

# Introduction à C++ comparé à Java

P. Chassignet

15 avril 2011

# Plan

## 1 Introduction

# Plan

- 1 Introduction
- 2 Principales différences entre Java et C++ (sauf objets)

# Plan

- 1 Introduction
- 2 Principales différences entre Java et C++ (sauf objets)
- 3 Objets

# Plan

- 1 Introduction
- 2 Principales différences entre Java et C++ (sauf objets)
- 3 Objets
- 4 S'il reste du temps

# Plan

- 1 Introduction
- 2 Principales différences entre Java et C++ (sauf objets)
- 3 Objets
- 4 S'il reste du temps
- 5 La suite

## 1 Introduction

- Historique
- Pourquoi utiliser C++
- Premier programme
- Similitudes entre Java et C++

# L'histoire de C

AT&T Bell Labs :

**1969** premier Unix, en assembleur sur PDP-7,  
... portage vers PDP-11 ...

**1972** C [Dennis Ritchie]

**1973** Unix en C

# L'histoire de C

AT&T Bell Labs :

**1969** premier Unix, en assembleur sur PDP-7,  
... portage vers PDP-11 ...

**1972** C [Dennis Ritchie]

**1973** Unix en C

**1978** manuel de référence :  
B. Kernighan, D. Ritchie, *The C Programming Language*  
= C K&R, toujours très répandu.

## L'histoire de C

AT&T Bell Labs :

**1969** premier Unix, en assembleur sur PDP-7,  
... portage vers PDP-11 ...

**1972** C [Dennis Ritchie]

**1973** Unix en C

**1978** manuel de référence :  
B. Kernighan, D. Ritchie, *The C Programming Language*  
= C K&R, toujours très répandu.

**1989** C ANSI ou C89

# L'histoire de C

AT&T Bell Labs :

**1969** premier Unix, en assembleur sur PDP-7,  
... portage vers PDP-11 ...

**1972** C [Dennis Ritchie]

**1973** Unix en C

**1978** manuel de référence :  
B. Kernighan, D. Ritchie, *The C Programming Language*  
= C K&R, toujours très répandu.

**1989** C ANSI ou C89

**1999** C ISO  
ouverture au calcul numérique  
en remplacement de Fortran ...

# L'histoire de C++

AT&T Bell Labs :

**1979** ...

**1983** C++, [Bjarne Stroustrup]  
= C + objets, héritage

**1985** B. Stroustrup, *The C++ Programming Language*

# L'histoire de C++

AT&T Bell Labs :

**1979** ...

**1983** C++, [Bjarne Stroustrup]  
= C + objets, héritage

**1985** B. Stroustrup, *The C++ Programming Language*

**1990** généricité, ...

**1994** STL : conteneurs, itérateurs, comparateurs, ...

# L'histoire de C++

AT&T Bell Labs :

**1979** ...

**1983** C++, [Bjarne Stroustrup]  
= C + objets, héritage

**1985** B. Stroustrup, *The C++ Programming Language*

**1990** généricité, ...

**1994** STL : conteneurs, itérateurs, comparateurs, ...

**1998** C++98 ISO

# L'histoire de C++

AT&T Bell Labs :

**1979** ...

**1983** C++, [Bjarne Stroustrup]  
= C + objets, héritage

**1985** B. Stroustrup, *The C++ Programming Language*

**1990** généricité, ...

**1994** STL : conteneurs, itérateurs, comparateurs, ...

**1998** C++98 ISO

**20 ...** C++0x

# L'histoire de C++

AT&T Bell Labs :

**1979** ...

**1983** C++, [Bjarne Stroustrup]  
= C + objets, héritage

**1985** B. Stroustrup, *The C++ Programming Language*

**1990** généricité, ...

**1994** STL : conteneurs, itérateurs, comparateurs, ...

**1998** C++98 ISO

**20 ...** C++0x

**2011** elle arrive !

# L'histoire de Java

Sun :

**1991** ...

# L'histoire de Java

Sun :

**1991** ...

**1995** Java 1.0

# L'histoire de Java

Sun :

**1991** ...

**1995** Java 1.0

...

**2004** Java 5.0 : génériques, `for ( : )`

# L'histoire de Java

Sun :

**1991** ...

**1995** Java 1.0

...

**2004** Java 5.0 : génériques, `for ( : )`

...

Pourquoi utiliser C++

## Pourquoi utiliser C++

- un langage relativement propre,
- des objets pour structurer,
- gestion précise de la mémoire,
- un peu moins rapide que C mais plus rapide que Java,
- un grand nombre de bibliothèques disponibles.

