

Cours C++

Quelques spécificités du C++ (hors POO)

Spécifités notables du C++ par rapport au C

- ▶ Bibliothèque standard
- ▶ Déclaration des variables (emplacement libre)
- ▶ Arguments par défaut de fonction
- ▶ Surdéfinition de fonction
- ▶ Type *bool*
- ▶ Espace de noms
- ▶ Références
- ▶ Allocation dynamique de mémoire : opérateur *new* et *delete*
- ▶ Exceptions

Bibliothèque standard

- Les bibliothèques du C ont été adaptées aux besoins du C++ :

```
// Inclusion en C  
#include <math.h>  
#include <time.h>
```

```
// Inclusion en C++  
#include <cmath>  
#include <ctime>
```

De nombreuses autres spécifiques au C++ existent (*cf.* diapos 3), l'ensemble formant la bibliothèque standard du C++ (*standard library*, ou *std lib*).

Déclaration de variables, gestion de la mémoire

- Possibilité de déclarer les variables locales quel que soit l'endroit

```
for (int i = 0 ; i < 10 ; i++) {  
    double x = 1./i;  
}
```



La variable `x` n'existe plus en dehors de la boucle

- ☞ À garder en tête ! Notion de cycle de vie (*scope*). Très différent de Python ou Java (*garbage collector*).

Arguments par défaut de fonctions

C++ autorise l'utilisation de valeurs par défaut dans les prototypes de fonction

```
// Prototype de fonction (déclaration)
void initialize(const double abs, const double ord = 3.4);

int main() {
    initialize(1.0); // Utilisation
}

// Définition
void initialize(const double abs, const double ord)
{ ... }
```

- ▶ En l'absence de second argument, le compilateur assigne la valeur 3.4 à la variable ord
- ▶ Les arguments concernés doivent **obligatoirement** être les **derniers** dans la liste.

Surdéfinition (surcharge) de fonctions

En C++ deux fonctions peuvent porter le même nom dès lors qu'elles n'ont pas les mêmes types d'argument

```
void dummy(int i, double a, char c);  
void dummy(int i, int a, char c);  
void dummy(int i);
```

Le compilateur se charge d'appeler la "bonne" fonction par rapport au contexte *i.e.* au vu de la liste d'arguments donnée lors de l'appel.

```
int main()  
{  
    dummy(1, 1.0, 'a');  
    dummy(1, 1, 'a');  
    dummy(1);  
}
```

Surdéfinition (surcharge) de fonctions

En C++ deux fonctions peuvent porter le même nom dès lors qu'elles n'ont pas les mêmes types d'argument

```
void dummy(int i, double a, char c);  
void dummy(int i, int a, char c);  
void dummy(int i);
```

Le compilateur se charge d'appeler la "bonne" fonction par rapport au contexte *i.e.* au vu de la liste d'arguments donnée lors de l'appel.

```
int main()  
{  
    dummy(1, 1.0, 'a');  
    dummy(1, 1, 'a');  
    dummy(1);  
}
```

Le type bool

Ce type est naturellement formé de deux valeurs notées `true` et `false` (remplaçant 1 et 0).

```
bool is_wrong = true;
```

```
if (is_wrong) {
```

```
    ...
```

```
}
```

```
bool is_nice = false;
```

```
bool is_ok = is_nice && !is_wrong;
```

OU

```
bool is_ok = is_nice and not is_wrong;
```


Les espaces de noms

Les espaces de noms sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur.

```
namespace util {  
    double angle (double re, double im) {  
        /* Calcul l'angle d'un nombre complexe */  
    }  
}  
  
namespace ondes {  
    double angle (double phase1, double phase2) {  
        /* Calcule l'angle entre deux phases, ramené dans  $[-\pi, +\pi]$  */  
    }  
}
```

Leur but est principalement d'éviter les conflits de noms entre plusieurs parties d'un projet :

```
util::angle(1, 0.4);  
ondes::angle(pi/5, 9*pi/5);
```

Les espaces de noms

Toute la bibliothèque standard est dans l'espace `std`.

```
#include <iostream>
/* using namespace std; */
...
std::cout << "Standard output" << std::endl;
...
```

