

Ver.3

2019.09 bug fixed

# Object-Oriented Programming for Beginners

期中後正課總複習

Yu-Hsuan Chen

June 2019

# Outline

- Reference(參考)
- 物件與類別
- 繼承與多型
- 運算子多載
- 樣板
- 例外處理

# 參考 Reference

期中考那一天，大家都想起了\*\*的恐懼

# 參考 Reference

期中考那一天，大家都想起了\*\*的恐懼

我是說指標(pointer)啦，想歪的給我去面壁哦

# Why Reference

- 看膩...阿不是，覺得C裡面大量的指標看了很煩躁嗎？

```
void delete(Node **head, string target)
{
    if((*head)->name == target)
    {
        Node *temp = *head;
        (*head) = (*head)->next;
    }
    else
    {
        Node *current = *head;
        while(current->next !=NULL)
            if(current->next->name != target)
                current = current->next;
        Node *temp = current->next;
        current->next = temp->next;
    }
    delete temp;
}
```



p5-Linked List Intro.pdf

# Reference

- C++引入Reference 就是為了減少指標傳遞的麻煩
- 參考可以看成是「不用 & 跟 \* 運算子的指標變數」
- 參考也是「另一個變數的別名(Alias)」
- 怎麼說..?

# Reference (cont.)

# Reference (cont.)

- 參考變數是「另一個變數的別名」

```
int x;  
int &y = x;
```

# Reference (cont.)

- 參考變數是「另一個變數的別名」

```
int x;  
int &y = x;
```

- 上面的宣告之後，程式碼可以變成這樣

```
y = 10;           // 相當於x = 10;  
scanf( "%d" , &y); // 相當於scanf( "%d" ,&x);
```

# Reference (cont.)

- 參考變數是「另一個變數的別名」

```
int x;  
int &y = x;
```

- 上面的宣告之後，程式碼可以變成這樣

```
y = 10;           // 相當於x = 10;  
scanf( "%d" , &y); // 相當於scanf( "%d" ,&x);
```

- 此時的y就代表了x

# Reference (cont.)

- 傳遞參數可以更方便

```
void Timer::setValue(Timer& T)
{
    this->min = T.min;
    this->sec = T.sec;
}
```

這個時候的T就是Call by reference，  
T不會在呼叫setValue的時候複製一份，  
T是唯一的。

# Reference (cont.)

- 傳遞參數可以更方便

```
void Timer::setValue(Timer& T)
{
    this->min = T.min;
    this->sec = T.sec;
}
```

這個時候的T就是Call by reference，  
T不會在呼叫setValue的時候複製一份，  
T是唯一的。

當然，你如果手很忙...把T的min, sec給改了，那麼main裡面傳入的那個T也就跟著被改了。  
如何解決？

請善用const規範傳入的參數

# Reference的小知識

# Reference的小知識

- Reference不具備強制轉型的能力

```
int main(){
    int data;
    f(data);
}
```

```
void f(double &data){
    data=10;
}
//Error: cannot convert
'int' to 'double&'
```

# Reference的小知識

- Reference不具備強制轉型的能力

```
int main(){
    int data;
    f(data);
}
```

```
void f(double &data){
    data=10;
}
//Error: cannot convert
'int' to 'double&'
```

- Reference其實是Lvalue，而且Reference跟const變數一樣需要初始化

```
int x=10;
int &y; // Error: 'y' declared as reference but not initialized
y = x; // 這樣還是不行 (Initialization跟Assignment是不一樣的概念)
int &z = x; // 這樣才對
```

# Reference的小知識

- Reference不具備強制轉型的能力

```
int main(){
    int data;
    f(data);
}
```

```
void f(double &data){
    data=10;
}
//Error: cannot convert
‘int’ to ‘double&’
```

- Reference其實是Lvalue，而且Reference跟const變數一樣需要初始化

```
int x=10;
int &y; // Error: ‘y’ declared as reference but not initialized
y = x; // 這樣還是不行 (Initialization跟Assignment是不一樣的概念)
int &z = x; // 這樣才對
```

- Reference 無法重複指派(Reassigned)

```
int &aliasX = x;
int &aliasX = y; // Error: identifier ‘aliasX’ re-declared.
```

# Reference的小知識

- Reference不具備強制轉型的能力

```
int main(){
    int data;
    f(data);
}
```

```
void f(double &data){
    data=10;
}
//Error: cannot convert
‘int’ to ‘double&’
```

- Reference其實是Lvalue，而且Reference跟const變數一樣需要初始化

```
int x=10;
int &y; // Error: ‘y’ declared as reference but not initialized
y = x; // 這樣還是不行 (Initialization跟Assignment是不一樣的概念)
int &z = x; // 這樣才對
```

- Reference 無法重複指派(Reassigned)

```
int &aliasX = x;
int &aliasX = y; // Error: identifier ‘aliasX’ re-declared.
```

