Chapter 9

類別初步

類別初步

- 類別：為一種可由使用者自行設計的資料型別

  資料成員 (data member)：用來儲存資料

  成員函式 (member function) ：用來操縱資料成員的函式

- 物件：類別變數

  - 物件導向程式設計
    - 以類別設計為主軸的程式設計

      eg. Small Talk , C++ , Java

  - 函式導向程式設計
    - 以函式設計為主軸的程式設計

      eg. Fortran 77 , basic , pascal , lisp

class, object

集合範例

- 集合：
  - 由多個同類型的資料所構成，這些資料稱為元素 (element)
  - 元素間沒有次序等級的區別
  - 元素可以重複(或不重複)出現
- 集合功能：
  - 將元素加入集合內
  - 檢查元素是否在集合內
  - 去除集合內某元素
  - 計算集合內元素總數
  - 計算兩集合間的交集，聯集，差集等
  - 檢查兩集合是否相等

集合結構（一）

- 集合須儲存多個同類型的元素：使用陣列
- 元素可以儲存在任何陣列位置：
  - 使用輔助陣列紀錄元素陣列的使用狀態

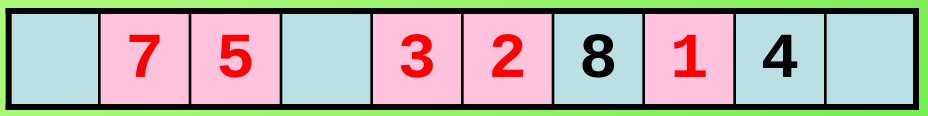
```
// 定義集合最多能儲存多少元素
const int SIZE = 10;

// 設定集合元素陣列的儲存狀態
enum   Status   { empty , occupied };
struct Set {
   int      element[SIZE]  ;    // 元素陣列
   Status   status[SIZE]   ;    // 輔助陣列
   int      element_no     ;    // 元素個數
};
```

集合結構（二）

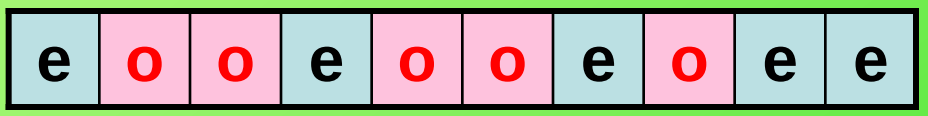
- 當狀態為 **empty** 表示對應的陣列位置並無元素
  當狀態為 **occupied** 表示對應的陣列位置存有元素

元素陣列：



element[SIZE]

狀態陣列：



status[SIZE]

元素個數：**5**（即狀態陣列中 **o** 的個數）

element_no

❖ 以上集合雖存有 **7** 個數值，但只有 **5** 個為集合內元素

設定集合初值

- 在使用集合結構資料型別之前，須先將集合內所有資料設定為適當的初值

```
// 設定集合的初始資料
void set_initialization( Set& foo ){
    int i ;
    for(i=0 ; i<SIZE ; ++i) foo.status[i] = empty ;
    foo.element_no = 0;
}

...

Set  foo ;                          // 定義 foo 物件
set_initialization(foo) ;           // 設定 foo 的初始資料
```

加入集合元素

- 加入元素至集合內

```
bool insert_element( Set& s , int data ){
    //若集合已滿了，則禁止加入任何元素
    if( s.element_no == SIZE ) return false ;
    for( int i = 0 ; i < SIZE ; ++i ){
        if( s.status[i] == empty ){
            s.element[i] = data ;
            s.status[i]  = occupied ;
            ++s.element_no ;
            return true ;
        }
    }
}
```

❖ 函式回傳元素是否被加入

刪除集合元素

- 刪除集合內的元素

```
int delete_element( Set& s , int data ){
    int no_elements_deleted = 0 ;  // 元素被去除的個數
    for( int i = 0 ; i < SIZE ; ++i ){
        if( s.status[i] == occupied &&
            s.element[i] == data ) {
            s.status[i] = empty ;
            --s.element_no ;
            ++no_elements_deleted ;
        }
    }
    return no_elements_deleted ;
}
```

