LES CLASSES ABSTRAITES EN C++

1. ROLE

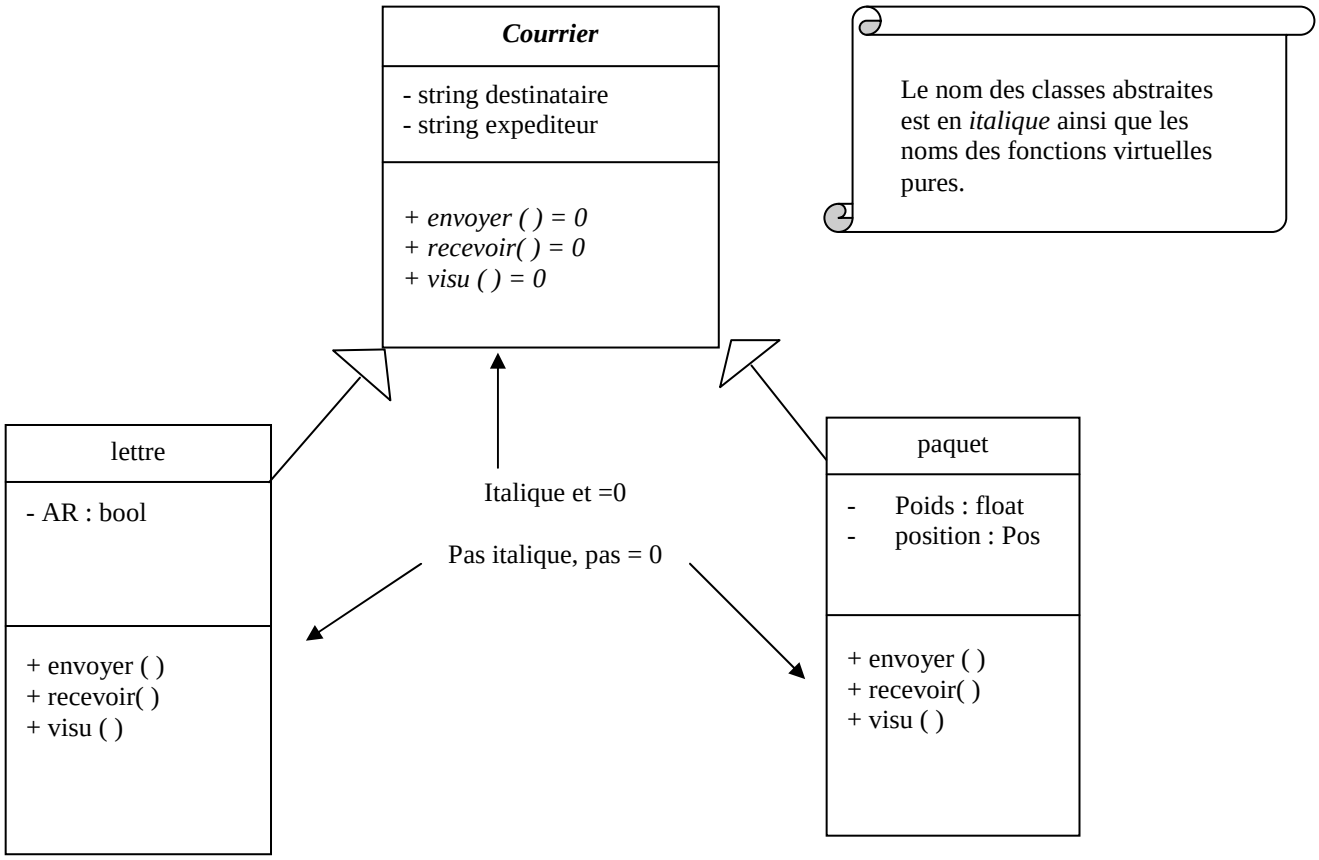
On peut être parfois amené à créer des classes destinées à donner naissance à d'autres classes et non pas à des objets. Ces classes sont appelées « **classes abstraites** ».
Leurs particularités, sont :

- Interdiction d'instancier des objets (déclaration possible de pointeurs).
- Construite avec au moins une fonction virtuelle pure.

