

資料結構與C++進階班 類別

講師：黃銀鵬

E-mail: yinpenghuang@gmail.com

物件導向 (Object-Oriented Programming, OOP)

▶ 封裝

- ▶ 將變數及函式包裝進**Class**成為其中的成員，並且針對實際需求設定存取權限。
- ▶ 好處：程式碼共用、容易管理、符合最小權限原則。

▶ 繼承

- ▶ 父類別將符合存取權限的成員交給子類別，子類別可根據實際需求擴展(修改函式功能)或是新增(增加成員)。

▶ 多型

- ▶ 父類別指標新增子類別實體，以支援動態時期型(跑程式時)別決定。

類別與物件

Class describes object

- ▶ 類別就是物件的藍圖，物件就是類別的實體，利用類別建立大量的實體(宣告變數)。
- ▶ 實體以是否占用記憶體區塊為判定標準。
 - ▶ `CClassNmae className;` ✓
 - ▶ `CClassNmae className[10];` ✓
 - ▶ `CClassNmae* pClassName;` x 不是放class的，本身指標不是物件
 - ▶ `CClassNmae* pClassName = new CClassNmae;` ✓
- ▶ 所有類別都包含狀態與行為。 要會用!
 - ▶ 狀態(特徵)：成員變數。
 - ▶ 行為：成員函式。
- ▶ 成員存取範圍
 - ▶ **public**：公開的成員，不只能直接透過物件來呼叫使用，也可以在繼承了該類別的衍生類別中使用。
 - ▶ **protected**：受保護的成員，繼承了基底類別的衍生類別，能夠直接存取呼叫基底類別中的成員。
 - ▶ **private**：私用成員，只能在類別物件中使用，不能直接透過物件來呼叫使用，而即使是繼承了該類別的衍生類別也是如此。
- ▶ 結構(struct)在C++被視為全成員存取範圍為public的類別。

沒有指向任何地方
指向記憶體0的位置，但指向0的位置一定會被弄掉，所以其實是沒有指向任何處的指標

直系子孫，
不只包含
兒子！

最小權限
原則

C內用函數指標拿
C++內可struct

類別宣告

前贅詞，好辨識，
避免命名衝突

- CClassName.h

```
class CClassName
: Public CFatherClass {
public:
    CClassName ();
    ~ CClassName ();
    FunctionA();
    成員變數A;
protected:
    FunctionB ();
    成員變數B;
private:
    FunctionC ();
    成員變數C;
};
```

- CClassName.cpp

```
#include "CClassName.h"

CClassName::CClassName ()
:成員變數A(初始值)
,成員變數B(初始值)
,成員變數C(初始值) {
    行為內容
}

CClassName::~~CClassName () {
    行為內容
}

CClassName::FunctionA () {
    行為內容
}

CClassName::FunctionB () {
    行為內容
}

CClassName::FunctionC () {
    行為內容
}
```

