

2. Définition de classes

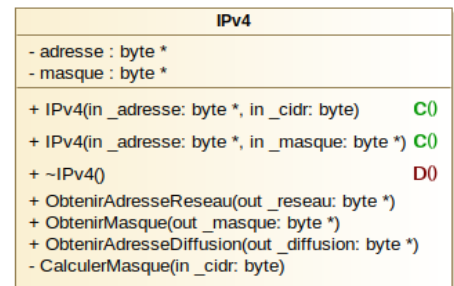
2.1. Introduction

La programmation objet consiste dans un premier temps à spécifier les différentes classes de l'application. D'une manière générale, un fichier `.h` dit « d'en-tête » contient la déclaration d'une classe, un deuxième fichier `.cpp` contient la définition des différentes méthodes.

2.2. Notion d'encapsulation

L'encapsulation représente l'idée de regrouper sous une même entité les attributs et les méthodes. Ce concept apporte une certaine intégrité aux **données membres** ou **attributs** qui ne peuvent être manipulés que par les **fonctions membres** de la classe ou **méthodes**. L'autre intérêt est la portabilité du code produit puisque les programmes et les données attachées sont encapsulés dans une même entité pour ne former qu'un tout. Cette vision de la programmation permet de modéliser plus facilement les objets du monde réel.

Comme exemple, on propose d'étudier la classe **IPv4**. Cette classe possède deux attributs privés, un signe `-` devant chaque attribut dans la représentation UML, les rend uniquement accessibles uniquement par les méthodes de la classe. Comme les adresses IPv4 peuvent être représentées de deux manières, elle dispose de deux constructeurs, le premier permet d'initialiser les attributs à partir d'un tableau d'octets contenant une adresse IP et de son CIDR, nombre de bits utilisés pour son masque associé, exemple : "192.168.1.1 /24". Le deuxième constructeur initialise les attributs sous la forme d'un tableau contenant l'adresse IP et un second tableau représentant le masque, soit par exemple : "192.168.1.1" et "255.255.255.0".



La classe possède un destructeur, l'utilisation de pointeurs parmi les attributs laisse entendre une allocation dynamique de la mémoire et donc une restitution de cette mémoire dans le destructeur.

Elle possède également une méthode privée permettant de calculer le masque à partir du CIDR, ici le /24. Il n'y a pas de raison pour que cette méthode soit accessible par un autre programme d'où le qualificatif **privé**, le signe `-` dans la représentation UML, devant la méthode **CalculerMasque**.

Par contre, il est possible d'obtenir l'adresse réseau, l'adresse de diffusion et éventuellement le masque du réseau d'une adresse IPv4. Ces trois dernières méthodes possèdent donc un qualificatif **public** correspondant au signe `+` d'UML. Elles sont donc visibles par tous les programmes qui utilisent cette classe.

Déclaration de la classe IPv4

Déclaration de la classe IPv4 :

ipv4.h

```
#ifndef __IPV4_H
#define __IPV4_H

class IPv4
{
private:
    unsigned char * adresse;
    unsigned char * masque ;
    void CalculerMasque(unsigned char _cidr);

public:
    IPv4(const unsigned char * _adresse, const unsigned char _cidr);
    IPv4(const unsigned char * _adresse, const unsigned char * _masque);
    ~IPv4();
    void ObtenirMasque(unsigned char * _masque);
    void ObtenirAdresseReseau(unsigned char * _reseau);
    void ObtenirAdresseDiffusion(unsigned char * _diffusion);
};

#endif
```

Voici quelques règles de qualité logicielle pour une programmation maintenable, efficace et compréhensible.

Codage C++	Explications
<code>#ifndef _IPV4_H</code> <code>#define _IPV4_H</code> ... <code>#endif</code>	Protège contre les inclusions multiples, la plupart des outils ajoutent ces lignes automatiquement lors de la création d'une classe C++.
<code>_IPV4_H</code>	Une constante ou définition est écrite en lettres majuscules, éventuellement séparées par des <code>_</code> .
<code>class IPv4</code>	Le nom d'une classe commence par une majuscule.
<code>private:</code>	Les attributs sont privés et ne sont donc pas modifiables en dehors de la classe.
<code>unsigned char * adresse;</code>	Les attributs et les variables d'une manière générale commencent par une minuscule.
<code>const unsigned char * _adresse</code>	Un paramètre passé à une fonction commencent par un <code>_</code> et porte le nom de l'attribut qu'il va initialiser.
<code>const</code>	Ce qualificateur indique que le paramètre est uniquement en entrée.
<code>unsigned char * _adresse</code>	Aurait pu être remplacé par <code>unsigned char _adresse[]</code> .
<code>void CalculerMasque();</code>	Le nom d'une méthode est un verbe à l'infinitif suivi d'un complément. La première lettre commence par une majuscule.
<code>ObtenirAdresseReseau</code>	Une majuscule sépare chaque mot différent dans un identifiant.

