

# **P00**

# **Les Classes en C++**



# Les Classes

---

- ❑ Le C++ est une sorte de sur-ensemble du C destiné à implémenter les concepts de l'objet dans le langage C.
- ❑ La classe C++ est une généralisation de la structure struct de C, elle comporte des champs, qui sont les attributs de la classe (les données), et des fonctions, qui sont les méthodes agissant sur ces données.
- ❑ La définition d'une classe en C++ est identique à celle d'une structure struct de C, dans laquelle on a placé des définitions de fonctions concernées par les données de la structure



# Exemple :classe cercle

---

```
typedef struct cercle
{
    float Rayon, abs_c, ord_c;
    // abscisse et ordonnée
} C1;

float perimetre (int R)
{
    return 2*3.141593*R;
}

main() { perimetre (C1.rayon); ...}
```



# Exemple :classe cercle

---

```
class cercle
{
    float Rayon, abs_c, ord_c;
        // abscisse et ordonnée
    float perimetre (void)
    {
        // La fonction périmètre est déclarée dans
        // la structure qui va l'utiliser
        return 2*3.141593*Rayon;
    };
};

main() { cercle C1; C1.perimetre(); }
```



## Exemple 2 : classe Point

---

```
class Point
{
public:
    void afficher()
    {cout << '(' << x << ',' << y << ')'; }
    void placer(int a, int b)
    {
        //validation des valeurs de a et b
        x = a; y = b;
    }
private:
    int x, y;
};
```



## Exemple 2 : classe Point

---

```
class Point
{
public:
    void afficher()
    {cout << '(' << x << ',' << y << ')'; }
    void placer(int a, int b)
    {
        //validation des valeurs de a et b
        x = a; y = b;
    }
private:
    int x, y;
};
```

```
main()
{
    Point P;
    P.placer(3,5);
    P.afficher();
};
```

```
main()
{
    Point P;
    P.x=3; P.y=5; //NON
    P.afficher();
};
```



# Classe point

---

- Chaque objet de la classe Point comporte un peu de mémoire, composée de deux entiers  $x$  et  $y$ , et de deux fonctions :
  - afficher, qui accède à  $x$  et  $y$  sans les modifier,
  - et placer, qui change les valeurs de  $x$  et  $y$ .
- L'**association** de **membres** et **fonctions** au sein d'une **classe**, avec la possibilité de rendre privés certains d'entre eux, s'appelle l'*encapsulation* des données



# Intérêt

---

- Intérêt de la démarche : puisqu'elles ont été déclarées privées, les coordonnées x et y d'un point ne peuvent être modifiées autrement que par un appel de la fonction placer sur ce point.
- Or, en prenant les précautions nécessaires lors de l'écriture de cette fonction (ce que nous avons noté « *validation des valeurs* de a et b ») le programmeur responsable de la classe Point peut garantir aux utilisateurs de cette classe que tous les objets créés auront toujours des coordonnées correctes.
- Autrement dit : *chaque objet peut prendre soin de sa propre cohérence interne.*





# Définition

---

La définition d'une classe désigne la spécification des attributs et des méthodes de la classe.

- Règles syntaxiques de la définition :
  - La définition commence par le mot-clé *class*.
  - Ensuite, il y a lieu de déclarer les collections d'attributs et de méthodes. Ces déclarations doivent être placées entre deux *accolades*.
  - La définition d'une classe se termine enfin par un *point virgule*.

Exemple :

```
class Date
{ int jour, mois, annee;
  void Saisir();
  void Afficher();
};
```



# Instanciación

---

- Une classe définit un nouveau type. De ce fait, il est possible de déclarer des variables ou des constantes ayant un type classe.
- Ces variables et constantes sont appelées *des objets* ou *des instances (occurrence)* de la classe.
- Le processus de création d'objets est appelé *instanciation*.
- **Exemple :**

Date D; // D est un objet de type classe Date.



# Déclaration des attributs d'une classe

- Les attributs d'une classe peuvent être des variables ou des constantes. Toutefois par rapport à ces dernières, les attributs se distinguent par le fait **qu'au moment de leur déclaration dans la définition de la classe aucun espace mémoire ne leur est réservé.**
- En effet la définition d'une classe constitue un processus de définition de **type** et non un processus de déclaration d'objet.
  - De ce fait, il devient impossible d'initialiser un attribut au moment de la définition d'une classe !!

