



03 類別與物件 (II)

Fundamental Computer Programming- C++ Lab(II)

元智大學 | C++ 程式設計實習 (二)

張維元

課程投影片：[請從元智個人 Portal 下載](#)

Outline

- 01 物件導向與開發環境
- 02 陣列、向量與結構
- 03 類別與物件
- 04 物件導向程式設計: 多載
- 05 物件導向程式設計: 繼承
- 06 樣板
- 07 C++ 進階用法

什麼是程式？

程式語言是用來命令電腦執行各種作業的工具，是人與電腦溝通的橋樑。當電腦藉由輸入設備把程式讀入後，會儲存在主記憶體內，然後指令會依序被控制單元提取並解碼或翻譯成電腦可以執行的信號，並把信號送到各個裝置上，以執行指令所指派的動作。也就是說，人類與電腦溝通的語言稱為程式語言。

程式 = 利用一系列的指令告訴電腦如何執行工作



物件導向程式設計

物件導向程式設計（Object-oriented programming，OOP）是種具有物件概念的程式設計典範，同時也是一種程式開發的抽象方針。它可能包含資料、屬性、程式碼與方法。物件則指的是類別（class）的實例。它將物件作為程式的基本單元，將程式和資料封裝其中，以提高軟體的重用性、靈活性和擴充性。

把物件作為程式最小的單位，模擬實體世界的運作



從結構 (struct) 到類別 (class)

C/C++ 而結構 (Struct) 提供在一個變數內自定義儲存的資料屬性。

```
1  
2 struct Book {  
3     string title;  
4     int price;  
5 };  
6
```

```
1  
2 class Book {  
3     public:  
4         string title;  
5         int price;  
6 };  
7
```

結構與類別的差別

差別	C 語言 Struct	C++ 語言 Struct	C++ 語言 Class
成員變數	不支援靜態變數	支援靜態變數	支援靜態變數
成員函式	不支援成員函式	支援成員函式	支援成員函式
成員預設值	不支援成員預設值	支援成員預設值	支援成員預設值
成員範圍	public	public、protect、private	public、protect、private
預設成員範圍	X	public	private
物件導向特性	X	繼承、多型、建構子 ...	繼承、多型、建構子 ...

→ C++ Class 就像是多了繼承特性的 C 語言 Struct

C++ 的結構 (struct) 很像 C++ 的類別 (class)

→ C++ 結構支援了 C++ 類別大部分的性質與用法

```
1  
2 struct Book {  
3     string title;  
4     int price;  
5 };  
6
```

≠

```
1  
2 class Book {  
3     string title;  
4     int price;  
5 };  
6
```

C++ 的結構 (struct) 很像 C++ 的類別 (class)

→ 其中一個主要的差別在於預設的變數範圍不同

```
1  
2 struct Book {  
3     string title;  
4     int price;  
5 };  
6
```

≠

```
1  
2 class Book {  
3     public:  
4         string title;  
5         int price;  
6 };  
7
```