因為沒有產生實體，所以沒有讓成員函數產生數字，但新的c++的可以

成員初始化順序與在class內宣告的順序相同，不相依的用member initialize，相依的要用命令列的initialize。

建構子與解構子

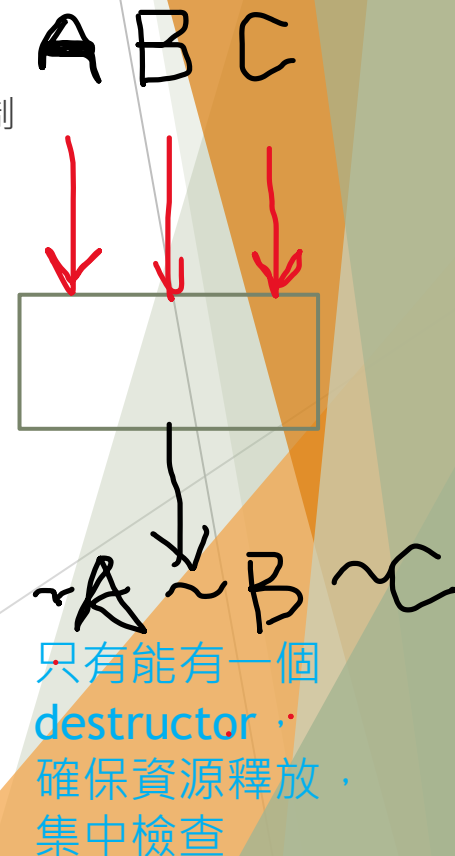
► 建構子(建構函式)：

- 與類別同名的函式。
- 沒有回傳資料型別，連void也沒有。
- 於類別宣告為變數時自動被呼叫，並且保證第一個被呼叫。
- 負責類別初始化動作。
 - 成員變數初始化、動態記憶體配置、開啟檔案、取得硬體控制權...等。
- 允許函式重載(同名異式)。

► 解構子(解構函式)：

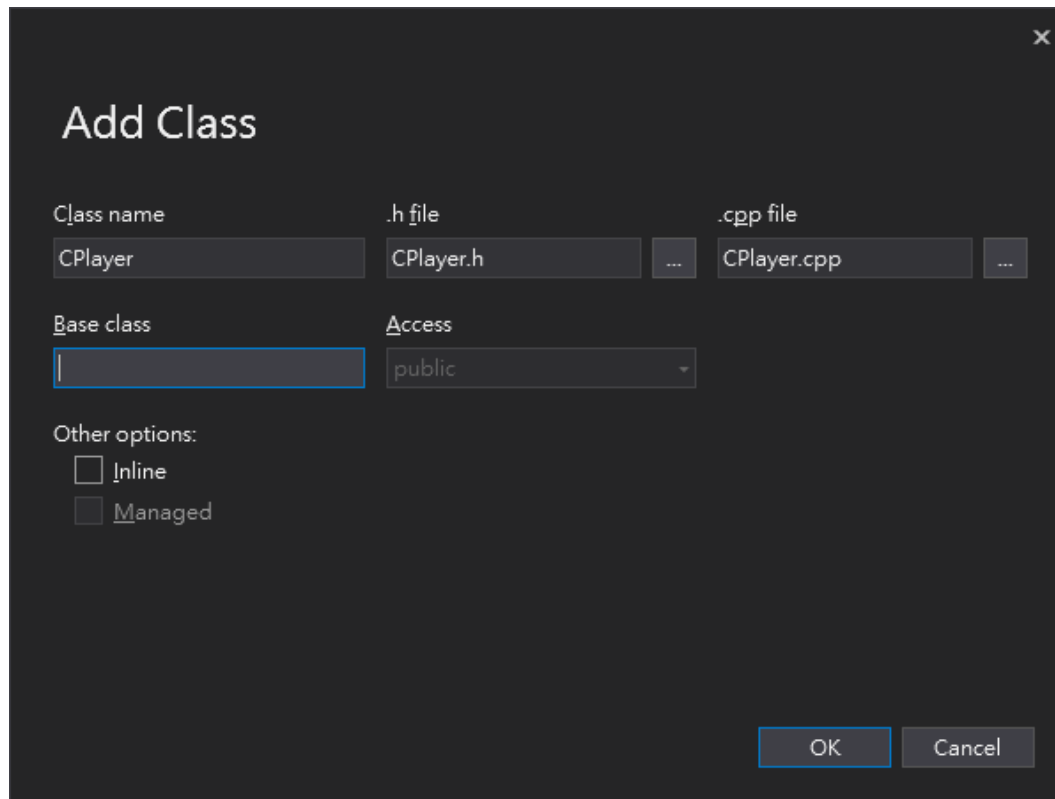
- 與類別同名，但是函式前有“~”。
- 沒有回傳資料型別，連void也沒有。
- 於變數消失前自動被呼叫，並且保證最後一個被呼叫。
- 負責持有資源的釋放。
 - 釋放記憶體、關閉檔案、歸還硬體控制權...等。
- 不允許函式重載。

main是最初的進入點，之後才可分岔出去，**constructor**在main後面，所以才可多建構子(**overloading**)，**資源運用最小化**。



