Bienvenue au cours “C,C#,.NET Intermediaire”

Nom de l’enseignant : Fabrice Meynckens

Promo : 2025-2026

Classe : B3CDEV

Objectifs de la session

- rappels sur la gestion de mémoire en C et C++
  - pointeurs
  - new - delete
  - pile et tas
  - pointeurs intelligents (smart pointers)

- Présentation de C# .Net
  - Historique, versions
  - Différences avec C++
    - Typage, gestion mémoire, organisation des fichiers

- Installation de l'environnement de développement
  - Premiers programmes simples

C++ : variables

Les types de base en C++

Le type d'une variable définit l'ensemble des valeurs qu'elle peut prendre et les opérations qui peuvent être effectuées sur elle. Les types de base les plus courants sont :

- **int:** Pour les nombres entiers (ex: 42, -10)
- **float:** Pour les nombres à virgule flottante simple précision (ex: 3.14)
- **double:** Pour les nombres à virgule flottante double précision (plus précise que float)
- **char:** Pour les caractères (ex: 'a', 'Z')
- **bool:** Pour les valeurs booléennes (true ou false)

Les suffixes

Les suffixes sont utilisés pour préciser le type d'une constante numérique littérale. Les plus courants sont :

- **f ou F:** Pour indiquer qu'un nombre est de type float (ex: 3.14f)
- **u ou U:** Pour indiquer qu'un nombre entier est de type unsigned (non signé, donc toujours positif)
- **l ou L:** Pour indiquer qu'un nombre entier est de type long (long entier)
- **ll ou LL:** Pour indiquer qu'un nombre entier est de type long long (très long entier)

C++ : portée des variables

La portée (scope) d'une variable en C++ détermine la partie du code où cette variable est accessible et valide.

Il existe trois types de portées principales : la **portée locale**, la **portée de bloc** et la **portée globale**.

Portée Locale (Fonction)

La portée locale s'applique aux variables déclarées à l'intérieur d'une fonction. Ces variables ne sont accessibles que dans cette fonction. Quand la fonction se termine, la mémoire de ces variables est libérée, et elles sont détruites.

Exemple :
C++
```
void maFonction() {
    int x = 10; // x a une portée locale
} // x n'existe plus après cette accolade
```

- **Utilité :** C'est le type de portée le plus courant pour les variables temporaires ou spécifiques à une tâche.

C++ : portée des variables

Portée de Bloc

La portée de bloc est plus restrictive que la portée locale. Une variable déclarée à l'intérieur d'un bloc (délimité par des accolades `{}`) est accessible uniquement dans ce bloc. C'est souvent le cas pour les boucles `for`, les structures conditionnelles `if`, etc.

Exemple :
C++
```
if (true) {
    int y = 20; // y a une portée de bloc
} // y n'existe plus ici
```

- **Utilité :** Permet de limiter l'existence d'une variable au strict nécessaire, ce qui évite les conflits de noms et améliore l'organisation du code.

C++ : portée des variables

Portée Globale

La portée globale s'applique aux variables déclarées en dehors de toute fonction ou classe. Ces variables sont accessibles depuis n'importe quelle partie du fichier source où elles sont déclarées. Leur durée de vie est celle du programme entier.

Exemple :
C++
```
int z = 30; // z a une portée globale


void autreFonction() {
    // z est accessible ici
}
```

- **Utilité :** Pour les données qui doivent être partagées par de nombreuses fonctions.
- **Mise en garde :** L'utilisation excessive de variables globales est généralement déconseillée car elle rend le code plus difficile à suivre et à déboguer. Il est souvent préférable de passer les valeurs en paramètres ou d'utiliser des classes pour gérer l'état.

C++ : déclarations des variables avec “auto”

Le mot-clé **`auto`** en C++ permet au compilateur de déduire automatiquement le type d'une variable lors de sa déclaration. Plutôt que de spécifier explicitement le type (comme `int`, `string` ou `MonObjet`), vous laissez le compilateur le déterminer en fonction de la valeur que vous assignez à la variable.

Fonctionnement et Utilité 🤔

L'utilisation de `auto` simplifie l'écriture du code, le rend plus concis et moins sujet aux erreurs de frappe, en particulier avec les types longs et complexes. C'est particulièrement utile dans les cas suivants :

- **Itérateurs** : Les types d'itérateurs peuvent être très longs et verbeux.
  - Au lieu de : `std::vector<std::pair<std::string, int>>::const_iterator it = maCarte.begin();`
  - Utilisez : `auto it = maCarte.begin();`
- **Expressions de lambda** : Les types des expressions lambda sont des types anonymes que vous ne pouvez pas nommer vous-même. `auto` est donc indispensable.
  - Exemple : `auto maFonction = [](int x, int y) { return x + y; };`
- **Types complexes et génériques** : Lorsque vous travaillez avec des templates ou des types imbriqués.

Il est important de noter que `auto` ne change pas le type de la variable. Le type est déterminé à la compilation et reste fixe. Il ne s'agit pas d'un typage dynamique.

