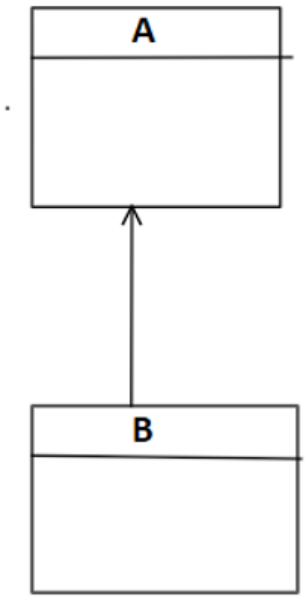
Héritage – Classes Abstraites – Polymorphisme en C++

I - Héritage

L'héritage est une technologie orientée objet fondamentale qui permet de mettre en place les classes abstraites, le polymorphisme et la réutilisation de code.

Soient 2 classes A et B quelconques, chaque fois que l'on peut dire qu'un objet B est aussi un objet de A , on dit alors que la classe de B hérite de la classe A. La relation « est un « doit être définie entre 2 entités pour qu'il y ait héritage .



La classe A est alors appelée classe de base ou superclasse tandis que la classe B est appelée sous- classe ou classe fille ou classe dérivée.

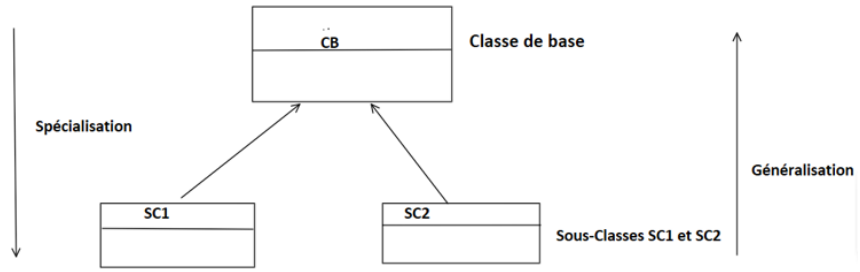
Dans A, on déclare les attributs propres à A ,un constructeur , des getters et setters.

Dans B, on déclare les attributs spécifiques à B, un constructeur, des getters et setters. Les attributs et méthodes de A sont automatiquement disponibles dans B sans les y réécrire. Il y alors réutilisation de code dans B.

Pour construire un objet B, il faut d'abord construire un objet A et l'habiller comme B. Cela veut dire que le constructeur de la classe B doit d'abord faire obligatoirement appel au constructeur de A.

2 cas sont possibles :

- **La classe de base et la sous-classe possèdent chacune un constructeur par défaut ou un constructeur interactif .Dans ce cas , le constructeur de B fait automatiquement appel au constructeur de A.**
- **La classe de base et la sous-classe possèdent chacune un constructeur avec paramètres. Dans ce cas, le constructeur de B fait appel explicitement au constructeur de A par une instruction d'appel.**



De manière générale, l'héritage utilise 2 techniques : la généralisation et la spécialisation.

La généralisation consiste à regrouper les attributs communs aux sous-classes (SC1 et SC2) et à les placer dans la classe de base (CB). Ces attributs seront ensuite automatiquement disponibles dans les sous-classes par héritage.

La spécialisation consiste à laisser les attributs non communs dans leurs sous-classes respectives.

Du point de vue code, on décrira les classes A et B de la manière suivante :

```
class A
{
private :
// attributs de A
public :
// méthodes de A
}
```

class B : public A

{

private :

// attributs de B uniquement

public :

// méthodes de B uniquement

}

L'héritage multiple existe. Dans ce cas , une sous-classe peut hériter de plusieurs classes de base .

1-1 Héritage simple avec des constructeurs interactifs

Exemple 1 : La classe de base et la sous- classe possèdent chacun un constructeur interactif. Le constructeur de la sous-classe appelle automatiquement le constructeur de la classe de base sans instruction. On dispose d'une classe de base personne et d'une sous-classe etudiant avec etudiant qui hérite de personne.

