



Programmation C++ moderne

Dr. Eng. Chiheb Ameer ABID

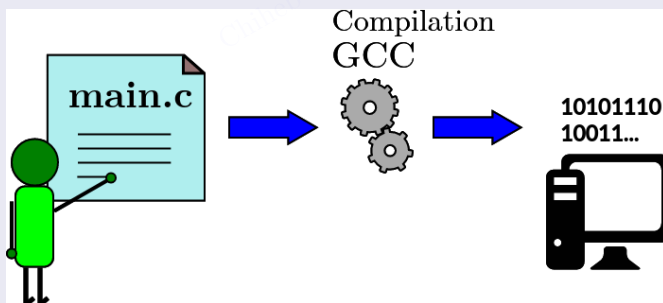
[in /in/chiheb-ameur-abid](#)
[✉ chiheb.abid@gmail.com](mailto:chiheb.abid@gmail.com)

Les langages C/C++

Les langages C et C++

- ↳ Performance (Langages compilés)
- ↳ Empreinte mémoire faible
- ↳ Portabilité
- ↳ Bibliothèques existantes (C depuis 1971, C++ depuis 1983)

Compilation



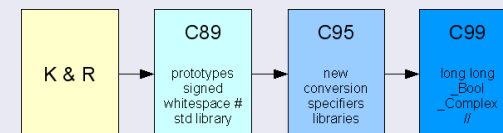
Plan

- 1 Historique et évolution du langage C++
- 2 Les espaces de noms
- 3 Littéraux
- 4 Les nouveautés du langage C++11/C++17
- 5 Références et expressions

Les langages C/C++

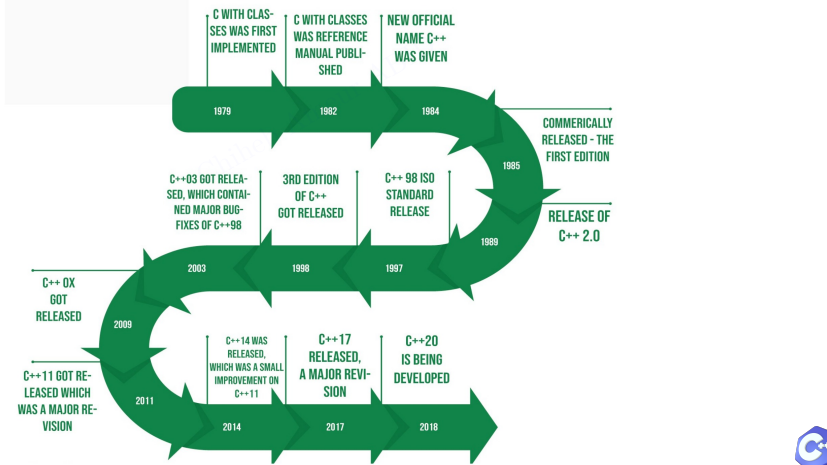
Historique

- ↳ En 1970, Ken Thompson, créa un nouveau langage : Le B
 - ↳ Simple, mais trop dépendant de l'architecture utilisée
- ↳ En 1971, Dennis Ritchie commence à mettre au point le successeur du B, le C.
 - ↳ Portable,
 - ↳ Langage bas niveau : performant (peut créer du code aussi rapide que de l'assembleur)
 - ↳ Permet de traiter des problèmes de haut niveau
- ↳ En 1989, l'ANSI (American National Standards Institute) normalisa le C sous les dénominations ANSI C ou C89



Historique

- En 1983, Bjarne Stroustrup des laboratoires Bell crée le C++
 - Basé sur le C, il garde une forte compatibilité avec le C
- La première normalisation de C++ date de 1998 (C++98)

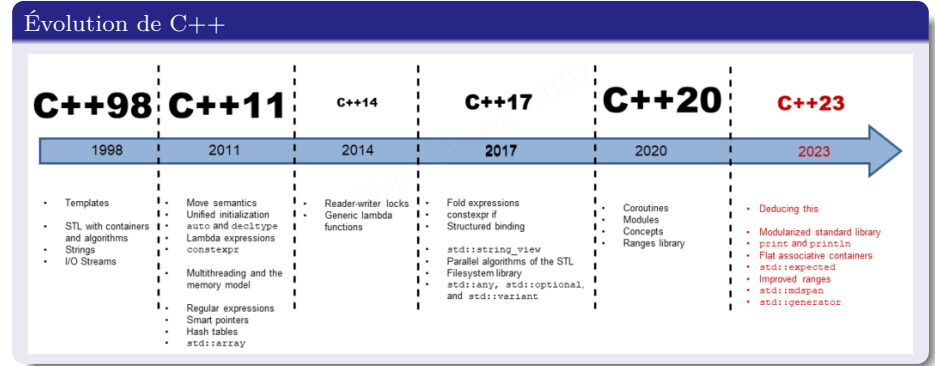


Compilation

Principaux compilateurs C/C++

- GCC (GNU Compiler Collection) : il s'agit d'un ensemble d'outils Open Source analysant plusieurs langages (C, C++, Objective C, Ada, Go, Fortran...)
 - Supporte une multitude d'architectures cibles
 - GCC autorise la compilation croisée
 - GCC est disponible sous la plupart des systèmes de type Unix (comme Linux) et sous Windows en utilisant l'environnement Cygwin
- Clang : c'est la principale alternative au compilateur C++ de GCC
 - Il s'intègre au sein du projet plus global LLVM (framework de développement de compilateur) pour générer une représentation compilée intermédiaire traduisible ensuite sur différentes architectures
 - Clang, de conception plus récente, possède une architecture plus modulaire que GCC et permet d'obtenir des programmes offrant des performances plus ou moins similaires à GCC
- Intel C++ Compiler : compilateur propriétaire (et gratuit) est développé par Intel pour la compilation à destination de ses propres processeurs (de type x86) en utilisant certaines optimisations maison.
- Visual C++ Compiler : compilateur propriétaire de Microsoft pour Windows accompagne l'environnement de développement Visual Studio de Microsoft