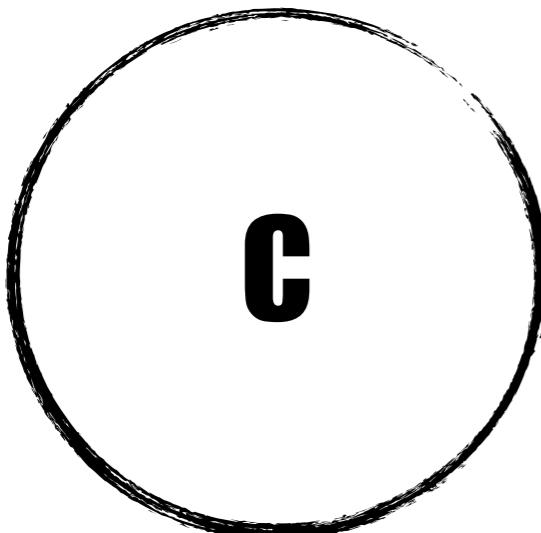
Lvalue: 運算式執行完之後還保留的物件

Rvalue: 運算式執行後不會保留的物件（即暫存的變數）

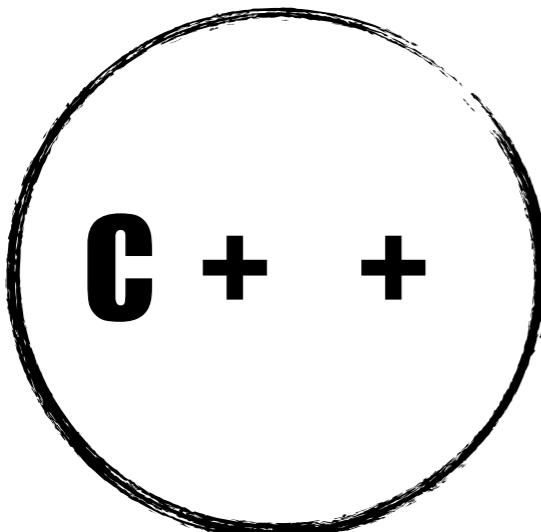
物件 Object

# 如何去看待物件導向？

助教個人的簡單比喻



C就像加法，少了什麼功能就加什麼

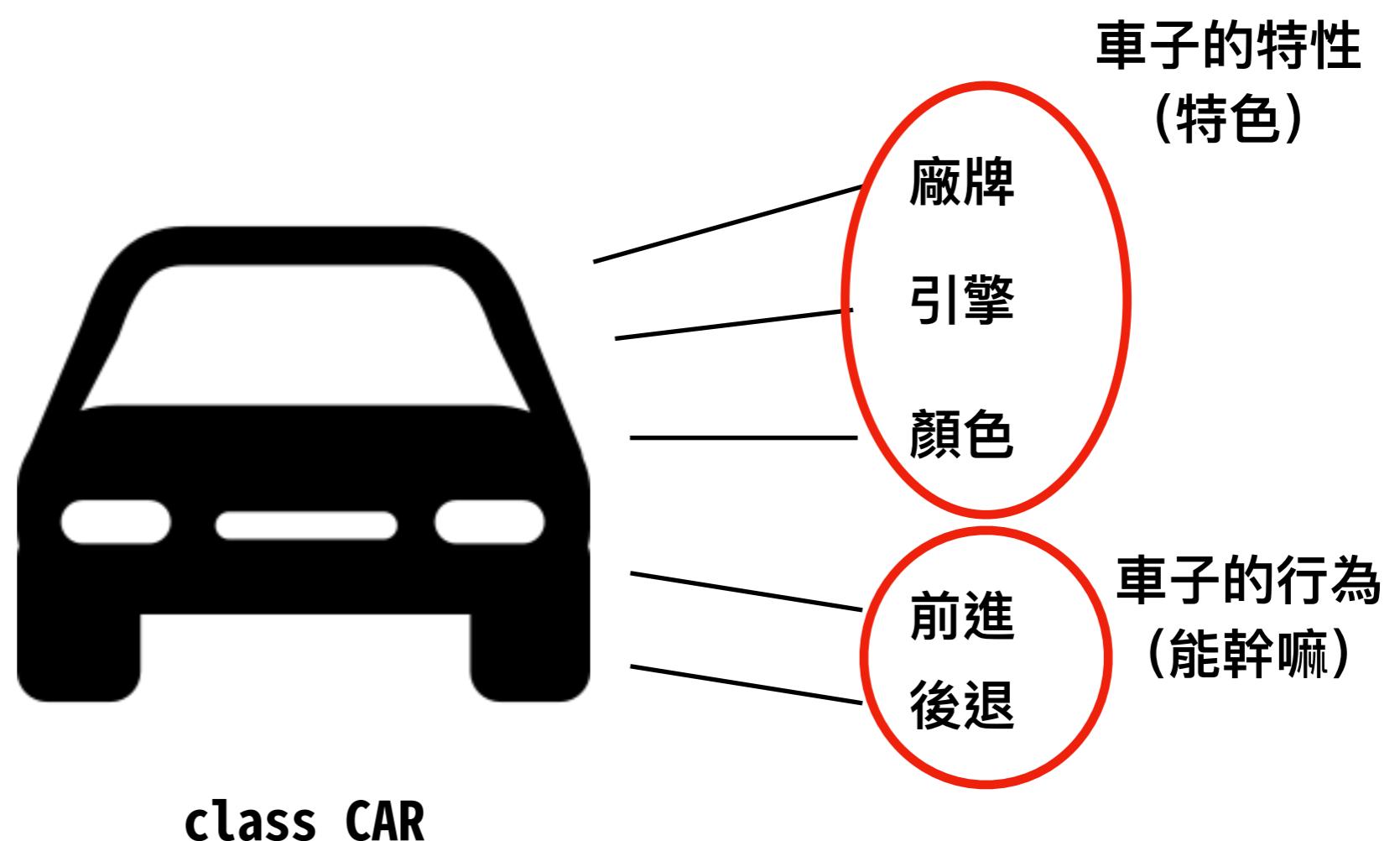


C++就像減法，單一物件功能越精簡越好

# 物件 Object

- 白話一點，他就是「東西」
- 物件的組成
  - 屬性(Attribute)：他有什麼特色、有什麼欄位
  - 方法(Method)：他能做什麼
- 先定義了類別(Class)，才會造出物件(Object)

# Example



# 這些都是CAR型態的物件

小客車



廠牌: 不重要

引擎: 1800cc

顏色: 星空灰

前進()

後退()

大客車



廠牌: 不重要

引擎: 13267cc

顏色: 藍

前進()

後退()

計程車



廠牌: 不重要

引擎: 2000cc

顏色: 黃

前進()

後退()