Tout en permettant une syntaxe légère si désiré :

```
#include <iostream>
using namespace std;
...
cout << "Standard output" << endl;
...
```

Les espaces de noms

```
namespace utl
{
    void dump() { cout << "utl::dump" << endl; }
}

namespace io
{
    void dump() { cout << "io::dump" << endl; }
}

// Utilisation de l'espace de nom io
using namespace io;

int main()
{
    // Par défaut, io::dump()
    dump();

    // Précision de l'espace de nom
    utl::dump();
    io::dump();
}
```

Les espaces de noms

```
namespace utl
{
    void dump() { cout << "utl::dump" << endl; }
}

namespace io
{
    void dump() { cout << "io::dump" << endl; }
}

// Utilisation de l'espace de nom io
using namespace io;

int main()
{
    // Par défaut, io::dump()
    dump();

    // Précision de l'espace de nom
    utl::dump();
    io::dump();
}
```

Alias d'espace de noms

```
#include <iostream>

namespace foo {
    namespace bar {
        namespace baz {
            int qux = 42;
        }
    }
}

namespace fbz = foo::bar::baz;

int main()
{
    std::cout << fbz::qux << '\n';
}
```

Pointeurs, références et allocation dynamique

Rappels sur les adresses et pointeurs

- ▶ Tout objet manipulé par l'ordinateur est stocké en mémoire. Selon la nature de l'objet, l'espace en mémoire alloué varie : par exemple, entier = 32 ou 64 bits
- ▶ L'adresse est l'endroit où se trouve la variable en mémoire. Elle s'obtient via la syntaxe suivante : `&NomDeLaVariable`
- ▶ L'adresse n'étant ni plus ni moins qu'une valeur, on peut donc stocker cette valeur dans une variable : un pointeur est ainsi un conteneur d'adresse
- ▶ Déclaration d'un pointeur :

```
int i = 10; // variable
int *pt_i = &i; // déclaration du pointeur et assign. à l'adresse de i
int j = *pt_i; // dépointage et copie dans j
```

Notion de référence en C++

Le C++ introduit la notion de référence afin de faciliter la manipulation des variables

// Pointeur

```
int i = 10;
```

```
int * pt_i;
```

```
pt_i = &i;
```

```
(*pt_i)++;
```

// Référence

```
int i = 10;
```

```
int & ref_i = i; // ref_i est une référence à i
```

```
ref_i++;
```

- ▶ La déclaration d'une référence ne crée pas de nouvel objet, c'est simplement un alias vers une vraie variable. Il n'y a pas besoin de dépointer : utiliser une référence, c'est directement utiliser la variable référencée.
- ▶ Toute référence doit se référer à un identificateur. Il est donc nécessaire d'initialiser une référence : `int & ref_i;` ne compilera pas. Une référence pointe toujours vers le même objet, on ne peut pas changer sa destination.
- ▶ Moins flexible, mais plus pratique et sûr. Suffisant dans la majorité des cas.

Références en tant qu'arguments de fonctions

Transmission par adresse

```
void echange (int * a, int * b) {  
    int c = *a;  
    *a = *b;  
    *b = c;  
}  
...  
int x = 10;  
int y = 20;  
echange(&x, &y);
```

Transmission par référence

```
void echange (int & a, int & b) {  
    int c = a;  
    a = b;  
    b = c;  
}  
...  
int x = 10;  
int y = 20;  
echange(x, y);
```

Références en tant qu'arguments de fonctions

Transmission par adresse

```
void echange (int * a, int * b) {  
    int c = *a;  
    *a = *b;  
    *b = c;  
}  
...  
int x = 10;  
int y = 20;  
echange(&x, &y);
```

Transmission par référence

```
void echange (int & a, int & b) {  
    int c = a;  
    a = b;  
    b = c;  
}  
...  
int x = 10;  
int y = 20;  
echange(x, y);
```

Allocation dynamique de mémoire

L'allocation dynamique de mémoire est nécessaire dès lors que la taille d'un objet, sa nature ou même son existence, n'est connue que lors de l'exécution du programme. Dans le cas d'un tableau de taille n , la déclaration

```
unsigned int n = 0;  
std::cin >> n;  
double tableau[n];
```

n'est pas correcte du fait que le compilateur ne connaît pas, au préalable, l'espace mémoire nécessaire à l'allocation (statique).

