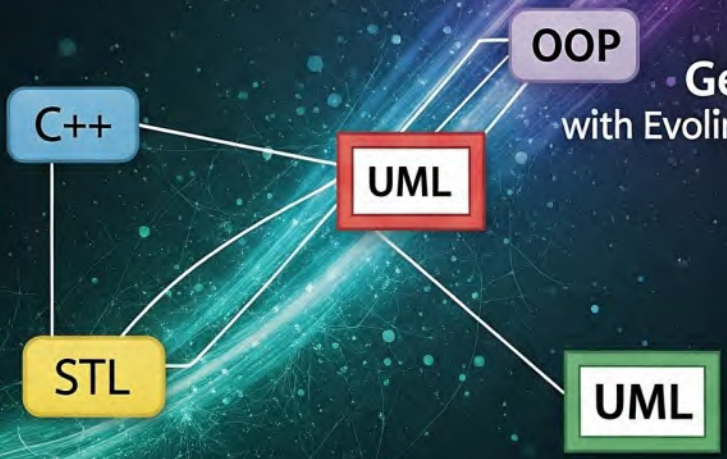
C++

C++ OOP PROFESSIONAL

(Integrative-Generative AI Edition)



Generative AI
with Evoling, Dynamic
Patterns
Standard Template
Library (STL)
Design Patterns
SOLID Principles
SOLID Principles
more

Student Price Book Center

คำนำ

หนังสือ **C++ OOP Professional** เล่มนี้ จัดทำขึ้นด้วยจุดมุ่งหมายที่จะเป็นทั้ง **คู่มือการเรียนรู้** และ **แนวทางปฏิบัติจริง** (Reference & Practice Guide) เพื่อช่วยผู้เรียนพัฒนาและยกระดับทักษะการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming: OOP) ด้วยภาษา C++ ตั้งแต่ **ผู้เริ่มต้น** จนถึง **ผู้ชำนาญ** โดยผู้เขียนตั้งใจอธิบายทั้ง **แนวคิดเชิงทฤษฎี** และ **การนำไปใช้จริง** (Concept and Practical Use) เพื่อให้ผู้เรียนเข้าใจทั้งแก่นแท้ของ OOP และรู้วิธีลงมือเขียนโปรแกรมเพื่อแก้โจทย์และปัญหาต่าง ๆ ในโลกจริงได้อย่างมั่นใจ

โครงสร้างเนื้อหาของหนังสือ

ผู้เขียนได้ออกแบบและจัดลำดับหัวข้อการเรียนรู้เป็น **4 ระดับ** อย่างชัดเจน เป็นขั้นเป็นตอน เพื่อให้ผู้เรียนเลือกศึกษาและพัฒนาตนเองได้ตรงตามศักยภาพและเป้าหมาย:

□ *ระดับผู้เริ่มต้น (Beginner Level)*

เป็นหัวใจของผู้ที่เพิ่งเข้ามาทำความรู้จัก C++ และ OOP

- ปูพื้นฐาน **ภาษา C++** ตั้งแต่โครงสร้างโปรแกรม ตัวแปร เงื่อนไข (if, switch) และลูป (for, while)
- ฟังก์ชันและการส่งค่าผ่าน parameter
- Pointer & Reference พื้นฐาน
- การรับและส่งค่าด้วย **Input/Output** (cin, cout, getline)
- แนวคิดของ **Object** และ **Class**: เข้าใจกระบวนทัศน์เชิงอ็อบเจกต์และความแตกต่างจากการเขียนโปรแกรมแบบ Procedural
- การสร้างคลาสและอ็อบเจกต์จริง: การใช้ Constructor, Destructor, Access Specifiers (public, private, protected) และ Getters / Setters เพื่อให้เข้าใจ Encapsulation ขั้นต้น

□ *ระดับกลาง (Intermediate Level)*

เรียนรู้หัวใจของ OOP เพื่อเขียนโปรแกรมขนาดใหญ่และเป็นระบบ

- เจาะลึก Encapsulation และ Abstraction: เทคนิคการซ่อนรายละเอียด (Data Hiding) และการออกแบบคลาสด้วย Abstract Class / Pure Virtual Function
- Inheritance (การสืบทอด): public / protected / private, Constructor & Destructor Chain และการอ้างอิงคลาสแม่ (Base::)
- Polymorphism:
    - Compile-Time Polymorphism (Overloading): ฟังก์ชันและโอเปอเรเตอร์

    - Run-Time Polymorphism (Overriding): Virtual Functions, Pure Virtual Functions, Dynamic Binding
- การบริหารจัดการหน่วยความจำ: new, delete, Shallow Copy vs Deep Copy, Copy Constructor, Move Constructor, Rule of 3 / Rule of 5
- แนวคิดเกี่ยวกับ Static และ Const: static member, static method, const method, const parameter และ mutable keyword

☐ *ระดับสูง (Advanced Level)*

ยกระดับทักษะ C++ และ OOP เพื่อรองรับงานขนาดใหญ่และต้องการความยืดหยุ่น

- Templates และ Generic Programming:
    - Function Templates
    - Class Templates
    - การออกแบบคลาสให้เป็นอิสระจากชนิดข้อมูล
- การจัดการข้อผิดพลาด (Exception Handling): try, catch, throw และการออกแบบคลาส Exception เพื่อรองรับ error handling ในโปรแกรมจริง
- STL + OOP: เรียนรู้การใช้ vector, list, map, set และโครงสร้าง STL ร่วมกับคลาสของตนเอง
- Smart Pointers (C++11 และใหม่กว่า): unique_ptr, shared_ptr, weak_ptr และแนวคิด RAII เพื่อการบริหารหน่วยความจำอัตโนมัติ

☐ *ระดับมืออาชีพ (Professional Level)*

ก้าวสู่การเป็นผู้พัฒนาซอฟต์แวร์อาชีพ ด้วยแนวคิดและแนวปฏิบัติสากล

- **Design Patterns**:
    - Creational Patterns: Singleton, Factory Method
    - Structural Patterns: Adapter, Decorator, Composite
    - Behavioral Patterns: Observer, Strategy, State
    - แนวคิดและกรณีศึกษาการใช้ Design Patterns ในงานจริง
- **SOLID Principles**:
    - Single Responsibility Principle
    - Open/Closed Principle
    - Liskov Substitution Principle
    - Interface Segregation Principle
    - Dependency Inversion Principle
- การออกแบบโปรแกรมแบบ Modular / Component-Based Architecture

- Test-Driven Development: Unit Test, Mock Objects และการใช้ Framework เช่น **Google Test** หรือ **Catch2** เพื่อควบคุมคุณภาพโค้ด

ขอให้สนุก มีไฟ และท้าทายไปกับการเรียนรู้ C++ OOP! เพื่อก้าวไปสู่การเป็นโปรแกรมเมอร์ C++ มืออาชีพได้อย่างเต็มภาคภูมิ!

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 1
พื้นฐานภาษา C++ ที่ต้องรู้ก่อนเข้า OOP
(Introduction to C++ Basic)

เนื้อหา

- โครงสร้างโปรแกรม C++
- ขยายรายละเอียดของ พื้นฐานภาษา C++ ก่อนเข้า OOP
- เครื่องมือที่ต้องใช้ในการเขียน C++
- ตัวอย่างโปรแกรม C++

บทนำ

การเรียนรู้ **การเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming: OOP)** ในภาษา C++ นั้นเป็นก้าวสำคัญสู่การเป็นโปรแกรมเมอร์ที่เข้าใจกระบวนทัศน์สมัยใหม่และรองรับงานพัฒนาซอฟต์แวร์ขนาดใหญ่ได้อย่างเป็นระบบ แต่ก่อนจะก้าวไปถึงจุดนั้น ผู้เรียนจำเป็นต้องเข้าใจ **พื้นฐานของภาษา C++** อย่างชัดเจนเสียก่อน เนื้อหาตอนแรกของบทนี้จึงมุ่งเน้นไปที่การทบทวนและอธิบายหัวข้อสำคัญของ C++ ที่ผู้เรียนต้องรู้ เพื่อเป็นรากฐานให้เข้าใจและต่อยอดสู่การเขียนโปรแกรมแบบ OOP ได้อย่างเต็มศักยภาพ

หัวข้อแรกคือ **โครงสร้างโปรแกรม C++** โดยจะอธิบายตั้งแต่รูปแบบไฟล์ .cpp และ .h โครงสร้างของ main() จนถึงการใช้งาน #include และ namespace เพื่อให้ผู้เรียนเข้าใจกระบวนการทำงานของโปรแกรม C++ ตั้งแต่ขั้นตอนการคอมไพล์ไปจนถึงการรันจริง จัดเป็นพื้นฐานสำคัญที่ต้องเข้าใจก่อนเขียนโปรแกรมใหญ่ ๆ หรือต่อยอดไปถึง OOP

ต่อมาจะอธิบายถึง **การใช้งานตัวแปร**, ประเภทข้อมูล, และโครงสร้างควบคุมโปรแกรม เช่น if, switch, และการทำงานของลูป (for, while) ซึ่งเป็นหัวใจของการควบคุมลำดับและทิศทางของโปรแกรม โดยผู้เรียนจะได้เห็นตัวอย่างและแนวคิดการเลือกใช้ให้เหมาะสมตามบริบทของโจทย์หรือโปรเจกต์จริง

อีกทั้งผู้เรียนจะได้ทบทวนและทำความเข้าใจกับ **ฟังก์ชัน**และการส่งค่าผ่าน parameter เพื่อเข้าใจกระบวนการแตกโจทย์เป็นหน่วยย่อย ๆ (decomposition) และเลือกใช้ฟังก์ชันเป็นหน่วยทำงานอิสระ โดยเนื้อหาจะครอบคลุมทั้งการส่งค่าผ่านค่าปกติ (pass by value), การส่งค่าผ่าน reference (pass by reference) และการส่งค่าด้วย pointer เพื่อปูทางสู่หัวข้อการใช้ pointer และ reference ในขั้นสูง

สุดท้ายผู้เรียนจะได้เรียนรู้ **Pointer & Reference** เบื้องต้น และเข้าใจกระบวนการจัดการ **Input/Output** ด้วย cin, cout และ getline() อย่างถูกต้องและมีประสิทธิภาพ เนื้อหาทั้งหมดนี้เป็นหัวใจของการเขียนโปรแกรม C++ และเป็นจุดเริ่มต้นอันมั่นคงเพื่อก้าวไปสู่โลกของ OOP ในบทต่อ ๆ ไป โดยผู้เรียนจะมีทั้งทฤษฎี ตัวอย่าง และโจทย์จริงเพื่อช่วยเสริมทักษะและความเข้าใจ จนสามารถต่อยอดสู่การออกแบบและพัฒนาโปรแกรม C++ เชิงอ็อบเจกต์ได้อย่างมั่นใจ

โครงสร้างโปรแกรม C++

☐ 1. โครงสร้างโปรแกรม C++

ทุกโปรแกรมภาษา C++ ต้องมีฟังก์ชัน main() เป็นจุดเริ่มต้นของโปรแกรม

```
#include <iostream> // ไฟล์ header สำหรับ I/O

using namespace std;

int main() {
    cout << "Hello, C++!" << endl; // แสดงข้อความ
    return 0; // ส่งค่ากลับให้ OS ว่าโปรแกรมจบปกติ
}
```

คำอธิบาย:

- #include <iostream> ใช้สำหรับการรับ/ส่งข้อมูล
- using namespace std; เพื่อเรียกใช้งาน std:: โดยไม่ต้องพิมพ์ซ้ำ
- int main() คือจุดเริ่มต้นของโปรแกรม
- cout คือการแสดงผลทางหน้าจอ

☐ 2. การใช้งานตัวแปร, เงื่อนไข และลูป

2.1 ตัวแปร (Variables)

```
int age = 25;
float score = 89.5;
char grade = 'A';
string name = "John";
bool isPassed = true;
```

2.2 เงื่อนไข (Condition)

if-else

```
if (age >= 18) {
    cout << "You are an adult.";
```

```
} else {
    cout << "You are a minor.";
}
```

switch

```
switch (grade) {
    case 'A': cout << "Excellent!"; break;
    case 'B': cout << "Good!"; break;
    default: cout << "Try harder.";
}
```

2.3 ลูป (Loop)

for loop

```
for (int i = 0; i < 5; i++) {
    cout << "i = " << i << endl;
}
```

while loop

```
int i = 0;
while (i < 5) {
    cout << "i = " << i << endl;
    i++;
}
```

---

3. ฟังก์ชันและการรับค่าผ่าน Parameter

ฟังก์ชันแบบรับค่าพารามิเตอร์

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

int main() {
    cout << add(5, 3); // แสดงผล 8
}
```

ฟังก์ชันไม่คืนค่า (void)

```
void greet(string name) {
    cout << "Hello, " << name << endl;
}
```

4. Pointer และ Reference เบื้องต้น

Pointer (ตัวชี้)

```
int x = 10;
int* ptr = &x; // ptr ชี้ไปที่ address ของ x

cout << "Value of x: " << x << endl;
cout << "Address of x: " << ptr << endl;
cout << "Value via pointer: " << *ptr << endl;
```

Reference (ตัวอ้างอิง)

```
int x = 5;
int& ref = x; // ref คือชื่ออีกชื่อหนึ่งของ x

ref = 10;
cout << x << endl; // ผลลัพธ์คือ 10 เพราะ ref คือ x
```

5. การจัดการ Input/Output (cin, cout, getline)

cin / cout

```
int age;
cout << "Enter your age: ";
cin >> age; // รับค่าจากผู้ใช้
cout << "You are " << age << " years old." << endl;
```

getline (ใช้รับข้อความทั้งบรรทัด)

```
string name;
cout << "Enter your full name: ";
getline(cin, name);
cout << "Hello, " << name << "!" << endl;
```

สรุป

พื้นฐานเหล่านี้คือรากฐานสำคัญก่อนเข้าสู่การเขียน **OOP** ใน C++ เพราะคุณต้อง:

- เข้าใจ **โครงสร้างภาษา**
- สามารถควบคุม **ลำดับการทำงานของโปรแกรม**
- เข้าใจ **แนวคิด pointer/reference** เพื่อไปใช้ใน class/constructor หรือ dynamic memory
- รู้จัก **ฟังก์ชัน** เพื่อเรียนรู้การสร้าง method ใน class

ขยายรายละเอียดของ พื้นฐานภาษา C++ ก่อนเข้า OOP

☐ 1. โครงสร้างโปรแกรม C++

```
#include <iostream>     // ใช้สำหรับ cin, cout
#include <string>       // ใช้สำหรับตัวแปร string

using namespace std;

int main() {
    cout << "สวัสดีครับ C++" << endl;
    return 0;
}
```

ส่วนประกอบหลัก:

ส่วนประกอบ	อธิบาย
#include	สั่งให้รวมไฟล์ header
using namespace std	ไม่ต้องเขียน std::cout, std::string ทุกครั้ง
main()	จุดเริ่มต้นของโปรแกรม
return 0;	ส่งค่ากลับให้ OS (0 = ปกติ)

☐ 2. ตัวแปร, เงื่อนไข, ลูป

☐ ตัวแปรและชนิดข้อมูล

```
int a = 10;
float b = 5.5;
char c = 'A';
bool flag = true;
string name = "Kittisak";
```

ชนิดข้อมูล	ความหมาย
int	ตัวเลขจำนวนเต็ม
float / double	ตัวเลขทศนิยม
char	ตัวอักษร 1 ตัว
bool	ค่าความจริง true/false

ชนิดข้อมูล	ความหมาย
string	ข้อความหลายตัวอักษร

☐ **เงื่อนไข (if / else / switch)**

ตัวอย่าง if-else แบบซ้อน

```
int score = 75;
if (score >= 80)
    cout << "เกรด A";
else if (score >= 70)
    cout << "เกรด B";
else
    cout << "เกรด C";
```

ตัวอย่าง switch-case

```
char grade = 'B';
switch (grade) {
    case 'A': cout << "ดีเยี่ยม"; break;
    case 'B': cout << "ดี"; break;
    case 'C': cout << "พอใช้"; break;
    default: cout << "ปรับปรุง";
}
```

☐ **ลูป (for / while / do-while)**

```
// for loop: นับ 1 ถึง 5
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}

// while loop
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

```
// do-while: ทำก่อนตรวจเงื่อนไข
int x = 1;
do {
    cout << x << " ";
    x++;
} while (x <= 5);
```

☐ **3. ฟังก์ชันและ Parameter**

☐ **แบบมีพารามิเตอร์และคืนค่า**

```
int multiply(int a, int b) {
    return a * b;
}

int main() {
    cout << multiply(3, 4); // 12
}
```

☐ **แบบไม่มีพารามิเตอร์**

```
void greet() {
    cout << "สวัสดีครับ!";
}
```

☐ **Pass by Value vs Pass by Reference**

```
void modifyByValue(int x) {
    x = 100;
}

void modifyByRef(int& x) {
    x = 100;
}

int main() {
    int num = 10;
    modifyByValue(num);   // ไม่เปลี่ยน
    cout << num << endl;  // 10
```

```
    modifyByRef(num);      // เปลี่ยน
    cout << num << endl;  // 100
}
```

☐ **4. Pointer & Reference**

☐ **Pointer**

```
int x = 10;
int* p = &x;      // p เก็บ address ของ x

cout << *p;       // แสดงค่า x (10)
```

