Chapitre 2

PROGRAMMATION DES CLASSES

2.1 Un exemple

(2.1.1) Voici la déclaration et la définition d'une classe `Complexe` décrivant les nombres complexes, et un programme qui en montre l'utilisation.

```
// ------------------------ complexe.h ------------------------
//            déclaration de la classe Complexe

class Complexe
{
public:
    Complexe(float x, float y);       // premier constructeur de la classe :
                                      // fixe la partie réelle à x, la partie imaginaire à y
    Complexe();                       // second constructeur de la classe :
                                      // initialise un nombre complexe à 0
    void Lis();                       // lit un nombre complexe entré au clavier
    void Affiche();                   // affiche un nombre complexe
    Complexe operator+(Complexe g);   // surcharge de l'opérateur d'addition +
private:
    float re, im;                     // parties réelle et imaginaire
};



// ------------------------ complexe.cpp ----------------------
//              définition de la classe Complexe

#include <iostream.h>          // pour les entrées-sorties
#include "complexe.h"          // déclaration de la classe Complexe

Complexe::Complexe(float x, float y)   // constructeur avec paramètres
{
    re = x;
    im = y;
}

Complexe::Complexe()                   // constructeur sans paramètre
{
    re = 0.0;
    im = 0.0;
}

void Complexe::Lis()                   // lecture d'un complexe
{
    cout << "Partie réelle ?  ";
    cin >> re;
    cout << "Partie imaginaire ?  ";
    cin >> im;
}

void Complexe::Affiche()               // affichage d'un complexe
{
    cout << re << " + i " << im;
}

Complexe Complexe::operator+(Complexe g)       // surcharge de l'opérateur +
{
```

```
    return Complexe(re + g.re, im + g.im);    // appel du constructeur
}



// ------------------------ usage.cpp ---------------------
//         exemple d'utilisation de la classe Complexe

#include <iostream.h>          // pour les entrées-sorties
#include "complexe.h"          // pour la déclaration de la classe Complexe

void main()                    // traitement principal
{
    Complexe z1(0.0, 1.0);     // appel implicite du constructeur paramétré
    Complexe z2;               // appel implicite du constructeur non paramétré

    z1.Affiche();              // affichage de z1
    cout << "\nEntrer un nombre complexe : ";
    z2.Lis();                  // saisie de z2
    cout << "\nVous avez entré : ";
    z2.Affiche();              // affichage de z2

    Complexe z3 = z1 + z2;     // somme de deux complexes grâce à l'opérateur +
    cout << "\n\nLa somme de ";
    z1.Affiche();
    cout << " et ";
    z2.Affiche();
    cout << " est ";
    z3.Affiche();
}
```

(2.1.2) **Remarques**

Les constructeurs permettent d'initialiser les objets. Nous verrons plus précisément leur usage au paragraphe 3.

Nous reviendrons également sur la surcharge des opérateurs (paragraphe 4). Dans ce programme, nous donnons l'exemple de l'opérateur `+` qui est redéfini pour permettre d'additionner deux nombres complexes. Cela permet ensuite d'écrire tout simplement `z3 = z1 + z2` entre nombre complexes. Cette possibilité de redéfinir (on dit aussi *surcharger*) les opérateurs usuels du langage est un des traits importants du C++.

2.2 Fonctions-membres

(2.2.1) **L'objet implicite**

Rappelons que pour décrire une classe (cf (1.2.5)), on commence par déclarer les données et fonctions-membres d'un objet de cette classe, puis on définit les fonctions-membres de ce même objet. Cet objet n'est jamais nommé, il est *implicite* (au besoin, on peut y faire référence en le désignant par `*this`).

Ainsi dans l'exemple du paragraphe (2.1.1), lorsqu'on écrit les définitions des fonctions-membres de la classe `Complexe`, on se réfère directement aux variables `re` et `im`, et ces variables sont les données-membres du nombre complexe implicite qu'on est en train de programmer et qui n'est jamais nommé. Mais s'il y a un autre nombre complexe, comme `g` dans la définition de la fonction `operator+`, les données-membres de l'objet `g` sont désignées par la notation pointée habituelle, à savoir `g.re` et `g.im` (1.5.2). Notons au passage que, bien que ces données soient privées, elles sont accessibles à ce niveau puisque nous sommes dans la définition de la classe `Complexe`.