Évolution de C++



Compilation

Compilateur GCC

- GCC est un logiciel libre capable de compiler divers langages de programmation, dont C, C++, Objective-C, Java, Ada et Fortran
- Pour faire référence précisément aux compilateurs de chaque langage
 - gcc : le compilateur C de GNU
 - g++ : le compilateur C++ de GNU
 - gcj : le compilateur Java de GNU
 - gobjc : le compilateur Objective-C de GNU
 - gobjc++ : le compilateur Objective-C++ de GNU
 - gnat : le compilateur Ada de GNU
 - gfortran : le compilateur Fortran de GNU

Compilation

Compilateur GCC

➤ Installer GCC depuis la CLI

```
sudo apt-get install build-essential
```

➤ Compiler un programme

📄 Programme écrit C

```
gcc main.c fonctions.c -o Programme
```

📄 Programme écrit en C++

```
g++ main.cpp fonctions.cpp -o Programme -std=c++17
```

➤ Pour le lancer le programme

```
./Programme
```



Modèle mémoire d'un processus

Un processus

➤ Un processus est une instance d'un programme en cours d'exécution chargé en mémoire vive

	(a) Generic	(b) UNIX	(c) Win32
Start of memory	Code	Text	Reserved by OS
	Data	Data	Stack
	Heap	BSS	Heap
	↓	Heap	Code
		↓	Constants
End of memory	↑	↑	Static variables
	Stack	Stack	Uninitialized variables



Étapes de compilation

Un toolchain est constitué par un ensemble d'outils permettant de produire le fichier exécutable

➤ C++ Preprocessor : Inclure le contenu des fichiers entêtes (headers), générer les macros et remplacer les constantes définies par *#define*

```
g++ -E fichier.cpp
```

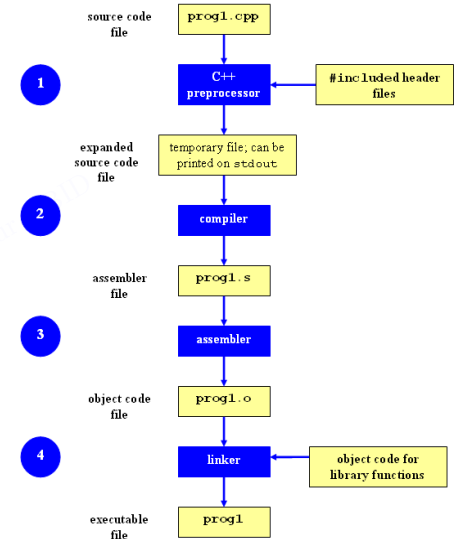
➤ Compiler : Traduire en assembleur

```
g++ -S fichier.cpp
```

➤ Assembleur : produire fichier objet binaire en traduisant les instruction assembleur en langage machine (binaire)

```
g++ -c fichier.cpp
```

➤ Linker : édition des liens du fichier objet avec les autres fichiers objets de n'importe quelle bibliothèque utilisée afin de produire le fichier exécutable



Modèle mémoire

➤ Code/Text : contient les instructions du programme (lecture seule)

➤ Data : variables globales et statiques initialisées par le programmeur

```
static int v=0;
```

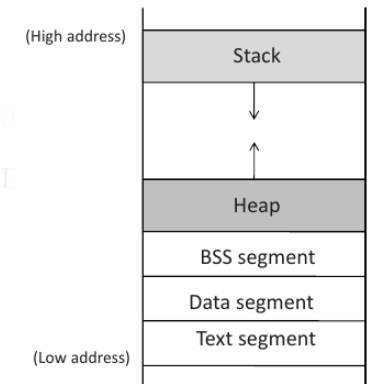
➤ BSS (Block Started by Symbol) : variables globales et statiques non initialisées

```
static int v;
```

➤ Heap : allocation dynamique

```
new int;
```

➤ Stack : fonctionne en mode LIFO ; stockage de variables locales (définies dans une fonction) et les informations relatives aux appels des fonctions



Plan

- 1 Historique et évolution du langage C++
- 2 Les espaces de noms
- 3 Littéraux
- 4 Les nouveautés du langage C++11/C++17
- 5 Références et expressions

Chiheb Ameer ABID

Espace de noms

Présentation

- ↳ Un espace de noms est une région qui fournit une portée aux identificateurs déclarés à l'intérieur
 - ↳ Les espaces de noms fournissent un mécanisme pour réduire le risque de conflits de noms
- ↳ Évolutions des espaces de noms
 - ↳ C++11 : espaces de noms `inline`
 - ↳ C++17 : espaces de noms imbriqués

↳ Éviter le problème dit de **pollution de l'espace de noms**

- ↳ Un même identificateur peut être utilisé par plusieurs modules ou bibliothèques
- ↳ Éviter les ambiguïtés

Présentation

1 Déclaration d'un espace de noms

```
1 namespace nom {
2   // Corps de l'espace de noms (code)
3   ...
4 }
```

- ↳ `nom` : Identifiant qui nomme l'espace de noms
- ↳ Corps de l'espace de noms : Code contenu dans l'espace de noms

```
1 namespace mon_module {
2   // Déclarations usuelles
3   double taux;
4   double conversion(double);
5 }
```

2 Extension d'un espace de noms

- ↳ Un espace de noms peut être étendu dans plusieurs fichiers ou sections du code

```
1 namespace nom_espace {
2   // Nouvelles déclarations ajoutées
3   // ...
4 }
```

Présentation

- 3 Accès et directives
 - ↳ L'accès aux éléments se fait via l'opérateur de résolution de portée `::`
- ↳ Accès qualifié

```
1 nom_espace::element(); // Appel de fonction
2 nom_espace::Classe obj; // Instanciation
```

