**Ch3. 資料型別抽象化**

這章中，首先會簡介物件導向程式設計，因為後續在實作資料結構時會使用到物件導向的精神與概念，再來，會定義一些物件導向程式設計相關的名詞，並說明什麼叫做「資料型別抽象化」。

接下來的重點是實作資料型別的抽象化，按照下列步驟，把一個抽象資料型別完整的建立出來。

A. 類別宣告

B. 定義類別下的方法

C. 類別的權限設定

D. 建構式與解構式

E. 重載運算子

F. 函式模板與類別模板

G. 標頭檔的建立

**第一節：物件導向程式設計簡介**

本門課不是物件導向課程，因此不會講解完整的物件導向內容，如多型、設計模式等主題都不會涵蓋，只會介紹與後續章節相關的概念。

1. 物件導向程式設計的由來

要了解為什麼會產生物件導向程式設計，要回顧電腦的歷史：早期程式設計以硬體導向為主，因當時電腦體積龐大，然而運算能力卻不如今日智慧型手機。

由於運算能力和儲存空間的雙重限制，在開發程式時需要時時注意節省記憶體與效能。當時電腦可能只有數 KB 至數 MB 的記憶體，所以過去常使用 short （2 Bytes）來修飾 int（4 Bytes），在硬體效能逐漸增加的今天，short 已經很少被使用。

判定資料型別部分，C/C++ 交給開發者手動進行，比如通常宣告變數時必須指定資料型別（屬於強型態程式語言）；相對的，較為晚近的 Python 和 Matlab 等語言中，變數的型態可以交由編譯器判斷，宣告時不需要手動指定資料型態。事實上，C/C++ 發展到今日，也已經可以使用關鍵字 auto ，讓編譯器直接指定變數的資料型別。

這顯示出一種典範轉移，由早期硬體導向的各種考量，轉為兼顧程式開發者開發上的便利。

2. 為何會發展出物件導向程式語言

C 語言是 C++ 語言的前身，正是因為 C 語言有其不足，才發展出 C++。

C 語言的強項是硬體開發，非常貼近硬體，然而 C 也有明顯的不足，當硬體越來越好，程式碼也越來越長，C 語言的瓶頸也越來越明顯。像 word 這樣的軟體有數十萬行程式，用 C 編寫會難以維護與擴展，比如新增功能就很困難。

「維護與擴展」問題的解方，就是「物件導向」。

物件導向是程式設計的一種精神，在開發的當下就想到未來的擴展性與維護性，而非像學校作業完全不需考慮到未來使用。商業上使用的程式沒有「開發完」的一天，會不斷需要修正，因此當下就要考慮維護與擴展的方便性。

3. 物件導向的精神：Once and only once

「Once and only once」的精神也可以用「DRY」這個縮寫來表示。

「DRY」指的是「Don't Repeat Yourself」，意即「不要重複你自己」。當有同一個功能、同一段程式碼需要在不同地方使用時，不要複製貼上，同樣的程式碼段落應該只出現一次。

與之對應的，是「WET」這個縮寫。「WET」可以是很多短句的縮寫，比如「Write everything twice」、「We enjoy typing」、「Waste everyone's time」，這些很明顯都不是優秀的程式設計師應有的習慣。

4. 物件導向程式設計的發展過程

最早的程式設計方式是「非結構化程式設計」，這也就是想到什麼就打什麼，這適用於程式碼小的時候。

「結構化程式設計」則是導入函式的概念，把單一常用到的功能放到函式之中，讓程式碼可以依據功能被切割成不同部分以利重複使用。

「模組化程式設計」更進一步，在開始進行開發程式的工作之前，就先以函式為單位，評估需要用到哪些功能。一個例子是開發賽車遊戲的時候，以「前進」、「後退」、「左轉」、「右轉」四個函式為基礎開發，就符合「模組化程式設計」，這樣開發出來的程式會是由多個函式建立起來的。

然而即使是模組化程式設計，仍有牽一髮動全身的問題。像骨牌一樣，函式 A 呼叫函式 B，函式 B 又呼叫函式 C，…，只要函式 C 壞掉，函式 A、B 也都會受到波及而無法正常運作，也很難知道問題究竟出在哪個函式裡。

在發展的最後，產生了「物件導向程式設計」，程式不再是以「函式」為單位，而是由「物件」為單位，這些物件各有功能、可以互相溝通，會是我們後續實作資料結構時的基礎。

傳統程式設計程式是由許多函式、指令組成；相對的，物件導向設計程式由獨立又可互相呼叫的物件組成，這些物件能獨立被撰寫、使用、呼叫、儲存、運作，且物件間可以彼此合作。這就像是台北、高雄等大城市中有許多居民，每個居民都如同一個物件，可以獨立工作，公車司機的職責是開車、便利商店店員的職責是管理店舖、結帳等，各司其職也能互相合作，像司機和店員都是「物流」的一部分。

5. 從物件導向到設計模式

物件導向是一種不同程式語言間共通的精神，在使用 C++、Python 等支持物件導向的程式語言時，程式設計師必須熟悉如何將程式拆解成物件。

設計模式是程式設計經驗的總結，能夠確保開發出的程式碼可靠性、複用、易讀。常見的設計模式共有 23 種，採用某種設計模式，就決定了每種物件應該怎麼寫、如何更新、維護。設計模式是程式的「架構」，這樣的架構又建立在物件導向之上，如果目標是程式架構師，物件導向與設計模式都是必定要熟悉的課題。

6. 物件導向的三大基石

「物件導向」的概念最早由 Brad Cox 在 1980 年代提出，在 C 語言中導入了物件與訊息的概念。同時，他也建立起了物件導向的三大基石：繼承（Inheritance）、封裝（Encapsulate）和動態連結（Dynamic binding）。

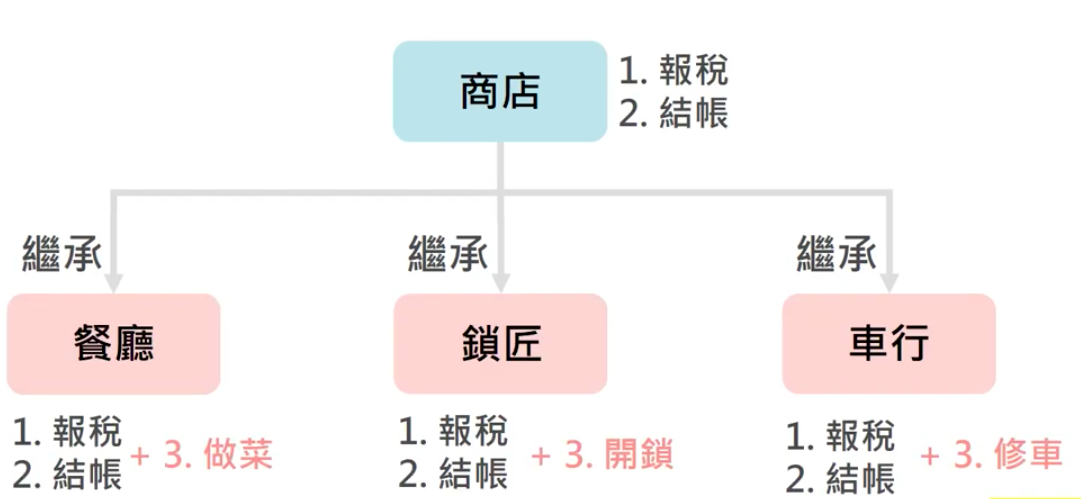
根據這三大基石，Objective-C 這個「物件化的 C 語言」被開發出來，早期的 iOS App 開發就是使用 Objective-C（現在已經改用 Swift）。

(1) 繼承

「繼承」指的是一個物件可以「直接得到其他物件的內容」。當「子類別」繼承了「父類別」，「子類別」就得到了「父類別」中含有的內容。

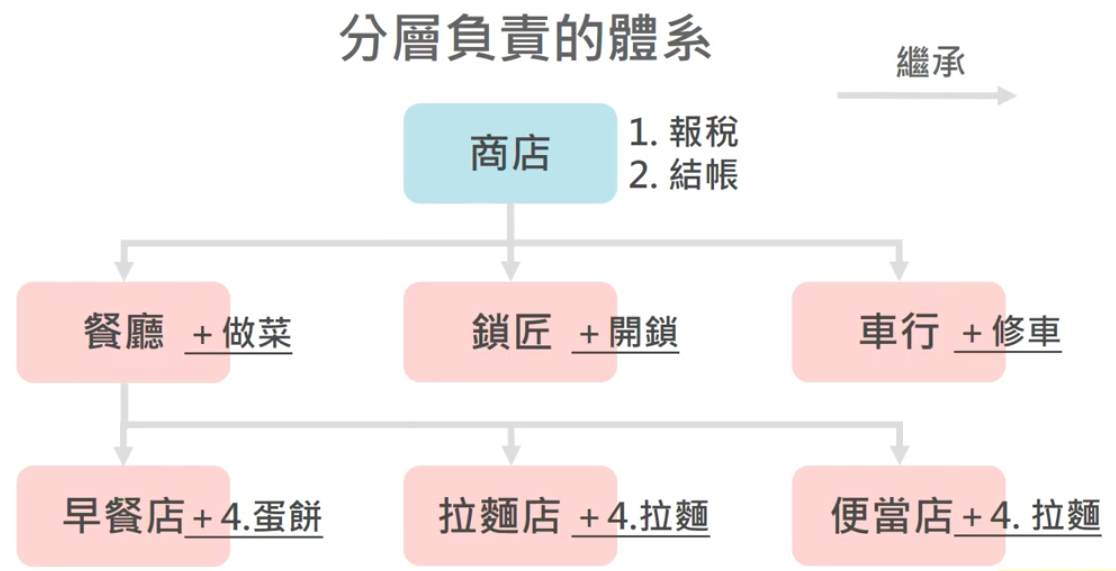
比如「餐廳」可以繼承「商店」，其中「商店」是父類別，「餐廳」是子類別，商店如果本來有兩個功能：「報稅」與「結帳」，那麼由於餐廳也是商店的一種，餐廳只要直接「繼承」商店即可。這樣一來，餐廳自然會帶有報稅與結帳的功能，不需要重寫一次這兩個函式。

繼承可以先讓一個類別取得其他類別的功能後，再進行擴寫，像「商店」只需要加上「做菜」的功能，就成為一家「餐廳」。



如果要設計許多家不同種類的商店，可以直接繼承父類別「商店」的功能，讓每種商店都有「報稅」和「結帳」的功能，之後再擴充每種商店獨自的功能，如餐廳加上「做菜」、鎖匠加上「開鎖」、車行加上「修車」。

這樣一來，共同的功能報稅與結帳不需要重複設計，且子類別（如餐廳）的程式碼有問題時，不會影響到父類別（商店）和其他子類別（鎖匠和車行）的使用。



除了單一層的繼承以外，也可以分成很多層，以分層負責的概念設計，一層負責一件事情。例如餐廳是繼承商店而來，「早餐店」可以繼承餐廳並加上蛋餅，拉麵店一樣繼承餐廳再加上拉麵，便當店繼承餐廳後再加上便當。

(2) 封裝

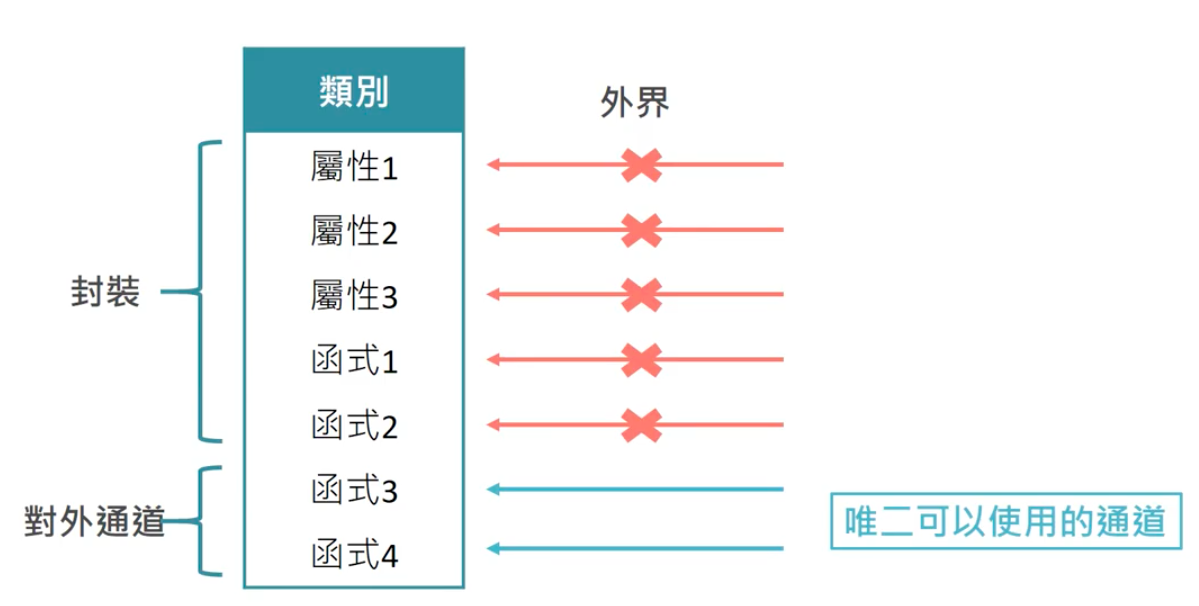
封裝的目的是降低物件跟介面的複雜度，透過封裝，可以達到資訊隱藏、限制外界使用的功能，並且減少物件間的互相干擾，確保修改物件時，不會牽一髮而動全身。

封裝之後，對外通道應該有「小而美」的特性，「小」指的是只開啟必要的對外通道、其他通道預設是關閉的（越少越好）；「美」指的是方便使用者使用（常用的功能都有提供）。

若不封裝，各個函式會像骨牌一樣，A 叫 B、B 叫 C、…，沒有經過完善的封裝，難以得知哪些函式會被呼叫。若一個類別下寫了 20 個函式，沒有經過封裝的情況下，這 20 個函式都有可能被外界呼叫，之後修改任何一部分都必須全部檢查。

