前言

本篇教學文章分為兩個部分。

第一個部分是對基礎的 C++物件導向程式作文獻的整理與說明，而物件導向是 C 與 C++差異最大的地方，所以建議大家在進入物件導向程式之前，可以先複習 C 或 C++更基礎的資料結構、控制結構、函數或指標的使用與宣告，這可以使我們更透徹地了解物件導向程式與非物件導向程式的差異。

第二個部分是介紹一些可以增進程式生產效率的技巧，像利用最佳化編譯器可以簡單使程式運行速度增加；利用前置處理器可以使 debug 更加的容易；利用資料流重導向與讀檔的技巧使我們在分析大量程式結果更加簡單；利用撰寫腳本讓程式可以自動化的執行；選擇適當的資料結構與運算子讓變數讀寫或計算更快速。

Part 1. C++物件導向程式技巧

1. Classes (類別)

類別是將資料以及函數組織在同一個結構的方法。類別的關鍵字為 class，功能與 C 語言的 struct 類似，不過 C 語言中的 struct 只能包含資料，不能包含函數。

class 宣告的形式如下：

```
class class_name {
    permission_label_1:
        member1;
    permission_label_2:
        member2;
    ...
} object_name;
```

其中，class_name 是用戶定義的類別名稱，而 object_name 是一個或多個宣告的物件名稱。class 宣告的內部包含可以是資料或函數的成員，以及表示權限範圍的標示 permission labels，權限範圍標示可以是以下三個關鍵字任意一個：private:, public: 或 protected:。

它們的意義如下所示：

- private：class 的 private 成員，只有同一個 class 的其他成員或該 class 的 “friend” class 可以使用這些成員。
- protected：class 的 protected 成員，只有同一個 class 的其他成員，該 class 的 “friend” class，或該 class 的 derived classes (子類別) 可以使用這些成員。
- public：class 的 public 成員，任何可以看到這個 class 的地方都可以使用這些成員。

如果我們在定義一個 class 成員的時候沒有宣告其允許範圍，這些成員將被默認為 private 範圍。

例如：

```
class CRectangle {
        int x, y;
    public:
        void set_values (int,int);
        int area (void);
} rect;
```

上面例子定義了一個 class CRectangle 和宣告了該 class 的物件變數 rect 。這個 class 有 4 個成員：屬於 private 部分的兩個整數變數 x 和 y (因為 private 是默認的允許範圍)；以及屬於 public 部分的兩個函數：set_values() 和 area()，這裡只包含了函數的 prototype (原型)。

注意 class 名稱與物件(object)名稱的不同：在上面的例子中，CRectangle 是 class 名稱 (即用戶定義的類型名稱)，而 rect 是一個屬於 CRectangle 類型的物件名稱。它們的區別就像下面例子中類型名 int 和 變數名 a 的區別一樣：

```
int a;
```

int 就相當於是 class 名稱，而 a 是屬於 int 的物件名稱。

在程式中，我們可以通過使用物件名後面加一點再加成員名稱（同使用 C structs 一樣），來引用物件 rect 的任何 public 成員，就像它們只是一般的函數或變數。例如：

```
rect.set_value (3,4);
myarea = rect.area();
```

但我們不能夠引用 x 或 y ，因為它們是該 class 的 private 成員，它們只能夠在該 class 的其它成員中被引用。下面是關於 class CRectangle 的一個較完整的例子：

```
// classes example
#include <iostream.h>
class CRectangle {
        int x, y;
    public:
        void set_values (int,int);
        int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

int main () {
    CRectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area();
}
```

```
area: 12
```

上面程式碼中，使用範圍運算子(雙冒號::) 在一個 class CRectangle 之外定義該 class 函數成員 set_values().。注意，我們在 class CRectangle 內部已經定義了函數 area() 的內容，因為這個函數非常簡單。而對函數 set_values() ，在 class 內部只是定義了它的原型 prototype，而其操作內容是在 class 之外定義的。這種在 class 之外定義其成員的情況必須使用範圍運算子::。

範圍運算子 (::) 宣告了被定義的成員所屬的 class 名稱，並賦予被定義成員適當的範圍屬性，這些範圍屬性與在 class 內部定義成員的屬性是一樣的。例如，在上面的例子中，我們在函數 set_values() 中引用了 private 變數 x 和 y，這些變數只有在 class 內部和它的成員中才是可見的。

在 class 內部直接定義完整的函數，和只定義函數的原型而把具體實現放在 class 外部的唯一區別在於，在第一種情況中，編譯器(compiler) 會自動將函數作為 inline 考慮，而在第二種情況下，函數只是一般的 class 成員函數。

我們把 x 和 y 定義為 private 成員 (記住，如果沒有特別宣告，所有 class 的成員均默認為 private)，原因是我們已經定義了一個設置這些 private 變數值的函數 (set_values()) 。這樣一來，能保護 private 成員在程式的其它地方就沒有辦法直接使用它們。也許在一個這樣簡單的例子中，你無法看到這樣保護兩個變數有什麼意義，但在比較複雜的程式中，這是非常重要的，因為它使得變數不會被意外修改 （只能在允許使用此物件的範圍內修改）。

使用 class 的一個更大的好處是我們可以用它來定義多個不同物件(object)。例如，

接著上面 class CRectangle 的例子，除了物件 rect 之外，我們還可以定義物件 rectb ：

```
// classes example
#include <iostream.h>

class CRectangle {
        int x, y;
    public:
        void set_values (int,int);
        int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}

int main () {
    CRectangle rect, rectb;
    rect.set_values (3,4);
    rectb.set_values (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

注意: 調用函數 rect.area() 與調用 rectb.area()所得到的結果是不一樣的。這是因為每一個 class CRectangle 的物件都擁有它自己的變數 x 和 y，以及它自己的函數 set_value() 和 area()。
這是基於物件(object) 和 物件導向程式設計 (object-oriented programming)的概念的。這個概念中，資料和函數是物件(object)的屬性(properties)，而不是像以前在結構化程式設計 (structured programming) 中所認為的物件(object)是函數參數。在後面的內容中，我們將討論物件導向程式設計的好處。
在這個具體的例子中，我們討論的 class (object 的類型)是 CRectangle，有兩個實例(instance)，或稱物件(object)：rect 和 rectb，每一個有它自己的成員變數和成員函數。

另外，類別不僅可以用關鍵字 class 來定義，也可以用 struct 或 union 來定義。

因為在 C++中類別和結構的概念太相似了，所以這兩個關鍵字 struct 和 class 的作用幾乎是一樣的（也就是說在 C++中 struct 定義的類別也可以有成員函數，而不僅僅有資料成員，C 中的 struct 只能有資料成員)。兩者定義的類別的唯一區別在於由 class 定義的類別所有成員的默認存取權限為 private，而 struct 定義的類

別所有成員默認存取權限為 public。除此之外，兩個關鍵字的作用是相同的。

union 的概念與 struct 和 class 定義的類別不同， 因為 union 在同一時間只能存儲一個資料成員。但是由 union 定義的類別也是可以有成員函數的。union 定義的類別存取權限默認為 public。union 的用法可以參閱其他文獻，這裡不詳細敘述。

2. Constructors and Destructors (建構子與解構子)

物件(object)在生成過程中通常需要初始化變數或分配動態記憶體(能夠在程式執行的時候才決定使用多少記憶體)，以方便我們能夠操作或防止在執行過程中返回意外結果。

例如，在前面的例子中，如果我們在調用函數 set_values( ) 之前就調用了函數 area()，將會產生什麼樣的結果呢？可能會是一個不確定的值，因為成員 x 和 y 還沒有被賦於任何值。

為了避免這種情況發生，一個 class 可以包含一個特殊的函數：建構子 constructor，它透過一個與 class 同名的函數來定義。當且僅當要生成一個 class 的新的物件的時候或給該 class 的一個物件分配記憶體的時候，這個建構子將自動被調用。下面，我們將實現包含一個建構子的 CRectangle ：

```
// classes example
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 30
```

這個例子的輸出結果與前面一個沒有區別。在這個例子中，我們只是把函數

set_values()換成了 class 的建構子 constructor。注意這裡參數是如何在 class 物件生成的時候傳遞給建構子的：

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

同時你可以看到，建構子的原型和實現中都沒有返回值(return value)，也沒有 void 型別宣告。建構子必須這樣寫。一個建構子永遠沒有返回值，也不用宣告成 void，就像在例子中看到的。

解構子 destructor 完成相反的功能。它在 objects 被從記憶體中釋放的時候被自動調用。釋放可能是因為它存在的範圍已經結束了（例如，如果 object 被定義為一個函數內的 local (區域) 物件變數，而該函數結束了）；或者是因為它是一個動態分配的物件，而被使用運算子 delete 釋放了。

解構子必須與 class 同名，加上水波號 tilde (~) 首碼，一樣必須無返回值。

解構子特別適用於當一個物件被動態分別記憶體空間，而在物件被銷毀的時我們希望釋放它所佔用的空間的時候。例如：

