Guide d’usage du template CMake – Cube World Alpha (CWSDK)

> **But** : partir d’un template minimal où **seuls** `dllmain.h` **et** `dllmain.cpp` **sont modifiés**, avec un **unique include** `#include "cwsdk.h"` qui agrège tout le SDK. Ce guide explique **où regarder** dans `cwsdk/`, **comment compiler**, **comment accrocher le jeu** et fournit **recettes de code** prêtes à adapter.

---

0) TL;DR

- Édite uniquement `dllmain.h` et `dllmain.cpp`.
- Mets `#include "cwsdk.h"` en haut : ça expose l’ensemble du SDK.
- Crée un **thread** au chargement du DLL (dans `DllMain`) pour éviter le loader lock.
- Utilise `cube::InitGlobals()` (si exposé) puis récupère des pointeurs vers **GameController**, **World**, **Creature** (joueur local), etc.
- Loggue soit via **console** (`OpenConsole()` / `AllocConsole`), soit via `OutputDebugStringA/W`.
- Compile en **Win32 / Release**, copie le `.dll` dans le dossier des mods **Alpha** (loader requis).

---

1) Arborescence utile du template

Le RAR contient (extraits clés) :

- **Racine du template**
- `CMakeLists.txt`, `CMakePresets.json`, `CMakeSettings.json`
- `dllmain.h`, `dllmain.cpp` ← **seuls fichiers à éditer**
- `cwsdk.h` ← **include unique** qui relie tout le SDK
- `README.md`
- `cwsdk/` (API du jeu & utilitaires)
- **Racine** : `cube.h`, `cube.cpp`, `cube_funcs.h/.cpp`, `cube_util.h/.cpp`, `globals.h/.cpp`, `memory_management.h/.cpp`, `Vector3.h/.cpp`, `Matrix4x4.h/.cpp`, `dll_exports.h`, `dx9_stub.h`…
- `cube/` (cœur gameplay/monde/UI) :
  - `GameController.h/.cpp` (boucle de jeu, accès world/UI)
  - `World.h/.cpp`, `Region.h/.cpp`, `Zone.h/.cpp`, `Chunk.h/.cpp`
  - `Creature.h/.cpp` (entités : joueur/PNJ)
  - `ChatWidget.h/.cpp`, `Speech.h/.cpp`, `Sprite*.h/.cpp`, `OptionsWidget.h`
  - `Database.h`, `Field.h/.cpp`
- `plasma/` (widgets/graph UI low-level) : `Widget.h`, `Display.h`, `Node.h`, `Matrix.h`, etc.
- `msvc_bincompat/` : implémentations internes type `vector`, `map`, `string`, `wstring` adaptées binaire MSVC (pour compat exécutable du jeu Alpha).

Astuce : quand tu cherches *« où est X ? »*, pense `cube/` pour gameplay/monde/UI, **racine** `cwsdk/` pour utilitaires & glue, `plasma/` pour bas niveau UI.

---

2) Compilation avec CMake + Visual Studio 2022 (Win32)

1. **Génération** (PowerShell/CMD) :

```
cmake -S . -B build -G "Visual Studio 17 2022" -A Win32 -
DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
```

2. Le résultat attendu : un `.dll` (ex. `EmptyModTemplate.dll`).
3. Place le `.dll` selon ton **loader Alpha** (ex. CubeModLoader) :
4. Généralement : `Cube World Alpha/Mods/MonMod/CubeAlpha-MonMod.dll` (ou nom similaire, selon le loader).
5. Vérifie la convention exacte de ton loader.

Important : en Alpha, les **offsets mémoire changent** selon l'exécutable. Le SDK intègre des `static_assert` pour vérifier les tailles/adresses. Si ça casse, voir §7.

---

3) Points d'entrée d'un mod

- `DllMain` : appelé au chargement/déchargement du DLL.
- Évite de faire du lourd directement ici. Lance un **thread**.
- **Thread mod** :
- Initialise les globals (`cube::InitGlobals()` s'il existe dans ta version).
- Optionnel : affiche une **console** (`OpenConsole()` ou `AllocConsole`) pour logs.
- Boucle principale (ex. `while (running) { ... Sleep(10); }`).

---

4) Accès aux systèmes du jeu (cartographie rapide)

Les noms ci-dessous correspondent aux en-têtes vus dans l'archive ; vérifie la signature exacte dans les headers.