❖ 函式回傳元素刪除的個數

列印集合與集合程式碼

- 撰寫輸出函式來驗證集合元素的正確性

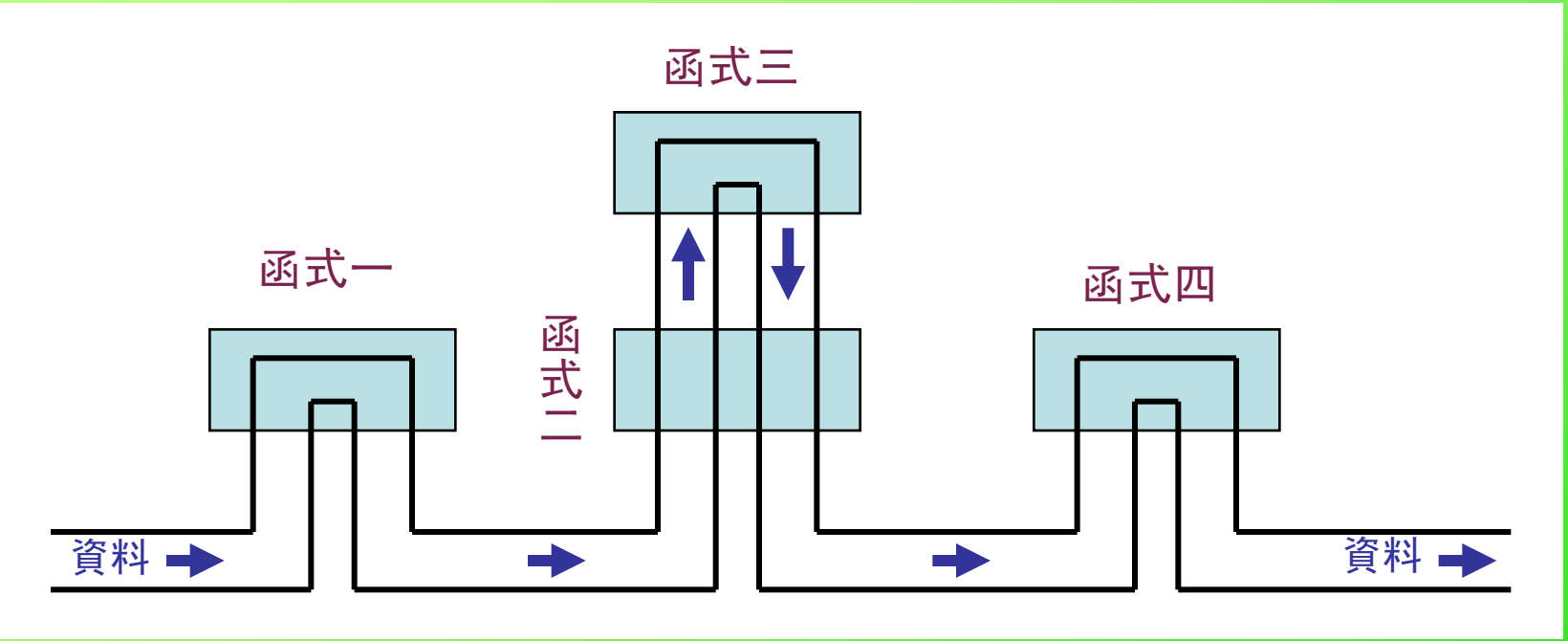
```
// 列印集合
void print_set( const Set& foo ){
   cout << "集合有 " << foo.element_no << "元素 :  " ;
   for( int i = 0 ; i < SIZE ; ++i ){
       if( foo.status[i] == occupied )
           cout << foo.element[i] << ' ';
   }
   cout << endl;
}
```

❖ 僅列印元素狀態為 **occupied** 的元素

程式
輸出
集合程式

函式導向程式設計

- 函式導向程式設計



程式碼的設計主軸就是將變數一次又一次地送入函式處理，當函式執行結束後，變數也隨之更改

procedure-oriented programming

集合函式導向程式設計的缺點

- 不熟悉的使用者可以隨時更改集合資料，使得資料違反設計規則

```
Set foo                          ;  //  定義集合變數
set_initialization(foo)          ;  //  設定集合初值
insert_element(foo,5)            ;  //  將 5 加入集合內
foo.status[3] = occupied ;          //  直接更改元素狀態
print_set(foo)                   ;  //  列印集合
```

❖ 以上的程式語法依然正確，但結果不對

集合類別

- 之前的缺失可以透過類別
  - 將集合內的元素資料與操作的函式看成一體
  - 規範資料的存取權限，藉以避免資料不當的被修改

```
const int SIZE = 10 ;
enum   Status   { empty , occupied } ;
class Set {
   private:
     int      element[SIZE] , element_no ;
     Status   status[SIZE] ;
   public:
     void     initialization()  ;
     bool     insert_element( int data ) ;
     int      delete_element( int data ) ;
     void     print_set();
} ;
```

私有區域

公共區域

❖ 類別末尾須加分號

類別內資料與函式使用權限

- 公共區域：

  此存取區域內的資料或函式可供類別物件與類別內的函式使用

- 私有區域：

  此存取區域內的資料或函式僅可為類別內的函式所使用

```
Set   foo ;
foo.initialization() ;
foo.insert_element(5) ;
foo.delete_element(6) ;
foo.print_set() ;
```