```
// example on constructors and destructors
#include <iostream.h>

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
    width = new int;
    height = new int;
    *width = a;
    *height = b;
}

CRectangle::~CRectangle () {
    delete width;
    delete height;
}

int main () {
    CRectangle rect (3,4), rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}
```

```
rect area: 12
rectb area: 30
```

3. Overloading Constructors (建構子重載)

像其它函數一樣，一個建構子也可以被多次重載(overload)為同樣名字的函數，但要有不同的參數類型或參數個數。記住，編譯器會調用與在調用時刻要求的參數類型和個數一樣的那個函數。在這裡則是調用與類別物件被宣告時一樣的那個建構子。

實際上，當我們定義一個 class 而沒有定義建構子的時候，編譯器會自動假設兩個重載的建構子 (預設建構子"default constructor" 和複製建構子"copy constructor")。例如，對以下 class：

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
};
```

沒有定義建構子，編譯器自動假設它有以下 constructor 成員函數：

- Empty constructor：它是一個沒有任何參數的建構子，被定義為 nop (沒有語句)。它什麼都不做。

```
CExample::CExample () { };
```

- Copy constructor：它是一個只有一個參數的建構子，該參數是這個 class 的一個物件，這個函數的功能是將被傳入的物件（object）的所有非靜態（non-static）成員變數的值都複製給自身這個 object。

```
CExample::CExample (const CExample& rv) {
    a=rv.a;  b=rv.b;  c=rv.c;
}
```

注意：這兩個預設建構子（empty construction 和 copy constructor ）只有在沒有其它建構子被定義的情況下才存在。如果任何其它有任意參數的建構子被定義了，這兩個建構子就都不存在了。在這種情況下，如果你想要有 empty construction 和 copy constructor ，就必需要自己定義它們。

當然，如果你也可以重載 class 的建構子，定義有不同的參數或完全沒有參數的建構子，見如下例子：

```
// overloading class constructors
#include <iostream.h>

Class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
    width = 5;
    height = 5;
}

CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
}
```

```
rect area: 12
rectb area: 25
```

在上面的例子中，rectb 被宣告的時候沒有參數，所以它被使用沒有參數的建構子進行初始化，也就是 width 和 height 都被賦值為 5。

注意在我們宣告一個新的 object 的時候，如果不想傳入參數，則不需要寫括弧()：

```
CRectangle rectb; // right
CRectangle rectb(); // wrong!
```

4. Pointers to Classes (類別指標)

類別也是可以有指標的，要定義類別的指標，我們只需要認識到，類別一旦被定義就成為一種有效的資料類型，因此只需要用類別的名字作為指標的名字就可以了。例如：

```
CRectangle * prect;
```

prect 是一個指向 class CRectangle 物件的指標。

就像資料結構中的情況一樣，要想直接引用一個由指標指向的物件(object)中的成員，需要使用運算子 ->。這裡是一個例子，顯示了幾種可能出現的情況：

```
// pointer to classes example
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
};

void CRectangle::set_values (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle a, *b, *c;
    CRectangle * d = new CRectangle[2];
    b= new CRectangle;
    c= &a;
    a.set_values (1,2);
    b->set_values (3,4);
    d->set_values (5,6);
    d[1].set_values (7,8);
    cout << "a area: " << a.area() << endl;
    cout << "*b area: " << b->area() << endl;
    cout << "*c area: " << c->area() << endl;
    cout << "d[0] area: " << d[0].area() << endl;
    cout << "d[1] area: " << d[1].area() << endl;
    return 0;
}
```

```
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56
```

以下解釋是怎樣讀前面例子中出現的一些指標和類別運算子 (*, &, ., ->, [ ])：

- *x 讀作: pointed by x （由 x 指向的）
- &x 讀作: address of x（x 的地址）

- x.y 讀作: member y of object x （對象 x 的成員 y）
- (*x).y 讀作: member y of object pointed by x（由 x 指向的對象的成員 y）
- x->y 讀作: member y of object pointed by x (同上一個等價)
- x[0] 讀作: first object pointed by x（由 x 指向的第一個對象）
- x[1] 讀作: second object pointed by x（由 x 指向的第二個對象）
- x[n] 讀作: (n+1)th object pointed by x（由 x 指向的第 n+1 個對象）
-

這些指標與運算子的邏輯含義，一定要搞清楚。它們不只在類別會用到，在其他資料結構中也常常出現。

5. Overloading Operators (運算子重載)

C++ 實現了在類別(class)之間也可以使用標準運算子，而不只有在基礎資料型別之間使用而已。例如：

```
int a, b, c;
a = b + c;
```

上述例子是有效操作，因為加號兩邊的變數都是基礎資料型別。然而，我們是否可以進行下面的操作：

```
struct { char product [50]; float price; } a, b, c;
a = b + c;
```

將一個類別 class (或結構 struct)的物件賦給另一個同種類型的物件是允許的(通過使用預設的複製建構子 copy constructor)。但相加操作就有可能產生錯誤，理論上它在非基礎資料型別之間是無效的。

但歸功於 C++ 的運算子重載（overload）能力，我們可以完成這個操作。像上述例子中組合類型(有 char 與 float 成員)的物件在 C++中可以透過運算子重載來完成原本不能被接受的相加操作，我們甚至可以修改這些運算子操作的效果。以下是所有可以被重載的運算子列表：

```
+    -    *    /    =    <    >    +=   -=   *=   /=   <<   >>
<<=  >>=  ==   !=   <=   >=   ++   --   %    &    ^    !    |
  ~    &=   ^=   |=   &&   ||   %=   []   ()   new  delete
```

要想重載一個運算子，我們只需要編寫一個成員函數，名為 operator ，後面跟我們要重載的運算子，遵循以下原型定義：

```
type operator sign (parameters);
```

這裡是一個運算子 +的例子。我們要計算二維向量(bidimensional vector) a(3,1) 與 b(1,2)的和。兩個二維向量相加的操作很簡單，就是將兩個 x 軸的值相加獲得結果的 x 軸值，將兩個 y 軸值相加獲得結果的 y 值。在這個例子裡，結果是

(3+1,1+2) = (4,3)。

```
// vectors: overloading operators example
#include <iostream.h>

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
    x = a;
    y = b;
}

CVector CVector::operator+ (CVector param) {
    CVector temp;
    temp.x = x + param.x;
    temp.y = y + param.y;
    return (temp);
}

int main () {
    CVector a (3,1);
    CVector b (1,2);
    CVector c;
    c = a + b;
    cout << c.x << "," << c.y;
    return 0;
}
```

```
4,3
```

為什麼看到這麼多遍的 Cvector？那是因為其中有些是指 class 名稱 CVector ，而另一些是以它命名的函數名稱，不要把它們搞混了：

```
CVector (int, int);           // 函數名稱 CVector (constructor)
CVector operator+ (CVector);   // 函數 operator+ 返回 CVector 類型的值
```

Class CVector 的函數 operator+ 是對數學運算子+進行重載的函數。這個函數可以用以下兩種方法進行調用：

```
c = a + b;
c = a.operator+ (b);
```

注意：我們在這個例子中包括了一個空建構子 empty constructor (無參數)，而且我們將它定義為無任何操作：

```
CVector ( ) { };
```

因為例子中已經有另一個建構子，所以我們必須自己宣告空建構子。如果我們沒有宣告空建構子的話，main( )中包含的語句：

```
CVector c;
```

將會變成不合法。

```
CVector (int, int);
```

另外，如果我們不像上面這樣定義一個有輸入參數的建構子的話，main( )中包含的語句：

```
CVector a;
CVector b;
```

也會變成不合法。

因為空語句塊 (no-op block)並不是一種值得推薦的建構子實現方式，因為它不能實現一個建構子至少應該完成的基本功能，也就是初始化 class 中的所有變數。在我們的例子中，這個建構子沒有完成對變數 x 和 y 的定義。因此一個更值得推薦的建構子定義應該像下面這樣：

```
CVector ( ) { x=0; y=0; };
```

就像一個 class 預設包含一個空建構子和一個複製建構子一樣，它同時包含一個對賦值運算子 assignation operator (=)的默認定義，該運算子用於兩個同類別物件之間。這個運算子將其參數物件(等號右邊的物件) 的所有非靜態 (non-static) 資料成員複製給其左邊的物件。當然，你也可以將它重新定義為你想要的任何功能，例如，只複製某些特定的 class 成員。

重載一個運算子並不一定要符合其原本常規的數學含義，雖然這是推薦的。例如，雖然我們可以將運算子+定義為取兩個物件的差值，或用==運算子將一個物件賦為 0，但這樣做是沒有什麼邏輯意義的。
雖然函數 operator+ 的原型定義看起來很明顯，因為它取運算子右邊的物件為其左邊物件的函數 operator+的參數，其它的運算子就不一定這麼明顯了。以下清單總結了不同的運算子函數是怎樣定義宣告的 (用運算子替換每個@):