新增類別

- ▶ Menu -> Project -> Add Class...
- ▶ 需要用手動輸入程式碼的方式新增建構子與解構子。



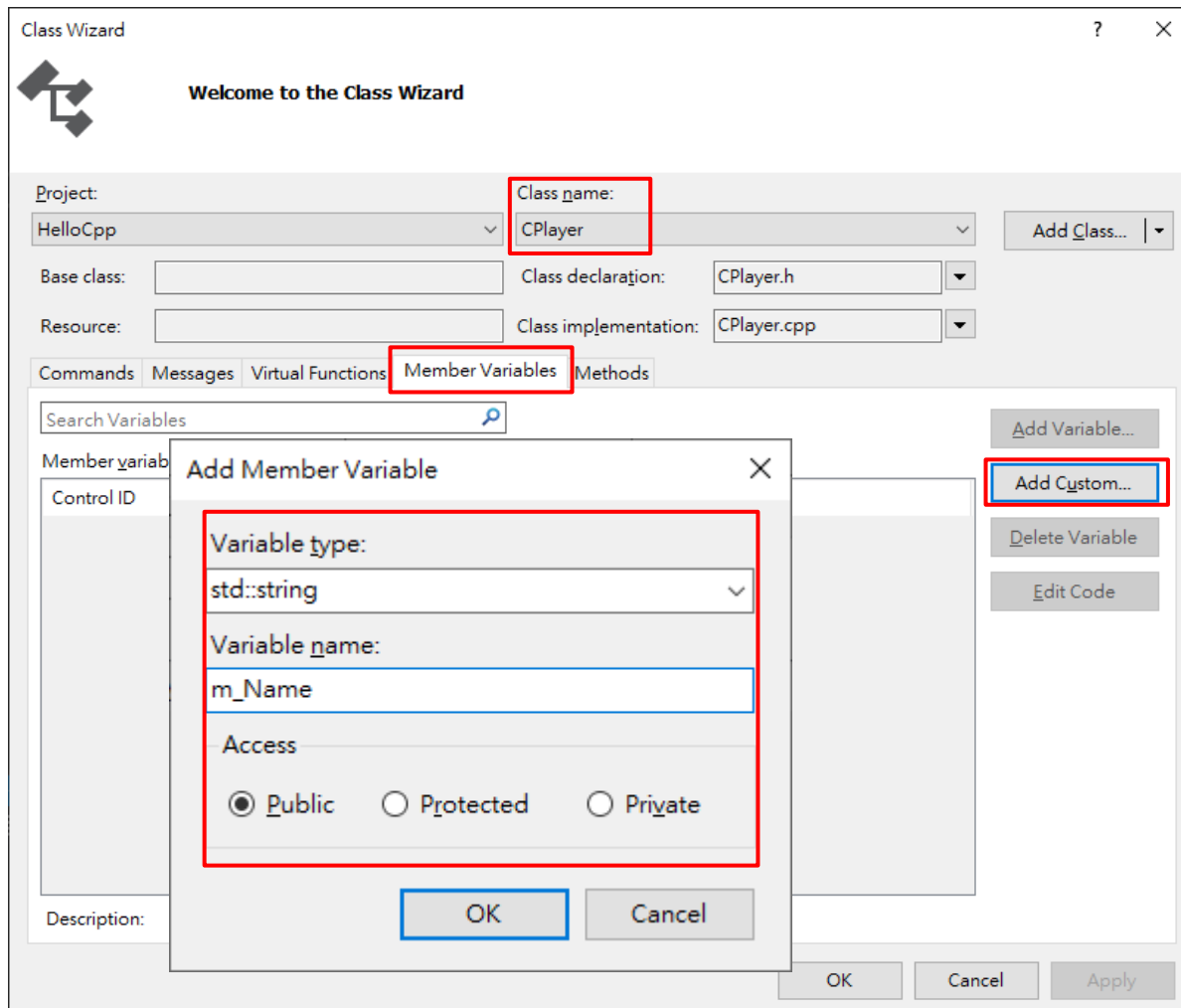
The screenshot shows a dark-themed 'Add Class' dialog box. It contains the following fields and options:

- Class name:** A text box containing 'CPlayer'.
- .h file:** A text box containing 'CPlayer.h' with a browse button ('...') to its right.
- .cpp file:** A text box containing 'CPlayer.cpp' with a browse button ('...') to its right.
- Base class:** An empty text box.
- Access:** A dropdown menu currently set to 'public'.
- Other options:**
 - ☐ **I**nline
 - ☐ **M**anaged

At the bottom right, there are 'OK' and 'Cancel' buttons.

