La surcharge des opérateurs

Introduction

Comme nous l'avons vu précédemment, **les nouveaux types de données créés en C++ doivent avoir le même statut que les types prédéfinis (→ abstraction)**.

Pour cela, il faudrait pouvoir utiliser les **opérateurs classiques** (lorsque cela a un sens) comme l'addition « + », la soustraction « - », etc… sur nos nouveaux types de données.

Cela est bien entendu possible en C++. Comment ? Simplement parce que les opérateurs ne sont jamais que des fonctions particulières.

Ainsi, il est possible de compléter nos types de données en définissant les opérateurs que l'on désire surcharger.

Un opérateur est vu en C++ **comme une fonction particulière**.

Le nom de cette fonction est un peu particulier : il est composé du **mot « operator » suivi du symbole de l'opérateur**.

Exemples :

Opérateur	Signification
Type_retourné **operator+**(argument)	Permet d'additionner un type donné à un objet de notre classe
Type_retourné **operator-**(argument)	Permet de soustraire un type donné à un objet de notre classe
…	…

Le **polymorphisme** s'applique aussi aux opérateurs. Il est donc possible de **surcharger de plusieurs manières un opérateur donné** pour nos types de données, si cela a un sens.

Remarque :

Tous les opérateurs ne peuvent être surchargés. Voici la liste des opérateurs ne pouvant être surchargés :

Opérateur	Signification
.	Permettre l'accès aux membres d'une classe
.*	Permettre l'accès aux membres d'une classe
::	« opérateur » de résolution de portée : il s'agit plutôt d'un type
?:	Opérateur ternaire pour lequel le C++ n'a pas prévu de surcharge
sizeof	Obtenir la taille occupée en mémoire centrale : est-ce vraiment un opérateur ?

De plus,

- Il n'est **pas possible de créer de nouveaux opérateurs**
- Un opérateur originellement binaire (unaire) doit rester binaire (unaire) lors d'une surcharge. Sa syntaxe générale est maintenue.
- **Les ordres des priorités des opérateurs restent inchangés**.

Une classe date

```
…
#define NUMJ unsigned int
#define NUMM unsigned int
#define NUMA unsigned int
…
class cdate
{
  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
    cdate();
    cdate(NUMJ j, NUMM m, NUMA a): jour(j), mois(m), annee(a) {}
    cdate(NUMJ j, NUMM m);
    cdate(NUMJ j);
    cdate(NUMJ j, char *m, NUMA a);
```

```
    cdate(char *d);
    NUMJ getJour() const { return jour; }
    NUMM getMois() const { return mois; }
    NUMA getAnnee() const { return annee; }
    void affiche();
};

… // Definitions rapportees des methodes

int main()
{
  cdate today, // sous-entendu date courante
  d1(23,1,1958),
  d2(2,5), // sous-entendu annee courante
  d3(4), // sous-entendu mois et annee courants
  d4(2,"JAN",1996),
  d5("3/7/92");

  cout << "aujourd'hui : "; today.affiche();
  cout << "d1 : "; d1.affiche();
  cout << "d2 : "; d2.affiche();
  cout << "d3 : "; d3.affiche();
  cout << "d4 : "; d4.affiche();
  cout << "d5 : "; d5.affiche();

  return 0;
}
```

Surcharge de l'opérateur =

Tout comme pour les constructeurs par défaut et de copie, **il existe par défaut** un opérateur d'affectation pour chaque classe que l'on crée.

Mais… Il **se contente d'affecter membre à membre** chaque variable de la classe.

- Cela convient lorsqu'il n'y a pas de variables membres pointeurs ou d'un type plus complexe défini par l'utilisateur. Dans un tel cas, il importe le plus souvent de redéfinir cet opérateur.

