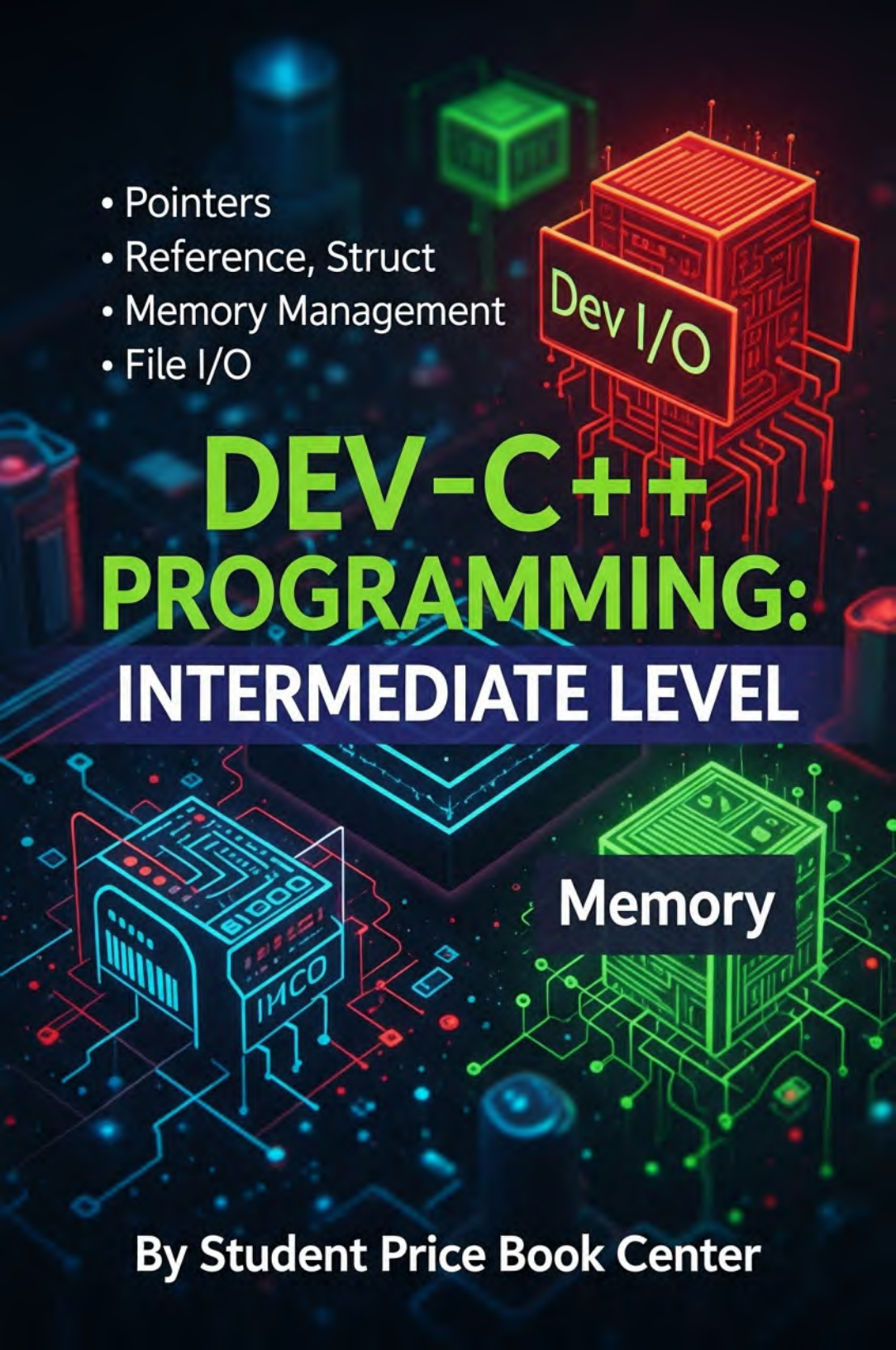
- Pointers
- Reference, Struct
- Memory Management
- File I/O



DEV-C++ PROGRAMMING:

INTERMEDIATE LEVEL

By Student Price Book Center

คำนำ

การเรียนรู้การเขียนโปรแกรมในระดับกลาง (Intermediate Level) ถือเป็นก้าวสำคัญที่นักพัฒนาซอฟต์แวร์ทุกคนต้องก้าวผ่าน เพื่อเพิ่มพูนความเข้าใจจากพื้นฐานสู่การสร้างโปรแกรมที่มีโครงสร้างซับซ้อนมากขึ้น หนังสือเล่มนี้จัดทำขึ้นเพื่อเสริมสร้างความรู้และทักษะในด้านที่เป็นแกนหลักของการพัฒนาโปรแกรมเชิงโครงสร้างและการจัดการหน่วยความจำในภาษา C++ ซึ่งเป็นภาษาที่ทรงพลังและใช้กันอย่างแพร่หลายในแวดวงวิศวกรรมซอฟต์แวร์ ระบบฝังตัว และงานวิจัยคอมพิวเตอร์

เนื้อหาเริ่มต้นจากหัวข้อสำคัญอย่าง **พอยน์เตอร์ (Pointers)** ซึ่งเป็นแนวคิดพื้นฐานของการเข้าถึงและจัดการหน่วยความจำโดยตรง ผู้อ่านจะได้เรียนรู้การประกาศและใช้งานพอยน์เตอร์ การใช้ตัวดำเนินการ & และ * ตลอดจนการประยุกต์ใช้ร่วมกับอาเรย์และฟังก์ชัน ต่อเนื่องด้วย **อ้างอิง (Reference)** ซึ่งเปรียบเทียบความแตกต่างระหว่าง pointer และ reference พร้อมนำเสนอเทคนิคการส่งค่าผ่าน reference เพื่อเพิ่มประสิทธิภาพในการเขียนฟังก์ชันและลดการคัดลอกข้อมูล

ถัดมา หนังสือจะพาผู้อ่านเข้าสู่หัวข้อ **โครงสร้างข้อมูลเบื้องต้น (Structs)** ซึ่งเป็นการรวมข้อมูลหลายประเภทให้เป็นชุดเดียวกัน โดยมีทั้งการสร้าง struct ซ้อนกัน การจัดการ array of struct และ pointer to struct เพื่อให้สามารถจัดการข้อมูลได้อย่างยืดหยุ่นและมีระบบยิ่งขึ้น หัวข้อถัดไปคือ **การจัดการหน่วยความจำ (Memory Management)** ซึ่งเน้นการใช้ new และ delete การสร้าง dynamic array และการป้องกันปัญหา memory leak ซึ่งเป็นหนึ่งในจุดอ่อนที่พบได้บ่อยในโปรแกรม C++

ท้ายที่สุด หนังสือได้นำเสนอเนื้อหาเกี่ยวกับ **การจัดการไฟล์ (File I/O)** ครอบคลุมทั้งการใช้งานคลาส ifstream, ofstream และ fstream สำหรับการอ่านและเขียนไฟล์ในรูปแบบข้อความและไบนารี ซึ่งเป็นทักษะที่สำคัญสำหรับการพัฒนาโปรแกรมที่ต้องจัดเก็บและดึงข้อมูลจากภายนอก เช่น ฐานข้อมูลหรือไฟล์บันทึกต่าง ๆ

ผู้เขียนหวังว่าหนังสือเล่มนี้จะเป็นแนวทางที่ชัดเจนในการพัฒนาทักษะเขียนโปรแกรมในระดับกลาง ช่วยให้ผู้อ่านเข้าใจหลักการสำคัญอย่างลึกซึ้ง และสามารถนำความรู้ไปประยุกต์ใช้ในการสร้างโปรแกรมจริงได้อย่างมั่นใจและมีประสิทธิภาพ ทั้งยังเป็นรากฐานสำหรับการต่อยอดสู่การเรียนรู้ระดับสูงต่อไปในอนาคต

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 6
พอยน์เตอร์
(Pointers)

เนื้อหา

- ความรู้เบื้องต้นเกี่ยวกับพอยน์เตอร์ (Pointers)
- พอยน์เตอร์ (Pointers) ใน C++ เชิงลึก
- พอยน์เตอร์กับอาเรย์ (Pointers and Arrays)
- พอยน์เตอร์กับฟังก์ชัน (Pointers and Functions)
- พอยน์เตอร์ (Pointers) ใน C++ (Dev-C++) เพิ่มเติม
- การประกาศและใช้ Pointer ใน C++
- ข้อมูลเพิ่มเติมแบบเจาะลึก
- การใช้ & (address-of) และ * (dereference) ใน C++
- รายละเอียด Pointer กับอาเรย์ (Array and Pointer)
- พอยน์เตอร์กับอาเรย์ และฟังก์ชัน
- ตัวอย่างบูรณาการและประยุกต์ใช้

