LES CLASSES

1. PRESENTATION

Une classe est semblable à une structure à la seule différence qu'au lieu du mot clé struct on utilise le mot clé class. Il est légitime alors de se poser la question : Faut-il utiliser class ou struct pour définir des types composés comme le type Date définit précédemment ?

Pour répondre à cette question, considérons les deux codes suivant :

```
class Date {
    int  Day, Month, Year;
} ;
```

```
struct Date {
    int  Day, Month, Year;
} ;
```

L'une comme l'autre est une déclaration valide. Mais :

- Pour la première (class), comme le qualificatif d'accès pour les trois membres Day, Month et Year n'est pas précisé, il est fixé par défaut comme étant private. Ceci est également vrai pour les fonctions membres;
- Pour la deuxième (struct), le qualificatif d'accès n'est pas également précisé, cependant il est fixé par défaut comme étant public. Ceci est également vrai pour les fonctions membres.

☞ Si la classe a été déclarée comme suit :

```
class Date {
    public :
        int    Day, Month, Year;
} ;
```

La structure et la classe auraient été identiques.

On peut donc déjà noter que la différence entre une classe et une structure CPP est que, sans déclaration explicite, les données membre de la première sont considérées comme private alors que pour la deuxième elles sont considérées comme public. Cette différence est d'une importance capitale parce qu'elle fait apparaitre le principe d'encapsulation propre à la programmation objet et attaché à l'utilisation des classes. Mais pour répondre à notre question précédente nous précisons que :

1. D'abord un objet est une entité indépendante qui stocke ses propres données et détiens ses propres fonctions.
2. Bien que nous nous sommes autorisés, jusqu'à maintenant, à appeler objets des variables de type structure, comme Date, ce mot est habituellement utilisé pour décrire des variables

de type class. Ainsi la « *programmation par objet* » concerne les programmes qui utilisent des classes.

Aussi et conformément à l'usage courant, nous utiliserons désormais le mot clé class.

Une déclaration d'une classe se fait donc tout simplement comme suit :

```
class class_Identifier
{
    public :
        Public_Members ;
    private :
        Private_Members ;
} ;
```

Où :

class : mot clé de création de la classe ;

class_Identifier : désigne le nom de la classe ;

public : mot clé qui déclare que les membres suivants sont de type public;

Public_Members : L'ensemble des membres publics de la classe. Ces membres sont donc accessibles depuis l'extérieur de la classe et peuvent être des *données* aussi bien que des méthodes;

private : mot clé qui déclare que les membres suivants sont de type privé;

Private_Members : L'ensemble des membres privés de la classe. Ces membres ne sont donc accessibles que depuis l'intérieur de la classe (par des membres de cette classe) et peuvent être des données aussi bien que des méthodes.

Toujours conformément à l'usage courant, nous avons placé le mot clé public avant le mot clé private. Si on avait placé les champs private avant les champs public on n'aurait pas besoin d'utiliser explicitement le mot private puisqu'il est considéré implicitement :

```
class Identificateur {
    public :
        Public_Members ;
    private :
        Private_Members ;
} ;
```

```
class Identificateur {
    Public_Members ;
    public :
        Private_Members ;
} ;
```

Ces deux structures sont exactement équivalentes, parce que les membres privés n'ont pas à être déclarés explicitement. Cependant la déclaration public, dans la classe de droite, est obligatoire parce que sans elle tous les membres seront considérés comme private.

Reprenons notre structure Date, dans sa toute dernière forme du chapitre II, et utilisons le mot le mot clé class au lieu de struct.