- ↳ À l'intérieur de `mon_module`, on utilise directement le nom `taux`

```
1 double mon_module::conversion(double a) {
2   return a*taux; // mon_module::taux
3 }
```

↳ Directive `using` pour importer

- ↳ `using namespace nom_espace;` : Importe tout (utiliser avec prudence)
- ↳ `using nom_espace::element;` : Import sélectif

↳ Exemple d'utilisation

```
1 using namespace MaLib;
2 fonction(); // Accès direct après using
```



- ↳ L'espace des déclarations globales d'un programme est aussi un espace de noms dit portée globale

- ↳ `::x` fait référence à l'identificateur `x` de la portée globale

Espace de noms

Utiliser les espaces de noms

- La déclaration `using` permet de faire entrer (connaître) un identificateur dans la portée courante

```
1 using mon_module::taux;  
2 std::cout << taux;
```

- Si on fait entrer `taux` dans la portée globale alors `::taux` et `mon_module::taux` deviennent des écritures équivalentes

- Il ne doit pas y avoir d'autre `taux` dans la portée courante

- La directive `using` permet de rendre visibles les noms définis dans un espace de noms

```
1 using namespace mon_module;  
2 std::cout << conversion(3);
```

Espace de noms

Utilisation de la directive `using`

- Risques d'ambiguïtés si plusieurs espaces de noms comportant des identificateurs identiques sont rendus visibles

```
1 using namespace mon_module;  
2 //espace de nom déclarant taux  
3 using namespace son_module;  
4 //espace de nom déclarant taux  
5 std::cout << taux; //appel ambigu
```

- Le compilateur signale

- Les ambiguïtés entre identifiants des espaces de noms rendus visibles
- Mais pas les surcharges de fonctions à travers les espaces de noms!

- Il est parfois plus prudent d'utiliser une déclaration `using` qu'une directive `using`!

Espace de noms

Espace de noms imbriqués

- Permettent de créer une hiérarchie d'espaces de noms

- Une organisation plus fine du code et une meilleure gestion des conflits de noms.

- Utilisation des espaces de noms imbriqués

```
1 namespace foo {  
2     namespace bar {  
3         namespace impl {  
4             // ...  
5         }  
6     }  
7 }
```

- En C++, une écriture équivalente

```
1 // Une autre écriture équivalente  
2 namespace foo::bar::impl {  
3     // ...  
4 }
```

Espace de noms

Espace de noms inline

- Permet de créer des espaces de noms imbriqués dont les membres sont automatiquement considérés comme faisant partie de l'espace de noms englobant.

- Les inline namespaces sont utilisés pour gérer les versions d'une bibliothèque et éviter les conflits de noms entre les différentes versions

Exemple de gestion des versions avec `inline namespace`

```
1 namespace modernlib  
2 {  
3     #ifndef LIB_VERSION_2  
4     inline namespace version_1 {  
5         template<typename T>  
6         int test(T value) { return 1; }  
7     }  
8     #endif  
9     #ifdef LIB_VERSION_2  
10    inline namespace version_2 {  
11        template<typename T>  
12        int test(T value) { return 2; }  
13    }  
14    #endif  
15 }
```

Alias d'espaces de noms

- ↳ Un alias de namespace est un identificateur qui peut être introduit comme un alias pour un espace de noms entier

- ▀ permet de créer des noms plus courts pour les namespaces imbriqués en profondeur ou les namespaces avec des noms longs

- ↳ Syntaxe

```
namespace alias_name = ns::name;.
```

Chiheb Ameer ABID

```
1 namespace foobar {
2   namespace miscellany {
3     namespace experimental {
4       int get_meaning_of_life() {return 42;}
5       void greet() {std::cout << "hello\n";}
6     }
7   }
8 }
9 int main() {
10  namespace n = foobar::miscellany::experimental; // Définition d'un alias
11  n::greet();
12  std::cout << n::get_meaning_of_life() << '\n';
13 }
```



Plan

- 1 Historique et évolution du langage C++
- 2 Les espaces de noms
- 3 Littéraires
- 4 Les nouveautés du langage C++11/C++17
- 5 Références et expressions

Chiheb Ameer ABID



Espace de noms

Espaces de noms anonymes

- ↳ Les entités définies dans un namespace anonyme ne sont visibles que dans son unité de traduction associée

- ▀ Elles ne sont pas accessibles depuis d'autres unités de traduction

- ↳ Syntaxe

```
namespace alias name = ns::name;.
```

```
1 namespace {
2   const int forty_two = 42;
3   int x;
4 }
5 int main() {
6   x = forty_two;
7   std::cout << x << '\n';
8 }
```

- ▀ La constante `forty_two` et la variable statique `x` sont uniquement accessibles dans le fichier source où ils sont définis



Les littéraires

Présentation

- ↳ Un littéral (aussi appelé constante littérale) est une valeur écrite exactement comme elle doit être interprétée.

