

C/C++ 進階班 資料結構

抽象資料型別 (Abstract Data Type)

李耕銘

課程大綱

- 物件導向程式設計簡介
- 名詞定義&資料型別抽象化
- 實作資料型別抽象化
 1. 類別宣告
 2. 定義類別下的方法
 3. 類別的權限設定
 4. 建構式與解構式
 5. 重載運算子
 6. 函式模板與類別模板
 7. 標頭檔的建立

物件導向程式設計

本門課非物件導向

因時間因素只會講授到之後會用到的相關觀念與語法，完整的物件導向 (多型、設計模式) **不會**講授。

物件導向程式設計

- 早期以硬體導向的程式設計
 - 強調節省記憶體與效能
 - short



硬體效能逐漸增加...

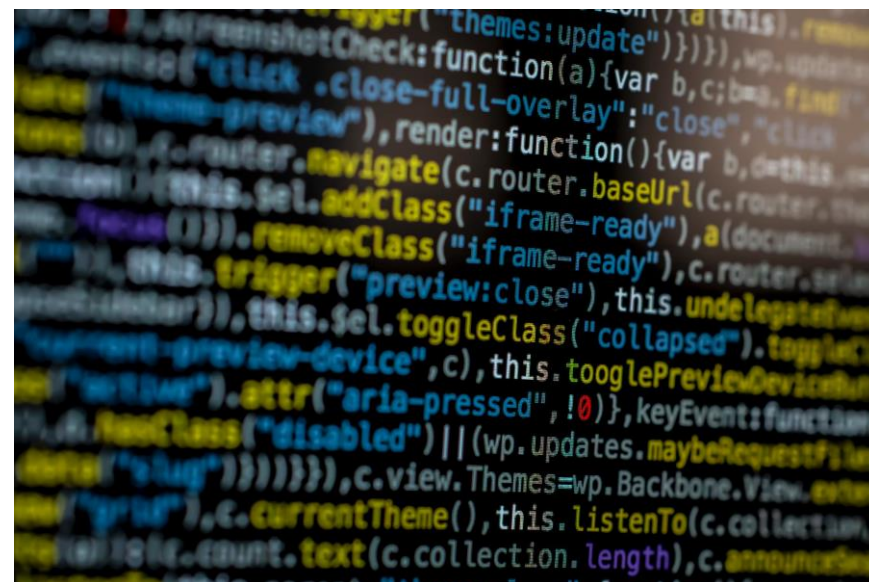


PDP-7
Ref: [Wikipedia](#)

物件導向程式設計

- 當你的程式越寫越長.....

- 擴展性？
- 好維護？
- 容易複用？



物件導向是程式設計的一種**精神**

物件導向程式設計

Once and only once.

DRY

Don't Repeat Yourself.

減少重複的資訊

WET

Write everything twice.

We enjoy typing.

Waste everyone's time.

物件導向程式設計

非結構
程式設計

結構化
程式設計

模組化
程式設計

物件導向
程式設計

Function

Functions

Classes

物件導向程式設計

傳統程式設計

程式為眾多函式/
指令組成

物件導向設計

- 程式由獨立又可互相呼叫的物件組成
- 物件能獨立被撰寫、使用、呼叫、儲存、運作
- 物件間也可以彼此合作

物件導向程式設計

物件導向

- 不同程式語言間共通的精神
- 把程式拆解成以物件為主的方式

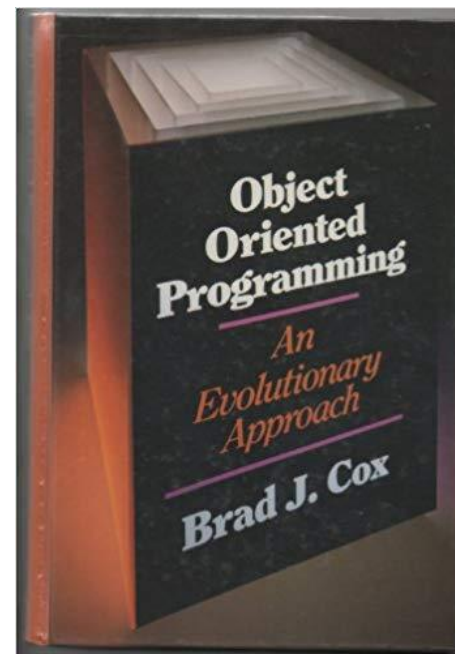
設計模式

- 程式設計經驗的總結
- 確保可靠性、複用、易讀
- 設計模式的基礎是物件導向
- 常見的設計模式有23種

*Design Patterns: Elements of
Reusable Object-Oriented Software*

物件導向程式設計

- Brad Cox 提出的物件導向
 - 物件 (Object) 與訊息 (Message)
 - 繼承 (Inheritance)
 - 封裝 (Encapsulation)
 - 動態連結 (Dynamic binding)
- 繼承、封裝、動態連結
 - OOP最重要的三大基石
 - 以此概念開發了Objective-C



Ref: [Amazon](#)

物件導向程式設計

- Brad Cox於1980年代發明
 - 物件化的C語言
 - 導入訊息與物件的概念
- [被傳的類別 傳入訊息: 傳入參數];
- Mac OS開發的首選語言
- iOS過去的開發語言
 - 現已經被Swift取代

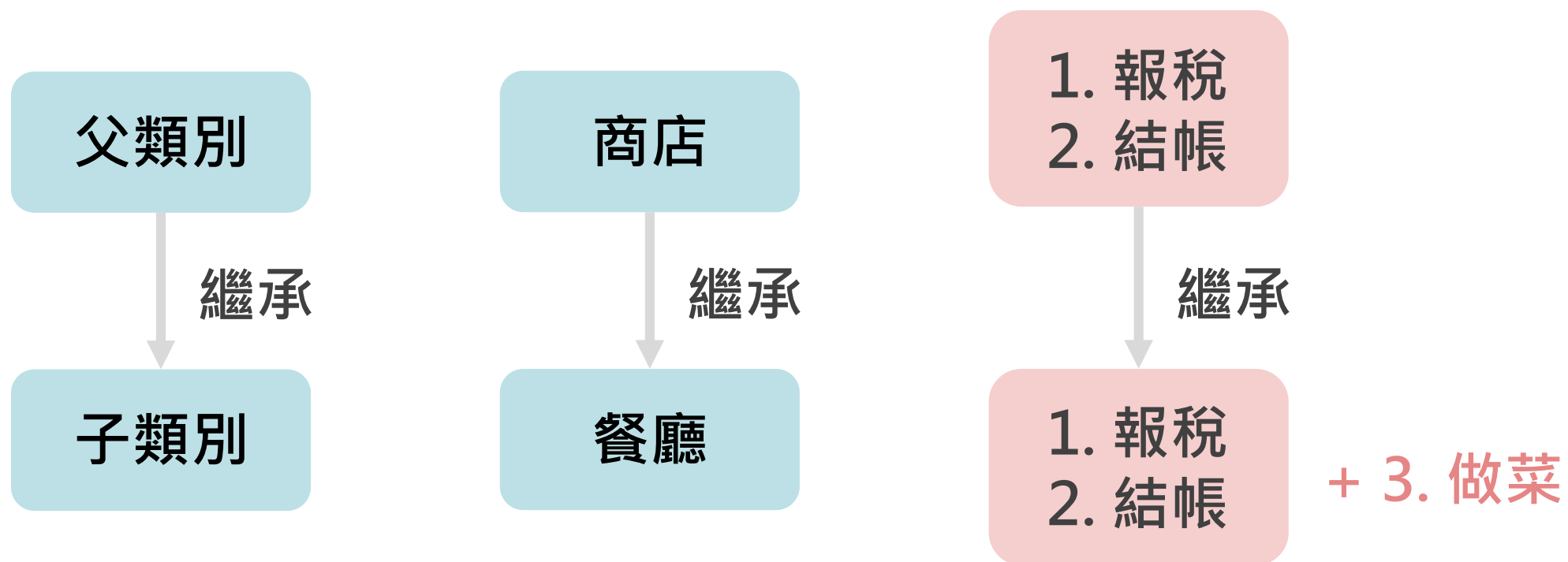


Objective-C

Ref: [Memorylack](#)

物件導向程式設計

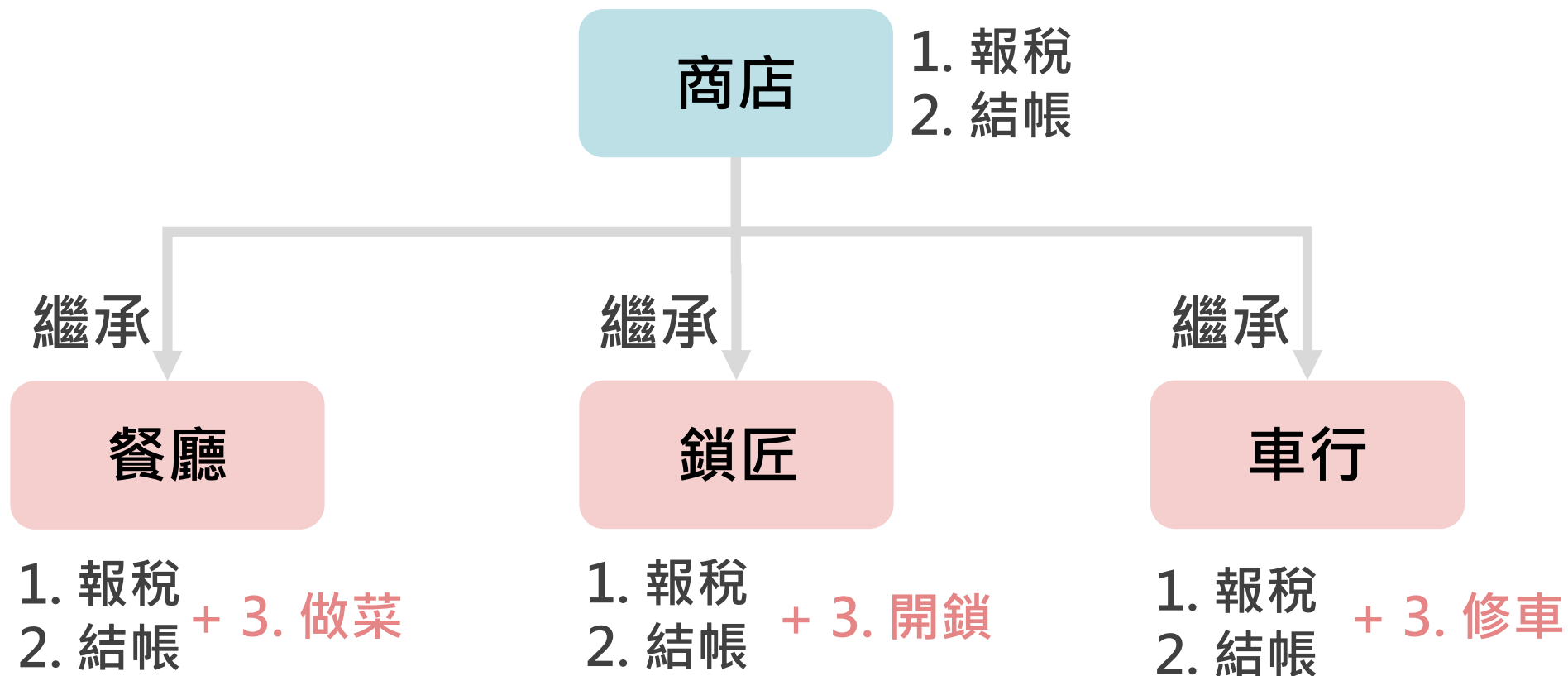
透過繼承得到其他物件的內容



本門課不提

物件導向程式設計

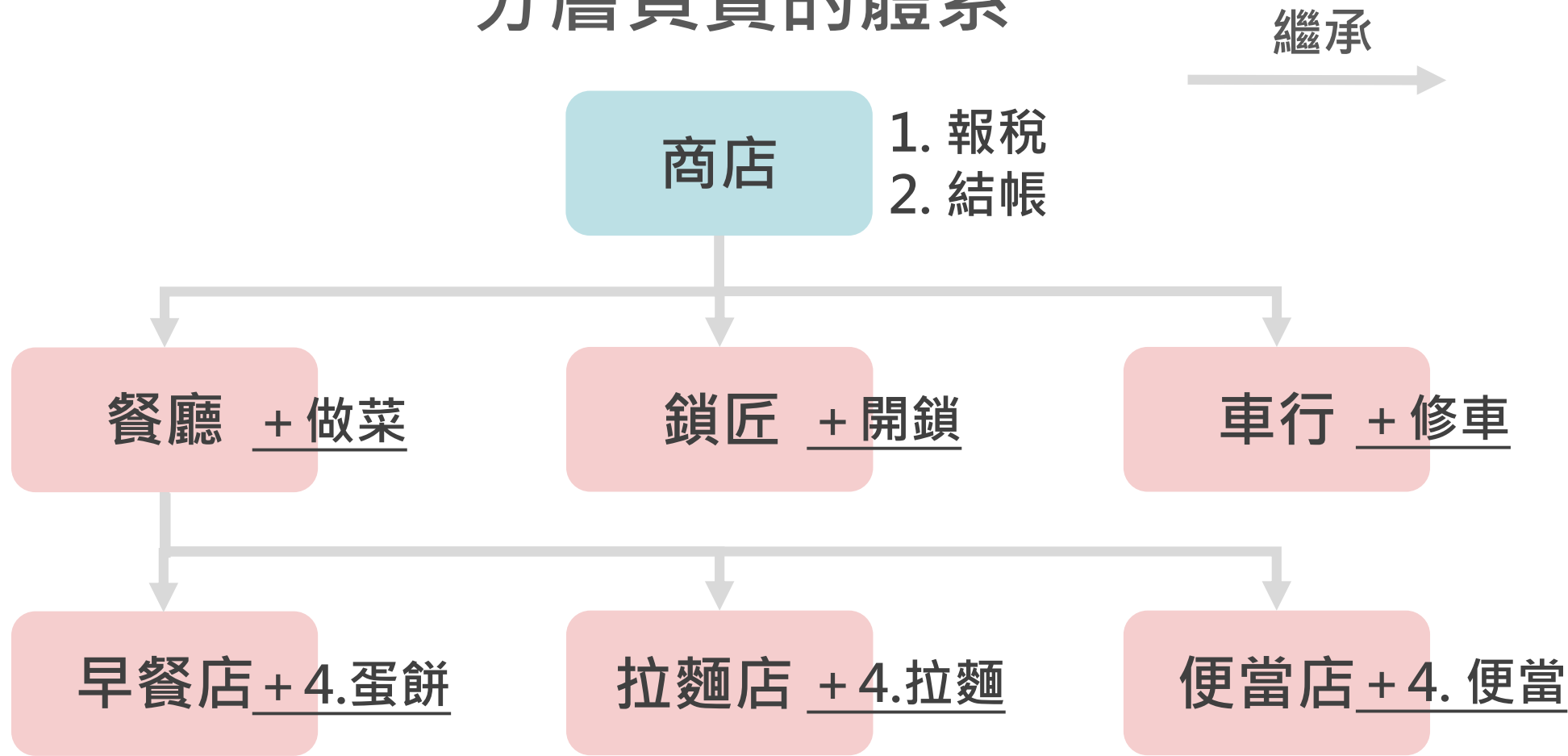
Once and only once.