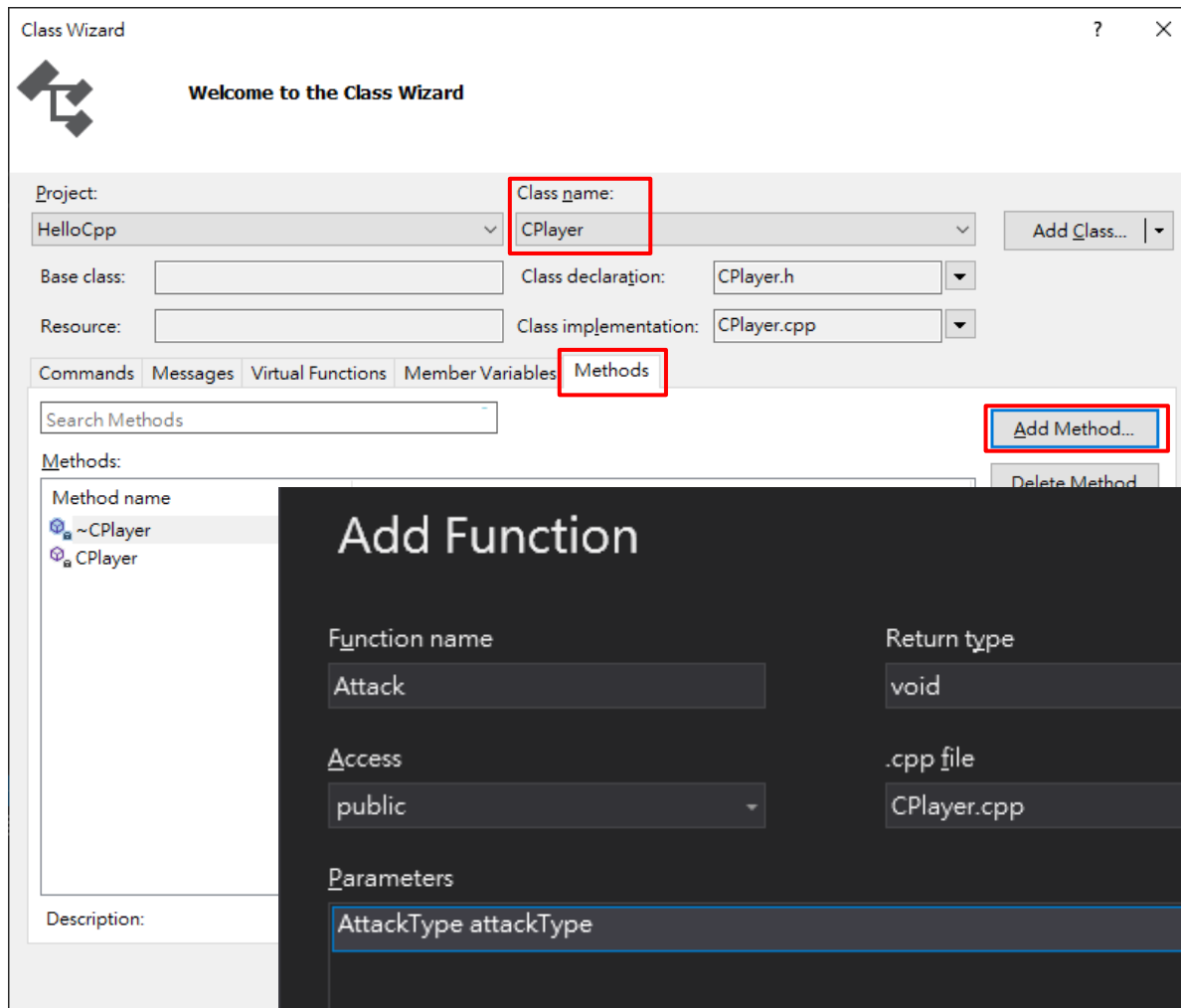
新增成員變數

- Menu -> Project -> Class Wizard...