# 建構元&解構元

## constructor & destructor

- 都沒有「回傳型態」
- 名字則是跟類別的名字一樣  
解構元前面多加一個 ~
- 建構元→物件生成出來的時候呼叫
- 解構元→物件被回收時呼叫

# 建構元&解構元

## constructor & destructor

- 都沒有「回傳型態」
- 名字則是跟類別的名字一樣  
解構元前面多加一個 ~
- 建構元→物件生成出來的時候呼叫
- 解構元→物件被回收時呼叫

```
class Car
{
    public:
        Car();
        ~Car();
}
```

# 建構元&解構元

## constructor & destructor

- 都沒有「回傳型態」
- 名字則是跟類別的名字一樣  
解構元前面多加一個 ~
- 建構元→物件生成出來的時候呼叫
- 解構元→物件被回收時呼叫

```
class Car
{
    public:
        → Car();
        ~Car();
}
```

無論如何至少都要一個建構元，即使do nothing也可以  
這個特別取名叫做“預設建構元”(default constructor)  
當然你可以定義多個建構元，接受不同的參數  
具體的作法請參考的Overload的部分

# 建構元constructor (cont.)

```
//Car.h  
class Car  
{  
private:  
    int cc;  
public:  
    Car();  
    Car(int);  
    ~Car();  
};
```

```
//Car.cpp  
#include "Car.h"  
Car::Car():cc(0)  
{  
}  
  
Car::Car(int cc)  
{  
    this->cc = cc;  
}  
  
Car::~Car()  
{}
```

這個叫做initialize List  
把想要初始化的值放入  
初始化的順序會依照**class定義的  
變數順序逐個初始化**

this指標平常是可以不寫的  
但是如果像這個情況  
就要使用this來區分是哪一個cc

# 拷貝建構元 copy constructor

- 什麼 $\Sigma(\overset{\circ}{\Delta}; \equiv; \overset{\circ}{\Delta})$ 又是新的名詞
- 顧名思義，拷貝建構元可以複製一份相同的物件
- 宣告原型：

```
Car::Car(const Car &rhs)
```

```
{
```

```
//複製類別成員
```

```
}
```

- 如果沒有定義，Compiler會幫你做  
但是當你的類別裡有動態配置的變數時，你最好自己寫，免得Compiler自己做的拷貝建構元只有複製了記憶體位址，沒有配置新的空間，造成複製不完整。

# 靜態成員 Static Members

- 如果在定義class的內容時，加上了static描述，那麼這個東西將會成為靜態成員。
- 它不屬於任何一個物件，而是這個class所構成的物件去共享的變數

# 靜態成員 Static Members

- 如果在定義class的內容時，加上了static描述，那麼這個東西將會成為靜態成員。
- 它不屬於任何一個物件，而是這個class所構成的物件去共享的變數

```
class Car
{
    public:
        static string country;
}
```

# 靜態成員 Static Members

- 如果在定義class的內容時，加上了static描述，那麼這個東西將會成為靜態成員。
- 它不屬於任何一個物件，而是這個class所構成的物件去共享的變數

```
class Car
{
public:
    static string country;
}
```

```
int main()
{
    Car van, taxi;
    van.country = "Keelung";
    cout << Car::country << endl; //印出Keelung
    cout << taxi.country << endl; //還是Keelung
}
```

//別忘了在使用static變數之前要賦予必要的初始值

# friend (友誼)

- 從class的角度來看，class自己對所有的成員有100%控制權，但是在class之外，僅能看到public的成員及函式
- 透過friend可以允許指定的函數共用class裡的成員

```
class Rectangle
{
    private:
        int width, height;
    public:
        void setValue(int, int);
        friend Rectangle duplicate(Rectangle);
}
```

```
Rectangle duplicate(Rectangle R)
{
    Rectangle newRect;
    newRect.width = R.width;
    newRect.height = R.height;
    return newRect;
}
//加了friend，duplicate實際上就不是Rectangle裡面的成員了
//這就是加了friend為什麼不需要Rectangle::的原因
```

# friend (友誼) (cont.)

- class之間也是可以有friend關係的!
- friend的特性是單向的，沒有相互存取的概念

```
class Rectangle
{
    private:
        int width, height;
    public:
        void setValue(int, int);
        void setValue(Square);
        int getWidth();
}
```

```
Class Square
{
    private:
        int side;
    public:
        void setSide(int);
        friend class Rectangle;
        void doSomething(Rectangle);
}
//讓Rectangle可以看得到Square的東西
//但是Square看不到Rectangle
```

```
void Rectangle::setValue(Square s)
{
    this->width = s.side;
    this->height = s.side;
}
```

```
void Square::doSomething(Rectangle r)
{
    cout << r.getWidth();
}
```

# 覆載 overload

- C語言中不允許同名的Function

```
void fun();  
void fun(int, int); //NO WAY!!
```

- C++則允許同名函式接受不同的參數  
(因為C++分辨函式的方法是透過名字跟參數)

- 舉例

```
void FUN(int ,int);  
void FUN(double, double);  
void FUN(int);
```

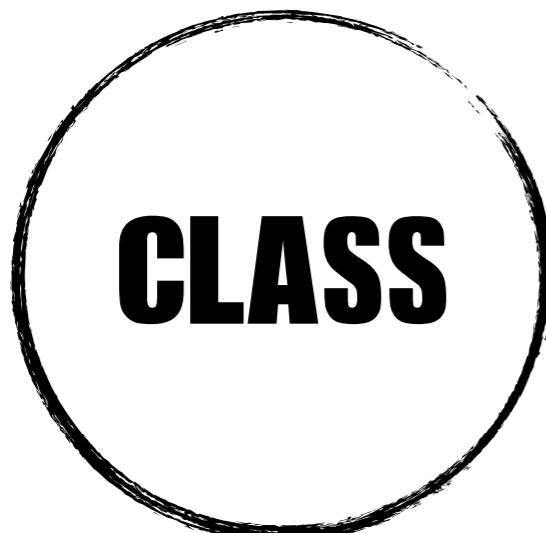
FUN(1, 1);	//呼叫int, int
FUN(1.2, 1.5);	//呼叫double, double
FUN(3);	//呼叫int