封裝又可以比擬成房間的「門」，房間是封裝好的場地，門是對外通道。一個房間最好只有一個門，若有十幾扇門，則很難維護和管理。

一般來說，只需要為了方便人進出而開一扇門，如果房間裡有衛浴，可以再多開一扇門，究竟要開多少通道需要視設計而定，但理想上通道不應過多。

經過封裝，外界能不能拿到類別內的屬性或函式？通常不行，為了和外界合作，封裝時需要額外設計對外通道。

上圖中，紅色叉叉代表屬性 1、2、3 與函式 1、2 都沒有辦法被外界取用，外界若要使用函式的內容或修改，必須透過通道，也就是函式 3 與函式 4。

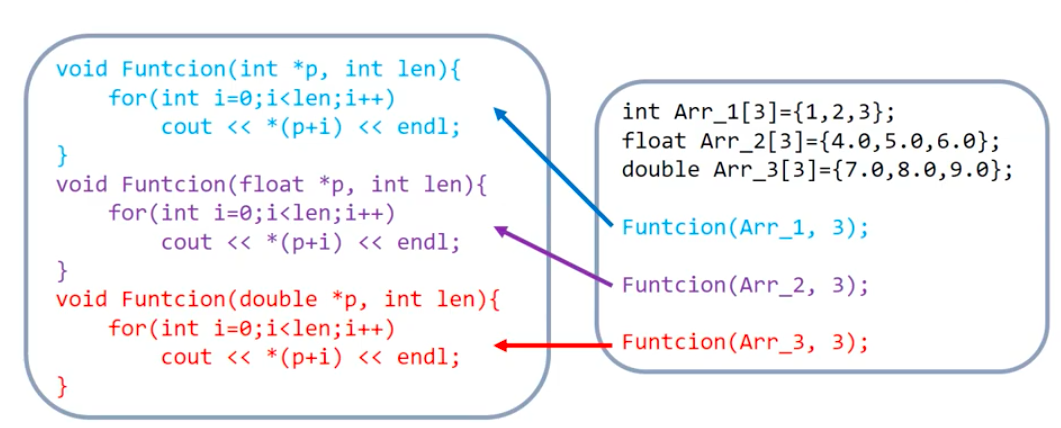
這樣一來，更新和維護就變得比較方便，因為前五個屬性 / 函式已經被封裝起來，因此改寫也沒關係，未來更新維護只需檢查對外通道。這就如同改裝自己家裡時，不需要昭告天下，只需和家人溝通即可，外人既然沒有進到家裡權限，也就不會受到影響。

(3) 靜態連結與動態連結



「連結」指的是把識別字「綁定」到某個記憶體位址，在編譯時，識別字名稱為何並沒有實際影響，因為識別字都已經被轉換為其綁定的記憶體位置了，無論一個 int 的名字是 a、b 或 c，都不會影響到執行結果。

「靜態」與「動態」連結的差異在於識別字是「何時」綁定到記憶體位置上：

靜態連結（Static Binding）指的是識別字對應到的記憶體在「編譯階段」決定，程式碼變成程式的過程中就已經綁定；動態連結（Dynamic Binding）指的是識別字對應到的記憶體在「執行階段」才決定。

|  |  |
| --- | --- |
| 靜態連結的例子：函式多載的綁定在「編譯」時進行 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | void Function(int \*p, int len){  for(int i=0;i<len;i++)  cout << \*(p+i) << endl;  }  void Function(float \*p, int len) {  for(int i=0;i<len;i++)  cout << \*(p+i) << endl;  }  void Function(double \*p, int len) {  for(int i=0;i<len;i++)  cout << \*(p+i) << endl;  }  int Arr\_1[3] = {1,2,3};  float Arr\_1[3] = {1,2,3};  double Arr\_1[3] = {1,2,3};  Function(Arr\_1,3); // 編譯階段就決定這個 "Function" 是哪個 Function  Function(Arr\_2,3);  Function(Arr\_3,3); |

動態連結的例子

商店裡有「報稅」函式，繼承商店的餐廳又覆寫了一個「報稅」函式，那麼呼叫「報稅」函式時，究竟該對應到兩者中的哪一個，則是在執行階段才決定。

**第二節：名詞定義與抽象資料型別**

1. 物件導向相關的名詞定義

接下來，我們來定義一些物件導向裡的常見名詞：



(1) 類別 Class

A. 記載物件內部含有哪些成員

B. 描述物件是如何被架構出來的

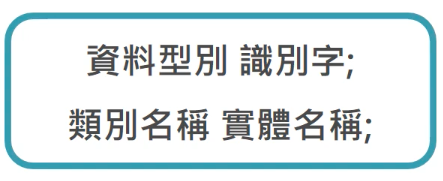
C. 相當於物件的「規格書」

D. 可被視作一種資料型別

(2) 實體 Instance

A. 根據類別 Class 開出的物件變數

B. 大部分操作都必須對實體（Instance，而非對物件）操作



宣告一個整數變數時使用的語法如：int a。

其中 int 是類別，a 就是依據 int 類別建立的實體。

|  |  |
| --- | --- |
| C++ 的檔案處理：ifstream 這個類別開出一個實體 myfile | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <iostream>  #include <fstream>  using namespace std;  int main(){  string str;  ifstream myfile; // 以 ifstream「類別」，開出「實體」myfile  myfile.open("example.txt"); // 接下來操作的對象是 myfile  myfile >> str;  cout << str << endl;  return 0;  } |

上面的程式碼中，ifrstream 是「類別」，沒有自己的資料；myfile 才是「實體」，資料存在記憶體中，操作自然應該針對「實體」myfile 操作。

(3) 屬性 Attribute

A. 類別所擁有的變數/資料成員

B. 儲存每一個物件實體的狀態

C. 不同物件實體的屬性可以有不同的值

D. 類似於結構（Structure）底下的成員



實體

實體

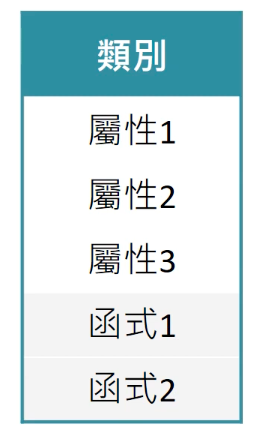
上圖中，字串 name、整數 ID、整數 score 都是屬性。其中一個實體的 name 屬性其值為 David，代表 David 這個人的資料。

name 為 David 的實體，其 ID 屬性值為 1，而 name 為 John 的實體，其 ID 屬性值則為 2，兩者的 ID 屬性有不同的值。

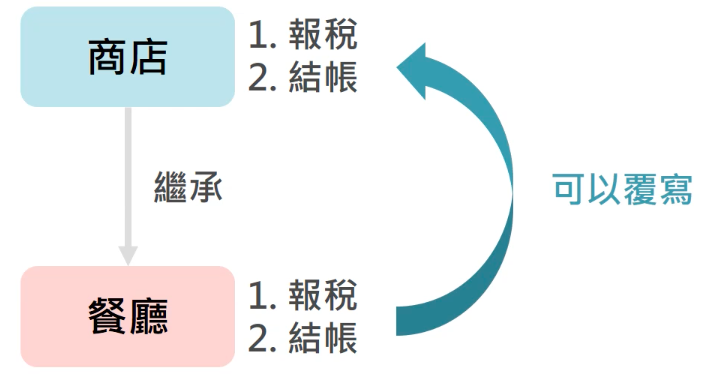
(4) 方法 Function

A. 方法是物件底下的函式或運算子

B. 不同的傳入值（訊息）對應到不同方法，即函式多載

 C. 通常透過方法去改變屬性（而非讓外界直接修改屬性）

類別底下有兩大類成員，一類是上面提到的屬性，另一類就是「方法」，即「函式成員」。

(5) 覆寫 Override：子類別重新定義父類別函式的功能

子類別繼承父類別之後，如果不希望直接使用父類別的某些函式，可以透過「覆寫」改變子類別透過繼承而來的函式行為。例如：商店類別有兩個函式成員「報稅」和「結帳」，餐廳類別繼承了商店類別，因此同樣有「報稅」和「結帳」兩個功能。

假設商店的「報稅」功能不適用於餐廳，比如餐廳可以不填報某些營業收入，那麼餐廳的「報稅」就不應該完全與商店的「報稅」相同，為了方便修改餐廳的「報稅」功能，我們可以進行「覆寫」。

(6) 訊息 Message

A. 物件間互相呼叫時溝通的工具

B. 物件收到訊息後，會執行相對應的方法

C. 多型的時候會用到，但與資料結構主題較不相關



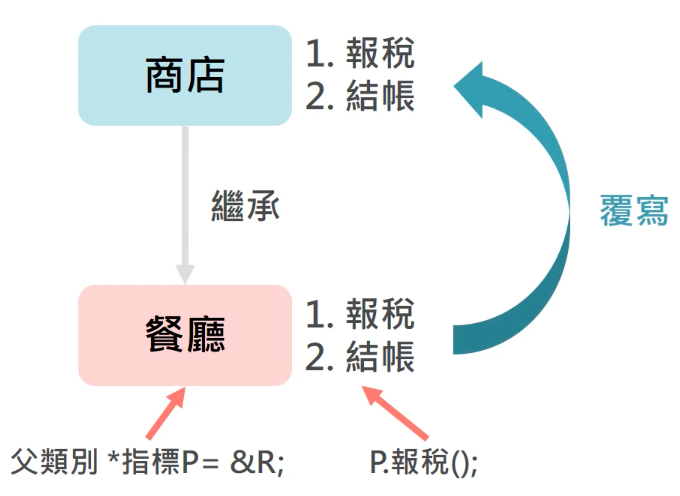
(7) 多型 Polymorphism

A. 透過父類別來操作子類別的方法

B. 多型三要素

a. 繼承

b. 覆寫父類別的方法

 c. 父類別指標指向子物件

宣告一個父類別指標 P 指向「餐廳」，此時 P 的「報稅」指向的是子類別

餐廳的「報稅」。因與資料結構課程無關，在此僅簡單帶過。

2. 資料型別抽象化

(1) 什麼是抽象？

回憶國中公民課提過的「國家」四要素：人民、領土、主權、政府。

上句中，「國家」一詞並不指向特定而具體的國家，而是代表一種「抽象」的概念：每一個符合「國家」概念的實體，都應該具有這四個要素。

A. 抽象概念：「國家」

B. 實體：美國、日本、韓國、…

(2) 抽象 Abstraction 的目的

抽象指的是「把目的與實作剝離開來」。

A. 函式抽象 Functional Abstraction：

許多程式篇幅龐大，使用者（除了研究編譯器的專家）不需要去追究當中

的每個函式，如 cin、cout 等是怎麼實作的。

一般來說，使用者只要知道一段程式碼的「目的」為何，知道可以怎麼使用這些函式即可，而不需搞懂底層的實現邏輯。

一個「堆疊」資料結構可能有特定操作方式，如「push」新增一筆資料、「pop」刪除一筆資料、「size」取得大小、…。要使用這些資料結構提供的操作方法，不要求瞭解實現的步驟，這就是「抽象」。同樣的，過去在學習陣列時，雖然知道陣列有哪些操作方式，但是不一定知道這些操作為何可以順利進行，這就代表陣列是「抽象」的。

把程式拆解成「標頭檔 .h」與「.cpp 檔」的目的，也是進行函式抽象，把宣告/介面放在標頭檔中，而把實作放在 cpp 檔。這樣一來，要知道有哪些方式可以進行操作，不需要看 cpp 檔，只需看標頭檔即可。

B. 資料抽象 Data Abstraction

幾乎所有語言都支援 process 抽象與資料型別抽象 data abstraction。

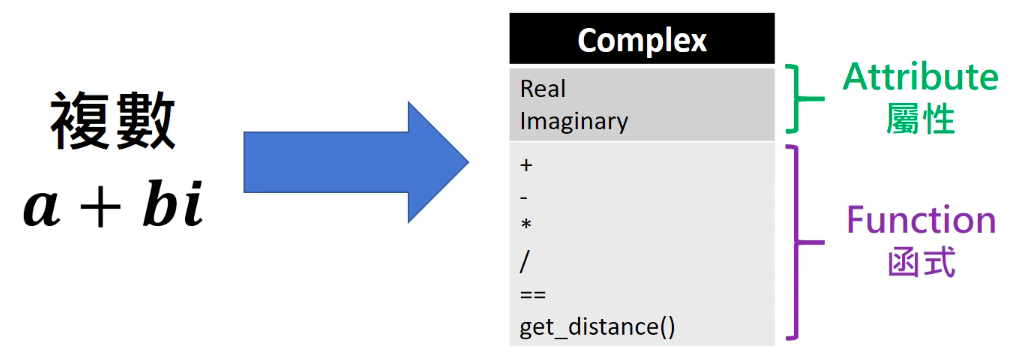
Abstract Data Type(ADT)

a. 由資料（Data）及資料間的操作（Operation）構成

b. 只要知道資料的特性，不需要知道操作是怎麼被實作出來的

c. 使用者只要會用，不需瞭解函式細部內容



在一個浮點數 float 中，資料被分成三個部分：「正負號」、「指數部分」和「小數部分」，而對一個浮點數可以進行的操作有「加減乘除」、「判斷是否相等」。即使我們不知道呼叫浮點數的「加法」、「乘法」時實際上是如何產生正確結果，還是可以順利使用，也就是說，浮點數是一個典型的「抽象資料型別」。