Utilisation des opérateurs new et delete

- ▶ Pour rappel, en langage C, la gestion dynamique de mémoire fait appel aux fonctions malloc et free (stdlib.h)
- ▶ C++ introduit quatre nouveaux opérateurs :
 - ▶ **new** alloue la quantité de mémoire nécessaire au type, et renvoie un pointeur sur la zone mémoire allouée.

```
double * ptr = new double;  
*ptr = 3.7;
```

- ▶ **delete** libère l'espace mémoire.

```
delete ptr;  
// bien sûr, ne pas utiliser 'ptr' après
```

- ▶ **new[]** alloue la mémoire nécessaire pour un tableau de taille *n* et renvoie un pointeur sur le début du tableau :

```
unsigned int n = 0;  
std::cin >> n;  
double * tableau = new double[n];
```

- ▶ **delete[]** libère l'espace mémoire d'un pointeur-tableau :

```
delete[] tableau;
```

Utilisation des opérateurs new et delete

- ▶ Pour rappel, en langage C, la gestion dynamique de mémoire fait appel aux fonctions malloc et free (stdlib.h)
- ▶ C++ introduit quatre nouveaux opérateurs :
 - ▶ **new** alloue la quantité de mémoire nécessaire au type, et renvoie un pointeur sur la zone mémoire allouée.

```
double * ptr = new double;  
*ptr = 3.7;
```

- ▶ **delete** libère l'espace mémoire.

```
delete ptr;  
// bien sûr, ne pas utiliser 'ptr' après
```

- ▶ **new[]** alloue la mémoire nécessaire pour un tableau de taille *n* et renvoie un pointeur sur le début du tableau :

```
unsigned int n = 0;  
std::cin >> n;  
double * tableau = new double[n];
```

- ▶ **delete[]** libère l'espace mémoire d'un pointeur-tableau :
delete[] tableau;

Utilisation des opérateurs new et delete

- ▶ C++ introduit quatre nouveaux opérateurs :
 - ▶ **new** alloue une la quantité de mémoire nécessaie au type
 - ▶ **delete** libère l'espace mémoire... **mais pas que !**
 - ▶ **new[]** alloue la mémoire nécessaire pour un tableau de taille n
 - ▶ **delete[]** libère l'espace mémoire d'un pointeur-tableau... **mais pas que !**
- ▶ **delete** et **delete[]** appellent aussi les *destructeurs* associés au type → cf. cours sur les constructeurs/destructeurs. Il faut utiliser **new** et **delete** en C++ .
- ▶ Ne surtout pas mélanger avec malloc et free !

Portée et durée de vie des variables

Durée de vie limitée au bloc (ici boucle for)

```
for (int i = 0; i < 10; i++) {  
    int k = 0;  
    // À la fin du bloc,  
    // destruction de k  
}
```

Durée de vie indépendante du bloc

```
for (int i = 0; i < 10; i++) {  
    int * k = new int(0);  
    // À la fin du bloc,  
    // k existe en mémoire  
}
```

Fuite de mémoire garantie

Portée et durée de vie des variables

Durée de vie limitée au bloc (ici boucle for)

```
for (int i = 0; i < 10; i++) {  
    int k = 0;  
    // À la fin du bloc,  
    // destruction de k  
}
```

Durée de vie indépendante du bloc

```
for (int i = 0; i < 10; i++) {  
    int * k = new int(0);  
    // À la fin du bloc,  
    // k existe en mémoire  
}
```

Fuite de mémoire garantie

Portée et durée de vie des variables

Durée de vie limitée au bloc (ici boucle for)

```
for (int i = 0; i < 10; i++) {  
    int k = 0;  
    // À la fin du bloc,  
    // destruction de k  
}
```

Durée de vie indépendante du bloc

```
for (int i = 0; i < 10; i++) {  
    int * k = new int(0);  
    ...  
    delete k;  
}
```

Portée et durée de vie des variables

Allocation sur la pile, "stack" (automatique)

```
int * pointeur_dix()
{
    int a = 10;
    return &a;
}

int main()
{
    int * pb = pointeur_dix();
    cout << *pb << endl;

    return 0;
}
```

