C++程式設計：指標

**目錄**

[C++程式設計：指標 1](#_Toc204871931)

[1. 基本概念 2](#_Toc204871932)

[1.1 範例：印出地址 2](#_Toc204871933)

[1.2 範例：印出指標值與地址 3](#_Toc204871934)

[2. 宣告指標 3](#_Toc204871935)

[2.1 範例：查看不同型態與指標大小 3](#_Toc204871936)

[3. Integer Pointers(整數指標) 4](#_Toc204871937)

[3.1 範例：call by value（傳值，無法交換） 4](#_Toc204871938)

[3.2 範例：Call by Pointer（透過指標交換） 5](#_Toc204871939)

[4. 元素指標（Element Pointer） 5](#_Toc204871940)

[5. 陣列位移 6](#_Toc204871941)

[5.1 範例：存取陣列元素的位址 6](#_Toc204871942)

[5.2 範例：存取陣列元素的值 7](#_Toc204871943)

[6. 陣列與指標作為函數參數做傳遞 8](#_Toc204871944)

[6.1 範例：將陣列作為函數參數傳遞 8](#_Toc204871945)

[6.2 範例：將陣列作為函數參數傳遞(較正式版) 10](#_Toc204871946)

[7. 我請GPT幫我生成練習的題目： 11](#_Toc204871947)

[7.1 範例01：Memory Address Inspector 11](#_Toc204871948)

[7.2 範例02：Pointer Addition 11](#_Toc204871949)

[7.3 範例03：Pointer vs Array Access 11](#_Toc204871950)

[7.4 範例04：Array as Function Argument 11](#_Toc204871951)

[7.5 範例05：Pointer to Max 12](#_Toc204871952)

[7.6 範例06：Pointer Search 12](#_Toc204871953)

[8. 陣列指標（Array Pointer） 12](#_Toc204871954)

[8.1 範例：區分【指向元素】跟【指向整個陣列】 13](#_Toc204871955)

[8.2 範例：區分【指向元素】跟【指向整個陣列】 13](#_Toc204871956)

[8.3 範例：將 Array Pointer 傳入函式 14](#_Toc204871957)

[9. 二維陣列 15](#_Toc204871958)

[9.1 操作方式 01：用 int\* 線性存取整個二維陣列 15](#_Toc204871959)

[9.2 操作方式 02：使用指標運算存取二維元素 16](#_Toc204871960)

[9.3 操作方式 03：搭配for迴圈使用指標運算存取二維元素 17](#_Toc204871961)

[9.4 操作方式 04：透過陣列指標的機制操作 17](#_Toc204871962)

[10. 我請GPT幫我生成練習的題目： 18](#_Toc204871963)

[10.1 範例01：Flatten 2D Array 18](#_Toc204871964)

[10.2 範例02：Address Shift 18](#_Toc204871965)

[10.3 範例03：Access by Double Dereference 18](#_Toc204871966)

[10.4 範例04：Access by Double Dereference 19](#_Toc204871967)

[10.5 範例05：Pointer to Row 19](#_Toc204871968)

[11. 函數指標（Function Pointer） 19](#_Toc204871969)

[11.1 範例：基本函數指標範例 20](#_Toc204871970)

[11.2 範例：函數回傳指標（如回傳 int\\*） 20](#_Toc204871971)

[11.3 範例：函數指標陣列（Function Pointer Array） 21](#_Toc204871972)

[11.4 範例01：Callback 機制應用（傳入函數當作參數） 21](#_Toc204871973)

[11.5 範例02：Callback 機制應用（傳入函數當作參數） 22](#_Toc204871974)

[12. 雙指標(Double Pointer) 23](#_Toc204871975)

[12.1 範例：基本雙指標操作 23](#_Toc204871976)

[12.2 範例：傳遞字串陣列給函數（使用 char\\*[]） 24](#_Toc204871977)

[12.3 範例：使用雙指標動態分配二維陣列 25](#_Toc204871978)

[13. 結構指標(Structure Pointer) 26](#_Toc204871979)

[14. 空指標(Null Pointer) 26](#_Toc204871980)

[14.1 範例：檢查 NULL 函數指標 27](#_Toc204871981)

[14.2 範例：含數指標 + 檢查 NULL 函數指標 27](#_Toc204871982)

[14.3 範例：檢查動態記憶體配置是否成功 28](#_Toc204871983)

[14.4 範例：在資料結構中表示結尾（空鏈結串列） 28](#_Toc204871984)

[14.5 比較：NULL Pointer vs Void Pointer 29](#_Toc204871985)

[15. 來做個指標優缺點的小總結吧 29](#_Toc204871986)

[16. 智慧指標（Smart Pointers） 29](#_Toc204871987)

[16.1 shared\_ptr（共享指標） 29](#_Toc204871988)

[16.2 unique\_ptr（唯一指標） 30](#_Toc204871989)

# 基本概念

指標（Pointer）是一種變數，用來儲存另一個變數的記憶體位址。我們先來複習複習。當我們跟記憶體宣告一個變數後，會有以下三個東西可以使用：

1. 空間(存取變數)
2. 變數名稱(例如 成功大學)
3. 空間地址(例如 台南市東區大學路1號)