C++ : “auto” bonnes pratiques

Restrictions et Bonnes Pratiques

- **Initialisation obligatoire** : Vous devez toujours initialiser une variable déclarée avec `auto` car le compilateur a besoin d'une valeur pour déterminer le type.
  - `auto x;` (ne fonctionne pas)
  - `auto x = 10;` (fonctionne)
- **Lisibilité** : Bien que `auto` soit pratique, il ne faut pas en abuser au détriment de la lisibilité. Pour les types simples comme `int` ou `double`, il est souvent plus clair de les spécifier explicitement.
- **Typage par déduction** : Le comportement de la déduction de type peut varier légèrement en fonction du contexte (référence, const, pointeur). Il est recommandé de bien comprendre comment cela fonctionne pour éviter des surprises.

Disponibilité 🕰️

Le mot-clé `auto` a été introduit et standardisé avec **C++11**. C'était l'une des fonctionnalités majeures de cette version, marquant un tournant vers un style de programmation plus moderne et expressif. Depuis C++11, `auto` est un élément essentiel et largement utilisé dans le développement C++.

C++ : Pointeurs

Les pointeurs en C++ : un outil puissant pour manipuler la mémoire

Qu'est-ce qu'un pointeur ?

Un pointeur en C++ est une variable spéciale qui **stocke l'adresse mémoire** d'une autre variable. En d'autres termes, au lieu de contenir directement une valeur, un pointeur indique où trouver cette valeur en mémoire.

Pourquoi utiliser des pointeurs ?

- **Allocation dynamique de mémoire:** Créer des variables à l'exécution, indépendamment de la pile.
- **Passage de paramètres par référence:** Modifier une variable à partir d'une fonction.
- **Structures de données complexes:** Créer des structures de données dynamiques comme les listes chaînées, les arbres, etc.

Déclaration d'un pointeur

```
type *nom_du_pointeur;
```

`Exemple`
`type` : le type de la donnée pointée.
`*nom_du_pointeur` : le nom du pointeur.

```
int *ptr; // Pointeur vers un entier
```

C++ : new et delete

L'opérateur `new` est utilisé pour **allouer dynamiquement de la mémoire** à l'exécution. Il retourne l'adresse de la mémoire allouée, qui est ensuite stockée dans un pointeur.

```
int *nombre = new int; // Alloue de la mémoire pour un entier
*nombre = 42; // Affecte la valeur 42 à l'entier pointé
```

L'opérateur `delete` est utilisé pour **libérer la mémoire** allouée dynamiquement avec `new`.

```
delete nombre; // Libère la mémoire pointée par nombre
```

```
#include <iostream>

using namespace std;

int main() {
    int *age = new int;
    cout << "Entrez votre âge : ";
    cin >> *age;
    cout << "Vous avez " << *age << " ans." << endl;

    delete age;

    return 0;
}
```

C++ : opérateur &

Qu'est-ce que l'opérateur & ?

En C++, l'opérateur & est utilisé pour obtenir l'**adresse mémoire** d'une variable. Il est souvent appelé **opérateur d'adressage**.

À quoi sert-il ?

1. **Initialisation de pointeurs:**
    - Lorsque vous déclarez un pointeur, vous devez lui affecter l'adresse d'une variable. C'est là que l'opérateur & intervient :

```
int x = 10;
int *ptr = &x; // ptr pointe maintenant vers l'adresse de x
```

C++ : la pile et le tas

En C++, la **pile** (stack) et le **tas** (heap) sont deux régions de mémoire distinctes utilisées pour stocker des données, et leur gestion est fondamentale pour comprendre le fonctionnement du langage.

La Pile (Stack) 🧠

La pile est une zone de mémoire à accès rapide, gérée de manière automatique par le compilateur. Elle fonctionne selon le principe **LIFO (Last-In, First-Out)** : la dernière chose qui entre est la première qui sort.

- **Utilité :** La pile est utilisée pour stocker les **variables locales** et les **paramètres de fonction**. Lorsqu'une fonction est appelée, un "cadre de pile" (stack frame) est créé pour contenir ses variables locales. Lorsque la fonction se termine, ce cadre est automatiquement détruit et la mémoire est libérée.
- **Moyens de l'utiliser :** C'est la méthode par défaut. Il suffit de déclarer une variable de manière standard, sans utiliser `new`.

Exemple :
C++
```
void maFonction() {
    int a = 10; // 'a' est sur la pile
    std::string s = "bonjour"; // 's' est sur la pile
} // Lorsque la fonction se termine, 'a' et 's' sont automatiquement détruits.
```

- **Avantages :** Allocation et désallocation extrêmement rapides, pas de fuites de mémoire.
- **Inconvénients :** Espace limité, ne convient pas pour les objets dont la durée de vie doit dépasser celle de la fonction qui les a créés.

Taille de la pile

La taille de la pile (stack) n'est pas une valeur unique et universelle. Elle dépend de plusieurs facteurs : le système d'exploitation, l'architecture du processeur, le compilateur et ses paramètres, et même les paramètres de configuration du projet.

Taille par défaut et limites

- **Sur les systèmes Windows**, la taille par défaut de la pile pour un programme compilé avec Visual C++ est généralement de **1 Mo**.
- **Sur les systèmes Linux**, elle est souvent plus grande, typiquement de **8 Mo**.
- **Sur macOS**, la taille par défaut est également d'environ **8 Mo**.