บทนำ

ในกระบวนการเขียนโปรแกรมเชิงโครงสร้างและระบบ การเข้าถึงหน่วยความจำและการจัดการข้อมูลในระดับต่ำถือเป็นทักษะที่สำคัญอย่างยิ่ง โดยเฉพาะในภาษาโปรแกรมอย่าง C หรือ C++ แนวคิดเกี่ยวกับ “**พอยน์เตอร์**” (Pointers) จึงมีบทบาทสำคัญในฐานะเครื่องมือที่ช่วยให้นักพัฒนาสามารถเข้าถึงตำแหน่งหน่วยความจำโดยตรง จัดการข้อมูลแบบไดนามิก และสร้างโครงสร้างข้อมูลที่ซับซ้อนได้อย่างมีประสิทธิภาพ

พอยน์เตอร์ คือ ตัวแปรชนิดพิเศษที่ใช้ในการเก็บที่อยู่ (address) ของตัวแปรอื่น ๆ ซึ่งแตกต่างจากตัวแปรทั่วไปที่มักเก็บเพียงค่าข้อมูลเท่านั้น ในบทนี้จะอธิบายพื้นฐานของการ **ประกาศและใช้งานพอยน์เตอร์**, ความเข้าใจในการใช้ **เครื่องหมาย & (address-of)** เพื่อเข้าถึงที่อยู่ของตัวแปร และ **เครื่องหมาย * (dereference)** เพื่อเข้าถึงค่าที่อยู่ในตำแหน่งหน่วยความจำนั้น

นอกจากนี้ยังจะศึกษาการประยุกต์ใช้พอยน์เตอร์กับ **อาเรย์ (Arrays)** และ **ฟังก์ชัน (Functions)** ซึ่งเป็นแนวทางสำคัญในการทำงานกับข้อมูลแบบกลุ่ม และการส่งค่าผ่านหน่วยความจำ

แทนค่าจริง ซึ่งช่วยให้โปรแกรมทำงานได้มีประสิทธิภาพมากขึ้น และสามารถพัฒนาโครงสร้างข้อมูลขั้นสูง เช่น linked list, tree หรือ graph ได้อย่างยืดหยุ่น

ความเข้าใจอย่างลึกซึ้งเกี่ยวกับพอยน์เตอร์ไม่เพียงแต่ช่วยให้ผู้เรียนสามารถเขียนโปรแกรมที่มีประสิทธิภาพเท่านั้น แต่ยังเป็นรากฐานสำคัญในการเรียนรู้วิชาอื่น ๆ เช่น ระบบปฏิบัติการ, โครงสร้างข้อมูล, และการพัฒนาโปรแกรมระบบ (System Programming) ในระดับสูงต่อไป

ความรู้เบื้องต้นเกี่ยวกับพอยน์เตอร์ (Pointers)

□ 1. การประกาศและใช้ Pointer

พอยน์เตอร์ (Pointer) คือ ตัวแปรที่เก็บ "ที่อยู่ของตัวแปรอื่น"

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

int main() {
    int a = 10;
    int* p;     // ประกาศ pointer ชนิด int
    p = &a;     // ให้ p ชี้ไปยังตัวแปร a

    cout << "ค่าของ a: " << a << endl;
    cout << "ที่อยู่ของ a: " << &a << endl;
    cout << "ค่าของ p (ที่อยู่ที่มันชี้): " << p << endl;
    cout << "ค่าที่ p ชี้ไปหา (*p): " << *p << endl;

    return 0;
}
```

□ ผลลัพธ์ตัวอย่าง

```
ค่าของ a: 10
ที่อยู่ของ a: 0x61ff08
ค่าของ p: 0x61ff08
ค่าที่ p ชี้ไปหา (*p): 10
```

□ 2. การใช้ & (address-of) และ * (dereference)

- &variable □ ใช้เพื่อ **ดึงที่อยู่ของตัวแปร**
- *pointer □ ใช้เพื่อ **ดึงค่าจากตำแหน่งที่ pointer ชี้**

```
#include <iostream>
```

```
using namespace std;

int main() {
    int b = 20;
    int* ptr = &b; // ptr ชี้ไปยัง b

    cout << "ที่อยู่ของ b: " << &b << endl;
    cout << "ค่าที่ ptr ชี้ไปหา: " << *ptr << endl;

    *ptr = 99; // เปลี่ยนค่าของ b ผ่าน pointer

    cout << "ค่าของ b หลังเปลี่ยนผ่าน ptr: " << b << endl;

    return 0;
}
```

ผลลัพธ์

```
ที่อยู่ของ b: 0x...
ค่าที่ ptr ชี้ไปหา: 20
ค่าของ b หลังเปลี่ยนผ่าน ptr: 99
```

---

3. Pointer กับอาเรย์ (Array)

พอยน์เตอร์สามารถใช้กับอาเรย์เพื่อเข้าถึงสมาชิกได้แบบเดียวกับ index

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

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int* p = arr; // พอยน์เตอร์ชี้ไปยัง arr[0]

    for (int i = 0; i < 5; i++) {
        cout << "arr[" << i << "] = " << *(p + i) << endl;
    }

    return 0;
```

```
}
```

☐ **ผลลัพธ์**

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

☐ **4. Pointer กับฟังก์ชัน**

ใช้พอยน์เตอร์ส่งค่าระหว่างฟังก์ชันแบบ "อ้างอิง"

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

void updateValue(int* x) {
    *x = *x + 100;
}

int main() {
    int num = 50;
    updateValue(&num); // ส่งที่อยู่ของ num เข้าไป
    cout << "ค่าหลัง update: " << num << endl;

    return 0;
}
```

☐ **ผลลัพธ์**

```
ค่าหลัง update: 150
```

☐ **สรุป:**

สัญลักษณ์	ความหมาย
*	การประกาศหรือเข้าถึงค่าจาก pointer (dereference)
&	การหา address ของตัวแปร (address-of)
int* p	ประกาศ pointer ที่เก็บ address ของ int
*p = 20	กำหนดค่าที่ตำแหน่ง pointer ชี้อยู่ให้เป็น 20

พอยน์เตอร์ (Pointers) ใน C++ เชิงลึก

พอยน์เตอร์ คือตัวแปรชนิดหนึ่งที่ไม่ได้เก็บค่าข้อมูลโดยตรง แต่เก็บ **ที่อยู่ของหน่วยความจำ (memory address)** ของตัวแปรอื่น ๆ พูดง่าย ๆ คือ มันชี้ไปยังตำแหน่งที่ข้อมูลจริงถูกเก็บไว้

ทำไมต้องใช้พอยน์เตอร์?

1. **การจัดการหน่วยความจำแบบพลวัต (Dynamic Memory Allocation):** สามารถจองและคืนหน่วยความจำในระหว่างที่โปรแกรมกำลังทำงานได้ ซึ่งจำเป็นสำหรับการจัดการข้อมูลที่มีขนาดไม่แน่นอน
2. **การเข้าถึงข้อมูลอย่างมีประสิทธิภาพ:** ในบางกรณี การใช้พอยน์เตอร์สามารถทำให้โปรแกรมทำงานได้เร็วขึ้น โดยเฉพาะเมื่อต้องจัดการกับโครงสร้างข้อมูลขนาดใหญ่ เช่น อาเรย์ หรืออ็อบเจกต์
3. **การทำงานกับฟังก์ชัน (Function Parameters):** สามารถส่งผ่านตัวแปรไปยังฟังก์ชันด้วย "reference" ผ่านพอยน์เตอร์ ทำให้ฟังก์ชันสามารถแก้ไขค่าของตัวแปรต้นฉบับได้ (pass by pointer)
4. **โครงสร้างข้อมูลที่ซับซ้อน:** ใช้สร้างโครงสร้างข้อมูลแบบ linked list, tree, graph ซึ่งจำเป็นต้องมีการเชื่อมโยงข้อมูลเข้าด้วยกัน

---

1. การประกาศและใช้งานพอยน์เตอร์

การประกาศพอยน์เตอร์จะบอกคอมไพเลอร์ว่าตัวแปรนี้จะเก็บที่อยู่ของหน่วยความจำ และที่อยู่นั้นชี้ไปยังข้อมูลชนิดใด

C++

