Chapitre3

Héritage

Plan

1- Notion d’héritage
2- Types d’héritage
3- Généralisation
4- Spécification
5- Syntaxe
6- Mode de dérivation
7- Héritage multiple
8- Héritage virtuel
9- Polymorphisme
10- Classe abstraite
11- Flots
12- Exceptions

1. Notion d'héritage

- L'héritage permet de créer des classes dérivées (Classes filles) d'autres classes (Classes mères), et qui en héritent toutes les caractéristiques, en modifier une partie et peuvent ajouter d'autres.
- Une classe B hérite d'une classe A quand on peut dire que chaque objet ou instance de type B est aussi de type A.

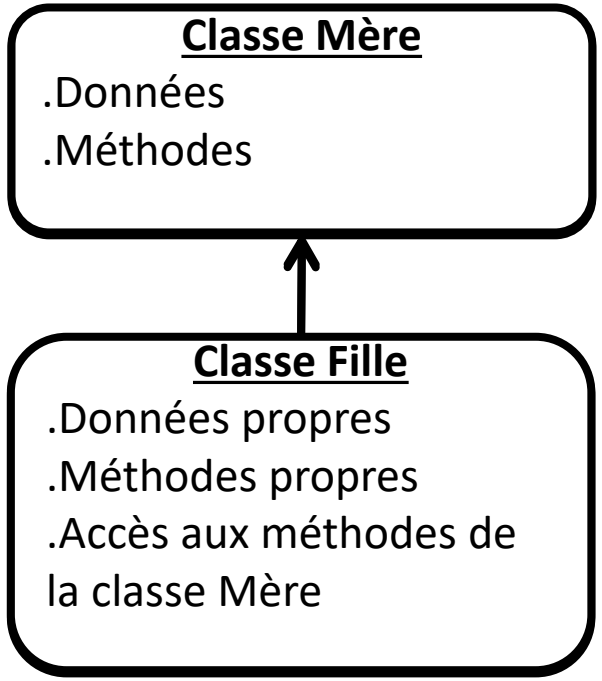
1. Notion d’héritage

Remarques:

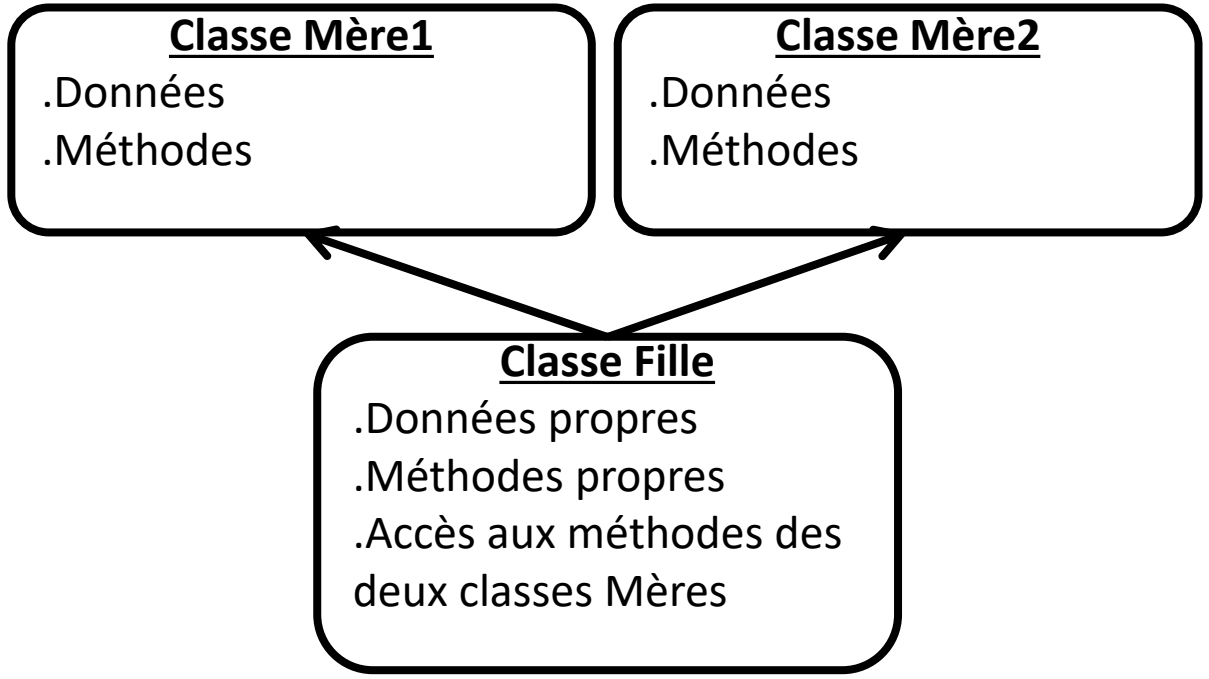
- La classe fille (ou classe dérivée ou sous classe) hérite de tous les membres, qui ne sont pas privés, de la classe de base (la super-classe ou la classe Mère) et ainsi réutiliser le code déjà écrit pour la classe de base.
- On peut aussi lui ajouter de nouveaux membres ou redéfinir des méthodes.

2. Types d'héritage

Héritage simple



Héritage multiple

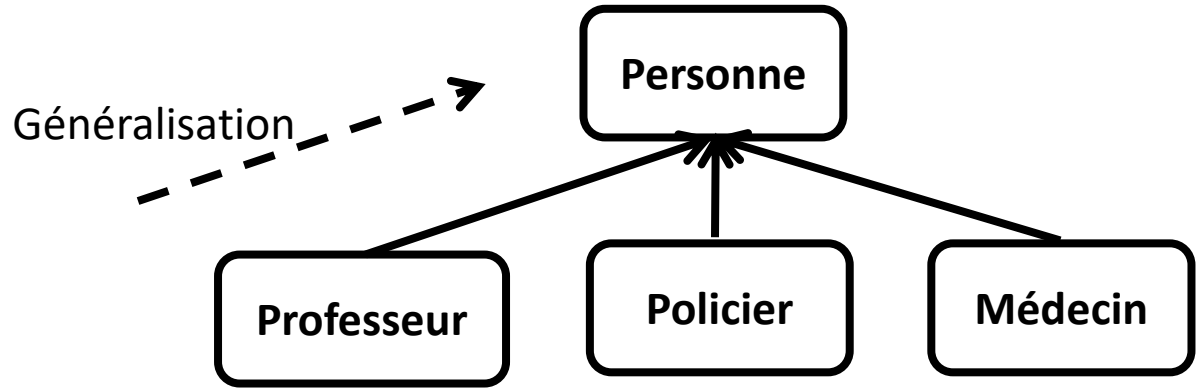


Les données privées ne sont pas accessibles par la classe fille

3. Généralisation

Généralisation:

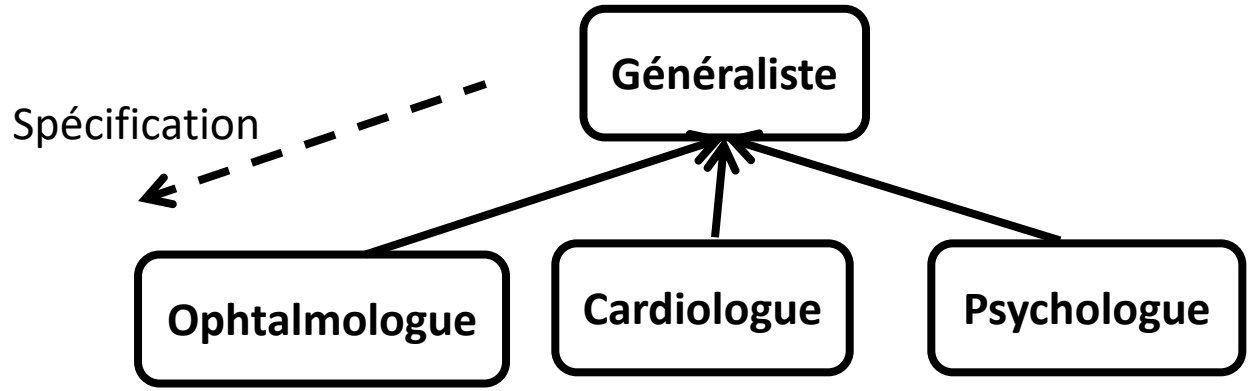
- Il s'agit de réunir des objets possédant des caractéristiques communes dans une nouvelle classe plus générale appelée super-classe (la classe mère)



4. Spécification

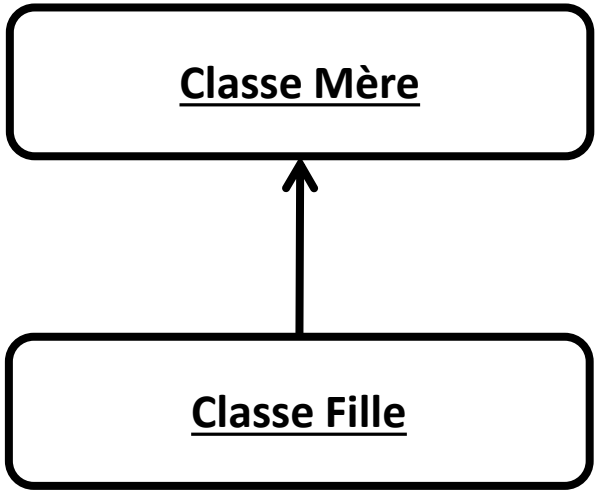
Spécification:

- Séparer des objets suivant des caractéristiques plus spécifiques dans une nouvelle classe plus spécifique appelée sous-classe (la classe fille).



