【第二講】

C++ 初探

講師: 李根逸 (Ken-Yi Lee), E-mail: feis.tw@gmail.com

★ ★ ★ ★

★ ★ ★ ★

課程大綱

- **C 與 C++**
- **資料型態、變數宣告與初始化**
  - 布林 (**bool**) 型態
  - 參考 (**type** *) 型態
  - 類別 (**class**) 型態
- **運算式、字面常數與暫時變數**
- **函式呼叫**
- **函式多載**
- **物件的建構、解構與隱性轉型**
- **運算子多載**
- **命名空間 (namespace)**
- **C++ 標準模版庫 (STL)**
  - 標準輸入輸出串流

C 與 C++

- **程式語言是可用來控制機器的人造語言**
  - C 與 C++ 都是高階程式語言
  - 高階語言需要經過編譯才能成為機器能執行的機器碼
- **常見 C 語言標準:**
  - ANSI C (C89) 與 ISO C (C90)
- **常見 C++ 語言標準:**
  - ISO/IEC 14882:1998 (C++98)
  - ISO/IEC 14882:2003 (C++03) [與 C++98 相似]
- **這門課程將以 C++98 為主要標準**
  - ISO/IEC 14882:2011 (C++11) 是將來的主流標準

C++ 的設計特色

- **C++ 語言的主要特色：**
  - 高度相容於 C，讓程式容易從 C 移植到 C++
  - 與 C 同樣具有相等高效與平台可攜的特性
  - 同時可使用多種設計方法:
    - 程序式設計 (procedural programming) [函式: 跟 C 一樣]
    - 物件導向程式設計 (object-oriented programming) [類別: class]
    - 泛型程式設計 (generic programming) [模版: template]
    - 例外處理 (exception handling) [例外]
- **了解『物件導向程式設計』與『泛型程式設計』等方法是學習或使用 C++ 和 C 之間最大的不同**
  - 這門課主要是透過了解『泛型程式設計』的優點來幫助我們使用各種現成資料結構與演算法開發高效的程式

資料型態、變數定義與初始化

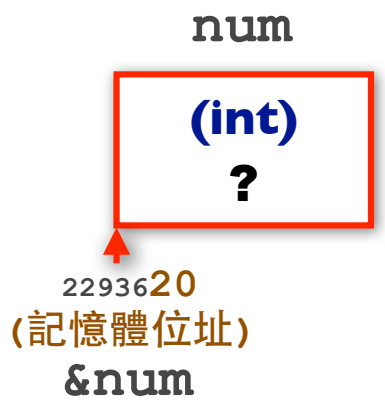
資料型態 (data type)

- 資料型態會決定我們怎麼儲存、表示和運算資料
- **C++** 裡的資料型態分為兩大類：
  - 基本資料型態
    - 整數 (**int, short int, long int, char, bool, ...**)
    - 浮點數 (**float, double, long double**)
    - 無 (**void**)
  - 複合資料型態
    - 陣列 (**type** [])
    - 函式 (**type** ())
    - 指標 (**type** *)
    - 參考 (**type &**) [C++ 特有]
    - 結構 (**struct**) [C 就有，但 C++ 對其做了擴充]
    - 類別 (**class**) [C++ 特有]
    - 列舉 (**enum**) [C 就有，但是我們為了簡化課程而不討論]
    - 聯合 (**union**) [C 就有，但是我們為了簡化課程而不討論]

變數定義

- 變數定義：
  - 資料型態 變數名稱;

```
int num;
```



- 變數名稱的限制：
  - 英文字母、數字或 _ 構成，數字不可開頭，且同一個英文字母的大小寫是視為相異的
  - 變數名稱不可以是『保留字』
- 變數定義時要指定該變數的資料型態，同時執行時會配置該變數所佔據的記憶體。

變數初始化

- 如果我們在變數定義時同時指定其值的話，我們稱為『初始化』
  - 未初始化的變數其值『不一定』為 0 (且通常不是)
  - 存取未給定過值的變數是『未定義行為』
- 變數初始化與賦值運算：

```
int num = 6;            // 初始化
num = 8;                // 賦值運算
```

我們需要區別這兩個 = 的差異

可視範圍與生命週期

- **變數定義的位置會決定該變數的『可視範圍』，同時也決定該變數空間的『生命週期』**
- **可視範圍 (scope)**：
  - 全域變數：定義在函式定義之外
  - 區域變數：定義在區塊以內
- **生命週期 (life time)**：
  - 全域變數：main 執行前配置，main 結束後釋放
  - 區域變數：執行到定義位置時配置，執行到區塊結尾（左大括號) 時釋放

```
#include <stdio.h>
#include <stdlib.h>

int i = 5;

void p(){
  /* int i = -1; */
  i = i + 1;
  printf("%d\n", i);
  return;
}

int main(){
  printf("%d\n", i);
  int i = 6;
  i = i + 1;
  p();
  printf("%d\n", i);
  system("pause");
  return 0;
}
```