- ↳ Exemples de littéraires

```
1 "Bonjour, le monde"
2 "Bjarne"
3 'a' // Caractère minuscule a
4 'A' // Caractère majuscule A
5 123
6 123U // Entier non signé 123
7 1'000'000 // Entier séparé par des apostrophes pour la lisibilité
8 3.1415 // Nombre décimal
9 1.0L // Nombre décimal avec suffixe L indiquant un long
10 1.23456789e-10 // Nombre scientifique
```



Littéraux de caractère

Littéraux de caractère

- Un littéral de caractère est constitué d'un préfixe optionnel suivi d'un ou plusieurs caractères placés entre apostrophes simples.
- Le type de littéral de caractère est déterminé par le préfixe (ou son absence)
 - ➊ Littéraux de caractères ordinaires de type `char`, par exemple `'a'`
 - ➋ Littéraux de caractères étendus de type `wchar_t`, par exemple `L'a'`
 - ➌ Littéraux de caractères UTF-16 de type `char16_t`, par exemple `u'a'`
 - ➍ Littéraux de caractères UTF-32 de type `char32_t`, par exemple `U'a'`
- Les caractères spéciaux peuvent être représentés par une séquence d'échappement

Les chaînes littérales brutes

Littéraux de chaînes de caractères

- C++11 a introduit les chaînes littérales brutes pour ne pas interpréter les caractères spéciaux
 - ☞ Il suffit de mettre la chaîne de caractères entre `R"` et `"`)

Exemple

- Soit à afficher la chaîne :

```
printf("Hello, %s%c\n", "World", '!');
```

- En C++03

```
std::cout << "printf(\\\"Hello, %s%c\\n\", \\\"World\\\", '!');"<<std::endl;
```

- En C++11

```
std::cout << R"(printf("Hello, %s%c\n", "World", '!'))" <<std::endl;
```

Les littéraux entiers

Littéraux de chaînes de caractères

- Les entiers peuvent être spécifiés en base décimale, binaire, hexadécimale et octale
- La base du nombre est indiquée par un préfixe (ou son absence)

Préfixe	Base
Aucun	Décimale
0 initial	Octale
0b ou 0B	Binaire
0x ou 0X	Hexadécimale

- Différents suffixes peuvent être spécifiés pour contrôler le type du littéral

Suffixe	Type
u ou U	unsigned (entier non signé)
l ou L	long (long int)
u ou U et l ou L (ensemble)	unsigned long long (entier non signé long long)
ll ou LL	long long (long long int)
u ou U et ll ou LL (ensemble)	unsigned long long (entier non signé long long)

- Un apostrophe simple est utilisé comme séparateur de chiffres : `1'000'000`
- Un littéral entier est toujours non négatif; ainsi, par exemple, `-1` est le littéral entier 1 avec l'opération de négation appliquée

Littéraux définis par l'utilisateur

Littéraux définis par l'utilisateur

- Les littéraux définis par l'utilisateur sont une fonctionnalité du langage qui permet de créer des raccourcis syntaxiques pour des valeurs ou des expressions courantes, en utilisant des suffixes personnalisés
- Étapes de création des littéraux définis par l'utilisateur
 - ☞ Définir les littéraux dans un espace de noms séparé pour éviter les conflits de noms.
 - ☞ Toujours préfixer le suffixe défini par l'utilisateur avec `_`
 - ☞ Définir un opérateur littéral de l'une des formes :

```
1 T operator "" _suffix(unsigned long int);
2 T operator "" _suffix(long double);
3 T operator "" _suffix(char);
4 T operator "" _suffix(wchar_t);
5 T operator "" _suffix(char16_t);
6 T operator "" _suffix(char32_t);
7 T operator "" _suffix(char const *, std::size_t);
8 T operator "" _suffix(wchar_t const *, std::size_t);
9 T operator "" _suffix(char16_t const *, std::size_t);
10 T operator "" _suffix(char32_t const *, std::size_t);
```

Littéraux définis par l'utilisateur

Exemple de définition d'un littéral

```
1 namespace compunits
2 {
3     constexpr size_t operator "" _KB(unsigned long long const size)
4     {
5         return static_cast<size_t>(size * 1024);
6     }
7 }
8 using compunits::operator "" _KB ; // using namespace compunits;
9
10 auto size{ 4_KB }; // size_t size = 4096;
11 using byte = unsigned char;
12 auto buffer {std::array<byte, 1_KB>{}}
```

Plan

- 1 Historique et évolution du langage C++
- 2 Les espaces de noms
- 3 Littéraux
- 4 Les nouveautés du langage C++11/C++17
- 5 Références et expressions

La constante nullptr

La constante nullptr (depuis C++11)

- En C++, la constante NULL correspond à la valeur entière 0 pour représenter un pointeur nul
 - ☞ Source d'ambiguïté
- La constante nullptr vise à résoudre ce problème

```
1 void foo(char *);
2 void foo(int);
3
4 ☞ L'appel foo(NULL) va appeler probablement foo(int)
5
6 char *pc = nullptr; // OK
7 int *pi = nullptr; // OK
8 bool b = nullptr; // OK. b is false.
9 int i = nullptr; // error
10 foo(nullptr); // calls foo(char *), not foo(int);
```

Les expressions constexpr

Présentation

- Les expressions constexpr ont été introduites en C++11
- constexpr
 - 1 Déclarer une variable constante dont la valeur est connue lors de la compilation
 - 2 Définir une fonction qui peut produire une valeur constante à la compilation si ses arguments sont des valeurs constantes
- L'objectif est de réaliser des calculs pendant la compilation
- En C++, la méta-programmation se base sur les templates méta-fonctions et les fonctions constexpr
 - ☞ Écrire du code qui sera interprété non pas à l'exécution, mais pendant la compilation

constexpr

Les fonctions constexpr

➤ En C++11, une `constexpr` est assez restrictive

- ☞ Elle ne peut rien lire ou écrire en dehors de la fonction
- ☞ Elle ne peut pas contenir de variables
- ☞ Ne peut pas contenir de structures de contrôle comme `if` ou `for`
- ☞ Elle ne peut contenir qu'une seule instruction de calcul
- ☞ Elle ne peut appeler que des fonctions qui sont également `constexpr`.