Expression	Operator (@)	Function member	Global function
@a	+ - * & ! ~ ++ --	A::operator@( )	operator@(A)
a@	++ --	A::operator@(int)	operator@(A, int)

a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A, B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= [ ]	A::operator@(B)	-
a(b, c...)	( )	A::operator()(B, C...)	-
a->b	->	A::operator->()	-

這裡 a 是 class A 的一個物件，b 是 class B 的一個物件，c 是 class C 的一個物件。

從上表可以看出有兩種方法重載一些 class 運算子：作為類別成員函數（member function）或作為全域函數(global function)。它們的用法沒有區別，但是請注意全域函數如果不是該 class 的 friend (friend 的含義將在後面的章節解釋)函數，則不能訪問該 class 的 private 或 protected 成員。

6. This (類別中 this 關鍵字)

關鍵字 this 通常被用在一個 class 內部，指正在被執行的該 class 的物件(object)在記憶體中的位址。它是一個指標，其值永遠是自身 object 的位址。

它可以被用來檢查傳入一個物件的成員函數的參數是否是該物件本身。例如：

```
// this
#include <iostream.h>

class CDummy {
  public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param) {
    if (&param == this) return 1;
    else return 0;
}

int main () {
    CDummy a;
    CDummy* b = &a;
    if ( b->isitme(a) )
        cout << "yes, &a is b";
    return 0;
}
```

```
yes, &a is b
```

它還經常被用在成員函數 operator= 中，用來返回物件的指標(避免使用臨時物件)。以下用前面看到的向量(vector)的例子來看一下函數 operator= 是怎樣實現的：

```
CVector& CVector::operator= (const CVector& param) {
   x=param.x;
   y=param.y;
   return *this;
}
```

實際上，如果我們沒有定義成員函數 operator=，編譯器自動為該 class 生成的程式碼有可能就是這個樣子的。

7. Static Members (靜態成員)

一個 class 可以包含靜態成員(static members)，可以是資料，也可以是函數。

一個 class 的靜態資料成員也被稱作類別變數"class variables"，因為它們的內容不依賴於某個物件，而是對同一個 class 的所有物件具有相同的值。

例如，它可以被用來計算一個 class 宣告的物件的個數，見以下程式碼：

```
// static members in classes
#include <iostream.h>

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
    CDummy a;
    CDummy b[5];
    CDummy * c = new CDummy;
    cout << a.n << endl;
    delete c;
    cout << CDummy::n << endl;
    return 0;
}
```

```
7
6
```

實際上，靜態成員與全域變數(global variable)具有相同的屬性，但它享有類別(class)的範圍。因此，根據 ANSI-C++ 標準，為了避免它們被多次重複宣告，在 class 的宣告中只能夠包括 static member 的原型(宣告)，而不能夠包括其定義(初

始化操作)。為了初始化一個靜態資料成員，我們必須在 class 之外(在全域範圍內)，包括一個正式的定義，就像上面例子中做法一樣。

因為它對同一個 class 的所有 object 是同一個值，所以它可以被作為該 class 的任何 object 的成員所引用，或者直接被作為 class 的成員引用：

```
cout << a.n;
cout << CDummy::n;
```

以上兩個調用都指同一個變數：class CDummy 裡的 static 變數 n 。

再一次提醒，它其實是一個全域變數。唯一的不同是它的名字跟在 class 的後面。

就像我們會在 class 中包含 static 資料一樣，我們也可以使它包含 static 函數。它們表示相同的含義：static 函數是全域函數(global functions)，但是像一個指定 class 的物件成員一樣被調用。它們只能夠引用 static 資料，永遠不能引用 class 的非靜態(nonstatic)成員。它們也不能夠使用關鍵字 this，因為 this 實際引用了一個物件指標，但這些 static 函數卻不是任何 object 的成員，而是 class 的直接成員。

8. Friend (好友)

■ 好友函數(Friend functions)

在前面我們已經看到了對 class 的不同成員存在 3 個層次的內部保護：public，protected 和 private。在成員為 protected 和 private 的情況下，它們不能夠被從所在的 class 以外的部分引用。然而，這個規則可以通過在一個 class 中使用關鍵字 friend 來繞過，這樣我們可以允許一個外部函數獲得訪問 class 的 protected 和 private 成員的能力。

為了實現允許一個外部函數訪問 class 的 private 和 protected 成員，我們必須在 class 內部用關鍵字 friend 來宣告該外部函數的原型，以指定允許該函數共用 class 的成員。在下面的例子中我們宣告了一個 friend 函數 duplicate：

```
// friend functions
#include <iostream.h>
class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};
```

```
24
```

```
void CRectangle::set_values (int a, int b) {
    width = a;
    height = b;
}

CRectangle duplicate (CRectangle rectparam) {
    CRectangle rectres;
    rectres.width = rectparam.width*2;
    rectres.height = rectparam.height*2;
    return (rectres);
}

int main () {
    CRectangle rect, rectb;
    rect.set_values (2,3);
    rectb = duplicate (rect);
    cout << rectb.area();
}
```

函數 duplicate 是 CRectangle 的 friend，因此在該函數之內，我們可以訪問 CRectangle 類型的各個 object 的成員 width 和 height。注意，在 duplicate()的宣告中，及其在後面 main()裡被調用的時候，我們並沒有把 duplicate 當作 class CRectangle 的成員，事實上它也不是。

friend 函數可以被用來實現兩個不同 class 之間的操作。廣義來說，使用 friend 函數是物件導向程式設計之外的方法，因此，如果可能，應儘量使用 class 的成員函數來完成這些操作。比如在以上的例子中，將函數 duplicate() 集成在 class CRectangle 可以使程式更短。

■ 好友類別 (Friend classes)

就像我們可以定義一個 friend 函數，我們也可以定義一個 class 是另一個的 friend，以便第二個 class 訪問第一個 class 的 protected 和 private 成員。

```
// friend class
#include <iostream.h>
class CSquare;
class CRectangle {
    int width, height;
  public:
    int area (void) {return (width * height);}
    void convert (CSquare a);
};
```

```
class CSquare {
  private:
    int side;
  public:
    void set_side (int a){side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
    width = a.side;
    height = a.side;
}

int main () {
    CSquare sqr;
    CRectangle rect;
    sqr.set_side(4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
}
```

在這個例子中，我們宣告了 CRectangle 是 CSquare 的 friend，因此 CRectangle 可以訪問 CSquare 的 protected 和 private 成員，更具體地說，可以訪問 CSquare::side，它定義了正方形的邊長。

在上面程式的第一句就是 class CSquare 的原型，這是必需的，因為在 CRectangle 的宣告中我們引用了 CSquare (作為 convert()的參數)。CSquare 的定義在 CRectangle 的後面，因此如果我們沒有在這個 class 之前包含一個 CSquare 的原型宣告，它在 CRectangle 中就是不可見的。

這裡要考慮到，如果沒有特別指明，好友關係（friendships）並不是相互的。在我們的 CSquare 例子中，CRectangle 是一個 friend 類別，但因為 CRectangle 並沒有對 CSquare 作相應的宣告，因此 CRectangle 可以訪問 CSquare 的 protected 和 private 成員，但反過來並不行，除非我們將 CSquare 也定義為 CRectangle 的 friend。

9. Inheritance (繼承)

■ 類別之間的繼承(Inheritance between classes)

類別的一個重要特徵是繼承，這使得我們可以基於一個類別生成另一個類別的物件，以便使後者擁有前者的某些成員，再加上它自己的一些成員。例如，假設我們要宣告一系列類型的多邊形，比如長方形 CRectangle 或三角形 CTriangle。它

們有一些共同的特徵，比如都可以只用兩條邊來描述：高（height）和底（base）。

這個特點可以用一個類別 CPolygon 來表示，基於這個類別我們可以衍生出上面提到的兩個類別 CRectangle 和 CTriangle 。

類別 CPolygon 包含所有多邊形共有的成員。在我們的例子裡就是： width 和 height。而 CRectangle 和 CTriangle 將視為 CPolygon 的子類別(derived classes)。

由其它類別衍生而來的子類別會繼承父類別的所有可視成員，意思是說，如果一個父類別包含成員 A ，而我們將它衍生為另一個包含成員 B 的類別，則這個子類別將同時包含 A 和 B。

要定義一個類別的子類別，我們必須在子類別的宣告中使用冒號(colon)運算子: ，如下所示：

```
class derived_class_name: public base_class_name;
```

這裡 derived_class_name 為子類別(derived class)的名稱，base_class_name 為父類別(base class)名稱。public 也可以根據需要換為 protected 或 private，描述了被繼承的子類別存取父類別的權限，我們在以下例子會看到：

```
// derived classes
#include <iostream.h>
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) { width=a; height=b;}
};

class CRectangle: public CPolygon {
    public:
        int area (void){ return (width * height); }
};

class CTriangle: public CPolygon {
    public:
        int area (void){ return (width * height / 2); }
};
```

```
20
10
```

```
int main () {
    CRectangle rect;
    CTriangle trgl;
    rect.set_values (4,5);
    trgl.set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
```

