Chapter 10

運算子覆載

運算子覆載(一)

- C++ 將數學運算式看成函式處理

```
3 + 5   <=>   operator+( 3 , 5 )
```

數學運算式的

- 運算子之前加上 operator => 函式名稱
- 運算元 => 函式的輸入參數
- 計算結果 => 函式的回傳值

運算子覆載(二)

■ 運算子覆載

C++ 並沒有可以處理非預設型別的加法運算函式，使用者須自行定義。如此同樣的 '+' 法符號，卻因運算元型別不同，有著不一樣的意義

若 Fraction 為分數類別

Fraction(a,b) 代表 $\frac{a}{b}$

則

$\frac{3}{2}$ + $\frac{5}{4}$ <=> operator+(Fraction(3,2),Fraction(5,4))

operator overloading

分數類別(一)

- 分數類別

```
class  Fraction {
   private :
     int num ;  // 分子
     int den ;  // 分母
   public :
     int get_num() const { return num } ;
     int get_den() const { return den } ;
};
```

❖ 程式將限制分母大於 0 ，分數的正負由分子控制

分數類別(二)

- 建構函式

```
Fraction::Fraction(){}

Fraction::Fraction( int n ,int d=1 ): num(n), den(d) {
   assert( d != 0 ) ;
   if( d < 0 ) { num *= -1 ; den *= -1 ; }
}

Fraction::Fraction( int a , int n , int d ) : den(d) {
   assert( d > 0 && n >= 0 ) ;
   num = ( a > 0 ? a * d + n : a * d – n ) ;
}
```

❖ 在 **cassert** 標頭檔內的 **assert** 函式是用來確認其內的邏輯式子執行是否為真，若為假則程式將中斷，並印出此邏輯式子

分數類別(三)

- 使用方式

```
Fraction  a ;              // 一個沒有初值的分數
Fraction  b(4) ;           // 一分之四
Fraction  c(3,5) ;         // 五分之三
Fraction  d(2,-3) ;        // 負三分之二
Fraction  e(-2,3,5) ;      // 負二又五分之三 (帶分數)
Fraction  f(-2,-3,5) ;     // 錯誤的設定
```

分數類別(四)

- 最大公因數

$$gcd(a, b) = \begin{cases} b & a \% b = 0 \\ gcd(b, a \% b) & a \% b \neq 0 \end{cases}$$

假設 a > b > 0

- 遞迴函式

```
int  Fraction::gcd( int a , int b ) const {
   if( a >= b )
      return  a%b == 0 ? b : gcd( b , a%b ) ;
   else
      return  b%a == 0 ? a : gcd( a , b%a ) ;
}
```

分數：加法運算子

- 分數的加法運算

```
Fraction  a(2,5) , b(-3,4) , c ;
c = a + b ;
```

- 加法運算子

```
Fraction operator+(const Fraction& a , const Fraction& b ) {
   int den = a.get_den() * b.get_den() ;
   int num = a.get_num() * b.get_den() +
             a.get_den() * b.get_num() ;
   Fraction sum(num,den) ;
   return sum ;
}
```

} `return Fraction(num,den);`

❖ 以上的末兩行可直接改寫成右側式子藉以省去一個暫時物件名稱

覆載運算子注意事項

- C++ 已有的運算子並不允許重新設計，以免造成天下大亂
- C++ 無法讓程式設計員自行設定新的運算子符號
- 運算子覆載不會改變運算子原有的運算優先順序，C++ 中所設定的運算符號優先順序仍然保持不變

```
3^5   <=>   3^5   <=>   operator^(3,5)

2+3^5   <=>    2 + 3^5
        <=>  operator^( operator+(2,3) , 5 )
        <=>  (2+3)^5
        <=>  (2+3)^5
```

+ 較 ^ 優先處理

運算元的自動型別轉換

- 許多分數加法運算子

```
Fraction  onehalf(1,2) ;
Fraction  foo , bar ;
foo = 3 + onehalf ;        // 整數 加 分數
bar = onehalf + 3 ;        // 分數 加 整數
```

- C++使用型別自動轉換機制來簡化運算子覆載程式碼，使用者可避免為同一個運算子設計許多功能相同的運算子函式

implicit type conversion

型別自動轉換(一)