Exemple

```
1 constexpr long fibonacci (long n)
2 {
3     return n <= 2 ? 1 : fibonacci ( n - 1) + fibonacci ( n - 2);
4 }
5
6 auto x1 {fibonacci(2)}; // Run-time execution
7 constexpr auto x2 {fibonacci(2)}; // Compile-time execution
```



constexpr

Exemple avec constexpr en C++17

```
1 constexpr int factorial(int n) {
2     int result {1};
3     for (int i {1}; i <= n; ++i) {
4         result *= i;
5     }
6     return result;
7 }
8
9 auto x1 (factorial(2)); // Run-time execution
10 constexpr auto x2 (factorial(2)); // Compile-time execution
```



Les fonctions constexpr

- En C++17, moins de restrictions ont été imposées sur une fonction `constexpr`
 - ☞ Doit avoir un type de retour littéral : un type qui peut être utilisé dans une expression constante, comme un type intégral, un type flottant, un pointeur, une référence, une énumération, ou une classe avec un constructeur `constexpr`.
 - ☞ La fonction ne doit pas avoir de spécificateur virtuel, `override`, `final`, ou `noexcept`
 - ☞ Pas des allocations dynamiques, pas des appels à des fonctions non constantes, pas des modifications de variables globales, ou des instructions `goto`
 - ☞ Peut appeler d'autres fonctions `constexpr`, mais pas des fonctions non constantes, sauf si elles sont marquées comme `constexpr` ou `constexpr`.
 - ☞ Peut contenir des instructions `if`, `switch`, `for`, `while`, ou `do-while`, mais pas des instructions `break` ou `continue` qui ne sont pas immédiatement suivies d'une instruction de fin de bloc.
 - ☞ Peut contenir des variables locales, mais elles doivent être initialisées avec des expressions constantes
 - ☞ Peut contenir des assertions statiques, des déclarations `using`, des déclarations `typedef`, ou des déclarations de types anonymes.
 - ☞ Peut contenir des expressions lambda, mais elles doivent être marquées comme `constexpr` et respecter les mêmes restrictions que la fonction englobante.



Le "range-based" for loop

Le range-based for loop

- Depuis C++11, une nouvelle variante de la boucle `for` a été introduite pour parcourir une collection des valeurs : tableaux ou conteneurs

```
1 for(type d'un élément ou auto identifiant : tableau/conteneur à parcourir) {
2     // Manipuler identifiant, en l'affichant par exemple.
3 }
```

- Exemple : par copie

```
1 int tab[] { 1, 2, 3, 4, 5 };
2 for (auto elt : v)
3     std::cout << elt << '\n';
```

- ☞ `elt` contient une copie d'un élément du tableau
- ☞ La modification de `elt` ne modifie pas la valeur dans le tableau

- Exemple : par référence

```
1 int tab[] { 1, 2, 3, 4, 5 };
2 for (auto &elt : v)
3     std::cout << elt << '\n';
```

- ☞ On ajoute `const` pour empêcher la modification des valeurs



Initialisation uniforme

- ↳ Fonctionnalité introduite avec le standard C++11
- ↳ Utiliser une syntaxe cohérente pour initialiser des variables et des objets, allant des types primitifs aux agrégats
 - ✓ Éviter certaines ambiguïtés syntaxiques
 - ✓ Prévenir les conversions implicites
- ↳ Basée sur l'utilisation des accolades `{}`

```
type var_name{arg1, arg2, ....arg n}
```

- ↳ Éviter l'ambiguïté avec l'opérateur d'affectation

```
1 std::vector<int> v1={1,2,3};
2 auto v2=v1; // Pas d'affectation ; constructeur par copie
3 auto v3 {v1};
```



Initialisation uniforme

- ↳ Initialisation des types primitifs

- ↳ Légèrement plus sûre pour les conversions étroites

```
1 int count1(7.5); // Compile avec conversion implicite
2 int count2 = 5.3; // Compile avec une conversion implicite
3 int count3{0.3}; // Erreur de compilation
```

- ↳ Initialisation par valeur

- ↳ Initialise à 0 les types primitifs
- ↳ Fait appel au constructeur par défaut pour les classes

```
1 int x; // Valeur quelconque
2 int i{}; // i prend 0
```

- ↳ Éviter une ambiguïté syntaxique

```
1 std::string s1(); // Déclaration d'une fonction qui renvoie un std:string
2 std::string s2{}; // Création d'une chaîne vide
3 std::string s3; // Création d'une chaîne vide
```



Initialisation uniforme

- ↳ Initialisation des agrégats

- ↳ Tableaux

```
1 int arr[] { 1, 2, 3, 4 };
2 float numbers[] = { 0.1f, 1.1f, 2.2f, 3.f, 4.f, 5. };
3 int nums[10] { 1 }; // 1, puis des 0s
```

- ↳ Structures

```
1 struct CarInfo {
2     std::string name;
3     unsigned year;
4     unsigned seats;
5     double power;
6 };
7 CarInfo any; // Des valeurs aléatoires pour les membres de types primitifs
8 CarInfo empty{}; // 0 pour les membres de types primitifs
9 CarInfo firstCar{"Megane", 2003, 5, 116 };
10 CarInfo partial{"unknown"}; // Les autres champs prennent la valeur 0
11 CarInfo largeCar{"large car", 1975, 10};
```



Initialisation uniforme

- ↳ Initialiser les objets

```
Type objet {...};
```

- ↳ L'initialisation d'un objet est effectuée selon l'ordre suivant :

- 1 Les constructeurs à liste d'initialisation utilisant un paramètre de type `std::initializer_list<T>`
- 2 Les autres constructeurs
- 3 L'affectation directe des membres : n'est autorisée que pour les tableaux et les classes si toutes les variables (non statiques) sont publiques et que la classe n'a pas de constructeur défini par l'utilisateur



Initialisation uniforme d'un objet

- ① Les constructeurs à liste d'initialisation utilisant un paramètre de type `std::initializer_list<T>`

```
1 struct MaClasse {
2     MaClasse(std::initializer_list<int> liste) {...}
3     MaClasse(int x) {...}
4 };
5
6 MaClasse obj1{1, 2, 3}; // Appelle le constructeur avec initializer_list
7 MaClasse obj2{10}; // Appelle le constructeur avec initializer_list
8 MaClasse obj3(10); // Appelle le constructeur avec un entier
```



