


Dans ce chapitre nous continuons notre étude des classes et de l'abstraction de données. Nous discutons de beaucoup d'autres notions avancées et exposons les préliminaires des classes et de la surcharge d'opérateurs, qui seront approfondis au chapitre 8.

Les chapitres 6, 7 et 8 encouragent les programmeurs à utiliser les objets, ce que nous appelons la *programmation orientée objets*. Par la suite, les chapitres 9 et 10 introduisent l'héritage et le polymorphisme, les véritables techniques de cette programmation.




Objectifs

- Créer et détruire des objets dynamiquement
- Manipuler des objets `const`
- Comprendre l'utilité de l'attribut `friend`
 - Fonctions `friend`
 - Classes `friend`
- Comprendre l'utilité du pointeur `this`

Marco Lavoie 14728 ORD - Langage C++

Les principaux sujets couverts dans ce chapitre sont :

- la création et destructions d'objets via la gestion dynamique de la mémoire via les opérateurs `new` et `delete`;
- la gestion d'objets non modifiables dans les fonctions membres de classes;
- permettre à une fonction cliente de classe l'accès aux membres privés de cette dernière en la déclarant amie (`friend`) de la classe;
- permettre à une fonction membre de classe l'accès à l'objet via lequel la fonction membre fut invoquée dans le code client exploitant la classe;
- définir des attributs membres destinés à être partagés par tous les objets instanciés de la classe.



Aperçu

- Objets `const` et fonctions membres `const`
- Objets comme attributs membres
- Fonctions `friend` et classes `friend`
- Utilisation du pointeur `this`
- Allocation dynamique de mémoire avec les opérateurs `new` et `delete`
- Membres de classes `static`
- La macro `assert`

Marco Lavoie 14728 ORD - Langage C++

Pour ce chapitre, ainsi que pour les quelques suivants, nous utilisons les chaînes de caractères propres au C, initialement introduites au chapitre 5. Cette pratique vise à aider le lecteur à maîtriser le sujet complexe des pointeurs et à le préparer pour le marché du travail, dans lequel le code écrit en C demeure omniprésent, du fait d'une utilisation constante depuis les vingt dernières années. Nous étudierons en fin de trimestre la classe `string`, un nouveau style de gestion de chaînes de caractères, où celles-ci sont considérées comme des objets de classe à part entière. De cette façon, le lecteur pourra se familiariser avec les deux méthodes répandues de création et de manipulation de chaînes de caractères en langage C++.



Introduction

4

- Le langage C++ est complexe car il offre une flexibilité accrue en rapport aux autres langages
 - Allocation statique et dynamique d'objets
 - Objets constants
 - Fonctions et classes amies
 - Pour assurer la surcharge d'opérateurs
- Ce chapitre présente ces particularités du C++ en préparation des chapitres suivants

Marco Lavoie

14728 ORD - Langage C++

Plusieurs des langages de programmation exploités dans l'industrie, tels que *Java* et *C#*, offrent une seule stratégie d'allocation de mémoire pour les objets : l'allocation dynamique. Cette technique d'allocation de la mémoire est caractérisée par l'utilisation de l'opérateur **new** pour créer des objets ou des tableaux. La plupart de ces langages offrant la gestion automatisée des blocs mémoire inutilisés (ce qu'on appelle la *collecte de déchets*), ils n'offrent pas explicitement un opérateur **delete** pour détruire les objets en fin de vie active, le collecteur de déchets s'acquittant de cette tâche de façon transparente.

Contrairement à ces langages, le C++ offre deux techniques d'allocation de mémoire aux objets : l'*allocation statique* et l'*allocation dynamique*. Ainsi, le programmeur dispose d'une plus grande flexibilité en termes de gestion de la mémoire afin de maximiser les performances de son application. Ce choix stratégique d'offrir deux techniques implique cependant une plus grande complexité du langage.



Objets et fonctions membres **const**

5

- Le mot-clé **const** permet de définir une variable en lecture seulement

```
const int taille = 20;
```

- Similairement, on peut instancier une classe sous forme d'objet en lecture seulement
 - La variable `heureSouper` est une constante fixée à 17:00:00 et ne pouvant pas être modifiée (i.e. en lecture seulement)

```
const Temps heureSouper( 17, 0, 0 );
```

Marco Lavoie

14728 ORD - Langage C++

Certains objets doivent demeurer modifiables, et d'autres pas. Le programmeur dispose du mot-clé **const** pour spécifier qu'un objet ne peut pas être modifié et pour s'assurer que toute tentative de modification sur cet objet provoque une erreur de syntaxe.

Les deux exemples ci-contre démontrent l'utilisation de **const** pour désigner une variable constante ainsi qu'un objet (du type de la classe **Temps**) constant.

La déclaration comme **const** des variables et objets constitue non seulement une pratique efficace pour la conception de logiciels, mais permet également une amélioration de la performance. En effet, les compilateurs sophistiqués d'aujourd'hui peuvent effectuer certaines opérations d'optimisation sur les constantes qui sont impossibles sur les variables modifiables.



Rappel : variable **const** et paramètre référence

6

- Que se passe-t-il dans l'exemple suivant ?

```
void fonce( int &x ) {
    x = 1;
}

int main() {
    const int a = 0;

    fonce( a );
    std::cout << a;

    return 0;
}
```

– L'invocation `fonce(a)` fait en sorte que la fonction tente de modifier la valeur de `a` via le paramètre référence `x`

– Conséquence : *erreur de compilation*

- Le compilateur ne permet pas l'invocation

L'appel par référence en C++ peut occasionner des difficultés lorsque combiné aux arguments constants. Dans l'exemple ci-contre, la fonction `fonce` dispose d'un paramètre référence (`x`) qui est conséquemment un alias de l'argument fourni lors de l'invocation de `fonce` par `main`, soit la variable non modifiable `a`. Ainsi, lorsque `fonce` affecte 1 à son paramètre `x`, en fait elle tente d'affecter cette valeur à la variable `a` dans `main`, qui est désignée **const**.

Une telle opération n'étant évidemment pas permise, le compilateur refuse de compiler ce code source, indiquant l'invocation de `fonce` dans `main` comme étant une erreur de syntaxe. Notez que l'erreur serait indiquée même si `fonce` ne tentait pas de modifier la valeur de son paramètre `x`. Dès qu'une fonction dispose d'un paramètre référence, il est interdit d'invoquer cette fonction avec un argument constant correspondant à ce paramètre. Il y a cependant des exceptions à cette règle, qui seront décrites plus tard.

Marco Lavoie

14728 ORD - Langage C++



Rappel : impact d'une fonction membre

7

- Une fonction membre peut modifier l'objet pour lequel elle s'exécute

```
class Temps {
    int heure, minute, seconde;
public:
    Temps( int = 0, int = 0, int = 0 );
    void ajusterTemps( int, int, int );
};

void Temps::ajusterTemps( int h, int m, int s ) {
    heure = h;
    minute = m;
    seconde = s;
}

int main() {
    Temps t( 0, 0, 0 );
    t.ajusterTemps( 17, 4, 39 );
}
```

Si la variable **t** est déclarée constante alors **ça ne compilerait pas**

Les attributs de la variable **t** sont modifiés par l'invocation

Marco Lavoie

14728 ORD - Langage C++

La restriction imposée aux fonctions conventionnelles ne pouvant recevoir en paramètre référence un argument correspondant à une variable **const** s'applique aussi aux fonctions membres invoquées via un objet constant. Puisqu'une fonction membre de classe peut modifier les valeurs des attributs membres de l'objet via lequel elle fut invoquée, le compilateur refusera une telle invocation, même si la fonction membre ne modifie aucun attribut membre.