Toutes ces recommandations ne sont pas obligatoires, mais vivement recommandées. Un code, même écrit par plusieurs personnes, doit toujours suivre les mêmes règles de codage.

Implémentation de la classe IPv4

Cette première partie du code montre comment implémenter en C++ les deux constructeurs de la classe. Pour chacun d'eux, il y a une allocation mémoire pour l'adresse IP et le masque. Le destructeur restitue la mémoire allouée. La méthode privée **CalculerMasque** est uniquement appelée dans le premier constructeur afin d'initialiser l'attribut **masque**.

Codage des constructeurs, et du destructeur :

ipv4.cpp

```
#include "IPv4.h"

IPv4::IPv4(const unsigned char * _adresse, const unsigned char _cidr)
{
    adresse = new unsigned char [4];
    masque = new unsigned char [4];
    for(int indice = 0 ; indice < 4 ; indice++)
        adresse[indice] = _adresse[indice];
    if(_cidr <= 32)
        CalculerMasque(_cidr);
}

IPv4::IPv4(const unsigned char * _adresse, const unsigned char * _masque)
{
    adresse = new unsigned char [4];
    masque = new unsigned char [4];
    for(int indice = 0 ; indice < 4 ; indice++)
    {
        adresse[indice] = _adresse[indice];
        masque[indice] = _masque[indice];
    }
}

IPv4::~IPv4()
{
    delete [] adresse;
    delete [] masque ;
}
```

Le détail de la fonction est donnée ci-après :

Codage de la fonction CalculerMasque

ipv4.cpp

```
void IPV4::CalculerMasque(unsigned char _cidr)
{
    int indice ;
    // Le masque est remis à 0 -> 0.0.0.0
    for(indice = 0 ; indice < 4 ; indice++)
        masque[indice] = 0 ;

    indice = 0;
    // tant que le cidr est un multiple de 8
    while(_cidr >= 8)
    {
        masque[indice++] = 255 ;
        _cidr -= 8 ;
    }

    // Complément pour la fin du cidr (<8)
    unsigned char puissance = 128 ;
    while(_cidr-- > 0) // Après le test la variable _cidr est décrémentée
    { // les puissances de 2 sont ajoutées à l'octet par valeur décroissante
        masque[indice] += puissance ;
        puissance /=2 ;
    }
}
```

Pour la suite du code, chaque méthode **Obtenir...** possède un paramètre de sortie, le choix, ici, est d'utiliser un pointeur, chaque fonction reçoit le nom d'un tableau (donc un pointeur constant) déclaré dans le programme principal. Ainsi, la mise à jour est automatique dans le programme appelant.

La méthode **ObtenirMasque** ne fait que recopier le masque dans le tableau passé en paramètre.

Codage de la méthode ObtenirMasque :

ipv4.cpp

```
void IPV4::ObtenirMasque(unsigned char * _masque)
{
    for(int indice = 0 ; indice < 4 ; indice++)
        _masque[indice] = masque[indice];
}
```

Pour obtenir une adresse réseau, on rappelle ici, qu'il s'agit de faire un **ET** binaire entre chaque bit de l'adresse IP et le masque de réseau. La fonction se code de la manière suivante :

Codage de la méthode ObtenirAdresseReseau :

ipv4.cpp

```
void IPV4::ObtenirAdresseReseau(unsigned char * _reseau)
{
    for(int indice = 0 ; indice < 4 ; indice++)
        _reseau[indice] = adresse[indice] & masque[indice] ;
}
```

ET	192	168	1	1	Adresse IP
	1100 0000	1010 1000	0000 0001	0000 0001	
soit	255	255	255	0	Masque
	1111 1111	1111 1111	1111 1111	0000 0000	
	1100 0000	1010 1000	0000 0001	0000 0000	Adresse réseau
	192	168	1	0	

L'opération **ET** binaire est effectuée bit à bit : 1 **ET** 1 = 1 sinon 0

Pour obtenir l'adresse de diffusion, il s'agit de compléter l'adresse réseau en plaçant le reste des bits à 1. On peut compléter le masque du réseau et faire un **OU** binaire avec l'adresse réseau.