**Exemple : class A**  
**{** .....  
**int i = 5; // Erreur**  
**};**



# Classe anonyme

---

- Une classe anonyme est une classe qui ne porte pas de nom (sans nom). Elle sert essentiellement à faire des déclarations directes d'objets (dans la même instruction qui définit la classe).

- **Exemple :**

```
class {.....} D;
```

```
// D est un objet de type classe sans nom
```

***NB : Une instance d'une classe anonyme ne peut pas figurer dans la liste d'arguments d'une fonction (Il est impossible en effet de spécifier le type du paramètre formel puisqu'il est sans nom).***

# Fonctions membres : méthodes

- La déclaration des méthodes se fait suivant la même syntaxe que la déclaration des fonctions classiques. Toutefois une méthode doit être obligatoirement déclarée à l'intérieur d'une classe. Il existe **deux façons** pour définir une méthode de classe :

- **Première façon :**

Elle consiste à définir la méthode au sein même de la classe.

*Exemple :* `class Date`

```
{  
    int jour, mois, annee;  
    void Saisir() {.....}  
    void Afficher(){.....}  
};
```



# Fonctions membres

---

- **Deuxième façon :**

- Elle consiste à : déclarer la fonction à l'intérieur de la classe,
- la définir ensuite à l'extérieur de la classe. Il faut pour cela indiquer la classe d'origine de la fonction.
- Ceci se fait comme suit :

**TypeRetour NomClasse::Methode(type1 arg1,..., typeN argN);**

- **::** est l'opérateur de résolution de portée.
- NomClasse::Methode est appelé le nom qualifié de la méthode.



# Exemple

---

```
class Date
{
    int jour, mois, annee;
    void Saisir();
    void Afficher();
};

void Date::Afficher()
cout<<jour<<"/"<<mois<<"/"<< annee;
void Date::Saisir() {
cout<<"Donnez dans l'ordre le
jour, le mois et l'année";
cin>>jour>>mois>>annee; }
```



# Remarques

---

- Une méthode définie à l'intérieur d'une classe est considérée par défaut comme étant *inline*.
- Il convient par conséquent de définir à l'intérieur de la classe les fonctions courtes, non récursives et ne comportant pas de boucles.
- Une fonction définie à l'extérieur de la classe n'est pas considérée comme *inline*. Pour la rendre ainsi, il faut ajouter explicitement la spécification *inline* devant sa définition.

**Exemple :**

```
inline void Date :: Afficher()  
{.....}
```

ENISO 2021-2022





# Encapsulation

---

- L'encapsulation est le mécanisme qui permet de regrouper les données et les méthodes au sein d'une même structure (classe). Ce mécanisme permet également à l'objet de définir le niveau de visibilité de ses membres.
- Un membre visible est un membre qui est accessible à partir de l'objet :
  - Si le membre est un attribut alors l'accès concerne une opération de lecture ou d'écriture de la valeur de cet attribut.
  - Si le membre est une méthode alors l'opération d'accès consiste en un appel de cette méthode.



# Encapsulation

---

- Généralement un objet ne doit exposer (rendre visible) que les membres qu'il juge utiles pour ses utilisateurs. Il doit cacher tous les détails de son implémentation (corps des méthodes ainsi que les attributs et méthodes qui sont utilisés en interne).
- Exemple:

```
Class étudiant {  
    public :  
        char nom[20], prenom[20];  
        int age;  
        void ajouter(); void supprimer();  
    private  
        float moyenne();  
}
```



# Spécificateurs des droits d'accès des membres

---

En C++, la spécification des droits d'accès aux membres d'une classe se fait à l'aide des trois mots-clés suivants :

- **public** : les membres d'une classe X (attributs ou méthodes) accessibles en dehors de la classe, généralement par les utilisateurs de cette dernière.
- **private** : restreint l'accès aux membres de la classe (attributs et méthodes) aux méthodes de la classe seulement, ainsi qu'aux fonctions déclarées amies de la classe.
- **protected** : effet similaire à private mais qui est toutefois moins sévère. En effet protected restreint l'accès aux membres d'une classe X aux méthodes de X, aux fonctions amies de X et aux méthodes des classes basées sur X.