如上所示，class CRectangle 和 CTriangle 的每一個物件都包含 CPolygon 的成員，即： width, height 和 set_values()。

識別字 protected 與 private 類似，它們的唯一區別在繼承時才表現出來。當定義一個子類別的時候，父類別的 protected 成員可以被子類別的其它成員所使用，然而 private 成員就不可以。因為我們希望 CPolygon 的成員 width 和 height 能夠被子類 CRectangle 和 CTriangle 的成員所訪問，而不只是被 CPolygon 自身的成員操作，我們使用了 protected 存取權限，而不是 private。

下表總結了不同存取權限類型：

訪問物件	public	protected	private
本類別的成員	yes	yes	yes
子類別的成員	yes	yes	no
非類別成員	yes	no	no

這裡"非類別成員"指從 class 以外的任何地方引用，例如從 main()中，從其它的 class 中或從全域(global)或本地(local)的任何函數中。
在我們的例子中，CRectangle 和 CTriangle 繼承來的成員與父類別 CPolygon 擁有同樣的訪問限制：

```
CPolygon::width          // protected access
CRectangle::width        // protected access
CPolygon::set_values()    // public access
CRectangle::set_values()  // public access
```

這是因為我們在繼承的時候使用的是 public：

```
class CRectangle: public CPolygon;
```

這裡關鍵字 public 表示子類別(CRectangle)從父類別(CPolygon)繼承的成員所必須獲得最低程度保護程度。這種被繼承成員最低程度的訪問限制可以透過使用 protected 或 private 而不是 public 來改變。例如，daughter 是 mother 的一個子類別，我們可以這樣定義：

```
class daughter: protected mother;
```

protected 將使得 daughter 從 mother 處繼承的成員其最低訪問限制為 protected。也就是說，原來 mother 中的所有 public 成員到 daughter 中將會成為 protected 成員。這最低存取權限限制只是建立在從 mother 中所繼承來的成員上的，並不代表 daughter 不能有它自己的 public 成員。

最常用的繼承限制除了 public 外就是 private ，它被用來將父類別完全封裝起來，因為在這種情況下，除了子類別自身外，其它任何程式都不能訪問那些從父類別繼承而來的成員。不過大多數情況下繼承都是使用 public 的。

如果沒有明確寫出訪問限制，所有由關鍵字 class 衍生的類別被默認為 private ，而所有由關鍵字 struct 衍生的類別被默認為 public。

■ 什麼會從父類別中繼承? (What is inherited from the base class?)
理論上說，子類別(drived class)繼承了父類別(base class)中 public 與 protected 的所有成員，除了：

- 建構子 Constructor 和解構子 destructor
- operator=() 成員
- friends

雖然父類別的建構子和解構子沒有被繼承，但是當一個子類別的 object 被生成或銷毀的時候，其父類別的預設建構子 (即，沒有任何參數的建構子)和解構子總是被自動調用的。

如果父類別沒有預設建構子，或你希望當子類別生成新的 object 時，父類別的某個重載的建構子被調用，你需要在子類別的每一個建構子的定義中指定它：

```
derived_class_name (parameters) : base_class_name (parameters) {}
```

例如：

```
// constructors and derivated classes
#include <iostream.h>

class mother {
    public:
        mother ()
        { cout << "mother: no parameters\n"; }
        mother (int a)
        { cout << "mother: int parameter\n"; }
};
```

```
mother: no parameters
daughter: int
parameter

mother: int parameter
son: int parameter
```

```
class daughter : public mother {
    public:
        daughter (int a)
        { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
    public:
        son (int a) : mother (a)
        { cout << "son: int parameter\n\n"; }
};

int main () {
    daughter cynthia (1);
    son daniel(1);
    return 0;
}
```

當一個新的 daughter object 生成的時候 mother 的哪一個建構子被調用了，而當新的 son object 生成的時候，又是哪一個建構子被調用了。不同的建構子被調用是因為 daughter 和 son 的建構子定義不同：

```
daughter (int a)          // 沒有特別指定：調用默認 constructor
son (int a) : mother (a)  // 指定了 constructor: 調用被指定的建構子
```

1. 多重繼承(Multiple inheritance)

在 C++ 中，一個 class 可以從多個 class 中繼承屬性或函數，只需要在子類別的宣告用逗號將不同父類別分開就可以了。例如，如果我們已有一個特殊的 class COutput 可以實現向螢幕列印的功能，我們同時希望我們的類別 CRectangle 和 CTriangle 在 CPolygon 之外還繼承 COutput 的成員，我們可以這樣寫：

```
class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {
```

以下是一個完整的例子：

```
// multiple inheritance
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class COutput {
  public:
    void output (int i);
};

void COutput::output (int i) {
    cout << i << endl;
}

class CRectangle: public CPolygon, public COutput {
  public:
    int area (void)
      { return (width * height); }
};

class CTriangle: public CPolygon, public COutput {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    rect.set_values (4,5);
    trgl.set_values (4,5);
    rect.output (rect.area());
    trgl.output (trgl.area());
    return 0;
}
```

```
20
10
```

10. Pointers to Base Class (父類別的指標)

繼承的好處之一是一個指向子類別(derived class)的指標與一個指向父類別(base class)的指標是 type-compatible 的。 本節就是重點介紹如何利用 C++的這一重要特性。例如，我們將結合 C++的這個功能，重寫前面關於長方形 rectangle 和三角形 triangle 的程式：

```
// pointers to base class
#include <iostream.h>

class CPolygon {
protected:
    int width, height;
public:
    void set_values (int a, int b) {
        width=a; height=b;
    }
};

class CRectangle: public CPolygon {
public:
    int area (void) {
        return (width * height);
    }
};

class CTriangle: public CPolygon {
public:
    int area (void) {
        return (width * height / 2);
    }
};
```

```
20
10
```

```
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
    return 0;
}
```

在主函數 main 中定義了兩個指向 class CPolygon 的物件的指標，即 *ppoly1 和 *ppoly2。 它們被賦值為 rect 和 trgl 的地址，因為 rect 和 trgl 是 CPolygon 子類別的物件，因此這種賦值是有效的。

使用*ppoly1 和 *ppoly2 取代 rect 和 trgl 的唯一限制是*ppoly1 和 *ppoly2 是 CPolygon* 類型的，因此我們只能夠引用 CRectangle 和 CTriangle 從父類別 CPolygon 中繼承的成員。正是由於這個原因，我們不能夠使用*ppoly1 和 *ppoly2 來調用成員函數 area()，而只能使用 rect 和 trgl 來調用這個函數。

要想使 CPolygon 的指標承認 area()為合法成員函數，必須在父類別中宣告它，而不能只在子類別進行宣告，詳細使用方法在下面會說明。

11. Polymorphism：Virtual Members (多態：虛擬成員)

如果想在父類別中定義一個成員留待子類別中進行細化，我們必須在它前面加關鍵字 virtual ，以便可以使用指標對指向相應的物件進行操作。以下是例子：

```
// virtual members
#include <iostream.h>

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

```
20
10
0
```

```
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void) {
            return (width * height / 2);
        }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
    return 0;
}
```

現在這三個類別(CPolygon, CRectangle 和 CTriangle) 都有同樣的成員：width, height, set_values() 和 area()。

area() 被定義為 virtual 是因為它後來在子類別中被細化了。你可以做一個試驗，如果在代碼種去掉這個關鍵字(virtual)，然後再執行這個程式，三個多邊形的面積計算結果都將是 0 而不是 20,10,0。這是因為沒有了關鍵字 virtual ，程式執行不再根據實際物件的不用而調用相應 area() 函數(即分別為 CRectangle::area(), CTriangle::area() 和 CPolygon::area())，取而代之，程式將全部調用 CPolygon::area()，因為這些調用是通過 CPolygon 類型的指標進行的。

因此，關鍵字 virtual 的作用就是在當使用父類別的指標的時候，使子類別中與父類別同名的成員在適當的時候被調用，如前面例子中所示。

注意，雖然本身被定義為虛擬類型，我們還是可以宣告一個 CPolygon 類型的物件並調用它的 area() 函數，它將返回 0 ，如前面例子結果所示。

12. Polymorphism：Abstract Base Classes (多態：抽象父類別)

基本的抽象類別與我們前面例子中的類別 CPolygon 非常相似，唯一的區別是在我們前面的例子中，我們已經為類別 CPolygon 的物件（例如物件 poly）定義了一個有效地 area()函數，而在一個抽象父類別（abstract base class）中，我們可以對它不定義，而簡單得在函式宣告後面寫 =0 (等於 0)。

類別 CPolygon 可以寫成這樣：

```
// abstract class CPolygon
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a;
            height=b;
        }
        virtual int area (void) =0;
};
```

注意我們是如何在 virtual int area (void)加=0 來代替函數的具體實現的。這種函數被稱為純虛擬函數（pure virtual function），而所有包含純虛擬函數的類別被稱為抽象父類別(abstract base classes)。

抽象父類別的最大不同是它不能夠有實例(物件)，但我們可以定義指向它的指標。因此，像這樣的宣告：