本門課不提

物件導向程式設計

分層負責的體系



本門課不提

物件導向程式設計

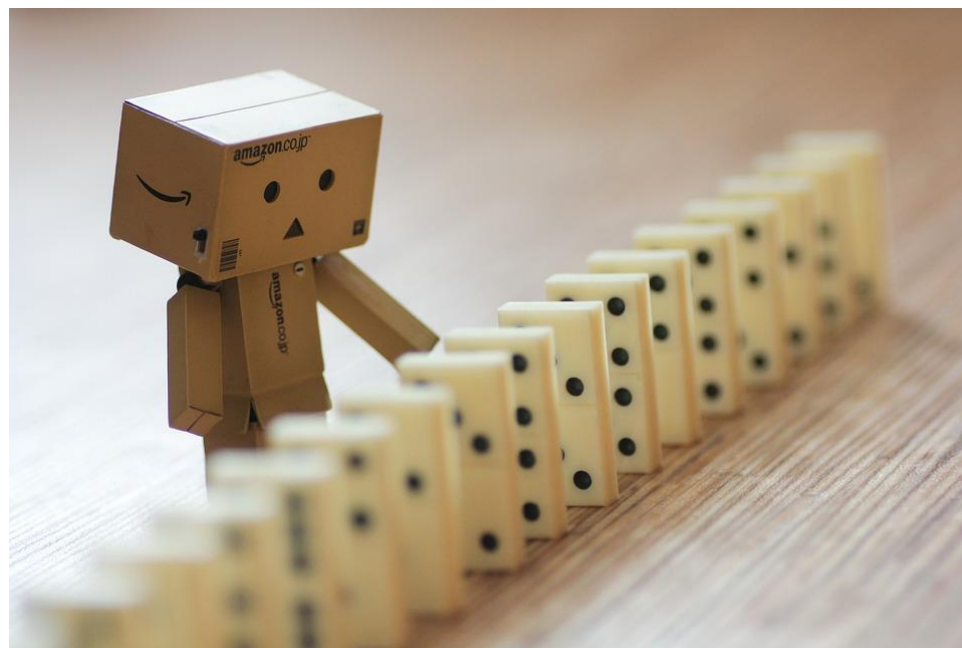
- 降低物件與介面的複雜度

- 透過封裝來達到資訊隱藏，限制外界使用
- 減少物件間的互相干擾
- 確保修改物件時，不會影響牽一髮動全身

- 小而美的對外通道

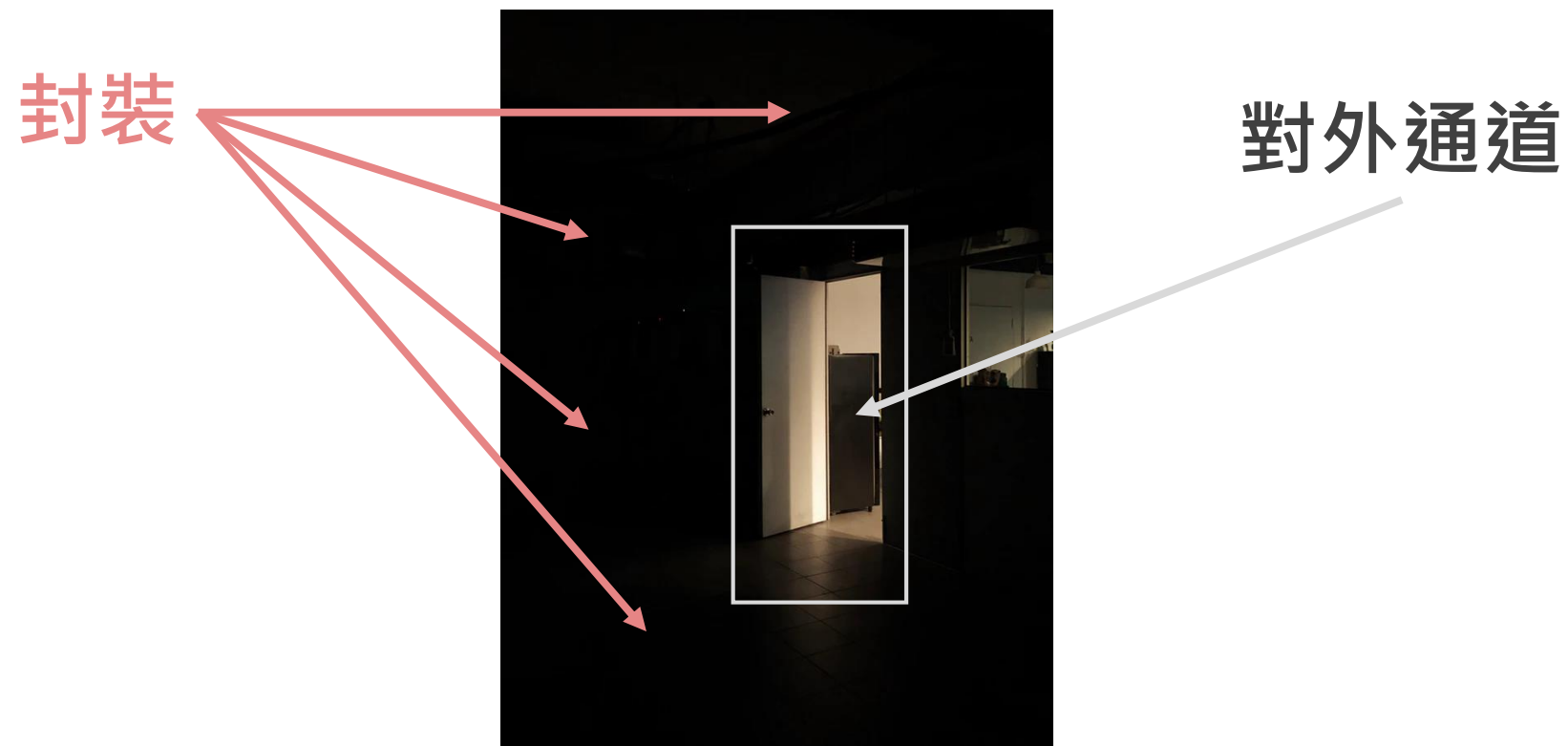
- 小：只開啟必要的對外通道，預設是關閉的
- 美：方便使用者使用

物件導向程式設計

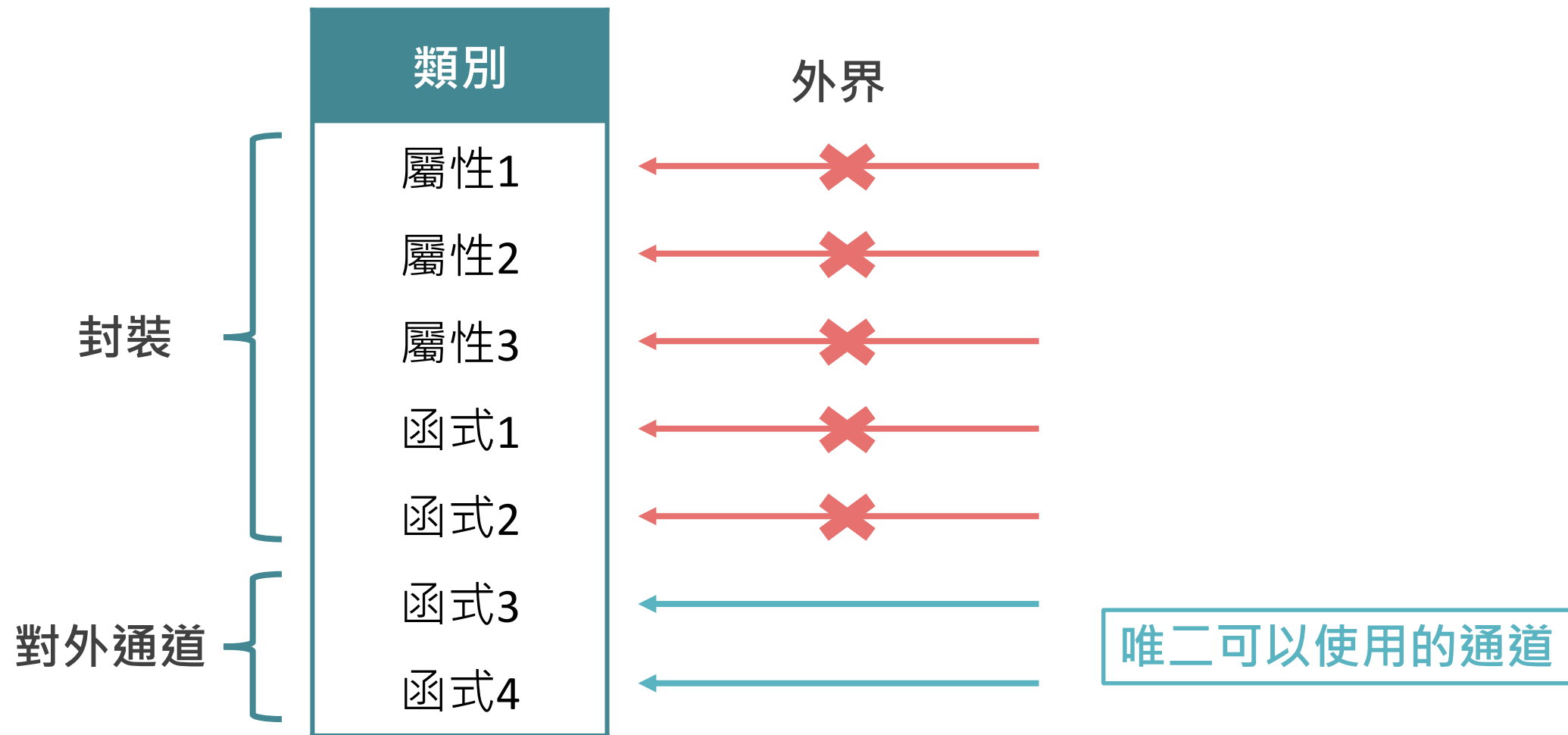


沒有封裝的後果.....

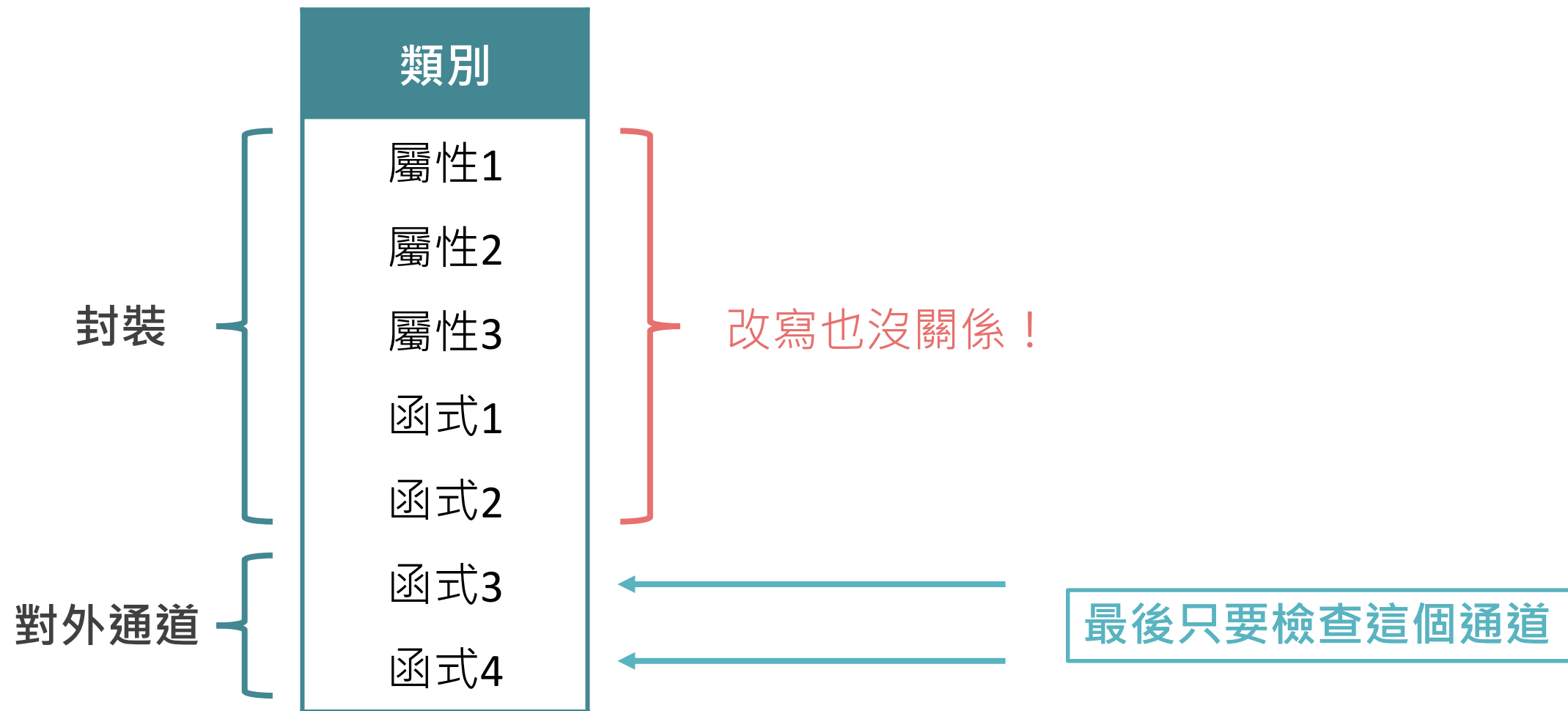
物件導向程式設計



物件導向程式設計



物件導向程式設計



物件導向程式設計

靜態連結(Static Binding)