# Portée d'un spécificateur d'accès

---

- La portée d'un spécificateur d'accès s'étend depuis l'endroit de sa définition jusqu'à la rencontre de la définition d'un autre spécificateur.

```
class X  
}  
  private :  
    int a1;  
    char a;2  
  public :  
    void f;()1  
    int f2(double k;(  
;{
```

```
class y  
}  
  public :  
    int  
    a;1  
    void f1();  
  private:  
    char a;2  
    int f2(double k;(  
;{
```



# Accès à l'intérieur aux membres d'une classe

- Les membres d'une classe sont locaux. Leur portée est définie par la classe dans laquelle ils sont déclarés.
- Tout membre d'une classe est visible par les autres membres de cette classe sans aucune considération des spécificateurs d'accès (*public*, *private*, *protected*).
- Les attributs d'une classe sont considérés comme des variables globales par rapport aux méthodes de cette classe. Tout attribut peut par conséquent être utilisé par n'importe quelle méthode de sa classe.
- Une méthode d'une classe peut être appelée par toute autre méthode de sa classe indépendamment de l'ordre de déclaration de ces dernières dans la classe.
- Les membres peuvent être manipulés directement par leurs noms sans avoir besoin d'aucune qualification.



# Exemple 1

---

```
class Date
{private :
    int jour, mois, annee;
public:
    void Saisir();
    void Afficher();
};

void Date::Saisir()
{
    cout<<"Donnez dans l'ordre le
jour, le mois et l'année;"
cin>>jour>>mois>>annee; }
```

```
void Date::Afficher()
{
    cout<<jour<<"/"<<mois>>"/">>
annee;
}

main();
{ Date D1;
  D1.Saisir();
  D1.Afficher(); }
```



## Exemple 2 : N!

```
class fact {
    int n, F;
    public:
    void saisie()
    {
        cout << "donner n";
        cin>>n;
    }
    void calcul()
    {   int i; F=1;
        for (i=1;i<=n;i++)
            F*=i;
    }
    void afficher()
    { cout<<F; }
};
```

```
main();
{
    fact f1;
    f1.saisie();
    f1.calcul();
    f1.afficher();
}
```



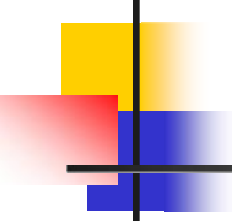
# Accès à l'extérieur aux membres d'une classe

---

- L'utilisation d'un membre d'une classe en dehors de cette dernière (par exemple par une fonction non membre de la classe) se fait généralement à partir d'une instance.
- Le membre doit être associé à l'objet auquel il appartient. Cette association se fait à l'aide de l'opérateur *point* de la manière suivante :
  - Pour les attributs : **Objet.Attribut;**
  - Pour les méthodes : **Objet.Methode(Paramètres effectifs);**
- Seuls *les membres publics* peuvent être accédés de l'extérieur d'une classe. Les autres membres (*private* et *protected*) **ne sont pas accessibles** de l'extérieur.



# Exemple



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

```
{
class Date
{ private :
int jour, mois;
public:
    int annee ;
    void Saisir();
    void Afficher();
};
Date D;
D.Jour = 5; // erreur
D.Mois = 10; // erreur
D.Annee = 2014 // OK
Cout<<D.jours; // erreur
Cout<<D.mois; // erreur
Cout<<D.annee // ok
}
```

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

```
{
class Date
{ int jour, mois;
public:
    int annee ;
    void Saisir();
    void Afficher();
};
Date D;
D.Saisir(); // OK
D.afficher(); // OK
}
```



# Portée des classes

---

- Un type "classe" n'est visible que dans le bloc dans lequel il est déclaré. Ainsi :
  - Toute classe déclarée dans une fonction n'est instanciable que dans cette fonction.
  - Toute classe X déclarée à l'intérieur d'une classe Y n'est instanciable que dans Y.
  - Une classe déclarée en dehors de tout bloc est une classe globale. Elle peut être instanciée n'importe où dans le programme.