集合物件能自由使用在定義於公共區域的函式

但是以下的方式已經被禁止

```
foo.element_no = 5 ;  // 錯誤，element_no 為集合類別私有區域的資料
```

❖ 由於類別私有區域內的資料成員受到保護，使得資料的安全性獲得保證

類別內資料成員

- 資料成員通常被放置於私有區域，類別物件可透過定義於公共區域的成員函式來存取這些資料

```
class  FOO  {
   private:
      // 資料成員
      int  dat  ;
   public:
      // 介面成員函式
      void  set_data( int d )  { dat = d ; }
      int   get_data(void)     { return dat ; }
} ;
```

data member

類別內成員函式

- 任何類別存取區域的成員函式都可直接取用類別內的所有資料成員，沒有任何限制
  - 類別內的成員函式不須將資料成員當成參數傳入函式內

```
void   print_set(void)  {
   cout << "集合有 " << element_no << " 元素 :  " ;
   for( int i = 0 ; i < SIZE ; ++i ) {
      if ( status[i] == occupied )
          cout << element[i] << ' ' ;
   }
   cout << endl ;
}
```

  - 類別物件僅能執行在公共區域成員函式，使用方式為

```
Set   foo ;
foo.print_set() ;
```

member function

領域運算子

- 若要將類別的成員函式定義於類別外部時，須使用**類別名稱::函式名稱**方式定義函式，這裡的 **::** 為領域運算子

- 若 **A** 為 **FOO** 類別的成員函式，**B** 為其內部敘述

```
class FOO  {
   private:
      int A(...)  {
         B ;
      }
};
```

```
class FOO  {
   private:
      int A(...) ;
};

int FOO::A(...) {
   B ;
}
```

scope operator

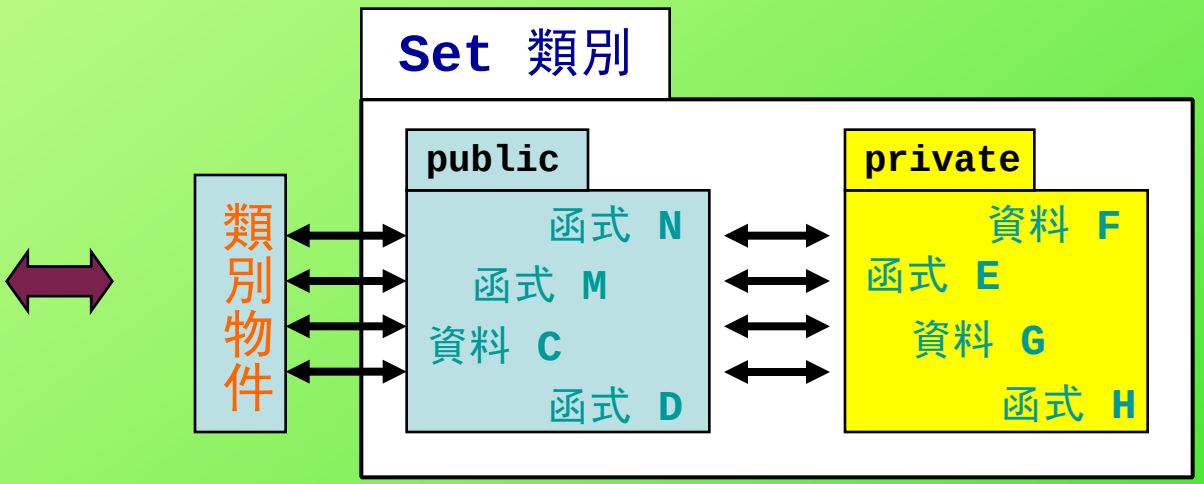
集合類別程式範例

- 將之前的集合結構改寫成集合類別，適當地調整操作函式的介面
- 在程式中，所有的資料成員都被置放在私有區域，這些資料成員僅能為類別的成員函式使用或修改，類別物件不能直接更動資料成員，如此資料的安全性就可以獲得保障
- 由於類別物件僅能透過執行定義於公共區成員函式來存取資料成員，因此公共區的成員函式可看成類別的介面函式

程式　輸出

集合類別程式設計的特點


```
class  Set {
   private:
        A
   public:
        B
};
```



- 類別私有資料為類別所獨有，不開放給類別物件使用
- 類別在公共區域的成員函式可看成類別的使用介面，物件僅能透過此介面間接地存取在私有區域內的成員

結構與類別

- 結構資料型別相當於結構內的所有成員(包含資料成員與成員函式)都被放置在公共區域的類別資料型別

```
struct F00 {
    ...
    ...
};
```

```
class F00 {
    public:
        ...
};
```

建構函式

- 建構函式
  - 類別物件產生時所自動執行的成員函式
  - 可用來設定類別內資料成員的初始值
  - 建構函式的函式名稱與類別名稱相同
  - 一類別可以有許多種建構函式
  - 不同的建構函式有不同的參數列
  - 建構函式不須回傳資料

constructor

集合類別建構函式（一）

- 無參數列的建構函式

```
class  Set {
   private:
      int       element[SIZE] ;   // 儲存集合元素
      Status    status[SIZE] ;    // 元素儲存狀態
      int       element_no ;      // 集合元素的個數
   public:
      Set(void) {
          for( int i = 0 ; i < SIZE ; ++i )
               status[i]  = empty ;
          element_no = 0 ;
      }
};
```

