Dev-C++ PROGRAMMING:

PROFESSIONAL

Contents:

System and Large-Scale Project Design,
High-Performance Programming Techniques,
Multithreading, Dev-C++ GUI
Dev-C++ Network Programming
Dev-C++ Advanced Tepics

Writer: Student Price Book Center

คำนำ

ในโลกของการพัฒนาซอฟต์แวร์สมัยใหม่ ภาษาซีพลัสพลัส (C++) ยังคงเป็นหนึ่งในภาษาที่มีความสำคัญที่สุด โดยเฉพาะอย่างยิ่งในบริบทของการเขียนโปรแกรมระบบ การพัฒนาเกม โปรแกรมฝังตัว และซอฟต์แวร์ที่ต้องการประสิทธิภาพสูง ภาษา C++ มีจุดเด่นที่ให้การควบคุมหน่วยความจำอย่างละเอียด การสนับสนุนแนวคิดเชิงวัตถุ (OOP) อย่างลึกซึ้ง และการรองรับแนวทางการพัฒนาแบบ low-level ที่นักพัฒนามืออาชีพนิยมใช้กันอย่างแพร่หลาย

หนังสือ *Dev-C++ Programming: Professional* เล่มนี้ถูกออกแบบมาเพื่อเป็นคู่มือที่ครบถ้วนและเป็นระบบสำหรับนักพัฒนาซอฟต์แวร์ที่ต้องการยกระดับความรู้จากระดับกลางสู่ระดับมืออาชีพ โดยมุ่งเน้นการใช้ **Dev-C++ IDE** ซึ่งแม้จะดูเรียบง่ายและล้าสมัยในบางด้าน แต่ก็ยังคงเป็นเครื่องมือยอดนิยมของผู้เริ่มต้น รวมถึงผู้ที่ต้องการควบคุมกระบวนการคอมไพล์ในระดับล่าง เนื้อหาในเล่มนี้จึงมุ่งหวังให้ผู้อ่านสามารถใช้ Dev-C++ เพื่อพัฒนาโปรแกรมเชิงระบบ โปรแกรมที่มีประสิทธิภาพสูง และโครงสร้างที่ซับซ้อน ได้อย่างเป็นมืออาชีพ

หนังสือเริ่มต้นด้วย **การออกแบบระบบและโครงสร้างโปรเจกต์** ซึ่งถือเป็นรากฐานที่สำคัญสำหรับการพัฒนาซอฟต์แวร์ในระดับองค์กรหรือโปรเจกต์ขนาดใหญ่ โดยจะกล่าวถึงการแยกไฟล์ .cpp (implementation file) และ .h (header file) อย่างถูกต้อง การใช้ *Header Guards* เพื่อป้องกันการ include ซ้ำซ้อน และการจัดโครงสร้างแบบโมดูลาร์ที่ช่วยให้โปรเจกต์มีความยืดหยุ่น ดูแลรักษาง่าย และสามารถแบ่งงานให้ทีมพัฒนาได้อย่างมีประสิทธิภาพ ทั้งนี้ยังให้แนวทางการวางโฟลเดอร์ โครงสร้างไฟล์ และแนวทางการตั้งชื่อที่สอดคล้องกับมาตรฐานอุตสาหกรรม

ถัดมา หนังสือจะพาผู้อ่านเข้าสู่ **เทคนิคการเขียนโปรแกรมประสิทธิภาพสูง** โดยเน้นการจัดการหน่วยความจำอย่างมีประสิทธิภาพ การใช้คำสั่งระดับล่างเพื่อควบคุมการจัดสรรและคืนค่าหน่วยความจำ (manual memory management) การทำ profiling เพื่อตรวจสอบและปรับปรุงประสิทธิภาพ และการใช้ bitwise operations ซึ่งเป็นเทคนิคที่มักพบในงานด้าน embedded systems หรือการประมวลผลที่ต้องใช้ทรัพยากรน้อยที่สุด

Multithreading หรือการเขียนโปรแกรมแบบหลายเธรด เป็นอีกหนึ่งหัวข้อที่ได้รับการกล่าวถึงอย่างละเอียดในหนังสือเล่มนี้ โดยเนื้อหาจะครอบคลุมถึงการใช้ **<thread>** ซึ่งเป็นคุณสมบัติที่ถูกเพิ่มเข้ามาในมาตรฐาน C++11 เพื่อให้สามารถสร้างเธรดได้อย่างง่ายดาย พร้อมทั้งเรียนรู้การใช้ mutex, lock_guard, และ condition_variable เพื่อจัดการกับ race condition และการประสานการทำงานระหว่างเธรดอย่างปลอดภัย ทั้งนี้ยังมีแนวทางการออกแบบระบบให้ thread-safe ซึ่งเป็นพื้นฐานของการพัฒนาโปรแกรมในยุคที่ hardware มี multi-core CPU แพร่หลาย

แม้ว่า Dev-C++ จะไม่ใช่เครื่องมือที่ถูกออกแบบมาสำหรับการสร้าง GUI โดยตรง แต่หนังสือยังคงนำเสนอหัวข้อ **การสร้าง GUI เบื้องต้น** โดยใช้ WinAPI ผ่าน windows.h ซึ่งเปิดโอกาสให้นักพัฒนาสามารถสร้างหน้าต่าง ปุ่ม และกล่องข้อความด้วยคำสั่งระดับล่าง ซึ่งเป็นพื้นฐานของการเขียน GUI แบบดั้งเดิมบน Windows หนังสือยังแนะนำให้ผู้อ่านที่สนใจ GUI อย่างจริงจังพิจารณาใช้ไลบรารี