這程式的執行結果是？

這程式的執行結果是？

```
#include <stdio.h>
#include <stdlib.h>

int main(){
  int i = 3;
  printf("%d\n", i);

  if (i == 3) {
    i = i + 1;
    int i = 6;
    printf("%d\n", i);
    i = i + 1;
  }

  if (i == 3) {
    printf("%d\n", i);
  }
  system("pause");
  return 0;
}
```

特殊可視範圍與生命週期

- 函式參數的可視範圍與生命週期在函式定義內
  - 只有該函式定義內可以使用函式參數
  - 一般函式參數在呼叫函式時配置，在離開函式時釋放
    - 後面我們會介紹 C++ 新增的參考型態
- **if 、 while 和 for 等流程控制語法的 () 裡面定義的變數可視範圍在該流程控制語法內，而生命週期在該流程控制語法開始時配置，結束時釋放**

```
int sum = 0;
for (int i = 1; i <= 10; ++i) {
  sum += i;
}
printf("%d\n", sum);
```

C++ 特有的資料型態

布林 (bool) 型態

- 真與假的概念在 C 語言中是使用『非 0』 與『 0 』兩種整數值來表示，在 C++ 語言中則使用了布林 (bool) 型態的值來表示。
  - bool 型態的值有 true (真) 與 false (假) 兩種
  - 因為要與 C 相容，所以 C++ 中『非 0 』值可以轉型後為 true ，而『 0 』可以轉型後為 false。相對地，true 可以轉型後為 『 1 』 而 false 可以轉型後為『 0 』
  - 實質上 bool 也是個整數型態
- 在 C++ 中使用 bool 型態最重要的意義是在於讓我們更容易了解程式碼 (可讀性)

[範例] bool.cpp

布林型態的運算

```
5 == 5              // true
5 != 5              // false
5 > 4               // true
5 < 4               // false
5 > 4 > 3           // ?
5 == 5 == 5         // ?

!true               // false
true && true        // true
true || false       // true

not true            // false
true and true       // true
true or false       // false

true > false        // true
```

參考 (type &) 型態

- 一般而言，指標的使用不易維護其正確性，所以要盡量避免在程式碼中使用裸指標 (type *)。但是指標在 C 語言中提供很多重要的功能，因此在 C++ 中提供一些替代方案來避免裸指標的使用，其中之一就是『參考』(type &) 型態

這裡的 & 不是取址運算子!

- 宣告並初始化參考 (參考必須要初始化)：

```
被參考變數的資料型態 &參考名稱 = 被參考變數的名稱;
```

  - 參考名稱 "等同於" 被參考變數的名稱 (可視為別名)
  - 參考名稱是參考哪個變數，在宣告後就不能改變
  - 不可以宣告參考型態的陣列
  - 不可以宣告參考的參考

[範例] reference.cpp

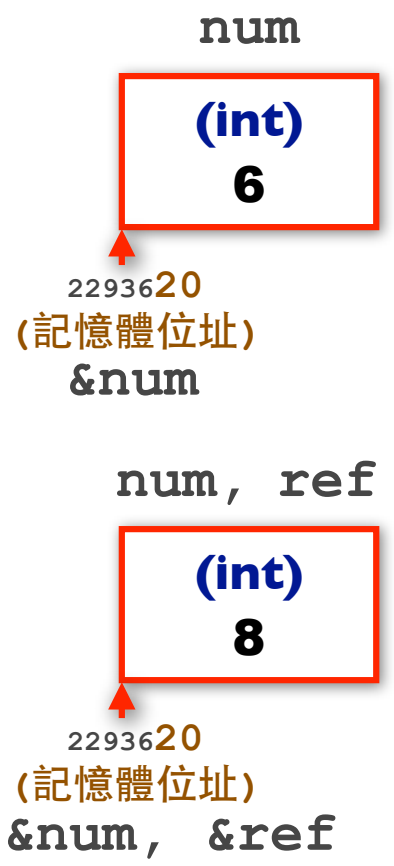
參考型態的運算

- 參考型態本身只是個別名的存在：

```
int num = 6;
 // num == ?
 // &num == ?

int &ref = num;
 // ref == ?
 // &ref == ?

ref = 8;
 // num == ?
```