5. Syntaxe

Syntaxe:




```
Class A{
//Membres    };
```

Mode de dérivation

```
Class B: public A{
//Membres ajoutés ou redéfinis    };
```

- La classe B hérite de façon public de la classe A.
- Tous les membres publics ou protégés de la classe A font partis de l'interface de la classe B.

6. Modes de dérivation

Modes de dérivation:

- Lors de la définition de la classe dérivée il est possible de spécifier le mode de dérivation par l'emploi d'un des mots-clé suivants : **public**, **protected** ou **private**.
- Ce mode de dérivation détermine quels membres de la classe de base sont accessibles dans la classe dérivée.
- Les membres privés de la classe de base ne sont jamais accessibles par les membres des classes dérivées.

6. Modes de dérivation

Héritage public:

- Il donne aux membres publics et protégés de la super classe le même statut dans la classe dérivée.

6. Modes de dérivation

Exemple:

```
Class Personne{
public:
    char nom[10];
   double poids;
private:
   char dateNais[20];
protected:
   calculAge();
public:
   saisrInfo();
…
};
```

Héritage public

```
Class Etudiant:public Personne{
public:
   int cne;
public:
   CompleterInfo(); //Ok pour saisirInfo()
   isAdulte(); // Erreur pour calculAge()
   afficheInfo(); //Ok pour nom
                  //Ok pour poids
                  //Erreur pour dateNais

…
};
```

```
Etudiant E;
E.afficheInfo(); // ok pour d'appel
```

6. Modes de dérivation

Héritage privé:

- Il donne aux membres publics et protégés de la super classe le statut de membres privés dans la classe dérivée.

6. Modes de dérivation

Exemple:

Héritage privé

```
Class Personne{
public:
    char nom[10];
   double poids;
private:
   char dateNais[20];
protected:
   calculAge();
public:
   saisrInfo();
…
};
```

```
Class Etudiant:private Personne{
public:
   int cne;
public:
   CompleterInfo(); //Ok pour saisirInfo()
   isAdulte(); // Erreur pour calculAge()
   afficheInfo(); //Ok pour nom
                  //Ok pour poids
                  //Erreur pour dateNais
…
};
```

```
Etudiant E;
E.afficheInfo(); // Erreur d'appel
```

6. Modes de dérivation

Héritage protégés:

- Il donne aux membres publics et protégés de la classe de base le statut de membres protégés dans la classe dérivée.

6. Modes de dérivation

Exemple:

```
Class A{
protected:
    int n;
…
};
```

Héritage protégé

```
Class B:protected A{
protected:
    fonct1(){ n++;}; // Ok pour n
…
};
```

```
Class C:public B{
protected:
    fonct2(){ n++;}; // Ok pour n
…
};
```

6. Modes de dérivation

Résumé:

		Statut dans la classe de base	Statut dans la classe dérivée
mode de dérivation	public	public	public
		protected	protected
		private	inaccessible
	protected	public	protected
		protected	protected
		private	inaccessible
	private	public	private
		protected	private
		private	inaccessible

Exercice1

On considère une classe Point :

Point
- int x - Int y
- Point(int,int) - Void affiche();

Réaliser une classe **PointNom**, dérivée de **Point** permettant de manipuler des points définis par leurs **coordonnées** et un **nom**. On y prévoit les méthodes suivantes:

- Un **constructeur** pour définir les **coordonnées** et le **nom** d'un objet de type **PointNom**.
- Une méthode **affiche()** pour afficher les **coordonnées** et le **nom** d'un objet de type **PointNom**.

Exercice1

Déclaration de Point

```
#include<iostream>
using namespace std;
//HeritPoint.hpp
class Point{
    public:
        int x;
        int y;
    public:
        Point(int,int);
        void affiche();
};
```

Déclaration de PointNom

```
//HeritPointCol.hpp
class PointNom:public Point{
    public:
        char nom;
    public:
        PointNom(int,int,char);
        void affiche();
};
```

Exercice1

Définition de Point

```
//Heritage.cpp
Point::Point(int a,int b)
{
    x=a;y=b;
}
void Point::affiche()
{
    cout<<" Les coordonnees sont: "<<x<<" et "<<y<<endl;
}
```

Définition de PointNom

```
PointNom::PointNom(int a,int b,char nm):Point(a,b){

    nom=nm;
}
void PointNom::affiche()
{
    cout<<"Le nom du point est: "<<nom<<endl;
    Point::affiche();
}
```

Exercice1

```
int main()
{
    cout<<"----------Tester la classe Point----------------"<<endl;
    Point test1(3,4);
    test1.affiche();
    cout<<"----------Tester la classe PointNom----------------"<<endl;
    PointNom test(5,7,'A');
    test.affiche();
}
```

Instanciation de Point

Instanciation de PointNom

```
----------Tester la classe Point----------------
 Les coordonnees sont: 3 et 4
----------Tester la classe PointNom----------------
Le nom du point est: A
 Les coordonnees sont: 5 et 7
```

7. Héritage multiple

- En langage C++, il est possible d'utiliser l'héritage multiple.
- Il permet de créer des classes dérivées à partir de plusieurs classes de base.
- Pour chaque classe de base, on peut définir le mode d'héritage.

7. Héritage multiple

Exemple:

```
Class A{
public:
    void fa();
...
};
```

```
Class B{
public:
    void fb();
...
};
```

```
Class C:public B,public A{
public:
    void fc();
...
};
```

Héritage multiple

```
void C::fc(){
Int i;
   fa();
...
}
```

Définition de fc()

7. Héritage multiple

- Dans l'héritage multiple, les constructeurs sont appelés dans l'ordre de déclaration de l'héritage.

```
Class A{
public:
    A(int n=0);
…
};
```

```
Class B{
public:
    B(int n=0);
…
};
```

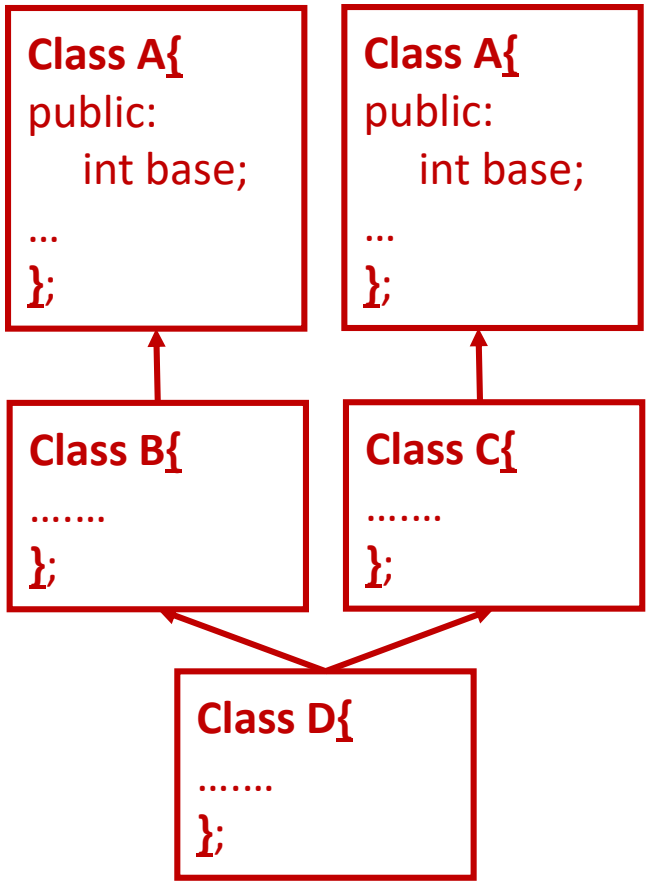
```
Class C:public B,public A{
public:
    C(int i,int j);
…
};
```

```
C::C(int i,int j):B(i),A(j){
….
….
…
}
```

