

C++ 17 template _auto_ 非型別樣版參數型別推導

C++ 中的三種樣板參數種類

在 C++ 的樣板系統中，我們可以定義以下三種類型的樣板參數：

1. 型別樣板參數 (Type Template Parameter)

這是最常見的樣板參數，用來傳遞「型別」給樣板類別或函式。

```
template <typename T>
struct A {};
```

例如：

```
A<int> a1;    // 傳入型別 int
A<std::string> a2; // 傳入型別 std::string
```

2. 樣板樣板參數 (Template Template Parameter)


這個參數本身是「一個樣板類別」！

常見用法是：傳入 `std::vector`、`std::deque` 等樣板容器的「樣板本體」：

```
template <template <typename, typename> class Container>
struct B {};
```

這裡表示 `Container` 是一個接受兩個型別樣板參數的樣板類別，也就是符合這種形式：

```
template <typename T, typename Alloc>
class SomeContainer {};
```

 符合這樣的有：`std::vector`、`std::deque`、`std::list` 等。

範例：

```
B<std::vector> b1; // OK, std::vector<T, Allocator> 符合
```

3.

非型別樣板參數（Non-type Template Parameter）

這類參數用來傳入「具體的值」，像是 `int`、`char`、`bool`，甚至是指標等 Literal Type。

```
template <int N>
struct C {};
```

例如：

```
C<10> c1; // N = 10
```



綜合範例：三種樣板參數同時出現

```
#include <vector>
#include <iostream>

template <
    typename T,                // #1 型別樣板參數
    template <typename, typename> class Container, // #2 樣板樣板參數
    int K                      // #3 非型別樣板參數
>
struct Example {
    Container<T, std::allocator<T>> data;

    void fill() {
        for (int i = 0; i < K; ++i)
            data.push_back(static_cast<T>(i));
    }
}
```

```

void print() {
    for (const auto& x : data)
        std::cout << x << " ";
    std::cout << std::endl;
}

};

int main() {
    Example<int, std::vector, 5> ex;
    ex.fill(); // 插入 0~4
    ex.print(); // 印出：0 1 2 3 4
}

```

解釋這段程式的樣板參數：

樣板參數名	類型	說明
T	型別樣板參數	決定容器中元素的型別（例如 int）
Container	樣板樣板參數	傳入如 std::vector 這種接受兩個型別樣板參數的容器
K	非型別樣板參數	傳入一個整數（用於控制迴圈次數）

延伸：C++17 之後 `template<auto>` 是什麼？

```

template <auto N>
struct Holder {
    static constexpr auto value = N;
};

```

這是 C++17 引入的新寫法，讓 非型別樣板參數 可以是任意 literal type，自動推導型別。

範例：

```
Holder<42> h1;    // 推導 N 為 int
Holder<'A'> h2;   // 推導 N 為 char
Holder<true> h3;  // 推導 N 為 bool
Holder<3.14> h4;  // 推導 N 為 double (注意要是 literal，不能是變數)
```

這比以前的 `template<int>` 或 `template<char>` 更彈性。

函式樣板參數推導 (Function Template Argument Deduction) VS 非型別樣板參數推導 (Non-type Template Argument Deduction)

基本名詞解釋

名詞	意思	例子	
樣板形式參數 (Template Formal Argument)	在 <code>template<></code> 中出現的變數，例如 <code>template<typename T></code> 中的 <code>T</code> 。	<code>{cpp}template<typename T> void f(T x);</code> 中的 <code>T</code>	
樣板實際參數 (Template Actual Argument)	實際用來替換樣板參數的值或型別。	<code>{cpp}f(42)</code> → <code>T = int</code>	
函式形式參數 (Function Formal Argument)	函式宣告時的參數名稱及型別。	<code>{cpp}void f(int x)</code> 中的 <code>x</code>	
函式實際參數 (Function Actual Argument)	呼叫函式時傳入的實際參數。	<code>f(42)</code> 中的 <code>42</code>	

一、函式樣板參數推導 (Function Template Argument Deduction)

◆ 重點：

編譯器會根據「函式實際參數」去推導「樣版型別參數 (Type Template Parameter)」的型別。

範例 1：單純的函式樣板推導

```
template <typename T> // T 是樣板形式參數
void printType(T x) { // x 是函式形式參數
    std::cout << typeid(T).name() << std::endl;
}

int main() {
    printType(42); // 傳入 int → 編譯器推導出 T = int
    printType(3.14); // 傳入 double → T = double
}
```