建構函式

集合類別建構函式（二）

- 有參數列的建構函式：普通建構函式

```
class  Set {
   public:
      Set( int no , int data = 0 ) {
          element_no = no ;
          for( int i = 0 ; i < SIZE ; ++i ){
               if( i < no ){
                  element[i] = data ;
                  status[i]  = occupied ;
               } else
                  status[i] = empty ;
          }
      }
};
```

建構函式

建構函式的自動使用

- 普通物件

```
Set a;          // 使用無參數列的建構函式
Set b(5);       // 使用普通建構函式，集合有 5 個 0 元素
Set c(2,3);     // 使用普通建構函式，集合有 2 個 3 元素
```

- 動態產生的物件

```
// 使用無參數列的建構函式
Set *ptr1 = new Set;

// 使用普通建構函式，集合有 5 個 0 元素
Set *ptr2 = new Set(5);

// 使用普通建構函式，集合有 2 個 3 元素
Set *ptr3 = new Set(2,3);
```

備用建構函式

- 如果使用者沒有設定任何型式的建構函式，則編譯器會自動提供無參數列且空無一物的建構函式，稱為備用建構函式，如此以下才能產生 **a** 物件

```
FOO  a ;
```

寫在 **FOO** 類別內

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

寫在 **FOO** 類別外

```
FOO::FOO(){};
```

- 如果使用者有定義任何一種建構函式，則編譯器將不會自動提供備用建構函式

default constructor

建構函式與物件陣列

- 使用備用建構函式產生物件

```
// 陣列共有 10 個集合物件，每個物件都使用備用建構函式產生
Set foo[10];

// 指標可以指到 10 個動態產生的集合物件的首位物件位址
Set *ptr = new Set[10];
```

- 使用參數式建構函式

```
Set a[3] = { Set(2) , Set(9,1) , Set(7) };

Set b[2] = { Set(2,1) , Set(9) };
```

解構函式（一）

- 解構函式：
  - 物件在消失前所自動執行的成員函式
  - 可用來清除類別內動態資料成員所佔有的空間
  - 函式名稱為類別名稱之前加上 ~
  - 解構函式沒有參數列也無回傳型別
  - 解構函式僅能有一個

destructor

解構函式(二)

- 使用方式：

```
class  FOO {
   private:
      int *ptr;              // 整數指標
   public:
      // 建構函式產生一個動態整數空間
      FOO(int a) { ptr = new int(a) ; }
      // 解構函式釋放使用的動態整數空間
      ~FOO(){ delete ptr ; }
};
```

- 備用解構函式：

未使用到動態空間的類別都可不撰寫解構函式，此時編譯器會自動產生空無一物的解構函式 `FOO::~FOO(){}`

`default destructor`

複製建構函式(一)

- 複製建構函式：

  是一種參數列為常數參考類別型別的特殊建構函式
  專門用來複製另一個同類別的物件

```
Set   foo ;          // 使用備用建構函式
Set   bar = foo ;    // 使用複製建構函式複製
Set   bar(foo) ;     // 同上
```

❖ 複製建構函式是用來複製傳入同類別物件的資料成員

copy constructor

複製建構函式(二)

- 基本型式：

```
class  FOO {
   public:
      FOO ( const FOO& a ) { ... } ;
};
```

- 集合類別的複製建構函式：

```
Set::Set( const Set&  foo ) {
   for( int i = 0 ; i < SIZE ; ++i ) {
      status[i]  = foo.status[i];
      element[i] = foo.element[i] ;
   }
   element_no = foo.element_no;
}
```

❖ 被複製的物件都是以常數參考方式傳入複製建構函式

備用複製建構函式

- 類別內若沒有定義複製建構函式，則編譯器會自動產生備用複製建構函式

- 備用複製建構函式的資料成員複製方式是以資料在類別內定義的順序依次複製，這種複製方式被稱為成員式複製

FOO 類別

```
class FOO {
   private:
      int     a , b ;
      char    c[10] ;
      double  d ;
};
```

複製次序

1. a , b
2. c[10]
3. d

memberwise copy

成員式複製的缺陷(一)

- 無法正確地複製動態記憶空間

```
class   FOO {
   private:
      int *ptr;                                  // 整數指標
   public:
      FOO(int a) { ptr = new int(a); } // 指向動態整數空間
      ~FOO() { delete ptr; }                     // 釋放動態整數空間
};
```