Ordre d'appel des constructeurs: B(), A() et C()

Ordre d'appel des deconstructeurs: ~C(), ~A() et ~B()

8. Héritage virtuel



- Dans la classe D, il y aura deux fois le contenu hérité de la classe A.
- L'accès aux membres de la classe A se fait en levant l'ambiguïté:

```
Int main(){
D d;
d.base=0:      // Erreur d'ambiguïté
d.B::base=1; //ok
d.C::base=2; //ok
Return 0;
};
```

8. Héritage virtuel

- L'héritage virtuel permet d'avoir uniquement une seule fois le contenu hérité de la classe de base.

```
Class A{
public:
    int base;
…
};
```

```
Class B:virtual public A{
…
};
```

```
Class C:virtual public A{
…
};
```

```
Int main(){
D d;
d.base=0:      // ok
….
Return 0;
};
```

- Les classe B et C héritent virtuellement de la classe de base A.
- Le mot clé **virtual** précise au compilateur les classes à ne pas dupliquer.

9. Polymorphisme

- L'héritage nous permet de réutiliser le code écrit de la classe de base dans les sous classes.
- En C++, le **polymorphisme** est mis en œuvre par l'utilisation des fonctions virtuelles.
- Une **fonction virtuelle** est une fonction membre déclarée dans une classe de base et redéfinie par une classe dérivée. Elle utilise le mot clé **virtual**.

9. Polymorphisme

Exemple:

```
class SuperClasse{
    public:
        int a;
    public:
        SuperClasse();
        SuperClasse(int);
        void affiche();
};
```

```
class SousClasse1:public SuperClasse{
    public:
        int b;
    public:
        SousClasse1();
        SousClasse1(int,int);
        void affiche();
};
```

```
class SousClasse2:public SuperClasse{
    public:
        int b;
    public:
        SousClasse2();
        SousClasse2(int,int);
        void affiche();
};
```

9. Polymorphisme

```
SuperClasse::SuperClasse(){}
SuperClasse::SuperClasse(int e){
    a=e;
}
void SuperClasse::affiche(){
    cout<<" SuperClasse  a : "<<a<<endl;
}
```

```
SousClasse1::SousClasse1(){}
SousClasse1::SousClasse1(int e,int f):SuperClasse(e){
    b=f;
}
void SousClasse1::affiche(){
    SuperClasse::affiche();
    cout<<" SousClasse1  b : "<<b<<endl;
}
```

```
SousClasse2::SousClasse2(){}
SousClasse2::SousClasse2(int e,int f):SuperClasse(e){
    b=f;
}
void SousClasse2::affiche(){
    SuperClasse::affiche();
    cout<<" SousClasse2  b : "<<b<<endl;
}
```

9. Polymorphisme

```
int main(){
    SuperClasse *sup;
    SousClasse1 *sous=new SousClasse1(1,2);
    sup=sous;
    sup->affiche();
    SousClasse2 *deriv=new SousClasse2(3,4);
    sup=deriv;
    sup->affiche();
    delete sup;
    delete sous;
    delete deriv;
    return 0;
}
```

Appel affiche() de la super classe

```
SuperClasse   a : 1
SuperClasse   a : 3

--------------------------
```

9. Polymorphisme

Ajout de virtual à affiche()

```
class SuperClasse{
    public:
        int a;
    public:
        SuperClasse();
        SuperClasse(int);
        virtual void affiche();
};
```

```
int main(){
    SuperClasse *sup;
    SousClasse1 *sous=new SousClasse1(1,2);
    sup=sous;
    sup->affiche();
    SousClasse2 *deriv=new SousClasse2(3,4);
    sup=deriv;
    sup->affiche();
    delete sup;
    delete sous;
    delete deriv;
    return 0;
}
```

Appel affiche() des sous classes

```
SuperClasse   a : 1
SousClasse1   b : 2
SuperClasse   a : 3
SousClasse2   b : 4


--------------------
```

Exercice2

Soit les classes suivantes:

une classe **Personne** qui comporte trois champs privés, nom, prénom et date de naissance. Cette classe comporte un constructeur pour permettre d'initialiser les données. Elle comporte également une méthode polymorphe Afficher pour afficher les données de chaque personne.

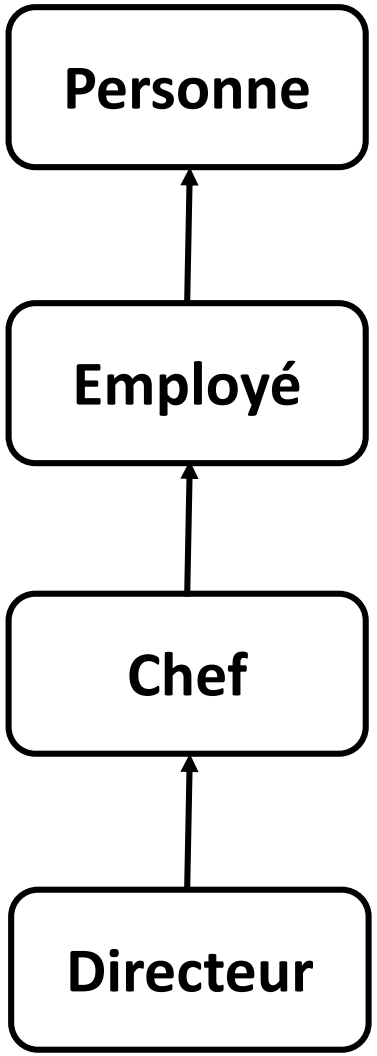
Une classe **Employé** qui dérive de la classe Personne, avec en plus un champ *Salaire* accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

Une classe **Chef** qui dérive de la classe Employé, avec en plus un champ *Service* accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

Une classe **Directeur** qui dérive de la classe Chef, avec en plus un champ *Société* accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

Exercice2

1.Ecrire les classes Personne, Employé, Chef et Directeur.
2.Créer un programme de test qui comporte tableau de huit personnes avec cinq employés, deux chefs et un directeur (8 références de la classe Personne dans lesquelles ranger 5 instances de la classe Employé, 2 de la classe Chef et 1 de la classe Directeur).



10. Classe abstraite

- Une classe est dite **abstraite** si elle contient au moins une **méthode virtuelle pure**.
- Une **méthode virtuelle pure** se déclare en ajoutant un **= 0** à la fin de sa déclaration.

```
class NomClasse{
…
virtual typeRetour fonction()=0;
…
};
```

10. Classe abstraite

- Une **classe abstraite** ne peut instancier aucun objet.
- Une **classe dérivée** qui ne redéfinit pas une **méthode virtuelle pure** est elle aussi **abstraite**.

Exercice3

Soit la classe suivante :

```
class  Forme {
        virtual float  perim()=0;
        virtual float  surf()=0;
        void affiche();
}
```

- Donner le code des classes **Disque** et **Rectangle** qui étendent de la classe **Forme**.
- Ecrire main() pour tester les classes crées.