```
#include <iostream>

int main() {
    // 1.1 การประกาศตัวแปรปกติ (value variable)
    int score = 100;
    double price = 99.99;

    std::cout << "--- การประกาศและใช้งานพอยน์เตอร์ ---" << std::endl;
    std::cout << "ค่าของ score: " << score << std::endl;       // แสดงค่า 100
    std::cout << "ค่าของ price: " << price << std::endl;       // แสดงค่า 99.99

    // 1.2 การประกาศพอยน์เตอร์ (pointer variable)
    // ชนิดข้อมูลที่พอยน์เตอร์ชี้ไป * ชื่อพอยน์เตอร์;
    int *ptrScore;      // พอยน์เตอร์ชื่อ ptrScore ชี้ไปยัง int
    double *ptrPrice;   // พอยน์เตอร์ชื่อ ptrPrice ชี้ไปยัง double
```

```
    char *ptrChar;       // พอยน์เตอร์ชื่อ ptrChar ชี้ไปยัง char

    std::cout << "\nพอยน์เตอร์เมื่อประกาศแล้ว (ยังไม่กำหนดค่า):" << std::endl;
    std::cout << "ptrScore (อาจเป็นค่าขยะ): " << ptrScore << std::endl;  // แสดงที่อยู่ขยะ หรือ NULL (0x0)
    std::cout << "ptrPrice (อาจเป็นค่าขยะ): " << ptrPrice << std::endl;  // แสดงที่อยู่ขยะ หรือ NULL (0x0)
    // การเข้าถึงค่าที่พอยน์เตอร์ขยะชี้ไปเป็นอันตราย (dereferencing a wild pointer)

    // 1.3 การกำหนดค่าให้พอยน์เตอร์
    // พอยน์เตอร์จะเก็บที่อยู่ของตัวแปรอื่น โดยใช้โอเปอเรเตอร์ & (address-of)
    ptrScore = &score;  // ptrScore ชี้ไปยังที่อยู่ของตัวแปร score
    ptrPrice = &price;  // ptrPrice ชี้ไปยังที่อยู่ของตัวแปร price

    std::cout << "\nหลังกำหนดค่าให้พอยน์เตอร์:" << std::endl;
    std::cout << "ที่อยู่ของ score (&score): " << &score << std::endl;
    std::cout << "ค่าที่ ptrScore เก็บ (ptrScore): " << ptrScore << std::endl; // ควรเป็นค่าเดียวกับ &score

    std::cout << "ที่อยู่ของ price (&price): " << &price << std::endl;
    std::cout << "ค่าที่ ptrPrice เก็บ (ptrPrice): " << ptrPrice << std::endl; // ควรเป็นค่าเดียวกับ &price

    // 1.4 การใช้พอยน์เตอร์ที่ไม่ได้ชี้ไปที่ไหน (nullptr)
    // การกำหนดค่าเป็น nullptr (C++11) หรือ NULL (C++ เก่า) เพื่อให้พอยน์เตอร์ไม่ชี้ไปที่ไหนเลย
    // เป็นการป้องกันการเข้าถึงหน่วยความจำโดยไม่ตั้งใจ (null pointer dereference)
    int *nullPtr = nullptr;
    std::cout << "\nค่าของ nullPtr: " << nullPtr << std::endl; // จะแสดง 0x0 หรือ 0 (หมายถึงไม่ชี้ไปไหน)
    // if (nullPtr != nullptr) { ... } // ตรวจสอบก่อน dereference เสมอ

    return 0;
}
```

ผลลัพธ์ (ตัวอย่าง):

(ที่อยู่หน่วยความจำจะแตกต่างกันไปในแต่ละครั้งที่รันโปรแกรม)

--- การประกาศและใช้งานพอยน์เตอร์ ---

ค่าของ score: 100

ค่าของ price: 99.99

พอยน์เตอร์เมื่อประกาศแล้ว (ยังไม่กำหนดค่า):

ptrScore (อาจเป็นค่าขยะ): 0x7ffe098319dc // ค่าขยะ

ptrPrice (อาจเป็นค่าขยะ): 0x7ffe098319e0 // ค่าขยะ

หลังกำหนดค่าให้พอยน์เตอร์:

ที่อยู่ของ score (&score): 0x7ffe098319d4

ค่าที่ ptrScore เก็บ (ptrScore): 0x7ffe098319d4 // ชี้ไปที่ score

ที่อยู่ของ price (&price): 0x7ffe098319d8

ค่าที่ ptrPrice เก็บ (ptrPrice): 0x7ffe098319d8 // ชี้ไปที่ price

ค่าของ nullPtr: 0

คำอธิบาย:

- **int *ptrScore;**: คือการประกาศตัวแปร ptrScore ให้เป็นพอยน์เตอร์ที่สามารถเก็บที่อยู่ของตัวแปรชนิด int ได้
- **ptrScore = &score;**: เป็นการกำหนดให้ ptrScore เก็บ **ที่อยู่ของหน่วยความจำ** ของตัวแปร score โดยใช้โอเปอเรเตอร์ **& (address-of operator)**
- ค่าของพอยน์เตอร์เองคือ **ที่อยู่ของหน่วยความจำ** ซึ่งมักจะแสดงเป็นเลขฐานสิบหก (hexadecimal) เช่น 0x7ffe098319d4
- การประกาศพอยน์เตอร์แล้วไม่ได้กำหนดค่าให้ (หรือกำหนดเป็นค่าขยะ) แล้วนำไปใช้ทันทีเป็นอันตรายมาก เพราะมันอาจชี้ไปยังหน่วยความจำที่ไม่ได้เป็นของโปรแกรมเรา ซึ่งจะทำให้เกิดข้อผิดพลาดรันไทม์ (runtime error)

---

2. การใช้ & (address-of) และ * (dereference)

สองโอเปอเรเตอร์นี้เป็นหัวใจสำคัญในการทำงานกับพอยน์เตอร์

- **& (Address-of Operator):**
  - ใช้วางหน้าชื่อตัวแปรปกติ
  - คืนค่า **ที่อยู่ของหน่วยความจำ (address)** ของตัวแปรนั้น ๆ
  - เช่น &score จะคืนค่าที่อยู่ของตัวแปร score

- * **(Dereference Operator หรือ Indirection Operator):**
  - ใช้วางหน้าชื่อพอยน์เตอร์
  - ใช้เพื่อ **เข้าถึงค่า (value)** ที่เก็บอยู่ในที่อยู่หน่วยความจำที่พอยน์เตอร์นั้นชี้ไป
  - เช่น *ptrScore จะคืนค่าที่ถูกเก็บอยู่ที่ตำแหน่งหน่วยความจำที่ ptrScore ชี้อยู่

C++

```
#include <iostream>

int main() {
    int num = 42;          // ตัวแปรปกติ
    int *ptr = &num;       // พอยน์เตอร์ ptr ชี้ไปที่ num

    std::cout << "--- การใช้ & (address-of) และ * (dereference) ---" << std::endl;

    // 2.1 ใช้ & (Address-of Operator) เพื่อหาที่อยู่ของตัวแปร
    std::cout << "ค่าของ num: " << num << std::endl;              // 42
    std::cout << "ที่อยู่ของ num (&num): " << &num << std::endl; // เช่น 0x7ffc8f3a3dcc

    // 2.2 ค่าที่พอยน์เตอร์เก็บ (คือที่อยู่ของ num)
    std::cout << "ค่าที่ ptr เก็บ (ptr): " << ptr << std::endl;   // เหมือนกับ &num

    // 2.3 ใช้ * (Dereference Operator) เพื่อเข้าถึงค่าที่พอยน์เตอร์ชี้ไป
    std::cout << "ค่าที่ ptr ชี้ไป (*ptr): " << *ptr << std::endl; // 42 (ค่าของ num)

    // 2.4 การแก้ไขค่าผ่านพอยน์เตอร์ (Dereferencing to modify)
    *ptr = 100; // เปลี่ยนค่าที่ ptr ชี้ไป (ซึ่งก็คือ num) ให้เป็น 100
    std::cout << "\nหลังเปลี่ยนค่าผ่าน *ptr = 100;" << std::endl;
    std::cout << "ค่าของ num: " << num << std::endl;              // ตอนนี้ num เป็น 100
    std::cout << "ค่าที่ ptr ชี้ไป (*ptr): " << *ptr << std::endl; // ตอนนี้ *ptr ก็เป็น 100