(2.2.2) **Flux de l'information**

Chaque fonction-membre est une unité de traitement correspondant à une fonctionnalité bien précise et qui sera propre à tous les objets de la classe.

Pour faire son travail lors d'un appel, cette unité de traitement dispose des informations suivantes :
- les valeurs des données-membre (publiques ou privées) de l'objet auquel elle appartient,

– les valeurs des paramètres qui lui sont transmises.

En retour, elle fournit un résultat qui pourra être utilisé après l'appel. Ainsi :

Avant de programmer une fonction-membre, il faudra identifier quelle est l'information qui doit y entrer (paramètres) et celle qui doit en sortir (résultat).

2.3 Constructeurs et destructeurs

(2.3.1) *Un constructeur est une fonction-membre déclarée du même nom que la classe, et sans type :*

`Nom_classe(`<*paramètres*>`);`

Fonctionnement : à l'exécution, l'appel au constructeur produit un nouvel objet de la classe, dont on peut prévoir l'initialisation des données-membres dans la définition du constructeur.

Exemple : avec la classe `Complexe` décrite en (2.1.1), l'expression `Complexe(1.0, 2.0)` a pour valeur un nombre complexe de partie réelle 1 et de partie imaginaire 2.

Dans une classe, il peut y avoir plusieurs constructeurs à condition qu'ils diffèrent par le nombre ou le type des paramètres. Un constructeur sans paramètre s'appelle *constructeur par défaut.*

(2.3.2) **Initialisation des objets**

Dans une classe, il est possible ne pas mettre de constructeur. Dans ce cas, lors de la déclaration d'une variable de cette classe, l'espace mémoire est réservé mais les données-membres de l'objet ne reçoivent pas de valeur de départ : on dit qu'elles ne sont pas *initialisées.* Au besoin, on peut prévoir une fonction-membre publique pour faire cette initialisation. En revanche :

S'il y a un constructeur, il est automatiquement appelé lors de la déclaration d'une variable de la classe.

Exemples avec la classe `Complexe` déclarée en (2.1.1) :

```
Complexe z;            // appel automatique du constructeur par défaut
                       // équivaut à : Complexe z = Complexe();

Complexe z (1.0, 2.0);           // appel du constructeur paramétré
                       // équivaut à : Complexe z = Complexe(1.0, 2.0);
```

On retiendra que :

L'utilité principale du constructeur est d'effectuer des initialisations pour chaque objet nouvellement créé.

(2.3.3) **Initialisations en chaîne**

Si une classe `Class_A` contient des données-membres qui sont des objets d'une classe `Class_B`, par exemple :

```
class Class_A
{
public:
       Class_A(...);          // constructeur
       ...
private:
       Class_B  b1, b2;       // deux objets de la classe Class_B
       ...
};
```

alors, à la création d'un objet de la classe `Class_A`, le constructeur par défaut de `Class_B` (s'il existe) est automatiquement appelé pour chacun des objets `b1`, `b2` : on dit qu'il y a des *initialisations en chaîne.*

Mais pour ces initialisations, il est également possible de faire appel à un constructeur paramétré de `Class_B`, à condition de définir le constructeur de `Class_A` de la manière suivante :

`Class_A :: Class_A(...)  :  b1 (...), b2 (...)`
<*instruction-bloc*>

Dans ce cas, l'appel au constructeur de `Class_A` provoquera l'initialisation des données-membres `b1`, `b2` (par appel au constructeur paramétré de `Class_B`) *avant* l'exécution de l'<*instruction-bloc*>.

(2.3.4) **Conversion de type**

Supposons que `Ma_classe` comporte un constructeur à un paramètre de la forme :

```
Ma_classe(Mon_type x);
```

où `Mon_type` est un type quelconque.

Alors, chaque fois que le besoin s'en fait sentir, ce constructeur assure la conversion automatique d'une expression `e` de type `Mon_type` en un objet de type `Ma_classe` (à savoir `Ma_classe(e)`).