- 利用型別所屬類別的建構函式將其它資料型別轉換過來

```
3 + onehalf     =>    Fraction(3) + onehalf
```

```
Fraction  onehalf(1,2) ;
Fraction  foo , bar ;

// foo = 3 + onehalf
foo = Fraction(3) + onehalf ;   // 自動轉成 分數 加 分數

// bar = onehalf + 3
bar = onehalf + Fraction(3) ;   // 自動轉成 分數 加 分數
```

型別自動轉換(二)

- 兩個整數相加後得到一個分數

```
Fraction  a , b , c ;

// 整數 加 分數 => 分數
a = 3 + Fraction(4) ;

// 整數 加 整數 => 整數，再利用建構函式轉成分數
b = Fraction( 3 + 4 ) ;

// 整數 加 整數 => 整數，再利用自動型別轉換機制
c = 3 + 4 ;
```

型別自動轉換(三)

- 若要禁止類別的建構函式被用來當成自動型別轉換，則可在函式名稱前加上 **explicit**

```
class  Fraction {
   public :
      explicit  Fraction( int n , int d = 1 ) {...}
};
```

如此

```
Fraction  foo ;
foo = 3 ;                  // 錯誤
foo = Fraction(3) ;        // 正確
```

型別自動轉換(四)

- 自動型別轉換不適用於非常數型參考物件

```
Fraction  operator+( Fraction& a ,
                     const Fraction& b ) ;
```

```
Fraction  foo ;
foo = 3 + Fraction(4) ;  // 錯誤，整數 3 不會被轉成分數
```

❖ 若函式參數在函式內並不會被更改，須盡量以常數型參考傳入函式

覆載模式(一)：全域函式

- 全域函式覆載模式

```
Fraction  operator+( const Fraction& a,
                      const Fraction& b ) {
    int d = a.get_den() * b.get_den() ;
    int n = a.get_num() * b.get_den() +
            a.get_den() * b.get_num() +
    return  Fraction( n , d ) ;
}

Fraction  a(2,3) , b(3,4) , c ;
```

```
c = a + b ;     <=>     c = operator+(a,b) ;
```

覆載模式(二)：成員函式

- 成員函式覆載模式

```
Fraction  Fraction::operator-( const Fraction& b )
                                const {
    int d = den * b.den ;
    int n = num * b.den - den * b.num ;
    return  Fraction( n , d ) ;
}
```

```
Fraction  a(2,3) , b(3,4) , c ;
```

```
c = a - b ;     <=>     c = a.operator-(b) ;
```

覆載模式(三)：比較

- 運算子覆載僅能在全域函式模式與成員函式模式擇一撰寫
- 全域函式內無法直接使用類別私有資料成員，成員函式則可
- 以成員函式撰寫的運算子，其第一個運算元一定要為類別物件本身
- 運算子 + - * / 等，若以全域函式撰寫較為便利

```
Fraction a(2,3) , b(3,4) , c , d ;
c = 2 + a ;   // 相當於 c = operator+(2,a)
c = a + 2 ;   // 相當於 c = operator+(a,2)
d = 3 – a ;   // 錯誤，無法執行 d = 3.operator-(a)
d = a – 3 ;   // 相當於 d = a.operator-(3)
```

運算子函式的運算程序

- 若加法為全域函式模式，乘法為成員函式模式

```
    a = b = c + d ;
<=> a = b = operator+(c,d) ;
<=> a = b.operator=( operator+(c,d) ) ;
<=> a.operator=( b.operator=( operator+(c,d) ) ) ;
```

```
    a = b * c + d ;
<=> a = b.operator*(c) + d ;
<=> a = operator+( b.operator*(c) , d ) ;
<=> a.operator=( operator+( b.operator*(c) , d ) ) ;
```

❖ 指定運算子為類別內的成員函式為右向結合運算子，
加減乘除運算子則為左向結合運算子

運算子的回傳型別(一)

- 運算子的回傳型別可以自行設定

```
// (1) 沒有回傳值
void  Fraction::operator=( const Fraction& ) ;

// (2) 回傳物件
Fraction  Fraction::operator=( const Fraction& ) ;

// (3) 回傳物件參考
Fraction&  Fraction::operator=( const Fraction& ) ;
```