    // ตัวอย่างการใช้พอยน์เตอร์กับ string
    std::string message = "Hello";
    std::string *ptrMessage = &message;

    std::cout << "\n--- พอยน์เตอร์กับ std::string ---" << std::endl;
```

```
    std::cout << "ค่าของ message: " << message << std::endl;
    std::cout << "ที่อยู่ของ message (&message): " << &message << std::endl;
    std::cout << "ค่าที่ ptrMessage เก็บ (ptrMessage): " << ptrMessage << std::endl;
    std::cout << "ค่าที่ ptrMessage ชี้ไป (*ptrMessage): " << *ptrMessage << std::endl;

    *ptrMessage = "Goodbye"; // เปลี่ยนค่าของ message ผ่านพอยน์เตอร์
    std::cout << "ค่าของ message หลังเปลี่ยน: " << message << std::endl;

    return 0;
}
```

ผลลัพธ์ (ตัวอย่าง):

```
--- การใช้ & (address-of) และ * (dereference) ---
ค่าของ num: 42
ที่อยู่ของ num (&num): 0x7ffe6b3a2a4c
ค่าที่ ptr เก็บ (ptr): 0x7ffe6b3a2a4c
ค่าที่ ptr ชี้ไป (*ptr): 42

หลังเปลี่ยนค่าผ่าน *ptr = 100;
ค่าของ num: 100
ค่าที่ ptr ชี้ไป (*ptr): 100

--- พอยน์เตอร์กับ std::string ---
ค่าของ message: Hello
ที่อยู่ของ message (&message): 0x7ffe6b3a2a50
ค่าที่ ptrMessage เก็บ (ptrMessage): 0x7ffe6b3a2a50
ค่าที่ ptrMessage ชี้ไป (*ptrMessage): Hello
ค่าของ message หลังเปลี่ยน: Goodbye
```

คำอธิบาย:

- int *ptr = # **ตรงนี้ * เป็นส่วนหนึ่งของการประกาศชนิดข้อมูล** หมายถึง ptr เป็นพอยน์เตอร์ชี้ไปที่ int
- *ptr = 100; **ตรงนี้ * เป็นโอเปอเรเตอร์ Dereference** หมายถึง "ไปที่ตำแหน่งที่ ptr ชี้อยู่แล้วเปลี่ยนค่าในนั้นเป็น 100"

พอยน์เตอร์กับอาเรย์ (Pointers and Arrays)

ใน C++ ชื่อของอาเรย์ (เมื่อใช้อย่างเดียวโดยไม่มีดัชนี) มักจะถูกตีความว่าเป็น **พอยน์เตอร์ไปยังสมาชิกตัวแรกของอาเรย์** นี่เป็นความสัมพันธ์ที่สำคัญมาก

C++

```
#include <iostream>

int main() {
    int numbers[] = {10, 20, 30, 40, 50}; // อาเรย์ 5 ช่อง
    int size = sizeof(numbers) / sizeof(numbers[0]);

    std::cout << "--- พอยน์เตอร์กับอาเรย์ ---" << std::endl;

    // 3.1 ชื่ออาเรย์คือพอยน์เตอร์ไปยังสมาชิกตัวแรก
    std::cout << "ที่อยู่ของ numbers[0] (&numbers[0]): " << &numbers[0] << std::endl;
    std::cout << "ชื่ออาเรย์ (numbers): " << numbers << std::endl; // จะแสดงที่อยู่เดียวกับ
&numbers[0]

    // 3.2 การสร้างพอยน์เตอร์ให้ชี้ไปที่อาเรย์
    int *ptrArray = numbers; // ptrArray ชี้ไปที่ numbers[0]
    std::cout << "ค่าของ ptrArray: " << ptrArray << std::endl;

    // 3.3 การเข้าถึงสมาชิกอาเรย์โดยใช้พอยน์เตอร์ (Pointer Arithmetic)
    // การเพิ่ม/ลดพอยน์เตอร์จะเลื่อนไปตามขนาดของชนิดข้อมูลที่ชี้
    std::cout << "\nเข้าถึงสมาชิกโดยใช้พอยน์เตอร์และ Pointer Arithmetic:" << std::endl;
    std::cout << "numbers[0] หรือ *ptrArray: " << *ptrArray << std::endl;       // 10
    std::cout << "numbers[1] หรือ *(ptrArray + 1): " << *(ptrArray + 1) << std::endl; // 20
    std::cout << "numbers[2] หรือ *(ptrArray + 2): " << *(ptrArray + 2) << std::endl; // 30

    // 3.4 การวนลูปผ่านอาเรย์โดยใช้พอยน์เตอร์
    std::cout << "\nวนลูปผ่านอาเรย์ด้วยพอยน์เตอร์:" << std::endl;
    for (int i = 0; i < size; ++i) {
        // สามารถใช้ numbers[i] หรือ *(numbers + i) หรือ *(ptrArray + i)
        std::cout << "Element " << i << ": " << *(numbers + i) << std::endl;
    }
```

```
    // อีกวิธีในการวนลูป: เลื่อนพอยน์เตอร์ไปข้างหน้า
    std::cout << "\nวนลูปผ่านอาเรย์ด้วยการเลื่อนพอยน์เตอร์:" << std::endl;
    for (int *p = numbers; p < (numbers + size); ++p) { // p ชี้ไปแต่ละตำแหน่ง
        std::cout << "Value: " << *p << std::endl; // Dereference p เพื่อเข้าถึงค่า
    }

    return 0;
}
```

ผลลัพธ์ (ตัวอย่าง):

```
--- พอยน์เตอร์กับอาเรย์ ---
ที่อยู่ของ numbers[0] (&numbers[0]): 0x7ffe098319a0
ชื่ออาเรย์ (numbers): 0x7ffe098319a0
ค่าของ ptrArray: 0x7ffe098319a0

เข้าถึงสมาชิกโดยใช้พอยน์เตอร์และ Pointer Arithmetic:
numbers[0] หรือ *ptrArray: 10
numbers[1] หรือ *(ptrArray + 1): 20
numbers[2] หรือ *(ptrArray + 2): 30

วนลูปผ่านอาเรย์ด้วยพอยน์เตอร์:
Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50

วนลูปผ่านอาเรย์ด้วยการเลื่อนพอยน์เตอร์:
Value: 10
Value: 20
Value: 30
Value: 40
Value: 50
```

คำอธิบาย:

- numbers (ชื่ออาเรย์เฉย ๆ) จะมีค่าเท่ากับ &numbers[0] เสมอ

- **Pointer Arithmetic:** เมื่อคุณเพิ่ม 1 เข้าไปในพอยน์เตอร์ (เช่น ptrArray + 1) พอยน์เตอร์จะเลื่อนไปข้างหน้าตามขนาดของชนิดข้อมูลที่ชี้ (เช่น ถ้า int มี 4 ไบต์ ptrArray + 1 จะเลื่อนไป 4 ไบต์) สิ่งนี้ทำให้ *(ptrArray + i) เทียบเท่ากับ ptrArray[i]

พอยน์เตอร์กับฟังก์ชัน (Pointers and Functions)

การใช้พอยน์เตอร์ในการส่งผ่านพารามิเตอร์ไปยังฟังก์ชัน หรือการส่งคืนพอยน์เตอร์จากฟังก์ชัน มีประโยชน์มาก

4.1 Pass by Pointer (ส่งผ่านด้วยพอยน์เตอร์)

เมื่อส่งพอยน์เตอร์เข้าไปในฟังก์ชัน ฟังก์ชันจะได้รับ **สำเนาของที่อยู่หน่วยความจำ** แต่ที่อยู่เหล่านั้นยังคงชี้ไปยังตำแหน่งหน่วยความจำเดิม ทำให้ฟังก์ชันสามารถ **แก้ไขค่าต้นฉบับ** ของตัวแปรที่อยู่ภายนอกฟังก์ชันได้

C++

```
#include <iostream>

// ฟังก์ชันสำหรับเพิ่มค่าตัวแปรผ่านพอยน์เตอร์
void increment(int *valuePtr) {
    std::cout << "  ในฟังก์ชัน increment:" << std::endl;
    std::cout << "  ที่อยู่ของ valuePtr ในฟังก์ชัน: " << &valuePtr << std::endl; // ที่อยู่ของพอยน์เตอร์เอง (ต่างจากที่อยู่ภายนอก)
    std::cout << "  ค่าที่ valuePtr เก็บ (ชี้ไป): " << valuePtr << std::endl;   // ที่อยู่เดียวกับ num ภายนอก
    std::cout << "  ค่าที่ valuePtr ชี้ไป (*valuePtr) ก่อนเพิ่ม: " << *valuePtr << std::endl;

