



ENHANCE TECH



+216 53249369



www.enhancetech.tn



info@enhancetech.tn



<https://www.linkedin.com/company/enhancetech-tn/>



Version 2026.4

C++ Moderne

Formations expertes pour entreprises



À PROPOS DE NOUS

EnhanceTech est une entreprise de services et de développement logiciel, qui propose également des sessions de formations spécialisées destinées aux entreprises souhaitant booster les compétences de leurs équipes en développement.

Fondée par des experts docteurs et ingénieurs en informatique, notre société s'appuie sur une expertise approfondie et une passion pour les technologies de pointe.

Nos formateurs, issus du monde académique et industriel, garantissent une approche complète et pragmatique répondant aux défis actuels du secteur IT.

Notre mission : accompagner la montée en compétence de vos collaborateurs grâce à des programmes ancrés dans la réalité industrielle et les standards modernes du développement.

NOUS CONTACTER

Téléphone : +216 53249369

Site web : www.enhancetech.tn

Mail : info@enhancetech.tn

LinkedIn : <https://www.linkedin.com/company/enhancetech-tn/>

NOS OBJECTIFS

Chez EnhanceTech, nos objectifs sont clairs et orientés vers l'excellence :

- ◆ Favoriser la montée en compétence rapide et efficace : Nous visons à équiper vos équipes avec les compétences avancées en développement mobile, embarqué et logiciel nécessaires pour innover et résoudre des problèmes complexes en programmation.



- ◆ Promouvoir l'innovation technologique : En nous concentrant sur les dernières avancées en développement, nous aidons les entreprises à adopter des pratiques modernes.
- ◆ Assurer une adaptabilité aux besoins du marché : Nos formations sont conçues pour aligner les compétences de vos employés avec les exigences évolutives des industries, en favorisant une culture d'apprentissage continu et de collaboration.
- ◆ Maximiser le retour sur investissement (ROI) : garantir un impact mesurable sur la productivité et la qualité logicielle.

NOS ATOUTS

EnhanceTech se distingue par plusieurs atouts qui font de nous le partenaire idéal pour vos formations en développement logiciel :

- ◆ Expertise reconnue : nos formateurs sont des praticiens expérimentés alliant rigueur académique et expérience terrain.
- ◆ Formations sur mesure et flexibles : adaptées à vos besoins, en présentiel, à distance ou en format hybride, du niveau débutant à expert.
- ◆ Approche pédagogique innovante : alternance de théorie, d'ateliers pratiques et d'études de cas réels.
- ◆ Partenariats : Nous collaborons avec des institutions académiques et industrielles de renom.
- ◆ Accompagnement post-formation : support continu, ressources en ligne et veille technologique.



THÉMATIQUES

1) C++ MODERNE & BEST PRACTICES

Pour des applications performantes et maintenables.

Nos formations dans cette thématique :

- ◆ Programmation C++ : les fondamentaux
- ◆ Programmation moderne C++ : de C++11 à C++17
- ◆ Les avancées du langage : du C++11 à C++20
- ◆ C++ Programmation Avancée Expert

Bénéfices pour votre entreprise :

- ◆ Productivité accrue avec les fonctionnalités modernes
- ◆ Performance optimisée grâce aux dernières avancées du langage
- ◆ Maintenabilité améliorée avec des codebases plus propres
- ◆ Compatibilité assurée avec les standards actuels et futurs

Public concerné : Tous vos développeurs C++, de juniors à seniors

2) CODAGE SÉCURISÉ & CYBERSÉCURITÉ

Pour des applications robustes face aux menaces modernes.

Nos formations dans cette thématique :

- ◆ Développeur Cybersécurité en C/C++
- ◆ C++ Programmation Avancée Expert (modules sécurité)

Bénéfices pour votre entreprise :

- ◆ **Sécurité** renforcée contre les vulnérabilités courantes
- ◆ **Conformité** aux standards de sécurité (CERT, CWE, OWASP)
- ◆ **Confiance** accrue dans vos produits logiciels
- ◆ **Réduction** des risques de failles de sécurité

Public concerné : Développeurs systèmes, responsables sécurité, architectes



3) APPLICATIONS GRAPHIQUES & INTERFACES UTILISATEUR

Pour des expériences utilisateur modernes et réactives.

Nos formations dans cette thématique :

- ◆ Conception d'interfaces graphiques avec Qt 6

Bénéfices pour votre entreprise :

- ◆ **Expérience** utilisateur optimale sur desktop et embarqué
- ◆ **Productivité** de développement multipliée
- ◆ **Portabilité** cross-plateforme native
- ◆ **Performance** graphique exceptionnelle

Public concerné : Développeurs d'applications, ingénieurs embarqués avec IHM

4) ROS & SYSTÈMES ROBOTIQUES

Pour maîtriser la robotique de nouvelle génération

Nos formations dans cette thématique :

- ◆ Créer des applications ROS2 en C++

Bénéfices pour votre entreprise :

- ◆ **Interopérabilité** avec l'écosystème robotique mondial
- ◆ **Productivité** grâce aux outils et bibliothèques ROS2
- ◆ **Innovation** accélérée dans vos projets robotiques
- ◆ **Maintenabilité** des systèmes complexes

Public concerné : Ingénieurs robotique, chercheurs, startups IoT/robotics

5) OBJETS CONNECTÉS & SYSTÈMES EMBARQUÉS

Pour dominer l'Internet des Objets et l'embarqué connecté

Nos formations dans cette thématique :

- ◆ Développer des applications IoT en C++



Bénéfices pour votre entreprise :

- ◆ **Connectivité** maîtrisée (LoRaWAN, Bluetooth, WiFi, MQTT)
- ◆ **Autonomie** optimisée pour l'embarqué
- ◆ **Cloud integration** seamless
- ◆ **Scalabilité** des solutions IoT

Public concerné : Ingénieurs embarqués, développeurs IoT, architectes systèmes

6) PERFORMANCE & SYSTÈMES CRITIQUES

Pour les applications où chaque cycle compte

Nos formations dans cette thématique :

- ◆ Programmer ses applications en Multicore en C++
- ◆ C++ Programmation Avancée Expert

Bénéfices pour votre entreprise :

- ◆ **Latence** réduite au minimum
- ◆ **Débit** maximalisé grâce au parallélisme
- ◆ **Efficacité** énergétique optimisée
- ◆ **Prédictibilité** temporelle garantie

Public concerné : Développeurs, systèmes temps-réel



POURQUOI CHOISIR ENHANCETECH POUR VOS FORMATIONS C++ ?

◆ Expertise terrain éprouvée

Nos formateurs sont des experts en activité intervenant sur des projets critiques dans divers secteurs industriels. Leur double casquette consultant-formateur garantit des formations **ancrées** dans les réalités techniques actuelles.

◆ Références sectorielles significatives

Nos programmes ont été déployés avec succès auprès de :

- ◆ Électronique & Systèmes Embarqués : Leaders en instrumentation et gestion d'énergie
- ◆ Télécommunications & IoT : Fabricants d'équipements connectés
- ◆ Recherche & Scientifique : Centres de recherche de rang mondial

◆ Pédagogie par la pratique

60% de travaux pratiques sur des cas réels issus de nos missions de consulting, permettant une application immédiate des concepts en environnement professionnel.

◆ Approche sur-mesure systématique

Chaque formation est adaptée à votre stack technique, vos processus de développement et vos enjeux métiers spécifiques.

◆ Accompagnement durable

Support post-formation inclus avec accès à nos ressources experts, forums dédiés et mises à jour techniques.



Programmation en C++

Les fondamentaux

Référence : CPP01			
5 jours (40h) 	Présentiel / Distanciel <i>Modalité</i>	Débutant à intermédiaire  Débutant  Intermédiaire	2 à 12 participants <i>Format</i>

Prérequis Expérience en programmation orientée objet dans un autre langage (Java, Python, C# ou équivalent). Aucune connaissance de C++ n'est nécessaire, mais une expérience en programmation est requise	Public concerné Développeurs souhaitant acquérir les bases solides du C++ moderne Développeurs Java/C#/Python migrant vers C++
---	---

OBJECTIFS DE LA FORMATION

- Maîtriser la syntaxe et les idiomes du C++ moderne (C++11 à C++17) dès le premier programme
- Mettre en œuvre les concepts de la Programmation Orientée Objet : encapsulation, héritage, polymorphisme
- Comprendre et gérer la mémoire : pile, tas, allocation manuelle puis smart pointers (RAII)
- Exploiter la bibliothèque standard STL : conteneurs, algorithmes, itérateurs
- Écrire du code robuste grâce aux exceptions, aux contrats et aux outils de diagnostic
- Appliquer les bonnes pratiques modernes

PROGRAMME DÉTAILLÉ

1 - Introduction au C++ moderne

- ☐ Historique et évolution
- ☐ Outils de développement
- ☐ Premier programme : structure de `main()`
- ☐ Variables et types fondamentaux : `int`, `double`, `bool`, `char`, `std::string`
- ☐ Initialisation moderne : `{ }` uniforme
- ☐ Constantes : `const`, `constexpr` (introduction)
- ☐ Pointeurs, références et tableaux

Maîtrisez les bases du C++ tout en intégrant les innovations du C++ moderne dès le départ, pour des applications robustes et maintenables qui respectent les standards industriels actuels.