- ❖ 函式的回傳型別不被視為特徵資料的一部份，因此為避免函式重複定義，以上三者僅能擇一設定

運算子的回傳型別(二)

- 三種指定運算子的使用差異

```
Fraction  a(2,3) , b(2,5) , c(3,4) ;
a = b ;              // 三種方式皆可
a = b = c ;          // 回傳物件 與 回傳物件參考 皆可
( a = b ) = c ;// 回傳物件 與 回傳物件參考 皆可，但答案不同
```

- 以上可改寫成

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

運算子的回傳型別(三)

- 綜合比較

```
a = 2/3 , b = 2/5 , c = 3/4
```

運算式	回傳 void	回傳 Fraction	回傳 Fraction&
a = b	a = b = 2/5 c = 3/4	a = b = 2/5 c = 3/4	a = b = 2/5 c = 3/4
a = b = c	錯誤	a = b = c = 3/4	a = b = c = 3/4
(a = b)= c	錯誤	a = b = 2/5 c = 3/4	a = c = 3/4 b = 2/5

❖ 覆載運算子時儘量不要改變運算子原有計算特性

自我指定的物件指標：this(一)

- 每個類別物件都有一個預設的指標指向物件本身，此指標稱為 **this**
- **this** 指標僅能在類別的成員函式內使用

- 成員函式內也可透過 **this** 取得物件內的資料成員

```
Fraction Fraction::operator-( const Fraction& a ) const {
   int d = this->den * a.den ;
   int n = this->num * a.den – this->den * a.num ;
   return  Fraction(n,d) ;
}
```

❖ 以上的方式有點大費周章，一般是不需如此使用

自我指定的物件指標：this(二)

- 分數類別的指定運算子：

```
Fraction&  Fraction::operator=(
           const Fraction& foo ) const {
    den = foo.den ;
    num = foo.num ;
    return *this ;
}
```

❖ this 為一永遠指向物件本身的指標，因此 *this 代表物件本身

自我指定的物件指標：this(三)

- 自我指定的指定運算子

```
Fraction  a(2,5) ;
a = a ;          // 物件自我指定
```

多此一舉，造成時間浪費，且可能造成錯誤
可將指定運算子改成以下方式

```
Fraction& Fraction::operator=(const Fraction& foo) {
   if ( this == &foo ) return *this ;
   den = foo.den ;  num = foo.num ;
   return *this ;
}
```

❖ 當 this 所指定的位址與輸入的參數物件位址相同時，則可以確認兩物件為同物件，不需再執行指定動作

單元運算子與雙元運算子

- 單元運算子：僅須要一個運算元的運算子

  -(負號) ++(遞增) --(遞減) !(否定) ...

- 雙元運算子：須要兩個運算元的運算子

  +(加) -(減) *(乘) =(指定) <(小於) ....

unary , binary operator

負號運算子

- 負號運算子：僅需一個運算元的 **operator-** 運算子
- 分數的負號運算子

成員函式模式

```
Fraction Fraction::operator-() const {
   return Fraction( -num , den ) ;
}
```

全域函式模式

```
Fraction operator-( const Fraction& a ) {
   return Fraction( -a.get_num(), a.get_den() );
}
```

各種指定運算子

- 先運算後指定：+= ， -= ， *= ， /= ， %=

operator+=	先加後指定
operator-=	先減後指定
operator*=	先乘後指定
operator/=	先除後指定
operator%=	先取餘數後指定

先加後指定運算子

- 成員函式模式

```
Fraction&  Fraction::operator+=( const Fraction& foo ) {
    num = num * foo.den + den * foo.num ;
    den = den * foo.den ;
    return *this ;
}
```

- 全域函式模式

```
Fraction& operator+=( Fraction& a , const Fraction& b ) {
    a = a + b ;     // 利用已有的 operator= 與 operator+
    return a ;
}
```

❖ 成員函式模式的運算子左側運算元一定須為分數型別物件
全域函式模式的第一個參數須以非常數式參考傳入函式內

無回傳的指定運算子

- 各種指定運算子也可選擇不回傳

```
void  Fraction::operator+=( const Fraction& foo ) {
    num = num * foo.den + den * foo.num ;
    den = den * foo.den ;
}
```