Syntaxe des surcharges d'opérateurs membres :

type_retourné **operator***symbole* (argument) { corps_fonction ; }

ou encore, dans le cas d'une définition rapportée de fonction :

type_retourné classe::**operator***symbole* (argument) { corps_fonction ; }

Attention ! Pour un opérateur binaire, la fonction ne prend qu'un seul argument : **la première opérande est en fait l'objet qui appelle l'opérateur**.

Par exemple, si A,B,C sont trois objets instanciant la classe « MaClasse », alors

```
A = B ;
```

équivaut à

```
A.operator=(B) ;
```

Et

```
A = B + C ;
```

équivaut à

```
A.operator=(B.operator+(C));
```

Un premier opérateur = "maladroit" :

```
class cdate
{
  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
    cdate();
    …
    void operator=(cdate d);
};

void cdate::operator=(cdate d)
{
  jour = d.jour;
  mois = d.mois;
  annee = d.annee;
}
```

Ce qui permettra d'écrire sans problème :

```
d3 = d1 ;      // ce qui équivaut à d3.operator=(d1) ;
```

mais pas :

```
d3 = d1 = d2 ;      // ce qui équivaut à d3.operator=(d1.operator=(d2)) ;
```

parce que l'affectation d1 = d2 ne renvoie pas une **date** (un objet « **cdate** » plus précisément) mais un **void** !

Un opérateur = presque complet :

```
class cdate
{
  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
    cdate();
    …
    cdate operator=(cdate d);
};

cdate cdate::operator=(cdate d)
{
  jour = d.jour;
  mois = d.mois;
  annee = d.annee;

  return (*this);
  //on retourne l'objet courant, grace au pointeur this, par valeur
}
```

En réalité, l'**affection du C++ renvoie plutôt une référence** qu'une valeur. Ce qui donnera comme version presque finale :

```
class cdate
{
  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
```

```
    cdate();
    …
    cdate& operator=(cdate& d);
};

cdate& cdate::operator=(cdate& d)
{
  jour = d.jour;
  mois = d.mois;
  annee = d.annee;

  return (*this);
//on retourne l'objet courant, grace au pointeur this, par reference
}
```

Exemple d'utilisation :

```
int main()
{
  cdate today, // sous-entendu date courante
  d1(23,1,1958),
  d2(2,5), // sous-entendu annee courante
  d3(4), // sous-entendu mois et annee courants
  d4(2,"JAN",1996),
  d5("3/7/92");

  d3 = d1;
  d4 = today;
  d5 = d2 = today;
  cout << "aujourd'hui : "; today.affiche();
  cout << "d1 : "; d1.affiche();
  cout << "d2 : "; d2.affiche();
  cout << "d3 : "; d3.affiche();
  cout << "d4 : "; d4.affiche();
  cout << "d5 : "; d5.affiche();

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
aujourd'hui : 22/9/2014
d1 : 23/1/1958
d2 : 22/9/2014
d3 : 23/1/1958
d4 : 22/9/2014
d5 : 22/9/2014
bash-3.00$
```

L'opérateur = complet :

Les types prédéfinis, comme les float par exemple, admettent l'affectation d'une constante.

On peut écrire par exemple :

```
float x ;
x = 2.6352 ;
```

Si on essaie d'affecter une **date constante** à notre opérateur = actuel, le compilateur le refuse :

```
cdate d3 ;
d3 = cdate(25,12,95) ;
```

En effet,

```
bash-3.00$ g++ Date3.cxx
Date3.cxx: In function `int main()':
Date3.cxx:133: error: no match for 'operator=' in 'd3 = cdate(25u, 12u,
95u)'
Date3.cxx:35: note: candidates are: cdate& cdate::operator=(cdate&)
bash-3.00$
```

En fait, le compilateur refuse de créer une référence pour une constante, c'est-à-dire une variable temporaire, s'il n'est pas explicitement prévu que celle-ci ne pourra être modifiée.