Codage de la méthode ObtenirAdresseDiffusion :

ipv4.cpp

```
void IPv4::ObtenirAdresseDiffusion(unsigned char *_diffusion)
{
    unsigned char adresseDuReseau[4];
    ObtenirAdresseReseau(adresseDuReseau);
    for(int indice = 0 ; indice < 4 ; indice++)
        _diffusion[indice] = adresseDuReseau[indice] | ~masque[indice] ;
}
```

OU	192	168	1	0	Adresse réseau
	1100 0000	1010 1000	0000 0001	0000 0000	
soit	0	0	0	255	Masque
	0000 0000	0000 0000	0000 0000	1111 1111	
	1100 0000	1010 1000	0000 0001	0000 0000	Adresse diffusion
	192	168	1	255	

L'opération **OU** binaire est effectuée bit à bit : 0 **OU** 0 = 0 sinon 1

Exercice d'application

1. La classe IPv4 peut être complétée par une méthode fournissant : l'adresse de la première machine du réseau, une autre fournissant celle de la dernière et éventuellement une troisième avec un paramètre de retour retournant le nombre de machines dans le réseau.

Vérification en ligne du résultat : <http://cric.grenoble.cnrs.fr/Administrateurs/Outils/CalculMasque/>

Pour tester cette classe, on propose le programme principal suivant :

Programme principal :

reseau.cpp

```
#include <iostream>
#include "IPv4.h"
using namespace std;

void AfficherTableau(unsigned char *tab);

int main()
{
    unsigned char adresse[4]= {192,168,1,1};
    unsigned char masque[4];
    unsigned char reseau[4];
    unsigned char diffusion[4];

    IPv4 uneAdresse(adresse, 24); // instanciation de la classe IPv4

    cout << "Adresse IPv4 : ";
    AfficherTableau(adresse);
    uneAdresse.ObtenirMasque(masque); // appel d'une méthode
    cout << "Masque : ";
    AfficherTableau(masque);
    uneAdresse.ObtenirAdresseReseau(reseau);
    cout << "Réseau : ";
    AfficherTableau(reseau);
    uneAdresse.ObtenirAdresseDiffusion(diffusion);
    cout << "Diffusion : ";
    AfficherTableau(diffusion);

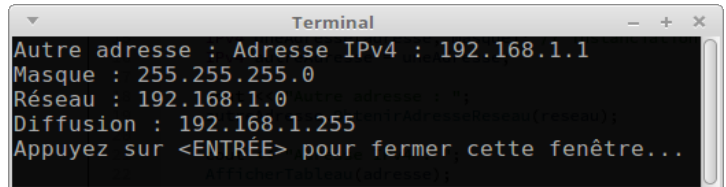
    return 0;
}
```

Cette fonction évite la répétition de code pour l'affichage

Fonction AfficherTableau

reseau.cpp

```
void AfficherTableau(unsigned char *tab)
{
    for(int indice=0 ; indice < 4 ; indice ++ )
    {
        cout << static_cast<int> (tab[indice]);
        if(indice < 3)
            cout << "." ;
    }
    cout << endl;
}
```



2.3. Modèle canonique dit Coplien

La forme canonique, dite de **Coplien**, définit un cadre de base à respecter pour les classes non triviales dont certains attributs peuvent être alloués dynamiquement.

Pour répondre à ce critère, une classe possède une forme canonique ou forme normale ou encore forme standard, si elle présente les méthodes suivantes :

- Un constructeur par défaut,
- Un constructeur de copie,
- Un destructeur éventuellement virtuel
- La surcharge de l'opérateur d'affectation

Dans notre exemple précédent, pour la classe **IPv4**, ce n'est pas le cas. Notre classe ne possède pas de constructeur par défaut c'est-à-dire sans paramètre. Elle ne possède pas non plus de constructeur de copie, et l'opérateur d'affectation n'est pas surchargé. Elle possède bien un destructeur. Par rapport à notre exemple de programme principal, cela était sans importance, le résultat obtenu est bien celui attendu. Maintenant, pour des traitements particuliers cela peut s'avérer problématique.

Pour illustrer ce propos, voici un nouveau programme principal, il déclare deux instances de la classe IPv4, la première est allouée dynamiquement, la seconde est une copie de la première. Ensuite, à partir de la copie, l'adresse réseau est affichée, avant et après destruction de l'instance de la première.