Exercice3

```
class Forme{
    public:
        Forme();
        virtual float  perim()=0;
        virtual float  surf()=0;
        void affiche();
};
```

```
class Disque:public Forme{
    public:
        float rayon;
        float const pi=3.14;
    public:
        Disque();
        Disque(float);
        float  perim();
        float  surf();
        void affiche();
};
```

```
class Rectangle:public Forme{
    public:
        float longueur;
        float largeur;
    public:
        Rectangle();
        Rectangle(float,float);
        float  perim();
        float  surf();
        void affiche();
};
```

Exercice3

```
Forme::Forme(){}
void Forme::affiche(){
    cout<<" Les informations concernant la forme sont: "<<endl;
}
```

Exercice3

```
Disque::Disque(){}
Disque::Disque(float r){
    rayon=r;
}
float Disque::perim(){
    float res;
    res=2*pi*rayon;
    return res;
}
float Disque::surf(){
    float res;
    res=pi*rayon*rayon;
    return res;
}
void Disque::affiche(){
    float p,s;
    p=perim();
    s=surf();
    Forme::affiche();
    cout<<"Disque: Rayon = "<<rayon;
    cout<<" --> Perimetre = "<<p<<" et Surface = "<<s<<endl;
}
```

Exercice3

```
Rectangle::Rectangle(){}
Rectangle::Rectangle(float a,float b){
    longueur=a; largeur=b;
}
float Rectangle::perim(){
    float res;
    res=2*(longueur+largeur);
    return res;
}
float Rectangle::surf(){
    float res;
    res=longueur*largeur;
    return res;
}
void Rectangle::affiche(){
    float p,s;
    p=perim();
    s=surf();
    Forme::affiche();
    cout<<"Rectangle: Longueur = "<<longueur<<" et Largeur = "<<largeur;
    cout<<" -->Perimetre = "<<p<<" et Surface = "<<s<<endl;
}
```

Exercice3

```
int main(){
    Disque d(5);
    d.affiche();
    Rectangle r(4,3);
    r.affiche();
    return 0;
}
```

```
 Les informations concernant la forme sont:
Disque: Rayon = 5 --> Perimetre = 31.4 et Surface = 78.5
 Les informations concernant la forme sont:
Rectangle: Longueur = 4 et Largeur = 3 -->Perimetre = 14 et Surface = 12
```

11. Flots

- Un flux (ou **stream**) est une représentation d'un flot de données entre une source produisant de l'information et une cible consommant cette information.
- Il existe 3 flots prédéfinis pour la sortie: **cout**, **cerr** et **clog.**

11. Flots

- **cout :** est le plus commun .Il permet d'écrire sur la sortie standard. Celle-ci désigne le plus souvent l'écran de la machine sur laquelle le code est exécuté.
- **cerr :** est la sortie d'erreur. Par défaut, elle désigne également l'écran de la machine. Il s'agit ce qu'on appelle une écriture non-bufferisée.
- **clog:** Elle désigne la sortie d'erreur en bufferisant les sorties.

11. Flots

```cpp
#include<iostream>
using namespace std;
int main()
{
    cout<<"La sortie standard "<<endl;
    cerr<<"La sortie erreur non bufferisee "<<endl;
    clog<<"La sortie erreur buffersisee "<<endl;
    return 0;
}
```

```
La sortie standard
La sortie erreur non bufferisee
La sortie erreur buffersisee

--------------------------------
```

11. Flots

- **Le flux d'entrée/sortie est représenté par le mot clé « iostream » qui est un diminutif de « InputOutputSTREAM »**

11. Flots

Flux d’entrée « istream »:

- Pour récupérer des informations, **cin** et **>>** sont utilisés souvent.
- Il existe d’autres fonctions pour la saisie des données.

11. Flots

- **int get() :**

Retourne la valeur du caractère lu, sinon EOF si la fin du fichier est atteinte.

```
#include<iostream>
using namespace std;
int main(){
char c;
while ((c=cin.get()) != EOF)
cout << c;
cout<<"Apres la boucle while"<<endl;
return 0;
}
```

- Le programme lit indéfiniment les caractères à partir du clavier
- On arrête la lecture si on tape la touche retour chariot (return) du clavier
- Après l'affichage, le programme se remet en état d'attente de lecture
- L'utilisation de « CTRL-Z » arrêtera la saisie
- L'utilisation de « CTRL-C » arrêtera complètement le programme

11. Flots

- **Istream& get(char&c) :**

Elle extrait le premier caractère du flux, même si c'est un espace et elle le place dans le caractère c

```
#include <iostream>
using namespace std;
int main(){
char c;
while (cin.get(c))
cout << c;
return 0;
}
```

11. Flots

- **istream& get (char* str, streamsize count):**

Elle extrait « count-1 » caractères du flux et elle place le résultat à l'adresse pointée par la variable « ch ». La lecture s'arrête à la rencontre de « \n » ou la fin du fichier.

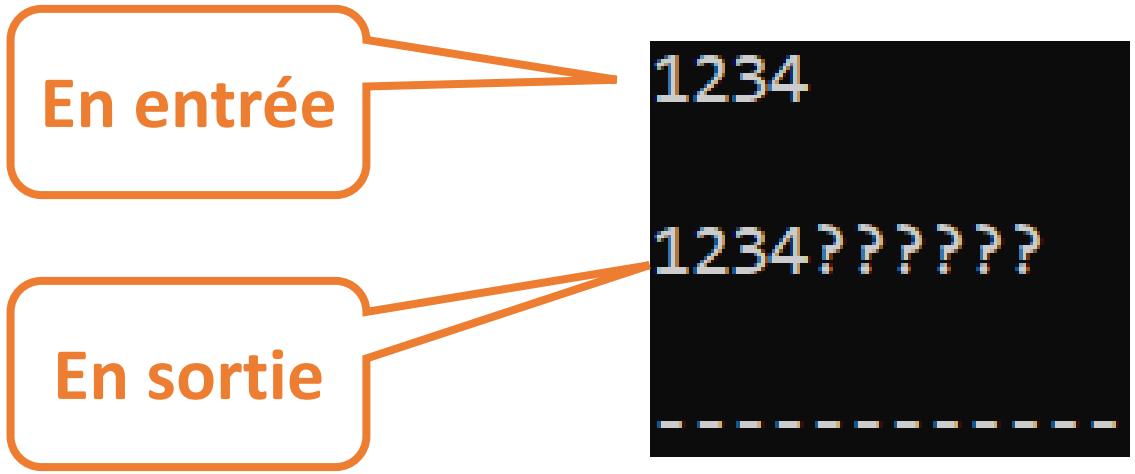
```
#include <iostream>
using namespace std;
int main(){
char c[80];
cin.get(c,79);
cout << c << endl;
return 0;
}
```

11. Flots

- **istream& read (char* str, streamsize count):**

Cette fonction extrait un bloc de taille « count » et elle le place dans la chaîne « str »

```
#include <iostream>
using namespace std;
int main(){
char c[] = "??????????";
// on ne va lire que 4 valeurs.
cin.read(c,4);
cout << endl << c << endl;
return 0;
}
```



11. Flots

Flux de sortie « ostream »:

- Pour récupérer des informations, **cout** et **<<** sont utilisés souvent.
- Il existe d'autres fonctions pour l'écriture des données.

11. Flots

- **ostream& put(char c) :**

Cette fonction écrit l'argument c dans le flux sélectionné en sortie.

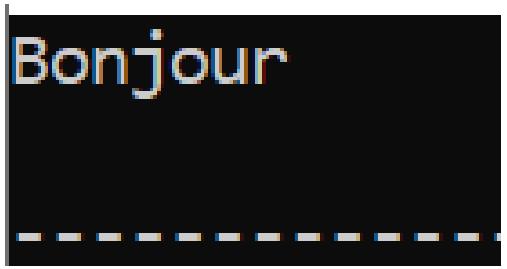
```
#include <iostream>
using namespace std;
int main(){
char c = 'a';
cout.put(c) << endl;
return 0;
}
```

11. Flots

- **ostream& write(const char* s, streamsize count):**

Cette fonction écrit « count » caractères dans le flux de sortie.

```
#include <iostream>
using namespace std;
int main(){
const char *c = "Bonjour tout le monde!";
cout.write(c,7) << endl;
return 0;
}
```



11. Flots

Formatages:

- L’écriture des données en respectant le format spécifié.
- Il faut inclure « **iomanip** ».