A / fichier personne.h

#ifndef PERSONNE_H

#define PERSONNE_H

#include <iostream>

#include <string>

using namespace std;

Commenté [n1]:

Commenté [n2R1]:

Commenté [n3R1]:

```
class personne
{
private:
   string nom;
   string prenom;

public:
   personne() ;
   string getnom();
   string getprenom ();

};
#endif
```

B/ fichier personne.cpp

```
#include "personne.h"
#include <iostream>
#include <string>
using namespace std;

personne::personne()
{
   cout<<"Nom ? "<<endl;
```

```
   getline(cin,nom);
   cout<<"Prénom ? "<<endl;
   getline(cin,prenom);

}

string personne::getprenom()
{
   return prenom;
}

string personne::getnom()
{
   return nom;
}
```

C/ fichier etudiant.h

```
#ifndef ETUDIANT_H
#define ETUDIANT_H
#include "personne.h"
#include <iostream>
#include <string>
using namespace std;
```

```
class etudiant : public personne
{
private:
    string section;
    string niveau;
public:
    etudiant();
   string getsection();
   string getniveau();
};
#endif
```

D / fichier etudiant.cpp

```
#include "personne.h"
#include "etudiant.h"
# include <string>
# include <iostream>
using namespace  std;
etudiant::etudiant()
{
        cout << "Section?"<<endl;
        getline(cin,section);
        cout << "Niveau?"<<endl;
```

```
        getline(cin,niveau);
}
string etudiant::getsection()
{
        return section;
}
string etudiant::getniveau()
{
        return niveau;
}
```

E/ fichier application.cpp

```
#include "personne.h"
#include "etudiant.h"
#include <iostream>
#include <string>
using namespace std;
int main()
{
   personne  p  ;
   cout<<"Nom  de la personne p:" <<p.getnom()<<endl;
   cout<<"Prénom de la personne  p:"
<<p.getprenom()<<endl;
```

```
    etudiant e;
    cout<<"Nom etudiant et:" <<e.getnom()<<endl;
    cout<<"Prénom etudiant et:" <<e.getprenom()<<endl;
    cout<<"Section etudiant et:" <<e.getsection()<<endl;
    cout<<"Niveau etudiant  et:" <<e.getniveau()<<endl;

    return 0;
}
```

1-2 Héritage simple avec des constructeurs avec paramètres et un vecteur d'objets polymorphe

Dans ce cas, le constructeur de la sous-classe fait explicitement appel au constructeur de la classe de base. En outre, au lieu de créer 2 vecteurs pour stocker les personnes et les étudiants, nous créerons un seul vecteur de type personne car les étudiants sont aussi des personnes et nous procéderons à un casting (transtypage) dynamique d'une personne en étudiant lorsque la personne est un étudiant.

Exemple :

A/ fichier personne.h

```
#ifndef PERSONNE_H
#define PERSONNE_H
#include <iostream>
#include <string>
using namespace std;
```

```
class personne
{
private:
   string nom;
   string prenom;
       string nomclasse;
public:
   personne(string anom,string aprenom,string anomclasse);
   string getnom();
   string getprenom();
   string getnomclasse();
};
#endif
```

B/ fichier personne.cpp

```
#include "personne.h"
#include <iostream>
#include <string>
using namespace std;

personne::personne(string anom,string aprenom,string
anomclasse)
{
```

```
    nom=anom;
    prenom=aprenom;
    nomclasse=anomclasse;

}

string personne::getprenom()
{
    return prenom;
}

string personne::getnom()
{
    return nom;
}

string personne::getnomclasse()
{
    return nomclasse;
}
```

C/ fichier etudiant.h

```
#ifndef ETUDIANT_H
```

```
#define ETUDIANT_H
#include "personne.h"
#include <iostream>
#include <string>
using namespace std;
class etudiant : public personne
{
private:
   string section;
   string niveau;
public:
   etudiant(string anom,string aprenom,string
anomclasse,string asection,string aniveau);
  string getsection();
  string getniveau();
};
#endif
```

D/ fichier etudiant.cpp

```
#include "personne.h"
#include "etudiant.h"
# include <string>
# include <iostream>
```

```
using namespace  std;
etudiant::etudiant(string anom,string aprenom,string
anomclasse,string asection,string aniveau):
personne(anom,aprenom,anomclasse)
{
      section=asection;
      niveau=aniveau;
}

string etudiant::getsection()
{
       return section;
}
string etudiant::getniveau()
{
       return niveau;
}
```