Nouveau programme principal :

reseau2.cpp

```
#include <iostream>
#include "IPv4.h"
using namespace std;

void AfficherTableau(const unsigned char *tab);

int main()
{
    unsigned char adresse[4]= {192,168,1,1};
    unsigned char reseau[4];

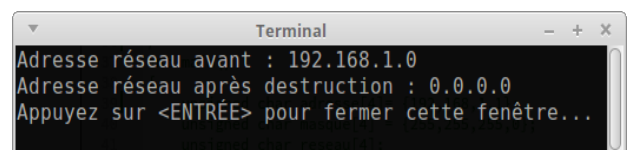
    IPv4 * uneAdresse = new IPv4(adresse, 24); // instantiation de la classe IPv4
    IPv4 adresseCopie = *uneAdresse; // l'instance est recopiée dans une autre

    cout << "Adresse réseau avant : ";
    adresseCopie.ObtenirAdresseReseau(reseau);
    AfficherTableau(reseau);

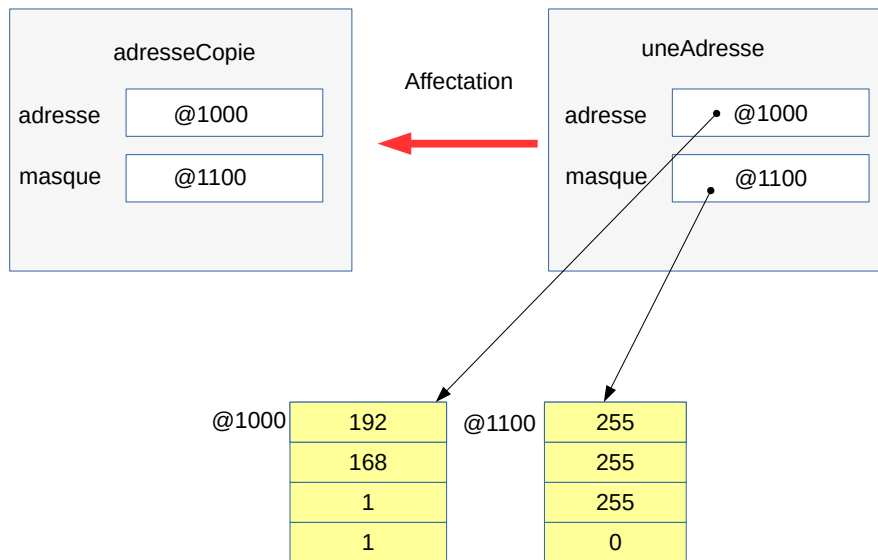
    delete uneAdresse; // destruction de la première instance

    cout << "Adresse réseau après destruction : ";
    adresseCopie.ObtenirAdresseReseau(reseau);
    AfficherTableau(reseau);

    return 0;
}
```



Constat fait, notre classe ne permet pas de réaliser cette opération. Lors de la copie, seuls les pointeurs ont été recopiés. La mémoire ayant été restituée lors de la destruction de la première instance, les pointeurs de la seconde pointent dans « le vide ».



Intérêt

La forme canonique dite de Coplien d'une classe impose la duplication de la mémoire allouée dynamiquement. Ainsi lorsqu'il y a une affectation ou un passage de paramètres par valeur, l'ensemble des données est bien dupliqué. Les deux instances possèdent une durée de vie dissociée.

Cette deuxième version présente la classe **IPv4** sous sa forme canonique dite de **Coplien**.

Déclaration de la classe IPv4 sous sa forme canonique :

ipv4_coplien.h

```
#ifndef _IPV4_H
#define _IPV4_H

class IPv4
{
private:
    unsigned char * adresse;
    unsigned char * masque ;

public:
    IPv4(const unsigned char * _adresse, const unsigned char _cidr);
    IPv4(const unsigned char * _adresse, const unsigned char * _masque);
    ~IPv4(); // destructeur
    // Ajout pour la forme canonique en plus du destructeur
    IPv4(); // constructeur par défaut
    IPv4(const IPv4& _ipv4); // constructeur de copie
    IPv4 &operator= (const IPv4& _ipv4); // opérateur d'affectation
private:
    void CalculerMasque(const unsigned char _cidr);
public:
    void ObtenirMasque(unsigned char * _masque);
    void ObtenirAdresseReseau(unsigned char * _reseau);
    void ObtenirAdresseDiffusion(unsigned char * _diffusion);
};

#endif
```

Définition des nouveaux constructeurs, le premier alloue simplement de la mémoire, le second alloue la mémoire et recopie les données transmises en paramètre.

