essaye d'accéder à de la mémoire que l'on a pas allouée précédemment.
3. Les erreurs d'**algorithme**. Ce type d'erreur est plus subtil. En effet le programme fonctionne, mais il ne fait pas ce que le programmeur avait prévu.
4. Les erreurs de **conception**. Ce sont les erreurs qui surviennent si vous avez mal défini ce que vous vouliez faire. Par exemple si vous aviez pensé qu'une grandeur était constante alors qu'elle ne l'est pas. Ce type d'erreur pousse souvent le programmeur à revoir une grande partie de son programme.

La gestion des exceptions permet, si elle est réalisée correctement, de *corriger les erreurs d'implémentation* en les prévoyant à l'avance. Ceci n'est pas toujours réalisable, car il faudrait penser à toutes les erreurs qui pourraient survenir, mais on peut facilement en éviter une grande partie.

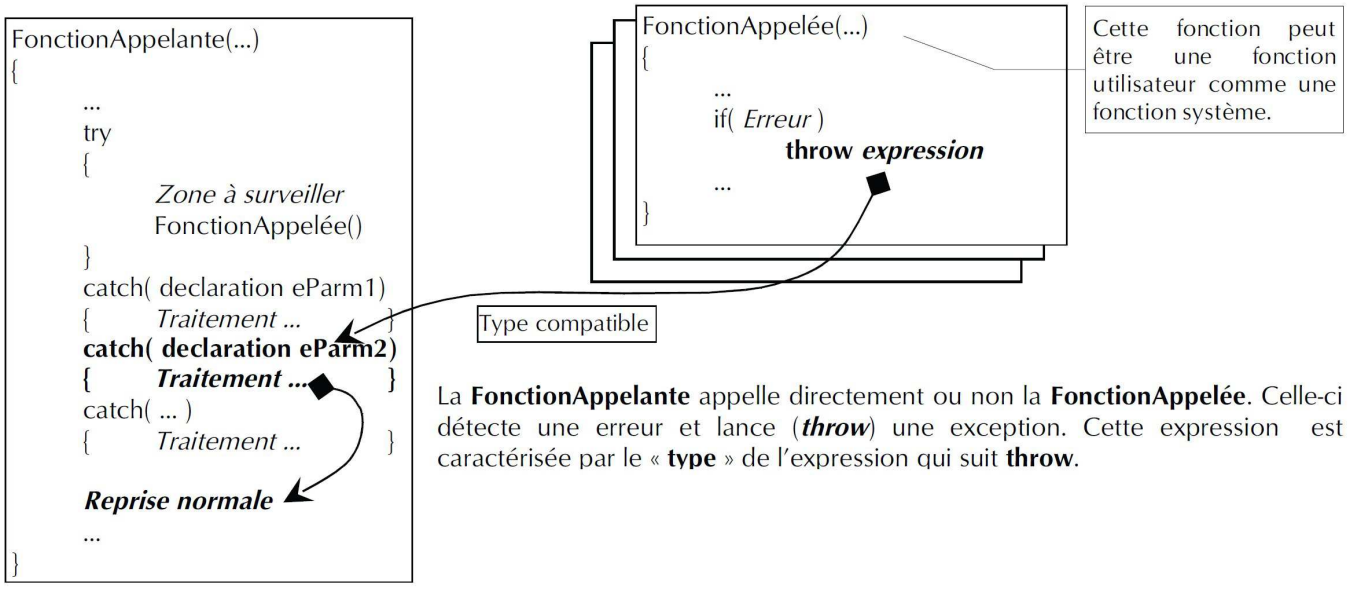
7.3) La gestion des exceptions

- Principe général :

Le langage C++ (comme de nombreux autres langages modernes) met en place un << **mécanisme des exceptions** >> permettant de manière générale a des fonctions appelées de notifier aux fonctions appelantes l'avènement d'une erreur interne.

Les caractéristiques de ce mécanisme sont les suivantes

- La fonction appelée qui (détecte un événement exceptionnel construit une exception et la lance (throw) vers la fonction appelante ;
- Une exception est une instance (objet) d'une classe spécialement définie clans ce but (souvent une classe dérivée de la classe exception) comportant diverses informations utiles à la caractérisation de l'événement à signaler ; Ce peut être un nombre ou une chaine.
- Une fois lancée, l'exception traverse la fonction qui la lancée, sa fonction appelante, la fonction appelante de la fonction appelante, etc., jusqu'à atteindre une fonction active (c.-a-d. une fonction commencée et non encore terminée) qui a prévu d'attraper (catch) ce type d'exception ;



La **FonctionAppelante** appelle directement ou non la **FonctionAppelée**. Celle-ci détecte une erreur et lance (***throw***) une exception. Cette expression est caractérisée par le « **type** » de l'expression qui suit **throw**.

Lors du lancement d'une exception, la fonction qui l'a lancée et les fonctions que l'exception traverse sont immédiatement terminées : les instructions qui restaient à exécuter dans chacune de ces fonctions sont abandonnées ; malgré son caractère prématuré, cette terminaison prend le temps de détruire les objets locaux de chacune des fonctions ainsi avortées ;

• si une exception arrive à traverser toutes les fonctions actives, car aucune de Ces onctions n'a prévu de l'attraper, alors elle produit la terminaison du programme.

Pour intercepter des exceptions au sein d'un bloc d'instructions, une fonction doit délimiter cette séquence ('instructions au sein d'un bloc fonctionnel try d'une part, et d'autre part doit associer un traitement à chaque exception qu'elle souhaite intercepter à l'aide d'un gestionnaire catch.

En résumé le principe général des exceptions est le suivant :

- On crée des zones où l'ordinateur va **essayer** le code en sachant qu'une erreur peut survenir.
- Si une erreur survient, on la signale en **lançant** un objet qui contient des informations sur l'erreur.
- A l'endroit où l'on souhaite gérer les erreurs survenues, on **attrape** l'objet et on gère l'erreur.

Exemple imagé : C'est un peu comme si on était coincés sur une île déserte. Nous lancerions une bouteille à la mer avec des informations dedans permettant de nous retrouver. Il n'y aurait alors plus qu'à espérer que quelqu'un attrape notre bouteille. Sinon on peut mourir de faim…

C'est la même chose ici, on lance un objet en espérant qu'un autre bout de code le rattrapera, sinon le programme plantera.

En C++, il y a trois mots-clés pour gérer ces exceptions qui ont été mis en gras dans le *principe général* ci-dessus (sauf qu'ils sont en anglais) :

- **try{ ... } :** (essaye en français) permet de signaler une portion de code où une erreur peut survenir.
- **throw :** (lance en français) permet de signaler l'erreur en lançant un objet.