อย่าง **Qt** ซึ่งแม้จะไม่รองรับ Dev-C++ โดยตรง แต่ก็เป็นแนวทางในการต่อยอดที่ดีสำหรับการพัฒนาแอปพลิเคชันในระดับมืออาชีพ

ในด้าน **การทำงานร่วมกับระบบและเครือข่าย** หนังสือให้ความรู้เกี่ยวกับการเขียนโปรแกรม socket เพื่อสื่อสารระหว่างเครื่องคอมพิวเตอร์ หรือสร้างระบบ client-server ขนาดเล็ก พร้อมตัวอย่างการใช้งานจริง เช่น chat server และระบบรับส่งข้อมูลภายในเครือข่าย LAN นอกจากนี้ยังมีเนื้อหาเกี่ยวกับการติดต่อกับ hardware เช่น serial port หรือพอร์ต USB เพื่ออ่านค่าจาก sensor หรือควบคุมอุปกรณ์ภายนอก ซึ่งเป็นทักษะที่จำเป็นสำหรับการพัฒนาโปรแกรมในโลกของ embedded systems และ IoT (Internet of Things)

นอกจากเนื้อหาหลักที่กล่าวมาแล้ว หนังสือเล่มนี้ยังมี **หัวข้อเพิ่มเติม (Optional Topics)** ที่ถูกออกแบบมาเพื่อสร้างแรงบันดาลใจและขยายขอบเขตความรู้ของผู้อ่าน ได้แก่ การเขียนเกมด้วย SDL และ SFML ซึ่งเป็นไลบรารีกราฟิกแบบ 2D ที่ทรงพลัง การพัฒนาไลบรารีของตนเองด้วย C++ สำหรับใช้ซ้ำในโปรเจกต์อื่น ๆ การเตรียมความพร้อมสำหรับการเข้าร่วมแข่งขัน **Competitive Programming** และการประยุกต์ใช้ OpenCV สำหรับงานด้าน Computer Vision ซึ่งเป็นหนึ่งในเทคโนโลยีที่เติบโตอย่างรวดเร็วในยุค AI

หนังสือ *Dev-C++ Programming: Professional* จึงไม่ใช่เพียงตำราที่ให้ความรู้ในเชิงทฤษฎีเท่านั้น แต่ยังเปี่ยมไปด้วยตัวอย่างภาคปฏิบัติ โค้ดที่ทดสอบได้จริง และแนวทางที่สามารถประยุกต์ใช้ได้ในโลกแห่งการทำงานจริง เหมาะสำหรับนักพัฒนาที่ต้องการฝึกฝนตนเองให้มีความพร้อมสำหรับอุตสาหกรรมซอฟต์แวร์ในระดับสากล

หวังเป็นอย่างยิ่งว่าหนังสือเล่มนี้จะเป็นแหล่งความรู้ที่มีคุณค่า จุดประกายความคิด และเป็นเครื่องมือสำคัญที่จะช่วยให้คุณก้าวสู่ความเป็นมืออาชีพในเส้นทางสาย C++ ได้อย่างมั่นใจ

ด้วยความปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 16
การออกแบบระบบและโครงสร้างโปรเจกต์
(System and Large Scale Project Design)

เนื้อหา

- การออกแบบระบบและโครงสร้างโปรเจกต์ C++
- การออกแบบระบบและโครงสร้างโปรเจกต์ C++ รายละเอียดเชิงลึก
- การแยกไฟล์ .cpp และ .h ใน C++
- ขั้นตอนการคอมไพล์และรันโปรแกรม C++ โดยใช้ Dev-C++
- Header Guards
- การจัดโครงสร้างแบบ Modular
- ตัวอย่างที่ใช้ STL, Exception Handling หรือ Template แบบแยกโมดูล

บทนำ

ในโลกของการพัฒนาซอฟต์แวร์ การออกแบบระบบและโครงสร้างโปรเจกต์ถือเป็นรากฐานสำคัญที่ส่งผลโดยตรงต่อคุณภาพ ความยืดหยุ่น และความสามารถในการบำรุงรักษาของโปรแกรม การเขียนโค้ดที่มีประสิทธิภาพเพียงอย่างเดียวอาจไม่เพียงพอ หากขาดกระบวนการจัดระเบียบไฟล์และการวางแผนโครงสร้างอย่างเป็นระบบ บทนี้จะนำเสนอแนวคิดและแนวปฏิบัติที่จำเป็นสำหรับการออกแบบโครงสร้างโปรเจกต์ที่ชัดเจนและสามารถพัฒนาได้ต่อเนื่องในระยะยาว

แนวคิดการแยกไฟล์ .cpp และ .h เป็นหนึ่งในกลยุทธ์หลักที่ช่วยให้นักพัฒนาสามารถแยกความรับผิดชอบระหว่างการประกาศ (Declaration) และการนิยาม (Definition) ได้อย่างชัดเจน โดยไฟล์ Header (.h) ทำหน้าที่ประกาศโครงสร้างข้อมูล ฟังก์ชัน และคลาส ขณะที่ไฟล์ Source (.cpp) จะเก็บรายละเอียดของการนิยามฟังก์ชันและตรรกะของโปรแกรม การจัดแยกเช่นนี้ทำให้โค้ดมีความเป็นระบบ ลดความซับซ้อน และเพิ่มความสามารถในการนำกลับมาใช้ซ้ำได้ง่ายขึ้น

การใช้ Header Guards เป็นเทคนิคที่สำคัญสำหรับการป้องกันปัญหาการนิยามซ้ำ (Multiple Inclusion) ที่มักเกิดขึ้นเมื่อไฟล์ Header เดียวกันถูกเรียกใช้งานหลายครั้งในโปรเจกต์เดียว การเพิ่ม Header Guards โดยใช้คำสั่ง #define และ #ifndef ช่วยให้คอมไพเลอร์รู้จักตรวจสอบการประกาศก่อนที่จะทำซ้ำ ส่งผลให้ลดข้อผิดพลาดระหว่างการคอมไพล์และทำให้โปรแกรมมีเสถียรภาพมากขึ้น

