Introduction au C/C++

Cours 7

Rémi Watrigant
(fortement inspiré du cours de M. Huchard de l'Université Montpellier 2)

Université de Nîmes

2013-2014

Opérateurs

En $C++$, les opérateurs n'ont pas le même comportement selon le type des valeurs utilisées :

- `3/2` $\longrightarrow$ division de valeurs entières (résultat $= 1$)
- `3./2.` $\longrightarrow$ division de valeurs flottantes (résultat $= 1.5$)

Opérateurs

En $C++$, les opérateurs n'ont pas le même comportement selon le type des valeurs utilisées :

- `3/2` $\longrightarrow$ division de valeurs entières (résultat $= 1$)
- `3./2.` $\longrightarrow$ division de valeurs flottantes (résultat $= 1.5$)

En fait les opérateurs sont des fonctions comme les autres :

- `int operator/(int, int)`
- `float operator/(float, float)`

`3/2` $\rightsquigarrow$ `operator/(3, 2)`

Opérateurs

En $C++$, les opérateurs n'ont pas le même comportement selon le type des valeurs utilisées :

- `3/2` $\longrightarrow$ division de valeurs entières (résultat $= 1$)
- `3./2.` $\longrightarrow$ division de valeurs flottantes (résultat $= 1.5$)

En fait les opérateurs sont des fonctions comme les autres :

- `int operator/(int, int)`
- `float operator/(float, float)`

`3/2` $\rightsquigarrow$ `operator/(3, 2)`

Elles ont été **SURCHARGEES**

Opérateurs

En $C++$, les opérateurs n'ont pas le même comportement selon le type des valeurs utilisées :

- `3/2` $\longrightarrow$ division de valeurs entières (résultat $= 1$)
- `3./2.` $\longrightarrow$ division de valeurs flottantes (résultat $= 1.5$)

En fait les opérateurs sont des fonctions comme les autres :

- `int operator/(int, int)`
- `float operator/(float, float)`

`3/2` $\rightsquigarrow$ `operator/(3, 2)`

Elles ont été **SURCHARGEES**

Surcharge

Fonction (resp. méthode) ayant le même nom qu'un autre fonction (resp. méthode) mais pouvant être distinguées par leur signature (type de retour, type des paramètres, nombre de paramètres).

Opérateurs

En $C++$, il est possible de surcharger les opérateurs

Opérateurs

En $C++$, il est possible de surcharger les opérateurs

On connait déjà des surcharges d'opérateurs :

```
string s1 = "cou";
string s2 = "cou!!!";
string s3 = s1 + s2;
s1 += s2;
if (s1 == s3) {
 cout « "les deux chaines sont egales" « endl;
 cout « "s1 = " « s1 « " s3 = " « s3 « endl;
}
```

Opérateurs

En $C++$, il est possible de surcharger les opérateurs

On connait déjà des surcharges d'opérateurs :

```
string s1 = "cou";
string s2 = "cou!!!";
string s3 = s1 + s2;
s1 += s2;
if (s1 == s3) {
 cout « "les deux chaines sont egales" « endl;
 cout « "s1 = " « s1 « " s3 = " « s3 « endl;
}
```

Opérateurs

En $C++$, il est possible de surcharger les opérateurs

On connait déjà des surcharges d'opérateurs :

```
string s1 = "cou";          operator=(char*)
string s2 = "cou!!!";
string s3 = s1 + s2;         operator+(string, string)
s1 += s2;                  operator+=(string)
if (s1 == s3) {          operator==(string, string)
 cout « "les deux chaines sont egales" « endl;
 cout « "s1 = " « s1 « " s3 = " « s3 « endl;
}          operator«(ostream, string)
```

Opérateurs

Opérateurs pouvant être surchargés :

```
 +      -      *      /      %      ^      &
 |      ~      !      =      <      >      +=
-=     *=     /=     %=     ^=     &=     |=
<<     >>     >>=    <<=    ==     !=     <=
>=     &&     ||     ++     --     ->*     ,
->     []     ()     new    delete type()
```

Opérateurs **ne pouvant pas** être surchargés :

```
.
.*
::
?:
sizeof
```

```
1  class Heure {
2  protected:
3        int nbHeures;
4        int nbMinutes;
5        int nbSecondes;
6  public:
7        Heure();
8        Heure(int, int int);
9        Heure(string);
10       ~Heure();
11       ...
12 };
```

```
1  class Heure {
2  protected:
3        int nbHeures;
4        int nbMinutes;
5        int nbSecondes;
6  public:
7        Heure();
8        Heure(int, int int);
9        Heure(string);
10       ~Heure();
11       ...
12 };
```