新增成員函示

- ▶ Menu -> Project -> Class Wizard...



01- 玩家角色

```
#include <iostream>
#include "CPlayer.h"
int main()
{
    CPlayer player;
    int occupation;
    std::cout << "請選擇角色職業" << std::endl;
    std::cout << "1.戰士\t2.精靈\t3.法師\n";
    std::cin >> occupation;
    player.SetOccupation((Occupation)occupation);
    std::cout << "請選擇角色名稱(不超過10中文字):" << std::endl;
    std::cin >> player.m_Name;
    std::cout << std::endl;
    player.Attack(smite);
    player.Attack(hit);
    return 0;
}
```

Microsoft Visual Studio ...

請選擇角色職業
1.戰士 2.精靈 3.法師
:1
請選擇角色名稱(不超過10中文字):
狂戰士

狂戰士使出重擊(旋風斬)
狂戰士使出普通攻擊(斬擊)

Microsoft Visual Studio ...

請選擇角色職業
1.戰士 2.精靈 3.法師
:2
請選擇角色名稱(不超過10中文字):
黑暗精靈

黑暗精靈使出重擊(連續射擊)
黑暗精靈使出普通攻擊(射擊)

Microsoft Visual Studio ...

請選擇角色職業
1.戰士 2.精靈 3.法師
:3
請選擇角色名稱(不超過10中文字):
死靈法師

死靈法師使出重擊(豪火球術)
死靈法師使出普通攻擊(火球術)

```
#pragma once
#include <string>
enum Occupation
{
    none,
    warrior = 1, //戰士
    elf, //精靈
    wizard //巫師
};

enum AttackType
{
    smite, //重擊
    hit //普通攻擊
};

class CPlayer
{
public:
    CPlayer();
    virtual ~CPlayer();
    void Attack(AttackType attackType);
    void SetOccupation(Occupation occupation);
    std::string m_Name;
private:
    Occupation m_Occupation;
};
```

有兒子的加virtual

列舉(enum)不行直接設置

```
#include "CPlayer.h"
#include <iostream>

CPlayer::CPlayer()
    : m_Occupation(none)
    , m_Name("")
{
}

CPlayer::~CPlayer()
{
}

void CPlayer::Attack(AttackType attackType)
{
    std::cout << m_Name;
    switch (m_Occupation) {
        case warrior:
            if (attackType == smite)
                std::cout << "使出重擊(旋風斬)" << std::endl;
            else if (attackType == hit)
                std::cout << "使出普通攻擊(斬擊)" << std::endl;
            break;
        case elf:
            if (attackType == smite)
                std::cout << "使出重擊(連續射擊)" << std::endl;
            else if (attackType == hit)
                std::cout << "使出普通攻擊(射擊)" << std::endl;
            break;
        case wizard:
            if (attackType == smite)
                std::cout << "使出重擊(豪火球術)" << std::endl;
            else if (attackType == hit)
                std::cout << "使出普通攻擊(火球術)" << std::endl;
            break;
    }
}

void CPlayer::SetOccupation(Occupation occupation)
{
    m_Occupation = occupation;
}
```