使用者也可以自定義抽象資料型別，比如上圖中我們可以寫出一個代表

「複數」的資料型別，形如 ，其中 是 。要處理這樣的複

數，可以自己寫出複數的資料型別，兩個屬性為「實部」a 和「虛部」b，另外還有一些加減乘除等操作可供使用。

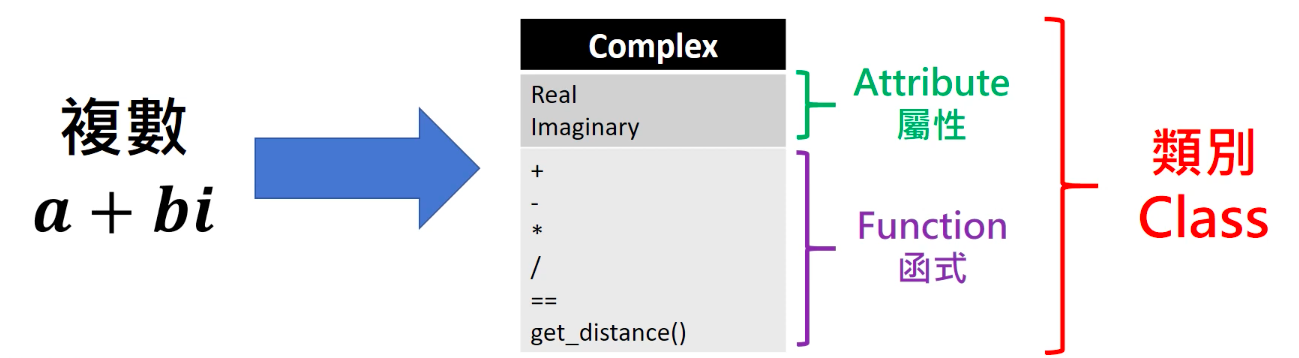
(3) 封裝 Encapsulation

A. 把資料與函式包在一起形成類別（Class）

B. 通常屬性（Attribute）不能被外界取用，只能透過函式存取

C. 資料封裝是物件導向程式設計的基本概念

D. 避免牽一髮而動全身的現象發生



(4) C++ 的抽象化實作

「類別Class」或「結構Structure」是使用者或開發者自行定義的資料型別，包含兩大類成員「資料」和「函式」，也就是「屬性」和「方法」。

「物件 Object」這個詞是「類別 Class」的「實體 Instance」，不過口語上有時會將「物件」與「類別」兩詞混同使用。

類別在記憶體中沒有自己的資料，是「規格書」，「物件實體」才有自己的資料。

int a;

float b;

double c;

上面的程式碼中，「int」、「float」與「double」是類別，可以操作的是 a、b、c等「實體」。

**第三節：類別的宣告與使用**

說明完物件導向的精神和一些名詞定義之後，接著就可以動手實作一個類別。

1. 類別宣告

物件導向程式設計的精神是「把程式以物件為基礎切割成一個個部份」。

設計物件時，第一個要考慮的就是「封裝」。這個物件根據程式需求，應有哪些屬性或函式，且應該如何封裝到物件裡。

「類別」就是這樣一個物件的規格書，記載了物件有哪些屬性與方法。

(1) 設計一個車子類別

A. 屬性：品牌

B. 屬性：引擎大小

C. 方法：發動引擎

D. 方法：煞車

設計一個車子類別時，第一個需要具有的屬性是「品牌」，看某輛車子是奧迪還是福特，第二個屬性可能是「引擎大小」，代表這是多大馬力的車子。描述了這台車後，可以用兩個函式來操作這個物件：「發動引擎」和「煞車」。

(2) 如何宣告類別

類別宣告的寫法與結構 structure 類似。首先寫出 class 類別名稱，後面接大括號並放入屬性和對應的資料型別。

|  |  |
| --- | --- |
| 宣告類別底下的屬性 | |
| 1  2  3  4  5 | class 類別名稱{  資料型別 屬性1;  資料型別 屬性2;  資料型別 屬性3;  }; |

宣告出實體的方法：

類別名稱 實體名稱;

操作大部分都是對實體進行，static 除外（本門課不提）。

(3) 長方形 rectangle 類別

下面是一個「長方形」類別，類別下有兩個屬性「寬 width」和「高 height」，另外有兩個函式：area 可以算出面積，set\_len 可以設定目前的寬和高。

|  |  |
| --- | --- |
| class rectangle | |
| 1  2  3  4  5  6  7  8  9  10 | class rectangle{  float width;  float height;  public:  float area();  void set\_len(float,float);  };  // 設計完 rectangle 類別後，就可以宣告出 rectangle 實體 R 來使用  rectangle R; // 類別名稱 rectangle 實體名稱 R |

(4) 物件的使用

一旦宣告出一個物件實體之後，可以透過「.」來使用或存取該方法或屬性，類似 C 語言的結構。

實體名稱.成員; // 取出該實體下的屬性或方法

若為物件指標，則是透過「->」來存取。

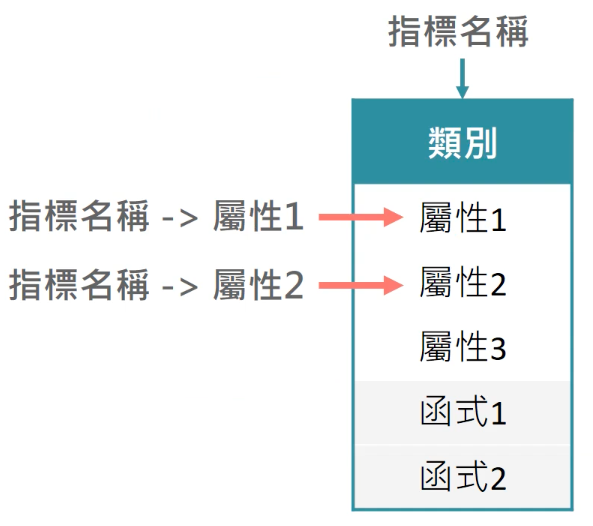
類別名稱 \*指標名稱 = &實體; // 宣告指向類別的指標

// \* 代表宣告指標，&取出實體的記憶體位置

(\*指標名稱).成員; // 第一種取出物件指標下成員的方法

指標名稱 -> 成員; // 第二種方法

|  |  |
| --- | --- |
| 使用 rectangle 實體的成員 | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class rectangle{  float width;  float height;  public:  float area();  void set\_len(float,float);  };  rectangle R;  R.set\_len(2.5,3.5); // 使用「.」設定 rectangle 實體 R 的寬和高  rectangle\* p = &R; // p是一個指向 rectangle 實體 R 的指標  cout << p->area << endl; // p->area：把 p 指到的 R 底下的成員取出來 |
| 執行結果 | |
| 8.75 | |



上圖中，有一個指標指向特定的物件（實體），「指標名稱 -> 屬性1」指的就是「把指標指到的物件，這個物件的屬性1取出」。

2. 定義類別下的方法

類別底下的成員主要分為兩大類

A. 屬性：描述狀態的變數

B. 方法：操作用的函式，可改變物件的狀態

最簡單的寫法，是把函式直接寫在類別底下：

|  |  |
| --- | --- |
| rectangle 類別 | |
| 1  2  3  4  5  6  7 | class rectangle{  float width;  float height;  // 回傳長方形的面積 area  float area(){  return width\*height;  };  }; |

但是如果把函式全部寫在類別裡，類別宣告會又臭又長。

類別裡函式的定義預設為 inline，一般情況下，盡量不要像下方一樣把方法都寫在類別的大括號裡面。

|  |  |
| --- | --- |
| rectangle 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class rectangle{  float width;  float height;  float area(){  return width\*height;  };  void setLen(float w, float h){  width = w;  height = h;  }  rectangle(){  width = 0;  height = 0;  }  }; |

習慣上，我們會把函式的宣告與定義分開：宣告放在類別宣告裡，定義則放在類別宣告外。

|  |  |
| --- | --- |
| 函式的宣告 | |
| 1  2  3 | class 類別名稱{  函式宣告;  }; |

因為把函式定義在類別外面，所以要使用「::」運算子向編譯器指明是哪個類別底下的哪個函式。

|  |  |
| --- | --- |
| 函式的定義 | |
| 1  2  3  4  5  6  7  8 | class 類別名稱{  函式宣告;  };  資料型態 類別名稱::方法名稱（引數1,引數2,...,引數n）  {  程式碼;  } |

修改過的 rectangle 類別寫法：

|  |  |
| --- | --- |
| rectangle 類別 | |
| 1  2  3  4  5  6  7  8 | class rectangle{  float width;  float height;  float area();  void setLen(float, float);  rectangle();  };  … // 把上面三個函式的定義寫在這裡 |

|  |  |
| --- | --- |
| area()：回傳面積的函式 | |
| 1  2  3 | float rectangle::area(){  return width\*height;  }; |

|  |  |
| --- | --- |
| setLen()：設定寬跟高的函式 | |
| 1  2  3  4 | void rectangle::setLen(float w, float h){  width = w;  height = h;  } |

|  |  |
| --- | --- |
| rectangle()：建構式 | |
| 1  2  3  4 | rectangle::rectangle(){  width = 0;  height = 0;  } |

記得在 rectangle 類別外面定義函式時，要加上「rectangle::」。

把函式的宣告（也就是介面）和定義（也就是實作）分開後，要知道 rectangle 有哪些特色，只要看 class 中的宣告就可以了，不需要特別去看這些方法是怎麼實作的。

3. 創建 Person 類別

創建一個人的類別 Person，使其包含以下成員

A. 姓名 name

B. 身高 height

C. 體重 weight

在人的類別中新增以下函式，設定跟回傳 3 個屬性

A. set\_name：設定姓名

B. get\_name：取得姓名

C. set\_height：設定身高

D. get\_height：取得身高

E. set\_weight：設定體重

F. get\_weight：取得體重

G. print：將人下面的所有成員印出來

|  |  |
| --- | --- |
| Person 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #include <iostream>  using namespace std;  class Person{  string name; // 姓名  float height; // 身高  float weight; // 體重    // 宣告方法（定義則放在類別外）  void set\_name(string);  string get\_name();  void set\_height(float);  float get\_height();  void set\_weight(float);  float get\_weight();  void print();  };  … // 把函式的定義寫在這裡  int main()  {  return 0;  } |

把函式定義寫在類別外時，寫法與一般函式大致一樣，但要記得在「回傳資料型別」與「函式名稱」之間加上「Person::」代表現在定義的是 Person 類別下的函式：

|  |  |
| --- | --- |
| set\_name：設定 name 屬性 | |
| 1  2  3 | void Person::set\_name(string n){  name = n; // 把 name 的值設定為 n  } |

|  |  |
| --- | --- |
| get\_name：回傳 name 屬性 | |
| 1  2  3 | string Person::get\_name(){  return name; // 回傳 name 的值  } |

|  |  |
| --- | --- |
| set\_height：設定 height 屬性 | |
| 1  2  3 | void Person::set\_height(float n){  height = n;  } |

|  |  |
| --- | --- |
| get\_height：取得 height 屬性 | |
| 1  2  3 | float Person::get\_height(){  return height;  } |

|  |  |
| --- | --- |
| set\_weight：設定 weight 屬性 | |
| 1  2  3 | void Person::set\_weight(float n){  weight = n;  } |

|  |  |
| --- | --- |
| get\_weight：取得 weight 屬性 | |
| 1  2  3 | float Person::get\_weight(){  return weight;  } |

|  |  |
| --- | --- |
| print：印出 Person 下所有的屬性值 | |
| 1  2  3  4  5 | void Person::print(){  cout << "Name: " << name << endl;  cout << "Height: " << height << endl;  cout << "Weight: " << weight << endl;  } |

之所以要用這麼多函式來取用屬性成員，是因為等一下設定權限的時候，所有的屬性都會設定成「私有」的，外面的人不能直接取得；另外，也是為了讓讀者練習撰寫類別中的方法。

4. 創建 Course 類別

創建一個課程類別，使其包含以下成員

A. 教師姓名

B. 學生人數

C. 成績列表（使用指標）：一個陣列，長度是學生人數，但值尚未確定

在課程類別中新增以下函式：

A. set\_lecturer：設定教師姓名

B. get\_lecturer：取得教師姓名

C. set\_student\_number：設定學生人數

D. get\_student\_number：取得學生人數

E. set\_grade\_list：設定成績列表

F. get\_grade\_list：取得成績列表

G. average：算出這個課程的平均成績

|  |  |
| --- | --- |
| Course 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <iostream>  using namespace std;  class Course{  string lecturer;  int student\_number;  float \*score;  void set\_lecturer(string);  string get\_lecturer();  void set\_student\_number(int);  string get\_student\_number();  void set\_grade\_list(float\*);  float\* get\_grade\_list();  float average();  };  … // 把函式的定義寫在這裡  int main()  {  return 0;  } |

|  |  |
| --- | --- |
| set\_lecturer：設定教師姓名 | |
| 1  2  3 | void Course::set\_lecturer(string name){  lecturer = name; // 設定 lecturer 的值  } |

|  |  |
| --- | --- |
| get\_lecturer：取得教師姓名 | |
| 1  2  3 | string Course::get\_lecturer(){  return lecturer; // 回傳 lecturer 的值  } |

|  |  |
| --- | --- |
| set\_student\_number：設定學生人數 | |
| 1  2  3 | void Course::set\_student\_number(int n){  student\_number = n;  } |

|  |  |
| --- | --- |
| get\_student\_number：取得學生人數 | |
| 1  2  3 | string Course::get\_student\_number(){  return student\_number;  } |

|  |  |
| --- | --- |
| set\_grade\_list：設定成績列表 | |
| 1  2  3 | void Course::set\_grade\_list(float\* p){  score = p;  } |

|  |  |
| --- | --- |
| get\_grade\_list：取得成績列表 | |
| 1  2  3 | float\* Course::get\_grade\_list(){  return score;  } |

|  |  |
| --- | --- |
| average：算平均數 | |
| 1  2  3  4  5  6  7  8  9  10 | float Course::average(){  float sum = 0;  // 用 for 迴圈加總所有成績  for(int i=0;i<student\_number;i++){  sum += \*(score+i);  }  // 把成績的和 sum 除以學生人數算出平均  sum /= student\_number;  return sum;  } |