- 識別字對應到的記憶體在編譯階段(Compile Time)決定
- final、static、private

動態連結(Dynamic Binding)

- 識別字對應到的記憶體在執行階段(Run Time)決定

識別字

綁定

變數或函式@記憶體

本門課不提

物件導向程式設計

- 靜態連結(Static Binding)

➤ 識別字對應到的記憶體在編譯階段(Compile Time)決定

```
void Funtcion(int *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << endl;  
}
```

```
void Funtcion(float *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << endl;  
}
```

```
void Funtcion(double *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << endl;  
}
```

```
int Arr_1[3]={1,2,3};  
float Arr_2[3]={4.0,5.0,6.0};  
double Arr_3[3]={7.0,8.0,9.0};
```

Funtcion(Arr_1, 3);

Funtcion(Arr_2, 3);

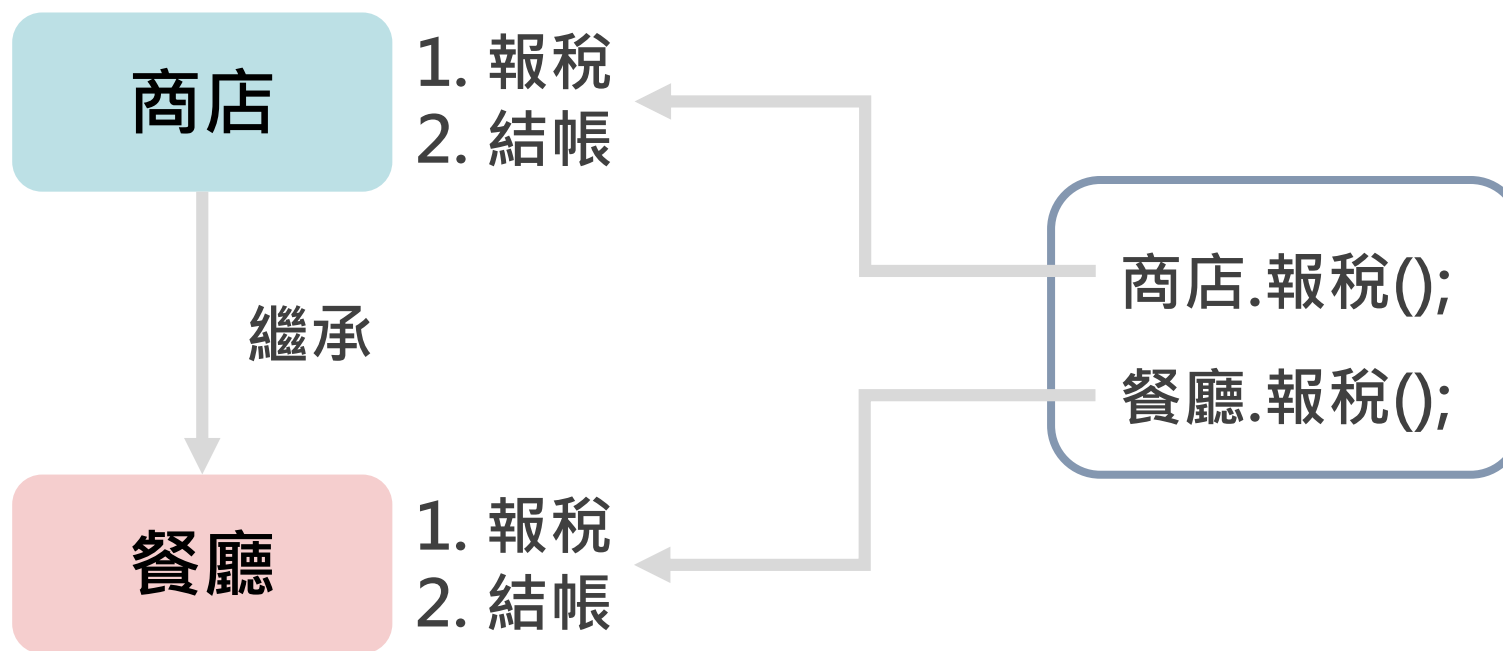
Funtcion(Arr_3, 3);

本門課不提

物件導向程式設計

- 動態連結(Dynamic Binding)

➤ 識別字對應到的記憶體在執行階段(Run Time)決定



本門課不提

物件導向名詞定義

類別(Class)

- 記載物件內部含有哪些成員
- 描述物件是如何被架構出來
- 相當於物件的規格書
- 可視作一種資料型別

類別

屬性1

屬性2

屬性3

函式1

函式2

函式3

函式4

實體(Instance)

- 根據類別(Class)開出的物件變數
- 大部分操作都必須對**實體(Instance)**操作

資料型別 識別字;
類別名稱 實體名稱;

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

```
int main () {  
    string str;  
    ifstream myfile;  
    myfile.open ("example.txt");  
    myfile >> str;  
    cout << str << endl;  
    return 0;  
}
```

實體



屬性(Attribute)

- 類別所擁有的變數/資料成員
- 儲存每一個物件實體的狀態
- 不同物件實體的屬性可以有不同的值
- 類似於結構(Structure)底下的成員



方法(Function)

- 方法是物件底下的函式或運算子
- 不同的傳入值(訊息)對應到不同方法
- 通常透過方法去改變屬性

類別

屬性1

屬性2

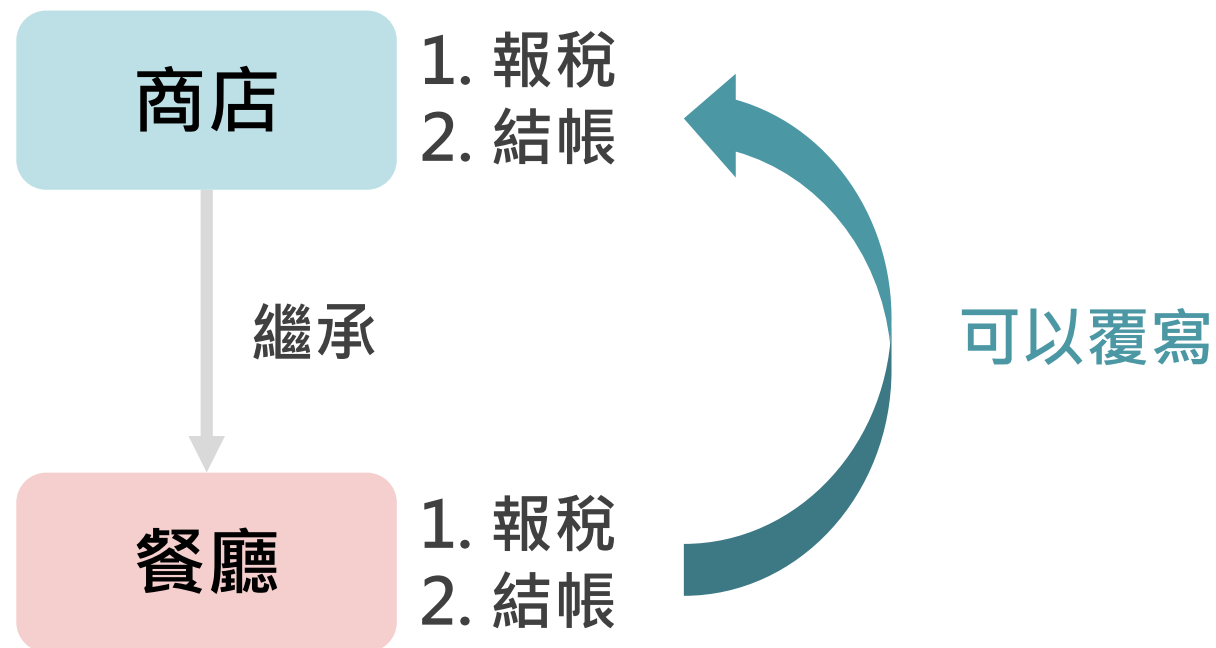
屬性3

函式1

函式2

覆寫(Override)

子類別將父類別的函式重新定義功能

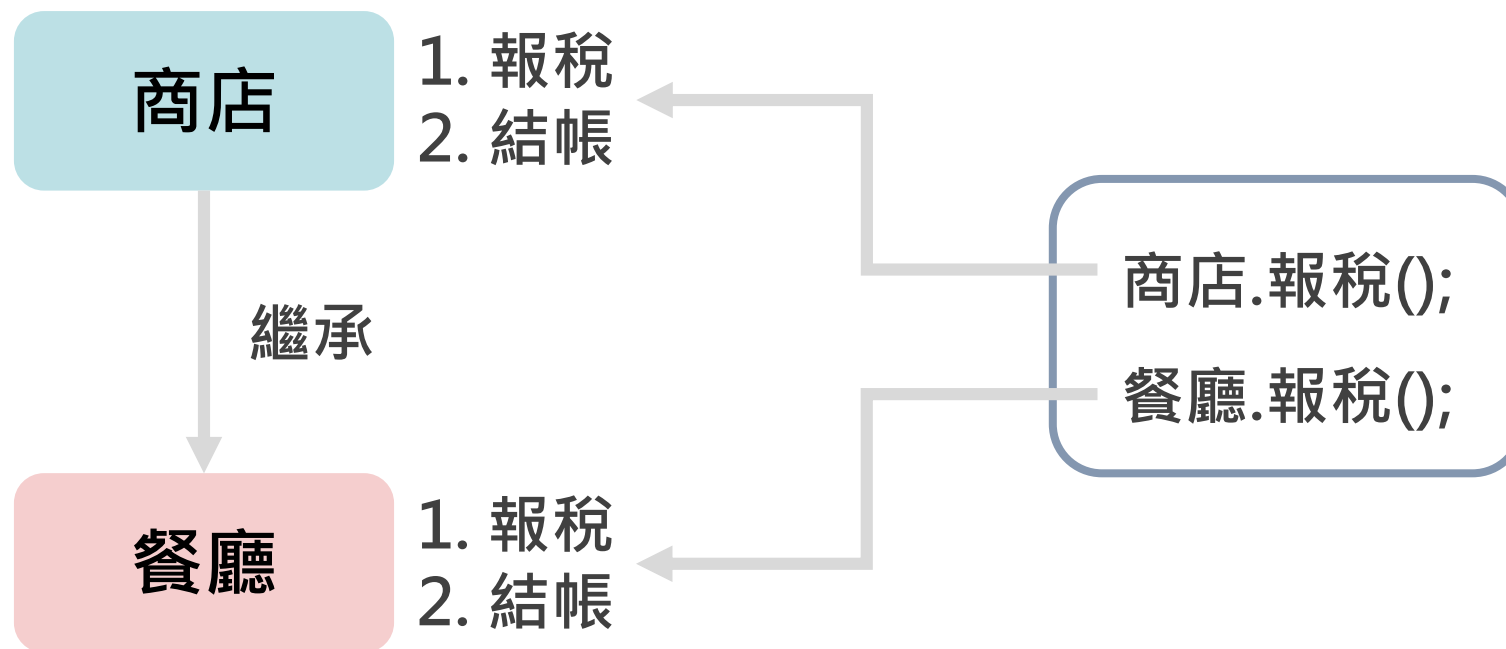


本門課不提

物件導向名詞定義

訊息(Message)