EXEMPLE:

```
#include <iostream>
#include <cstring>
using namespace std ;
class Date {
    public :
        int Set_Date(int day = 28, int  month = 2, int year = 2018)
        {
            if(day < 1 || day > 30 || month < 1 || month > 12 || year < 0)
                return(1) ;
            Day     = day ;
            Month = month;
            Year    = year;
            return(0) ;
        }
        void Display_Date()
        {
            cout <<  Day  << '/' << Month <<  '/' << Year << « \n » ;
        }
    private :
        int  Day , Month, Year;
} ;
main()
{
    int Flag ;
    Date D ;
    Flag = D.Set_Date(-3, 2, 2018 ) ;
    if(Flag)
        cout <<  " Mauvaise valeurs de la date : " ;
    else
    {
        cout <<  " La méthode Set_Date a fixée la date à : " ;
        D. Display_Date() ;
    }
    Flag = D. Set_Date(28, 2, 2018 ) ;
    if(Flag)
        cout <<  " Mauvaise valeurs de la date : " ;
    else
    {
        cout <<  " La méthode Set_Date a fixée la date à : " ;
        D. Display_Date() ;
    }
}
```

Ici D est déclaré comme objet de la classe Date;

☞ Quand on déclare un objet comme D, on dit qu'on a opérer une instanciation de la classe Date et que D est une instance de cette classe.

- Cet objet D a ses propres données membres internes (privés) Day, Month et Year. Les méthodes (fonctions) membres quant à elles, elles sont communes à tous les objets;

Les mêmes règles, que celles déjà utilisées pour l'accès et la manipulation des membres d'une structure, restent valables pour les classes à savoir :

- On appel une méthode membre comme Set_Date() en préfixant son nom du nom de son propriétaire : D.Set_Date(...). C'est la seule façon d'appeler une méthode membre. Nous dirons que l'objet D est *propriétaire* de l'appel.
- Les corps des méthodes Set_Date(...) et Display_Date() sont donnés à l'intérieur de leurs classe Date. Généralement l'inclusion, du corps d'une méthode membre à l'intérieur de sa classe même, n'est pratiquée que pour les fonctions dont le code est succinct. Ces fonctions sont appelés alors fonctions inline (en ligne) ;
- Dans la plupart des cas, on préfère définir les méthodes membres à l'extérieur de la déclaration de la classe, en utilisant l'opérateur de résolution de portée **::**. La syntaxe étant:

  Identificateur_classe**::**Identificateur_méthode

  Le qualificatif Identificateur_classe**::**, comme préfixe de chaque non de méthode (Identificateur_méthode), est indispensable à la définition d'une méthode membre figurant hors de la définition de sa classe. l'opérateur de résolution de portée (class scope resolution operator) **::** est utilisé pour rattacher la définition de la méthode à sa classe (Date). Sans ce qualificatif, le compilateur ne saurait pas que la méthode en cours de définition est membre de telle ou telle classe et cette façon de faire est en accord avec le principe d'encapsulation. En effet, les définitions sont souvent mises dans un fichier séparé et compilés séparément ce qui signifie que les programmes applicatifs, qui utilisent la classe, ont seulement besoin de savoir ***ce que font*** les objets ; ils n'ont pas besoin de savoir ***comment*** les objets le font ;

2. AFFECTATION D'OBJETS D'UNE CLASSE

Les données des objets instances d'une classe peuvent être affectées de deux manières :

2. 1. PAR UTILISATION DES FONCTIONS D'ACCES EN MODIFICATION

Ces fonctions appelées également accesseurs en modification (***Setters***), logées obligatoirement dans la partie public de la classe, permettent de modifier les champs donnés (private ou public) de cette classe, comme nous l'avons déjà fait dans les exemples précédents de la classe Date. Si on essaye d'accéder directement aux champs données private, sans passer par les ***Setters***, une erreur est générée.

EXEMPLE:

```
class Date {
    public :
        int Set_Date(int day = 28, int  month = 2, int year = 2018)
        {
            if(day <1 || day > 30 || month <1 || month > 12 || year < 0)
                return(1) ;
            Day   = day;
            Month = month;
            Year  = year;
            return(0) ;
        }
        void Display_Date() { cout <<  Day << '/' << Month <<  '/' << Year << « \n » ; }
    private :
        int   Day, Month, Year;
} ;

.....
int flag ;
Date D ;
Flag = D. Set_Date(...) ;
Flag = D(...)
```

L'instruction D.Set_Date(...) est tout à fait légale parce qu'elle se sert de la méthode membre, Set_Date, pour accéder aux données private de l'objet D de la classe Date, alors que l'instruction Flag = D(...) est rejetée parce qu'elle tente d'accéder directement aux membres private de cette même classe.