11. Flots

- Pour les entiers tout d'abord, on peut convertir l'affichage en différentes bases.
- Pour les booléens, on peut afficher ou non la valeur entière du booléen ou sa valeur 'true/false' .

```
#include <iostream>
#include<iomanip>
using namespace std;
int main(){
int i=42;
cout<<"Entier "<<endl;
cout<<"Deci : "<<i<<endl;
cout<<"Hexa : "<<hex<<i<<endl;
cout<<"Oct : "<<oct<<i<<endl;
cout<<"Bool : "<<endl;
cout<<true<<endl;
cout<<boolalpha<<true<<endl;
return 0;
}
```

```
Entier
Deci : 42
Hexa : 2a
Oct : 52
Bool :
1
true
```

11. Flots

- Une autre possibilité offerte et celle de la taille des gabarits.
- On va utiliser **setw(int)** avec un entier en paramètres qui va définir la taille du gabarit d'affichage.

```
#include <iostream>
#include<iomanip>
using namespace std;
int main(){
for(int i=0;i<=10;i++)
cout<<setw(2)<<i<<":"<<setw(i)<<2021<<endl;
return 0;
}
```

```
 0:2021
 1:2021
 2:2021
 3:2021
 4:2021
 5: 2021
 6:  2021
 7:   2021
 8:    2021
 9:     2021
10:      2021
```

11. Flots

- Pour les nombres à virgules flottantes, on peut déterminer la précision et l'écriture scientifique ou non;

```
#include <iostream>
#include<iomanip>
using namespace std;
int main(){
double d=42.1234567;
cout<<setprecision(4)<<d<<endl;
cout<<scientific<<d<<endl;
cout<<setprecision(8)<<fixed<<d<<endl;
return 0;
}
```

```
42.12
4.2123e+001
42.12345670
```

11. Flots

Fichiers:

- Un fichier est une séquence d'informations que l'on peut manipuler par l'intermédiaire d'une tête de lecture/écriture.
- Les types d'accès à un fichier: **Lecture seule**, **Ecriture seule** ou bien **Lecture/Ecriture**.
- Pour manipuler les fichiers en C++, on peut utiliser: **ifstream** et **ofstream** (Notion d'objet).

11. Flots

Fichiers textes:

- Un **fichier texte** est un fichier qui contient des informations sous la forme de caractères (en général en utilisant le code ASCII).
- Manipulation des fichiers (en lecture ou en écriture) s'effectue en **plusieurs étapes**.

11. Flots

Fichiers textes: (Ecriture seulement)

1. Déclaration d'une variable de type fichier:

```
#include <iostream>
#include <fstream>
using namespace std;
ofstream f; // o pour output
```

Manipuler les fichiers

Déclaration de f

11. Flots

2. Ouverture du fichier:

```
f.open("essai.txt");
```

Ouverture de essai.txt

Si le fichier n'existe pas sur le disque, il est créé, si il existe, son contenu est effacé !

11. Flots

3. Tester l'ouverture:

Problème d'ouverture

```
if (!f.good())
{
cerr << "Impossible d'écrire dans le fichier !\n";
exit(1); // arrêt du programme avec un code d'erreur différent de 0
}
```

11. Flots

4. Ecriture de données:

```
for(int i=0;i<5;i++)
f << i << "\n";
```

Ecriture du contenu de i dans le fichier

11. Flots

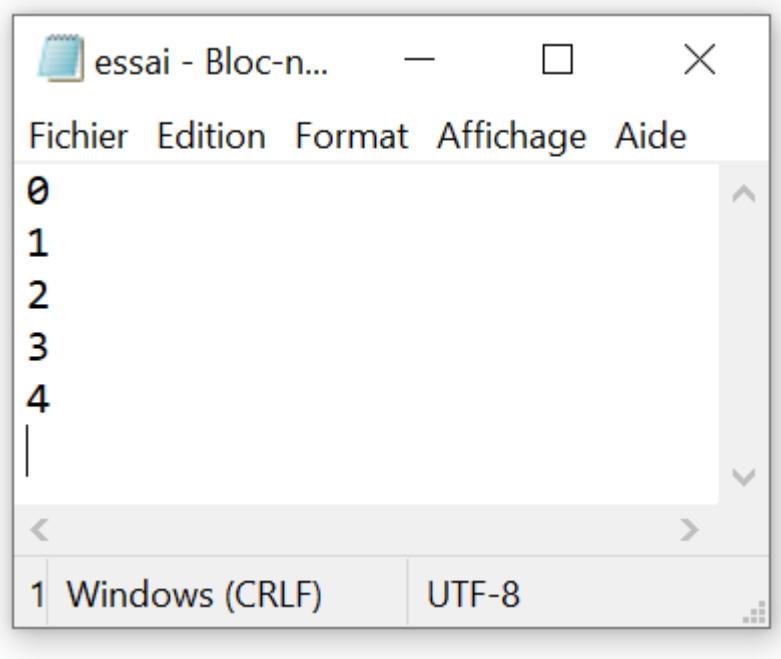
5. Fermeture du fichier:

```
f.close();
```

Fermeture du ficher. Cette étape est réalisée automatiquement par C++ dans le cas d'oublie.

11. Flots

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{

ofstream f; // 1

f.open("essai.txt"); // 2

if (!f.good())   //3
{
cerr << "Impossible d'écrire dans le fichier !\n";
exit(1);
}

for(int i=0;i<5;i++)    //4
f << i << "\n";

f.close();     //5

return 0;
}
```



11. Flots

Fichiers textes: (Lecture seulement)

1. Déclaration d'une variable de type fichier:

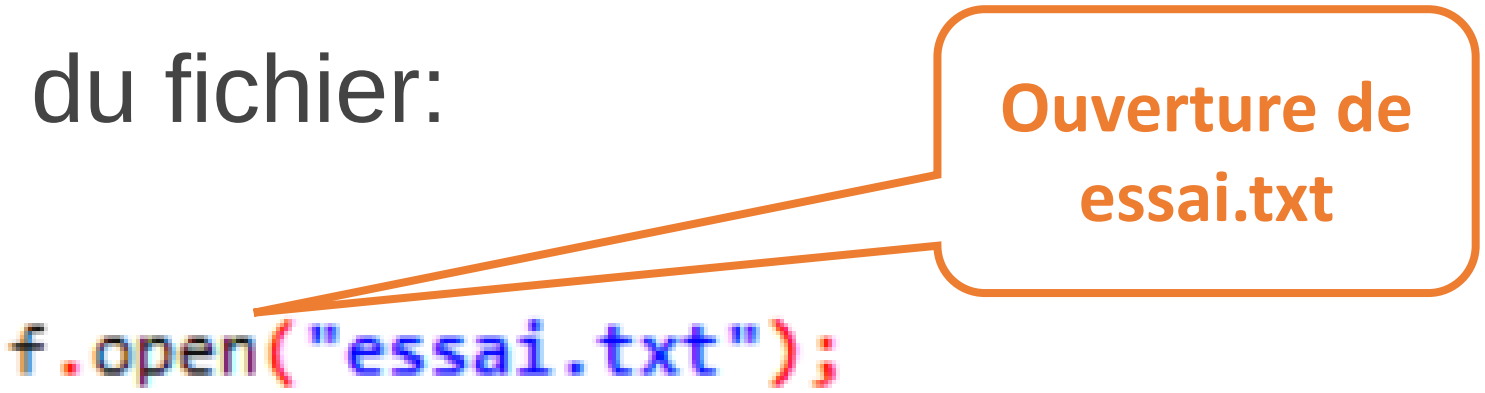
```
#include <iostream>
#include <fstream>
using namespace std;
ifstream f; // i pour input
```

Manipuler les fichiers

Déclaration de f

11. Flots

2. Ouverture du fichier:



11. Flots

3. Tester l'ouverture:

Problème d'ouverture

```
if (!f.good())
{
cerr << "Impossible de lire dans le fichier !\n";
exit(1); // arrêt du programme avec un code d'erreur différent de 0
}
```

11. Flots

4. Lecture de données:

Lecture du contenu du fichier

```
int i;
do
{
f >> i;
if (!f.eof())
cout << i << " ";
}
while (!f.eof());
```

11. Flots

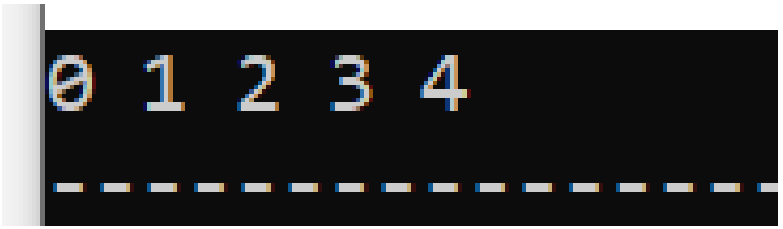
5. Fermeture du fichier:

```
f.close();
```

Fermeture du ficher. Cette étape est réalisée automatiquement par C++ dans le cas d'oublie.

11. Flots