- **Contrôle global** : `cube::GameController` (→ `GameController.h`)
- Accès à la **World**, aux **widgets UI** (chat…), à des callbacks.
- **Monde** : `cube::World` (→ `World.h`)
- `local_player` (souvent), zones, régions, chunks.
- **Entités** : `cube::Creature` (→ `Creature.h`)
- Position, PV, flags, inventaire (selon ce qui est exposé).
- **UI / Chat** : `cube::ChatWidget` (→ `ChatWidget.h`)
- Ajout de messages, commandes custom (à accrocher selon implémentation).
- **Utilitaires** : `cube_funcs.h` , `cube_util.h` , `globals.h`
- Helpers pour retrouver des singletons, conversions, global state.
- **Math** : `Vector3` , `Matrix4x4`

- **Compat MSVC** : `msvc_bincompat::*`
- Préfère **ces types** (vector/map/string) si le SDK te le suggère pour éviter des **ABI mismatch** avec l'exe Alpha.

---

5) Recettes de code (prêtes à coller)

> **NB** : ci-dessous, le code montre des patterns. Selon la version exacte du SDK Alpha, certaines fonctions peuvent s'appeler différemment. Reporte-toi aux headers pour les noms/signatures exacts.

5.1 Ouvrir une console & logger

```
// dllmain.cpp (extrait)
#include "dllmain.h"
#include <windows.h>
#include <thread>
#include <atomic>
#include <cstdio>

static std::atomic<bool> g_running{true};

static void open_console_safely() {
    // Si le SDK expose OpenConsole(), préférer :
    // cube::OpenConsole();
    AllocConsole();
    FILE* out; freopen_s(&out, "CONOUT$", "w", stdout);
    freopen_s(&out, "CONOUT$", "w", stderr);
    printf("[EmptyModTemplate] Console ouverte.\n");
}
```

5.2 Thread du mod depuis `DllMain`

```
// dllmain.h
#pragma once
#include "cwsdk.h"

namespace mod {
    void run();
    void stop();
}

// dllmain.cpp (suite)
static void mod_thread() {
    // Si dispo dans ton SDK Alpha
    // cube::InitGlobals();

    open_console_safely();
    printf("[EmptyModTemplate] Mod démarré.\n");
```

```
    while (g_running) {
        // Exemples de triggers clavier pour quick tests
        if (GetAsyncKeyState(VK_F6) & 1) {
            OutputDebugStringA("[EmptyModTemplate] F6 pressed\n");
        }
        Sleep(10);
    }

    printf("[EmptyModTemplate] Mod stoppé.\n");
}

void mod::run()  { std::thread(mod_thread).detach(); }
void mod::stop() { g_running = false; }

// Point d'entrée DLL
BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID) {
    if (reason == DLL_PROCESS_ATTACH) {
        DisableThreadLibraryCalls(hModule);
        mod::run();
    } else if (reason == DLL_PROCESS_DETACH) {
        mod::stop();
    }
    return TRUE;
}
```

5.3 Récupérer GameController / World / joueur

```
// Schéma générique – adapte aux helpers réels exposés par ton SDK
if (auto* gc = cube::GetGameController()) {           // souvent dispo via
cube_funcs/globals
    if (auto* world = gc->world) {
        if (auto* player = world->local_player) {
            // player est un cube::Creature*
        }
    }
}
```

Si `GetGameController()` n'existe pas tel quel, fouille `cube_funcs.h`, `globals.h` et `GameController.h`.

5.4 Lire la position du joueur (Vector3)

```
if (auto* gc = cube::GetGameController()) {
    if (auto* w = gc->world) {
        if (auto* p = w->local_player) {
            // Selon Creature.h : nom exact du champ (position / pos /
entity_data.pos ...)
            const auto pos = p->position; // à confirmer dans le header
```

```
            printf("Player pos: %.2f %.2f %.2f\n", pos.x, pos.y, pos.z);
        }
    }
}
```

5.5 Afficher un message dans le chat

```
if (auto* gc = cube::GetGameController()) {
    if (auto* chat = gc->chat_widget) { // ou accès via UI/Widget manager
        // Selon ChatWidget.h, il y a souvent une méthode type
AddMessage(wstring)
        // chat->AddMessage(L"[Mod] Hello from EmptyModTemplate!");
    }
}
```

5.6 Commande « /hello » (pattern)

> Deux approches courantes : 1) **Hook** d'une méthode du ChatWidget (ex. `OnSend`) et interception de la ligne saisie. 2) **Polling** léger : surveiller l'historique ou une variable, selon ce que le SDK expose.