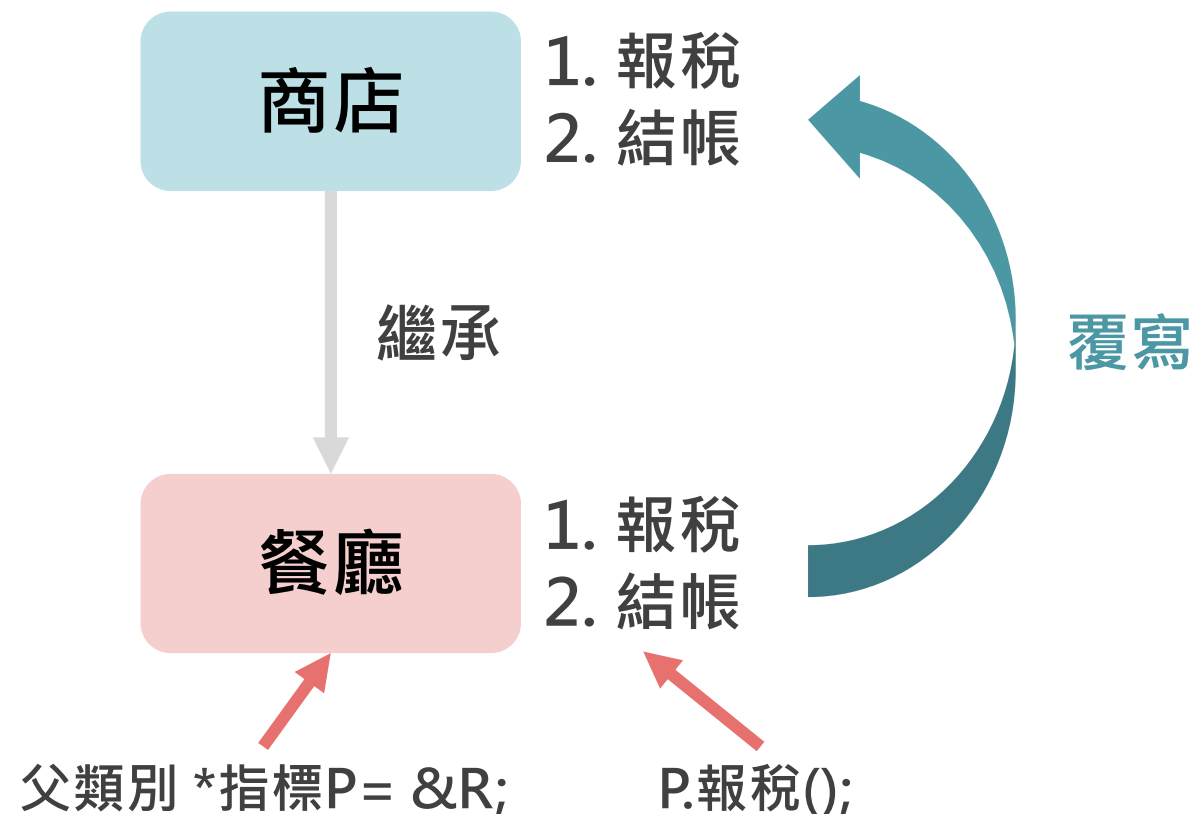
- 物件間互相呼叫時溝通的工具
- 物件收到訊息後，會執行相對應的方法



本門課不提

多型(Polymorphism)

- 透過父類別來操作子類別的方法
- 多型三要素
 - 繼承
 - 覆寫父類別的方法
 - 父類別指標指向子物件



本門課不提

資料型別抽象化

何為「抽象」(Abstraction)



- 「國家」四要素
 1. 人民
 2. 領土
 3. 主權
 4. 政府
- 並不指特定/具體的國家
- 「國家」是一種**抽象**的概念

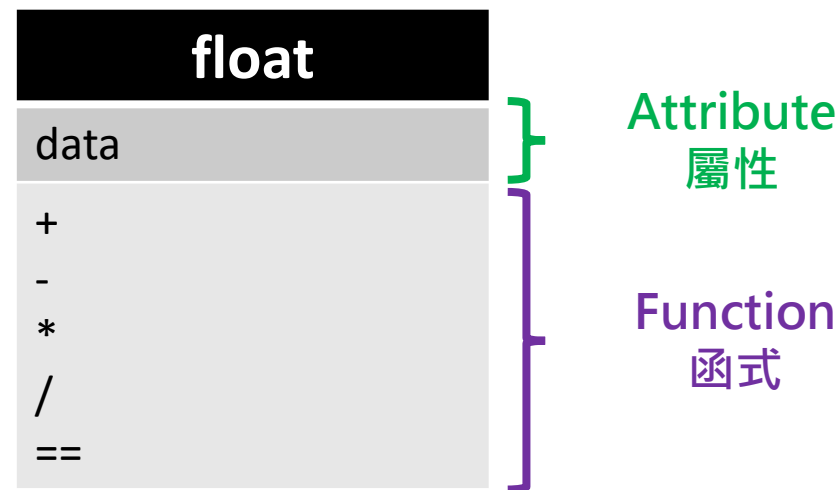
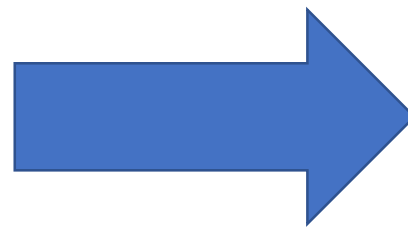
「抽象」(Abstraction)

- 「抽象」(Abstraction)
 1. Functional Abstraction
 - 把目的與實作撥離開來
 2. Data Abstraction
 - 有哪些**操作**資料的方式
 - 不在乎實際上是怎麼執行的
- 幾乎所有語言都支援 process 抽象
- 幾乎所有語言都支援資料型別抽象 (Data abstraction)

「抽象」(Abstraction)

- Abstract Data Type (ADT)
 - 由資料(Data)以及資料間的操作(Operation)構成
 - 只要知道資料的特性，不需要知道操作是怎麼被實作出的
 - 會用就好，管它實際上是怎麼寫的

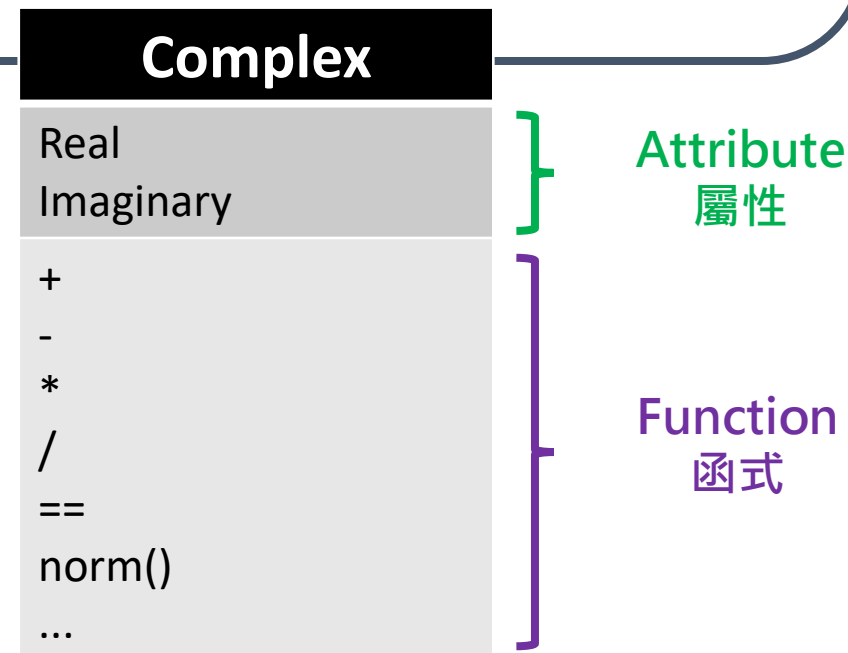
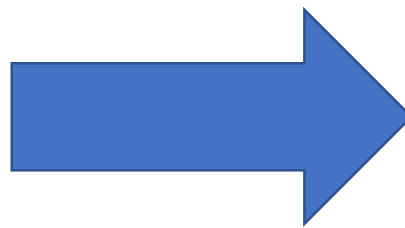
浮點數
float



「抽象」(Abstraction)

- Abstract Data Type (ADT)
 - 由資料(Data)以及資料間的操作(Operation)構成
 - 只要知道資料的特性，不需要知道操作是怎麼被實作出的
 - 會用就好，管它實際上是怎麼寫的
 - 使用者自定義的資料型別

複數
 $a + bi$

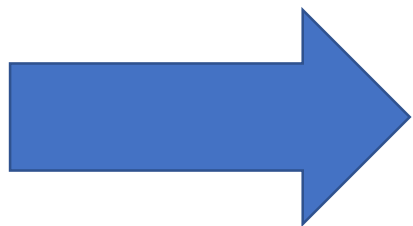


「抽象」(Abstraction)

- 封裝 (Encapsulation)

- 把資料與函式包在一起形成**類別(Class)**
- 通常屬性(Attribute)不能被外界取用，只能透過函式存取
- 資料封裝是物件導向程式設計的基本概念

複數
 $a + bi$



Complex	
Real	}
Imaginary	
+	}
-	
*	
/	
==	
get_distance()	

Attribute
屬性

Function
函式

類別
Class

「抽象」(Abstraction)

- C++ 的抽象化實作
 - 類別(Class)/結構(Structure)定義了一個新的資料型態，包含
 1. 資料成員
 2. 函式
 - 物件(Object)是類別(Class)的實體(Instance)

```
int a;  
float b;  
double c;
```

類別宣告

物件的產生與使用

把程式以物件為基礎切割成一個個物件

封裝(Encapsulation)

- 把描述該物件的屬性、方法封裝入物件
- 類別是物件的規格書，記載了物件有哪些屬性與方法

設計一個車子類別

- 屬性：品牌
- 屬性：引擎大小
- 方法：發動引擎
- 方法：煞車

物件的產生與使用

宣告類別底下的屬性

```
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;  
};
```

```
class rectangle{  
    float width;  
    float height;  
public:  
    float area();  
    void set_len(float,float);  
};
```

```
rectangle R;
```

宣告出實體來使用

```
類別名稱 實體名稱;
```

- 操作大部分都是**對實體做操作**
 - static除外(本門課不提)

物件的產生與使用

- 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;

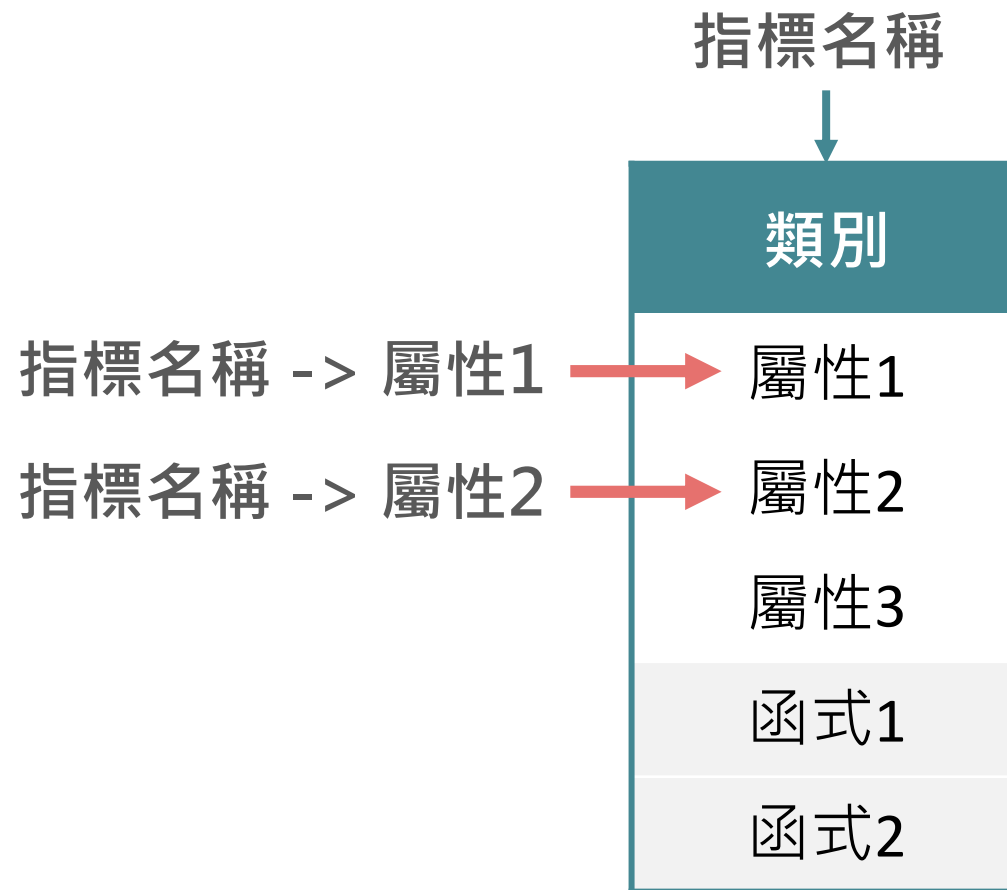
- 若為物件指標，透過->來使存取

類別名稱 *指標名稱 = &實體;
(*指標名稱).成員;
指標名稱->成員;

```
class rectangle{  
    float width;  
    float height;  
public:  
    float area();  
    void set_len(float,float);  
};  
  
rectangle R;  
R.set_len(2.5,3.5);  
rectangle* p = &R;  
cout << p->area() << endl;
```

物件的產生與使用

```
class rectangle{  
    float width;  
    float height;  
public:  
    float area();  
    void set_len(float,float);  
};  
  
rectangle R;  
R.set_len(2.5,3.5);  
rectangle* p = &R;  
cout << p->area() << endl;
```



定義類別下的方法

類別底下的方法

- 類別底下主要可分為兩大類
 - 描述狀態的變數(屬性)
 - 操作作用的函式(方法)，可改變物件的狀態
- 最基本的函式直接寫在類別底下

```
class rectangle{  
    float width;  
    float height;  
    float area(){  
        return width*height;  
    };  
};
```

類別底下的方法

- 如果把函式全部寫在類別裏頭，類別宣告會又臭又長
- 類別裡函式的定義預設為inline

```
class rectangle{  
    float width;  
    float height;  
    float area(){  
        return width*height;  
    };  
    void setLen(float w,float h){  
        width = w;  
        height = h;  
    };  
    void rectangle(){  
        width = 0;  
        height = 0;  
    };  
};
```

類別底下的方法

- 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**放在類別宣告外頭

```
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;  
}
```

因為定義在外頭，所以要指名是哪個類別底下的哪個函式

類別底下的方法

```
class rectangle{  
    float width;  
    float height;  
    float area();  
    void setLen(float,float);  
    rectangle();  
};
```

```
float rectangle::area(){  
    return width*height;  
};  
void rectangle::setLen(float w,float h){  
    width = w;  
    height = h;  
};  
rectangle::rectangle(){  
    width = 0;  
    height = 0;  
};
```