    (*valuePtr)++; // เพิ่มค่าที่พอยน์เตอร์ชี้ไป
    // หรือ *valuePtr = *valuePtr + 1;

    std::cout << "  ค่าที่ valuePtr ชี้ไป (*valuePtr) หลังเพิ่ม: " << *valuePtr << std::endl;
}

// ฟังก์ชันสำหรับแลกเปลี่ยนค่าของตัวแปรสองตัวผ่านพอยน์เตอร์
void swap(int *a, int *b) {
    int temp = *a; // เก็บค่าที่ a ชี้ไปไว้ใน temp
    *a = *b;       // เอาค่าที่ b ชี้ไปมาใส่ในตำแหน่งที่ a ชี้ไป
    *b = temp;     // เอาค่าใน temp (คือค่าเดิมของ a) มาใส่ในตำแหน่งที่ b ชี้ไป
```

```
}

int main() {
    std::cout << "--- พอยน์เตอร์กับฟังก์ชัน (Pass by Pointer) ---" << std::endl;

    int num = 10;
    std::cout << "ก่อนเรียก increment: num = " << num << std::endl;
    std::cout << "ที่อยู่ของ num (&num): " << &num << std::endl;

    increment(&num); // ส่งที่อยู่ของ num ไปยังฟังก์ชัน increment
    std::cout << "หลังเรียก increment: num = " << num << std::endl; // num จะเปลี่ยนเป็น 11

    std::cout << "\n--- ตัวอย่าง Swap ด้วยพอยน์เตอร์ ---" << std::endl;
    int x = 5, y = 10;
    std::cout << "ก่อนเรียก swap: x = " << x << ", y = " << y << std::endl;
    swap(&x, &y); // ส่งที่อยู่ของ x และ y
    std::cout << "หลังเรียก swap: x = " << x << ", y = " << y << std::endl; // x และ y จะสลับค่ากัน

    return 0;
}
```

ผลลัพธ์ (ตัวอย่าง):

```
--- พอยน์เตอร์กับฟังก์ชัน (Pass by Pointer) ---
ก่อนเรียก increment: num = 10
ที่อยู่ของ num (&num): 0x7ffe427b5e4c
  ในฟังก์ชัน increment:
  ที่อยู่ของ valuePtr ในฟังก์ชัน: 0x7ffe427b5e38
  ค่าที่ valuePtr เก็บ (ชี้ไป): 0x7ffe427b5e4c
  ค่าที่ valuePtr ชี้ไป (*valuePtr) ก่อนเพิ่ม: 10
  ค่าที่ valuePtr ชี้ไป (*valuePtr) หลังเพิ่ม: 11
หลังเรียก increment: num = 11

--- ตัวอย่าง Swap ด้วยพอยน์เตอร์ ---
ก่อนเรียก swap: x = 5, y = 10
หลังเรียก swap: x = 10, y = 5
```

คำอธิบาย:

- void increment(int *valuePtr): ฟังก์ชันนี้รับพอยน์เตอร์ชนิด int เป็นพารามิเตอร์
- increment(&num);: เมื่อเรียกใช้ เราส่ง **ที่อยู่** ของ num เข้าไป
- (*valuePtr)++;: ภายในฟังก์ชัน เราใช้ *valuePtr เพื่อเข้าถึงค่าจริงของ num และแก้ไขมัน ซึ่งจะส่งผลกระทบต่อ num ที่อยู่ใน main ด้วย

4.2 พอยน์เตอร์กับอาเรย์ในฟังก์ชัน

เมื่อส่งอาเรย์เข้าไปในฟังก์ชัน จริงๆ แล้ว C++ จะทำการ **ส่งที่อยู่ของสมาชิกตัวแรก** ของอาเรย์นั้นไป (decay to pointer) ซึ่งหมายความว่าเรากำลังทำงานกับพอยน์เตอร์

C++

```
#include <iostream>

// ฟังก์ชันสำหรับแสดงผลสมาชิกของอาเรย์ โดยรับพอยน์เตอร์ (ชื่ออาเรย์) และขนาด
void printArray(int *arr, int size) {
    std::cout << "  ในฟังก์ชัน printArray:" << std::endl;
    std::cout << "  ที่อยู่ของ arr ที่รับมา: " << arr << std::endl; // ที่อยู่เดียวกับ numbers ภายนอก
    std::cout << "  ขนาดของ arr ในฟังก์ชัน (เป็นขนาดพอยน์เตอร์): " << sizeof(arr) << " ไบต์" <<
std::endl;
    // หมายเหตุ: sizeof(arr) ในฟังก์ชันจะไม่ให้ขนาดอาเรย์จริง แต่เป็นขนาดของพอยน์เตอร์ (เช่น 8
ไบต์บน 64-bit system)

    for (int i = 0; i < size; ++i) {
        std::cout << "  Element " << i << ": " << arr[i] << std::endl; // ใช้ arr[i] ได้เหมือนอาเรย์ปกติ
        // หรือ std::cout << "  Element " << i << ": " << *(arr + i) << std::endl;
    }
}

// ฟังก์ชันสำหรับหาผลรวมของสมาชิกในอาเรย์
int sumArray(int *arr, int size) {
    int total = 0;
    for (int i = 0; i < size; ++i) {
        total += arr[i];
    }
    return total;
}
```

```
int main() {
    int data[] = {1, 2, 3, 4, 5};
    int dataSize = sizeof(data) / sizeof(data[0]);

    std::cout << "--- พอยน์เตอร์กับอาเรย์ในฟังก์ชัน ---" << std::endl;
    std::cout << "ใน main: ที่อยู่ของ data: " << data << std::endl;

    printArray(data, dataSize); // ส่งชื่ออาเรย์ (ซึ่งคือพอยน์เตอร์) และขนาด

    int arraySum = sumArray(data, dataSize);
    std::cout << "ผลรวมของสมาชิกในอาเรย์: " << arraySum << std::endl;

    return 0;
}
```

ผลลัพธ์ (ตัวอย่าง):