如此，則

```
Fraction  a(2,3) , b(2,5) , c ;
c = ( a += b ) ;            // 錯誤
```

無回傳

❖ 以上 c 因右側沒有任何指定資料而發生錯誤

比較運算子(一)

- 等於運算子：**operator==**

```
bool  operator==( const Fraction& a , const Fraction& b ) {
    return ( a.get_num() * b.get_den() ==
             a.get_den() * b.get_num() ) ;
}
```

- 小於運算子：**operator<**

```
bool  operator<( const Fraction& a , const Fraction& b ) {
    return ( a.get_num() * b.get_den() <
             a.get_den() * b.get_num() ) ;
}
```

❖ 以上的運算子也可以設計為成員函式模式，但如此一來運算子左邊的運算元一定須為分數類別物件

比較運算子(二)

- 不等於運算子：**operator!=**

```
bool operator!=( const Fraction& a , const Fraction& b ) {
    return !( a == b ) ;
}
```

- 小於等於運算子：**operator<=**

```
bool operator<=( const Fraction& a , const Fraction& b ) {
    return ( a == b || a < b ) ;
}
```

- 大於運算子：**operator>**

```
bool operator>( const Fraction& a , const Fraction& b ) {
    return !( a == b || a < b ) ;
}
```

- 大於等於運算子：**operator>=**

```
bool operator>=( const Fraction& a , const Fraction& b ) {
    return !( a < b ) ;
}
```

其他邏輯運算子

- **operator!**：非

```
bool operator!( const Fraction& a ) ;
```

- **operator||**：或者

```
bool operator||( const Fraction& a,
                 const Fraction& b ) ;
```

- **operator&&**：而且

```
bool operator&&( const Fraction& a,
                 const Fraction& b ) ;
```

❖ 以上三者邏輯運算子最好不要輕易自行設計，以免得不償失

輸出運算子(一)

- 輸出運算子的基本形式

```
ostream&  operator<< ( ostream& out , const FOO& foo ) {
    ....
    return out ;
}
```

- **ostream** 類別定義在 **iostream** 標頭檔內，專門負責資料的輸出，**cout** 則為 **C++** 預設的 **ostream** 類別物件

- 當輸出運算式被執行時，欲輸出的資料會存入 **cout** 物件

- 輸出運算子最後一定要回傳輸入的 **ostream** 物件參數，也就是 **cout** 物件

輸出運算子(二)

- 輸出運算子內定為全域函式模式

輸出整數

```
cout << 12 ;   <=>   operator<<( cout , 12 ) ;
```

輸出整數 **12** 之後再輸出分數 **1/2**

```
cout << 12 << Fraction(1,2) ;
<=>  operator<<(operator<<(cout,12),Fraction(1,2));
```

回傳 **cout**

- 上式共使用了兩種不同型別的輸出運算子，其一為C++所預設針對整數的輸出運算子，另一為使用者自定針對分數的輸出運算子

分數類別輸出運算子(一)

- 分數類別的輸出運算子

```
ostream& operator<<( ostream& out , const Fraction& a ) {
    out << a.get_num() << "/" << a.get_den() ;
    return  out ;
}
```

- 以上的輸出運算子並非分數類別的成員函式，因此在輸出運算子內須透過分數類別的公共介面函式間接地取得分數物件的私有資料成員

- 此輸出運算子共使用了三個預設型別的輸出運算子，分別用來輸出分子(整數)、斜線(傳統字串)與分母(整數)

分數類別輸出運算子(二)

- 輸出運算子經常須取用物件的私有資料成員，因此在使用上常將之定義為輸出類別的夥伴函式，便於取得物件私有資料

```
class  Fraction {
  public:
    friend ostream& operator<<( ostream& , const Fraction& ) ;
} ;

ostream& operator<<( ostream& out , const Fraction& foo ) {
   int  num = ( foo.num > 0 ? foo.num : -foo.num ) ;
   int  den = foo.den ;
   int  divisor = ( num == 0 ? 1 : foo.gcd(num,den) ) ;
   return  out << foo.num/divisor << "/" << foo.den/divisor ;
}
```

