

# Programmation en C++

EFREI - 2015/2016

Nicolas Sicard

v3

# 2

## Planning - Évaluations

- Cours de l'UE 43 «Informatique générale»
  - Pondération pour l'UE : 2 / 7
  - Évaluations : 1 DE (60%), TP (40%)
- Planning
  - Cours : 6 séances de 1h45 = 10h30
  - TP : 4 séances de 3h30 = 14h

# Références

- Thinking in C++ (2nd ed.) - *Bruce Eckel* - Prentice Hall
- Le Langage C++ - *Bjarne Stroustrup* - Pearson Education
- Cours C++ - *Olivier Carles* - EFREI
- Object-Oriented Programming with Objective-C - *Apple* -  
[http://developer.apple.com/documentation/Cocoa/Conceptual/OOP\\_ObjC/index.html](http://developer.apple.com/documentation/Cocoa/Conceptual/OOP_ObjC/index.html)
- Beyond C++ Standard Library : An Introduction to Boost - *Björn Karlsson* - Addison Wesley

# Enjeux

## TIOBE Index de Février 2016

Feb 2016	Feb 2015	Change	Programming Language	Ratings	Change
1	2	↑	Java	21.145%	+5.80%
2	1	↓	C	15.594%	-0.89%
3	3		C++	6.907%	+0.29%
4	5	↑	C#	4.400%	-1.34%
5	8	↑	Python	4.180%	+1.30%
6	7	↑	PHP	2.770%	-0.40%
7	9	↑	Visual Basic .NET	2.454%	+0.43%
8	12	↑↑	Perl	2.251%	+0.86%
9	6	↓	JavaScript	2.201%	-1.31%
10	11	↑	Delphi/Object Pascal	2.163%	+0.59%

<http://www.tiobe.com/>

## Enjeux (2)

- Langage répandu : référence dans l'entreprise
- Pont entre le C et la programmation orientée objet (POO)
  - Concepts communs avec Java, C#, Objective-C...
- Langage portable : norme mise à jour régulièrement, compilateurs présents sur toutes les plateformes
- Langage compilé : exécution rapide

# Compétences cibles

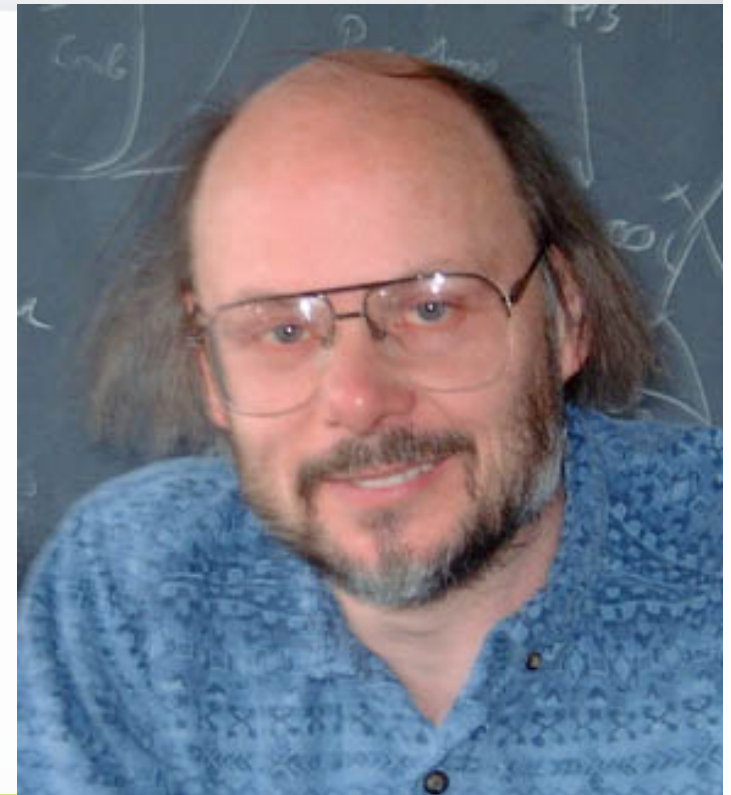
- ■ Modéliser un problème sous forme de classes
- ■ Programmer des classes en langage C++
- ■ Établir des relations (composition, agrégation, héritage) entre les classes pour mettre en œuvre un programme de bout en bout
- ■ Utiliser le polymorphisme et/ou les classes *templates* pour programmer des classes ou des fonctions génériques
- ■ Gérer les erreurs à l'aide des mécanismes d'exceptions
- ■ Manipuler des fichiers en utilisant les flux C++