Dans l'exemple ci-contre, si la variable **t** de **main** est déclarée constante, l'invocation de **ajusterTemps** pour l'objet **t** est refusée par le compilateur puisque cette dernière risque de modifier les attributs de **t** (soit **t.heure**, **t.minute** ou **t.seconde**).



Rappel : impact d'une fonction membre (suite)

8

- Et qu'en est-il d'une fonction membre ne modifiant pas l'objet ?

```
class Temps {
    int heure, minute, seconde;
public:
    Temps( int = 0, int = 0, int = 0 );
    void ajusterTemps( int, int, int );
    bool validerTemps();
};

bool Temps::validerTemps() {
    return ( heure >= 0 && heure <= 23 ) &&
           ( minute >= 0 && minute <= 59 ) &&
           ( seconde >= 0 && seconde <= 59 );
}

int main() {
    const Temps t( 0, 0, 0 );
    bool res = t.validerTemps();
}
```

– Ça ne compile pas non plus

- Le compilateur ne peut pas efficacement déterminer si la fonction **validerTemps()** va occasionner la modification de l'objet

– Est-ce à dire qu'une fonction membre ne peut traiter un objet constant ?

Marco Lavoie

14728 ORD - Langage C++

Telle que pour les fonctions conventionnelles ayant des paramètres référence, cette restriction empêchant d'invoquer des fonctions membres pour un objet constant est aussi applicable aux fonctions membres ne tentant pas de modifier les attributs de l'objet.

Dans l'exemple ci-contre, même si la fonction membre **validerTemps** ne modifie pas les attributs **heure**, **minute** et/ou **seconde** de **t**, le compilateur refuse de compiler le programme car il y a tout de même risques que **t** soit indirectement modifié par l'invocation de la fonction membre (celle-ci pourrait par exemple à son tour invoquer une autre fonction membre qui elle pourrait modifier les attributs de l'objet).

En règle générale, dès qu'un objet est déclaré **const**, il est impossible d'invoquer une fonction membre de la classe via cet objet. Mais est-ce vraiment le cas ?



Rappel : impact d'une fonction membre (suite)

9

- Non, mais il faut indiquer au compilateur que la fonction membre ne modifie pas l'objet sur lequel elle s'exécute

```
class Temps {
    int heure, minute, seconde;
public:
    Temps( int = 0, int = 0, int = 0 );
    void ajusterTemps( int, int, int );
    bool validerTemps() const;
};

bool Temps::validerTemps() const {
    return ( heure >= 0 && heure <= 23 ) &&
           ( minute >= 0 && minute <= 59 ) &&
           ( seconde >= 0 && seconde <= 59 );
}

int main() {
    const Temps t( 0, 0, 0 );
    bool res = t.validerTemps();
}
```

En indiquant que la fonction membre est **const**, on certifie au compilateur que celle-ci ne modifie pas les attributs de l'objet

Marco Lavoie

14728 ORD - Langage C++

En effet, il est impensable qu'il soit impossible d'invoquer une fonction membre de la classe sur un objet désigné **const**. Quel serait alors l'utilité de déclarer de tels objets ? En fait, il est possible d'invoquer via un objet **const** une fonction membre de classe en autant que celle-ci soit elle aussi désignée **const**.

Pour désigner une fonction **const**, il suffit d'ajouter en suffixe le mot-clé **const** après la parenthèse fermant la liste des paramètres de la fonction. Désigner ainsi une fonction membre **const** indique au compilateur que cette fonction ne fait aucune modification aux attributs membres de la classe. Ainsi, il est alors permis d'invoquer cette fonction membre pour un objet déclaré **const**.

Fonctions membres `const`

10

- Elles peuvent être invoquées par des objets `const` ou non

```
class Temps {
    int heure, minute, seconde;
public:
    Temps( int = 0, int = 0, int = 0 );
    void ajusterTemps( int, int, int );
    bool validerTemps() const;
};

int main() {
    const Temps tc( 0, 0, 0 );
    Temps tv( 0, 0, 0 );

    tc.validerTemps();
    tv.validerTemps();
    tv.ajusterTemps( 17, 4, 39 );
}
```

← Notez l'absence de

car elle ne peut être invoquée sur `tc`

Marco Lavoie

14728 ORD - Langage C++

Notez qu'une fonction membre est désignée `const` à la fin du prototype (c.à.d. après la liste de ses paramètres), telle que

```
bool validerTemps() const;
```

et non en début de prototype. Insérer `const` avant le type de valeur de retour d'une fonction signifie que cette dernière retourne une constante comme valeur de retour. Par exemple,

```
const bool validerTemps();
```

indique que la fonction membre retourne une constante de type `bool`, mais non que cette fonction ne modifie pas les attributs membres. Nous verrons plus tard l'utilité de retourner une valeur de retour constante.

Notez qu'une fonction membre désignée `const` peut toujours être invoquée pour un objet n'étant pas constant.

Fonctions membres `const` (suite)

11

- Et si une fonction membre `const` tente de modifier les attributs de l'objet ?

```
class Temps {
    int heure, minute, seconde;
public:
    Temps( int = 0, int = 0, int = 0 );
    void ajusterTemps( int, int, int ) const;
};

void Temps::ajusterTemps( int h, int m, int s ) const {
    heure = h;
    minute = m;
    seconde = s;
}
```

– Ça ne compile pas car la fonction membre n'est pas autorisée à modifier les attributs de l'objet

Marco Lavoie

14728 ORD - Langage C++

Le fait de désigner une fonction membre comme étant constante indique au compilateur que cette dernière ne modifie pas les valeurs d'attributs membres de classe pour l'objet sur lequel elle fut invoquée. Si une fonction désignée `const` tente de modifier les attributs membres directement, tel que dans l'exemple ci-contre, soit indirectement en invoquant à son tour une autre fonction membres n'étant pas désignée elle aussi `const`, le compilateur refuse de compiler l'invocation, générant une erreur de syntaxe à cet effet.

Fonctions membres `const` (suite)

12

- Et qu'en est-il des constructeurs ?

– On ne déclare jamais les constructeurs avec le qualificatif `const`, car ceux-ci peuvent modifier l'objet

– En C++, on ne peut pas définir un constructeur `const`

```
class Temps {
    int heure, minute, seconde;
public:
    Temps( int = 0, int = 0, int = 0 );
};

Temps::Temps( int h, int m, int s ) {
    heure = h;
    minute = m;
    seconde = s;
}

int main() {
    const Temps tc( 17, 4, 39 );
    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

Puisque le principal rôle d'un constructeur est d'initialiser les attributs membres d'un nouvel objet, un constructeur peut conséquemment modifier les valeurs de ces attributs même pour un objet désigné `const`. Dans l'exemple ci-contre, l'objet `tc` devient constant uniquement une fois ses attributs initialisés par le constructeur.



Attributs membres **const**

- Un attribut membre peut être **const**
 - Mais le constructeur ne peut pas initialiser cet attribut car il est **const**
 - Un attribut ne peut pas non être initialisé dans la déclaration de la classe
- Alors comment l'initialiser ?