- **catch(...) {...}** : (attrape en français) attrape l'objet et permet d'introduire la portion de code qui va s'occuper de gérer l'erreur.

- Exemples détaillés :

1. Commençons par **try** : il permet d'introduire un bloc sensible aux exceptions. On l'utilise comme ceci :

Exemple – Code en C++ - Mot-clé **try** :

```
…       //Ici Du code sans risque

try
{
        //Ici du code qui pourrait créer une erreur
}
```

Entre les crochets du bloc **try** peut se trouver n'importe quelle instruction C++, notamment un autre bloc **try**.

2. Ensuite **throw**, sa syntaxe est : throw expression; . On peut lancer n'importe quoi comme objet, par exemple un entier qui correspond au numéro de l'erreur ou une string contenant le texte de l'erreur. On l'utilise comme ceci :

Exemple – Code en C++ - Mot-clé **throw** :

```
throw 123;              //On lance l'entier 123, par exemple si l'erreur 123 est survenue
throw string("Erreur fatale. Contactez un administrateur");          //On peut lancer une string
throw maClasse;         //Ou un objet d'une classe que vous avez créés.
```

Le mot-clé **throw** peut se trouver n'importe où dans le code (pas d'erreur à la compilation), mais si il n'est pas dans un bloc **try**, l'erreur ne pourra pas être rattrapée et le programme plantera.

3. Terminons avec **catch** : il permet de créer un bloc de gestion d'une exception survenue. Il faut créer un bloc **catch** par type d'exception. Chaque bloc **try** doit obligatoirement être suivi d'un bloc **catch**. De manière réciproque, tout bloc **catch** doit être précédé d'un bloc **try** ou d'un autre bloc **catch**.

Sa syntaxe est la suivante : catch (const type& e) { }

On attrape les exceptions par référence constante (d'où la présence du &) et pas par valeur, ceci afin d'éviter une copie et de conserver le polymorphisme de l'objet reçu.

Exemple – Code en C++ - Mot-clé **catch** :

```
try
{
    //Le bloc sensible aux erreurs
}

catch(const int& e)      //On rattrape les entiers lancés
{
    //On gère l'erreur
}

catch(const string& e)   //On rattrape les strings lancés
{
    //On gère l'erreur
}
```

A l'exécution, le programme va se dérouler normalement comme si les instructions **try** et les blocs **catch** n'étaient pas là.

Par contre, au moment ou l'ordinateur arrive sur une instruction **throw**, il va sauter tous les instructions suivantes

, appeler le destructeur de tous les objets déclarés à l'intérieur des fonctions appelées dans le bloc **try**. Arrivé au bloc **catch**, il va exécuter ce qui se trouve dans le bloc et reprendre l'exécution du programme après le bloc **catch**.

=> Attention, L'exécution reprend après le bloc **catch**, et pas à l'endroit où se trouve le **throw**.

Utilisation de la classe exception sous borland C++

```
try {
PFA2=StrToInt(Edit4->Text);
if(PFA2>37) // on vérifie la valeur de PFA2 rentrée
throw Exception("saisie invalide");// Si PFA2 >37 on lance l'exception
}
```

```
catch (Exception &ex) {
MessageBoxA(NULL,AnsiString("PFA2 < ou = à 37").c_str(),"erreur",MB_OK);
}
```

Initiation au langage objet C++ - COMPLEMENTS

8) Rappels - Flux standards les plus communs

8.1) Les entrées – sorties

- Première chose, il faut inclure la bibliothèque d'entrées-sorties du C++, qui est « **iostream** ».

- Pour les **entrées**, on connaît le mot-clé « **cin** » en C++, équivalent au « **scanf( )** » du langage C. Ces fonctions sont pour la saisie clavier, à la seule différence que « scanf » attend qu'on lui spécifie le type (%..) de la valeur entrée au clavier, et qu'elle corresponde à la variable déclarée dans laquelle on enregistre cette valeur.

Au contraire, « cin » s'adapte au type de la variable déclarée dans laquelle on enregistre la valeur entrée au clavier.

Exemple « cin » :

```
#include < iostream >

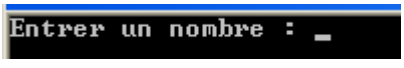
int var;
cout <<"Entrer un nombre : ";
cin >>var;                          //Attend la saisie au clavier d'un nombre pour le mettre dans "var"
```

Exemple « scanf » :

```
#include < stdio.h >

int var;
printf("Entrer un nombre : ");
scanf("%d", &var);      //Attend la saisie au clavier d'un nbre de type INT  pour le mettre dans "var"
```

Affichage à l'écran pour les deux cas :



- Pour les **sorties**, on connaît le mot-clé « **cout** » en C++, équivalent au « **printf( )** » du langage C. Ces fonctions sont pour l'affichage dans la fenêtre de commande (d'exécution), à la seule différence que « printf » attend qu'on lui spécifie le type (%..) de la valeur affichée à l'écran.

Au contraire, « cout » s'adapte au type de la variable déclarée dont on veut afficher la valeur à l'écran.

Exemple « cout » :

```
#include < iostream >

int var;
var=3;
cout <<"La valeur de var est : "<< var;          //On affiche 3
```

Exemple « printf » :

```
#include < stdio.h >

int var;
var=3;
printf("La valeur de var est : %d", var);          //On affiche 3
```

Affichage à l'écran pour les deux cas :



Dans les **sorties**, on peut aussi utiliser le mot-clé « **cerr** », qui a exactement la même action que « cout ». On se sert de « cerr » pour faire la différence, dans la lecture d'un programme, entre l'affichage à l'écran d'un message d'erreur (on utilise « cerr »), et l'affichage de tout autre message.

Exemple « cout - cerr » :

```
#include < iostream >

cout <<"Traitement en cours.."<< endl;
cerr <<"ERREUR"<< endl;
```

Affichage à l'écran :



Pour finir, la terminaison facultative « **endl** » que l'on place en fin de « cout » ou « cerr » permet que le curseur d'écran revienne à la ligne après avoir affiché ce qu'on lui demande à l'écran.

8.2) Les fonctions d'attente

Selon les compilateurs et logiciels de codage, on a parfois besoin de mettre dans notre programme une fonction permettant de faire attendre la fenêtre d'affichage ("l'écran") pour qu'on puisse voir le résultat de notre programme. Car parfois les fenêtres s'ouvrent puis se ferment instantanément.

Pour palier à ce problème, on peut utiliser une fonction d'attente. Les deux fonctions : « **system("PAUSE");** » ou « **getch( );** » bloquent la fenêtre ouverte et attend que l'utilisateur appuie sur une touche.