我們會有兩個常用的運算子：

* 解參考運算子（\*）：用來宣告指標變數，或用來存取某個位址所儲存的值。
* 位址運算子（&）用來取得變數的位址，或把某變數的位址傳給指標使用。

## 範例：印出地址

#include <iostream>

using namespace std;

int main() {

// int\* ptr 宣告一個指標

// &m：取得變數m的地址

    int m = 100;

    int\* ptr = &m;

    cout << "Value of Variable m is: " << m << endl; // 100

    cout << "Memory Address of Variable m is: " << &m << endl; // 0x7ffffcbf4

    cout << "Memory Address of Variable m is using ptr: " << ptr << endl; // 0x7ffffcbf4

    return 0;

}

## 範例：印出指標值與地址

接著，我們示範將int\* ptr = &m;拆成兩步：

#include <iostream>

using namespace std;

int main() {

    int a = 10;

    int\* ptr;

    ptr = &a;

    cout << "ptr = " << ptr << endl; // 0x7ffffcbf4

    cout << "\*ptr = " << \*ptr << endl; // 10

    return 0;

}

# 宣告指標

當我們宣告完指標後，是無法更改指標所指向的data type。宣告指標方式如下：

int \*x;

int \*x, \*y;

int \*y=NULL;

float \*fptr;

double \*dptr;

## 範例：查看不同型態與指標大小

這個範例示範不同data type類型的指標

#include <iostream>

using namespace std;

int main() {

    int a1 = 40, a2 = 50, a3 = 60;

    int\* p1 = &a1;

    int\* p2 = &a2;

    int\* p3 = &a3;

    cout << p1 << "\t" << \*p1 << endl; // 0x7ffffcbe4     40

    cout << p2 << "\t" << \*p2 << endl; // 0x7ffffcbe0     50

    cout << p3 << "\t" << \*p3 << endl; // 0x7ffffcbdc     60

    // 查看不同型態的指標大小

    int\* iptr;

    float\* fptr;

    double\* dptr;

    cout << "sizeof(iptr) = " << sizeof(iptr) << endl; // sizeof(iptr) = 8

    cout << "sizeof(fptr) = " << sizeof(fptr) << endl; // sizeof(fptr) = 8

    cout << "sizeof(dptr) = " << sizeof(dptr) << endl; // sizeof(dptr) = 8

    return 0;

}

// 雖然他們是不同值，但因為指標型態都是int，因此指標的資料型態都是int -> 8

# Integer Pointers(整數指標)

## 範例：call by value（傳值，無法交換）

#include <iostream>

using namespace std;

void swap(int a, int b) {

    int temp = a;

    a = b;

    b = temp;

}

int main() {

    int i = 1;

    int j = 2;

    swap(i, j);

    cout << "i = " << i << ", j = " << j << endl; // i = 1, j = 2

    return 0;

}

## 範例：Call by Pointer（透過指標交換）

#include <iostream>

using namespace std;

void swap(int\* a, int\* b) {

    int temp = \*a;

    \*a = \*b;

    \*b = temp;

}

int main() {

    int i = 1;

    int j = 2;

    swap(&i, &j);

    cout << "i = " << i << ", j = " << j << endl; // i = 2, j = 1

    return 0;

}

# 元素指標（Element Pointer）

元素指標（Element Pointer）是指陣列第0個元素的地址(也就是整個陣列的統稱地址)

當你寫int \*ptr = arr;，代表 ptr 指向arr[0]，也就是陣列的第 0 個元素。

#include <iostream>

using namespace std;

int main() {

    int\* ptr; // 宣告整數指針

    int arr[5] = {1, 2, 3, 4, 5};

    // 正確寫法

    ptr = arr;  // ptr 指向 arr[0]

    cout << ptr << endl; // 0x7ffffcbe0

    // 正確寫法

    ptr = &arr[0];

    cout << ptr << endl; // 0x7ffffcbe0

    // 錯誤寫法

    // ptr = &arr;  // &arr是「指向整個陣列」的指標，而上面都是「指向單一 int 元素」的指標

    int (\*arrayPtr)[5] = &arr;  // 若你真的需要取得整個陣列的指標，則必須使用正確的型別

    cout << arrayPtr << endl; // 0x7ffffcbe0

    return 0;

}

這邊就可以來個小整理：

|  |  |
| --- | --- |
| int\* ptr = arr; | 指向 arr[0] 的指標（陣列退化） |
| int\* ptr = &arr[0]; | 明確取得第一個元素的指標 |
| int (\*arrayPtr)[5] = &arr; | 指向整個陣列的指標 |

至於我們在範例中，會發現說，其實三者輸出的值都是會一樣。但它們的型別和語意是不一樣的。後面我們會在位移運算中再來討論一次~

# 陣列位移

這邊我們將【取值/地址】跟【陣列位移】整理在一起做討論。

## 範例：存取陣列元素的位址

我們宣告int\* ptr;

* ptr + i代表陣列第i 個元素的位址。
* &ptr[i]則是一種語法糖(這邊可以理解為ptr[i]是陣列取元素，那&ptr[i]就是陣列取元素取地址)

#include <iostream>

using namespace std;

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    int\* ptr = arr; // 指向 arr[0]

    // int\* ptr = &arr[0]; // 也可以

    // 指標運算

    cout << "Pointer arithmetic (ptr + i):\n";

    for (int i = 0; i < 5; ++i) {

        cout << (ptr + i) << '\n';