- Il faut préciser à l'opérateur = que le deuxième opérande est une référence d'un objet constant.

```
cdate& cdate::operator=(const cdate& d)
{
  jour = d.jour;
  mois = d.mois;
  annee = d.annee;

  return (*this);
}
```

Le retour du mot réservé « delete »

Comme nous l'avons vu, le C++ nous créé **implicitement un opérateur =** pour toute classe que l'on créée, même si on ne le désire pas ! Dans certains cas cependant, on n'a pas envie d'autoriser l'affectation de nos objets. Pour éviter cela, on a de nouveau recours au mot réservé **delete**.

Soit la classe **MyClass** dont par son fichier **MyClass.h** :

```
class MyClass {
private:
    int a;

public:
    MyClass();
    MyClass(int);
    MyClass(const MyClass&);

    MyClass& operator=(const MyClass&) = delete; // interdit l'affection
};
```

et son fichier **MyClass.cpp** :

```
#include "MyClass.h"

MyClass::MyClass() {
    this->a = 0;
}

MyClass::MyClass(int a) {
    this->a = a;
}

MyClass::MyClass(const MyClass& obj) {
    this->a = obj.a;
}
```

où on observe que

- Dans le fichier **MyClass.h**, l'opérateur = a été décoré avec **« = delete »** afin de supprimer l'opérateur = implicitement créé par le C++
- Dans le fichier **MyClass.cpp**, l'opérateur = n'apparaît forcément pas

Soit le fichier de test **main.cpp** suivant :

```
#include "MyClass.h"
```

```
int main() {
      MyClass obj;
      MyClass obj2(3);
      MyClass obj3(obj2);

      obj = obj2;
      return 0;
}
```

La compilation de la classe **MyClass** et de ce fichier fournit :

```
[student@moon]$ g++ main.cpp MyClass.o -o main
main.cpp: Dans la fonction « int main() »:
main.cpp:8:8: error: utilisation de la fonction supprimée « MyClass&
MyClass::operator=(const MyClass&) »
  obj = obj2;
        ^~~~
In file included from main.cpp:1:
MyClass.h:10:14: note: déclaré ici
     MyClass& operator=(const MyClass&) = delete; // interdit l'affection
              ^~~~~~~~
[student@moon]$
```

où on observe que la compilation échoue vu que l'on tente d'exécuter l'opérateur = de l'objet obj.

Affecter autre chose qu'une date

Comme il est possible de surcharger une fonction en modifiant au moins un élément de sa liste de paramètres, on peut surcharger notre opérateur d'affectation pour gérer convenablement le cas où l'on veut affecter un entier à une date.

Actuellement, si on écrit

```
…
int main()
{
  cdate d3(25,12,95);
  d3 = 12;
  cout << "d3 = ";
  d3.affiche();

  return 0;
}
```

Le résultat affiché à l'écran (si on exécute le programme le 23/09/2014) ne sera pas le résultat espéré :

```
bash-3.00$ a.out
d3 = 12/9/2014
bash-3.00$
```

Le compilateur a fait appel au **constructeur de date ne réclamant qu'un seul paramètre pour créer un objet constant qu'il a affecté à d3** ☹ !

Définition de l'opérateur = affectant un entier à une date :

```
class cdate
{
  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
    cdate();
    …
    cdate& operator=(const cdate& d);
    cdate& operator=(const NUMJ& j);
};
cdate& cdate::operator=(const NUMJ& j)
{
  jour = j;
  return (*this);
}
```

L'exécution est à présent conforme à ce que l'on attend :

```
bash-3.00$ a.out
d3 = 12/12/95
bash-3.00$
```

- On peut donc définir pour une classe plusieurs opérateurs d'affectation selon la nature de l'objet à affecter.

Opérateur d'affectation / Constructeur de copie :

L'opérateur d'affectation présente certaines analogies avec le constructeur de copie. Mais,

- L'objet n'existe pas encore lorsque le constructeur de copie agit, alors qu'il existe déjà lors de l'affectation. En général, cette dernière réalise le travail d'un destructeur suivi de celui d'un constructeur de copie.