- ☐ Opérateurs et conversions
- ☐ Les espaces de nommage
- ☐ Le typage automatique avec le mot-clé **auto**
- ☐ Les entrées/sorties avec C++

Travaux pratiques

- Prise en main de l'environnement de développement et programmation d'un programme simple.
- Manipulation de types, conversions et espaces de noms

2 - Structures de contrôle, fonctions et références

- ☐ Structures conditionnelles : **if**, **switch**
- ☐ Boucles : **for**, **while**, **do while**
- ☐ Boucle moderne range-based **for** [C++11]
- ☐ Définition et appel de fonctions
- ☐ Passage de paramètres : par valeur et par référence
- ☐ Valeurs par défaut
- ☐ Surcharge de fonctions

Travaux pratiques

- Implémentation d'un ensemble de fonctions de calcul et manipulation de données.
- Implémentation de fonctions de calcul (moyenne, factorielle, Fibonacci)
- Manipulation de références vs copies
- Utilisation de range-based for sur tableaux

3 - Programmation Orientée Objet - Fondamentaux

- ☐ Principes de la programmation orientée objet
- ☐ Définition d'une classe : attributs et méthodes
- ☐ Contrôle d'accès : **private**, **protected**, **public**
- ☐ Constructeurs et liste d'initialisation
- ☐ Délégation de constructeurs
- ☐ Destructeur et cycle de vie des objets
- ☐ Méthodes **const**
- ☐ Champs et méthodes statiques
- ☐ Méthodes et classes **friend** : quand et pourquoi
- ☐ Bonnes pratiques : **explicit**, responsabilité unique, invariants de classe

Travaux pratiques

- Implémentation des classes avec encapsulation stricte.
- Implémentation de constructeurs avec délégation
- Méthodes const et statiques : cas d'usage concrets

4 - Allocation dynamique et gestion mémoire classique

- ☐ Différence pile et tas : durée de vie, performance, fragmentation



- ☐ Allocation dynamique : **new, delete, new[], delete[]**
- ☐ Problématiques : fuites mémoire, dangling pointers, double free
- ☐ Constructeur de copie : sémantique de copie profonde
- ☐ Constructeur de déplacement : sémantique de transfert
- ☐ Règle des **3/5** : gestion complète du cycle de vie

Travaux pratiques

- Classe TableauDynamique avec allocation manuelle complète (constructeur, destructeur, copie, déplacement)

5 - Smart pointers et gestion mémoire moderne

- ☐ RAII : principe fondamental
- ☐ Smart pointers : **std::unique_ptr, std::shared_ptr, std::weak_ptr**
- ☐ **std::make_unique** [C++14], **std::make_shared**
- ☐ Règle du zéro : classes sans gestion manuelle

Travaux pratiques

- Refactorisation du TableauDynamique avec std::unique_ptr et std::vector
- Résolution d'un cycle de références shared_ptr

6 - Héritage et polymorphisme

- ☐ Principe de la dérivation.
- ☐ Les aspects syntaxiques : la définition des classes dérivées, les constructeurs.
- ☐ Le contrôle d'accès.
- ☐ La mise en œuvre du polymorphisme : les fonctions virtuelles.
- ☐ La réutilisation de code : classes abstraites et interfaces.
- ☐ Les aspects sémantiques et méthodologiques : la factorisation du code.

Travaux pratiques

- Implémentation d'une hiérarchie de classes avec polymorphisme
- Utilisation de override et final pour sécuriser l'héritage
- Classe abstraite avec méthodes virtuelles pures

7 - La surcharge des opérateurs

- ☐ Principe de la surcharge
- ☐ Surcharge interne et externe des opérateurs
- ☐ Surcharge des opérateurs binaires
- ☐ Surcharge particulière : l'opérateur indice, fonction, conversion

Travaux pratiques

- La surcharge des opérateurs + et ==
- Opérateur [] pour accès sécurisé aux éléments d'un tableau

8 - Les modèles

- ☐ Modèle de classe. Principes et mécanismes généraux. Spécialisation et surcharge de méthodes des modèles.



- ☐ Modèle de fonction : création des fonctions génériques.
- ☐ Spécialisation des modèles

Travaux pratiques

- Création de classes et fonctions génériques utilisant les modèles
- Spécialisation pour un type particulier

9 - Bibliothèque standard C++ (STL)

- ☐ Présentation de la Standard Template Library (STL)
- ☐ Conteneurs séquentiels : **std::vector**, **std::array**, **std::list**
- ☐ Conteneurs associatifs : **std::map**, **std::unordered_map**
- ☐ Itérateurs
- ☐ Algorithmes génériques
- ☐ Parcours avec range-based **for**

Travaux pratiques

- Manipulation de **std::vector** : ajout, suppression, tri
- Utilisation de **std::map** pour un annuaire téléphonique
- Algorithmes STL : **find_if**, **sort**, transform personnalisé

10 - Les exceptions

- ☐ Mécanisme des exceptions
- ☐ Blocs **try**, **catch**
- ☐ Génération d'exceptions (**throw**)
- ☐ Conception d'une hiérarchie d'exceptions
- ☐ **std::optional** [C++17] pour les résultats potentiels
- ☐ **noexcept**, garanties d'exceptions (basique/forte/nothrow)

Travaux pratiques

- Mise en œuvre de gestion d'exceptions
- Création d'une hiérarchie d'exceptions personnalisée
- Utilisation de **std::optional** pour éviter les valeurs nulles



Programmation moderne en C++

de C++11 à C++17

Référence : CPP02			
4 jours (32h) 	Présentiel / Distanciel <i>Modalité</i>	Intermédiaire 	2 à 12 participants <i>Format</i>

Prérequis Maîtrise des bases du C++ (syntaxe, classes, pointeurs, références) Expérience pratique en C++03	Public concerné Développeurs C++ souhaitant moderniser leur codebase C++03/98 Développeurs Java/C#/Python migrant vers C++ (avec bases C++ acquises)
---	---

OBJECTIFS DE LA FORMATION

- Comprendre les évolutions des standards C++11, C++14 et C++17.
- Écrire du code C++ plus sûr, expressif et maintenable
- Maîtriser la sémantique de déplacement et l'optimisation des copies
- Utiliser les smart pointers et l'idiome RAII pour une gestion mémoire sûre
- Exploiter la programmation fonctionnelle avec les lambdas et `std::function`
- Mettre en œuvre la programmation concurrente avec `std::thread`
- Utiliser efficacement les nouvelles fonctionnalités de la STL

PROGRAMME DÉTAILLÉ

1 - Introduction et écosystème C++

- ☐ Historique et évolution du langage C++
 - Bref historique
 - Philosophie de C++ moderne
- ☐ Outils de développement
- ☐ Modèle Mémoire : rappel sur la pile, le tas, les sections de données. Modèle mémoire d'un programme exécutable.

C++11 à C++17 : la machine à remonter le temps pour votre code obsolète – voyagez vers l'avenir et multipliez la vitesse de vos développements par dix !



2 - Évolutions syntaxiques majeures (C++11, C++14, C++17)

- ☐ La constante **nullptr**
- ☐ **auto** et **decltype**
- ☐ Initialisation uniforme **{ }**
- ☐ Les expressions **constexpr**
- ☐ Littéraux et espaces de noms
 - Littéraux modernes : littéraux de caractère et les chaînes littérales brutes
- ☐ Espaces de noms : espaces de nommage imbriqués [C++17], espaces de noms inline [C++11], alias d'espaces de noms, espaces de nommage anonymes.

Travaux pratiques

- Refactoring d'un code legacy : remplacement de NULL, boucles for, enum
- Création de fonctions constexpr pour calculs à la compilation
- Appliquer l'initialisation uniforme pour la déclaration de variables et l'initialisation de conteneurs, en observant les erreurs de compilation.
- Utiliser les chaînes littérales brutes pour définir un chemin de fichier complexe ou une expression régulière, et observer la simplification du code.
- Utiliser des espaces de noms imbriqués et des alias pour organiser le code

3 - Améliorations de la syntaxe et attributs (C++11, C++14, C++17)

- ☐ Améliorations Syntaxiques
 - Range-based **for** [C++11]
 - Énumérations fortement typées (**enum class**) [C++11]
 - Structured bindings [C++17]
 - Initialisation dans **if** et **switch** [C++17]
- ☐ Attributs de [C++11, C++14, C++17]

Travaux pratiques

- Utiliser range-based for pour parcourir différentes collections
- Créer des enum class pour représenter des états ou des codes d'erreur, et les utiliser dans des switch avec initialisation de variables.
- Appliquer les structured bindings pour simplifier l'accès aux éléments d'une structure de std::pair ou std::tuple

4 - Sémantique de déplacement et optimisation (C++11, C++14, C++17)

- ☐ Catégories d'expressions (lvalue, rvalue, prvalue)
- ☐ Sémantique de déplacement
- ☐ **std::move**
- ☐ Le perfect forwarding avec **std::forward**
- ☐ Règle des **0/3/5**
- ☐ Copy elision et RVO (Return Value Optimization) [C++17]
- ☐ Les opérateurs de casting **const_cast**, **static_cast**, **reinterpret_cast**.



Travaux pratiques

- Implémenter une classe de gestion de ressources avec constructeur de copie, constructeur de déplacement, opérateur d'affectation de copie et opérateur d'affectation de déplacement.
- Utiliser `std::move` et `std::forward` dans des fonctions génériques pour implémenter le perfect forwarding.