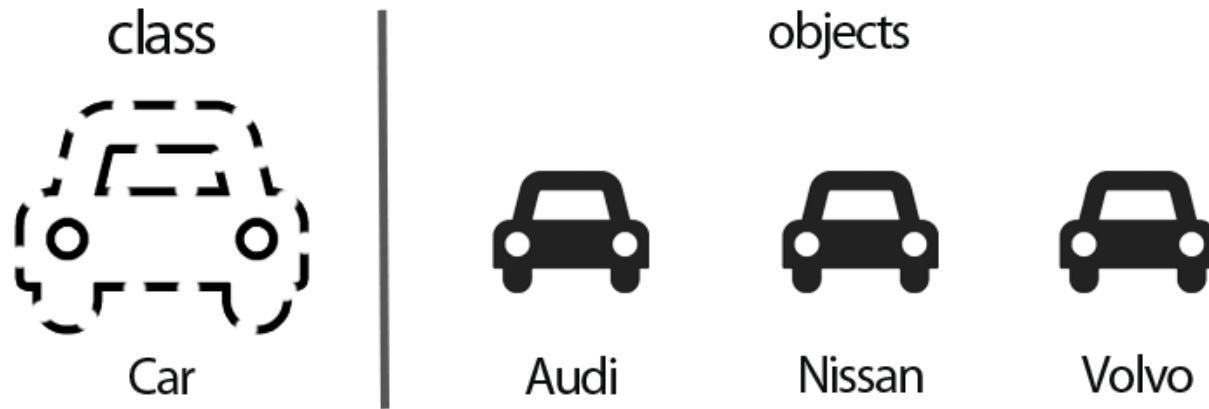

從結構 (struct) 到類別 (class)

```
1
2 class Book {
3     public:
4         string title;
5         int price;
6     };
```

```
int main( ){
    Book book1;
    book1.title = "C++ How to Program";
    book1.price = 636;
    cout << book1.title << endl;
    cout << book1.price << endl;
    return 0;
}
```

類別與物件

- 類別 (Class) 利用屬性與方法定義了物件的抽象樣板/結構
- 物件 (Object) 是類別的實體 (instance) ，可以使用類別方法



→ Class 是抽象的描述， Object 才是實體的存在

類別與物件

- 類別 (Class) 利用屬性與方法定義了物件的抽象樣板/結構
- 物件 (Object) 是類別的實體 (instance) ，可以使用類別方法

```
1  
2  
3 class Book {  
4     public:  
5         string title;  
6         int price;  
7 };  
8  
9
```

```
1 int main( ){  
2  
3     Book book1;  
4  
5     book1.title = "C++ How to Program";  
6     book1.price = 636;  
7  
8     cout << book1.title << endl;  
9     cout << book1.price << endl;  
10    return 0;  
11  
12 }
```

→ 定義了一個叫做「Book」的變數型態

成員函式

(member function)

```
1
2 class Book {
3     public:
4         string title;
5         int price;
6         void f(void);
7 };
8
9 void Book::f(){
10     cout << "I'm a book: " + title;
11 };
12
```

```
1
2 int main( ){
3
4     Book book1;
5     book1.title = "C++ How to Program";
6     book1.f();
7
8     return 0;
9 }
10
11
12
```

成員函式

(member function)

```
1
2 class Book {
3     public:
4         string title;
5         int price; ← 成員變數
6         void f(void); ← 成員函式
7 };
8
9 void Book::f(){
10     cout << "I'm a book: " + title;
11 };
12
```

```
1
2 int main( ){
3
4     Book book1;
5     book1.title = "C++ How to Program";
6     book1.f();
7
8     return 0;
9 }
10
11
12
```

成員函式

(member function)

```
1
2 class Book {
3     public:
4         string title;
5         int price; ← 成員變數
6         void f(void); ← 成員函式
7 };
8
9 void Book::f(){
10     cout << "I'm a book: " + title;
11 };
12     → 成員函式可存取成員變數
```

```
1
2 int main( ){
3
4     Book book1;
5     book1.title = "C++ How to Program";
6     book1.f();
7
8     return 0;
9 }
10
11
12
```

結構與類別的差別

差別	C 語言 Struct	C++ 語言 Struct	C++ 語言 Class
成員變數	不支援靜態變數	支援靜態變數	支援靜態變數
成員函式	不支援成員函式	支援成員函式	支援成員函式
成員預設值	不支援成員預設值	支援成員預設值	支援成員預設值
成員範圍	public	public、protect、private	public、protect、private
預設成員範圍	X	public	private
物件導向特性	X	繼承、多型、建構子 ...	繼承、多型、建構子 ...