```
CPolygon poly;
```

對於前面定義的抽象父類別是不合法的。

然而，指標：

```
CPolygon * ppoly1;
CPolygon * ppoly2
```

這是因為該類別包含的純虛擬函數(pure virtual function) 是沒有被實現的，所以這是完全合法的。

下面是一個完整的例子：

```
// virtual members
#include <iostream.h>
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a;
            height=b;
        }
        virtual int area (void) =0;
};
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void) {
            return (width * height / 2);
        }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
    return 0;
}
```

```
20
10
```

這段程式用同一種類型的指標(CPolygon*)指向不同類別的對象，這一點非常有用。 想像一下，現在我們可以寫一個 CPolygon 的成員函數，使得它可以將函數 area()的結果列印到螢幕上，而不必考慮具體是為哪一個子類別。

```
// virtual members
#include <iostream.h>
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a;
            height=b;
        }
        virtual int area (void) =0;
        void printarea (void) {
            cout << this->area() << endl;
        }
};
class CRectangle: public CPolygon {
    public:
        int area (void) { return (width * height); }
};
class CTriangle: public CPolygon {
    public:
        int area (void) {
            return (width * height / 2);
        }
};
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    return 0;
}
```

```
20
10
```

記住，this 代表正在被程式執行的這一個物件的指標。

抽象父類別和虛擬成員賦予了 C++ 多態(polymorphic)的特徵，使得物件導向的程式設計 object-oriented programming 成為一個有用的工具。這裡只是展示了這些功能最簡單的用途。想像一下如果在物件陣列或動態分配的物件上使用這些功能，將會節省多少麻煩。