Example Code

Mission

創建一個人的類別，使其包含以下成員：

1. 姓名
2. 身高
3. 體重

Example Code

Mission

在人的類別中新增以下函式：

1. `set_name`
2. `get_name`
3. `set_height`
4. `get_height`
5. `set_weight`
6. `get_weight`
7. `print`

Practice

Mission

創建一個課程類別，使其包含以下成員：

- 教師姓名
- 學生人數
- 成績列表 (應用指標)

Practice

Mission

在課程類別中新增以下函式：

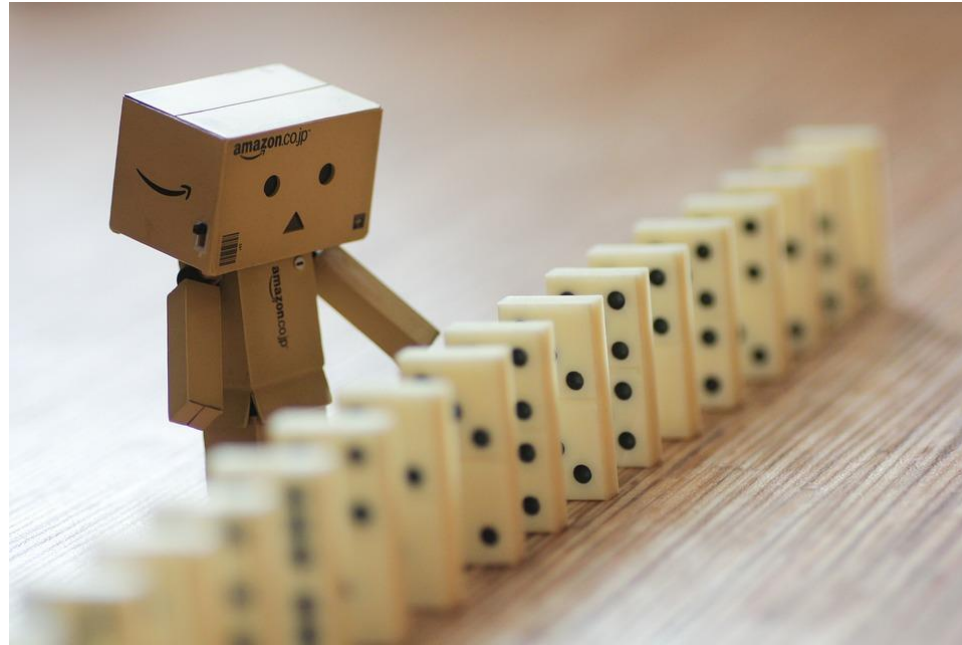
1. `set_lecturer`
2. `get_lecturer`
3. `set_student_number`
4. `get_student_number`
5. `set_grade_list`
6. `get_grade_list`
7. `average`

類別的權限設定

類別的權限設定

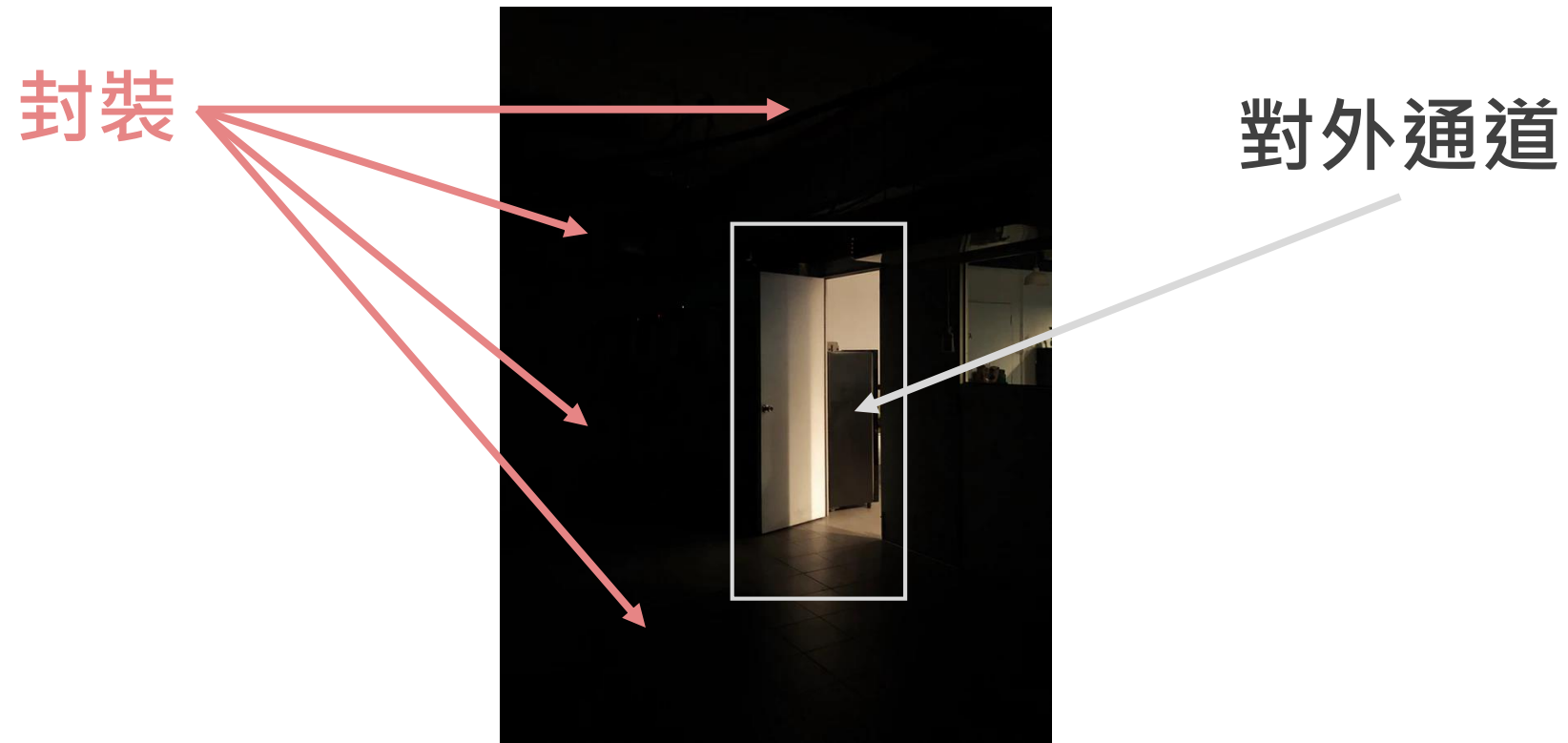
- 降低物件與介面的複雜度
 - 透過封裝來達到資訊隱藏，限制外界使用
 - 減少物件間的互相干擾
 - 確保修改物件時，不會影響牽一髮動全身
- 小而美的對外通道
 - 小：只開啟必要的對外通道，預設是關閉的
 - 美：方便使用者使用
- 原則上所有的屬性都會被封裝起來

類別的權限設定

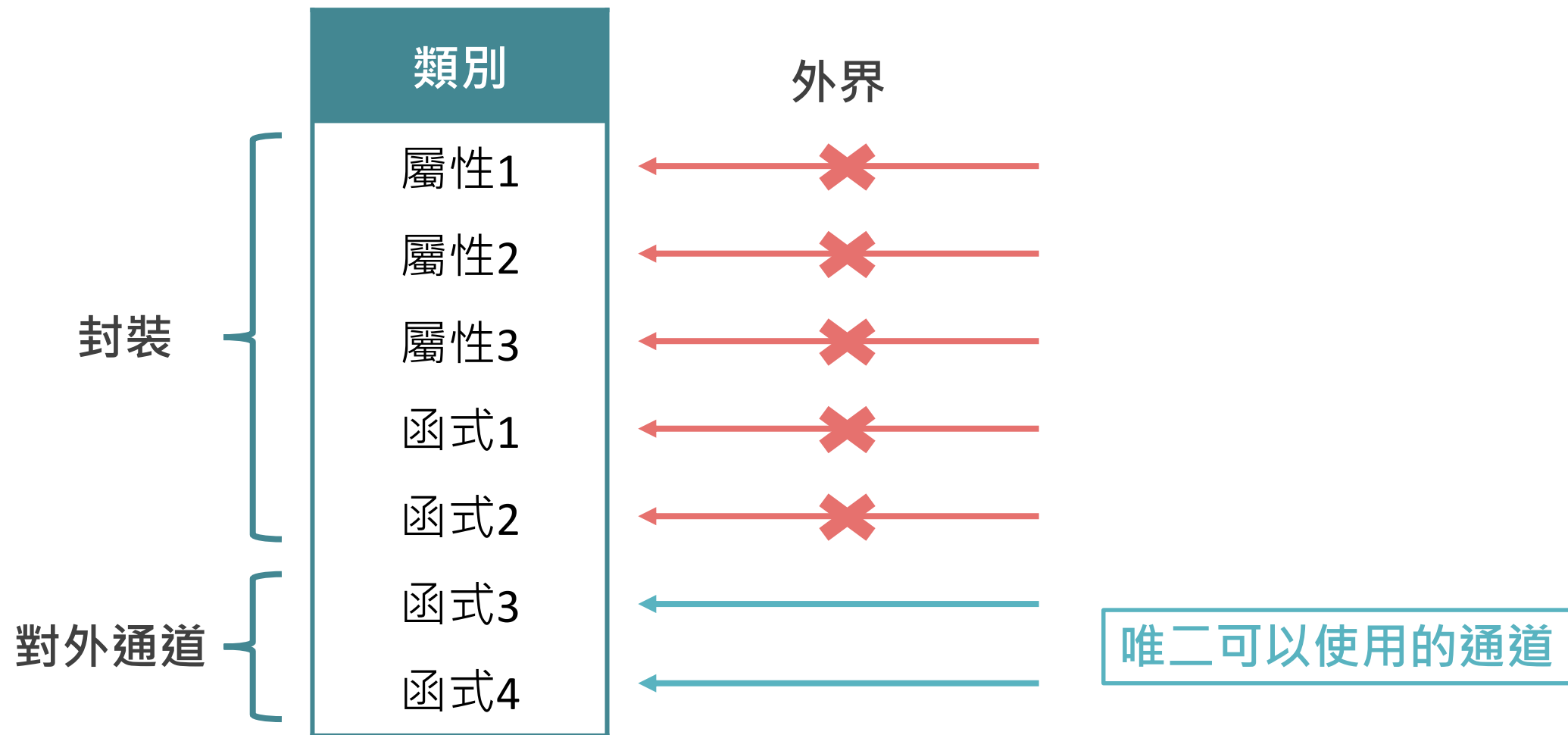


沒有封裝的後果.....

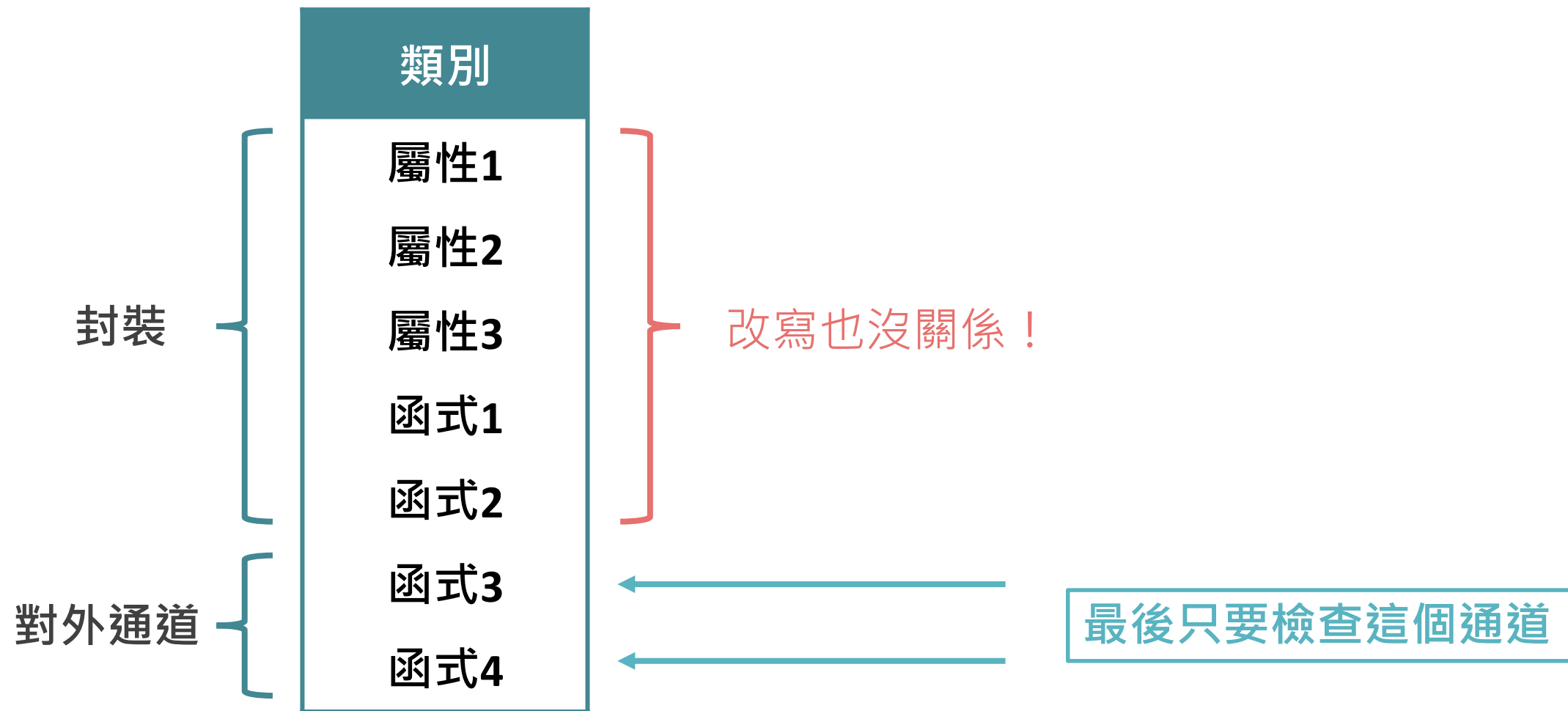
類別的權限設定



類別的權限設定



物件導向程式設計



高內聚 低耦合

物件裡複雜的運算只留在物件內部
供外界呼叫使用的通道是少量且單純的



類別的權限設定

public(公開成員)