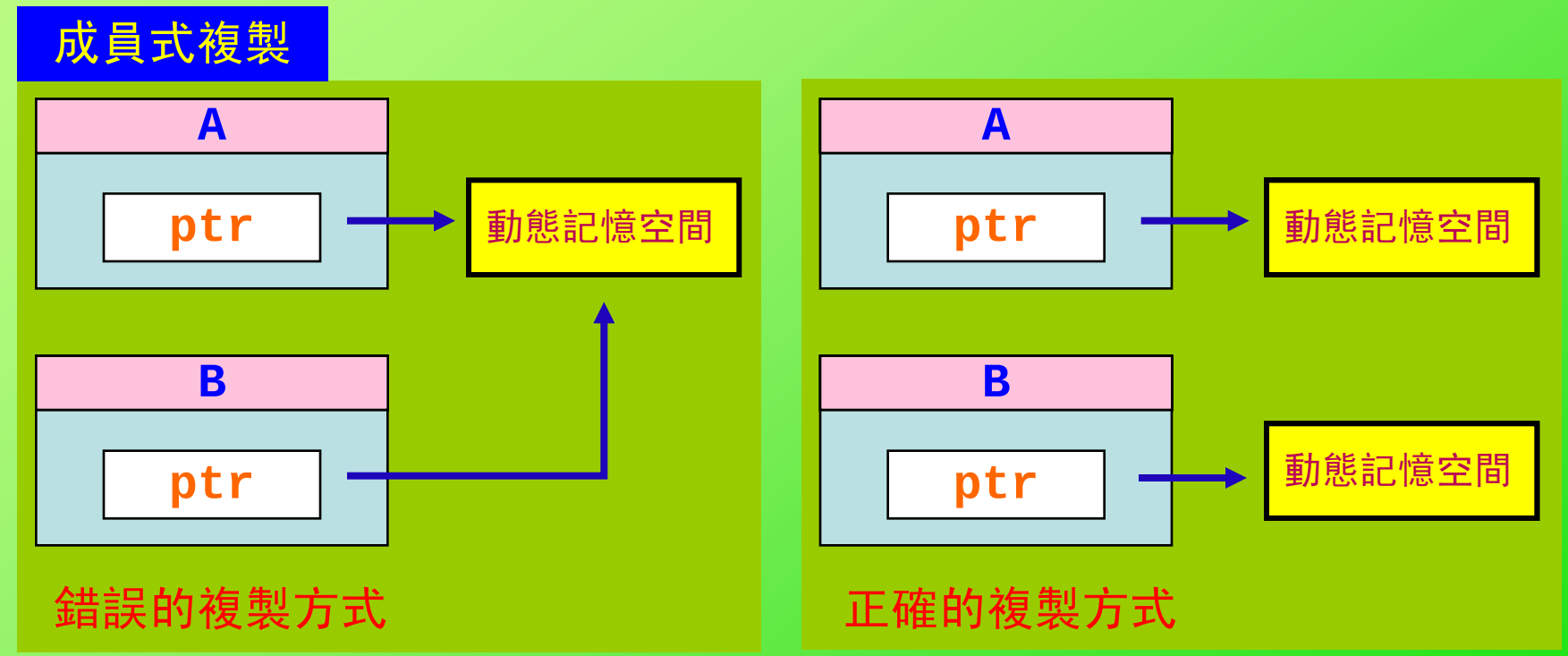
- 備用複製建構函式：成員式複製

```
FOO::FOO( const FOO& foo ) { ptr = foo.ptr ; }
```

❖ C++ 預設的備用複製建構函式，只複製指標資料沒有複製動態空間

成員式複製的缺陷(二)

- 當類別內使用到動態記憶空間時，使用者應自行撰寫複製建構函式以正確地複製動態記憶空間



成員式複製的缺陷(三)

- 正確的動態空間複製方式

```
class  FOO {
   private:
      int  *ptr;
   public:
      FOO(int a){ ptr = new int(a) ; }
      FOO( const FOO& foo ) {
          ptr = new int( *(foo.ptr) ) ;
      }
      ~FOO(){ delete ptr ; }
};
```

指定運算子(一)

- 指定運算子 ：
  - 功能與複製建構函式作用相當，但使用的時機不同
  - 複製建構函式為在物件產生時複製自其它物件
  - 指定建構函式為在物件產生後複製自其它物件

```
Set  foo , bar ;     // 使用預設建構函式產生兩個整數集合
foo.insert(3) ;      // 將整數 3 插入 foo 集合內
bar = foo ;          // 使用集合類別的指定運算子
                     // 將 foo 物件複製給 bar 物件
```

assignment operator

指定運算子(二)

- 基本型式

```
class FOO {
   public:
      FOO& operator=( const FOO& bar ) { ... }
};
```

- 使用方式

```
Set foo , bar ;
foo.insert(3) ;
bar.operator=(foo) ;      // 相當於 bar = foo
```

❖ 指定運算子的函式名稱為 **operator=**，此函式在讀入同類別的物件參數後，回傳物件本身的參考

指定運算子(三)