Portée et durée de vie des variables

Allocation sur la pile, "stack" (automatique)

```
int * pointeur_dix()
{
    int a = 10;
    return &a;
}
```

Le pointeur retourné contient
une adresse obsolète

```
int main()
{
    int * pb = pointeur_dix();
    cout << *pb << endl;

    return 0;
}
```

Allocation sur le tas, "heap"

```
int * pointeur_dix()
{
    int * pa = new int(10);
    return pa;
}
```

```
int main()
{
    int * pb = pointeur_dix();
    cout << *pb << endl;
    delete pb;
    return 0;
}
```

Portée et durée de vie des variables

Allocation sur la pile, "stack" (automatique)

```
int * pointeur_dix()
{
    int a = 10;
    return &a;
}
```

Le pointeur retourné contient
une adresse obsolète

```
int main()
{
    int * pb = pointeur_dix();
    cout << *pb << endl;

    return 0;
}
```

Allocation sur le tas, "heap"

```
int * pointeur_dix()
{
    int * pa = new int(10);
    return pa;
}
```

```
int main()
{
    int * pb = pointeur_dix();
    cout << *pb << endl;
    delete pb;
    return 0;
}
```

Allocation dynamique de mémoire

Ceci dit, en C++ moderne, la manipulation de pointeurs nus et l'allocation/libération manuelle de mémoire n'est **pas une bonne pratique**.

On préférera l'utilisation d'objets encapsulant et effectant automatiquement ces opérations → bibliothèque standard (`std::vector`, `std::string...`).

Et à défaut, on créera nos propres objets pour encapsuler ces opérations.

Principales notions en un exemple

- ▶ Le C++ introduit la notion de référence afin de faciliter la manipulation des variables notamment au sein des fonctions
→ transmission des arguments par référence
- ▶ En C++ plusieurs fonctions peuvent porter le même nom dès lors qu'elles n'ont pas le même contexte d'appel
→ surdéfinition ou surcharge de fonction
- ▶ Les macros sont à éviter. Les seules véritables directives de préprocesseur à utiliser sont `#include`, `#ifndef` / `#define`
→ compilation séparée

Principales notions en un exemple

- ▶ Le C++ introduit la notion de référence afin de faciliter la manipulation des variables notamment au sein des fonctions
→ transmission des arguments par référence
- ▶ En C++ plusieurs fonctions peuvent porter le même nom dès lors qu'elles n'ont pas le même contexte d'appel
→ surdéfinition ou surcharge de fonction
- ▶ Les macros sont à éviter. Les seules véritables directives de préprocesseur à utiliser sont `#include`, `#ifndef` / `#define`
→ compilation séparée

Principales notions en un exemple

- ▶ Le C++ introduit la notion de référence afin de faciliter la manipulation des variables notamment au sein des fonctions
→ transmission des arguments par référence
- ▶ En C++ plusieurs fonctions peuvent porter le même nom dès lors qu'elles n'ont pas le même contexte d'appel
→ surdéfinition ou surcharge de fonction
- ▶ Les macros sont à éviter. Les seules véritables directives de préprocesseur à utiliser sont `#include`, `#ifndef` / `#define`
→ compilation séparée

dummy.h

```
#ifndef __dummy_h__
#define __dummy_h__ 1
void dummy();
void dummy(int i);
void dummy(int & i, double & a, char & c);
#endif
```

dummy.cpp

```
#include<iostream>
#include "dummy.h"
using namespace std;
void dummy() {cout << "fct type void, pas d'argument" << endl;}
void dummy(int i) {
    cout << "fct type void, 1 argument type entier" << endl;}
void dummy(int & i, double & a, char & c) {
    cout << "fct type void, 3 arguments : ref sur entier, réel et caractère"
    << endl;}
```

test_dummy.cpp

```
#include<iostream>
using namespace std;
#include "dummy.h"
int main()
{
    cout << "Programme test des fonctions dummy" << endl;
    dummy();
    int i = 10; double d = 2.0; char c = 'a';
    dummy(i, d, c);
}
```

dummy.h

```
#ifndef __dummy_h__
#define __dummy_h__ 1
void dummy();
void dummy(int i);
void dummy(int & i, double & a, char & c);
#endif
```