接下來要來設定這些函式的權限，之後就可以實際操作這些類別了。

**第四節：類別的權限設定**

有了類別的權限設定之後，我們才能實際使用類別的實體。

1. 為什麼要有權限設定？

(1) 降低物件與介面的複雜度

A. 透過封裝來達到資訊隱藏，限制外界使用

B. 減少物件間的互相干擾

C. 確保修改物件時，不會牽一髮動全身

如果班上有一個邊緣人消失了，你很可能不會發現，但如果班上有個夯哥

消失了，他跟每個人連結很緊密，大家都會受到影響，因此我們希望盡量

減少物件與物件間的互相干擾。

然而，實際上不可能達到零耦合。零耦合指的是一個物件完全不會影響到

其他物件，但是既然物件完全不跟其他物件溝通，也就沒有存在的必要。

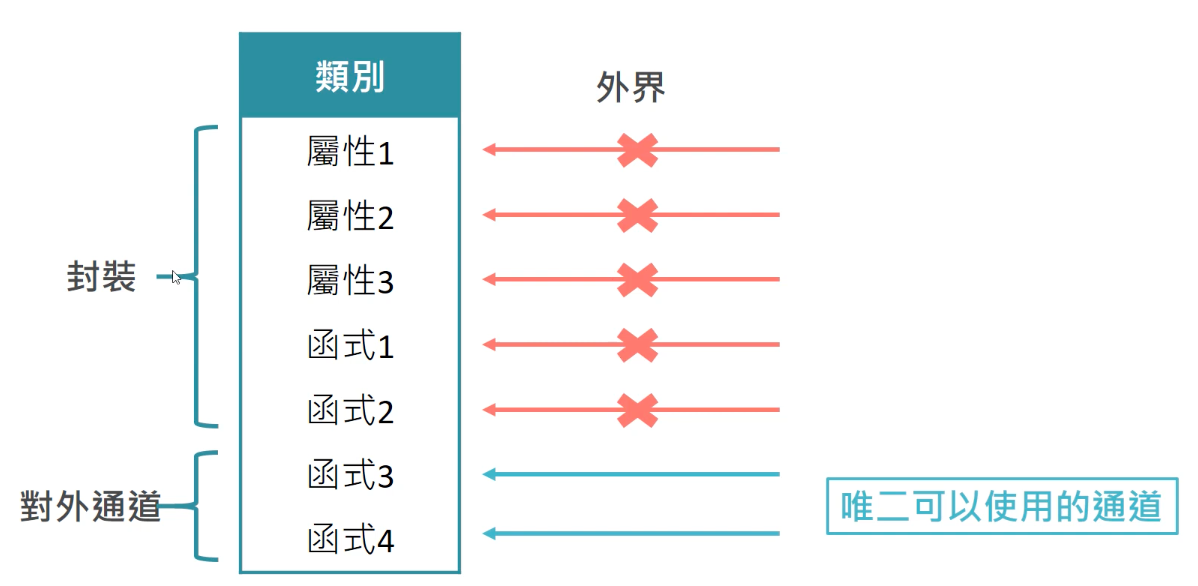
(2) 小而美的對外通道

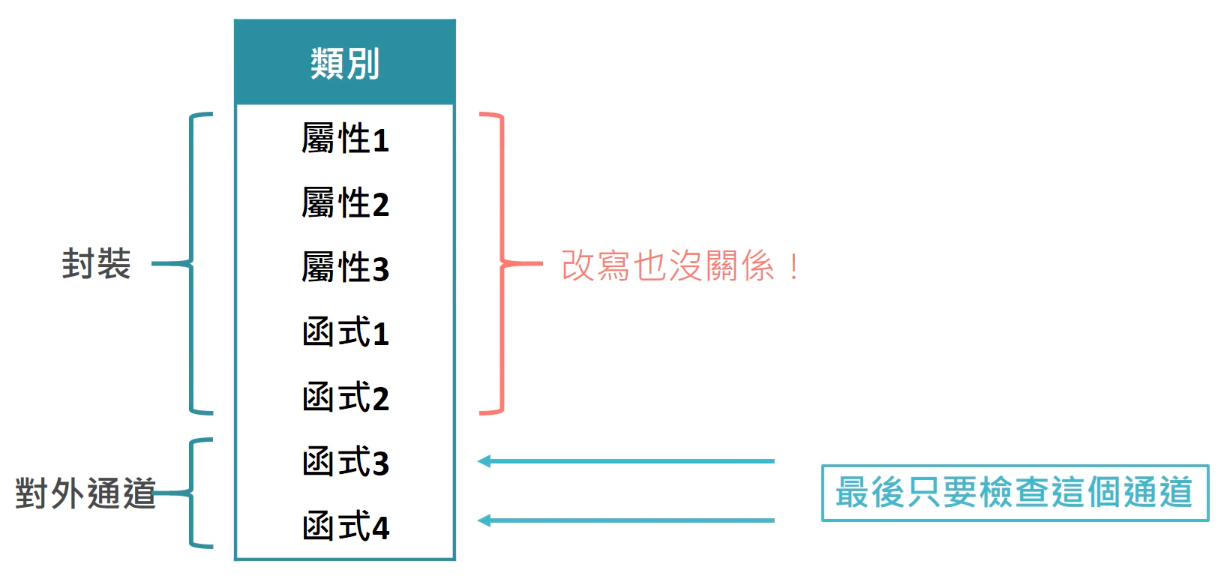
小：只開啟必要的對外通道，預設是關閉的

美：方便使用者使用

(3) 原則上所有的屬性都應該被封裝起來

就像對於家裡的房間來說，牆壁就是封裝，從房間要到外部，必須要通過

為了與外界聯繫而特別開出的門。



被封裝起來的成員不會被外部取用，因此若有修改時，只要檢查作為對外

通道的「函式3」與「函式4」即可。

(4) 物件封裝的目標：高內聚 + 低耦合

A. 複雜的運算只留在物件內部

B. 供外界呼叫使用的通道是少量且單純的



電腦硬體也符合高內聚跟低耦合的特性：記憶體如果壞掉，把記憶體拔

掉，換一個新的記憶體上去就好。

雖然記憶體和主機板內部的運算各自都很複雜，有很多輸入輸出、電路，

但要讓兩者順利溝通，只要通路（插槽）對的起來就好，複雜的運算都被

包在物件內部。

2. 類別的權限設定

我們只介紹兩種權限，另一個繼承才會用到，因此在本書裡不會使用。

A. public（公開成員）：可以透過物件名稱被外界呼叫

B. private（私有成員）

a. 不可以透過物件名稱被外界呼叫

b. 類別封裝的原則，理想上所有屬性都應被設定為 private

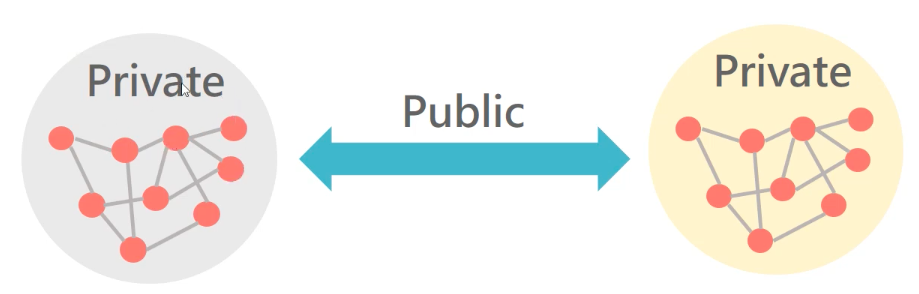
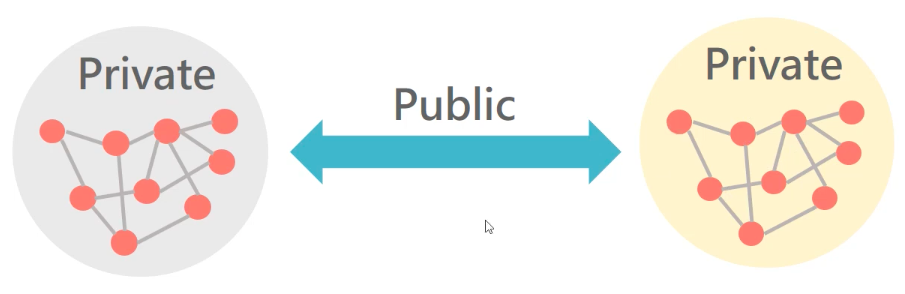
c. 能不公開就不公開

d. 取得（get）或設定（set）屬性盡量透過方法來進行

e. 最小化公開的資訊，只公開給外界呼叫的部分

f. 如果沒有寫權限，預設為「private」

一個物件就像一個家庭一樣，只有家人才能取用家裡的東西（私有成員）。

(1) 高內聚 + 低耦合

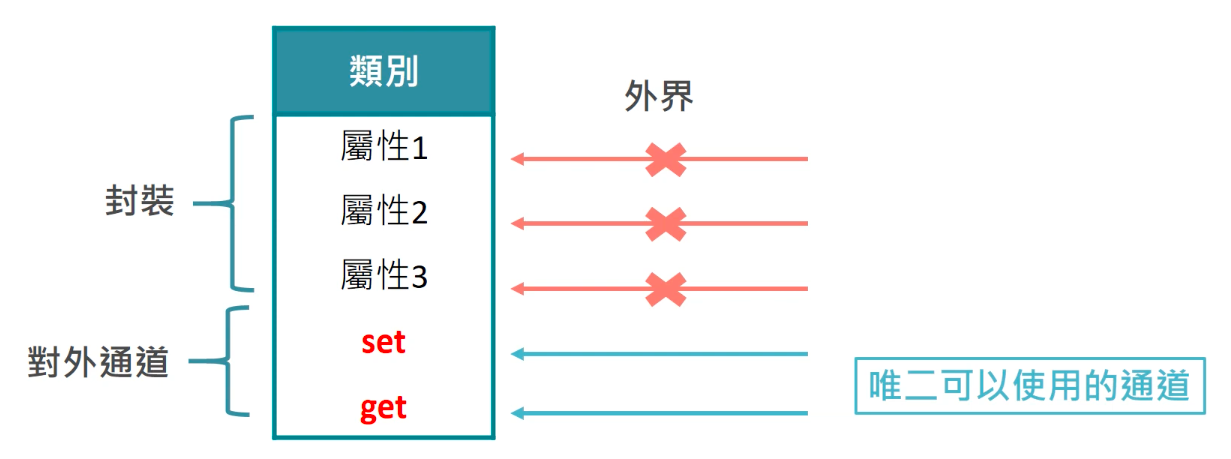
物件內部的屬性或函式如果只有內部會用到的話，就應設定成 Private，物件內部的運算很複雜也沒關係，但是物件和物件間溝通的通道 Public 成員則應該越單純越好。

(2) Private 和 Public 的不同功能

A. 透過 Private 來達到資訊隱藏

B. 透過 Public 創建給外界用的通道

|  |  |
| --- | --- |
| 改寫 rectangle 類別加上權限設定 | |
| 1  2  3  4  5  6  7  8  9 | class rectangle{  private: // 可省略，因預設值為 private  float width;  float height;  public: // 只有 public 成員可以被外界呼叫  float area();  void setLen(float, float);  rectangle();  }; |



|  |  |
| --- | --- |
| 測試 rectangle 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main(){  float w,h;  cout << "Please enter width and height:";  cin >> w >> h;  rectangle t;  // Public 成員能夠直接在 main 裡被呼叫  t.setLen(w,h);  // Private 成員不能直接在 main 裡被呼叫  ~~t.width = w;~~  ~~t.height = h;~~  } |

原則上將所有的資料成員都設定成 private 來達到資訊隱藏，若有存取需求，統一透過 public 函式來操作（set、get）。

(3) friend

private 成員原則上不能被物件外部呼叫，只限於物件內部間呼叫。但有時候，我們會希望特定函式能夠取用私有成員，這該如何達成呢？

「friend」關鍵字就像一張邀請函，讓 private 成員可被外部函式呼叫，函式（function）、類別（class）都可被設定為 friend。

class 類別名稱{

friend 函式;

friend 類別名稱;

};

因為這些函式和類別實際上不屬於類別內部，前面不需加上權限。

把外部函式設定為 friend，就像告訴編譯器：這個外部函式是我的好朋友，可以讓它取用私有成員。

通常使用 friend 關鍵字是基於效率考量，讓外部函式可以直接存取私有成員，而非間接透過公開函式存取，減少函式呼叫並使得效率較高。

(4) 實際使用 friend

在 rectangle 中使用兩個外部函式（判斷方式：前面不加 rectangle::）

A. area：回傳面積

B. setLen：設定 rectangle 的寬和高

因為寬跟高設定成 private，所以要使用 friend 關鍵字把這兩個外部函式作為例外，使它們可以取用 rectangle 中的私有成員 width 和 height。

|  |  |
| --- | --- |
| 外部函式 area 與 setLen | |
| 1  2  3  4  5  6  7 | class rectangle{  friend float area(rectangle);  friend void setLen(rectangle);  private:  float width;  float height;  }; |

|  |  |
| --- | --- |
| 外部函式 area | |
| 1  2  3 | float area(rectangle& r){  return r.width\*r.height;  } |

|  |  |
| --- | --- |
| 外部函式 setLen | |
| 1  2  3  4 | void setLen(rectangle& r, float w, float h){  r.width = w;  r.height = h;  } |

也可以把一個特定類別設定成 friend，設定後就可以在 ruler 類別底下取用 rectangle 的私有成員：

|  |  |
| --- | --- |
| 在 rectangle 類別中把 ruler 類別設成 friend | |
| 1  2  3  4  5  6 | class rectangle{  friend class ruler;  private:  float width;  float height;  }; |

|  |  |
| --- | --- |
| ruler | |
| 1  2  3  4 | class ruler{  public:  void print(rectangle);  } |

|  |  |
| --- | --- |
| print：印出 ruler 類別的屬性值 | |
| 1  2  3  4 | void ruler::print(rectangle& r){  // 可以取用 rectangle 的私有成員  cout << "Width: " << r.width << endl;  cout << "Height: " << r.height << endl;  } |