```
class Increment {
public:
    Increment( int = 0, int = 1 );
    void Incrementer();
    int getCompteur();
private:
    int compteur;
    const int increment = 1;
};

Increment::Increment( int c, int i ) {
    compteur = c;
    increment = i;
}

void Increment::Incrementer() {
    compteur += increment;
}

int Increment::getCompteur() {
    return compteur;
}
```

Marco Lavoie

14728 ORD - Langage C++

Jusqu'à présent nous avons vu qu'un objet peut être désigné **const** lors de sa déclaration, et qu'une fonction membre de classe peut être désignée **const** afin de manipuler des objets constants de cette classe.

Un attribut membre de classe peut aussi être désigné **const**, indiquant ainsi que sa valeur ne peut pas être modifiée par les fonctions membres de la classe, ni pas les clients de la classe lorsque l'attribut est d'accès **public**.

La seule façon d'attribuer une valeur à un attribut membre constant d'une classe est via ses constructeurs. Cependant, un constructeur n'est pas autorisé à affecter une valeur à un tel attribut dans son corps, puisqu'il pourrait successivement affecter deux valeurs différentes à l'attribut **const**, ce qui est contraire au concept de constance d'un attribut (il ne devrait pouvoir recevoir qu'une valeur).



Attributs membres **const** (suite)

- Attributs membres **const** initialisés dans le constructeur via un *initialiseur*
 - Énumération des attributs à initialiser avec sa valeur initiale entre parenthèses, séparés par des virgules
 - Les initialiseurs peuvent aussi initialiser les attributs non constants

```
class Increment {
public:
    Increment( int = 0, int = 1 );
private:
    int compteur;
    const int increment;
};

Increment::Increment( int c, int i )
    : increment( i ), compteur( c ) {
    compteur = c;
}
```

Notez que le bloc de code du constructeur est vide puisqu'il n'y a rien d'autre à y mettre

Marco Lavoie

14728 ORD - Langage C++

Pour palier au fait qu'un constructeur ne peut pas directement affecter une valeur à un attribut constant dans son corps, le C++ offre les *initialiseurs* à cette fin. Un initialiseur permet d'initialiser un attribut membre (**const** ou pas) avant l'exécution du corps du constructeur. Dans l'exemple ci-contre, la notation `: increment(i)` initialise **increment** à la valeur **i**. Si de multiples initialiseurs d'attributs membres sont requis, ils peuvent être énumérés par cette syntaxe en les séparant par des virgules.



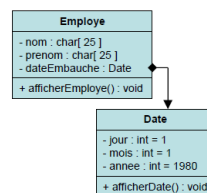
Objets membres

- Un attribut membre peut être un objet (i.e. instance) d'une autre classe

```
class Date {
public:
    Date( int = 1, int = 1, int = 1980 );
    void afficherDate();
private:
    int jour, mois, annee;
};

class Employee {
public:
    Employee( char [], char [], int, int, int );
private:
    char nom[ 25 ], prenom[ 25 ];
    Date dateEmbauche;
    void afficherEmployee();
};
```

– En UML, la flèche \blacklozenge représente cette relation



Marco Lavoie

14728 ORD - Langage C++

Un objet de type **Employee** a besoin de connaître quand l'employé qu'il représente a été embauché; pourquoi alors ne pas inclure un objet **Date** comme attribut membre de la classe **Employee**? On appelle *composition* une telle possibilité qui permet à une classe de posséder des membres provenant d'une autre classe.

En UML la composition est représentée par une flèche entre les deux classes impliquées. La direction de cette flèche indique quelle classe (le point de départ de la flèche) est composée en partie d'un attribut membre du type de la seconde classe (la destination de la flèche).

La composition est une forme de réutilisation en génie logiciels. Dans l'exemple ci-contre, pourquoi définir trois attributs (**jour**, **mois** et **annee**) dans la classe **Employee** pour gérer la date d'embauche, alors qu'une instance de la classe **Date** peut très bien faire l'affaire tout en offrant du même coup toutes les fonctionnalités requises (via ses fonctions membres) pour manipuler la date d'embauche (telle que la lecture et l'affichage).

Objets membres : initialiseurs

16

- Comment initialiser un objet membres ?

- Avec des initialiseurs exploitant un constructeur de l'objet membre

```
Employee::Employee( char nm[], char pnm[], int j, int m, int a )
: dateEmbauche( j, m, a ) {
    strcpy( nom, nm, 25 );
    strcpy( prenom, pnm, 25 );
}
```

- C'est la seule façon d'initialiser `dateEmbauche` car ses attributs `jour`, `mois` et `annee` étant privés, la classe `Employee` n'y a pas directement accès

Marco Lavoie

14728 ORD - Langage C++

Revenons à l'exemple de la page précédente. Lorsqu'un code client déclare une variable de type `Employee`, l'objet obtenu dispose implicitement d'un objet `Date` via son attribut `dateEmbauche`. Une des tâches du constructeur de `Employee` est d'initialiser les attributs membres de l'objet, incluant `dateEmbauche`. Notez cependant que la classe `Date` impose un accès `private` à ses attributs `jour`, `mois` et `annee`. Ainsi, la classe `Employee` n'est pas autorisée à manipuler directement les attributs membres de son objet `dateEmbauche`.

Pour palier à cette restriction, le constructeur de la classe `Employee` exploite un initialiseur pour initialiser les attributs membres de son objet `dateEmbauche`. Le constructeur accepte cinq arguments. Les deux points (`:`) de l'en-tête sépare de la liste de paramètres les initialiseurs des attributs membres, qui spécifient que les arguments du constructeur passés aux constructeurs des attributs membres objets, tel que `dateEmbauche`.

Fonctions amies

17

- Une fonction n'a pas directement accès aux membres privés d'une classe
- Il y a parfois des circonstances dans lesquelles on doit autoriser une fonction à accéder aux membres privés d'une classe
 - Nous verrons ces circonstances plus tard

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

void ajusterDate( Date &d, int j,
                 int m, int a ) {
    d.jour = j;
    d.mois = m;
    d.annee = a;
};
```

*Erreur de compilation car
ajusterDate n'est pas
une fonction membre de
la classe Date*

Marco Lavoie

14728 ORD - Langage C++

Une fonction *friend* (amie) d'une classe se définit en dehors de la portée de cette classe, bien qu'elle puisse accéder aux membres `private` (et aussi, comme nous le verrons au chapitre 9, aux membres `protected`) de la classe. Une fonction conventionnelle ou une classe entière peut être déclarée comme *friend* d'une autre classe.

L'exemple ci-contre illustre une situation où la clause *friend* est requise. La fonction conventionnelle `ajusterDate` reçoit en paramètre un objet `Date` (paramètre `d`) et tente d'initialiser ses attributs aux valeurs reçues via ses paramètres `j`, `m` et `a`. Puisque les attributs correspondants `jour`, `mois` et `annee` de `Date` sont d'accès `private` et que `ajusterDate` n'est pas une fonction membre de `Date`, la fonction ne peut conséquemment pas affecter les valeurs `j`, `m` et `a` aux attributs `jour`, `mois` et `annee` du paramètre `d`.