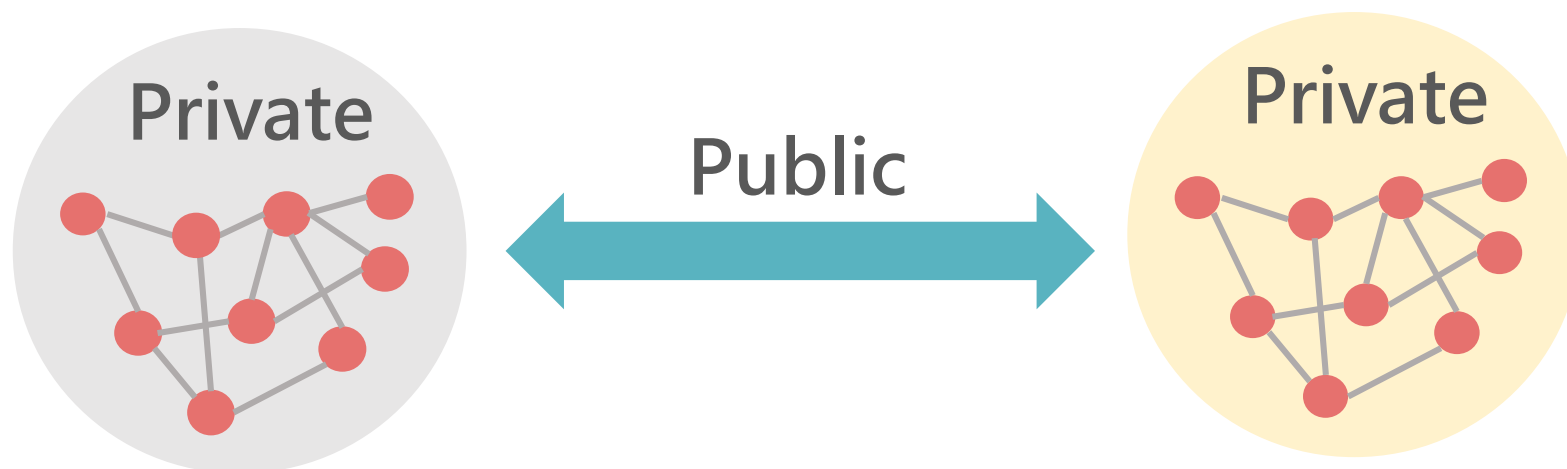
- 可以透過物件名稱被外界呼叫

private(私有成員)

- 不可以透過物件名稱被外界呼叫
- 類別封裝的原則
- 能不公開就不公開
- 取得(get)或設定(set)屬性儘量透過方法來進行
- 最小化公開的資訊，只公開給外界呼叫的部分
- 如果沒有寫權限，預設為private

類別的權限設定

物件裡複雜的運算只留在物件內部
供外界呼叫使用的通道是少量且單純的



類別的權限設定

- 透過**Private**來達到資訊隱藏
- 透過**Public**創建給外界用的通道

```
class rectangle{  
    private: ← 可省略，預設值即為private  
        float width;  
        float height;  
    public:  
        float area();  
        void setLen(float,float);  
        void rectangle();  
};
```

類別的權限設定

```
class rectangle{  
    private:  
        float width;  
        float height;  
    public:  
        float area();  
        void setLen(float,float);  
        rectangle();  
};
```

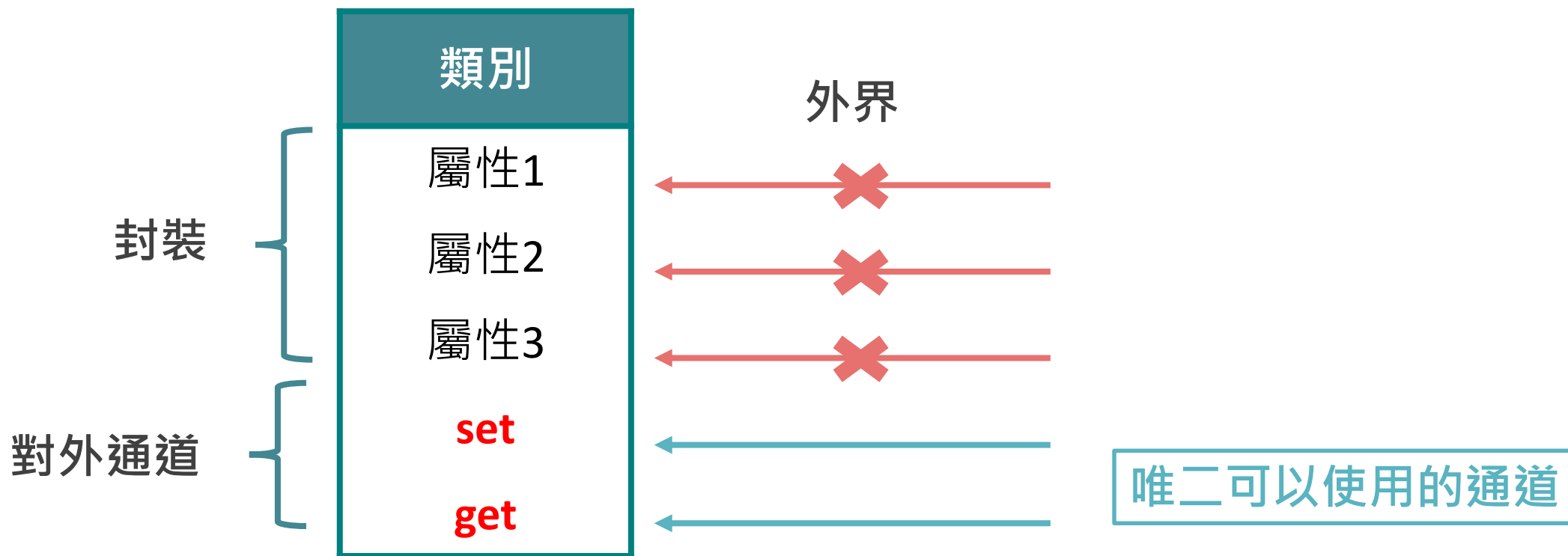
Public成員能夠直接在main裏頭被呼叫

```
int main(){  
    float w,h;  
    cout << "Please enter width and height:";  
    cin >> w >> h;  
  
    rectangle t;  
    t.setLen(w,h);  
    t.width = w;  
    t.height = h;  
}
```

Private成員不能直接在main裏頭被呼叫

類別的權限設定

- 原則上所有的資料成員都會設定成 **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)



friend

- **private**成員不能被物件外部呼叫：只限於物件內部間呼叫
- **friend**可讓private成員被外部函式呼叫
- 函式(function)、類別(class)都可設定成friend



friend : 類別

```
class 類別名稱{  
    friend 函式;  
    friend 類別名稱;  
};
```

- 前面不須加權限
- 基於效率的考量，透過直接存取私有成員而非間接透過公開函式存取

friend : 函式

```
class rectangle{  
    friend float area(rectangle&);  
    friend void setLen(rectangle&);  
    private:  
        float width;  
        float height;  
};
```

```
float area(rectangle& r){  
    return r.width*r.height;  
}  
void setLen(rectangle& r, float w,float h){  
    r.width = w;  
    r.height = h;  
}
```

透過friend取用私有成員

friend : 類別

```
class rectangle{  
    friend class ruler;  
private:  
    float width;  
    float height;  
};
```

```
class ruler{  
    public:  
        void print(rectangle);  
};  
void ruler::print(rectangle r){  
    cout << "Width: " << r.width << endl;  
    cout << "Height: " << r.height << endl;  
}
```

透過friend取用私有成員



建構式與解構式

建構式與解構式

物件的初始值怎麼長這樣？
怎麼設定初始值？

```
Width: 1.12104e-044  
Height: 6.86636e-044
```

建構式與解構式

- 透過建構式給定實體化後的**初始值**
- 建構式與解構式都**不需要回傳資料型別**
- 需設定成**public**

建構式(Constructor)

- 物件被實體化時執行，通常用作**初始化**
- **類別名稱()**

解構式(Destructor)

- 物件從記憶體被釋放時執行，通常用作**清理**
- **~類別名稱()**

建構式與解構式

```
class rectangle{  
    friend ostream& operator<<(ostream&,rectangle);  
    private:  
        float width;  
        float height;  
    public:  
        rectangle();  
};
```

建構式名稱就是類別名稱

```
rectangle::rectangle(){  
    width = height = 0;  
}
```

不需回傳資料型別

建構式與解構式

建構式可以帶入引數方便使用者建構

```
class rectangle{  
    friend ostream& operator<<(ostream&,rectangle);  
    private:  
        float width;  
        float height;  
    public:  
        rectangle(float,float);  
};
```

```
rectangle::rectangle(float w,float h){  
    width = w;  
    height = h;  
}
```

建構式與解構式

建構式可以帶入引數方便使用者建構

```
int main()
{
    rectangle rectangleInstance(3,5);
    cout << rectangleInstance;
    return 0;
}
```



```
Width: 3
Height: 5
```

建構式與解構式

```
class rectangle{  
    friend ostream& operator<<(ostream&,rectangle);  
    private:  
        float width;  
        float height;  
    public:  
        ~rectangle(); ← 解構式名稱就是~類別名稱  
};
```

```
rectangle::~~rectangle(){  
    cout << "Bye" << endl;  
} ← 不需回傳資料型別
```

參考與解構

```
class rectangle{  
    friend ostream& operator<<(ostream&,rectangle);  
    friend float area(rectangle);  
private:  
    float width;  
    float height;  
public:  
    rectangle();  
    rectangle(float,float);  
    ~rectangle();  
};
```


參考與解構

```
float area(rectangle r){  
    return r.width*r.height;  
}
```

```
rectangle::~~rectangle(){  
    cout << "Bye" << endl;  
}
```

為什麼會觸發兩次解構式？

```
int main()  
{  
    rectangle rectangleInstance(3,5);  
    cout << area(rectangleInstance) << endl;  
    return 0;  
}
```



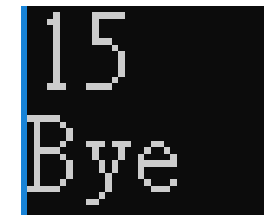
```
15  
Bye  
Bye
```

參考與解構

```
float area(rectangle& r){  
    return r.width*r.height;  
}
```

```
rectangle::~~rectangle(){  
    cout << "Bye" << endl;  
}
```

```
int main()  
{  
    rectangle rectangleInstance(3,5);  
    cout << area(rectangleInstance) << endl;  
    return 0;  
}
```



```
15  
Bye
```

建構式與解構式

- 建構式 Constructors

1. 常用來做資料成員的初始化或是配置記憶體空間
2. 當實體被宣告時自動觸發建構式
3. 沒有回傳資料型態、跟類別名稱同名
4. 可以有多种建構式，可接收不同引數做建構
5. 類別一定有建構式，若未定義編譯器會自動產生

- 解構式 Destructor

1. 常用來做清理、計數或記憶體釋放
2. 當物件的生命周期結束時觸發
3. 類別一定有解構式，若未定義編譯器會自動產生

Example Code

Mission

在人的類別中新增權限：

1. 所有的資料成員都應該被設定成私有
2. 特定設定給外界的函式可以設定成公開
3. 另外寫一個外部函式 `print`，並利用 `friend` 操作類別
4. 試著使用它！

Example Code

Mission

在人的類別中新增兩種建構式：

1. 沒有任何引數的建構式
2. 帶有引數的建構式
 - A. 姓名
 - B. 身高
 - C. 體重

Practice

Mission

在課程類別中新增權限：

1. 所有的資料成員都應該被設定成私有
2. 特定設定給外界的函式可以設定成公開
3. 另外寫一個外部函式 `print`，並利用 `friend` 操作類別
4. 試著使用它！

Practice

Mission

在課程類別中新增兩種建構式：

1. 沒有任何引數的建構式
2. 帶有引數的建構式
 - A. 教師姓名
 - B. 學生人數
 - C. 成績指標

重載運算子

重載運算子

- 運算子：+ - * / % & | > > < <
- 運算子的行為都是由人類定義出來的
- C/C++ 支援**運算子重載**
- 重新/追加定義運算子的功能
- 只能重載自定義的資料型別運算子

重載運算子

- 運算子重載是函式重載的延伸
- 函式名稱 => **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;  
}
```

重載運算子

定義： $\text{rectangle} \times N = \text{長寬伸縮}N\text{倍}$

```
class rectangle{  
    friend ostream& operator<<(ostream&,rectangle);  
    friend float area(rectangle&);  
private:  
    float width;  
    float height;  
public:  
    rectangle();  
    rectangle(float,float);  
    ~rectangle();  
    rectangle operator*(float);  
};
```

重載運算子

定義： $\text{rectangle} \times N = \text{長寬伸縮}N\text{倍}$

```
rectangle rectangle::operator*(float r){  
    rectangle rec(width*r,height*r);  
    return rec;  
}
```

```
int main()  
{  
    rectangle rectangleInstance(3,5);  
    cout << rectangleInstance*2 << endl;  
    return 0;  
}
```

```
Width: 6  
Height: 10
```


重載運算子

```
class rectangle{  
    friend ostream& operator<<(ostream&, rectangle);  
private:  
    float width;  
    float height;  
};
```

`cout << rectangle`

```
ostream& operator<<(ostream& os, rectangle r){  
    os << "Width: " << r.width << endl;  
    os << "Height: " << r.height << endl;  
    return os;  
}
```

為何要回傳ostream?

重載運算子