Ces valeurs sont des tailles par défaut pour les **threads principaux** de l'application. Pour les threads secondaires, la taille peut être différente.

Comment la taille est-elle gérée ?

La taille de la pile est une valeur fixe qui est déterminée au début de l'exécution du programme. Elle n'augmente pas de manière dynamique. Si le programme essaie d'utiliser plus de mémoire que la pile n'en a, cela provoque un **débordement de pile** (stack overflow), ce qui fait planter le programme.

Comment modifier la taille de la pile ?

Dans la plupart des environnements de développement, on peut modifier la taille de la pile.

- **Avec Visual Studio (Windows)** : On peut ajuster la taille de la pile dans les propriétés de la configuration du projet. Aller dans **Propriétés du projet > Éditeur de liens > Système > Taille de la pile**.
- **Avec GCC ou Clang (Linux/macOS)** : On peut utiliser le flag de compilation `-Wl,--stack=<taille>` pour spécifier une taille de pile différente.

Il est important de noter que changer la taille de la pile devrait être fait avec prudence. Augmenter la taille consomme plus de mémoire pour l'application, ce qui peut ne pas être souhaitable. Dans la plupart des cas, si on a un problème de débordement de pile, il est préférable de repenser l'architecture ducode pour stocker de grandes structures de données sur le **tas** (heap) au lieu de la pile.

C++ : le tas

Le Tas (Heap) 🗂️

Le tas est une grande zone de mémoire dynamique, gérée manuellement par le programmeur.

- **Utilité :** Le tas est utilisé pour stocker des objets dont la taille n'est pas connue à l'avance, ou dont la durée de vie doit dépasser la portée de la fonction qui les a créés. Les données restent allouées jusqu'à ce que le programmeur les libère explicitement.
- **Moyens de l'utiliser :** Pour allouer de la mémoire sur le tas, on utilise l'opérateur `new`. Cet opérateur renvoie un **pointeur** vers l'emplacement de la mémoire allouée. Pour libérer cette mémoire, il faut impérativement utiliser l'opérateur `delete`. Ne pas le faire conduit à une **fuite de mémoire** (memory leak).

Exemple :
C++
```
void maFonction() {
    int* ptr = new int; // 'ptr' est sur la pile, mais l'entier est sur le tas.
    delete ptr; // Il faut libérer manuellement la mémoire allouée.
}
```

- **Avantages :** Espace très grand, durée de vie des objets flexible.
- **Inconvénients :** Plus lent que la pile, et nécessite une gestion manuelle de la mémoire, source potentielle de fuites de mémoire et d'erreurs.

C++ : Smart pointers

Les pointeurs intelligents en C++ : une gestion de la mémoire plus sûre

Les pointeurs intelligents sont des objets qui encapsulent un pointeur brut et offrent une gestion de la mémoire plus sûre et plus automatique que les pointeurs classiques. Ils sont conçus pour prévenir les fuites de mémoire et les erreurs liées à la désallocation prématurée ou multiple.

Pourquoi utiliser des pointeurs intelligents ?

- **Sécurité:** Ils gèrent automatiquement la libération de la mémoire, réduisant ainsi les risques de fuites.
- **Facilité d'utilisation:** Ils simplifient la gestion de la durée de vie des objets alloués dynamiquement.
- **Sémantique claire:** Ils offrent des interfaces intuitives pour les opérations courantes sur les pointeurs.

c++ : std::unique_ptr

std::unique_ptr:

- **Propriété exclusive:** Un seul `unique_ptr` peut pointer vers un objet à un moment donné.
- **Destruction automatique:** Lorsque le `unique_ptr` sort de portée, l'objet pointé est automatiquement détruit.
- **Transfert de propriété:** Il est possible de transférer la propriété d'un objet d'un `unique_ptr` à un autre, mais pas de faire des copies.
- **Utilisation typique:** Pour gérer des ressources uniques qui ne doivent pas être partagées.

std::unique_ptr: Pour la plupart des cas où vous avez besoin d'un propriétaire unique pour un objet.

```cpp
class MyClass {
public:
    MyClass() {
        std::cout << "Constructeur de MyClass appelé" << std::endl;
    }
    ~MyClass() {
        std::cout << "Destructeur de MyClass appelé" << std::endl;
    }

    void doSomething() {
        std::cout << "Je fais quelque chose" << std::endl;
    }
};

int main() {
    // Création d'un unique_ptr pointant vers un nouvel objet MyClass
    std::unique_ptr<MyClass> myObject = std::make_unique<MyClass>();

    // Utilisation de l'objet
    myObject->doSomething();

    // Le destructeur de MyClass sera appelé automatiquement
    // lorsque myObject sortira de portée à la fin du bloc
}
```

c++ : std::make_unique

std::unique_ptr:

- **Propriété exclusive:** Un seul `unique_ptr` peut pointer vers un objet à un moment donné.
- **Destruction automatique:** Lorsque le `unique_ptr` sort de portée, l'objet pointé est automatiquement détruit.
- **Transfert de propriété:** Il est possible de transférer la propriété d'un objet d'un `unique_ptr` à un autre, mais pas de faire des copies.
- **Utilisation typique:** Pour gérer des ressources uniques qui ne doivent pas être partagées.