- 連續指定

```
// 先將 c 複製給 b ， 再將 b 複製給 a
a = b = c ;

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

- 類別內若未定義指定運算子，則編譯器會自行產生備用指定運算子函式以供使用，此函式的資料成員複製方式也是採用成員式複製法

C++自動產生的成員函式(一)

- 備用建構函式：
  在缺少其他建構函式情形下，才會自動產生
- 備用解構函式
- 備用複製建構函式：
  採用成員式複製方式複製資料成員
- 備用指定運算子：
  採用成員式複製方式複製資料成員

C++自動產生的成員函式(二)

- 備用建構函式只在無其他建構函式時才會自動產生

```
class FOO {
   public:
      FOO(int i){ ... }      // 定義某建構函式
};
```

以上的定義無法以第二種方式來產生 **b** 物件

```
FOO  a(3) ;        // 正確，使用參數型建構函式
FOO  b ;           // 錯誤，找不到備用建構函式
```

❖ 若 **FOO** 類別建構函式定義成以下方式，則 **b** 物件即可產生

```
FOO::FOO( int i = 0 ) { ... }
```

C++自動產生的成員函式(三)

- 成員函式的自動運作

```
int main(){
    Set foo(3,0) ;      // foo 執行建構函式
    {
        Set  bar ;      // bar 執行備用建構函式
        bar = foo ;     // bar 執行指定運算子
    }                   // bar 執行解構函式
}                       // foo 執行解構函式
```

- 不可同時定義無法區分的建構函式

```
FOO::FOO() { ... }
FOO::FOO( int i = 0 ){ ... }
```

類別物件的產生時機

- 缺乏效率：

```
for( int i = 0 ; i < 10 ; ++i ){
   FOO  a ;
}
```

❖ 以上 a 物件在迴圈內共產生與消失了 10 次

- 改善方法：

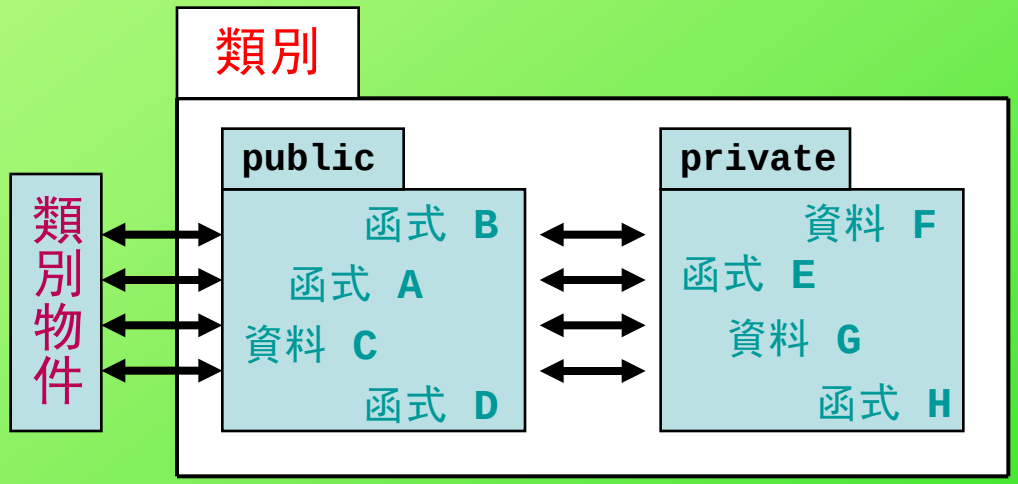
```
FOO  a ;
for( int i = 0 ; i < 10 ; ++i ){
        . . .
}
```

集合類別物件導向程式碼

- 集合類別：完整版本

集合類別物件程式特點

- 物件導向程式設計：
  - 物件導向程式設計是以類別設計為主
  - 類別的資料成員通常被放置在私有區域
  - 類別物件僅能透過公共區域的介面函式間接地取用私有資料成員



object-oriented programming

建構函式初值設定列(一)

- 初值設定列
  - 類別的建構函式可以使用初值設定列來設定物件內資料成員的初值
  - 初值設定列是在建構函式參數列後以冒號：為起始

```
class  Point {
   private :
      int  x , y ;
   public :
      Point( int a , int b ) : x(a) , y(b) {}
      ...
} ;
```

initialization list

建構函式初值設定列(二)

- 若分數類別被定義為

```
class  Fraction{
   private:
      int  num , den ;        // 分子與分母
   public:
      // (1) 使用個別成員指定方式設定初值
      Fraction( int n , int d = 1 ){
           num = n ;   den = d ;
      }
      // (2) 使用初值設定列方式設定初值
      Fraction( int n , int d = 1 ) : num(n) , den(d) {}
};
```

(1) 與 (2) 僅能擇一使用

(1): 資料成員是先被產生，之後才在建構函式內設定初值，包含兩個動作

(2): 資料成員在產生之際就直接設定其初值，僅包含一個動作

建構函式初值設定列(三)

- 一般而言，使用初值設定列的建構函式較有效率
- 在某些特殊情況下，類別的某些資料成員僅能以初值設定列方式設定初值