Par exemple, si nous avons dans la classe `Complexe` le constructeur suivant :

```
Complexe::Complexe(float x)
{
       re = x;
       im = 0.0;
}
```

alors ce constructeur assure la conversion automatique `float` → `Complexe`, ce qui nous permet d'écrire des instructions du genre :

```
z1 = 1.0;
z3 = z2 + 2.0;
```

(`z1`, `z2`, `z3` supposés de type `Complexe`).

2.4 Surcharge des opérateurs

(2.4.1) En C++, on peut *surcharger* la plupart des opérateurs usuels du langage, c'est-à-dire les reprogrammer pour que, dans un certain contexte, ils fassent autre chose que ce qu'ils font d'habitude. Ainsi dans l'exemple (2.1.1), nous avons surchargé l'opérateur d'addition `+` pour pouvoir l'appliquer à deux nombres complexes et calculer leur somme.

Notons également que les opérateurs d'entrées-sorties `<<` et `>>` sont en réalité les surcharges de deux opérateurs de décalages de bits (appelés respectivement "shift left" et "shift right").

La surcharge d'un opérateur <*op*> se fait en déclarant, au sein d'une classe `Ma_classe`, une fonction-membre appelée `operator` <*op*>. Plusieurs cas peuvent se présenter, selon que <*op*> est un opérateur *unaire* (c'est-à-dire à un argument) ou *binaire* (c'est-a-dire à deux arguments). Nous allons voir quelques exemples.

(2.4.2) **Cas d'un opérateur unaire**

Nous voulons surcharger l'opérateur unaire `-` pour qu'il calcule l'opposé d'un nombre complexe. Dans la classe `Complexe` décrite en (2.1.1), nous déclarons la fonction-membre publique suivante :

```
Complexe operator-();
```

que nous définissons ensuite en utilisant le constructeur paramétré de la classe :

```
Complexe Complexe::operator-()
{
        return Complexe(-re, -im);
}
```

Par la suite, si `z` est une variable de type `Complexe`, on pourra écrire tout simplement l'expression `-z` pour désigner l'opposé de `z`, sachant que cette expression est équivalente à l'expression `z.operator-()` (message `operator-` destiné à `z`).

(2.4.3) **Cas d'un opérateur binaire**

Nous voulons surcharger l'opérateur binaire `-` pour qu'il calcule la différence de deux nombres complexes. Dans la même classe `Complexe`, nous déclarons la fonction-membre publique suivante :

```
Complexe operator-(Complexe u);
```

que nous définissons ensuite en utilisant également le constructeur paramétré de la classe :

```
Complexe Complexe::operator-(Complexe u)
{
        return Complexe(re - u.re, im - u.im);
}
```

Par la suite, si `z1` et `z2` sont deux variables de type `Complexe`, on pourra écrire tout simplement l'expression `z1 - z2` pour désigner le nombre complexe obtenu en soustrayant `z2` de `z1`, sachant que cette expression est équivalente à l'expression `z1.operator-(z2)` (message `operator-` destiné à `z1`, appliqué avec le paramètre d'entrée `z2`).

(2.4.4) **Autre cas d'un opérateur binaire**.

Cette fois, nous désirons définir un opérateur qui, appliqué à deux objets d'une même classe, donne une valeur d'un type différent. Ce cas est plus compliqué que le précédent.

On considère la classe "culinaire" suivante :

```
class Plat                    // décrit un plat proposé au menu d'un restaurant
{
public:
        float Getprix();      // fonction d'accès donnant le prix : voir (1.3.2)
private:
        char nom[20];         // nom du plat
        float prix;           // et son prix
}
```

Nous voulons surcharger l'opérateur `+` pour qu'en écrivant par exemple `poulet + fromage`, cela donne le prix total des deux plats (`poulet` et `fromage` supposés de type `Plat`).

Nous commençons par déclarer la fonction-membre :

```
float operator+(Plat p);
```

que nous définissons par :

```
float Plat::operator+(Plat p)
{
        return  prix + p.Getprix();
}
```

et que nous pouvons ensuite utiliser en écrivant par exemple `poulet + fromage`. Nous définissons ainsi une loi d'addition `+` : (`Plat` × `Plat`) → `float`.