E/ fichier application.cpp

```
#include "personne.h"
#include "etudiant.h"
#include <iostream>
#include <string>
```

```
#include <vector>

int main()
{
   vector <personne *> liste;
   etudiant * et;
   etudiant * v;
   personne * p;
   int choix=0;
   string anom,aprenom,asection,aniveau,anomclasse;
   do
   {
      cout <<"1.Ajouter une personne"<<endl;
      cout <<"2.Ajouter un etudiant"<<endl;
      cout <<"3.Lister les personnes"<<endl;
      cout <<"4.Lister les etudiants"<<endl;
      cout <<"5.Sortie"<<endl;
      cout <<"   Votre choix?"<<endl;
      cin>>choix;
      cin.get();
      switch(choix)
      {
```

```
        case 1:
            anomclasse="personne";
            cout <<"Nom personne?"<<endl;
            getline(cin,anom);
            cout <<"Prenom personne?"<<endl;
            getline(cin,aprenom);
            p=new personne(anom,aprenom,anomclasse);
            liste.push_back(p);
            break;
        case 2:
            anomclasse="etudiant";
            cout <<"Nom etudiant?"<<endl;
            getline(cin,anom);
            cout <<"Prenom etudiant?"<<endl;
            getline(cin,aprenom);
            cout <<"Section etudiant?"<<endl;
            getline(cin,asection);
            cout <<"Niveau etudiant?"<<endl;
            getline(cin,aniveau);
 et =new
etudiant(anom,aprenom,anomclasse,asection,aniveau);
            liste.push_back(et);
```

```
        break;
    case 3:
        for (int i=0;i<liste.size();i++)
        {
       if (liste[i]->getnomclasse().compare("personne")==0)
        {
         cout<<"_________________"<<endl;
        cout<<"Nom personne:"<<liste[i]->getnom()<<endl;
cout<<"Prenom personne:"<< liste[i]->getprenom()<<endl;
        }

        }
        cout<<"_________________"<<endl;
        break;
    case 4:
        for (int i=0;i<liste.size();i++)
        {
       if (liste[i]->getnomclasse().compare("etudiant")==0)
        {
          v = (etudiant *)liste[i];
          cout<<"_________________"<<endl;
          cout<<"Nom etudiant:"<<v->getnom()<<endl;
```

```
            cout<<"Prenom etudiant:"<<v->getprenom()<<endl;
              cout<<"Section etudiant:"<<v->getsection()<<endl;
              cout<<"Niveau etudiant:"<<v->getniveau()<<endl;
             }
            }
            cout<<"_________________"<<endl;
            break;
       case 5:
            break;
       default:
            cout <<"choix invalide!!!!!"<<endl;
       }
   }
   while (choix!=5);
   cout<<"Fin du programme "<<endl;
}
```

II – Classes abstraites et Classes concrètes

Une classe abstraite est définie dans le cadre de l'héritage. Une classe abstraite est d'abord toujours une classe de base pour d'autres classes mais qui n'a réellement pas d'existence dans le contexte où l'on modélise l'application.

Contrairement aux classes abstraites, les classes concrètes ont une existence dans l'environnement de modélisation.

Une classe concrète peut être une classe de base ou une sous-classe.

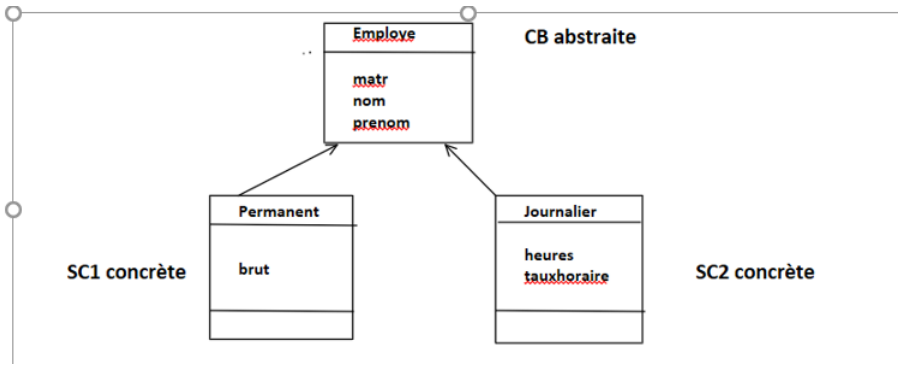
Considérons l'exemple précédent vu dans l'héritage avec les classes Personne, Etudiant et Professeur.