std::unique_ptr: Pour la plupart des cas où vous avez besoin d'un propriétaire unique pour un objet.

```cpp
class MyClass {
public:
    MyClass() {
        std::cout << "Constructeur de MyClass appelé" << std::endl;
    }
    ~MyClass() {
        std::cout << "Destructeur de MyClass appelé" << std::endl;
    }

    void doSomething() {
        std::cout << "Je fais quelque chose" << std::endl;
    }
};

int main() {
    // Création d'un unique_ptr pointant vers un nouvel objet MyClass
    std::unique_ptr<MyClass> myObject = std::make_unique<MyClass>();

    // Utilisation de l'objet
    myObject->doSomething();

    // Le destructeur de MyClass sera appelé automatiquement
    // lorsque myObject sortira de portée à la fin du bloc
}
```

c++ : fonctions de std::unique_ptr

Ces fonctions permettent de manipuler et de contrôler la durée de vie de l'objet pointé par le `unique_ptr`.

`get()`

- **Fonctionnalité:** Retourne un pointeur brut vers l'objet détenu.
- **Utilisation:** Utile lorsqu'une fonction ou une bibliothèque externe nécessite un pointeur brut. Cependant, il faut faire attention à ne pas le supprimer manuellement, car le `unique_ptr` se chargera de la désallocation.

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
int* rawPtr = ptr.get(); // rawPtr pointe maintenant vers l'entier 42
```

`release()`

- **Fonctionnalité:** Libère la propriété de l'objet pointé et retourne un pointeur brut vers celui-ci. Le `unique_ptr` devient alors vide.
- **Utilisation:** Permet de transférer la propriété de l'objet à un autre pointeur (brut ou intelligent).

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
int* rawPtr = ptr.release(); // rawPtr possède maintenant l'objet
// ptr est maintenant vide
```

c++ : fonctions de std::unique_ptr

`reset()`

- **Fonctionnalité:** Libère l'objet actuel (si existant) et prend en option un nouveau pointeur brut.
- **Utilisation:** Permet de remplacer l'objet pointé par un autre, ou de vider le `unique_ptr`.

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
ptr.reset(); // L'objet 42 est détruit
ptr.reset(new int(10)); // ptr pointe maintenant vers un nouvel entier
```

`swap()`

- **Fonctionnalité:** Échange le contenu de deux `unique_ptr`.
- **Utilisation:** Utile pour déplacer la propriété d'un objet d'un `unique_ptr` à un autre.

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(42);
std::unique_ptr<int> ptr2 = std::make_unique<int>(10);
ptr1.swap(ptr2); // ptr1 pointe maintenant vers 10 et ptr2 vers 42
```

c++ : std::shared_ptr

Qu'est-ce que std::shared_ptr ?

`std::shared_ptr` est un pointeur intelligent de la bibliothèque standard C++ conçu pour gérer des ressources qui peuvent être partagées par plusieurs parties de votre code. Il implémente un mécanisme de comptage de références : tant qu'il existe au moins un `std::shared_ptr` pointant vers un objet, cet objet n'est pas détruit. Lorsque le dernier `std::shared_ptr` sort de portée, l'objet est automatiquement supprimé.

Pourquoi utiliser std::shared_ptr ?

- **Partage de ressources:** Plusieurs parties de votre code peuvent accéder à un même objet sans risque de le supprimer prématurément.
- **Sécurité:** Le comptage de références garantit que la mémoire est libérée de manière sûre lorsqu'elle n'est plus utilisée.
- **Facilité d'utilisation:** `std::shared_ptr` simplifie la gestion de la durée de vie des objets partagés.

```cpp
#include <memory>
#include <iostream>

class MyClass {
public:
    MyClass() { std::cout << "Constructeur\n"; }
    ~MyClass() { std::cout << "Destructeur\n"; }
};

int main() {
    // Création de deux shared_ptr pointant vers le même objet
    std::shared_ptr<MyClass> ptr1 = std::make_shared<MyClass>();
    std::shared_ptr<MyClass> ptr2 = ptr1;

    // L'objet est toujours en mémoire tant que ptr1 ou ptr2 existe
    std::cout << "ptr1 et ptr2 pointent vers le même objet\n";

    // Quand ptr1 et ptr2 sortent de portée, le destructeur de MyClass es
}
```

c++ : std::shared_ptr

Points à retenir

- **Comptage de références:** C'est le cœur du mécanisme de `std::shared_ptr`.
- **Cycles de références:** Il faut faire attention aux cycles de références qui peuvent empêcher la destruction d'objets.
- **`std::make_shared`:** Préférez `std::make_shared` à `new` pour créer des `shared_ptr` car il assure une allocation et une initialisation atomiques, réduisant ainsi les risques de fuites de mémoire.
- **`std::weak_ptr`:** Pour observer un objet sans en prendre la propriété et éviter les cycles de références.
- **Custom deleter:** Vous pouvez spécifier un destructeur personnalisé pour l'objet.

Quand utiliser `std::shared_ptr` ?

- **Ressources partagées:** Lorsque plusieurs parties de votre code doivent accéder à un même objet.
- **Gestion de la durée de vie complexe:** Pour des scénarios où la durée de vie d'un objet dépend de plusieurs facteurs.

c++ : std::weak_ptr

Qu'est-ce que std::weak_ptr ?