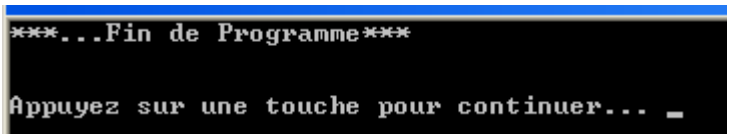
La différence entre les deux c'est que « system("PAUSE"); » écrit sur l'écran : « Appuyer sur une touche… » pour spécifier à l'utilisateur qu'il attend l'appuie d'une touche pour fermer la fenêtre. La fonction « getch( ); » n'écrit rien à l'écran, mais fonctionne similairement : dès qu'une touche est appuyée, il laisse la fenêtre d'affichage se fermer.

Exemple « system("PAUSE"); » :

```
#include < iostream >

cout <<"***...Fin de Programme***\n\n";
system("PAUSE");
```

Affichage à l'écran :

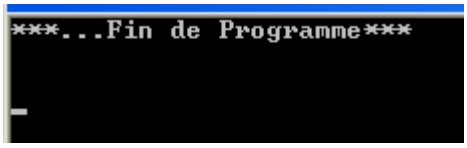


Exemple « getch( ); » :

```
#include < conio.h >
#include < iostream >

cout <<"***...Fin de Programme***\n\n";
getch( );
```

Affichage à l'écran :



Rem : Pour la fonction « **getch( )** », il faut inclure la bibliothèque « **conio.h** ».

9) Allocations dynamiques

9.1) Introduction

Rappels : On connaît en langage C les fonctions « malloc » et « free ».

Exemple – Code en C :

```
int main( )
{
        int *variable = NULL;
        variable = malloc(sizeof(int));          //Allocation de mémoire
        free(variable);                          //Libération de mémoire
        return 0;
}
```

L'allocation dynamique est une technique qui nous permet de gérer l'allocation de mémoire pour des variables. C'est notamment très pratique dans le cas de l'allocation de tableaux dont on ne connaît pas la taille avant compilation.

En C++, on fait toujours des allocations dynamiques. D'ailleurs, les fonctions « malloc » et « free » sont toujours utilisables. Cependant, le C++ dispose de nouveaux opérateurs spécialisés dans les allocations dynamiques : « new » et « delete ».

Les opérateurs **new** et **delete** sont des mots-clé dans le langage C++. Contrairement à **malloc** et **free**, ce ne sont pas des fonctions. Les opérateurs **new** et **delete** font en fait eux-mêmes appels aux fonctions **malloc** et **free** (on se sert de l'existant). Cependant, ils font des tests et des initialisations supplémentaires, ce qui fait qu'on préfèrera toujours utiliser **new** et **delete** au lieu de **malloc** et **free**. Ils sont plus adaptés en C++.

9.2) Allocation dynamique d'une variable

- Allocation de mémoire :

Les opérateurs **new** et **delete** ne s'utilisent pas exactement de la même manière que **malloc** et **free**.
On va dans un premier temps apprendre à s'en servir pour allouer une variable

simple, puis on verra ensuite le cas de l'allocation de tableaux.

On souhaite donc allouer dynamiquement une variable (de type **int** par exemple). En C++, on va d'abord devoir créer le pointeur et l'initialiser à NULL :

Exemple – Code en C++ - *new* :

```
int *variable = NULL;
variable = new int;          //Allocation dynamique
```

On n'a plus besoin d'utiliser l'opérateur de taille **sizeof( )** du C. Ici, on indique juste le type de variable à créer.

- Libération de mémoire :

Lorsqu'on a fini d'utiliser notre variable et qu'on en a plus besoin, vous devez la libérer avec l'opérateur **delete**.

Exemple – Code en C++ - *delete* :

```
delete variable;          //Libération de mémoire
```

- Résumé :

new et **delete** étant des opérateurs, et non des fonctions, on ne met pas de parenthèses.

Le programme ci-dessous résume un exemple complet de l'allocation / libération de mémoire en C++ :

Exemple – Code en C++ - *new et delete* :

```
int main( )
{
    int *variable = NULL;
    variable = new int;          //Allocation de mémoire
    delete variable;             //Libération de mémoire
    return 0;
}
```

9.3) Allocation dynamique d'un tableau

- Allocation de mémoire :

Si on veut allouer un tableau, on n'a plus besoin de faire un calcul du type **20*sizeof(int)** comme on devait le faire en langage C. On va utiliser l'opérateur **new[ ]**.
On commence par créer le pointeur, puis on alloue la mémoire :

Exemple – Code en C++ - *new[ ]* :

```
int *tableau = NULL;
tableau = new int[20];          //Allocation de mémoire (20 cases)
```

Dans ce cas, un tableau de 20 cases sera alloué. Bien entendu, il est aussi possible de remplacer ce nombre par une variable :

Exemple – Code en C++ - *new[ ]* :

```
int *tableau = NULL;
tableau = new int[taille];      //Allocation de mémoire ("taille" cases)
```

La longueur du tableau sera définie par la valeur de la variable taille.

- Libération de mémoire :

Lorsqu'on n'a plus besoin du tableau, on doit le libérer... avec cette fois l'opérateur **delete[ ]** pour bien préciser qu'il s'agit d'un tableau. On n'a pas besoin de préciser la taille entre crochets, mais il ne faut pas oublier de les mettre quand même, c'est important pour le compilateur.

Exemple – Code en C++ - *delete[ ]* :

```
delete[ ] tableau;              //Libération de mémoire
```

- *Résumé* :

Le programme ci-dessous résume un exemple complet de l'allocation / libération de mémoire pour les tableaux en C++ :

Exemple – Code en C++ - *new[ ] et delete[ ]* :

```
int main( )
{
        int *tableau = NULL;
        tableau = new int[20];          //Allocation de mémoire (tableau)
        delete[ ] tableau;              //Libération de mémoire (tableau)
        return 0;
}
```

Il y a donc en tout 4 opérateurs :

a. **new** s'utilise avec **delete** pour allouer une variable.
b. **new[ ]** s'utilise avec **delete[ ]** pour allouer un tableau.

10) Les fonctions virtuelles et classes abstraites

10.1) Les fonctions virtuelles

Soit **f** une fonction membre d'une classe nommée **C**. Si les conditions suivantes sont réunies :

- **f** est redéfinie dans des classes dérivées (directement ou indirectement) de **C**,
- **f** est souvent appelée à travers des pointeurs ou des références sur des objets de la classe **C** ou de classes dérivées de **C**,

=> Alors **f** mérite d'être une **fonction virtuelle**. On exprime cela en faisant précéder sa déclaration du mot réservé « **virtual** ».