ในระดับโครงสร้างโปรเจกต์ การจัดรูปแบบแบบ Modular เป็นแนวทางที่ได้รับการยอมรับอย่างกว้างขวางในวงการพัฒนาซอฟต์แวร์ Modular Design ช่วยให้โปรเจกต์ถูกแบ่งออกเป็นส่วนย่อยที่มีความสัมพันธ์กันอย่างหลวม (Loose Coupling) และมีขอบเขตชัดเจน (High Cohesion) แต่ละโมดูล

สามารถพัฒนา ทดสอบ และปรับปรุงได้โดยไม่กระทบกับส่วนอื่น การวางโครงสร้างแบบนี้จึงช่วยเพิ่มประสิทธิภาพในการพัฒนาและลดความเสี่ยงจากการเปลี่ยนแปลงโค้ด

การออกแบบโครงสร้างโปรเจกต์ที่ดีจำเป็นต้องพิจารณาความสามารถในการอ่านและการดูแลรักษาของโค้ด การตั้งชื่อไฟล์และโฟลเดอร์ให้สอดคล้องกับหน้าที่ของแต่ละโมดูล รวมถึงการกำหนดลำดับขั้นความสัมพันธ์ระหว่างไฟล์ Header และไฟล์ Source ล้วนเป็นส่วนสำคัญที่ช่วยให้ทีมพัฒนาเข้าใจโครงสร้างได้ชัดเจนและลดความผิดพลาดระหว่างการทำงานร่วมกัน

นอกจากประโยชน์ในแง่ความชัดเจน การจัดโครงสร้างที่เหมาะสมยังเอื้อต่อการปรับปรุงประสิทธิภาพการคอมไพล์ การจัดการ Dependency ระหว่างโมดูล และการรวมระบบเข้ากับเครื่องมือทดสอบอัตโนมัติ (Automated Testing) การออกแบบระบบจึงไม่ใช่เพียงการจัดไฟล์ให้เป็นระเบียบ หากแต่เป็นการวางรากฐานของซอฟต์แวร์ให้มีคุณภาพพร้อมรองรับการพัฒนาในอนาคต

บทนี้จึงมุ่งหวังที่จะสร้างความเข้าใจในหลักการแยกไฟล์ การใช้ Header Guards และการจัดโครงสร้างโปรเจกต์แบบ Modular เพื่อให้นักพัฒนาสามารถสร้างระบบที่มีความยืดหยุ่น แข็งแรง และตอบโจทย์ความต้องการที่ซับซ้อนของซอฟต์แวร์ยุคปัจจุบันได้อย่างมั่นใจ

การออกแบบระบบและโครงสร้างโปรเจกต์ C++

หัวข้อ

- การแยกไฟล์ .cpp และ .h
- การใช้ Header Guards
- การจัดโครงสร้างแบบ Modular

---

1. การแยกไฟล์ .cpp และ .h

แนวคิด

- **Header files (.h)**: เก็บการประกาศ (declarations) ของคลาส, ฟังก์ชัน, ตัวแปร, คอนสแตนต์ หรือ template เพื่อให้ไฟล์อื่นสามารถนำไปใช้ได้
- **Source files (.cpp)**: เก็บการนิยาม (definitions) ของฟังก์ชันและเมธอดต่าง ๆ ที่ประกาศใน .h

ข้อดี

- แยกความรับผิดชอบของโค้ด
- เพิ่มความชัดเจนและง่ายต่อการบำรุงรักษา
- ลดการคอมไพล์ซ้ำซ้อน

ตัวอย่างโครงสร้าง

MyClass.h // ประกาศ class และฟังก์ชัน

MyClass.cpp // นิยามฟังก์ชัน

main.cpp // โค้ดหลักที่เรียกใช้ MyClass

2. การใช้ Header Guards

ปัญหา

- เมื่อ include header ซ้ำหลายครั้งจะเกิดปัญหา "multiple definition" หรือ "redefinition"

วิธีแก้

- ใช้ **Header Guards** เพื่อป้องกันการ include ซ้ำ

รูปแบบ Header Guard

```
#ifndef HEADER_NAME_H
#define HEADER_NAME_H

// เนื้อหา header file

#endif // HEADER_NAME_H
```

หรือใช้ #pragma once (ไม่ใช่มาตรฐานแต่ใช้งานแพร่หลาย)

3. การจัดโครงสร้างแบบ Modular

แนวคิด

- แยกโปรแกรมออกเป็นโมดูลย่อย ๆ ที่มีหน้าที่เฉพาะเจาะจง
- โมดูลแต่ละตัวมีไฟล์ .h และ .cpp ของตัวเอง
- ทำให้โปรเจกต์ขนาดใหญ่ดูแลง่าย และสามารถพัฒนาโดยทีมงานได้

ตัวอย่าง

```
project/
│
├── include/          # Header files (.h)
│   ├── MyClass.h
│   └── Utils.h
│
├── src/              # Source files (.cpp)
│   ├── MyClass.cpp
│   └── Utils.cpp
│
└── main.cpp          # โค้ดหลัก
```

ตัวอย่างโปรเจกต์เล็ก ๆ

1. MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
private:
    int data;
public:
    MyClass(int val);
    void showData() const;
};

#endif // MYCLASS_H
```

2. MyClass.cpp

```
#include <iostream>
#include "MyClass.h"

MyClass::MyClass(int val) : data(val) {}

void MyClass::showData() const {
    std::cout << "Data = " << data << std::endl;
}
```

3. main.cpp