    }

    // 語法糖：&ptr[i]

    cout << "\nAddress using &ptr[i]:\n";

    for (int i = 0; i < 5; ++i) {

        cout << &ptr[i] << '\n';

    }

    return 0;

}

// Pointer arithmetic (ptr + i):

// 0x7ffffcbd0

// 0x7ffffcbd4

// 0x7ffffcbd8

// 0x7ffffcbdc

// 0x7ffffcbe0

// Address using &ptr[i]:

// 0x7ffffcbd0

// 0x7ffffcbd4

// 0x7ffffcbd8

// 0x7ffffcbdc

// 0x7ffffcbe0

## 範例：存取陣列元素的值

我們宣告int\* ptr;

* \*(ptr + i)代表陣列第i 個元素的位址的值。
* ptr[i] 則是一種語法糖(這邊可以理解成，ptr就是陣列，ptr[i]就是陣列取值)

#include <iostream>

using namespace std;

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    int\* ptr = arr;

    // int\* ptr = &arr[0]; // 也可以

    // 指標算術

    cout << "Values via \*(ptr + i): ";

    for (int i = 0; i < 5; ++i) {

        cout << \*(ptr + i) << " "; // 1 2 3 4 5

    }

    cout << "\n";

    // 語法糖 ptr[i]

    cout << "Values via ptr[i]:     ";

    for (int i = 0; i < 5; ++i) {

        cout << ptr[i] << " "; // 1 2 3 4 5

    }

    cout << "\n";

    return 0;

}

# 陣列與指標作為函數參數做傳遞

## 範例：將陣列作為函數參數傳遞

#include <iostream>

using namespace std;

// 這邊都是用語法糖

void f1(int arr[]) {

    // 位移運算

    for (int i = 0; i < 5; ++i) {

        cout << arr+i << " "; // 取地址

    }

    cout << "\n";

    for (int i = 0; i < 5; ++i) {

        cout << \*(arr+i) << " "; // 取值

    }

    cout << "\n";

    // 語法糖

    for (int i = 0; i < 5; ++i) {

        cout << arr[i] << " "; // 取值

    }

    cout << "\n";

    for (int i = 0; i < 5; ++i) {

        cout << &arr[i] << " "; // 取地址

    }

    cout << "\n";

}

void f2(int\* arr) {

    // 位移運算

    for (int i = 0; i < 5; ++i) {

        cout << arr + i << " "; // 取地址

    }

    cout << "\n";

    for (int i = 0; i < 5; ++i) {

        cout << \*(arr + i) << " "; // 取值

    }

    cout << "\n";

    // 語法糖

    for (int i = 0; i < 5; ++i) {

        cout << arr[i] << " "; // 取值

    }

    cout << "\n";

    for (int i = 0; i < 5; ++i) {

        cout << &arr[i] << " "; // 取地址

    }

    cout << "\n";

}

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    f1(arr);

    cout << "========================\n";

    f2(arr);

    return 0;

}

## 範例：將陣列作為函數參數傳遞(較正式版)

我們有提過，通常傳陣列會傳兩格東西：

1. 陣列名稱(同時也代表陣列本身地址，也代表陣列第0個元素的地址)。
2. 陣列長度

#include <iostream>

using namespace std;

// 使用陣列語法

void printArray01(int arr[], int size) {

    // 邏輯位移

    for (int i = 0; i < size; ++i) {

        cout << arr + i << " ";

    }

    cout << "\n";

    for (int i = 0; i < size; ++i) {

        cout << \*(arr + i) << " ";

    }

    cout << "\n";

    // 語法糖

    for (int i = 0; i < size; ++i) {

        cout << arr[i] << " ";

    }

    cout << "\n";

    for (int i = 0; i < size; ++i) {

        cout << &arr[i] << " ";

    }

    cout << "\n";

}

int main() {

    int arr[] = {1, 2, 3, 4, 5};

    printArray01(arr, 5); // 傳遞

    return 0;

}

# 我請GPT幫我生成練習的題目：

## 範例01：Memory Address Inspector

|  |  |
| --- | --- |
| 題目描述：  給定一個整數 x，請輸出該變數的值與記憶體位址（模擬輸出即可，實際位址用 placeholder 模擬）。 | |
| Input  單一整數 x（0 ≤ x ≤ 1000） | Output  輸出兩行：  第一行輸出 Value = x  第二行輸出 Address = 0xABCDEF01（用此值模擬地址即可） |
| Sample Input  123 | Sample Output  Value = 123  Address = 0xABCDEF01 |

## 範例02：Pointer Addition

|  |  |
| --- | --- |
| 題目描述：  輸入一個含 n 個整數的陣列，請用指標運算輸出每個元素的值（不可使用陣列語法）。 | |
| Input  第一行一個整數 n（1 ≤ n ≤ 100） 第二行 n 個整數 | Output  輸出一行，陣列元素值以空格分隔，使用指標加法存取 |
| Sample Input  5  1 2 3 4 5 | Sample Output  1 2 3 4 5 |

## 範例03：Pointer vs Array Access

|  |  |
| --- | --- |
| 題目描述：  請同時使用 ptr[i] 與 \*(ptr + i) 兩種方式，輸出同一陣列的所有值。 | |
| Input  一行輸入 5 個整數 | Output  輸出兩行： 第一行：使用 ptr[i] 輸出陣列內容 第二行：使用 \*(ptr + i) 輸出陣列內容 |
| Sample Input  10 20 30 40 50 | Sample Output  10 20 30 40 50  10 20 30 40 50 |