# Classes imbriquées

- Une classe X peut avoir un membre de type une autre classe Y. Les deux classes sont dites dans ce cas *des classes imbriquées*.

- **Exemple :**

```
class Personne
{
    char Nom[20];
    char Prenom[20];
    private
        Date Dn; // ou int jour,mois, annee;
    Public :
        void Saisir();
        void Afficher();
}
```

*Date* et *Personne* sont dans ce cas deux classes imbriquées.



# Manipulation des attributs et des méthodes des classes imbriquées

---

- On considère les classes *Personne* et *Date déjà* définies.
- Possibilité d'accéder aux champs de *Date* à partir de la classe *Personne* et ce en utilisant différents spécificateurs d'accès.
- Dn est un champ public et ses champs sont publics.
  - `Personne P;`
  - `P.Dn.Jour; // OK`
- Dn est un champ privé et ses champs sont publics.
  - `Personne P;`
  - `P.Dn.Jour; // Erreur`
- Dn est un champ public et ses champs sont privés.
  - `Personne P;`
  - `P.Dn.Jour; // Erreur`



# Les tableaux d'objets

---

- Déclaration : **TypeClasse** **NomTableau[NbElements]** ;
- Accès aux attributs d'un élément du tableau:  
**NomTableau[indice].NomAttribut**
- Accès aux méthodes d'un élément du tableau:  
**NomTableau[indice].NomMéthode(<args>)**

- ***Exemple :***

... ..

**Date TD[5];**

**// Saisie de 5 dates**

**for(int i=0;i<5;i++) TD[i].Saisir();**

**// Affichage des dates**

**for(int i=0;i<N;i++) TD[i].Afficher();**

... ..



# Pointeur sur un objet

---

- Il est possible de déclarer un pointeur vers un objet. Ce pointeur doit être obligatoirement initialisé pour être utilisable. L'accès aux membres d'une classe à partir d'un pointeur se fait à l'aide de l'opérateur ->
- Déclaration : **TypeClasse \* PtrObj;**
- Initialisation : **PtrObj = new TypeClasse;**
- Accès aux attributs : **PtrObj->Attribut** ou également : **(\*PtrObj).Attribut**
- Accès aux méthodes : **PtrObj->Methode(liste d'arguments)**

ou également : **(\*PtrObj).Methode(liste d'arguments)**

*Exp : **date \*d;** d un pointeur qui pointe sur un objet de type class  
date*



# Le pointeur this

---

- *this* est un pointeur particulier en C++ qui pointe toujours sur l'objet en cours d'utilisation. Ce pointeur possède plusieurs champs d'applications.

## **Exemple:**

La méthode *Saisir* définie comme suit :

```
void Date::Saisir()  
{  
    cout<<"Donnez dans l'ordre le jour, le  
    mois et l'année;"  
    cin>>jour>> mois>>annee;  
}
```

En réalité implicitement définie comme suit:

```
void Date::Saisir(Date *this(  
}  
cout<<"Donnez dans l'ordre le jour, le  
mois et l'année;"  
cin>>this->jour>>this->mois>>this-  
<annee;  
{
```



# Utilisation explicite de this

---

- Un des problèmes qui peuvent être résolus par l'utilisation du pointeur *this* est celui qui se pose lors de l'utilisation d'une méthode qui possède des paramètres portant les mêmes noms que les attributs.

- **Exemple :**

```
class A
{
    int x, y;
public :
    void f(int x, int y)           // x,y paramètres de fonction
    {
        this->x=x;                // x attribut d'objet
        this->y=y;                // y attribut d'objet
    }
};
```





# Lecture de l'adresse et de la taille d'un objet

---

- Il est possible de récupérer l'adresse mémoire d'un objet et ce à l'aide de l'opérateur **&**.
- Il est possible de récupérer la taille en mémoire d'un objet et ce à l'aide de l'opérateur ***sizeof***. Cette taille est égale à la somme des tailles des attributs de l'objet.