L'important service obtenu est celui-ci : si **f** est appelée à travers un pointeur ou une référence sur un objet de la classe **C**, le choix de la fonction effectivement activée, parmi les diverses redéfinitions de **f**, se fera d'après le type dynamique de cet objet.

Une classe possédant des fonctions virtuelles est dite classe « polymorphe ». La qualification **virtual** devant la redéfinition d'une fonction virtuelle est facultative : les redéfinitions d'une fonction virtuelle sont virtuelles d'office.

Exemple – Code en C++ : (**f** équivaut à la fonction ***seDessiner*** et **C** équivaut à la classe ***Figure***)

```
class Figure
{
        public:
                virtual void seDessiner( )
                {
                        // fonction passe-partout, probablement sans grand intérêt
                }
                ...
};

class Triangle : public Figure
{
        public:
                void seDessiner( )
                {
                        // implémentation du tracé d'un triangle
                }
                ...
};

class Ellipse : public Figure
```

```
{
        public:
                void seDessiner( )
                {
                        // implémentation du tracé d'une ellipse
                }
                ...
};

Figure *image[maxFigure];
int n = 0;
image[n++] = new Triangle(...);
image[n++] = new Figure(...);
image[n++] = new Ellipse(...);
...
for (int i = 0; i < N ; i++) image[i]->seDessiner( );
```

Maintenant, l'appel **image[i]->seDessiner( )** active la fonction redéfinie dans la classe de l'objet effectivement pointé par **image[i]**. Cela constitue, bien entendu, le comportement intéressant, celui qu'on recherche presque toujours. On notera que cet appel est légal parce que la fonction **seDessiner** a été déclarée une première fois dans la classe qui correspond au type statique de **image[i]**.

10.2) Les classes abstraites

Les fonctions virtuelles sont souvent introduites dans des classes placées à des niveaux si élèvés de la hiérarchie (d'héritage) qu'on ne peut pas leur donner une implémentation utile ni même vraisemblable.

Une première solution consiste à en faire des fonctions vides :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( )
                {
                }       // sera définie dans les classes dérivées
                ...
};
```

En fait, si la fonction **seDessiner** peut être vide c'est parce que :

- La classe **Figure** ne doit pas avoir d'objets ;
- La classe **Figure** doit avoir des classes dérivées (qui, elles, auront des objets) ;
- La fonction **seDessiner** doit être redéfinie dans les classes dérivées.

Bref, la classe **Figure** est une *abstraction*. Un programme ne créera pas d'objets **Figure**, mais des objets **Rectangle**, **Ellipse**, etc., de classes dérivées de **Figure**. Le rôle de la classe Figure n'est pas de produire des objets, mais de représenter ce que les objets Rectangles, Ellipses, etc., ont en commun. Or, s'il y a une chose que ces objets ont en commun, c'est bien la faculté de se dessiner. Nous dirons que ***Figure est une classe abstraite.***

On remarque que la fonction **seDessiner** introduite au niveau de la classe **Figure** n'est pas un service rendu aux programmeurs des futures classes dérivées de Figure, mais une contrainte : son rôle n'est pas de dire ce qu'est le dessin d'une figure, mais d'obliger les futures classes dérivées de Figure à le dire.

On peut traduire cet aspect de la chose en faisant que cette fonction déclenche une erreur. En effet, si elle est appelée, c'est que ou bien on a créé un objet de la classe Figure, ou bien on a créé un objet d'une classe dérivée de Figure dans laquelle on n'a pas redéfini la fonction seDessiner :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( )
                {
                        erreur("on a oublie de redéfinir la fonction seDessiner");
                }
                ...
};
```

Cette manière de faire est peu pratique, car l'erreur ne sera déclenchée qu'a l'exécution (c'est le programmeur qui a commis l'oubli mais c'est l'utilisateur qui se fera réprimander !).

En C++ on a une meilleure solution :

Définir une **fonction virtuelle pure**, par la syntaxe :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( ) = 0;          // Une fonction virtuelle pure
                ...
};
```

D'une part, cela officialise le fait que la fonction ne peut pas, à ce niveau, posséder une implémentation.

D'autre part, cela crée, pour le programmeur, l'obligation de définir cette implémentation dans une classe dérivée ultérieure, et permet de vérifier pendant la compilation que cela a été fait. Une fonction virtuelle pure reste virtuelle pure dans les classes dérivées, aussi longtemps qu'elle ne fait pas l'objet d'une redéfinition (autre que « = 0 »).

Pour le compilateur, une classe abstraite est une classe qui a des fonctions virtuelles pures. Tenter de créer des objets d'une classe abstraite est une erreur, qu'il signale :

Exemple – Code en C++ :

```
class Figure
{
        public:
                virtual void seDessiner( ) = 0;
                ...
};

class Rectangle : public Figure
{
        int x1, y1, x2, y2;
        public:
                void seDessiner( )
                {
                        trait(x1, y1, x2, y1);
                        trait(x2, y1, x2, y2);
                        trait(x2, y2, x1, y2);
                        trait(x1, y2, x1, y1);
```

```
        }
        ...
};
```

Utilisation :

```
Figure f;           // ERREUR : Création d'une instance (objet) de la classe abstraite Figure
Figure *pt;         // Oui : c'est un pointeur
pt = new Figure;    // ERREUR : Création d'une instance (objet) de la classe abstraite Figure
Rectangle r;        // Oui : la classe Rectangle n'est pas abstraite
pt = new Rectangle; // Oui
```

11) Annexes

11.1)Classes abstraites

Une classe est ***abstraite*** lorsque l'une au moins de ses méthodes **virtuelles** est ***pure***. Lorsqu'une classe est abstraite, elle ne peut être utilisée directement ; en particulier, on ne peut pas déclarer d'objets de cette classe. Elle a pour vocation de posséder des sous classes par héritage. Elle sert à factoriser des données membres et des méthodes communes.