- Généralités
- Programmation orientée objet
- Les classes
- Surcharges d'opérateurs
- Héritage, héritage multiple
- Polymorphisme et méthodes virtuelles
- Les exceptions
- Introduction aux “*templates*” - STL *Library*

# Généralités

## > Historique

- ❑ 1980 - Bjarne Stroustrup (Bell Labs AT&T)
- ❑ Initialement “ *C with classes* ”
- ❑ Norme ANSI/ISO C++ en juillet 1998 (ISO/IEC 14882)
- ❑ Norme **C++11** publiée le 10 octobre 2011 !
- ❑ Combiner un langage classique populaire (C) avec un modèle de programmation moderne
- ❑ Compromis entre performance, réutilisation et abstraction





# Généralités

## > Langage C / C++

- Un programme C ou C++ est compilé pour chaque plate-forme cible (≠ Java)
- C++ apporte certaines spécificités non liées à la POO
  - commentaires C : `/* bloc de texte sur plusieurs lignes */`
  - en C++ : `//commentaire` sur une seule ligne
  - transtypages de pointeurs doivent être explicites :

```
void*    gen;  
int* adi;  
  
/* en C */  
gen = adi;  
adi = gen;  
// en C++  
gen = adi;  
adi = (int*)gen;
```

# Généralités

## > Langage C / C++

- N'importe quel fichier (`.c` ou `.h`) ou code source en langage C (propre) peut être compilé par un compilateur C++
- Les fichiers en-têtes portent également le suffixe `.h` et les fichiers sources l'en-tête `.cpp` ou `.cc`
- En ligne de commande (UNIX), le compilateur est GNU `g++` qui accepte presque toutes les mêmes options que `gcc`
- Pour "linker" contre des bibliothèques compilées avec `gcc`, il faut utiliser `extern "C" {#include "entete_c.h" ...}`

# Spécificités du C++

## > Constantes (**const**)

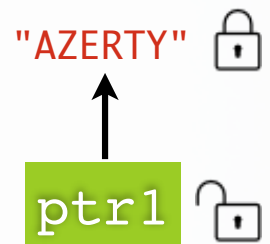
- **En C et C++** : directive préprocesseur (avant compilation) pour définir des constantes - ou *macro* - en remplacement de texte
  - ex : `#define MAX 50`
- **En C++** : on utilise plutôt le mot-clé **const** pour définir une variable constante (non modifiable) correctement typée
  - exemple : `const int MAX=50;`
  - alors : `int tab[MAX];` //autorisé en C++ (pas en C) !!
- Portées comparables à l'équivalent variable (sans **const**)

# Constantes (const)

## > Cas des pointeurs

- La donnée pointée est constante (non modifiable)

```
const char *ptr1 = "AZERTY";  
ptr1++;           // OK !  
*ptr1 = 'B';      // ERROR: assignment to const type
```



- Le pointeur est constant mais la donnée est modifiable

```
char const *ptr2 = "AZERTY";  
ptr2++;           // ERROR: increment of const type  
*ptr2 = 'B';      // OK !
```



- Le pointeur et la donnée sont constants

```
const char const *ptr3 = "AZERTY";  
ptr3++;           // ERROR: increment of const type  
*ptr3 = 'B';      // ERROR: assignment to const type
```



# Constantes (const)

## > Pointeurs et paramètre de f°

```
void f(const int *p1, int *p2)
{
    *p1 = 7;           // ERROR: l-value specifies const object
    int i = *p1;
    *p2 = 7;
    int j = *p1;
    if (i != j)
        cout << "j'ai changé *p1 grace a p2 !" << endl;
}

int main() {
    int x = 0;
    f(&x, &x); // OK !
    return 0;
}
```

`const int *p1` indique seulement que le contenu de `p1` ne peut être modifié en passant par `p1`, mais rien ne l'interdit via `p2` !

