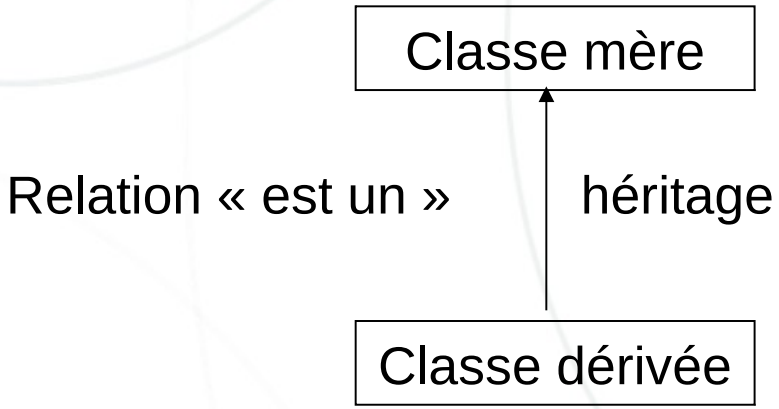
PROGRAMMATION ORIENTEE OBJET EN C++

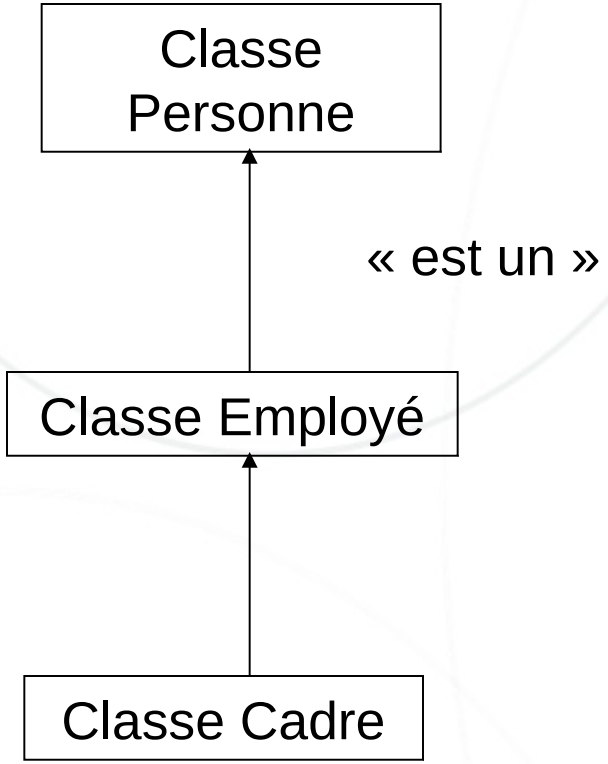
Héritage

Permet de spécialiser une classe

Traduit la relation « est un »



Héritage




```
class Personne
        {
                ...
        } ;
class Employe : Personne
        {
                ...
        };
class  Cadre : Employe
        {
                ...
        };
```

Héritage

Les appels de constructeurs se font dans l’ordre hiérarchique, de la classe de base vers la classe la plus spécialisée.

L'appel explicite d'un constructeur de la classe de base se fait avant la liste d'initialisation

Les destructeurs sont appelés dans l’ordre inverse des constructeurs.

Héritage

```
class base {
   int i;

   public :
   base(int n): i(n)
   { cout <<« constructeur base « << endl;}

   ~base()
   { cout<<« destructeur base »<<endl;}

    int val_i()
   { cout<<« i=« <<i<<endl;  return i ;}
   };
```

Héritage

```
#include « base.h »

class derive : public base {
    int j;

    public :
    derive(int a) : base(a)
    { cout<<« constructeur derive »<<endl;
      j=2*val_i();
       cout<<« j=« <<j<<endl; }

     ~derive()
     { cout<<destructeur derive »<<endl;}

};
```

```
int main()
{
   derive d(5);
   return 0;
 }
```

Affichage obtenu :

constructeur base
constructeur derive
i=5
j=10
destructeur derive
destructeur base

Encapsulation

Contrôle d'accès (protected):

```
class A {
                protected :     int x;
                void fa(...) {   ... x    // autorisé
                        }
        };
class B : A {   ...
                void fb(...) {  ...  x  // autorisé
                              }
                friend void fb_amie(B b1);
            };
void fb_amie(B b1)  {   ... b1.x;  b1.fa(...); }  // autorisé
```

Héritage

Public, protégé ou privé (par défaut) :

class derivee : specif_acces nom_classe_base { ... };

ex : class B : public A { ... };

- public : membres **public** de A sont **public** dans B.
membres **protected** de A sont **protected** dans B.
membres **private** de A sont inaccessibles dans B.