Mais que se passe-t-il si nous voulons calculer `salade + poulet + fromage` ? Par associativité, cette expression peut également s'écrire :

```
salade + (poulet + fromage)
(salade + poulet) + fromage
```

donc il nous faut définir deux autres lois :

- une loi `+` : (`Plat` × `float`) → `float`,
- une loi `+` : (`float` × `Plat`) → `float`.

La première se programme en déclarant une nouvelle fonction-membre :

```
float operator+(float u);
```

que nous définissons par :

```
float Plat::operator+(float u)
{
       return  prix + u;
}
```

La deuxième ne peut pas se programmer avec une fonction-membre de la classe `Plat` puisqu'elle s'adresse à un `float`. Nous sommes contraints de déclarer une fonction libre (c'est-à-dire hors de toute classe) :

```
float operator+(float u, Plat p);
```

que nous définissons par :

```
float operator+(float u, Plat p)
{
       return  u + p.Getprix();
}
```

2.5 Réalisation d'un programme

(2.5.1) Sur un exemple, nous allons détailler les différentes étapes qui mènent à la réalisation d'un programme. Il s'agira de simuler le jeu du "c'est plus, c'est moins" où un joueur tente de deviner un nombre choisi par le meneur de jeu.

Le fait de programmer avec des objets nous force à modéliser soigneusement notre application avant d'aborder le codage en C++.

(2.5.2) 1ère étape : identification des classes

Conformément à (1.1.3), nous commençons par décrire le jeu de manière littérale :

> "Le jeu oppose un joueur à un meneur. Le meneur choisit un numéro secret (entre 1 et 100). Le joueur propose un nombre. Le meneur répond par : "c'est plus", "c'est moins" ou "c'est exact". Si le joueur trouve le numéro secret en six essais maximum, il gagne, sinon il perd."

Le jeu réunit deux acteurs avec des rôles différents : un meneur et un joueur. Nous définirons donc deux classes : une classe `Meneur` et une classe `Joueur`.

(2.5.3) 2ème étape : fiches descriptives des classes

Il nous faut déterminer les données et fonctions-membres de chaque classe.

Le meneur détient un numéro secret. Ses actions sont :
- choisir ce numéro,
- répondre par un diagnostic ("c'est plus", "c'est moins" ou "c'est exact").

D'où la fiche descriptive suivante :

classe : `Meneur`	
privé :	*public :*
`numsecret`	`Choisis`
	`Reponds`

Le joueur détient un nombre (sa proposition). Son unique action est de proposer ce nombre. Mais au cours du jeu, il doit garder à l'esprit une fourchette dans laquelle se situe le numéro à deviner, ce qui nous amène à la fiche descriptive suivante :

classe : `Joueur`	
privé :	*public :*
`proposition`	`Propose`
`min`	
`max`	

(2.5.4) 3ème étape : description détaillée des fonctions-membres

Nous allons décrire le fonctionnement de chaque fonction-membre et en préciser les informations d'entrée et de sortie (voir (2.2.2)).

`Choisis` (de `Meneur`) :
- entrée : rien
- sortie : rien
- choisit la valeur de `numsecret`, entre 1 et 100. On remarque que ce choix ne se fait qu'une fois, au début de la partie. Il est donc logique que ce soit le constructeur qui s'en charge. Nous transformerons donc cette fonction en constructeur.

`Reponds` (de `Meneur`) :
- entrée : la proposition du joueur
- sortie : un diagnostic : "exact", "plus" ou "moins" (que nous coderons respectivement par 0, 1 ou 2)
- compare la proposition du joueur avec le numéro secret et rend son diagnostic.

`Propose` (de `Joueur`) :
- entrée : le diagnostic du précédent essai
- sortie : un nombre
- compte tenu des tentatives précédentes, émet une nouvelle proposition.