```
#include <iostream>
#include <fstream>
using namespace std;
int main(){

ifstream f; //1

f.open("essai.txt"); //2

if (!f.good())    //3
{
cerr << "Impossible de lire dans le fichier !\n";
exit(1);
}
int i;
do {          //4
    f >> i;
    if (!f.eof())
    cout << i << " ";
   }while (!f.eof());

f.close();          //5

return 0;
}
```



11. Flots

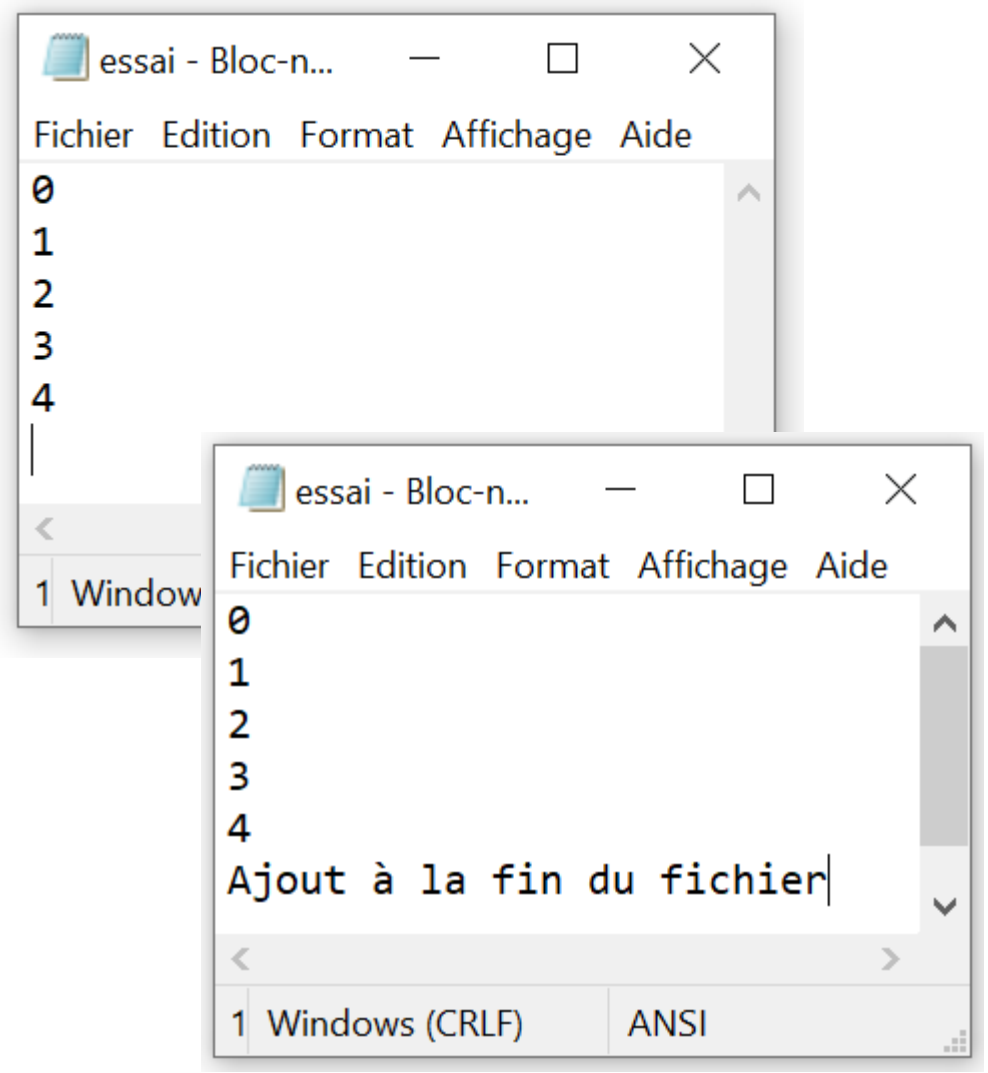
Mode d'ouvertures:

Mode d'ouverture	Signification
ios::app	Ajout en fin de fichier (append)
ios::ate	Ouverture et positionnement à la fin du fichier (at end)
ios::binary	Fichier binaire et non pas texte
ios::in	Ouverture en lecture
ios::out	Ouverture en écriture
ios::trunc	Vider le fichier lors de l'ouverture

11. Flots

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{

ofstream f; // 1

f.open("essai.txt",ios::app); // 2

if (!f.good())   //3
{
cerr << "Impossible d'écrire dans le fichier !\n";
exit(1);
}

f << "Ajout à la fin du fichier" << "\n";  //4


f.close();     //5

return 0;
}
```

Ajout à la fin du fichier



11. Flots

Fichiers textes: (Lecture/Ecriture des types de base)

- Toute variable ou constante des types de base (bool, char, int, float, double, string,...) peut être écrite ou lue.
- Lorsqu'une variable est lue, le programme commence par sauter tous les caractères assimilables à un espace (espace, tabulation, fin de ligne,...) puis lit tous les caractères qui peuvent constituer une représentation ascii du type lu.

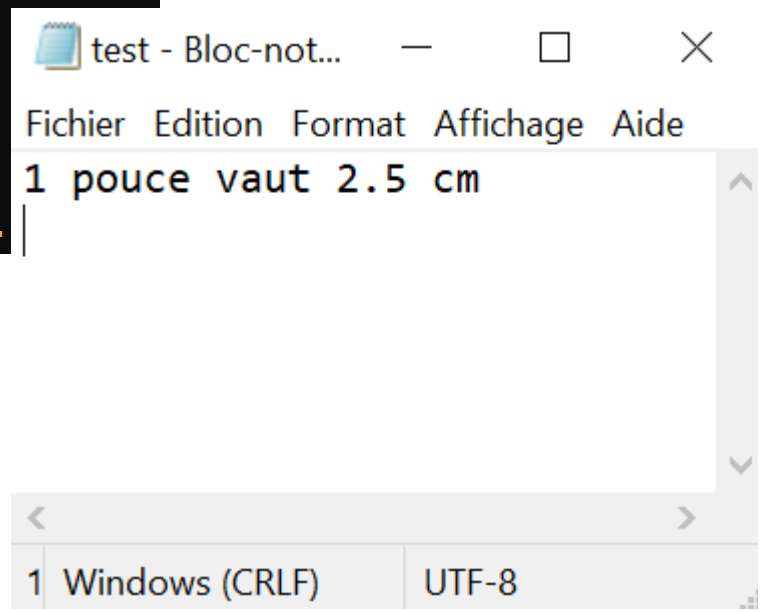
11. Flots

- La lecture s’arrête au premier espace ou au premier caractère qui ne peut pas faire partie de la représentation ascii du type lu (ce sera alors le prochain caractère à lire).
- Pour une string, la lecture s’arrête au premier espace.

11. Flots

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main(){
int i;
string s;
float r;
ifstream f;                    //1
f.open("test.txt");            //2
if (!f.good())                 //3
{
cerr << "Impossible de lire dans le fichier !\n";
exit(1);
}
f>>i;                          //4
cout<<" i = "<<i<<endl;
f>>s;
cout<<" s = "<<s<<endl;
f>>s;
cout<<" s = "<<s<<endl;
f>>r;
cout<<" r = "<<r<<endl;
f>>s;
cout<<" s = "<<s<<endl;
f.close();                     //5
return 0;
}
```

```
i = 1
s = pouce
s = vaut
r = 2.5
s = cm
------------
```



11. Flots

Fichiers textes: (Lecture/Ecriture des types définis)

- Lorsque le programmeur définit ses propres types (structures ou tableaux), il a la possibilité de définir comment ils doivent être écrits à l'écran ou dans un fichier et lus à partir du clavier ou d'un fichier
- Cette possibilité peut être réalisée via la définition des fonctions.

11. Flots

- Pour l'écriture:

```
ostream &operator <<(ostream &s,MonType &v)
{
  // écriture du paramètre v dans le fichier s
  ...
  return s; // return indispensable
}
```

11. Flots

- Pour la lecture:

```
istream &operator >>(istream &s,MonType &v)
{
  // lecture du paramètre v à partir du fichier s
  ...
  return s; // return indispensable
}
```

Exercice4

Rédiger un programme en c++ qui permet de manipuler un fichier en assurant les deux opérations suivantes: Ecriture et Lecture.

L'exécution du programme suit les étapes citées ci-dessous:

1- Saisir un nombre complexe

2- Ecrire le nombre complexe saisi dans un fichier

3- Lire le nombre complexe écrit à partir du fichier en affichant le résultat sur l'écran

Exercice4

```
#include <iostream>
#include <fstream>
using namespace std;
typedef struct
        { float re,im;
        } complexe;
ostream &operator <<(ostream &s, complexe &c)
{
    s << c.re;
    if (c.im>=0)
        s << "+";
    s << c.im << "*i";
    return s;
}
```

Exercice4