```
#include "MyClass.h"

int main() {
    MyClass obj(100);
    obj.showData();
    return 0;
}
```

คำอธิบาย

- ไฟล์ MyClass.h ประกาศคลาส MyClass พร้อมฟังก์ชัน และใช้ header guard ป้องกันซ้ำ
- ไฟล์ MyClass.cpp นิยามฟังก์ชันของคลาสโดย include header ของตัวเอง

- main.cpp นำเข้า MyClass.h เพื่อสร้างและเรียกใช้ object ของคลาส
- การแยกไฟล์แบบนี้ช่วยให้โปรแกรมง่ายต่อการขยายและบำรุงรักษา

สรุปข้อดีของการแยกไฟล์และโครงสร้าง Modular

ประเด็น	ประโยชน์
แยก .h / .cpp	ลดความซับซ้อนของโค้ด
Header Guards	ป้องกันการ include ซ้ำ
Modular structure	พัฒนาและบำรุงรักษาง่าย, ทีมงานทำงานร่วมกันได้ดี

การออกแบบระบบและโครงสร้างโปรเจกต์ C++ รายละเอียดเชิงลึก

---

1. การแยกไฟล์ .h และ .cpp อย่างละเอียด

ทำไมต้องแยกไฟล์?

- **ไฟล์ header (.h)** จะเก็บแค่การประกาศ (declaration) เท่านั้น เช่น การประกาศคลาส, ฟังก์ชัน , ตัวแปรคงที่, ตัวแปร extern ฯลฯ
- **ไฟล์ source (.cpp)** จะเก็บการนิยาม (definition) ของฟังก์ชัน และเมธอดต่างๆ ที่ประกาศไว้ใน header

ประโยชน์

- ถ้าเปลี่ยนแปลงใน .cpp จะไม่กระทบไฟล์อื่นๆ ที่ include header นั้น ๆ
- ลดเวลาคอมไพล์ เพราะ header ที่ไม่เปลี่ยนไม่ต้องคอมไพล์ใหม่
- แยกความรับผิดชอบให้ชัดเจน ช่วยให้โค้ดอ่านง่าย
- สนับสนุนการทำงานแบบทีม เพราะแต่ละคนทำไฟล์ .cpp ของตัวเองได้

ตัวอย่าง

MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass(int val);       // constructor
    void printValue() const;

private:
```

```
    int value;
};

#endif // MYCLASS_H
```

MyClass.cpp

```
#include <iostream>
#include "MyClass.h"

MyClass::MyClass(int val) : value(val) {}

void MyClass::printValue() const {
    std::cout << "Value is " << value << std::endl;
}
```

main.cpp

```
#include "MyClass.h"

int main() {
    MyClass obj(42);
    obj.printValue();
    return 0;
}
```

---

2. Header Guards (ป้องกัน include ซ้ำ)

ปัญหาการ include ซ้ำ

สมมติถ้าในโปรเจกต์เรา #include "MyClass.h" หลายๆ ครั้งในหลายไฟล์ หรือมีไฟล์ header อื่นๆ ที่ include ไฟล์นี้อีกทีหนึ่ง จะเกิดการนิยามซ้ำ (multiple definition) ทำให้คอมไพล์ไม่ผ่าน

วิธีแก้

ใช้ **Header Guards** ซึ่งเป็น macro ของ C++ ทำหน้าที่ตรวจสอบว่าไฟล์ header ถูก include แล้วหรือยัง ถ้ายังไม่เคย include จะให้ compiler อ่านโค้ดในไฟล์นั้น แต่ถ้าเคย include แล้ว จะข้ามการอ่านไฟล์ซ้ำ

```
#ifndef MYCLASS_H      // ถ้ายังไม่มีการประกาศ MYCLASS_H
#define MYCLASS_H      // ประกาศ MYCLASS_H เพื่อบอกว่ามีการ include แล้ว

// เนื้อหา header file
```

```
#endif // MYCLASS_H
```

ตัวอย่างใน MyClass.h (จากตัวอย่างข้างบน)

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
    // ประกาศ class
};

#endif
```

คำสั่ง #pragma once

- เป็นวิธีที่ง่ายและสั้นกว่า header guard
- ใช้บอก compiler ให้ include ไฟล์นี้ครั้งเดียวเท่านั้น
- ไม่ใช่มาตรฐาน C++ แต่ compiler ส่วนใหญ่รองรับ (เช่น GCC, MSVC, Clang)

```
#pragma once

class MyClass {
    // ...
};
```

---

3. การจัดโครงสร้างแบบ Modular

Modular Programming คืออะไร?

- การแบ่งโปรแกรมเป็น **โมดูลย่อย ๆ** ที่แยกออกจากกัน
- แต่ละโมดูลทำหน้าที่เฉพาะเจาะจง
- โมดูลแต่ละตัวจะมีไฟล์ header (.h) และไฟล์ source (.cpp) ของตัวเอง
- เมื่อโมดูลถูกพัฒนาและทดสอบแยกกันได้ จะช่วยเพิ่มประสิทธิภาพการพัฒนาและบำรุงรักษา

ตัวอย่างโครงสร้างโปรเจกต์

```
ProjectRoot/
│
├── include/            # เก็บไฟล์ header (.h)
│   ├── MyClass.h
│   └── Utils.h
│
```

```
├── src/                # เก็บไฟล์ source (.cpp)
│   ├── MyClass.cpp
│   └── Utils.cpp
│
├── main.cpp            # โค้ดหลักของโปรแกรม
└── Makefile / CMakeLists.txt  # สคริปต์สำหรับ build project
```

การทำงานของโมดูล

- **ไฟล์ header** ประกาศ interface (คลาส, ฟังก์ชันที่โมดูลนั้นมีให้คนอื่นเรียกใช้)
- **ไฟล์ source** นิยามฟังก์ชันและรายละเอียดการทำงานจริง
- โปรแกรมหลัก (เช่น main.cpp) รวมโมดูลต่าง ๆ โดยการ include header และลิงก์ไฟล์ source เข้าด้วยกัน

ตัวอย่าง

Utils.h