Initialisation uniforme d'un objet

- ② Si aucun constructeur avec `std::initializer_list<T>` n'est disponible, le compilateur utilise les autres constructeurs correspondants

```
1 struct MaClasse {
2     MaClasse(int x) { ... }
3     MaClasse(int x, int y) { ... }
4 };
5
6 MaClasse obj1{10}; // Appelle MaClasse(int x)
7 MaClasse obj2{10, 20}; // Appelle MaClasse(int x, int y)
8 MaClasse obj3(10); // Appelle MaClasse(int x)
```



Initialisation uniforme d'un objet

- ③ L'affectation directe des membres si toutes les variables (non statiques) sont publiques et que la classe n'a pas de constructeur défini par l'utilisateur
- Exemple d'une classe dont tous les membres non statiques sont publics et pas de constructeur défini

```
1 struct Point {
2     int x, y;
3     int getX() const { ... }
4     int getY() const { ... }
5 };
6
7 Point p{10, 20}; // Affectation directe des membres
```

- Si la classe a un constructeur défini par l'utilisateur ou des membres privés, l'affectation directe n'est pas autorisée

```
1 struct Point {
2     Point(int x, int y) : x(x), y(y) {} // Constructeur défini par l'utilisateur
3     int x, y, couleur;
4 };
5
6 Point p1{10, 20, 13}; // Erreur de compilation : affectation directe interdite
7 Point p2(10, 20); // Correct : utilise le constructeur
```



Initialisation uniforme d'un objet

```
1 struct Box { };
2 struct Product {
3     Product(): name{"default product"} { }
4     Product(const Box& b) : name{"box"} { }
5     std::string name;
6 };
7
8 Product p(); // Déclaration d'une fonction !?
9 Product p1; // OK
10 Product p2{ }; // OK
11
12 Product q(Box()); // Déclaration d'une fonction !?
13 Product p3{Box()}; // OK
14 Product p4{Box{ }}; // OK
```





➤ Pour les conteneurs, les parenthèses (...) sont généralement utilisées pour spécifier le nombre d'éléments

- ☛ Appel de constructeurs prenant des arguments

```
1 std::vector<int> v1(10); // 10 éléments, initialisés à 0
2 std::vector<int> v2(10,2); // 10 éléments, initialisés à 2
```

➤ Pour les conteneurs, les accolades {...} sont généralement utilisées pour spécifier une liste de valeurs

- ☛ Appel du constructeur ayant comme paramètre std::initializer_list

```
1 std::vector<int> v1{10}; // 1 élément avec la valeur 10
2 std::vector<int> v2{10,2}; // 2 éléments avec les valeurs 10 et 2
```



➤ Lorsque vous utilisez des initialiseurs accolés avec le mot-clé auto, il est important de faire attention

- ☛ Avec C++11 et C++14, l'initialisation d'une valeur unique est interprétée comme std::initializer_list<int>
- ☛ Par contre avec C++17, l'initialisation d'une valeur unique est interprétée comme valeur unique

Chiheb Ameur ABID

```
1 /* C++11 and C++14 */
2 auto i {10}; // i est de type std::initializer_list<int> !!!
3 auto pi = {3.14159}; // pi est de type std::initializer_list<double>
4 auto list1{1, 2, 3}; // list1 est de type std::initializer_list<int>
5 auto list2 = {4, 5, 6}; // list2 est de type std::initializer_list<int>
6
7 /* C++17 */
8 auto i {10}; // i has type int
9 auto pi = {3.14159}; // pi has type std::initializer_list<double>
10 auto list1{1, 2, 3}; // error: does not compile!
11 auto list2 = {4, 5, 6}; // list2 has type std::initializer_list<int>
```

Les énumérations typées

Les énumérations typées

➤ En C++ traditionnel, les types énumérés ne sont pas sûrs pour le type

- ☛ Les types énumérés sont traités comme entiers
- ☛ Comparer directement deux types énumérés complètement différents!?

```
1 enum Couleur {Rouge, Vert} couleur;
2 enum Forme {Rectangle, Ellipse} forme;
3 couleur=Rouge;
4 forme=Rectangle;
5 if (forme==couleur) ....// Le compilateur ne signale rien
```

➤ Les énumérations typées visent à éviter ce problème

```
1 enum class Couleur {Rouge, Vert} couleur;
2 enum class Forme {Rectangle, Ellipse} forme;
3 couleur=Couleur::Rouge;
4 forme=Forme::Rectangle;
5 if (forme==couleur) ....// Erreur signalé par le compilateur
```

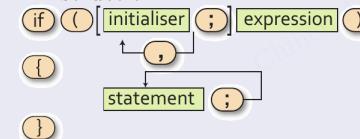
Initialisation dans les instructions if et switch

Initialisation dans les instructions if et switch

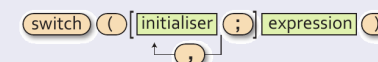
➤ Fonctionnalité introduite en C++17

➤ Permettre l'initialisation de variables directement dans les instructions if et switch

☛ L'instruction if



☛ L'instruction switch



Initialisation dans l'instruction if

```
1 int valeur {10};
2
3 if (int result = valeur * 2; result > 20) {
4     std::cout << "Le résultat est > 20 et vaut : " << result << std::endl;
5 } else {
6     std::cout << "Le résultat est =< à 20 et vaut : "<< result << std::endl;
7 }
```

Initialisation dans l'instruction switch

```
1 #include <iostream>
2 #include <cstdlib>
3 int main() {
4     srand(time(NULL));
5     switch (int i = rand() % 100; i) {
6     case 42:
7         std::cout << "La chance! Vous avez obtenu 42." << std::endl;
8         break;
9     default:
10        std::cout << "Vous avez obtenu " << i << "." << std::endl;
11        break;
12    }
13 }
```