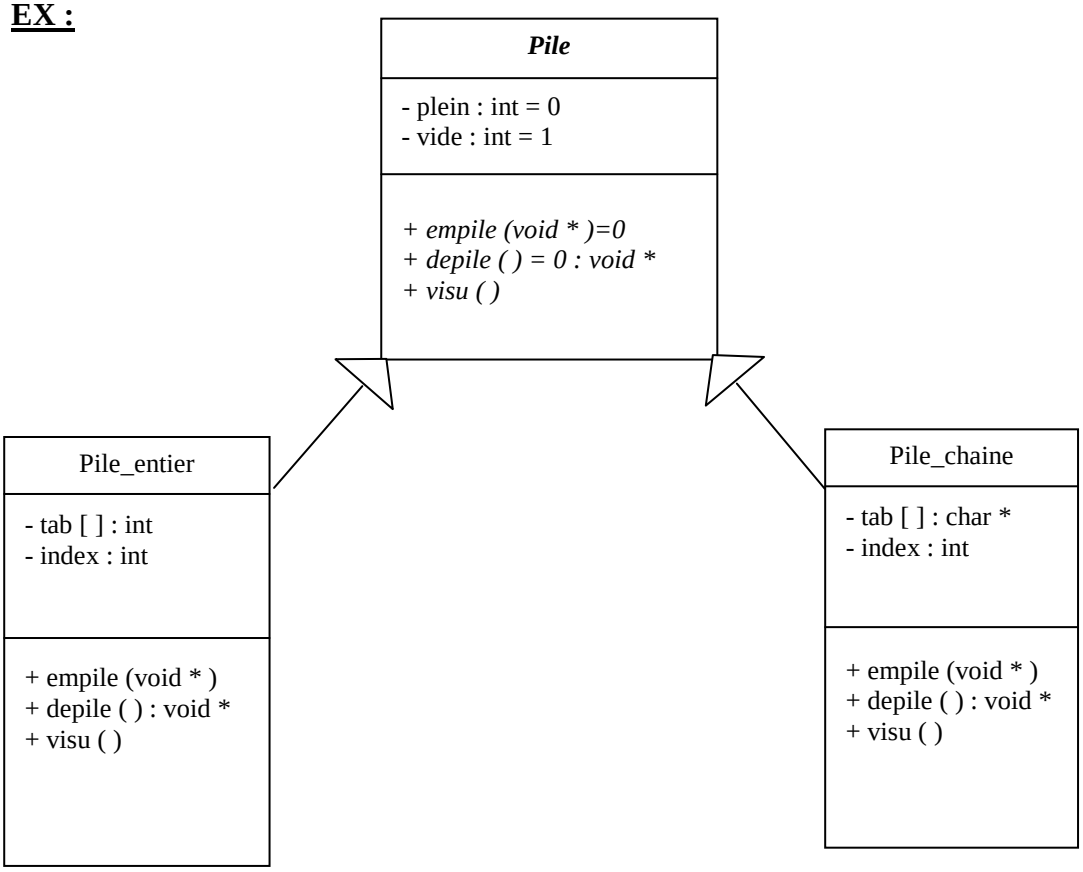
Illustration :

Courrier
- string destinataire - string expediteur
+ envoyer ( ) + recevoir( ) + visu ( )

Cette classe permet de gérer du courrier sans tenir compte de la spécificité de celui-ci : lettre, paquet, lettre avec AR… Cependant, tous ces « courriers » ont des points communs. Nous allons donc créer une classe générique regroupant les points communs. Cette classe ne servira pas à instancier des objets mais à créer d'autres classes dérivées gérant les spécificités de chaque type de courrier en respectant ce qui est imposé par la classe mère.