# Exemple

---

```
class CX
{ public :
    int a1;
    int a2;
    void Affiche()
        { cout<<" a1: "<<a1<<" et a2: "<<a2;}
};

void main()
{   CX x;
    cout<<"\n Adresse de x : "<<&x;
    cout<<"\n Taille de x : "<<sizeof(x);
}
```



# Affectation d'objets

---

- Il est possible d'affecter d'une manière globale un objet O1 à un autre objet O2 de la même classe. La valeur de chaque attribut de O1 est alors copiée dans l'attribut qui lui correspond dans O2.

- ***Exemple :***

```
void main()  
{    CX x1,x2;  
  
    x1.a1=5;  
    x1.a2=8;  
    x2=x1;  
    x2.Affiche();  
    x1.a1=18;  
    x1.Affiche();  
    x2.Affiche();  
}
```



# Affectation entre pointeurs sur des objets !!!

---

## ***Exemple :***

```
void main()
{  CX *px1,*px2;
   px1=new CX(); // Objet sans nom
   px1->a1=5;    //  (*px1).a1=5;
   px1->a2=8;
   px2=px1; // px1 et px2 pointent sur le même objet
   px2->Affiche();
   px1->a1=18;
   px1->Affiche();
   px2->Affiche();
   delete px1;
   delete px2; // Erreur car l'objet a été déjà libéré
}
```

# Initialisation des attributs d'un objet (**Explicite**)

- Il est possible d'attribuer des valeurs initiales aux attributs d'un objet à travers une liste de valeurs

```
class Point2D
{public:
int x;
int y;
};
```

```
class Point3D
{public:
int x;
int y;
int z;
};
```

- Point2D P1={5,6};
- Point3D P2={5,6,9};
- Point3D P3={4,,9} // le deuxième attribut est initialisé à 0.
- Point2D T[3] = {1,5,6,8,9,10};
- Point2D T[3] = {{1,5},{6,8},{9,10}};
- Les attributs doivent être déclarés **public**



# Initialisation des attributs d'un objet (**Explicite**)

**class TabEntiers**

**{**

**int Nb; int\* T;**

**public :**

**void Init(int Ni)**

**{**

**Nb=Ni;**

**T=new int [Nb];**

**//ou T=(int \*)malloc(Nb\*sizeof(int);**

**}**

**void Init(int\* Ti,int Ni)**

**{**

**Nb=N;**

**T=new int ]Nb;[**

**for(int k=0;k<Nb;k(++**

**T[k]=Ti[k;[**

**{**

**void main()**

**}**

**TabEntiers TE;**

**TE.init;(5**

**{**

Exemple :



## Initialisation explicite : Inconvénients

---

- Pour pouvoir utiliser l'objet, il est nécessaire d'appeler à chaque fois, d'une manière explicite, la méthode *Init*.
- Cet appel explicite peut engendrer des erreurs surtout dans les programmes de grande taille où le risque d'oubli devient élevé.
- L'utilisation de *Init* ne peut pas être considérée comme étant une initialisation au vrai sens technique du terme car une initialisation se fait généralement dans la même instruction que la déclaration.



## 2. Initialisation implicite

### **Notion de constructeur**

---

- Un constructeur est une méthode particulière d'une classe qui s'exécute automatiquement d'une manière implicite, lors de la création d'un objet de cette classe.
  
- **Rôle et intérêt**
  - Un constructeur d'une classe assure la réservation de l'espace mémoire nécessaire à la création de tout objet de cette classe.
  - L'espace dont on parle ici désigne l'espace de base nécessaire à la création de l'objet.
  - Les constructeurs sont généralement exploités par les programmeurs pour réaliser l'initialisation des attributs de la classe et pour allouer des espaces mémoires nécessaires aux membres dynamiques de la classe.