Liaison structurée

Liaison structurée avec les tableaux

- ➔ Le nombre d'identifiants doit correspondre exactement au nombre d'éléments dans le tableau.

```
1 double myArray[3] { 1.0, 2.0, 3.0 };
2 auto [a, b, c] = myArray;
3 auto [d, e, f] {myArray};
4 auto [g, h] myArray; // Erreur de compilation
```



Liaison structurée

Présentation

- ➔ Fonctionnalité introduite en C++17
 - ☞ Faciliter l'accès aux éléments individuels de l'objet
- ➔ Syntaxe générale

```
1 auto [a, b, c, ...] = expression;
2 auto [a, b, c, ...] { expression };
3 auto [a, b, c, ...] ( expression );
```

- ➔ Appliquée aux types suivants

- 1 Tableaux
- 2 `std::pair<>` et `std::tuple<>`
- 3 Les structures sur les membres non statiques et publiques



Liaison structurée

Liaison structurée avec `std::pair<>` et `std::tuple<>`

- ➔ Le nombre d'identifiants doit correspondre exactement au nombre d'éléments dans les objets

```
1 // Avec std::pair
2 std::pair<int, double> maPaire = {1, 4.5};
3 auto [monEntier, monDouble] = maPaire; // Décomposition de la paire
4
5 // Avec std::tuple
6 std::tuple<int, double, std::string> monTuple = {2, 6.7, "Texte"};
7 auto [entier, dbl, texte] = monTuple; // Décomposition du tuple
```



Liaison structurée

Liaison structurée avec les structures

- Par défaut, applicable avec les structures sont tous les membres sont publics et non statiques
- Pour les classes personnalisées, il y a la possibilité d'activer la liaison structurée
 - ☞ En redéfinissant les modèles `get<N>`, `std::tuple_size` et `std::tuple_element`
 - ☞ En utilisant, la fonction `std::tie`

```

1 struct S {
2     int n;
3     std::string st;
4     float d;
5 };
6
7 S s;
8 auto [a, b, c] = s;
```



Plan

- 1 Historique et évolution du langage C++
- 2 Les espaces de noms
- 3 Littéraux
- 4 Les nouveautés du langage C++11/C++17
- 5 Références et expressions



Liaison structurée

Utilisation de `std::tie`

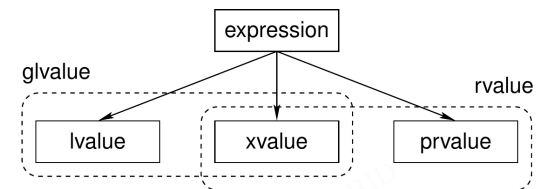
- `std::tie` crée un tuple de références à partir de ses arguments
 - ☞ Le tuple créé ne possède pas ses propres valeurs mais agit comme un alias pour les variables existantes
- Utile pour décomposer des tuples et pour l'affectation multiple

```

1 int tab[] {0,1,2,3,4,5};
2 auto [a,b] {std::tie(tab[0],tab[3])}; // a=tab[0] et b=tab[3]
3 std::tie(tab[0],tab[3])=std::tuple{0,0}; // tab[0]=0 et tab[3]=0
4
5 struct S {
6     int n;
7     std::string st;
8     float d;
9 } o;
10 auto [a,b] = std::tie(o.n,o.st);
11 std::tie(a,b) = std::tie(o.n,o.st);
```



Catégories d'expressions



Notion de value-ness

- Chaque expression peut être classée dans exactement l'une des trois catégories suivantes :
 - ☞ `lvalue` (left value)
 - ☞ `prvalue` (pure rvalue)
 - ☞ `xvalue` (valeur expirante)
- Une expression qui est une `lvalue` ou une `xvalue` est appelée une `glvalue` (generalized lvalue).
- Une expression qui est une `prvalue` ou une `xvalue` est appelée une `rvalue`
- Chaque expression est soit une `lvalue`, soit une `rvalue` (mais pas les deux).
- La possibilité de déplacer (au lieu de copier) une valeur dépend de son type : `lvalue` ou `rvalue`



Notion de value-ness

Lvalue

↳ lvalue (left value) est une expression qui :

- Désigne une fonction ou un objet ; et
- Possède une identité : elle occupe un emplacement identifiable en mémoire et, par conséquent, on peut en principe prendre son adresse

Une expression lvalue est tout ce qui pouvait traditionnellement être à gauche de l'opérateur d'affectation

Expressions lvalue

↳ Les objets et fonctions nommés sont des lvalues

```
1 int getValue(); // getValue est une lvalue (mais pas getValue()) non assignable
2 int i = 0;
3 const int j = 1;
4 i = j + 1; // i et j sont des lvalues, j est une lvalue non assignable
```



Notion de value-ness

Expressions prvalue

↳ Une prvalue est une expression qui :

- Est un objet temporaire ou un sous-objet d'un objet temporaire, ou une valeur qui n'est pas associée à un objet
- N'a pas d'identité (c'est-à-dire, qu'on ne peut pas en principe prendre son adresse).

↳ Une prvalue est un type spécifique de rvalue

- Tous les prvalues sont des rvalues, mais tous les rvalues ne sont pas des prvalues.
- Le terme "rvalue" est plus général et englobe les prvalues ainsi que d'autres types d'expressions qui ne désignent pas des objets.