Définition des nouveaux constructeurs :

ipv4_coplien.cpp

```
IPv4::IPv4()
{
    adresse = new unsigned char [4];
    masque = new unsigned char [4];
}

IPv4::IPv4(const IPv4 &_ipv4)
{
    adresse = new unsigned char [4];
    masque = new unsigned char [4];
    for(int indice = 0 ; indice < 4 ; indice++)
    {
        adresse[indice] = _ipv4.adresse[indice];
        masque[indice] = _ipv4.masque[indice];
    }
}
```

Remarque

Dans le cas présent, bien que privés les attributs de l'objet `_ipv4` sont accessibles dans le nouvel objet en construction.

Pour la surcharge de l'opérateur d'affectation, la mission est un peu plus délicate. En effet, il faut s'assurer :

- que l'objet n'est pas lui même, dans ce cas, il est retourné directement.
- ou, si la mémoire avait déjà été allouée dynamiquement pour ces variables, il faut la restituer avant d'affecter de nouvelles données.

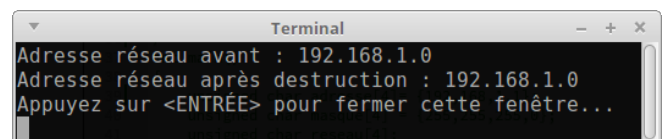
Voici le code correspondant pour la classe **IPv4** :

Définition de l'opérateur d'affectation :

ipv4_coplien.cpp

```
IPv4 &IPv4::operator=(const IPv4 &_ipv4)
{
    if(adresse != _ipv4.adresse || masque != _ipv4.masque)
    {
        if(adresse != nullptr && masque != nullptr)
        {
            delete [] adresse;
            delete [] masque ;
        }
        adresse = new unsigned char [4];
        masque = new unsigned char [4];
        for(int indice = 0 ; indice < 4 ; indice++)
        {
            masque[indice] = _ipv4.masque[indice];
            adresse[indice] = _ipv4.adresse[indice];
        }
    }
    return *this
}
```

Le reste de la classe est inchangée. Voici le résultat :



Exercice d'application :

Décrivez les lignes de code suivant en indiquant quelle partie de la classe **IPv4** sous forme canonique a été mise en jeu :

Exercice

1. `IPv4 adresseIP;`
2. `IPv4 adresse1(adresseIP);`
3. `IPv4 adresse2 = adresseIP;`
4. `adresse1 = adresseIP;`

1. :

2. :

3. :


4. :

2.4. Application

Note au lecteur

Dans ce paragraphe, il est demandé de faire l'étude d'un menu pour sélectionner une option parmi plusieurs proposées. Cette application représente une synthèse des éléments vus jusqu'à maintenant. La classe **Menu**, que nous allons construire, sera réutilisée dans d'autres parties de ce cours.

L'objectif est d'afficher un menu chargé à partir d'un fichier et d'assurer son fonctionnement. Ce menu doit être parfaitement dynamique et s'adapter au fichier d'entrée.

menu.txt	Résultat attendu	Représentation UML
<pre>Option 1 Option 2 Option 3 Option 4 Quitter</pre>		<pre> classDiagram class Menu { - nom : string - options : string * - nbOptions : integer - longueurMax : integer + Menu(in _nom: string) C0 + ~Menu() D0 + Afficher() : integer + AttendreAppuiTouche() } </pre>

Description des Attributs

nom	Désigne le nom du fichier
options	Représente un tableau de chaînes de caractères implémentées sous la forme de string. Ce tableau sera alloué dynamiquement en fonction du nombre de lignes du fichier.
NbOptions	Contient le nombre d'options du Menu
LongueurMax	Taille de la plus grande chaîne contenue dans le tableau

La méthode **AttendreAppuiTouche()** de la classe **Menu** est déclarée **statique** comme l'indique. Elle pourra être appelée même si la classe **Menu** n'est pas instanciée en invoquant **Menu::AttendreAppuiTouche()** ; Pour les autres méthodes rien de particulier.

- a) Déclarez la classe **Menu** dans un fichier **menu.h**

b) Complétez le code du constructeur à partir de l'algorithme suivant

Constructeur de la classe Menu

menu.cpp

```
Menu::Menu(const string &_nom):nom(_nom), longueurMax(0)
{
    // ouvrir le fichier
    // Si il y a une erreur
    //     alors Afficher un message indiquant une erreur de lecture
    //     et mettre nbOptions à 0
    // Sinon calculer nbOptions, le nombre d'options dans le fichier
    //     allouer dynamiquement le tableau options en fonction de nbOptions
    //     Pour chaque option dans le fichier
    //         Lire l'option et l'affecter dans le tableau options
    //         Si la taille de l'option est plus grande que longueurMax
    //             alors longueurMax reçoit la taille de l'option
    //         FinSi
    //     FinPour
    // FinSi
}
```

Quelques compléments d'information pour le codage de la méthode :

Pour calculer le nombre de lignes dans un fichier, on peut faire une première lecture et compter le nombre de lignes au fur et à mesure.