Pseudo-code (hook vtable – avancé):

```
// 1) Obtenir l'adresse de la méthode d'origine (via vtable ou symbole résolu
dans cwsdk)
// 2) Installer un trampoline (MinHook/polyhook) – si ton template/loader le
propose
// 3) Dans le detour, si la ligne commence par "/hello", consommer et agir;
sinon, appeler l'originale.
```

> Comme le hook exact dépend des signatures, pars des noms présents dans `ChatWidget.h/.cpp`.

5.7 Tick par frame (pattern sans hook hard)

```
// Dans ton thread : tu peux faire un tick soft à 100 Hz
while (g_running) {
    // ... logique périodique ici ...
    Sleep(10);
}
```

> Pour un vrai **per-frame**, on accroche la **game loop** du `GameController` (hook vtable) – voir `GameController.h/.cpp`.

---

6) Conseils d'ABI/compat

- **N'utilise pas arbitrairement** `std::vector` **/** `std::string` si le SDK fournit `msvc_bincompat::vector/string`. Mélanger des ABI peut crasher.
- Respecte les `assert_size` du SDK (ex. `assert_size<cube::Creature, 0x????>()`) : ils vérifient que tes entités correspondent au binaire cible.
- Compile en **/MD** ou **/MT** selon ce que le CMake du template impose — ne change pas sans raison.

---

7) Problèmes fréquents & diagnostic

7.1 `static_assert failed: 'cube::X != 0xYYYY'`

- Ta version de l'exécutable **Alpha** ne correspond pas aux offsets assumés par le SDK.
- **Solutions** :
- Prendre le **SDK Alpha exact** ciblant ton `Cube.exe` (même build/CRC).
- Ou **re-scannér** et ajuster les offsets (Ghidra/IDA) → long.

7.2 Champs introuvables (ex. `parent_owner` / `hostility_flags`)

- Le champ **n'existe pas** (ou **nom différent**) dans **ta** version d'Alpha.
- Ouvre `Creature.h`, vérifie **les noms réels**; pars des **accès utilisés** dans `Creature.cpp`.

7.3 LNK2019 / symboles non résolus

- Tu as référencé une fonction que **ton** template ne build pas (ex. `start_probe_async`).
- Supprime l'appel, ou **ajoute** la source correspondante dans `CMakeLists.txt`.

7.4 `strcasestr` introuvable (MSVC)

- Remplacer par `_stricmp` / `_wcsicmp` selon le type.

7.5 Rien dans les logs / console

- Assure-toi d'appeler `AllocConsole()` **ou** `OpenConsole()` **avant** de `printf`.
- Ou bien utilise `OutputDebugStringA/W` et lis via **DebugView**.

---

8) Packaging & chargement

- Nom du DLL : souvent `CubeAlpha-<Nom>.dll` (selon loader).
- Dossier : `Mods/<Nom>/`.
- Certains loaders lisent un `manifest.json` – si ton loader en demande un, ajoute-le.

---

9) Check-list « avant de builder »

- [ ] `#include "cwsdk.h"` **unique** dans `dllmain.h` (puis inclus par `dllmain.cpp`).
- [ ] Aucune dépendance exotique non prévue par le template.

- [ ] Générateur VS2022, plateforme **Win32**, config **Release**.
- [ ] Le loader est compatible **Alpha** et actif.
- [ ] Teste une touche rapide (F6/F7) pour vérifier que ton thread tourne.