`std::weak_ptr` est un autre type de pointeur intelligent en C++ qui offre une référence non propriétaire à un objet géré par un `std::shared_ptr`. Contrairement à `std::shared_ptr` qui incrémente le compteur de référence, `std::weak_ptr` n'a pas d'impact sur la durée de vie de l'objet. Il est principalement utilisé pour :

- **Éviter les cycles de référence:** Lorsqu'il y a une relation circulaire entre plusieurs `shared_ptr`, `weak_ptr` peut aider à rompre ce cycle.
- **Observer un objet sans en prendre la propriété:** Vous pouvez observer l'état d'un objet géré par un `shared_ptr` sans prolonger sa durée de vie.

Pourquoi utiliser std::weak_ptr ?

- **Éviter les fuites de mémoire:** En rompant les cycles de référence, `weak_ptr` aide à prévenir les fuites de mémoire.
- **Flexibilité:** Il offre plus de contrôle sur la durée de vie des objets.

```cpp
#include <memory>
#include <iostream>

class Node {
public:
    Node(int data) : data(data) {}
    std::shared_ptr<Node> parent;
    std::weak_ptr<Node> child;
    int data;
};

int main() {
    // Création d'un nœud
    auto root = std::make_shared<Node>(0);
    root->child = root; // Cycle de référence si on utilise un shared_ptr

    // Utilisation de weak_ptr pour éviter le cycle
    std::weak_ptr<Node> weak_root = root;

    // Vérifier si l'objet est toujours valide
    if (auto locked = weak_root.lock()) {
        std::cout << "Le nœud existe toujours : " << locked->data << std:
    } else {
        std::cout << "Le nœud a été supprimé." << std::endl;
    }
}
```

c++ : std::weak_ptr

Qu'est-ce que std::weak_ptr ?

`std::weak_ptr` est un autre type de pointeur intelligent en C++ qui offre une référence non propriétaire à un objet géré par un `std::shared_ptr`. Contrairement à `std::shared_ptr` qui incrémente le compteur de référence, `std::weak_ptr` n'a pas d'impact sur la durée de vie de l'objet. Il est principalement utilisé pour :

- **Éviter les cycles de référence:** Lorsqu'il y a une relation circulaire entre plusieurs `shared_ptr`, `weak_ptr` peut aider à rompre ce cycle.
- **Observer un objet sans en prendre la propriété:** Vous pouvez observer l'état d'un objet géré par un `shared_ptr` sans prolonger sa durée de vie.

Pourquoi utiliser std::weak_ptr ?

- **Éviter les fuites de mémoire:** En rompant les cycles de référence, `weak_ptr` aide à prévenir les fuites de mémoire.
- **Flexibilité:** Il offre plus de contrôle sur la durée de vie des objets.

```cpp
#include <memory>
#include <iostream>

class Node {
public:
    Node(int data) : data(data) {}
    std::shared_ptr<Node> parent;
    std::weak_ptr<Node> child;
    int data;
};

int main() {
    // Création d'un nœud
    auto root = std::make_shared<Node>(0);
    root->child = root; // Cycle de référence si on utilise un shared_ptr

    // Utilisation de weak_ptr pour éviter le cycle
    std::weak_ptr<Node> weak_root = root;

    // Vérifier si l'objet est toujours valide
    if (auto locked = weak_root.lock()) {
        std::cout << "Le nœud existe toujours : " << locked->data << std:
    } else {
        std::cout << "Le nœud a été supprimé." << std::endl;
    }
}
```

C# : Introduction

Description et historique de C#

C# (prononcez "C sharp") est un langage de programmation orienté objet, développé par Microsoft en 2000. Il a été conçu pour faire partie de la plateforme **.NET**, un écosystème de développement unifié. L'objectif de C# était d'être un langage moderne, simple, de type sécurisé, et optimisé pour le développement d'applications sur la plateforme Windows. Son design est fortement influencé par C++ et Java, tout en intégrant des fonctionnalités qui simplifient la programmation.

Caractéristiques principales

- **Orienté objet** : C# est entièrement orienté objet. Tout est un objet ou peut être traité comme tel, ce qui inclut l'héritage, le polymorphisme, l'encapsulation et les interfaces.
- **Gestion de la mémoire automatique** : Le langage utilise un **"garbage collector"** pour la gestion de la mémoire. Le développeur n'a pas besoin de libérer manuellement la mémoire, ce qui réduit considérablement le risque de fuites de mémoire et d'erreurs de gestion de la mémoire.
- **Typage fort** : C# est un langage fortement typé. Le type d'une variable doit être déclaré, et le compilateur vérifie la compatibilité des types à la compilation, ce qui aide à prévenir les erreurs courantes.
- **Interopérabilité** : C# peut interagir avec d'autres langages et technologies. Il peut notamment s'interfacer avec le code natif via le service **P/Invoke** (Platform Invoke) et s'intégrer facilement avec des bibliothèques écrites en C++ ou en C.
- **LinQ** : Le "Language-Integrated Query" est une fonctionnalité de C# qui permet d'écrire des requêtes de données directement dans le langage, de manière uniforme, pour différentes sources (collections, bases de données, XML).
- **Support multiplateforme** : Grâce à **.NET Core** (et son évolution, **.NET**), C# n'est plus limité à Windows. Les applications C# peuvent désormais être exécutées sur Linux, macOS et d'autres plateformes.