推導過程：

呼叫	函式實際參數	編譯器推導結果
{cpp}printType(42)	int	T = int
{cpp}printType(3.14)	double	T = double

二、非型別樣板參數推導 (Non-type Template Argument Deduction)


◆ 重點：

編譯器會根據傳入的「值」推導出 非型別樣板參數 (例如整數、指標、字元等) 的型別。

範例 2：非型別樣板結構

```
template <auto N> // N 是非型別樣版參數
struct ConstValue {
    static void print() {
        std::cout << "Value: " << N << std::endl;
    }
};

int main() {
    ConstValue<10>::print(); // N = 10, 推導型別為 int
    ConstValue<'A'>::print(); // N = 'A', 推導型別為 char
}
```

 推導過程：

呼叫	樣板實際參數	編譯器推導結果
ConstValue<10>	10	N = 10, 型別 int
ConstValue<'A'>	'A'	N = 'A', 型別 char

綜合範例（文章中的程式碼）

這段程式碼融合了兩種推導方式：

```
template <typename T, auto Size> // #3 樣板形式參數
void test(T (&array)[Size]) { // #2 函式形式參數
    std::cout << typeid(decltype(Size)).name()
        << " Size = " << Size << std::endl;
}

int main() {
    int a[3];
    test(a); // #1 函式呼叫
}
```

◆ {cpp}template <typename T, auto Size>

這裡定義了一個函式樣版（function template），有兩個樣版參數：

1. `T` : 型別參數 (type parameter)
2. `auto Size` : 非型別樣版參數，意思是 `Size` 是一個「值」，而不是型別。這裡使用 `auto` 讓編譯器自行推導 `Size` 的實際型別。

◆ `{cpp}void test(T (&array)[Size])`

這是一個接受固定大小陣列參考的參數：

- `T (&array)[Size]` 表示：`array` 是一個「參考 (reference) 到 `T` 類型的陣列，且長度是 `Size`」。

例如：`int a[3]`; 被推導成 `T == int`、`Size == 3`，所以參數型別就是 `int (&array)[3]`。

◆ `{cpp}test(a);` → 發生什麼事？

當呼叫 `test(a)` 時，編譯器會依據 `a` 的型別來進行樣版推導 (template argument deduction)。

🧠 分析推導流程：

1. `{cpp}test(a)` → `a` 是一個 `int[3]` 的陣列。
2. `{cpp}T (&array)[Size]` 是一個參考到陣列的參數，所以：
 - 推導出 `{cpp}T = int`
 - 推導出 `{cpp}Size = 3`
3. 這時，`Size` 是一個「非型別樣版參數」，而它的型別是根據 3 推導出來的 → `int`。

🔍 「非型別樣版參數推導」是什麼？

通常「非型別樣版參數」(如 `template<int N>`) 的型別必須明確指定。但是在這個例子中：

```
template <typename T, auto Size>
```

`auto` 讓編譯器根據 `array` 的大小自動推導出 `Size` 的型別與值。

所以這裡的說法：

「先以『函式樣版參數推導』推導出 `T == int` 與 `Size == 3`，然後再以『非型別樣版參數推導』推導 `Size` 的型別」

是說：

1. 先根據 `a` 的型別 `int[3]` 判斷 `T` 與 `Size`；
2. 然後根據 `Size == 3`，推論出 `Size` 的型別是什麼（如 `int` 或 `std::size_t`）。

這個「兩階段」推導在你寫 `auto Size` 的時候才會發生。如果你一開始寫 `int Size`，就只會做一步（不需要型別推導）。

? 那這個有什麼實用嗎？

就像原文說的：

「這個例子沒有什麼實際用途，因為陣列大小是少數能被『函式樣版參數推導』推導出的非型別樣版參數。」

簡單說：

- 大多數情況下，非型別樣版參數要手動指定；
- 陣列大小剛好是一個例外，可以從參數型別中間接推導出來；
- 所以你也可以寫成更簡單的形式，例如直接用 `template<typename T, int Size>`。

✅ 差異對照表

面向	函式樣版參數推導	非型別樣版參數推導
推導對象	<code>typename T</code> 、 <code>class T</code>	<code>auto N</code> 、 <code>int N</code> 、 <code>char N</code> 等
根據什麼推導	函式呼叫時的實際參數型別	傳入的具體值
發生時機	呼叫函式樣板時	使用樣板結構或函式時，包含值
推導結果	型別	值 + 值的型別
可否自動	✅（編譯器自動判斷）	✅（若使用 <code>auto</code> 型別）
範例	<code>printType(42) → T=int</code>	<code>ConstValue<10> → N=10 (int)</code>