```
cout << 變數1 << 變數2 << 變數3 << endl;
```

回傳
cout

回傳
cout

回傳
cout

重載運算子

```
class rectangle{  
    friend istream& operator>>(istream&, rectangle);  
private:  
    float width;  
    float height;  
};
```

`cin >> rectangle`

```
istream& operator>>(istream& is, rectangle r){  
    is >> r.width >> r.height;  
    return is;  
}
```

函式模板與類別模板

函式多載

同樣函式名稱，但具有不同的引數型態

- 因為引數型態不同，函式簽名也不同→可同時存在
- 但要寫三次很麻煩，有沒有辦法只寫一次？

```
void Funtcion(int *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

```
void Funtcion(float *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

```
void Funtcion(double *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

```
int Arr_1[3]={1,2,3};  
float Arr_2[3]={4.0,5.0,6.0};  
double Arr_3[3]={7.0,8.0,9.0};
```

Funtcion(Arr_1, 3);

Funtcion(Arr_2, 3);

Funtcion(Arr_3, 3);

模板 (Template)

想要設計函式，可以處理不同型態的引數？

- 一般函式 (Function)
 - 每個函式都取不同的名稱
- 函式重載 (Function Overloading)
 - 函式有相同的名稱，不同的引數型態
- 函式模板 (Function Template)
 - 把引數型態參數化，再由編譯器產生程式碼

函式模板 (Function Template)

函式模板 (Function Template)

1. 將資料型態參數化
2. 以簡化的符號表示，通常是 T, T1, T2, ...
3. 由編譯器透過類似巨集代換，根據內容產生程式碼

```
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;  
}
```

把資料型態變成變數

本門課不提

函式模板 (Function Template)

```
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;  
}
```

```
template <class T>  
bool isBigger(T a, T b){  
    return a>b;  
}
```

1. 以關鍵字 template 開頭

- 用 <> 把參數包起來
- 習慣上用 T, T1, T2 作為模板參數名稱

2. 再寫上函式的程式碼

- 又稱原型(prototype)
- 函式名稱即為該函式樣板的名稱

- 參數型態可用 **class** 或 **typename**
 - 表泛用型態 (即任何型態)
- 參數型態也可以用已宣告的資料型態
 - 如 int、float 或自定義 (如structure)

本門課不提

模板 (Template)

```
void Funtcion(int *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

```
void Funtcion(float *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

```
void Funtcion(double *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

- 函式間具有相同的函式名稱、實作方式
- 但不同的引數型態
- 把引數型態當作參數傳入

```
template<typename T>  
void Funtcion(T *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

本門課不提

模板 (Template)

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

```
template<typename T>
void Funtcion(T *p, int len){
    for(int i=0;i<len;i++)
        cout << *(p+i) << " ";
    cout << endl;
}
```

```
int main() {
    int arr_int[5] = {1,2,3,4,5};
    float arr_float[5] = {1.2,3.5,4.8,6.1,5.9};
    char arr_char[5] = {'a','b','c','d','e'};

    Funtcion(arr_int,5);
    Funtcion(arr_float,5);
    Funtcion(arr_char,5);
    return 0;
}
```

T = int

T = float

T = char

本門課不提

類別模板 (Class Template)

類別模板 (Class Template)

1. 將資料型態參數化
2. 以簡化的符號表示，通常是 T, T1, T2, ...
3. 由編譯器透過類似巨集代換，根據內容產生程式碼

```
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;
}
```

類別模板 (Class Template)

```
template <typename T>
class Data{
    private:
        T data;
    public:
        void set_data(T);
        T get_data();
};
```

```
int main() {
    Data<int> i;
    Data<float> f;
    Data<double> d;
    return 0;
}
```

T = int

T = float

T = double

類別模板 (Class Template)

```
template <typename T>
class Data{
    private:
        T data;
    public:
        void set_data(T);
        T get_data();
};
```

void Data::set_data(T a){...}



void Data<T>::set_data(T a){...}



```
template<typename T>
void Data<T>::set_data(T a){...}
```

類別模板 (Class Template)

類別模板 (Class Template) 下也可以有函式模板

```
template <typename T>
class Data{
    private:
        T data;
    public:
        void set_data(T);
        T get_data();
        template<typename T2> T2 sum(T2, T2);
};

template<typename T>
template<typename T2>
T2 Data<T>::sum(T2 a, T2 b){
    return a+b;
}
```

模板 (Template)

編譯器如何實作 template

只產生這三種！

```
int main() {  
    int arr_int[5] = {1,2,3,4,5};  
    float arr_float[5] = {1.2,3.5,4.8,6.1,5.9};  
    char arr_char[5] = {'a','b','c','d','e'};  
  
    Funtcion(arr_int,5);  
    Funtcion(arr_float,5);  
    Funtcion(arr_char,5);  
    return 0;  
}
```

產生

```
void Funtcion(int *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

產生

產生

```
void Funtcion(char *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

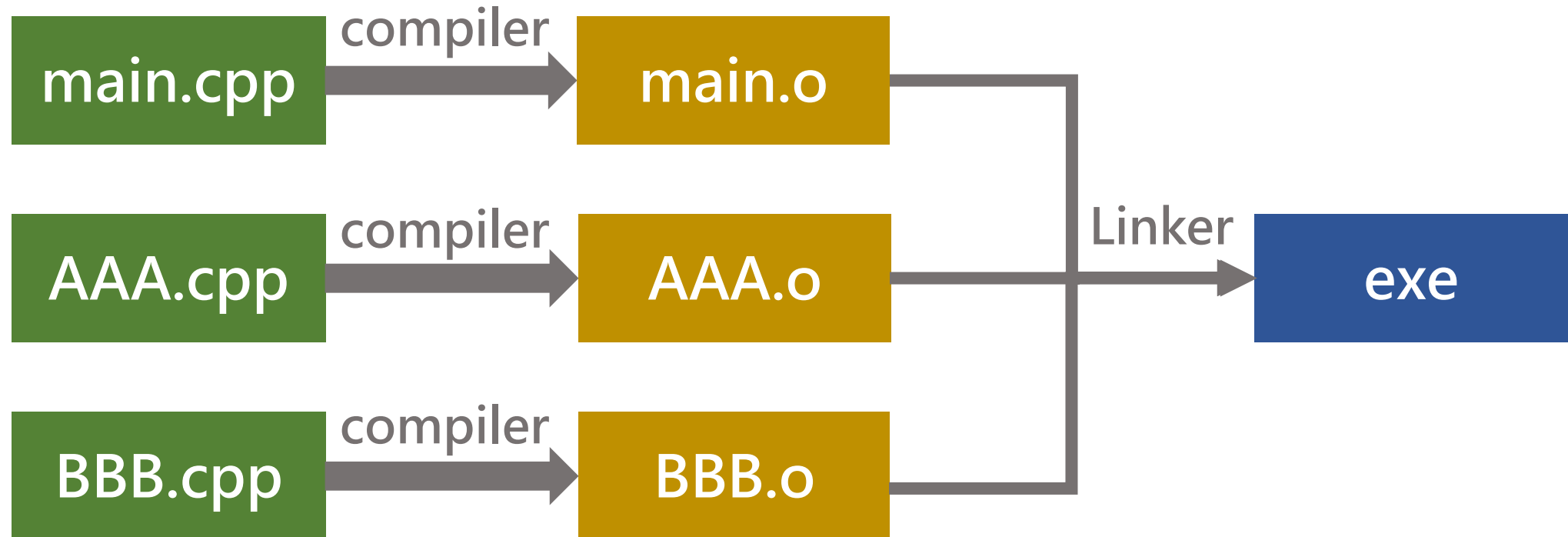
```
void Funtcion(float *p, int len){  
    for(int i=0;i<len;i++)  
        cout << *(p+i) << " ";  
    cout << endl;  
}
```

本門課不提

模板 (Template)

Template 是在編譯 (Compile) 階段生成，執行檔的大小沒有改變

- 有用到才會生成！



模板 (Template)

Template 的可視度 (Scope) 只有在 template 下的一個函式或類別中

```
template<typename T>  
T MIN(T a, T b){  
    return a<b?a:b;  
}
```

模板一

```
template<typename T>  
T MAX(T a, T b){  
    return a>b?a:b;  
}
```

模板二

模板 (Template)

Template 可以不換行

```
template<typename T>
void Funtcion(T *p, int len){
    for(int i=0;i<len;i++)
        cout << *(p+i) << " ";
    cout << endl;
}
```

```
template<typename T> void Funtcion(T *p, int len){
    for(int i=0;i<len;i++)
        cout << *(p+i) << " ";
    cout << endl;
}
```

模板 (Template)

Template 與函式間不能有任何指令

```
template<typename T>
void Funtcion(T *p, int len){
    for(int i=0;i<len;i++)
        cout << *(p+i) << " ";
    cout << endl;
}
```

錯誤！

```
template<typename T>
int a;
void Funtcion(T *p, int len){
    for(int i=0;i<len;i++)
        cout << *(p+i) << " ";
    cout << endl;
}
```

模板 (Template)

在找不到對應的函式時才會生成
(候補選手)

```
template<typename T>
T sum(T a, T b){
    cout << "Called by template function." << endl;
    return a+b;
}
```

```
int sum(int a, int b){
    cout << "Called by normal function." << endl;
    return a+b;
}
```

```
int main() {
    cout << sum(3,5) << endl;
    cout << sum(1.8,2.7) << endl;
    return 0;
}
```

已有直接用

沒有→透過 template 生成

標頭檔的建立

標頭檔

把函式拆解成宣告與定義兩個部分

- 宣告—函式的介面
 - 定義—函式的實作
-
- 宣告放在header file中(.h)
 - 定義放在cpp file中(.cpp)
 - 需要用到就include header file

標頭檔

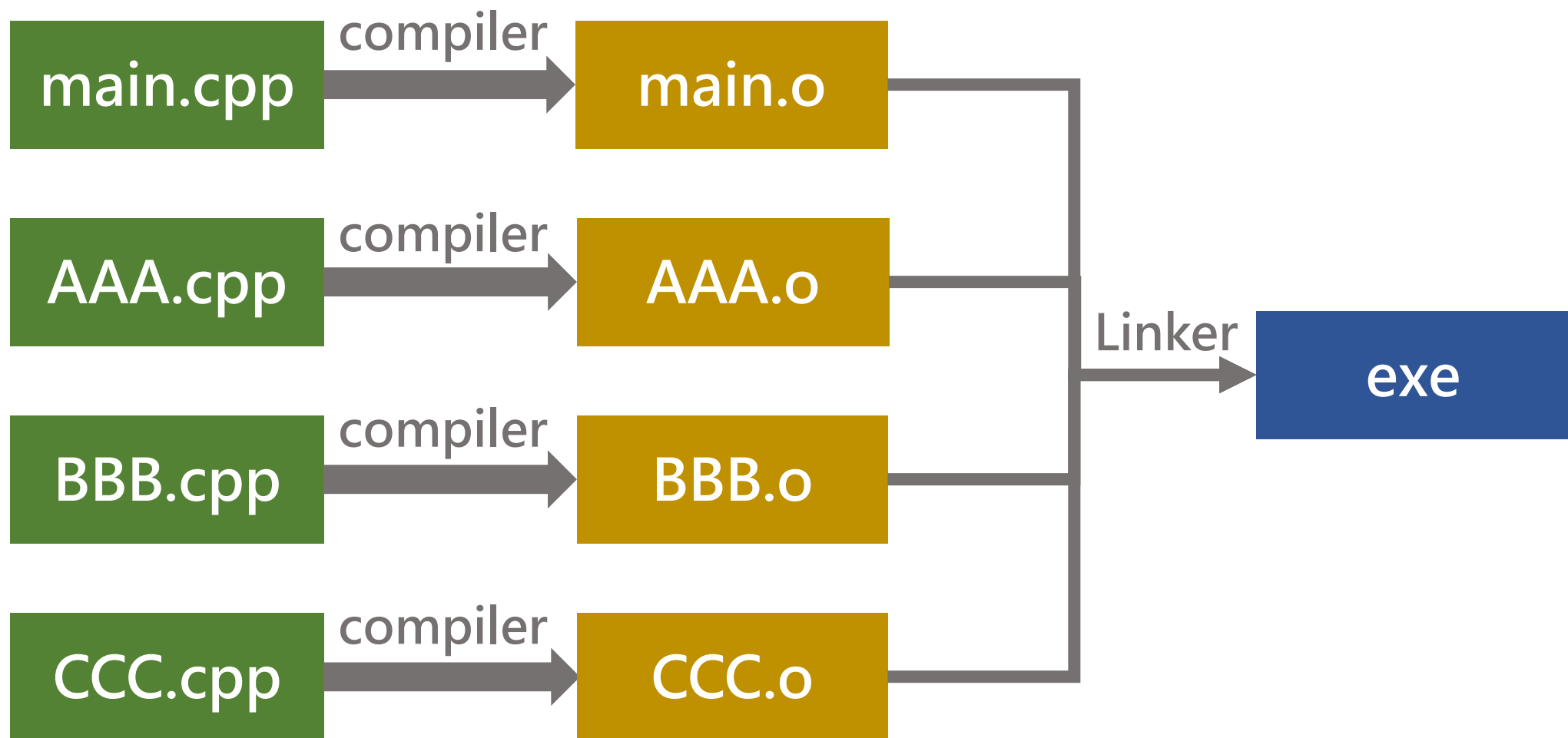
```
ostream& operator<< (ostream &, Student);  
bool operator>(Student, Student);  
bool operator<(Student, Student);
```