C# : Introduction

La plateforme .NET

La plateforme **.NET** est l'environnement d'exécution de C#. Elle inclut :

- **CLR (Common Language Runtime)** : L'environnement d'exécution qui gère l'exécution du code, le garbage collection, la sécurité et la compilation **JIT** (Just-In-Time) du code en langage machine.
- **BCL (Base Class Library)** : Une immense bibliothèque de classes qui fournit des fonctionnalités pour tout, de la manipulation de chaînes de caractères à la gestion de fichiers, en passant par le réseau et la sécurité.

Dernière version de C#

La dernière version majeure du langage est **C# 12**, qui a été publiée en novembre 2023. Elle a été livrée avec **.NET 8**.

Les nouveautés de C# 12 se concentrent sur l'amélioration de la productivité et de la lisibilité, notamment avec :

- **Collections expression** : Une syntaxe simplifiée pour initialiser des tableaux et des collections.
- **Alias de type** : La possibilité d'utiliser la directive `using` pour créer des alias pour n'importe quel type, y compris des types génériques et des types d'alias de tuple.
- **Paramètres optionnels** pour les expressions lambda.
- **Améliorations des intercepteurs** pour l'optimisation des performances dans certains scénarios.

C# : installation

1. Installer le .NET SDK

Le **.NET SDK** est un kit de développement logiciel qui inclut le compilateur C#, la bibliothèque de classes .NET et d'autres outils nécessaires.

1. Allez sur le site officiel de Microsoft pour télécharger le SDK : **dotnet.microsoft.com/download**.
2. Téléchargez la dernière version du **.NET SDK** pour Windows.
3. Exécutez l'installeur et suivez les instructions.

Une fois l'installation terminée, ouvrez un terminal (comme PowerShell ou l'invite de commandes) et tapez : `dotnet --version`. Cela devrait afficher la version du SDK installée, confirmant que l'installation a réussi.

2. Installer Visual Studio Code

Si vous ne l'avez pas déjà, VS Code est un éditeur de code léger mais puissant.

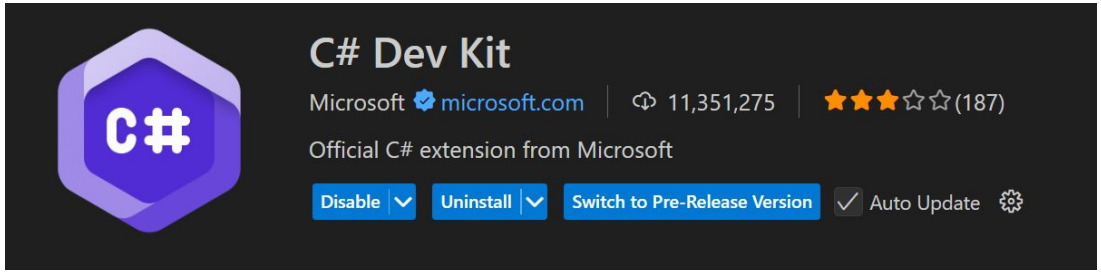
1. Téléchargez VS Code sur le site officiel : **code.visualstudio.com**.
2. Exécutez le fichier d'installation et suivez les instructions.

C# : installation

3. Installer l'extension C# pour VS Code

Cette extension est cruciale car elle ajoute des fonctionnalités essentielles pour le développement en C#, comme l'autocomplétion (IntelliSense), le débogage et l'exploration de code.

1. Ouvrez VS Code.
2. Cliquez sur l'icône des **Extensions** dans la barre d'activité à gauche (elle ressemble à un carré).
3. Dans la barre de recherche, tapez **"C# Dev Kit"**.
4. Cliquez sur **Installer** pour l'extension **C# Dev Kit** publiée par Microsoft. Cette extension installe aussi automatiquement l'extension C# et l'extension .NET Runtime Install Tool.



C# : installation

4. Créer votre premier projet C#

Maintenant que votre environnement est prêt, vous pouvez créer votre premier projet.

1. Ouvrez VS Code.
2. Ouvrez la palette de commandes en appuyant sur `Ctrl + Maj + P`.
3. Tapez **".NET: New Project"** et sélectionnez cette option.
4. VS Code vous demandera quel type de projet créer. Pour commencer, choisissez **"Console App"**.
5. VS Code vous demandera où enregistrer le projet. Choisissez un dossier et donnez un nom à votre projet.

5. Exécuter le code

Un fichier `Program.cs` contenant un simple "Hello, World!" devrait être généré automatiquement.

1. Le terminal de VS Code est le meilleur endroit pour exécuter votre code.
2. Pour lancer le programme, vous pouvez taper `dotnet run` dans le terminal de VS Code, depuis le répertoire de votre projet.
3. Vous devriez voir le message "Hello, World!" s'afficher dans la console.

Félicitations, vous êtes prêt à coder en C# !