- L'opérateur d'affectation peut avoir une valeur de retour, alors qu'un constructeur n'en a pas.

- Le premier argument d'un constructeur de copie est une référence de la classe et il peut y avoir d'autres arguments. Par contre, l'opérateur d'affectation n'a qu'un seul argument, mais de type quelconque.

Exemples d'appel du constructeur de copie et/ou de l'opérateur d'affectation :

```
cdate d1(25,12,95), d3 ;
cdate d2(d1) ;           // Constructeur de copie

d3 = d2 ;                // opérateur d'affectation

d3 = cdate(25,8,95) ;    // création d'un objet temporaire constant
                         // puis appel de l'operateur d'affectation

d3 = cdate(d2) ;         // création d'un objet temporaire à partir
                         // du constructeur de copie puis appel de
                         // l'opérateur d'affectation

cdate d4=cdate(30,12,95) ;  // création d'un objet temporaire
                            // constant puis appel du constructeur
                            // de copie et non de l'opérateur
                            // d'affectation !!!
```

La surcharge de l'opérateur +

Il n'y a pas de sens à vouloir additionner deux dates. Par contre, additionner (ou soustraire) un nombre entier à une date peut avoir un sens. Par exemple, lorsqu'on reçoit une facture à une date déterminée, à payer dans les 40 jours, et que l'on souhaite connaître la date ultime de paiement.

Notre classe cdate devrait pouvoir permettre d'écrire :

```
cdate d1(23,11,1995), d ;

d = d1 + 40 ;
```

Il nous faut donc surcharger l'opérateur d'addition d'un entier dans notre classe cdate :

```
class cdate
{
  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;
  public :
    cdate();
    …
    cdate& operator=(const cdate& d);
    cdate& operator=(const NUMJ& j);
    cdate  operator+(int nj);
};

cdate cdate::operator+ (int nj)
{
  cdate d(*this);  // on crée un objet temporaire à partir de
                   // l'objet courant car celui-ci ne doit pas
                   // être modifié !
  if (nj < 0) return d;
  if (nj <= njmois[mois-1]-jour)
    d.jour += nj;  // on reste dans le mois
  else
  {
    …
    // changement de mois et d'année si nécessaire
    …
  }
  return d;
}
int main()
{
```

```
  cdate today;
  cout << "aujourd'hui : "; today.affiche();

  cdate d;
  d = today + 3;
  cout << "aujourd'hui + 3 : "; d.affiche();
  d = today + 365;
  cout << "aujourd'hui + 365 : "; d.affiche();
  d = d + 40;
  cout << "40 jours plus tard : "; d.affiche();

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
aujourd'hui : 22/9/2014
aujourd'hui + 3 : 25/9/2014
aujourd'hui + 365 : 22/9/2015
40 jours plus tard : 1/11/2015
bash-3.00$
```

Opérateurs membres et opérateurs libres : les fonctions amies

Il est possible de surcharger un opérateur au moyen d'une fonction « **libre** », c'est-à-dire **non-membre de la classe**.

Exemple :

```
cdate operator+(cdate d1,int nj)
{
  cdate d(d1);
  if (nj < 0) return d;
  if (nj <= cdate::njmois[d.mois-1]-d.jour)
    d.jour += nj; // on reste dans le mois
  else
  {
    …
  }
  return d;
}
```