3. 建構式與解構式

(1) 什麼時候會使用到建構式？

如果未手動初始化，物件屬性的初始值會是亂數值，就像變數沒有初始化時一樣：

|  |  |
| --- | --- |
| 印出未初始化的 Person 屬性 | |
| 1  2  3  4  5  6  7  8 | class Person(){…};  …  int main()  {  Person p;  p.print();  return 0;  } |
| 執行結果 | |
| Width: 1.12104e-044  Height: 6.86636e-044 | |

(2) 建構式的宣告

透過建構式可以給定實體的初始值。建構式與解構式都不需要回傳資料型別，權限則皆需設定成 public。

A. 建構式 Constructor

a. 物件被實體化時執行，通常用作「初始化」

b. 名稱：類別名稱()

B. 解構式 Destructor

a. 物件從記憶體被釋放時執行，通常用作「清理」

b. 名稱：~類別名稱()

|  |  |
| --- | --- |
| rectangle 建構式的宣告 | |
| 1  2  3  4  5  6  7 | class rectangle{  private:  float width;  float height;  public:  rectangle(); // rectangle 的建構式（與類別名稱相同）  }; |

在建構式中，可以把寬跟高都初始化成 0：

|  |  |
| --- | --- |
| rectangle 建構式 | |
| 1  2  3 | rectangle::rectangle(){  width = height = 0;  } |

另外，也可以帶入引數，根據使用者給的資料做建構：

|  |  |
| --- | --- |
| rectangle 建構式的宣告（帶引數） | |
| 1  2  3  4  5  6  7 | class rectangle{  private:  float width;  float height;  public:  rectangle(float, float); // 帶引數的建構式  }; |

|  |  |
| --- | --- |
| rectangle 建構式（帶參數） | |
| 1  2  3  4 | rectangle::rectangle(float w, float h){  width = w;  width = h;  } |

這種使用傳入參數的建構式本質上是一種「賦值 assign」，而不是真正的「初始化」。

(3) 建構式的使用方式

|  |  |
| --- | --- |
| 使用 rectangle 建構式 | |
| 1  2  3  4  5 | int main(){  rectangle rectangleInstance(3,5);  cout << rectangleInstance;  return 0;  } |
| 執行結果 | |
| Width：3  Height：5 | |

(4) 解構式的宣告與定義

解構式在實體從記憶體中被抹去時會被自動觸發。

當類別內有使用動態記憶體配置 malloc 的時候，更要記得寫解構式。

|  |  |
| --- | --- |
| rectangle 解構式的宣告 | |
| 1  2  3  4  5  6  7  8 | class rectangle{  friend ostream& operator<<(ostream&,rectangle);  private:  float width;  float height;  public:  ~rectangle();  }; |

|  |  |
| --- | --- |
| ~rectangle：解構式 | |
| 1  2  3 | rectangle::~rectangle(){  cout << "Bye" << endl;  } |

(5) 傳值呼叫與傳參考呼叫

A. 傳值呼叫 pass by value：當我們把值傳給函式的時候，把變數的值 copy 一

份丟進去

B. 傳參考呼叫 pass by reference：在參數列的參數前加上 & 運算子，會讓函式

內的變數與函式外的變數有同一個記憶體位置，是同一筆資料

為了更清楚兩種呼叫的效果，我們可以透過修改解構式，讓每次發生解構的時候就 cout 一次 "Bye"，這樣每次輸出都代表有一個實體被解構了。

|  |  |
| --- | --- |
| 傳值呼叫版 area | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class rectangle{  friend float area(rectangle);  private:  float width;  float height;  public:  ~rectangle();  };  rectangle::~rectangel(){  cout << "Bye" << endl;  }  float area(rectangle r){  return r.width\*r.height;  } |

如果使用傳值呼叫 pass by value，呼叫後事實上是另外跑出一個物件實體 rectangle，因此解構式會被呼叫兩次，印出兩次 "Bye"：

|  |  |
| --- | --- |
| 測試使用傳值呼叫 | |
| 1  2  3  4  5  6 | int main()  {  rectangle rectangleInstance(3,5);  cout << area(rectangleInstance) << endl;  return 0;  } |
| 執行結果 | |
| 15  Bye  Bye | |

第一個 Bye：area 被呼叫時產生的區域變數 r 被解構

第二個 Bye：main 函式裡的 rectangleInstance 被解構

如果使用傳參考呼叫 pass by reference 則不會在呼叫時新開出一個實體，也只會觸發一次解構式。

|  |  |
| --- | --- |
| 傳參考呼叫版 area | |
| 1  2  3 | float area(rectangle& r){  return r.width\*r.height;  } |

|  |  |
| --- | --- |
| 測試使用傳參考呼叫 | |
| 1  2  3  4  5  6 | int main()  {  rectangle rectangleInstance(3,5);  cout << area(rectangleInstance) << endl;  return 0;  } |
| 執行結果 | |
| 15  Bye | |

通常我們在傳遞參數給函式的時候，會使用傳參考呼叫 pass by reference 來避免時間和空間的浪費。

(6) 總結建構式與解構式

A. 建構式 Constructors

a. 常用來做資料成員的「初始化」或是配置記憶體空間

b. 當實體被宣告時自動觸發建構式

c. 沒有回傳資料型態、跟類別名稱同名

d. 可以有多種建構式，接收不同引數做建構（函式多載）

e. 類別一定有建構式，若未定義編譯器會自動產生

B. 解構式 Destructor

a. 常用來做「清理」、計數或記憶體釋放

b. 當物件的生命週期結束時觸發

c. 類別一定有解構式，若未定義編譯器會自動產生

4. 改寫 Person 類別

在 Person 類別中新增權限

A. 原則上所有的資料成員都應該被設定成私有

B. 特定設定給外界使用的函式可以設定成公開

C. 另外寫一個外部函式 print，並利用 friend 操作類別

D. 試著使用它

在 Person 類別中新增兩種建構式

A. 沒有任何引數的建構式

B. 帶有引數的建構式

a. 姓名

b. 身高

c. 體重

(1) 改寫 Person 類別

|  |  |
| --- | --- |
| Person 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Person{  friend void print(Person& p); // 外部函式 print  private: // 預設  string name;  float height;  float weight;  public: // 可以被外界取用  void set\_name(string)  string get\_name();  void set\_height(float);  float get\_height();  void set\_weight(float);  float get\_weight();  void print();  // 利用函式初始值，把兩種建構式結合成一個  Person(string="anonymous", float=0, float=0);  ~Person(); // 解構式  } |

|  |  |
| --- | --- |
| 外部的 print 函式，需要被設定成 friend | |
| 1  2  3  4  5 | void print(Person& p){  cout << "Name: " << p.name << endl;  cout << "Height: " << p.height << endl;  cout << "Weight: " << p.weight << endl;  } |

(2) Person 建構式與解構式的定義

|  |  |
| --- | --- |
| Person 建構式 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | /\*  Person::Person(){ // 使用者沒有傳入引數  name = "anonymous"; // 名稱設為「匿名」  height = 0; // 身高體重都設定為 0  weight = 0;  }  \*/  // 使用者給出引數的時候，就使用這個函式做建構  Person::Person(string n, float h, float w){  name = n;  height = h;  weight = w;  } |

|  |  |
| --- | --- |
| Person 解構式 | |
| 1  2  3 | Person::~Person(){  cout << "Object destructed!" << endl;  } |

(3) 測試 Person 類別

|  |  |
| --- | --- |
| 測試 Person 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | int main()  {  // 沒有傳入引數，會印出預設值 (anonymous, 0, 0)  Person p1;  // 印出傳入的引數  Person p2("Mick",168,60);  p1.print();  p2.print();  return 0;  } |
| 執行結果 | |
| Name：anonymous  Height：0  Weight：0  Name：Mick  Height：168  Weight：60  Object was destructed!  Object was destructed! | |

friend 就像「白名單」的概念，黑名單代表的是原則上大家都可以進行某個動作，只有名單上的人不行，白名單則與此相反，原則上大家都不能進行某個動作，只有名單上的人可以。

原則上外部函式都不能取用私有成員，只有設定成 friend 的外部函式可以，這就是「白名單」。

5. 改寫課程類別

在課程類別中新增權限

A. 原則上所有的資料成員都應該被設定成私有 private

B. 特定設定給外界的函式可以設定成公開 public

C. 另外寫一個外部函式 print，並利用 friend 操作類別

D. 試著使用它

在課程類別中新增兩種建構式：

A. 沒有任何引數的建構式

B. 帶有引數的建構式

a. 教師姓名

b. 學生人數

c. 成績指標

(1) 改寫 Course 類別

|  |  |
| --- | --- |
| Course 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Course{  friend void print(Course&); // 讓外部 print 函式可以取用私有成員  private: // 預設  string lecturer;  int student\_number;  float \*score;  public:  void set\_lecturer(string);  string get\_lecturer();  void set\_student\_number(int);  int get\_student\_number(int);  void set\_grade\_list(float\*);  float\* get\_grade\_list();  float average();  // float\* =0 要空格，不然會誤認為 \*=  Course(string="unknown",int = 0,float\* = 0); // 建構式  ~Course(); // 解構式  } |

|  |  |
| --- | --- |
| 外部 print 函式，盡量使用參考方式傳入 | |
| 1  2  3  4  5  6  7 | void print(Course& c){  cout << "Lecturer:" << c.lecturer << endl;  cout << "Student number:" << c.student\_number << endl;  // 利用 for 迴圈印出每個學生的成績  for(int i=0;i<c.student\_number;i++){  cout << i+1 << "-th:" << \*(c.score+i) << endl;  }  } |

(2) Course 建構式與解構式的定義

|  |  |
| --- | --- |
| Course 的兩種建構式 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\* Course::Course(){  lecturer = "unknown";  student\_number = 0;  score = 0;  } \*/  // 有引數的情況下，宣告後再 assign  Course::Course(string name, int number, float\* score\_list){  lecturer = name;  student\_number = number;  // 當 number 不是 0，但 score\_list 沒有值，開出新陣列  if(number!=0 && score\_list == 0){  score = (float\*)calloc(number, sizeof(float));  } else {  score = score\_list; // 一般情況，把 score\_list 指定給 score  }  } |

|  |  |
| --- | --- |
| Course 的解構式 | |
| 1  2  3  4  5 | Course::~Course(){  cout << "Course finished!" << endl;  // 因為用到建構式 calloc，沒有 free 掉會造成 memory leakage  free(score);  } |

(3) 測試 Course 類別

|  |  |
| --- | --- |
| 測試 Course 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main()  {  // Course math;  float score[5]={85,95,60,82,78};  Course math("LKM",5,score);  /\* math.set\_lecturer("LKM");  math.set\_student\_number(5);  math.set\_grade\_list(score); \*/  // 注意student\_number 是否還沒有初始化  print(math);  } |
| 執行結果 | |
| Lecturer：LKM  Student number：5  1-th：85  2-th：95  3-th：60  4-th：82  5-th：78  Course finished! | |

**第五節：重載運算子、模板與標頭檔**

1. 重載運算子

運算子指的是「+ - x / %」等符號，這些運算子的行爲是由人類定義出來的，包括加減乘除如何運作，都是由工程師決定。

C / C++ 支援運算子重載，但只能重載「自定義」的運算子，並不能改掉如 int + int 的運算方式。這是一種保護機制，否則其他人使用時可能出錯，所以包含 int float double boolean 等已有的資料型別都不能被重載。

(1) 運算子重載的宣告與定義

運算子重載是函式重載的延伸，因此宣告和定義的方式都與函式類似。差別在於函式名稱要改為「operator運算子符號」：

回傳資料型態 operator運算子符號(引數1,引數2){

...