☐ **Reference**

```
int a = 5;
int& ref = a;     // ref ชี้มาที่ a

ref = 20;         // a ถูกเปลี่ยนเป็น 20
```

☐ **5. การรับ/แสดงผล Input/Output**

☐ **cin และ cout**

```
int age;
cout << "Enter age: ";
cin >> age;

cout << "Your age is: " << age;
```

☐ **getline() สำหรับข้อความหลายคำ**

```
string fullName;
cout << "Enter your full name: ";
getline(cin, fullName); // รับข้อความรวมช่องว่าง

cout << "Hello, " << fullName << endl;
```

☐ **เทคนิคเสริม: การใช้ cin.ignore() กับ getline**

```
int age;
string name;
```

```
cin >> age;
cin.ignore(); // ล้าง newline ที่ค้างจาก cin
getline(cin, name);
```

☐ **สรุปเพิ่มเติม**

หัวข้อพื้นฐาน	ใช้ใน OOP อย่างไร
function	พื้นฐานของ method ใน class
pointer / reference	ใช้ใน constructor, dynamic memory, polymorphism
cin, cout	ใช้รับส่งค่าระหว่าง object กับ user
loop, condition	ใช้ใน method ของ class
string, int, bool	ใช้เป็นสมาชิกข้อมูลของคลาส (data members)

เครื่องมือที่ต้องใช้ในการเขียน C++

☐ **Compiler (ตัวแปลภาษา)**

- แปลงไฟล์ .cpp ไปเป็นไฟล์ .exe หรือ .out
- ตัวอย่าง compiler ที่นิยม:
  - g++ (GNU Compiler)
  - clang++
  - MSVC (Microsoft Visual C++)

☐ **IDE หรือ Code Editor**

- โปรแกรมที่ใช้เขียนโค้ด
- ตัวอย่าง:
  - **Code::Blocks** (รวม compiler แล้ว)
  - **Dev-C++**
  - **Visual Studio**
  - **Visual Studio Code** (ต้องติดตั้ง g++)
  - **CLion** (ของ JetBrains)
  - **Xcode** (สำหรับ macOS)

☐ **สำหรับผู้ใช้ Windows**

☐ **วิธีที่ 1: ติดตั้ง Code::Blocks (แนะนำสำหรับผู้เริ่มต้น)**

1. ไปที่เว็บไซต์:
   - https://www.codeblocks.org/downloads/
2. เลือกดาวน์โหลด **"codeblocks-20.03mingw-setup.exe"**
   (มี compiler g++ ในตัว ไม่ต้องติดตั้งแยก)
3. ติดตั้งตามขั้นตอน (Next ⟶ Next ⟶ Install)
4. เปิดโปรแกรม ⟶ สร้างโปรเจกต์ใหม่:
   - File ⟶ New ⟶ Project ⟶ Console Application ⟶ เลือก C++
   - เขียนโค้ด แล้วกด Build & Run (F9)

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, C++ on Windows!" << endl;
    return 0;
}
```

วิธีที่ 2: ใช้ Visual Studio Code + g++

1. ติดตั้ง **VS Code**:
   - https://code.visualstudio.com
2. ติดตั้ง **MinGW (g++)**:
   - ดาวน์โหลดจาก: https://www.mingw-w64.org
   - เมื่อติดตั้งเสร็จ ต้องเพิ่ม path ของ bin ลงใน Environment Variables
3. ตรวจสอบว่า g++ ทำงานหรือไม่:
4. g++ --version
5. เปิด VS Code ⟶ ติดตั้ง Extension: C/C++ จาก Microsoft
6. เขียนโค้ดและรันด้วย:
7. g++ main.cpp -o main
8. ./main

สำหรับผู้ใช้ macOS

1. เปิด Terminal แล้วติดตั้ง Xcode Command Line Tools:
2. xcode-select --install
3. เขียนไฟล์ main.cpp ด้วย Text Editor หรือ VS Code
4. Compile และ Run:

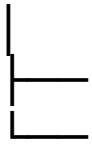
5. g++ main.cpp -o main
6. ./main

☐ **สำหรับผู้ใช้ Linux (Ubuntu/Debian)**

1. ติดตั้ง g++ compiler:
2. sudo apt update
3. sudo apt install g++
4. สร้างไฟล์ชื่อ main.cpp แล้วเขียนโค้ด
5. Compile และ Run:
6. g++ main.cpp -o main
7. ./main

☐ **โครงสร้างไฟล์ที่แนะนำ**

MyCppProject/



main.cpp ← ไฟล์หลัก

[อื่นๆในอนาคต เช่น class.cpp, utils.h]

☐ **ตัวอย่างโปรแกรมแรก**

```
#include <iostream>
using namespace std;

int main() {
    cout << "สวัสดี C++!" << endl;
    return 0;
}
```

Compile แล้วรัน:

```
g++ main.cpp -o hello
./hello
```

☐ **เคล็ดลับเพิ่มเติม**

เคล็ดลับ	ประโยชน์
ใช้ Ctrl + S บ่อยๆ	บันทึกไฟล์ก่อน compile

เคล็ดลับ	ประโยชน์
ตั้งชื่อไฟล์เป็น .cpp	ระบบจะรู้ว่าเป็น C++
ใช้ endl หรือ \n	ขึ้นบรรทัดใหม่
ตรวจสอบ error ที่ compile บอก	C++ ต้องแม่น syntax

ตัวอย่างโปรแกรม C++

☐ **1. ตัวแปร, เงื่อนไข (if/switch), และลูป (for/while)**

☐ **โค้ด:**

```
#include <iostream>
using namespace std;

int main() {
    int age;
    char grade;

    cout << "กรุณากรอกอายุของคุณ: ";
    cin >> age;

    if (age >= 18) {
        cout << "คุณเป็นผู้ใหญ่" << endl;
    } else {
        cout << "คุณยังเป็นเยาวชน" << endl;
    }

    cout << "กรุณากรอกเกรด (A/B/C): ";
    cin >> grade;

    switch (grade) {
        case 'A': cout << "เยี่ยมมาก!" << endl; break;
        case 'B': cout << "ดีมาก!" << endl; break;
        case 'C': cout << "พอใช้!" << endl; break;
        default: cout << "ไม่รู้จักเกรดนี้!" << endl;
```

```
    }

    cout << "นับเลขจาก 1 ถึง 5 ด้วย for loop: ";
    for (int i = 1; i <= 5; i++) {
        cout << i << " ";
    }
    cout << endl;

    cout << "นับเลขจาก 5 ถึง 1 ด้วย while loop: ";
    int j = 5;
    while (j >= 1) {
        cout << j << " ";
        j--;
    }

    return 0;
}
```

ผลลัพธ์ (เมื่อป้อนอายุ 20 และเกรด B):

```
กรุณากรอกอายุของคุณ: 20
คุณเป็นผู้ใหญ่
กรุณากรอกเกรด (A/B/C): B
ดีมาก!
นับเลขจาก 1 ถึง 5 ด้วย for loop: 1 2 3 4 5
นับเลขจาก 5 ถึง 1 ด้วย while loop: 5 4 3 2 1
```

---

2. ฟังก์ชันและ Parameter

โค้ด:

```
#include <iostream>
using namespace std;

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

```
// ฟังก์ชันไม่คืนค่า ใช้แสดงผล
void greet(string name) {
    cout << "สวัสดีคุณ " << name << "!" << endl;
}

int main() {
    int x = 10, y = 20;
    cout << "ผลบวกของ " << x << " และ " << y << " คือ: " << add(x, y) << endl;

    greet("สมชาย");

    return 0;
}
```

ผลลัพธ์:

```
ผลบวกของ 10 และ 20 คือ: 30
สวัสดีคุณ สมชาย!
```

---

3. Pointer & Reference เบื้องต้น

โค้ด:

```
#include <iostream>
using namespace std;

void modifyValue(int* ptr) {
    *ptr = 100;
}

void modifyByReference(int& ref) {
    ref = 200;
}

int main() {
    int num = 50;
```

```
    cout << "ค่าก่อนเปลี่ยน: " << num << endl;

    modifyValue(&num);  // ส่ง pointer
    cout << "หลังจาก modifyValue: " << num << endl;

    modifyByReference(num);  // ส่งโดย reference
    cout << "หลังจาก modifyByReference: " << num << endl;

    return 0;
}
```

 ผลลัพธ์:

ค่าก่อนเปลี่ยน: 50

หลังจาก modifyValue: 100

หลังจาก modifyByReference: 200

 คำอธิบาย:

- *ptr = 100; เปลี่ยนค่าที่ pointer ชี้อยู่ (ตัวแปร num)
- ref = 200; เป็นอีกชื่อหนึ่งของ num เลยเปลี่ยนค่าจริง

---

 4. การจัดการ Input/Output (cin, cout, getline)

 โค้ด:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string fullName;
    int age;

    cout << "กรุณากรอกชื่อ-นามสกุล: ";
    getline(cin, fullName); // รับชื่อเต็ม

    cout << "กรุณากรอกอายุ: ";
    cin >> age;
```

```
    cout << "ยินดีต้อนรับคุณ " << fullName << " อายุ " << age << " ปี" << endl;

    return 0;
}
```

☐ **ตัวอย่างผลลัพธ์:**

```
กรุณากรอกชื่อ-นามสกุล: สมชาย ใจดี
กรุณากรอกอายุ: 25
ยินดีต้อนรับคุณ สมชาย ใจดี อายุ 25 ปี
```

☐ **สรุปภาพรวม**

หัวข้อ	ความสามารถหลักที่ควรรู้
ตัวแปร & เงื่อนไข	ควบคุมการตัดสินใจในโปรแกรม
ลูป	ทำงานซ้ำแบบอัตโนมัติ
ฟังก์ชัน	แยกงานให้เป็นระบบ ส่งค่าเข้าและคืนค่าได้
Pointer	ใช้ควบคุมหน่วยความจำโดยตรง
Reference	เปลี่ยนค่าตัวแปรได้สะดวกแบบปลอดภัย
cin / cout / getline	รับข้อมูลจากผู้ใช้ แสดงผลอย่างยืดหยุ่น

สรุป

บทนี้เป็นจุดเริ่มต้นสำคัญของการเรียนรู้ C++ เพื่อก้าวไปสู่การเขียนโปรแกรมเชิงวัตถุ (OOP) โดยผู้เรียนจะได้ทบทวนและเข้าใจพื้นฐานของภาษา C++ ตั้งแต่โครงสร้างโปรแกรมและการคอมไพล์ การใช้งานตัวแปร เงื่อนไข (if, switch) และลูป (for, while) ฟังก์ชันและการส่งค่าผ่าน parameter ตลอดจนแนวคิดและการใช้งาน pointer และ reference เบื้องต้น นอกจากนี้ยังครอบคลุมการจัดการ Input/Output ด้วย cin, cout และ getline เพื่อให้ผู้เรียนมีพื้นฐานที่แข็งแรงและพร้อมต่อยอดไปสู่การเรียนรู้และพัฒนาโปรแกรมแบบ OOP ได้อย่างมั่นใจและเป็นระบบ

บทที่ 2
แนวคิดพื้นฐานของ OOP และ UML
(OOP Basic and UML)

เนื้อหา

- แนวคิดพื้นฐานของ OOP
- แนวคิดพื้นฐานของ OOP เชิงลึก
- แนวคิดของ Class&Object
- แนวคิดของ Object และ Class — ระดับแก่น (Conceptual Depth)
- ทำไม OOP จึงสำคัญมาก ในการพัฒนาโปรแกรมขนาดใหญ่?
- ความสำคัญของ OOP ในการพัฒนาโปรแกรมขนาดใหญ่
- UML (Unified Modeling Language) คืออะไร และทำไมถึงเป็น “เครื่องมือออกแบบ” ที่สำคัญของ OOP
- Class Diagram
- Class Diagram เชิงลึกที่สุดใน UML

บทนำ

การเขียนโปรแกรมเป็นทั้งศาสตร์และศิลป์ที่ต้องอาศัยแนวคิดและโครงสร้างที่ชัดเจน เพื่อให้ผู้พัฒนาและผู้ดูแลรักษาโปรแกรมเข้าใจและควบคุมได้ บทนี้จะพาผู้อ่านไปทำความรู้จักแนวคิดพื้นฐานของ OOP (Object-Oriented Programming) ตั้งแต่จุดเริ่มต้นของแนวคิด จนถึงการออกแบบและพัฒนาโปรแกรมขนาดใหญ่ที่ต้องรองรับผู้ใช้จำนวนมากและมีความซับซ้อนสูง

หัวข้อแรกคือการเปรียบเทียบแนวคิดของ Programming Paradigm ระหว่างรูปแบบดั้งเดิมแบบ Procedural กับ Object-Oriented โดยผู้เรียนจะเห็นข้อแตกต่างทั้งในมุมมองของการออกแบบโปรแกรม การควบคุมข้อมูลและหน้าที่ รวมถึงข้อดีของ OOP ที่ช่วยให้ผู้พัฒนารวมข้อมูล (Data) และหน้าที่ (Function) เป็นหน่วยเดียวกัน จัดการและบำรุงรักษาได้ง่ายขึ้น และแก้ปัญหาการเขียนโปรแกรมแบบเก่าที่ต้องเจอกับโครงสร้างโค้ดขนาดใหญ่และสลับซับซ้อน

ผู้เรียนจะได้ทำความเข้าใจกับแนวคิดของ Object และ Class หัวใจของ OOP โดย Object เป็นตัวแทนของสิ่งที่ต้องแก้ปัญหาหรืออ้างอิงถึงในโปรแกรม และ Class เป็นพิมพ์เขียวของ Objects เหล่านั้น การเรียนรู้เรื่องนี้ช่วยให้ผู้พัฒนาเข้าใจและเลือกใช้ Encapsulation เพื่อปกป้องข้อมูลและควบคุมการเข้าถึงได้อย่างเป็นระบบและเป็นสัดเป็นส่วน

บทนี้ยังเน้นถึงความสำคัญของ OOP ในการพัฒนาโปรแกรมขนาดใหญ่ ด้วยข้อดีทั้งในด้านการออกแบบและบริหารจัดการระบบ ช่วยลดข้อขัดแย้งของโค้ด รองรับการทำงานเป็นทีม และทำให้โปรแกรมมีโครงสร้างชัดเจน สามารถแก้ไข ปรับปรุง และขยายต่อได้อย่างมีประสิทธิภาพทั้งในปัจจุบันและอนาคต

สุดท้ายผู้เรียนจะได้รู้จัก UML (Unified Modeling Language) ซึ่งเป็นเครื่องมือสำคัญของนักพัฒนา OOP ในการออกแบบและสื่อสารโครงสร้างโปรแกรมทั้งในภาพใหญ่และภาพย่อย UML ช่วยให้ผู้เรียนเข้าใจกระบวนการออกแบบคลาส วัตถุ และความสัมพันธ์ขององค์ประกอบต่าง ๆ ได้เป็นมาตรฐานและเป็นสากล เป็นรากฐานสำคัญสู่การเป็นโปรแกรมเมอร์ C++ OOP มืออาชีพต่อไป

แนวคิดพื้นฐานของ OOP

☐ **Programming Paradigm: Procedural vs Object-Oriented**

☐ **Procedural Programming (การเขียนโปรแกรมเชิงโครงสร้าง / เชิงลำดับ)**

- ใช้แนวคิดการแบ่งโปรแกรมเป็นฟังก์ชันหรือโปรซีเยอร์ (procedure / function)
- เน้นการทำงานเป็นลำดับขั้นตอน (sequence of instructions)
- ข้อมูล (data) และฟังก์ชันแยกจากกัน
- ตัวอย่างภาษา: C, Pascal

ตัวอย่างแนวคิด:

```
// C style
int add(int a, int b) {
    return a + b;
}
```

การประมวลผลจะไหลจากบนลงล่าง เรียกใช้ฟังก์ชันเป็นลำดับ

☐ **Object-Oriented Programming (การเขียนโปรแกรมเชิงวัตถุ)**

- แบ่งโปรแกรมเป็นหน่วยย่อยที่เรียกว่า **object**
- object ประกอบด้วย **data (attributes)** + **methods (operations)** อยู่ด้วยกัน
- เน้นการจัดกลุ่มข้อมูลและพฤติกรรมที่เกี่ยวข้องเข้าด้วยกัน (Encapsulation)
- สนับสนุน **inheritance, polymorphism, abstraction**

ตัวอย่างแนวคิด:

```
// C++ style
class Calculator {
public:
    int add(int a, int b) {
        return a + b;
```

```
    }
};
```

ทุกอย่างถูกรวมอยู่ใน class ที่เป็นตัวแทนของ concept หรือสิ่งของ

☐ **แนวคิดของ Object และ Class**

☐ **Class**

- คือ *แม่พิมพ์* หรือ *แบบพิมพ์เขียว* (blueprint)
- กำหนดว่าข้อมูลอะไรและพฤติกรรมอะไรบ้างที่วัตถุประเภทนั้นต้องมี
- ข้างใน class ประกอบด้วย:
    - **Attributes (data members)** — ข้อมูล
    - **Methods (member functions)** — การกระทำ

ตัวอย่าง:

```
class Car {
public:
    string brand;
    int year;

    void start() {
        cout << brand << " started.\n";
    }
};
```

☐ **Object**

- คือ **อินสแตนซ์** (instance) หรือ **ตัวแทนที่สร้างขึ้นจริง** จาก class
- แต่ละ object มี data ของตัวเอง

ตัวอย่าง:

```
Car myCar;
myCar.brand = "Toyota";
myCar.year = 2020;
myCar.start();  // แสดง: Toyota started.
```

☐ **ความสำคัญของ OOP ในการพัฒนาโปรแกรมขนาดใหญ่**

☐ **เหตุผลหลักที่ OOP สำคัญสำหรับโปรแกรมขนาดใหญ่:**

1☐ **จัดโครงสร้างโปรแกรมให้เป็นระบบ**

- OOP ทำให้ซอฟต์แวร์แบ่งออกเป็นหน่วยย่อย (class / object) ซึ่งทำให้เข้าใจและจัดการง่าย
- ตัวอย่าง: ในระบบ e-commerce อาจมี User, Product, Order, Cart ซึ่งแต่ละ class ดูแลหน้าที่ตัวเอง

2 **ลดความซับซ้อน และเพิ่มการใช้ซ้ำ**

- Class และ object ออกแบบให้ใช้ซ้ำได้ (Reusable)
- Code ที่เขียนครั้งเดียว สามารถนำไปใช้ซ้ำในส่วนอื่นได้

3 **ขยายและบำรุงรักษาง่าย**

- เมื่อระบบใหญ่ขึ้น OOP ช่วยให้เพิ่ม feature ใหม่โดยไม่กระทบส่วนอื่นมาก
- การแก้ไขบั๊กหรือเปลี่ยนแปลง logic เฉพาะจุดง่ายกว่า

4 **สนับสนุนหลักการ abstraction และ encapsulation**

- ซ่อนรายละเอียดภายใน class ให้ผู้ใช้สนใจแค่สิ่งจำเป็น
- ลดโอกาสเกิดข้อผิดพลาดเพราะเข้าถึงข้อมูลโดยตรง

5 **เหมาะกับทีมพัฒนา**

- แต่ละทีมสามารถพัฒนา class หรือ module ของตัวเองได้ โดยไม่ต้องเข้าใจระบบทั้งหมดในเชิงลึก

สรุปภาพรวม

ประเด็น	Procedural	OOP
โครงสร้าง	ฟังก์ชัน + ข้อมูลแยกกัน	ข้อมูล + ฟังก์ชันรวมใน object
การ reuse	reuse ฟังก์ชันได้ยากขึ้นในระบบใหญ่	class / object reuse ได้สูง
การจัดการความซับซ้อน	ควบคุมยากเมื่อโค้ดยาว	ควบคุมง่ายเพราะแยกเป็น class
ขยายระบบ	ลำบาก	สะดวก ด้วย inheritance และ polymorphism

แนวคิดพื้นฐานของ OOP เชิงลึก

---

1 Programming Paradigm: Procedural vs Object-Oriented

การเลือก **รูปแบบการเขียนโปรแกรม (Programming Paradigm)** มีผลโดยตรงต่อ:

- โครงสร้างของโปรแกรม
- วิธีคิดของผู้พัฒนา

- วิธีแก้ไขปัญหาต่าง ๆ

Procedural Programming

- **ลักษณะเด่น:**
  - โค้ดถูกจัดเป็น **ฟังก์ชัน** (functions) และ **โปรซีเยอร์** (procedures)
  - ข้อมูล (data) ถูกเก็บเป็นตัวแปรอิสระ แยกจากฟังก์ชัน
  - ฟังก์ชันต้องรู้โครงสร้างของข้อมูลโดยตรง
- **ข้อเสีย:**
  - เมื่อโปรแกรมใหญ่ขึ้น โครงสร้างจะยุ่งเหยิง (spaghetti code)
  - การแก้ไขต้องเข้าไปแก้ทั้งในฟังก์ชันและส่วนของข้อมูล
- **ตัวอย่าง:**

```
// Procedural
struct Point {
    int x;
    int y;
};

void movePoint(Point *p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}
```

ข้อมูล (Point) และ ฟังก์ชัน (movePoint) แยกกันโดยสิ้นเชิง

Object-Oriented Programming (OOP)

- **ลักษณะเด่น:**
  - รวม **ข้อมูล (Attributes)** และ **พฤติกรรม (Methods)** เป็นหน่วยเดียวกันคือ **Class**
  - ข้อมูลและเมธอดอยู่ภายในอ็อบเจกต์เดียวกัน ⟶ มี **Encapsulation**
- **ข้อดี:**
  - จัดการง่าย แม้โปรแกรมใหญ่ขึ้น
  - โครงสร้างเป็นโมดูล (Module) แต่ละคลาสมีหน้าที่ของตัวเอง
  - รองรับแนวคิด **Reusability**, **Extensibility**, และ **Maintainability**
- **ตัวอย่าง:**

```
// OOP
class Point {
public:
```

```
    int x, y;

    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
};
```

Point เป็นทั้งข้อมูลและเมธอดอยู่ในคลาสเดียวกัน

---

2 แนวคิดของ Object และ Class

Class

- เป็น **แม่แบบ** (blueprint) หรือต้นแบบของอ็อบเจกต์
- กำหนดทั้ง:
    - **Attributes** (ข้อมูล) — เช่น ชื่อ, ราคา
    - **Methods** (พฤติกรรม) — เช่น คำนวณราคา, พิมพ์ใบเสร็จ
- เป็นตัวแทนของสิ่งของจริง ๆ หรือแนวคิด

Object

- เป็น **Instance** ของ Class
- แต่ละอ็อบเจกต์มีค่าของ Attribute ของตัวเอง
- ตัวอ็อบเจกต์คือ **สิ่งที่เราสร้างและใช้งานจริง**

ตัวอย่างเปรียบเทียบ

Class: Shirt

Attributes: size, color, price

Methods: buy(), showDetails()

Object:

- Shirt shirt1("L", "red", 299);

- Shirt shirt2("M", "blue", 199);

---

3 ความสำคัญของ OOP ในการพัฒนาโปรแกรมขนาดใหญ่

3.1 จัดโครงสร้างโปรแกรมเป็นหน่วยย่อย

โปรแกรมใหญ่ ๆ มีโครงสร้างซับซ้อน แต่ OOP จัดการได้โดย:

- แบ่งเป็นคลาสต่าง ๆ
- แต่ละคลาสดูแลหน้าที่เฉพาะ

- ตัวอย่าง: ระบบร้านค้าออนไลน์
- User
- Product
- Order
- Payment
- ShoppingCart

แต่ละคลาสเข้าใจได้โดยตรงว่าเกี่ยวกับหน้าที่ใด

3.2 Encapsulation (การห่อหุ้ม)

- ซ่อนรายละเอียดภายในคลาส (internal details)
- เผยแพร่เฉพาะสิ่งจำเป็นผ่าน public methods
- ผลลัพธ์:
  - คนใช้คลาสต้องรู้แค่ “ต้องเรียกเมธอดใด” ไม่ต้องรู้ว่า “ทำงานภายในอย่างไร”

3.3 Inheritance (การสืบทอด)

- คลาสใหม่ (subclass) สร้างขึ้นจากคลาสเดิม (parent class)
- นำคุณสมบัติเดิมกลับมาใช้ และขยายต่อได้
- ประโยชน์:
  - ลดการเขียนโค้ดซ้ำ
  - ปรับแก้เฉพาะส่วนใหม่
- ตัวอย่าง:
- class Animal { ... };
- class Dog : public Animal { ... };
- class Cat : public Animal { ... };

Dog และ Cat มีทั้งคุณสมบัติของ Animal และคุณสมบัติใหม่ของตัวเอง

3.4 Polymorphism (พหุรูป)

- เรียกใช้เมธอดเดียวกัน แต่ทำงานแตกต่างกันไปตามอ็อบเจกต์
- ประโยชน์:
  - รองรับการขยายโปรแกรมโดย **ไม่ต้องแก้โค้ดเก่า**
  - เช่น
  - Animal* a1 = new Dog();
  - Animal* a2 = new Cat();
  -
  - a1->speak(); // ผลลัพธ์: "Woof!"
  - a2->speak(); // ผลลัพธ์: "Meow!"

3.5 Abstraction (นามธรรม)

- ให้ผู้ใช้เข้าใจว่า **ต้องทำอะไร** แต่ **ไม่ต้องรู้ว่าทำอย่างไร**
- ตัวอย่าง:
- class Payment {
- public:
- virtual void pay() = 0; // abstract method
- };

ผู้ใช้เลือกคลาสจริง เช่น CreditCardPayment หรือ PaypalPayment แต่ใช้เมธอด pay() เหมือนกัน

สรุป

ประเด็น	Procedural	OOP
โครงสร้าง	ฟังก์ชัน + ข้อมูลแยกกัน	คลาส = ข้อมูล + ฟังก์ชัน
แนวคิด	จัดการตามลำดับ	จัดการเป็นอ็อบเจกต์
การขยายและบำรุงรักษา	ยากขึ้นเรื่อย ๆ	จัดการได้เป็นระบบ
การใช้ซ้ำ	ฟังก์ชันใช้ซ้ำได้ แต่ต้องจัดการข้อมูลเอง	คลาส/อ็อบเจกต์ใช้ซ้ำได้เต็มรูปแบบ
การรองรับโปรเจ็กต์ใหญ่	มีข้อจำกัด	ดีมาก (Encapsulation, Inheritance, Polymorphism)

⚡ **ผลลัพธ์ของ OOP ในงานจริง**

- โค้ดอ่านง่าย / แก้ไขสะดวก
- รองรับการขยายระบบขนาดใหญ่
- ประหยัดเวลาและลดข้อผิดพลาด
- มีมาตรฐาน จัดการเป็นระบบ (Modern Development)

แนวคิดของ Class&Object

---

1. Class คืออะไร?

Class = *"แม่แบบ"* (Blueprint)

- เป็นโครงร่างที่ **อธิบายคุณสมบัติ (Attributes)** และ **พฤติกรรม (Methods)** ของสิ่ง ๆ หนึ่ง

- เป็น *นามธรรม* (Abstract Concept) ไม่ใช่สิ่งจริง แต่เป็น *ต้นแบบ* ให้สร้างสิ่งจริงได้
- ตัวอย่างในชีวิตจริง:
    - **Class = พิมพ์เขียวของบ้าน**

      มีการอธิบายว่า:
        - มีห้องนอนกี่ห้อง
        - มีห้องน้ำกี่ห้อง
        - มีพื้นที่เท่าไร

      แต่ตัว “พิมพ์เขียว” นั้นเองยัง **อาศัยอยู่ไม่ได้** เพราะเป็นเพียงแบบแปลน

ตัวอย่างในโค้ด:

```
class Car {
public:
    // Attribute: ข้อมูลเกี่ยวกับรถ
    std::string brand;
    std::string color;
    int year;

    // Method: พฤติกรรมของรถ
    void start() {
        std::cout << brand << " is starting.\n";
    }
};
```

☐ ตรงนี้ Car เป็น Class

มีข้อมูล (brand, color, year)

มีพฤติกรรม (start())

☐ **2. Object คืออะไร?**

☐ **Object** = *“สิ่งจริง”*

- เป็น **อินสแตนซ์ (Instance)** ของ Class
- มีค่า **Attributes** ของตัวเอง
- มี **Methods** เพื่อควบคุมการทำงาน
- ตัวอย่างในชีวิตจริง:
    - **Object = บ้านจริง**
        - ตั้งอยู่จริงบนที่ดิน
        - มีเลขที่บ้าน มีผู้อยู่อาศัย

- มีห้องจริง ๆ ที่ใช้งานได้

ตัวอย่างในโค้ด:

```
int main() {
    // สร้าง object จาก class Car
    Car car1;
    car1.brand = "Toyota";
    car1.color = "Red";
    car1.year = 2022;

    car1.start();
    // Output: Toyota is starting.

    Car car2;
    car2.brand = "Honda";
    car2.color = "Blue";
    car2.year = 2023;

    car2.start();
    // Output: Honda is starting.

    return 0;
}
```

☐ car1 และ car2 เป็น **Object** ที่ถูกสร้างจาก **Class** Car แต่มีค่าของตนเอง

☐ **3. สรุปความแตกต่าง**

หัวข้อ	Class	Object
สถานะ	พิมพ์เขียว / แม่แบบ	สิ่งจริง / ผลลัพธ์
ตัวตน	ไม่มีตัวตนจริง	มีตัวตนจริง
ประกอบด้วย	Attributes และ Methods (ต้องนิยาม)	ค่าของ Attributes และ Methods
ตัวอย่างจริง	แบบแปลนของบ้าน	บ้านจริง ๆ
ในโค้ด	class Car { ... };	Car car1;

☐ **4. เหตุผลและข้อดีของแนวคิด Class-Object**

☐ จัดกลุ่มข้อมูล + พฤติกรรมเป็นหน่วยเดียวกัน ⟶ จัดการง่าย

☐ ส่งเสริม **Encapsulation** (ห่อหุ้ม) ข้อมูลและเมธอดให้อยู่ร่วมกัน

☐ ส่งเสริม **Reusability** (ใช้ซ้ำ) เพราะ Class เป็นแม่แบบให้นำไปสร้าง Object กี่ตัวก็ได้

☐ ปรับแก้และขยายโปรแกรมได้โดย:

- ปรับแก้ Class ⟶ Objects ทุกตัวอัปเดตอัตโนมัติ
- สร้าง Class ใหม่โดยใช้คุณสมบัติของ Class เดิม (**Inheritance**)

☐ ตัวอย่างแนวคิดภาพใหญ่

Class

Class: Animal

Attributes:

- name
- age

Methods:

- eat()
- sleep()

Objects

dog = Animal()

name = "Rex"

age = 3

cat = Animal()

name = "Kitty"

age = 2

☐ สรุปเป็นภาพ

- ⚡☐ **Class** = แม่แบบ, แนวคิด
- ☐ **Object** = สิ่งจริง, มีตัวตน
- ☐ Class กำหนดว่า Object “ต้องเป็นแบบไหน” แต่ Object เป็นผู้ “ใช้จริง” และ “เก็บค่าต่าง ๆ”

แนวคิดของ Object และ Class — ระดับแก่น (Conceptual Depth)

---

1. Class เป็นมากกว่า “พิมพ์เขียว”

ในเชิงลึก Class ไม่ได้เป็นเพียง “พิมพ์เขียว” แต่เป็น **หน่วยของ Abstraction** (หน่วยของแนวคิด)

ในการออกแบบ OOP:

- เราจะ **เลือกกลุ่มของคุณลักษณะ (Attributes)** และ **พฤติกรรม (Methods)** มารวมกันเป็นคลาสเดียว
- เป้าหมายคือเพื่ออธิบาย “สิ่ง” หรือ “แนวคิด” หนึ่ง ๆ ในโดเมนปัญหาของโปรแกรม

ตัวอย่างเชิงลึก:

หากเรากำลังเขียนโปรแกรมบริหารห้องสมุด

- เราเลือกคลาส Book เพราะเป็น “แนวคิด” สำคัญในบริบทนี้
- Book มีทั้ง **Attributes** (title, author, year) และ **Methods** (borrow(), returnBook())

Class = “Concept” (แนวคิด) + “Implementation” (การเขียนโค้ด)

---

2. Object เป็น Instantiation of a Concept

Object คือ **ตัวตนจริง (Concrete Instance)** ที่ถูก “หล่อ” มาจากคลาส

Object มีลักษณะดังนี้:

- มี **State** (สถานะ) คือค่าของ Attributes ในช่วงเวลาหนึ่ง
- มี **Behavior** (พฤติกรรม) คือความสามารถของเมธอด
- มี **Identity** (อัตลักษณ์) ของตัวเอง (เช่น book1 และ book2 แม้เป็น Book แต่เป็นคนละเล่ม)

ตัวอย่างเชิงลึก:

คลาส Book:

```
class Book {
public:
    std::string title;
    std::string author;

    void read() {
        std::cout << title << " is being read.\n";
    }
};
```

อ็อบเจกต์ Book:

```
Book book1;
book1.title = "1984";
book1.author = "George Orwell";
```

Book book2;

book2.title = "The Hobbit";

book2.author = "J.R.R. Tolkien";

⚡ book1 และ book2 เป็น Objects ของคลาส Book แต่ทั้งสองมี state (title, author) ต่างกัน

---

3. ความลึกของแนวคิด:

a) Encapsulation (การห่อหุ้ม)

Object = **Data + Methods**

- แทนที่จะให้ “ข้อมูล” และ “ฟังก์ชัน” กระจายตัวไปทั่วโปรแกรม
- เราเลือกห่อทั้งสองสิ่งไว้ด้วยกันเป็น **Unit of Abstraction** (หน่วยของแนวคิด)

b) Abstraction (นามธรรม)

Class คือ **โมเดล** ของสิ่งใดสิ่งหนึ่ง

- ไม่ต้องอธิบายทุกรายละเอียด แต่เลือกอธิบาย **เฉพาะแก่นสำคัญ** ของสิ่งนั้น
- เช่น Car มี speed และ color แต่เราเลือกอธิบายเฉพาะสิ่งเหล่านี้ เพราะเป็นแก่น

c) Identity (อัตลักษณ์)

Object แต่ละตัวมี **ตำแหน่งในหน่วยความจำ** และมี state เฉพาะ

- แม้ Objects สองตัวจะมี Attribute เหมือนกัน แต่เป็นคนละอัตลักษณ์ (เช่น book1 != book2)

d) Polymorphism + Inheritance

- Class เป็นแม่แบบที่ช่วยให้นำแนวคิดมาต่อยอดเป็นคลาสใหม่ (Inheritance) และเลือกให้ Objects แต่ละตัวทำงานแตกต่างกันได้ (Polymorphism)

---

4. Class / Object กับมุมมอง Software Architecture

Class เป็นหน่วยเล็กสุดของ Architecture ใน OOP

- แต่ละ Class มีหน้าที่ชัดเจน
- Objects เป็น “ผู้ให้บริการ” ของหน้าที่นั้น ๆ

⚡ ตัวอย่าง:

Class User จัดการเรื่องผู้ใช้ (สมัครสมาชิก, เข้าสู่ระบบ)

Objects user1, user2 เป็นผู้ใช้จริง ๆ แต่ละคน

---

5. ผลลัพธ์: Class + Objects นำไปสู่...

- โครงสร้างโปรแกรมที่เป็น Modular
- ป้องกัน Side-Effects เพราะ Objects แยก State ของตนเอง
- รองรับการขยาย ปรับแก้ และทดสอบได้ง่าย
- เปิดโอกาสให้เลือกใช้ Patterns เช่น:

- Factory Pattern (ใช้คลาสเป็นผู้ผลิต Objects)
- Singleton Pattern (มี Objects ตัวเดียวทั้งโปรแกรม)

⚡ **บทสรุปสุดท้าย**

- **Class** = แนวคิด + โครงสร้าง + พฤติกรรม
- **Object** = สิ่งจริง + State + Identity
- ⚡ **ทั้งสองเป็นหัวใจของ OOP** ที่ช่วยให้:
  - จัดการโปรแกรมใหญ่ ๆ ได้
  - ป้องกันข้อผิดพลาด
  - นำไปสู่การคิดเป็นโมเดลใกล้เคียงโลกจริง
  - ปรับแก้และขยายได้เป็นระบบ

ทำไม OOP จึงสำคัญมาก ในการพัฒนาโปรแกรมขนาดใหญ่?

“OOP” เป็นมากกว่าการเขียนโปรแกรมด้วยคลาสและอ็อบเจกต์

แต่เป็น **แนวคิดและสถาปัตยกรรม** ที่ช่วยควบคุมและจัดการ **โปรแกรมใหญ่ ๆ** ให้:

- มีโครงสร้างชัดเจน
- แก้ไขและขยายได้ง่าย
- มีคุณภาพและลดข้อผิดพลาด

⚡ **1 จัดการ Complexity (ความซับซ้อน) ด้วย Encapsulation**

- ในโปรแกรมใหญ่ ฟังก์ชันและข้อมูลมักกระจายอยู่ทั่วทั้งโปรแกรม
- OOP รวมทั้งข้อมูลและพฤติกรรมไว้ด้วยกันเป็น **คลาส** (Encapsulation) ⟶ แต่ละคลาสเป็น “กล่อง” (black box) จัดการหน้าที่ของตัวเอง
- ผลลัพธ์: โค้ดเข้าใจง่าย แต่ละคลาสมีหน้าที่ชัดเจน

⚡ **2 รองรับ การขยายและแก้ไข (Maintainability)**

- โปรแกรมใหญ่ต้องมีการอัปเดตเป็นธรรมชาติ
- OOP ช่วยให้แก้ไขและขยายโดย:
  - ปรับแก้ **ภายในคลาส** (implementation) โดยไม่กระทบคลาสหรือโมดูลอื่น
  - สร้างคลาสใหม่โดย **สืบทอด (Inheritance)** จากคลาสเดิม
- ผลลัพธ์: ประหยัดเวลา ป้องกันบั๊กที่อาจเกิดจากแก้ตรงกลาง

⚡ **3 Reusability (การใช้ซ้ำ) ด้วย Classes**

- โปรแกรมใหญ่ต้องการโครงสร้างชิ้นเล็ก ๆ ที่ใช้ซ้ำได้
- OOP มีคลาสเป็น **Unit of Reusability**:
  - สร้างคลาสใหม่เพื่อใช้ในหลาย ๆ โครงการ
  - จัดเป็นไลบรารี (Library) / เฟรมเวิร์ก (Framework)
- □ ผลลัพธ์: ประหยัดเวลา เพิ่มคุณภาพ (Code Quality)

⚡□ **4.□ รองรับ Team Development (การทำงานเป็นทีม)**

- โครงการใหญ่ต้องการให้นักพัฒนา **แบ่งหน้าที่และทำงานขนานกันได้**
- OOP ช่วยโดย:
  - แบ่งระบบเป็นคลาส / โมดูล
  - แต่ละคนพัฒนาและทดสอบคลาสของตัวเองได้โดยอิสระ
- □ ผลลัพธ์: ประสิทธิภาพทีมสูงขึ้น ป้องกันการชนกันของโค้ด

⚡□ **5.□ สนับสนุน Polymorphism เพื่อรองรับการขยาย**

- โปรแกรมใหญ่ต้องรองรับการเพิ่มชนิดใหม่ ๆ
- Polymorphism ช่วยให้:
  - เขียนโค้ดโดยอ้างอิงคลาสแม่ แต่ใช้งานคลาสลูกได้
  - เช่น List<Shape> shapes รองรับทั้ง Rectangle, Circle, Triangle
- □ ผลลัพธ์: เปิดกว้างให้รองรับ Requirement ใหม่โดยไม่ต้องแก้โค้ดเดิม

⚡□ **6.□ รองรับ Design Patterns / Architecture**

- OOP เป็นพื้นฐานของ Pattern เช่น:
  - MVC, MVVM, MVP
  - Observer, Strategy, Command, Decorator ฯลฯ
- □ ผลลัพธ์: โครงงานใหญ่มี Pattern จัดระเบียบ มีคุณภาพ และบำรุงรักษาง่าย

⚡□ **7.□ ป้องกัน Code Duplication & Spaghetti Code**

- โค้ด Procedural มักเกิดปัญหาการทำงานซ้ำ ๆ จนควบคุมยาก
- OOP จัดกลุ่มและใช้การสืบทอด เพื่อหลีกเลี่ยงการเขียนโค้ดซ้ำ
- □ ผลลัพธ์: โค้ดเป็นระบบ เข้าใจง่าย มีข้อผิดพลาดน้อยลง

⚡□ **8.□ เป็นมาตรฐานอุตสาหกรรม**

- OOP เป็นหัวใจของภาษาและ Framework สมัยใหม่ เช่น:

- Java, C#, C++, Kotlin, Swift, Dart
- Framework เช่น .NET, Spring Framework, Laravel

- □ ผลลัพธ์: รองรับงานจริง และจ้างงานได้จริง

□ **สรุปสั้น ๆ**

⚡□ OOP ช่วยแก้โจทย์ใหญ่ ๆ ของงาน Software Development:

- จัดการ Complexity
- รองรับ Maintainability & Scalability
- Enables Reusability & Teamwork
- Supports Polymorphism & Patterns
- เป็นรากฐานของ Architecture ในงานจริง

□ **ภาพใหญ่ของ OOP ในโปรแกรมใหญ่**

โปรแกรมใหญ่ ๆ มีคลาสเป็นตึกเล็ก ๆ แต่ละตึกมีหน้าที่ชัดเจน มีผู้ดูแล มีหน้าต่าง มีประตู มีการเข้าออก

⟶ จัดการได้ ⟶ ปรับแก้ได้ ⟶ ขยายได้

“ใหญ่ แต่เป็นระเบียบ”

ความสำคัญของ OOP ในการพัฒนาโปรแกรมขนาดใหญ่

□ **1□ จัดการ Complexity ด้วย Encapsulation & Modularization**

□ **โจทย์ใหญ่:**

โปรแกรมขนาดใหญ่มีโค้ดเป็นหมื่นเป็นแสนบรรทัด หากไม่มีโครงสร้าง จะแก้ตรงไหนก็กระทบตรงนั้น ตัวโปรแกรมจะคล้าย **“ชามสปาเกตตี”** (Spaghetti Code)

□ **OOP แก้ด้วย Encapsulation:**

- จัดกลุ่มทั้ง **ข้อมูล (Attributes)** และ **พฤติกรรม (Methods)** เป็น **คลาส**
- แต่ละคลาสเป็น “กล่อง” มีหน้าที่ชัดเจน (Single Responsibility)

□□ *ผลลัพธ์:*

- โครงสร้างเป็น Modular ⟶ หาต้นตอและแก้ปัญหาง่าย
- คน 10–100 คนแก้โค้ดได้โดยไม่ชนกัน
- ป้องกันการ “ล้นขอบ” ของโค้ด (Code Spill)

□ **2□ ขยายโปรแกรมได้โดยไม่ต้องรื้อทิ้งทั้งระบบ**

ในงานจริง ฟีเจอร์ใหม่ต้องถูกเพิ่มตลอดอายุโปรเจกต์ แต่ต้องทำโดย:

- “**ไม่ต้องแก้ของเก่ามาก**”
- “**ลดความเสี่ยง**” ใน Production

 OOP สนับสนุนผ่าน:

- **Inheritance (การสืบทอด):**

  คลาสใหม่สืบทอดคุณสมบัติเดิม แต่เลือกเพิ่ม/แก้ไขได้
- **Polymorphism:**

  โค้ดเลือกทำงานตามคลาสจริง ณ Runtime
  - ตัวอย่าง:

    List<Shape> shapes = [new Circle(), new Square()];

    shapes.forEach(s -> s.draw());
  - ขยายเป็น new Polygon() ได้โดย “**ไม่ต้องแก้ draw() ในคลาสอื่น**”

---

 3 จัดการและควบคุม State ในระบบใหญ่

ในโปรแกรมใหญ่ แต่ละ “อ็อบเจกต์” มี **state (สถานะ)** ของตนเอง

- เช่น User มี username, email, roles
- Product มี price, category, quantity
- Order มี orderId, items, status

 OOP จัด state ของอ็อบเจกต์ให้เป็นเอกเทศ

- ป้องกัน “state bleed” ที่จะเกิดเมื่อใช้ตัวแปร global
- แต่ละอ็อบเจกต์บริหาร state ของตนเอง (Encapsulation) ⟶ **ควบคุมได้, ทดสอบได้**

⚡ **4 Reusability & Maintainability**

 ในโปรแกรมใหญ่:

- มีโค้ดต้องใช้ซ้ำเป็น 100+ ครั้ง
- แต่ต้องหลีกเลี่ยง **Duplicate Logic** (หายนะของ Maintenance)

 OOP ช่วยโดย:

- สร้าง **Base Class** + ใช้ Inheritance
- จัดกลุ่มเป็น **Library / Framework** ของตนเอง
- เช่น DatabaseHandler ตัวเดียวใช้ได้ทั้งโปรเจกต์

  ผลลัพธ์:

- โค้ดมี Single Source of Truth
- Maintenance น้อยลง

- เวลาแก้บั๊ก แก้ตรงเดียว มีผลทั้งระบบ

□ **5□ รองรับการทำงานเป็น ทีมใหญ่ (Collaboration)**

□ ในโปรเจกต์ใหญ่ มีโปรแกรมเมอร์ 10–100 คนต้องทำงานร่วมกัน

□ ถ้าไม่มี OOP: โค้ดชนกันตลอดเวลา

□ OOP จัดการโดย:

- จัดโครงโปรแกรมเป็น **คลาสและแพ็กเกจ**
- แต่ละคลาสเป็น “สัญญา” (contract) ของหน้าที่
- แต่ละทีมหรือผู้พัฒนาเลือกทำหน้าที่ในคลาสนั้น ๆ
- มี Interface / Abstract Class เป็น **ข้อตกลงกลาง** ในทีม

□□ ***ผลลัพธ์:***

- แต่ละทีมทำงานได้แบบ **เป็นอิสระ**
- บำรุงรักษาได้โดย “**รู้หน้าที่**” ไม่ต้องอ่านทั้งโปรเจกต์
- มี **Code Review / Testing** ที่เจาะตรงคลาสได้

□ **6□ เป็นรากฐานของ Architecture & Design Patterns**

□ OOP เป็นหัวใจของ Architecture:

- MVC / MVVM / MVP
- Domain-Driven Design
- Layers Architecture
- Pattern ต่าง ๆ เช่น Strategy, Observer, Command, Factory

□□ ***ผลลัพธ์:***

- โครงสร้างโปรแกรมใหญ่เข้าใจได้
- ปรับขยายได้ตรงตามโจทย์ใหม่ ๆ
- มี Pattern ที่ช่วยแก้ปัญหาที่พบบ่อยโดยตรง

□ **7□ ผลลัพธ์สุดท้าย: Robustness, Scalability, Maintainability**

□ OOP ไม่ใช่ “ไวยากรณ์” แต่เป็น **แนวคิดการออกแบบ**

□ ให้โปรแกรมใหญ่เป็นระบบ

□ ป้องกันการเสียหายเป็นวงกว้าง

□ รองรับผู้ใช้เป็นล้าน ๆ คน และผู้พัฒนาเป็นร้อย ๆ คน

⚡□ **ตัวอย่างจริง**

□ **Facebook / YouTube**

- มีผู้พัฒนาเป็นพัน ๆ คน
- ใช้ OOP + Patterns เพื่อควบคุม Complexity
- มี Services และ Objects เป็น Units จัดการอิสระ
- รองรับผู้ใช้เป็นพันล้านคนโดยโครงสร้างต้องชัดเจน

□ **Banking System**

- มีคลาสบัญชีผู้ใช้, บัญชีธุรกรรม, การแจ้งเตือน, การรักษาความปลอดภัย
- แต่ละคลาสมีหน้าที่ชัดเจนและอิสระ
- ปรับปรุงอัพเกรดได้โดย “แตะต้องคลาสที่เกี่ยวข้อง” เท่านั้น

⚡□ **สรุปแก่น**

□ OOP มีค่าต่อโปรแกรมใหญ่เพราะ:

- จัดการ Complexity ด้วย Encapsulation
- ขยายได้ด้วย Inheritance + Polymorphism
- จัดการ State ของ Objects แต่ละตัวเป็นอิสระ
- สนับสนุน Reusability + Maintainability
- เปิดทางให้ Architecture / Patterns จัดการโปรแกรมใหญ่
- เป็นแกนกลางของการพัฒนาเป็นทีมและต่อยอดระบบ

UML (Unified Modeling Language) คืออะไร และทำไมถึงเป็น “เครื่องมือออกแบบ” ที่สำคัญของ OOP

□ **1□ UML คืออะไร**

- **UML = Unified Modeling Language**
- เป็น **ภาษาภาพมาตรฐาน** (Visual Language) ที่ใช้เป็น **“แบบพิมพ์เขียว”** (Blueprint) เพื่ออธิบาย:
  - โครงสร้างของโปรแกรม
  - พฤติกรรมของโปรแกรม
  - ปฏิสัมพันธ์ขององค์ประกอบ (Objects, Classes)

□ **คล้ายแบบแปลนตึก** แต่เป็น **แบบแปลนของโปรแกรม**

⚡□ **2□ UML สำคัญต่อ OOP เพราะ...**

- OOP มีแนวคิดเป็น **object**, **class**, **interaction**

- UML เป็นเครื่องมือที่ช่วย:
  - สื่อสารแนวคิด OOP ให้เป็นภาพ
  - ช่วยทั้งผู้พัฒนา, ผู้ออกแบบ, ผู้บริหาร, และผู้เกี่ยวข้องเข้าใจกันตรงกัน
- เป็น **สะพาน** ระหว่าง **แนวคิด (concept)** ➔ **โค้ดจริง (implementation)**

---

3 ประโยชน์ของ UML ในการพัฒนาโปรแกรมขนาดใหญ่

ช่วยให้เห็นภาพใหญ่ (Big Picture):

- ชัดเจนว่าโปรแกรมมีกี่คลาส แต่ละคลาสเกี่ยวข้องกันอย่างไร

ช่วยจัดโครงสร้างโปรแกรม:

- แยกหน้าที่ (Responsibilities) ของแต่ละคลาส
- จัดสรรผู้พัฒนาให้รับผิดชอบตรงตามคลาส / โมดูล

ลดข้อขัดแย้งในการทำงานเป็นทีม:

- ทีมเห็นตรงกันเกี่ยวกับหน้าตาและหน้าที่ของคลาส / โมดูล

เป็นเอกสารอ้างอิง (Documentation):

- แม้ผู้พัฒนาเดิมจะย้ายไป คนใหม่อ่าน UML ก็เข้าใจโครงสร้างได้เร็ว

รองรับการออกแบบเชิงสถาปัตยกรรม:

- เช่น MVC, Layer Architecture, Domain-Driven Design (DDD)

---

4 ประเภทของ UML Diagram

ประเภท UML	ใช้ทำอะไร
⚡ **Class Diagram**	แสดงคลาส, Attribute, Method และ Relationship ของคลาสต่าง ๆ
⚡ **Object Diagram**	แสดง Objects จริง ๆ ณ ช่วงเวลาหนึ่ง
⚡ **Use Case Diagram**	แสดงผู้ใช้ (Actor) และ Use Cases ของระบบ
⚡ **Sequence Diagram**	แสดงลำดับการสื่อสาร (Message) ระหว่าง Objects ใน Use Case
⚡ **Activity Diagram**	แสดงลำดับกิจกรรม / ขั้นตอนใน Workflow
⚡ **State Diagram**	แสดงสถานะและการเปลี่ยนผ่านของ Objects
⚡ **Component Diagram**	แสดงองค์ประกอบเชิงโครงสร้างของระบบ
⚡ **Deployment Diagram**	แสดงการกระจายองค์ประกอบของระบบไปยัง Hardware/Environment

---

5 ตัวอย่างจริง

🔲 *Class Diagram*

ตัวอย่าง: ระบบห้องสมุด

Book

- title: String

- author: String

- ISBN: String

+ borrow()

+ return()

🔲 *Use Case Diagram*

ตัวอย่าง:

ผู้ใช้ ⟶ [ยืมหนังสือ], [คืนหนังสือ]

🔲 *Sequence Diagram*

ตัวอย่าง:

ผู้ใช้ ➔ BookController ➔ BookService ➔ BookRepository ➔ Database

---

⚡🔲 6. 🔲 UML และ OOP ผสานกันอย่างไร

🔲 OOP ให้: Objects, Classes, Methods, Attributes

🔲 UML ให้: Visual Representation ของสิ่งเหล่านี้

🔲 ผลลัพธ์:

OOP = แนวคิดและโครงสร้าง

UML = “ภาพ” และ “แบบ” ของแนวคิดและโครงสร้าง

⚡🔲 7. 🔲 ข้อดีของ UML ในงานจริง

- 🔲 **ลดข้อผิดพลาด**: เข้าใจก่อนเขียนโค้ดจริง
- 🔲 **รองรับ Agile / Scrum**: เป็นสื่อสารร่วมกันในทีม
- 🔲 **เป็นแบบเรียนรู้และอ้างอิงได้**: แม้ผู้พัฒนาใหม่เข้าร่วม
- 🔲 **ช่วยเลือก Patterns**: เช่น Observer, Strategy, Command

⚡🔲 สรุปแก่น

⚡🔲 **UML = Visual Language for OOP**

🔲 ช่วยให้นักพัฒนา “เห็นภาพตรงกัน” ก่อนลงมือเขียนโค้ดจริง

- จัดระบบการออกแบบและสื่อสารทั้งโครงสร้างและพฤติกรรมของโปรแกรม
- เป็นหัวใจของการพัฒนาโปรแกรมขนาดใหญ่ (Enterprise Systems)

Class Diagram

⚡ Class Diagram คืออะไร?

- เป็น **UML Diagram** ประเภทหนึ่งที่ใช้เพื่อ **แสดงโครงสร้างของระบบแบบเชิงวัตถุ (Object-Oriented)**
- แสดงรายละเอียดของ **คลาส (Class)** ในระบบ
- แสดง **Attributes (ข้อมูล/คุณสมบัติ)** ของแต่ละคลาส
- แสดง **Methods (ฟังก์ชัน/พฤติกรรม)** ที่คลาสนั้น ๆ สามารถทำได้
- แสดง **ความสัมพันธ์ (Relationships)** ระหว่างคลาสต่าง ๆ เช่น
  - **Association (การเชื่อมโยงทั่วไป)**
  - **Aggregation (ความเป็นส่วนประกอบแบบหลวม)**
  - **Composition (ความเป็นส่วนประกอบแบบแข็งแรง)**
  - **Inheritance (การสืบทอด)**

องค์ประกอบหลักของ Class Diagram

องค์ประกอบ	คำอธิบาย	ตัวอย่าง
Class	กล่องสี่เหลี่ยมแบ่งเป็น 3 ส่วน	Book
Attributes	รายการคุณสมบัติของคลาส	- title: String- author: String
Methods	รายการฟังก์ชันของคลาส	+ borrow(): void+ return(): void
Relationships	การเชื่อมโยงระหว่างคลาสต่าง ๆ	Inheritance, Association, Aggregation, Composition

---

โครงสร้างของ Class ใน Diagram