Seulement, **le compilateur interdira l'accès aux variables membres privées** de l'objet d ! En effet,

```
bash-3.00$ g++ Date7.cxx
Date7.cxx: In function `cdate operator+(cdate, int)':
Date7.cxx:17: error: `unsigned int cdate::mois' is private
Date7.cxx:50: error: within this context
Date7.cxx:16: error: `unsigned int cdate::jour' is private
Date7.cxx:50: error: within this context
Date7.cxx:16: error: `unsigned int cdate::jour' is private
Date7.cxx:51: error: within this context
Date7.cxx:17: error: `unsigned int cdate::mois' is private
Date7.cxx:54: error: within this context
Date7.cxx:16: error: `unsigned int cdate::jour' is private
Date7.cxx:54: error: within this context
Date7.cxx:17: error: `unsigned int cdate::mois' is private
Date7.cxx:55: error: within this context
Date7.cxx:18: error: `unsigned int cdate::annee' is private
Date7.cxx:59: error: within this context
Date7.cxx:18: error: `unsigned int cdate::annee' is private
Date7.cxx:70: error: within this context
Date7.cxx:17: error: `unsigned int cdate::mois' is private
Date7.cxx:73: error: within this context
Date7.cxx:16: error: `unsigned int cdate::jour' is private
Date7.cxx:73: error: within this context
bash-3.00$
```

Pour résoudre ce problème, on peut :

- Utiliser les méthodes nécessaires pour accéder à ces données (méthodes get…() et set…()) : c'est un peu lourd ☹…

- Déclarer l'opérateur-fonction comme étant une « **fonction amie** » de la classe

Une **fonction amie** d'une classe possède les mêmes privilèges vis-à-vis de celle-ci que ses fonctions membres.

Elle peut donc accéder à sa partie privée. Il faut donc être prudent dans la création de telles fonctions afin de ne pas remettre en cause le principe de l'encapsulation.

Déclaration de la fonction amie :

```
class cdate
{
  friend cdate operator+(cdate d1,int nj);

  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
    cdate();
    …
    cdate& operator=(const cdate& d);
    cdate& operator=(const NUMJ& j);
};
```

Comment déterminer si un opérateur doit être membre ou ami ?

- Un **opérateur membre** assure que **la première opérande est forcément un objet de la classe** : aucune conversion n'est possible.

- Au contraire, **un opérateur libre** (ou ami) a pour **première opérande un argument de fonction normal**. Il peut donc être **quelconque**.

- **Un opérateur libre est souvent utilisé lorsque les opérandes sont des objets de type différent**, qui peuvent se présenter dans n'importe quel ordre.

Ainsi les deux versions de l'opérateur d'addition vu précédemment permettent d'écrire :

```
cdate d1(23,11,95) , d ;

d = d1 + 40 ;
```

mais pas

```
cdate d1(23,11,95), d ;

d = 40 + d1 ;
```

Car pour cela, il faudrait

- Soit un opérateur d'addition membre de la « classe entier » avec une date comme second opérande.

- Soit un opérateur libre prenant comme premier opérande un entier et comme second une date

Notre classe devient alors :

```
class cdate
{
  friend cdate operator+(cdate d1,int nj);
  friend cdate operator+(int nj,cdate d1);

  private :
    NUMJ jour;
    NUMM mois;
    NUMA annee;

  public :
    cdate();
    …
    cdate& operator=(const cdate& d);
    cdate& operator=(const NUMJ& j);
};

cdate operator+(int nj,cdate d1)
{
  return d1 + nj;  // Appel à l'autre version de la fonction qui
                   // effectue le calcul nécessaire
}
```

Dès lors,

- Un opérateur libre s'impose **pour une classe qu'il n'est pas possible de modifier**.

- Mais, **certains opérateurs ne laissent cependant pas le choix** : le langage C++ impose qu'ils soient membre de la classe. Il s'agit de := , () , [] , -> , new et delete.

Récapitulatif concernant les opérateurs membres et amis :

Les opérateurs binaires :

	Opérande gauche	**Opérande droite**
Opérateur fonction membre	Objet pointé par *this*	Premier paramètre
Opérateur fonction libre (amie)	Premier paramètre	Second paramètre

Les opérateurs unaires :

	La seule opérande
Opérateur fonction membre	Objet pointé par *this*
Opérateur fonction libre (amie)	Premier paramètre

D'autres surcharges :

- La surcharge des opérateurs de soustraction (**operator-**), de multiplication (**operator***) et de division (**operator/**) s'effectue de la même manière que celle de l'opérateur d'addition.