...

return 回傳值;

}

透過運算子重載，能夠讓 rectangle 物件可以「乘以」某個數。一般來說，「長方形乘以某個數」是沒有定義的，也不能順利執行，所以需要手動寫出遇到 「rectangle 物件x N」時的處理方式。下面我們將其定義為該 rectangle 物件的長寬各伸縮 N 倍：

|  |  |
| --- | --- |
| rectangle 中 \* 運算子重載的宣告 | |
| 1  2  3  4  5  6  7  8  9  10  11 | class rectangle{  friend float area(rectangle&);  private:  float width;  float height;  public:  rectangle();  rectangle(float,float);  ~rectangle();  rectangle operator\*(float); // 運算子重載  } |

|  |  |
| --- | --- |
| rectangle 中 \* 運算子重載的定義 | |
| 1  2  3  4 | rectangle rectangle::operator\*(float r){  rectangle rec(width\*r,height\*r); // 開出一個新的 rectangle  return rec; // 回傳長寬都是原來 r 倍的 rectangle 物件  } |

|  |  |
| --- | --- |
| 測試 \* 運算子重載 | |
| 1  2  3  4  5  6 | int main()  {  rectangle rectangleInstance(3,5);  cout << rectangleInstance\*2 << endl;  return 0;  } |
| 執行結果 | |
| Width：6  Height：10 | |

(2) 輸入 / 輸出運算子重載

透過重載輸入 / 輸出運算子，可以教會編譯器「cout << rectangle物件」應該要進行什麼處理，讓自定義的物件能直接進行 cin / cout。

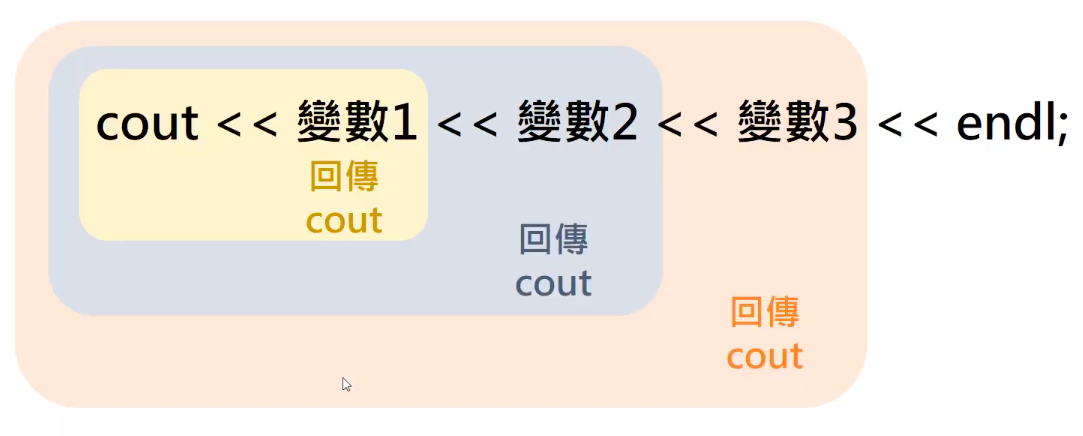
如果不做重載，編譯器並不知道應該如何正確處理「cout << rectangle物件」，所以會無法執行。

A. 重載 << 運算子

|  |  |
| --- | --- |
| rectangle 中 << 運算子重載的宣告 | |
| 1  2  3  4  5  6 | class rectangle{  friend ostream& operator<<(ostream&, rectangle&);  private:  float width;  float height;  }; |

|  |  |
| --- | --- |
| rectangle 中 << 運算子重載的定義 | |
| 1  2  3  4  5 | ostream& operator<<(ostream& os, rectangle& r){  os << "Width: " << r.width << endl;  os << "Height: " << r.height << endl;  return os;  } |

因為取用了 rectangle 中的 width 和 height 等私有成員，所以必須在 rectangle 類別中把運算子重載設為 friend。完成後，當我們寫「cout << rectangle物件」，編譯器就會按照設計輸出 rectangle 物件的寬和高。

另外，cin 是一個 istream（輸入串流），cout 則是一個 ostream（輸出串流），回傳 ostream 的理由是這樣才能連續輸出，否則一次只能 cout 一個變數，不能使用形如「cout << a << b << c」的語法。

上面這種語法能夠順利執行，是因為每次進行一次 cout 後，都會回傳一個 ostream，可以繼續對下一個變數做輸出。

B. 重載 >> 運算子

|  |  |
| --- | --- |
| rectangle 中 >> 運算子重載的宣告 | |
| 1  2  3  4  5  6 | class rectangle{  friend istream& operator>>(istream&,rectangle&);  private:  float width;  float height;  }; |

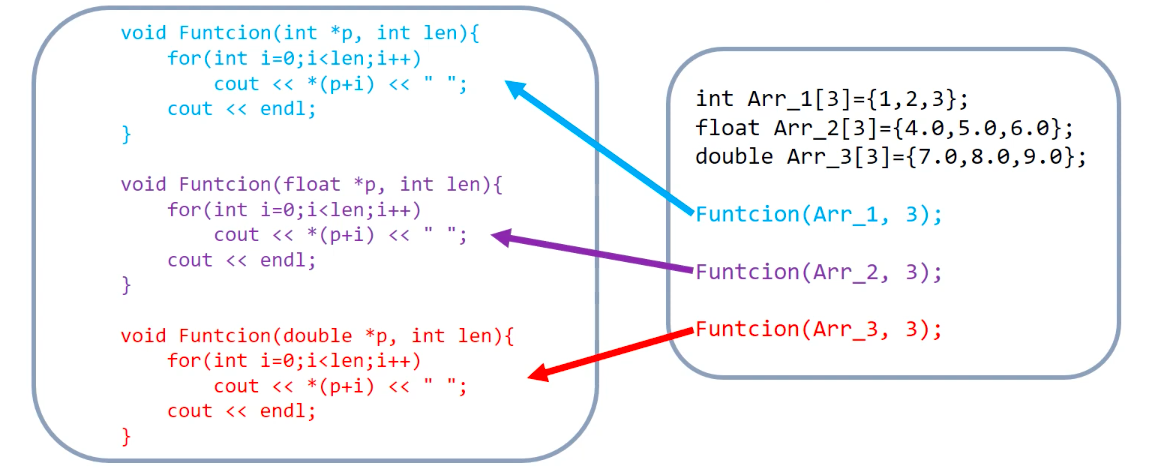
|  |  |
| --- | --- |
| rectangle 中 >> 運算子重載的宣告 | |
| 1  2  3  4 | istream& operator>>(istream& is, rectangle& r){  is >> r.width >> r.height;  return is;  } |

輸入和輸出運算子 >> 與 << 的箭頭方向其實是人為定義的，透過運算子重載，我們也可以將其調換，讓 << 變為輸入、>> 變為輸出，只是這樣容易造成混淆，因此並不建議。

2. 函式模板與類別模板

(1) 函式多載

A. 指的是同樣的函式名稱，但有不同的引數型態

 B. 引數型態不同，函式簽名也不同，可以同時存在

但是觀察上面的程式碼，每個函式的內容大同小異，差別只在輸入引數的

型態而已，違反了「Do not repeat yourself」的精神，有沒有辦法只寫一

次就達成同樣的效果？

(2) 模板 Template

A. 一般函式（Function）：每個函式都取不同的名稱

B. 函式重載（Function Overloading）：函式有相同的名稱，但有不同引數型態

C. 函式模板（Function Template）：把引數型態參數化，由編譯器根據參數產生

程式碼

(3) 函式模板 Function Template

A. 資料型態參數化：把資料型態變成變數

B. 以簡化的符號表示，通常是 T (typename), T1, T2, ...

C. 由編譯器透過類似巨集代換，依據內容產生程式碼

template<樣板參數型態 樣板參數名稱,...>

回傳資料型態 函式名稱（引數型態 引數名稱,...）

{

程式碼;

}

(4) 函式模板的實際例子

想實現「判斷 a 是不是大於 b」的函式 isBigger，但不想為 int、float、char 各寫一個類似的函式，把資料型別參數化（變成一個變數 T）：

template <class T>

bool isBigger(T a, T b){

return a>b;

}

A. 以關鍵字 template 開頭

a. 用 <> 把參數包起來

b. 習慣上用 T, T1, T2 作為模板參數名稱

B. 寫上函式的程式碼

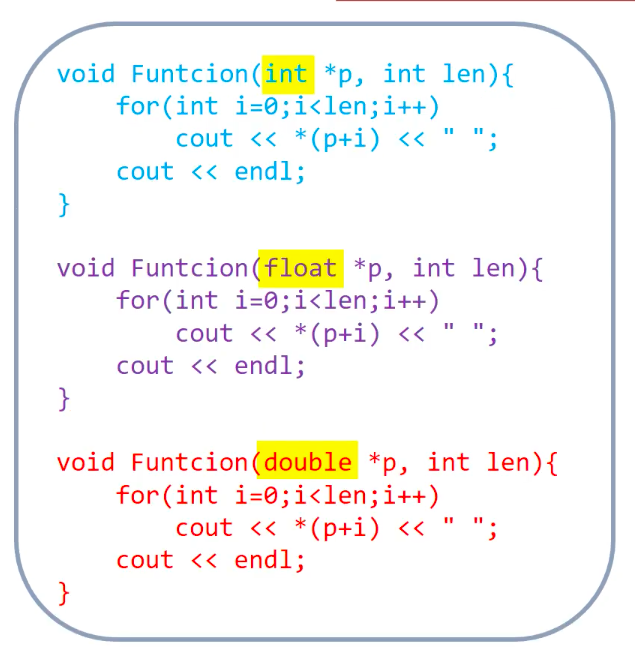
a. 又稱原型（prototype）

b. 函式名稱即為該函式樣版的名稱

參數型態可用 class 或 typename，表示泛用型態（即任何型態）。因此參數型態也可以代入如 int、float 或其他自定義的型態（結構等等），使用者傳入 int 時，T 就代入 int 並產生對應 int 的程式碼；使用者傳入 float 時，T 則代入 float 產生對應 float 的程式碼。

(5) 另一個函式模板的例子

|  |  |
| --- | --- |
| 利用函式模板印出陣列 | |
| 1  2  3  4  5  6 | template<typename T>  void Function(T \*p, int len){  for(int i=0;i<len;i++)  cout << \*(p+i) << " ";  cout << endl;  } |

使用者呼叫函式時如果傳入 int、float 或 double 型態的資料，會分別生成下列三個對應的函式：

(6) 類別模板 Class Template

A. 將資料型態參數化

B. 以簡化的符號表示，通常是 T (typename), T1, T2, ...

C. 由編譯器透過類似巨集代換，依據內容產生程式碼

template <樣板參數型態 樣板參數名稱,...>

class 類別名稱()

{

private:

資料成員;

public:

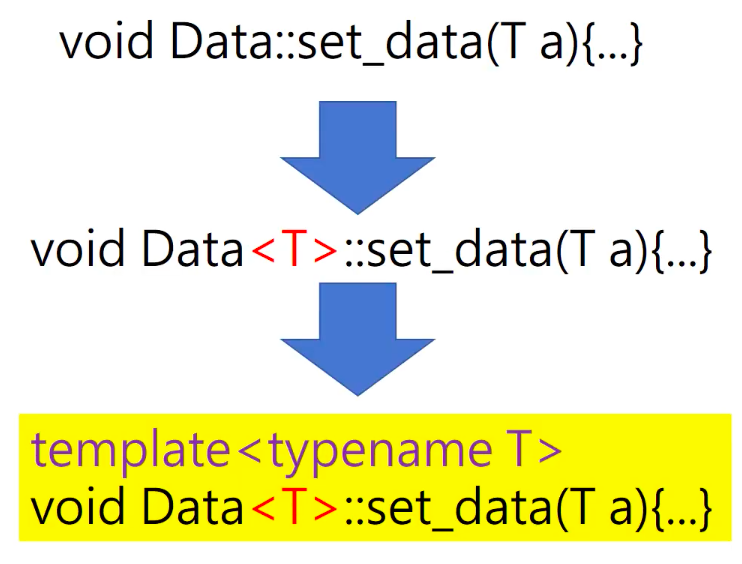
函式成員;

}

我們希望 Data 這個類別裡可以存不同型態的資料 int、float、double：

|  |  |
| --- | --- |
| 利用模板寫 Data 類別 | |
| 1  2  3  4  5  6  7  8 | template <typename>  class Data{  private:  T data;  public:  void set\_data(T);  T get\_data();  }; |

|  |  |
| --- | --- |
| 宣告不同資料型態的 Data 實體 | |
| 1  2  3  4  5  6 | int main(){  Data<int> i; // T=int  Data<float> f; // T=float  Data<double> d; // T=double  return 0;  } |

(7) 類別模板的宣告

上面是類別模板的宣告和定義方式，它的演進方式是

A. 回傳資料型別 類別名稱::函式名稱(引數){…}

B. 因為要指定資料型別，類別名稱要寫成「類別名稱<T>」

C. 寫明 T 的意義，有用到 T 的部分前面都要加上 template<typename T>

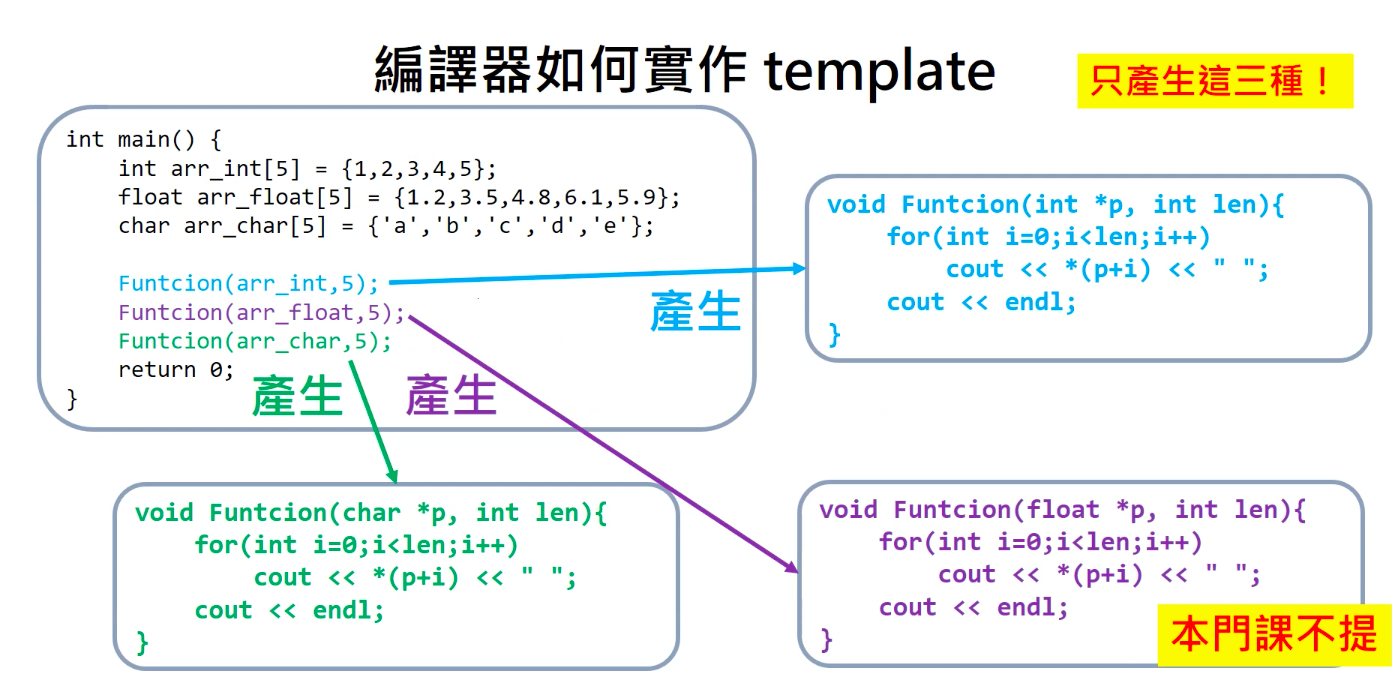
(8) 類別模板下的函式模板

|  |  |
| --- | --- |
| 類別模式下的函式模板 | |
| 1  2  3  4  5  6  7  8  9 | template <typename T> // 用類別模板寫 class Data  class Data{  private:  T data;  public:  void set\_data(T);  T get\_data();  // 在類別模板下，用函式模板寫 sum 函式  template<typename T2> T2 sum(T2, T2);  } |