→ 其中一個主要的差別在於預設的變數範圍不同

未指定的成員預設是私有變數

```
1
2 class Book {
3     int price;
4     public:
5         void set(int p);
6         double get(void);
7 };
8
9 void Book::set(int p){
10     price = p;
11 };
12
13 double Book::get(void){
14     return price * 0.9;
15 };
16
```

```
1 int main( ){
2
3     Book book1;
4     book1.price = 600;
5     cout << "price: " << book1.get();
6
7     return 0;
8 }
9
10
```


未指定的成員預設是私有變數

```
1
2 class Book {
3     private:
4         int price;
5     public:
6         void set(int p);
7         double get(void);
8 };
9
10 void Book::set(int p){
11     price = p;
12 };
13
14 double Book::get(void){
15     return price * 0.9;
16 };
```

```
1 int main( ){
2
3     Book book1;
4     book1.price = 600;
5     cout << "price: " << book1.get();
6
7     return 0;
8 }
9
10
```

Private 和 Public

```
1
2 class Book {
3     private: ← 私有成員
4         int price;
5     public: ← 公開函式
6         void set(int p);
7         double get(void);
8 };
9
10 void Book::set(int p){
11     price = p;
12 };
13
14 double Book::get(void){
15     return price * 0.9;
16 };
```

```
1 int main( ){
2
3     Book book1;
4     book1.price = 600;
5     cout << "price: " << book1.get();
6
7     return 0;
8 }
9
10
```

不能直接存取私有變數

```
1
2 class Book {
3     private: ← 私有成員
4         int price;
5     public: ← 公開函式
6         void set(int p);
7         double get(void);
8 };
9
10 void Book::set(int p){
11     price = p;
12 };
13
14 double Book::get(void){
15     return price * 0.9;
16 };
```

```
1 int main( ){
2
3     Book book1;
4     book1.price = 600;
5     cout << "price: " << book1.get();
6
7     return 0;
8 }
9
10
```

利用公開成員函式存取私有成員變數

```
1
2 class Book {
3     private: ← 私有成員
4         int price;
5     public: ← 公開函式
6         void set(int p);
7         double get(void);
8 };
9
10 void Book::set(int p){
11     price = p;
12 }; → 成員函式可存取成員變數
13
14 double Book::get(void){
15     return price * 0.9;
16 };
```

```
1 int main( ){
2
3     Book book1;
4     book1.set(600);
5     cout << "price: " << book1.get();
6
7     return 0;
8 }
9
10
```

static 修飾子

```
1
2 class Book {
3     static int i;
4     public:
5         int id;
6         void f(void);
7 };
8 int Book::i = 20210330;
9
10 void Book::f(void){
11     cout << "I'm a book: " << id;
12     cout << "i = " << i << endl;
13 };
```

```
1
2 int main( ){
3
4     Book books[10];
5     for(int i = 0; i < 10; i++){
6         books[i].id = i;
7         books[i].f();
8     }
9     return 0;
10 }
```

function 也有 static 修飾子

```
1 #include <iostream>
2 using namespace std;
3
4 void count(){
5     static int c = 1;
6     cout << c << endl;
7     c++;
8 }
9
10 int main(){
11     for(int i = 0; i < 10; i++) {
12         count();
13     }
14     return 0;
15 }
```

→ static 可以保留變數在同一個 function 使用

static 修飾子 = 靜態成員變數

```
1
2 class Book {
3     static int i;
4     public:
5         int id;
6         void f(void);
7 };
8 int Book::i = 20210330;
9
10 void Book::f(void){
11     cout << "I'm a book: " << id;
12     cout << "i = " << i << endl;
13 };
```

```
1
2 int main( ){
3
4     Book books[10];
5     for(int i = 0; i < 10; i++){
6         books[i].id = i;
7         books[i].f();
8     }
9     return 0;
10 }
```

static 修飾子

```
1
2 class Book {
3     static int i;
4     public:
5         int id;
6         void f(void);
7 };
8 int Book::i = 20210330; ← static
9                               要記得初始化
10 void Book::f(void){
11     cout << "I'm a book: " << id;
12     cout << "i = " << i++ << endl;
13 };
```

```
1
2 int main( ){
3
4     Book books[10];
5     for(int i = 0; i < 10; i++){
6         books[i].id = i;
7         books[i].f();
8     }
9     return 0;
10 }
```

