

## Section 13-3 Delegating Constructor

### 委派建構子

#### 什麼是 Delegating Constructor ?

在 C++11 之後，建構子可以呼叫同一類別裡的其他建構子，來避免重複初始化邏輯。

#### 為什麼需要它？

在過去（C++11 以前）我們常這樣寫多載建構子：

```
class Student {  
private:  
    string name;  
    int age;  
  
public:  
    Student() {  
        name = "Unknown";  
        age = 0;  
    }  
  
    Student(string name) {  
        this->name = name;  
        age = 0;  
    }  
  
    Student(string name, int age) {  
        this->name = name;  
        this->age = age;  
    }  
};
```

❗ 問題：很多「重複的邏輯」反覆寫在每個建構子裡，難以維護。

## ✓ 使用 Delegating Constructor 改寫

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

public:
    Student() : Student("Unknown", 0) {}           // 委派給第三個建構子
    Student(string name) : Student(name, 0) {}      // 委派給第三個建構子
    Student(string name, int age) {
        this->name = name;
        this->age = age;
    }
};
```

### 🎯 關鍵句：

```
Student() : Student("Unknown", 0) {}
```

意思是：

🧠 「呼叫 Student(string name, int age) 這個建構子，傳入指定的預設值」

## ✓ 使用方式

```
Student s1;           // name="Unknown", age=0
Student s2("Tom");    // name="Tom", age=0
Student s3("Amy", 20); // name="Amy", age=20
```

## 🎯 詳細分解建構子的「初始化列表」語法 + 委派建構子的呼叫

```
Student()           // 建構子本體的定義
: Student("Unknown", 0) // <-- 初始化列表，這裡在委派呼叫另一個建構子
```

```
{ } // 建構子函式的本體（可以是空的）
```

## 拆解逐句說明

部分	解釋
<code>Student()</code>	宣告無參數建構子
<code>: Student("Unknown", 0)</code>	利用初始化列表，委派呼叫自己類別內另一個建構子（這是 C++11 新語法）
<code>{ }</code>	實作主體，這裡是空的，因為初始化工作已經在被呼叫的建構子裡完成了

## 它實際上做了什麼？

這句等於說：

「當我用 `Student()`（無參數建構子）建立物件時，請自動去呼叫 `Student(string name, int age)` 並傳入 `"Unknown"` 和 `0`」

然後那個有參數的建構子負責真正的初始化：

```
Student(string name, int age) {
    this->name = name;
    this->age = age;
}
```

## 等價於什麼？

這行：

```
Student() : Student("Unknown", 0) {}
```

功能上就等價於傳統寫法：

```
Student() {
    name = "Unknown";
    age = 0;
}
```

但差別在於：

👉 用委派寫法可以避免在每個建構子都重複寫 `name = "Unknown"` 這種初始化邏輯

## ✨ 使用委派的優點再強調一次：

- ☒ 把重複的初始化邏輯集中寫在一個建構子裡（通常是「最大參數版」）
- ☒ 其他建構子只負責「轉呼叫」那個建構子 → 程式碼簡潔、好維護
- ☒ 若未來初始化邏輯改變，只要改一個地方！

## 🔧 補充例子（對照預設參數）

委派建構子寫法：

```
class Student {
public:
    Student() : Student("Unknown", 0) {}
    Student(string name) : Student(name, 0) {}
    Student(string name, int age) {
        this->name = name;
        this->age = age;
    }
};
```

預設參數寫法：

```
class Student {
public:
    Student(string name = "Unknown", int age = 0) {
        this->name = name;
        this->age = age;
    }
};
```

# 委派建構子 (Delegating Constructor) vs 預設參數 (Default Parameters)

✅ 表面上：可以達成相同效果

這兩段程式碼都可以用來支援不同數量的參數建立物件：

**A** 委派建構子寫法：

```
class Student {  
public:  
    Student() : Student("Unknown", 0) {}  
    Student(string name) : Student(name, 0) {}  
    Student(string name, int age) {  
        this->name = name;  
        this->age = age;  
    }  
};
```

**B** 預設參數寫法：

```
class Student {  
public:  
    Student(string name = "Unknown", int age = 0) {  
        this->name = name;  
        this->age = age;  
    }  
};
```

使用時一樣：