```
-------------------------
|        ClassName       |      <-- ชื่อคลาส
-------------------------
| - attributeName: Type  |      <-- Attributes (ข้อมูล)
| - ...                  |
-------------------------
```

| + methodName(): ReturnType | <-- Methods (พฤติกรรม)

| + ... |

- เครื่องหมาย - แสดงว่า **private** (ซ่อนข้อมูล)
- เครื่องหมาย + แสดงว่า **public** (เปิดเผยให้ใช้งานได้)

☐ **ตัวอย่าง Class Diagram**

สมมติระบบห้องสมุดมี 3 คลาสหลัก:

Class	Attributes	Methods	ความสัมพันธ์
Book	- title: String- author: String- ISBN: String	+ borrow(): void+ return(): void	Association กับ Member
Member	- name: String- memberId: int	+ register(): void+ borrowBook(): void	Association กับ Book
Library	- name: String- address: String	+ addBook(): void+ removeBook(): void	Aggregation ของ Book

แสดงใน Class Diagram

```
+------------------+          1       *      +------------------+
|       Member     |----------------------|        Book        |
+------------------+                       +------------------+
| - name: String   |                       | - title: String    |
| - memberId: int  |                       | - author: String   |
+------------------+                       +------------------+
| + register()     |                       | + borrow()         |
| + borrowBook()   |                       | + return()         |
+------------------+                       +------------------+


          1
   +---------------+
   |    Library    |
   +---------------+
   | - name:String |
   | - address:String|
   +---------------+
```

```
| + addBook()   |
| + removeBook()|
+---------------+
        *
      owns
        *
        |
      Book
```

- **Member** กับ **Book** มีความสัมพันธ์แบบ **Association** (สมาชิกยืมหนังสือได้หลายเล่ม)
- **Library** กับ **Book** มีความสัมพันธ์แบบ **Aggregation** (ห้องสมุดมีหนังสือเป็นส่วนประกอบ แต่หนังสือสามารถอยู่ได้เองโดยไม่ขึ้นกับห้องสมุด)

☐ **ความสัมพันธ์ใน Class Diagram แบบละเอียด**

ชนิดความสัมพันธ์	สัญลักษณ์ใน Diagram	ความหมาย
Association	เส้นตรง	การเชื่อมโยงทั่วไป ระหว่างคลาส 1 ตัวกับอีกตัว
Multiplicity	ตัวเลขที่ข้างเส้น	จำนวนของอ็อบเจกต์ที่เกี่ยวข้อง เช่น 1, 0.., *1..*
Inheritance	ลูกศรปากกา (ลูกศรชี้ไปแม่แบบ)	การสืบทอดคุณสมบัติและพฤติกรรมจากคลาสแม่ (Super Class)
Aggregation	เส้นมีข้าวหลามตัดกลวง	ส่วนประกอบแบบหลวม อ็อบเจกต์ที่เป็นส่วนประกอบมีชีวิตอิสระ
Composition	เส้นมีข้าวหลามตัดทึบ	ส่วนประกอบแบบแข็งแรง อ็อบเจกต์ส่วนประกอบมีชีวิตร่วมกับเจ้าของ

ตัวอย่าง Inheritance

```
+------------------+          +--------------------+
|      Vehicle     |<|------- |        Car         |
+------------------+          +--------------------+
| - speed: int     |          | - numDoors: int    |
+------------------+          +--------------------+
| + move()         |          | + openDoor()       |
+------------------+          +--------------------+
```

- Car สืบทอด (extends) จาก Vehicle

สรุป

ประเด็น	Class Diagram
แสดงอะไร	โครงสร้างคลาส, Attribute, Method, ความสัมพันธ์
ประโยชน์หลัก	ทำให้เห็นภาพรวมโปรแกรมและความเชื่อมโยง
เหมาะกับ	ออกแบบระบบที่ใช้ OOP, วิเคราะห์ความซับซ้อน
ใช้เมื่อ	เริ่มวางโครงสร้างโปรแกรม, สื่อสารกับทีม

Class Diagram เชิงลึกที่สุดใน UML

1. ความหมายและบทบาทของ Class Diagram

- **Class Diagram** เป็นหนึ่งในประเภทของ UML Diagram ที่สำคัญที่สุด
- ใช้แสดง **โครงสร้างแบบคงที่ (Static Structure)** ของระบบเชิงวัตถุ
- บอกว่าในระบบมี **Class อะไรบ้าง** แต่ละคลาสมีอะไรบ้าง และมีความสัมพันธ์แบบไหนกัน
- เป็นแผนภาพที่นักออกแบบและนักพัฒนานำมาใช้เพื่อ **วางแผน พูดคุย และสื่อสารโครงสร้างระบบ** ก่อนเริ่มเขียนโค้ด

2. องค์ประกอบหลักของ Class Diagram

2.1 Class (คลาส)

- คลาสเป็น **แบบแผน (Blueprint)** สำหรับสร้างอ็อบเจกต์
- แสดงเป็นสี่เหลี่ยม 3 ช่อง (เรียกว่า **“แพ็กเกจ”** หรือ **“คอมปาร์ตเมนต์”**)

โครงสร้างคลาส:

```
+--------------------------+
|        Class Name        |  ← ชื่อคลาส (สำคัญที่สุด)
+--------------------------+
| - attribute1: Type       |  ← คุณสมบัติ (Attributes) หรือ ข้อมูลของคลาส
| - attribute2: Type       |
+--------------------------+
| + method1(params): Return |  ← ฟังก์ชันหรือพฤติกรรม (Operations/Methods)
| + method2(params): Return |
+--------------------------+
```

- **ชื่อคลาส** เขียนให้อยู่บนสุด

- **Attributes** คือ ตัวแปรข้อมูลที่คลาสนั้นเก็บไว้
- **Methods** คือ ฟังก์ชันที่คลาสนั้นสามารถทำได้

2.2 Visibility (การมองเห็น)

กำหนดว่า attribute หรือ method นั้นเข้าถึงได้จากที่ใดบ้าง โดยใช้สัญลักษณ์นำหน้า

สัญลักษณ์	ความหมาย
+	Public (เปิดเผยทุกที่)
-	Private (เฉพาะในคลาส)
#	Protected (เฉพาะคลาสและลูก)
~	Package (เฉพาะภายในแพ็กเกจ)

2.3 Attributes (คุณสมบัติ)

- แสดงชื่อและชนิดข้อมูล เช่น - age: int
- อาจกำหนดค่าเริ่มต้น เช่น - name: String = "John"

2.4 Methods (พฤติกรรม)

- แสดงชื่อ, พารามิเตอร์, และชนิดผลลัพธ์ เช่น

  + getName(): String

  + setAge(age: int): void

---

3. ความสัมพันธ์ระหว่างคลาส (Relationships)

3.1 Association (ความสัมพันธ์ทั่วไป)

- เส้นตรงเชื่อมระหว่างคลาส
- แสดงว่าคลาสสองตัวรู้จักกัน หรือมีการติดต่อกัน
- อาจมีทิศทาง (ลูกศร) หรือไม่มีทิศทาง
- มักกำหนด **Multiplicity** (จำนวนความสัมพันธ์)

ตัวอย่าง:

- 1 ฝ่ายหนึ่งสัมพันธ์กับ * อีกฝ่ายหนึ่ง (หนึ่งต่อหลาย)

3.2 Multiplicity (ความหลายหลาย)

- กำหนดจำนวนของอ็อบเจกต์ที่มีส่วนร่วมในความสัมพันธ์
- ใช้ notation เช่น
  - 1 หมายถึงหนึ่งตัวเท่านั้น

- 0..1 หมายถึงศูนย์หรือหนึ่งตัว
- * หมายถึงหลายตัว (0 หรือมากกว่า)
- 1..* หมายถึงหนึ่งหรือมากกว่า

3.3 Aggregation (การรวมแบบหลวม)

- เส้นตรงมีรูปข้าวหลามตัด (empty diamond) ที่ปลายด้านเจ้าของ
- แสดงความสัมพันธ์ “has-a” แบบส่วนประกอบที่มีชีวิตแยกได้
- ตัวอย่าง: ห้องสมุด **มี** หนังสือ แต่หนังสือยังสามารถอยู่ได้เอง

3.4 Composition (การรวมแบบแน่น)

- เส้นตรงมีรูปข้าวหลามตัดทึบ (filled diamond) ที่ปลายเจ้าของ
- แสดงความสัมพันธ์แบบส่วนประกอบที่ชีวิตขึ้นกับเจ้าของ
- ตัวอย่าง: ห้อง **มี** ห้องน้ำ ห้องน้ำไม่สามารถอยู่ได้ถ้าไม่มีห้อง

3.5 Inheritance / Generalization (การสืบทอด)

- ลูกศรที่มีหัวแบบกลวง (open arrowhead) ชี้จากคลาสลูกไปยังคลาสแม่
- บอกว่าคลาสลูก **สืบทอดคุณสมบัติและพฤติกรรม** จากคลาสแม่
- สนับสนุน **Polymorphism** และ **Reuse**

3.6 Dependency (การพึ่งพิง)

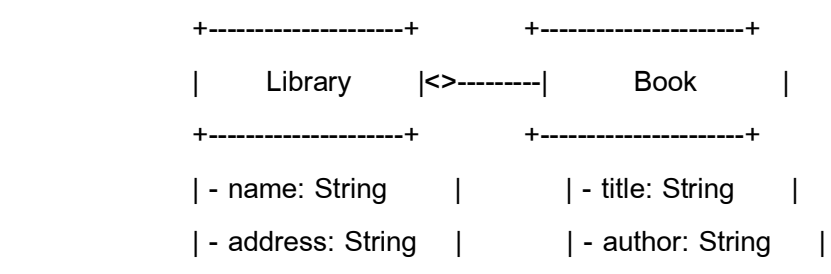
- เส้นประที่มีลูกศรชี้
- บอกว่าคลาสหนึ่งใช้หรือขึ้นกับอีกคลาสหนึ่งชั่วคราว
- เช่น method ในคลาส A ใช้ parameter หรือคืนค่าจากคลาส B

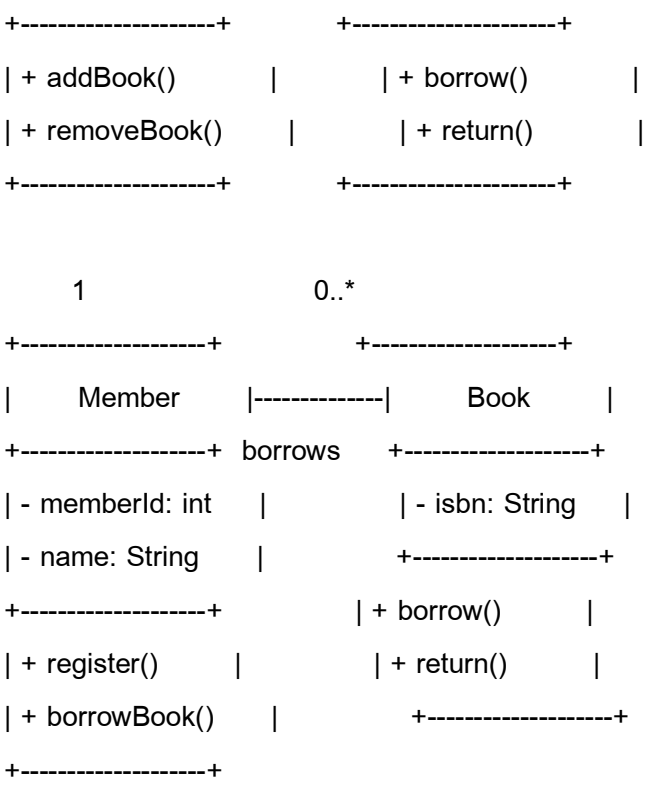
3.7 Realization (การทำสัญญา)

- ใช้ในกรณี Interface กับคลาสที่ implements
- เส้นประที่มีลูกศรกลวงชี้ไปยัง Interface

---

4. ตัวอย่าง Class Diagram แบบสมบูรณ์ (Library System)





- Library **Aggregation** หนังสือหลายเล่ม
- Member กับ Book มี **Association** แบบยืมหนังสือกัน

---

5. การใช้ Class Diagram จริงในโปรเจกต์

- ใช้ในขั้นตอน **ออกแบบระบบ (Design Phase)**
- ช่วยให้ทีมเห็นภาพรวมของข้อมูลและพฤติกรรม
- ใช้เป็น **เอกสารประกอบการพัฒนาและบำรุงรักษา**
- ใช้ช่วยวางแผนการเขียนโค้ดที่สอดคล้องตามแบบแผน

---

6. เครื่องมือสร้าง Class Diagram ยอดนิยม

- **StarUML**
- **Visual Paradigm**
- **Enterprise Architect**
- **PlantUML** (เขียนเป็นโค้ดแล้ว generate ภาพ)
- **IBM Rational Rose**
- **Microsoft Visio**

---

7. เคล็ดลับออกแบบ Class Diagram ให้ดี

- รักษา **Single Responsibility Principle (SRP)**: คลาสหนึ่งควรมีหน้าที่เดียว

- อย่าทำคลาสใหญ่มาก (God Class)
- ใช้ **Inheritance** เฉพาะเมื่อมีความสัมพันธ์ “is-a” ชัดเจน
- ใช้ **Aggregation** และ **Composition** เพื่อแสดงความเป็นเจ้าของและส่วนประกอบ
- ใส่ **Multiplicity** เพื่อชี้ชัดจำนวนอ็อบเจกต์
- สื่อสารให้ทีมเข้าใจง่ายด้วย **Naming ที่ชัดเจน**

สรุป

ประเด็นสำคัญ	Class Diagram
หน้าที่หลัก	แสดงโครงสร้างคลาสและความสัมพันธ์
องค์ประกอบหลัก	Class, Attribute, Method, Relationships
ความสัมพันธ์	Association, Aggregation, Composition, Inheritance, Dependency
ประโยชน์	วางแผนระบบ, สื่อสารทีม, เอกสารประกอบการพัฒนา
ใช้เมื่อ	ออกแบบระบบ, วางโครงสร้าง OOP, วิเคราะห์ความซับซ้อน

สรุป

บทนี้เป็นจุดเริ่มต้นสำคัญของการเรียนรู้แนวคิดพื้นฐานของ OOP (Object-Oriented Programming) โดยอธิบายถึงความแตกต่างระหว่างการเขียนโปรแกรมแบบ Procedural และ Object-Oriented เพื่อให้ผู้เรียนเข้าใจถึงข้อดีของ OOP ในการรวมข้อมูลและหน้าที่เป็นหน่วยเดียว จัดการและบำรุงรักษาโค้ดได้เป็นระบบและรองรับโปรแกรมขนาดใหญ่ได้อย่างมีประสิทธิภาพ เนื้อหาครอบคลุมทั้งแนวคิดของ Object และ Class ที่เป็นหัวใจของ OOP ตลอดจนความสำคัญของ OOP ในการพัฒนาโปรแกรมขนาดใหญ่ และการใช้ UML (Unified Modeling Language) เป็นเครื่องมือช่วยออกแบบและสื่อสารโครงสร้างของโปรแกรม เพื่อปูพื้นฐานไปสู่การเป็นโปรแกรมเมอร์ C++ OOP ระดับมืออาชีพต่อไป

บทที่ 3
การสร้างคลาสและวัตถุ
(Class and Object)

เนื้อหา

- พื้นฐานการสร้างคลาสและวัตถุใน C++
- การสร้างคลาสและวัตถุใน C++ โดยละเอียด พร้อม ตัวอย่าง UML Class Diagram
- การประกาศและใช้งาน class และ object ใน C++
- การประกาศและใช้งาน class และ object ใน C++ แบบเจาะลึก
- ตัวอย่างโปรแกรม: ระบบจัดการบุคคล (Person Management)
- Constructor และ Destructor ใน C++
- Constructor และ Destructor ใน C++ แบบเชิงลึก
- Access Specifier ใน C++: public, private, protected
- Access Specifier ใน C++ — รายละเอียดเชิงลึก พร้อม UML
- Getter / Setter (Encapsulation)
- Getter / Setter (Encapsulation) — เชิงลึก

บทนำ

การสร้างคลาสและอ็อบเจกต์ (Class and Object) เป็นหัวใจสำคัญของการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming) ใน C++ เพราะเป็นจุดเริ่มต้นของการออกแบบและพัฒนาโปรแกรมที่สอดคล้องกับแนวคิด OOP แท้จริง โดยผู้เรียนจะได้เรียนรู้ว่า Class เป็นเสมือนแม่แบบหรือพิมพ์เขียวของสิ่งที่ต้องการสร้าง ในขณะที่ Object เป็นชิ้นงานจริงที่ถูกผลิตขึ้นมาด้วยพิมพ์เขียวนั้น การเข้าใจกระบวนการประกาศและใช้งาน Class และ Object จึงเป็นก้าวแรกของผู้พัฒนาโปรแกรม C++ OOP

สิ่งสำคัญอีกข้อคือ **Constructor และ Destructor** ที่เป็นฟังก์ชันพิเศษของคลาส มีหน้าที่ช่วยผู้พัฒนาในการควบคุมอายุของอ็อบเจกต์ ตั้งแต่ตอนเริ่มต้นจนถึงตอนสิ้นสุด Constructor ใช้เพื่อเตรียมค่าหรือทรัพยากรเริ่มต้นของอ็อบเจกต์ ในขณะที่ Destructor ใช้เพื่อเก็บกวาดและคืนทรัพยากรเมื่ออ็อบเจกต์หมดอายุ สิ่งเหล่านี้ช่วยให้โปรแกรมบริหารจัดการหน่วยความจำและทรัพยากรได้เป็นระบบและลดข้อผิดพลาดที่อาจเกิดขึ้น

ผู้เรียนจะได้ทำความรู้จักและเข้าใจกับ **Access Specifier** ได้แก่ public, private และ protected ที่ช่วยควบคุมการเข้าถึงและแก้ไขข้อมูลภายในคลาส การเลือกใช้ Access Specifier ที่

ถูกต้องเป็นหัวใจของ Encapsulation เพื่อป้องกันข้อมูลและควบคุมสิทธิ์การเข้าถึงจากภายนอกคลาส ผู้เรียนจะเข้าใจว่า Access Specifier แต่ละแบบมีหน้าที่และลักษณะการใช้งานแตกต่างกัน และเลือกใช้ได้อย่างเหมาะสม

หัวข้อถัดไปคือ **Getter และ Setter** เป็นแนวคิดและวิธีการที่ช่วยให้ผู้เรียนออกแบบและควบคุมการเข้าถึงข้อมูลภายในคลาสได้เป็นระบบ โดยใช้เมธอด Getter เพื่ออ่านค่าของตัวแปรภายในคลาส และเมธอด Setter เพื่ออัพเดตค่าของตัวแปรโดยมีตรรกะควบคุมสิทธิ์และตรวจสอบค่าก่อนแก้ไขจริง ๆ หลักการเหล่านี้ช่วยให้โปรแกรมมีความปลอดภัย มีโครงสร้างที่ชัดเจน และแก้ไขได้ง่ายเมื่อต้องขยายหรือปรับปรุง

สุดท้าย บทนี้จะเป็นรากฐานของการเขียนโปรแกรมเชิงอ็อบเจกต์ด้วย C++ ในระดับสูงขึ้น ไม่ว่าจะเป็น Inheritance, Polymorphism หรือต่อยอดไปถึง Design Patterns และ Architecture ของโปรแกรมจริง ผู้เรียนจะเข้าใจทั้งทฤษฎีและแนวคิดปฏิบัติของการออกแบบคลาสและอ็อบเจกต์ และสามารถนำไปใช้แก้โจทย์ปัญหาหรือพัฒนาโปรแกรม C++ OOP ได้อย่างเป็นระบบ มีประสิทธิภาพ และตรงตามมาตรฐานของการเขียนโปรแกรมมืออาชีพ.

พื้นฐานการสร้างคลาสและวัตถุใน C++

3.1 การประกาศและใช้งาน Class และ Object

Class คืออะไร?

- เป็นแบบแผน (Blueprint) ที่กำหนดว่าอ็อบเจกต์ (วัตถุ) จะมีข้อมูลอะไร (Attributes) และทำงานอะไรได้บ้าง (Methods)
- คลาสไม่ใช่วัตถุจริง แต่คือแม่แบบให้สร้างวัตถุ

Object คืออะไร?

- เป็น instance หรือวัตถุจริงที่สร้างจากคลาส
- แต่ละอ็อบเจกต์มีสถานะข้อมูลเป็นของตัวเอง

ตัวอย่างการประกาศ Class และสร้าง Object