```
Class Personne
{
      public:
              // Méthode virtuelle pure
              virtual double calculerSalaire(Mois m)  = 0;
              ----------------------------------------
} ;

class Personnel : public Personne
{
      protected:
              unsigned int numero_bureau;
      public:
              // Constructeur
              Personnel(unsigned int numBureau, const char*name, const char*adress=0);
              // Accès au numéro de bureau
              unsigned int getNumeroBureau() const { return numero_bureau; }
              void setNumeroBureau(unsigned int n)  { numero_bureau = n; }
              // Redéfinition Méthode virtuelle pure
              double calculerSalaire(Mois m) {____________________};
};
```

Nous définissons une méthode **virtuelle *calculerSalaire*,** mais à ce niveau de la hiérarchie, nous ne pouvons pas donner de méthode de calcul du salaire : par défaut, nous avons renvoyé zéro. Il est possible de ne pas définir une méthode virtuelle dans une classe : elle devra alors obligatoirement être définie dans les classes dérivées (« classe Personnel »).

11.2) Le pointeur this

Rappel et résumé :

Une classe constitue une sorte de type, et les objets de cette classe en sont des ***instances***. La classe définit donc la structure des données, alors appelées *champs* ou *variables d'instances*, que les objets correspondants auront, ainsi que les méthodes de l'objet.

À chaque instanciation, une allocation de mémoire est faite pour les données du nouvel objet créé. L'initialisation de l'objet nouvellement créé est faite par une méthode spéciale, le *constructeur*. Lorsque l'objet est détruit, une autre méthode est appelée : le *destructeur*. L'utilisateur peut définir ses propres constructeurs et destructeurs d'objets si nécessaire.

Le pointeur this :

Chaque fois qu'un programme crée une **instance** d'une classe, le C++ crée un pointeur spécial appelé ***this*** contenant l'adresse de l'instance courante de la classe.

Le C++ connaît le pointeur ***this*** uniquement lorsqu'une méthode membre non-statique est en cours d'exécution.

L'utilisation du pointeur ***this*** est inutile mais beaucoup de programmeur exploitent ce fait pour améliorer la clarté.

```
class uneclasse
{
   public:
      uneclasse(char *titre, char *auteur);
      void afficher_avec_this( );
      void afficher_sans_this(uneclasse livre);

   private:
      char titre[256];
      char auteur[256];
};
/* *****************Programme Principal ************** */
```

```
void uneclasse::afficher_avec_this( )
{
   cout << "LIVRE: "<< this->titre <<endl;
   cout << "Auteur: "<< this->auteur <<endl;
}

void uneclasse::afficher_sans_this(uneclasse livre)
{
   cout << "LIVRE: "<< livre.titre <<endl;
   cout << "Auteur: "<< livre.auteur <<endl;
   cout << "LIVRE: "<< this->titre <<endl;
   cout << "Auteur: "<< this->auteur <<endl;
}

uneclasse::uneclasse(char *titre, char *auteur)
{
   strcpy(uneclasse::titre, titre);
   strcpy(uneclasse::auteur, auteur);
}


int main( )
{
   //Déclarations et initialisations d'objets de classe "uneclasse"
   uneclasse livre("La bible", "Jamsa");
    uneclasse livre1("La bible1", "Jamsa");
   uneclasse livre2("La bible2", "Jamsa");

   livre.afficher_avec_this( );
   livre.afficher_sans_this(livre2);

    cout <<"\n\n" ;
    system("PAUSE") ;
    return 0;
}
```

```
LIVRE: La bible
Auteur: Jamsa
LIVRE: La bible2
Auteur: Jamsa
LIVRE: La bible
Auteur: Jamsa

