

Section 13-1 Class and Object

概念講解 | Concept

◆ 類別 (Class)

類別就像是一張「設計藍圖」，它定義了一種資料結構，這個結構包含了：


- 屬性 (Attributes)：也稱為成員變數，用來描述這個類別有哪些資料。
- 行為 (Behaviors)：也就是成員函式 (Member Functions)，描述這個類別能做什么事情。

 類別本身不會佔用記憶體空間，它只是個模板。

◆ 物件 (Object)

物件是根據類別「建造出來的實體」，會佔用記憶體空間。

你可以把類別想成「建築圖」，物件就是「真正蓋出來的房子」。

 每次從類別產生一個物件，就像用模具倒出一個產品一樣。

範例程式碼 (以 C++ 撰寫)

我們來定義一個 Dog 類別，並創建幾個 Dog 物件：

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

// 類別定義
class Dog{
public:
    // 成員變數
    string name;
    int age;

    // 成員函式
    void bark() {
        cout << name << "：汪汪！" << endl;
    }
}
```

```
};

int main() {
    // 建立物件
    Dog dog1;
    dog1.name = "Lucky";
    dog1.age = 3;
    dog1.bark(); // 呼叫成員函式

    Dog dog2;
    dog2.name = "Coco";
    dog2.age = 5;
    dog2.bark(); // 呼叫成員函式

    return 0;
}
```

關鍵點整理

類別 (Class)	物件 (Object)
設計藍圖	類別的實體
不佔記憶體	佔記憶體
用 class 定義	用 類別名 變數名; 宣告

物件的動態宣告

在 C++ 中，靜態宣告的物件（像前面範例裡的 `Dog dog1;`）會：

- 在堆疊（stack）區域配置記憶體
- 函式執行完畢後自動釋放，不需要 `delete`

而有時候，我們需要：

- 動態建立物件（在執行階段才決定）
- 回傳一個在函式裡創建的物件指標
- 或物件太大，不適合放在 stack 裡

這時就需要 動態宣告。

範例：動態建立類別物件

我們再用 Dog 這個類別，示範動態建立物件的方法：

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

class Dog {
public:
    string name;
    int age;

    void bark() {
        cout << name << "：汪汪！（我今年 " << age << " 歲）" << endl;
    }
};

int main() {
    // 動態建立物件（用 new 配置）
    Dog* dogPtr = new Dog;

    // 用指標存取成員（記得用 -> 而不是 .）
    dogPtr->name = "Buddy";
    dogPtr->age = 2;
    dogPtr->bark();

    // 用完記得釋放記憶體
    delete dogPtr;

    return 0;
}
```

重點說明

語法	意義
Dog* dogPtr = new Dog;	在 heap 上建立一個 Dog 物件，回傳其指標

語法	意義
<code>dogPtr->name</code>	使用箭頭 <code>-></code> 存取成員（因為是指標）
<code>delete dogPtr;</code>	手動釋放記憶體，防止記憶體洩漏

靜態 vs 動態 比較

項目	靜態宣告	動態宣告
宣告方式	<code>Dog d;</code>	<code>Dog* d = new Dog;</code>
記憶體區域	Stack	Heap
存取成員	<code>d.name</code>	<code>d->name</code>
釋放方式	自動	手動（ <code>delete</code> ）
彈性	固定	高，可動態配置多個物件

實際應用場景 | 什麼情況「必須」動態宣告物件？

1 數量不固定（執行期間才知道）

你無法在寫程式時就知道會有幾個物件，這時動態宣告就變得必要。

◆ 情境範例：

你做一個購物網站，使用者購物車裡放幾項商品是不固定的：

```
int n;
cin >> n;
Product* products = new Product[n]; // n 是執行時才知道
```

2 物件「生命週期」要延長（跨函式存在）

靜態物件的生命週期是「宣告的區塊」，出了那個函式會被自動釋放；但如果你想從函式中建立物件並傳出去，就要用動態宣告。

◆ 範例：

```

Dog* createDog() {
    Dog* d = new Dog;
    d->name = "Puppy";
    return d;
}

int main() {
    Dog* myDog = createDog();
    myDog->bark(); // OK !
    delete myDog; // 不刪會 memory leak
}

```

🧠 如果 `createDog()` 裡用 `Dog d;`，出了函式那隻狗就不見了（記憶體釋放了），會變成懸空指標（dangling pointer）。

3 建立大型物件，避免 Stack Overflow

Stack 的大小有限（例如 1~8 MB），如果你建立超大結構（例如一整張 10000x10000 的圖片），會造成 stack overflow。
這時應該放在 **heap** 裡（動態宣告）。

◆ 圖像編輯軟體中的圖片資料：

```
Image* img = new Image(width, height); // 如果你用 Image img; 可能會爆掉 stack
```

4 多型（Polymorphism）應用

當你用父類別指標指向子類別物件，動態宣告是必備的。

◆ 例子：

```

class Animal {
public:
    virtual void speak() {}
};

class Cat : public Animal {
public: void speak() {
    cout << "喵 !" << endl;
}
}

```

```
};

class Dog : public Animal {
public: void speak() {
    cout << "汪 !" << endl;
}
};

int main() {
    Animal* a1 = new Cat;
    Animal* a2 = new Dog;
    a1->speak(); // 輸出：喵！
    a2->speak(); // 輸出：汪！
    delete a1;
    delete a2;
}
```

這是 C++ 物件導向的重點之一：虛擬函式（virtual function）+ 指標。

補充：避免記憶體洩漏

用 `new` 就一定要配 `delete`，不然會造成：

- 記憶體永遠無法回收（記憶體洩漏）
- 長時間運作的程式會逐漸佔滿 RAM → 系統效能下降

小小結論

情境	為什麼用動態宣告
執行時才知道要幾個物件	<code>new</code> 配合變數數量建立
建立的物件要跨函式存在	不能用區域變數，只能 <code>new</code>
避免 stack overflow	大型資料應該丟 heap
多型 + 虛擬函式	通常用父類別指標指向子類別物件
實作資料結構（linked list, tree）	節點間的連接都是動態配置