# Spécificités du C++

## > Déclaration de variables

```
{  
    ...  
    int i = 0;  
    ↓  
}
```



En C++ **on peut déclarer les variables n'importe où dans le code !**

- La portée de la variable va de la ligne de sa déclaration jusqu'à la fin du bloc courant (y compris une boucle `for`)
- Une variable `const` peut être initialisée à sa déclaration à partir du résultat d'un calcul (mais n'est plus modifiable après)
- Intérêt : déclarer une variable au plus près de son utilisation offre une meilleure lisibilité

```
#include <stdio.h>
```

```
int main() {  
    int i = 0;  
    i++;  
    int j = i;  
    j++;  
    int ma_fonction(int a, int b);  
  
    const int k = ma_fonction(i,j);  
    k = 2; // ERROR : assignment of read-only variable 'k'  
    cout << "resultat = " << k << endl;  
}
```

on peut déclarer une variable **const** au milieu d'un bloc, mais elle ne pourra plus être modifiée par la suite



**Déclaration de variables en C++**

```
int main()  
{  
    for (int i=0; i<25; i++)  
        cout << "[" << i << "]" ";  
    cout << "[" << i << "]" ;  
    // ERROR : 'for' loop initial declaration used outside C99 mode  
}
```

on peut même déclarer un compteur directement dans une boucle **for**, mais attention à ne pas l'utiliser en dehors !

15



# Spécificités du C++ > Références (synonymes)

- Permet d'attribuer un synonyme (alias)
  - à une variable : modifier le contenu d'une variable via un autre nom
  - un paramètre : remplacer le passage de paramètre par adresse
  - à une valeur de retour d'une fonction (au lieu d'un pointeur)
- Syntaxe
  - `int a;` // déclaration d'un entier
  - `int& b=a;` // `b` est synonyme de `a` (doit être initialisé à la déclaration)
  - `!! const int a;` puis `int& b=a;` n'est pas possible
  - !! ne pas confondre avec l'opérateur `&` (adresse de)





## ■ Références comme variables

```
double a = 1.0;
```

```
double& b = a;           // a et b désignent la même variable en  
mémoire !
```

```
double& c;           // ERROR : 'c' declared as reference but not  
initialized
```

```
a = 5.4;
```

```
cout << "a = " << a << " et b = " << b << endl;  
// affiche : a = 5.4 et b = 5.4
```

```
double *ptr = &b; //ptr <- adresse de b
```

```
*ptr = -2.3;
```

```
cout << "a = " << a << " et b = " << b << endl;  
// affiche : a = -2.3 et b = -2.3
```



**Utilisation des  
références en  
C++**

# 17

## ■ Références comme paramètres (remplace le passage par adresse)

//échange local seulement

```
void echangeCpy(int a, int b) { int tmp=a; a=b; b=tmp; }
```

//passage par adresse

```
void echangePtr(int *a, int *b) { int tmp=*a; *a=*b; *b=tmp; }
```

//passage par référence

```
void echangeRef(int& p, int& q) { int tmp=p; p=q; q=tmp; }
```

```
int main(){  
    int a=3, b=5;
```

```
    echangeCpy(a,b); cout<<"a = "<<a<<" b = "<<b<<endl;  
    // affiche : a=3 b=5
```

```
    echangePtr(&a,&b); cout<<"a = "<<a<<" b = "<<b<<endl;  
    // affiche : a=5 b=3
```

```
    echangeRef(a,b); cout<<"a = "<<a<<" b = "<<b<<endl;  
    // affiche : a=3 b=5
```

```
}
```