# Déclaration et définition

---

- Un constructeur est une méthode de la classe.
- Un constructeur doit porter le même nom que celui de sa classe.
- Un constructeur peut admettre 0, un ou plusieurs paramètres.
- Un constructeur ne retourne aucune valeur
- Une classe peut comporter un ou plusieurs constructeurs. Dans le cas d'existence de plusieurs constructeurs, ces derniers doivent respecter les règles de surcharge des méthodes.
- Comme toute méthode un constructeur peut être défini à l'intérieur de la classe ou à l'extérieur.
- **Syntaxe : NomClasse (<paramètres>);**



# Exemple

---

```
class Time
{
    int Hour;
    int Minute;
    int Second;
public :
    Time(int H,int M, int S)
    {
        Hour = H;
        Minute = M;
        Second = S;
    }
}
```

```
void Affiche()
{
    cout<<"L'heure est  ;" :
    cout<<Hour<<':'<<Minute`:'>>
        >>Second<<endl;
    {
};{

main()
} Time T(;(33,5,7
    T.affiche;()
    {
}
```



# Appel d'un constructeur

---

- La création d'un objet de type auto peut se faire à travers un appel explicite ou implicite du constructeur.

## ***Syntaxe de l'appel explicite :***

- Le constructeur est appelé par son nom comme une méthode classique.
- **NomClasse NomObjet = NomClasse(<paramètres effectifs>);**

Time T = **Time(16,30,25);**

## ***Syntaxe de l'appel implicite :***

- **NomClasse NomObjet(<paramètres effectifs>);**

Time **T(16,30,25);**



# Cas d'un objet dynamique

---

- Pour la création dynamique d'objets, seul l'appel explicite du constructeur est possible.

- ***Syntaxe :***

**NomClasse\* ptrObjet = new NomClasse(<paramètres effectifs>);**

Time\* T = new Time(16,30,25);

- ***Exemple :***

Time T1(16,30,25);

Time\* T2 = new Time(17,44,59);

T1.Affiche();

T2->Affiche(); // ou (\*T2).Affiche()



# Exemple

---

```
class Time
{
    int Hour;
    int Minute;
    int Second;
public:
    Time(int H,int M, int S(
    {
        Hour =H;
        Minute =M;
        Second =S;
    }
```

```
void Affiche()
{
    cout<<"L'heure est  ";
    cout<<Hour<<':'<<Minute\':'>>
        >>Second<<endl;
    {
;{
main} ()
    Time T1(16,30,25
    Time* T =2 new Time(17,44,59);
    T1.Affiche();
    T2->Affiche(); // (*T2).affiche();
    {
```

# Surcharge des constructeurs



```
class TabEntiers
```

```
{ int Nb;
```

```
int* T;
```

```
public :
```

```
TabEntiers(int Ni)
```

```
{ Nb=Ni;
```

```
  T=new int [Nb];
```

```
}
```

```
TabEntiers(int* Ti,int Ni)
```

```
{ Nb=Ni;
```

```
  T=new int [Nb];
```

```
  for(int k=0; k<Nb; k++)
```

```
    { T[k]=Ti[k]; }
```

```
}
```

```
void Saisir()
```

```
{
```

```
  for(int k=0;k<Nb;k++)
```

```
  {cout<<"Donner l'élément d'indice "<<k;  
    cin>>T[k];}
```

```
void Afficher()
```

```
{
```

```
  for(int k=0;k<Nb;k++)
```

```
  {cout<<T[k]<<' ';
```

```
}
```

```
}
```

```
}
```

```
void main()
```

```
{
```

```
  int Ti[3]={21,3,45};
```

```
  TabEntiers TE1(Ti,3);
```

```
  TE1.Afficher();
```

```
  TabEntiers TE2(2);
```

```
  TE2.Saisir();
```

```
  TE2.Afficher(); }
```



# Constructeur par défaut

- On appelle constructeur par défaut tout constructeur qui peut être appelé sans lui passer d'arguments.

```
class Time
```

```
{
```

```
    int Hour, Minute, Second;
```

```
    public :
```

```
    Time() // Constructeur qui initialise l'heure par défaut à Midi
```

```
    {Hour = 12; Minute = 0; Second = 0; }
```

```
    Time(int H, int M, int S)
```

```
    {Hour = H; Minute = M; Second = S;}
```

```
    .....
```

```
    Time T; // T contient Hour=12, Minute=0, Second=0;
```