5 - Conception de classes et héritage moderne

- ☐ Initialisation des membres non-statiques [C++11]
- ☐ Constructeurs délégués [C++11]
- ☐ Les directives **=default**, et **=delete**
- ☐ Héritage et délégation des constructeurs
- ☐ Contrôle de l'héritage et de la virtualité : **override** et **final**
- ☐ Contrôle du downcasting à l'aide de l'opérateur **dynamic_cast**
- ☐ Modèle de classe **std::initializer_list**
- ☐ **std::string_view** : remplacement performant des chaînes

Travaux pratiques

- Concevoir une hiérarchie de classes polymorphiques en utilisant `override` et `final` pour garantir la bonne implémentation des interfaces.
- Utiliser `=default` et `=delete` pour contrôler précisément les opérations de copie et de déplacement d'une classe.
- Optimisation de paramètres de fonctions avec `string_view`

6 - Programmation fonctionnelle et les Lambdas

- ☐ Pointeurs de fonctions
- ☐ Foncteurs
- ☐ Expressions lambda
 - Syntaxe complète
 - Captures de variables
 - Closures et durée de vie des captures
 - Intérêt de **auto** et **decltype**
 - Lambdas génériques [C++14]
- ☐ **std::function**
 - Type-erasure pour les objets appelables (callables).
 - Utilisation de **std::function** pour stocker et passer des callbacks génériques.

Travaux pratiques

- Implémenter un tri personnalisé pour une structure de données complexe en utilisant une lambda générique comme comparateur.
- Créer un système de gestion d'événements ou de notifications utilisant `std::function`

7 - Pointeurs intelligents

- ☐ L'idiome RAII : principe fondamental



- ☐ Les pointeurs intelligents : **unique_ptr**, **weak_ptr** et **shared_ptr**
- ☐ Fonctions de création : **std::make_shared** et **std::make_unique**
- ☐ Pièges courants

Travaux pratiques

- Re-factorisation d'un code utilisant **new** / **delete**
- Détection et résolution d'un cycle de références **shared_ptr**

8 - Templates avec C++11/C++17

- ☐ Alias de template avec **using** [C++11]
- ☐ Variadic templates [C++11]
- ☐ Fold expressions [C++17]
- ☐ Types locaux et non nommés comme arguments template
- ☐ Sélection de branche à la compilation avec **if constexpr** [C++17]
- ☐ CTAD (Class Template Argument Deduction) [C++17]

Travaux pratiques

- Implémenter une fonction variadique générique
- Créer un template de fonction qui utilise **if constexpr** pour adapter son comportement en fonction des propriétés du type de ses arguments
- Refactoriser du code existant pour tirer parti de CTAD lors de l'instanciation de templates de classe complexes.

9 - Conteneurs et algorithmes STL modernes

- ☐ Nouveaux conteneurs : **std::array**, **std::forward_list**, **std::unordered_map**, **std::unordered_set**, **std::tuple**
- ☐ Nouveaux algorithmes et utilitaires
 - Algorithmes de conteneurs : **std::all_of**, **std::any_of**, **std::none_of**, **std::iota** [C++11]
 - Algorithmes utilitaires : **std::clamp**, **std::gcd** et **std::lcm** [C++17]
- ☐ Types sûrs : **std::optional**, **std::variant** et **std::any**
- ☐ Manipuler les chemins des fichiers avec **std::filesystem**
- ☐ L'espace de noms **std::chrono**
 - Les classes **duration** et **time_point**
 - Les horloges : **system_clock**, **steady_clock**, **high_resolution_clock**
 - Mesure des performances d'algorithmes
 - Littéraux de durée [C++14] : **h**, **min**, **s**, **ms**, **us**, **ns**

Travaux pratiques

- Manipulation de collections de données avec les algorithmes STL.



10 - Gestion des threads

- ☐ Création et terminaison des threads **std::thread**
- ☐ Partage de données entre les threads
- ☐ Synchronisation des opérations concurrentes
- ☐ Les types atomiques

Travaux pratiques

- Créer une application multithreadée simple où plusieurs threads accèdent à une ressource partagée, en utilisant `std::mutex` et `std::lock_guard` pour éviter les race conditions et garantir la sécurité des données.

ENHANCE

TECH



Les avancées du langage de C++11 à C++20

Référence : CPP03			
4 jours (32h) 	Présentiel / Distanciel <i>Modalité</i>	Intermédiaire / Avancé  	2 à 12 participants <i>Format</i>

Prérequis Bonne connaissance du C++ de base : classes, héritage, pointeurs Connaissance de la STL standard (vector, map, algorithmes de base)	Public concerné Développeurs C++ souhaitant se mettre à jour sur les standards modernes C++11 à C++20 Équipes migrant une base de code de C++03 vers le C++ moderne
--	--

OBJECTIFS DE LA FORMATION

- Comprendre l'évolution du langage C++ de C++11 à C++20 et identifier les principales nouveautés introduites dans ces versions.
- Utiliser les nouvelles fonctionnalités du langage telles que auto, decltype, constexpr, nullptr, les énumérations fortement typées et les boucles range-based for.
- Exploiter les améliorations apportées aux classes en C++ moderne, notamment l'initialisation des membres, les constructeurs délégués, les directives =default et =delete, ainsi que la sémantique de déplacement (move semantics).
- Mettre en œuvre des techniques modernes de programmation, telles que les expressions lambda, la programmation générique avec templates avancés et l'utilisation des smart pointers pour la gestion de la mémoire.
- Utiliser efficacement les nouvelles fonctionnalités de la bibliothèque standard (STL).
- Développer des applications modernes et performantes en utilisant les mécanismes de concurrence et les nouvelles fonctionnalités introduites par C++20 telles que les coroutines, modules et concepts.

PROGRAMME DÉTAILLÉ

1 - Nouveautés fondamentales

- ☐ Bref historique des versions de C++ passées, présentes et futures
- ☐ Outils de développement et compilateurs
- ☐ La constante **nullptr**

Explorez les révolutions C++11 à C++20, des coroutines aux concepts, pour des logiciels ultra-modernes qui anticipent les défis de demain et maximisent vos performances



- ☐ Initialisation uniforme **{ }**
- ☐ Inférence et déduction de type : **auto** et **decltype**
- ☐ Évaluation à la compilation : **constexpr**, **constexpr** et **constexpr**
- ☐ Structured bindings [C++17]
- ☐ Le range-based **for**
- ☐ Énumérations fortement typées : **enum class**
- ☐ Littéraux définis par l'utilisateur : suffixes, cas d'usage
- ☐ Espaces de noms **inline** et imbriqués
- ☐ Initialisation dans **if** et **switch** [C++17]
- ☐ **noexcept** : contrat de non-exception, remplacement de **throw()**
- ☐ Nouveaux attributs

Travaux pratiques

- Réécrire un module de traitement de données pour utiliser **auto**, **nullptr**, **range-based for** et **enum class**, en mesurant l'impact sur la lisibilité et la sécurité.
- Développer des fonctions **constexpr** pour des calculs de configuration, et expérimenter avec **constexpr/constexpr** pour garantir l'évaluation à la compilation.
- Appliquer les structured bindings pour simplifier l'accès aux résultats de fonctions retournant des paires ou des tuples, et pour itérer sur des structures de données complexes.

2 - Classes modernes

- ☐ Initialisation des membres
- ☐ Constructeurs délégués
- ☐ **=default** et **=delete**
- ☐ Contrôle de l'héritage et de la virtualité : **override** et **final**
- ☐ **std::string_view** : paramètre de chaîne sans copie [C++17]
- ☐ Références **rvalue** : syntaxe **T&&**, catégories d'expressions **lvalue** / **rvalue**
- ☐ Sémantique de déplacement
- ☐ **std::move** : conversion en rvalue reference
- ☐ Règle des **0 / 3 / 5**
- ☐ Copy elision et RVO (Return Value Optimization) [C++17]
- ☐ Initialisateurs désignés [C++20]
- ☐ Opérateur spaceship **<=>** : ordre total/partiel [C++20]
- ☐ **[[no_unique_address]]** : optimisation de la taille des classes [C++20]

Travaux pratiques

- Implémenter une classe **Buffer** avec sémantique de déplacement
- Appliquer la règle des 5 avec **=default/=delete** appropriés
- Utiliser **std::string_view** comme paramètre de fonction pour éviter les copies

3- Programmation fonctionnelle

Formez vos experts aux modules C++20 comme les ranges et le spaceship operator : transformez vos codebases legacy en solutions agiles, prêtes pour l'IA et le cloud.



- ☐ Les lambda-expressions : syntaxe, typage, utilisation
- ☐ Captures de variables
- ☐ Closures et durée de vie des captures
- ☐ Intérêt de **auto** et **decltype** avec les lambda-expressions
- ☐ **std::function** : type-erasure pour callables

Travaux pratiques

- Créer un système de callbacks avec lambdas et `std::function`
- Implémenter un tri personnalisé avec lambda de comparaison
- Démontrer un piège de capture by-reference avec durée de vie

4- Nouveautés dans les templates

- ☐ Alias de template avec `using` [C++11]
- ☐ Variadic templates
- ☐ Fold expressions [C++17]
- ☐ Types locaux et non nommés comme arguments template
- ☐ **if constexpr** : sélection de branche à la compilation sans SFINAE [C++17]
- ☐ CTAD (Class Template Argument Deduction) [C++17]
- ☐ Concepts et contraintes [C++20]
 - Définition et utilisation des concepts pour des templates
 - Contraintes sur les paramètres de template : la clause **requires** et l'expression **requires**
 - Concepts prédéfinis de la bibliothèque standard [C++20]

Travaux pratiques

- Implémentation d'une fonction variadique avec fold expression
- Remplacement d'un SFINAE par un Concept C++20

5 - Nouveautés dans la STL

- ☐ Nouveaux conteneurs et vues
 - Conteneurs : **array**, **forward_list**, **unordered_***, **tuple**
 - Vues : **string_view**, **span**
 - Types sûrs : **optional**, **variant**, **any**
- ☐ Manipuler les fichiers avec **std::filesystem**
- ☐ Nouveaux algorithmes
 - Algorithmes de conteneurs : **erase**, **erase_if**
 - Fonctions utilitaires numériques et bas niveau : **bit_cast**, **midpoint**, **lerp**
 - Formatage texte : **format**