EX :

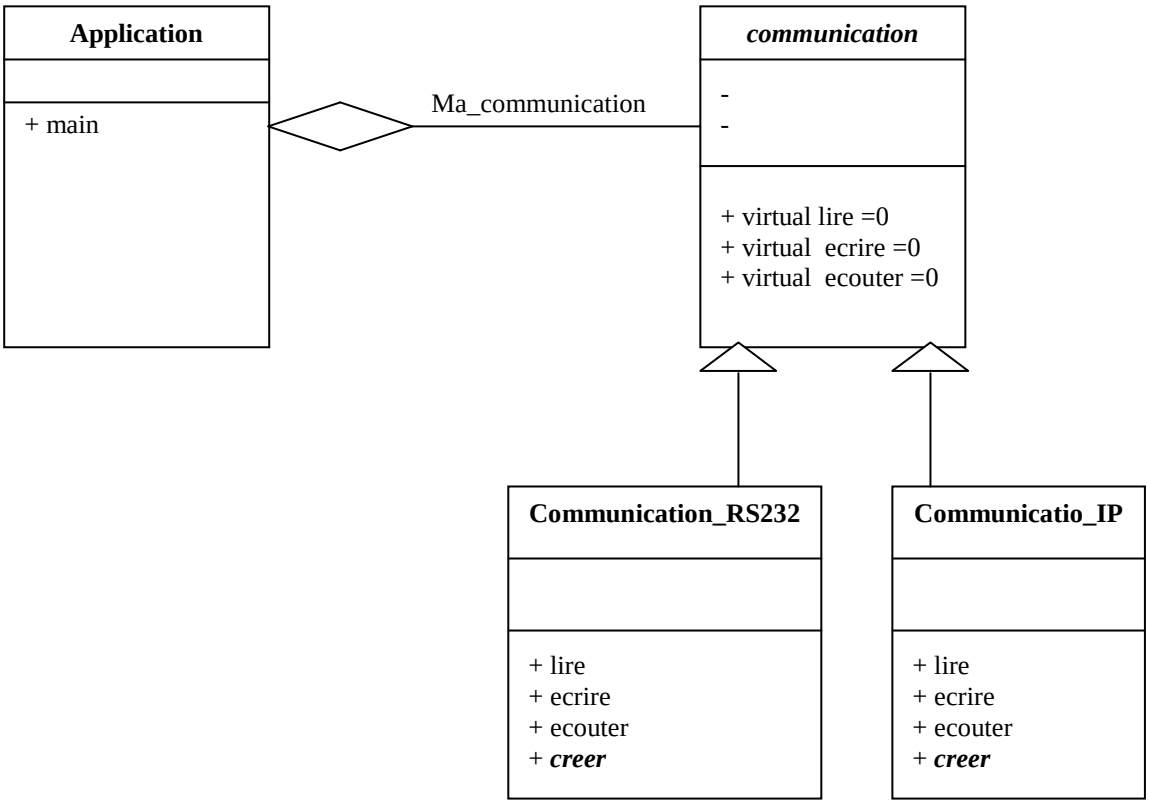


Dans ce cas, la classe « **Pile** » est abstraite et ne peut donner naissance à des objets de type « **Pile** » car on ne connaît pas le type des données à empiler. Elle doit donc simplement permettre de créer des classes dérivées « pile_entier » ou « pile_chaine » qui elles donneront naissance à des objets capables d'empiler des entiers ou des chaînes. Les fonctions déclarées virtuelles pures dans la classe Pile, doivent être obligatoirement définies dans les classes dérivées.

Autre exemple :

Modification de la nature d'un objet pendant l'exécution de l'application :

Supposons une application quelconque devant par moment communiquer certaines données à d'autres systèmes. Ces communications pouvant se faire via une liaison série ou réseau, avec un lien IP ou RPC. L'application devra être écrite le plus indépendamment possible du type de liaison. C'est pourquoi, il s'avère judicieux de réaliser une classe abstraite « **communication** » dont dériveront par exemple « **communication_RS232** » ou « **communication_IP** ». Cette classe abstraite devra imposer aux classes dérivées d'utiliser des méthodes nommées par exemple « lire », « ecrire » ou « ecouter ».
L'application manipulera des objets de types **communication** (type abstrait), eux-mêmes implémentés par (type concret) **communication_serie** ou **communication_IP.** Le programme choisit l'implémentation en cours d'exécution (ligature dynamique) et non pas à la compilation (ligature statique).
De cette manière l 'application principale fera toujours appel aux mêmes méthodes quelque soit le type réel de liaison.