```
Student s1;           // name = "Unknown", age = 0  
Student s2("Tom");    // name = "Tom", age = 0  
Student s3("Amy", 18); // name = "Amy", age = 18
```

## 🧠 但實際上有「語意」和「設計風格」上的差異！

比較項目	委派建構子	預設參數
定義位置	多個建構子	單一建構子
重複初始化邏輯	可集中在一個主建構子中處理 <input checked="" type="checkbox"/>	所有邏輯要塞在一個建構子裡 🤪
可讀性/維護性	好（每個建構子語意清楚）	當參數多、邏輯複雜時難維護
彈性（可定義不同行為）	<input checked="" type="checkbox"/> 每個建構子可以自訂不同行為	<input checked="" type="checkbox"/> 所有邏輯只能放在一個函式中
效能	幾乎相同（編譯器會優化）	幾乎相同
支援版本	C++11 才支援	C++98 就支援

## 🎯 總結一句話：

如果只是想快速寫個小範例、參數邏輯很簡單：可以用預設參數。

但若參數多、邏輯複雜、或不同建構子需要不同初始化邏輯：請用委派建構子，程式碼更乾淨、可讀性更高。

## 🧠 使用委派建構子時，建構子本體 {} 還是可以寫程式碼！

### 🧪 範例：加上「印出訊息」的邏輯

```
#include <iostream>
#include <string>
using namespace std;
class Student {
private:
    string name;
    int age;
public:
    // 主建構子：實際做初始化
```

```

Student(string name, int age) {
    this->name = name;
    this->age = age;
    cout << "🏠 正在建立學生：" << name << ", 年齡：" << age << endl;
}

// 無參數建構子：委派 + 額外邏輯
Student() : Student("Unknown", 0) {
    cout << "✅ 使用預設值建立學生物件" << endl;
}

// 單參數建構子：委派 + 額外邏輯
Student(string name) : Student(name, 0) {
    cout << "✅ 未輸入年齡，使用預設值 0" << endl;
}
};

```

## 🔧 測試程式：

```

int main() {
    Student s1;      // 無參數
    Student s2("Alice"); // 只有名字
    Student s3("Bob", 20); // 完整資訊
    return 0;
}

```

### 📄 執行結果會是：

```

```sql
🏠 正在建立學生：Unknown, 年齡：0 ✅ 使用預設值建立學生物件
🏠 正在建立學生：Alice, 年齡：0 ✅ 未輸入年齡，使用預設值 0
🏠 正在建立學生：Bob, 年齡：20

```

## ✅ 小結論

類型	建構子本體 {} 可以做什麼？
✅ 委派建構子	可以寫額外的行為（例如 <code>cout</code> 、 <code>log</code> 、計算等等）
✅ 主建構子	實際處理變數初始化的邏輯

你可以把主建構子想成「🔧 真正初始化的工廠」，其他委派建構子像是「📦 預設組合包裝方案」，但他們可以在出貨前偷偷加上貼紙、保護泡泡紙之類的額外動作

🧠 如果只用預設參數 `= ...` 的方式，在建構子裡就必須手動用 `if-else` 判斷來模擬委派建構子能做的事。

## A 使用委派建構子 ✅

👉 每一種建構子情境的行為可以清楚分開，程式碼簡單、易讀：

```
class Student {
private:
    string name;
    int age;
public:
    Student(string name, int age) {
        this->name = name;
        this->age = age;
        cout << "📦 建立學生：" << name << ", 年齡：" << age << endl;
    }

    Student() : Student("Unknown", 0) {
        cout << "✅ 使用預設值建立學生物件" << endl;
    }

    Student(string name) : Student(name, 0) {
        cout << "✅ 未輸入年齡，使用預設年齡 0" << endl;
    }
};
```

## B 如果只用預設參數 🧨

👉 你就得在一個建構子裡面搞定所有情況，變得很難維護，像這樣：



```

class Student {
private:
    string name;
    int age;
public:
    Student(string name = "Unknown", int age = 0) {
        this->name = name;
        this->age = age;
        if (name == "Unknown" && age == 0) {
            cout << "✅ 使用預設值建立學生物件" << endl;
        } else if (age == 0) {
            cout << "✅ 未輸入年齡，使用預設年齡 0" << endl;
        }
        cout << "🏠 建立學生：" << name << ", 年齡：" << age << endl;
    }
};

```

## 🔍 結果與比較

做法	特點
<b>A</b> 委派建構子	✅ 每種情境對應一個建構子，邏輯集中、可讀性高、維護方便
<b>B</b> 預設參數建構子	❌ 所有邏輯都塞在一個建構子裡，需要用 if-else 判斷使用者的參數意圖

## 🧠 額外補充：預設參數其實不知道「你有沒有真的傳進來」

```

Student s1();           // 會配到 `name = "Unknown", age = 0`
Student s2("Amy");      // 配到 `name = "Amy", age = 0`