---

10) Exemple complet minimal (coller tel quel et adapter)

`dllmain.h`

```
#pragma once
#include "cwsdk.h"

namespace mod {
    void run();
    void stop();
}
```

`dllmain.cpp`

```
#include "dllmain.h"
#include <windows.h>
#include <thread>
#include <atomic>
#include <cstdio>

static std::atomic<bool> g_running{true};

static void open_console_safely() {
    // Préfère cube::OpenConsole() si disponible dans ton SDK
    AllocConsole();
    FILE* out; freopen_s(&out, "CONOUT$", "w", stdout);
    freopen_s(&out, "CONOUT$", "w", stderr);
}

static void mod_thread() {
    // Si exposé par ton SDK
    // cube::InitGlobals();

    open_console_safely();
    printf("[EmptyModTemplate] Hello Cube!\n");

    while (g_running) {
        if (GetAsyncKeyState(VK_F6) & 1) {
            printf("F6 pressed\n");
            // Démo d'accès – adapte au header réel :
            // if (auto* gc = cube::GetGameController()) {
            //     if (auto* world = gc->world) {
            //         if (auto* p = world->local_player) {
```

```
            //             auto pos = p->position; // vérifie le nom exact
            //             printf("Pos: %.2f %.2f %.2f\n", pos.x, pos.y,
pos.z);
            //         }
            //         if (auto* chat = gc->chat_widget) {
            //             // chat->AddMessage(L"[Mod] Salut !"); //
signature à confirmer
            //         }
            //     }
            // }
        }
        Sleep(10);
    }
}

void mod::run()  { std::thread(mod_thread).detach(); }
void mod::stop() { g_running = false; }

BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID) {
    if (reason == DLL_PROCESS_ATTACH) {
        DisableThreadLibraryCalls(hModule);
        mod::run();
    } else if (reason == DLL_PROCESS_DETACH) {
        mod::stop();
    }
    return TRUE;
}
```

---

11) Aller plus loin

- **Hook par vtable** de `GameController` / `ChatWidget` pour tick précis/commandes chat.
- **Spawner** des entités : inspecter `World` / `Creature` / `Database`.
- **UI custom** : via `plasma::*` (bas niveau) ou en réutilisant des widgets `cube::*`.
- **Offsets** : si tu veux stabiliser pour *ta* build Alpha, documente les sizes/offsets en t'aidant des `static_assert` existants.

Besoin d'exemples « réels » ?

Ce guide suffit pour démarrer. **Des exemples de mods existants** (mêmes headers/signatures que *ton* SDK Alpha) sont très utiles pour : - le *hook* du chat, - les patterns d'accès à `GameController` / `World`, - les conventions de packaging du loader.

Tu peux m'en partager : je proposerai des **extraits ciblés** adaptés à ce template.

Annexes — Patterns des mods coremaze & Portage vers le template `cwsdk.h` unique

> Cette annexe synthétise les 15 dépôts que tu m'as donnés, explique **ce que chaque mod modifie**, **où intervenir dans le SDK Alpha**, et donne un **plan de portage** vers ton template (où seul `dllmain.h/.cpp` inclut `cwsdk.h`).
>
> ⚠️ Beaucoup de ces mods ciblent l'**Alpha** (client ou serveur) et s'appuient sur le **Mod Launcher** (ou **Server Mod Launcher**) pour les callbacks. Attends-toi à **des offsets variables** d'une build à l'autre : garde toujours un œil sur les `static_assert` du CWSDK et re-scanne si nécessaire.

A. Rappels d'architecture

- **Client Alpha** : logique jeu (GameController/World/Creature/UI), persistance locale (SQLite), rendu DX9, chronos système 32-bit.
- **Serveur Alpha** : logique réseau (packets, GUID, états de créatures), persistance (DB), callbacks de connexion/chat.
- **Launcher (client/serveur)** : charge `Mods/*.dll` ou `Server_Mods/*.dll`, expose des **callbacks d'événements** (tick, chat, join/leave, packet, etc.). Dans ce template, on peut **émuler** ces hooks via :
- un **thread** maison (polling léger) + accès aux singletons exposés par `cwsdk.h` ;
- ou un **detour** ciblé (vtable / fonction libre) si l'événement n'est pas observable autrement.

B. Cookbook de portage par mod

1) Adaption Rebalance — *coût d'adaptation* (client)

But : rendre le coût d'adaptation raisonnable. **Zones** : boutique/forgeron, calcul du coût d'adaptation d'un item. **Types probables** : `cube::World`, `cube::Creature` (joueur), DB/Items, éventuellement un helper coût. **Approche** : 1. Localiser la **fonction de calcul** (recherche dans headers/ implémentations : « adapt », « cost », « upgrade »). 2. Installer un **detour** sur la fonction (ou wrapper le path UI qui l'appelle) pour **remapper la formule**. 3. Conserver les bornes (min/max) et types entiers attendus par l'UI. **Test** : item low/high level, comparer coût vanilla vs mod, vérifier affichage et débit d'or.