約分　　約分

❖ 以上利用到分數類別內私有成員函式 **gcd** 以計算分數的最簡形式

輸入運算子(一)

- 輸入運算子的基本形式

```
istream&  operator>> ( istream& in , FOO& foo ) {
    ....
    return in ;
}
```

- 類別 **istream** 為定義在 **iostream** 標頭檔內之專門負責資料輸入的類別，**cin** 為 **C++** 預設之 **istream** 類別物件
- 當輸入運算式被執行時，資料會由 **cin** 物件取出存入運算子右側的運算元，因此運算元不能設定為常數型式
- 輸入運算子回傳輸入的 **istream** 物件參數，也就是 **cin** 物件

輸入運算子(二)

- 輸入運算子為全域函式

輸入整數

```
int a ;
cin >> a ;          // operator>> ( cin , a )
```

輸入整數再輸入分數

```
int a ;
Fraction b ;
cin >> a >> b ;  // operator>>( operator>>(cin,a), b )
```

回傳 cin

- 上式共使用了兩個不同的輸入運算子，其一為C++所預設針對整數的輸入運算子，另一為使用者自定針對分數的輸入運算子

分數類別輸入運算子(一)

- 分數類別的輸入運算子

```
class  Fraction {
   public:
      friend istream& operator>>( istream& , Fraction& ) ;
} ;

istream& operator>>( istream& in , Fraction& foo ) {
    char slash ;
    in >> foo.num >> slash >> foo.den ;
    return in ;
}
```

- 為方便將讀入的資料直接存入分數私有資料成員內，通常會將輸入運算子定義為類別的夥伴函式

分數類別輸入運算子(二)

- 分數輸入運算子

```
istream&  operator>>( istream& in , Fraction& foo ) {
    char slash ;
    in >> foo.num >> slash >> foo.den ;
    return  in ;
}
```

- 分數輸入運算子總共使用了三個預設型別的輸入運算子，分別用來輸入分子(整數)、斜線(字元)與分母(整數)，這裡的斜線讀入後並沒有任何作用
- 分數可以接受以下輸入型式

`-2/5 , -2 /5 , -2 / 5` 甚至 `-2 * 5`

遞增與遞減運算子

- 前置遞增/遞減：先執行遞增/遞減

```
int i , j , s = 3 , t = 3 ;
i = ++s + 1 ;    // 先讓 s = 4 , 再令 i = 5
j = --t + 1 ;    // 先讓 t = 2 , 再令 j = 3
```

- 後置遞增/遞減：後執行遞增/遞減

```
int i , j , s = 3 , t = 3 ;
i = (s++) + 1 ;     // 先讓 i = 4 , 再令 s = 4
j = (t--) + 1 ;     // 先讓 j = 4 , 再令 t = 2
```

分數遞增 / 遞減運算子(一)

■ 成員函式模式

前置遞增

```
Fraction&  Fraction::operator++ () {
    num += den ;
    return *this ;
}
```

後置遞增

```
Fraction&  Fraction::operator++ ( int a ) {
    Fraction bar(*this) ;
    num += den ;
    return bar ;
}
```

❖ 後置遞增/遞減運算子的參數列須多一個無用的整數參數

分數遞增 / 遞減運算子(二)

- 全域函式模式

```
Fraction&  operator++ ( Fraction& foo ) {
   foo += 1   ;
   return foo ;
}
```

前置遞增

```
Fraction& operator++ ( Fraction& foo , int a ) {
   Fraction bar(foo) ;              // bar 為暫時物件
   foo += 1 ;
   return bar ;
}
```

後置遞增

- 後置運算子也可利用前置運算子來完成

```
Fraction& operator++ ( Fraction& foo , int a ) {
    Fraction old = foo ;            // old 為暫時物件
    ++foo ;
    return old ;
}
```

分數遞增 / 遞減運算子(三)

- 基於執行效率考量，若運算式中單純地使用遞增/遞減運算子時，以前置形式為較佳的選擇，如此可避免產生暫時物件

```
// 效率佳
for ( Fraction bar(1,2) ; bar < 10 ; ++bar ){
    cout << bar << endl ;
}

// 效率差
for ( Fraction bar(1,2) ; bar < 10 ; bar++ ){
    cout << bar << endl ;
}
```