Avec la surcharge d'opérateurs :

```
1 Heure h1(2, 56, 23), h2(0, 4, 43);
2 Heure h3 = h1 + h2;
3 cout << h1 << " + " << h2 << " = " << h3 << endl;
```

$\Rightarrow$ 2h56min23s + 4min43s = 3h02min6s

Opérateur d'addition : `h1 + h2`

Opérateur d'addition : `h1 + h2`

Signature :

```
1 Heure operator+(const Heure& a, const Heure& b);
```

Opérateur d'addition : `h1 + h2`

Signature :

```
1 Heure operator+(const Heure& a, const Heure& b);
```

Implémentation :

```
1  Heure operator+(const Heure& a, const Heure& b) {
2      int nbSec = a.getNbSec() + b.getNbSec();
3      int nbMin = a.getNbMin() + b.getNbMin() + nbSec/60;
4      nbSec %= 60;
5      int nbH = a.getNbH() + b.getNbH() + nbMin/60;
6      nbMin %= 60;
7
8      Heure copie(nbH, nbMin, nbSec);
9      return copie;
10 }
```

Opérateur d'addition : `h1 + h2`

Signature :

```
1 Heure operator+(const Heure& a, const Heure& b);
```

Implémentation :

```
1  Heure operator+(const Heure& a, const Heure& b) {
2      int nbSec = a.getNbSec() + b.getNbSec();
3      int nbMin = a.getNbMin() + b.getNbMin() + nbSec/60;
4      nbSec %= 60;
5      int nbH = a.getNbH() + b.getNbH() + nbMin/60;
6      nbMin %= 60;
7
8      Heure copie(nbH, nbMin, nbSec);
9      return copie;
10 }
```

Où placer ce code ? Réponse :

Opérateur d'addition : `h1 + h2`

Signature :

```
1 Heure operator+(const Heure& a, const Heure& b);
```

Implémentation :

```
1  Heure operator+(const Heure& a, const Heure& b) {
2      int nbSec = a.getNbSec() + b.getNbSec();
3      int nbMin = a.getNbMin() + b.getNbMin() + nbSec/60;
4      nbSec %= 60;
5      int nbH = a.getNbH() + b.getNbH() + nbMin/60;
6      nbMin %= 60;
7
8      Heure copie(nbH, nbMin, nbSec);
9      return copie;
10 }
```

Où placer ce code ? Réponse : à peu près n'importe où ! mais...

Opérateur d'addition : `h1 + h2`

Signature : dans Heure.h (mais hors de la classe)

```
1 Heure operator+(const Heure& a, const Heure& b);
```

Implémentation : dans Heure.cpp (à la suite des autres méthodes)

```
1  Heure operator+(const Heure& a, const Heure& b) {
2      int nbSec = a.getNbSec() + b.getNbSec();
3      int nbMin = a.getNbMin() + b.getNbMin() + nbSec/60;
4      nbSec %= 60;
5      int nbH = a.getNbH() + b.getNbH() + nbMin/60;
6      nbMin %= 60;
7
8      Heure copie(nbH, nbMin, nbSec);
9      return copie;
10 }
```

Où placer ce code ? Réponse : à peu près n'importe où ! mais...

Intermède : mot clé **const** :

Intermède : mot clé **const** : se place :

- devant un paramètre de fonction
- à la fin d'une signature de méthode

Intermède : mot clé **const** : se place :

- devant un paramètre de fonction
- à la fin d'une signature de méthode

But : indique au compilateur que le paramètre ou l'objet ne va pas être modifié dans la fonction

Intermède : mot clé **const** : se place :

- devant un paramètre de fonction
- à la fin d'une signature de méthode

But : indique au compilateur que le paramètre ou l'objet ne va pas être modifié dans la fonction

```
1  class Heure {
2  protected:
3        int nbH;    int nbMin;      int nbSec;
4  public:
5        //constructeurs, destructeurs (...)
6        int getNbH() const;
7        int getNbMin() const;
8        int getNbSec() const;
9
10       void setNbH(const int);
11       void setNbMin(const int);
12       void setNbSec(const int);
13
14       void affiche() const;
15 };
```

Intermède : mot clé **const** : se place :

- devant un paramètre de fonction
- à la fin d'une signature de méthode

But : indique au compilateur que le paramètre ou l'objet ne va pas être modifié dans la fonction.

Permet de "passer un contrat" envers les prochains utilisateurs de la classe.

accélération lors de la compilation/exécution.