Un Professeur est une Personne. Un Etudiant est une Personne. Par conséquent les classes Professeur et Etudiant sont des sous- classes qui héritent de la classe Personne et sont des classes concrètes car elles ont une existence dans l'environnement. De même, la classe Personne est une classe de base concrète pour représenter les personnes qui ne sont ni des étudiants ni des professeurs.

Dans la plupart des cas, une classe de base est en général abstraite et donc n'a pas d'existence propre. Elle sert à la généralisation pour les sous-classes. En outre, il sera interdit de créer des objets d'une classe abstraite. Nous ne pourrons créer que des objets de classes concrètes.

Prenons l'exemple suivant :

On considère une entreprise dans laquelle il n'y a que 2 types d'employés : des permanents et des journaliers.



La classe Employe n'a pas d'existence dans cette entreprise car un employé est soit un permanent, soit un journalier. Par conséquent, la classe Employe est une classe de base abstraite et on ne créera pas d'objets de type Employe.

Les sous-classes Permanent et Journalier sont des sous-classes concrètes et ont une existence propre.

Pourquoi alors la classe Employe est- elle utile ?

- **Tout d'abord pour la généralisation : regroupement des attributs communs aux sous-classes Permanent et Journalier (matr, nom, prenom)**
- **La classe Employe bien qu'abstraite doit posséder un constructeur non pas pour créer des objets de type Employe mais pour la construction des objets des sous-classes Permanent et Journalier. En effet pour construire un objet d'une sous classe , il faut construire d'abord l'objet de sa classe de base .**

III – Propriétés des classes abstraites et le Polymorphisme

Il existe 2 types de polymorphisme :

- **Polymorphisme de structures**
- **Polymorphisme de méthodes**

3-1 Polymorphisme de structures

Au lieu de créer un vecteur pour chaque sous-classe concrète afin de stocker ses objets, on crée un seul vecteur qui est du type de la classe de base abstraite pour stocker les différents objets des sous-classes concrètes. En effet, ces objets des sous-classes sont aussi des objets de la classe de base. Ce vecteur est polymorphe puisqu'il stocke des objets différents

3-2 Polymorphisme de méthodes

Une méthode polymorphe est une méthode qui est présente dans toutes les classes abstraites et concrètes sous le même nom avec un corps d'instructions différent de classe à classe.

Lorsqu'on invoque une méthode polymorphe sur un objet, ce dernier est analysé et la méthode correspondante est alors exécutée.

Il existe 2 types de méthodes polymorphes :

- **<u>Les méthodes virtuelles</u> : ce sont des méthodes qui portent le même nom dans toutes les classes abstraites et concrètes qui possèdent un corps d'instructions. La méthode virtuelle à exécuter parmi les méthodes de même nom sera celle de l'objet sur lequel elle opère. Une méthode virtuelle sera déclarée avec le mot virtual au niveau des classes de base abstraites et sans le mot virtual dans les sous-classes concrètes. Elles se définissent de la manière suivante au niveau des classes abstraites :**

  <u>virtual type nom_methoder();</u>

- **<u>Les méthodes virtuelles pures encore appelées méthodes abstraites</u> : ce sont des méthodes sans corps d'instructions définies au niveau des classes de bases abstraites pour qu'elles soient rendues concrètes par les classes concrètes (obligation).Pour qu'une classe soit abstraite, elle doit posséder au moins une méthode abstraite qui se définit de la manière suivante au niveau des classes de base :**

  <u>virtual type nom_methode()=0;</u>

Exemple

A/ fichier employe.h

```
# ifndef EMPLOYE_H
# define EMPLOYE_H
# include <string>
# include <iostream>
using namespace std;
class employe
{
private:
      string matr;
      string nom;
      string prenom;
public:
      employe();
virtual   void afficher();
      virtual int salaire()=0;
};
#endif
```

B/ fichier employe.cpp

```
#include "employe.h"
#include <iostream>
#include <string>
using namespace std;

employe::employe()
{
	cout <<"matricule ?"<<endl;
	getline(cin,matr);
	cout <<"nom ?"<<endl;
	getline(cin,nom);
	cout <<"prenom ?"<<endl;
	getline(cin,prenom);

}
void employe::afficher()
{
	cout<<"matricule: "<<matr<<endl;
	cout<<"nom: "<<nom<<endl;
```

```
   cout<<"prenom: "<<prenom<<endl;
}
```