Voir fichier joint au cours : program.cs

C# : packages

Le site : https://www.nuget.org/ propose le téléchargement de nombreux packages utilitaires

Pour installer un nouveau package, tapez simplement la commande d'installation comme spécifié sur le site

Ex : un lecteur de fichiers XML

C# : variables et types

1. Types de base et uniformité du système de types

En C#, tous les types, y compris les types primitifs (`int`, `bool`, `double`), dérivent finalement de la classe `System.Object`. Cela signifie qu'un `int` peut être traité comme un objet, un processus appelé "boxing" (emballage). En C++, les types primitifs sont des entités de bas niveau qui ne sont pas des objets.

- **C# :** Un système de types unifié, ce qui facilite la manipulation des types de base de manière uniforme (par exemple, les mettre dans des collections d'objets).
- **C++ :** Les types primitifs comme `int` ou `char` ne font pas partie d'une hiérarchie de classes.

2. Gestion de la mémoire et pointeurs

C'est l'une des plus grandes divergences. C# utilise la **gestion automatique de la mémoire** via un "garbage collector" (ramasse-miettes). Les développeurs n'ont généralement pas besoin de se soucier de l'allocation ou de la désallocation de la mémoire.

- **C# :** Les pointeurs sont limités à un mode de code non sécurisé (`unsafe`). L'utilisation de `new` pour créer une classe alloue l'objet sur le **tas** (heap), et le ramasse-miettes s'occupe de le libérer quand il n'est plus utilisé.
- **C++ :** Le développeur est responsable de la **gestion manuelle de la mémoire** en utilisant `new` et `delete`. Il est crucial de libérer la mémoire allouée dynamiquement pour éviter les fuites de mémoire. Les pointeurs (`*`) sont largement utilisés pour la manipulation directe de la mémoire.

C# : variables et types

3. Chaînes de caractères (`string`)

En C#, le type `string` est un type de référence **immuable**. Une fois qu'une chaîne de caractères est créée, son contenu ne peut pas être modifié. Toute opération qui semble la modifier (comme la concaténation) crée en fait une nouvelle chaîne en mémoire.

- **C# :** Les chaînes sont immuables (`System.String`). Cela les rend plus sûres mais peut avoir des implications sur la performance lors de manipulations intensives.
- **C++ :** Les chaînes (`std::string`) sont **mutables**. Vous pouvez les modifier directement, ce qui peut être plus efficace dans certains cas, mais aussi plus complexe à gérer.

4. Types valeur (`struct`) et types référence (`class`)

Les deux langages ont des types valeur et des types référence, mais avec des implications différentes.

- **C# :** Une **`struct`** est un type valeur. Les variables de type `struct` contiennent directement leurs données et sont stockées sur la **pile** (stack). Une **`class`** est un type référence ; ses variables contiennent une référence (une sorte de pointeur) vers un objet qui est sur le **tas** (heap).
- **C++ :** Les `struct` et les `class` sont très similaires et se distinguent principalement par la visibilité par défaut des membres (publique pour `struct`, privée pour `class`). La gestion de la mémoire, qu'elle soit sur la pile ou sur le tas, dépend de la manière dont l'objet est alloué (par exemple, `MyClass obj;` sur la pile vs `MyClass* obj = new MyClass();` sur le tas).

C# : surcharges

Au niveau des déclarations et des appels de méthodes de classes, les différences entre C++ et C# résident principalement dans la gestion de l'héritage, du polymorphisme et dans la syntaxe.

Surcharges (Overloading)

L'**overloading** de méthodes (fonctions en C++) existe dans les deux langages et fonctionne de manière similaire. Il s'agit de définir plusieurs méthodes avec le même nom dans la même classe, mais avec des signatures différentes (nombre ou type de paramètres). Le compilateur détermine quelle version appeler en fonction des arguments fournis.

- **C#** : La surcharge est un concept de base et est résolue à la compilation (polymorphisme statique).
- **C++** : C'est également un concept de base. Il est possible de surcharger des fonctions, des méthodes et même des opérateurs.

C# vs C++ : Redéfinitions (Overriding) et Polymorphisme

C'est là que les différences sont les plus marquées. La manière dont le polymorphisme est géré dans le contexte de l'héritage varie considérablement.

En C++ :

Pour qu'une méthode d'une classe dérivée **redéfinisse** (override) une méthode de sa classe de base, la méthode de la classe de base doit être explicitement déclarée comme `virtual`. Sans ce mot-clé, une méthode de la classe dérivée avec le même nom et la même signature "cache" simplement celle de la classe de base, et l'appel de la méthode dépend du type de la référence ou du pointeur utilisé.

- **Déclaration** : `virtual void myMethod();` dans la classe de base.
- **Redéfinition** : `void myMethod() override;` dans la classe dérivée (l'utilisation de `override` est une bonne pratique depuis C++11 pour éviter les erreurs).
- **Appel** : Le compilateur résout l'appel au moment de l'exécution (polymorphisme dynamique) si vous utilisez des pointeurs ou des références à la classe de base.

C# vs C++ : Redéfinitions (Overriding) et Polymorphisme

En C# :

La gestion du polymorphisme est plus explicite et moins sujette à erreur.