La ligne ci-dessous réalise le même traitement. Elle est à utiliser telle quelle, des explications supplémentaires interviendront à la fin du document dans les chapitres sur les **templates** et les algorithmes génériques.

```
nbLignes = static_cast<int>(count(istreambuf_iterator<char>(fichierMenu), istreambuf_iterator<char>(), '\n'));
```



Réalisation d'un transtypage, le résultat doit être un entier



Nom de la variable de type ifstream

La fonction `count` nécessite l'inclusion de la librairie Algorithm : `#include <algorithm>`

Il ne faut pas oublier de revenir au début du fichier pour reprendre la lecture de chaque ligne après.

```
fichierMenu.seekg(0, ios::beg);
```

<http://www.cplusplus.com/reference/istream/istream/seekg/>

La lecture de la ligne se fait par la fonction `getline`. La taille de la chaîne de caractères est donnée par la méthode `length` de la classe `std::String`. Ces éléments sont décrits dans la documentation de référence de cette classe. <http://www.cplusplus.com/reference/string/string/>.

c) Codez le destructeur, il y a eu une allocation dynamique dans le constructeur

d) Codez à présent la méthode `int Menu::Afficher()` en respectant la représentation donnée en exemple. Elle assure également la saisie du choix de l'utilisateur. Si ce choix n'est pas dans les valeurs admises l'affichage devient :

Pour la saisie, on ajoute à `cin` un test pour être bien sûr d'effectuer une saisie et supprimer les retours chariot superflus

```
if(! (cin>>choix))
{
    cin.clear();
    cin.ignore(std::numeric_limits<streamsize>::max(), '\n');
    choix = -1;
}
```

<http://www.cplusplus.com/reference/ios/ios/clear/>
<http://www.cplusplus.com/reference/istream/istream/ignore/>

Les deux lignes `cin.clear()` et `cin.ignore(..)` peuvent être ajoutée avant le retourner le choix de l'utilisateur pour vider complètement le tampon d'entrée pour les saisies à venir dans le programme.

Pour effacer l'écran sous Linux avant l'affichage du menu et en quittant la fonction un appel système peut être réalisé avec :

- sous Linux : `system("clear");`
- sous Windows : `system("cls");`

e) Pour la méthode **void Menu::AttendreAppuiTouche()** on propose le code suivant :

Constructeur de la classe Menu

menu.cpp

```
void Menu::AttendreAppuiTouche()
{
    string uneChaine;
    cout << endl << "appuyer sur la touche Entrée pour continuer...";
    getline(cin, uneChaine);
    cin.ignore( std::numeric_limits<streamsize>::max(), '\n' );
    system("clear");
}
```

f) Complétez le programme principal à partir des éléments suivants :

Constructeur de la classe Menu

menu.cpp

```
int main()
{
    int choix;
    Menu leMenu("menu.txt");
    do
    {
        choix = leMenu.Afficher();
        switch (choix)
        {
            case OPTION_1:
                cout << "Vous avez choisi l'option n°1" << endl;
                Menu::AttendreAppuiTouche();
                break;
            // à compléter
        }
    } while(choix != QUITTER);

    return 0;
}
```

```
enum CHOIX_MENU
{
    OPTION_1 = 1,
    OPTION_2,
    OPTION_3,
    OPTION_4,
    QUITTER
};
```

Remarque : Le fichier menu.txt doit être dans le même répertoire que l'exécutable de votre programme.

2.5. Traitement des erreurs avec une Classe d'exception

Afin d'améliorer et de simplifier le traitement des erreurs, il est possible de créer des classes susceptibles de gérer les exceptions. Prenons l'exemple d'une classe **Tableau** dont le rôle est de créer un tableau dynamiquement. Les risques de plantage peuvent être liés :

- à la création du tableau avec une dimension négative,
- à l'accès à un élément du tableau avec un indice négatif,
- et à l'accès en lecture ou en écriture à un élément du tableau d'indice supérieur ou égale à la taille du tableau, même si cela ne fait pas directement planter le programme, le risque est juste d'écraser des variables adjacentes dans la mémoire.