# programme Java

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

# programme C

C K&R :

```
#include <stdio.h>
main()
{
    printf("Hello World\n");
}
```

# programme C

## C K&R :

```
#include <stdio.h>
main()
{
    printf("Hello World\n");
}
```

## C ISO :

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0; // code d'erreur Unix
}
```

# programme C++

```
#include <iostream>
int main()
{
    std::cout << "Hello World\n";
}
```

on préfère :

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World" << endl;
}
```

**astuce** : cout << xxx **vaut** cout

# Similitudes entre Java et C++

- une grande part de syntaxe commune avec C (hors objets)
- concepts objets similaires (mais syntaxe différente)

## 2 Principales différences entre Java et C++ (sauf objets)

- Mise en œuvre
- Divers “détails”
- Préprocesseur
- Fonctions
- Tableaux

## Mise en œuvre

- en Java :
  - compilation : `javac Hello.java`
  - produit un fichier `Hello.class` (bytecode)
  - exécution : `java Hello`  
= interprétation de `Hello.main(String[] args)` dans la JVM
  - si plusieurs `.java` (au bon endroit) : `javac` et `java` se débrouillent

## Mise en œuvre

- en Java :
  - compilation : `javac Hello.java`
  - produit un fichier `Hello.class` (bytecode)
  - exécution : `java Hello`  
= interprétation de `Hello.main(String[] args)` dans la JVM
  - si plusieurs `.java` (au bon endroit) : `javac` et `java` se débrouillent
- en C/C++ :
  - prétraitement., compilation, édition de liens :  
`gcc bonjour.c` ou `g++ bonjour.cpp`
  - produit un fichier `a.out` (code machine exécutable)  
mais on peut changer son nom
  - commande `./a.out` : exécution de **la** fonction `main`
  - si plusieurs fichiers `.h` et `.c` ou `.cpp` + des bibliothèques,  
pour faire un seul `a.out` :  
il y a des utilitaires de type *make*, cf le TD

## Mise en œuvre

- en Java :
  - compilation : `javac Hello.java`
  - produit un fichier `Hello.class` (bytecode)
  - exécution : `java Hello`  
= interprétation de `Hello.main(String[] args)` dans la JVM
  - si plusieurs `.java` (au bon endroit) : `javac` et `java` se débrouillent
- en C/C++ :
  - prétraitement., compilation, édition de liens :  
`gcc bonjour.c` ou `g++ bonjour.cpp`
  - produit un fichier `a.out` (code machine exécutable)  
mais on peut changer son nom
  - commande `./a.out` : exécution de **la** fonction `main`
  - si plusieurs fichiers `.h` et `.c` ou `.cpp` + des librairies,  
pour faire un seul `a.out` :  
il y a des utilitaires de type *make*, cf le TD
- avec un IDE, c'est en principe plus simple !

Divers "détails"

## En C/C++, "au choix de l'implémentation" :

●  $-5/3 = ?$

Divers "détails"

## En C/C++, "au choix de l'implémentation" :

- $-5/3 = ?$
- taille des entiers `int` ?

## En C/C++, "au choix de l'implémentation" :

- $-5/3 = ?$
- taille des entiers `int` ?
- pseudo-opérateur `sizeof(type)` ou `sizeof(expr)`
- `short int` (ou `short` simplement) : 16 bits,
- `long int` (ou `long` simplement)
- `long long int` : généralement 64 bits,

## En C/C++, "au choix de l'implémentation" :

- $-5/3 = ?$
  
- taille des entiers `int` ?
- pseudo-opérateur `sizeof(type)` ou `sizeof(expr)`
- `short int` (ou `short` simplement) : 16 bits,
- `long int` (ou `long` simplement)
- `long long int` : généralement 64 bits,
  
- `sizeof(char) = 1` (taille en octets) bit de signe ?
- `signed char` : un octet, intervalle `-128..127`,
- `unsigned char` : un octet, intervalle `0..255`,

## En C/C++, "au choix de l'implémentation" :

- $-5/3 = ?$
- taille des entiers `int` ?
- pseudo-opérateur `sizeof(type)` ou `sizeof(expr)`
- `short int` (ou `short` simplement) : 16 bits,
- `long int` (ou `long` simplement)
- `long long int` : généralement 64 bits,
- `sizeof(char) = 1` (taille en octets) bit de signe ?
- `signed char` : un octet, intervalle `-128..127`,
- `unsigned char` : un octet, intervalle `0..255`,
- `cout << ( sizeof('a'=='b') == sizeof(int) );`  
ça fait quoi ?