- protected:
membres **public** et **protected** de A sont **protected** ds B.
membres **private** de A sont inaccessibles dans B.

- private :
membres **public** et **protected** de A sont **private** dans B.
membres **private** de A sont inaccessibles dans B.

Héritage

```
class A {
        int a; // membre privé
      public :
        int b,c;
        int Afonc();
      };
class B : private A        //  b, c, Afonc() privés dans B
      {   int d;
       public :
        A::c; // c est redéfini comme public dans B
        A::a;   // impossible
        int e;
        int Bfonc();    // Bfonc() a accès à :
       };                       //  c,b,d,e, Afonc()
void externfonc(B &x);  // externfonc a accès à : c,e, Bfonc()
```

Les fonctions virtuelles

C++ possède à la base un typage statique(pas de ligature dynamique) :

```
class Base {
    public :
    void f1() { ... }
};
class Derive : public Base {
    public :
 // redéfinition de f1() de Base
    void f1() { ... }
    void f2() { .... }
};
```

```
Base b;
Derive d;
Base * ptb= &b;
Derive * ptd=&d;
ptb->f1();   // équivalent à b.f1()
ptd->f1();    // équivalent à d.f1()
ptb= ptd;        // autorisé
ptb->f2();  //  => illicite car ptb catalogué »
                    comme de classe base
ptb->f1();   // équivalent à l'appel
                    de f1() de Base
```

Les fonctions virtuelles

Pour réaliser une ligature dynamique, on déclare la fonction *virtual* (dans le prototype uniquement):

```
class Base {
        public :
                virtual void f1() { ...  }
        };
class Derive : public Base {
        public :
                void f1() { ...  }     // redéfinition de f1() de Base
                void f2() { .... }
        };
...
ptb= ptd;       // autorisé
ptb->f1();   // équivalent à l'appel de f1() de Derive , tient compte du
                type actuel de l'objet pointé
```

Les fonctions virtuelles

Si une fonction est déclarée virtuelle dans une classe, elle conservera sa « virtualité » pour toutes ses classes dérivées

```
class DeriveEncore : public Derive {
        public :
                void f1() { …  }     // redéfinition de fonc() de Derive
        };
Derive d;
DeriveEncore de;
Derive * ptd=&d;
DeriveEncore *ptde =&de;
ptd= ptde;        // autorisé
ptd->f1();   // équivalent à l'appel de f1() de DeriveEncore
```