Fonctions amies (suite)

18

- Une classe peut cependant autoriser des fonctions à accéder à ses membres privés
 - Ces fonctions sont dites *amies* (*friend*) de la classe
 - Il suffit de spécifier le prototype de la fonction amie dans la classe, précédée du mot-clé *friend*

```
class Date {
    friend void ajusterDate( Date &, int, int, int );
public:
    Date( int, int, int );
private:
    int jour, mois, annee;
};
```

```
void ajusterDate( Date &d, int j,
                 int m, int a ) {
    d.jour = j;
    d.mois = m;
    d.annee = a;
};
```

*Maintenant ça
compile*

Marco Lavoie

14728 ORD - Langage C++

Déclarer une fonction *friend* d'une classe signifie que cette fonction « amie » peut accéder à tous les membres de la classe, qu'ils soient `private`, `public` ou `protected`. Pour déclarer une fonction comme *friend* d'une classe, faites précéder le prototype de fonction par le mot-clé *friend* dans la définition de la classe. Les prototypes des fonctions *friend* sont placés au début du corps de la classe, avant le premier identificateur d'accès aux membres.

En dépit du fait que les prototypes de fonctions friend apparaissent dans la définition de la classe, ces fonctions ne constituent pas des fonctions membres de la classe.

L'utilisation de fonctions *friend* améliore les performances. Nous verrons au prochain chapitre l'utilisation des fonctions *friend* pour surcharger les opérateurs, afin de combiner leurs opérations avec des objets de classes.



Classes amies

19

- Comme une fonction, une classe peut aussi être amie d'une autre classe

- Dans l'exemple ci-contre, la classe `Employe` a accès aux attributs privés de la classe `Date`
- L'amitié n'est pas pas réciproque
 - `Date` n'a pas accès aux membres privés de `Employe`

```
class Date {
    friend class Employe;
public:
    Date( int = 1, int = 1, int = 1980 );
    void afficherDate();
private:
    int jour, mois, annee;
};

class Employe {
public:
    Employe( char [], char [], int, int, int );
private:
    char nom[ 25 ], prenom[ 25 ];
    Date dateEmbauche;
    void afficherEmploye();
};
```

Marco Lavoie

14728 ORD - Langage C++

Une déclaration d'amitié est accordée par une classe à une autre, mais ne peut être réclamée. Par exemple, pour qu'une classe `B` devienne amie d'une classe `A`, cette dernière doit explicitement déclarer que la classe `B` est son amie dans sa définition. De plus, l'amitié n'est pas symétrique ni transitive, c'est-à-dire que si on déclare la classe `A` **friend** de la classe `B` et qu'à son tour, on définit la classe `B` **friend** de la classe `C`, on ne peut pas déduire que la classe `B` est **friend** de la classe `A` (l'amitié n'étant pas symétrique), ni que la classe `A` est **friend** de la classe `C`, l'amitié n'étant pas transitive non plus.



Exercice 7.1

20

- Considérez la déclaration de classe suivante :

```
const int MAX = 100;
class Donnees {
public:
    Donnees(); // constructeur par défaut (aucune données)
    Donnees( double *, int ); // constructeur paramétré
    double moyenne(); // calcule la moyenne des données

private:
    double valeurs[ MAX ]; // stockage de données
    int nbValeurs; // nombre de données dans valeurs[]
};
```

- Complétez cette classe de sorte qu'elle fonctionne avec le programme ci-contre

```
int main() {
    const double v[] = { 3.2, 4.6, 2.9 };
    const Donnees data( v, 3 );
    cout << data.moyenne() << endl;
    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

Pour solutionner cet exercice, vous devez modifier la définition de la classe `Donnees` de sorte que ses fonctions membres puissent être invoquées pour un objet constant.



Pointeur implicite `this`

21

- Chaque instance de classe (i.e. objet) a accès à sa propre adresse mémoire via le pointeur `this`

- Chaque fonction membre d'une classe a accès implicitement au pointeur `this` de l'objet pour lequel elle s'exécute
 - Seule exception : les *fonctions membres statiques* (qu'on verra dans les prochaines acétates)
- Principale utilisation : retourner l'objet ou son adresse

Marco Lavoie

14728 ORD - Langage C++

Chaque objet a accès à sa propre adresse par le biais d'un pointeur appelé `this`. Le pointeur `this` d'un objet ne fait pas partie de l'objet en tant que tel; en d'autres termes, le pointeur `this` n'est pas stocké dans le bloc mémoire alloué à l'objet, mais plutôt passé (par le compilateur) comme premier argument implicite (c.à.d. caché) lors de chaque appel de fonction membre non **static** de l'objet (nous discutons des membres **static** plus loin dans ce chapitre).

this est exploité implicitement

22

- Même si on ne fait pas référence à **this**, le compilateur le fait implicitement

```
class Test {
public:
    Test( int = 0 );
    void afficher();
private:
    int x;
};

Test::Test( int a ) {
    x = a;
}

void Test::afficherDate() {
    cout << "    x = " << x
    << "\n  this->x = " << this->x
    << "\n(*this).x = " << (*this).x;
}
```

```
int main() {
    Test t( 12 );
    t.afficher();

    return 0;
}
```

```
C:\>test.exe
    x = 12
  this->x = 12
(*this).x = 12
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

On utilise implicitement le pointeur **this** pour référencer à la fois les attributs membres et les fonctions membres d'un objet dans les fonctions membres de la classe; il peut également intervenir explicitement. La fonction membre **afficherDate** dans l'exemple ci-contre illustre l'utilisation implicite (référence à **x**) et explicite (références à **this->x**) du pointeur **this**.

Le pointeur **this** possède le type **const** de la classe correspondant à la fonction membre dans laquelle il est utilisé. Ainsi, dans l'exemple ci-contre, le pointeur **this** est de type **const Test ***, c'est-à-dire un pointeur constant vers un objet de type **Test**.

this comme valeur de retour

23

- Une fonction membre peut retourner **this** afin de permettre le *chaînage*

```
class Test {
public:
    Test( int = 0 );
    Test & incr();
private:
    int x;
};

Test::Test( int a ) {
    x = a;
}

Test & Test::incr() {
    x++;
    return *this;
}
```

```
int main() {
    Test t( 0 );

    t.incr(); // t.x = 1
    t.incr().incr().incr(); // t.x = 4

    return 0;
}
```

Ça fonctionne car **t.incr()** retourne l'objet, pour lequel on invoque à nouveau la fonction membre

Retourne une référence à l'objet

– Nous verrons plus tard l'utilité du *chaînage*

Marco Lavoie

14728 ORD - Langage C++

L'exemple précédent ne démontre pas une utilisation intéressante du pointeur **this** (la fonction **afficherDate** peut très bien se passer d'utiliser **this** explicitement). Une utilisation plus intéressante du pointeur **this** est de permettre des appels en cascade de fonctions membres. L'exemple ci-contre illustre le renvoi d'une référence vers un objet **Test**. La fonction membre **incr** retourne ***this** avec le type de retour **Test &**.