## 範例04：Array as Function Argument

|  |  |
| --- | --- |
| 題目描述：  設計一個函數 printArray(int\* arr, int size)，輸入陣列與長度後，請用三種方式輸出陣列內容：   * arr[i] * \*(arr + i) * &arr[i] | |
| Input  第一行整數 n（1 ≤ n ≤ 100） 第二行輸入 n 個整數 | Output  三行：依序使用上述三種方法輸出，空格分隔 |
| Sample Input  4  7 8 9 10 | Sample Output  7 8 9 10  7 8 9 10  0x1 0x2 0x3 0x4 |

## 範例05：Pointer to Max

|  |  |
| --- | --- |
| 題目描述：  請實作一個函數 int\* findMax(int\* arr, int n) 回傳指向最大值的指標，主程式輸出其值與位址。 | |
| Input  第一行一個整數 n（1 ≤ n ≤ 100） 第二行 n 個整數 | Output  兩行： 第一行最大值 第二行記憶體地址（模擬輸出） |
| Sample Input  5  3 9 2 7 1 | Sample Output  9  0x12345678 |

## 範例06：Pointer Search

|  |  |
| --- | --- |
| 題目描述：  請實作一個 int\* binarySearch(int\* arr, int size, int target) 回傳指向 target 的指標，若未找到則回傳 nullptr，並輸出其值或 Not Found。 | |
| Input  第一行一個整數 n（1 ≤ n ≤ 1000） 第二行 n 個整數 | Output  若找到 x，輸出 Found x at address 0xABCDEF01，否則輸出 Not Found |
| Sample Input  5 7  1 3 5 7 9 | Sample Output  Found 7 at address 0xABCDEF01 |

# 陣列指標（Array Pointer）

陣列指標是指向整個陣列的指標，不是指向單一元素。最常用於多維陣列工作與保留陣列大小資訊。

|  |  |
| --- | --- |
| **宣告方式** | **說明** |
| int \*p = arr; | 指向 arr[0]，第 0 個元素 |
| int (\*p)[5] = &arr; | 指向整個 arr[5] 陣列 |

這邊可以先提一下，在存取時：

(\*p) [i]：取指向陣列的第 i 個元素(就是arr[i] 的值)

\*p[i]：會先讀p[i]：從指標p，視為陣列p(就是陣列指標)，再去做解參考運算子(\*) ，❗️結果是未定義行為

## 範例：區分【指向元素】跟【指向整個陣列】

#include <iostream>

using namespace std;

int main() {

    int arr[5] = {10, 20, 30, 40, 50};

    int\* p1 = arr;         // 指向 arr[0]

    int (\*p2)[5] = &arr;   // 指向整個陣列 arr[5]

    // 存取元素

    cout << p1[2] << endl;        // 印出 30

    cout << (\*p2)[2] << endl;     // 印出 30

    // 指向資料的大小

    cout << "sizeof(\*p1) = " << sizeof(\*p1) << endl;  // 4（int）

    cout << "sizeof(\*p2) = " << sizeof(\*p2) << endl;  // 20（5 ints）

    // 指標本身的大小（通常是 8 bytes on 64-bit systems）

    cout << "sizeof(p1) = " << sizeof(p1) << endl; // 8

    cout << "sizeof(p2) = " << sizeof(p2) << endl; // 8

    return 0;

}

## 範例：區分【指向元素】跟【指向整個陣列】

#include <iostream>

using namespace std;

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    int\* p1;         // 指向單一整數（arr[0]）

    int (\*p2)[5];    // 指向整個陣列（arr）

    p1 = arr; // 也可以寫p1 = &arr[0];

    p2 = &arr;

    // 印出地址

    cout << "p1  = " << p1 << endl; // 0x7ffffcbd0

    cout << "p2  = " << p2 << endl; // 0x7ffffcbd0

    cout << "\*p2 = " << \*p2 << endl; // 0x7ffffcbd0

    cout << endl;

    // 印出值

    cout << "p1[0] = " << \*p1 << endl;       // 1

    cout << "p1[1] = " << \*(p1 + 1) << endl; // 2

    cout << "p1[0] = " << \*p2 << endl;       // 0x7ffffcbd0

    cout << "p1[0] = " << \*p2[0] << endl;    // 1

    // 向後移動

    p1++;  // 前進 1 個 int（4 bytes）

    p2++;  // 前進 1 個陣列（5 \* sizeof(int) = 20 bytes）

    // 印出地址

    cout << "p1  = " << p1 << endl; // 0x7ffffcbd0

    cout << "p2  = " << p2 << endl; // 0x7ffffcbd0

    cout << "\*p2 = " << \*p2 << endl; // 0x7ffffcbd0

    cout << endl;

    // 印出值

    cout << "p1[0] = " << \*p1 << endl;       // 1

    cout << "p1[1] = " << \*(p1 + 1) << endl; // 2

    cout << "p1[0] = " << \*p2 << endl;       // 0x7ffffcbd0

    cout << "p1[0] = " << \*p2[0] << endl;    // 7(未知)

    cout << "p1[0] = " << (\*p2)[0] << endl;  // 7(未知)

    return 0;

}

## 範例：將 Array Pointer 傳入函式

#include <iostream>

using namespace std;