- 那本尊跟別名有什麼不同？
  - 別名不會額外配置記憶體空間，也無法決定生命週期

類別 (class) 型態

- 類別是物件導向的基石，也是 C++ 語言早期最為人熟知的新特性。類別可以將資料和函式做包裝與封藏，使得我們可以更乾淨、方便與直觀的使用。
- 類別定義：

```
class 類別名稱 {
 存取限制:
  資料型態   資料成員;
  回傳值型態  函式成員(參數型態 參數名稱,...);
  ...
};
```

可以有零到多個資料或函式成員

類別定義內的成員在程式碼中宣告的順序可以任意改變

  - 每個資料跟函式成員都可以設定不同的存取限制：
    - **public** (公開) 表示任何函式都可以存取或呼叫
    - **private** (私有) 表示只有該類別的函式成員可以存取或呼叫
    - **protected** (保護) 表示除了該類別外其衍生類別的函式成員也可以存取或呼叫

類別的設計

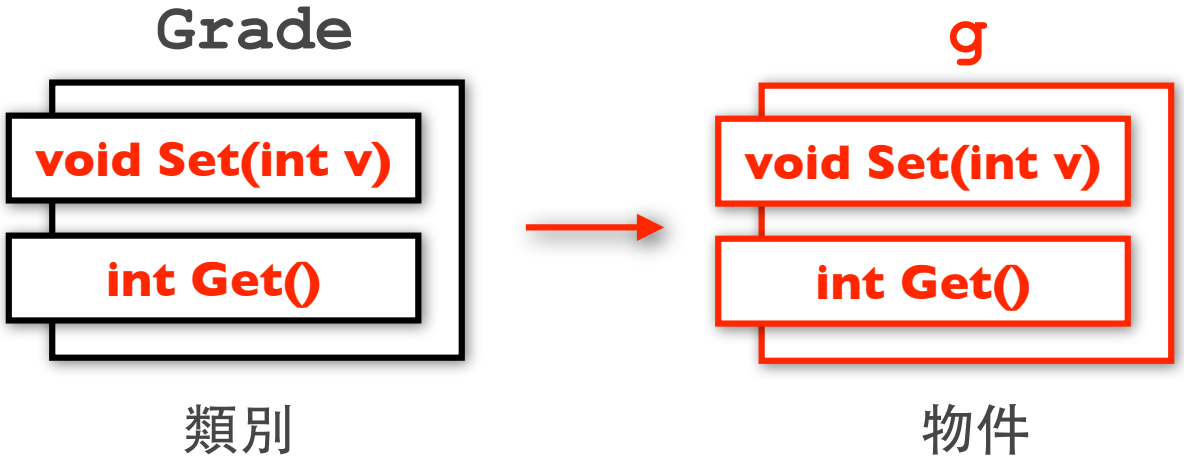
- 定義一個自訂的 **Grade** 類別：
  - 先想好我們要怎麼使用他？(user code)

```
Grade g;
g.Set(80);
printf("%d\n", g.Get());
```

這裡的 **g** 我們稱作為一個 **Grade** 物件

提供 **Set** 操作將成績設定為 **80** 分

提供 **Get** 操作可以得到成績



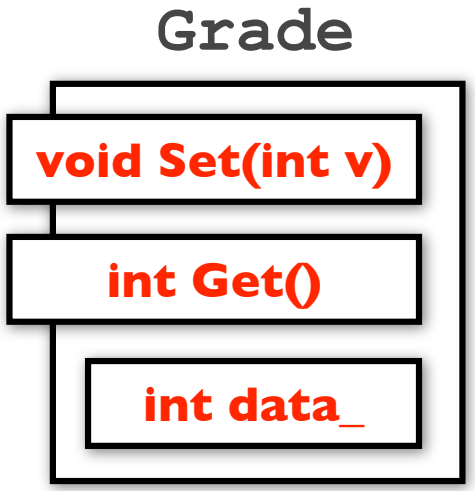
【範例】Grade

```
class Grade {
 public:
  void Set(int v);
  int Get();

 private:
  int data_;
};
```

將我們要提供的操作設定為 **public (公開)** 函式

將實作這些公開函式需要的其他成員設為 **private (私有)**



【思考】這樣的設計有什麼好處？

【寫作風格】存取限制依照 **public**、**protected**、**private** 順序排列

【寫作風格】函式名稱以大寫字母開頭

【寫作風格】私有資料成員名稱以 _ 結尾

【寫作風格】存取限制前留一格空白

[範例] class.cpp

【練習】Grade

- **試修改 Grade 類別，將成績的上限跟下限分別設為 100 與 0，也就是當成績超過 100 分時自動設為 100，而成績低於 0 分時自動設為 0 分**
  - 隨作業會附一個預期結果 ref 的執行檔，最少目標是要讓執行結果跟預期的一樣，但必須要合理並只修改 TODO 的部份。