Manipulation d'objets répartis ou distribués (entre plusieurs systèmes différents)

Les objets répartis sont des objets de même nature distribués sur différentes plates formes et donc implémentés de façon différente.
Ex : jeu sur Internet, l'implémentation des objets n'est pas la même sur le serveur et les postes clients).
Afin de garantir la cohérence, chaque objet doit être issu d'une classe concrète dérivée d'une classe abstraite (ou interface au sens UML) appelée contrat.
En général, le contrat est écrit dans un langage normalisé appelé IDL (Interface Definition Language).

Exemple : représentation UML d'une gestion distante d'un ascenseur.

Diagramme de déploiement :

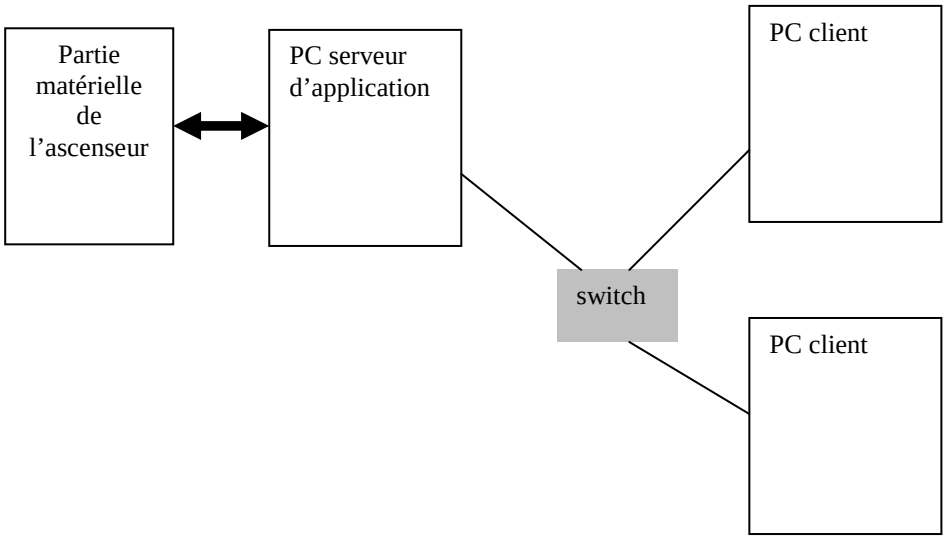
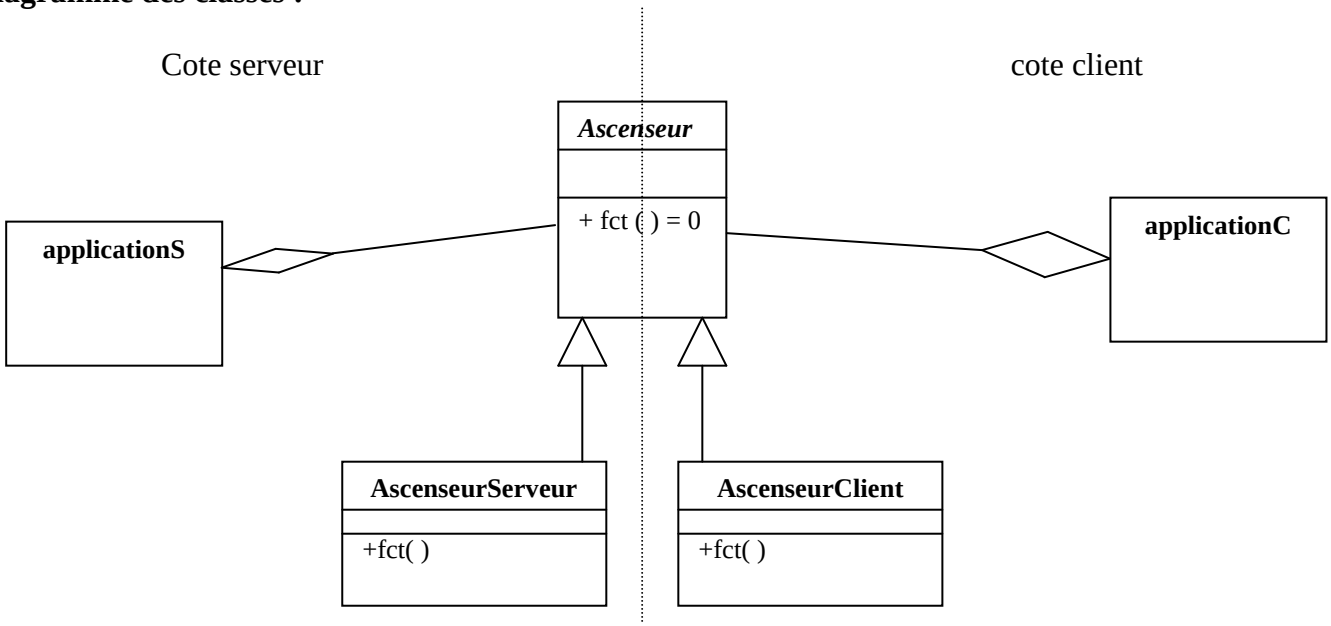