```
istream &operator >>(istream &s, complexe &c)
{    char tmp;
     bool error=false;
     s >> c.re;
     s >> c.im;
     s >> tmp;
     if (tmp!='*')
     error=true;
     else
     {    s >> tmp;
          if (tmp!='i')
               error=true;
     }
     if (error)
     {    cerr << "Le complexe n'a pas été écrit dans le bon format !\n";
          exit(1);
     }
     return s;
}
```

Exercice4

```
int main()
{
    complexe c;
    ofstream ecr;
    ifstream lect;
    cout << "Tapez un complexe sous la forme 1+6*i :";
    cin >> c;
    cout << "Vous avez tapé " << c << "\n";
    cout << "Ecriture du complexe dans le fichier\n";
    ecr.open("fic.txt");
    if (!ecr.good())
    {   cerr << "Impossible de créer le fichier !\n";
        exit(1);
    }
    ecr << c;
    ecr.close();
    cout << "Relecture du complexe à partir du fichier\n";
    lect.open("fic.txt");
    if (!lect.good())
    {   cerr << "Impossible de créer le fichier !\n";
        exit(1);
    }
    lect >> c;
    lect.close();
    cout << "J'ai relu " << c << "\n";
    return 0;
}
```

```
Tapez un complexe sous la forme 1+6*i :3+8*i
Vous avez tapÚ 3+8*i
Ecriture du complexe dans le fichier
Relecture du complexe Ó partir du fichier
JÆai relu 3+8*i


--------------------------------
```

11. Flots

Fichiers binaires:

- Un **fichier binaire** est un fichier qui contient directement la représentation mémoire des informations
- La manipulation d'un fichier binaire se diffèrent d'un fichier texte dans les étapes: 2 (Désignation du fichier) et 4 et (écriture ou lecture de données).
- On n'utilise plus **<<** et **>>** mais **f.write(...)** et **f.read(...)**.

11. Flots

Fichiers binaires: (Ecriture seulement)

1. Déclaration d'une variable de type fichier:

```
#include <iostream>
#include <fstream>
using namespace std;
ofstream f; // o pour output
```

Manipuler les fichiers

Déclaration de f

11. Flots

2. Ouverture du fichier:

```
f.open("testb.txt",ios::out|ios::binary|ios::trunc);
```

Ouverture de testb.txt

Si le fichier n'existe pas sur le disque, il est créé, si il existe, son contenu est effacé !

Fichier binaire

11. Flots

3. Tester l'ouverture:

Problème d'ouverture

```
if (!f.good())
{
cerr << "Impossible d'écrire dans le fichier !\n";
exit(1); // arrêt du programme avec un code d'erreur différent de 0
}
```

11. Flots

4. Ecriture de données:

```
f.write((char *)&v,sizeof(v));
```

Ecriture du contenu de v dans le fichier

11. Flots

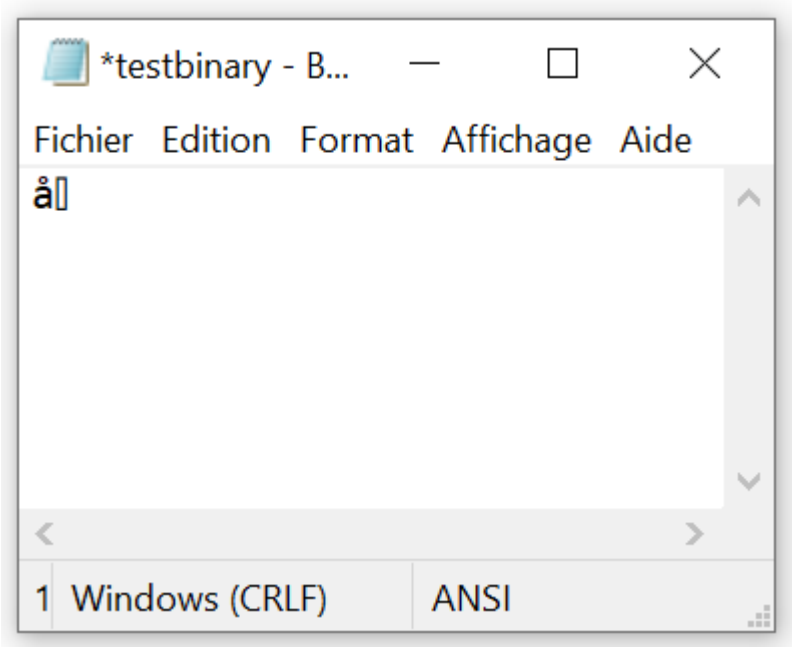
5. Fermeture du fichier:

```
f.close();
```

Fermeture du ficher. Cette étape est réalisée automatiquement par C++ dans le cas d'oublie.

11. Flots

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main()
{

ofstream f; // 1

f.open("testbinary.txt",ios::out|ios::binary|ios::trunc); // 2

if (!f.good())   //3
{
cerr << "Impossible d'écrire dans le fichier !\n";
exit(1);
}
int v=2021;

f.write((char *)&v,sizeof(v));    //4

f.close();     //5

return 0;
}
```



11. Flots

Fichiers binaires: (Lecture seulement)

1. Déclaration d'une variable de type fichier:

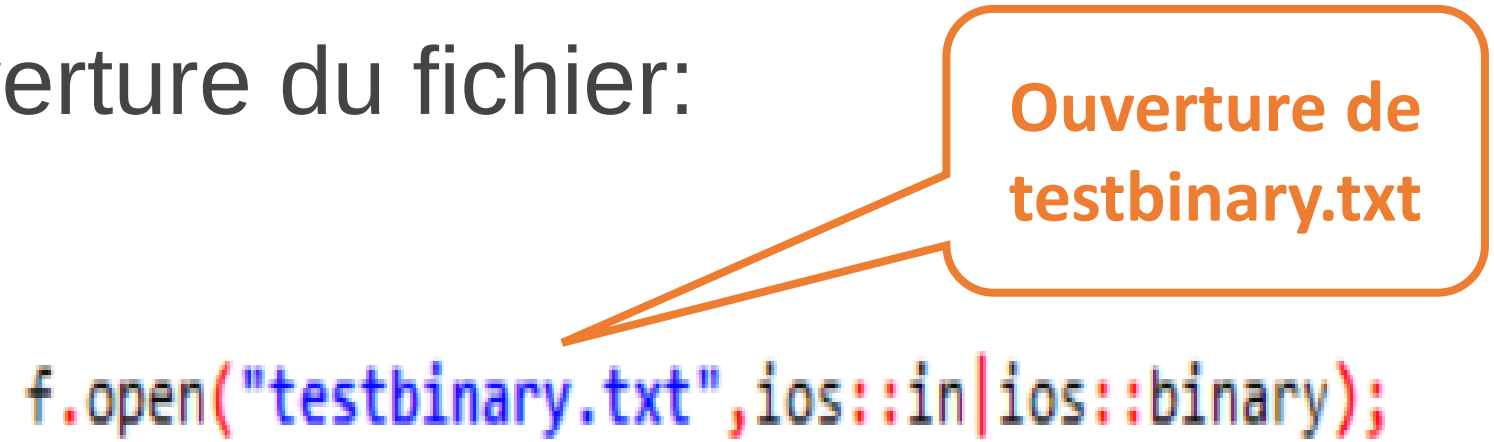
```
#include <iostream>
#include <fstream>
using namespace std;
ifstream f; // i pour input
```

Manipuler les fichiers

Déclaration de f

11. Flots

2. Ouverture du fichier:



11. Flots

3. Tester l'ouverture:

Problème d'ouverture

```
if (!f.good())
{
cerr << "Impossible de lire dans le fichier !\n";
exit(1); // arrêt du programme avec un code d'erreur différent de 0
}
```

11. Flots

4. Lecture de données:

```
f.read((char *)&v,sizeof(v));
```

Lecture du contenu du fichier

11. Flots

5. Fermeture du fichier:

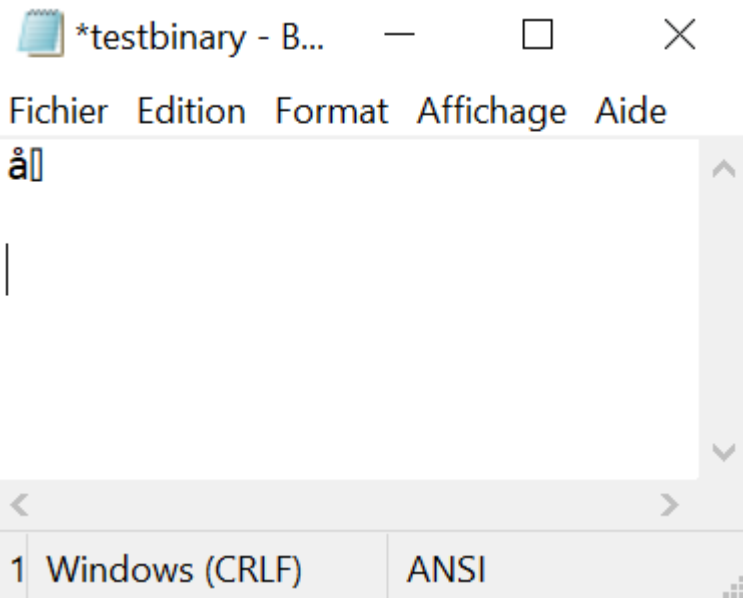
```
f.close();
```

Fermeture du ficher. Cette étape est réalisée automatiquement par C++ dans le cas d'oublie.