- La surcharge des **opérateurs de comparaison** nécessite de déterminer sur quels éléments (champs) celle-ci s'effectue.

Surcharge des opérateurs de comparaison pour la classe « cdate »

Si l'on désire pouvoir déterminer si une date est égale (**operator==**), plus grande (**operator>**) ou plus petite (**operator<**) qu'une autre, il suffit de surcharger chacun de ces opérateurs.

```
class cdate
{
  friend cdate operator+(cdate d1,int nj);
  friend cdate operator+(int nj,cdate d1);

  private :
    …
    int compD(const cdate& d);

  public :
    cdate();
    …
    cdate& operator=(const cdate& d);
    cdate& operator=(const NUMJ& j);
    int    operator<(const cdate& d);
    int    operator>(const cdate& d);
    int    operator==(const cdate& d);
};
```

Il s'agit de comparer deux dates entre elles. Il est donc logique d'en faire des **opérateurs membres**. Les fonctions possèdent donc un seul argument : la date faisant office de second opérande. Elles retournent un entier permettant de déterminer si la réponse à la comparaison est positive ou négative.

Définition des opérateurs :

```
int cdate::operator<(const cdate& d)
{
  return compD(d)==-1;
}

int cdate::operator>(const cdate& d)
{
  return compD(d)==1;
}

int cdate::operator==(const cdate& d)
{
  return compD(d)==0;
}
int cdate::compD(const cdate& d)
```

```
{
  if (annee < d.annee) return -1;
  if (annee > d.annee) return 1;
  // meme annee
  if (mois < d.mois) return -1;
  if (mois > d.mois) return 1;
  // meme mois
  if (jour < d.jour) return -1;
  if (jour > d.jour) return 1;
  // dates egales
  return 0;
}
```

où **compD** est une fonction membre (**privée**, car utilisée uniquement par elle-même) de la classe cdate permettant d'effectuer la comparaison entre deux dates (comparaison de l'année, du mois si nécessaire puis du jour si nécessaire).

Exemple d'utilisation :

```
int main()
{
  cdate today, // sous-entendu date courante
  d1(23,1,1958),
  d2(2,5), // sous-entendu annee courante
  d3(4), // sous-entendu mois et annee courants
  d4(2,"JAN",1996),
  d5("3/7/92");

  cout << "aujourd'hui : "; today.affiche();
  cout << "d1 : "; d1.affiche();
  cout << "d4 : "; d4.affiche();
  cout << "d1 < d4 : " << (d1<d4) << endl;
  cout << "d1 > d4 : " << (d1>d4) << endl;
  cout << "d1 == d4 : " << (d1==d4) << endl;

  d4 = d1;
  cout << "d1 : "; d1.affiche();
  cout << "d4 : "; d4.affiche();
  cout << "d1 < d4 : " << (d1<d4) << endl;
  cout << "d1 > d4 : " << (d1>d4) << endl;
  cout << "d1 == d4 : " << (d1==d4) << endl;

  cout << "d2 : "; d2.affiche();
  cout << "d3 : "; d3.affiche();
  cout << "d3 < d2 : " << (d3<d2) << endl;
  cout << "d3 > d2 : " << (d3>d2) << endl;
  cout << "d3 = d2 : " << (d3==d2) << endl;

  return 0;
```

```
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
aujourd'hui : 22/9/2014
d1 : 23/1/1958
d4 : 2/1/1996
d1 < d4 : 1
d1 > d4 : 0
d1 == d4 : 0
d1 : 23/1/1958
d4 : 23/1/1958
d1 < d4 : 0
d1 > d4 : 0
d1 == d4 : 1
d2 : 2/5/2014
d3 : 4/9/2014
d3 < d2 : 0
d3 > d2 : 1
d3 = d2 : 0
bash-3.00$
```