```

你在建構子裡是無法直接知道 `age` 是不是使用了預設值，因為對建構子來說：

```
Student("Amy"); // 就是等於 Student("Amy", 0)
```

它不知道 `0` 是你寫的，還是預設的，除非你自己寫 `if (age == 0)` 這種判斷。

## ★ 逐步解析委派建構子的呼叫順序：

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

class Student {
private:
    string name;
    int age;

public:
    // 主建構子
    Student(string name, int age) {
        this->name = name;
        this->age = age;
        cout << "🏠 建立學生：" << name << ", 年齡：" << age << endl;
    }

    // 委派建構子 1：無參數
    Student() : Student("Unknown", 0) {
        cout << "✅ 使用預設值建立學生物件" << endl;
    }

    // 委派建構子 2：只有名字
    Student(string name) : Student(name, 0) {
        cout << "✅ 未輸入年齡，使用預設年齡 0" << endl;
    }
};
```

## 🧪 測試程式：

```
int main() {
    Student s1;
    Student s2("Amy");
}
```

```
Student s3("Bob", 20);
}
```

## 呼叫順序解析

### ◆ Student s1; 呼叫順序：

s1 → 呼叫 Student() → 委派到 Student("Unknown", 0)

- └—— 實際初始化 name 和 age
- └—— 印出 🏠 建立學生：Unknown, 年齡：0

然後繼續執行 Student() 的本體

- └—— 印出 ✅ 使用預設值建立學生物件

🧠 換句話說：

Student() → Student("Unknown", 0) → 印出 → 回到 Student() → 印出

### ◆ Student s2("Amy"); 呼叫順序：

s2 → 呼叫 Student(string name) → 委派到 Student(name, 0)

- └—— 實際初始化 name = "Amy", age = 0
- └—— 印出 🏠 建立學生：Amy, 年齡：0

然後繼續執行 Student(name) 的本體

- └—— 印出 ✅ 未輸入年齡，使用預設年齡 0

🧠 換句話說：

Student("Amy") → Student("Amy", 0) → 印出 → 回到 Student("Amy") → 印出

### ◆ Student s3("Bob", 20); 呼叫順序：






s3 → 直接呼叫 Student("Bob", 20)

- └—— 初始化 name = "Bob", age = 20
- └—— 印出 🏠 建立學生：Bob, 年齡：20

🧠 換句話說：

`Student("Bob", 20)` → 印出

## ✓ 小結論

呼叫方式	會呼叫哪個建構子	委派到哪	最終印出順序
<code>Student()</code>	<code>Student()</code>	→ <code>Student("Unknown", 0)</code>	 → 
<code>Student("Amy")</code>	<code>Student(string name)</code>	→ <code>Student(name, 0)</code>	 → 
<code>Student("Bob", 20)</code>	<code>Student(string name, int)</code>	無	

## ✓ 實務經驗補充：




實務上，很多中大型專案會：

- 使用 **委派建構子** 負責組合邏輯
- 然後使用 **初始化列表** 處理初始化
- 把「邏輯和數值」的來源分開

## 🧠 委派建構子 vs 預設參數 vs 多載

技術	優點	缺點
預設參數	簡單一個建構子搞定	所有邏輯都要寫在一個函式內
多載建構子	彈性高，可以個別寫邏輯	程式碼重複多
委派建構子 	避免重複邏輯、每個建構子仍可客製化	需要 C++11 支援

## 📌 補充注意事項

1.  只能委派給同一類別裡的建構子（不能呼叫父類別的建構子）
2.  只能出現在初始化列表中
3.  不能委派給自己（會無限遞迴）

## const 與 reference 成員必須在建構子初始化列表中初始化，而且只能初始化一次！

 我們先用一個簡單的範例：

```
#include <iostream>
#include <string>
using namespace std;
class Student {
private:
    const string school;
    int age;
public:
    Student(string schoolName, int a) : school(schoolName), age(a) {
        cout << "
```

```
Student(string schoolName, int a) {
    school = schoolName; // ❌ 錯誤：const 不能這樣賦值
    age = a;
}

};
```

❌ 錯誤訊息會像這樣：

error: assignment of read-only member 'Student::school'

## 🎯 進一步舉例：reference 成員