強制型別轉換(一)

- 若要將分數明確地轉型為整數或浮點數

```
Fraction  foo(3,2) ;
int    a = static_cast<int>(foo) ;     // a = 1
double b = static_cast<double>(foo) ;  // b = 1.5
```

  - 可以在分數類別內定義兩個轉型成員函式如下

```
Fraction::operator int() const {
   return num/den ;
}
```

```
Fraction::operator double() const {
   return 1. * num/den ;
}
```

❖ 此兩種轉型函式都沒有回傳型別

強制型別轉換(二)

- 強制型別轉換的困境

```
Fraction  foo(2,3) ;                  // foo 為 2/3
cout << 3 + foo << endl ;             // 無從判斷，整數 ? 分數 ?
```

- 以上的 **3 + foo** 可以被認為以下兩種之一
  - **3 + static_cast<int>(foo)**
  - **Fraction(3) + foo** // 使用自動型別轉換

- 可在分數類別的建構函式前加上 **explicit** 字樣藉以明確地禁止此建構函式被用來自動型別轉換

分數類別程式碼

- 所有的 `+=` `-=` `*=` `...` 等運算子都被定義成類別內的成員函式
- 所有的 `+` `-` `*` `/` `...` 等運算子都被定義成全域函式
- 將之前所有的分數函式加以整理，分數類別程式碼可以寫成

程式 輸出

向量類別(一)

- 向量類別主要功能：

  可在程式執行中，依需要自動調整儲存資料陣列的長度

- 資料儲存機制：

  動態配置空間

- 類別設計重點：

  複製建構函式，指定運算子，解構函式須自行設定

向量類別(二)

- 基本向量類別

```
class  Vector {
  private:
    int *data ;     // 向量元素
    int size  ;     // 向量元素個數
  public:
    Vector() : data(NULL) , size(0) {}  ;  // 備用建構函式
    Vector( unsigned int s , int v = 0) ;  // 建構函式
    Vector( const Vector& ) ;              // 複製建構函式
    Vector& operator=( const Vector& ) ;   // 指定運算子
    ~Vector() ;                            // 解構函式
} ;
```

❖ NULL 為系統預設的整數常數其值為 0

向量類別：建構、解構函式

- 備用建構函式

```
Vector::Vector() : data(NULL) , size(0) {}
```

- 一般建構函式

```
// 向量長度為 s 每個元素預設值為 v
Vector::Vector(unsigned int s, int v = 0) : size(s) {
    data = new int[size] ;
    for ( int i = 0 ; i < s ; ++i ) data[i] = v ;
}
```

- 解構函式

```
Vector::~Vector() { if( size > 0 ) delete [] data ; }
```

向量類別：複製建構函式

- 複製建構函式

```
Vector::Vector( const Vector& foo ) : size(foo.size) {
    data = ( size > 0 ? new int[size] : NULL ) ;
    for ( int i = 0 ; i < size ; ++i )
       data[i] = foo.data[i] ;
}
```

❖ 所有建構函式都不需回傳任何資料

向量類別：指定運算子

- 指定運算子

```
Vector&  Vector::operator= ( const Vector& foo ) {
    if ( this == &foo ) return *this ;
    if ( foo.size != size ) {
        if ( size ) delete [] data  ;
        size = foo.size ;
        data = ( size ? new int[size] : NULL ) ;
    }
    for ( int i = 0 ; i < size ; ++i )
        data[i] = foo.data[i] ;
    return *this ;
}
```

- ❖ 為避免物件自我指定，程式須先檢查輸入的物件是否為原有的物件
- ❖ 若指定物件與被指定物件的資料長度相等，則可直接使用原有的動態空間，以增加程式執行效率

向量類別：輸出/輸入運算子

- 向量類別的輸出/輸入運算子可定義為

輸出運算子

```
ostream& operator<< ( ostream& out , const Vector& a ){
   out << '(' << ' ' ;
   for ( int i = 0 ; i < a.size ; ++i )
      out << a.data[i] << ' ' ;
   return out << ')' ;
}
```