- 定義多個建構元裡面也有overload的概念

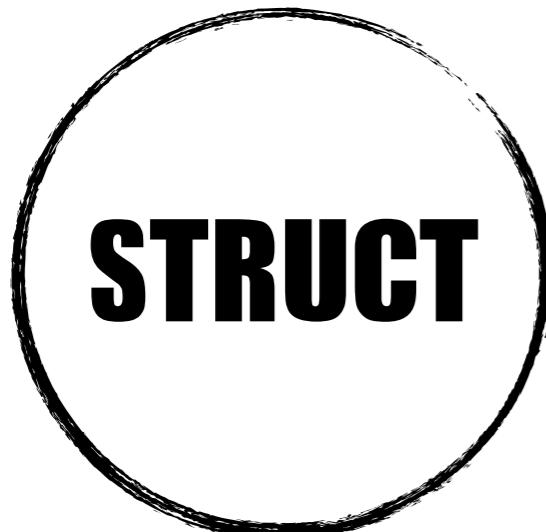
# class vs struct

長得很像 但是大家對於他們寄予的功能又不太一樣

source: <http://screamlab-ncku-2008.blogspot.com/2009/10/c-struct-class-keywords.html>



如果不指定存取權限預設是Private  
建構與解構元是必要的  
Object oriented



如果不指定存取權限預設是Public  
其實可以有建構元（只是你平常不會特別寫）  
Data aggregation

運算子多載

Operator overloading

# 運算子多載

- 讓自定義的類別也可以套用類似四則運算(+,-,\*,/ )的方法

- 提升程式可讀性

```
Complex a, b;
```

```
a.add(b);
```

```
a = a + b; //Which one is better?
```

- 因應不同的運算子會需要不一樣的定義

這些東西**不可以**被覆載

- . 成員選取
- .\* 成員指標選取
- :: 範圍解析
- ??: 條件式
- # 前置處理器轉成字串
- ## 前置處理器串連

換言之，剩下的就可以

- 二元的四則運算(+ - \* /)
- 遞增遞減(++, --)
- 指派 =
- 左移右移運算子 << >>
- 陣列註標 []
- 函式呼叫()
- 成員選取 ->
- 新增/刪除 new delete
- ...

# 一元運算子

用正課作業的Timer類別示範

operator : ++ --

前置運算子(運算式需要左值)  
Timer &timer::operator++()

```
{  
    sec++;  
    return *this;  
}
```

//如何使用

```
Timer Foo(0); //0秒  
++Foo;  
Foo.print(); //1秒
```

後置運算子

輸入的int參數只是用來分辨前置後置

Timer Timer::operator++(int)

```
{  
    Timer temp = *this;  
    sec++;  
    return tmp;  
}
```

//如何使用

```
Timer Foo(0); //0秒  
Foo++;  
Foo.print(); //1秒
```

# 二元運算子

用正課作業的Timer類別示範

operator : + -

兩個Timer 加在一起

```
Timer Timer::operator+(const Timer &secondTime)
{
    int t_sec, t_min
    t_sec = sec + secondTime.sec;
    t_min = min + secondTime.min;
    t_min += t_sec/60;
    t_sec %= 60;
    return Timer(t_min, t_sec);
}
```

# 二元運算子

用正課作業的Timer類別示範

operator : + -

兩個Timer 加在一起

Timer Timer::operator+(const Timer &secondTime)

```
{  
    int t_sec, t_min  
    t_sec = sec + secondTime.sec;  
    t_min = min + secondTime.min;  
    t_min += t_sec/60;  
    t_sec %= 60;  
    return Timer(t_min, t_sec);  
}
```

secondTime是在運算元的右邊



# 二元運算子

用正課作業的Timer類別示範

operator : + -

兩個Timer 加在一起

```
Timer Timer::operator+(const Timer &secondTime)
{
    int t_sec, t_min
    t_sec = sec + secondTime.sec;
    t_min = min + secondTime.min;
    t_min += t_sec/60;
    t_sec %= 60;
    return Timer(t_min, t_sec);
}
```

secondTime是在運算元的右邊

```
//使用方式
Timer T1, T2, T3;
T1 = 100;
T2 = 100;
T3 = T1 + T2;
```

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : + -

- Timer foo;  
foo = 0 ;  
**foo = foo + 100;**
- 在設計運算元的時候必須考慮左值(Lvalue)跟右值(Rvalue)。  
看上面的例子，如果你看到foo + 100，會不會覺得100 + foo  
也是可行的？
- 該怎麼定義？

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : + -

- 解法：friend

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : + -

```
Timer foo(10);  
foo = foo + 100;
```

- 解法：friend

```
friend Timer operator+(Timer T1, Timer T2)  
{  
    int min, sec;  
    min = T1.min + T2.min;  
    sec = T1.sec + T2.sec;  
    min += sec/60;  
    sec %= 60;  
    Return Timer(min, sec);  
}
```

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : + -

```
Timer foo(10);  
foo = foo + 100;
```

## ● 解法：friend

咦，我不是用Foo + 100嗎  
怎麼參數是兩個Timer？

```
friend Timer operator+(Timer T1, Timer T2)  
{  
    int min, sec;  
    min = T1.min + T2.min;  
    sec = T1.sec + T2.sec;  
    min += sec/60;  
    sec %= 60;  
    Return Timer(min, sec);  
}
```

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : + -

```
Timer foo(10);  
foo = foo + 100;
```

## ● 解法：friend

咦，我不是用Foo + 100嗎  
怎麼參數是兩個Timer？

```
friend Timer operator+(Timer T1, Timer T2)  
{  
    int min, sec;  
    min = T1.min + T2.min;  
    sec = T1.sec + T2.sec;  
    min += sec/60;  
    sec %= 60;  
    Return Timer(min, sec);  
}
```