```
#ifndef UTILS_H
#define UTILS_H

int add(int a, int b);
void printHello();

#endif
```

Utils.cpp

```
#include <iostream>
#include "Utils.h"

int add(int a, int b) {
    return a + b;
}

void printHello() {
    std::cout << "Hello, Modular World!" << std::endl;
}
```

main.cpp

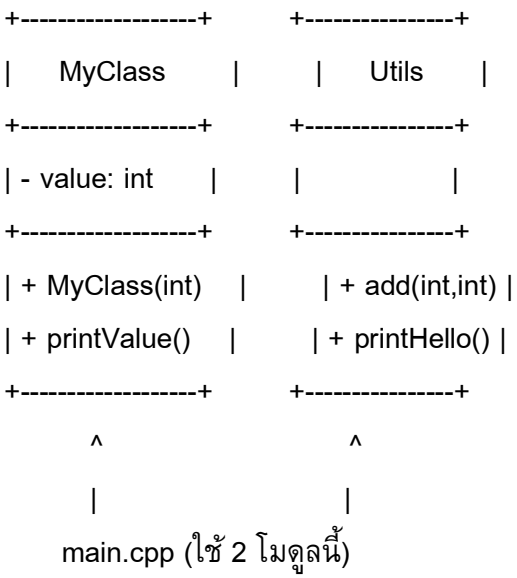
```
#include "MyClass.h"
#include "Utils.h"
```

```
int main() {
    printHello();
    MyClass obj(10);
    obj.printValue();

    int sum = add(5, 7);
    std::cout << "Sum is " << sum << std::endl;

    return 0;
}
```

ทำไม Modular Programming ถึงดี?

ข้อดี	รายละเอียด
ลดความซับซ้อน	แต่ละโมดูลมีหน้าที่ชัดเจน
สะดวกในการดูแลรักษา	แก้ไขหรือเพิ่มฟีเจอร์ในโมดูลใดโมดูลหนึ่งได้โดยไม่กระทบส่วนอื่น
รองรับการทำงานเป็นทีม	สมาชิกแต่ละคนสามารถดูแลโมดูลแยกกันได้
ประหยัดเวลาคอมไพล์	เปลี่ยนเฉพาะโมดูลที่แก้ไขเท่านั้น

ตัวอย่าง UML Diagram สำหรับโปรเจกต์ Modular



สรุป

หัวข้อ	รายละเอียดสำคัญ
แยกไฟล์ .h และ .cpp	แยก declaration กับ definition เพื่อความชัดเจนและบำรุงรักษาง่าย
Header Guards	ป้องกันการ include ซ้ำ ป้องกัน error เวลา compile
Modular Structure	แบ่งโปรแกรมเป็นโมดูลเล็กๆ ที่ชัดเจน แยกไฟล์ ทำให้ทีมงานพัฒนาได้ดี

ถ้าต้องการ ผมช่วยออกแบบตัวอย่างโปรเจกต์ใหญ่ที่ใช้โครงสร้าง Modular และมีไฟล์หลายโมดูลจริงจัง พร้อม Makefile หรือ CMake script ได้ครับ!
หรือจะให้ผมอธิบายการเชื่อมต่อโมดูล การลิงก์ไฟล์ การจัดการ dependencies ก็แจ้งได้เลยครับ :)
ได้ครับ! มาดูรายละเอียดเชิงลึกของ **การแยกไฟล์ .cpp และ .h ใน C++** กัน

การแยกไฟล์ .cpp และ .h ใน C++

1. ความหมายและบทบาทของไฟล์แต่ละชนิด

ไฟล์	บทบาทหลัก	เนื้อหาที่เก็บ
.h (Header file)	ประกาศ (Declaration) ให้กับโปรแกรมอื่นรู้จัก	ประกาศคลาส, ฟังก์ชัน, ตัวแปร, คอนสแตนต์, typedef, macros ฯลฯ
.cpp (Source file)	นิยาม (Definition) ของฟังก์ชันและเมธอด	รหัสการทำงานจริงของฟังก์ชันและเมธอดที่ประกาศไว้ใน header

---

2. ทำไมต้องแยกไฟล์?

- **แยกความรับผิดชอบ**: header แค่บอก “มีอะไรบ้าง” ส่วน source บอก “ทำงานอย่างไร”
- **ลดเวลา compile**: เวลาแก้ .cpp ไฟล์ไม่ต้อง re-compile ทุกไฟล์ที่ include header
- **แก้ไขง่าย**: เปลี่ยนฟังก์ชันใน .cpp ไม่กระทบไฟล์อื่นที่ include .h
- **สนับสนุน modular programming** และช่วยทำงานเป็นทีมได้ง่ายขึ้น

---

3. รูปแบบการแยกไฟล์แบบพื้นฐาน

ตัวอย่างโครงสร้างไฟล์

MyClass.h // ประกาศคลาส MyClass
MyClass.cpp // นิยามฟังก์ชันของ MyClass
main.cpp // โปรแกรมหลัก (เรียกใช้ MyClass)

ตัวอย่างโค้ด

MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass(int val);            // constructor ประกาศไว้
    void printValue() const;     // เมธอดประกาศไว้

private:
    int value;
};

#endif // MYCLASS_H
```