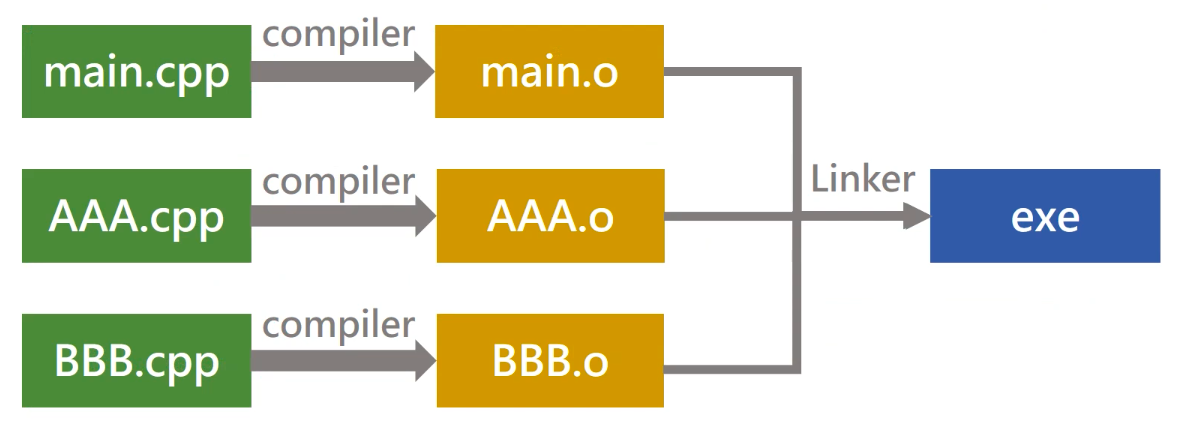
|  |  |
| --- | --- |
| 函式模板 sum 的定義 | |
| 1  2  3  4  5 | template<typename T> // 類別模板  template<typename T2> // 函式模板  T2 Data<T>::sum(T2 a, T2 b){  return a+b;  } |

函式模板 sum 可以用來計算兩個 int、兩個 float 或兩個 double 的和，因為 T 已經被用過了，函式的類別變數要換成使用 T2。

定義時，前面要依序加上 「template<typename T>」與「template<typename T2>」。

(9) 編譯器如何實作模板

傳入的引數是整數指標時，生成對應整數的程式碼；傳入的引數是浮點數時，則生成對應浮點數的程式碼。這些程式碼是在編譯階段生成，有呼叫到才會產生，未呼叫就不會產生對應類型的函式，像上圖中，就不會生成對應倍精度浮點數 double 的函式。

模板是在編譯階段生成，是否寫成 template，對執行檔的大小沒有影響，只是寫程式的時候比較精簡，也不會漏定義某個資料型態。

(10) 模板 Template 的注意事項

A. Template 的可視度(Scope)只有在 template 下方的「一個」函式或類別中

|  |  |
| --- | --- |
| Template 的可視度 | |
| 1  2  3  4  5  6  7  8  9 | template<typename T> // 模板一  T MIN(T a,T b){  return a<b?a:b;  }  template<typename T> // 模板二，不會看到模板一的 T  T MAX(T a,T b){  return a>b?a:b;  } |

B. Template 可以不換行

|  |  |
| --- | --- |
| 下面兩種寫法意義相等 | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | template<typename T>  void Function(T \*p, int len){  for(int i=0;i<len;i++)  cout << \*(p+i) << " ";  cout << endl;  }  template<typename T> void Function(T \*p, int len){ // 不換行意義相同  for(int i=0;i<len;i++)  cout << \*(p+i) << " ";  cout << endl;  } |

C. Template 與函式間不能有任何指令

|  |  |
| --- | --- |
| Template 與函式間不能有任何指令 | |
| 1  2  3  4  5  6  7 | template<typename T>  int a; // 錯誤！  void Function(T \*p, int len){  for(int i=0;i<len;i++)  cout << \*(p+i) << " ";  cout << endl;  } |

D. 找不到對應的函式時才會生成（候補選手），若程式碼中已有，則不會生成

|  |  |
| --- | --- |
| Template 與函式間不能有任何指令 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | template<typename T>  T sum(T a, T b){  cout << "Called by template function.";  return a+b;  }  int sum(int a, int b){  cout << "Called by normal function.";  return a+b;  }  int main(){  cout << sum(3,5) << endl; // 已有對應，直接用 sum(int, int)  cout << sum(1.8,2.7) << endl; // 沒有對應，透過模板生成  return 0;  } |
| 執行結果 | |
| Called by normal function.8 // 3 跟 5 為 int，因此輸出 "normal function"  Called by template function.4.5 // 1.8, 2.7 為 float，對應函式由 template 生成 | |

3. 標頭檔的建立

把函式拆解成宣告與定義兩個部分

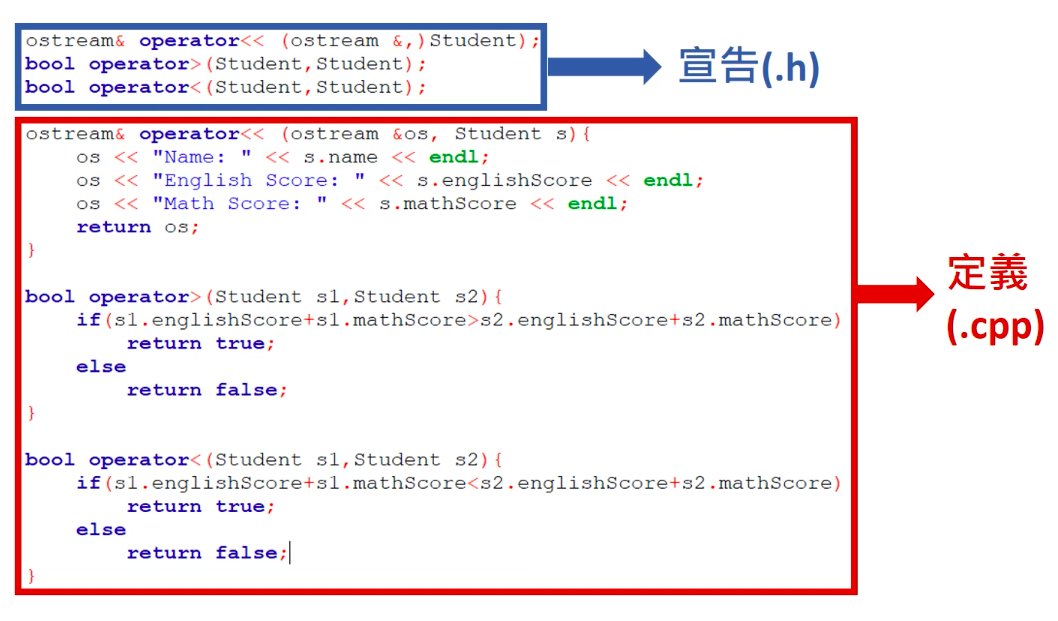
A. 宣告：函式的介面

B. 定義：函式的實作方式

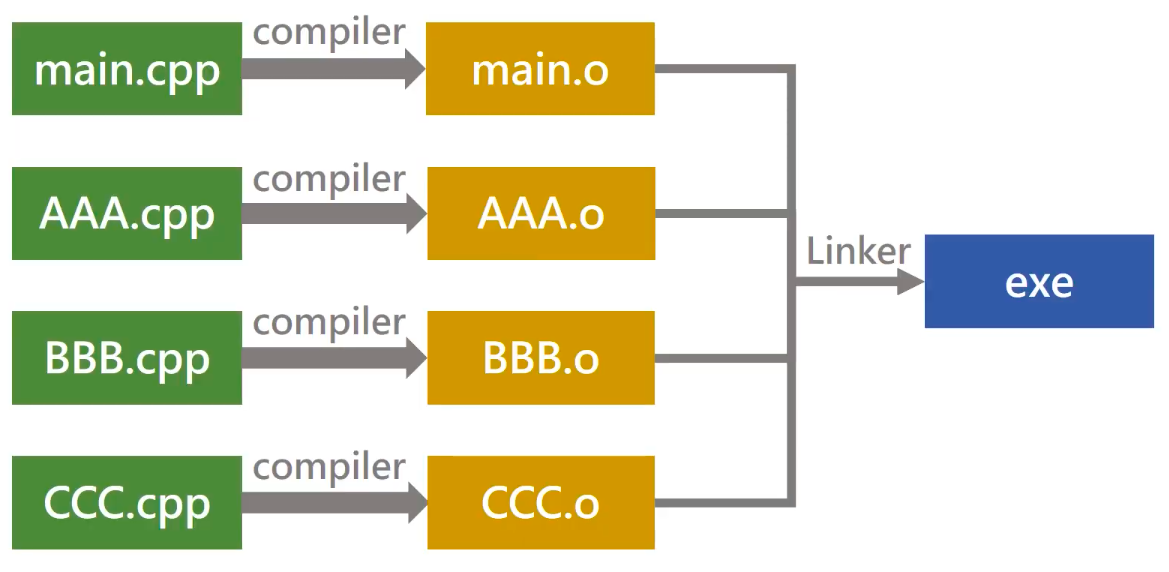
看完函式的宣告後，就知道可以如何操作了。

(1) 標頭檔與 cpp 檔

一般來說，宣告放在 header file(.h)，定義則放在 cpp file(.cpp)，需要用到時，利用「include header file」就可以使用。平常不需要知道函式如何實作，就像可以直接使用，而不需研究 cin、cout 是怎麼做的。



宣告通常只有一行：「資料型態 函式名稱/運算子符號(引數列);」；定義則是有大括號的程式碼部分。



原則上編譯的時候，每個 cpp 檔都會產生一個 .o 檔，最後所有 .o 檔再透過連結器 Linker 連接在一起，變成一個 exe 檔。

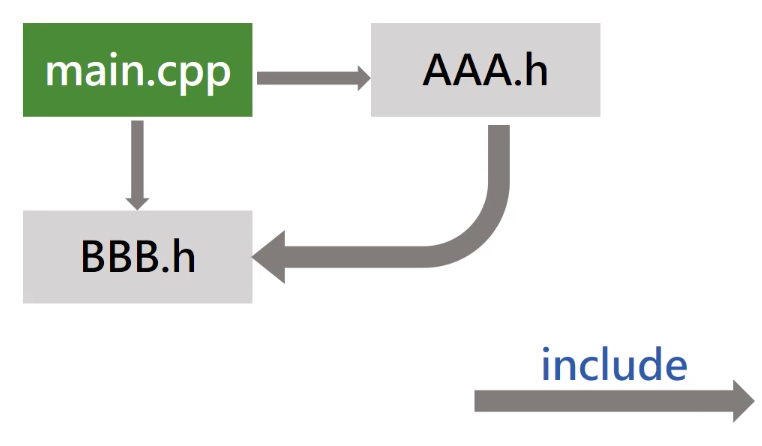
(2) 引用標頭檔

include 有兩種方式，"" 與 <> 找的路徑不同

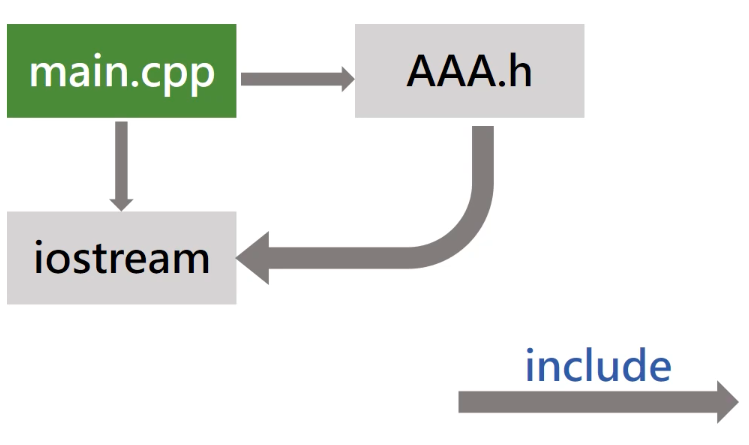
A. #include "file.h" 找的路徑是資料夾內

B. #include <file.h> 找的路徑則是編譯器 Compiler 的內部路徑內

(3) 使用 ifndef 避免重複宣告



上圖中，main.cpp 引入了 BBB.h，又引入了 AAA.h，而 AAA.h 裡也引入了 BBB.h。

這樣一來，main.cpp 是不是會引入兩次 BBB.h，造成重複宣告的問題？

最常見的例子是 iostream，因為許多檔案中都會引入 iostream。但是就算引入了多次 iostream，還是可以正常執行，不會有重複宣告的問題，這是因為其中有避免重複 include 的機制。

這個機制類似糖果屋的故事，一對兄妹為了避免迷路，在森林裡邊走邊撒麵包屑，以便之後知道這條路有沒有走過。利用類似的方法，我可以在 include 過一個標頭檔後，就紀錄類似「XXX 到此一遊」的訊息，避免重複。