[練習] ex2A.cpp

運算式、字面常數與暫時變數

運算式

- **C++ 程式碼主體由各種宣告、定義與運算式構成**
- **運算式讓我們對資料（運算元）執行運算子後得到運算的結果。因為同時只會有一個運算子執行，所以運算子之間的優先順序會影響運算的結果**
- **運算子的運算結果是存放在一個『暫時變數』**
  - 暫時變數一樣具有資料型態以及『可能』佔有記憶體空間
  - 『可能』佔有記憶體空間意味著我們不應該對『暫時變數』取址，同時也不應該去修改其值

```
int a = 3, b = 5;
a + b = 8;                  // 這裡的 a 跟 b 是運算元
                            // + 和 = 是運算子
                            // + 與 = 誰先執行？
```

字面常數

- 字面常數指的是我們在程式碼中直接寫出內建型態的值，字面常數可想成也是個『暫時變數』：

字面	資料型態	名稱
`3`	整數 (Integer)	`int, long int`
`3u`	無號整數 (Unsigned Integer)	`unsigned int`
`3l`	長整數 (Long Integer)	`long int`
`3.`	雙精度浮點數 (Double Precision Floating Point)	`double`
`3.f`	單精度浮點數 (Single Precision Floating point)	`float`
`'3'`	字元 (Character)	`char`
`true`	布林 (Boolean)	`bool`

對暫時變數取址與參考

- **暫時變數只是我們為了方便教學取的名稱，C++ 標準內稱其為右值 (r-value)。相對的，一般變數或動態配置的記憶體空間稱為左值 (l-value)**
  - 左或右可以從是不是可以放在賦值運算子左右邊判斷
  - 左值可以作為右值，反之則不行
- **因為暫時變數的生命週期在運算式算完後就結束，所以對其取址意義不大，同時參考暫時變數也意義不大，所以這兩種操作都是不合法的。**
  - 但是暫時變數可以作為固定值參考 (reference-to-const)，這設計帶給我們在函式呼叫傳遞引數和回傳值時使用上的好處
    - C++11 內更明確地分別了右值參考與左值參考

[範例] constref.cpp

函式呼叫

函式呼叫

```
int Addone(int v) {
  v = v + 1;
  return v;
}
```

```
int main() {
  int a = 4;
  int b = Addone(v);
  return 0;
}
```

- 函式呼叫時參數與回傳值的運作方式：

城牆！

無知的人們

城牆！

```
  int a = 4;
----------------
  int v = a;
----------------
  v = v + 1;
----------------
  int ret = v;
----------------
  int b = ret;
```

參數初始化

函式本體

回傳值初始化

這裡的 **ret** 類似個暫時變數

注意的是只有在參數初始化時可以存取 **a**

也只有在函式呼叫的地方可以使用 **ret**

函式傳值

- 由函式呼叫的方式可以知道，一般情況下，呼叫函式時會額外配置函式參數和回傳值的空間。會造成兩個主要效應：
  - 在傳入函式後，函式內部參數是引數的複製品
    - 無法修改到原本的引數
    - 複製一份的成本可能很高
  - 在函式回傳值的時候，使用的是『暫時變數』
    - 該回傳值的生命週期離開運算式就結束了，如果我們之後還要使用的話需要另外再定義一個變數去儲存 (同時需要做複製)
    - 我們不應該對回傳值直接做修改

[範例] passbyvalue.cpp

函式傳址 [1]

- **C 裡面的指標在函式參數中扮演重要的角色是因為要做出一些函式傳值的機制中無法直接達到的目的**
  - 在傳入函式後，函式內部參數是引數的複製品
    - 我們可以利用取址運算子 (&) 將變數的位址作為引數傳入函式，同時在函式內部用間接運算子 (*) 去取得該變數的值或者修改該變數
  - 在函式回傳值的時候，使用的是『暫時變數』
    - 函式可以傳回某配置好的記憶體空間的位址，此時就可以對其回傳值使用間接運算子 (*) 後修改其指向的記憶體空間的值
- **C / C++ 的函式傳址同樣建立在函式傳值的基礎上**

[範例] passbyaddr.cpp

函式傳址 [2]

```
int *Addone(int *v) {
  *v = *v + 1;
  return v;
}
```

```
int main() {
  int a = 4;
  int b = *Addone(&a);
  *Addone(&a) = 0;
  return 0;
}
```

```
  int a = 4;
----------------
  int *v = &a;
----------------
  *v = *v + 1;
----------------
  int *ret = v;
----------------
  int b = *ret;
```

函式傳參考 [1]