**Utilisation des  
références en  
C++**

# 18

## ■ Référence comme valeur de retour de fonction

la fonction retourne une référence (alias) sur l'élément d'indice `index` du tableau `tab` et non pas une copie de sa valeur !

```
int& valueAt(int *tab, int index) {  
    int i = 0;  
    return i;  
    return tab[index+i]; // OK!  
}
```

attention ! il ne faudrait pas retourner une référence sur une variable locale ou un paramètre passé par copie...

```
int main() {  
    int tablo[5] = {10, 20, 30, 40, 50};
```

```
    cout << "tablo[" << 2 << "]" = " << valueAt(tablo, 2);
```

\$ tablo[2] = 30

```
    valueAt(tablo, 3) = 35;
```

```
    cout << "tablo[" << 3 << "]" = " << valueAt(tablo, 3);
```

\$ tablo[3] = 35

```
    return 0;
```

```
}
```

l'appel de la fonction `valueAt` est remplacé par la référence retournée (sur `tablo[3]`) : on peut donc faire une affectation !

# 19

# Références

## > Paramètres référence `const`

- On passe souvent une référence `const` en paramètre plutôt qu'une adresse :
- passage par copie parfois problématique (données de grande taille)
- passage par adresse parfois dangereux (modification de la mémoire) et syntaxe lourde (\* et ->)

■ Exemple : `typedef struct { double x; double y; } Point;`

on peut ainsi exercer un **contrôle strict de la modification des paramètres**

```
void affiche(const Point& p) {  
    cout << "(" << p.x << "," << p.y << ")";  
    p.x = 0.0; // ERROR : assignment of data-member  
               'point::x' in read-only structure  
}
```



# Fonctions

## > Arguments à valeur par défaut

- Attribuer une valeur par défaut à un ou plusieurs paramètres qui deviennent implicites à l'appel de la fonction

- permet d'alléger l'écriture pour des valeurs souvent utilisées

```
void afficher(const Point& p, int couleur=0, const char *nom="Point") {  
    cout << nom << " = (" << p.x << "," << p.y << ")[" << couleur << "]" << endl;  
}
```

attention, les paramètres  
par défaut doivent être  
en fin de liste !

```
int main() {  
    Point p1 = {0.0,0.0};  
    afficher(p1);  
    afficher(p1, 127);  
    affiche(p, "Zero"); // ERROR: invalid conversion from 'const char*' to 'int'  
    affiche(p1, 255, "Origine");  
  
    return 0;  
}
```



```
$ Point = (0.000000,0.000000)[0]  
$ Point = (0.000000,0.000000)[127]  
$ Origine = (0.000000,0.000000)[255]
```

# Spécificités du C++

## > Entrées / Sorties

- **En C** : on utilise des fonctions d'E/S standard (`scanf`, `printf...`)
- **En C++** : on utilise des opérateurs d'E/S
  - << pour envoyer des valeurs dans un **flot de sortie**
  - >> pour extraire des valeurs d'un **flot d'entrée**
- Un flot peut être défini à partir d'un fichier, d'une entrée/sortie standard, d'un buffer de caractères *etc.*
  - **cin** / **cout** : entrée / sortie standard (équ. à `stdin` / `stdout`)
  - **cerr** / **clog** : sorties standard d'erreur
  - les principaux types standards sont reconnus (`char`, `float`, `int` *etc.*)



```
#include <iostream>
using namespace std;
```

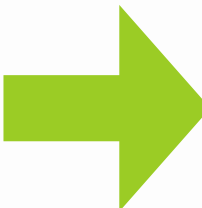
**utilisation de l'espace de nom std :**  
permet d'accéder directement aux flots  
std::cout et std::cin

```
int main() {
    int i=123;
    float f=1234.567
    char str[]="Bonjour\n", rep;
```

