Chapitre 10 – Particularités du C++ embarqué

Le développement embarqué (*embedded C++*) consiste à utiliser C++ sur des systèmes contraints comme :

- microcontrôleurs (ARM Cortex-M, AVR, ESP32…)
- FPGA/SoC (Zynq, Cyclone…)
- systèmes temps réel (RTOS, FreeRTOS…)
- électronique automobile, avionique, médical, IoT

Dans ces environnements, le C++ doit être utilisé **avec précaution**, car les ressources sont limitées.

1. La librairie std en C++ embarqué

La librairie standard (STL + libc++) est souvent **trop lourde** pour un microcontrôleur :

Parts of STL souvent interdites en embarqué

- std::vector, std::string, std::map → peuvent faire des *allocations dynamiques* dangers pour les RAM limitées
- Exceptions (throw) → coût élevé
- RTTI (dynamic_cast, typeid) → augmente la taille du binaire
- Threads <thread> → non disponibles sur microcontrôleurs sans OS

Autorisé / recommandé

- std::array : tableau à taille fixe
- std::span (C++20) : vue sécurisée sur une zone mémoire
- std::optional : alternative légère aux erreurs
- std::pair, std::tuple
- std::chrono pour le timing (si supporté)

Domaine embarqué : choix courants

- Suppression totale des exceptions
- Suppression de la heap (pas de new, pas de malloc)
- Utilisation d'un allocateur mémoire spécialisé
- Remplacement de std::string par des buffers statiques char[]

2. Compilation & Linkage en C++ embarqué

2.1 Chaîne de compilation croisée (Cross-Compilation)

Les microcontrôleurs utilisent un compilateur spécifique :

- ARM : arm-none-eabi-g++
- AVR : avr-g++

- RISC-V : riscv64-unknown-elf-g++

Ce compilateur doit produire :

- du code **sans OS** (bare metal)
- sans dépendances dynamiques
- sans exceptions (option -fno-exceptions)
- sans RTTI (option -fno-rtti)

2.2 Le linkage statique

En embarqué :

- On utilise uniquement du linkage statique**.**
- Pas de DLL, pas de libs dynamiques (pas disponibles).

Exemple de linkage statique :

arm-none-eabi-g++ main.cpp -o firmware.elf -static

2.3 Sections mémoire critiques

Les microcontrôleurs ont des sections mémoire :

- .text → programme en Flash
- .data → données initialisées en RAM
- .bss → données non-initialisées
- .stack → pile
- .heap → allocations dynamiques (souvent interdite)

Le fichier **linker script (.ld)** organise ces sections :

```
SECTIONS {
   .text 0x08000000 : { *(.text) }
   .data 0x20000000 : { *(.data) }
}
```

3. Règles de codage spécifiques en C++ embarqué

En industrie (aéronautique, automobile, défense) : des normes strictes interdisent une partie du C++

Les plus connues :

- MISRA-C++
- AUTOSAR C++ 14
- DO-178C (avionique)
- CERT C++ (sécurité)

Règles typiques

1. Pas d'allocation dynamique

Interdiction de **new, delete, malloc, free.**

➞ Risque de fragmentation mémoire.

2. Pas d'exceptions

Interdit :

throw;

try {} catch() {}

Option à activer : -fno-exceptions

3. Pas de RTTI

Interdit :

dynamic_cast<T*>(ptr);

typeid(obj);

Option :

-fno-rtti

4. Préférer des classes simples

Pas d'héritage complexe, pas d'abstractions inutiles.

5. Structures mémoire déterministes

Utiliser :

- std::array
- static
- petites structures alignées
- buffers circulaires (**ring buffer**)

6. Programmer sans heap : *zero-dynamic-allocation*

Utilisé dans : **FreeRTOS, Zephyr, STM32 HAL, AUTOSAR, etc.**

4. Classe virtuelle dans le C++ embarqué

Les classes virtuelles permettent :

- le **polymorphisme**
- le **recouvrement dynamique** des fonctions

Mais attention…

Elles ont un coût mémoire important car elles nécessitent :

- une table virtuelle (**vtable**)
- un pointeur virtuel par objet (**vptr**)

Exemple:

```
class Base {
public:
   virtual void run() = 0;   // classe abstraite
};

class Motor : public Base {
public:
   void run() override {
      // code moteur
   }
};
```

Pourquoi c'est risqué en embarqué ?

- Chaque objet stocke un vptr
- La vtable est stockée en Flash
- L'appel virtuel est **plus lent** qu'un appel direct
- Le polymorphisme dynamique réduit l'optimisation du compilateur

Recommandations embarquées

Fonctionnement	**Recommandation**
Polymorphisme dynamique	✗ éviter autant que possible
Interfaces via gabarits (CRTP)	✓ recommandé
Classes virtuelles	✓ seulement si justifié
Destructeurs virtuels	✓ si utilisation via pointeurs de base

Alternative recommandée : CRTP

Le Curiously Recurring Template Pattern permet un polymorphisme sans virtual **:**

```
template <typename T>
class Base {
public:
   void run() {
      static_cast<T*>(this)->runImpl();
   }
};

class Motor : public Base<Motor> {
```

```
public:
    void runImpl() {
        // exécution du moteur
    }
};
```

✓ Pas de vtable

✓ Appels optimisés à la compilation

✓ Aucun coût mémoire