



Programmation en C++

EFREI - L'3 2009/2010

Nicolas Sicard

v1.5

Références

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

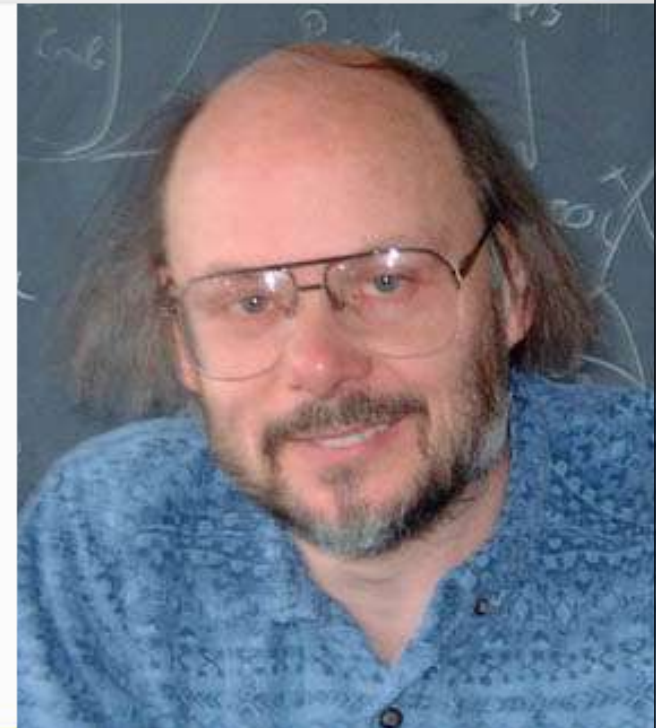
3

- 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)
- 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 via le ptr)

```
const char *ptr1 = "AZERTY";  
ptr1++;           // OK !  
*ptr1 = 'A';      // 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 = 'A';      // OK !
```

- Le pointeur et la donnée sont constants

```
const char const *ptr3 = "AZERTY";  
ptr3++;           // ERROR: increment of const type  
*ptr3 = 'A';      // 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;
}

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

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



Spécificités du C++

> Déclaration de variables

- 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>
```

```
void 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;
```

```
}
```

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

```
void main() {
```

```
    for (int i=0; i<25; i++)
```

```
        cout << "[" << i << "]" ";
```

```
    cout << "[" << i << "]" ";
```

```
    // ERROR : 'for' loop initial declaration used outside C99 mode
```

```
}
```

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++**

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

12

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 le même espace 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
```

■ Références comme paramètres (remplace le passage par adresse)

```
void echangeCpy(int a, int b) { int tmp=a; a=b; b=tmp; } //echange local seulement

void echangeAdd(int *a, int *b) { int tmp=*a; *a=*b; *b=tmp; } //passage par adresse

void echangeRef(int& a, int& b) { int tmp=a; a=b; b=tmp; } //passage par référence

void main() {
    int a=3, b=5;
    echangeCpy(a,b); cout<<"a = "<<a<<" b = "<<b<<endl; // affiche : a=3 b=5
    echangeAdd(&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++**

14

■ 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...

```
void 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

```
}
```

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

15

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 :

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

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

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 !

```
void 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");  
}
```



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

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;
    affiche(somme(p,q), 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++**

19

```
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;
affiche(p+q, 0, "p+q"); // <----- on somme des couples avec l'opérateur !
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 d'
opérateurs en
C++ (voir aussi
chapitre dédié)**

20

Fonctions

> Fonctions `inline`

- Le compilateur remplace l'appel d'une fonction `inline` dans le programme par son code machine
 - exécution plus rapide (pas de gestion de contexte *etc.*)
 - privilégier les fonctions de taille courte appelées très souvent
 - s'utilise exactement comme une fonction classique
 - vérification des types des arguments ($\neq$ macro) !
- Exemple :

```
inline long carre(long x) { return x*x; }
```

Fonctions

> #define vs. inline

- Effets indésirable d'une macro

```
#define carre(x) x*x
```

```
long a=3, a2;  
a2=carre(a++); // remplacé par a++ * a++ (a incrémenté 2 fois !)  
cout << "a2=" << a2 << " a=" << a << endl; // affiche a2=9 a=5
```

- Pas de problème avec une fonction inline

```
inline long carre(long x) { return x*x; }
```

```
long a=3, a2;  
a2=carre(a++); // comme une fonction classique  
cout << "a2=" << a2 << " a=" << a << endl; // affiche a2=9 a=4
```

Fonctions

> Comparatif

	avantages	inconvénients
macro	vitesse d'exécution	risque d'effet de bord pas de compilat° séparée perte d'espace mémoire
fonction	économie mémoire compilation séparée	perte de temps d'exécut° des fonctions courtes
fonction inline	vitesse d'exécution compilation séparée	perte d'espace mémoire