C/ fichier permanent.h

```
#ifndef PERMANENT_H
#define PERMANENT_H
#include <iostream>
#include "employe.h"
using namespace std;
class permanent:public employe
{
private:
      int brut;
      int ret;
public:
      permanent();
      void afficher();
      int salaire();
};
#endif
```

D/ fichier permanent.cpp

```
# include "employe.h"
# include "permanent.h"
#include <iostream>
using namespace std;

permanent::permanent()
{
	cout << "brut ?"<<endl;
	cin>>brut;
	cout << "retenues ?"<<endl;
	cin>>ret;
	cin.get();
}
void permanent::afficher()
{
	employe::afficher();
	cout << "brut: "<< brut<<endl;
	cout <<"retenues: "<<ret<<endl;
}
int permanent::salaire()
```

```
{
	return (brut-ret);
}
```

E/ fichier journalier.h

```
#ifndef JOURNALIER_H
#define JOURNALIER_H
#include "employe.h"
#include <iostream>
using namespace std;
class journalier:public employe
{
private:
	int nh;
	int th;
public:
	journalier();
	void afficher();
	int salaire();
};
#endif
```

F/ fichier journalier.cpp

```
#include "employe.h"
#include "journalier.h"
#include <iostream>
using namespace std;

journalier::journalier()
{
	cout << "nombre d\'heures ?"<<endl;
	cin>>nh;
	cout << "taux horaire ?"<<endl;
	cin >>th;
	cin.get();
}

void journalier::afficher()
{
	employe::afficher();
	cout <<"heures effectuees: "<<nh<<endl;
```

```
        cout << "taux par heure: "<<th<<endl;
}

int journalier::salaire()
{
        return (nh*th);
}
```

G/ fichier commissionnaire.h

```
#ifndef COMMISSIONNAIRE_H
#define COMMISSIONNAIRE_H
#include "employe.h"
#include <iostream>
using namespace std;
class commissionnaire:public employe
{
private:
        int fixe;
        int na;
        int ta;
public:
```

```
        commissionnaire();
        void afficher();
        int salaire();
};
#endif
```

H/ fichier commissionnaire.cpp

```
#include "employe.h"
#include "commissionnaire.h"
#include <iostream>
using namespace std;

commissionnaire::commissionnaire()
{
        cout << "salaire fixe ?" <<endl;
        cin>>fixe;
        cout << "nombre d\'articles vendus ?"<<endl;
        cin>>na;
        cout << "taux par article ?"<<endl;
        cin>>ta;
        cin.get();
```

```
}

void commissionnaire::afficher()
{
	employe::afficher();
	cout << "salaire fixe: "<<fixe<<endl;
	cout << "nombre aticles vendus: "<<na<<endl;
	cout << "taux par article: " << ta;
}

int commissionnaire::salaire()
{
	return(fixe+na*ta);
}
```

I/ application.cpp

```
# include "employe.h"
# include "permanent.h"
# include "journalier.h"
# include "commissionnaire.h"
#include <iostream>
```

```
#include <string>
#include <vector>
using namespace std;
int main()
{
   vector <employe *> liste;
   permanent * p;
   journalier * j;
   commissionnaire * c;

      int i,choix=0;

            do
            {
            cout << "1.ajouter un permanent" << endl;
            cout << "2.ajouter un journalier" << endl;
            cout << "3.ajouter un commissionnaire" <<
endl;
            cout << "4.Afficher employés et salaires" <<
endl;
            cout << "5.Sortie" << endl;
```

```
        cout << "   Votre choix?" << endl;
cin >>choix;
        cin.get();
        switch(choix)
        {
          case 1:

    p=new permanent;
              liste.push_back(p);
     break;
   case 2:
        j=new journalier;
        liste.push_back(j);
        break;
   case 3:
         c=new commissionnaire;
              liste.push_back(c);
        break;
   case 4:
```

```
            cout << "******calcul des salaires et
affichage des donnees des employes****" <<endl;
            for (i=0;i<liste.size();i++)
            {
            cout <<
"_____"<<endl;
              cout << "employe N°: " << i+1<<endl;
              liste[i]->afficher();
               cout<< "\n salaire: "<<liste[i]-
>salaire()<<endl;
          }
            cout <<
"_____"<<endl;
             break;
             case 5:
           break;
        default:
          cout << "choix invalide!!!!!"<<endl;


       }
```

```
        }
        while (choix!=5);
cout << "Fin du programme"<<endl;

       return 0;
}
```