2) Bad Item ID Crash Fix — *drop d'ID invalide* (client)

But : éviter un crash quand un item avec ID invalide est lâché. **Zones** : `Creature::DropItem` / gestion d'inventaire / validation DB. **Approche** : 1. Avant création de l'item au sol, **valider l'ID** (présent dans DB / table des items). 2. Si invalide : ignorer l'item ou substituer un **fallback** neutre. 3. Journaliser via console pour diagnostiquer. **Test** : forcer un ID hors bornes ; s'assurer qu'il n'y a ni crash ni duplication.

3) Multithreaded Terrain Generation — *génération de chunks* (client)

But : paralléliser la génération terrain. **Zones** : `cube::World` / `Chunk` / `Zone` / worker pool. **Approche** : 1. Abstraire la **file de jobs** `ChunkGenJob` (coord, seed, biome…). 2. **Thread pool** (N

threads) qui calcule la géométrie **hors thread principal**. 3. Synchroniser **l'insertion** (meshes, heightmaps) avec **mutex/critical section** au moment d'attacher au monde. **Test** : fly rapide sur bords de régions ; vérifier pas de data-race (artefacts / crash intermittent).

4) Unlimited Item Stacks — *suppression limite de stack* (client)

But : lever la limite de pile pour tous les items. **Zones** : `ItemStack` / inventaire / sérialisation. **Approche** : 1. Localiser `GetMaxStack()` / `max_stack` (ou clamp à 99/50). 2. **Detour** pour retourner une valeur élevée (ex. 9999) ou **neutraliser le clamp**. 3. Vérifier **sauvegarde/chargement** (types : `uint8` vs `uint16/32`). Si la sauvegarde tronque, patcher la (dé)serialisation. **Test** : cumuler >255, redémarrer le jeu, confirmer persistance & pas de corruption.

5) Moving Clouds — *nuages animés* (client)

But : translater/faire dériver les nuages avec le temps. **Zones** : structure « Cloud » (ou bloc type), tick/world time. **Approche** : 1. Accéder au **conteneur des nuages** (liste/voxels dédiés). 2. À chaque tick, appliquer un **offset fonction du temps** (ex. sin/cos ou vitesse constante) sur leur position de rendu. 3. Préserver l'intangibilité (no-collision). **Test** : time-warp, météo, visibilité longue distance.

6) System Time Overflow Fix — *overflow 32-bit* (client & serveur)

But : corriger les bugs liés au wrap de `GetTickCount`/chrono 32-bit (connexion, particules inversées…). **Approche** : 1. Centraliser le **temps système** dans un wrapper 64-bit (QPC ou `GetTickCount64`). 2. **Detour** les points qui lisent la version 32-bit pour renvoyer **un delta monotone** (sans wrap). 3. Revoir les calculs qui font des différences de tick (cast/overflow signés). **Test** : run prolongé (>49.7 jours simulés), réseaux, effets particules.

7) Server Mod Launcher — *injection serveur* (serveur)

But : charger `Server_Mods/*.dll`, exposer callbacks basiques. **Approche côté mod** : 1. `DllMain` → thread → souscrire aux **événements** (join/leave, chat, tick, packet). 2. Pour notre template (sans header launcher), **émuler** via : polling sur la liste des joueurs + hooks ponctuels (ex. handler chat) si nécessaire. **Test** : connexion de clients multiples, affichage liste mods au démarrage.

8) GUID Fix — *téléportation globale due aux GUID* (serveur)

But : corriger collisions/overflows de GUID réseau. **Zones** : générateur de GUID / table de mapping entités. **Approche** : 1. Remplacer RNG/compteur 32-bit par **64-bit** (ou vérifier **unicité** avant assignation). 2. Assurer la **persistance**/recyclage propre à la déconnexion/despawn. **Test** : spawn/despawn massifs, sessions longues, aucun « tp vers coin map ».

9) Server Chat Crash Fix — *crash au chat* (serveur)

But : éviter crash lors de messages spécifiques. **Approche** : 1. **Sanitiser** l'entrée (taille, encodage, caractères de contrôle). 2. Envoi côté serveur en **taille bornée** ; éviter format strings dangereuses. **Test** : spam, messages très longs/UTF-8, caractères spéciaux.