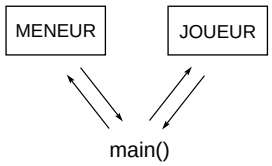
(2.5.5) 4ème étape : description du traitement principal

La fonction `main()` sera le chef-d'orchestre de la simulation. Son travail consiste à :

- déclarer un joueur et un meneur
- faire :
  - prendre la proposition du joueur
  - la transmettre au meneur
  - prendre le diagnostic du meneur
  - le transmettre au joueur

  jusqu'à la fin de la partie
- afficher le résultat

Remarquons que nos deux objets-acteurs ne communiquent entre eux que de manière indirecte, par l'intermédiaire de la fonction `main()` :



On pourrait mettre directement en rapport les objets entre eux, à l'aide de pointeurs (paragraphe 6).

(2.5.6) 5ème étape : déclaration des classes

Nous en arrivons à la programmation proprement dite. Nous commençons par écrire les fichiers de déclarations des classes `Meneur` et `Joueur` :

```
// ------------------------ meneur.h ------------------------
// ce fichier contient la déclaration de la classe Meneur

class Meneur
{
public:
    Meneur();                   // initialise un meneur
    int Reponds(int prop);      // reçoit la proposition du joueur
                                // renvoie 0 si c'est exact, 1 si c'est plus
                                // et 2 si c'est moins
private:
    int numsecret;              // numéro secret choisi au départ
};

// ------------------------ joueur.h ------------------------
// ce fichier contient la déclaration de la classe Joueur

class Joueur
{
public:
    Joueur();                   // initialise un joueur
    int Propose(int diag);      // reçoit le diagnostic du précédent essai
                                // renvoie une nouvelle proposition
private:
    int min, max,               // fourchette pour la recherche
        proposition;            // dernier nombre proposé
};
```

(2.5.7) 6ème étape : écriture du traitement principal

Ce fichier contient l'utilisation des classes.

```
// ------------------------ jeu.cpp ---------------------
// programme de simulation du jeu "c'est plus, c'est moins"

#include <iostream.h>           // pour les entrées-sorties
#include "joueur.h"             // pour la déclaration de la classe Joueur
#include "meneur.h"             // pour la déclaration de la classe Meneur

void main()                     // gère une partie ...
{
    Joueur j;                   // ...  avec un joueur ...
    Meneur m;                   // ...  et un meneur
    int p, d = 1,               // variables auxiliaires
        cpt = 0;                // nombre d'essais
    do                          // simulation du déroulement du jeu
    {
        p = j.Propose(d);       // proposition du joueur
        d = m.Reponds(p);       // diagnostic du meneur
        cpt++;
    }
    while (d && cpt < 6);
    if (d)                      // défaite du joueur
        cout << "\nLe joueur a perdu !";
    else                        // victoire du joueur
        cout << "\nLe joueur a gagné !";
}
```

Nous pourrions dès à présent compiler ce fichier `jeu.cpp`, alors que les classes `Joueur` et `Meneur` ne sont pas encore définies.

(2.5.8) 7ème étape : définition des classes

C'est l'ultime étape, pour laquelle nous envisagerons deux scénarios différents. Dans le premier, l'utilisateur du programme tiendra le rôle du joueur tandis que l'ordinateur tiendra le rôle du meneur. Dans le second, ce sera le contraire : l'utilisateur tiendra le rôle du meneur et l'ordinateur celui du joueur. Nous écrirons donc deux versions des classes `Meneur` et `Joueur`.

Première version :

```
// ------------------------ meneur.cpp ---------------------
// ce fichier contient la définition de la classe Meneur
// le rôle du meneur est tenu par l'ordinateur

#include <iostream.h>
#include <stdlib.h>
#include <time.h>                       // pour les nombres aléatoires
#include "meneur.h"

Meneur::Meneur()
{
    srand((unsigned) time(NULL));       // initialisation du générateur aléatoire
    numsecret = 1 + rand() % 100;       // choix du numéro secret
}

int Meneur::Reponds(int prop)           // prop = proposition du joueur
{
    if (prop < numsecret)
    {
        cout << "\nC'est plus";
        return 1;
    }
    if (prop > numsecret)
    {
        cout << "\nC'est moins";
        return 2;
    }
    cout << " \nC'est exact";
    return 0;
}

// ------------------------ joueur.cpp ---------------------
// ce fichier contient la définition de la classe Joueur
// le rôle du joueur est tenu par l'utilisateur du programme

#include <iostream.h>
#include "joueur.h"

Joueur::Joueur()
{
    cout << "\nBonjour !  Vous allez jouer le rôle du joueur.";
}

int Joueur::Propose(int diag)           // la valeur de diag est ignorée
{
    int p;
    cout << "\nProposition ?  ";
    cin >> p;
    return p;
}
```