Les classes abstraites

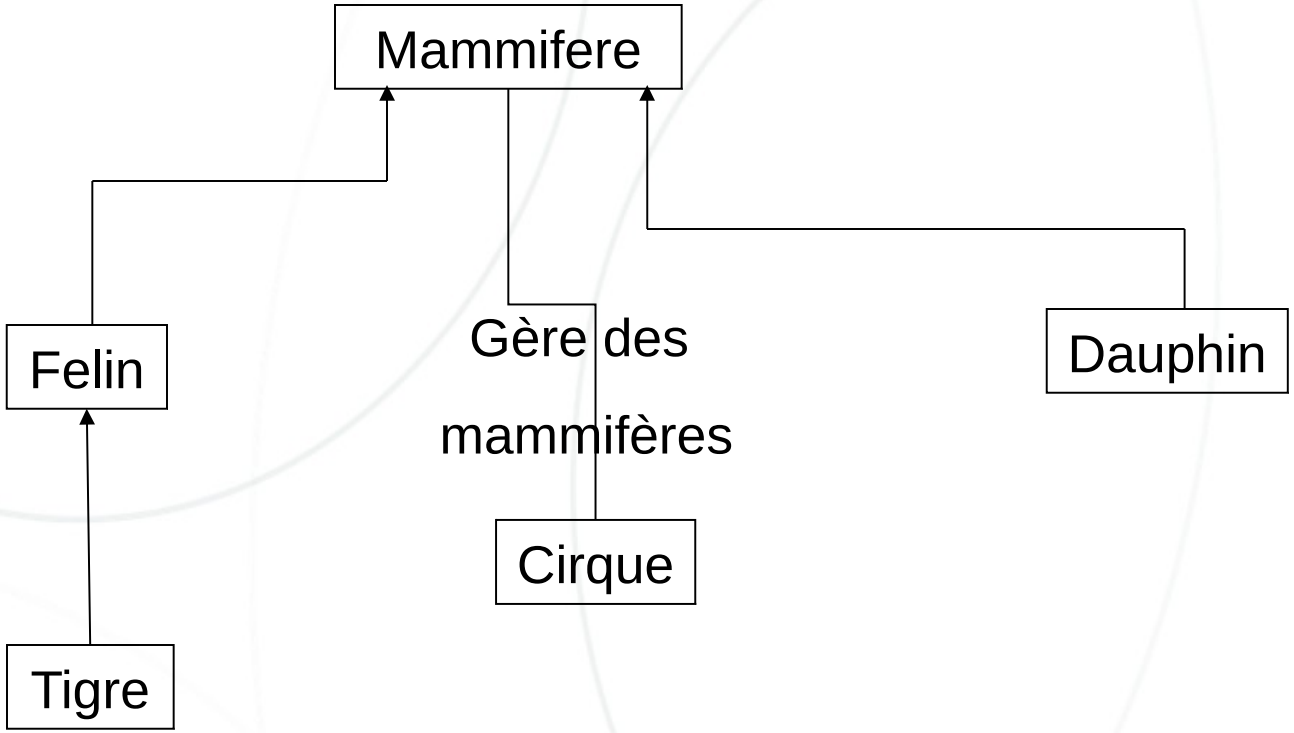
- Classe servant de « squelette » à des classes dérivées, ces dernières devant compléter les objets.
- Réservoir de connaissances.
- Ne peuvent pas être instanciées.
- Possèdent au moins une fonction abstraite (fonction virtuelle pure) :

  virtual type_retour nom_fonction(liste params) **=0** ;

- Une interface en Java est en C++ une classe abstraite ne possédant que des fonctions virtuelles pures

Les classes abstraites

– Exemple récapitulatif :



Les classes abstraites

```
class Mammifere  {   // classe abstraite
    protected  :
        String nom;
    public :
        Mammifere(String nom= « inconnu »);

          virtual void affiche()     // fonction virtuelle
        { cout <<‘’c’est un ’’<<nom<<endl; }

          virtual void parler() =0;     // fonction virtuelle pure
    };

Mammifere::Mammifere(String s) { nom=s; }
```

Les classes abstraites

```
class Felin : public Mammifere
        {
                bool domestique;
        public :
                Felin(String s, bool d) ;
                void est_domestique();
                virtual void parler() =0; //  parler reste une fonction
                                          //   virtuelle pure
};
Felin::Felin(String s, bool d): Mammifere(s)
        {     domestique=d;   }

void Felin::est_domestique()
        {      if (domestique)
                cout <<« le »<<nom<<« est domestique »;
              else  cout<<« le »<<nom<<« n'est pas domestique »;
              cout <<endl;
        }
```

Les classes abstraites

```
class Tigre : public Felin
{
        String race;
        public :
                Tigre(String r) ;
                void affiche();
                void parler() {  cout <<« il feule »<<endl;   }
};

Tigre::Tigre(String r): Felin(« Tigre »,false) {     race=r;   }

void Tigre::affiche()
{
          Felin::affiche();
          cout<<«c'est un »<<race<<endl;
}
```

Les classes abstraites

```
class Dauphin : public Mammifere
{
        public :
                Dauphin() ;
                void affiche() { Mammifere::affiche();}
                void parler()
                        {  cout <<« il crie (?) »<<endl;   }
};

Dauphin::Dauphin(): Mammifere(« Dauphin ») { }
```

Les classes abstraites

```
class Cirque
{
        list<Mammifere *> liste PT;

            public :
                Cirque() ;
                void saisirliste(Mammifere &m);
                void editerliste();
};
```

Les classes abstraites

```
void Cirque::saisirliste(Mammifere * m)
{
      liste.push_back(m);
}

void Cirque::editerliste()
{
   Int i=0 ;
   list<Mammifere*>::const_iterator it ;
    for( it=liste.begin();it!=liste.end(); it++,i++)
    {
      cout<<endl<<« Mammifere numero « <<(i+1)<<« : «<<endl;
      (*it)->affiche();
      (*it)->parler();
     }
}
```