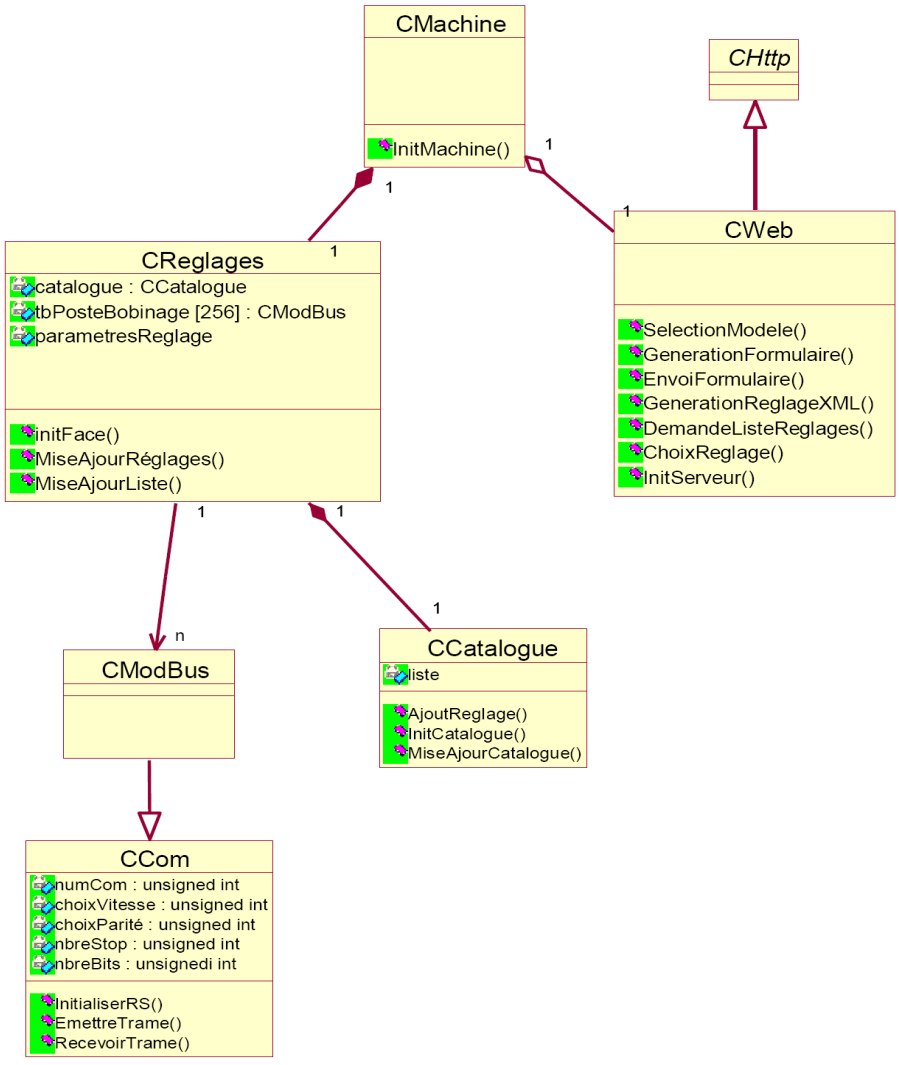

Appuyez sur une touche pour continuer...
```

Le pointeur ***this*** est le pointeur de l'objet courant ; il permet ici (voir exemple ci-dessus) que l'attribut et le paramètre puisse avoir le même nom.

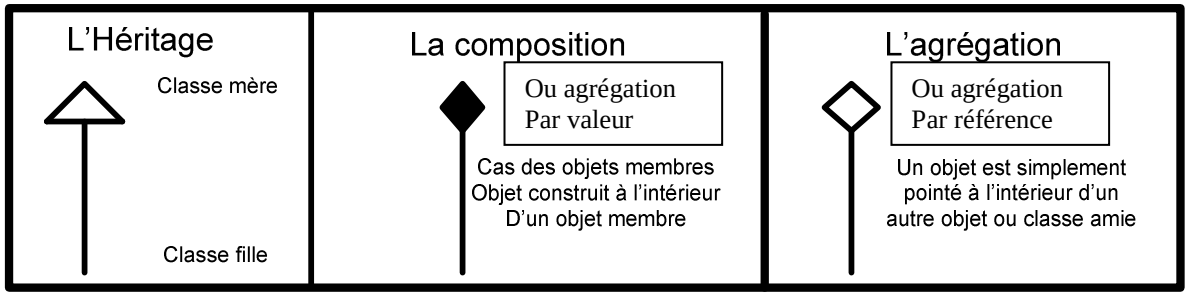
11.3) Le diagramme de classe

Un diagramme de classe permet de décrire les interactions, les différentes relations entre différentes classes dans un modèle d'un logiciel.

a) Exemple de diagramme de classe UML (BTS IRIS 2006)



b) Schématisation de la relation entre les différentes classes

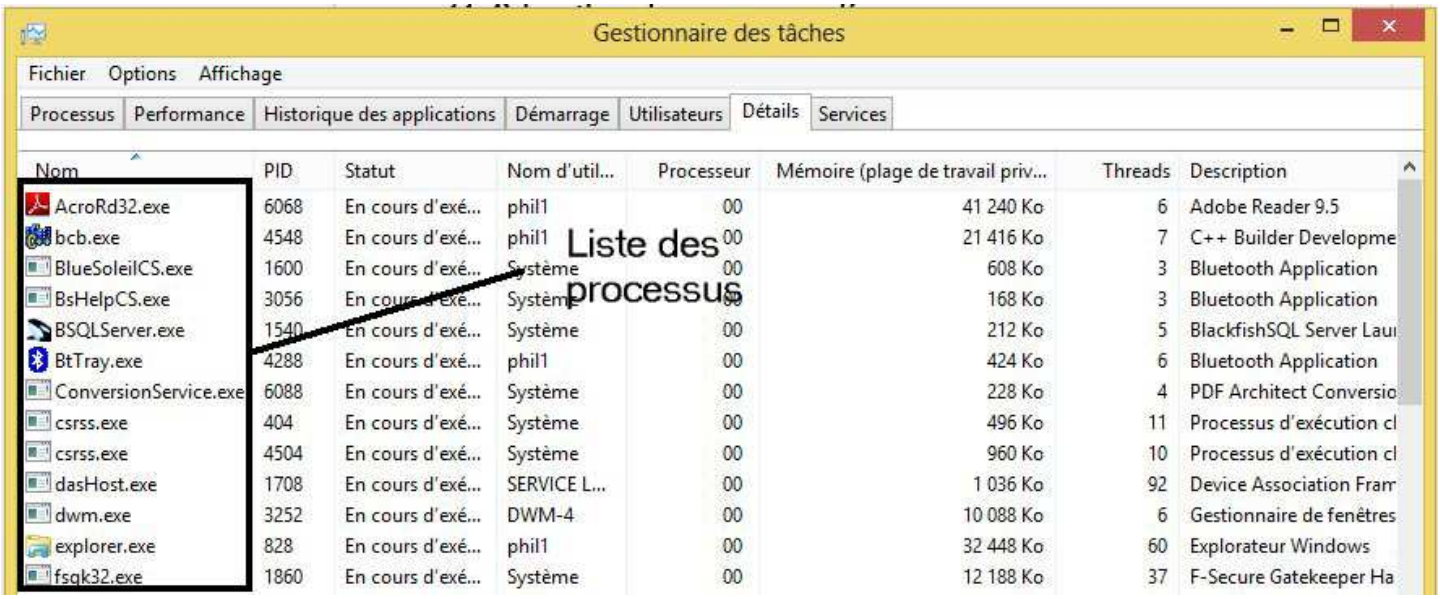


11.4) Les threads ou processus léger

Un processus lourd est un programme en cours d'exécution. Chaque processus lourd implique la gestion d'un espace d'adressage virtuel MMU et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc…).
Le principal avantage de ces processus est la protection mémoire entre les processus du même système.
Par contre, l'utilisation des processus lourds présente les inconvénients suivants :

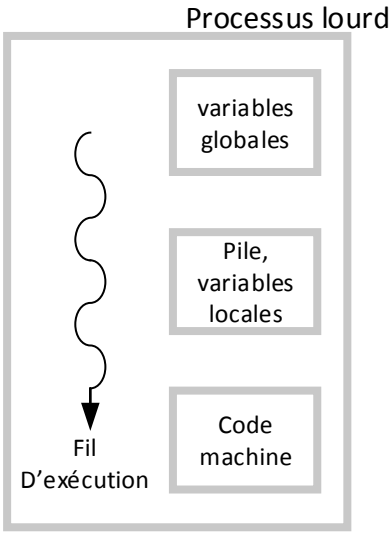
- leur création nécessite des appels systèmes coûteux en temps,
- le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- le coût des mécanismes de protection associés au processus,
- l'interaction, la synchronisation ou la communication entre processus nécessite l'utilisation de mécanismes de communication spéciaux (tube communicant appelé "*pipe*", *socket*, boîte aux lettres),
- le partage de mémoire entre processus s'effectue par ajout de mécanismes lourds (bibliothèque de partage de mémoire).

La liste des processus lourd est donnée par le gestionnaire des taches :

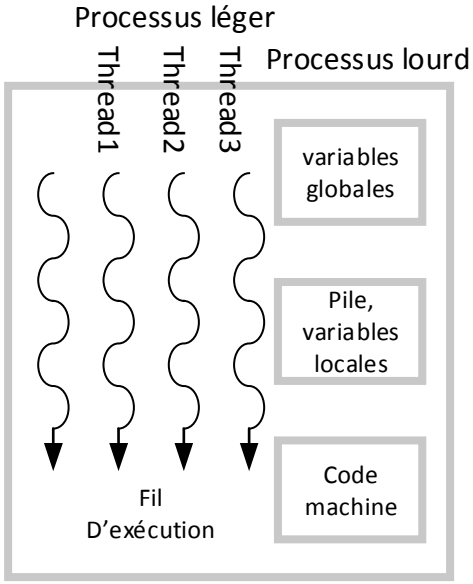


Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	Bluetooth Application
BsHelpCS.exe	3056	En cours d'exé...	Système	00	168 Ko	3	Bluetooth Application
BSQLServer.exe	1540	En cours d'exé...	Système	00	212 Ko	5	BlackfishSQL Server Lau
BtTray.exe	4288	En cours d'exé...	phil1	00	424 Ko	6	Bluetooth Application
ConversionService.exe	6088	En cours d'exé...	Système	00	228 Ko	4	PDF Architect Conversio
csrss.exe	404	En cours d'exé...	Système	00	496 Ko	11	Processus d'exécution cl
csrss.exe	4504	En cours d'exé...	Système	00	960 Ko	10	Processus d'exécution cl
dasHost.exe	1708	En cours d'exé...	SERVICE L...	00	1 036 Ko	92	Device Association Fram
dwm.exe	3252	En cours d'exé...	DWM-4	00	10 088 Ko	6	Gestionnaire de fenêtres
explorer.exe	828	En cours d'exé...	phil1	00	32 448 Ko	60	Explorateur Windows
fsgk32.exe	1860	En cours d'exé...	Système	00	12 188 Ko	37	F-Secure Gatekeeper Ha

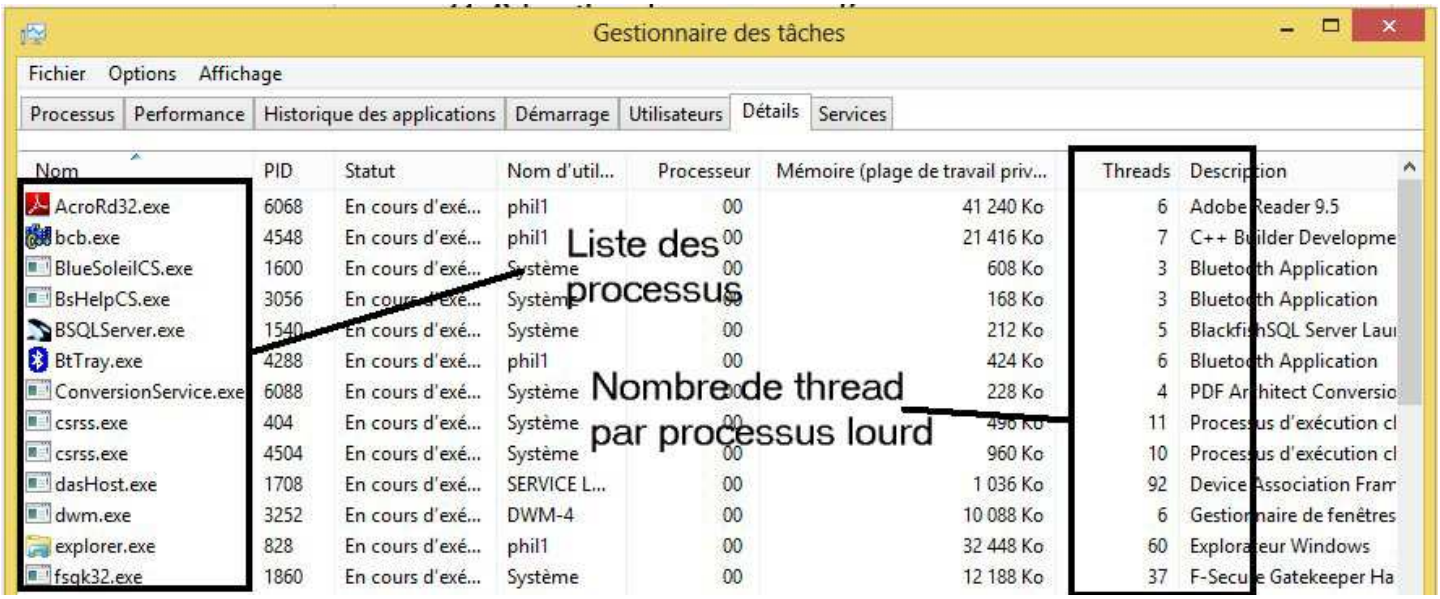
Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).



Les threads ou processus léger permettent de dérouler plusieurs blocs d'instructions, en PARALLELE, à l'intérieur du même processus. Un thread exécute une fonction. Chaque fonction associée à un thread va être exécutée de façon parallèle et indépendante.



Le système d'exploitation gère donc des centaines de thread :