Mémoire dynamique

> Allocation / Libération

- **En C** : on utilise la fonction **malloc(size_t)** qui alloue une zone mémoire non typée et non initialisée
- **En C++** : l'opérateur **new** réserve l'espace mémoire demandé et retourne l'adresse (typée) de début de zone. Il retourne **NULL** en cas d'erreur
 - l'allocation peut s'accompagner d'une initialisation
 - l'opérateur **new type[]** permet d'allouer des tableaux typés
- L'opérateur **delete** libère une zone mémoire à partir de son adresse (pointeur)

```
// allocation puis initialisation d'un réel double précision
```

```
double *ptr1 = new double;
```

```
*ptr1 = 3.0;
```

```
// allocation/initialisation et libération d'un réel double préc.
```

```
double *ptr2 = new double(2.81);
```

```
delete ptr2;
```

```
// allocation d'un tableau de 10 réels
```

```
double *tab1 = new double[10];
```

```
// allocation d'un tableau avec initialisation impossible
```

```
double *tab1 = new double[10](2.0); //ERROR : ISO C++ forbids  
initialization in array new
```

```
// allocation puis libération d'un tableau de n réels
```

```
int n = 30;
```

```
double *tab2 = new double[n];
```

```
delete [] tab2;
```

```
//allocation d'un point
```

```
Point *p1 = new Point;
```

```
//allocation d'un tableau de n points
```

```
Point *ptab = new Point[n];
```



**Allocation
mémoire en
C++**

25

Mémoire dynamique

> Opérateur `delete`

- L'application de l'opérateur `delete` à un pointeur de valeur `NULL` est possible et n'entraîne pas d'arrêt prématuré du programme — l'opération est ignorée.
- Attention, le comportement du programme est indéterminé si :
 - on applique `delete` à un pointeur correspondant à une zone déjà libérée
 - on fournit à `delete` une adresse impropre ou qui n'a pas été obtenue par l'opérateur `new`
 - on applique `delete` à une adresse obtenue par `malloc` !!



Mémoire dynamique

> Gestion de l'échec d'allocation

- On peut écrire et “enregistrer” une fonction d’interception qui sera automatiquement appelée en cas d’échec de l’allocation

```
#include <stdio.h>
#include <new>
using std::set_new_handler;

void MemErrorHandler() {
    printf("Memoire disponible insuffisante !\n");
    exit(1);
}

int main() {
    set_new_handler(MemErrorHandler);
    while (1) {
        double *giant = new double[1000000000];
        printf("%d floats allocated\n", 1000000000);
    }
    return 0;
}
```

```
localhost:~ nicolas$ ./a.out
allocation de 50000000 reels
allocation de 50000000 reels
allocation de 50000000 reels
allocation de 50000000 reels
allocation de 50000000 reels
a.out(9777) malloc: *** mmap(size=4
*** error: can't allocate region
*** set a breakpoint in malloc_erro
Memoire disponible insuffisante !
```

Mémoire dynamique

> Gestion de l'échec d'allocation

- Le handler est appelé tant que l'allocation n'a pas réussi, il permet de libérer de la mémoire inutile pour pouvoir continuer

```
#include <stdio.h>
#include <new>
using std::set_new_handler;

void MemErrorHandler() {
    cout << "Memoire disponible insuffisante !" << endl;
    // libération de zones mémoire inutiles sinon exit
}

int main() {
    set_new_handler(MemErrorHandler);
    while (1) {
        double *giant = new double[100000000];
        cout << 100000000 << " floats allocated" << endl;
    }
    return 0;
}
```

Espaces de nom (*namespaces*)

> Description

- **Module logique** cohérent de variables, de fonctions ou de classes C++ sous un même nom
- Permet de **limiter les conflits de nommage** de variables, fonctions ou classes dans des projets de grande ampleur avec utilisation de modules indépendants

```
/* fichier objects.h */
```

```
namespace Moteur3D {  
    // contenu de l'espace de nom  
  
    class Object {  
    public:  
        double x,y,z; //position  
  
        Object(double a, double b, double c);  
    };  
  
    void print_objects();  
    // etc...  
}
```

Espaces de nom (*namespaces*)

> Description

```
/* fichier objects.cc */  
  
#include "test3.h"  
  
namespace Moteur3D {  
    // contenu de l'espace de nom  
    Object*      objects[10];  
    unsigned      nb_objects = 0;  
  
    Object::Object(double a, double b, double c) : x(a), y(b), z(c) {  
        printf("Création d'un objet\n");  
    }  
  
    void print_objects() {  
        for (int i=0; i < nb_objects; i++) {  
            printf("[%lf %lf %lf]\n", objects[i]->x, objects[i]->y, objects[i]->z)  
        }  
    }  
    // etc...  
}
```

Espaces de nom (*namespaces*)

> Description

```
/* fichier main.cc */
```

```
#include "test3.h"
```

```
int main() {  
    Moteur3D::objects[0] = new Moteur3D::Object(1.0,1.0,1.0);  
  
    Moteur3D::nb_objects = 1;  
    Moteur3D::print_objects();  
    delete Moteur3D::objects[0];  
}
```

ou bien :

```
/* fichier main.cc */
```

```
#include "test3.h"  
using namespace Moteur3D; ★
```

```
int main() {  
    objects[0] = new Object(1.0,1.0,1.0);  
  
    nb_objects = 1;  
    print_objects();  
    delete objects[0];  
}
```

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

```
void 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;
```

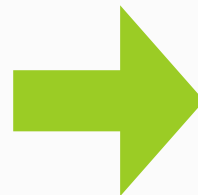
```
    cout << "str = ? ";
```

```
    cin >> str;
```

```
    cout << "i=" << i << " f=" << f << " str=" << str << endl;
```

```
    cout << "adresse de str = " << (void*)str << endl;
```

```
}
```



```
> 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++**

33

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