MyClass.cpp

```
#include <iostream>
#include "MyClass.h"

// นิยาม constructor
MyClass::MyClass(int val) : value(val) {}

// นิยามเมธอด
void MyClass::printValue() const {
    std::cout << "Value is: " << value << std::endl;
}
```

main.cpp

```
#include "MyClass.h"

int main() {
    MyClass obj(100);
    obj.printValue();   // Output: Value is: 100
    return 0;
}
```

4. สรุปขั้นตอนแยกไฟล์

ขั้นตอน	รายละเอียด
สร้าง header file (.h)	ประกาศ class, ฟังก์ชัน, ตัวแปร, prototype
สร้าง source file (.cpp)	นิยามฟังก์ชันและเมธอดที่ประกาศใน header
ในไฟล์ source #include header	เพื่อเชื่อมโยง declaration กับ definition
ในโปรแกรมหลัก #include header	เพื่อเรียกใช้ class หรือฟังก์ชัน
คอมไพล์ทุก .cpp รวมกัน	ลิงก์ไฟล์ source เพื่อสร้าง executable

---

5. ข้อควรระวัง

- อย่าประกาศนิยามฟังก์ชันใน header (ยกเว้น inline functions หรือ template) เพราะจะทำให้เกิดการนิยามซ้ำตอนคอมไพล์
- ต้องมี **Header Guards** ในไฟล์ header เพื่อป้องกัน include ซ้ำ
- ใช้ include ให้เหมาะสม อย่า include header ที่ไม่จำเป็นใน header อื่นๆ เพื่อช่วยลดเวลา compile

---

นี่คือตัวอย่างโปรแกรม 5 โปรแกรมที่แยกไฟล์ .h และ .cpp อย่างถูกต้อง พร้อมคำอธิบายง่าย ๆ

ตัวอย่างที่ 1: คลาสง่าย ๆ แสดงข้อความ

MyClass1.h

```
#ifndef MYCLASS1_H
#define MYCLASS1_H

class MyClass1 {
public:
    void sayHello();
};

#endif
```

MyClass1.cpp

```
#include <iostream>
#include "MyClass1.h"

void MyClass1::sayHello() {
    std::cout << "Hello from MyClass1!" << std::endl;
```

```
}
```

main1.cpp

```
#include "MyClass1.h"

int main() {
    MyClass1 obj;
    obj.sayHello();
    return 0;
}
```

ตัวอย่างที่ 2: คลาสเก็บและแสดงค่าเลขจำนวนเต็ม

MyClass2.h

```
#ifndef MYCLASS2_H
#define MYCLASS2_H

class MyClass2 {
private:
    int number;

public:
    MyClass2(int n);
    void printNumber() const;
};

#endif
```

MyClass2.cpp

```
#include <iostream>
#include "MyClass2.h"

MyClass2::MyClass2(int n) : number(n) {}

void MyClass2::printNumber() const {
    std::cout << "Number is: " << number << std::endl;
}
```

main2.cpp

```
#include "MyClass2.h"

int main() {
    MyClass2 obj(42);
    obj.printNumber();
    return 0;
}
```

ตัวอย่างที่ 3: คลาสจัดการพนักงาน (Employee) เก็บชื่อและเงินเดือน

Employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>

class Employee {
private:
    std::string name;
    double salary;

public:
    Employee(const std::string& n, double s);
    void display() const;
};

#endif
```

Employee.cpp

```
#include <iostream>
#include "Employee.h"

Employee::Employee(const std::string& n, double s) : name(n), salary(s) {}

void Employee::display() const {
```

```
    std::cout << "Employee: " << name << ", Salary: $" << salary << std::endl;
}
```

main3.cpp

```
#include "Employee.h"

int main() {
    Employee emp("John Doe", 35000.0);
    emp.display();
    return 0;
}
```

ตัวอย่างที่ 4: คลาสคำนวณพื้นที่สี่เหลี่ยมผืนผ้า

Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h);
    double area() const;
};

#endif
```

Rectangle.cpp

```
#include "Rectangle.h"

Rectangle::Rectangle(double w, double h) : width(w), height(h) {}

double Rectangle::area() const {
    return width * height;
```

```
}
```

main4.cpp

```
#include <iostream>
#include "Rectangle.h"

int main() {
    Rectangle rect(5.0, 3.0);
    std::cout << "Area of rectangle: " << rect.area() << std::endl;
    return 0;
}
```

ตัวอย่างที่ 5: คลาสธนาคาร (BankAccount) เก็บยอดเงิน และฝากถอน

BankAccount.h

```
#ifndef BANKACCOUNT_H
#define BANKACCOUNT_H

class BankAccount {
private:
    double balance;

public:
    BankAccount(double initialBalance);
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance() const;
};

#endif
```

BankAccount.cpp

```
#include <iostream>
#include "BankAccount.h"

BankAccount::BankAccount(double initialBalance) : balance(initialBalance) {}
```

```
void BankAccount::deposit(double amount) {
    balance += amount;
}

void BankAccount::withdraw(double amount) {
    if (amount <= balance) {
        balance -= amount;
    } else {
        std::cout << "Insufficient balance!" << std::endl;
    }
}

double BankAccount::getBalance() const {
    return balance;
}
```

main5.cpp

```
#include <iostream>
#include "BankAccount.h"

int main() {
    BankAccount acc(1000.0);
    acc.deposit(500);
    acc.withdraw(200);
    std::cout << "Current balance: " << acc.getBalance() << std::endl;
    acc.withdraw(2000);  // ทดลองถอนเกินยอด
    return 0;
}
```

วิธีคอมไพล์ (ตัวอย่าง)

```
g++ MyClass1.cpp main1.cpp -o program1
g++ MyClass2.cpp main2.cpp -o program2
g++ Employee.cpp main3.cpp -o program3
g++ Rectangle.cpp main4.cpp -o program4
g++ BankAccount.cpp main5.cpp -o program5
```