Nom	PID	Statut	Nom d'util...	Processeur	Mémoire (plage de travail priv...	Threads	Description
AcroRd32.exe	6068	En cours d'exé...	phil1	00	41 240 Ko	6	Adobe Reader 9.5
bcb.exe	4548	En cours d'exé...	phil1	00	21 416 Ko	7	C++ Builder Developme
BlueSoleilCS.exe	1600	En cours d'exé...	Système	00	608 Ko	3	Bluetooth Application
BsHelpCS.exe	3056	En cours d'exé...	Système	00	168 Ko	3	Bluetooth Application
BSQLServer.exe	1540	En cours d'exé...	Système	00	212 Ko	5	BlackfishSQL Server Lau
BtTray.exe	4288	En cours d'exé...	phil1	00	424 Ko	6	Bluetooth Application
ConversionService.exe	6088	En cours d'exé...	Système	00	228 Ko	4	PDF Architect Conversio
csrss.exe	404	En cours d'exé...	Système	00	496 Ko	11	Processus d'exécution cl
csrss.exe	4504	En cours d'exé...	Système	00	960 Ko	10	Processus d'exécution cl
dasHost.exe	1708	En cours d'exé...	SERVICE L...	00	1 036 Ko	92	Device Association Fram
dwm.exe	3252	En cours d'exé...	DWM-4	00	10 088 Ko	6	Gestionnaire de fenêtres
explorer.exe	828	En cours d'exé...	phil1	00	32 448 Ko	60	Explorateur Windows
fsgk32.exe	1860	En cours d'exé...	Système	00	12 188 Ko	37	F-Secure Gatekeeper Ha

Dans un système monoprocesseur l'illusion sera donnée que chaque thread est exécuté de façon parallèle ou concurrente. En réalité chaque thread sera exécuté à tour de rôle mais très rapidement.

Avantages :

- **Multi-tâche moins coûteux** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux threads est moins coûteuse que la commutation de contexte de processus lourd
- **Communication entre threads plus rapide et plus efficace** : grâce au partage de certaines ressources entre threads, IPC (*Inter Processus Communication*) inutile pour les threads

- **Inconvénients** :

Programmation utilisant des threads est toutefois plus difficile : obligation de mettre en place des **mécanismes de synchronisation**.

Sous borland C++ un thread est un objet héritant de la classe Tthread. Cette classe se compose au minimum d'un constructeur public et d'une méthode protégée , Execute.
Pour créer un thread sous borland C++, choisissez fichier nouveau puis sélectionnez objet thread.



Enregistrez le fichier unit2.cpp en thread.cpp. Automatiquement le fichier unit2.h devient thread.h. Le code suivant apparait dans thread.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "thread.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------

//   Important : les méthodes et les propriétés des objets de la VCL ne peuvent être
//   utilisées que dans une méthode appelée en utilisant Synchronize, comme :
//
//      Synchronize(UpdateCaption);
//
//   où UpdateCaption serait de la forme :
//
//      void __fastcall thread::UpdateCaption()
//      {
//        Form1->Caption = "Mise à jour dans un thread";
//      }
//---------------------------------------------------------------------------

__fastcall thread::thread(bool CreateSuspended)
      : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall thread::Execute()
{
      //---- Placer le code du thread ici ----
}
//---------------------------------------------------------------------------
```