2. 2. PAR UTILISATION D'AFFECTATIONS GLOBALES

Si a et b sont deux objets de la même classe, l'instruction :

```
b = a ;
```

Recopie l'ensemble des valeurs des champs de l'objet a dans ceux de l'objet b, sans se soucié des qualificatifs de ces champs public ou private.

On verra, dans une étape ultérieure, que cette copie fait appel à un constructeur spécialisé appelé constructeur par copie par défaut (implicite) et que ce dernier opère une copie superficielle, qui s'avère insuffisante dans le cas d'objets disposants de membres dynamiques.

3. LES CONSTRUCTEURS (CONSTRUCTOR)

3. 1. PRESENTATION

Avant de parler du constructeur proprement dit, voyons comment ça marche l'instanciation d'un objet classe. Considérons pour cela le code suivant :

```
Date D ;
```

```
D.Set_Date(28, 2, 2018) ;
```

La première instruction opère une instanciation de la classe Date dans l'objet D. Cette instanciation implique la construction de l'objet D. Pour cela le compilateur effectue les opérations suivantes:

- Réservation de l'espace mémoire nécessaire à la représentation de l'objet ;
- Implantation, dans l'espace ainsi réservé, des données membres définies par la classe. Si les objets sont statiques, leurs données se voient initialisées à zéros. Si les objets sont automatiques leurs données restent indéterminées. Si les membres sont eux-mêmes des objets, les mécanismes d'instanciation des classes correspondantes seront appelés à leur tour.

A ce niveau, signalons que chaque objet a ***ses propres données*** mais ***les méthodes sont communes à tous les objets***. En effet, si les méthodes sont également instanciées pour chaque objet ça aurait été un gaspillage de mémoire non justifié.

Une telle instanciation s'appelle ***construction*** et elle est exécutée par une méthode membre implicite qui s'appelle ***constructeur par défaut***. A défaut d'existence d'un constructeur explicite, qu'on va voir dans les paragraphes suivants, toute classe est automatiquement équipée d'un constructeur par défaut automatiquement invoqué à chaque instanciation, quelque soit la classe d'allocation mémoire de l'objet : statique, automatique ou dynamique. Maintenant, si on veut équiper une classe d'un constructeur explicite, il faut respecter les points suivants:

- C'est une méthode membre, figurant obligatoirement dans la section public:
- Cette méthode ne spécifie aucun type de retour même pas void ;
- Son identificateur est le celui de sa classe.

Cette méthode est appelée constructeur parce qu'elle spécifie une manière de construire une instance de la classe. Voici comment peut-on inclure un constructeur explicite dans notre classe Date.

```
class  Date
{
    public :
        Date(int day, int month, int year)
        {                                          // Cette méthode membre est le constructeur
            Day = day; Month = month; Year = year;     // parce qu'elle porte le même nom que la classe
        }
        void Display_Date() { cout <<  Day << '/' << Month <<  '/' << Year << « \n » ; }
    private :
        int  Day, Month, Year;
```

```
};
```

Ainsi quand l'instruction :

```
Date D(28, 2, 2018);
```

Est exécutée, le constructeur est appelé automatiquement et les entiers 28, 2 et 2018 sont passés aux membres privés de l'instance D. Deux remarques sont à soulever quant à l'écriture D(28, 2, 2018) :

- D'abord cette écriture est bien différente de l'écriture D={28, 2, 2018}, qui procède à un accès direct pour l'assignement des données membres, et qui viole le principe d'encapsulation du moment que les données manipulées sont déclarés comme private;
- Ensuite, cette écriture fait penser, de part sa syntaxe, à un appel de fonction au nom D avec les valeurs 28, 2 et 2018. C'est vrai sauf que le nom de la fonction appelé n'est pas l'objet D mais son constructeur Data. En effet D(28, 2, 2018) est un appel à la méthode membre qui est le constructeur de l'objet. Le constructeur effectue bien le travail qu'effectuait la méthode membre Set_Date.