Diagramme des classes :



2. LES FONCTIONS VIRTUELLES :

Les fonctions virtuelles permettent de choisir le code exécutable d'une fonction au moment de l'appel de la fonction (ligature dynamique) et non pas à la compilation (ligature statique).
Une fonction virtuelle se déclare avec le mot clé « **virtual** ».

Illustration :

Cas de la ligature statique : pas de fonctions virtuelles.

```
class mere
{
private :
int dm;
public :
void voir ( ){cout << "\nvoir de mere";}
} ;
```

```
class fille : public mere
{
private :
int df;
public :
void voir(){cout << "\nvoir de fille";}
};
```

```
main()
{
mere *m1 = new mere;
fille f1;
m1->voir();
f1.voir();
m1 = &f1;
m1->voir();
}
```

C'est le cas de la ligature statique, le choix du code à exécuter au moment de « m1.fct1 ( ) » est défini à la compilation et ne peut plus être modifié.

voir de mere
voir de fille
voir de mere

Programme principal identique mais résultat différent !!!

Cas de la ligature dynamique : « voir » de la classe mère est déclarée virtuelle.

```
class mere
{
private :
int dm;
public :
virtual  void voir ( ){cout << "\nvoir de
mere";}
} ;
```

```
class fille : public mere
{
private :
int df;
public :
void voir(){cout << "\nvoir de fille";}
};
```

```
main()
{
mere *m1 = new mere;
fille f1;
m1->voir();
f1.voir();
m1 = &f1;
m1->voir();
}
```

C'est le cas de la ligature dynamique, le choix du code à exécuter au moment de « m1.fct1 ( ) » est défini au moment de l'appel. L'objet m1 étant devenu égal à f1, c'est la fonction fct1 de la classe fille qui sera exécutée.
voir de mere
voir de fille
voir de fille

Une fonction virtuelle pure est une fonction n'ayant pas de code associé. Pour être utilisée, elle doit être redéfinie dans les classes dérivées. Elle se déclare de la façon suivante :

virtual type nom_fct (liste des arguments) = 0 ;

3. MISE EN OEUVRE

Pour qu'une classe soit abstraite, il faut et il suffit qu'une méthode soit déclarée virtuelle pure. Dès lors, il devient impossible de créer des objets de cette classe ce qui oblige le concepteur des classes dérivées à définir toutes les fonctions virtuelles pures. Cela impose à tous les utilisateurs de cette classe abstraite, de respecter les noms de fonctions imposées (et leur signature) par la classe mère.
Si une des méthodes virtuelles pures n'est pas redéfinie dans la classe dérivée, Cette méthode doit être redéfinie virtuelle pure et la classe devient donc abstraite.
Le passage des arguments vers les constructeurs des classes abstraites est identique à l'héritage classique.
Les constructeurs ne peuvent être virtuels contrairement au destructeurs.

Déclaration d'une fonction virtuelle et virtuelle pure :

```
virtual     fct_1 ( int truc )      //fonction virtuelle
        {…//corps de la fonction
        }

virtual     fct_1 ( int truc ) = 0  //fonction virtuelle pure
```

4. EXEMPLE

Déclaration de la classe abstraite « Pile ».