Transmission par référence

Déclaration de la fonction

dummy.cpp

```
...
#include "dummy.h"
void dummy() {...}
void dummy(int i) {...}
void dummy(int & i, double & a, char & c) {...}
```

Définition de la fonction

test_dummy.cpp

```
#include "dummy.h"
int main()
{
    dummy();
    int i = 10; double d = 2.0; char c = 'a';
    dummy(i, d, c);
}
```

Appel de la fonction

dummy.h

```
#ifndef __dummy_h__
#define __dummy_h__ 1

void dummy();
void dummy(int i);
void dummy(int & i, double & a, char & c);
#endif
```

Surdéfinition de fonction

Déclaration des fonctions

dummy.cpp

```
...
#include "dummy.h"
void dummy() {...}
void dummy(int i) {...}
void dummy(int & i, double & a, char & c) {...}
```

Définition des fonctions

test_dummy.cpp

```
#include "dummy.h"
int main()
{
    dummy();
    int i = 10; double d = 2.0; char c = 'a';
    dummy(i, d, c);
}
```

Appel des fonctions

dummy.h

```
#ifndef __dummy_h__
#define __dummy_h__ 1
void dummy();
void dummy(int i);
void dummy(int & i, double & a, char & c);
#endif
```

dummy.cpp

```
...
#include "dummy.h"
void dummy() {...}
void dummy(int i) {...}
void dummy(int & i, double & a, char & c) {...}
```

test_dummy.cpp

```
#include "dummy.h"
int main()
{
    dummy();
    int i = 10; double d = 2.0; char c = 'a';
    dummy(i, d, c);
}
```

Directives de préprocesseur

Compilation

```
$ g++ dummy.cpp test_dummy.cpp -o test_dummy.exe
```

👉 Les fichiers d'en-tête ne sont jamais compilés !

Objets constants

Objets constants

Le mot-clé **const** interdit la modification de l'objet auquel il s'applique.

```
const int i = 42;  
i = 3; // erreur de compilation
```

Deux utilités :

- ▶ Protège l'objet d'une modification (contrainte imposée à soi-même et aux autres utilisateurs du code)
- ▶ Indique au compilateur qu'il peut effectuer des optimisations supplémentaires

Objets constants

Le mot-clé **const** interdit la modification de l'objet auquel il s'applique.

```
const int i = 42;  
i = 3; // erreur de compilation
```

Deux utilités :

- ▶ Protège l'objet d'une modification (contrainte imposée à soi-même et aux autres utilisateurs du code)
- ▶ Indique au compilateur qu'il peut effectuer des optimisations supplémentaires

Pointeur constant ou pointeur sur constante ?

Petite subtilité :

- ▶ **char** * **const** str; est un *pointeur constant* sur un objet de type **char** mutable
⇒ str[0]='a' est légal, mais str=NULL est illégal.
- ▶ **const char** * str; est un pointeur (mutable) sur un objet de type **const char**, ie. une constante
⇒ str[0]='a' est illégal, mais str=NULL est légal.

La question ne se pose pas pour les références, qui sont nécessairement constantes. Seule la forme **const** type& a une utilité, pour empêcher l'objet référencé d'être modifié.

Pointeur constant ou pointeur sur constante ?

Petite subtilité :

- ▶ `char * const str;` est un *pointeur constant* sur un objet de type `char` mutable
⇒ `str[0]='a'` est légal, mais `str=NULL` est illégal.
- ▶ `const char * str;` est un pointeur (mutable) sur un objet de type `const char`, ie. une constante
⇒ `str[0]='a'` est illégal, mais `str=NULL` est légal.

La question ne se pose pas pour les références, qui sont nécessairement constantes. Seule la forme `const type&` a une utilité, pour empêcher l'objet référencé d'être modifié.