Divers "détails"

## Pièges :

- ```
int i = 1;
if (i = 0) cout << i << endl;
cout << i << endl;
```

ça fait quoi ?

Divers "détails"

## Pièges :

- ```
int i = 1;
if (i = 0) cout << i << endl;
cout << i << endl;
```

ça fait quoi ?

ça écrit 0

Divers "détails"

## Pièges :

- ```
int i = 1;
if (i = 0) cout << i << endl;
cout << i << endl;
```

ça fait quoi ?

ça écrit 0

affecte 0 à i et vaut 0, ie. "faux".

## Préprocesseur

## Préprocesseur : lignes qui commencent par #

- #include <*fichier-emplacement-standard*>
- #include "*fichier-local*"

## Préprocesseur : lignes qui commencent par #

- `#include <fichier-emplacement-standard>`
- `#include "fichier-local"`
  
- constantes "à l'ancienne" :
- `#define PI 3.14159`
- `#define NEWLINE '\n'`
- = substitution de chaînes avant la compilation
- typiquement `#include "constantes.h"`

## Préprocesseur : lignes qui commencent par #

- `#include <fichier-emplacement-standard>`
- `#include "fichier-local"`
- constantes "à l'ancienne" :
  - `#define PI 3.14159`
  - `#define NEWLINE '\n'`
  - = substitution de chaînes avant la compilation
  - typiquement `#include "constantes.h"`
  - autrefois utile aussi pour des définitions manquantes ou pour "inliner" :
    - `#define neg(x) -x`  
mais `neg(a+b)` réécrit en `-a+b` raté !

## Préprocesseur : lignes qui commencent par #

- `#include <fichier-emplacement-standard>`
- `#include "fichier-local"`
- constantes "à l'ancienne" :
  - `#define PI 3.14159`
  - `#define NEWLINE '\n'`
  - = substitution de chaînes avant la compilation
  - typiquement `#include "constantes.h"`
  - autrefois utile aussi pour des définitions manquantes ou pour "inliner" :
    - `#define neg(x) -x`  
mais `neg(a+b)` réécrit en `-a+b` raté !
    - `#define neg(x) -(x)`

## Préprocesseur : lignes qui commencent par #

- `#include <fichier-emplacement-standard>`
- `#include "fichier-local"`
- constantes "à l'ancienne" :
  - `#define PI 3.14159`
  - `#define NEWLINE '\n'`
  - = substitution de chaînes avant la compilation
  - typiquement `#include "constantes.h"`
  - autrefois utile aussi pour des définitions manquantes ou pour "inliner" :
    - `#define neg(x) -x`  
mais `neg(a+b)` réécrit en `-a+b` raté !
    - `#define neg(x) -(x)`
    - `#define abs(x) ((x)<0?- (x) : (x))`

## Préprocesseur : lignes qui commencent par #

- `#include <fichier-emplacement-standard>`
- `#include "fichier-local"`
- constantes "à l'ancienne" :
  - `#define PI 3.14159`
  - `#define NEWLINE '\n'`
  - = substitution de chaînes avant la compilation
  - typiquement `#include "constantes.h"`
  - autrefois utile aussi pour des définitions manquantes ou pour "inliner" :
    - `#define neg(x) -x`  
mais `neg(a+b)` réécrit en `-a+b` raté !
    - `#define neg(x) -(x)`
    - `#define abs(x) ((x)<0?- (x) : (x))`  
mais `abs(f(...))` = double évaluation de `f(...)` !

## Fonctions :

- `int somme(int a, int b) { return a+b; }`
- `float somme(float a, float b) { return a+b; }`  
= surcharge
- comme en Java ?

```
int i = 1, j = 2;  
cout << somme(i, j) << endl;  
cout << somme(1.1, j) << endl;  
cout << somme(i, 2.2) << endl;  
cout << somme(1.1, (float)j) << endl;
```

## Fonctions :

- `int somme(int a, int b) { return a+b; }`
- `float somme(float a, float b) { return a+b; }`  
= surcharge
- comme en Java ?

```
int i = 1, j = 2;
cout << somme(i, j) << endl;
cout << somme(1.1, j) << endl;
cout << somme(i, 2.2) << endl;
cout << somme(1.1, (float)j) << endl;
```

- La déclaration doit toujours précéder l'utilisation :

```
int somme(int a, int b); // déclaration sans définition
int moyenne(int a, int b) { return somme(a, b) / 2; }
int somme(int a, int b) { return a+b; } // définition
```

- typiquement :
  - on déclare les fonctions dans un fichier `.h` partagé
  - on définit les fonctions dans un fichier `.cpp`