# Constructeur par défaut

---

- **Attention !!!**

```
class Time
```

```
{
```

```
    int Hour, Minute, Second;
```

```
    public :
```

```
    Time() // Constructeur qui initialise l'heure par défaut à Midi
```

```
    {Hour = 12; Minute = 0; Second = 0; }
```

```
    Time(int H=12, int M=0, int S=0)
```

```
    {Hour = H; Minute = M; Second = S;}
```

```
    .....
```

```
    Time T; // Erreur de Compilation !!
```





# Constructeur par défaut

**Attention :** Le compilateur ne génère aucun constructeur par défaut pour les classes qui possèdent au moins un constructeur explicitement défini.

```
class Time
{ int Hour, Minute, Second;
public :
    Time (int H, int M, int S){ .... }
    Affiche(){ .... }
};
void main()
{ ....
    Time T; // Erreur pas de constructeur par défaut
    ....
}
```



# Le destructeur

---

- Un destructeur joue le rôle symétrique du constructeur. Il est automatiquement appelé pour libérer l'espace mémoire de base nécessaire à l'existence même de l'objet (espace occupé par les attributs). Cette libération est implicite. Elle est effectuée par tout destructeur.
- Il est généralement recommandé d'exploiter le destructeur pour faire :
  - La libération d'éventuels espaces mémoires dynamiques.
  - La fermeture d'éventuels fichiers utilisés par l'objet.
  - L'enregistrement des données, etc.



# Déclaration et définition d'un destructeur

---

- Un destructeur porte le même nom que celui de la classe mais précédé d'un tilde  $\sim$  (pour le distinguer du constructeur).
- Un destructeur ne retourne aucune valeur.
- Un destructeur ne prend jamais de paramètres.
- Un destructeur est une méthode de la classe. Toutes les règles qui s'appliquent aux méthodes (portée, qualification d'accès, déclaration, définition) s'appliquent donc également au destructeur.
- Une classe ne peut disposer que d'un seul destructeur.



# Exemples

---

## **Exemple 1**

```
class Time
{
    int Hour;
    int Minute;
    int Second;
public :
    Time () { .... }
    Time (int H, int M, int S) { .... }
    Time (const char* str) { .... }
    ~Time() {} // Destructeur vide
    Affiche() { .... }
};
```

## **Exemple 2**

```
class TabEntiers
}public :
int Nb;
int* T;
TabEntiers(int Ni){...}
TabEntiers(int* Ti,int Ni{...}{
~TabEntiers(){delete[] T{;
//Destructeur
void Saisir{...}()
void Afficher{...}()
;{
```



# Appel du destructeur

---

- **Appel explicite :**

NomObjet.~NomClasse()  
ou PointeurObjet->.~NomClasse()

Cet appel reste toutefois rare d'utilisation.

- **Appel implicite :**

***Exemple :***

```
void main()
{
    TabEntiers TE(2);
    TE.Saisir();
    TE.Afficher();
}
```



# Exemple

---

A- Étant donné le programme suivant :

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

```
void main()
```

```
{ cout<<"Bonjour tout le monde"<<endl; }
```

Modifier le programme sans toucher toutefois à la fonction *main* pour que le résultat d'exécution soit le suivant :

*Avant bonjour*

*Bonjour tout le monde*

```
Class Toto {
```

```
    void Toto() {cout<<"Avant bonjour"<<endl;}
```

```
}
```

```
Toto x;
```

```
void main()
```

```
{ cout<<"Bonjour tout le monde"<<endl; }
```

ENISO 2021-2022



# Exercice 1

Réaliser une classe FUSION permettant de fusionner deux tableaux triés A et B en un tableau C qui soit lui aussi trié ?

```
class FUSION {
```

```
int *A, *B, *C ;
```

```
Int NA, NB, NC;
```

```
public :
```

```
fusion(int x, int y)
```

```
saisie_A() ;
```

```
saisie_B() ;
```

```
fusionner() ;
```

```
afficher_C() ;
```

```
~fusion()
```

```
}
```

3	8	12	20	40
---	---	----	----	----

**A**

1	2	9	18	30	32	34
---	---	---	----	----	----	----

**B**

1	2	3	8	9	12	18	20	30	32	34	40
---	---	---	---	---	----	----	----	----	----	----	----

**C**