// 接收一個指向含有 5 個 int 的陣列指標

void printArray(int (\*p)[5]) {

    for (int i = 0; i < 5; ++i) {

        cout << (\*p)[i] << " "; // 取陣列的值

        // cout << \*p[i] << " "; // 為定義行為

    }

    cout << endl;

}

int main() {

    int arr[5] = {1, 2, 3, 4, 5};

    printArray(&arr);  // 傳整個陣列的位址

    return 0;

}

# 二維陣列

接下來我們來討論二微陣列的範疇，首先要先了解二微陣列再記憶體中的擺放位置。

int arr[2][3] = {

    {1, 2, 3},

    {4, 5, 6}

};

// 其實在記憶體中是一條直線

// [1][2][3][4][5][6]

了解擺放位置後，我們就來看看，如何操作二微陣列

## 操作方式 01：用 int\* 線性存取整個二維陣列

#include <iostream>

using namespace std;

int main() {

    int arr[2][3] = {

        {1, 2, 3},

        {4, 5, 6}

    };

    int\* p = &arr[0][0];  // 指向第一個元素

    for (int i = 0; i < 6; ++i) {

        // 使用位移運算

        cout << \*(p + i) << " ";

        // 或使用語法糖：

        cout << p[i] << " ";

    }

    cout << endl;

    return 0;

}

## 操作方式 02：使用指標運算存取二維元素

#include <iostream>

using namespace std;

int main() {

    int num[2][4] = {

        {312, 16, 35, 65},

        {52, 111, 77, 80}

    };

    // 二維陣列本體即為指標（指向第一列）

    cout << "num         = " << num << endl; // 0x7ffffcbc0

    cout << "num + 1     = " << (num + 1) << endl; // 0x7ffffcbd0 (從0x7ffffcbc0後移16格byte)

    // 解參考(一層)取得列地址

    cout << "\*(num)      = " << \*(num) << endl; // 0x7ffffcbc0

    cout << "\*(num + 1)  = " << \*(num + 1) << endl; // 0x7ffffcbd0 (從0x7ffffcbc0後移16格byte)

    // 解參考(兩層)取得值

    cout << "\*(\*(num)+0)     = " << \*(\*(num + 0)+0) << endl; // 312 = num[0][0]

    cout << "\*(\*(num + 1)+0)  = " << \*(\*(num + 1)+0) << endl; // 52 = num[1][0]

    // 再示範一次解參考(兩層)取得值

    cout << "\*(num) + 2         = " << \*(num) + 1 << endl; // 0x7ffffcbe4

    cout << "\*(num + 1) + 2     = " << \*(num + 1) + 1 << endl; // 0x7ffffcbf4

    cout << "\*(\*(num) + 2)      = " << \*(\*(num + 0) + 1)  << endl;       // 16  = num[0][2]

    cout << "\*(\*(num + 1) + 2)  = " << \*(\*(num + 1) + 2) << endl;   // 77 = num[1][2]

    return 0;

}

## 操作方式 03：搭配for迴圈使用指標運算存取二維元素

#include <iostream>

using namespace std;

int main() {

    int num[2][4] = {

        {312, 16, 35, 65},

        {52, 111, 77, 80}

    };

    // 使用迴圈逐一印出所有元素

    size\_t m = sizeof(num) / sizeof(num[0]);       // 列數

    size\_t n = sizeof(num[0]) / sizeof(num[0][0]); // 行數

    for (size\_t i = 0; i < m; ++i) {

        for (size\_t j = 0; j < n; ++j) {

            cout << \*(\*(num + i) + j) << "\t"; // 或寫作 num[i][j]

        }

        cout << endl;

    }

    return 0;

}

## 操作方式 04：透過陣列指標的機制操作

#include <iostream>

using namespace std;

int main() {

    int arr[2][3] = {

        {1, 2, 3},

        {4, 5, 6}

    };

    int (\*p)[3] = arr; // p 是指向 3 個 int 陣列的指標

    for (int i = 0; i < 2; ++i) {

        for (int j = 0; j < 3; ++j) {

            cout << p[i][j] << " ";

        }

    }

    cout << endl;

    return 0;

}

# 我請GPT幫我生成練習的題目：

## 範例01：Flatten 2D Array

|  |  |
| --- | --- |
| 題目描述：  給定一個 2×3 的整數矩陣，請用 int\* 指標線性印出所有元素。 | |
| Input  一行輸入 6 個整數，依序填入 2×3 陣列。 | Output  一行輸出 6 個整數，使用 p[i] 或 \*(p + i) 印出 |
| Sample Input  1 2 3 4 5 6 | Sample Output  1 2 3 4 5 6 |

## 範例02：Address Shift

|  |  |
| --- | --- |
| 題目描述：  建立一個 2×4 的陣列，請輸出 num、num+1、\*(num)、\*(num+1) 四個位址（可模擬輸出）。 | |
| Input  不需輸入 | Output  輸出四行模擬地址（使用 0xABCDEF01 型式） |
| Sample Input | Sample Output |

## 範例03：Access by Double Dereference

|  |  |
| --- | --- |
| 題目描述：  請從一個 2×4 的陣列中，透過 \*(\*(arr+i)+j) 印出以下元素：   * arr[0][0] * arr[0][2] * arr[1][1] * arr[1][3] | |
| Input  一行 8 個整數，依照 row-major 填入 2×4 陣列 | Output  四行對應數值 |
| Sample Input | Sample Output |