為了達成這個目的，其實你還要定義一個東西  
**Timer::Timer(int);**  
這個可以讓100透過這個建構元去產生一個Timer型態的物件，就可以滿足我定義的operator+了

```
1 #include<iostream>
2 using namespace std;
3
4 class Timer{
5     private:
6         int min,sec;
7     public:
8         Timer(int s): min(s/60), sec(s%60){
9             cout << s;
10            cout << " In CTOR\n";
11        }
12        Timer(int m,int s): min(m), sec(s)
13        {
14            cout << "In two CTOR\n";
15        }
16        ~Timer()
17        {
18            cout << "In DTOR\n";
19        }
20
21        friend Timer operator+(Timer T1, Timer T2)
22        {
23            return Timer(T1.min + T2.min, T1.sec + T2.sec);
24        }
25    };
26
27
28    int main()
29    {
30        Timer T1(50);
31        Timer T2 = T1 + 100;
32        return 0;
33    }
34
```

# Try this

```

1 #include<iostream>
2 using namespace std;
3
4 class Timer{
5     private:
6         int min,sec;
7     public:
8         Timer(int s): min(s/60), sec(s%60){
9             cout << s;
10            cout << " In CTOR\n";
11        }
12        Timer(int m,int s): min(m), sec(s)
13        {
14            cout << "In two CTOR\n";
15        }
16        ~Timer()
17        {
18            cout << "In DTOR\n";
19        }
20
21        friend Timer operator+(Timer T1, Timer T2)
22        {
23            return Timer(T1.min + T2.min, T1.sec + T2.sec);
24        }
25    };
26
27
28    int main()
29    {
30        Timer T1(50);
31        Timer T2 = T1 + 100;
32        return 0;
33    }
34

```

# Try this

## OUTPUT...

**50 In CTOR**

**100 In CTOR**

**In two CTOR**

**In DTOR**

**In DTOR**

**In DTOR**

**In DTOR**

輸出對應的Object呼叫是這樣的：

**Line 30 - T1生成**

**Line 31 - 100這個整數透過Timer(int)**

**Line 23 - 回傳Timer物件，用兩個參數的建構元呼叫**

**解構整數100 產生的Timer**

**解構operator+複製的T1(call by value)**

**解構main的T2**

**解構main的T1**

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator :=

- 我想寫出類似 Timer Foo2 = Foo 可以嗎？

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator :=

- 我想寫出類似 Timer Foo2 = Foo 可以嗎？

```
Timer &operator=(const Timer &rhs)
{
    if(this == &rhs)
        return *this; //預防自己等於自己
    this->min = rhs.min;
    this->sec = rhs.sec;
    return *this;
}
```

寫operator=的時候，要特別注意動態配置的變數是否有正確的釋放與配置。  
通常會跟著dtor, copy ctor一起使用。

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator ==

- 比較兩個物件是否相等

```
bool &operator==(const Timer &rhs)
{
    if(sec == rhs.sec && min == rhs.min)
        return true;
    return false;
}
```

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : << >>

- 如何自由地把成員變數在串流之間傳遞？

# 二元運算子 (cont.)

用正課作業的Timer類別示範

operator : << >>

- 如何自由地把成員變數在串流之間傳遞？
- 答案還是那個：Friend (共享給ostream/istream型別的物件)

```
friend ostream& operator<<(ostream &os, const Timer& T)
{
    os << T.min << ":" << T.sec << endl;
}
```

//注意串流物件只有一個，所以要加上&

繼承 Inheritance  
&  
多型 Polymorphism

# 繼承的價值

- 物件導向的重要原則：  
Open-Closed Principle:  
**擴充要很容易、但寫好的模組不能輕易被修改**
- 用現實例子說明：  
**剪頭髮用的剃頭刀，換刀頭不需要把整個給拆了**

# 繼承的例子

Base Class [基礎類別]

```
class Student
{
public:
    Student(string, int);
    ~Student();
    int getAge() const;
    string getName() const;
private:
    string m_name;
    int m_age;
}
```

# 繼承的例子

Base Class [基礎類別]

```
class Student
{
public:
    Student(string, int);
    ~Student();
    int getAge() const;
    string getName() const;
private:
    string m_name;
    int m_age;
}
```

這個const 表示該成員函式不能更動class裡面的任何成員

# 繼承的例子

## Base Class [基礎類別]

```
class Student
{
public:
    Student(string, int);
    ~Student();
    int getAge() const;
    string getName() const;
private:
    string m_name;
    int m_age;
}
```

這個const 表示該成員函式不能更動class裡面的任何成員

## Derived Class [衍生類別]

```
class Graduate: public Student
{
public:
    Graduate(string, int ,int);
    int getStipend();  
新的成員函式
private:
    int m_stipend;  
新的資料成員
}
```

# 繼承的例子

## Base Class [基礎類別]

```
class Student
{
public:
    Student(string, int);
    ~Student();
    int getAge() const;
    string getName() const;
private:
    string m_name;
    int m_age;
}
```

這個const 表示該成員函式不能更動class裡面的任何成員

## Derived Class [衍生類別]

```
class Graduate: public Student
{
public:
    Graduate(string, int ,int);
    int getStipend();  
新的成員函式
private:
    int m_stipend;
}  
新的資料成員
```

除此之外, Graduate還會保有Student的成員  
但是在Student 中 private的 依然是private

```
int Graduate::getStipend() const
{
    if(m_age>30) //無法存取
        return 0;
    else return m_stipend;
}
```

# 繼承的例子

## Base Class [基礎類別]

```
class Student
{
public:
    Student(string, int);
    ~Student();
    int getAge() const;
    string getName() const;
private:
    string m_name;
    int m_age;
}
```