```
--- พอยน์เตอร์กับอาเรย์ในฟังก์ชัน ---
ใน main: ที่อยู่ของ data: 0x7ffe427b5e50
  ในฟังก์ชัน printArray:
  ที่อยู่ของ arr ที่รับมา: 0x7ffe427b5e50
  ขนาดของ arr ในฟังก์ชัน (เป็นขนาดพอยน์เตอร์): 8 ไบต์
  Element 0: 1
  Element 1: 2
  Element 2: 3
  Element 3: 4
  Element 4: 5
ผลรวมของสมาชิกในอาเรย์: 15
```

คำอธิบาย:

- เมื่อคุณส่ง data (ชื่ออาเรย์) ไปยัง printArray จริงๆ แล้วคุณกำลังส่ง &data[0] (ที่อยู่ของสมาชิกตัวแรก) ไป
- ดังนั้น พารามิเตอร์ int *arr ก็คือพอยน์เตอร์ที่รับที่อยู่นั้นมา
- แม้จะรับเป็น int *arr แต่คุณยังสามารถใช้สัญลักษณ์ arr[i] เพื่อเข้าถึงสมาชิกได้ตามปกติ ซึ่งคอมไพเลอร์จะแปลเป็น *(arr + i) โดยอัตโนมัติ

ข้อควรระวังในการใช้พอยน์เตอร์:

- **Null Pointer Dereference:** การพยายามเข้าถึงค่าจากพอยน์เตอร์ที่เป็น nullptr (ชี้ไปที่ 0x0) จะทำให้โปรแกรม crash ได้ ควรตรวจสอบเสมอว่าพอยน์เตอร์ไม่เป็น nullptr ก่อนใช้งาน
- **Dangling Pointer:** พอยน์เตอร์ที่ชี้ไปยังหน่วยความจำที่ถูกคืนไปแล้ว (deallocated) การพยายามเข้าถึงหน่วยความจำนั้นเป็นอันตราย
- **Memory Leaks:** การจองหน่วยความจำแบบพลวัตด้วย new แล้วลืม delete จะทำให้หน่วยความจำนั้นไม่ถูกคืนกลับสู่ระบบ ซึ่งนำไปสู่การใช้หน่วยความจำเกินความจำเป็น
- **Wild Pointers:** พอยน์เตอร์ที่ยังไม่ได้ถูกกำหนดค่าให้ชี้ไปที่ไหน การใช้งานพอยน์เตอร์เหล่านี้เป็นอันตรายอย่างยิ่ง

พอยน์เตอร์เป็นหัวข้อที่ต้องฝึกฝนทำความเข้าใจและระมัดระวังในการใช้งาน แต่เมื่อเชี่ยวชาญแล้ว มันจะช่วยให้คุณควบคุมการทำงานของโปรแกรมในระดับต่ำได้อย่างมีประสิทธิภาพมากครับ

คุณมีคำถามเพิ่มเติมเกี่ยวกับพอยน์เตอร์ หรืออยากให้ผมแสดงตัวอย่างในสถานการณ์อื่นอีกไหมครับ? แน่นอนครับ! ต่อไปนี้คือรายละเอียดเพิ่มเติมแบบครบถ้วน เข้าใจง่าย พร้อม **ภาพจำทางความคิด + ตัวอย่าง + คำอธิบายลึกขึ้น** สำหรับหัวข้อ:

พอยน์เตอร์ (Pointers) ใน C++ (Dev-C++) เพิ่มเติม

☐ 1. พอยน์เตอร์คืออะไร?

พอยน์เตอร์ (Pointer) คือ ตัวแปรชนิดพิเศษที่ "เก็บที่อยู่ (Address)" ของตัวแปรอื่น ไม่ใช่ค่าโดยตรง

☐ เปรียบเทียบให้เข้าใจง่าย:

ตัวแปรธรรมดา	พอยน์เตอร์
เก็บ "ค่า"	เก็บ "ที่อยู่ของตัวแปรอื่น"
เช่น int a = 10;	int *p = &a;

☐ ภาพจำ:

[a] ----> [10] (ตัวแปรทั่วไป)

[p] ----> [ที่อยู่ของ a] (พอยน์เตอร์)

☐ ตัวอย่างที่ 1: การประกาศและใช้งานพื้นฐาน

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

int main() {
    int a = 10;
```

```
    int* p = &a;

    cout << "ค่า a = " << a << endl;
    cout << "ที่อยู่ของ a (&a) = " << &a << endl;
    cout << "ค่าของ p (ที่อยู่ที่ p เก็บ) = " << p << endl;
    cout << "ค่าที่ *p ชี้ไปหา = " << *p << endl;

    return 0;
}
```

□ จุดสังเกต:

- int* p □ ประกาศ pointer สำหรับ int
- p = &a □ p ชี้ไปยังตัวแปร a
- *p □ เข้าถึงค่าที่อยู่ใน a (เพราะ p ชี้ไปที่ a)

□ 2. การใช้ & และ * อย่างลึกซึ้ง

สัญลักษณ์	ชื่อ	ความหมาย
&x	Address-of operator	คืนค่าที่อยู่ของตัวแปร
*p	Dereference operator	เข้าถึงค่าที่ pointer ชี้อยู่

□ ตัวอย่างที่ 2: ใช้เปลี่ยนค่าผ่าน pointer

```
int b = 5;
int* pb = &b;

*pb = 99; // เปลี่ยนค่าของ b ผ่าน pointer
```

□ ตอนนี้ b จะเปลี่ยนเป็น 99

□ 3. Pointer กับอาเรย์ (Array)

เมื่อคุณประกาศอาเรย์ เช่น:

```
int arr[3] = {10, 20, 30};
int* p = arr;  // p = &arr[0]
```

คุณสามารถใช้ pointer p เดินในอาเรย์ได้

□ ตัวอย่างที่ 3: เดินอาเรย์ด้วย pointer

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

```
int main() {
    int arr[3] = {10, 20, 30};
    int* p = arr;

    for (int i = 0; i < 3; i++) {
        cout << "ค่าที่ *(p + " << i << ") = " << *(p + i) << endl;
    }

    return 0;
}
```

□ ความเข้าใจ: *(p + i) เทียบเท่ากับ arr[i]

□ **4. Pointer กับฟังก์ชัน**

คุณสามารถส่งพอยน์เตอร์เข้าไปในฟังก์ชัน เพื่อ:

- รับค่าจากหลายตัวแปร (เหมือน return ได้หลายค่า)
- เปลี่ยนค่าจากฟังก์ชันหลัก (main)

□ ตัวอย่างที่ 4: ส่ง pointer เพื่อเปลี่ยนค่าจริง

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

void square(int* num) {
    *num = (*num) * (*num);
}

int main() {
    int x = 5;
    square(&x);
    cout << "x^2 = " << x << endl;

    return 0;
}
```

□ ผลลัพธ์: x^2 = 25

☐ **5. Pointer กับ Pointer (Double Pointer)**

```
int x = 10;
int* p = &x;
int** pp = &p;  // พอยน์เตอร์ของพอยน์เตอร์
```

☐ ใช้งาน:

```
cout << **pp;  // ได้ค่า x = 10
```

☐ ภาพจำ:

```
[pp] ---> [p] ---> [x] = 10
```

การประกาศและใช้ Pointer ใน C++

☐ **1. คำจำกัดความ**

Pointer คือ ตัวแปรชนิดพิเศษที่ **เก็บที่อยู่ (memory address)** ของตัวแปรอื่น

☐ **2. รูปแบบการประกาศ**

```
type* pointerName;
```

หรือ

```
type *pointerName;
```

☐ ทั้งสองรูปแบบใช้ได้เหมือนกัน (นิยมแบบแรกเพื่อความชัดเจน)

☐ **3. ตัวอย่างการประกาศ**

```
int a = 100;
int* ptr;      // ประกาศ pointer สำหรับเก็บที่อยู่ของตัวแปร int
ptr = &a;      // ให้ ptr ชี้ไปยังตัวแปร a
```

☐ &a คือ address ของตัวแปร a

☐ *ptr คือการเข้าถึงค่าที่ pointer ptr ชี้อยู่ (ในที่นี้คือ a)

☐ **ตัวอย่างโปรแกรม: การประกาศและใช้ Pointer**

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

int main() {
    int a = 100;
    int* p = &a;

    cout << "ค่า a = " << a << endl;
```

```
    cout << "ที่อยู่ของ a (&a) = " << &a << endl;
    cout << "ค่าของ p (ที่อยู่ที่มันชี้) = " << p << endl;
    cout << "ค่าที่อยู่ใน *p (ค่าที่ pointer ชี้ไปหา) = " << *p << endl;

    return 0;
}
```

☐ **ผลลัพธ์ที่ได้ (ตัวอย่าง):**

ค่า a = 100

ที่อยู่ของ a (&a) = 0x61ff08

ค่าของ p (ที่อยู่ที่มันชี้) = 0x61ff08

ค่าที่อยู่ใน *p (ค่าที่ pointer ชี้ไปหา) = 100

☐ **คำอธิบายเพิ่มเติม**

คำสั่ง	ความหมาย
int* p	ประกาศตัวแปร p เป็น pointer ที่เก็บ address ของ int
p = &a	ให้ p ชี้ไปยัง a (เก็บ address ของ a)
*p	ดึงค่าที่อยู่ใน address ที่ p ชี้อยู่
&a	หาที่อยู่ของตัวแปร a

☐ **สรุปสำคัญ:**

สัญลักษณ์	ความหมาย	ตัวอย่าง
*	ใช้ประกาศ pointer หรือเข้าถึงค่าที่ pointer ชี้อยู่	*p = 50;
&	ใช้ดึงที่อยู่ของตัวแปร	p = &a;

☐ **ภาพจำ (Conceptual Diagram)**

int a = 100;

int* p = &a;

[a] = 100 --> Address: 0x1234

[p] = 0x1234 --> ชี้ไปยังตัวแปร a

ข้อมูลเพิ่มเติมแบบเจาะลึก

☐ **“การประกาศและใช้ Pointer”**

ครอบคลุมแนวคิดพื้นฐาน ⟶ ขั้นกลาง ⟶ ข้อควรระวัง ⟶ แบบฝึกหัด พร้อม **ภาพอธิบายจำลอง**

☐ **1. โครงสร้างหน่วยความจำเบื้องต้น**

```
int a = 10;
int* p = &a;
```

☐ **หน่วยความจำภายในจะเป็นแบบนี้:**

ตัวแปร	ค่า	ที่อยู่ (สมมุติ)
a	10	0x1000
p	0x1000 (ที่อยู่ของ a)	0x2000

☐ **ความสัมพันธ์:**

- a = 10
- p = &a = 0x1000
- *p = 10

☐ **2. การใช้งานจริง: อ่านและเขียนค่าผ่าน pointer**

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

int main() {
    int a = 5;
    int* p = &a;

    cout << "ค่าเดิมของ a: " << a << endl;