Travaux pratiques

- Re-factorisation d'une API retournant des codes d'erreur spéciaux avec `std::optional`
- Parcourir un arbre de répertoires et filtrer des fichiers par extension



6 - Gestion mémoire

- ☐ L'idiome RAII : principe fondamental
- ☐ Les pointeurs intelligents : **unique_ptr**, **weak_ptr** et **shared_ptr**
- ☐ Utilisation avec la STL
- ☐ Pièges courants

Travaux pratiques

- Re-factorisation d'un code pour remplacer les pointeurs classiques par des pointeurs intelligents

7 - Le namespace chrono

- ☐ Les classes **duration** et **time_point**
- ☐ Les horloges **system_clock**, **steady_clock** et **high_resolution_clock**
- ☐ Littéraux de durée (**h**, **min**, **s**, **ms**, **us**, **ns**)
- ☐ Les fonctions de conversion : **duration_cast** et **time_point_cast**
- ☐ Les types de durée prédéfinis : **hour**, **minute**, etc.
- ☐ Calendrier moderne C++20
 - Nouveaux types : **year**, **month**, **day**, **year_month_day**, etc.
 - Opérateurs de conversion : **sys_days**
- ☐ Gestion des fuseaux horaires avec **std::chrono::zoned_time**
- ☐ Formatage de dates avec **std::format** et les types **chrono**

Travaux pratiques

- Mesurer le temps d'exécution d'une fonction et afficher le résultat en ms
- Convertir une durée en différentes unités

8 - Utilisation des threads

- ☐ Les threads
 - Création et gestion de threads **std::thread**
 - Les threads auto-joignable à la destruction (RAII) **std::jthread** [C++20]
 - Annulation coopérative d'un thread sans variable partagée [C++20]
 - Fonctions utilitaires : **yield**, **get_id**, **sleep_for**, **sleep_until**
- ☐ Synchronisation et exclusion mutuelle
 - Protection des données partagées avec les mutex
 - Exécution unique : **once_flag**, **call_once**
 - Variables conditionnelles
 - Synchronisation par sémaphores [C++20]
 - **std::latch** : barrière de synchronisation à usage unique [C++20]
 - **std::barrier** : barrière réutilisable avec fonction de complétion [C++20]
- ☐ Futures et asynchronisme : **promise**, **packaged_task**, **future**, **shared_future**, **async**



☐ Types atomiques

Travaux pratiques

- Calcul parallèle d'une somme : partitionnement sur plusieurs `std::jthread` avec annulation via `stop_token`
- Coordination de threads producteurs/consommateurs avec les sémaphores
- Synchronisation de N threads à un point de rendez-vous avec `std::barrier`
- Communication asynchrone avec `promise/future` : traitement d'une tâche longue en arrière-plan

9- Autres nouveautés

☐ Introduction aux coroutines

- Comparaison fonction / coroutine / thread
- **`co_yield`** : production paresseuse de valeurs
- **`co_await`** : suspension en attente d'une opération asynchrone
- **`co_return`** : clôture d'une coroutine avec une valeur de retour
- **`std::coroutine_handle`** : contrôle du cycle de vie

☐ Modules

- Motivation
- Syntaxe : **`module units, export, import, module partitions`**
- Comparaison header/module

Travaux pratiques

- Générateur paresseux de séquences Fibonacci avec `co_yield`
- Conversion d'un header simple en module C++20, et vérification du temps de compilation.



C++ Programmation

Avancée Expert

Référence : CPP04			
4 jours (32h) 	Présentiel / Distanciel <i>Modalité</i>	Expert  Expert	2 à 12 participants <i>Format</i>

Prérequis Solide expérience en C++ (niveau intermédiaire minimum) Maîtrise des concepts OO : classes, héritage, polymorphisme Connaissance des smart pointers et de la STL de base	Public concerné Développeurs C++ confirmés souhaitant maîtriser les techniques avancées du langage Architectes logiciels concevant des systèmes à haute performance ou contrainte temps-réel Tech leads souhaitant établir des standards de code et bonnes pratiques C++ modernes
--	---

OBJECTIFS DE LA FORMATION

- Maîtriser les mécanismes avancés de la programmation orientée objet en C++, notamment l'héritage, le polymorphisme et la gestion du cycle de vie des objets..
- Comprendre et utiliser les conversions de types, les pointeurs sur membres et les idiomes de conception .
- Mettre en œuvre la programmation générique avec les templates et les techniques de méta-programmation.
- Gérer efficacement les ressources et les erreurs à l'aide du principe RAII, des smart pointers et des mécanismes de gestion des exceptions.
- Exploiter les fonctionnalités modernes du langage C++, telles que les expressions lambda, les références rvalue, les move semantics et le perfect forwarding.
- Développer des applications performantes et modernes en utilisant les mécanismes de concurrence , les coroutines et les structures de la bibliothèque STL.

PROGRAMME DÉTAILLÉ

1 - Rappels, pièges et bonnes pratiques modernes

- ☐ Forme canonique d'une classe, règle des **0/3/5/6**
- ☐ Constructeurs : délégation, ordre d'initialisation, pièges de l'initialisation en deux phases
- ☐ Allocation dynamique et destructeur virtuel

Devenez maître absolu du C++ avec nos modules experts : de la méta-programmation aux coroutines, pour des architectures critiques qui défient les limites de la performance



- ☐ Surcharge de l'opérateur d'affectation : copie vs déplacement
- ☐ Bonnes pratiques C++ modernes : **const**, **constexpr**, **auto**, **override**, **final**, **[[nodiscard]]**
- ☐ Undefined Behavior courant : accès hors bornes, UB arithmétique, strict aliasing

Travaux pratiques

- Exemples illustrant les pièges classiques du C++ moderne.
- Code review : repérer les violations de la règle des 0/3/5 dans un code fourni

2 - Introspection avancée : pointeurs sur membres et RTTI

- ☐ Syntaxe et Utilisation des Pointeurs sur Membres
 - Pointeurs sur membres de données
 - Pointeurs sur méthodes
 - Applications : implémentation de tables de fonctions, de callbacks génériques.
- ☐ Principe du RTTI (Run-Time Type Information)
 - **dynamic_cast** : pour la navigation sécurisée dans une hiérarchie de classes polymorphiques.
 - **typeid** et **type_info** : pour l'identification du type dynamique d'un objet.
 - Limitations et coûts du RTTI.

Travaux pratiques

- Utilisation de pointeurs sur méthodes pour implémenter des actions génériques.
- Utilisation de **dynamic_cast** pour déterminer le type réel de l'objet et appeler une méthode spécifique à ce type.
- Créer un petit outil d'analyse de types qui utilise **typeid**

3 - Utilisation avancée de l'héritage

- ☐ Comparaison héritages public, privé et protégé
- ☐ Héritages multiple et multiple répété
- ☐ Héritage virtuel et conséquences
- ☐ Principe de substitution de Liskov

Travaux pratiques

- Concevoir une hiérarchie de classes complexe utilisant l'héritage multiple et l'héritage virtuel pour modéliser un scénario concret
- Refactoriser une hiérarchie de classes existante pour respecter le Principe de Substitution de Liskov, en identifiant et corrigeant les violations

4 - Conversions et types modernes

- ☐ Conversions implicites
- ☐ Conversions explicites (**const_cast**, **static_cast**, **reinterpret_cast**, **dynamic_cast**)
- ☐ Conversion via constructeurs ou l'opérateur de conversion
- ☐ Usage de **explicit**



- ☐ Introduction aux types modernes : **std::optional**, **std::variant**, **std::any**, **std::string_view**, **std::span**

Travaux pratiques

- Choix du type de cast et mise en œuvre dans différentes situations
- Développer une fonction qui retourne un `std::optional<T>` pour indiquer la réussite ou l'échec d'une opération. Utiliser `std::variant` pour gérer des données de types hétérogènes et `std::string_view` pour optimiser le passage de chaînes

5 - Utilisation des références rvalue

- ☐ Rappels sur les références lvalue
 - Définition et utilisation des références lvalue (**T&**).
 - Utilité des adaptateurs de référence (**std::reference_wrapper**).
- ☐ Définition des références rvalue (**T&&**) [C++11]
- ☐ Applications des références rvalue : move semantics [C++11]
- ☐ Perfect forwarding [C++11] : **std::move** et **std::forward**
- ☐ Principe du copy elision avec RVO (Return Value Optimization)

Travaux pratiques

- Mise en œuvre des références rvalue pour la création et la copie d'objet par déplacement
- Utiliser `std::move` et `std::forward` dans des fonctions génériques pour implémenter le perfect forwarding
- Démonstration du RVO sur une fonction retournant un objet complexe

6 - Foncteurs et programmation fonctionnelle

- ☐ Foncteurs : principe, intérêt vs pointeurs de fonctions, état interne
- ☐ Foncteurs prédéfinis de la bibliothèque standard
- ☐ Les lambda-expressions : syntaxe, typage, utilisation
- ☐ La gestion des fermetures (closures) liées au contexte
- ☐ Lambdas génériques [C++14] et lambdas avec template explicite [C++20]
- ☐ **std::function** : type-erasure pour callables
- ☐ **decltype** et retour de type **auto** : trailing return types
- ☐ Adaptateurs modernes

Travaux pratiques

- Foncteur de comparaison avec état interne pour trier une collection selon un critère configurable

7 - Templates et programmation générique

- ☐ Templates de fonctions et classes
- ☐ Surcharge et spécialisation partielle/complète
- ☐ Variadic templates et variables templates
- ☐ **if constexpr** [C++17]
- ☐ Concepts et contraintes [C++20]



- ☐ Déduction des arguments template (CTAD) : simplification de l'instanciation [C++17]

Travaux pratiques

- Implémentation de classes génériques avec variadic templates.
- Remplacement d'un SFINAE complexe par un Concept C++20

8 - Idiomes C++ et la méta-programmation

- ☐ Idiomes de conception courants : Traits, Policy, SFINAE, CRTP
- ☐ Métaprogrammation à la compilation
- ☐ Les mot-clés **constexpr** et **constexpr**
- ☐ Optimisations

Travaux pratiques

- Mise en œuvre de chacun des 4 idiomes étudiés
- Implémentation d'idiomes génériques et calculs à la compilation.