ขั้นตอนการคอมไพล์และรันโปรแกรม C++ โดยใช้ Dev-C++

การคอมไพล์และรันโปรแกรมใน Dev-C++

1. เปิดโปรแกรม Dev-C++

- เปิดโปรแกรม Dev-C++ ขึ้นมา
- ถ้ายังไม่มีโปรเจกต์ ให้เลือก **File > New > Project...** เพื่อสร้างโปรเจกต์ใหม่ หรือเลือก **File > New > Source File** เพื่อสร้างไฟล์ .cpp ใหม่

2. สร้างโปรเจกต์ (Project)

- เมื่อเลือก New Project
- เลือกประเภทโปรเจกต์ เช่น **Console Application**
- ตั้งชื่อโปรเจกต์และเลือกโฟลเดอร์เก็บไฟล์
- Dev-C++ จะสร้างไฟล์หลัก เช่น main.cpp ให้

3. เพิ่มไฟล์ .h และ .cpp

- ถ้ามีหลายไฟล์ เช่น MyClass.cpp, MyClass.h, main.cpp
- ให้เลือก **Project > Add to Project...** แล้วเลือกไฟล์ .cpp และ .h ที่ต้องการเพิ่มเข้ามาในโปรเจกต์
- Dev-C++ จะจัดการคอมไพล์ทุกไฟล์รวมกัน

4. เขียนโค้ดในไฟล์ .cpp และ .h

- แก้ไขไฟล์ .cpp และ .h ตามต้องการ
- ตัวอย่างเช่น main.cpp จะมีฟังก์ชัน main() เป็นจุดเริ่มต้นของโปรแกรม

5. คอมไพล์โปรเจกต์

- คลิกที่ไอคอน **Compile** (ปุ่มรูปฟันเฟือง หรือกด Ctrl + F9)
- Dev-C++ จะตรวจสอบโค้ดว่ามีข้อผิดพลาดหรือไม่
- ถ้าไม่มีข้อผิดพลาด จะสร้างไฟล์ executable (.exe)

6. รันโปรแกรม

- คลิกที่ไอคอน **Run** (ปุ่มรูปสามเหลี่ยมสีเขียว หรือกด Ctrl + F10)
- โปรแกรมจะทำงาน และแสดงผลลัพธ์ในหน้าต่าง Console (หน้าต่างคำสั่ง)

- ถ้าต้องการคอมไพล์และรันพร้อมกัน ให้กด F9

---

7. แก้ไขข้อผิดพลาด

- ถ้ามี Error หรือ Warning ในการคอมไพล์ Dev-C++ จะแสดงข้อความในส่วน Output
- อ่านข้อความผิดพลาด และแก้ไขโค้ดตามนั้น
- จากนั้นคอมไพล์ใหม่จนไม่มีข้อผิดพลาด

ตัวอย่าง

สมมุติคุณมีโปรเจกต์ชื่อ MyProject มีไฟล์

- MyClass.h
- MyClass.cpp
- main.cpp

ขั้นตอน

1. เปิด Dev-C++ แล้วโหลดโปรเจกต์ MyProject
2. เพิ่มไฟล์ทั้งหมดเข้ามาในโปรเจกต์ (ถ้ายังไม่เพิ่ม)
3. กด Ctrl+F9 เพื่อคอมไพล์
4. กด Ctrl+F10 เพื่อรัน

หมายเหตุ

- ไฟล์ .h (header) จะไม่ถูกคอมไพล์แยก แต่จะถูก include เข้าไปใน .cpp
- Dev-C++ จะคอมไพล์ .cpp ทุกไฟล์ในโปรเจกต์รวมกัน
- อย่าลืมเซฟไฟล์ก่อนคอมไพล์ทุกครั้ง (Ctrl + S)

ตัวอย่างโปรเจกต์เล็ก ๆ โดยแยกไฟล์ .h กับ .cpp พร้อมแสดง UML, โค้ด, ผลรัน และคำอธิบายโค้ดทีละส่วน

ตัวอย่างโปรเจกต์: ระบบจัดการหนังสือ (Book Management)

---

1. UML Diagram

```
+------------------+
|      Book        |
+------------------+
| - title: string  |
| - author: string |
```

+------------------+

| + Book(t, a) |

| + getTitle(): string |

| + getAuthor(): string|

| + display(): void |

+------------------+

- - หมายถึง private
- + หมายถึง public

---

2. โครงสร้างไฟล์ในโปรเจกต์

- Book.h (Header file)
- Book.cpp (Implementation file)
- main.cpp (ไฟล์หลัก)

---

3. โค้ดแต่ละไฟล์

Book.h

```
#ifndef BOOK_H
#define BOOK_H

#include <string>
using namespace std;

class Book {
private:
    string title;
    string author;

public:
    Book(string t, string a);  // Constructor
    string getTitle();
    string getAuthor();
    void display();
};
```

```
#endif
```

Book.cpp

```
#include <iostream>
#include "Book.h"

Book::Book(string t, string a) {
    title = t;
    author = a;
}

string Book::getTitle() {
    return title;
}

string Book::getAuthor() {
    return author;
}

void Book::display() {
    cout << "Title: " << title << ", Author: " << author << endl;
}
```

main.cpp

```
#include <iostream>
#include "Book.h"
using namespace std;

int main() {
    // สร้าง object book1
    Book book1("The Alchemist", "Paulo Coelho");

    // เรียกใช้ method display
    book1.display();
```

```
    // สร้าง object book2
    Book book2("1984", "George Orwell");
    book2.display();

    return 0;
}
```

---

4. วิธีคอมไพล์และรันบน Dev-C++

- เปิด Dev-C++
- สร้างโปรเจกต์ Console Application
- เพิ่มไฟล์ Book.h, Book.cpp และ main.cpp ลงในโปรเจกต์
- กด Ctrl + F9 เพื่อคอมไพล์
- กด Ctrl + F10 เพื่อรันโปรแกรม