## Passage des paramètres :

- `void swap(int a, int b) { int t = a; a = b; b = t; }`  
`swap(x, y);` ne marche pas (idem Java)  
**passage par valeur = copie**

## Passage des paramètres :

- `void swap(int a, int b) { int t = a; a = b; b = t; }`  
`swap(x, y);` ne marche pas (idem Java)

**passage par valeur = copie**

- `void swap2(int& a, int& b) { int t = a; a = b; b = t; }`  
 appel inchangé : `swap(x, y);` OK car **passage par référence**

## Passage des paramètres :

- `void swap(int a, int b) { int t = a; a = b; b = t; }`  
`swap(x, y);` ne marche pas (idem Java)

**passage par valeur = copie**



`void swap2(int& a, int& b) { int t = a; a = b; b = t; }`  
 appel inchangé : `swap(x, y);` OK car **passage par référence**

- en C++ référence veut dire *alias* :

```
int x = 0;
```

```
int& a = x; // a est un alias de x
```

```
a = 4; // on modifie x
```

## Passage des paramètres :

- `void swap(int a, int b) { int t = a; a = b; b = t; }`  
`swap(x, y);` ne marche pas (idem Java)  
**passage par valeur = copie**

- `void swap2(int& a, int& b) { int t = a; a = b; b = t; }`  
appel inchangé : `swap(x, y);` OK car **passage par référence**

- en C++ référence veut dire *alias* :  
`int x = 0;`  
`int& a = x; // a est un alias de x`  
`a = 4; // on modifie x`

- en C++ on peut choisir **passage par valeur** ou **passage par référence**

# Tableaux

- déclaration, construction :  

```
int t[5];
```
- un tableau n'est pas un objet (comme en Java),  
ce n'est pas une référence,  
juste une adresse sur un bloc de mémoire

# Tableaux

- déclaration, construction :

```
int t[5];
```

- un tableau n'est pas un objet (comme en Java),  
ce n'est pas une référence,  
juste une adresse sur un bloc de mémoire

- **pas d'initialisation**

```
cout << t[0] << endl; affiche quelle valeur ?
```

# Tableaux

- déclaration, construction :

```
int t[5];
```

- un tableau n'est pas un objet (comme en Java), ce n'est pas une référence, juste une adresse sur un bloc de mémoire

- **pas d'initialisation**

```
cout << t[0] << endl; affiche quelle valeur ?
```

- **pas de limites**

```
cout << t[0] << ' ' << t[-1] << ' ' << t[10] << endl;
ça affiche !
```

# Tableaux

- déclaration, construction :

```
int t[5];
```

- un tableau n'est pas un objet (comme en Java), ce n'est pas une référence, juste une adresse sur un bloc de mémoire

- **pas d'initialisation**

```
cout << t[0] << endl; affiche quelle valeur ?
```

- **pas de limites**

```
cout << t[0] << ' ' << t[-1] << ' ' << t[10] << endl;
ça affiche !
```

- `t[0] = 1;` OK

# Tableaux

- déclaration, construction :

```
int t[5];
```

- un tableau n'est pas un objet (comme en Java), ce n'est pas une référence, juste une adresse sur un bloc de mémoire

- **pas d'initialisation**

```
cout << t[0] << endl; affiche quelle valeur ?
```

- **pas de limites**

```
cout << t[0] << ' ' << t[-1] << ' ' << t[10] << endl;
ça affiche !
```

- `t[0] = 1;` OK
- `t[-1] = 1;` AÏE!

# Tableaux

- déclaration, construction :

```
int t[5];
```

- un tableau n'est pas un objet (comme en Java), ce n'est pas une référence, juste une adresse sur un bloc de mémoire

- **pas d'initialisation**

```
cout << t[0] << endl; affiche quelle valeur ?
```

- **pas de limites**

```
cout << t[0] << ' ' << t[-1] << ' ' << t[10] << endl;
ça affiche !
```

- `t[0] = 1;` OK
- `t[-1] = 1;` AÏE!
- `t[10000] = 1;` OUILLE!

## Tableaux

## Tableaux, suite

- la construction/initialisation sympathique

```
int t[] = { 1, 2, 3, 4 };
```

- longueur de `t` ?