9 - Pointeurs intelligents (smart pointers)

- ☐ Principe de la gestion RAII des ressources
- ☐ Mise en œuvre de **unique_ptr**, **shared_ptr**, **weak_ptr**
- ☐ Utilisation de **make_unique** et **make_shared**
- ☐ Destructeur personnalisé et danger des pointeurs bruts

Travaux pratiques

- Refactoring d'une classe legacy avec RAII et smart pointers

10 - Gestion des exceptions

- ☐ Principe du traitement des anomalies
- ☐ Utilisation des mots-clés **try**, **catch** et **throw**
- ☐ Traiter les exceptions sur place
- ☐ Propager les exceptions
- ☐ Créer des classes d'exception personnalisées
- ☐ Le mot-clé **noexcept**
- ☐ Échange d'exceptions entre threads avec **std::exception_ptr**

Travaux pratiques

- Calcul de la somme des soldes mensuels d'un compte bancaire par lecture d'un fichier CSV avec mise en œuvre d'un traitement sur place d'exceptions puis en propageant les exceptions à la fonction appelante

11 - Évolutions dans la bibliothèque STL

- ☐ Nouveaux itérateurs : **cbegin**, **cend**
- ☐ Nouveaux conteneurs : **std::array**, **unordered_map**, **unordered_set**, **tuples**
- ☐ Adaptateurs d'itérateurs et stream iterators
- ☐ Algorithmes ensemblistes et pipelines avec **std::ranges** [C++20]

**Travaux pratiques**

- Mise en œuvre de conteneurs modernes et algorithmes avec ranges et pipelines

12 - Programmation concurrente

- ☐ Création et gestion de threads
- ☐ Synchronisation avec mutex
- ☐ RAII avec les mutex : **lock_guard**, **unique_lock**, **shared_lock**
- ☐ Exécution unique avec **once_flag**, **call_once**
- ☐ Les variables conditionnelles
- ☐ Futures et programmation asynchrone
- ☐ Types atomiques

Travaux pratiques

- Parallélisation d'un programme avec les threads pour améliorer les performances
- Mise en œuvre de mutex pour éviter la corruption de données dans l'accès des structures de données partagées
- Utilisation de promise et future pour communication asynchrone entre threads

13 - Coroutines

- ☐ Comparaison fonction/coroutine
- ☐ Mise en œuvre des opérateurs **co_await**, **co_yield**, **co_return**
- ☐ Rôle de **std::coroutine_handle**
- ☐ Applications des coroutines

Travaux pratiques

- Mise en œuvre d'un serveur asynchrone simple avec coroutines

14 - Autres nouveautés de C++20/23

- ☐ Modules et partitions
- ☐ Opérateur spaceship **<=>**, initialisateurs désignés, pattern matching (inspect)
- ☐ **constinit**, **std::views** et pipelines
- ☐ **std::format** [C++20]

Travaux pratiques

- Démonstration de **std::format**, **std::views** et modules sur des exemples concrets



Programmation multicore et concurrence avancée en C++

Référence : MCC01

4 jours (32h) 	Présentiel / Distanciel <i>Modalité</i>	Expert  Expert	2 à 12 participants <i>Format</i>
---	---	---	---

Prérequis

Solide expérience en C++ (niveau intermédiaire minimum)
Maîtrise des concepts OO : classes, héritage, polymorphisme
Connaissance des smart pointers et de la STL de base
Notions bas niveau (mémoire/cache/CPU) appréciées

Public concerné

Développeurs C++ expérimentés souhaitant maîtriser les concepts et les outils de la programmation parallèle et concurrente.
Architectes logiciels et chefs de projets techniques impliqués dans la conception d'applications haute performance.
Ingénieurs souhaitant optimiser des applications existantes ou développer de nouvelles solutions exploitant les architectures Multicore.

OBJECTIFS DE LA FORMATION

- Maîtriser les enjeux de la programmation Multicore avec le C++ moderne
- Concevoir et développer des applications multithreadées performantes
- Protéger les ressources partagées avec les mutex
- Concevoir des structures de données thread-safe (avec ou sans verrou)
- Exploiter les outils modernes du C++ (C++17/C++20) pour la concurrence
- Déboguer et profiler des applications Multicore

PROGRAMME DÉTAILLÉ

1 - Fondamentaux de la programmation concurrente et parallèle

- ☐ Parallélisme vs concurrence
 - Définitions et distinctions.
 - Exemples d'applications concurrentes et parallèles.
- ☐ Architectures Matérielles
 - CPU, cœurs, cache (L1, L2, L3), hyperthreading.
 - Modèle mémoire partagée vs mémoire distribuée.

Vos processeurs multicore dorment-ils ? Réveillez-les avec notre formation C++ pour des apps qui tournent comme une symphonie orchestrée, sans fausses notes de concurrence !



☐ Pourquoi la Programmation Multicore en C++ moderne?

☐ Défis de la Programmation Concurrente

- Race conditions, deadlocks, livelocks, starvation.
- Problèmes de visibilité et d'ordre des opérations.

☐ Rappels essentiels de C++ moderne pour la concurrence

- RAII et gestion sûre des ressources (mutex, threads)
- Move semantics en contexte concurrent
- Lambdas et callbacks thread-safe

Travaux pratiques

- Modernisation d'un code existant et analyse comparative

2 - Gestion des threads C++ standard

☐ Création et lancement de threads

- **std::thread** : constructeurs, fonctions membres (**join**, **detach**).
- Passage d'arguments aux fonctions de thread.
- Gestion des exceptions dans les threads.
- **std::jthread** [C++20] : thread auto-joignable avec RAII
- **std::stop_token** et **stop_source** : annulation coopérative
- **std::latch** / **std::barrier** [C++20] : synchronisation moderne
- Fonctions utilitaires : **yield**, **get_id**, **sleep_for**, **sleep_until**

☐ Partage des données entre les threads

- Variables locales aux threads (**thread_local**)

Travaux pratiques

- Implémenter une application multithread simple où plusieurs threads exécutent des tâches indépendantes
- Passer des arguments à une fonction de thread et récupérer un résultat simple

3 - Protection des données partagées et synchronisation

☐ Problèmes liés au partage de données entre threads

- Exemples concrets de race conditions et leurs conséquences.
- Propriété de l'exclusion mutuelle.

☐ Les mutex C++ standard

- **std::mutex** : verrouillage et déverrouillage de base.
- **std::recursive_mutex** : pour les fonctions récursives.
- **std::timed_mutex** : tentatives de verrouillage avec timeout

☐ Gestion RAII des mutex : **std::lock_guard** et **std::unique_lock**

☐ Structuration du code pour protéger les données partagées

Travaux

- Différents cas pour résoudre les problèmes liés au partage des données entre



pratiques

- threads
- Implémenter un reader-writer lock avec `std::shared_mutex`

4 - Synchronisation avancée et communication inter-threads

- ☐ Variables de condition
- ☐ Futures et tâches asynchrones : **`std::promise`**, **`std::packaged_task`**, **`std::future`**, **`std::shared_future`** et **`std::async`**
- ☐ Initialisation thread-safe : **`std::call_once`**
- ☐ Gestion des erreurs et propagation d'exceptions entre threads

Travaux pratiques

- Implémenter un modèle producteur-consommateur avec `std::condition_variable` pour synchroniser l'accès à une file d'attente partagée
- Utiliser `std::async` et `std::future` pour exécuter une tâche en arrière-plan et récupérer son résultat de manière non bloquante
- Développer un singleton thread-safe en utilisant `std::call_once`

5 - Modèle mémoire de C++ et les opérations sur les types atomiques

- ☐ Concepts fondamentaux du modèle de mémoire en C++
 - Ordre des opérations, visibilité des écritures.
 - Problèmes de ré-ordonnancement par le compilateur et le processeur.
- ☐ Types atomiques
 - Opérations garanties atomiques (lecture, écriture, RMW).
 - **`std::atomic_flag`** : le type atomique de base
 - Utilisation de **`std::atomic<T>`**
- ☐ Ordre de la Mémoire (`memory_order`)
 - **`memory_order_relaxed`**, **`memory_order_acquire`**, **`memory_order_release`**, **`memory_order_acq_rel`** et **`memory_order_seq_cst`**
 - Impact sur les performances et la correction du code
- ☐ Barrières Mémoire **`std::atomic_thread_fence`** : imposer un ordre entre les opérations non-atomiques.

Travaux pratiques

- Implémentation de sémaphores avec les variables atomiques
- Conception et implémentation d'une pile thread-safe non bloquante

6 - Structures de données lock-free

- ☐ Introduction aux structures lock-free
 - Avantages et inconvénients par rapport aux structures basées sur des verrous
 - Concepts de base : CAS (Compare-And-Swap)
- ☐ Implémentation de structures lock-free simples
 - Pile ou file lock-free avec **`std::atomic`** et CAS.



❑ Problèmes et Solutions

- Problème ABA (ABA problem) et solutions.
- Introduction aux techniques avancées (hazard pointers, RCU).

Travaux pratiques

- Implémenter une pile lock-free
- Comparer les performances de la pile lock-free avec une version basée sur les mutex
- Étudier le problème ABA et proposer une solution pour une file lock-free.