```
#include <iostream>
using namespace std;

class Person {
public:
    string name;    // attribute
    int age;        // attribute

    void sayHello() {  // method
```

```
        cout << "Hello, my name is " << name << " and I'm " << age << " years old.\n";
    }
};

int main() {
    Person p1;          // สร้าง Object p1 จาก Class Person
    p1.name = "Alice";  // กำหนดค่า Attribute
    p1.age = 30;
    p1.sayHello();      // เรียกใช้ Method

    return 0;
}
```

ผลลัพธ์:

Hello, my name is Alice and I'm 30 years old.

3.2 Constructor และ Destructor

Constructor (ตัวสร้าง)

- เป็นฟังก์ชันพิเศษชื่อเหมือนกับชื่อคลาส
- ถูกเรียกโดยอัตโนมัติเมื่อสร้างอ็อบเจกต์
- ใช้สำหรับ **กำหนดค่าเริ่มต้น** ให้กับอ็อบเจกต์
- สามารถมีหลายแบบ (Overloading) ได้

Destructor (ตัวทำลาย)

- เป็นฟังก์ชันพิเศษชื่อเหมือนคลาสแต่มี ~ นำหน้า
- ถูกเรียกโดยอัตโนมัติเมื่ออ็อบเจกต์ถูกทำลาย
- ใช้สำหรับ **เคลียร์ทรัพยากร** เช่น memory หรือไฟล์

ตัวอย่าง Constructor และ Destructor

```
#include <iostream>
using namespace std;

class Person {
public:
    string name;
    int age;
```

```
    // Constructor
    Person(string n, int a) {
        name = n;
        age = a;
        cout << "Constructor called for " << name << endl;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }

    void sayHello() {
        cout << "Hello, my name is " << name << ", age " << age << endl;
    }
};

int main() {
    Person p1("Bob", 25);
    p1.sayHello();

    return 0;
}
```

ผลลัพธ์:

```
Constructor called for Bob
Hello, my name is Bob, age 25
Destructor called for Bob
```

3.3 Access Specifier: public, private, protected

Access Specifier	ความหมาย	เข้าถึงได้จากที่ไหน
public	เข้าถึงได้จากทุกที่	ทั้งภายในและภายนอกคลาส
private	เข้าถึงได้เฉพาะภายในคลาสเท่านั้น	ไม่สามารถเข้าถึงจากภายนอกคลาส
protected	เข้าถึงได้ภายในคลาสและคลาสลูก	เหมาะสำหรับ inheritance

ตัวอย่างการใช้งาน Access Specifier

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;  // private attribute

public:
    int age;       // public attribute

    void setName(string n) {    // public setter method
        name = n;
    }

    string getName() {          // public getter method
        return name;
    }
};

int main() {
    Person p;
    // p.name = "Alice";       // ERROR! name เป็น private ไม่สามารถเข้าถึงได้ตรง ๆ

    p.setName("Alice");        // ต้องใช้ setter แทน
    p.age = 30;                // age เป็น public สามารถเข้าถึงได้ตรง ๆ

    cout << "Name: " << p.getName() << ", Age: " << p.age << endl;

    return 0;
}
```

3.4 Getter / Setter (Encapsulation)

- **Encapsulation** คือการปกป้องข้อมูลภายในคลาสไม่ให้ถูกเข้าถึงโดยตรง

- ใช้ **private** กับข้อมูล และเปิดให้เข้าถึงข้อมูลผ่าน **getter** และ **setter**
- ช่วยควบคุมการแก้ไขและตรวจสอบข้อมูลก่อนกำหนดค่า

ตัวอย่าง Getter / Setter

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    void setName(string n) {
        if (n.length() > 0) {
            name = n;
        }
    }

    string getName() {
        return name;
    }

    void setAge(int a) {
        if (a >= 0) {
            age = a;
        }
    }

    int getAge() {
        return age;
    }
};

int main() {
```

```
    Person p;
    p.setName("Charlie");
    p.setAge(28);

    cout << "Name: " << p.getName() << ", Age: " << p.getAge() << endl;

    return 0;
}
```

สรุป

หัวข้อ	คำอธิบายสั้น ๆ
Class	แบบแผนสำหรับสร้าง Object
Object	อ็อบเจกต์ที่สร้างจาก Class
Constructor	ฟังก์ชันพิเศษสร้าง Object และกำหนดค่าเริ่มต้น
Destructor	ฟังก์ชันพิเศษทำลาย Object และเคลียร์ทรัพยากร
Access Specifier	กำหนดการเข้าถึงสมาชิก class (public/private/protected)
Getter/Setter	วิธีเข้าถึงและแก้ไขข้อมูลแบบปลอดภัย (Encapsulation)

การสร้างคลาสและวัตถุใน C++ โดยละเอียด พร้อม ตัวอย่าง UML Class Diagram

---

3. การสร้างคลาสและวัตถุใน C++ พร้อม UML

3.1 การประกาศและใช้งาน Class และ Object

- **Class** คือ แบบแผน (Blueprint) ที่บอกว่าอ็อบเจกต์จะมีข้อมูล (Attributes) และพฤติกรรม (Methods) อะไรบ้าง
- **Object** คือ อ็อบเจกต์จริงที่สร้างจากคลาส (Instance)

ตัวอย่างโค้ด C++

```
class Person {
public:
    string name;
    int age;
```

```
    void sayHello() {
        cout << "Hello, I'm " << name << ", " << age << " years old." << endl;
    }
};

int main() {
    Person p1;  // สร้าง Object p1
    p1.name = "Alice";
    p1.age = 30;
    p1.sayHello();

    return 0;
}
```

UML Class Diagram (เบื้องต้นของ Person)

```
+---------------------+
|       Person        |
+---------------------+
| - name: string      |
| - age: int          |
+---------------------+
| + sayHello(): void  |
+---------------------+
```

- \- = private (ถ้าไม่ระบุใน C++ จะเป็น private โดยค่าเริ่มต้นถ้าใช้ struct จะเป็น public)
- \+ = public

3.2 Constructor และ Destructor

- **Constructor:** ฟังก์ชันพิเศษชื่อเดียวกับคลาส เรียกตอนสร้าง Object เพื่อกำหนดค่าเริ่มต้น
- **Destructor:** ฟังก์ชันพิเศษชื่อเดียวกับคลาสแต่มี ~ หน้า เรียกตอน Object ถูกทำลาย (ปล่อยทรัพยากร)

ตัวอย่างโค้ด

```
class Person {
private:
```

```
    string name;
    int age;

public:
    // Constructor
    Person(string n, int a) {
        name = n;
        age = a;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }

    void sayHello() {
        cout << "Hello, I'm " << name << ", " << age << " years old." << endl;
    }
};
```

UML Class Diagram (Person กับ Constructor/Destructor)

```
+----------------------------+
|           Person           |
+----------------------------+
| - name: string             |
| - age: int                 |
+----------------------------+
| + Person(n: string, a: int) |  <<constructor>>
| + ~Person()                |  <<destructor>>
| + sayHello(): void         |
+----------------------------+
```

3.3 Access Specifier: public, private, protected

Access Specifier	ความหมาย	ตัวอย่างการใช้งาน
public	เข้าถึงได้ทุกที่	ตัวแปรและเมธอดที่ต้องการให้เรียกใช้ได้จากภายนอกคลาส
private	เข้าถึงได้เฉพาะภายในคลาส	ตัวแปรและเมธอดที่ต้องการซ่อน ไม่ให้แก้ไขโดยตรงจากภายนอก
protected	เข้าถึงได้ภายในคลาสและคลาสลูก	เหมาะสำหรับการสืบทอด (inheritance)

ตัวอย่างโค้ด

```
class Person {
private:
    string name;    // ซ่อนข้อมูล
public:
    int age;

    void setName(string n) {
        name = n;
    }

    string getName() {
        return name;
    }
};
```

UML Class Diagram ที่แสดง Access Specifier

```
+-----------------------+
|        Person         |
+-----------------------+
| - name: string        |
| + age: int            |
+-----------------------+
| + setName(n: string)  |
| + getName(): string   |
```

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

3.4 Getter / Setter (Encapsulation)

- ใช้ **ซ่อนข้อมูล** (private) และเปิดช่องทางเข้าถึงผ่าน **getter/setter**
- ป้องกันการแก้ไขข้อมูลโดยตรง เพื่อให้ควบคุมค่าที่ถูกต้อง เช่น เช็คค่าก่อนกำหนด

ตัวอย่างโค้ด

```
class Person {
private:
    string name;
    int age;

public:
    void setName(string n) {
        if (n.length() > 0) name = n;
    }

    string getName() {
        return name;
    }

    void setAge(int a) {
        if (a >= 0) age = a;
    }

    int getAge() {
        return age;
    }
};
```

UML Class Diagram พร้อม Getter/Setter

```
+--------------------------+
|          Person          |
+--------------------------+
| - name: string           |
```

| - age: int |
+----------+
| + setName(n: string) |
| + getName(): string |
| + setAge(a: int) |
| + getAge(): int |
+----------+

สรุปภาพรวมใน UML Class Diagram

องค์ประกอบ	รายละเอียด	ตัวอย่าง
Class Name	ชื่อคลาส	Person
Attributes	ตัวแปรข้อมูลพร้อมชนิดและระดับการเข้าถึง	- name: string (private) + age: int (public)
Methods	ฟังก์ชันที่เปิดเผยหรือซ่อน	+ setName(), + getName()
Constructor/Destructor	ฟังก์ชันพิเศษสำหรับสร้างและทำลาย Object	+ Person(), + ~Person()
Access Specifier	public, private, protected	ใช้ +, -, # แทนใน UML

การประกาศและใช้งาน class และ object ใน C++

การประกาศและใช้งาน Class และ Object ใน C++

1. การประกาศ Class

- Class คือ โครงสร้างที่ใช้เป็นแบบแผนในการสร้างอ็อบเจกต์
- ประกอบด้วย **Attributes** (ข้อมูล) และ **Methods** (ฟังก์ชันหรือพฤติกรรม)
- Syntax พื้นฐาน:

```
class ClassName {
    // access specifiers and members
};
```

- ในคลาส เราจะประกาศตัวแปร (attributes) และฟังก์ชัน (methods) ภายใน
- โดยค่าปริยาย ถ้าไม่ระบุ access specifier จะเป็น **private**

ตัวอย่างการประกาศ Class

```
class Person {
public:                 // กำหนดให้สมาชิกนี้เข้าถึงได้จากภายนอก
    string name;        // attribute
    int age;

    void sayHello() { // method
        cout << "Hello, my name is " << name << endl;
    }
};
```

---

2. การสร้าง Object (Instantiation)

- Object คือ instance ที่สร้างจาก class
- สร้าง object โดยใช้ชื่อคลาสเป็นชนิดข้อมูล เช่น

