شرح تفصيلي لملف main.cpp

نظرة عامة

ملف **main.cpp** هو **نقطة البداية** للبرنامج بالكامل. هذا الملف هو المايسترو الذي ينسق بين جميع الأجزاء:

- **Utils**: تحميل الألغاز من الملف
- **Library**: بناء هيكل المكتبة والكتب
- **Game**: تشغيل اللعبة والتفاعل مع اللاعب

البنية العامة

```
main.cpp
├── تحميل الألغاز من الملف
├── بناء المكتبة
├── التحقق من صحة البناء
└── تشغيل اللعبة
```

الشرح التفصيلي

1. المكتبات المستخدمة

```cpp
#include "Library.h"  // فئة المكتبة (إدارة الكتب والبنية)
#include "Game.h"     // فئة اللعبة (منطق اللعب)
#include "Utils.h"    // الأدوات المساعدة (تحميل الألغاز)
#include <iostream>   // الإدخال والإخراج
using namespace std;
```

ملاحظة مهمة: ترتيب الـ includes منطقي

- أولاً: الملفات الخاصة بالمشروع ("...")
- ثانياً: مكتبات C++ القياسية (<...>)

---

2. دالة main() - نقطة البداية

```cpp
```

```cpp
int main() {
    // ... كود البرنامج
    return 0;  // نجاح البرنامج
}
```

القيمة المُرجعة:

- `0` → البرنامج نجح وانتهى بشكل طبيعي
- `1` (أو أي رقم آخر) → حدث خطأ

الخطوة 1: تحميل الألغاز من الملف

أ. رسالة التحميل

```cpp
cout << "Loading puzzles from file..." << endl;
```

الغرض (User Experience): إعلام المستخدم بما يحدث

ب. تحديد مسار الملف

```cpp
string puzzleFile = "data/puzzles.txt";
```

تحليل المسار:

```
Project/
├── main.cpp
├── data/
│   └── puzzles.txt  ← الملف المطلوب
```

- `data/` مجلد فرعي
- `puzzles.txt` الملف الذي يحتوي على الألغاز

لماذا مجلد منفصل؟

- تنظيم أفضل للملفات
- فصل الكود عن البيانات

- سهولة التعديل بدون إعادة ترجمة

ج. تحميل الألغاز

```cpp
vector<Puzzle> puzzles = loadPuzzlesFromFile(puzzleFile);
```

ماذا يحدث هنا؟

1. استدعاء دالة `loadPuzzlesFromFile()` من Utils.cpp
2. الدالة تفتح الملف وتقرأه سطراً سطراً
3. تستخرج الألغاز وتخزنها في `vector<Puzzle>`
4. تُرجع المصفوفة الديناميكية

مثال على محتوى الملف:

```
[General]
Riddle: What has keys but no locks? | Answer: keyboard
Riddle: I'm tall when young, short when old | Answer: candle

[Math]
Riddle: What is 2 + 2? | Answer: 4
```

النتيجة:

```cpp
puzzles[0] = {riddle: "What has keys...", answer: "keyboard", category: "General"}
puzzles[1] = {riddle: "I'm tall when...", answer: "candle", category: "General"}
puzzles[2] = {riddle: "What is 2 + 2?", answer: "4", category: "Math"}
```

د. التحقق من نجاح التحميل

```cpp
```

```cpp
if (puzzles.empty()) {
    cout << "\nError: No puzzles found!" << endl;
    cout << "Make sure 'data/puzzles.txt' exists and has puzzles." << endl;
    return 1;  // خروج من البرنامج بخطأ
}
```

السيناريوهات المحتملة:

✅ الملف موجود ويحتوي على ألغاز:

```cpp
puzzles.size() > 0
puzzles.empty() → false
// نكمل البرنامج عادي
```

❌ الملف غير موجود:

```cpp
puzzles.size() = 0
puzzles.empty() → true
// نطبع رسالة خطأ ونخرج
```

❌ الملف موجود لكن فارغ أو صيغته خاطئة:

```cpp
puzzles.size() = 0
puzzles.empty() → true
// نطبع رسالة خطأ ونخرج
```

لماذا `return 1`؟

- `return 0` يعني نجاح
- `return 1` يعني فشل (أو أي رقم غير صفري)
- نظام التشغيل يستطيع معرفة أن البرنامج فشل

هـ. رسالة النجاح

```cpp
```

```
cout << "✓ Loaded " << puzzles.size() << " puzzles!\n" << endl;
```

مثال على الخرج:

```
Loading puzzles from file...
Loaded 50 puzzles successfully!
✓ Loaded 50 puzzles!
```