這個const 表示該成員函式不能更動class裡面的任何成員

## Derived Class [衍生類別]

```
class Graduate: public Student
{
public:
    Graduate(string, int ,int);
    int getStipend();  
新的成員函式
private:
    int m_stipend;
}  
新的資料成員
```

除此之外, Graduate還會保有Student的成員  
但是在Student 中 private的 依然是private

```
int Graduate::getStipend() const
{
    if(getAge() >30) //合法的存取
        return 0;
    else return m_stipend;
}
```

# Protected

- 讓父類別的protected成員可以被子類別存取。

```
class Student
{
public:
    Student(string, int);
    ~Student();
    int getAge() const;
    string getName() const;
protected:
    string m_name;
    int m_age;
}
```

```
int Graduate::getStipend() const
{
    if(m_age>30) //OK!
        return 0;
    else return m_stipend;
}
```

# 繼承與建構元/解構元

- 建構元(ctor)/解構元(dtor)不會被繼承，但是在子類別object生成跟銷毀時還是會呼叫到父類別的**default ctor/dtor**。

# 繼承與建構元/解構元

- 建構元(ctor)/解構元(dtor)不會被繼承，但是在子類別object生成跟銷毀時還是會呼叫到父類別的**default ctor/dtor**。

```
int main()
{
    Graduate stu( "Mike" , 24, 600);
    cout << stu.getName() << ", " << stu.getAge() << " years old.\n" ;
}
```

# 繼承與建構元/解構元

- 建構元(ctor)/解構元(dtor)不會被繼承，但是在子類別object生成跟銷毀時還是會呼叫到父類別的**default ctor/dtor**。

```
int main()
{
    Graduate stu( "Mike" , 24, 600);
    cout << stu.getName() << ", " << stu.getAge() << " years old.\n" ;
}
```

- 上面的輸出會像是右邊那樣

```
In Student CTOR
In Graduate CTOR
Mike, 24 years old.
In Graduate DTOR
In Student DTOR
```

# 繼承與建構元/解構元

- 建構元(ctor)/解構元(dtor)不會被繼承，但是在子類別object生成跟銷毀時還是會呼叫到父類別的**default ctor/dtor**。

```
int main()
{
    Graduate stu( "Mike" , 24, 600);
    cout << stu.getName() << ", " << stu.getAge() << " years old.\n" ;
}
```

- 上面的輸出會像是右邊那樣
- 這是compiler主動呼叫處理的

```
In Student CTOR
In Graduate CTOR
Mike, 24 years old.
In Graduate DTOR
In Student DTOR
```

# 總整理

訪問的物件	public	protected	private
本類別的成員	0	0	0
子類別的成員	0	0	X
非類別成員	0	X	X

- 會被繼承的東西
  - 父類別中public/protected的所有成員
- 不會被繼承的東西
  - 建構元/解構元
  - operator=()
  - friends
- 善用::來存取不同類別間可能同名的變數或function

# 多重繼承

- 如果子類別需要繼承至少一個以上的父類別

```
class Mul : public Base1 , private Base2
```

# 虛擬函式 & 多型 Polymorphism

- 有些時候，父類別不需要定義的這麼的詳細
- 透過virtual關鍵字，讓父類別僅保留函式宣告，函式細節則由子類別負責定義。
- 通常搭配著類別的指標使用

```
class Polygon
{
protected:
    int width, height;
public:
    void setValue(int a, int b){
        width=a, height=b;
    }
    virtual int area(){
        return 0;
    }
}

class Rectangle: public Polygon
{
public:
    int area(){
        return width*height;
    }
}

class Triangle: public Polygon
{
public:
    int area(){
        return width*height/2;
    }
}
```

# 多型與類別指標的使用

```

class Polygon
{
protected:
    int width, height;
public:
    void setValue(int a, int b){
        width=a, height=b;
    }
    virtual int area(){
        return 0;
    }
}

class Rectangle: public Polygon
{
public:
    int area(){
        return width*height;
    }
}

class Triangle: public Polygon
{
public:
    int area(){
        return width*height/2;
    }
}

```

# 多型與類別指標的使用

```

int main()
{
    Polygon p;
    Triangle t;
    Rectangle r;
    Polygon *P1 = &p;
    Polygon *P2 = &t;
    Polygon *P3 = &r;
    P1->setValue(4,5);
    P2->setValue(4,5);
    P3->setValue(4,5);
    cout << P1->area() << endl;
    cout << P2->area() << endl;
    cout << P3->area() << endl;
    return 0;
}

```

**OUTPUT**

0
20
10

雖然指標都是Polygon型態的，但因為Rectangle, Triangle都是Polygon的子類別，所以操作是合法的  
在物件導向的世界裡，可以透過父類別的指標去靈活的運用及呼叫

# 純虛擬函式(Pure virtual function)

# 純虛擬函式(Pure virtual function)

```
class Animal
{
public:
    virtual void bark() =0;
}
```

純虛擬函式：完全不定義  
此時Animal又叫做「抽象父類別」

# 純虛擬函式(Pure virtual function)

```
class Animal
{
public:
    virtual void bark() =0;
}
```

純虛擬函式：完全不定義  
此時Animal又叫做「抽象父類別」

```
class Cat : public Animal
{
public:
    void bark(){
        cout << "Meow~\n" ;
    }
}
```

```
class Dog : public Animal
{
public:
    void bark(){
        cout << "Woof~\n" ;
    }
}
```

# 純虛擬函式(Pure virtual function)

```
class Animal
{
public:
    virtual void bark() =0;
}
```

純虛擬函式：完全不定義  
此時Animal又叫做「抽象父類別」

```
class Cat : public Animal
{
public:
    void bark(){
        cout << "Meow~\n" ;
    }
}
```

```
class Dog : public Animal
{
public:
    void bark(){
        cout << "Woof~\n" ;
    }
}
```