```
Person p1;    // สร้างอ็อบเจกต์ชื่อ p1 จาก class Person
```

- แต่ละ object มีค่าของ attribute เป็นของตัวเองแยกจาก object อื่น ๆ

---

3. การใช้งาน Object

- สามารถเข้าถึง attribute และ method ผ่าน **dot operator (.)**
- เช่น

```
p1.name = "Alice";    // กำหนดค่า attribute name
p1.age = 25;          // กำหนดค่า attribute age
p1.sayHello();        // เรียก method sayHello()
```

---

4. ตัวอย่างโปรแกรมเต็ม

```
#include <iostream>
using namespace std;

class Person {
public:
    string name;
    int age;

    void sayHello() {
        cout << "Hello, my name is " << name << " and I am " << age << " years old." << endl;
```

```
    }
};

int main() {
    Person p1;          // สร้าง object p1
    p1.name = "Alice";  // กำหนดค่า attribute
    p1.age = 25;
    p1.sayHello();      // เรียก method

    Person p2;          // สร้าง object p2
    p2.name = "Bob";
    p2.age = 30;
    p2.sayHello();

    return 0;
}
```

ผลลัพธ์:

```
Hello, my name is Alice and I am 25 years old.
Hello, my name is Bob and I am 30 years old.
```

สรุป

ขั้นตอน	คำอธิบาย
ประกาศ Class	กำหนดโครงสร้างข้อมูลและฟังก์ชันในคลาส
สร้าง Object	ใช้ชื่อคลาสเป็นชนิดข้อมูล ประกาศตัวแปร object
กำหนดค่า Attribute	ใช้ dot operator เข้าถึงและกำหนดค่า attribute
เรียกใช้งาน Method	ใช้ dot operator เรียกฟังก์ชันภายใน object

---

การประกาศและใช้งาน class และ object ใน C++ แบบเจาะลึก

การประกาศและใช้งาน Class และ Object ใน C++ (เชิงลึก) พร้อม UML

---

1. แนวคิดเบื้องต้น: Class และ Object

- **Class** คือ แบบแผน (Blueprint) สำหรับสร้างอ็อบเจกต์ (Object)

- กำหนดข้อมูล (Attributes) และพฤติกรรม (Methods) ที่อ็อบเจกต์จะมี
- **Object** คือ instance ของ class ที่มีสถานะข้อมูลและพฤติกรรมของตัวเอง

เปรียบเสมือน **แม่พิมพ์ (Class)** กับ **สินค้าที่ถูกผลิต (Object)**

---

2. การประกาศ Class ใน C++

Syntax

```
class ClassName {
    access_specifier:
        // Attributes (ตัวแปร)
        // Methods (ฟังก์ชัน)
};
```

- **access_specifier**: ระบุระดับการเข้าถึงของสมาชิก (เช่น public, private)
- ถ้าไม่ระบุ default จะเป็น **private**
- สมาชิก (attributes/methods) ที่เป็น public จะสามารถถูกเรียกจากภายนอกได้

ตัวอย่างประกาศ Class Person

```
class Person {
public:                  // เปิดให้เข้าถึงได้ทุกที่
    string name;         // Attribute ชื่อ name
    int age;             // Attribute ชื่อ age

    void sayHello() {    // Method พูดทักทาย
        cout << "Hello, my name is " << name << ", age " << age << endl;
    }
};
```

---

3. การสร้าง Object จาก Class

- ใช้ชื่อ class เป็นชนิดข้อมูล แล้วประกาศตัวแปร object
- ทุก object มีข้อมูล attribute ของตัวเองแยกจาก object อื่น ๆ

ตัวอย่าง

```
Person p1;   // สร้าง object p1
Person p2;   // สร้าง object p2
```

---

4. การใช้งาน Object

- ใช้ **dot operator (.)** เพื่อเข้าถึง attributes และ methods
- แต่ละ object สามารถเก็บข้อมูลแตกต่างกันได้

ตัวอย่าง

```
p1.name = "Alice";
p1.age = 30;
p1.sayHello();

p2.name = "Bob";
p2.age = 25;
p2.sayHello();
```

---

5. ตัวอย่างโปรแกรมเต็ม

```
#include <iostream>
using namespace std;

class Person {
public:
    string name;
    int age;

    void sayHello() {
        cout << "Hello, my name is " << name << ", age " << age << endl;
    }
};

int main() {
    Person p1;
    p1.name = "Alice";
    p1.age = 30;
    p1.sayHello();

    Person p2;
    p2.name = "Bob";
    p2.age = 25;
```

```
    p2.sayHello();

    return 0;
}
```

---

6. UML Class Diagram ของ Person

```
+------------------------+
|         Person         |
+------------------------+
| + name: string         |
| + age: int             |
+------------------------+
| + sayHello(): void     |
+------------------------+
```

- + หมายถึง **public**
- แสดง attributes และ methods ที่ class มี

---

7. ความเข้าใจเชิงลึกเพิ่มเติม

ประเด็น	รายละเอียด
Class เป็นแบบแผน	Class เพียงแค่เป็นแม่แบบ ไม่มีหน่วยความจำถูกจัดสรรจนกว่าจะสร้าง Object
Object เป็น Instance	Object คือหน่วยความจำที่ถูกจัดสรรตามแบบของ class มีสถานะข้อมูล (attributes) ของตัวเอง
Dot operator (.)	ใช้เข้าถึงสมาชิก (attributes/methods) ของ object
หลาย Object	เราสามารถสร้าง object หลายตัวจาก class เดียว แต่ละตัวเก็บสถานะข้อมูลต่างกัน
Encapsulation	ควรซ่อนข้อมูล (private) และเข้าถึงผ่าน public methods (getter/setter) เพื่อควบคุมความปลอดภัย

---

8. การขยายความ: Access Specifier

```
class Person {
private:
```

```
    string name;      // ซ่อนจากภายนอก
public:
    int age;

    void setName(string n) { name = n; }
    string getName() { return name; }
};
```

- name เป็น private ไม่สามารถเข้าถึงโดยตรงจาก object ได้
- ต้องใช้ method public ในการอ่าน/เขียนข้อมูลแทน

---

9. UML Class Diagram พร้อม Access Specifier

```
+---------------------------+
|          Person           |
+---------------------------+
| - name: string            |
| + age: int                |
+---------------------------+
| + setName(n: string)      |
| + getName(): string       |
| + sayHello(): void        |
+---------------------------+
```

สรุป

ขั้นตอน	สาระสำคัญ
ประกาศ Class	กำหนดแบบแผน มี Attributes และ Methods
สร้าง Object	ประกาศตัวแปรชนิด class เพื่อสร้าง instance ใหม่
กำหนดค่า Attribute	ใช้ dot operator กำหนดค่า attribute ของ object
เรียก Method	ใช้ dot operator เรียก method ที่ class กำหนด
ใช้ Access Specifier	ควบคุมการเข้าถึงข้อมูลภายใน class (public/private)

ตัวอย่างโปรแกรม: ระบบจัดการบุคคล (Person Management)

---

1. โจทย์

สร้างโปรแกรมจัดการข้อมูลบุคคลโดยใช้คลาส Person โดยข้อมูลแต่ละคนมี

- ชื่อ (name)
- อายุ (age)

โปรแกรมต้องสามารถ

- กำหนดชื่อและอายุให้กับบุคคล
- แสดงข้อมูลบุคคลออกทางหน้าจอ

---

2. UML Class Diagram

```
+---------------------------+
|          Person           |
+---------------------------+
| - name: string            |    // ข้อมูลส่วนตัวเก็บแบบ private
| - age: int                |
+---------------------------+
| + setName(n: string)      |    // กำหนดชื่อ
| + getName(): string       |    // อ่านชื่อ
| + setAge(a: int)          |    // กำหนดอายุ
| + getAge(): int           |    // อ่านอายุ
| + displayInfo(): void     |    // แสดงข้อมูล
+---------------------------+
```

---

3. โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;   // เก็บชื่อ (private ซ่อนข้อมูล)
    int age;       // เก็บอายุ

public:
    // Setter สำหรับชื่อ
    void setName(string n) {
```

```
        name = n;
    }

    // Getter สำหรับชื่อ
    string getName() {
        return name;
    }

    // Setter สำหรับอายุ
    void setAge(int a) {
        if(a >= 0) {    // ตรวจสอบให้ค่ามีความถูกต้อง
            age = a;
        }
    }

    // Getter สำหรับอายุ
    int getAge() {
        return age;
    }

    // แสดงข้อมูลบุคคล
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p1;              // สร้าง Object p1 จากคลาส Person
    p1.setName("Alice");      // กำหนดชื่อผ่าน setter
    p1.setAge(28);            // กำหนดอายุผ่าน setter

    Person p2;              // สร้าง Object p2
    p2.setName("Bob");
    p2.setAge(35);
```

```
    // แสดงข้อมูลบุคคลทั้งสองคน
    p1.displayInfo();
    p2.displayInfo();

    return 0;
}
```

---

4. ผลการรันโปรแกรม

Name: Alice, Age: 28

Name: Bob, Age: 35

---

5. คำอธิบายแทรก

- คลาส Person กำหนดข้อมูลส่วนตัว name และ age เป็น **private** เพื่อซ่อนข้อมูลจากภายนอก
- สร้างเมธอด **setter** และ **getter** เพื่อควบคุมการเข้าถึงและป้องกันข้อมูลเสียหาย (Encapsulation)
- เมธอด displayInfo() ใช้แสดงข้อมูลที่เก็บอยู่ในอ็อบเจกต์
- ใน main() สร้างอ็อบเจกต์ 2 ตัว (p1 กับ p2) แต่ละตัวเก็บข้อมูลคนละชุด
- เรียกใช้เมธอดเพื่อกำหนดและแสดงข้อมูลตามลำดับ

ตัวอย่างโปรแกรม 5 ชุด แบบครบทั้ง โจทย์ + UML + โค้ด C++ พร้อมคำอธิบายแทรก + ผลการรัน

ตัวอย่างโปรแกรม 1 - ระบบจัดการบุคคล (Person Management)

โจทย์

สร้างคลาส Person เก็บชื่อและอายุ พร้อมแสดงข้อมูลออกทางหน้าจอ

UML Class Diagram

```
+---------------------------+
|          Person           |
+---------------------------+
| - name: string            |
| - age: int                |
+---------------------------+
```

```
| + setName(n: string)       |
| + getName(): string        |
| + setAge(a: int)           |
| + getAge(): int            |
| + displayInfo(): void      |
+----------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    void setName(string n) { name = n; }
    string getName() { return name; }
    void setAge(int a) { if(a >= 0) age = a; }
    int getAge() { return age; }
    void displayInfo() { cout << "Name: " << name << ", Age: " << age << endl; }
};

int main() {
    Person p1;
    p1.setName("Alice");
    p1.setAge(28);

    Person p2;
    p2.setName("Bob");
    p2.setAge(35);

    p1.displayInfo();
```

```
    p2.displayInfo();

    return 0;
}
```

ผลการรัน

Name: Alice, Age: 28

Name: Bob, Age: 35

ตัวอย่างโปรแกรม 2 - ระบบบัญชีธนาคาร (Bank Account)

โจทย์

สร้างคลาส BankAccount มีเลขที่บัญชีและยอดเงิน พร้อมฝากและถอนเงินได้

UML Class Diagram

```
+-----------------------------+
|         BankAccount         |
+-----------------------------+
| - accountNumber: string     |
| - balance: double           |
+-----------------------------+
| + deposit(amount: double)   |
| + withdraw(amount: double)  |
| + getBalance(): double      |
| + setAccountNumber(acc: string) |
| + getAccountNumber(): string |
+-----------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class BankAccount {
private:
```

```
    string accountNumber;
    double balance;

public:
    BankAccount() { balance = 0; }  // Constructor กำหนดยอดเงินเริ่มต้นเป็น 0

    void setAccountNumber(string acc) { accountNumber = acc; }
    string getAccountNumber() { return accountNumber; }

    void deposit(double amount) {
        if(amount > 0) balance += amount;
    }

    void withdraw(double amount) {
        if(amount > 0 && amount <= balance) balance -= amount;
        else cout << "Insufficient balance!" << endl;
    }

    double getBalance() { return balance; }
};

int main() {
    BankAccount acc1;
    acc1.setAccountNumber("123456789");
    acc1.deposit(1000);
    acc1.withdraw(300);
    cout << "Account " << acc1.getAccountNumber() << " balance: " << acc1.getBalance() <<
endl;

    acc1.withdraw(800);  // เกินยอดเงินจะมีข้อความเตือน

    return 0;
}
```

ผลการรัน

Account 123456789 balance: 700

Insufficient balance!

ตัวอย่างโปรแกรม 3 - ระบบรถยนต์ (Car)

โจทย์

สร้างคลาส Car มียี่ห้อ, รุ่น, และปีผลิต พร้อมฟังก์ชันแสดงข้อมูลรถ

UML Class Diagram

```
+-------------------------+
|           Car           |
+-------------------------+
| - brand: string         |
| - model: string         |
| - year: int             |
+-------------------------+
| + setBrand(b: string)   |
| + setModel(m: string)   |
| + setYear(y: int)       |
| + displayCarInfo(): void |
+-------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Car {
private:
    string brand;
    string model;
    int year;

public:
```

```
    void setBrand(string b) { brand = b; }
    void setModel(string m) { model = m; }
    void setYear(int y) { year = y; }

    void displayCarInfo() {
        cout << "Car: " << brand << " " << model << " (" << year << ")" << endl;
    }
};

int main() {
    Car car1;
    car1.setBrand("Toyota");
    car1.setModel("Corolla");
    car1.setYear(2020);
    car1.displayCarInfo();

    Car car2;
    car2.setBrand("Honda");
    car2.setModel("Civic");
    car2.setYear(2018);
    car2.displayCarInfo();

    return 0;
}
```

ผลการรัน

Car: Toyota Corolla (2020)

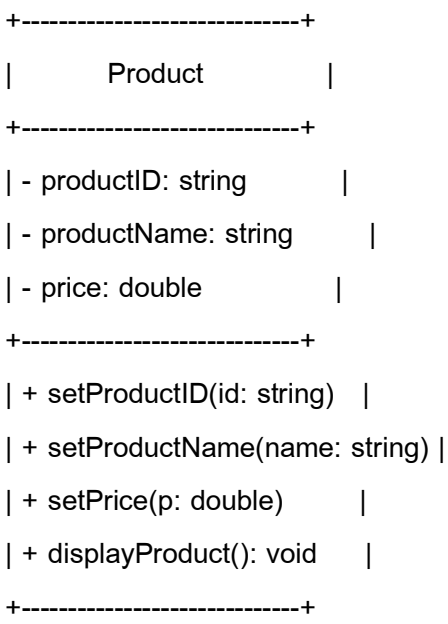
Car: Honda Civic (2018)

ตัวอย่างโปรแกรม 4 - ระบบสินค้า (Product)

โจทย์

สร้างคลาส Product เก็บรหัสสินค้า, ชื่อสินค้า, และราคาสินค้า พร้อมแสดงข้อมูลสินค้า

UML Class Diagram



โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Product {
private:
    string productID;
    string productName;
    double price;

public:
    void setProductID(string id) { productID = id; }
    void setProductName(string name) { productName = name; }
    void setPrice(double p) { if(p >= 0) price = p; }

    void displayProduct() {
        cout << "Product ID: " << productID << ", Name: " << productName << ", Price: $" <<
price << endl;
    }
};
```

```
int main() {
    Product prod;
    prod.setProductID("P1001");
    prod.setProductName("Laptop");
    prod.setPrice(1500.75);
    prod.displayProduct();

    return 0;
}
```

ผลการรัน

Product ID: P1001, Name: Laptop, Price: $1500.75

ตัวอย่างโปรแกรม 5 - ระบบนักเรียน (Student)

โจทย์

สร้างคลาส Student เก็บรหัสนักเรียน, ชื่อ, และเกรดเฉลี่ย พร้อมฟังก์ชันแสดงผลและตรวจสอบสถานะผ่าน/ไม่ผ่าน

UML Class Diagram

```
+-------------------------------+
|            Student            |
+-------------------------------+
| - studentID: string           |
| - name: string                |
| - gpa: double                 |
+-------------------------------+
| + setStudentID(id: string)    |
| + setName(n: string)          |
| + setGPA(g: double)           |
| + displayStudent(): void      |
| + isPassed(): bool            |
+-------------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Student {
private:
    string studentID;
    string name;
    double gpa;

public:
    void setStudentID(string id) { studentID = id; }
    void setName(string n) { name = n; }
    void setGPA(double g) { if(g >= 0 && g <= 4.0) gpa = g; }

    void displayStudent() {
        cout << "Student ID: " << studentID << ", Name: " << name << ", GPA: " << gpa;
        if(isPassed())
            cout << " (Passed)" << endl;
        else
            cout << " (Failed)" << endl;
    }

    bool isPassed() {
        return gpa >= 2.0;
    }
};

int main() {
    Student s1;
    s1.setStudentID("S001");
    s1.setName("John");
    s1.setGPA(3.5);
```

```
    s1.displayStudent();