Les classes abstraites

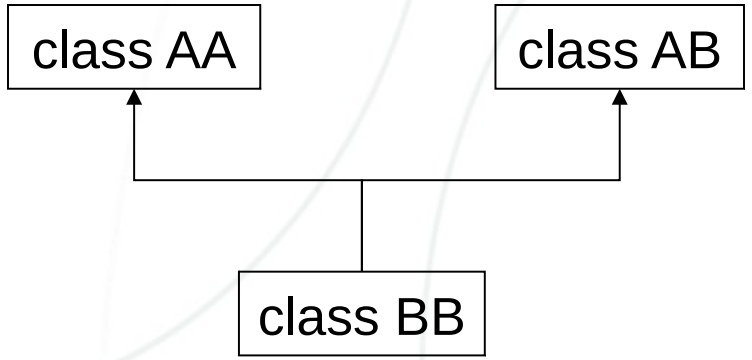
```
void main()
    {
      Tigre t(« Royal »);
      t.affiche();                    c'est un Tigre c'est un Royal
      t.parler();                     il feule
      Dauphin d;
      Mammifere &pm=d;  // accepté car référence
      d.affiche();                    c'est un Dauphin
      d.parler();                     il crie (?)
      Cirque ci;
      Tigre t1(« Bengale »);
      ci.saisirliste(&t); ci.saisirliste(&d); ci.saisirliste(&t1);
      ci.editerliste();
      cin.get();
    }
```

Mammifere numero 1 : c'est un Tigre c'est un Royal il feule

Mammifere numero 2 : c'est un Dauphin il crie (?)

Mammifere numero 3 : c'est un Tigre c'est un Bengale il feule

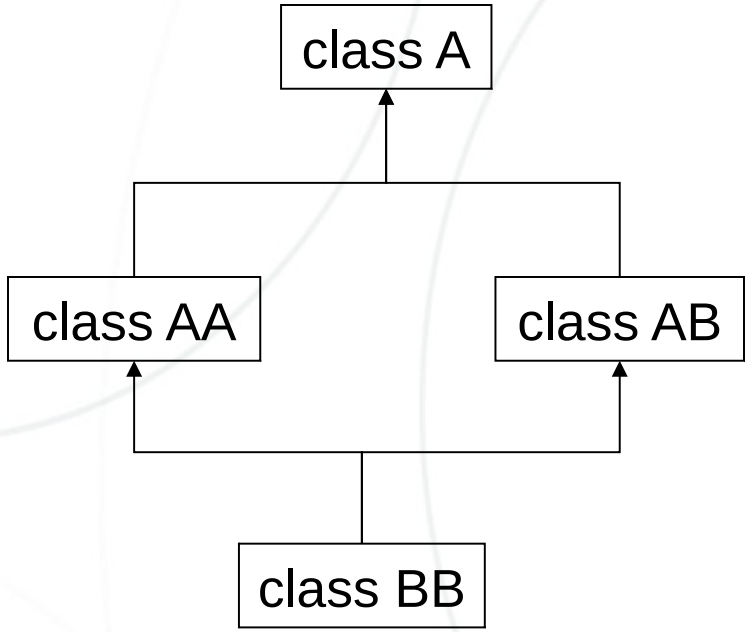
Héritage multiple




```
class BB : public AB, public AA
        {
        public :
                BB(int n) : AB(2*n), AA(n)
                        { cout <<''constructeur de BB''<<endl; }
        };
```

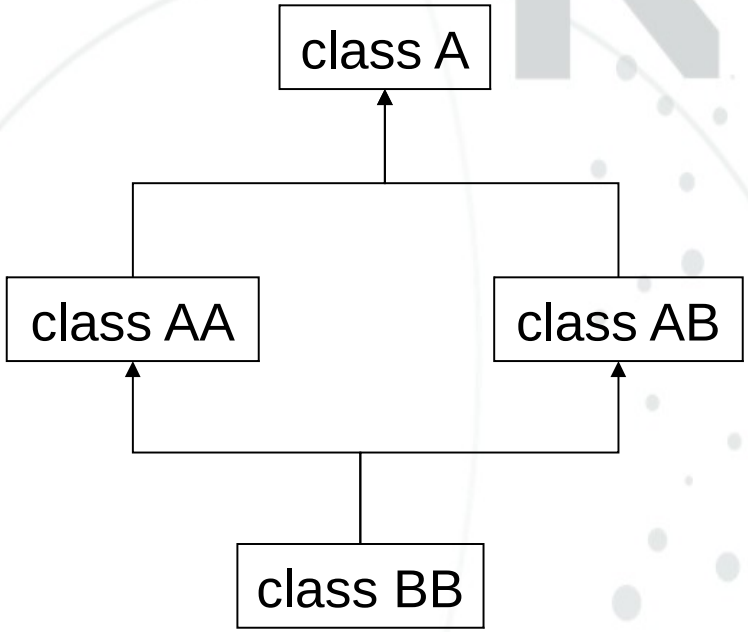
L’héritage multiple

Problème d’un héritage multiple :