L'opérateur++

L'opérateur de pré-incrémentation

On désire pouvoir écrire

```
cdate d1(25,10,95), demain ;

demain = ++d1 ;  // incrémentation de 1 jour avant affectation
```

Pour cela, il faut surcharger l'opérateur ++ :

```
cdate cdate::operator++() // pré-incrémentation
{
  (*this) = (*this) + 1;  // utilisation des opérateurs + et =
                          // redéfinis auparavant.
  return (*this);
}
```

L'opérateur de post-incrémentation

Cette fois, on désire pouvoir écrire :

```
cdate d1(25,10,95) , ajd ;

ajd = d1++ ;  // incrémentation de 1 jour après l'affectation
```

Pour cela, il faut surcharger l'opérateur ++ :

```
cdate cdate::operator++()
{
  cdate temp(*this);
  (*this) = (*this) + 1;
  return temp;
}
```

Cependant, **comment le compilateur peut-il différencier les deux versions de la fonction-operateur surchargée** ? **Elles ne peuvent avoir la même liste de paramètres** : on va fonc attribuer à la deuxième version (post-incrémentation) **un paramètre sans signification**, par exemple de type int, dont le seul rôle sera de permettre la différentiation recherchée.

Ce qui donne :

```
cdate cdate::operator++(int) // post-incrémentation
{
  cdate temp(*this);  // copie non modifee de l'objet courant
  (*this) = (*this) + 1;
  return temp;
}
```

Exemple d'utilisation :

```
int main()
{
  int j1,m1,a1,j2,m2,a2;
  cout << "une date : " << endl;
  cout << "jour (0 si fini) : ";cin >> j1;
  cout << "mois : "; cin >> m1;
  cout << "annee : "; cin >> a1;
  cdate d1(j1,m1,a1);
  cdate demain;
  demain = d1++;
  cout << "post demain : ";demain.affiche();d1.affiche();
  demain = ++d1;
  cout << "pre demain : ";demain.affiche();d1.affiche();
```

```
  return 0;
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ a.out
une date :
jour (0 si fini) : 24
mois : 9
annee : 2014
post demain : 24/9/2014
25/9/2014
pre demain : 26/9/2014
26/9/2014
bash-3.00$
```

Les opérateurs << et >> : histoires de flux

Nous avons vu que les entrées/sorties en C++ s'effectuent à l'aide de flux. Plus précisément dans le cas d'entrée clavier et de sortie écran, on utilise respectivement les flux cin et cout. En fait,

- le flux **cin est un objet instanciant la classe istream**
- le flux **cout est un objet instanciant la classe ostream**

Nous avons vu aussi qu'un des grands principes recherchés dans l'utilisation des flux est la souplesse d'utilisation. L'affichage des types prédéfinis s'effectue de manière identique et transparente à l'utilisateur de l'objet cout.

Exemple :

```
int main()
{
  int i=5 ;
  float f = 7.2 ;
  char *msg = "Hello world";

  cout << i << endl;
  cout << f << endl;
  cout << msg << endl;

  return 0 ;
}
```

Cela est rendu possible car on a **redéfini l'opérateur "<<" dans la classe ostream pour les principaux types de données prédéfinis**. Il est en de même avec l'opérateur ">>" dans la classe istream.

Pour que nos types de données aient le même statut que les types prédéfinis, on doit pouvoir écrire :

```
int main()
{
  cdate d ;
  cout << "Entrez une date : " << endl ;
  cin >> d ;
  cout << "La date entree est : " << d << endl ;
  return 0 ;
}
```

Cela revient à surcharger l'opérateur << (>>) de la classe ostream (istream) prenant un objet du type cdate comme deuxième opérande. **Cet opérateur peut-il être membre ? Non**, car nous n'avons pas accès au code de la classe ostream (istream). Il doit être « libre » (ou « ami).