宣告(.h)

```
ostream& operator<< (ostream &os, Student s){  
    os << "Name: " << s.name << endl;  
    os << "English Score: " << s.englishScore << endl;  
    os << "Math Score: " << s.mathScore << endl;  
    return os;  
}  
  
bool operator>(Student s1, Student s2){  
    if(s1.englishScore+s1.mathScore>s2.englishScore+s2.mathScore)  
        return true;  
    else  
        return false;  
}  
  
bool operator<(Student s1, Student s2){  
    if(s1.englishScore+s1.mathScore<s2.englishScore+s2.mathScore)  
        return true;  
    else  
        return false;  
}
```

定義
(.cpp)

標頭檔



標頭檔

宣告放在.h

定義放在.cpp

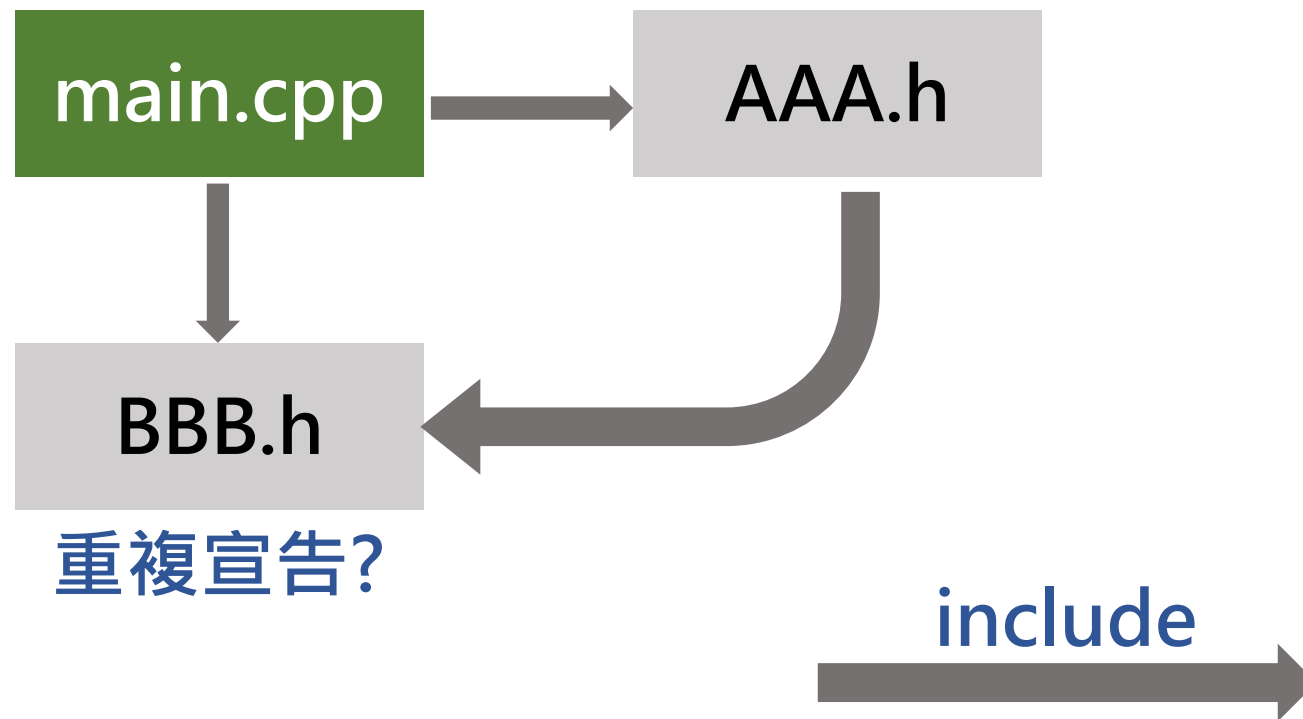
標頭檔的位置在資料夾內要用" "

- #include "file.h "
 - ✓ 資料夾內
- #include <file.h>
 - ✓ Compiler的路徑內

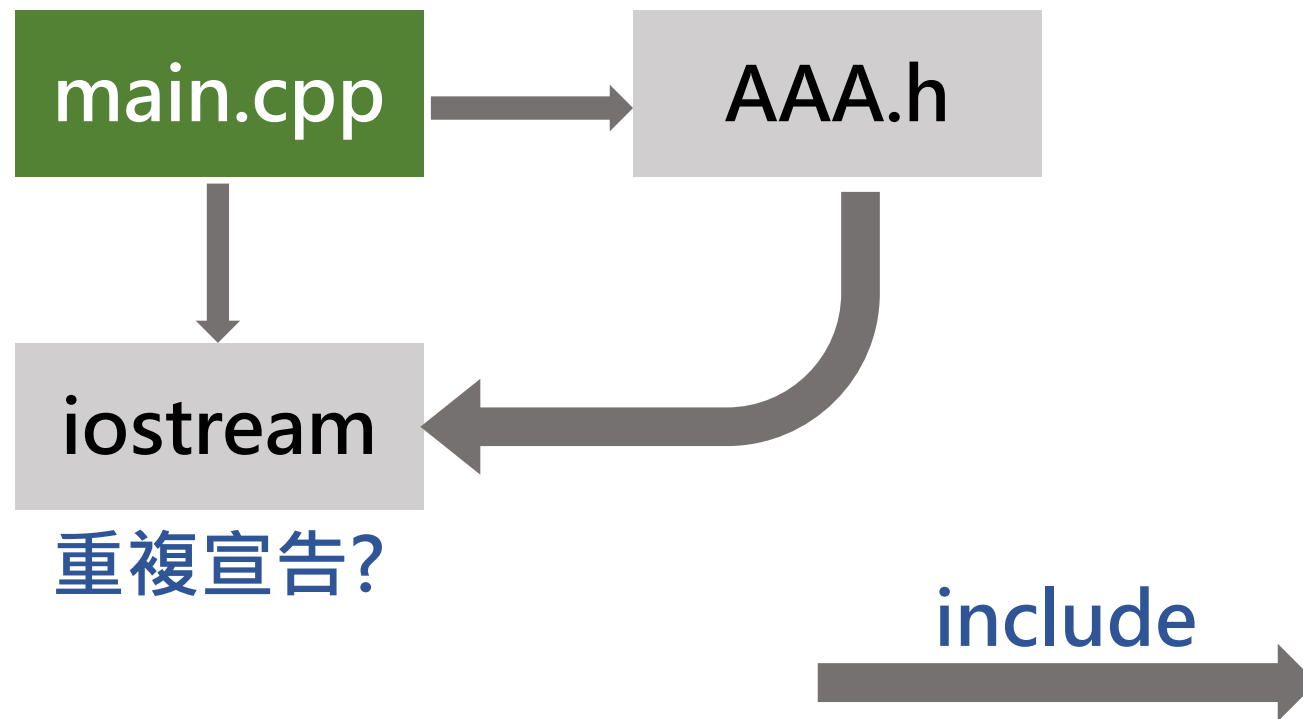
使用ifndef避免重複宣告

拆解方式同我們上次提到的

標頭檔



標頭檔



ifndef, define

- 留下足跡表示這段程式碼已經被include過
- ifndef = if not define
- 在header file中留下麵包屑！



```
#ifndef XXXXX_H_INCLUDED  
#define XXXXX_H_INCLUDED
```

要宣告的函式

```
#endif
```

ifndef, define

第一次include

- 關鍵字沒有被定義過
- #define 關鍵字
- 宣告函式

第二次include

- 關鍵字被定義過了(代表來過了)
- 跳到endif結束

```
#ifndef XXXXX_H_INCLUDED  
#define XXXXX_H_INCLUDED
```

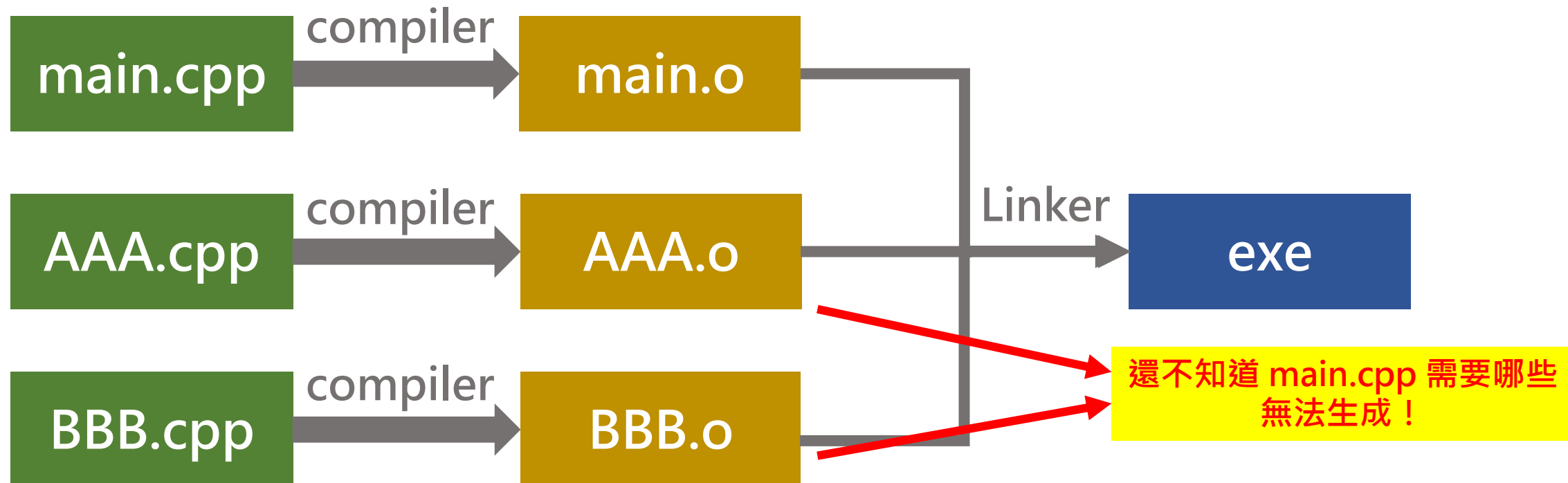
要宣告的函式

```
#endif
```

模板 (Template)

Template 是在編譯 (Compile) 階段生成，執行檔的大小沒有改變

- 有用到才會生成！template 不能放在 .cpp 下！
- template 請放在 .h 下



模板 (Template)

Template 是在編譯 (Compile) 階段生成

- 有用到才會生成！template 不能放在 .cpp 下！
- 或在 .cpp 檔下指定要生成哪些 data type 的函式(危險)

main.cpp

```
int main() {  
    Data<int> d;  
    d.set_data(2);  
    cout << d.get_data();  
    沒問題，有這個類別！  
    Data<float> d2;  
    d2.set_data(2.5);  
    cout << d.get_data();  
    不行，沒有這個類別！  
    return 0;  
}
```

data.h

```
template <typename T>  
class Data{  
    private:  
        T data;  
    public:  
        void set_data(T);  
        T get_data();  
};
```

data.cpp

```
template<typename T>  
void Data<T>::set_data(T input){  
    data = input;  
}  
  
template<typename T>  
T Data<T>::get_data(){  
    return data;  
}  
    強制生成 int 型態的 class  
template class Data<int>;
```

總結

宣告放在.h

定義放在.cpp(模板除外)

標頭檔的位置在資料夾內要用" "

- #include "file.h"

- ✓資料夾內

- #include <file.h>

- ✓Compiler的路徑內

使用ifndef避免重複宣告

Overview

set_lecturer

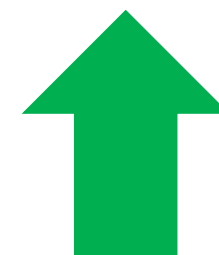
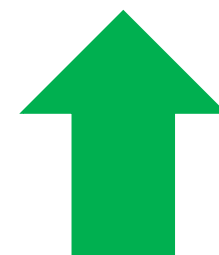
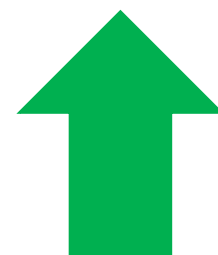
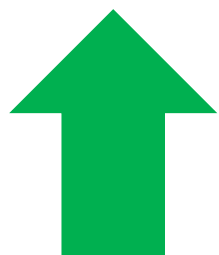
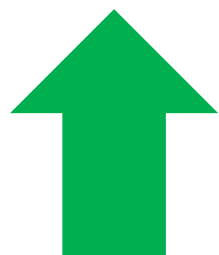
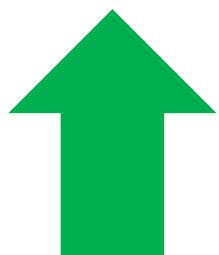
get_lecturer

average

<

>

.....



```
string lecturer;  
int student_counts;  
float *grade_list;
```

因為時間限制，只有這堂課會寫完整的 ADT

Example Code

Mission

重載人的類別中的運算子：

1. 重載 `<<` 和 `>>` 運算子，讓類別可以直接進行輸入與輸出。
2. 重載 `+` 運算子，回傳這兩個人的體重和

Example Code

Mission

把人的類別改成模板，使其可以指定身高與體重的資料型別

Example Code

Mission

1. 把程式碼切割成標頭檔(.h)與原始碼 (.cpp)
2. 試著在 main.cpp 中呼叫該類別

Practice

Mission

重載成績類別中的運算子：

1. 重載 `<<` 和 `>>` 運算子，讓類別可以直接進行輸入與輸出。
2. 重載 `[]` 運算子，回傳特定編號學生的成績

Practice

Mission

把成績類別改成模板，使其可以指定成績的資料型別

Practice

Mission

1. 把程式碼切割成標頭檔(.h)與原始碼 (.cpp)
2. 試著在 `main.cpp` 中呼叫該類別

Take Home Message

- Abstract Data Type (ADT) 有哪些意涵？目的為何？
- 要如何在類別外定義函式？
- 有哪些權限可以設定？兩者有甚麼差異？
- 建構式與解構式分別負責甚麼？甚麼時候會觸發？
- 運算子重載要如何進行？
- 模板(Template)的功能為何？
- .h 與 .cpp 如何分工？