```
class  Number {
    private:
       int &no ;
    public:
       Number( int& a ) : no(a){}
      int&  val() { return no ; }
};

int  s = 3 ;
Number  foo(s) ;          // foo::no 為 s 的參考
foo.val() = 5 ;           // 修改 foo::no 為 5
cout << s << endl ;       // 列印 5
```

❖ 以上 no 為參考型別，參考型別的參考對象須在建構之際就要設定完成

常數物件與常數成員函式(一)

- 常數物件：

  物件的資料成員不能在程式執行中被更動

- 常數成員函式：

  成員函式在執行時不會更動任何資料成員

❖ 常數物件僅能執行常數成員函式藉以避免物件內部資料遭受更動

constant member function

常數物件與常數成員函式(二)

- 集合類別：

```
class  Set {
   public:
      bool  insert_element( int ) ;      // 加入元素
      int   delete_element( int ) ;      // 刪除元素
      void  print_set()           ;      // 列印集合
};
. . .
const  Set  foo ;

foo.insert_element(1) ;      // 錯誤
foo.delete_element(1) ;      // 錯誤
foo.print_set()   ;          // 錯誤
```

- 最後一式僅是單純地列印集合元素，並不會更動集合內資料成員之值，若仍不能使用，則不太合理

常數物件與常數成員函式（三）

- 常數成員函式： 在成員函式參數列後加註 **const**

```
class Set{
   public:
      // 註明其為常數成員函式
      void print_set() const {
           . . .
      }
};
```

- 非常數物件也可使用常數成員函式

❖ 若類別內的成員函式並不會修改任何資料成員，則要毫不吝嗇地加註其為常數成員函式，使得其同時可被常數與非常數物件所使用

常數物件與常數成員函式(四)

- 在成員函式參數列後的 const 也會被視為函式特徵資料的一部份

```
class  Set {
   public:
      int elements( int i ) ;         // (1) 非常數成員函式
      int elements( int i ) const ;   // (2) 常數成員函式
};
. . .
Set         foo ;
const  Set  bar ;
. . .
cout << foo.elements(2) << endl;    // 使用 (1)
cout << bar.elements(2) << endl;    // 使用 (2)
```

常數物件與常數成員函式(五)

- 如果類別內僅有常數型的成員函式，則不管是常數或者是非常數物件都可使用

```
class  Set {
   public:
      int size() const ;          // 回傳集合元素的數量
      int add( int i ) ;          // 加入新元素於集合內
};
```

```
Set  foo ;
foo.add(3);   foo.add(5);         // 將 3 , 5 加入集合內
cout << foo.size() << endl;       // 列印集合元素的數量
```

```
const  Set  bar ;
bar.add(3);   bar.add(5);         // 錯誤，bar 為常數物件
                                  // 不能使用非常數成員函式
cout << bar.size() << endl;       // 列印集合元素的數量
```

靜態資料成員(一)

- 靜態資料成員：

  為類別所有物件所共享的資料成員

```
class  FOO {
   private:
      static int count ;          // 類別的靜態資料成員
   . . .
};
```

- 每個類別物件都有各自獨立的資料成員，但所有的類別物件共用一份靜態資料成員

❖ 每個類別物件都可將靜態資料成員當成自己的資料成員使用

static data member

靜態資料成員(二)

- 初值設定：

  靜態資料成員可在類別之外設定其初值

```
class FOO {
   private:
      static int count ;
   . . .
} ;

int  FOO::count = 0 ;
```

❖ 靜態資料成員若不設初值，則會以 0 為其初值

靜態成員函式(一)

- 靜態成員函式：

類別內專門存取靜態資料成員的函式

```
// 以下的 count 是用來計算類別物件的總個數
class FOO {
    private:
        static int count ;    // 靜態資料成員，用來計算類別的物件個數
    public:
        // 當執行建構函式時，產生了新的物件，因此 count 加一
        FOO(){ ++count ; }
        FOO (const FOO& a) { ++count ; }
        // 當備用建構函式被執行時，物件消失，因此 count 減一
        ~FOO(){ --count ; }

        // 不須加 const
        static int object_no() { return count ; }
};
```

static member function

靜態成員函式(二)

- 用法：

類別名稱::靜態成員函式

```
FOO  a ;
for( int i = 0 ; i < 3; ++i ){
   FOO b = a ;
   cout << FOO::object_no() << endl  ;  // 輸出 2
}
cout << FOO::object_no() << endl  ;     // 輸出 1

// 也可以輸出 1，但不建議使用
cout << a.object_no() << endl;
```

❖ 以上的靜態成員函式是置放在類別的公共區內。此外應儘量避免直接使用非靜態成員函式修改靜態資料成員，以免造成物件間不完全獨立的現象

銀行利息範例

- 銀行的利率適用於所有的帳號
  - 使用靜態資料成員儲存利率
  - 使用靜態成員函式調整利率