L'héritage multiple

```
class A  {
    protected  : int i;
    public :
    A(int n): i(n)
    { cout <<''constructeur de A''<<endl; }
};
class AA : public A
{
    public :
    AA(int n) : A(n)
    { cout <<''constructeur de AA''<<endl; }
};
class AB : public A
{
    public :
    AB(int n) : A(2*n)
    { cout <<''constructeur de AB''<<endl; }
};
```



L’héritage multiple

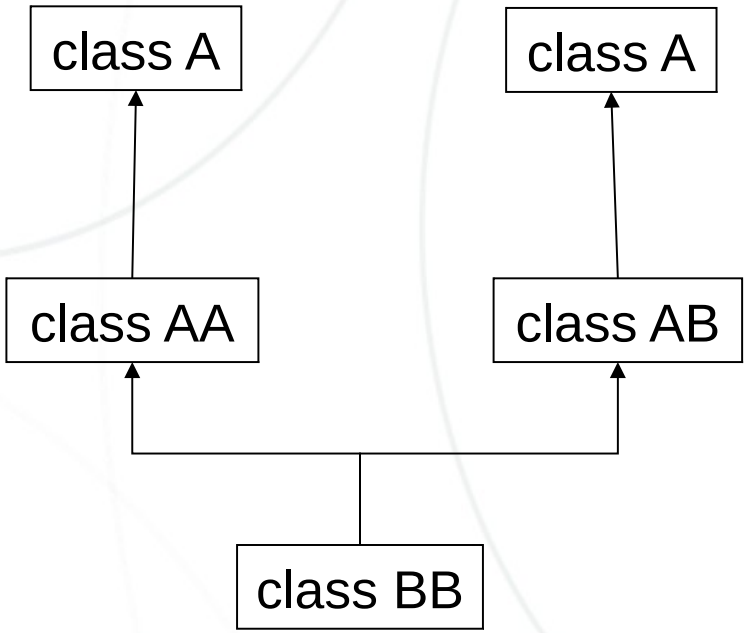
```
class BB : public AB, public AA
{
public :
BB(int n) : AB(2*n), AA(n)
{ cout <<‘’constructeur de BB’’<<endl; }
void affiche() {
     cout <<‘’i de AB=‘’<<AB::i<<end;
     cout <<‘’i de AA=‘’<<AA::i<<endl
}
};
BB bb(4);
bb.affiche();
```

-> constructeur de A
constructeur de AB
constructeur de A
constructeur de AA
constructeur de BB
i de AB=16
i de AA=4

L’héritage multiple

Problème d’un héritage multiple :

- Tout ce passe comme si le graphe d’héritage était **:**



L’héritage multiple

Problème d’un héritage multiple : les classes virtuelles

- Pour éviter ce problème, on utilise le mot **virtual** dans la déclaration des classes intermédiaires :

```
class AA : public  virtual A
{
public :
AA(int n) : A(n)
{ cout <<‘’constructeur de AA’’<<endl; }
};
class AB : public virtual A
{
public :
AB(int n) : A(2*n)
{ cout <<‘’constructeur de AB’’<<endl; }
};
```

L'héritage multiple

Problème d'un héritage multiple :

- De plus, le constructeur de BB devra faire un appel au constructeur de la classe de base :

```
class BB : public AB, public AA
        {
        public :
                BB(int n) :A(n), AB(2*n), AA(n)
                        { cout <<''constructeur de BB''<<endl; }
                void affiche() {
                        cout <<''i de AB=''<<AB::i<<end;
                        cout <<''i de AA=''<<AA::i<<endl
                        }
        };
BB bb(4);
```

```
constructeur de A
constructeur de AB
constructeur de AA
 constructeur de BB
```

```
bb.affiche();
```

```
    i de AB=4
    i de AA=4
```

=> TP 5

Boost

Ensemble de bibliothèques gratuites et portables, visant à être intégrées au prochain standart C++, pour :

- Les threads
- La gestion du temps (date et heure)
- La sérialisation
- La manipulation de chaines
- ...