Un programme bien fait s'interdit toute ces opérations. Toutefois, rien n'empêche un programmeur de mal utiliser cette classe **Tableau**. Dans ce cas, il faut lui indiquer et, en général, stopper l'exécution. Pour cela, il est nécessaire de lever une exception lorsque les dimensions ne sont pas convenables ou que l'indice est en dehors des limites.

Définition de la classe Tableau

tableau.h

```
#ifndef TABLEAU_H
#define TABLEAU_H

class Tableau
{
private:
    int *ptr;
    int taille;
public:
    Tableau(int _taille);
    ~Tableau();
    void Affecter(int _indice, int _valeur);
    int &operator[](int _indice);
};

#endif // TABLEAU_H
```

Tableau	
- ptr : integer *	
- taille : integer	
+ Tableau(in _taille: integer)	C0
+ ~Tableau()	D0
+ Affecter(in _indice: integer, in _valeur: integer)	
+ Operator [] (in _indice: integer): integer &	

Comme nous pouvons le constater, la définition de la classe reste tout à fait classique.

Lors de l'implémentation de cette classe, il faut veiller à ce que les erreurs ne puissent pas se produire. Pour cela deux nouvelles classes peuvent être définies la première, la classe **ErreurCreation**, va réagir lors de problèmes de création du tableau, la seconde la classe **ErreurIndice** sera appelée lors d'un défaut d'accès aux éléments du tableau.

La définition de ces deux classes est réalisée sur le même modèle et peut être déclarée dans le fichier `tableau.h`.

Définition des classes d'exception	<code>tableau.h</code>
<pre>class ErreurCreation { private: int codeErreur; string message; public: ErreurCreation(int _codeErreur, string _message); int ObtenirCodeErreur() const; string ObtenirDescription() const; };</pre>	<pre>class ErreurIndice { private: int codeErreur; string message; public: ErreurIndice(int _codeErreur, string _message); int ObtenirCodeErreur() const; string ObtenirDescription() const; };</pre>

Pour chaque type d'erreur, un code spécifique est déterminé ainsi qu'un message indiquant la nature de l'erreur. Les codes d'erreur peuvent être définis par une constante énumérée.

Définition des constantes énumérée	<code>tableau.h</code>
<pre>enum erreurs { CREATION, INDICE };</pre>	<p>Ici les valeurs sont 0 et 1, mais rien n'empêche de fixer d'autres valeurs, 100 et 101 par exemple en fonction de la gestion d'erreurs pour le reste du programme.</p>

L'implémentation de la classe `Tableau` devient alors :

Implémentation de la classe <code>tableau</code>	<code>tableau.cpp</code>
<pre>Tableau::Tableau(int _taille): taille(_taille) { if(taille <= 0) { ptr = nullptr; ErreurCreation excep(CREATION, "taille incorrecte lors de la création du tableau"); throw (excep); } ptr = new int[taille]; } Tableau::~~Tableau() { if(ptr != nullptr) delete[] ptr; }</pre>	

Dans le constructeur, si la taille de tableau est fixée de manière incorrecte, le pointeur est initialisé à **nullptr**, qui représente la valeur du pointeur nul directement supportée par le langage C++11. Ensuite une instance de la classe **ErreurCreation** est définie et l'exception est levée avec **throw**. L'absence du **else** ne fait pas défaut ici, car, dès l'instant où l'instruction **throw** est exécutée, le programme est dérivé vers le bloc **catch** correspondant à l'objet lancé. Reste simplement à l'utilisateur de cette classe à instancier la classe `tableau` dans un bloc **try** qui sera présenté dans le programme principal. L'autre conséquence de l'exécution de l'instruction **throw** est l'appel de tous les destructeurs des objets déclarés dans le bloc **try**. C'est pour cela qu'il est de bonne facture de tester si le pointeur **ptr** est bien différent de **nullptr** avant de libérer la mémoire, cette règle évite des plantages lorsque le programme s'arrête anormalement.

Remarque

L'exécution du programme reprend après le bloc **catch** et non pas à l'endroit où se trouve l'instruction **throw** en cas de levée d'exception.

La suite de l'implémentation de la classe Tableau reprend la même logique en utilisant la deuxième classe d'exception.