Pourquoi utiliser cette technique de retour de ***this** comme travail de référence? L'opérateur point (**.**) étant *associatif de gauche à droite*, l'expression

```
t.inc().inc().inc();
```

évalue d'abord **t.inc()**, qui retourne une référence vers l'objet **t** comme valeur pour l'invocation suivante de **inc**, qui elle aussi retourne une référence vers **t**, pour lequel le troisième appel à **inc** est effectué.

Allocation dynamique

24

- Opérateurs d'allocation dynamique
 - new** : permet de réserver un espace mémoire durant l'exécution du programme
 - delete** : permet de libérer un espace mémoire (durant l'exécution) obtenu précédemment via un **new**
- Distinction importante
 - Allocation statique : le compilateur gère le bloc mémoire
 - Allocation dynamique : le programmeur gère le bloc mémoire avec **new** et **delete**

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné précédemment, le C++ offre deux techniques d'allocation de mémoire aux objets : l'*allocation statique* et l'*allocation dynamique*. Le programmeur dispose donc d'une plus grande flexibilité en termes de gestion de la mémoire afin de maximiser les performances de son application. C'est en partie pourquoi les applications écrites en C++ s'exécutent généralement plus rapidement que celles écrites en *Java*, *C#* ou *Visual Basic*.

En C++, les opérateurs **new** (nouveau) et **delete** (supprimer) fournissent un moyen commode d'effectuer les tâches d'allocation dynamique de la mémoire pour tout type prédéfini (tels que **int** ou **double**) ou défini par l'utilisateur (tels que les structures et classes).

Les opérateurs **new** et **delete** de C++ remplacent respectivement **malloc** et **free** du langage C. Comme nous le verrons plus loin, **new** et **delete** offrent plusieurs avantages sur **malloc** et **free**, donc l'invocation implicite de constructeurs et destructeurs.

Opérateur **new**

25

- Retourne un *pointeur* au bloc mémoire alloué

– Exemple :

```
Date *p;
p = new Date( 24, 2, 2010 );
```

- Notes importantes

- Le type du pointeur doit correspondre au type spécifié après **new**
- **new** permet d'invoquer un constructeur si le type spécifié est une classe

Marco Lavoie

14728 ORD - Langage C++

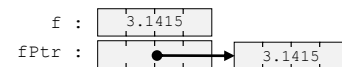
L'opérateur **new** crée automatiquement un objet de la classe spécifiée, lui alloue de la mémoire pour stocker ses valeurs d'attributs membres, et retourne un pointeur vers ce bloc mémoire (c.à.d. l'adresse du bloc mémoire alloué à l'objet).

Le C++ permet de fournir un initialiseur pour un objet nouvellement créé, comme dans l'expression

```
float *fPtr = new float( 3.1415 );
```

qui stocke la valeur 3.1415 dans les octets alloués par **new**. Notez dans l'exemple ci-dessus que la **fPtr** est une variable pointeur dans laquelle est stockée l'adresse du bloc mémoire (c.à.d. l'adresse du premier octet) alloué par **new**. Le diagramme suivant démontre la distinction entre la déclaration statique ci-dessus et

```
float f = 3.1415;
```



Allocation statique vs dynamique

26

- Distinction entre les deux types d'allocation

– À l'exécution

```
{
    Date d( 12, 11, 1998 );
    d.afficher();

    Date *p;
    p = new Date( 24, 2, 2009 );
    p->afficher();
}
```

- À la fin d'exécution du bloc, l'objet **Date** créé par l'instruction **new** demeure en mémoire :

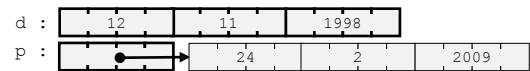
```
jour : 24
mois : 2
annee : 2009
```

- Le bloc mémoire associé à l'objet statique **d** est automatiquement géré par le compilateur

Marco Lavoie

14728 ORD - Langage C++

Dans l'exemple ci-contre, l'objet **d** est statique alors que l'objet pointé par **p** est dynamique (car créé à l'aide de l'opérateur **new**). Voici une représentation graphique de ces deux objets en mémoire (architecture 32 bits) :



Puisque la portée des deux variables **d** et **p** est le bloc de code à l'intérieur duquel elles sont déclarées, ces deux variables sont éliminées à la fin d'exécution du bloc, et la mémoire leur ayant été allouée statiquement (blocs à bordures gras dans le diagramme) est libérée. Notez que le bloc mémoire alloué dynamiquement n'est pas automatiquement libéré à la fin du bloc. Il doit être explicitement libéré par le programme avec l'opérateur **delete**.

Opérateur **delete**

27

- Libère un bloc mémoire obtenu via l'opérateur **new**

– Exemple à l'exécution :

```
{
    Date d( 12, 11, 1998 );
    d.afficher();

    Date *p;
    p = new Date( 24, 2, 2009 );
    p->afficher();
    delete p;
}
```

- L'instruction **delete** détruit le bloc mémoire spécifié
- Notez qu'on donne à **delete** l'adresse mémoire du bloc à détruire (adresse dans le pointeur **p**)

→

Marco Lavoie

14728 ORD - Langage C++

Pour libérer l'espace mémoire alloué à un objet avec l'opérateur **new** en C++, on doit utiliser l'opérateur **delete** en lui fournissant l'adresse du bloc mémoire à libérer. Dans l'exemple ci-contre, l'instruction **delete p** libère l'espace mémoire dont l'adresse est stockée dans la variable pointeur **p**. Notez que l'objectif de cette instruction n'est pas de supprimer la variable pointeur **p**, mais plutôt de supprimer le bloc mémoire dont l'adresse est dans la variable **p** :



La variable pointeur **p** elle-même étant automatique, sa portée ainsi que sa durée de vie est limitée au bloc dans lequel elle est déclarée, et donc elle sera automatiquement détruite à la sortie de ce bloc.



Gestion dynamique de tableaux

28

- Les opérateurs `new` et `delete` permettent de gérer dynamiquement des tableaux
 - En conjonction avec l'opérateur `[]`
 - Puisque `t` est un pointeur, {
 - il faut indiquer à `delete` que celui-ci pointe à un tableau, d'où le besoin d'inclure l'opérateur `[]` après le `delete`

```
int *t = new int[ 10 ];
t[0] = 14;
t[1] = 17;
...
delete [] t;
```

Marco Lavoie

14728 ORD - Langage C++

Comme nous l'avons vu au chapitre 5, en C++ il est possible de créer des tableaux **statiques** (c'est-à-dire dont la taille ne varie pas) en utilisant les `[]`. On peut de cette manière créer un tableau de n'importe quel type de base ou de n'importe quelle classe. En voici des exemples :

```
int    tab1[10]; // Crée un tableau de 10 entiers
maClasse tab2[2]; // Crée un tableau de 2 objets de
                // type « maClasse »
```

Il est également possible en C++ de créer dynamiquement des tableaux, en leur allouant de la mémoire à l'aide de l'opérateur `new` et la récupérant avec `delete`. Dans un tel contexte, l'opérateur `new` requiert un type (afin de connaître l'espace à allouer pour chaque élément) ainsi que les dimensions du tableau à créer (pour déterminer la quantité totale d'espace à allouer au tableau).

Lors de la suppression d'un tableau dynamique, il faut accompagner l'opérateur `delete` des `[]` afin d'indiquer au compilateur que le bloc mémoire à libérer constitue un tableau.



Tableaux dynamiques

29

- Pourquoi gérer un tableau de façon dynamique plutôt que statique ?