Boost Serialization

Processus visant à coder l'état d'une information en mémoire sous la forme d'une suite d'informations plus petites.

Ce mécanisme pourra par exemple être utilisé pour la sauvegarde (persistance) ou le transport sur le réseau.

L' activité symétrique s'appelle la désérialisation.

Boost Serialization

```
Int main()
{       Point p1(0,0), p2(1,1) ;
        std ::ofstream ofs(« monfic.sav »);
        {
             boost::archive::text_oarchive  oa(ofs);
             oa<<p1<<p2 ;
        } // destruction de l'archive oa

          … // modifications de p1 et p2

        std::istream ifs(« monfic.sav »);
        boost::archive::text_iarchive ia(ifs);
        ia >>p1>>p2 ;  // retour état initial p1 et p2
        …
```

Sérialization de base

Pour qu'un objet soit sérializable, il faut qu'il implémente la fonction template :

template <class Archive>

void serialize (Archive &ar, const unsigned int version)

Les types d'Archive :

- Texte : text_oarchive , text_iarchive
- Binaire : binary_oarchive , binary_iarchive
- XML : xml_oarchive, xml_iarchive

Sérialization de Base

```
#include <boost::serialize::text_iarchive.hpp>
#include <boost/ archive /text_iarchive.hpp>
#include <boost::serialize::text_oarchive.hpp>
#include <boost/ archive /text_oarchive.hpp>
Class Point{
        float x,y ;
        friend class boost::serialize::access; // nécessaire si fonction serialize private
        friend class boost::serialization::access;
        template <class Archive>
        void serialize(Archive & ar, const unsigned int version)
        {
                ar & x ;
                ar & y ;
                // ar & x & y ;  possible  aussi
        }  // NB : fonction template à définir dans le header !!!
        ...
};
```

Sérialization de Base

VERSIONS de SERIALIZATION :

```
#include <boost/serialization/version.hpp>
Class Point{
        float x,y,z ;
        friend class boost::serialize::access;
        template <class Archive>
        void serialize(Archive & ar, const unsigned int version)
        {
              ar & x &y ;
              if (version >0 )
                   ar & z
        }  // NB : fonction template à définir dans le header !!!
        ...
};
 BOOST_CLASS_VERSION(Point , 1)
```

Sérialization membres

```
#include <boost/ archive /text_iarchive.hpp>
#include <boost/ archive /text_oarchive.hpp>
Class Point{
        Position x,y ;
        friend class boost::serialization::access; // nécessaire si fonction serialize
          private
        template <class Archive>
        void serialize(Archive & ar, const unsigned int version)
        {
                ar & x ;  // OK si classe Position serializable
                ar & y ;
                // ar & x & y ;  possible  aussi
        }  // NB : fonction template à définir dans le header !!!
        ...
};
```

Sérialization Array STL

Tableaux / Pointeurs :

```
Point poly[5] , *poly2 = new Point[3] ;
 ...
ar & poly ;
ar & poly2 ;   // ok sans polymorphisme
```

STL Collections :

```
#include <boost / serialization / list.hpp>
list<Point> l ;
...
ar & l ;
```

Sérialization classes dérivées

```
#include <boost/serialization/base_object.hpp>

Class PointCouleur : public Point{  //Point doit être sérializable !!!
        string couleur ;
        template <class Archive>
        void serialize(Archive & ar, const unsigned int version)
        {
                ar & boost::serialization::base_object<Point> (*this) ;
                ar & couleur ;
        }
        ...
       };
```

Exceptions

Une exception est l'interruption de l'exécution d'un programme à la suite d'un évènement particulier.