7 - Algorithmes parallèles de la STL

❑ Politiques d'exécution

❑ Algorithmes STL supportant le parallélisme

❑ Performance et seuils de rentabilité

- Overhead de parallélisation
- Taille minimale des données pour bénéficier du parallélisme
- Impact du nombre de cœurs disponibles
- Cache locality et faux partage (false sharing)

❑ Pièges et bonnes pratiques

- L'overhead du parallélisme
- Gestion des exceptions dans les algorithmes parallèles
- Thread-safety des foncteurs
- Localité des données

Travaux pratiques

- Benchmark : `std::sort` séquentiel vs `std::sort` avec `execution::par`
- Utiliser `std::transform_reduce` pour une somme parallèle sur grand vecteur
- Mesurer le speedup en fonction de la taille des données (1K, 1M, 10M éléments)
- Identifier le seuil de rentabilité du parallélisme pour différents algorithmes

8 - Débogage, profilage et bonnes pratiques

❑ Débogage d'Applications Multicore

- Outils de débogage et leurs fonctionnalités Multicore.
- Techniques de débogage spécifiques : points d'arrêt conditionnels, watchpoints, débogage à distance.
- Outils d'analyse de race conditions et deadlocks (ThreadSanitizer, Helgrind).

❑ Profilage de performances

- Outils de profilage (perf, Valgrind/Callgrind, VTune, Visual Studio Profiler).
- Identification des goulots d'étranglement (contention de verrous, faux partage de cache).

❑ Bonnes Pratiques et Design Patterns

- Design patterns de concurrence (Producteur-Consommateur, Lecteur-Écrivain, Thread Pool).



- Éviter les deadlocks : ordre de verrouillage, **std::lock**.
- Minimiser la contention : granularité des verrous, structures lock-free.
- Alignement mémoire (alignas)
- Notion de false sharing illustrée
- Tests unitaires pour le code concurrent.

Travaux pratiques

- Utiliser ThreadSanitizer (TSan) pour détecter les race conditions
- Profiler avec perf/VTune pour identifier les goulots d'étranglement
- Déboguer un deadlock avec gdb
- Analyser le faux partage de cache avec perf

ENHANCE

TECH



Développeur Cybersécurité en C/C++

Référence : SEC01			
5 jours (40h) 	Présentiel / Distanciel <i>Modalité</i>	Avancé / Expert  	2 à 12 participants <i>Format</i>

Prérequis Solide expérience en C++ (niveau intermédiaire minimum) Connaissance en assembleur : souhaitée	Public concerné Développeurs C++ confirmés (systèmes, embarqué, critique) Architectes logiciels responsables de la sécurité Responsables sécurité applicative (AppSec)
---	--

OBJECTIFS DE LA FORMATION

- Sensibiliser les développeurs sur la sécurité du code
- Connaître le rôle des acteurs et la classification des risques : CERT, CWE, OWASP
- Comprendre la structure mémoire (heap, stack, etc.), et savoir comment les vulnérabilités exploitent ces parties pour compromettre un système
- Savoir repérer les failles de sécurité dans le code
- Étudier des cas d'attaques contre un code tout en connaissant les mécanismes pour se protéger
- Connaître les bonnes pratiques à appliquer et les différentes approches pour produire un code sécurisé
- Renforcer la protection des logiciels contre les tentatives de rétro-conception et de piratage
- Étudier les mécanismes de protection offerts au niveau d'un système d'exploitation et par les compilateurs

PROGRAMME DÉTAILLÉ

1 - Introduction : risques, acteurs et classification

- ☐ C et C++ des langages peu sécurisés ?
- ☐ Connaître les risques liés à la programmation
- ☐ Les traces laissées par les développeurs
- ☐ Acteurs de la sécurité : CERT/CC, MITRE CWE, OWASP, PCI-DSS, MISRA - rôles et périmètres
- ☐ Classification des risques

C++ : le géant endormi des vulnérabilités – réveillez vos développeurs pour transformer chaque ligne de code en bouclier impénétrable contre les cyberpirates



- ☐ Les CWE les plus courants en C/C++ : CWE-119 (buffer overflow), CWE-416 (use-after-free), CWE-125 (out-of-bounds read)
- ☐ Guides de codage sécurisé
- ☐ Cycle de vie sécurisé du développement (SDLC) : threat modeling, code review, tests de sécurité

Travaux pratiques

- Analyse d'un code C++ fourni : identification des CWE présents et calcul du score CVSS pour chaque vulnérabilité
- Consultation du guide CERT C++ : trouver les règles applicables à 3 situations de code données
- Démonstration de désassemblage d'un binaire : repérer les informations laissées par le compilateur

2 - Vulnérabilités : mémoire et arithmétiques

- ☐ Modèle mémoire d'un programme exécutable
- ☐ Modèle mémoire d'un programme C/C++
- ☐ Pile d'appel (call stack)
- ☐ Buffer overflow sur la pile
- ☐ Stack smashing : corruption de la pile et détournement du flux d'exécution
- ☐ Heap overflow : corruption du tas
- ☐ Use-after-free et double free
- ☐ Dangling pointers : pointeurs sur des objets détruits
- ☐ Sécurité des entiers : integer overflow, signedness confusion, integer truncation
- ☐ Attaques avancées : return-to-libc, Return-Oriented Programming (ROP)
- ☐ Pointer subterfuge : modification de pointeurs de fonctions ou de vtables

Travaux pratiques

- Exploration de certaines failles de sécurité : buffer overflow, stack smashing, pointer subterfuge, exécution d'un code arbitraire, return-to-libc et ROP attaques, etc.
- Mettre des solutions pour remédier à chaque type d'attaque.

3 - Vulnérabilités liées aux entrées et aux ressources

- ☐ Chaînes de caractères en C : taille fixe, fonctions dangereuses et remplacements sûrs
- ☐ Format string attacks
- ☐ Injection de code via les entrées utilisateur : validation, sanitisation, principe du moindre privilège
- ☐ Gestion des fichiers : path traversal, symlink attacks, TOCTOU (Time-Of-Check Time-Of-Use)
- ☐ Gestion des ressources système : descripteurs de fichiers, handles, sockets

Travaux pratiques

- Exploitation d'une format string attack
- Path traversal : accès à un fichier hors du répertoire autorisé
- TOCTOU : démonstration de la race condition et correction



4 - C++ moderne pour un code sécurisé

- ☐ Gestion mémoire sécurisée
- ☐ Tableaux sûrs : **std::array** et **std::vector** à la place des tableaux C bruts
- ☐ **std::span** : vue non-propritaire sur un tableau avec taille connue
- ☐ Entiers sûrs : éviter les conversions implicites signées/non signées
- ☐ Macros et préprocesseur : dangers des macros
- ☐ Gestion des erreurs
- ☐ Exceptions : garanties basique/forte/nothrow, **noexcept** et sécurité des ressources
- ☐ Classes sécurisées
- ☐ Multithreading sécurisé : protection des données partagées
- ☐ **std::string_view** et **std::string**
- ☐ Initialisation défensive : initialisation uniforme **{}**

Travaux pratiques

- Refactoring sécurisé : transformer un module C legacy en C++ moderne.
- Classe de gestion de buffer sécurisé
- Code review croisé : les stagiaires audient mutuellement leur code

5 - Sécurisation des communications

- ☐ Fondamentaux de la cryptographie
- ☐ Fonctions de hachage
- ☐ Chiffrement symétrique
- ☐ Chiffrement asymétrique
- ☐ Bibliothèques cryptographiques en C/C++ : comparaison et cas d'usage
- ☐ Protocoles SSL/TLS
- ☐ Pièges courants en cryptographie
- ☐ Gestion des certificats numériques
- ☐ Bonnes pratiques

Travaux pratiques

- Chiffrement/déchiffrement AES-GCM d'un fichier avec OpenSSL
- Connexion TLS client-serveur en C++ avec vérification du certificat serveur

6 - Rétroconception et ingénierie inverse (Reverse Engineering)

- ☐ Concepts de base de la rétroconception (reverse engineering)
- ☐ Outils de rétroconception : Introduction à IDA Pro et Ghidra
- ☐ Comprendre et naviguer dans les appels de fonction et les conventions d'appel (cdecl, stdcall, fastcall)
- ☐ Analyse des tables de dispatch virtuelles (vtables) pour le C++ orienté objet
- ☐ Décomposition des binaires : Analyse des segments de code et reconstruction du flux de contrôle
- ☐ Mise en place de protections contre la rétro-conception

Pensez comme un attaquant, codez comme un défenseur : transformez vos applications C/C++ en systèmes résilients face aux attaques les plus avancées.