10) Server Ghost Damage Fix — *morts infinies* (serveur)

But : corriger application de dégâts sur entités « fantômes ». **Approche** : 1. Dans la pipeline dégâts, **valider** que l'entité cible est **valide/en vie/same world**. 2. Ignorer les events anciens/répliqués après despawn. **Test** : PvP/PvE intensif, changements de zone, respawn.

11) Server MOTD — *Message du jour* (serveur)

But : envoyer un message à la connexion. **Approche** : 1. Hook **OnPlayerJoin** → push message dans le **chat serveur**. 2. Prévoir **config** (fichier `.ini` / `.json`) pour le texte MOTD. **Test** : multiples connexions, accents/UTF-8.

12) Multiplayer Pet Leveling Fix — *progression pet non sauvegardée* (serveur)

But : assurer la bonne persistance de l'XP pet. **Approche** : 1. À chaque tick ou event XP, **flusher** dans la DB serveur (ou flag « dirty » + batch périodique). 2. Vérifier **(dé)serialisation** et **réplication** au client. **Test** : changer de zone/serveur, crash-recovery → niveau OK.

13) Stutter Fix — *SQL limité par disque* (client)

But : réduire les micro-freeze dus aux accès disques SQLite. **Approche** : 1. À l'ouverture DB : exécuter `PRAGMA journal_mode=WAL; PRAGMA synchronous=NORMAL/OFF; PRAGMA temp_store=MEMORY;` (selon risque toléré). 2. Mettre en cache certaines requêtes ou **déplacer I/O** hors du thread principal. **Test** : profilage frame-time avec/ sans PRAGMA.

14) Efficient Crafting — *UI craft plus fluide* (client)

But : réduire le coût CPU/UI du crafting. **Approche** : 1. **Memoize** les calculs (recettes valides, coûts) par item/quantité. 2. Debounce les rafraîchissements UI ; batch des mises à jour. **Test** : lots de matériaux, scroll rapide, aucune saccade.

15) EXP Buff — *formule d'XP adoucie* (client)

But : modifier la courbe XP. **Approche** : 1. Localiser la fonction « XP needed for level ». 2. Remplacer par une **fonction continue** plus douce (ex. poly ou exponentielle à base réduite), garder type & bornes. **Test** : grapher vanilla vs mod (outil rapide Python) ; valider progression multi-sessions.

16) Building Mod — *construction libre* (client)

But : permettre de placer/retirer des voxels. **Zones** : `World` / `Chunk`, picking 3D, MAJ meshes/chunks. **Approche** : 1. Raycast → déterminer voxel cible. 2. Opérations `set_block(x,y,z, id)` avec **rebuild mesh** minimal (greedy ou chunk dirty flag). 3. Persistance : écrire dans DB/region file custom. **Test** : sauvegarde/recharge, bordures de chunk, perf.

C. Gabarits de code (adaptables au template)

C.1 Démarrage + console + boucle

Voir section 10 du guide principal : `DllMain` → thread → console → boucle `Sleep(10)`.

C.2 Lecture joueur/monde (pattern générique)

```
if (auto* gc = cube::GetGameController()) {
    if (auto* w = gc->world) {
        if (auto* p = w->local_player) {
            // TODO: adapter noms de champs depuis Creature.h
            auto pos = p->position;
        }
    }
}
```

C.3 Patch d'un clamp « max stack » (pseudo-detour)

```
// Attention: illustre le principe; implémente le vrai hook selon ton SDK/
loader.
int __stdcall MyGetMaxStack(/*... item args ...*/) {
    return 9999; // ou lis depuis config
}
// InstallHook(OriginalGetMaxStack, MyGetMaxStack);
```

C.4 PRAGMA SQLite au démarrage (stutter)

```
// Après ouverture DB (exposée via cwsdk)
// db->Exec("PRAGMA journal_mode=WAL;");
// db->Exec("PRAGMA synchronous=NORMAL;");
// db->Exec("PRAGMA temp_store=MEMORY;");
```

C.5 Wrapper temps 64-bit (overflow)

```
static inline uint64_t NowTicks() {
    LARGE_INTEGER q; QueryPerformanceCounter(&q); return
(uint64_t)q.QuadPart;
}
```

D. Stratégie de test & compat Alpha

- **Sanity** : pas de crash au boot, console OK, F6 trigger OK.
- **Fonctionnel** : pour chaque mod, définir 3 cas (min/typique/max) + persistance.
- **Perf** : micro-freeze < 5 ms en terrain/craft ; pas de leak.
- **Réseau** (serveur) : 3+ clients, spam chat, tp, pet-xp ; pas de désync.
- **Offsets** : si `static_assert` casse, re-scanner ou revenir à la build SDK correspondante.