## Tableaux, suite

- la construction/initialisation sympathique

```
int t[] = { 1, 2, 3, 4 };
```

- longueur de `t` ?
- pas de `t.length`

## Tableaux, suite

- la construction/initialisation sympathique

```
int t[] = { 1, 2, 3, 4 };
```

- longueur de `t` ?
- pas de `t.length`

- `int t2[3][4];` n'est pas un tableau de tableaux
- simplement 12 cases consécutives en mémoire
- `t2[i][j];` case  $4*i+j$

## Tableaux

## Tableaux, suite

- la construction/initialisation sympathique  
`int t[] = { 1, 2, 3, 4 };`
- longueur de `t` ?
- pas de `t.length`
  
- `int t2[3][4];` n'est pas un tableau de tableaux
- simplement 12 cases consécutives en mémoire
- `t2[i][j];` case  $4*i+j$
  
- un tableau est alloué **sur la pile**  
comme une variable d'un type primitif
- donc durée de vie limitée !

## Tableaux, suite

- la construction/initialisation sympathique

```
int t[] = { 1, 2, 3, 4 };
```

- longueur de `t` ?
- pas de `t.length`

- `int t2[3][4]` ; n'est pas un tableau de tableaux

- simplement 12 cases consécutives en mémoire

- `t2[i][j]` ; case  $4*i+j$

- un tableau est alloué **sur la pile**  
comme une variable d'un type primitif

- donc durée de vie limitée !

- tableau persistant (alloué dans le tas) : il peut aussi le faire !  
si on le demande autrement

## Tableaux

# Fonctions et tableaux

- `void f(int arg[]) { .... }`

## Tableaux

# Fonctions et tableaux

- `void f(int arg[]) { .... }`  
n'a pas de sens, sauf convention spéciale pour la longueur

## Fonctions et tableaux

- `void f(int arg[]) { .... }`  
n'a pas de sens, sauf convention spéciale pour la longueur
- ```
void clear(int t[], int length) {  
    for ( int i = 0; i < length; ++i )  
        t[i] = 0;  
}
```
- tableau : **passage par adresse** (recopie de l'adresse)

## Fonctions et tableaux

- `void f(int arg[]) { .... }`  
n'a pas de sens, sauf convention spéciale pour la longueur
- ```
void clear(int t[], int length) {  
    for ( int i = 0; i < length; ++i )  
        t[i] = 0;  
}
```
- **tableau : passage par adresse** (recopie de l'adresse)
- ```
void println(const int t[], int length) {  
    for ( int i = 0; i < length; ++i )  
        cout << t[i] << ' '  
    cout << endl;  
}
```

## Fonctions et tableaux

- `void f(int arg[]) { .... }`  
n'a pas de sens, sauf convention spéciale pour la longueur
- ```
void clear(int t[], int length) {  
    for ( int i = 0; i < length; ++i )  
        t[i] = 0;  
}
```
- **tableau : passage par adresse** (recopie de l'adresse)
- ```
void println(const int t[], int length) {  
    for ( int i = 0; i < length; ++i )  
        cout << t[i] << ' '  
        cout << endl;  
}
```
- `const` protège vraiment le contenu (pas vrai avec `final` en Java)

3

## Objets

- Définition d'une classe
- Surcharge d'opérateurs
- Objets `const`
- Désallocation

## Définition d'une classe

## Définition minimale

```
#include <iostream>
using namespace std;
class Duration {
    public:                                // sinon private par défaut
        int hours, minutes, seconds;
}; // <---- point virgule !
```

## Définition d'une classe

## Définition minimale

```

#include <iostream>
using namespace std;
class Duration {
    public:                                // sinon private par défaut
        int hours, minutes, seconds;
}; // <---- point virgule !

int main()
{
    Duration d1;                          // constructeur implicite
    d1.minutes = 20;
    cout << d1.hours << ':' << d1.minutes << ':' << d1.seconds <
}

```

écrit :

## Définition d'une classe

## Définition minimale

```

#include <iostream>
using namespace std;
class Duration {
    public:                                // sinon private par défaut
        int hours, minutes, seconds;
}; // <---- point virgule !

int main()
{
    Duration d1;                          // constructeur implicite
    d1.minutes = 20;
    cout << d1.hours << ':' << d1.minutes << ':' << d1.seconds <
}

```

écrit :

n'importe quoi:20:n'importe quoi

les champs d'un objet ne sont pas initialisés

## Constructeurs

```
class Duration {
    public:
        int hours, minutes, seconds;
        Duration(int h, int m, int s) {
            this->hours = h; minutes = m; seconds = s; }
        Duration() { hours = 0; minutes = 0; seconds = 0; }
};
```