✅ 總結整理

名詞	意義
<code>T</code>	一般樣版型別參數

名詞	意義
auto Size	非型別樣版參數，值會從陣列大小推導
T (&array)[Size]	參考到固定大小陣列的參數型別
推導順序	先從參數型別推導出 T 與 Size 的值，再推導出 Size 的型別
特殊之處	陣列大小是少數可推導出非型別樣版參數的例子
實用性	不高，多為語法實驗性質，但可做為範例

型別檢查輸出：

```
typeid(decltype(Size)).name() // 編譯期顯示 Size 的型別（可能是 int、size_t 依編譯器而異）
```

使用情境：將常數表達式包裝為型別

C++ 泛型程式設計（Generic Programming）通常會將計算結果儲存於以 `value` 為名字的靜態常數成員。

C++ 標準函式庫提供的 [std::integral_constant](#) 就是典型的範例：

```
template <typename T, T Value>
struct integral_constant {
    static constexpr T value = Value;
};
```

1. integral_constant 是什麼？

`integral_constant` 是 C++ 標準庫裡一個模板類別，它用來包裝一個常數值，讓這個常數成為一個「型別的靜態成員」。簡單說，就是把「數值」變成型別的一部分。

```
template <typename T, T Value>
struct integral_constant {
    static constexpr T value = Value;
};
```

- T 是型別（例如 `int`、`bool`）

- `Value` 是該型別的常數值

範例：

```
#include <iostream>

int main() {
    std::cout << integral_constant<int, 5>::value << std::endl; // 印出 5
}
```

這裡：

- `integral_constant<int, 5>` 是一個型別，裡面有一個叫 `value` 的靜態常數，其值是 5。

2. 「樣版別名 (Template Alias)」是什麼？

因為 `integral_constant<int, 5>` 每次都要寫兩個參數，寫起來麻煩，如果我們想做布林值版本，就可以用樣版別名簡化：

```
template <bool Value>
using bool_constant = integral_constant<bool, Value>;
```

這樣做有什麼好處？

你以後要用布林值常數，只要寫：

```
bool_constant<true> // 代表 integral_constant<bool, true>
bool_constant<false> // 代表 integral_constant<bool, false>
```

不需要每次都寫完整的 `integral_constant<bool, true>`。

範例：

```
#include <iostream>

template <bool Value>
using bool_constant = integral_constant<bool, Value>;

int main() {
    std::cout << bool_constant<true>::value << std::endl; // 印出 1
}
```

3. C++17 的 `template <auto>` 讓型別推導更方便

C++17 引入了「非型別樣板參數自動推導」，讓你可以這樣寫：

```
template <auto Value>
struct constant {
    static constexpr auto value = Value;
};
```

這裡的 `Value` 可以是任何型別的常量，例如 `int`、`bool`、`char`、`constexpr` 數字等等。

使用時，只需要傳入數值 `Value`：

```
constant<5> // 自動推導 Value 為 int 5
constant<true> // Value 為 bool true
```

範例：

```
#include <iostream>

template <auto Value>
struct constant {
    static constexpr auto value = Value;
};

int main() {
```

```
std::cout << constant<5>::value << std::endl; // 5
std::cout << constant<true>::value << std::endl; // 1 (bool true)
}
```

小結

用法	說明
<code>integral_constant<T, Value></code>	傳入型別與數值，明確指定
<code>bool_constant<Value></code>	使用別名，簡化寫法，固定型別為 <code>bool</code>
<code>constant<Value></code> (C++17)	使用 <code>auto</code> 讓編譯器自動推導數值型別，更簡潔

使用情境：泛型成員存取函式

有時候我們會使用 `std::transform` 搭配 Unary Function（一元函式）提取物件資料成員：

```
#include <algorithm>
#include <iostream>
#include <iterator>

struct X {
    int a;
    double b;
};

int GetA(const X &x) {
    return x.a;
}

double GetB(const X &x) {
    return x.b;
}

int main() {
    X xs[] = {{1, 2.2}, {3, 4.4}, {5, 6.6}, {7, 8.8}};
```

```

std::cout << "int:" << std::endl;
std::transform(std::begin(xs), std::end(xs),
               std::ostream_iterator<int>(std::cout, "\n"),
               GetA);

std::cout << "double:" << std::endl;
std::transform(std::begin(xs), std::end(xs),
               std::ostream_iterator<double>(std::cout, "\n"),
               GetB);

return 0;
}