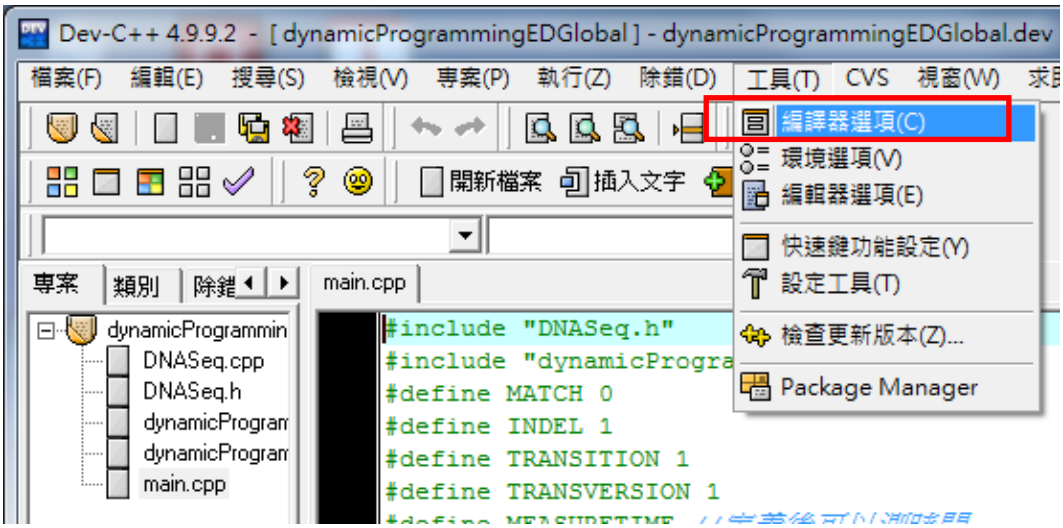
Part 2. 增進程式生產效率的技巧

1. Compiler Optimization (最佳化編譯器)

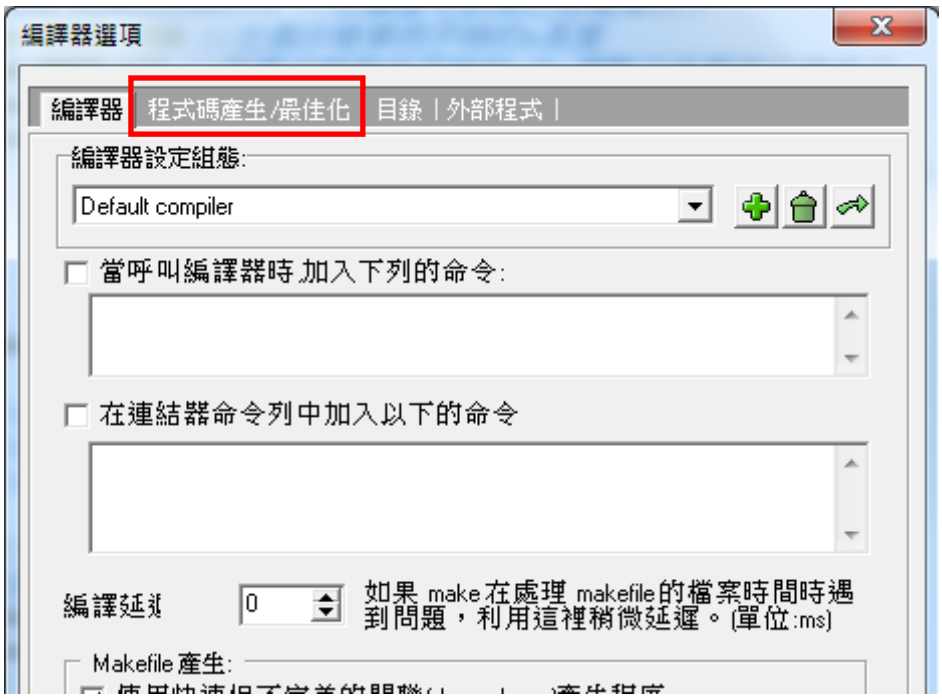
最佳化編譯是編譯器對一個程式在編譯時作程序上的調整，使得效能能夠最佳化。通常目標是要縮短執行的時間，或者是縮小整個程式所需的記憶體，底下會示範在 Dev-C++以及 Visual C++下怎麼最佳化編譯程式。

一般來說，編譯器提供三種不同程度的最佳化：O1(minimize size)，O2(maximize speed)，及 O3(full optimization)，數字越大編譯器作的最佳化越多，可以根據所需的狀況以及穩定度來做選擇。

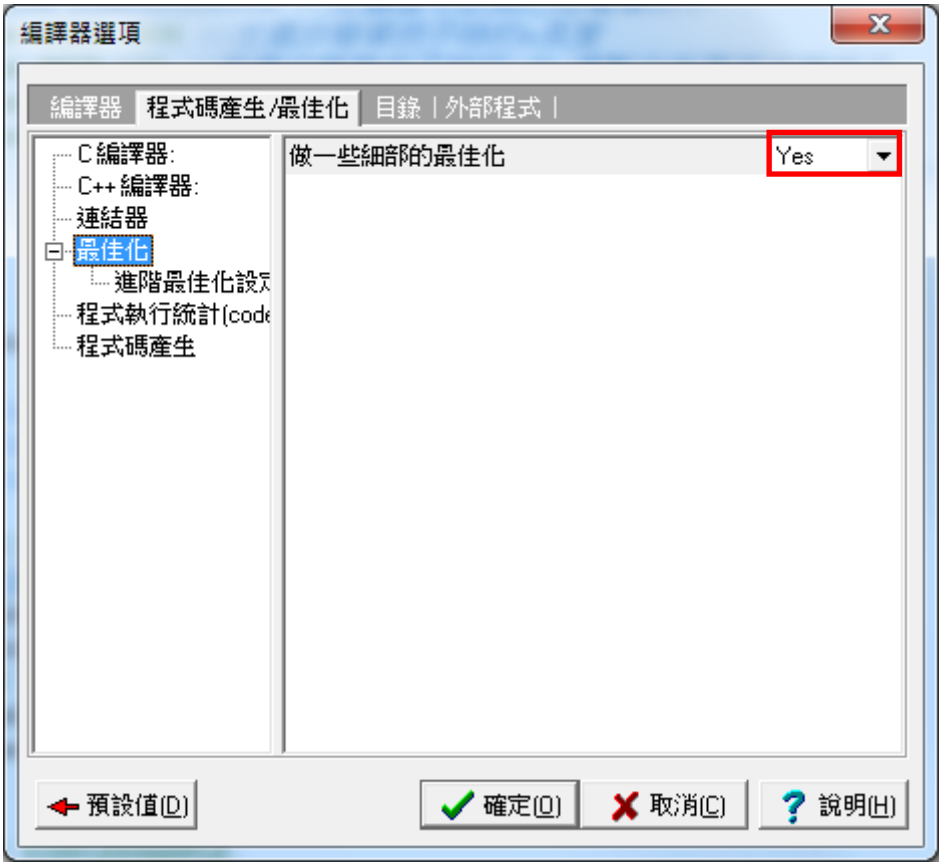
- Dev-C++設定最佳化編譯過程：
  - 選擇工具->編譯器選項



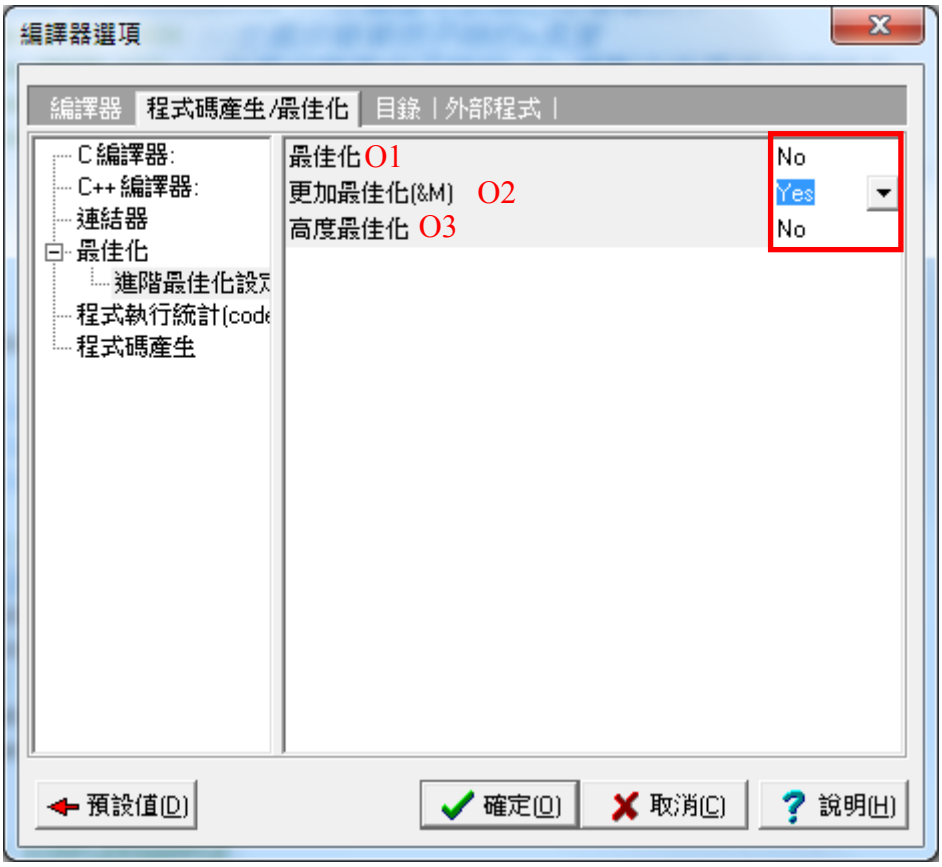
  - 選擇程式碼產生/最佳化



◆ 選擇最佳化->做一些細部的最佳化選 yes



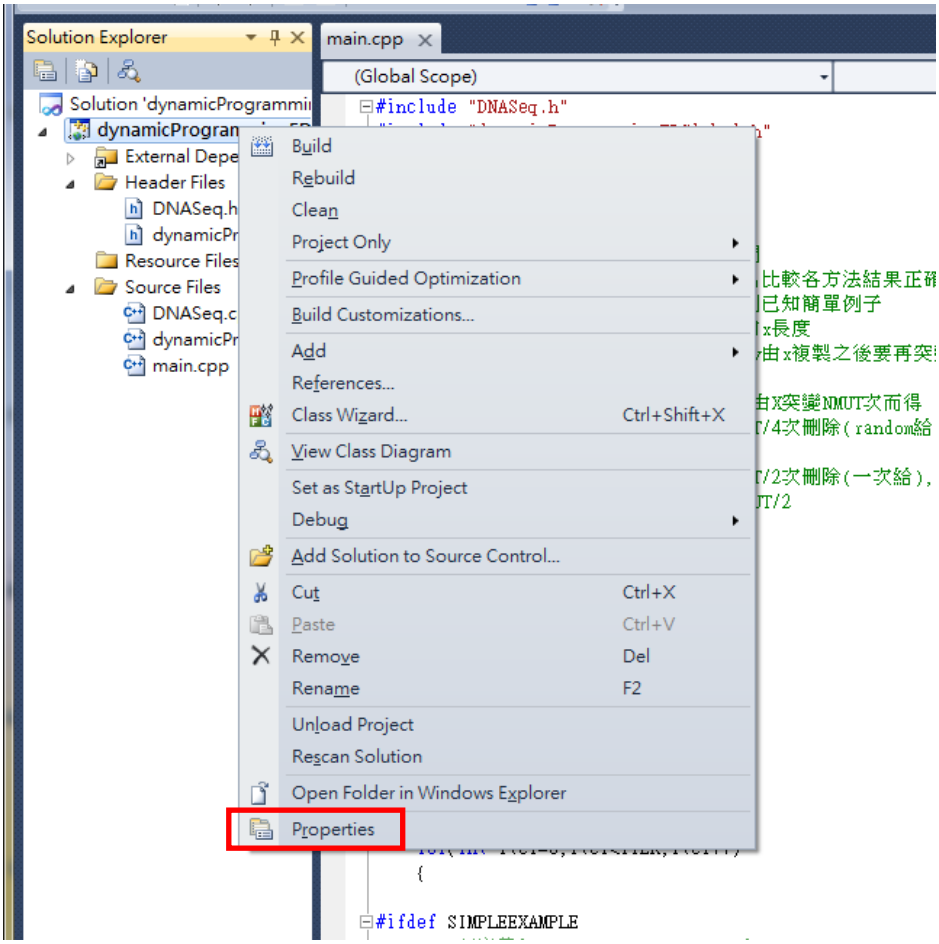
◆ 進階最佳化設定->決定所需的最佳化程度選 yes



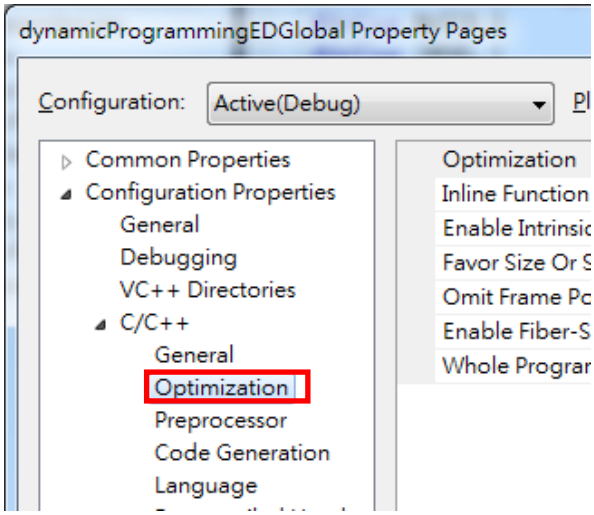
◆ 刪除前一次編譯產生的 output 檔案，再重新編譯。

■ Visual-C++設定最佳化編譯過程：

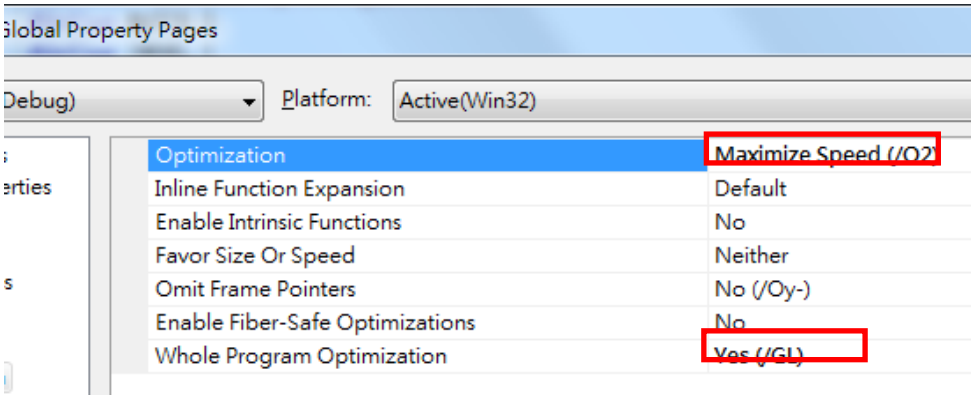
◆ 在專案名稱按右鍵-> 選擇 Properties



◆ 選擇 Configuration Properties->C/C++->Optimization



◆ Whole Program Optimization 選 yes; Optimization 依所需的最佳化選擇



◆ 刪除前一次編譯產生的 output 檔案，再重新編譯。

2. Preprocessor Directives (前置處理器指令)

預處理指令是我們寫在程式碼中的給前置處理器(preprocessor)的命令，而不是程式本身的語句。前置處理器在我們編譯一個 C++程式時由編譯器自動執行，它負責控制對程式碼的第一次驗證和處理。

所有這些指令必須寫在單獨的一行中，它們不需要加結尾的分號;。

■ #define

#define ，可以被用來生成巨集定義常數(defined constantants 或 macros)，它的形式是：

```
#define name value
```

它的作用是定義一個叫做 name 的巨集定義，然後每當在程式中遇到這個名字的時候，它就會被 value 代替，例如：

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
char str2[MAX_WIDTH];
```