## 範例04：Access by Double Dereference

|  |  |
| --- | --- |
| 題目描述：  請寫出一段程式碼，利用指標解參考操作與巢狀 for 迴圈，輸出 2×4 陣列的所有值。 | |
| Input  一行 8 個整數 | Output  輸出兩行，每行 4 整數 |
| Sample Input | Sample Output |

## 範例05：Pointer to Row

|  |  |
| --- | --- |
| 題目描述：  使用 int (\*p)[3] 指向 2×3 陣列，請用 p[i][j] 印出所有元素。 | |
| Input  一行輸入 6 個整數 | Output  兩行，每行 3 個整數 |
| Sample Input | Sample Output |

# 函數指標（Function Pointer）

函數指標可以儲存「函數的位址」。用於實現 callback（回呼）機制、行為抽象化，甚至模擬簡單 策略模式。

#include <iostream>

using namespace std;

// 建立函數

int add(int a, int b) {

    return a + b;

}

int main() {

    // 宣告函數指標：指向返回 int 且接受兩個 int 的函數

    int (\*fptr)(int, int);

    // 初始化函數指標

    fptr = &add;   // 可寫作 fptr = add;

    // 呼叫方式

    int result1 = (\*fptr)(3, 4); // 解參考方式

    int result2 = fptr(3, 4);    // 語法糖：直接呼叫方式

    cout << result1 << endl;

    cout << result2 << endl;

    return 0;

}

## 範例：基本函數指標範例

#include <iostream>

using namespace std;

int add(int a, int b) {

    return a + b;

}

int main() {

    int (\*fptr)(int, int);  // 宣告指向函數的指標

    fptr = add;             // 初始化（可省略 &）

    cout << "Result: " << fptr(5, 3) << endl;  // 輸出: 8

    return 0;

}

## 範例：函數回傳指標（如回傳 int\\*）

使用指標作為參數回傳：

#include <iostream>

using namespace std;

int\* firstPositive(int\* ptr) {

    while (\*ptr == 0) {

        ++ptr;

    }

    return ptr;

}

int main() {

    int array[] = {0, 1, 0, 5, 9, 0, 0, 6, 0, 2};

    int\* iptr = firstPositive(array);

    cout << "\*iptr = " << \*iptr << endl; // \*iptr = 1

    iptr = firstPositive(iptr);

    cout << "\*iptr = " << \*iptr << endl; // \*iptr = 1

    iptr = firstPositive(iptr + 1);

    cout << "\*iptr = " << \*iptr << endl; // \*iptr = 5

    return 0;

}

## 範例：函數指標陣列（Function Pointer Array）

#include <iostream>

using namespace std;

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

int sub(int a, int b) { return a - b; }

int mul(int a, int b) { return a \* b; }

int main() {

    // 宣告一個函數指標陣列

    int (\*farr[])(int, int) = {add, sub, mul};

    cout << "Add: " << farr[0](10, 5) << endl; // 15

    cout << "Sub: " << farr[1](10, 5) << endl; // 5

    cout << "Mul: " << farr[2](10, 5) << endl; // 50

    return 0;

}

## 範例01：Callback 機制應用（傳入函數當作參數）

Callback（回呼）在 C++ 中是一種「把函式當成參數」的做法，也就是你可以把某個函式的指標或函式物件，傳給另一個函式使用。這種技巧常用在處理事件、排序自訂邏輯、或想讓某個函式可以根據外部定義的邏輯來執行時。

Callback 就是「你定義好某個行為邏輯，把它交給別人來執行」，是一種控制反轉的技巧。

#include <iostream>

using namespace std;

void greet() {

    cout << "Hello from callback!" << endl;

}

// 接收 callback 的函式

void process(void (\*callback)()) {

    cout << "Start processing..." << endl;

    callback();  // 執行回呼

    cout << "Done!" << endl;

}

int main() {

    process(greet); // 將函式當成參數傳入

    return 0;

}

// Start processing...

// Hello from callback!

// Done!

## 範例02：Callback 機制應用（傳入函數當作參數）

#include <iostream>

using namespace std;

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

int subtract(int a, int b) { return a - b; }

// 接收一個函數指標作為參數

void calc(int a, int b, int (\*op)(int, int)) {

    cout << "Result: " << op(a, b) << endl;

}

int main() {

    calc(10, 5, add);      // Output: Result: 15

    calc(10, 5, subtract); // Output: Result: 5

    return 0;