Il existe une fonction virtuelle « **visu ( )** »n'ayant pas nécessairement besoin d'être redéfinie.
Il existe deux fonctions virtuelles pures «**`empile ()`** » et « **`depile ()`** » qui elles rendent la classe abstraite (interdiction d'instancier) et qui doivent être redéfinies dans les classes dérivées.

```
#define TAILLE  5
//---classe abstraite pile
class pile
{
public :
          int plein,vide;
pile()
    {
      plein =0 ; vide =1 ;
    cout << "\nconstructeur pile adr :"<<this;
    }
virtual void visu()     //fonction virtuelle
      { cout << « \nvisu virtuelle de pile »;}

//fonction virtuelle pure doivent être redéfinies
virtual int empile( void *) =0 ;
virtual void * depile() = 0;
};
//---FIN de classe pile
```

Déclaration de la classe concrète « pile_int ».

```
class pile_int  : public pile
{
private :
    int index,tab[TAILLE];
public :
//ecriture du constructeur à compléter le cas échéant
pile_int()
    {
    cout << "\nconstructeur pile_int adr :"<<this;
    index = 0;tab[index]=0;
    }
void visu(){cout << "\ndernier element : "<<tab[index-1];}

int empile(void * pt)
    {
    if (index == TAILLE){plein = 1;return(0);}
    else {vide =0;tab[index++] = *(int *)pt; }
    return(1);
    }
void * depile()
    {
    if(index == 0){vide =1;return(NULL);}
    else plein = 0;
    return(&tab[--index]);
    }
};
```

Déclaration de la classe concrète « pile_str ».

```
class pile_str  : public pile
{
private :
    char *tab[TAILLE];
    int index;
public :
//ecriture du constructeur à compléter le cas échéant
pile_str()
    {
    cout << "\nconstructeur pile_str adr :"<<this;
    index = 0;tab[index]=NULL;
    }
void visu(){cout << "\ndernier element : "<<tab[index-1];}

int empile(void * pt)
    {
    if (index == TAILLE){plein = 1;return(0);}
    else {vide =0;tab[index] =(char *)malloc(127);tab[index++] =
(char *)pt; }
    return(1);
    }
void * depile()
    {
    if(index == 0){vide =1;return(NULL);}
    else plein = 0;
    return((void *)tab[--index]);
    }
};
```

Deux exemples de programme de gestion de pile d'entier et de pile de chaînes de caractères.

```
//---programme de gestion de pile de string
int main()
{
int k;
void  *pt;
char* tt[] = {"chaine 0","chaine 1","chaine 2","chaine 3","chaine 4","chaine
5","chaine 6","c 8","c 9","c 10"}  ;
pile_str p;
char * texte = (char *)malloc(20);
for(k=0;k<7;k++)
    {
    if(p.empile(tt[k]) == 0){cout << "\npile pleine : "<<k;}
    else p.visu();
    }

for(k=0;k<7;k++)
    {
    if((pt= (void *)p.depile()) == NULL){cout << "\npile vide : "<<k;}
    else {cout << "\nvaleur depilée : "<<(char *)pt;}
    }
cin >>texte;

return 0;
}
```

```
//---programme de gestion de pile d'entirer
int main()
{
int k;
int *pt;
int tt[] = {10,20,30,40,50,60,70,80}  ;
pile_int p;
char * texte = (char *)malloc(20);
for(k=0;k<7;k++)
    {
    if(p.empile(&tt[k]) == 0){cout << "\npile pleine : "<<k;}
    else p.visu();
    }

for(k=0;k<7;k++)
    {
    if((pt= (int *)p.depile()) == NULL){cout << "\npile vide : "<<k;}
    else {cout << "\nvaleur depilée : "<<*pt;}
    }
cin >>texte;

return 0;
}
```

TD : héritage et classe abstraite.

Ecrire les classes représentées par le diagramme UML suivant :

- Les constructeurs ne sont pas représentés.
- Les prototypes des méthodes ne sont pas complets.