    Student s2;
    s2.setStudentID("S002");
    s2.setName("Mary");
    s2.setGPA(1.8);
    s2.displayStudent();

    return 0;
}
```

ผลการรัน

Student ID: S001, Name: John, GPA: 3.5 (Passed)

Student ID: S002, Name: Mary, GPA: 1.8 (Failed)

Constructor และ Destructor ใน C++

---

1. Constructor คืออะไร?

- เป็นฟังก์ชันพิเศษที่ชื่อเหมือนกับชื่อคลาส
- ถูกเรียกโดยอัตโนมัติ **เมื่อมีการสร้าง (instantiate) object**
- ใช้สำหรับ **กำหนดค่าเริ่มต้น** ให้กับ attributes ของ object
- สามารถมี **หลาย constructor (constructor overloading)** ได้ โดยมีพารามิเตอร์ต่างกัน
- ไม่มีชนิดข้อมูลคืนค่า (ไม่มี void หรือค่าอื่น)
- ถ้าไม่เขียน constructor โปรแกรมจะสร้าง constructor ปริยาย (default constructor) ให้โดยอัตโนมัติ

---

2. Destructor คืออะไร?

- เป็นฟังก์ชันพิเศษที่ชื่อเหมือนกับชื่อคลาส แต่มีเครื่องหมาย ~ นำหน้า
- ถูกเรียกโดยอัตโนมัติ **เมื่อ object ถูกทำลาย (destroyed)** เช่น เมื่อออกจาก scope หรือถูก delete
- ใช้สำหรับ **เคลียร์ทรัพยากรที่ object ใช้** เช่น memory, file, connection
- ไม่มีพารามิเตอร์และไม่มีค่าคืน
- มีเพียงหนึ่ง destructor ต่อคลาสเท่านั้น (ไม่มี overloading)

---

3. ตัวอย่างโค้ดประกอบ Constructor และ Destructor

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    // Constructor แบบมีพารามิเตอร์
    Person(string n, int a) {
        name = n;
        age = a;
        cout << "Constructor called for " << name << endl;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p1("Alice", 30);
    p1.display();

    {
        Person p2("Bob", 25);
        p2.display();
```

```
    } // p2 ออกจาก scope ที่นี่ destructor จะถูกเรียก

    cout << "End of main()" << endl;
    return 0;
}
```

---

4. ผลการรันโปรแกรม

```
Constructor called for Alice
Name: Alice, Age: 30
Constructor called for Bob
Name: Bob, Age: 25
Destructor called for Bob
End of main()
Destructor called for Alice
```

---

5. คำอธิบายแทรก

- ตอนสร้าง p1 เรียก constructor และกำหนดชื่อและอายุ
- p1.display() แสดงข้อมูล
- ในบล็อก { ... } สร้าง p2 และเรียก constructor
- เมื่อบล็อกจบ p2 ออกจาก scope จึงเรียก destructor ของ p2
- เมื่อโปรแกรมจบ จะเรียก destructor ของ p1 อัตโนมัติ
- Constructor ใช้กำหนดค่าเริ่มต้น
- Destructor ใช้ปล่อยทรัพยากรหรือจัดการงานทำความสะอาด

---

6. Constructor แบบ Default และ Overloading

```
class Person {
private:
    string name;
    int age;

public:
    // Default constructor
    Person() {
        name = "Unknown";
```

```
        age = 0;
        cout << "Default constructor called" << endl;
    }

    // Parameterized constructor (Overloading)
    Person(string n, int a) {
        name = n;
        age = a;
        cout << "Parameterized constructor called for " << name << endl;
    }

    ~Person() {
        cout << "Destructor called for " << name << endl;
    }
};
```

---

7. สรุป

หัวข้อ	รายละเอียด
Constructor	ฟังก์ชันพิเศษเรียกตอนสร้าง object ใช้กำหนดค่าเริ่มต้น ไม่มีค่าคืน ชื่อเหมือนคลาส
Destructor	ฟังก์ชันพิเศษเรียกตอนทำลาย object ใช้ปล่อยทรัพยากร ชื่อเหมือนคลาสแต่มี ~ หน้า ไม่มีพารามิเตอร์และค่าคืน
Overloading	Constructor สามารถมีหลายแบบได้ โดยต่างพารามิเตอร์
Scope	Destructor เรียกตอน object ออกจาก scope หรือถูก delete

Constructor และ Destructor ใน C++ แบบเชิงลึก

Constructor และ Destructor ใน C++ (เชิงลึก) พร้อม UML

---

1. Constructor (ตัวสร้าง)

1.1 ลักษณะสำคัญ

- ชื่อเหมือนกับชื่อคลาส
- ไม่มีชนิดข้อมูลคืนค่า (ไม่ต้องใส่ void หรืออื่น ๆ)

- ถูกเรียกโดยอัตโนมัติเมื่อสร้างอ็อบเจกต์ (object instantiation)
- ใช้สำหรับ **กำหนดค่าเริ่มต้น** ให้กับอ็อบเจกต์ เช่น กำหนดค่าเริ่มต้นให้ attributes หรือจอง resource ที่จำเป็น
- สามารถมี **หลาย constructor** (constructor overloading) ด้วยพารามิเตอร์ต่างกันเพื่อความยืดหยุ่นในการสร้างอ็อบเจกต์

1.2 รูปแบบ Constructor

ชนิด Constructor	คำอธิบาย	ตัวอย่าง
Default Constructor	ไม่มีพารามิเตอร์ ใช้สร้าง object แบบไม่มีข้อมูลเริ่มต้น	Person()
Parameterized Constructor	รับพารามิเตอร์เพื่อกำหนดค่าเริ่มต้น	Person(string name, int age)
Copy Constructor	รับอ็อบเจกต์ตัวอื่นมาเป็นพารามิเตอร์ เพื่อสร้างสำเนา	Person(const Person &p)

1.3 ตัวอย่าง Constructor Overloading

```
class Person {
private:
    string name;
    int age;

public:
    Person() {          // Default constructor
        name = "Unknown";
        age = 0;
    }

    Person(string n, int a) {  // Parameterized constructor
        name = n;
        age = a;
    }

    Person(const Person &p) {  // Copy constructor
        name = p.name;
        age = p.age;
```

```
    }
};
```

---

2. Destructor (ตัวทำลาย)

2.1 ลักษณะสำคัญ

- ชื่อเหมือนชื่อคลาสแต่มี ~ นำหน้า เช่น ~Person()
- ไม่มีพารามิเตอร์และไม่มีค่าคืน
- เรียกโดยอัตโนมัติ **เมื่ออ็อบเจกต์ถูกทำลาย** (เช่น เมื่อออกจาก scope, delete pointer)
- ใช้สำหรับ **ปล่อยทรัพยากร** เช่น memory, ไฟล์, หรือการเชื่อมต่อที่เปิดไว้
- มีเพียง 1 destructor ต่อคลาส (ไม่มีการ overloading)

2.2 ทำไมต้องมี Destructor?

ถ้าใช้ dynamic memory (เช่น new), ต้องใช้ destructor เพื่อ delete ทรัพยากรนั้น มิฉะนั้นจะเกิด memory leak (หน่วยความจำรั่ว)

---

3. UML Class Diagram ตัวอย่าง

```
+-----------------------------------+
|              Person               |
+-----------------------------------+
| - name: string                    |
| - age: int                        |
+-----------------------------------+
| + Person()                        |  <<default constructor>>
| + Person(n: string, a: int)       |  <<parameterized constructor>>
| + Person(p: Person)               |  <<copy constructor>>
| + ~Person()                       |  <<destructor>>
| + display(): void                 |
+-----------------------------------+
```

---

4. ตัวอย่างโค้ดพร้อม Constructor และ Destructor

```
#include <iostream>
using namespace std;

class Person {
private:
```

```
    string name;
    int age;

public:
    // Default constructor
    Person() {
        name = "Unknown";
        age = 0;
        cout << "Default constructor called" << endl;
    }

    // Parameterized constructor
    Person(string n, int a) {
        name = n;
        age = a;
        cout << "Parameterized constructor called for " << name << endl;
    }

    // Copy constructor
    Person(const Person &p) {
        name = p.name;
        age = p.age;
        cout << "Copy constructor called for " << name << endl;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
```

```
int main() {
    Person p1;                  // เรียก default constructor
    p1.display();

    Person p2("Alice", 30);     // เรียก parameterized constructor
    p2.display();

    Person p3 = p2;             // เรียก copy constructor
    p3.display();

    return 0;
}
```

---

5. ผลการรันโปรแกรม (Output)

```
Default constructor called
Name: Unknown, Age: 0
Parameterized constructor called for Alice
Name: Alice, Age: 30
Copy constructor called for Alice
Name: Alice, Age: 30
Destructor called for Alice
Destructor called for Alice
Destructor called for Unknown
```

---

6. การทำงานเชิงลึก

- **Constructor** จะช่วยให้เราสร้างอ็อบเจกต์ได้อย่างปลอดภัยและสะดวก เพราะกำหนดค่าเริ่มต้นที่เหมาะสมทันที
- **Copy constructor** สำคัญในการสร้างสำเนาของ object แบบ deep copy (ถ้าต้องใช้ pointer ภายใน class)
- **Destructor** เป็นจุดสำคัญในการจัดการ resource lifecycle โดยเฉพาะ resource ที่ไม่ได้ถูกจัดการอัตโนมัติ (เช่น memory)
- การเรียก constructor และ destructor เป็นไปโดยอัตโนมัติ จึงช่วยลดความผิดพลาดและความยุ่งยากในการจัดการหน่วยความจำ

- C++ รองรับ constructor overloading เพื่อรองรับสถานการณ์ต่าง ๆ ในการสร้าง object

---

7. ข้อควรระวัง

- หาก class มี **pointer** ชี้ไปยัง memory ที่ dynamic allocate จำเป็นต้องเขียน copy constructor และ destructor เพื่อจัดการ memory อย่างถูกต้อง
- ถ้าไม่เขียน copy constructor เอง จะเกิด **shallow copy** ที่อาจทำให้เกิดปัญหา double delete หรือ memory corruption
- การใช้ smart pointer (เช่น std::unique_ptr) ช่วยลดภาระการเขียน destructor เอง

นี่คือตัวอย่างโปรแกรมที่ใช้ **Constructor** และ **Destructor** โดยแสดงครบทั้ง

- โจทย์
- UML Class Diagram
- โค้ด C++ พร้อมคำอธิบายแทรก
- ผลการรัน

ตัวอย่างโปรแกรม: คลาส Person ใช้ Constructor และ Destructor

---

1. โจทย์

สร้างคลาส Person ที่

- มีข้อมูลชื่อ (name) และอายุ (age)
- ใช้ **Constructor** เพื่อกำหนดค่าเริ่มต้นตอนสร้าง object
- ใช้ **Destructor** เพื่อแสดงข้อความเมื่อ object ถูกทำลาย
- มีฟังก์ชันแสดงข้อมูลชื่อและอายุ

---

2. UML Class Diagram

```
+---------------------------+
|          Person           |
+---------------------------+
| - name: string            |
| - age: int                |
+---------------------------+
| + Person()                | <<Default Constructor>>
| + Person(n: string, a: int)| <<Parameterized Constructor>>
| + ~Person()               | <<Destructor>>
```

| + display(): void |
+---------------------------+

---

3. โค้ด C++ พร้อมคำอธิบายแทรก

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    // Default constructor
    Person() {
        name = "Unknown";
        age = 0;
        cout << "Default constructor called" << endl;
    }

    // Parameterized constructor
    Person(string n, int a) {
        name = n;
        age = a;
        cout << "Parameterized constructor called for " << name << endl;
    }

    // Destructor
    ~Person() {
        cout << "Destructor called for " << name << endl;
    }

    // ฟังก์ชันแสดงข้อมูล
    void display() {
```

```
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p1;  // เรียก Default constructor
    p1.display();

    Person p2("Alice", 30);  // เรียก Parameterized constructor
    p2.display();

    {
        Person p3("Bob", 25);  // สร้าง object ใน scope ย่อย
        p3.display();
    } // p3 ออกจาก scope => destructor ถูกเรียก

    cout << "End of main()" << endl;

    return 0;
}
```

---

4. ผลการรันโปรแกรม

```
Default constructor called
Name: Unknown, Age: 0
Parameterized constructor called for Alice
Name: Alice, Age: 30
Parameterized constructor called for Bob
Name: Bob, Age: 25
Destructor called for Bob
End of main()
Destructor called for Alice
Destructor called for Unknown
```

---

5. คำอธิบายแทรก

- เมื่อสร้าง p1 ไม่มีพารามิเตอร์ จึงเรียก **Default constructor** กำหนดค่าเริ่มต้น "Unknown" และ 0
- เมื่อสร้าง p2 ส่งพารามิเตอร์ "Alice" และ 30 จึงเรียก **Parameterized constructor**
- ตัวแปร p3 สร้างในบล็อก {...} พอจบบล็อก p3 จะถูกทำลายและเรียก **Destructor** ทันที
- เมื่อ main() จบ จะเรียก Destructor ของ p1 และ p2 ตามลำดับอัตโนมัติ
- การใช้ Constructor และ Destructor ช่วยจัดการค่าเริ่มต้นและการทำความสะอาดอ็อบเจกต์ได้อย่างอัตโนมัติและปลอดภัย

ตัวอย่างโปรแกรม 5 ชุด ที่ใช้ **Constructor และ Destructor** พร้อม

- โจทย์
- UML Class Diagram
- โค้ด C++ พร้อมอธิบายแทรก
- ผลการรัน

มาให้ครบทุกโปรแกรม

โปรแกรมที่ 1: คลาส Person (ใช้ Constructor และ Destructor)

โจทย์

สร้างคลาส Person ที่เก็บชื่อและอายุ ใช้ Constructor กำหนดค่าเริ่มต้น และ Destructor แสดงข้อความเมื่อ object ถูกทำลาย

UML Class Diagram

```
+--------------------------+
|          Person          |
+--------------------------+
| - name: string           |
| - age: int               |
+--------------------------+
| + Person()               |
| + Person(n: string, a: int)|
| + ~Person()              |
| + display(): void        |
+--------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Person {
private:
    string name;
    int age;

public:
    Person() {  // Default constructor
        name = "Unknown";
        age = 0;
        cout << "Default constructor called" << endl;
    }
    Person(string n, int a) {  // Parameterized constructor
        name = n;
        age = a;
        cout << "Parameterized constructor called for " << name << endl;
    }
    ~Person() {  // Destructor
        cout << "Destructor called for " << name << endl;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p1;
    p1.display();

    Person p2("Alice", 30);
    p2.display();
```

```
    {
        Person p3("Bob", 25);
        p3.display();
    }  // p3 ออกจาก scope destructor ถูกเรียก

    cout << "End of main()" << endl;
    return 0;
}
```

ผลการรัน

```
Default constructor called
Name: Unknown, Age: 0
Parameterized constructor called for Alice
Name: Alice, Age: 30
Parameterized constructor called for Bob
Name: Bob, Age: 25
Destructor called for Bob
End of main()
Destructor called for Alice
Destructor called for Unknown
```

โปรแกรมที่ 2: คลาส BankAccount

โจทย์

สร้างคลาส BankAccount ที่เก็บเลขบัญชีและยอดเงิน ใช้ Constructor กำหนดเลขบัญชีและยอดเงินเริ่มต้น มีฟังก์ชันฝากและถอนเงิน และ Destructor แสดงข้อความเมื่อทำลาย object

---

UML Class Diagram

```
+-----------------------------+
|         BankAccount         |
+-----------------------------+
| - accountNumber: string     |
| - balance: double           |
```

```
+-----------------------------+
| + BankAccount(acc: string, bal: double) |
| + deposit(amount: double)     |
| + withdraw(amount: double)    |
| + getBalance(): double        |
| + ~BankAccount()              |
+-----------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class BankAccount {
private:
    string accountNumber;
    double balance;

public:
    BankAccount(string acc, double bal) {
        accountNumber = acc;
        balance = bal;
        cout << "BankAccount constructor for account " << accountNumber << endl;
    }

    void deposit(double amount) {
        if(amount > 0) balance += amount;
    }

    void withdraw(double amount) {
        if(amount > 0 && amount <= balance)
            balance -= amount;
        else
            cout << "Insufficient funds!" << endl;
    }
```

```
    double getBalance() {
        return balance;
    }

    ~BankAccount() {
        cout << "BankAccount destructor for account " << accountNumber << endl;
    }
};

int main() {
    BankAccount acc1("123456", 1000);
    acc1.deposit(500);
    acc1.withdraw(200);
    cout << "Balance: " << acc1.getBalance() << endl;

    return 0;
}
```

ผลการรัน

```
BankAccount constructor for account 123456
Balance: 1300
BankAccount destructor for account 123456
```

โปรแกรมที่ 3: คลาส Car

โจทย์

สร้างคลาส Car เก็บยี่ห้อ รุ่น และปีผลิต ใช้ Constructor กำหนดค่าเริ่มต้น มีฟังก์ชันแสดงข้อมูลรถ และ Destructor

---

UML Class Diagram

```
+--------------------------+
|           Car            |
+--------------------------+
```

```
| - brand: string            |
| - model: string            |
| - year: int                |
+--------------------------+
| + Car(b: string, m: string, y: int) |
| + display(): void          |
| + ~Car()                   |
+--------------------------+
```

โค้ด C++ พร้อมอธิบายแทรก

```
#include <iostream>
using namespace std;

class Car {
private:
    string brand;
    string model;
    int year;

public:
    Car(string b, string m, int y) {
        brand = b;
        model = m;
        year = y;
        cout << "Car constructor called for " << brand << endl;
    }

    void display() {
        cout << "Car: " << brand << " " << model << " (" << year << ")" << endl;
    }

    ~Car() {
        cout << "Car destructor called for " << brand << endl;
    }
```