輸入運算子

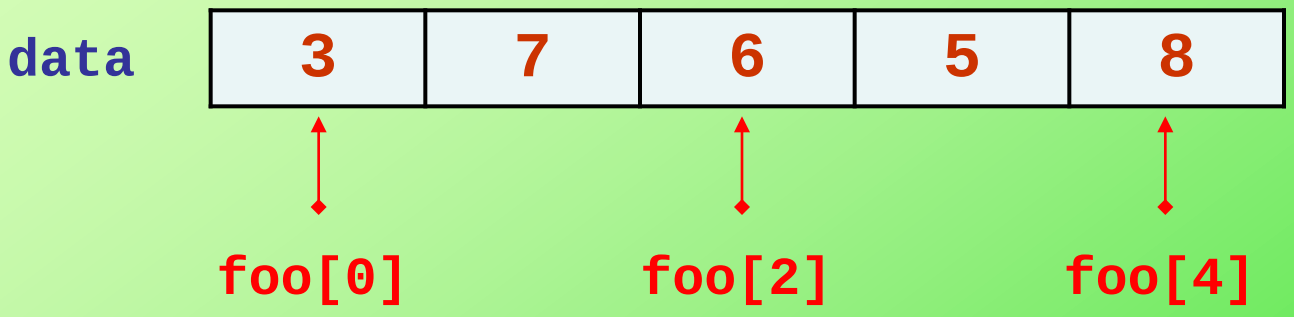
```
istream& operator>> ( istream& in , Vector& a ) {
   for ( int i = 0 ; i < a.size ; ++i ) in >> a.data[i] ;
   return in ;
}
```

❖ 以上的輸出與輸入運算子要設定為向量類別的夥伴函式

向量類別：下標運算子(一)

- 下標運算子：**operator[]**

  可直接用來存取向量物件內 **data** 資料成員的某分量



- 兩種形式：非常數與常數成員函式

```
int&       Vector::operator[](int i)       {return data[i];}
const int& Vector::operator[](int i) const {return data[i];}
```

向量類別：下標運算子(二)

■ 常數下標運算子與非常數下標運算子的使用

```
const Vector a(10,5) ;   // 常數向量，10個分量，初值為 5
Vector  b(10) ;          // 非常數向量，10個分量，初值為 0
int  i ;

// 使用常數下標運算子
for ( i = 0 ; i < 10 ; ++i ) cout << a[i] << endl ;

// 使用非常數下標運算子
for ( i = 0 ; i < 10 ; ++i ) cout << b[i] << endl ;
```

❖ C++會根據物件是否為常數來選擇相對類型的下標運算子

向量類別：加法、減法運算子

- 兩向量相加：成員函式

```
Vector Vector::operator+( const Vector& foo ) const {
    Vector  bar(size) ;
    for ( int i = 0 ; i < size ; ++i )
        bar[i] = data[i] + foo.data[i] ;
    return  bar ;
}
```

❖ 減法運算子的型式類似

向量類別：內積

- 內積：計算兩等長向量的內積

```
int Vector::operator*( const Vector& foo ) const {
    int sum = 0 ;
    for ( int i = 0 ; i < size ; ++i )
        sum += data[i] * foo.data[i] ;
    return sum ;
}
```

- 也可利用下標運算子取得物件資料分量

```
int Vector::operator*( const Vector& foo ) const {
    int sum = 0 ;
    for ( int i = 0 ; i < size ; ++i )
        sum += data[i] * foo[i] ;
    return sum ;
}
```

向量物件陣列與下標運算子

- 向量物件

```
Vector a(3,1) ;     // 3 個 1
```

- 向量物件陣列

```
Vector a(3,1) , b(2,3) ;  // a : 3 個 1 , b : 2 個 3
Vector c[2]               ;  // 兩個向量物件
c[0] = a , c[1] = b       ;
```

- 向量下標運算子

```
c[0][1] = 5 ;  // 讓向量物件陣列的第1個向量的
               // 第 2 筆資料設定為 5
```

相當於 `c[0].operator[](1) = 5`

陣列下標　物件下標

向量類別程式碼

- 將之前所有相關的函式集結在一起，可得程式碼如下