Implémentation de la classe Tableau

tableau.cpp

```
void Tableau::Affecter(int _indice, int _valeur)
{
    if(_indice < 0 || _indice > taille)
    {
        ErreurIndice excep(INDICE, "L'indice du tableau n'est pas correct lors de l'affectation");
        throw (excep);
    }
    ptr[_indice] = _valeur;
}

int &Tableau::operator[](int _indice)
{
    if(_indice < 0 || _indice > taille)
    {
        ErreurIndice excep(INDICE, "L'indice du tableau n'est pas correct pour l'opérateur []");
        throw (excep);
    }
    return ptr[_indice];
}
```

Pour l'implémentation des deux classes d'exception, elles peuvent également être associées au code de la classe Tableau et réalisées dans le même fichier.

Implémentation de la classe tableau

tableau.cpp

```
ErreurIndice::ErreurIndice(int _codeErreur, string _message):
    codeErreur(_codeErreur),
    message(_message)
{
}

int ErreurIndice::ObtenirCodeErreur() const
{
    return codeErreur;
}

string ErreurIndice::ObtenirDescription() const
{
    return message;
}

ErreurCreation::ErreurCreation(int _codeErreur, string _message):
    codeErreur(_codeErreur),
    message(_message)
{
}

int ErreurCreation::ObtenirCodeErreur() const
{
    return codeErreur;
}

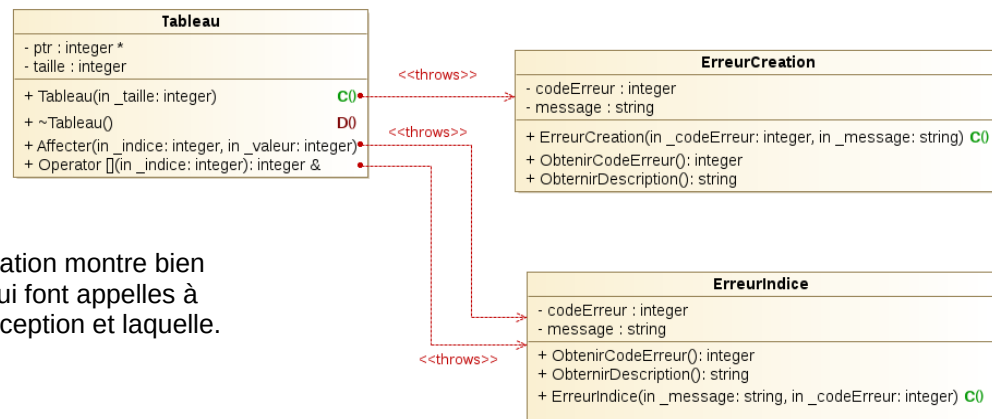
string ErreurCreation::ObtenirDescription() const
{
    return message;
}
```

Elles sont construites, toutes deux, sur le même modèle.

Remarque

Par la suite, il pourra être envisagé d'améliorer ces classes d'exception en utilisant une classe abstraite, notion abordée dans le prochain chapitre.

La représentation UML de la classe tableau et ces deux classes d'exception est donnée ici :



Cette représentation montre bien les méthodes qui font appelées à une classe d'exception et laquelle.

Pour finir cette étude sur les classes d'exception, il ne reste plus que l'implémentation du programme principal et notamment l'utilisation du bloc **try** et des blocs catch pour le traitement des exceptions.

Implémentation du programme principal

utilisationtableau.cpp

```

#include "tableau.h"
#include <iostream>
#include <stdlib.h>
using namespace std;
int main()
{
    try
    {
        Tableau leTableau(10);
        leTableau.Affecter(5,8);
        leTableau.Affecter(-5,15);
        cout << leTableau[-5] << endl;
        Tableau autre(-12);
    }
    catch(ErreurCreation const &exp)
    {
        cout << "Erreur " << exp.ObtenirCodeErreur() << endl;
        cout << exp.ObtenirDescription() << endl;
        exit (EXIT_FAILURE);
    }
    catch (ErreurIndice const &exp)
    {
        cout << "Erreur " << exp.ObtenirCodeErreur() << endl;
        cout << exp.ObtenirDescription() << endl;
        exit (EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
  
```

Dans ce programme principal, les différents types d'erreurs ont été cumulés, lors de vos tests il est nécessaire de mettre en commentaire les lignes pour lesquelles on ne souhaite pas lever d'exception.

Remarque

Ici, le choix a été de stopper le programme, mais il aurait pu être envisagé tout autre cas de figure comme redemander une taille de tableau correcte...

Exercice d'application :

Reprenez la gestion de menu étudiée précédemment, recherchez quelles pourraient être les causes de plantages de cette gestion et réalisez un traitement d'erreur approprié en utilisant une ou plusieurs classes d'exception.