---

5. ผลลัพธ์จากการรัน

Title: The Alchemist, Author: Paulo Coelho

Title: 1984, Author: George Orwell

---

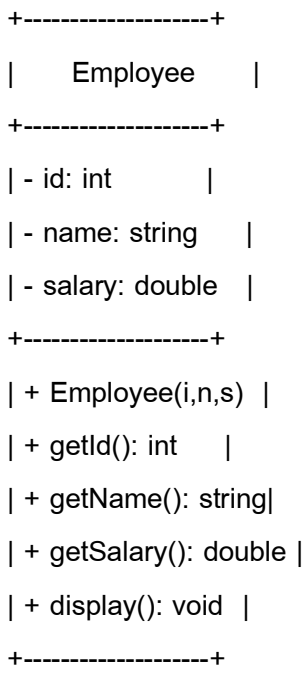
6. คำอธิบายโค้ด

- Book.h คือไฟล์ Header ที่ประกาศ class Book มี attribute title กับ author เป็น private
- Book มี constructor รับพารามิเตอร์สองตัวคือชื่อหนังสือและชื่อผู้แต่ง
- มี method getTitle(), getAuthor() เพื่อดึงข้อมูล และ display() เพื่อแสดงข้อมูล
- Book.cpp เป็นไฟล์ที่เขียนรายละเอียดการทำงานของ constructor และ method ต่าง ๆ
- ใน main.cpp สร้าง object book1 และ book2 จาก class Book โดยส่งค่าข้อมูลตอนสร้าง object
- เรียกใช้ method display() เพื่อแสดงข้อมูลหนังสือ

นี่คือตัวอย่างโปรเจกต์แยกไฟล์ .h กับ .cpp อีก 3 ตัวอย่าง พร้อม UML, โค้ด, ผลรัน และคำอธิบายเชิงลึก

ตัวอย่างที่ 1: ระบบจัดการพนักงาน (Employee Management)

UML Diagram



Employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

class Employee {
private:
    int id;
    string name;
    double salary;

public:
    Employee(int i, string n, double s);
    int getId();
    string getName();
    double getSalary();
    void display();
};
```

```
#endif
```

Employee.cpp

```
#include <iostream>
#include "Employee.h"

Employee::Employee(int i, string n, double s) {
    id = i;
    name = n;
    salary = s;
}

int Employee::getId() {
    return id;
}

string Employee::getName() {
    return name;
}

double Employee::getSalary() {
    return salary;
}

void Employee::display() {
    cout << "ID: " << id << ", Name: " << name << ", Salary: " << salary << endl;
}
```

main.cpp

```
#include <iostream>
#include "Employee.h"
using namespace std;
```

```
int main() {
    Employee emp1(101, "John Doe", 50000.50);
    emp1.display();

    Employee emp2(102, "Jane Smith", 62000.75);
    emp2.display();

    return 0;
}
```

ผลลัพธ์

ID: 101, Name: John Doe, Salary: 50000.5

ID: 102, Name: Jane Smith, Salary: 62000.75

คำอธิบาย

- คลาส Employee มีข้อมูลพนักงานคือ id, name และ salary
- มี constructor เพื่อกำหนดค่าเริ่มต้น
- ฟังก์ชัน display() แสดงข้อมูลพนักงาน

ตัวอย่างที่ 2: ระบบจัดการสินค้า (Product Management)

UML Diagram

```
+--------------------+
|      Product       |
+--------------------+
| - code: string     |
| - name: string     |
| - price: double    |
+--------------------+
| + Product(c,n,p)   |
| + getCode(): string |
| + getName(): string |
| + getPrice(): double|
| + display(): void  |
```

```
+--------------------+
```

Product.h

```
#ifndef PRODUCT_H
#define PRODUCT_H

#include <string>
using namespace std;

class Product {
private:
    string code;
    string name;
    double price;

public:
    Product(string c, string n, double p);
    string getCode();
    string getName();
    double getPrice();
    void display();
};

#endif
```

Product.cpp

```
#include <iostream>
#include "Product.h"

Product::Product(string c, string n, double p) {
    code = c;
    name = n;
    price = p;
}
```

```
string Product::getCode() {
    return code;
}

string Product::getName() {
    return name;
}

double Product::getPrice() {
    return price;
}

void Product::display() {
    cout << "Code: " << code << ", Name: " << name << ", Price: " << price << endl;
}
```

---

main.cpp

```
#include <iostream>
#include "Product.h"
using namespace std;

int main() {
    Product prod1("P001", "Laptop", 15000.00);
    prod1.display();

    Product prod2("P002", "Smartphone", 8000.00);
    prod2.display();

    return 0;
}
```

ผลลัพธ์

```
Code: P001, Name: Laptop, Price: 15000
```

Code: P002, Name: Smartphone, Price: 8000

คำอธิบาย

- คลาส Product มีข้อมูลสินค้า code, name, และ price
- constructor ใช้กำหนดค่าข้อมูล
- display() แสดงข้อมูลสินค้า

ตัวอย่างที่ 3: ระบบจัดการนักเรียน (Student Management)

UML Diagram

```
+---------------------+
|       Student       |
+---------------------+
| - studentId: int    |
| - fullName: string  |
| - grade: double     |
+---------------------+
| + Student(id,n,g)   |
| + getStudentId(): int |
| + getFullName(): string |
| + getGrade(): double |
| + display(): void   |
+---------------------+
```

Student.h

```
#ifndef STUDENT_H
#define STUDENT_H

#include <string>
using namespace std;

class Student {
private:
    int studentId;
```