    *p = 20;  // เปลี่ยนค่าของ a ผ่าน pointer

    cout << "ค่าของ a หลังใช้ pointer เปลี่ยน: " << a << endl;

    return 0;
}
```

☐ **ผลลัพธ์:**

ค่าเดิมของ a: 5

ค่าของ a หลังใช้ pointer เปลี่ยน: 20

☐ **3. การเปลี่ยนค่าตัวแปรในฟังก์ชัน**

☐ **แบบไม่ใช้ pointer (ค่าไม่เปลี่ยน)**

```
void modify(int x) {
    x = 999;
}
```

☐ **แบบใช้ pointer (ค่าเปลี่ยนจริง)**

```
void modify(int* x) {
    *x = 999;
}
```

☐ **ตัวอย่าง:**

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

void modify(int* ptr) {
    *ptr = 999;
}

int main() {
    int value = 10;
    modify(&value);
    cout << "ค่าหลัง modify: " << value << endl;  // ได้ 999
    return 0;
}
```

☐ **4. การอ่าน pointer ด้วย cout**

Expression	ความหมาย
p	แสดงที่อยู่ที่ pointer ชี้
&p	ที่อยู่ของตัว pointer เอง
*p	ค่าที่อยู่ในตำแหน่งที่ pointer ชี้

☐ 5. ข้อควรระวังในการใช้ Pointer

กรณี	ปัญหา
ลืมกำหนดค่าเริ่มต้นให้ pointer	อาจชี้ไปยังที่อยู่สุ่ม (Segmentation Fault)
ใช้ pointer หลัง delete	**Dangling Pointer**
ใช้ pointer แบบ null โดยไม่ได้ตรวจ	อาจ crash ได้

☐ วิธีป้องกัน:

```
int* p = nullptr;  // กำหนดค่าเริ่มต้นให้ pointer ปลอดภัย
```

☐ 7. ภาพประกอบแนวคิด

```
int x = 50;
int* p = &x;

Memory:
+----------+-----------+
| Address  | Value     |
+----------+-----------+
| 0x1000   | 50 (x)    |
| 0x2000   | 0x1000 (p)|
+----------+-----------+

*p => ไปที่ 0x1000 => ได้ค่า 50
```

☐ สรุปภาพรวม

คำสั่ง	ความหมาย
int* p;	ประกาศ pointer ที่เก็บที่อยู่ของ int
p = &a;	ให้ pointer ชี้ไปยังตัวแปร a
*p = 20;	เปลี่ยนค่าที่ pointer ชี้อยู่ให้เป็น 20
&a	หา address ของ a

แนวคิดสำคัญ:

- **ที่อยู่หน่วยความจำ (Address):** ตำแหน่งเฉพาะใน RAM ที่ข้อมูลถูกเก็บไว้
- **ค่า (Value):** ข้อมูลที่ถูกเก็บอยู่ที่ที่อยู่หน่วยความจำนั้น

- **& (Address-of Operator):** ใช้เพื่อ **หาที่อยู่** ของตัวแปรปกติ
- * **(Dereference Operator):** ใช้เพื่อ **เข้าถึงค่า** ที่อยู่ ณ ที่อยู่หน่วยความจำที่พอยน์เตอร์ชี้ไป

C++

```
#include <iostream> // สำหรับ std::cout และ std::endl
#include <string>   // สำหรับ std::string

int main() {
    std::cout << "===== การประกาศและการใช้งานพอยน์เตอร์เบื้องต้น =====" << std::endl;

    // 1. ประกาศตัวแปรปกติ (Value Variables)
    //    ตัวแปรเหล่านี้เก็บค่าโดยตรง
    int age = 25;
    double salary = 50000.75;
    std::string name = "Alice";

    std::cout << "\n--- ข้อมูลของตัวแปรปกติ ---" << std::endl;
    std::cout << "ชื่อตัวแปร: age" << std::endl;
    std::cout << "  ค่า: " << age << std::endl;                    // แสดงค่า 25
    std::cout << "  ที่อยู่ (&age): " << &age << std::endl;        // แสดงที่อยู่หน่วยความจำของ age

    std::cout << "ชื่อตัวแปร: salary" << std::endl;
    std::cout << "  ค่า: " << salary << std::endl;                 // แสดงค่า 50000.75
    std::cout << "  ที่อยู่ (&salary): " << &salary << std::endl; // แสดงที่อยู่หน่วยความจำของ salary

    std::cout << "ชื่อตัวแปร: name" << std::endl;
    std::cout << "  ค่า: " << name << std::endl;                   // แสดงค่า "Alice"
    std::cout << "  ที่อยู่ (&name): " << &name << std::endl;      // แสดงที่อยู่หน่วยความจำของ
name

    // 2. ประกาศพอยน์เตอร์ (Pointer Variables)
    //    พอยน์เตอร์จะเก็บที่อยู่หน่วยความจำของตัวแปรอื่น
    //    รูปแบบ: ชนิดข้อมูล_ที่_พอยน์เตอร์_ชี้_ไป * ชื่อพอยน์เตอร์;
    int *ptrAge;          // พอยน์เตอร์ที่ชี้ไปยัง int
    double *ptrSalary;    // พอยน์เตอร์ที่ชี้ไปยัง double
```

```
    std::string *ptrName; // พอยน์เตอร์ที่ชี้ไปยัง std::string

    std::cout << "\n--- พอยน์เตอร์เมื่อประกาศ (ยังไม่กำหนดค่า) ---" << std::endl;
    // พอยน์เตอร์ที่ยังไม่ถูกกำหนดค่าจะเก็บ "ค่าขยะ" (garbage value)
    // การพยายามเข้าถึงค่าที่พอยน์เตอร์เหล่านี้ชี้ไปเป็นอันตราย (undefined behavior)
    std::cout << "ptrAge (ยังไม่กำหนด): " << ptrAge << std::endl;       // อาจแสดงค่าขยะ
    std::cout << "ptrSalary (ยังไม่กำหนด): " << ptrSalary << std::endl; // อาจแสดงค่าขยะ
    std::cout << "ptrName (ยังไม่กำหนด): " << ptrName << std::endl;     // อาจแสดงค่าขยะ

    // 3. กำหนดค่าให้พอยน์เตอร์: ให้ชี้ไปยังที่อยู่ของตัวแปรปกติ
    //    ใช้โอเปอเรเตอร์ & (address-of) เพื่อรับที่อยู่ของตัวแปร
    ptrAge = &age;          // ptrAge ชี้ไปยังที่อยู่ของตัวแปร 'age'
    ptrSalary = &salary;    // ptrSalary ชี้ไปยังที่อยู่ของตัวแปร 'salary'
    ptrName = &name;        // ptrName ชี้ไปยังที่อยู่ของตัวแปร 'name'

    std::cout << "\n--- พอยน์เตอร์หลังกำหนดค่า (ชี้ไปที่ตัวแปรปกติ) ---" << std::endl;
    std::cout << "ptrAge เก็บที่อยู่: " << ptrAge << std::endl;       // จะแสดงที่อยู่เดียวกับ &age
    std::cout << "ptrSalary เก็บที่อยู่: " << ptrSalary << std::endl; // จะแสดงที่อยู่เดียวกับ &salary
    std::cout << "ptrName เก็บที่อยู่: " << ptrName << std::endl;     // จะแสดงที่อยู่เดียวกับ &name

    // 4. เข้าถึงค่าที่พอยน์เตอร์ชี้ไป: ใช้โอเปอเรเตอร์ * (dereference)
    std::cout << "\n--- การเข้าถึงค่าผ่านพอยน์เตอร์ด้วย * ---" << std::endl;
    std::cout << "ค่าที่ ptrAge ชี้ไป (*ptrAge): " << *ptrAge << std::endl;       // จะได้ค่า 25
    std::cout << "ค่าที่ ptrSalary ชี้ไป (*ptrSalary): " << *ptrSalary << std::endl; // จะได้ค่า
50000.75
    std::cout << "ค่าที่ ptrName ชี้ไป (*ptrName): " << *ptrName << std::endl;     // จะได้ค่า "Alice"

    // 5. เปลี่ยนค่าของตัวแปรผ่านพอยน์เตอร์
    //    เราสามารถใช้ * เพื่อเข้าถึงค่าและทำการแก้ไขได้โดยตรง
    *ptrAge = 26;           // เปลี่ยนค่าของ age เป็น 26 ผ่าน ptrAge
    *ptrSalary += 10000;    // เพิ่มค่า salary อีก 10000 ผ่าน ptrSalary
    *ptrName = "Bob";       // เปลี่ยนค่า name เป็น "Bob" ผ่าน ptrName

    std::cout << "\n--- ค่าของตัวแปรปกติหลังการแก้ไขผ่านพอยน์เตอร์ ---" << std::endl;
```

```
    std::cout << "age (เปลี่ยนผ่าน *ptrAge): " << age << std::endl;              // แสดง 26
    std::cout << "salary (เปลี่ยนผ่าน *ptrSalary): " << salary << std::endl;     // แสดง 60000.75
    std::cout << "name (เปลี่ยนผ่าน *ptrName): " << name << std::endl;           // แสดง "Bob"

    std::cout << "\n--- ตรวจสอบค่าผ่านพอยน์เตอร์อีกครั้ง ---" << std::endl;
    std::cout << "ค่าที่ *ptrAge ชี้ไป: " << *ptrAge << std::endl;       // แสดง 26
    std::cout << "ค่าที่ *ptrSalary ชี้ไป: " << *ptrSalary << std::endl; // แสดง 60000.75
    std::cout << "ค่าที่ *ptrName ชี้ไป: " << *ptrName << std::endl;     // แสดง "Bob"

    // 6. กำหนดให้พอยน์เตอร์ไม่ชี้ไปที่ใด (nullptr/NULL)
    //    เป็นแนวทางปฏิบัติที่ดีเพื่อป้องกันการใช้พอยน์เตอร์ที่ชี้ไปที่อยู่ที่ไม่ถูกต้อง
    ptrAge = nullptr; // ใช้ nullptr ใน C++11 ขึ้นไป
    // หรือ ptrAge = NULL; // สำหรับ C++ รุ่นเก่า

    std::cout << "\n--- พอยน์เตอร์ที่กำหนดเป็น nullptr ---" << std::endl;
    std::cout << "ptrAge (หลังเป็น nullptr): " << ptrAge << std::endl; // มักแสดง 0x0 หรือ 0

    // การพยายาม dereference พอยน์เตอร์ที่เป็น nullptr จะทำให้โปรแกรม crash
    // if (ptrAge != nullptr) {
    //     std::cout << *ptrAge << std::endl; // บรรทัดนี้จะทำงานได้ถ้า ptrAge ไม่ใช่ nullptr
    // }

    return 0; // จบการทำงาน
}
```