الخطوة 2: بناء المكتبة

```
cpp

Library library(puzzles);
```

ماذا يحدث في هذا السطر؟

1. استدعاء Constructor الخاص بـ Library:

```
cpp

Library::Library(vector<Puzzle>& puzzles) {
    // بناء 4 كتب مدخل (entrance)
    // بناء كتب المراحل (stage)
    // بناء كتب الممرات (path)
    // بناء الكتاب النهائي (final)
    // توزيع الألغاز على الكتب
    // ربط الكتب ببعضها (next1, next2)
}
```

2. هيكل المكتبة الناتج:

```
    [E1]  [E2]  [E3]  [E4]
     ↓     ↓     ↓     ↓
    Stage Books (متعددة)
     ↓     ↓     ↓     ↓
    Path Books (متعددة)
     ↓     ↓     ↓     ↓
       [FINAL BOOK]
```

3. كل كتاب يحصل على:

- عدد معين من الألغاز

- مستوى صعوبة (EASY أو HARD)
- نوع (ENTRANCE, STAGE, PATH, FINAL)
- روابط للكتب التالية (next1, next2)

الخطوة 3: التحقق من صحة البناء

```cpp
if (!library.checkIfValid()) {
    cout << "\nError: Library structure is broken!" << endl;
    return 1;
}
```

ما الذي يتم التحقق منه؟

1. الكتب موجودة:

```cpp
- هل توجد 4 بوابات دخول؟
- هل يوجد كتاب نهائي؟
```

2. الروابط صحيحة:

```cpp
- هل كل كتاب (غير النهائي) يشير لكتاب تالي؟
- هل الروابط لا تشير إلى nullptr؟
```

3. المسارات تؤدي للنهاية:

```cpp
- هل يمكن الوصول للكتاب النهائي من أي بوابة؟
- هل هناك كتب معزولة (orphaned)؟
```

السيناريوهات:

✅ **البنية صحيحة:**

```cpp
checkIfValid() → true
// نكمل البرنامج
```

البنية معطوبة ❌:

```cpp
checkIfValid() → false
// نطبع خطأ ونخرج
```

أسباب محتملة للفشل:

- خطأ برمجي في بناء المكتبة
- ذاكرة غير كافية
- منطق خاطئ في الروابط

رسالة النجاح

```cpp
cout << "✓ Library structure is valid!\n" << endl;
```

الخرج المتوقع:

```
✓ Loaded 50 puzzles!
✓ Library structure is valid!
```

الخطوة 4: تشغيل اللعبة

```cpp
Game game(&library);
game.start();
```

أ. إنشاء كائن اللعبة

```cpp
Game game(&library);
```

ماذا يحدث؟

1. إنشاء كائن `game` من فئة `Game`

2. تمرير مؤشر للمكتبة (`&library`)
3. استدعاء Constructor:

```cpp
Game::Game(Library* lib) {
    library = lib;
    currentBook = nullptr;
}
```

لماذا مؤشر؟

- لتجنب نسخ المكتبة بالكامل (مكلف جداً)
- المكتبة كبيرة وتحتوي على كتب كثيرة
- نحتاج فقط للإشارة إليها

ب. بدء اللعبة

```cpp
game.start();
```

ماذا يحدث داخل start()؟

```cpp
```

```
void Game::start() {
    showIntro();          // عرض المقدمة
    chooseEntrance();     // اختيار بوابة

    while (currentBook != nullptr) {
        solvePuzzles();       // حل الألغاز

        if (فشل) {
            return;           // نهاية اللعبة
        }

        if (وصل للنهاية) {
            break;            // !فوز
        }

        chooseNextPath();     // الانتقال للكتاب التالي
    }
}
```

دورة اللعبة الكاملة:

```
المقدمة
  ↓
اختيار بوابة (1-4)
  ↓
حل ألغاز الكتاب
  ↓
(لا)—— ؟نجح→ Game Over
  ↓ (نعم)
(نعم)—— ؟النهاية→ Congratulations!
  ↓ (لا)
اختيار المسار التالي
  ↓
(عودة للحلقة)
```

الخطوة 5: النهاية

```
cpp

return 0;
```

متى نصل هنا؟

- بعد انتهاء `game.start()`
- سواء فاز اللاعب أو خسر
- البرنامج ينتهي بنجاح

تدفق البرنامج الكامل

✅ سيناريو ناجح

```
1. main() يبدأ
   ↓
2. تحميل الألغاز من data/puzzles.txt
   → نجح: 50 لغز
   ↓
3. بناء المكتبة
   → بنية الكتب والروابط جاهزة
   ↓
4. التحقق من الصحة
   → checkIfValid() = true
   ↓
5. إنشاء كائن اللعبة
   ↓
6. game.start()
   ├─ showIntro()
   ├─ chooseEntrance()
   ├─ حلقة اللعب
   │  ├─ solvePuzzles()
   │  └─ chooseNextPath()
   └─ نهاية اللعبة
   ↓
7. return 0 (نجاح)
```