- C++ 為了避免裸指標 (type *) 的使用提供了參考型態讓我們可以作為函式參數或回傳值型態傳遞：

```
int &Addone(int &v) {
  v = v + 1;
  return v;
}
```

```
int main() {
  int a = 4;
  int b = Addone(a);
  Addone(a) = 0;
  return 0;
}
```

```
  int a = 4;
---------------
  int &v = a;
---------------
  v = v + 1;
---------------
  int &ret = v;
---------------
  int b = ret;
```

- 參考在函式呼叫傳遞中，傳遞的是該變數別名而非複製品，提供類似指標的作用

[範例] passbyref.cpp

【練習】變數交換

- 試使用參考型態來修改下面程式讓 **a, b** 變數的值可以交換：

```
void Swap(int a, int b) {
  int t = a;
  a = b;
  b = t;
}

int main() {
  int a = 3, b = 5;
  Swap(a, b);
  printf("a = %d, b = %d\n", a, b);
  system("pause");
  return 0;
}
```

[練習] ex2B.cpp

函式傳參考 [2]

- 如前所述，我們可以透過使用參考型態來讓被呼叫的函式可以修改原本的引數值
- 此外用參考型態傳遞還有一個很重要的用途：
  - 在呼叫函式的時候不用『複製』一份引數，增加效率
    - 也就是我們可能因為不想複製一份而使用參考型態:

```
Grade Add1(Grade &lhs, Grade &rhs) {
  Grade ret;
  ret.Set(lhs.Get()+rhs.Get());
  return ret;
}
```

不是為了要修改 **rhs** 或 **lhs** 的本尊而使用參考型態

【思考】為什麼 **Add1** 的回傳值不用參考型態來增加效率？

【思考】那為什麼前幾頁的 **Addone** 為可以回傳參考型態？

函式傳參考 [3]

- 但是我們可以這麼使用嗎？

```
Grade x, y, z;

x.Set(3);
y.Set(4);
z.Set(5);

Grade sum = Add1(Add1(x, y), z);
```

  - 我們不能去用一般參考去參考暫時變數！
    - 固定值參考 (reference-to-const) 可以參考暫時變數：

```
Grade Add2(const Grade &lhs, const Grade &rhs) {
  Grade ret;
  ret.Set(lhs.Get()+rhs.Get());
  return ret;
}
```

回傳值為類別型態

- 當回傳值為類別型態時，我們可以對回傳值做修改：

```
Grade x, y;
x.Set(3);
y.Set(5);
Add2(x, y).Set(10);   // 對暫時變數修改值不合理但是因為
                      // Grade 是類別而沒有編譯錯誤
```

- 所以通常會將回傳值設定為固定值 (const)：

```
const Grade Add3(const Grade &lhs,
                 const Grade &rhs);

Add3(x, y).Set(10);   // [編譯錯誤]
```

  - 那下面這個會編譯錯誤嗎？該怎麼解決？

```
Add3(x, y).Get());
```

[範例] returnobject.cpp

類別的 const 成員函式

- 在宣告類別時，我們可以使用 const 修飾成員函式，表示呼叫該成員函式時不會修改到物件內容：

```
class 類別名稱 {
 存取限制:
  回傳值型態 函式成員(參數型態 參數名稱,...) const;
  ...
};
```

注意 const 放的位置

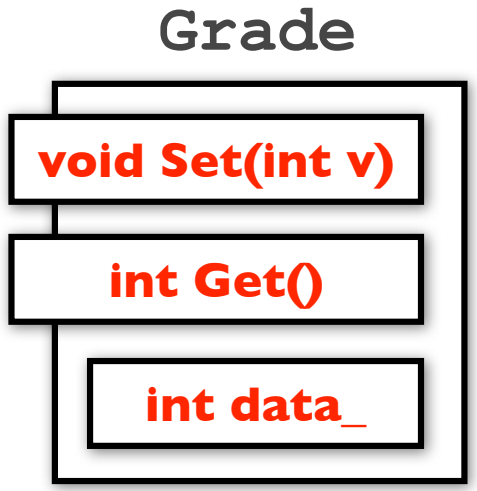
  - const 物件可以呼叫 const 成員函式，但不可以呼叫 non-const 成員函式
  - non-const 物件可以呼叫 const 或 non-const 成員函式
    - 當兩種成員函式都存在時呼叫 non-const 成員函式

【範例】Grade

```
class Grade {
 public:
  void Set(int v) {
    data_ = v;
  }
  int Get() const {
    return data_;
  }

 private:
  int data_;
};

const Grade Add(const Grade &lhs, const Grade &rhs) {
  Grade ret;
  ret.Set(lhs.Get() + rhs.Get());
  return ret;
}
```