它定義了兩個最多可以儲存 100 個字元的字串。

#define 也可以被用來定義巨集函數：

```
#define getmax(a,b) a>b?a:b
int x=5, y;
y = getmax(x,2);
```

這段程式執行後 y 的值為 5 。

■ #undef

#undef 完成與 #define 相反的工作，它取消對參數的巨集定義：

```
#define MAX_WIDTH 100
char str1[MAX_WIDTH];
#undef MAX_WIDTH
#define MAX_WIDTH 200
char str2[MAX_WIDTH];
```

■ #ifdef, #ifndef, #if, #endif, #else and #elif

這些指令可以使程式的一部分在某種條件下被忽略。

#ifdef 可以使一段程式只有在某個指定常量已經被定義了的情況下才被編譯，無論被定義的值是什麼。它的操作是：

```
#ifdef name
// code here
#endif
```

例如：

```
#ifdef MAX_WIDTH
char str[MAX_WIDTH];
#endif
```

在這個例子中，語句 char str[MAX_WIDTH]; 只有在巨集定義常量 MAX_WIDTH 已經被定義的情況下才被編譯器考慮，不管它的值是什麼。如果它還沒有被定義，這一行程式碼則不會被包括在程式中。

#ifndef 與 #ifdef 作用相反：在指令#ifndef 和 #endif 之間的程式碼只有在某個常量沒有被定義的情況下才被編譯，例如：

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 100
#endif
char str[MAX_WIDTH];
```

這個例子中，如果當處理到這段程式碼的時候 MAX_WIDTH 還沒有被定義，則它會被定義為值 100。而如果它已經被定義了，那麼它會保持原值 (因為#define 語句這一行不會被執行) 。

指令#if, #else 和 #elif (elif = else if) 用來使得其後面所跟的程式部分只有在特定條件下才被編譯。這些條件只能夠是常量運算式，例如：

```
#if MAX_WIDTH>200
#undef MAX_WIDTH
#define MAX_WIDTH 200

#elif MAX_WIDTH<50
#undef MAX_WIDTH
#define MAX_WIDTH 50

#else
#undef MAX_WIDTH
#define MAX_WIDTH 100
#endif

char str[MAX_WIDTH];
```

注意看這一連串的指令 #if, #elif 和 #else 是怎樣以 #endif 結尾的。

■ #line

當我們編譯一段程式的時候，如果有錯誤發生，編譯器會在錯誤前面顯示出錯檔案的名稱以及檔案中的第幾行發生的錯誤。

指令#line 可以使我們對這兩點進行控制，也就是說當出錯時顯示檔案中的行數以及我們希望顯示的檔案名。它的格式是：

#line number "filename"

這裡 number 是將會賦給下一行的新行數。它後面的行數從這一點逐行遞增。

filename 是一個可選參數，用來替換自此行以後出錯時顯示的檔案名，直到有另外一個#line 指令替換它或直到檔的末尾。例如：

```
#line 1 "assigning variable"
int a?;
```

這段代碼將會產生一個錯誤，顯示為在檔案"assigning variable", line 1 。

■ #error

這個指令將中斷編譯過程並返回一個參數中定義的出錯資訊，例如：

```
#ifndef __cplusplus
#error A C++ compiler is required
#endif
```

這個例子中如果 __cplusplus 沒有被定義就會中斷編譯過程。

■ #include

當前置處理器找到一個#include 指令時，它用指定檔的全部內容替換這條語句。聲明包含一個檔有兩種方式：

```
#include "file"
#include <file>
```

兩種表達的唯一區別是編譯器應該在什麼路經下尋找指定的檔。第一種情況下，檔案名被寫在雙引號中，編譯器首先在包含這條指令的檔案所在的目錄下進行尋找，如果找不到指定檔，編譯器再到被配置的預設路徑下（也就是標準標頭檔路徑下）進行尋找。

如果檔案名是在尖括弧 <> 中，編譯器會直接到預設標準標頭檔路徑下尋找。

■ #pragma

這個指令是用來對編譯器進行配置的，針對你所使用的平臺和編譯器而有所不同。要瞭解更多資訊，請參考你的編譯器手冊。

如果你的編譯器不支持某個#pragma 的特定參數，這個參數會被忽略，不會產生

出錯。

■ 預定義的巨集常數名稱 (Predefined macro names)

以下巨集常數名稱在任何時候都是定義好的：

macro	value
__LINE__	整數值，表示當前正在編譯的行在原始檔案中的行數。
__FILE__	字串，表示被編譯的原始檔案的檔案名。
__DATE__	一個格式為 "mm dd yyyy" 的字串，儲存編譯開始的日期。
__TIME__	一個格式為 "hh:mm:ss" 的字串，儲存編譯開始的時間。
__cplusplus	整數值，所有 C++編譯器都定義了這個常量為某個值。如果這個編譯器是完全遵守 C++標準的，它的值應該等於或大於 199711L，具體值取決於它遵守的是哪個版本的標準。

例如：

```
// Predefined macro names
#include <iostream>
using namespace std;

int main()
{
  cout << "This is the line number "
       << __LINE__;
  cout << " of file " << __FILE__
       << ".\n";
  cout << "Its compilation began "
       << __DATE__;
  cout << " at " << __TIME__ << ".\n";
  cout << "The compiler gives a "
       <<"__cplusplus value of "
       << __cplusplus;
  return 0;
}
```

```
This is the line number 7 of file
/home/jay/stdmacronames.cpp.
Its compilation began Nov 1 2005
at 10:12:29.
The compiler gives a __cplusplus
value of 1
```

前置處理器還有更多方便的地方，只要善加利用前置處理器的指令，它能夠簡單的改變或選擇要被編譯的程式碼區塊，當程式碼區塊要依不同狀況分別執行時，前置處理器便利的優點顯而易見。例如我們可以在程式中需要印出 debug 資訊的地方加上：

```
#ifdef DEBUG
cout<<...; // print debug information
#endif
```

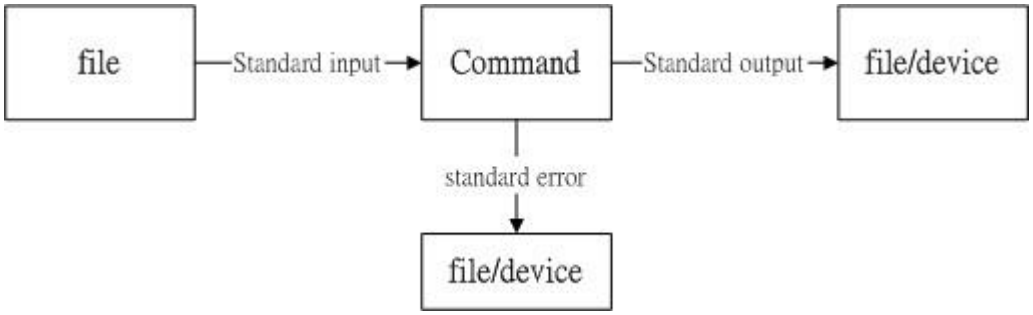
這樣當程式需要印出 debug 資訊時，只要在程式開頭簡單定義：

```
#define DEBUG
```

在程式中所有有加上#ifdef DEBUG 的程式碼就會被編譯。使印 debug 資訊變得很方便使用。

3. Redirection (資料流重導向)

什麼是資料流重導向？這得要由指令的執行結果談起！ 一般來說，如果你要執行一個指令，通常他會是這樣的：



我們執行一個指令的時候，這個指令可能會由檔案讀入，經過處理之後，再將資料輸出到螢幕上。 在上圖中， standard output 與 standard error 分別代表標準輸出與標準錯誤輸出， 這兩個預設輸出設都是輸出到螢幕上面來。

在這樣的過程當中，我們可以將 standard error (簡稱 stderr) 與 standard output (簡稱 stdout) 傳送到其他不同的地方而不是螢幕上頭，傳送的目標通常是檔案或者是裝置， 而傳送的指令則是如下所示：

- 標準輸入(stdin) ：重導向指令 <；
- 標準輸出(stdout)：重導向指令 > 或 >> ；
- 標準錯誤輸出(stderr)：重導向指令 2> 或 2>> ；

舉例來說，如果想要將我目前執行程式的結果所有輸出都記錄到檔案的話，可以簡單的使用>或>>指令將螢幕原本輸出結果重導向至給定的檔案名稱並儲存下來。

而>與>>差異在於要儲存的檔案若存在時，前者會清空檔案內容重新寫入檔案，後者會在檔案最後接著繼續寫下去。

- 使用>：將 stdout 重導向到檔案（command > file）
- 使用>>：將 stdout 資料串加到檔案內容之後（command >> file）

底下是在命令提示字元執行印出"hello world"程式的一個範例，我們可以簡單地利用重導向指令將程式執行完會輸出到螢幕的結果儲存到給定名稱的檔案：

```
// test.cpp
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World!!" <<endl;
  return 0;
}
```

以上是 test.cpp 的檔案內容，經由編譯出的執行檔為 test.exe。