## Définition d'une classe

# Constructeurs

```

class Duration {
    public:
        int hours, minutes, seconds;
        Duration(int h, int m, int s) {
            this->hours = h; minutes = m; seconds = s; }
        Duration() { hours = 0; minutes = 0; seconds = 0; }
};

int main() {
    Duration d1;           // suffit pour construire (0,0,0)
    Duration d2(10,20,30); // constructeur (int,int,int)
    Duration d3 = d2;     // fait une copie de d2
    ...
}

```

il y a un constructeur par recopie (implicite) :

```

Duration(const Duration& d) {
    this->hours = d.hours; .... }

```

d est passé par référence, pour éviter une première recopie,

## Définition d'une classe

# Constructeurs

```

class Duration {
    public:
        int hours, minutes, seconds;
        Duration(int h, int m, int s) {
            this->hours = h; minutes = m; seconds = s; }
        Duration() { hours = 0; minutes = 0; seconds = 0; }
};

int main() {
    Duration d1;           // suffit pour construire (0,0,0)
    Duration d2(10,20,30); // constructeur (int,int,int)
    Duration d3 = d2;     // fait une copie de d2
    ...
}

```

il y a un constructeur par recopie (implicite) :

```

Duration(const Duration& d) {
    this->hours = d.hours; .... }

```

`d` est passé par référence, pour éviter une première recopie, mais `const`

## Méthodes

```
class Duration {
    int hours, minutes, seconds; // maintenant private
public:
    Duration(int h, int m, int s) { ... }
    Duration() { ... }
    int getHours() { return hours; }
    void normalize() { minutes += seconds/60; seconds %= 60;
        ... }
};
```

## Définition d'une classe

## Méthodes

```
class Duration {
    int hours, minutes, seconds; // maintenant private
public:
    Duration(int h, int m, int s) { ... }
    Duration() { ... }
    int getHours() { return hours; }
    void normalize() { minutes += seconds/60; seconds %= 60;
        ... }
};

int main() {
    ...
    d3.normalize();
    ...
    cout << d3.getHours() << ....
}
```

## Définitions externes

```
class Duration {
    int hours, minutes, seconds; // maintenant private
public:
    Duration(int h, int m, int s) { ... }
    Duration() { ... }
    int getHours() { return hours; }
    void normalize();
};
```

## Définition d'une classe

## Définitions externes

```
class Duration {
    int hours, minutes, seconds; // maintenant private
public:
    Duration(int h, int m, int s) { ... }
    Duration() { ... }
    int getHours() { return hours; }
    void normalize();
};
```

plus loin ou, par exemple, dans `math.cpp` :

```
void Duration::normalize() {
    minutes += seconds/60;
    seconds %= 60;
    ...
}
```

## Surcharge d'opérateur

```
class Duration {
    ...
    void normalize();
    Duration operator + (const Duration& d) {
        Duration dd;
        dd.hours = this->hours + d.hours;
        dd.minutes = this->minutes + d.minutes;
        dd.seconds = this->seconds + d.seconds;
        dd.normalize();
        return dd;
    }
};
```

## Surcharge d'opérateur

```
class Duration {
    ...
    void normalize();
    Duration operator + (const Duration& d) {
        Duration dd;
        dd.hours = this->hours + d.hours;
        dd.minutes = this->minutes + d.minutes;
        dd.seconds = this->seconds + d.seconds;
        dd.normalize();
        return dd;
    }
};

....
Duration d4 = d2.operator+(d3); // forme littérale
```

## Surcharge d'opérateur

```

class Duration {
    ...
    void normalize();
    Duration operator + (const Duration& d) {
        Duration dd;
        dd.hours = this->hours + d.hours;
        dd.minutes = this->minutes + d.minutes;
        dd.seconds = this->seconds + d.seconds;
        dd.normalize();
        return dd;
    }
};

....
Duration d4 = d2.operator+(d3); // forme littérale

Duration d5 = d2 + d3;         // raccourci

....

```

## Un autre très pratique

```
class Duration {  
    ...  
    // permet à cette fonction d'accéder  
    // aux champs private  
    friend ostream& operator << (ostream&, const Duration&);  
};
```

## Un autre très pratique

```

class Duration {
    ...
    // permet à cette fonction d'accéder
    // aux champs private
    friend ostream& operator << (ostream&, const Duration&);
};

// surcharge de <<
ostream& operator << (ostream& out, const Duration& d) {
    out << d.hours << ':' << d.minutes << ':' << d.seconds;
    return out; // astuce
}

....
cout << d4 << ' ' << d5 << endl;
....

```

## Objets et méthodes `const`

.....

```

Duration d2(10,20,3001);
d2.normalize();
const Duration d3 = d2;
d3.normalize();                // INTERDIT
Duration d4 = d2+d3;
Duration d5 = d3+d2;          // INTERDIT !