☐ Recommandations pour sécuriser le code et prévenir la rétro-conception

Travaux pratiques

- Désassemblage et décompilation d'un binaire C++ fourni avec Ghidra.
- Identification des conventions d'appel dans le code assembleur
- Mise en place d'un binaire durci

7 - Durcissement des applications (Hardening)

- ☐ Outils d'analyse syntaxique
- ☐ Outils d'analyse dynamique
- ☐ Techniques de mitigation des vulnérabilités (ASLR, DEP, canaries, etc.).
- ☐ Mécanismes de protection offerts par les systèmes d'exploitation
- ☐ Mécanismes et options de protection offerts par GCC/Clang
- ☐ Sécurité au niveau de l'éditeur de liens
- ☐ Validation du Hardening

Travaux pratiques

- Utiliser un analyseur syntaxique pour identifier les failles
- Détection des fuites de mémoire avec Valgrind
- Tester plusieurs configurations de compilation avec gcc/clang en appliquant le hardening.
- Validation d'un binaire durci

ENHANCE

TECH



Développer des applications IoT en C++

Référence : IOT01			
4 jours (32h) 	Présentiel / Distanciel <i>Modalité</i>	Intermédiaire / Avancé  	2 à 12 participants <i>Format</i>

Prérequis Expérience pratique en développement C++ (au moins 6 mois) Connaissance des bases : pointeurs, classes, STL (vector, map) Notions de programmation système recommandées (processus, fichiers, réseau) Aucune connaissance préalable en IoT ou en électronique embarquée requise	Public concerné Développeurs C++ souhaitant concevoir des applications pour objets connectés Ingénieurs embarqués souhaitant étendre leurs systèmes vers le cloud et l'IoT Chefs de projets techniques supervisant des équipes IoT et souhaitant en maîtriser les enjeux
--	--

OBJECTIFS DE LA FORMATION

- Comprendre l'écosystème IoT : architectures, acteurs, standards et contraintes spécifiques
- Connaître les différentes technologies réseau pour l'IoT et savoir en utiliser certains mécanismes de base
- Comprendre et mettre en œuvre les technologies LoRa/LoRaWAN
- Prototyper une communication LoRa/LoRaWAN intégrée dans une chaîne IoT
- Être capable de prototyper une solution communicante LoRaWAN sur base Arduino ou Raspberry
- Mettre en œuvre une chaîne IoT complète, de la collecte de données capteurs LPWAN à la visualisation en temps réel via le cloud
- Développer des applications C++ intégrant communication, stockage et supervision des données IoT

PROGRAMME DÉTAILLÉ

1 - Concepts IoT, architectures et réseaux de proximité

- ☐ Présentation des concepts associés à l'IoT
- ☐ Les 4 couches de l'architecture IoT : perception, réseau, middleware, application
- ☐ Acteurs et écosystème



- ☐ Contraintes spécifiques IoT
- ☐ Systèmes d'exploitation pour l'IoT
- ☐ Modèles de déploiement cloud : vue d'ensemble
- ☐ Panorama des technologies sans fil courte portée
 - Bluetooth Low Energy (BLE)
 - WiFi Direct : communication peer-to-peer sans point d'accès
 - RFID et NFC
 - Critères de sélection : portée, débit, consommation, coût, sécurité

Travaux pratiques

- Étude critique de l'architecture d'une solution IoT

2 - Réseaux LPWAN : SigFox, LoRa et LoRaWAN

- ☐ Spécificités des réseaux dédiés IoT
- ☐ Les spécificités des réseaux IoT (distance, positionnement des équipements, autonomie)
- ☐ Caractéristiques des réseaux LPWAN
- ☐ Étude de SigFox et LoRaWAN
- ☐ Quelle programmation ?
- ☐ Réseaux privés ou publics
- ☐ Les opérateurs
- ☐ Sécuriser un réseau IoT

Travaux pratiques

- Choix d'une technologie LPWAN sous contraintes réelles
- Estimation de la consommation énergétique d'un nœud IoT

3 - LoRa et LoRaWAN

- ☐ Présentation du standard LoRa/LoRaWAN
- ☐ La modulation radio LoRa et la spécification LoRaWAN
- ☐ L'architecture générale d'un réseau LoRaWAN
- ☐ Les classes A-B-C d'équipements ("mote")
- ☐ Les mécanismes ABP et OTAA d'inscription sur le réseau
- ☐ Prototypage avec Arduino Uno et le shield SX1276
- ☐ Présentation de la bibliothèque LMIC
- ☐ Les fonctions des concentrateurs ("gateways") et des serveurs réseaux et applicatifs
- ☐ Sécuriser un réseau LoRaWAN

Travaux pratiques

- Communication peer-to-peer via modules LoRa
- Configuration d'une gateway et enregistrement sur le serveur LoRaWAN
- Développement d'une mote ABP ou OTAA en C++

4 - Développement embarqué

- ☐ Présentation de la Raspberry Pi

Passez du prototype à la production en développant des systèmes IoT complets, interconnectés et prêts à être déployés dans des environnements réels



- ☐ Choix du système d'exploitation et outils de développement C++
- ☐ Utilisation de la bibliothèque pigpio sous Raspberry Pi
- ☐ Gestion des GPIO et signaux PWM
- ☐ Communication M2M : UART, I2C et SPI

Travaux pratiques

- Acquisition de données capteurs via bus série/I2C/SPI

5 - Communication MQTT et intégration cloud

- ☐ Protocole MQTT
 - Introduction au protocole MQTT
 - Concepts clés : publisher, subscriber, topics, QoS
 - Présentation de la bibliothèque Paho MQTT C++
 - Développement d'un client MQTT pour la transmission de données IoT
- ☐ API REST et intégration cloud
 - Principes des API REST pour les échanges de données dans l'IoT
 - Méthodes HTTP (GET, POST, PUT, DELETE) pour interagir avec des objets connectés
 - Consommation des API REST depuis une application C++
 - Manipulation de données JSON
 - Sécurité : authentification (OAuth, API keys) et chiffrement (HTTPS)

Travaux pratiques

- Publication des mesures capteurs (T, H, CO2) depuis la Raspberry Pi vers un broker MQTT Mosquitto
 - Réception des messages LoRaWAN et republication sur le broker MQTT local
 - Développement d'un client REST en C++ : envoi de données JSON vers une API cloud

6- Stockage time-series avec InfluxDB

- ☐ Bases de données time-series
- ☐ Concepts InfluxDB
- ☐ Intégration avec des applications C++
- ☐ Intégration IoT temps réel

Travaux pratiques

- Configuration d'InfluxDB pour stocker des données IoT, requêtes sur des séries chronologiques
 - Développement d'une application C++ pour insérer des données via l'API HTTP
 - Implémentation d'un envoi batch de mesures capteurs pour optimiser les performances

7- Techniques de visualisation et monitoring

- ☐ Outils pour visualiser et monitorer les données IoT en temps réel
- ☐ Création de dashboards interactifs pour l'analyse des métriques
- ☐ Alertes et monitoring proactif pour détecter les anomalies



- ❑ Outil présenté : Grafana, plateforme open-source pour la visualisation et l'observabilité
- ❑ Intégration avec InfluxDB et MQTT pour des graphiques dynamiques et alertes

**Travaux
pratiques**

- Configuration de Grafana pour afficher des données IoT
- Création de panels et tableaux de bord dynamiques

ENHANCE

TECH



Créer des applications ROS2 en C++

Référence : ROS01			
4 jours (32h) 	Présentiel / Distanciel <i>Modalité</i>	Intermédiaire / Avancé  	2 à 12 participants <i>Format</i>

Prérequis Bonne maîtrise du C++ moderne Expérience en programmation Linux Notions de robotique souhaitées mais non obligatoires	Public concerné Développeurs C++ souhaitant créer des applications robotiques avec ROS2 Ingénieurs robotique migrant de ROS1 vers ROS2 ou démarrant avec ROS2 Chercheurs et ingénieurs R&D travaillant sur des robots autonomes ou industriels
---	--

OBJECTIFS DE LA FORMATION

- Comprendre l'architecture distribuée de ROS2 et le rôle du middleware DDS dans la communication inter-processus.
- Sélectionner et configurer les politiques QoS adaptées aux contraintes de leur projet (capteurs temps réel, commandes critiques, tolérance aux pannes).
- Concevoir, développer et structurer une application robotique en C++ avec ROS2 selon les bonnes pratiques de l'écosystème.
- Exploiter les simulateurs et les outils de débogage pour développer, tester et valider une application ROS2.
- Exploiter les fonctionnalités avancées de ROS2 en C++, notamment les composants, la composition de nœuds, les exécuteurs, les callback groups et les plugins.

PROGRAMME DÉTAILLÉ

1 - Introduction à ROS2

- ☐ Historique de ROS : genèse, limites de ROS1, naissance de ROS2
- ☐ Contraintes et besoins
- ☐ Licence et communauté
- ☐ Comparaison technique ROS1 vs ROS2
- ☐ Tour d'horizon des robots compatibles
- ☐ Installation et configuration

Codez des robots comme des systèmes critiques : alliez ROS2 et C++ sécurisé pour garantir fiabilité, sûreté et résilience dans les environnements industriels exigeants.

**Travaux pratiques**

- Installation et configuration de l'environnement

2 - Architecture et concepts fondamentaux

- ☐ Architecture middleware basée sur DDS
- ☐ QoS (Quality of Service) : introduction et concepts clés
- ☐ Éléments fondamentaux d'une application ROS2
- ☐ Principe du graphe de calcul ROS2
- ☐ Architecture d'une application ROS2
- ☐ Organisation des fichiers : workspaces, packages, sources, builds, installs
- ☐ Structure d'un package ROS2
- ☐ Présentation des packages disponibles

Travaux pratiques

- Mise en place d'un workspace ROS2

3 - Interface CLI de ROS2

- ☐ Présentation de l'interface en ligne de commande ROS2
- ☐ ROS2 CLI : philosophie et cohérence des commandes
- ☐ Gestion des packages
- ☐ Gestion des nœuds
- ☐ Gestion des topics et des services
- ☐ Actions

Travaux pratiques

- Création de packages, nœuds, topics, services et actions.