ผลลัพธ์ (Output):

(ที่อยู่หน่วยความจำ 0x... จะแตกต่างกันไปในแต่ละครั้งที่รันโปรแกรม)

```
===== การประกาศและการใช้งานพอยน์เตอร์เบื้องต้น =====

--- ข้อมูลของตัวแปรปกติ ---
ชื่อตัวแปร: age
  ค่า: 25
  ที่อยู่ (&age): 0x7ffd1e83f3e4
ชื่อตัวแปร: salary
  ค่า: 50000.75
```

```
  ที่อยู่ (&salary): 0x7ffd1e83f3e8
ชื่อตัวแคร: name
  ค่า: Alice
  ที่อยู่ (&name): 0x7ffd1e83f3f0

--- พอยน์เตอร์เมื่อประกาศ (ยังไม่กำหนดค่า) ---
ptrAge (ยังไม่กำหนด): 0x7ffd1e83f3c4
ptrSalary (ยังไม่กำหนด): 0x7ffd1e83f3c8
ptrName (ยังไม่กำหนด): 0x7ffd1e83f3d0

--- พอยน์เตอร์หลังกำหนดค่า (ชี้ไปที่ตัวแปรปกติ) ---
ptrAge เก็บที่อยู่: 0x7ffd1e83f3e4
ptrSalary เก็บที่อยู่: 0x7ffd1e83f3e8
ptrName เก็บที่อยู่: 0x7ffd1e83f3f0

--- การเข้าถึงค่าผ่านพอยน์เตอร์ด้วย * ---
ค่าที่ ptrAge ชี้ไป (*ptrAge): 25
ค่าที่ ptrSalary ชี้ไป (*ptrSalary): 50000.75
ค่าที่ ptrName ชี้ไป (*ptrName): Alice

--- ค่าของตัวแปรปกติหลังการแก้ไขผ่านพอยน์เตอร์ ---
age (เปลี่ยนผ่าน *ptrAge): 26
salary (เปลี่ยนผ่าน *ptrSalary): 60000.75
name (เปลี่ยนผ่าน *ptrName): Bob

--- ตรวจสอบค่าผ่านพอยน์เตอร์อีกครั้ง ---
ค่าที่ *ptrAge ชี้ไป: 26
ค่าที่ *ptrSalary ชี้ไป: 60000.75
ค่าที่ *ptrName ชี้ไป: Bob

--- พอยน์เตอร์ที่กำหนดเป็น nullptr ---
ptrAge (หลังเป็น nullptr): 0x0
```

สรุปขั้นตอนและหลักการ:

1. **ประกาศตัวแปรปกติ:** สร้างตัวแปรที่เก็บค่าข้อมูลโดยตรง

   ```
   int age = 25;
   ```

2. **ประกาศพอยน์เตอร์:** ระบุชนิดข้อมูลที่พอยน์เตอร์จะชี้ไป ตามด้วยเครื่องหมายดอกจัน * และชื่อพอยน์เตอร์

   ```
   int *ptrAge; // ptrAge สามารถเก็บที่อยู่ของตัวแปร int ได้
   ```

3. **กำหนดค่าให้พอยน์เตอร์:** ใช้โอเปอเรเตอร์ **& (address-of)** หน้าตัวแปรปกติ เพื่อ "รับที่อยู่" ของตัวแปรนั้นมาเก็บไว้ในพอยน์เตอร์

   ```
   ptrAge = &age; // ตอนนี้ ptrAge ชี้ไปที่ตัวแปร age
   ```

4. **เข้าถึงค่าผ่านพอยน์เตอร์:** ใช้โอเปอเรเตอร์ * **(dereference)** หน้าชื่อพอยน์เตอร์ เพื่อ "เข้าถึงค่า" ที่อยู่ในตำแหน่งหน่วยความจำที่พอยน์เตอร์นั้นชี้อยู่

   ```
   std::cout << *ptrAge; // จะแสดงค่า 25 (ค่าของ age)
   ```

5. **แก้ไขค่าผ่านพอยน์เตอร์:** ใช้โอเปอเรเตอร์ * เพื่อเข้าถึงค่า แล้วกำหนดค่าใหม่ให้

   ```
   *ptrAge = 26; // จะทำให้ค่าของ age เปลี่ยนเป็น 26
   ```

6. **nullptr (หรือ NULL):** ใช้เพื่อกำหนดให้พอยน์เตอร์ไม่ชี้ไปที่ใด ซึ่งเป็นสิ่งสำคัญในการป้องกันการเข้าถึงหน่วยความจำโดยไม่ตั้งใจ (null pointer dereference)

การทำความเข้าใจการประกาศและใช้พอยน์เตอร์ รวมถึงความแตกต่างระหว่าง & และ * เป็นสิ่งสำคัญมากในการเรียนรู้ C++ ครับ หากมีข้อสงสัยเพิ่มเติม ถามได้เลยนะครับ!

แน่นอนครับ! หัวข้อนี้คือ **หัวใจของการเข้าใจ Pointer ในภาษา C++**

เราจะเจาะลึกคำสั่ง & และ * อย่างชัดเจน พร้อมตัวอย่างและคำอธิบายภาพจำแบบเข้าใจง่าย □

การใช้ & (address-of) และ * (dereference) ใน C++

□ **& (Address-of Operator)**

ใช้เพื่อ **ดึง “ที่อยู่ในหน่วยความจำ” ของตัวแปร**

□ **รูปแบบ:**

```
int a = 10;
int* p = &a;
```

คำสั่ง	ความหมาย
&a	ที่อยู่ของตัวแปร a
p = &a	ให้ pointer p ชี้ไปยังตัวแปร a

□ *** (Dereference Operator)**

ใช้เพื่อ **เข้าถึงค่าที่อยู่ใน address ที่ pointer ชี้ไปหา**