→ static 是同一個 class 下，所有 object 共享的變數 24

建構子 (constructor)

建構子（constructor；ctor），是一個類別裡用於建立物件時初始化成員變數函式。在初始化一個新建的物件，能夠同時輸入參數用以設定實例變數。

```
1
2 class Book {
3     public:
4         int price;
5         string title;
6         Book(int p){
7             price = p;
8         };
9     };
10
```


```
1
2
3 int main( ){
4     Book book1(600);
5     cout << "price: " << book1.price;
6     return 0;
7 }
8
9
10
```

建構子 (constructor)

建構子（constructor；ctor），是一個類別裡用於建立物件時初始化成員變數函式。在初始化一個新建的物件，能夠同時輸入參數用以設定實例變數。

```
1
2 class Book {
3     public:
4         int price;
5         string title;
6         Book(int p){
7             price = p;
8         };
9     };
10
```

```
1
2
3 int main( ){
4     Book book1(600);
5     cout << "price: " << book1.price;
6     return 0;
7 }
8
9
10
```



建構子多載

(Constructor Overloading)

```
1 class Book {  
2     public:  
3         int price;  
4         string title;  
5         Book(int p){  
6             price = p;  
7         };  
8         Book(int p, string t){  
9             price = p;  
10            title = t;  
11        };  
12};
```

```
1  
2  
3 int main( ){  
4     Book book1(600);  
5     Book book2(536, "C++ Prime");  
6  
7 }  
8  
9  
10
```

建構子多載

(Constructor Overloading)

```
1 class Book {  
2     public:  
3         int price;  
4         string title;  
5         Book(int p){ ← 一個輸入  
6             price = p;  
7         };  
8         Book(int p, string t){  
9             price = p;  
10            title = t;  
11        };  
12    };
```

```
1  
2  
3 int main( ){  
4     Book book1(600);  
5     Book book2(536, "C++ Prime");  
6  
7 }  
8  
9  
10
```

→ 多載 = 同一個函式能夠接受不同的參數輸入

成員變數的預設值

```
1 class Book {  
2     public:  
3         int price;  
4         string title = "default";  
5         Book(int p){  
6             price = p;  
7         };  
8         Book(int p, string t){  
9             price = p;  
10            title = t;  
11        };  
12    };
```

```
1  
2  
3 int main( ){  
4     Book book1(600);  
5     Book book2(536, "C++ Prime");  
6  
7 }  
8  
9  
10
```

解構子 (destructor)

解構子 (destructor ; dtor) 在物件導向程式設計裡是一個方法，當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {
2     public:
3         int id;
4         Book(int i){
5             id = i;
6             cout << id << "建構子呼叫";
7         };
8         ~Book(){
9             cout << id << "解構子呼叫";
10        }
11};
```

```
1
2
3 int main( ){
4     Book b1(1);
5     Book b2(2);
6     return 0;
7 }
8
9
10
```

解構子的發生時機

解構子（destructor ; dtor）在物件導向程式設計裡是一個方法，
當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，
清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {
2     public:
3         int id;
4         Book(int i){
5             id = i;
6             cout << id << "建構子呼叫";
7         };
8         ~Book(){
9             cout << id << "解構子呼叫";
10        }
11    };
```

```
1 void f(){
2     Book b2(2);
3     return ;
4 };
5
6 int main( ){
7     Book b1(1);
8     f();
9     return 0;
10 }
11
```

解構子的發生時機

解構子（destructor ; dtor）在物件導向程式設計裡是一個方法，
當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，
清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {
2     public:
3         int id;
4         Book(int i){
5             id = i;
6             cout << id << "建構子呼叫";
7         };
8         ~Book(){
9             cout << id << "解構子呼叫";
10        }
11    };
```