4 - Développement avec ROS2

- ☐ Aperçu de **CMake** et **ament**
- ☐ Compilation avec **colcon** : commandes essentielles
- ☐ Gestion de l'espace de travail avec **colcon**
- ☐ Création d'un nouveau package
- ☐ Organisation du code d'un package ROS2
- ☐ Implémentation d'un nœud
- ☐ Cycle de vie des nœuds

Travaux pratiques

- Création d'un nœud minimal

5 - Topics, QoS et interfaces personnalisées

- ☐ Publishers et Subscribers
 - Développement d'un éditeur (publisher)



- Développement d'un abonné (subscriber)
- Messages de journalisation (logging)
- ☐ QoS (Quality of Service)
 - Politiques QoS
 - Profils QoS prédéfinis
 - Compatibilité QoS publisher/subscriber
- ☐ Interfaces personnalisées
 - Définition de messages
 - Génération de code C++ pour une interface personnalisée

Travaux pratiques

- Contrôler et déplacer un robot
- Expérimentation des politiques QoS

6 - Services, actions et communication synchrone

- ☐ Services ROS2
 - Communication synchrone avec les services
 - Implémentation d'un serveur et d'un client de service
- ☐ Actions ROS2
 - Actions vs services
 - Implémentation d'un serveur et d'un client d'action
 - Annulation d'une action en cours

Travaux pratiques

- Implémentation d'un serveur de service et d'action

7 - Paramètres, launch files et configuration

- ☐ Paramètres ROS2
 - Système de paramètres
 - Lecture et écriture de paramètres
 - Validation et application dynamique des changements
 - Fichiers de paramètres YAML
- ☐ Présentation des fichiers de lancement (launch files)
 - Lancement de plusieurs nœuds
 - Configuration d'applications ROS2

Travaux pratiques

- Configuration d'une application ROS2 avec launch files et paramètres

8 - Fonctionnalités avancées de ROS2 avec C++

- ☐ Composants (Components) et composition de nœuds
- ☐ Les exécuteurs
- ☐ Callback groups



☐ Plugins avec pluginlib

Travaux pratiques

- Conversion d'un nœud en composant
- Traitement parallèle de topics haute fréquence

9 - Outils avancés de développement et de diagnostic

- ☐ Compilation croisée d'applications ROS2
- ☐ Débogage des applications ROS2
- ☐ Utilisation de fichiers **rosvbag2** pour stocker des messages et les réutiliser
- ☐ Analyse et supervision des échanges

Travaux pratiques

- Compilation croisée d'une application ROS2 pour la cible Raspberry
- Déboguer une application ROS2
- Superviser et rejouer le déplacement d'un robot dans un espace virtuel

10 - Simulation

- Simulation avec **Gazebo**
- Les fichiers **SDF**
- Plugins **Gazebo**
- **ros2_control** : interface entre la simulation et les contrôleurs ROS2
- Lancement de la simulation depuis un launch file

☐ Fichiers URDF

☐ Fichiers Xacro

Travaux pratiques

- Création et simulation d'un objet 3D et interfaçage avec ROS2

11 - Communication distribuée et sécurité dans ROS2

- ☐ Communication sur un réseau
- ☐ Communiquer avec les applications ROS 1
- ☐ Configuration sécurisée

Travaux pratiques

- Faire communiquer un nœud ROS1 avec un nœud ROS2
- Mise en place d'une configuration sécurisée pour communiquer les nœuds sur un réseau



Conception d'interfaces graphiques avec Qt 6 pour développeurs C++

Référence : QTC01			
5 jours (40h) 	Présentiel / Distanciel <i>Modalité</i>	Intermédiaire / Avancé  	2 à 12 participants <i>Format</i>

Prérequis Maîtrise des bases du C++ (syntaxe, classes, pointeurs, références)	Public concerné Développeurs C++ souhaitant créer des IHM desktop/embarqués
---	---

OBJECTIFS DE LA FORMATION

- Maîtriser le modèle objet Qt
- Développer des interfaces graphiques complètes avec Qt Widgets
- Créer des IHM modernes et animées avec Qt Quick et QML
- Exploiter QGraphicsView pour des scènes 2D interactives
- Développer des widgets personnalisés et les intégrer dans Qt Designer
- Utiliser les APIs Qt pour les applications embarquées
- Internationaliser, styler et rendre accessibles les applications Qt
- Déployer une application Qt en production : compilation croisée, packaging, tests automatisés

PROGRAMME DÉTAILLÉ

1 - Fondamentaux Qt

- ☐ Présentation du Framework Qt
 - Historique, architecture générale et modules clés de Qt 6.
 - Avantages de Qt pour le développement multiplateforme
- ☐ Environnement de développement Qt
 - Compilation, plates-formes supportées
 - Aperçu de **CMake**



- Meta-Object Compiler
- Principes de la compilation croisée pour les cibles embarquées (Raspberry Pi)

❑ Modèle objet Qt

- Classe **QObject**
- Gestion parent-enfant de la mémoire
- Métadonnées
- Système de propriétés
- Signaux et slots

Travaux pratiques

- Premier projet Qt 6 avec CMake : configuration, compilation et exécution
- Créer un arbre d'objets, observer la destruction en cascade avec un destructeur tracé
- Signaux et slots : connecter un QSlider à un QLabel avec les trois syntaxes (string, macro, function pointer)

2 - Qt Widgets

❑ Structure d'une application Qt

- Qt Widgets vs Qt Quick
- Gestion des menus
- Principaux composants graphiques, menus et barres d'outils
- Utilisation des boîtes de messages et boîtes de dialogue standards

❑ Widgets de base et gestion des Layouts

- Présentation des widgets essentiels
- Utilisation des gestionnaires de mise en page
- Conception d'interfaces avec Qt Designer et les fichiers **.ui**

❑ Événements et gestion avancée

- Le concept de boucle d'événements Qt.
- Gestion des événements souris, clavier et timers.
- Introduction au Drag & Drop pour des interactions utilisateur avancées.

Travaux pratiques

- Conception d'une fenêtre principale avec menus, barres d'outils et une zone centrale. Utilisation de Qt Designer pour la mise en page initiale.
- Implémentation d'un formulaire de saisie de données avec différents widgets (QLineEdit, QComboBox, QCheckBox) et validation des entrées.
- Implémentation d'une fonctionnalité Drag & Drop simple.

3 - Bases de Qt Quick et QML

❑ Architecture de Qt Quick

❑ Syntaxe QML

- Structure d'un fichier QML, éléments visuels fondamentaux.
- Propriétés, liaisons de propriétés et états.
- Gestion du positionnement et des ancres.



☐ Interaction utilisateur en QML

- Gestion des événements souris et clavier
- Utilisation des Qt Quick Controls 2 pour des composants UI modernes

☐ Animations et transitions

- Création d'animations de propriétés
- Transitions entre états

Travaux pratiques

- Création d'une application QML simple affichant des éléments graphiques de base.
- Implémentation d'interactions utilisateur.
- Ajout d'animations et de transitions pour rendre l'interface plus fluide et réactive.

4 - Intégration C++ / QML

☐ Exposition d'objets C++ à QML

- Enregistrement de types C++ personnalisés dans le moteur QML
- Utilisation de **Q_PROPERTY** pour exposer les propriétés C++ à QML.
- Utilisation de **Q_INVOKABLE** pour appeler des méthodes C++ depuis QML.

☐ Modèles C++ pour les Vues QML

- Communication bidirectionnelle
- Appeler des fonctions QML depuis C++.
- Émettre des signaux C++ capturés par QML.

Travaux pratiques

- Création d'un objet C++ simple et exposition de ses propriétés et méthodes à QML.
- Développement d'une interface QML qui interagit avec l'objet C++
- Implémentation d'un modèle de liste C++ pour afficher une collection de données dans QML.

5 - QGraphicsView : scènes 2D interactives

☐ Architecture **QGraphicsView**

- Le concept de scène, vue et éléments
- Modèle-Vue-Contrôleur (MVC) appliqué à **QGraphicsView**

☐ Création et manipulation d'éléments

☐ Rendu et interactivité

- Rendu avec **QPainter**
- Gestion des événements souris et clavier sur les éléments de la scène
- Animations et transitions dans **QGraphicsView**

Travaux pratiques

- Création d'une application utilisant QGraphicsView d'affichage d'une scène 2D, ajout d'éléments graphiques et implémentation de la sélection, déplacement et redimensionnement.
- Intégration d'un menu contextuel pour manipuler les éléments

6 - Création des widgets personnalisés



☐ API de **QWidget**

- Héritage de **QWidget**
- Redéfinition des événements

☐ Intégration dans Qt Designer

☐ **QWidgets** Composites

Travaux pratiques

- Création d'un widget personnalisé simple, son intégration dans Qt Designer et utilisation dans une application existante

7 - Les APIs Qt avancées

☐ Collections et algorithmes génériques

☐ Processus et communication Inter-Processus (IPC)

☐ Programmation Concurrentielle

- Gestion des threads avec **QThread**
- Mécanismes de synchronisation (mutex, sémaphores, variables de condition).
- Utilisation de Qt Concurrent pour le parallélisme de haut niveau

☐ Communication avec les périphériques (embarqués)

- Acquisition et streaming vidéo à partir d'une caméra avec **openCV**

Travaux pratiques

- Création d'une application multithreadée pour effectuer une tâche en arrière-plan sans bloquer l'interface utilisateur.
- Implémentation d'une communication série avec un périphérique externe
- Acquisition d'un flux vidéo depuis une caméra et affichage dans l'interface Qt

8 - Compléments et Bonnes Pratiques

☐ Création de systèmes d'aide

☐ Internationalisation (i18n) et Localisation (l10n)

☐ Personnalisation de l'Apparence (Look and Feel)

Travaux pratiques

- Internationalisation d'une application existante
- Application de styles personnalisés pour modifier l'apparence de l'interface.

9 - Déploiement et Maintenance

☐ Compilation et packaging

☐ Tests automatisés

☐ Optimisation et débogage

- Outils de profilage pour les applications Qt.
- Techniques de débogage spécifiques à Qt.

Travaux pratiques

- Préparation d'une application Qt pour le déploiement sur une cible Windows/Linux/macOS.
- Création d'un test unitaire simple avec Qt Test.



ENHANCE

TECH