constructor要存在

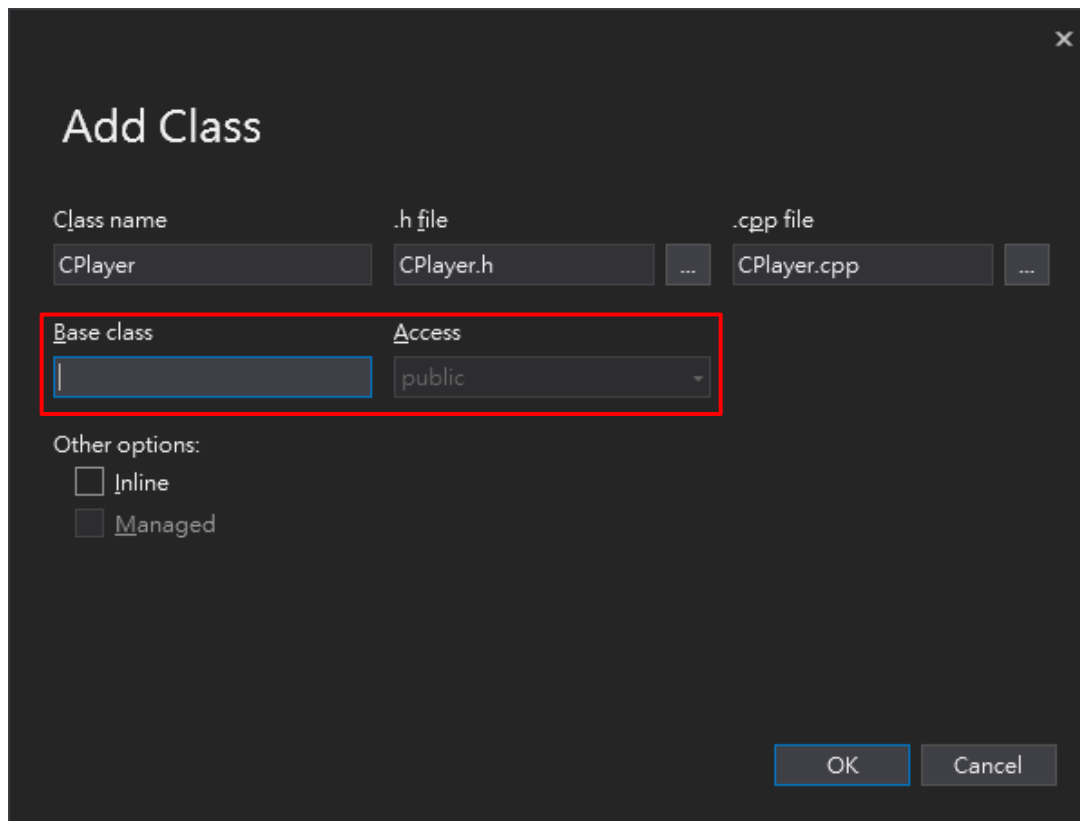
組合與繼承

- ▶ 組合：在類別內使用既有類別所產生的物件(宣告變數)。
 - ▶ 組件：類似零件的概念，在完整的物件中所包含的另一種完整物件，透過操作此物件使設計的類別功能完整。
 - ▶ 聚集：多個同類型的零件在類別中被使用。
 - ▶ 類別建構比較加
 - ▶ 多用組合少用繼承
- ▶ 繼承：承接既有類別的架構，並增加新的成員或修改既有成員。**增加耦合性(關聯性)，但會降低維護性。**

繼承方式

- ▶ **public**：父類別所有成員存取範圍不變。
- ▶ **protected**：父類別**public**成員存取範圍變**protected**，其餘不變。
- ▶ **private**：父類別**public**及**protected**成員存取範圍變**private**，其餘不變。

類別繼承



The image shows a 'Add Class' dialog box with a dark background. It contains fields for 'Class name' (CPlayer), '.h file' (CPlayer.h), and '.cpp file' (CPlayer.cpp). A red rectangle highlights the 'Base class' and 'Access' fields. The 'Base class' field is empty, and the 'Access' dropdown is set to 'public'. Below these are 'Other options' with checkboxes for 'Inline' and 'Managed'. 'OK' and 'Cancel' buttons are at the bottom right.