- **Déclaration** : Une méthode doit être déclarée comme `virtual` dans la classe de base pour pouvoir être redéfinie.
- **Redéfinition** : La classe dérivée doit utiliser le mot-clé `override` pour explicitement redéfinir la méthode virtuelle. Si le mot-clé est omis, la nouvelle méthode "cache" l'ancienne (comme en C++ sans `virtual`), mais le compilateur émet un avertissement pour informer le développeur du comportement.
- **Appel** : Le polymorphisme dynamique est le comportement par défaut des méthodes `virtual` et `override`, ce qui rend le code plus prévisible.

Un autre point clé est l'utilisation des mots-clés `new` en C# et `final`.

- Le mot-clé `new` en C# est utilisé pour "cacher" une méthode de la classe de base, sans la redéfinir. Le comportement est alors déterminé par le type de la variable au moment de la compilation.
- Le mot-clé `final` en C++ et C# permet d'empêcher une méthode `virtual` d'être redéfinie dans une classe dérivée.

C# vs C++ : paramètres de fonctions

Les principales différences entre C++ et C# concernant les paramètres de fonctions (méthodes) se situent dans la gestion du passage par valeur, par référence, et dans la présence de mots-clés spécifiques pour contrôler ce comportement.

1. Passage par valeur et par référence

C++

Par défaut, les paramètres sont passés **par valeur**. Cela signifie que le compilateur crée une copie de l'argument et la transmet à la fonction. Les modifications apportées au paramètre dans la fonction ne sont pas visibles à l'extérieur.

- **Passage par valeur :** `void maFonction(int a)`
- **Passage par référence :** Pour modifier l'argument original, vous devez utiliser une **référence** (`&`) ou un **pointeur** (`*`).
  - `void maFonction(int& a)` (référence)
  - `void maFonction(int* a)` (pointeur)

Le passage par référence en C++ est souvent utilisé pour des raisons d'efficacité, notamment pour les objets volumineux, afin d'éviter la copie.

C# vs C++ : paramètres de fonctions

C#

En C#, le passage des paramètres est plus nuancé en raison de la distinction entre les **types valeur** (`struct`, `int`, `bool`, etc.) et les **types référence** (`class`).

- **Types valeur :** Par défaut, ils sont passés **par valeur**. Une copie de l'argument est transmise.
- **Types référence :** Par défaut, ils sont passés **par valeur**, mais la "valeur" transmise est la **référence** (l'adresse mémoire) de l'objet. Cela signifie que la méthode peut modifier l'objet d'origine. C'est un comportement qui peut sembler être un passage par référence, mais il est important de noter que si vous réaffectez la référence dans la méthode, cela ne changera pas la référence d'origine.

Pour un véritable passage par référence, C# utilise des mots-clés spécifiques.

2. Mots-clés spécifiques de C#

C# offre des mots-clés pour clarifier et contrôler le passage des paramètres :

- **`ref`** : Ce mot-clé force le passage par **référence** d'un paramètre. Cela permet à la méthode de modifier l'argument original, qu'il soit un type valeur ou un type référence. L'argument doit être initialisé avant d'être passé à la fonction.
  - Exemple : `void maFonction(ref int a)`
- **`out`** : Similaire à `ref`, il force le passage par **référence**. La différence est que l'argument n'a pas besoin d'être initialisé avant l'appel de la méthode, car son seul but est de recevoir une valeur de la méthode.
  - Exemple : `void maFonction(out int a)`
- **`in`** : Introduit avec C# 7.2, il permet de passer un paramètre par **référence**, mais il est **en lecture seule** à l'intérieur de la méthode. Cela offre un gain de performance en évitant la copie de gros `struct` sans risquer de les modifier accidentellement.
  - Exemple : `void maFonction(in MonGrosStruct a)`

C# : organisation et séparation des fichiers sources

En C#, il n'y a pas de séparation de fichiers entre les déclarations (`.h` ou `.hpp`) et les implémentations (`.cpp`) comme en C++.

L'organisation des fichiers en C# est basée sur un modèle plus simple et plus direct. Chaque fichier `.cs` contient à la fois les déclarations de classes, d'interfaces, de structures et d'autres types, ainsi que leurs implémentations.

Comment ça marche en C# ?

- **Un seul fichier, tout-en-un** : Dans un fichier `.cs`, vous écrivez la définition de votre classe, y compris ses méthodes, propriétés et champs, directement à l'intérieur de la classe elle-même.

- **Les "usings" remplacent les "includes"** : Au lieu d'inclure des fichiers d'en-tête, C# utilise le mot-clé **`using`** pour importer des **espaces de noms** (namespaces). Un namespace est une collection de classes et de types liés. L'instruction `using` permet au compilateur de trouver les types que vous utilisez sans avoir à spécifier leur nom complet.

- **Rôle du compilateur** : Le compilateur C# se charge de parcourir tous les fichiers `.cs` d'un projet et de construire l'assemblage (assembly), qui est l'équivalent d'un fichier exécutable (`.exe`) ou d'une bibliothèque (`.dll`). Il n'a pas besoin de la séparation en deux étapes d'un fichier d'en-tête et d'un fichier d'implémentation.

- **Classes partielles** : Pour les grandes classes, C# offre le mot-clé **`partial`**. Il permet de diviser la définition d'une seule classe en plusieurs fichiers `.cs`. C'est utile pour organiser de très grands blocs de code ou lorsque du code est généré automatiquement (par exemple, dans les frameworks UI).