E. Portage pas-à-pas (recommandé)

1) **Unlimited Stacks** → easy win, visible, faible risque. 2) **EXP Buff** → formule isolée, rapide à valider. 3) **Stutter Fix** → gains confort immédiats. 4) **System Time Overflow** → robustesse long terme. 5) **Server MOTD** → pipeline serveur simple. 6) **Ghost Damage / Chat Crash / GUID** → corrections réseau. 7) **Multithreaded Terrain** → itérer prudemment (profil/perf).

> Si tu me donnes un header précis (signature réelle) pour **Stack max** ou **Formule XP**, je te fournis le **detour complet** prêt à coller dans `dllmain.cpp`.

Verrou EmptyModTemplate — respecter strictement le CWSDK du ZIP

Tu as raison : on **s'aligne exclusivement** sur le CWSDK et la config fournis dans **EmptyModTemplate** (et non sur les offsets/noms des mods indépendants).

Règles d'or

1) **Fichiers modifiables** : uniquement `dllmain.h` et `dllmain.cpp`. 2) **Include unique** : `#include "cwsdk.h"` dans `dllmain.h` → **aucun** autre include du SDK directement depuis `dllmain.cpp`. 3) **Aucune dépendance offsets externes** (pas de hard-addr, pas de taille codée en dur) : on ne touche qu'aux **API/types** exposés par **ce** `cwsdk.h`. 4) **Types bincompat** : si le `cwsdk` fourni expose `msvc_bincompat::vector/string/map`, **les utiliser** (éviter `std::` si le header du ZIP ne le permet pas). 5) **Flags de build** : conserver **Win32**, **Release**, l'ABI, la version C++ et les options telles que définies dans le **CMake** du ZIP. 6) **Assertions de taille** : ne pas supprimer/altérer les `assert_size` et `static_assert` présents ; s'appuyer dessus pour détecter les divergences. 7) **Hooks/Detours** : si nécessaires, ils se font **via les symboles/fonctions exposés** par ce `cwsdk.h`. Aucun scan mémoire. 8) **Logs** : préférer `OpenConsole()` (s'il existe dans le ZIP) ou `AllocConsole()` + `OutputDebugString`. 9) **Threading** : démarrage dans un **thread détaché** (pas de logique lourde dans `DllMain`). 10) **Tests** : sanity (boot, F6), fonctionnels (3 cas/feature), perf, persistance.

Patron "capteurs" sans offsets (copier-coller dans `dllmain.cpp`)