- Regroupement des cas anormaux ou erreurs
- Différentier les anomalies

  (types d'exception)
- Reporter le traitement des erreurs

Exceptions

Exemple :

```
int main()
{
vector<int> test ={ 5,4,3,2,1} ;    // c++11
... //saisie d 'une valeur n  entre 0 et 4
cout<<test.at(n) ;
return 0 ;
}
```

terminate called after throwing an instance of 'std::out_of_range'

Exception

Traitement de l'exception :

```
try{
        code susceptible de lancer des exceptions
}
catch(Exception1& e1)
{ ...message1 }
catch (Exception1& e2)
{...message2 }
catch (std::exception& e)
{ ... message général}
```

Exception

Récupération d'exception :

```
vector<int> v={4,3,2,1};
try{
        … //saisie de n
        cout<<v.at(n);
    }
    catch(std::exception& e)
    {
        cout<<"valeur comprise entre 0 et 4 !!! ";
    }
```

Exception

Lancer une exception :

```
#include <stdexcept>
...
vector<int> v={4,3,2,1};
try{
        ... //saisie de n
        if (n >4)  throw out_of_range (« indice trop grand ») ;
        cout<<v[n];
    }
    catch(std::exception& e)
    {
      cout<<e.what() <<"valeur comprise entre 0 et 4 !!! ";
    }
```

Exception

Création :

```
#include <exception>
class MonException : public std ::exception
{
      string mess ;
      public :
            MonException (string nomfic, int no_lig){
            mess = « Erreur MonException dans »+ nomfic+
                        « line  »+ to_string(no_lig) ; //c++11
            }

            const char* MonException::what() const throw()
            {
               return mess.c_str();
            }
 };
```

=> TP 6

Le constructeur par recopie

Par défaut, celui-ci consiste à recopier chaque membre d'un objet dans le membre correspondant de l'autre objet

Il est appelé dans les cas suivants :

- Création d'un objet à partir d'un autre de même type
- Passage par valeur d'un objet
- Listes d'initialisation

Le constructeur par recopie

```
Point p1 ; // ici constructeur sans paramètre
Point p2=p1;
Point p3(p1);
p1.distance(p2);
```

```
class Point{
                ...
                distance(Point p){ ....}
};
```

```
Segment s(p1,p2);
```

```
class Segment{
        Point x, y;

        Segment(Point px, Point py) :x (px)
        {y=py;}
  }; //fin de la classe Segment
```

=> appel constructeur par recopie => appel constructeur sans paramètre=> appel opérateur égalité

Relations entre classes

Composition :

```
class Segment{
    Point x1;
    Point x2;
 public:
Segment(Point px,Point py):
    x1(px) , x2(py) {}
};
```

Composition (dynamique):

```
class Segment{
    Point  *x1;
    Point  *x2;
 public:
Segment(Point px, Point py) :
      {   x1 = new Point(px);
          x2 = new Point(py);
       }
};
```

Agrégation :

```
class Segment{
    Point *x1;
    Point *x2;
 public:
Segment(Point *px, Point
    *py) : x1(px), x2(py) {}
};
```

Constructeur par recopie

Si un objet possède soit des champs constants, soit des attributs alloués dynamiquement, le constructeur par recopie doit être redéfini :

```
Point  p0(0,0), p1(0,5);
Segment s1(p0,p1) , s2 = s1;
 s1.translater(10) ;
 => les champs de s2 seront également translatés
```

Constructeur par recopie

```
class Segment{
        Point  *x1;
        Point  *x2;
        public:
        Segment(Point px, Point py) : x1(new Point(px)), x2(new
Point(py))                {}

        Segment(const Segment& s){
            x1 = new Point(*(s.x1));
            x2 = new Point(*(s.x2));
        }
};
```

NB : le paramètre étant une référence constante, ce constructeur sera appelé pour des objets constants ou non-constants

Héritage

Constructeurs par recopie :

- Si la classe dérivée nécessite un constructeur par recopie,

  => appel explicite du constructeur par recopie de la classe mère
- Sinon, appel du constructeur sans paramètres de la classe mère
- Si ce dernier n'existe pas => erreur à la compilation

Héritage

```
class base {
    int *i;     // attribut  pointeur sur entier
    public :
    base(int n) {
    i=new int;
    *i=n;
    cout <<« constructeur base « << endl;
    cout <<« *i=« <<*i<< endl;}

    ~base() { cout<<« destructeur base »<<endl;
    delete i;}
    int val_i() { cout<<« *i=« <<*i<<endl;  return *i ;}

                    base (const base &b) {  // constructeur par recopie
    i=new int;
    *i=*b.i;
    cout << « *i recopie =« <<*i<<endl;}
    };  // fin classe base
```