Expressions prvalue

↳ Les objets temporaires sont des prvalues

```
1 std::vector<int> v;
2 v = std::vector<int>(10, 2); // std::vector<int>(10, 2) est une prvalue
3 std::complex<double> u;
4 u = std::complex<double>(1, 2); // std::complex<double>(1, 2) est une prvalue
```



Expressions lvalue

↳ Pointeur déréférencé

- Si e est une expression de type pointeur, alors *e est une lvalue

```
1 char buffer[] = "Hello"; // buffer une lvalue non assignable
2 char* s = buffer;
3 *s = 'a'; // *s est une lvalue
4 *(s + 1) = 'b'; // *(s + 1) est une lvalue
```

↳ Le résultat de l'appel d'une fonction dont le type de retour est une référence lvalue est une lvalue

```
1 std::vector<int> v = {1, 2, 3};
2 // int& std::vector<int>::operator[](int);
3 int i = v[0]; // v[0] is lvalue
```

↳ Un littéral de chaîne est une lvalue. Exemple : "Hello World"

↳ Les références rvalue nommées sont des lvalues

```
1 int&& i = 1 + 3;
2 int j = i; // i est une lvalue
```

↳ Les références rvalue vers des fonctions (nommées et non-nommées) sont des lvalues.



Notion de value-ness

Expressions prvalue

↳ Un appel de fonction dont le type de retour n'est pas un type de référence est une prvalue

```
1 int func();
2 int i = func(); // func() est une prvalue
```

↳ Tous les littéraux sauf les littéraux de chaîne sont des prvalues

```
1 double pi = 3.1415; // 3.1415 est une prvalue
2 int i = 42; // 42 est une prvalue
3 i = 2 * i + 1; // 2 et 1 sont des prvalues
4 char c = 'A'; // 'A' est une prvalue
```

↳ Le résultat produit par certains opérateurs intégrés (par exemple, +, -, *, /) est une prvalue

```
1 int i, j;
2 i = 3 + 5; // 3 + 5 est une prvalue
3 j = i * i; // i * i est une prvalue
```

↳ Le mot-clé this est une expression prvalue.



Notion de value-ness

Expressions xvalue

↳ **xvalue** (expiring value) est une expression qui :

- ☛ Désigne un objet ou un bit-field dont les ressources peuvent être réutilisées (généralement parce qu'il est près de la fin de sa durée de vie);
- ☛ Possède une identité (comme une glvalue), mais est traitée comme une rvalue pour permettre le déplacement.

```
1 std::string s {"Hello"};
2 std::string t {std::move(s)}; // std::move(s) est une xvalue
```



↳ Une expression **xvalue** est une glvalue qui peut être déplacée, souvent issue d'une conversion explicite vers une **référence rvalue**

Notion de value-ness

rvalue et lvalue

- ↳ **lvalue** (left value) : référence sur une variable (dans la pile) ou un espace alloué dynamiquement (dans le tas) doté d'une adresse
 - ☛ Tout ce qui pouvait traditionnellement être à gauche de l'opérateur d'affectation
- ↳ **rvalue** (right value - prvalue depuis C++11) : les valeurs (ie. contenus) que l'on peut mettre dans une lvalue
 - ☛ Ces valeurs peuvent très bien être retournées par une fonction
 - ☛ Une valeur n'a pas d'adresse
- ↳ Il est possible de transformer une lvalue en une rvalue à l'aide de la fonction `std::move`
 - ☛ `std::move` ne déplace rien, juste elle effectue une sorte de casting (conversion)

Notion de value-ness

Les références

- ↳ Les références sont des alias d'objets ou de fonctions déjà existants
- ↳ Comme il existe deux types de valeurs (lvalue et rvalue), alors il existe deux types de références
 - 1 les références lvalue, denotées par `&`, tel que `&x`, sont des références de lvalues.
 - 2 les références rvalue, denotées par `&&`, tel que `&&x`, sont des références de rvalues

Inférence de type

Inférence de type avec le mot clé `auto`

- ↳ Le mot clé `auto` permet de déduire automatiquement le type d'un objet lors de sa définition à partir du type de l'objet utilisé pour l'initialisation

```
1 auto f = fonctionRenvoyantUnType(toto);
2 auto g=1; //int g=1;
```

- ☛ Utile pour éviter les répétitions

```
1 LaClasse * p=new LaClasse;
2 auto c2=new LaClasse;
```

- ☛ Utile pour éviter d'avoir à construire un nom compliqué lié à l'usage des templates

```
1 vector<vector<int> > v;
2 // C++98 (sans auto)
3 for (vector<vector<int> >::iterator
4 i=v.begin(); i!=v.end();++i)
5 // C++11
6 for (auto i=v.begin(); i!=v.end();++i)
```

Inférence de type

Avertissement : récupérer la value-ness

- Le mot clé `auto` ne permet pas de récupérer la value-ness de ce qui est à droite de l'affectation !

```
1 int f1();
2 int& f2();
3 const int& f3();
4
5 auto i1 {f1()}; // i1 de type int
6 auto a2 {f2()}; // i2 de type int
7 auto a3 {f3()}; // i3 de type int
```

- La value-ness est déterminée de manière manuelle

```
1 int & f1();
2 const int f2();
3 int f3();
4 const int & f4();
5
6 auto & i1 {f1()}; // int &
7 const auto & i2 {f1()}; //const int &
8 auto && i3 {f2()}; // int &&
9 auto && i4 {f3()}; // int &&
10 auto && i5 {f1()}; // int &
11 auto && i6 {f4()}; // const int &
```



Inférence de type

Inférence de type avec le mot clé `decltype`

- Introduit en C++11
- Le mot clé `decltype` permet de déduire le type y compris la value-ness d'une expression sans l'évaluer
- Syntaxe

```
decltype(expression)
```

```
1 int i;
2 decltype(i) j {5}; // j est donc de type int
3 ///////////////////////////////////////////////////
4 int f1();
5 int& f();
6 const int& f3();
7
8 decltype (f1()) i1 {f1()}; //type int
9 decltype (f2()) i2 {f2()}; //type int&
10 decltype (f3()) i3 {f3()}; //type const int&
```



Merci pour votre attention



Questions ?