```

我們能編寫一個泛型函式同時涵蓋 `GetA` 與 `GetB`：

```

template <typename DataType, typename ClassType,
          DataType ClassType::*Ptr>
DataType Getter(const ClassType &obj) {
    return obj.*Ptr;
}

```

上述 `Getter` 函式分別有三個樣版參數：

- `DataType` 是資料成員的型別
- `ClassType` 是物件類別型別
- `Ptr` 是資料成員指標（Pointer to Data Member）

透過這三個樣版參數我們可以明確定義 `Getter` 要從什麼類別存取什麼資料成員：

```

#include <algorithm>
#include <iostream>
#include <iterator>

template <typename DataType, typename ClassType, DataType ClassType::*Ptr>
DataType Getter(const ClassType &obj) {
    return obj.*Ptr;
}

```

```

struct X {
    int a;
    double b;
};

int main() {
    X xs[] = {{1, 2.2}, {3, 4.4}, {5, 6.6}, {7, 8.8}};

    std::cout << "int:" << std::endl;
    std::transform(std::begin(xs), std::end(xs),
        std::ostream_iterator<int>(std::cout, "\n"),
        Getter<int, X, &X::a>); // Modified

    std::cout << "double:" << std::endl;
    std::transform(std::begin(xs), std::end(xs),
        std::ostream_iterator<double>(std::cout, "\n"),
        Getter<double, X, &X::b>); // Modified

    return 0;
}

```

但是在這個例子中，我們先宣告 `DataType` 與 `ClassType` 型別樣版參數是為了宣告 `Ptr` 「非型別樣版參數」的型別。實際上 `Ptr` 已經提供足夠的資訊。我們能以 `template <auto>` 改寫：

```

template <typename T>
struct MemPtrTraits {};

template <typename U, typename V>
struct MemPtrTraits<U (V::*)> {
    typedef V ClassType;
    typedef U DataType;
};

template <auto Ptr>
const typename MemPtrTraits<decltype(Ptr)>::DataType &
Getter(const typename MemPtrTraits<decltype(Ptr)>::ClassType &obj) {

```

```
return obj.*Ptr;
}
```

✓ 1. 什麼是 std::transform ？

std::transform 是 C++ 標準函式庫中的一個演算法，用來對一個序列的每個元素，套用一個函式，然後輸出到另一個容器或迭代器中。

🔑 語法（常見版本）：

```
std::transform(input_begin, input_end, output_iterator, unary_function);
```

- input_begin 、 input_end : 輸入序列的起訖
- output_iterator : 輸出資料要寫入的位置
- unary_function : 接受一個參數並回傳新值的函式（或函式物件）

✓ 範例：

```
#include <algorithm>
#include <iostream>
#include <vector>

int square(int x) { return x * x; }

int main() {
    std::vector<int> input = {1, 2, 3, 4};
    std::transform(input.begin(), input.end(),
        std::ostream_iterator<int>(std::cout, " "),
        square); // 輸出：1 4 9 16
}
```

✓ 2. Pointer to Data Member 是什麼？

它是一種指向類別中成員（而非物件）的特殊指標型別。

📌 語法形式：

```
<資料型別> <類別型別>::*
```

例如：

```
int X::* ptr = &X::a; // 指向 X 類別中的 int 資料成員 a
```

這種指標只能在某個 `X` 物件上使用，例如：

```
X obj{42, 3.14};
std::cout << obj.*ptr << std::endl; // 等價於 obj.a
```

✅ 實用範例（用在泛型 Getter 中）：

```
template <typename T, typename C, T C::*Ptr>
T Getter(const C& obj) {
    return obj.*Ptr;
}

Getter<int, X, &X::a>(some_x); // 從物件 some_x 取出 a 成員的值
```

✅ 3. 為什麼要定義 MemPtrTraits ？

💡 問題背景：

我們希望寫一個函式 `Getter`，接受任意成員指標（例如 `&X::a`、`&X::b`），並自動知道：

- 該成員的型別（`int`，`double` ...）
- 該成員所屬的類別型別（`X`）

🧠 但 `auto Ptr` 本身沒有顯式提供型別資訊：

你只有：

```
template <auto Ptr>
const ??? &Getter(const ??? &obj);
```

所以我們需要用 `decltype(Ptr)` 來從型別推導成員的資訊。這就需要 type traits 來解析「資料成員指標」的型別結構。

實作：MemPtrTraits

```
template <typename T>
struct MemPtrTraits {}; // 空的預設模板，留給特化使用

// 特化：當 T 是某個類別的資料成員指標時
template <typename U, typename V>
struct MemPtrTraits<U V::*> {
    using ClassType = V;
    using DataType = U;
};
```