```
1 void f(){
2     Book b2(2);
3     return ;
4 };
5
6 int main( ){
7     Book b1(1);
8     f();
9     return 0;
10 }
11
```

1 建構子呼叫

解構子的發生時機

解構子（destructor ; dtor）在物件導向程式設計裡是一個方法，
當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，
清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {
2     public:
3         int id;
4         Book(int i){
5             id = i;
6             cout << id << "建構子呼叫";
7         };
8         ~Book(){
9             cout << id << "解構子呼叫";
10        }
11    };
```

```
1 void f(){
2     Book b2(2);
3     return ;
4 };
5
6 int main( ){
7     Book b1(1);
8     f();
9     return 0;
10 }
11
```

1 建構子呼叫

解構子的發生時機

解構子（destructor ; dtor）在物件導向程式設計裡是一個方法，
當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，
清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {
2     public:
3         int id;
4         Book(int i){
5             id = i;
6             cout << id << "建構子呼叫";
7         };
8         ~Book(){
9             cout << id << "解構子呼叫";
10        }
11    };
```

```
1 void f(){
2     Book b2(2);
3     return ;
4 };
5
6 int main( ){
7     Book b1(1);
8     f();
9     return 0;
10 }
11
```

- 1 建構子呼叫
- 2 建構子呼叫

解構子的發生時機

解構子（destructor ; dtor）在物件導向程式設計裡是一個方法，
當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，
清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {  
2     public:  
3         int id;  
4         Book(int i){  
5             id = i;  
6             cout << id << "建構子呼叫";  
7         };  
8         ~Book(){  
9             cout << id << "解構子呼叫";  
10        }  
11};
```

```
1 void f(){  
2     Book b2(2);  
3     return ;  
4 };  
5  
6 int main( ){  
7     Book b1(1);  
8     f();  
9     return 0;  
10 }  
11
```

- 1 建構子呼叫
- 2 建構子呼叫
- 2 解構子呼叫

解構子的發生時機

解構子（destructor ; dtor）在物件導向程式設計裡是一個方法，
當物件的生命週期結束時自動地被呼叫執行。它最主要的目的在於，
清空並釋放物件先前建立或是佔用的記憶體資源。

```
1 class Book {
2     public:
3         int id;
4         Book(int i){
5             id = i;
6             cout << id << "建構子呼叫";
7         };
8         ~Book(){
9             cout << id << "解構子呼叫";
10        }
11};
```

```
1 void f(){
2     Book b2(2);
3     return ;
4 };
5
6 int main( ){
7     Book b1(1);
8     f();
9     return 0;
10 }
11
```

1 建構子呼叫
2 建構子呼叫
2 解構子呼叫
1 解構子呼叫

解構子 (destructor)

→ 動態生成的物件不會自動呼叫解構子

```
1 class Book {  
2     public:  
3         int id;  
4         Book(int i){  
5             id = i;  
6             cout << id << "建構子呼叫";  
7         };  
8         ~Book(){  
9             cout << id << "解構子呼叫";  
10        }  
11    };
```

```
1  
2 int main( ){  
3     Book *b1 = new Book(1);  
4     Book *b2 = new Book(2);  
5     Book *b3 = new Book(3);  
6     return 0;  
7 }  
8  
9  
10
```

1 建構子呼叫
2 建構子呼叫
3 建構子呼叫

解構子 (destructor)

→ 動態生成的物件釋放的時候才會呼叫解構子

```
1 class Book {  
2     public:  
3         int id;  
4         Book(int i){  
5             id = i;  
6             cout << id << "建構子呼叫";  
7         };  
8         ~Book(){  
9             cout << id << "解構子呼叫";  
10        }  
11    };
```

```
1 int main( ){  
2     Book *b1 = new Book(1);  
3     Book *b2 = new Book(2);  
4     Book *b3 = new Book(3);  
5  
6     delete b2;  
7     delete b3;  
8     delete b1;  
9     return 0;  
10 }
```

1 建構子呼叫
2 建構子呼叫
3 建構子呼叫
2 解構子呼叫
3 解構子呼叫
1 解構子呼叫

Thanks for listening.

元智大學 | C++ 程式設計實習

Wei-Yuan Chang