子類別再負責定義bark()的內容，如果沒定義會Compile Error

# 純虛擬函式(Pure virtual function)

```
class Animal
{
public:
    virtual void bark() =0;
}
```

純虛擬函式：完全不定義  
此時Animal又叫做「抽象父類別」

```
class Cat : public Animal
{
public:
    void bark(){
        cout << "Meow~\n" ;
    }
}
```

```
class Dog : public Animal
{
public:
    void bark(){
        cout << "Woof~\n" ;
    }
}
```

子類別再負責定義bark()的內容，如果沒定義會Compile Error

```
int main()
{
    Animal A; // Error，抽象父類別無法生成物件
    Cat c;
    Dog d;
    c.bark(); // 貓叫：Meow~
    d.bark(); // 狗叫：Woof~
}
```

```
//指標的應用
int main()
{
    Animal *c, *d;
    c = new Cat(); //繼承特性讓Animal指標轉型
    d = new Dog();
    c->bark(); // 貓叫：Meow~
    d->bark(); // 狗叫：Woof~
}
```

樣板 Template

# Template

- 雖然我們有overload可以做到同名函式不同型態的參數，但是這樣你必須定義很多個overload function出來。

# Template

- 雖然我們有overload可以做到同名函式不同型態的參數，但是這樣你必須定義很多個overload function出來。
- 而且少一個都不可以哦！

```
void Fun(int a, int b);
void Fun(double a, double b);

//呼叫階段
Fun(123L, 123L); //完蛋,沒有定義long 的引數
```

# Template

- 雖然我們有overload可以做到同名函式不同型態的參數，但是這樣你必須定義很多個overload function出來。
- 而且少一個都不可以哦！

```
void Fun(int a, int b);
void Fun(double a, double b);

//呼叫階段
Fun(123L, 123L); //完蛋,沒有定義long 的引數
```

- Template就是為此而生的

# 函數樣板 Function Template

- **Template原型** : template <參數型態 參數名稱>

# 函數樣板 Function Template

- Template原型：template <參數型態 參數名稱>

有兩種可以選：**class**或是**typename**

# 函數樣板 Function Template

- Template原型：template <參數型態 參數名稱>  
有兩種可以選：**class**或是**typename**
- 只要用到Template就要在原來的函式前面宣告

# 函數樣板 Function Template

- Template原型： template <參數型態 參數名稱>
- 只要用到Template就要在原來的函式前面宣告

```
template<class T> ↑
T addFunc(T a, T b)
{
    return a+b;
}

template<class T>
T subFunc(T a, T b)
{
    return a-b;
}
```

這兩行可以也可以縮成一行

```
template<class T> T addFunc(T a,T b){}
```

```
//呼叫
int main()
{
    int a=1,b=2;
    addFunc(a, b);
    double c=3.2, d=4;
    subFunc(c, d);
}
```

# 類別樣板 class Template

- 跟Function Template很像，但是可以處理class裡面的成員

```
Class Array
{
    private:
        int arrSize;
        int* arr;
    public:
        Array(int size)
            :arrSize(size){
                arr = new int[arrSize];
            }
        ~Array(){
            delete[] arr;
        }
} //正常的class
```

```
template <class T>
Class Array
{
    private:
        int arrSize;
        T* arr;
    public:
        Array(int size): arrSize(size){
            arr = new T[arrSize];
        }
        ~Array(){
            delete[] arr;
        }
} //有template的class
```

```
int main()
{
    Array<int> a(20);
}
```

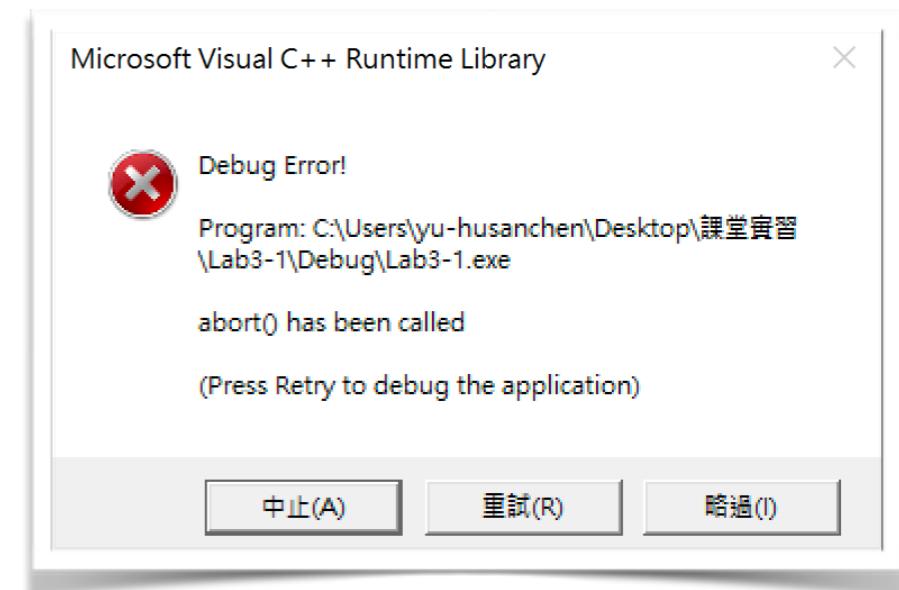
# 例外處理 Exception

# Why Exception?

- 程式「執行時」總會有出錯的時候  
不管是使用者操作不當、或是Coding的  
時候少了一個分號(笑)