```
class  Deposit {
   private :
      // 所有存款帳號的利率
      static double interest_rate ;
   public :
      // 設定所有存款帳號的新利率
      static void set_rate( double rate ){
         interest_rate = rate ;
      }
} ;
// 設定存款帳號類別的內定靜態資料初值，不可省略
double  Deposit::interest_rate = 0.05 ;
```

程式

輸出

類別常數(一)

- 類別常數：類別內專用的常數

使用靜態資料成員

```
class  FOO {
   private:
       //類別專用的整數常數
       static const  int     TOL = 10 ;
       //類別專用的浮點數常數
       static const  double  Err ;
};
//設定 FOO 類別的常數初值
const double FOO::Err = 1.0e-10 ;
```

- 整數類型的常數靜態資料成員可以直接將初值寫在類別內但其它型別則須將初值寫在類別外

類別常數(二)

- 類別常數：類別內專用的常數

使用列舉型別

```
class Card {
   private:
      enum { Jack=11 , Queen=12 , king=13 , Ace=14 };
      enum { SIZE=52 };
      int card[SIZE] ;
   public:
      int get_jack() const { return Jack ; }
};
```

❖ 列舉型別無法設定非整數的常數資料值

行內成員函式

- 行內成員函式：

  將編譯後的成員函式程式碼直接置放在呼叫處，使得程式在執行時以非函式型式執行函式，如此可提高函式執行效率

```
class FOO {
   public:
      inline int get(){ ... }  // 正確的行內函式定義方式
      inline int print() ;     // 錯誤的行內函式定義方式
};
int FOO::print(){ ... }        // 錯誤的行內函式定義方式
```

- ❖行內成員函式一定要將程式碼定義於類別內
- ❖定義於類別內的成員函式皆會被自動視為行內函式
- ❖太複雜的成員函式仍然無法以行內函式的方式編譯

inline member function

夥伴函式與類別

- 夥伴函式

  物件在夥伴函式內可以直接存取物件類別內的所有成員，就像此夥伴函式為類別的成員函式一般

- 夥伴類別

  物件在夥伴類別的所有成員函式內使用，都可以自由地取用其類別內的所有成員

```
class  FOO {
   public:
      friend void bar(); // bar 為類別 FOO 的夥伴函式
      friend class BAR ; // 類別 BAR 為類別 FOO 的夥伴類別
};
```

❖ 夥伴函式不是類別的成員函式

friend function
friend class

使用夥伴函式

- 享有「夥伴」待遇的夥伴函式

```
class  FOO {
   private :
      int a ;
   public :
      ...
      friend void print( const FOO& foo ) {
         cout<< foo.a << endl ;
      }
};
FOO  bar ;
...
print(bar) ;
```

❖ 物件在夥伴函式與夥伴類別內使用時，可自由地取用物件所屬類別內的所有成員，但如此會降低資料成員的安全性，應儘量減少使用

集合範例：使用動態空間(一)

- 靜態陣列的缺點：
  - 可能浪費記憶空間
  - 無法儲存超過設定的元素個數

```
const int M = 100 ;
int a[M] ;

int n = 100 ;
int* p = new int[n] ;      // 動態空間可儲存 100 個元素
. . .
delete [] p ;              // 退還動態空間回系統
p = new int[2*n] ;         // 動態空間可儲存 200 個元素
```

集合範例：使用動態空間(二)

- 集合類別設計：

  由於系統預設的**成員式複製方式**無法正確地複製動態資料空間，因此

  - 複製建構函式
  - 指定運算子
  - 解構函式

  都要自行定義，不能依靠編譯器自動產生

集合範例：使用動態空間(三)

- 在程式中，我們特別讓指定運算子函式沒有回傳任何資料，但這會造成以下的使用問題

```
Set  a , b , c ;

// 正確的使用方式，即 a.operator=(b)
a = b ;

// 錯誤的使用方式，即 a.operator=(b.operator=(c))
a = b = c ;
```

❖ 以上 b = c 並沒有回傳任何物件，使得 a 物件的指定運算子右側沒有資料，因而造成錯誤

成員函式的連續使用

- 若成員函式回傳物件，則可以接續執行此物件所屬類別的成員函式，達到連續使用的效果

```
class  Polynomial {
   private:
      int       deg  ;
      double  c[100] ;
   public:
      ...
      Polynomial  derivative() const ;  // 回傳多項式的微分
      double      value(int a) const ;  // 計算 f(a)
};
```

```
Polynomial  f ;
cout << f.derivative() ;                  // 相當於 f'(x)
cout << f.derivative().value(3) ;         // 相當於 f'(3)
cout << f.derivative().derivative() ;     // 相當於 f''(x)
```