```

.....

OK : `normalize` modifie le contenu de l'objet

Objets et méthodes `const`

.....

```

Duration d2(10,20,3001);
d2.normalize();
const Duration d3 = d2;
d3.normalize();                // INTERDIT
Duration d4 = d2+d3;
Duration d5 = d3+d2;          // INTERDIT !

```

.....

OK : `normalize` modifie le contenu de l'objet

MAIS : `+` ne modifie pas le premier opérande !

## Objets et méthodes const

```

....
    Duration d2(10,20,3001);
    d2.normalize();
    const Duration d3 = d2;
    d3.normalize();                // INTERDIT
    Duration d4 = d2+d3;
    Duration d5 = d3+d2;          // INTERDIT !
....

```

OK : `normalize` modifie le contenu de l'objet

MAIS : `+` ne modifie pas le premier opérande !

```

class Duration {
    ...
    int getHours() const { return hours; }
    ...
    Duration operator + (const Duration& d) const { ... }
};

```

## Désallocation de variables

- Les objets affectés à une variable locale sont désalloués à la fin du bloc.
- exemple :

```
Duration f(....) {
    Duration d1;
    .....
    Duration d2;
    .....
    if ( ... )
        return d1;
    else
        return d2;
    // d1 et d2 désalloués
}
.....
Duration d = f( .... );
```

## Désallocation de variables

- Les objets affectés à une variable locale sont désalloués à la fin du bloc.
- exemple :

```
Duration f(....) {  
    Duration d1;  
    .....  
    Duration d2;  
    .....  
    if ( ... )  
        return d1;  
    else  
        return d2;  
    // d1 et d2 désalloués  
}  
.....  
Duration d = f( .... );
```

ouf! recopie dans d avant

# Optimisation

- cas particulier :

```
Duration f(....) {  
    // pas d'allocation, d1 est un alias de d  
    Duration d1;  
    .....  
    return d1;  
    // pas de désallocation  
}  
.....  
Duration d = f( .... );
```

## Désallocation d'objets temporaires

- Les objets temporaires qui apparaissent dans une expression, sont désalloués à la fin de l'évaluation de l'expression.
- exemple :

```
cout << d4+d5 << endl;  
// le résultat de l'addition est désalloué maintenant
```

## Désallocation d'objets temporaires

- Les objets temporaires qui apparaissent dans une expression, sont désalloués à la fin de l'évaluation de l'expression.
- exemple :

```
cout << d4+d5 << endl;  
// le résultat de l'addition est désalloué maintenant
```

- pour voir les destructions :

```
class Duration {  
    ...  
    // on redéfinit le destructeur  
    ~Duration() {  
        cout << "bye " << this << .... << endl;  
    }  
}
```

4

## S'il reste du temps

- Généricité
- Pointeurs et tas

# Template

```
template<typename T>
class Couple {
    public:
        T e1, e2;
        Couple(T a, T b) { e1 = a; e2 = b; }
        void swap() { T t = e1; e1 = e2; e2 = t; }
        Couple copy_swap() { return Couple(e2,e1); }
};
.....
```

## Template

```

template<typename T>
class Couple {
    public:
        T e1, e2;
        Couple(T a, T b) { e1 = a; e2 = b; }
        void swap() { T t = e1; e1 = e2; e2 = t; }
        Couple copy_swap() { return Couple(e2,e1); }
};

.....

Couple<int> i(10,20);
cout << i.e1 << ' ' << i.e2 << endl;
Couple<string> s("Hello","World");
Couple<string> t = s.copy_swap();
cout << t.e1 << ' ' << t.e2 << endl;
Couple<Duration*> d(new Duration(1,2,3),new Duration(4,5,6))
d.swap();
cout << *(d.e1) << ' ' << *(d.e2) << endl;

```

# Pointeurs

- rappel :

## **passage par référence**

```
void swap2(int& a, int& b) { int t = a; a = b; b = t; }
```

appel inchangé : `swap(x, y);`

## Pointeurs

- rappel :

### passage par référence

```
void swap2(int& a, int& b) { int t = a; a = b; b = t; }  
appel inchangé : swap(x, y);
```

- pointeur = adresse

```
int x = 0;  
int& a = x; // alias de x  
a = 2; // x est modifié  
int* p = &x; // adresse de x  
*p = 4; // x est modifié
```

## Pointeurs

- rappel :

### passage par référence

```
void swap2(int& a, int& b) { int t = a; a = b; b = t; }
appel inchangé : swap(x, y);
```

- pointeur = adresse

```
int x = 0;
int& a = x; // alias de x
a = 2; // x est modifié
int* p = &x; // adresse de x
*p = 4; // x est modifié
```