> Ce bloc **ne suppose aucun offset**. Les deux *hooks* optionnels sont protégés par des `#ifdef` imaginés pour **ce** SDK ; si ces macros n'existent pas, le code compile toujours (il se contente de logger).

```
// dllmain.h
#pragma once
#include "cwsdk.h"
namespace mod { void run(); void stop(); }
```

```
// dllmain.cpp
#include "dllmain.h"
#include <windows.h>
#include <thread>
#include <atomic>
#include <cstdio>

static std::atomic<bool> g_running{true};

static void open_console_safely() {
    // Préfère cube::OpenConsole() si le ZIP l'expose
    AllocConsole();
    FILE* f; freopen_s(&f, "CONOUT$", "w", stdout); freopen_s(&f, "CONOUT$",
"w", stderr);
    OutputDebugStringA("[EmptyModTemplate] Console ready
");
}

static cube::GameController* try_get_gc() {
    // Ces chemins sont **exclusivement** basés sur le ZIP.
    // Chemin 1 : helper global (exposé par cwsdk du ZIP)
#ifdef CWSKD_HAS_GET_GAME_CONTROLLER
    if (auto* gc = cube::GetGameController()) return gc;
#endif
    // Chemin 2 : global/Singleton exposé (noms à confirmer dans le ZIP)
#ifdef CWSKD_HAS_GLOBALS_GAME_CONTROLLER
    return cube::globals::g_GameController; // si fourni par le ZIP
#endif
    return nullptr;
}

static void tick_once() {
    if (auto* gc = try_get_gc()) {
        // Monde
        auto* w = gc->world; // **nom exact à confirmer dans le ZIP**
        if (w) {
            // Joueur local
            auto* p = w->local_player; // **nom exact à confirmer**
            if (p) {
                // Nom de champ position selon le ZIP (position/pos/
physics.pos ...)
#ifdef CWSKD_CREATURE_HAS_POSITION
                auto pos = p->position;
                printf("Player pos: %.2f %.2f %.2f
", pos.x, pos.y, pos.z);
#endif
            }
        }
        // Chat UI (si exposé par le ZIP)
#ifdef CWSKD_GAMECONTROLLER_HAS_CHATWIDGET
```

```
        if (auto* chat = gc->chat_widget) {
#ifdef CWSKD_CHATWIDGET_HAS_ADDMESSAGE
            // chat->AddMessage(L"[Mod] Hello from template");
#endif
        }
#endif
    }
}

static void mod_thread() {
    open_console_safely();
#ifdef CWSKD_HAS_INIT_GLOBALS
    cube::InitGlobals(); // seulement si présent dans le ZIP
#endif
    printf("[EmptyModTemplate] started.
");
    while (g_running) {
        if (GetAsyncKeyState(VK_F6) & 1) {
            tick_once();
        }
        Sleep(10);
    }
    printf("[EmptyModTemplate] stopped.
");
}

void mod::run()  { std::thread(mod_thread).detach(); }
void mod::stop() { g_running = false; }

BOOL APIENTRY DllMain(HMODULE hModule, DWORD reason, LPVOID) {
    if (reason == DLL_PROCESS_ATTACH) { DisableThreadLibraryCalls(hModule);
mod::run(); }
    else if (reason == DLL_PROCESS_DETACH) { mod::stop(); }
    return TRUE;
}
```

Comment utiliser les macros `CWSKD_*` **?**
— Si le ZIP expose déjà des macros de présence, réutilise-les.
— Sinon, laisse ces `#ifdef` **commentés** : le code restera neutre (il ne fait que logger), sans référence à des offsets.

“Packs” prêts à porter (sans dépendre d’offsets externes)

1) Stutter Fix (client / DB)

- Si le ZIP expose un handle DB via `cube::Database*` ou `world->db`, ajoute (au moment où la DB est ouverte) :

```
// db->Exec("PRAGMA journal_mode=WAL;");
// db->Exec("PRAGMA synchronous=NORMAL;");
// db->Exec("PRAGMA temp_store=MEMORY;");
```

- Si aucun accès DB de haut niveau n'est exposé : **ne pas bricoler** → rester aligné ZIP.

2) System Time 64-bit

- Fournir un **wrapper monotone** et l'utiliser **uniquement** dans ton code (pas de patch global) :

```
static inline uint64_t NowQPC() { LARGE_INTEGER q;
QueryPerformanceCounter(&q); return (uint64_t)q.QuadPart; }
```

3) MOTD serveur (si callbacks exposés)

- Utiliser **les callbacks de join** fournis par le ZIP ; sinon, **ne rien supposer** → se limiter aux logs.

4) Unlimited Stacks / EXP Buff

- Implémenter **uniquement** si le ZIP expose la **fonction** (ex. `GetMaxStack()` ou `GetXPForLevel()`), par **detour symbolique** fourni par le ZIP.
- **Pas d'adresses brutes.** Si la fonction n'existe pas en clair dans le `cwsdk.h` du ZIP, on **s'abstient**.

Stratégie de validation (100% ZIP)

1. Compiler en l'état (aucun hook → sanity).
2. Activer progressivement les blocs protégés par `#ifdef` **qui existent réellement dans le ZIP**.
3. Ajouter des `static_assert(sizeof(...))` **uniquement** si le ZIP expose les tailles attendues.
4. Pour chaque feature, écrire 3 tests (min/typique/max) + reboot jeu (persistance).

Avec ceci, on reste **strictement conforme** au `cwsdk.h` de ton **EmptyModTemplate**, sans fuite vers offsets ou headers d'anciens dépôts.