```
C:\Users\Housheng\Desktop> test.exe
Hello World!!
C:\Users\Housheng\Desktop> test.exe >> out.txt
```

在命令提示字元執行 test.exe，會在螢幕輸出 Hello World!!。若將輸出重導向至檔案 out.txt，以下是 out.txt 的內容：

```
Hello World!!
```

非常類似地，2>與 2>>就是將 stderr 訊息重導向到檔案或裝置。

- 使用 2>：將 stderr 重導向到檔案（command 2> file）
- 使用 2>>：將 stderr 資料串加到檔案內容之後（command 2>> file）

那<又是什麼？以最簡單的說法來說， 就是將原本需要由鍵盤輸入的資料，經由檔案來讀入的意思。這個部分比較不常使用，詳細說明可以參閱資料流重導向的相關文獻。

底下列出 Windows 可以使用的重導向指令：

重導向指令	動作
command > *file*	將 *command* 的 stdout 寫入至 *file*
command 1> *file*	將 *command* 的 stdout 寫入至 *file*(跟上述指令一樣)
command 2> *file*	將 *command* 的 stderr 寫入至 *file*
command > *file* 2>&1	將 *command* 的 stdout 及 stderr 寫入至 *file*
command >> *file*	將 *command* 的 stdout 接續著寫入至 *file*
command 1>> *file*	將 *command* 的 stdout 接續著寫入至 *file*(跟上述指令一樣)
command 2>> *file*	將 *command* 的 stderr 接續著寫入至 *file*
command >> *file* 2>&1	將 *command* 的 stdout 及 stderr 接續著寫入至 *file*
commandA | *commandB*	將 *commandA* 的 stdout 重導向至 *commandB* 的 stdin
commandA 2>&1 | *command*	將 *commandA* 的 stdout 及 stderr 重導向至 *commandB* 的 stdin

command < *file*	將 *file* 當成 *command* 的 stdin
command 2>&1	將 *command* 的 stderr 重導向至 stdout
command 1>&2	將 *command* 的 stdout 重導向至 stderr

重導向指令很簡單的將程式的 stdout 輸出至檔案，而不用在 C 或 C++程式碼中開檔寫檔。如果程式輸出為我們要分析的數據，而且我們事先將程式輸出的資料格式撰寫成 MATLAB 輸入點數資料的指令(例如：a[1]=64436; a[2]=46654; ...)，在跑完程式之後只要將重導向所儲存的檔案全部複製貼上至 MATLAB，就可以直接完成大量數據資料的輸入了，接著我們可以很方便利用 MATLAB 對數據作分析或畫圖。

4. Scripts (撰寫腳本)

在 windows 中有.bat 檔在 Linux 有 shell 腳本(.sh 檔)，這兩者都可以將多行指令按一定順序排列而形成一個批次處理的檔案。

底下給一個範例，如果我們想要執行不同參數輸入(共有 100 組)的 test.exe，並將 100 組執行的結果都輸出至檔案 out.txt。以下是 test.bat 的內容(為 100 組不同的參數與執行檔 test.exe 所形成的指令批次檔)：

```
test.exe 100 1000 10
test.exe 100 1000 20
test.exe 100 1000 30
test.exe 100 1000 40
…
```

接著在命令提示字元執行 test.bat，就會依序執行這一百行指令了，再加上重導向指令>>，很簡單就可以將 100 組不同參數的程式輸出結果儲存至檔案 out.txt 了。

```
C:\Users\Housheng\Desktop> test.bat >> out.txt
```

我們很簡單利用腳本的撰寫，讓原本可能要一行一行輸入的指令接續著執行，自動化執行程式在欲執行程式數目很多或執行時間很長的時候是非常有效率的，我們讓電腦自動化去執行並將結果儲存至檔案，我們只需要在所有程式都執行完之後分析輸出的檔案就可以了。

想要更加深入了解腳本或批次檔的撰寫，請參閱 Windows 的.bat 檔與 Linux 的.sh 檔的使用指令相關文獻。

5. Choose Appropriate Data Structures and Operators (選擇適當的資料結構與運算子)

在運算的時候選擇適當資料結構以及運算子是很重要的，以下會從這兩個方面去討論一些例子。

- 選擇適當資料結構：
  在處理二維矩陣時可以用一維矩陣代替，再加上利用指標可以使讀取更快速。底下是隨機產生一個二維矩陣讀取每個數值算總平均的例子：

```
// average_2dim_matrix.cpp
#include <iostream>
using namespace std;
#define ROW 10
#define COL 10
#define RAND_MAX 100
int main()
{
  int matrix[M][N];
  int row,col;
  //write matrix with random value
  for(row=0; row<ROW; row++) {
    for(col=0; col<COL; col++) {
      matrix[row][col]=rand()%RAND_MAX;
    }
  }
  //calculate the matrix average value
  int average=0;
  for(row=0; row<ROW; row++) {
    for(col=0; col<COL; col++) {
      average+=matrix[row][col];
    }
  }
  cout<<"average="<<average/(ROW*COL)<<endl;
  return 0;
}
```

上面程式碼把寫入數值與運算分開寫的原因是，寫入矩陣數值的動作可以改成由外部讀取檔案進來的(例如：處理影像檔時，可以把第一段改成讀檔將圖檔數值存進二維矩陣)，請注意實際上產生與平均兩段程式一起寫在同一個迴圈會更有效率，但是這裡為了方便觀念釐清使用分開寫的方式。

底下的程式將上述二維矩陣改存成一維處理，並加上適當的指標處理正確的index：

```
// average_1dim_matrix.cpp
#include <iostream>
using namespace std;
#define ROW 10
#define COL 10
#define ALL ROW*COL
#define RAND_MAX 100
int main()
{
  int matrix[ALL];
  int row, col;
  int *pMatrix;
  //write matrix with random value
  pMatrix=matrix;
  for(row=0; row<ROW; row++) {
    for(col=0; col<COL; col++) {
      *pMatrix=rand()%RAND_MAX;
      pMatrix++;
    }
  }
  //calculate the matrix average value
  int average=0;
  pMatrix=matrix;
  for(row=0; row<ROW; row++) {
    for(col=0; col<COL; col++) {
      average+=*pMatrix;
      pMatrix++;
    }
  }
  cout<<”average=”<<average/ALL<<endl;
  return 0;
}
```

這種寫法會比較快，因為二維的矩陣在記憶體位置運算會有乘法出現(儲存matrix[row][col]實際上會寫入記憶體 matrix+row*COL+col 的位置)，而一維的寫法只用加法就可以計算出正確的記憶體位置。因為我們知道所有矩陣元素個數，而且每個元素都匯處理到，所以底下可以再更進一步改寫上述程式將迴圈簡化：

```
// average_1dim_matrix_1loop.cpp
#include <iostream>
using namespace std;
#define ROW 10
#define COL 10
#define ALL ROW*COL
#define RAND_MAX 100
int main()
{
  int matrix[ALL];
  int index;
  int *pMatrix;
  //write matrix with random value
  pMatrix=matrix;
  for(index=0; index<ALL; index++) {
    *pMatrix=rand()%RAND_MAX;
    pMatrix++;
  }
  //calculate the matrix average value
  int average=0;
  pMatrix=matrix;
  for(index=0; index<ALL; index++) {
    average+=*pMatrix;
    pMatrix++;
  }
  cout<<"average="<<average/ALL<<endl;
  return 0;
}
```

一個迴圈就把原本兩個迴圈才可處理的程式完成了，這種寫法會更加有效率。

- 選擇適當運算子：
  在運算子的選擇上面有一些先後順序：
  - 盡量用加法替代乘法。
  - 盡量用 bit shift (<<或>>)替代 2 的冪次方之乘除法。
  - 盡量不要使用除法或取餘數(%)。

  簡單來說，利用越快的運算(加法或 bit shift)來替代較慢的運算(乘法或除法)會使程式執行更加有效率。

- 選擇適當控制結構：
  在控制的時候盡量使用巢狀判斷結構來取代平行化的判斷結構，前者可以預先把一些不符合判斷條件的狀況排除，故會比較有效率。

另外，在使用判斷的時候，盡量將可以篩選掉大部分資料(較嚴苛)的條件放在越外層，這樣可以是判斷所需次數減少。

底下分別是巢狀以及平行化的判斷結構：

◆ 巢狀判斷結構

```
if (條件式一)
    陳述一;
else if(條件式二)
    陳述句二;
else if(條件式三)
    陳述句三;
else
    陳述句四;
```

```
if (條件式一)
{
    陳述一;
}
else
{
    if(條件式二)
    {
        陳述句二;
    }
    else
    {
        if(條件式三)
        {
            陳述句三;
        }
        else
        {
            陳述句四;
        }
    }
}
```

◆ 平行化判斷結構

```
if (條件式一)
{
    陳述一;
}
if(條件式二)
{
    陳述句二;
}
if(條件式三)
{
    陳述句三;
}
if(條件式四)
{
    陳述句四;
}
```