```
class Student {
private:
    string& nickname; // reference 必須初始化
    int age;
public:
    Student(string& nick, int a) : nickname(nick), age(a) {
        cout << "✅ nickname 指向的是：" << nickname << endl;
    }
};
```

- `string& nickname` 是一個 reference（參考），一樣只能在初始化時設定。
- 如果你沒有在初始化列表中指定 `nickname` 的對象，編譯器也會報錯！

💣 **結論：如果類別成員有這些，你不能依賴委派建構子的初始化邏輯！**

成員類型	是否能在建構子本體內賦值？	是否需要初始化列表？	可否靠委派初始化？
一般變數（如 <code>int</code> ）	✅ 可以	❌ 不一定	✅ 可以
<code>const</code> 變數	❌ 不行	✅ 必須	✅ 可以
reference 變數	❌ 不行	✅ 必須	✅ 可以，但需小心順序

## ⚠ 委派建構子 + const 成員的注意點

```
class Student {
    private:
        const string school;

    public: // 主建構子，處理 const 初始化
        Student(string s) : school(s) {
            cout << "主建構子\n";
        } // ❌ 委派建構子若沒初始化 const，會編譯錯

        Student() : Student("Default School") {
            cout << "委派建構子\n";
        }
};
```

✅ 這樣可以，但要記住：只有主建構子能初始化 const 成員，因為一旦委派了，就無法在自己這層初始化 const ！

## 🔗 圖解補充：成員初始化在哪裡？

```
class Student {
    const string school;
    string& nickname;
    int age;

    // ✅ 建構子初始化列表處理：
    Student(string s, string& n, int a)
        : school(s), nickname(n), age(a) {
        // ❌ 無法在這裡改 school 或 nickname
    }
};
```

# 加入 const 和 reference 成員到委派建構子的例子

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

class Person {
private:
    const string school;    // const 成員，必須在初始化列表初始化
    string& nickname;       // reference 成員，必須在初始化列表初始化
    int age;

    // 用來提供參考用的「預設暱稱」
    static string default_nick;

public:
    // 主建構子：最完整的初始化入口
    Person(const string& schoolName,
           string& nick,
           int a)
        : school(schoolName), nickname(nick), age(a)
    {
        cout << "[主建構子] 初始化完成："
              << "school = " << school
              << ", nickname = " << nickname
              << ", age = " << age
              << endl;
    }

    // 委派建構子 1：只給學校名稱
    Person(const string& schoolName)
        : Person(schoolName, default_nick, 0) // → 委派給「主建構子」
    {
        cout << "[委派建構子] 只指定 school，使用 default_nick、age = 0" << endl;
    }
}
```



```
// 委派建構子 2：不給任何參數
Person()
: Person("Default School", default_nick, 18) // → 委派給「主建構子」
{
    cout << "[委派建構子] 使用完全預設值 (Default School, default_nick, 18)" << endl;
}
};

// 靜態成員定義
string Person::default_nick = "NoNickname";

int main() {
    cout << "=== 建立 p1 ===" << endl;
    Person p1;           // 呼叫無參數委派建構子

    cout << "\n=== 建立 p2 ===" << endl;
    Person p2("High School"); // 呼叫單參委派建構子

    cout << "\n=== 建立 p3 ===" << endl;
    string myNick = "TomCat";
    Person p3("University", myNick, 20); // 直接呼叫主建構子

    return 0;
}
```

## 執行結果 (示意)

=== 建立 p1 ===

[主建構子] 初始化完成：school = Default School, nickname = NoNickname, age = 18

[委派建構子] 使用完全預設值 (Default School, default\_nick, 18)

=== 建立 p2 ===

[主建構子] 初始化完成：school = High School, nickname = NoNickname, age = 0

[委派建構子] 只指定 school，使用 default\_nick、age = 0

=== 建立 p3 ===

[主建構子] 初始化完成：school = University, nickname = TomCat, age = 20

## 呼叫順序拆解（以 `Person p2("High School");` 為例）

1. 編譯器看到 `Person(const string&)` → 執行此委派建構子
2. 初始化列表：`Person(schoolName, default_nick, 0)` → 跳去呼叫主建構子
3. 主建構子執行初始化列表：
  - `school(schoolName)`
  - `nickname(default_nick)`
  - `age(0)`
4. 主建構子本體印出 `[主建構子] ...`
5. 返回到委派建構子本體，印出 `[委派建構子] ...`

這樣的設計能確保：

- `const`、`reference` 成員在最底層的主建構子裡一次完成初始化。
  - 其他委派建構子 **不需要重複「初始化」邏輯**，只專注在「呼叫不同預設參數」和「印出對應訊息」。
  - 程式結構清晰、易維護。
-