【思考】Add 可以回傳固定值參考嗎？

函式多載

函式多載

- 在 **C** 語言裡面，不允許有兩個以上的函式具有相同的函式名稱。但是在 **C++** 語言裡面，函式在參數的型態或個數不同時，是可以同名的。換句話說，同樣名稱的函式可以有很多個，這個特色叫做函式多載 **(function overloading)**
  - 注意如果同名函式只有回傳值型態不同是不行的
- 同樣名稱的函式雖然可以因為多載而存在，但 **C++** 編譯器必須要能依照呼叫時給予的引數型態與個數決定要呼叫哪一個版本的函式，當編譯器無法找到適當的函式或者無法決定要呼叫哪一個的時候，會產生編譯錯誤

情境一:

```
int   Echo(int v)    { return v;}
float Echo(double v) { return v;}

int main() {
  printf("%d\n", Echo(3));      // 呼叫哪個 Echo ?
  printf("%f\n", Echo(3.5));    // 呼叫哪個 Echo ?
  return 0;
}
```

情境二:

```
float  Echo(float v)    { return v;}
double Echo(double v)   { return v;}

int main() {
  printf("%f\n", Echo(3));      // 呼叫哪個 Echo ?
  return 0;
}
```

// 如果我把其中一個拿掉會成功嗎?

【範例】自動判斷型態的輸出

- C 語言中要使用 printf 印出不同型態的資料時需要使用不同的格式符：

```
int a = 3;
printf("%d\n", a);

float b = 3.2;
printf("%f\n", b);

char c = 'A';
printf("%c\n", c);
```

  - 試著利用函式多載的設計讓我們不需要額外加格式符：

```
Print(a);
Print(b);
Print(c);
```

[範例] print.cpp

【練習】自動判斷型態的輸入

- C 語言中要使用 scanf 輸入不同型態的資料時需要使用不同的格式符，而且需要用指標：

```
int a;
scanf("%d", &a);

float b;
scanf("%f", &b);
```

  - 試著利用函式多載和參考的設計讓我們不需要額外加格式符與使用指標：

```
Scan(a);
Scan(b);
```

[練習] ex2C.cpp

物件的建構、解構與隱性轉型

物件的生與死

- **一份物件（變數）在生命週期中會經歷兩個必然的過程：生與死**
  - **出生**：當一份物件出生時，會配置儲存該物件所需要的記憶體空間，然後自動呼叫『建構式』
  - **死亡**：當一份物件死亡時，會自動呼叫『解構式』後再釋放儲存該物件的記憶體空間
- **我們可以透過對一個類別定義『建構式』或『解構式』來讓物件在「出生後」或「死亡前」自動執行一些指令**

類別的建構式與解構式

- **當你配置一個類別物件的時候，該類別定義裡對應的建構式（constructor）會被執行，用來作為物件的『初始化』或提供『隱性轉型』使用**
  - 建構式在類別定義中看起來是一個「名稱與類別相同」且「沒有回傳值資料型態」的函式
    - 例如: Grade 類別內的 Grade 函式
- **當你釋放一個物件的時候，該類別定義裡對應的解構式 (destructor) 會被執行，用來回收物件所佔用的資源**
  - 解構式在類別定義中看起來是一個是「名稱為類別名稱前加上 ~ 符號」且「沒有回傳值資料型態」的函式
    - 例如: Grade 類別內的 ~Grade 函式

[範例] allocation.cpp

預設建構式

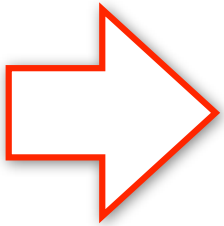
- 預設建構式指的是不帶有參數的建構式：

```
Grade a;                    // 使用 Grade() 建構式
                            // 我們稱為預設建構式
```

  - 當物件被配置時會自動呼叫建構式，所以當我們沒替該類別宣告任何建構式時，會有一個公開的空白預設建構式被自動定義：

```
class Grade {
};
```

沒有建構式的類別



```
class Grade {
 public:
  Grade() {}
};
```

自動加入空白預設建構式

使用物件的建構式

- 我們也可以直接在運算時使用建構式來產生暫時變數(物件)：

```
Grade();                    // 呼叫預設建構式，產生一個暫時物件
```

  - 要注意他跟一般物件定義的差別：

```
Grade a;                    // 呼叫預設建構式，產生 a 物件
```

  - 我們可以用暫時物件來初始化物件：

```
Grade b = Grade(); // 這裡跟只有寫 Grade b; 等義
```

- 宣告物件陣列的時候也會呼叫建構式：

```
Grade c[3];                 // 會呼叫三次 Grade() 建構式
```

[範例] constructor.cpp

帶有參數的建構式

- 建構式可以帶有參數 (反之解構式不行)
- 使用帶有參數的建構式來產生一般物件：

```
Grade d(10);        // 在名稱後加上 () 呼叫建構式
                    // Grade(int);
```