- passage par adresse

```
void swap3(int* a, int* b) { int t = *a; *a = *b; *b = t; }
appel modifié : swap(&x, &y);
```

## Pointeurs

- rappel :

### passage par référence

```
void swap2(int& a, int& b) { int t = a; a = b; b = t; }  
appel inchangé : swap(x, y);
```

- pointeur = adresse

```
int x = 0;  
int& a = x; // alias de x  
a = 2; // x est modifié  
int* p = &x; // adresse de x  
*p = 4; // x est modifié
```

- passage par adresse

```
void swap3(int* a, int* b) { int t = *a; *a = *b; *b = t; }  
appel modifié : swap(&x, &y);
```

- pour l'instant : adresses vers la pile
- ne jamais retourner une adresse vers la pile

## Tableaux et pointeurs

- allocation d'un tableau dans le tas (à la Java)

```
int* t = new int[5];
```

- `t[i] = ... etc.`

- ```
int t1[5];  
int* t2 = t1; // l'adresse de t1 est t1
```

- mais

## Tableaux et pointeurs

- allocation d'un tableau dans le tas (à la Java)

```
int* t = new int[5];
```

- `t[i] = ... etc.`

- ```
int t1[5];  
int* t2 = t1; // l'adresse de t1 est t1
```

- mais
- **pas de désallocation automatique**

## Tableaux et pointeurs

- allocation d'un tableau dans le tas (à la Java)

```
int* t = new int[5];
```

- `t[i] = ... etc.`

- ```
int t1[5];  
int* t2 = t1; // l'adresse de t1 est t1
```

- mais
- **pas de désallocation automatique**
- `delete[] t;` **1 seule fois !**

## Tableaux et pointeurs

- allocation d'un tableau dans le tas (à la Java)

```
int* t = new int[5];
```

- `t[i] = ... etc.`

- ```
int t1[5];
int* t2 = t1; // l'adresse de t1 est t1
```

- mais
- **pas de désallocation automatique**
- `delete[] t; 1 seule fois !`
- `delete[] t2; NON !`

## Tableaux et pointeurs

- allocation d'un tableau dans le tas (à la Java)

```
int* t = new int[5];
```

- `t[i] = ... etc.`

- `int t1[5];`  
`int* t2 = t1; // l'adresse de t1 est t1`

- mais
- **pas de désallocation automatique**
- `delete[] t; 1 seule fois !`
- `delete[] t2; NON !`
- **qui peut/doit désallouer ?**

## Attention

- ```
int* copy(int t[], int length) {
    int t2[length];
    for ( int i = 0; i < length; ++i )
        t2[i] = t[i];
    return t2;          // NON sur la pile
}
```

- ```
int* copy(int t[], int length) {
    int* t2 = new int[length];
    for ( int i = 0; i < length; ++i )
        t2[i] = t[i];
    return t2;          // OK dans le tas
}
```

## Attention

- ```

int* copy(int t[], int length) {
    int t2[length];
    for ( int i = 0; i < length; ++i )
        t2[i] = t[i];
    return t2;          // NON sur la pile
}

```

- ```

int* copy(int t[], int length) {
    int* t2 = new int[length];
    for ( int i = 0; i < length; ++i )
        t2[i] = t[i];
    return t2;          // OK dans le tas
}

```

- ```

piège int* i, j;

```

## Attention

- ```
int* copy(int t[], int length) {
    int t2[length];
    for ( int i = 0; i < length; ++i )
        t2[i] = t[i];
    return t2;          // NON sur la pile
}
```

- ```
int* copy(int t[], int length) {
    int* t2 = new int[length];
    for ( int i = 0; i < length; ++i )
        t2[i] = t[i];
    return t2;          // OK dans le tas
}
```

- piège `int* i, j;`

- se lit : `int *i, j;`

## Objets et pointeurs

```
Duration* p1 = new Duration(10,20,30); // dans le tas
Duration d(10,20,3001);
Duration* p2 = &d; // sur la pile = danger
Duration* p3 = new Duration(d); // copie dans le tas
```

accès aux champs :

```
p1->hours
(*p1).hours // idem p1->hours
p1->getHours()
cout << *p3 + *p2 << endl;
```

this est un pointeur

## 5 La suite

## N'oubliez pas :

- cet après-midi, TD en salle 32
- envoyer un mail à Renaud Keriven  
quel matériel, portable, système, ...
- prochaines fois : tout ici
- idées de projet ...