11. Flots

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main(){

ifstream f; //1

f.open("testbinary.txt",ios::in|ios::binary); //2

if (!f.good())    //3
{
cerr << "Impossible de lire dans le fichier !\n";
exit(1);
}
int v;
f.read((char *)&v,sizeof(v));       //4
cout<<v<<endl;

f.close();          //5

return 0;
}
```

```
2021

----------------
```



11. Flots

Accès aléatoire:

- Il est possible de positionner la **tête** de lecture/écriture à une position arbitraire du fichier.
- En C++, il y a deux têtes : une **tête de lecture** et une **tête d'écriture** .

11. Flots

Les opérations suivantes permettent de manipuler ces têtes :

Opération	Signification
f.tellp	Fonction qui renvoie la position de la tête d'écriture
f.tellg	Fonction qui renvoie la position de la tête de lecture
f.seekp(déplacement, origine)	Procédure qui change la position de la tête d'écriture
f.seekg(déplacement, origine)	Procédure qui change la position de la tête de lecture

11. Flots

- Pour les procédures, **seekp** et **seekg**, le déplacement représente un nombre d'octets à parcourir à partir de l'origine.
- L'origine peut valoir l'une des valeurs suivantes :

ios::beg	Déplacement par rapport au début du fichier (le déplacement doit être positif ou nul)
iso::end	Déplacement par rapport à la fin du fichier (le déplacement doit être négatif ou nul)
ios::cur	Déplacement par rapport à la position courante

12. Exceptions

- Une **exception** est l'interruption de l'exécution du programme à la suite d'un problème survenu (l'insuffisance de mémoire, la perte d'un fichier, la saisie non valide d'une valeur, la division par zéro, …).
- La **gestion des exceptions** consiste à prévoir ces erreurs (ces problèmes), à en informer les utilisateurs et éventuellement à mettre en œuvre des solutions de reprise et de correction de ces erreurs d'exécution.

12. Exceptions

```
#include <iostream>
using namespace std;
int main() {
int c,a=1,b=0;
c = a/b; // division par zéro!
cout << "c: " << c << endl;
return 0;
}
```

Exception
(Division par zéro)

12. Exceptions

- La mise en œuvre des exceptions nécessite trois étapes:
  - Définir une classe d'exception.
  - Lancer l'exception (throw).
  - Intercepter l'exception.

12. Exceptions

Définir une classe d'exception:

- Une classe d'exception correspond à une classe C++ qui peut fournir des informations sur une erreur.
- Exemple:

```
class Erreur
{
   ... // Données et fonctions membres
}
```

12. Exceptions

Lancer l'exception:

- Toute fonction qui souhaite lancer une exception doit utiliser l'opérateur **throw**, suivi par un objet créé à partir d'une classe d'exception.
- Cet opérateur permet de quitter la fonction qui l'utilise et d'informer la fonction appelante qu'une exception à été générée.

12. Exceptions

- Exemple:

```
void diviseur(int div)
{
   if(div==0) {Erreur er;
                  throw er;}
   cout<<" Diviseur non nul " <<endl;
}
```

Si div = 0 la fonction construit un objet statique de la classe d'exception Erreur et envoie cet objet avec l'opérateur throw qui ressemble à return.

12. Exceptions

La fonction précédente peut être simplifiée :

```
void diviseur(int div)
{
   if(div==0) {  throw Erreur();}
   cout<<" Diviseur non nul " <<endl;
}
```

12. Exceptions

Toute fonction susceptible d'envoyer un ou plusieurs exceptions peut l'indiquer dans sa déclaration juste après la liste de ces arguments. :

```
void controle(int v) throw (erreur)
{
   …
   …
}
```

Indication de l'exception

12. Exceptions

Intercepter l'exception:

- Pour intercepter une exception, C++ fournit une syntaxe basée sur l'utilisation des blocs **try** et d'un ou plusieurs blocs **catch**.

12. Exceptions

```
try
{    // Fonctions pouvant générer des erreurs
}
catch(...)
{    // Bloc s'exécutant pour l'exception indiquée en argument
}
catch(...)
{    // Bloc s'exécutant pour l'exception indiquée en argument
}
```

12. Exceptions

```
#include <iostream>
using namespace std;
class Erreur{
    public:
        Erreur();
        void message();
};
Erreur::Erreur(){
}
void Erreur::message(){
    cout<<"valeur incorrecte! \n";
}
```

12. Exceptions

```
void positive(int v)
{
    if(v<0) throw Erreur();
    cout<<"Valeur positive \n";
}
void inf(int v,int max)
{
    if(v>max) throw Erreur();
    cout<<"Valeur inférieure à : "<<max<<endl;
}
void sup(int v,int min)
{
    if(v<min) throw Erreur();
    cout<<"Valeur supérieure à : "<<min<<endl;
}
```

12. Exceptions

```
int main()
{
    int minimum=10; int maximum=100;
    try
    {
        int n;
        cout<<"Saisir une valeur entière : "; cin>>n;
        positive(n);
        inf(n,maximum);
        sup(n,minimum);
        cout<<"Valeur correcte !\n";
    }
    catch(Erreur er)
    {
        er.message();
    }
    catch(...)
    {
        cout<<"Erreur inconnue !\n";
    }
return 0;
}
```

Exercice5

L’exemple précédent contient une seule classe pour gérer toutes les erreurs.
Refaire le programme en rédigeant une classe pour chaque type d’erreur.

Exercice5

```
#include <iostream>
using namespace std;
int minimum=10; int maximum=100;
class Erreur_positive{
    public:
        Erreur_positive();
        void message();
};
Erreur_positive::Erreur_positive(){
}
void Erreur_positive::message(){
    cout<<"Erreur ! valeur negative ! \n";
}
```

Exercice5

```
class Erreur_inf{
    public:
        Erreur_inf();
        void message();
};
Erreur_inf::Erreur_inf(){
}
void Erreur_inf::message(){
    cout<<"Erreur ! valeur superieure A : "<<maximum<<endl;
}
//-----------------------------------------------------
class Erreur_sup{
    public:
        Erreur_sup();
        void message();
};
Erreur_sup::Erreur_sup(){
}
void Erreur_sup::message(){
    cout<<"Erreur ! valeur inferieure A : "<<minimum<<endl;
}
```

Exercice5

```
void positive(int v)
{
    if(v<0) throw Erreur_positive();
    cout<<"Valeur positive \n";
}
void inf(int v,int max)
{
    if(v>max) throw Erreur_inf();
    cout<<"Valeur inférieure à : "<<max<<endl;
}
void sup(int v,int min)
{
    if(v<min) throw Erreur_sup();
    cout<<"Valeur supérieure à : "<<min<<endl;
}
```

Exercice5

```
int main()
{ int n;
    try
    {
        cout<<"Saisir une valeur entière : "; cin>>n;
        positive(n);
        inf(n,maximum);
        sup(n,minimum);
        cout<<"Valeur correcte !\n";
    }
    catch(Erreur_positive er)
    {
        er.message();
    }
    catch(Erreur_inf er)
    {
        er.message();
    }
    catch(Erreur_sup er)
    {
        er.message();
    }
    catch(...)
    {
        cout<<"Erreur inconnue !\n";
    }
    return 0;
}
```