  - 但是不能用上面這個方法來呼叫沒有參數的建構式：

```
Grade e();          // 因為語法本身有兩種可能的意思
                    // 這行會被當做函式宣告!
```

  - 當然也可以呼叫帶有參數的建構式來產生暫時物件：

```
Grade(10);          // 呼叫 Grade(int) 產生一個暫時物件
```

  - 然而產生物件陣列時只能呼叫預設建構式而不能呼叫帶有參數的建構式：

```
Grade c[3];         // 會呼叫三次 Grade() 建構式
```

隱性轉型成類別物件

- 單一參數的建構式可以讓我們將該參數型態的值『隱性轉型』成該類別的物件：

```
Grade f = Grade(10);
Grade g = 10;          // 會呼叫 Grade(int); [隱性轉型]
```

  - 舉個例子：

```
const Grade Add(const Grade &lhs, const Grade &rhs) {
  return Grade( lhs.Get() + rhs.Get() );
}
// 在這個定一下，我們可以呼叫 Add(3, 5) 嗎 ?
```

  - 實際上因為回傳時也可隱性轉型，所以也可定義為：

```
const Grade Add(const Grade &lhs, const Grade &rhs) {
  return lhs.Get()+rhs.Get();
}
```

[範例] implicit.cpp

【複習】建構式的使用

- 用來宣告定義一般物件：

```
Grade a;                  // 呼叫預設建構式 Grade();
Grade b = Grade()         // 呼叫預設建構式 Grade();
Grade c[3];               // 呼叫三次預設建構式 Grade();
Grade d(10);              // 呼叫建構式 Grade(int);
Grade e();                // 函式宣告
Grade f = Grade(10);      // 呼叫建構式 Grade(int);
Grade g = 10;             // [隱性轉型] 呼叫 Grade(int);
```

- 用來產生暫時物件：

```
Grade();                  // 呼叫預設建構式 Grade();
Grade(10);                // 呼叫建構式 Grade(int);
```

【範例】Grade

```
class Grade {
 public:
  Grade()           { data_ = 0; }
  Grade(int v)      { data_ = v; }
  void Set(int v)  { data_ = v; }
  int Get() const { return data_; }

 private:
  int data_;
};

const Grade Add(const Grade &lhs, const Grade &rhs) {
  return lhs.Get() + rhs.Get();
}

// 我們可以呼叫 Add(Add(3, 5),10) 得到一個 const Grade 物件
```

運算子多載

運算子多載

- **C++ 支援運算子多載 (operator overloading)** 允許定義自訂型態的運算子行為。
  - 舉個例子：

```
Grade a = 10;
Grade b = 20;
Grade c = Add(a, b); // 可以寫成 a + b 嗎 ?
```

  - 新增一個函式去多載 + 號運算子：

```
const Grade operator+(const Grade &rhs,
                      const Grade &lhs) {
  return Add(rhs, lhs);
}
```

  - 當我們運算 + 號時遇到自訂型態 (例如類別) 會去尋找名稱為 **operator+** 的函式
    - 依此類推, - 號會找 operator- 等等

[範例] operator.cpp

Grade 的運算子多載

- 想想我們 **Grade** 要提供哪些運算？
  - 我們希望『可以怎麼用』以及『不能怎麼用』 Grade？

```
Grade a = 10, b = 20;
Grade c;

c = a + b;    // const Grade operator+(Grade, Grade);
c = a + 20;   // const Grade operator+(Grade, int);
c = 20 + a;   // const Grade operator+(int, Grade);

c = c - a;    // const Grade operator-(Grade, Grade);
c = c - 5;    // const Grade operator-(Grade, int);
c = 30 - c;   // const Grade operator-(int, Grade);

c = 2 * c;    // const Grade operator*(int, Grade);
c = c * 2;    // const Grade operator*(Grade, int);

c = c / 2;    // const Grade operator/(Grade, int);
```

預設你什麼都不能用！

我們需要定義幾個函式？

【練習】成績運算

- 試著替 **Grade** 類別新增一個 **>=** 運算：

```
Grade a = 50;
if (a >= 60) {
  printf("及格\n");
} else {
  printf("不及格\n");
}
```

  - 此時 a >= 60 會試著呼叫 **operator>=(a, 60)**，該函式應該要回傳一個 **bool** 型態的值，當 a 有大於等於 60 時回傳 **true**, 反之回傳 **false**
  - 那如果反過來寫 a < 60 會怎樣？