}

// Result: 15

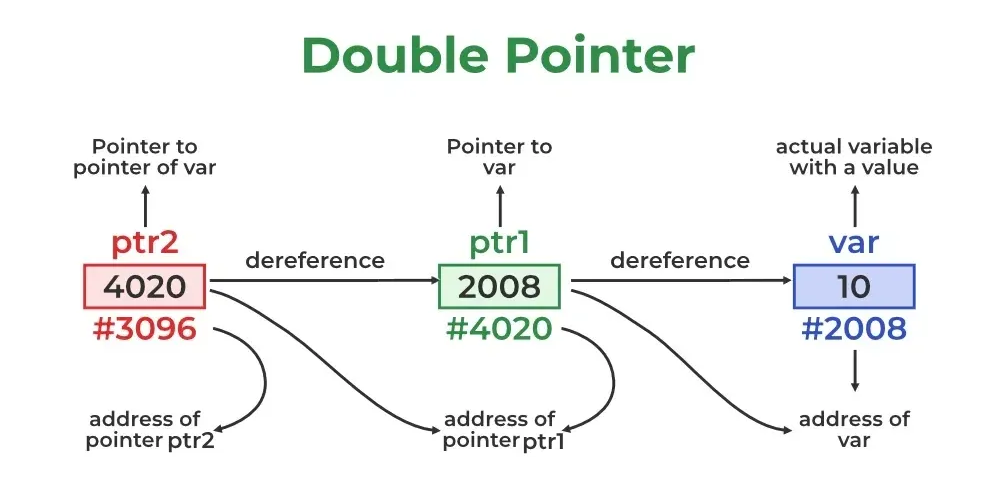
// Result: 5

# 雙指標(Double Pointer)

雙指標是指向「指標變數」的指標

在 C++ 中常用於：

* 修改指標本身的值（例如函式中動態配置記憶體）
* 表示動態配置的 二維陣列
* 字串陣列（例如 char\*\* argv）



// 宣告雙指針

int \*\*dptr; // 宣告一個指向 int\* 的雙指針

// 初始化雙指針

pointer\_name = &single\_pointer;

## 範例：基本雙指標操作

#include <iostream>

using namespace std;

int main() {

    int var = 10;

    int\* ptr = &var;

    int\*\* dptr = &ptr;

    cout << "變數 var 的值: " << var << endl; // 10

    cout << "\*ptr 指向的值: " << \*ptr << endl; // 10

    cout << "\*\*dptr 指向的值: " << \*\*dptr << endl; // 10

    cout << "ptr 的地址（dptr 指向的位置）: " << dptr << endl; // 0x7ffffcbe8

    cout << "ptr 的值（也就是 var 的位址）: " << ptr << endl; // 0x7ffffcbf4

    return 0;

}

## 範例：傳遞字串陣列給函數（使用 char\\*[]）

#include <iostream>

using namespace std;

// void printString(const char\* arr[], int n) {

// 用陣列指標去接陣列指標，沒毛病

void printStrings01(const char\* arr[], int n) {

    for (int i = 0; i < n; ++i) {

        cout << arr[i] << endl; // 語法糖

        cout << arr+i << endl; // 位移運算(地址)

        cout << \*(arr+i) << endl; // 位移運算

    }

}

// 用陣列指標去接陣列指標，沒毛病

void printStrings02(const char\*\* arr, int n) {

    for (int i = 0; i < n; ++i) {

        cout << arr[i] << endl; // 語法糖

        cout << arr+i << endl; // 位移運算(地址)

        cout << \*(arr+i) << endl; // 位移運算

    }

}

int main() {

    // 陣列指標

    const char\* arr[] = {"Geek", "Geeks", "Geekfor"};

    cout << arr << endl; // 0x7ffffcbe0

    printStrings01(arr, 3);

    printStrings02(arr, 3);

    return 0;

}

## 範例：使用雙指標動態分配二維陣列

#include <iostream>

using namespace std;

int main() {

    int m = 2, n = 3;

    // 分配 m 行

    int\*\* arr = new int\*[m];

    // 每行分配 n 個整數

    for (int i = 0; i < m; ++i) {

        arr[i] = new int[n];

    }

    // 初始化陣列

    for (int i = 0; i < m; ++i) {

        for (int j = 0; j < n; ++j) {

            arr[i][j] = i \* n + j + 1;

        }

    }

    // 輸出陣列

    for (int i = 0; i < m; ++i) {

        for (int j = 0; j < n; ++j) {

            cout << arr[i][j] << " "; // 1 2 3 4 5 6

        }

        cout << endl;

    }

    // 釋放記憶體

    for (int i = 0; i < m; ++i) {

        delete[] arr[i];

    }

    delete[] arr;

    return 0;

}

# 結構指標(Structure Pointer)

J個我們後續講結構(Structure)會在提到，不著急

先示範結構的簡單用法：

#include <iostream>

using namespace std;

struct A {

    int var;

};

int main() {

    A a = {30};   // C++ 中可省略 struct 關鍵字

    A\* ptr = &a;  // 指標指向 a

    cout << a.var << endl; // 30

    cout << (\*ptr).var << " ";  // Dot Operator（解參考再存取）

    cout << ptr->var << endl;   // Arrow Operator（推薦用法）

    return 0;

}

# 空指標(Null Pointer)

在 C++ 中，空指標（Null Pointer） 表示「不指向任何有效記憶體位置」的指標。

* C++11 之前使用 NULL（本質上是 0）
* C++11 之後建議使用 nullptr（專門為指標設計的常數，型別安全）

常見用途

* 初始化指標，避免未定義行為
* 判斷記憶體分配是否成功
* 作為資料結構的結尾標記（如 linked list）

int\* ptr = nullptr; // 建議用法（C++11+）

// 舊寫法（仍可用，但不推薦）

int\* ptr2 = NULL;

int\* ptr3 = 0;  // 可隱式轉為 nullptr

## 範例：檢查 NULL 函數指標

#include <iostream>

using namespace std;

int main() {

    int\* ptr = nullptr;

    if (ptr == nullptr) {

        cout << "Pointer is NULL (nullptr)" << endl;

    } else {

        cout << "Pointer is NOT NULL" << endl;