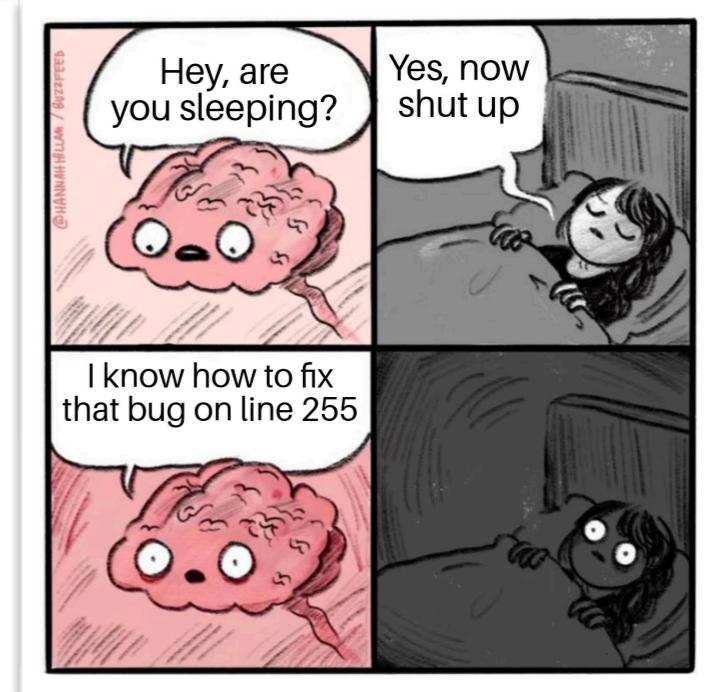
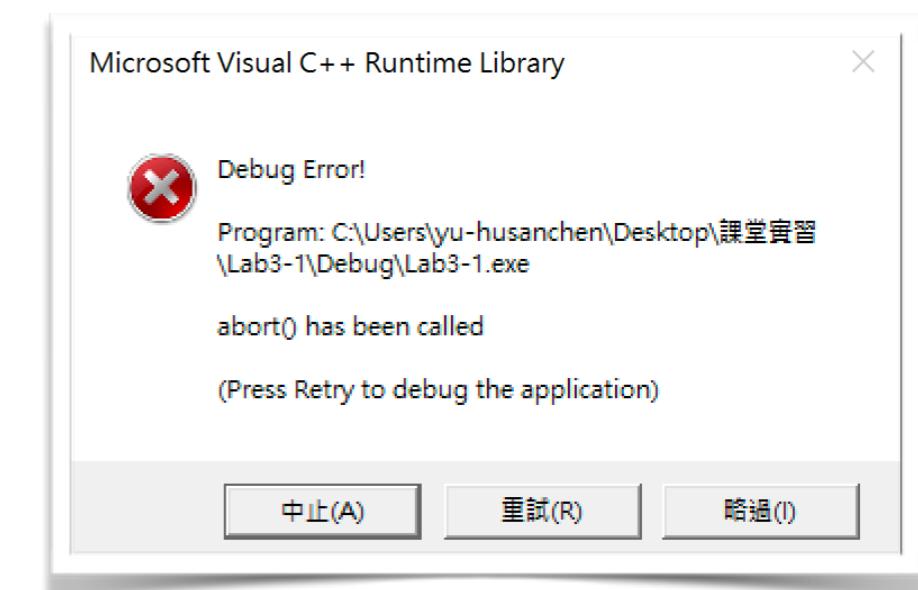
# Why Exception?

- 程式「執行時」總會有出錯的時候  
不管是使用者操作不當、或是Coding的時候少了一個分號(笑)
- 試想一種情況：程式一出錯所有剛剛做的進度全部得重來，煩不煩人？



# Why Exception?

- 程式「執行時」總會有出錯的時候  
不管是使用者操作不當、或是Coding的時候少了一個分號(笑)
- 試想一種情況：程式一出錯所有剛剛做的進度全部得重來，煩不煩人？
- 好的例外處理可以
  - 讓程式有容錯空間：即使出錯程式可以迴避有問題的段落、順利完成
  - 甩鍋的時候比較方便(看到Exception跳出來就表示有人寫的code炸鍋了)



# 呈現方式

```
try{
    int a=10,b;
    cin >> b;
    if(b==0)
        throw -1;
    cout << a/b << endl; ← 正常該執行的
}
catch(int err)
{
    if(err == -1)
        cout << "除法不可以除以0 \n" ;
}
catch(std::exception e) ←
{
    cout << e.what();
}
```

可能會出事的段落  
自己主動throw exception  
只要甩exception出來  
下面的cout就被跳過了

catch到exception之後該如何處理  
這裡的例子是印出訊息

Exception物件則是系統自己丟出來的

# 再一個例子

因為Java的例外機制很詳實所以用Java解釋

# 再一個例子

因為Java的例外機制很詳實所以用Java解釋

```
try {
    FileWriter ff = new FileWriter("./test.json");
    JSONObject json = new JSONObject();
    try {
        JSONArray jsonMember = new JSONArray();
    }
    json.put("CRAW", jsonMember);
    String writeString = json.toString();
    ff.write(writeString);
    ff.close();
} catch (JSONException jse) {
    jse.printStackTrace();
}
} catch (IOException ioe) {
    ioe.printStackTrace();
}
}
```

# 再一個例子

因為Java的例外機制很詳實所以用Java解釋

```
try {  
    FileWriter ff = new FileWriter("./test.json");  
    JSONObject json = new JSONObject();  
    try {  
        JSONArray jsonMember = new JSONArray();  
    }  
    json.put("CRAW", jsonMember);  
    String writeString = json.toString();  
    ff.write(writeString);  
    ff.close();  
} catch (JSONException jse) {  
    jse.printStackTrace();  
}  
} catch (IOException ioe) {  
    ioe.printStackTrace();  
}  
}
```

JAVA針對大部分的例外情況幾乎都有對應的稱呼  
所以可以很輕鬆的處理  
老實說我覺得Java在exception這塊比C++好看多了  
(當然有可能是我還太菜XD)  
左邊的箭頭指出這行程式碼可能會甩出什麼Exception

扯遠了....回來回來

**剩下沒講到的呢？**

**自己去翻投影片啊！！**