**les opérateurs << sont associatifs à droite :** on peut concaténer les E/S de gauche à droite

```
    cout << "i=" << i << " f=" << f << " str=" << str;
    cout << "i = ? ";
    cin >> i;
    cout << "f = ? ";
    cin >> f;
    cout << "rep = ? " ;
    cin >> rep;
    cout << "str = ? ";
    cin >> str;
    cout << "i=" << i << " f=" << f << " str=" << str << endl;
    cout << "adresse de str = " << (void*)str << endl;

    return 0;
}
```



```
> i=123 f=1234.57 str=Bonjour
> i = ? 54
> f = ? 2.8
> rep = ? z
> str = ? C++ is easy
> i=54 f=2.8 str=C++
> adresse de str = 0xbffffa9b
```



**Entrées/Sorties  
en C++**

23

**endl est un flot particulier qui insère un retour chariot** et vide le buffer (comme fflush)



# Fonctions

## > Surcharges de fonctions

- Une fonction peut être surchargée : **plusieurs fonctions différentes peuvent porter le même nom**
- Le compilateur choisit une fonction grâce à la liste (typée) de ses arguments au moment de son appel
- On peut aussi surcharger des opérateurs (\*, +, etc...)
- La signature d'une fonction est composé de son nom et de la liste de ses arguments. **Attention, le type de la valeur de retour n'est pas un élément discriminant !**



```
int somme(int n1, int n2) { return n1+n2; }
double somme(int n1, int n2) { return (double)(n1+n2); }
// ERROR : new declaration 'double somme(int, int)'
// ERROR : ambiguates old declaration 'int somme(int, int)'
double somme(double d1, double d2) { return d1+d2; } // OK
int somme(int n1, int n2, int n3) { return n1+n2+n3; } // OK
```

```
typedef struct { double x; double y; } Point;
```

```
Point somme(const Point& p1, const Point& p2) {
    Point p;
    p.x = p1.x + p2.x; p.y = p1.y + p2.y;
    return p;
} // OK
```

```
int main() {
    Point p = {3.0,1.0}, q = {0.2,2.7};
    affiche(p, 0, "p");
```

```
cout << "3+4 = " << somme(3,4) << endl;
cout << "2.6+7.2 = " << somme(2.6,7.2) << endl;
```

```
Point r = somme(p,q);
```

```
affiche(r, 0, "p+q");
return 0;
```

```
}
```



```
$ p = (3.000000,1.000000)[0]
$ 3+4 = 7
$ 2.6+7.2 = 9.800000
$ p+q = (3.200000,3.700000)[0]
```



## Surcharge de fonctions en C++

# 25

```
int somme(int n1, int n2) { return n1+n2; }
double somme(int n1, int n2) { return (double)(n1+n2); }
// ERROR : new declaration 'double somme(int, int)'
// ERROR : ambiguates old declaration 'int somme(int, int)'
double somme(double d1, double d2) { return d1+d2; } // OK
int somme(int n1, int n2, int n3) { return n1+n2+n3; } // OK
```

```
typedef struct { double x; double y; } Point;
```



```
Point operator+(const Point& p1, const Point& p2) { <-----
    Point p;
    p.x = p1.x + p2.x; p.y = p1.y + p2.y;
    return p;
} // OK
```

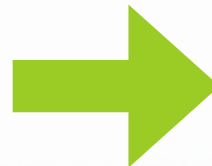
on remplace le nom de la fonction **somme** par le mot clé **operator** suivi du symbole + de l'opérateur d'addition

```
int main() {
    Point p = {3.0,1.0}, q = {0.2,2.7};
    affiche(p, 0, "p");
```

```
cout << "3+4 = " << somme(3,4) << endl;
cout << "2.6+7.2 = " << somme(2.6,7.2) << endl;
```



```
Point r = p+q;
affiche(r, 0, "p+q");
return 0;
}
```



```
$ p = (3.000000,1.000000)[0]
$ 3+4 = 7
$ 2.6+7.2 = 9.800000
$ p+q = (3.200000,3.700000)[0]
```

on somme directement des structures grâce à l'opérateur +



**Surcharge d'opérateurs en C++ (voir aussi chapitre dédié)**

**26**