← Mémoire allouée à t ici

```
int main() {
    int n;

    cout << "Nombre de valeurs ? ";
    cin >> n;

    int t[ n ]; // créer le tableau ✗
    for ( int i = 0; i < n; i++ )
        cin >> t[ i ];

    ...
}
```

– Le tableau `t` ne peut pas être de taille `n` car son bloc mémoire est alloué au début du bloc, mais la valeur de `n` est lue dans ce même bloc!

Taille du tableau t seulement connue ici

Marco Lavoie

14728 ORD - Langage C++

Lorsqu'on veut créer des tableaux dont la taille **varie** au cours de l'exécution du programme ou si la taille d'un tableau n'est pas connue lors de la compilation, des problèmes apparaissent avec l'utilisation de tableaux statiques. En effet, la taille de l'espace mémoire allouée à un tableau statique est déterminée par le compilateur lors de la compilation du programme. Ainsi si les dimensions du tableau ne sont pas connus lors de la compilation, ou s'ils doivent changer durant l'exécution du programme, le tableau en question doit être dynamique.

Pour créer un tableau dont la taille varie, il faut allouer soi-même la mémoire avec l'opérateur `new`. Nous avons alors un *tableau dynamique*.



Tableaux dynamiques (suite)

30

- Si on se limite à l'allocation statique, on doit limiter la flexibilité du programme

```
int main() {
    const unsigned int N = 10;
```

```
    do {
        cout << "Nombre de valeurs ? ";
        cin >> n;
    } while ( n > N );
```

```
    int t[ N ]; // créer un tableau approprié
```

```
    for ( int i = 0; i < n; i++ )
        cin >> t[ i ];
    ...
}
```

Le programme ne pourra jamais traiter plus de 10 valeurs

Marco Lavoie

14728 ORD - Langage C++

Sans l'allocation dynamique de mémoire aux tableaux, il est impossible d'écrire un programme qui ajuste la taille d'un tableau statique aux besoins de l'utilisateur. Par exemple, le programme ci-contre ne permet pas de gérer plus de valeurs que ce que permet la taille du tableau `t` (soit 10 valeurs maximum).

Cette restriction est inhérente aux tableaux statiques puisque c'est la responsabilité du compilateur (lors de la compilation du programme) de déterminer la quantité de mémoire que sera allouée au tableau lors de l'exécution du programme. Puisque le compilateur n'a aucun moyen de connaître les besoins des futurs utilisateurs en termes de quantité de valeurs à traiter, le programmeur doit allouer suffisamment de mémoire au tableau pour les besoins maximum des utilisateurs (`N = 10` valeurs maximum dans l'exemple ci-contre).

Les tableaux dynamiques permettent de contourner cette restriction.



Tableaux dynamiques (suite)

31

- L'allocation dynamique permet d'allouer la mémoire à `t` seulement lorsque la valeur de `n` est connue

```
int main() {
    unsigned int n;

    cout << "Nombre de valeurs ? ";
    cin >> n;

    int *t = new int[ n ]; ← Aucune limite de nombre de
                             valeurs à traiter (sauf celle
                             imposée par la quantité de RAM
                             dans l'ordinateur)

    for ( int i = 0; i < n; i++ )
        cin >> t[ i ];
    ...
    delete [] t;
}
```

Marco Lavoie

14728 ORD - Langage C++

Puisque la mémoire allouée au tableau est obtenue via l'opérateur `new`, il est possible pour un programme de déterminer au préalable la quantité de mémoire requise par le tableau avant l'allocation de cette mémoire.

Dans l'exemple ci-contre, le programme demande premièrement à l'utilisateur le nombre de valeurs à traiter (ce nombre est stocké dans la variable `n`), puis alloue la quantité de mémoire requise au tableau `t` pour lui permettre de stocker ces valeurs. En conséquence, le tableau `t` est dimensionné selon les besoins de l'utilisateur.

Notez que la valeur `n` fournie dans l'instruction `new int[n]` n'indique pas le nombre d'octets à allouer, mais plutôt le nombre d'éléments à stocker. Ainsi, dans l'exemple ci-contre, si l'utilisateur entre 20 pour `n`, alors 80 octets seront allouer au tableau `t` afin de stocker les 20 entiers non signés (à raison de 4 octets chacun sur une architecture 32 bits).



Dangers de l'allocation dynamique

32

- Lorsqu'on fait un `new`, le programmeur ne doit pas oublier d'éventuellement faire le `delete` correspondant
 - Sinon il y aura *fuite de mémoire*

```
{
    Date d( 12, 11, 1998 );
    d.afficher();

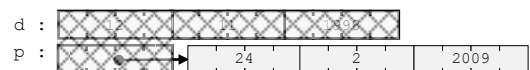
    Date *p;
    p = new Date( 24, 2, 2009 );
    p->afficher();
}
```

jour : 24
mois : 2
annee : 2009

Marco Lavoie

14728 ORD - Langage C++

Dans l'exemple ci-contre, le programme crée deux objets de type de la classe `Date` : un objet statique stocké dans la variable `d` ainsi qu'un objet dynamique (à l'aide de l'opérateur `new`) dont l'adresse est stockée dans la variable pointeur `p`. Lorsque l'exécution atteint la fin du bloc de code ci-contre, les variables automatiques `d` et `p` sont détruites, mais le bloc mémoire dont l'adresse est stockée dans `p` ne l'est pas :



Les programmeurs inexpérimentés (et même parfois ceux ayant de l'expérience en C++) oublient fréquemment d'invoquer l'opérateur `delete` pour libérer l'espace mémoire alloué dynamiquement à une variable pointeur. Dans l'exemple ci-dessus, le bloc mémoire dynamique est irrémédiablement perdu (sans être libéré) puisque son adresse (qui était stockée dans `p`) est perdue à jamais.



Exercice 7.2

33

- Considérez la classe de l'exercice 7.1 modifiée :

```
class Donnees {
public:
    Donnees( int = 100 ); // constructeur par défaut
    Donnees( double *, int, int = 100 ); // Constructeur paramétré

    int ajout( double ); // ajoute une donnée à valeurs[]
    double moyenne() const; // calcule la moyenne des données

private:
    double *valeurs; // stockage de données
    int nbValeurs; // nombre de données dans valeurs[]
    int szValeurs; // taille du tableau valeurs[]
};

int main() {
    double v[] = { 3.2, 4.6, 3.9 };
    Donnees data( v, 3 );
    data.ajout( 1.7 ).ajout( 9.4 );
    cout << data.moyenne() << endl;
    return 0;
}
```

Marco Lavoie

14728 ORD - Langage C++

Il y a deux faits à noter dans le code source fourni pour cet exercice :

- La classe `Donnees` dispose d'un attribut membre pointeur `valeurs` pour stocker les valeurs fournies via sa fonction membre publique `ajout`. Conséquemment, de la mémoire doit être dynamiquement allouée pour cet attribut via les constructeurs de la classe. Ceux-ci reçoivent en argument le nombre maximum de valeurs pouvant être stockées dans l'attribut `valeurs` (par défaut, l'attribut `valeurs` stockera jusqu'à 100 valeurs). N'oubliez pas d'ajouter un destructeur à la classe afin de libérer l'espace mémoire alloué dans les constructeurs.
- Le type de valeur de retour de `ajout` doit être modifié afin de permettre le chaînage des invocations de la fonction membre.