Héritage

```
class derive : public base {
    int *j;
    public :
    derive(int a) : base(a)
    { cout<<« constructeur derive »<<endl;
      j=new int;
      *j=2*::val_i();
     cout<<« *j=« <<*j<<endl;}

    ~derive() { cout<< « destructeur derive »<<endl;
          delete j ; }

    derive(const derive & d) : base(d) {
    j=new int;
    *j=*d.j;
    cout<<« *j recopie=« <<*j<<endl; }
    };
```

Héritage

```
int main(){
derive d(8);
derive e=d;
return 0;}
```

Affichage obtenu :

```
constructeur base   // objet d
*i=8
constructeur derive
*i=8
*j=16
*i recopie=8  // objet e
*j recopie=16
destructeur derive  // objet e
destructeur base
destructeur derive   // objet d
destructeur base
```

Héritage

Surcharge opérateur = :

=>De même que pour le constructeur par recopie, appel explicite de la surcharge de cet opérateur dans la classe mère

```
class derive : public base{
...
derive& operator=(const derive& d){
base::operator=(d);
...}
};
```

Surcharge d’opérateurs

- Si la classe possède des membres pointeurs :

1. constructeur par recopie
2. opérateur d’affectation
3. destructeur

=> *Forme de Coplien*

=> TP 7

Les classes patrons

Un patron de classe :

- Définit un modèle de classe.
- Sert à modéliser une famille de classe paramétrée par une liste d'identificateurs qui représentent des valeurs et des types.
- On parle de classe générique.
- À partir d'une classe générique => multitude de classes concrètes, différentes entre elles.

Les classes patrons

Un patron de classe :

- Syntaxe :template <class T1, class T2,...> class nom_class { ... };
- La classe « nom_class » est paramétrée par les types formels T1,T2, ...
- Les paramètres T1, T2, ... symbolisent des identificateurs de types inconnus (au moment de la définition de la classe) qu'il faudra obligatoirement préciser par des types connus lorsque l'utilisateur instanciera la classe.

  Exemple :

  nom_class<int, Pile> i1, i2;
- Le type « int » remplace maintenant le type inconnu T1, Pile le type T2 pour les instances i1 et i2.

  nom_class <Pile,int> j1,j2;

  =>i1 et i2 de même type

  =>j1 et j2 de même type

  =>i1 et j1 de types différents

Les classes patrons

Un fonction membre d'une classe patron :

```
template <class T1, class T2,class T3> class nom_class{ ...
  void fonc(T1 t1, T2 t2, T3 t3);   // fonction membre de la classe
  ...
 };  // fin de classe

template  <class T1,class T2, class T3>
     void nom_class<T1,T2,T3>::fonc(T1 t1, T2 t2, T3 t3)
                { ...}
```

Les fonctions et classes template sont instanciées lorsque la liste de leurs paramètres est fournie explicitement

=>les déclarations et les définitions de fonctions sont regroupées dans les fichiers d'en-tête

Les classes patrons

Exemple : une classe tableau paramétrée

```
template <class T> class Tableau
{
  int taille;
  T* pt;
public :
  Tableau();
  ~Tableau() { delete [] pt; }
  T & operator[](int i)  { // surcharge de []
            if (i<taille) return pt[i];
            else {
                    cerr<<'' indice trop grand ''<<endl;
                    return pt[0];  // par exemple
                 }
            }
  void affiche();
  void remplit();
  int size()  { return taille;}
};  // fin de classe
```

Les classes patrons

```
template <class T>  Tableau<T>::Tableau()
    {
     int s=10;
     cout <<''nombre d'éléments : '' ;
     cin>> s;
     cin.get();
     pt=new T[taille=s];
}    // fin du constructeur

template <class T> void Tableau<T>::affiche()
   {
     for (int i=0; i<taille; i++)
                cout<<pt[i]<<endl;
}
```

Les classes patrons

```
template <class T>  void Tableau<T>::remplit()
    {
     T tampon;
     for (int i=0; i<taille; i++)
                {
                cout<<''valeur :'';
                cin>>tampon;
                cin.get();
                pt[i]=tampon;
                }
    }
void main()
    {
      Tableau<int>  t1;
      t1.remplit();
      t1.affiche();
    }
```

Les classes patrons

Cas particulier de paramètre constant :

- Exemple 1:

```
template <int max,class T> class Tableau
   {
     ...
     Tableau(int dim)
     {          if (dim < max)  pt=new T[taille=dim];
                                else      pt=new T[taille=max];
     }
     ...
};
Tableau<50,int> t1;
Tableau<100,int> t2;    // t1 et t2 de type différents
```