La fonction exécutée par le thread s'appelle Execute(). On peut la modifier afin d'inclure dans notre thread des objets de la VCL

```
void __fastcall thread::Execute()
{
while ( !Terminated )
 {
//Synchronize gère le multi-tâche (accès concurrents aux mêmes données,etc)
      Sleep(40);// endormissement de thread pendant 40ms
      Synchronize(AssignCtrl); //execution des objets de la VCL dans notre thread
      //----  Placer le code du thread ici hors VCL ----
}

 }
```

```
//     void __fastcall thread::  AssignCtrl ()
//     {
//       Form1->Caption = "Mise à jour dans un thread";
//     }
```

- Pour utiliser notre thread dans notre processus lourd de classe TForm il faut créer l'objet thread dans la méthode constructeur de TForm1

```
Thread *monthread ;
monthread = new Thread(false ) ;       // démarrage du thread si false
                                       // thread endormi si true
```

- Pour démarrer le thread on fera appel à la méthode resume()

```
monthread ->Resume() ;
```

La méthode Execute et AssignCtrl de monthread seront exécutés de façon concurrente avec les autres threads.

- Pour endormir un thread on fera appel à la méthode Suspend()

```
monthread ->Suspend() ;
```

Exception exemple

Le mieux pour comprendre le fonctionnement expliqué ci-dessus, est encore de reprendre l'exemple de la calculatrice et de la division par 0.

Exemple – Code en C++ - Rappel fonction **division** et **main** :

```
int division(int a, int b)
{
	return a/b;
}

int main( )
{
	int a, b;
	cout << "Valeur pour a: ";
	cin >> a;
	cout << "Valeur pour b: ";
	cin >> b;
	cout << a << " / "<<b << " = " << division(a,b) << endl;
	return 0;
}
```

Nous savons qu'une erreur peut survenir si **b vaut 0**, il faut donc lancer une exception dans ce cas. On va lancer une chaîne de caractère, car on peut ainsi décrire le problème survenu.

Exemple – Code en C++ - Fonction **division** avec **throw** :

```
int division(int a, int b)
{
	if(b == 0) throw string("Division par zero !");
	else return a/b;
}
```

Pour rappel, un **throw** doit toujours se trouver dans un bloc **try** qui doit lui-même être suivi d'un bloc **catch**. Ce qui donne la structure suivante :

Exemple – Code en C++ - Fonction **division** avec **throw-try-catch** :

```
int division(int a, int b)
{
	try
	{
		if(b == 0) throw string("Division par zero !");
		else return a/b;
```

```
        }

        catch(const string& chaine)
        {
                //On gère l'exception
        }
}
```

Il ne reste plus alors qu'à gérer l'erreur, c'est-à-dire par exemple, afficher un message d'erreur.

Exemple – Code en C++ - Fonction **division** avec **throw-try-catch** – Et message d'erreur :

```
int division(int a, int b)
{
        try
        {
                if(b == 0) throw string("Division par zero !");
                else return a/b;
        }

        catch(const string& chaine)
        {
                cerr << "ERREUR: " << chaine << endl;
        }
}
```

Ce qui donne le résultat suivant :

```
Valeur pour a: 10
Valeur pour b: 0

ERREUR: Division par zero !
10 / 0 = 4478048


Appuyez sur une touche pour continuer...
```

Remarque : Pour l'utilisation de **cerr** plutôt que **cout**, voir le chapitre **1.1)** page **4**.

Cette manière de faire est correcte. Cependant, cela ressemble un peu au mauvais exemple numéro 2°) du **4.2)** ci-dessus.

En effet, la fonction **division** peut potentiellement écrire dans la console alors que ce n'est pas son rôle. Le mieux à faire serait alors de lancer l'exception dans la fonction **division** et de récupérer l'erreur, si elle se produit, dans le **main**.

Exemple – Code en C++ - La meilleure solution :

```
int division(int a, int b)                    //Calcule a divisé par b
{
        if(b==0) throw string("Division par zero !");
        else return a/b;
}

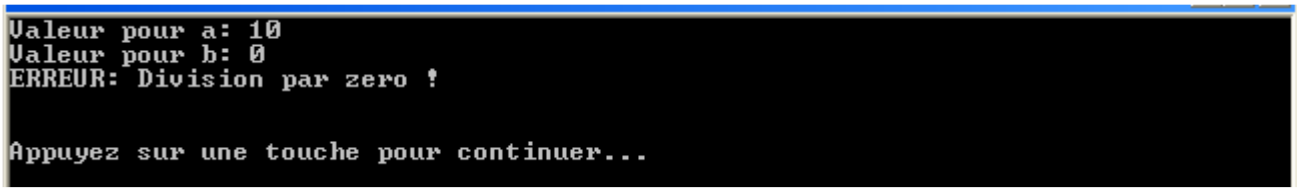
int main( )
{
        int a,b;
        cout << "Valeur pour a: ";
        cin >> a;
        cout << "Valeur pour b: ";
        cin >> b;

        try
        {
                cout << a << " / "<<b << " = " << division(a, b) << endl;
        }

        catch(const string& chaine)
        {
                cerr << "ERREUR: " << chaine << endl;
        }

         system("PAUSE");
        return 0;
}
```

Ce qui donne le résultat suivant :



Cette fois, le programme ne plante plus et la fonction n'a plus d'effet de bord (afficher n'importe quoi). C'est la meilleure solution.

- Conclusion :

Dans le cas d'une calculatrice, le fait de lever une exception peut se justifier, parce qu'il s'agit réellement d'une **erreur de l'utilisateur**. Mais ceci n'est pas toujours le cas.