❌ سيناريو فاشل (ملف الألغاز مفقود)

```
1. main() يبدأ
   ↓
2. تحميل الألغاز من data/puzzles.txt
   → فشل: الملف غير موجود
   ↓
3. puzzles.empty() = true
   ↓
```

4. طباعة رسالة خطأ:
"Error: No puzzles found!"
↓
5. return 1 (فشل)

الخرج:

```
Loading puzzles from file...
Error: Can't open file data/puzzles.txt

Error: No puzzles found!
Make sure 'data/puzzles.txt' exists and has puzzles.
```

❌ سيناريو فاشل (بنية المكتبة معطوبة)

1. main() يبدأ
↓
2. تحميل الألغاز
→ نجح: 50 لغز
↓
3. بناء المكتبة
→ تم البناء (لكن به خطأ منطقي)
↓
4. التحقق من الصحة
→ checkIfValid() = false
↓
5. طباعة رسالة خطأ:
"Error: Library structure is broken!"
↓
6. return 1 (فشل)

لماذا هذا التصميم؟

1. الفصل بين المسؤوليات (Separation of Concerns)

```
main.cpp    → التنسيق العام
Utils.cpp   → تحميل البيانات
Library.cpp → بناء الهيكل
Game.cpp    → منطق اللعب
```

الفوائد:

- سهولة الصيانة
- إمكانية تعديل جزء بدون التأثير على الأجزاء الأخرى
- سهولة إضافة مميزات جديدة

---

2. التحقق التدريجي (Progressive Validation)

```cpp
if (puzzles.empty())        → توقف مبكر
if (!library.checkIfValid()) → توقف قبل اللعبة
```

الفوائد:

- كشف الأخطاء في أقرب وقت
- عدم إضاعة الموارد في عمليات لن تنجح
- رسائل خطأ واضحة للمستخدم

---

3. استخدام المؤشرات (Pointers)

```cpp
Game game(&library);  // مؤشر وليس نسخة
```

الفوائد:

- كفاءة الذاكرة
- سرعة في التنفيذ
- تجنب نسخ البيانات الكبيرة

مقارنة: تصميم جيد vs تصميم سيء

❌ تصميم سيء (كل شيء في main)

```cpp
```

```
int main() {
    // سطر من كود تحميل الألغاز 500
    // سطر من كود بناء المكتبة 300
    // سطر من كود اللعبة 400
    // كل شيء مبعثر وغير منظم
    return 0;
}
```

المشاكل:

- صعوبة القراءة
- صعوبة الصيانة
- استحالة إعادة استخدام الكود
- صعوبة العمل الجماعي

✅ تصميم جيد (الحالي)

```
cpp

int main() {
    vector<Puzzle> puzzles = loadPuzzlesFromFile(puzzleFile);
    Library library(puzzles);
    Game game(&library);
    game.start();
    return 0;
}
```

المميزات:

- واضح ومختصر
- كل جزء في ملفه الخاص
- سهولة الفهم والصيانة
- قابلية إعادة الاستخدام

الأخطاء الشائعة وكيفية تجنبها

1. نسيان التحقق من الأخطاء

```
cpp
```

```cpp
// ❌ خطأ
vector<Puzzle> puzzles = loadPuzzlesFromFile(puzzleFile);
Library library(puzzles);  // ماذا لو puzzles فارغ؟

// ✅ صحيح
vector<Puzzle> puzzles = loadPuzzlesFromFile(puzzleFile);
if (puzzles.empty()) {
    // معالجة الخطأ
    return 1;
}
Library library(puzzles);
```

---

2. عدم إعطاء رسائل واضحة

cpp

```cpp
// ❌ خطأ
if (puzzles.empty()) {
    cout << "Error!" << endl;  // غامض جداً
    return 1;
}

// ✅ صحيح
if (puzzles.empty()) {
    cout << "\nError: No puzzles found!" << endl;
    cout << "Make sure 'data/puzzles.txt' exists and has puzzles." << endl;
    return 1;
}
```

---

3. نسيان return بعد الخطأ

cpp

```
// ❌ خطأ
if (puzzles.empty()) {
    cout << "Error!" << endl;
    // نسينا return - البرنامج سيكمل!
}
Library library(puzzles);  // سيحاول البناء بـ vector فارغ!

// ✅ صحيح
if (puzzles.empty()) {
    cout << "Error!" << endl;
    return 1;  // توقف فوري
}
```

الخلاصة

ملف **main.cpp** هو المايسترو الذي:

🚚 يحمّل

- الألغاز من الملف
- البيانات الضرورية

🏗️ يبني

- هيكل المكتبة
- الكتب والروابط

✅ يتحقق

- من وجود البيانات
- من صحة البنية

🎮 يشغّل

- اللعبة الرئيسية
- التفاعل مع اللاعب

قاعدة ذهبية: main.cpp يجب أن يكون قصيراً ومنسقاً، يستدعي الدوال الكبيرة من الملفات الأخرى، ولا يحتوي على منطق معقد.