    }

    return 0;

}

## 範例：含數指標 + 檢查 NULL 函數指標

如果今天指標被宣告之後，沒有去指派，那他就會是一個NULL。(這邊會跟void去做比較)

#include <iostream>

using namespace std;

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

int main() {

    int (\*fptr)(int, int) = nullptr;

    if (fptr == nullptr) {

        cout << "Pointer is NULL" << endl;

        cout << fptr << endl; // 0

    }

    fptr = add;

    if (fptr != nullptr) {

        cout << "Result: " << fptr(5, 3) << endl;  // Output: 8

    }

    return 0;

}

// Pointer is NULL

// 0

// Result: 8

## 範例：檢查動態記憶體配置是否成功

#include <iostream>

using namespace std;

int main() {

    int\* ptr = new(nothrow) int[5];  // 配置 5 個 int 空間，使用 nothrow 以防錯

    if (ptr == nullptr) {

        cout << "Memory Allocation Failed" << endl;

    } else {

        cout << "Memory Allocation Succeeded" << endl;

        delete[] ptr;  // 釋放記憶體

    }

    return 0;

}

## 範例：在資料結構中表示結尾（空鏈結串列）

#include <iostream>

using namespace std;

struct Node {

    int data;

    Node\* next;

};

int main() {

    Node\* head = nullptr;

    if (head == nullptr) {

        cout << "Linked list is empty" << endl;

    }

    return 0;

}

## 比較：NULL Pointer vs Void Pointer

|  |  |  |
| --- | --- | --- |
|  | Null Pointer（`nullptr`） | Void Pointer（`void\*`） |
| 範例 | `int\* p = nullptr;` | `void\* p = &someVar;` |
| 定義 | 不指向任何有效位址的特殊值 | 一種「通用指標型別」，可指向任意型別資料 |
| 類型 | 所有指標型別皆可設為 `nullptr` | 僅能宣告為 `void\*` |
| 比較性值 | 所有空指標皆等於 `nullptr` | 不同 `void\*` 值不一定相等 |
| 轉換限制 | 不可直接對 `nullptr` 做算術運算 | void\*` 需轉型才能存取實值 |

# 來做個指標優缺點的小總結吧

|  |  |
| --- | --- |
| 優點 | 缺點 |
| 可有效存取與操作記憶體，適用於動態記憶體管理 | 記憶體洩漏：配置記憶體後未釋放，導致系統無法回收資源 |
| 可用於構建靈活的資料結構（如 Linked List、Tree、Graph） | 未初始化指標：指標未設為 `nullptr`，就操作會造成 \*\*Segmentation Fault\*\* |
| 提高函式傳遞彈性（如傳入大型陣列的位址） | 指標算術錯誤：越界、錯誤的地址計算可能導致未定義行為 |
| 可減少複製，提高效能與彈性 | 多重 `delete`：重複釋放記憶體會導致不穩定行為與程式崩潰 |

# 智慧指標（Smart Pointers）

在傳統 C++ 中，我們常常這樣寫：

int\* p = new int(10);

// 做一些操作...

delete p;  // 要記得手動釋放

但這樣很容易忘記 delete，導致 記憶體洩漏（memory leak）！智慧指標會自動幫你 delete，不用手動釋放記憶體。

## shared\_ptr（共享指標）

多個 shared\_ptr 可以指向同一個物件。當最後一個 shared\_ptr 被銷毀時，該物件會自動被釋放。

#include <iostream>

#include <memory> // 要包含這個

using namespace std;

class Dog {

public:

    Dog(string name) : name(name) {

        cout << "Dog " << name << " 被建立" << endl;

    }

    ~Dog() {

        cout << "Dog " << name << " 被銷毀" << endl;

    }

    void bark() {

        cout << name << "：汪！" << endl;

    }

private:

    string name;

};

int main() {

    shared\_ptr<Dog> p1 = make\_shared<Dog>("小黑");  // 建立智慧指標

    {

        shared\_ptr<Dog> p2 = p1;  // p2 也指向同一隻狗

        p2->bark();               // 呼叫 bark

        cout << "use\_count: " << p1.use\_count() << endl;  // 👉 顯示有幾個 shared\_ptr 指向該物件

    } // p2 被銷毀，但 p1 還活著

    cout << "use\_count: " << p1.use\_count() << endl;

} // p1 被銷毀，此時 Dog 也被銷毀

## unique\_ptr（唯一指標）

一個 unique\_ptr 只能由一個擁有者擁有。

不可以複製，只能轉移（使用 std::move）。

#include <iostream>

#include <memory>

using namespace std;

class Cat {

public:

    Cat(string name) : name(name) {

        cout << "Cat " << name << " 被建立" << endl;

    }

    ~Cat() {

        cout << "Cat " << name << " 被銷毀" << endl;

    }

    void meow() {

        cout << name << "：喵！" << endl;

    }

private:

    string name;

};

int main() {

    unique\_ptr<Cat> c1 = make\_unique<Cat>("小白");

    c1->meow();

    // unique\_ptr 不能複製，這會錯誤：

    // unique\_ptr<Cat> c2 = c1;

    // 要用 move() 把擁有權轉移

    unique\_ptr<Cat> c2 = move(c1);

    if (!c1)

        cout << "c1 是空的" << endl;

    c2->meow();

} // 離開 scope 時，Cat 自動被刪除