Rappel : passage par valeur, par référence

Passage par valeur

```
void f (int i) {  
    i = 10;  
    cout << "f : " << i << endl;  
}  
  
void main () {  
    int e = 0;  
    f(e);  
    cout << "main : " << e << endl;  
}
```

```
f : 10  
main : 0
```

Passage par référence

```
void g (int& i) {  
    i = 10;  
    cout << "g : " << i << endl;  
}  
  
void main () {  
    int e = 0;  
    g(e);  
    cout << "main : " << e << endl;  
}
```

```
g : 10  
main : 10
```

Rappel : passage par valeur, par référence

Passage par valeur

```
void f (int i) {  
    i = 10;  
    cout << "f : " << i << endl;  
}  
  
void main () {  
    int e = 0;  
    f(e);  
    cout << "main : " << e << endl;  
}
```

```
f : 10  
main : 0
```

Passage par référence

```
void g (int& i) {  
    i = 10;  
    cout << "g : " << i << endl;  
}  
  
void main () {  
    int e = 0;  
    g(e);  
    cout << "main : " << e << endl;  
}
```

```
g : 10  
main : 10
```

Passage d'objets par valeur, par référence constante

Passage par valeur

```
double somme (BigData data) {  
    double s = 0;  
    for (...)  
        s += data[i];  
    return s;  
}
```

```
BigData test ("un_fichier.csv");  
cout << somme(test) << endl;
```

L'objet est **copié**
(constructeur par copie appelé)

Passage par référence constante

```
double somme (const BigData& data) {  
    double s = 0;  
    for (...)  
        s += data[i];  
    return s;  
}
```

```
BigData test ("un_fichier.csv");  
cout << somme(test) << endl;
```

Une copie inutile est évitée,
mais l'objet **reste protégé**
d'éventuelles modification.

→ Utile pour les objets volumineux (tableaux, listes...). Parfois indispensable (objet non copiable).

Passage d'objets par valeur, par référence constante

Passage par valeur

```
double somme (BigData data) {  
    double s = 0;  
    for (...)  
        s += data[i];  
    return s;  
}
```

```
BigData test ("un_fichier.csv");  
cout << somme(test) << endl;
```

L'objet est **copié**
(constructeur par copie appelé)

Passage par référence constante

```
double somme (const BigData& data) {  
    double s = 0;  
    for (...)  
        s += data[i];  
    return s;  
}
```

```
BigData test ("un_fichier.csv");  
cout << somme(test) << endl;
```

Une copie inutile est évitée,
mais l'objet **reste protégé**
d'éventuelles modification.

→ Utile pour les objets volumineux (tableaux, listes...). Parfois indispensable (objet non copiable).

Copie en C++

En C++ , le comportement par défaut lors d'un appel ou d'une assignation est la copie (profonde). C'est à vous de choisir entre la *copie*, le *passage par référence constante* (pas de modifications), ou le *passage par référence simple* (pour effectuer des modifications)

Différent de Python, où les objets composites ne sont pas copiés en profondeur :

```
def test (liste, nombre):  
    liste[2] = 0  
    nombre = 0
```

```
liste = [1,2,3]  
nombre = 4  
somme(liste, nombre)  
print(liste, nombre)
```

```
[1,2,0] 4
```


Copie en C++

En C++ , le comportement par défaut lors d'un appel ou d'une assignation est la copie (profonde). C'est à vous de choisir entre la *copie*, le *passage par référence constante* (pas de modifications), ou le *passage par référence simple* (pour effectuer des modifications)

Différent de Python, où les objets composites ne sont pas copiés en profondeur :

```
def test (liste, nombre):  
    liste[2] = 0  
    nombre = 0
```

```
liste = [1,2,3]  
nombre = 4  
somme(liste, nombre)  
print(liste, nombre)
```

```
[1,2,0] 4
```

Copie en C++

En C++ , le comportement par défaut lors d'un appel ou d'une assignation est la copie (profonde). C'est à vous de choisir entre la *copie*, le *passage par référence constante* (pas de modifications), ou le *passage par référence simple* (pour effectuer des modifications)

Différent de Python, où les objets composites ne sont pas copiés en profondeur :

```
def test (liste, nombre):  
    liste[2] = 0  
    nombre = 0
```

```
liste = [1,2,3]  
nombre = 4  
somme(liste, nombre)  
print(liste, nombre)
```

```
[1,2,0] 4
```