這段意思是：只要我們遇到 `U V::*`（即指向某個類別成員的指標），我們就能抽出：

- `U` 是該成員的型別
- `V` 是擁有這個成員的類別

使用：

```
template <auto Ptr>
const typename MemPtrTraits<decltype(Ptr)>::DataType &
Getter(const typename MemPtrTraits<decltype(Ptr)>::ClassType &obj) {
    return obj.*Ptr;
}
```

先傳入 `Ptr` 的型別，再透過 `MemPtrTraits` 推導出它指向的資料型別，最後取用該型別作為 `const reference`。

推導流程圖

傳入某型別 `Ptr`



| `decltype(Ptr)` | ← 取得 `Ptr` 的實際型別（例如：`int*`）



| `MemPtrTraits<T>` | ←— 使用 `decltype(Ptr)` 作為模板參數 `T`



| `MemPtrTraits<T>::Datatype` | ←— 在特化版本中定義的型別別名



✓✓✓ 得到實際的 `Datatype` 型別 (例如 `int`)

推導一下：

1. 呼叫 `Getter` 傳入 `(&X::*)`
2. 觸發 `<auto ptr> template`
3. 推導出 `ptr` 為 `(int X::*)`
4. 呼叫 `MemPtrTraits` 傳入 `(decltype(ptr))`
5. 觸發

```
template <typename U, typename V>
struct MemPtrTraits<U V::*> {
    using ClassType = V;
    using DataType = U;
};
```

6. 取得 `ClassType`

搭配具體範例說明：

範例目的

情境：你有一個成員指標，例如 `int MyStruct::*`，你想要寫一個泛型工具，可以從這個型別中萃取出 `int`。

我們會建立一個型別萃取器 `MemberPointerTraits<T>`，當你給它 `int MyStruct::*`，它就能提供別名 `Datatype = int` 和 `ClassType = MyStruct`。

程式碼分段解說

```

#include <iostream>
#include <type_traits>

// 📖 1. Primary template : 預設情況，沒有定義 Datatype 和 ClassType
template <typename T>
struct MemberPointerTraits;

// 📦 2. Specialization for member pointer : T = Datatype, C = ClassType
template <typename T, typename C>
struct MemberPointerTraits<T C::*> {
    using Datatype = T;
    using ClassType = C;
};

// 🛠 測試用的 struct
struct MyStruct {
    double value;
};

int main() {
    // 🌸 3. 定義成員指標型別
    double MyStruct::* ptr = &MyStruct::value;

    // 🧠 4. 使用 decltype 取得 ptr 的型別
    using Traits = MemberPointerTraits<decltype(ptr)>;

    // ✅ 驗證萃取成功
    static_assert(std::is_same_v<Traits::Datatype, double>);
    static_assert(std::is_same_v<Traits::ClassType, MyStruct>);

    std::cout << "Traits::Datatype is double.\n";
    std::cout << "Traits::ClassType is MyStruct.\n";
}

```

📌 一步步解析（推導流程圖）

你也可以想成以下的邏輯鏈：

```

ptr => double MyStruct::*
|
v
decltype(ptr) => double MyStruct::*
|
v
MemberPointerTraits<double MyStruct::*>
|
v (特化)
template<typename T, typename C>
struct MemberPointerTraits<T C::*> {
    using Datatype = T;    // => double
    using ClassType = C;   // => MyStruct
};

```

小筆記：為什麼這樣設計？

- `T C::*` 是 C++ 語言中「成員指標」的型別語法：表示「C 類別中的某個 T 型別的欄位」。
- 透過這種語法，我們可以在 `template` 特化中把 `T` 和 `C` 拆出來，進一步使用或轉換。

最終使用：

```

Getter<&X::a>(x); // 回傳 x.a
Getter<&X::b>(x); // 回傳 x.b

```

這就是將成員指標轉為泛型函式的一種技巧性寫法。

總結表：

名詞	意思	舉例
<code>std::transform</code>	對每個元素套用一個函式	<code>transform(xs, ys, f)</code>
Pointer to data member	指向類別中某個資料成員的指標	<code>int X::* ptr = &X::a</code>

名詞	意思	舉例
MemPtrTraits	從成員指標推導型別資訊	MemPtrTraits<int X::*>::DataType 是 int