Seconde version :

```
// ------------------------ meneur.cpp ---------------------
// ce fichier contient la définition de la classe Meneur
// le rôle du meneur est tenu par l'utilisateur du programme

#include <iostream.h>
#include "meneur.h"

Meneur::Meneur()
{
    cout << "\nBonjour !  Vous allez jouer le rôle du meneur.";
    cout << "\nChoisissez un numéro secret entre 1 et 100";
}

int Meneur::Reponds(int prop)           // la valeur de prop est ignorée
{
    int r;
    cout << "\n0 - C'est exact";
    cout << "\n1 - C'est plus";
    cout << "\n2 - C'est moins";
    cout << "\nVotre réponse (0,1,2) ?  ";
```

```
    cin >> r;
    return r;
}

// ------------------------ joueur.cpp ---------------------
// ce fichier contient la définition de la classe Joueur
// le rôle du joueur est tenu par l'ordinateur

#include <iostream.h>
#include "joueur.h"

Joueur::Joueur()
{
    min = 1;                          // fixe la fourchette dans laquelle
    max = 100;                        // se trouve le numéro à deviner
    proposition = 0;                  // première proposition fictive
}

int Joueur::Propose(int diag)
{
    if (diag == 1)                    // ajuste la fourchette
        min = proposition + 1;
    else
        max = proposition - 1;
    proposition = (min + max) / 2;    // procède par dichotomie
    cout << "\nJe propose :  " << proposition;
    return proposition;
}
```

Remarque.— Quand nous aurons abordé les notions d'héritage et de polymorphisme, nous serons en mesure d'écrire une version unique et beaucoup plus souple de ce programme, où la distribution des rôles pourra être décidée au moment de l'exécution.

2.6 Pointeurs et objets

(2.6.1) On peut naturellement utiliser des pointeurs sur des types classes. Nous pourrons ainsi utiliser des objets dynamiques, par exemple de la classe `Complexe` (cf (2.1.1)), en écrivant :

```
Complexe *pc = new Complexe;   // etc...
```

De la même manière qu'avec les objets statiques :

- si `new` est utilisé avec un type classe, le constructeur par défaut (s'il existe) est appelé automatiquement,
- il est possible de faire un appel explicite à un constructeur, par exemple :

```
Complexe *pc = new Complexe(1.0, 2.0);
```

(2.6.2) L'accès aux données et fonctions-membres d'un objet pointé peut se faire grâce à l'*opérateur flèche* `->`. Par exemple, avec les déclarations précédentes, on écrira :

`pc -> re`	plutôt que	`(*pc).re`
`pc -> Affiche()`	plutôt que	`(*pc).Affiche()`

(2.6.3) **Liens entre objets**

Supposons déclarées deux classes avec :

```
class A
{
public:
       machin truc(); // fonction-membre
.....
};
```

```
class B
{
private:
        A *pa;          // donnée-membre :  pointeur sur un objet de la classe A
.....
};
```

Comme un objet `obj_B` de la classe `B` contient un pointeur sur un objet de la classe `A`, cela permet à `obj_B` de communiquer directement avec cet objet en lui envoyant par exemple le message `pa -> truc()`. Ainsi :

En programmation-objet, l'utilité principale des pointeurs est de permettre à deux objets de communiquer directement entre eux.

On peut également construire des *listes chaînées* d'objets de la même classe, en déclarant :

```
class C
{
private:
        C *lien;        // donnée-membre :  pointeur sur un objet de la classe C
.....
};
```

Une telle liste peut être représentée par le schéma suivant :