3. 2. ROLE DU CONSTRUCTEUR

Un constructeur, comme son nom l'indique, a pour rôle de construire l'objet. Cette construction ne se limite pas à l'initialisation de l'objet, comme on l'a vue jusqu'à présent, mais le travail que peut accomplir un constructeur peut être beaucoup plus élaboré. Voici un programme exploitant une classe nommée somme_comulee comportant un constructeur qui remplie un tableau, membre donnée, par des nombres correspondants aux sommes cumulées des entiers, à partir de l'entier qu'on lui passe en argument (la somme cumulée jusqu'à 5 est 1+2+3+4+5).

```
class  somme_comulee
{
        int    tab[10] ;                          // le qualificatif private est sous entend
    public :
        somme_comulee(int );
        void  affiche() ;
};
somme_comulee::somme_comulee(int x)
{
    int i=0 ;
    for(int j = x ;  j < x+10  ; j++)
    {
        int somme = 0;
        for(int k = 1; k <= j ; k++)
            somme += k ;
```

```
            tab[i] = somme;
            i++;
        }
    }
    void somme_comulee::affiche()
    {
        for(int i=0 ;  j<10  ; i++)
            cout << tab[i] << " " ;
        cout << "\n" ;
    }
    main()
    {
        somme_comulee sum(4);
        sum.affiche() ;
    }
```

L'exécution de ce programme affiche :

10 15 21 28 36 45 55 66 78 91

3. 3. SURCHARGE DE CONSTRUCTEURS

Un constructeur a pour mission de construire l'objet d'une certaine manière. Peut-on alors construire cet objet de plusieurs manières ? La réponse est oui. En effet, comme nous l'avons déjà vu, CPP autorise la surcharge des identificateurs. Il autorise ainsi la définition de plusieurs fonctions avec le même identificateur à conditions que leurs *signatures* soient distinctes : On appel signature d'une fonction, son identificateur suivi du type de chacun de ses paramètres dans l'ordre de leur apparition, pendant le prototypage de cette fonction. On peut donc équiper une classe avec plusieurs constructeurs de signatures différentes en exploitant ce principe de surcharge des identificateurs. Appliquons cette notion sur notre classe Date.

```
class  Date
{
    public :
        Date()                                          // première forme du constructeur
        {
            Day = 28 ; Month = 2 ; Year = 2018;
        }
        Date(int day)                                   // deuxième forme du constructeur
        {
            Day = day; Month = 2 ; Year = 2018;
        }
        Date(int day, int month)                        // troisième forme du constructeur
        {
            Day = day; Mois = month; Year = 2018;
        }
        Date(int day, int month, int year)              // quatrième forme du constructeur
        {
            Day = day; Month = month; Year = year;
```

```
    }
    void Display_Date() { cout <<  Day << '/' << Month <<  '/' << Year << « \n » ; }
private :
    int    Day, Month, Year;
} ;
```

Dans cet exemple, quatre formes du constructeur sont données. L'appel de l'un d'entre eux est dirigé par la signature au moment de l'appel.