Surcharge de l'opérateur d'insertion <<

L'instruction d'affichage « **cout << d ;** » fait intervenir :

- un objet de la classe **ostream** comme premier opérande
- l'objet de la classe **cdate** à afficher

En effet,

```
cout << d ;
```

est équivalent à

```
operator<<(cout,d) ;
```

De plus, si on veut pouvoir écrire « **cout << d << endl ;** », il faut que notre fonction-opérateur **retourne le flux courant, c'est-à-dire un objet de la classe ostream**.

En effet,

```
cout << d << endl;
```

est équivalent à

```
(operator<<(cout,d)).operator<<(endl) ;
```

Ce qui donne le code suivant pour notre opérateur :

```
class cdate
{
  friend ostream& operator<<(ostream& s,const cdate& d);
  …
};

ostream& operator<<(ostream& s,const cdate& d)
{
  s << d.jour << "/" << d.mois << "/" << d.annee;
  return s;
}
```

Surcharge de l'opérateur >>

On obtient de manière analogue

```
class cdate
{
  friend ostream& operator<<(ostream& s,const cdate& d);
  friend istream& operator>> (istream& s, cdate& d);
  …
};

istream& operator>> (istream& s, cdate& d)
{
  int j,m,a,erreur = 0;
  // Saisie de j,m,a et verification de leur valeurs
  …
  d.jour = j; d.mois = m; d.annee = a;
  return s;
}

int main()
{
  cdate d1;
  cout << "une date : " << endl;
  cin >> d1;
```

```
  cout << d1 << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
une date :
annee : 2014
mois : -4
<<<erreur>>>
9
jour : 26
26/9/2014
bash-3.00$
```

L'opérateur [ ]

Cet opérateur

- est un peu curieux car il se place **de part et d'autre** de la deuxième opérande
- **doit être membre de la classe** : la première opérande est donc un objet de la classe. La deuxième opérande, ainsi que la valeur retournée, peuvent être quelconque

Imaginons vouloir écrire

```
int main()
{
  cdate d(25,10,95);
  cout << "jour = d[0] = " << d[0] << endl;
  cout << "mois = d[1] = " << d[1] << endl;
  cout << "annee = d[2] = " << d[2] << endl;

  return 0;
}
```

Il nous faut donc surcharger l'opérateur [].

Ce qui donne :

```
class cdate
{
  …
  public :
    …
    int operator[](int i);
};

int cdate::operator[] (int i)
{
  switch (i)
  {
    case 0 : return jour;
    case 1 : return mois;
    case 2 : return annee;
    default : return jour;
  }
}
```

L'exécution du code précédent fournit

```
bash-3.00$ a.out
jour = d[0] = 25
mois = d[1] = 10
annee = d[2] = 95
bash-3.00$
```

Les castings

Enfin, il est possible de surcharger les opérations de castings en C++. En fait, l'opération de casting est considérée comme l'**application d'un opérateur unaire de conversion de la forme**. Cet opérateur est **toujours membre**.

La syntaxe est

```
operator type() ;
```

Exemple :

Si nous désirons pouvoir « caster » un objet date en un **entier** (retourne le jour) ou en une **chaîne de caractères** (retourne la date au format « jj/mm/aaaa »), notre classe deviendra :

```
class cdate
{
  …
  public :
    cdate();
    …
    operator int() { return jour; }
    operator char*();
};

cdate::operator char*()
{
  char* buf = new char[11];
  sprintf(buf,"%2d/%2d/%4d",jour,mois,annee);
  return buf;
}
```

Exemple d'utilisation :

```
int main()
{
  cdate d(25,10,1995);
  cout << "jour = " << (int)d << endl;
  cout << "chaine = " << (char*)d << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
jour = 25
chaine = 25/10/1995
bash-3.00$
```