Membres de classe `static`

34

- Les membres d'une classe peuvent être déclarés `static`
 - Attention : ceci n'a rien à voir avec l'allocation statique
- Un attribut membre `static` permet à toutes les instances d'une même classe de partager le même attribut
 - Contrairement aux attributs membres conventionnels où chaque instance a sa propre copie de l'attribut

Marco Lavoie

14728 ORD - Langage C++

Tout objet d'une classe possède sa propre copie de tous les attributs membres de la classe (p.ex. chaque instance de la classe `Date` possède ses propres valeurs indépendantes pour les attributs `jour`, `mois` et `annee`). Il peut cependant arriver, selon les besoins, qu'une seule et même copie d'un attribut membre doive être partagée par toutes les instances de la classe. Dans ce cas (et aussi pour certaines autres raisons que nous verrons plus tard), on peut utiliser un attribut membre `static` représentant des informations liées à la classe elle-même plutôt qu'à chacun de ses objets. Il s'agit alors d'une propriété de la classe et non un objet spécifique de la classe.

Nous verrons aussi que non seulement des attributs membres de classe peuvent être désignés `static`, il en est de même pour les fonctions membres.

La déclaration d'un membre statique commence par le mot-clé `static`.

Attributs membres `static`

35

Exemple

```
class Test {
public:
    Test( int = 0 );
    static int nbInstances();
private:
    int x;
    static int n;
};

// Initialiser le membre statique
int Test::n = 0;

Test::Test( int a ) {
    x = a;
    n++; // Compter les instances
}

int Test::nbInstances() {
    return n;
}
```

```
int main() {
    for ( int i = 0; i < 100; i++ ) {
        Test t;
    }

    cout << "Nombre d'instances créés = "
    << Test::nbInstances() << endl;

    return 0;
}
```

- L'attribut `static` `n` est accessible par les instances et par le public via une fonction membre `static` (`nbInstances`), et ce même sans passer par une instance

Marco Lavoie

14728 ORD - Langage C++

Examinons l'exemple ci-contre. L'objectif d'utiliser l'attribut membre statique `n` dans la classe `Test` est de compter le nombre d'objets de type `Test` qui ont été créés (statiquement ou dynamiquement) par le code client de la classe (soit le programme principal `main`).

On constate que le constructeur de la classe `Test` incrémente la valeur de l'attribut `n`. Puisque `n` est un attribut membre `static` de `Test`, tous les objets créés de type `Test` partagent le même attribut `n`, donc la valeur de cet attribut correspond au nombre d'instance de `Test` créées durant l'exécution du programme.

Faits à noter concernant cet exemple :

- Puisqu'il n'est pas permis d'initialiser un attribut directement dans la définition de la classe, et qu'un attribut `static` ne peut pas être initialisé dans les constructeurs de la classe puisqu'il serait réinitialisé à chaque instanciation d'objet, il doit être initialisé à l'extérieur de la portée de la classe (par `int Test::n = 0;` dans notre exemple).

Attributs membres `static` (suite)

36

- Attention : une fonction membre statique peut s'exécuter sans faire référence à un objet instancié de la classe

```
int main() {
    for ( int i = 0; i < 100; i++ ) {
        Test t;
    }

    cout << "Nombre d'instances créés = "
    << Test::nbInstances() << endl;

    return 0;
}
```

- Conséquence : on ne peut pas faire référence au pointeur `this`

```
int Test::nbInstances() {
    cout << this->x; // X ←- Puisque nbInstances peut être
    return n;         // invoquée sans objet, le pointeur this
                    // n'est pas accessible
}
```

Marco Lavoie

14728 ORD - Langage C++

- Puisque `n` est d'accès `private` dans la classe `Test`, le programme principal n'a pas accès directement à l'attribut. Une fonction membre (`nbInstances`) est donc fournie par la classe `Test` pour donner accès à la valeur de l'attribut statique `n`. Notez que cette fonction membre est aussi désignée `static`, ce qui signifie qu'elle peut être invoquée sans passer par un objet de la classe `Test`, tel que le fait l'instruction `Test::nbInstances()` dans le programme principal.
- Une fonction membre désignée `static` pouvant être invoquée sans passer par un objet, il est donc interdit de faire référence au pointeur objet `this` dans le corps de ces fonctions membres (puisque cet objet n'existe pas dans le cas de l'invoque de `Test::nbInstances()` dans `main`).

assert

37

- Macro **assert** : utilitaire facilitant le débogage de programmes C++
 - Cette macro (exploitée par le précompilateur et le compilateur) permet d'interrompre l'exécution du programme si une condition est fausse
 - **Assert** signifie « valider l'hypothèse »
 - Lorsque le programmeur fait une hypothèse, il peut s'assurer qu'elle est toujours respectée via un **assert**

Marco Lavoie

14728 ORD - Langage C++

La macro **assert** (qui signifie « affirmer »), définie dans le fichier d'en-tête **<cassert>**, teste la valeur d'une condition. Si la valeur de cette condition est **false**, **assert** émet un message d'erreur et invoque la fonction **abort**, du fichier d'en-tête des utilitaires généraux **<cstdlib>**, afin de terminer l'exécution du programme. Si par contre la condition de l'assertion est **true**, le programme continue son exécution, sans interruption.

Il s'agit d'un outil de débogage pratique qui vérifie si une variable possède une valeur appropriée. La macro **assert** permet ainsi d'intégrer au code source du programme l'assertion du programmeur quant à la valeur attendue de cette variable. Si celle-ci ne contient pas la valeur attendue, le programme cessera immédiatement de s'exécuter.

assert (suite)

38

- Exemple d'application
- Comment s'assurer que la fonction ne sera pas appelée avec 0 comme argument pour **b**
 - Vérifier et interrompre l'exécution

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    return a - ( a / b ) * b ;
}

// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    if ( b == 0 )
        halt(); // Arrêter l'exécution
    return a - ( a / b ) * b ;
}
```

Marco Lavoie

14728 ORD - Langage C++

Voici un exemple d'utilisation de la macro **assert**. La fonction **monModulo** suppose que l'argument fourni au paramètre **b** ne doit jamais être 0. Pour s'en assurer, le programmeur a inséré une structure conditionnelle en début de fonction afin d'interrompre l'exécution du programme si jamais la fonction **monModulo** est invoquée avec une valeur de zéro pour **b** (afin d'éviter une division par zéro).

Puisque le programmeur ayant conçu la fonction **monModulo** a émis l'hypothèse que sa fonction ne sera jamais invoquée avec 0 comme valeur pour **b**, il est théoriquement impossible que la fonction **monModulo** cause l'interruption de l'exécution du programme. Cependant, afin de s'assurer que son hypothèse soit respectée, le programme incorpore une structure conditionnelle dans sa fonction afin d'empêcher tout code client de l'invoquer avec zéro comme second argument.

assert (suite)

39

- La stratégie du **halt()** est valide, mais elle ralentit l'exécution
 - La structure **if** est exécutée à chaque invocation de **monModulo**
 - Remplacer la structure par un **assert**
- Si jamais **monModulo** est invoquée avec **b = 0**, alors un **halt()** sera automatiquement exécuté

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    assert( b != 0 ); // b ne doit pas être 0
    return a - ( a / b ) * b ;
}
```