Add Class

Class name: CPlayer

.h file: CPlayer.h

.cpp file: CPlayer.cpp

Base class:

Access: public

Other options:

- ☐ Inline
- ☐ Managed

OK Cancel

多型的條件(缺一不可)

- ▶ 父類別的指標new子類別的實體。
- ▶ 父類別的解構子必須是虛擬的。
- ▶ 支援多型的函式必須父子都要有。(函式要一模一樣)
- ▶ 支援多型的函式必須是虛擬的。
- ▶ virtual會一直下去

02- 玩家角色(多型)

在switch內只
能用多型

```
#include <iostream>
#include "CWarrior.h"
#include "CElf.h"
#include "CWizard.h"

int main()
{
    CPlayer* player = NULL;    好習慣NULL
    int occupation;
    std::cout << "請選擇角色職業" << std::endl;
    std::cout << "1.戰士\t2.精靈\t3.法師\n";
    std::cin >> occupation;
    switch (occupation) {
        case warrior:
            player = new CWarrior;    父指標new子類別
            break;
        case elf:
            player = new CElf;
            break;
        case wizard:
            player = new CWizard;
            break;
    }
    std::cout << "請選擇角色名稱(不超過10中文字):" << std::endl;
    std::cin >> player->m_Name;
    std::cout << std::endl;
    player->Attack(smite);
    player->Attack(hit);
    delete player;
    return 0;
}
```

m_Name是爸爸
function 是兒子的

```
#pragma once
#include <string>
#include <iostream>
enum Occupation
{
    warrior = 1,
    elf,
    wizard
};

enum AttackType
{
    smite,
    hit
};

class CPlayer
{
public:
    CPlayer();
    virtual ~CPlayer();
    virtual void Attack(AttackType attackType);
    std::string m_Name;
};
```

```
#include "CPlayer.h"

CPlayer::CPlayer()
    : m_Name("")
{
}

CPlayer::~~CPlayer()
{
}

void CPlayer::Attack(AttackType attackType)
{
    std::cout << m_Name;
}
```

多型函式(用多型方式呼叫)


```
#pragma once
#include "CPlayer.h"
class CWarrior :
{
    public CPlayer
{
public:
    CWarrior();
    ~CWarrior();
    void Attack(AttackType attackType);
};
```

增加了類別耦合性，但兒子間增加類別獨立性，要修改時，放在不同的class可避免動到其他東西

多型函式

沒有必要的話已經穩定的code不要改

沒有要用到多型，就多組合，要多型則大膽用繼承

```
#include "CWarrior.h"
CWarrior::CWarrior()
{
}

CWarrior::~CWarrior()
{
}

void CWarrior::Attack(AttackType attackType)
{
    CPlayer::Attack(attackType);
    if (attackType == smite)
        std::cout << "使出重擊(旋風斬)" << std::endl;
    else if (attackType == hit)
        std::cout << "使出普通攻擊(斬擊)" << std::endl;
}
```

```
#pragma once
#include "CPlayer.h"
class CElf :
    public CPlayer
{
public:
    CElf();
    ~CElf();
    void Attack(AttackType attackType);
};
```

```
#include "CElf.h"
CElf::CElf()
{
}

CElf::~CElf()
{
}

void CElf::Attack(AttackType attackType)
{
    CPlayer::Attack(attackType);
    if (attackType == smite)
        std::cout << "使出重擊(連續射擊)" << std::endl;
    else if (attackType == hit)
        std::cout << "使出普通攻擊(射擊)" << std::endl;
}
```

```
#pragma once
#include "CPlayer.h"
class CWizard :
{
    public CPlayer
{
public:
    CWizard();
    ~CWizard();
    void Attack(AttackType attackType);
};
```