C'est juste pour montrer le principe d'utilisation de plusieurs constructeurs qu'on s'est permis une telle écriture. En effet, par utilisation du mécanisme de fonctions au nombre d'arguments variables, les quatre constructeurs ci-dessus peuvent être rassemblés en un seul comme suit.

```
class  Date
{
public :
    Date(int day = 28, int month = 2, int year = 2018)      // unique constructor
    {
        Day = day; Month = month; Year = year;
    }
    void Display_Date()  {cout <<  Day << '/' << Month <<  '/' << Year << « \n » ;}
private :
    int  Day, Month, Year;
} ;
```

4. LE DESTRUCTEUR DE CLASSE

5. 1. DESTRUCTEUR IMPLICITE (PAR DEFAULT) : PRESENTATION

Depuis le début de notre discussion sur les classes, on ne parlait que des techniques de construire, c'est à dire donner vies aux objets de ce type. Mais qu'en est-il de la destruction, c'est-à-dire la mort de ces objets. Comme tout objet élémentaire (int, float,…) :

- la durée de vie (la portée) d'un objet de type classe est limitée au bloc au sein duquel il a pris naissance ;
- L'objet vie donc tant qu'il est à l'intérieur de ce bloc. A la sortie de cet espace un mécanisme de destruction automatique, *destructeur par défaut*, est exécuté pour mettre fin à la vie de cet objet ;
- Au même titre qu'un constructeur par défaut et sauf définition explicite d'un destructeur, chaque classe dispose d'un destructeur par défaut offert par le système, qui effectue les opérations suivantes :
  - Suppression des données membres qui sont de type élémentaire ;
  - Appel du destructeur de la classe correspondante pour chaque donnée membre qui est de type objet (class).

☞ Pour des simplifications de notations, et sans que ça soit une source de confusion, nous appellerons désormais notre classe Date_Explicite simplement Date.

Pour mettre en évidence le rôle du destructeur par défaut, reprenons notre classe Date, sans faire apparaître aucun destructeur qu'il soit. Elle a la forma suivante:

```
class  Date
{
    public :
        Date(int day = 28, int month = 2, int year = 2018)    // constructeur unique
        {
            if(day <1 || day > 30 || month <1 || month > 12 || year < 0)
                return(1) ;

            Day = day; Month = month; Year = year;

            cout << "Constructeur  Explicite : " << this << "  Date : " << Day << "\t" << Month << "\t "
                                                    << Year << "\n ";
        }
    private :
        int  Day, Month, Year;
} ;
main()
{
    {
1.      Date D(3, 3, 2018) ;                    // l'objet D existe  uniquement dans ce bloc
    }
2.  D(5, 6, 2018) ;                             // faux parce que l'objet D est détruit à la sortie du bloc
}
```

La compilation de ce programme génère l'erreur 'D' was not declared in this scope. Cette erreur est due à l'instruction 2 du programme main, parce que l'instruction 1 est toute correcte, et la raison en est qu'à la sortie du bloc de l'instruction 1, l'objet D n'existe plus. Il est détruit par le compilateur, parce que son scope est justement limite à ce bloc.

☞ Notez que malgré qu'aucun destructeur explicite n'est défini, le système à équipé la classe par un destructeur par défaut et c'est celui la qui à détruit l'objet D à la sortie du bloc.

5. 2. INSUFFISANCE DU DESTRUCTEUR PAR DEFAUT

Le destructeur par défaut est il suffisant ? Pour répondre à cette question considérons de nouveau notre classe Date, toujours dépourvue d'un destructeur explicite, qu'on a :

- Enrichi d'un membre private P, de type pointeur vers des chaines de caractères (string). Le nombre de ces chaînes est deux et la réservation de l'espace mémoire correspondant est opérée au niveau du constructeur explicite;

```
class  Date
{
    public :
```

```
        Date(int day = 28, int month = 2, int year = 2018)
        {                                                    // unique constructor
            Day = day; Month = month; Year = year;
            if(day <1 || day > 30 || month <1 || month > 12 || year < 0)
                return(1) ;
            P = new string[2] ; if(P == NULL) exit() ;
        }
    private :
        int  Day, Month, Year;
        string  *P;

};
main()
{
    {
        Date D;
    }
    // suite du programme
}
```

L'exécution de l'instruction `Date D` a les conséquences suivantes:

1. L'objet `D` est crée (construit) par appel au constructeur par défaut (sans paramètres). Les valeurs des trois paramètres utilisés pendant cette construction sont celles par défaut : `28`, `2` et `2018`. Se sont ces valeurs qui vont servir pour l'initialisation des trois données membres `private : Day, Month` et `Year`;
2. Ce même constructeur procède à la réservation dynamique d'un espace mémoire de taille `=2 (sizeof(string))` et affecte leurs adresses au pointeur, membre `private P`.