Marco Lavoie

14728 ORD - Langage C++

Même si la stratégie du programmeur pour imposer son hypothèse est fonctionnelle, elle a le désavantage d'ajouter une structure conditionnelle à la fonction **monModulo**, ce qui a un impact négatif sur les performances de la fonction lors de l'exécution du programme. Cette dégradation de performance est cependant inacceptable puisque **monModulo** ne sera probablement jamais invoqué avec 0 comme valeur pour **b**. C'est dans de telles circonstances que la macro **assert** prouve son utilité.

assert (suite)

40

- Si `assert` vérifie aussi la condition et fait un `halt()`, quel est l'avantage ?
 - Tous les compilateurs C++ ont une option pour désactiver les `assert`
 - Ainsi, lorsque le programme aura été testé pour s'assurer qu'il n'invoque jamais `monModulo` avec `b = 0`, on pourra désactiver les `assert` et le code exécutable ne contiendra plus ceux-ci
 - Si on utilise des structures `if` à la place des `assert`, le programmeur devra manuellement enlever toutes ces `if` (ou les commenter) afin de s'en débarrasser

Marco Lavoie

14728 ORD - Langage C++

Le rôle de la macro `assert` invoquée dans le code ci-contre est donc de s'assurer que la fonction `monModulo` n'est jamais invoquée par le code client avec 0 comme valeur de second paramètre. Durant le processus de débogage du programme on s'assure ainsi que l'hypothèse `b != 0` est toujours respectée.

Une fois le débogage de l'application terminé, rien n'oblige le retrait des assertions d'un programme. Lorsque les assertions ne sont plus requises à des fins de débogage dans un programme, l'insertion de la ligne

```
#define NDEBUG
```

au début du fichier programme indique au précompilateur d'ignorer toutes les assertions; le programmeur n'a donc pas besoin de supprimer manuellement chacune des assertions. Par ailleurs, le compilateur offre habituellement parmi ses propres options la possibilité de désactiver les assertions.

assert (suite)

41

- Exemple de désactivation
 - Utilisation de structures `if`

```
// Calcul de a % b. Attention: b ne doit pas être 0
int monModulo( int a, int b ) {
    // if ( b == 0 )
    //   halt(); // Arrêter l'exécution

    return a - ( a / b ) * b ;
}
```
 - Utilisation de `assert`
 - Désactiver l'option du compilateur puis recompiler (sans enlever les `assert` dans le code source)
 - La commande de précompilateur suivante désactive aussi les `assert` :

```
#define NDEBUG
```

Marco Lavoie

14728 ORD - Langage C++

En désactivant les assertions lors de la compilation du programme, le compilateur ignore toutes les invocations de la macro `assert` dans le code source du programme (au même titre qu'il ignore les commentaires par exemple). Conséquemment l'évaluation des conditions d'assertions ne se retrouvent pas dans le code exécutable du programme, ne causant ainsi aucune dégradation des performances de celui-ci.

De plus, puisque les invocations à `assert` demeurent dans le code source, le programmeur aura tout le loisir de les réactiver au besoin ultérieurement si le besoin est.

Erreurs de programmation

42

- Invoquer une fonction membre non `const` sur un objet `const`
- Absence d'initialiseur de membre `const` dans un constructeur
- Utilisateur de l'opérateur point (`.`) sur une variable pointeur
 - L'utilisation de l'opérateur flèche (`-->`) sur une variable d'instance statique ou sur un pointeur non initialisé à une instance dynamique
- Oublier de faire correspondre un `delete` à un `new`
- Référence au pointeur `this` dans une fonction membre statique
 - Ou déclarer `const` une fonction membre statique

Marco Lavoie

14728 ORD - Langage C++

La plus grande erreur de programmation généralement occasionnée par les programmeurs en C++ est sans contredit les *fuites de mémoire*, c'est-à-dire l'allocation dynamique d'un bloc de mémoire (avec `new`) sans le libérer ultérieurement (avec `delete`). De telles erreurs sont tellement fréquentes qu'elles se retrouvent parfois dans des logiciels achetés sur étagère. Il est généralement possible de détecter ces erreurs lors de l'utilisation du logiciel : la quantité d'espace mémoire disponible pour les autres applications en fonction de l'ordinateur diminue graduellement durant l'utilisation de l'application défectueuse, menant éventuellement à une défaillance des autres applications par manque de mémoire, pouvant mener même jusqu'à la défaillance générale de l'ordinateur, exigeant alors un réamorçage de celui-ci.



Bonnes pratiques de programmation

43

- Déclarer `const` toutes les fonctions membres ne modifiant pas les attributs membres de l'instance
- Placer les relations d'amitié (`friend`) en premier dans la classe
 - Aucun identificateur d'accès n'est requis pour les prototypes `friend` car ils ne s'y appliquent pas
- Après avoir détruit une instance dynamique (avec `delete`), remettre le pointeur à 0

```
Date *p = new Date( 24, 2, 2009 );
...
delete p;
p = 0;
```

Marco Lavoie

14728 ORD - Langage C++

Pour faire suite à la discussion précédente sur les défaillances d'applications causées par les fuites de mémoire, il est important de toujours s'assurer qu'un bloc de mémoire alloué par `new` sera éventuellement libéré par un `delete` dans le code source du programme. Une telle précaution permet d'éviter bien des désagréments aux utilisateurs de notre programme.



Devoir #6

44

- Solutionnez le problème distribué par l'instructeur
 - Créer un nouveau projet console *Visual Studio C++*
 - Solutionner le problème tel que décrit, avec la spécification supplémentaire suivante
 - Le code source de votre projet doit être réparti dans trois fichiers : `CompteEpargne.h`, `CompteEpargne.cpp` et `Devoir_6.cpp`
 - L'instructeur vous fournit un programme principal (i.e. le fichier `Devoir_6.cpp`) qui vous permettra de tester votre classe `CompteEpargne`
 - N'oubliez pas les conventions d'écriture
 - Respectez l'échéance imposée par l'instructeur
 - Soumettez votre projet selon les indications de l'instructeur
 - **Attention** : respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, sinon la note EC sera attribuée à ceux-ci

Marco Lavoie

14728 ORD - Langage C++

Dans votre solution à ce devoir, prenez en compte que toutes les instances de la classe `CompteEpargne` partagent le même taux d'intérêt annuel. Conséquemment, l'attribut membre `tauxInteretAnnuel` de la classe `CompteEpargne` doit être désigné `static`.

Par contre, puisque chaque instance de `CompteEpargne` dispose de son propre solde, l'attribut membre `soldeEpargne` ne doit pas être désigné `static`.



Pour la semaine prochaine

45

- Vous devez relire le contenu de la présentation du chapitre 7
 - Il y aura un `quiz` sur ce contenu au prochain cours
 - À livres et ordinateurs fermés
 - Profitez-en pour réviser le contenu des chapitres précédents

Marco Lavoie

14728 ORD - Langage C++

En début de classe la semaine prochaine vous aurez à répondre à des questions sur le C++ sans consultation du matériel pédagogique. Vous êtes donc fortement encouragé à relire les notes de cours du chapitre 7.

Profitez-en pour réviser le contenu des chapitres précédents.