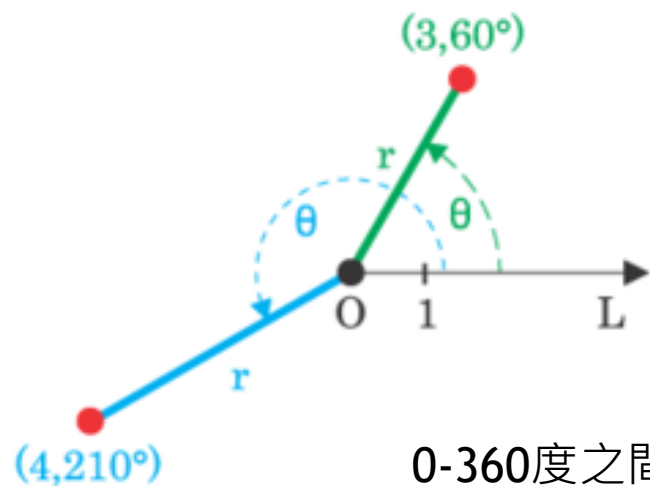
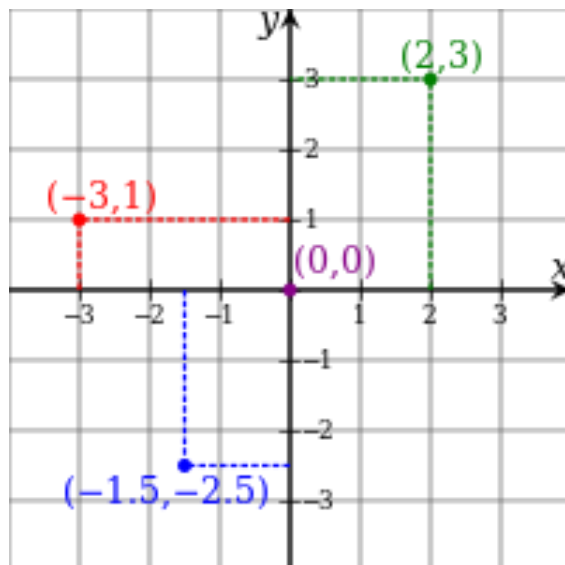
```
#include "CWizard.h"
CWizard::CWizard()
{
}

CWizard::~~CWizard()
{
}

void CWizard::Attack(AttackType attackType)
{
    CPlayer::Attack(attackType);
    if (attackType == smite)
        std::cout << "使出重擊(豪火球術)" << std::endl;
    else if (attackType == hit)
        std::cout << "使出普通攻擊(火球術)" << std::endl;
}
```

平面座標系

- ▶ 平面座標系分為”直角座標系”(Cartesian coordinate system)及”極座標系”(Polar coordinate system)。平面中的一點在直角座標系的座標是由該點於X軸的投影位置(x)及於Y軸的投影位置位置(y)所決定；而在極座標系是由該點與原點的距離(r)及該點與原點的連線相對於X軸的夾角(θ)所決定。



0-360度之間

作業：平面座標系

- ▶ 請撰寫三個類別。第一個類別為平面座標系類別，第二個為直角座標系類別，第三個為極座標系類別，平面座標系類別為另外兩個類別的父類別。
- ▶ 所有座標皆以整數表示，座標系必須具備鏡像的功能，且必須支援多型技術。
- ▶ 不給測資。
- ▶ 請問你要使用哪種系統，輸入點為何， x 、 y 軸都要鏡向出來的點座標為何
- ▶ 自己設計測資證明自己是Ok的，多式幾個，截圖沒問題

想想看

配置給變數的記憶體在編譯時期就要能確定，也就是說寫程式時當下確定，記憶體大小跟資料型別有關，就是要確定資料型別，實際上就是看記憶體大小。

右圖宣告a為A型態，但要先看A的內容為何，才能決定配給a的大小，但是決定A的內容時又要看a的大小，所以a看A，A又要看a，導致報錯。

```
class A
{
    A a; ✗
    A* pa; ✓
}
```

- ▶ 類別內是否可以用自己宣告變數；類別內是不可以用自己宣告指標。(見右上解說)
- ▶ 是否可以不用指標及多型，而是根據不同的輸入宣告不同類別的變數來使用。(可以，但不好管理)
- ▶ 使用多型時，解構子是否一定要是虛擬函式。(是)
大型的軟體系統，用多型較好管理

B b; 子先解構，父再解構，若相反，
則父親則沒辦法完全釋放空間。

B* pb = new B;

A* pa = new B;

若沒有virtual，則
~B()就沒有被呼叫
到，而是只有~A()