Après cette construction, on sort du bloc qui contient l'objet `D`. Le ***destructeur par défaut*** est alors appelé et tous les membres de type élémentaires se trouvent détruits. Il s'agit des membres `private` simple `Day`, `Month`, `Year` et le pointeur `P`. C'est dans cette dernière destruction que réside tout le problème :

☞ L'espace réservé, dynamiquement pour le pointeur `P`, lors de l'instanciation de `D`, subsiste toujours après la sortie du bloc. Il est toujours considéré par le système comme réservé. Cet espace n'a plus aucune chance d'être libéré, puisque le pointeur qui le désigne vient d'être détruit à la sortie du bloc. C'est ce qu'on appel ***fuite de mémoire***.

A la lumière de cet exemple, on voit bien que le destructeur implicite n'est pas suffisant. En effet, les allocations dynamiques effectuées dans une classe doivent être correctement gérées par le code de cette même classe. En particulier, les réservations de mémoires qui sont faites à l'instanciation doivent être libérées lors de la destruction : le programmeur doit, dans

ce cas, remplacer le mécanisme de destruction par défaut en définissant un destructeur explicite.

5. 3. DESTRUCTEUR EXPLICITE

Comme son nom l'indique, un destructeur explicite est une méthode membre, appelée automatiquement à chaque destruction d'un objet de sa classe. Ce destructeur est défini par le programmeur et explicite la manière de détruire l'objet. Ses caractéristiques sont :

- Un destructeur est une méthode membre, logée obligatoirement dans la partie public de la classe;
- Son identificateur est formé du caractère **~** immédiatement suivi de l'identificateur de sa classe;
- Il n'accepte aucun paramètre. En effet, le destructeur n'est jamais appelé explicitement par le programmeur : son exécution est déclenchée automatiquement quand un objet, défini localement dans un bloc, doit être détruit à la sortie de ce bloc;
- Il n'indique aucun type de retour (pas même void);
- Il ne peut y avoir qu'un seul destructeur. En effet, on peut définir plusieurs constructeurs pour une même classe, car il peut y avoir plusieurs façons de construire un objet, mais il n'y a qu'une seule façon de détruire cet objet.

☞ Bien sûr si on spécifie explicitement un destructeur, le destructeur par défaut est inhibé.

Pour la suite, nous considérons toujours notre classe Date, qui vient d'être enrichie ci-dessus, qu'on va encore enrichir avec un destructeur explicite ce qui donne :

EXEMPLE :

```
class  Date
{
    public :
        Date(int day = 28, int month = 2, int year = 2018)
        {
            Day = day; Month = month; Year = year;
            if(day <1 || day > 30 || month <1 || month > 12 || year < 0)
                return(1) ;
            P = new string [2] ; if(P == NULL) exit() ;
        }
        ~Date()
        {
            delete [] P ;                    // les crochets [] sont obligatoire pour effacer un tableau
        }                                    //  crée par new: int *pt= new int[20] ⇒ delete[] pt
    private :
        int  Day, Month, Year;
        string  *P ;
```

```
};
main()
{
    {
        Date D;
    }
    // suite du programme
}
```

Cette fois ci, à la sortie du bloc de l'objet D, le destructeur explicite est appelé. Il procède à la libération de l'espace mémoire alloué pendant l'instanciation de l'objet. Cette espace est rendu donc disponible pour une autre utilisation.

☞ Se rappelé donc que chaque fois qu'une classe définit un ou plusieurs membres qui sont initialisés par allocation dynamique, il faut pour cette classe, définir explicitement le destructeur approprié.