[練習] ex2D.cpp

【練習】自製串流輸出類別

```
class ConsoleOut {};
const ConsoleOut &operator<<(const ConsoleOut &lhs,
                             float rhs);
const ConsoleOut &operator<<(const ConsoleOut &lhs,
                             double rhs);
const ConsoleOut &operator<<(const ConsoleOut &lhs,
                             int rhs);
const ConsoleOut &operator<<(const ConsoleOut &lhs,
                             char rhs);
const ConsoleOut &operator<<(const ConsoleOut &lhs,
                             const char *rhs);
int main() {
  ConsoleOut cout;
  cout << 3.f << '\n' << 3. << '\n' << 3 << '\n';
  cout << "Hello world!\n";
  system("pause");
  return 0;
}
```

[練習] ex2E.cpp

命名空間 [1]

- 在製作大型程式時，時常會面臨名稱重複的問題。尤其在引入多個別人寫好的函式庫時，可能會因為名稱衝突而無法編譯：
  - 在 C 語言裡的解決方法是在取名時加上獨特的名稱:
    - `IplImage, CImage, QImage, ...`
  - 在 C++ 語言裡，我們可以用命名空間 (**namespace**) 來解決這個問題:

```
namespace 命名空間名稱 {
    /* 將變數、函式或類別宣告或定義於此 */
}
```

    - 例如：

```
namespace KenYiLee {
    int Max(int *v, int N);
} // 此時這個函式的全名是 ::KenYiLee::Max
```

[範例] namespace.cpp

命名空間 [2]

- 常見命名空間：
  - 全域命名空間 (::)
  - **std** 命名空間 (::std::)
    - 在 C++ 中我們使用 **<cstdio>** 取代 **<stdio.h>**, **<cstdlib>** 取代 **<stdlib.h>** 等等，主要就是要讓 C 標準函式庫的函式放置在 **::std** 這個命名空間下
- **using** 語法:
  - 指定使用的命名空間或直接指定使用的命名空間成員
    - 例如：
```
using namespace std;
using std::printf;
```
    - **using** 語法跟變數一樣具有可視範圍
      - 從宣告開始到所屬的區塊結束 (右大括號)

[範例] using.cpp

命名空間 [3]

```
void B() { C(); }          // ::B()
void C() {}                // ::C()

namespace A {              // ::A
  void B() { C(); }        // ::A::B()
  void C() {}              // ::A::C()
};

int main() {               // ::main
  B();                     // 哪個 B ?
  A::B();                  // 哪個 B ?

  using namespace A;
  B();                     // 哪個 B ?
  return 0;
}
```

:: 是範圍決議 (scope resolution) 運算子

:: 使用上有點像是我們平常檔案路徑中的 / 或 \

標準輸入輸出串流

- **std::cout** 和 **std::cin** 是定義在 **<iostream>** 內的變數，**std::cout** 是一個輸出串流類別 **(ostream)** 的物件，並多載了 **operator<<**。而 **std::cin** 是一個輸入串流類別 **(istream)** 的物件，並多載了**operator>>**
- 下面程式碼會讀入一個整數後印出：

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

  - **std::endl** 表示換行符號

- 使用這樣的設計法最大的好處是在於擴充性：
  - 我們可以使用運算子多載來輸入或輸出自定類別物件

```
Grade g;
std::cin >> g;
std::cout << g << std::endl;
```

[範例] stream.cpp

【複習】C++ 初探 [1]

- 設計一般類別的時候將該類別提供的操作設為公開函式，其他皆為私有成員
- 函式的參數型態如為大型的類別，參數通常會設定為該類別的參考型態以提升效率。如果想避免引數因透過參考傳遞而被修改時，可以使用固定值參考型態
  - 使用固定值參考型態做完參數型態還有一個好處是他可以參考暫時變數 (物件)，一般的參考型態不能參考暫時變數 (物件)
- 函式的回傳值型態如為類別，一般情況下請設其為固定值 (const) 以避免回傳值被不合理的使用

【複習】C++ 初探 [2]

- 函式多載：因為同名的函式可以有多個，呼叫函式時會依照呼叫的引數型態來決定要呼叫哪一個
- 運算子多載：當運算子的運算元其中一個是自定型態(例如類別)，該運算子執行時會嘗試呼叫名為『operator運算符號』(例如: operator+)的函式，所以我們可以透過定義這樣名稱的函式來定義這個運算子執行時的行為。
- 標準輸入輸出：C++ 提供 <iostream> 內的 std::cin 和 std::cout 兩種物件來做輸入和輸出。因為 cin 和 cout 在 std 的命名空間裡，所以初學者可以用 using namespace std; 來簡化使用。