#ifndef XXXXX\_H\_INCLUDED

#define XXXXX\_H\_INCLUDED

要宣告的函式

#endif

ifndef 是「if not defined」的縮寫，當第二次 include 到同一個標頭檔時，因為發現關鍵字已經被定義過了，就會直接跳過，避免重複宣告。

A. 第一次 include

a. 關鍵字沒有被定義過

b. #define 關鍵字

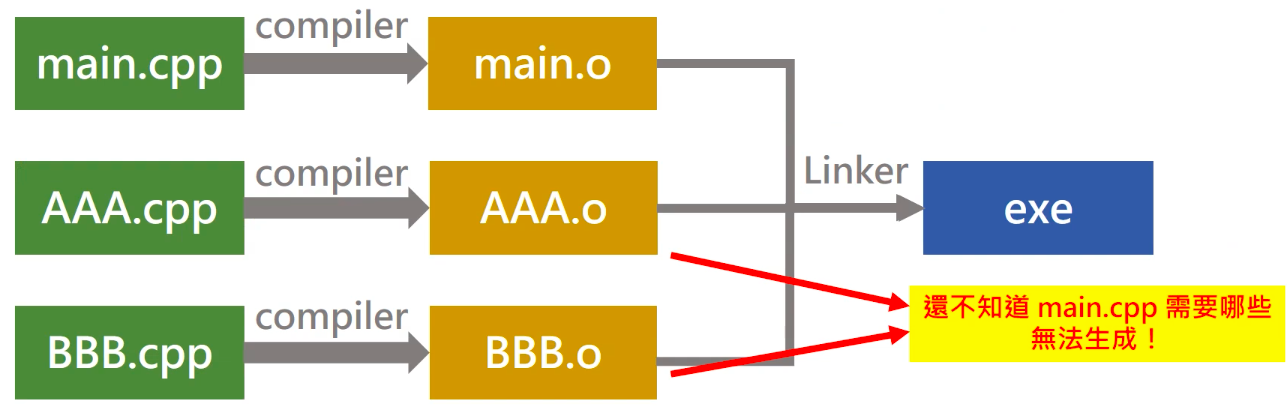
c. 宣告函式

B. 第二次 include

a. 關鍵字被定義過了（代表來過了）

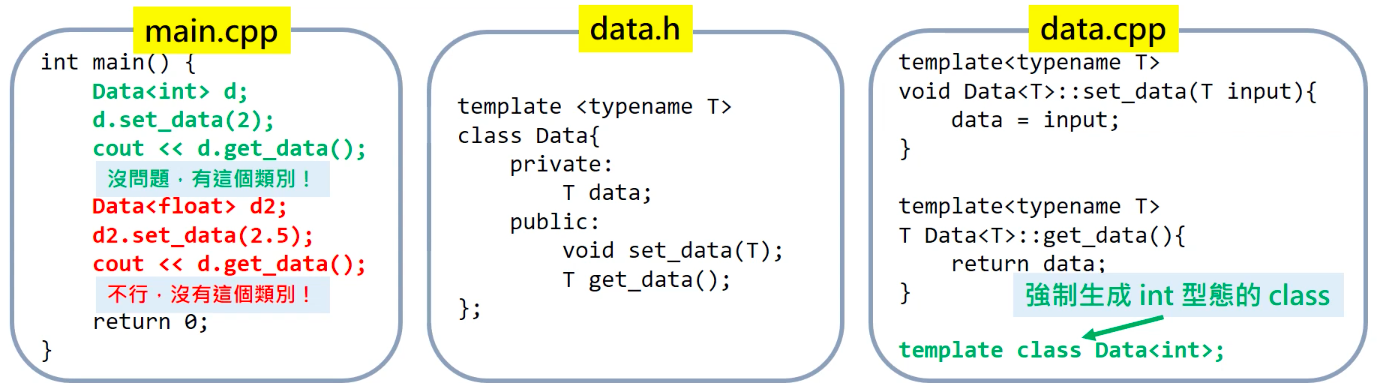
b. 跳到 endif 結束

很多情形下，這個機制會由編譯器自動生成。

(4) 模板要放在標頭檔中

雖然一般來說，宣告放在標頭檔 .h，定義放在 cpp 檔，但因為模板是在編譯階段生成，所以不能放在 .cpp 下，而應放在 .h 下！

上圖中，如果模板被放在 AAA.cpp 或 BBB.cpp 下，編譯階段並不會知道 main.cpp 傳入了哪些類型的資料，應該產生哪些對應的程式碼，所以模板應放在標頭檔裡。



如果模板放在 .cpp 檔而非標頭檔下，需要當下指定要生成哪些資料型別的函式。

比如 main.cpp 裡用到了 int，data.h 中有一個類別模板與函式的宣告，data.cpp 中則有函式的定義：main.cpp 編譯後會變為 main.o，data.cpp 則變為 data.o，因為data.cpp 不會知道 main.cpp 會用到哪些資料型態，所以只能在 data.cpp 中強制生成 int 型態的程式碼。

到這裡為止，我們已經可以寫完一個簡單的抽象資料型別 ADT

A. 有一組資料

B. 上面有一大堆操作

C. 在做操作時，不需要想是怎麼進行操作的

D. 開發者雖然需要花時間開發，但是使用者可以直接使用

4. 重載「Person類別」中的運算子：

A. 重載 << 和 >> 運算子，讓類別可以直接進行輸入輸出

B. 重載 + 運算子，將「Person + Person」定義為「回傳兩個人體重和」

C. 把 Person 類別改成模板，由使用者決定身高體重使用的資料型別

D. 切割成標頭檔 .h 和原始碼 .cpp，試著在 main.cpp 中呼叫這個類別

(1) 重載運算子

|  |  |
| --- | --- |
| Person 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <iostream>  using namespace std;  class Person{  friend void print(Person& p);  friend ostream& operator<<(ostream&, Person&);  friend istream& operator>>(istream&, Person&);  private:  string name;  float height;  float weight;  public:  void set\_name(string);  string get\_name();  void set\_height(float);  float get\_height();  void set\_weight(float);  float get\_weight();  void print();  Person(string="anonymous", float=0, float=0);  ~Person();  float operator+(Person&);  } |

|  |  |
| --- | --- |
| 重載 << 運算子 | |
| 1  2  3  4  5  6 | ostream& operator<<(ostream& os, Person& p){ // 傳入的是「參考」  os << "Name: " << p.name << endl;  os << "Height: " << p.height << endl;  os << "Weight: " << p.weight << endl;  return os;  } |

|  |  |
| --- | --- |
| 重載 >> 運算子 | |
| 1  2  3  4 | istream& operator>>(istream& is, Person& p){ // 傳入的是「參考」  is >> p.name >> p.height >> p.weight;  return is;  } |

|  |  |
| --- | --- |
| 重載 + 運算子 | |
| 1  2  3 | float Person::operator+(Person& p){ // operator+ 在 Person 類別內  // Person1 + Person2 -> 回傳 Person1 的 weight + Person2 的 weight  return weight + p.weight;  } |

|  |  |
| --- | --- |
| 測試 Person 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int main()  {  Person p1("Rallod",170,50);  Person p2("Mick",168,60);  print(p1);  print(p2);  // 測試 cin、cout、+ 運算子  cin >> p1;  cout << p2;  cout << p1+p2 << endl;  return 0;  } |

(2) 修改為 Template

|  |  |
| --- | --- |
| Template 版本 Person.h | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | ## Person.h  #ifndef PERSON\_H\_INCLUDED  #define PERSON\_H\_INCLUDED  #include <iostream>  using namespace std;  template<typename T>  class Person{  // T 已經被 Person 類別使用，函式重載要改用 T2  template<typename T2> friend ostream& operator<<(ostream&, Person<T2>&);  // T 已經被 Person 類別使用，函式重載要改用 T2  template<typename T2> friend istream& operator>>(istream&, Person<T2>&);  private:  string name;  T height;  T weight;  public:  void set\_name(string);  string get\_name();  void set\_height(T);  T get\_height();  void set\_weight(T);  T get\_weight();  Person(string="anonymous", T=0, T=0);  ~Person();  T operator+(Person&);  }  …  #endif |

|  |  |
| --- | --- |
| Template 版本 set\_name | |
| 1  2  3  4 | template<typename T>  void Person<T>::set\_name(string n){  name = n;  } |

|  |  |
| --- | --- |
| Template 版本 get\_name | |
| 1  2  3  4 | template<typename T>  string Person<T>::get\_name(){  return name;  } |

|  |  |
| --- | --- |
| Template 版本 set\_height | |
| 1  2  3  4 | template<typename T>  void Person<T>::set\_height(T n){  height = n;  } |

|  |  |
| --- | --- |
| Template 版本 get\_height | |
| 1  2  3  4 | template<typename T>  T Person<T>::get\_height(){ // 注意回傳型態為 T  return height;  } |

|  |  |
| --- | --- |
| Template 版本 set\_weight | |
| 1  2  3  4 | template<typename T>  void Person<T>::set\_weight(T n){  weight = n;  } |

|  |  |
| --- | --- |
| Template 版本 get\_weight | |
| 1  2  3  4 | template<typename T>  T Person<T>::get\_weight(){  return weight;  } |

|  |  |
| --- | --- |
| Template 版本 Person | |
| 1  2  3  4  5  6 | template<typename T>  Person<T>::Person(string n, T h, T w){  name = n;  height = h;  weight = w;  } |

|  |  |
| --- | --- |
| Template 版本 ~Person | |
| 1  2  3  4 | template<typename T>  Person<T>::~Person(){  cout << "Object was destructed!" << endl;  } |

|  |  |
| --- | --- |
| Template 版本 + 運算子重載 | |
| 1  2  3  4 | template<typename T>  T Person<T>::operator+(Person<T>& p){  return weight+p.weight;  } |

|  |  |
| --- | --- |
| Template 版本 << 運算子重載 | |
| 1  2  3  4  5  6  7 | template<typename T>  ostream& operator<<(ostream& os, Person<T>& p){  os << "Name: " << p.name << endl;  os << "Height: " << p.height << endl;  os << "Weight: " << p.weight << endl;  return os;  } |

|  |  |
| --- | --- |
| Template 版本 >> 運算子重載 | |
| 1  2  3  4  5 | template<typename T>  istream& operator>>(istream& is, Person<T>& p){  is >> p.name >> p.height >> p.weight;  return is;  } |

|  |  |
| --- | --- |
| 在 main.cpp 中測試 Person 類別 | |
| 1  2  3  4  5  6  7  8 | #include <iostream>  #include "Person.h" // 記得用 "" 引入  using namespace std;  int main()  {  …  } |

5. 重載成績類別中的運算子

A. 重載 << 和 >> 運算子，讓類別可以直接進行輸入與輸出

B. 重載 [] 運算子，回傳特定編號學生的成績，即類似陣列的操作方式

C. 把成績類別改成模板，讓使用者可以指定成績的資料型別

D. 切割成標頭檔 .h 與原始碼 .cpp，試著在 main.cpp 中呼叫該類別

|  |  |
| --- | --- |
| Course.h | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #ifndef COURSE\_H\_INCLUDED  #define COURSE\_H\_INCLUDED  #include <iostream>  #include <stdlib.h>  using namespace std;  template<class T> // 關鍵字用 class 或 typename 都可以  class Course{  template<class T2> friend istream& operator>>(istream&, Course<T>&);  template<class T2> friend ostream& operator<<(ostream&, Course<T>&);  private:  string lecturer;  int student\_number;  T \*score;  public:  void set\_lecturer(string);  string get\_lecturer();  void set\_student\_number(int);  int get\_student\_number();  void set\_score\_list(T \*);  T \* get\_score\_list();  float average();  Course(string="unknown",int=0); // 陣列改為統一由類別建構  ~Course();  T operator[](int);  }  …  #endif // COURSE\_H\_INCLUDED |

|  |  |
| --- | --- |
| << 運算子重載 | |
| 1  2  3  4  5  6  7  8  9 | template<class T>  ostream& operator<<(ostream& os, Course<T>& c){  os << "Lecturer:" << c.lecturer;  os << "Student number:" << c.student\_number << endl;  for(int i=0;i<c.student\_number;i++){  os << i+1 << "-th:" << \*(c.score+i) << endl;  }  return os;  } |

|  |  |
| --- | --- |
| >> 運算子重載 | |
| 1  2  3  4  5  6  7  8  9  10 | template<class T>  istream& operator>>(istream& is, Course<T>& c){  cout << "Please enter lecturer and number of students:" << endl;  is >> c.lecturer >> c.student\_number;  cout << "Please enter " << c.student\_number << " scores:" << endl;  for(int i=0;i<c.student\_number;i++){  is >> \*(c.score+i);  }  return is;  } |

|  |  |
| --- | --- |
| [] 運算子重載 | |
| 1  2  3  4 | template<class T>  T Course<T>::operator[](int index){ // [] 運算子重載在 Course 內部  return \*(score+index); // 回傳第 index 筆學生的成績  } |

測試修改過的 Course 類別之前，記得其他函式的定義前也要加上 template<class T>，且把「Course::」改為「Course<T>::」，另外，要分別更改 set\_score\_list 和 get\_score\_list 的傳入參數型態和回傳型態：

void Course<T>::set\_score\_list(T\* p){…}

T\* Course<T>::get\_score\_list(){…}

|  |  |
| --- | --- |
| 在 main.cpp 中測試 Course 類別 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <iostream>  #include "Course.h"  using namespace std;  int main()  {  Course<float> math("LKM",5);  cin >> math;  cout << math;    for(int i=0;i<math.get\_student\_number();i++)  cout << endl;  return 0;  } |
| 執行結果 | |
| Please enter lecturer and number of students:  >> lkm 5  Please enter 5 scores:  >> 40 70 50 60 80  Lecturer: lkm  Student number: 5  1-th:40  2-th:70  3-th:50  4-th:60  5-th:80  Course finished! | |

**Take Home Message**

A. Abstract Data Type(ADT)有哪些意涵？目的為何？

B. 要如何在類別外定義函式？

C. 有哪些權限可以設定？private 和 public 權限兩者之間有什麼差異？

D. 建構式與解構式分別負責什麼？什麼時候會觸發？

E. 運算子重載要如何進行？

F. 模板(Template)的功能為何？

G. (.h) 和 (.cpp) 兩種檔案如何分工？