Opérateur d'addition : `h1 + h2`

Signature : dans Heure.h (mais hors de la classe)

```
1 Heure operator+(const Heure& a, const Heure& b);
```

Implémentation : dans Heure.cpp (à la suite des autres méthodes)

```
1  Heure operator+(const Heure& a, const Heure& b) {
2      int nbSec = a.getNbSec() + b.getNbSec();
3      int nbMin = a.getNbMin() + b.getNbMin() + nbSec/60;
4      nbSec %= 60;
5      int nbH = a.getNbH() + b.getNbH() + nbMin/60;
6      nbMin %= 60;
7
8      Heure copie(nbH, nbMin, nbSec);
9      return copie;
10 }
```

Où placer ce code ? Réponse : à peu près n'importe où ! mais...

Opérateur d'incrémentation : `h1 += h2`

Opérateur d'incrémentation : `h1 += h2`

Signature : (comme méthode de la classe Heure)

```
1 void operator+=(const Heure& b);
```

Opérateur d'incrémentation : `h1 += h2`

Signature : (comme méthode de la classe Heure)

```
1 void operator+=(const Heure& b);
```

Implémentation : dans Heure.cpp

```
1 void operator+=(const Heure& h) {
2     this->nbSec += h.getNbSec();
3     this->nbMin += h.getNbMin() + this->nbSec/60;
4     this->nbSec %= 60;
5     this->nbH += h.getNbH() = + this->nbMin/60;
6     this->nbMin %= 60;
7 }
```

Opérateur de flux : `cout « h1;`

Opérateur de flux : `cout « h1;`
En théorie : méthode de la classe `ostream`, mais impossible...
donc :

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 1 : comme simple fonction

Signature : dans Heure.h (mais hors de la classe, simple fonction)

```
1 ostream& operator<<(ostream&, const Heure&);
```

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 1 : comme simple fonction

Signature : dans Heure.h (mais hors de la classe, simple fonction)

```
1 ostream& operator<<(ostream&, const Heure&);
```

Implémentation : dans Heure.cpp

```
1 ostream& operator<<(ostream& flux, const Heure& h) {
2     flux << h.getNbH() << "h" << h.getNbMin() << "min" << h.ge
3     return flux;
4 }
```

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 1 : comme simple fonction

Signature : dans Heure.h (mais hors de la classe, simple fonction)

```
1 ostream& operator<<(ostream&, const Heure&);
```

Implémentation : dans Heure.cpp

```
1 ostream& operator<<(ostream& flux, const Heure& h) {
2     flux << h.getNbH() << "h" << h.getNbMin() << "min" << h.ge
3     return flux;
4 }
```

Incovénient : on pourrait avoir besoin d'utiliser des attributs privés de la classe dans la fonction

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 2 : comme fonction amie

Utilisation : dans Heure.h (toujours comme fonction, pas méthode)

```
1 class Heure {
2 protected:
3       int nbH;    int nbMin;      int nbSec;
4 public:
5       //constructeurs, destructeurs (...)
6       //getters, setters...
7
8      friend ostream& operator<<(ostream&, const Heure&);
9 };
```

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 2 : comme fonction amie

Utilisation : dans Heure.h (toujours comme fonction, pas méthode)

```
1 class Heure {
2 protected:
3       int nbH;    int nbMin;      int nbSec;
4 public:
5       //constructeurs, destructeurs (...)
6       //getters, setters...
7
8      friend ostream& operator<<(ostream&, const Heure&);
9 };
```

Permet à la fonction de pouvoir accéder aux attributs et méthodes privés de la classe Heure.

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 2 : comme fonction amie

Utilisation : dans Heure.h (toujours comme fonction, pas méthode)

```
1 class Heure {
2 protected:
3       int nbH;    int nbMin;      int nbSec;
4 public:
5       //constructeurs, destructeurs (...)
6       //getters, setters...
7
8      friend ostream& operator<<(ostream&, const Heure&);
9 };
```

Permet à la fonction de pouvoir accéder aux attributs et méthodes privés de la classe Heure.

Inconvénient : l'utilisation de fonctions amies témoigne souvent d'un problème de conception

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 3 : fonction + méthode

dans Heure.h

```
1  class Heure {
2  protected:
3        int nbH;    int nbMin;      int nbSec;
4  public:
5        //constructeurs, destructeurs (...)
6        //getters, setters...
7       virtual void affiche(ostream&) const;
8  };
9
10 ostream& operator<<(ostream&, const Heure&);
```

Opérateur de flux : `cout « h1;`

En théorie : méthode de la classe `ostream`, mais impossible...

donc :

Solution 3 : fonction + méthode

dans Heure.cpp

```
1 void Heure::affiche(ostream& flux) {
2       flux << h.getNbH() << "h" << h.getNbMin() << "min" << h
3 }
4
5 ostream& operator<<(ostream& flux, const Heure& h) {
6     h.affiche(flux);
7     return flux;
8 }
```

Avantages :

- permet d'accéder aux attributs privés
- permet de pouvoir redéfinir la fonction affiche dans des classes filles