Chapter 15

C++字串

C++ 字串

- C++ 字串：字串樣版類別

```
typedef  basic_string<char>      string ;
typedef  basic_string<wchar_t>   wstring ;
```

- 類別的型別樣版參數分別為一般字元 (**char**)與寬字元 (**wchar_t**)，後者是由多個位元組所構成的字元，如中文字，日文字等

- 現少有編譯器支援寬字元字串
- **basic_string** 樣版類別：基本字串類別
- 使用 C++字串須使用 **string** 標頭檔

傳統字串與字元指標

- 傳統字串以空字元 '\0' 為字串末尾字元，經常透過字元指標存取字元

```
char foo[] = "Ship in a bottle" ;
char *ptr ;
// 利用迴圈與指標，將字串的小寫字元改成大寫
for ( ptr = foo ; *ptr != '\0' ; ++ptr ) {
    if ( *ptr >= 'a' && *ptr <= 'z')
        *ptr = *ptr – 'a' + 'A' ;
}
cout << foo << endl ;     // 輸出： SHIP IN A BOTTLE
```

C++字串與字串迭代器（一）

- C++字串不以空字元當末尾，但可用迭代器取代字元指標的功能

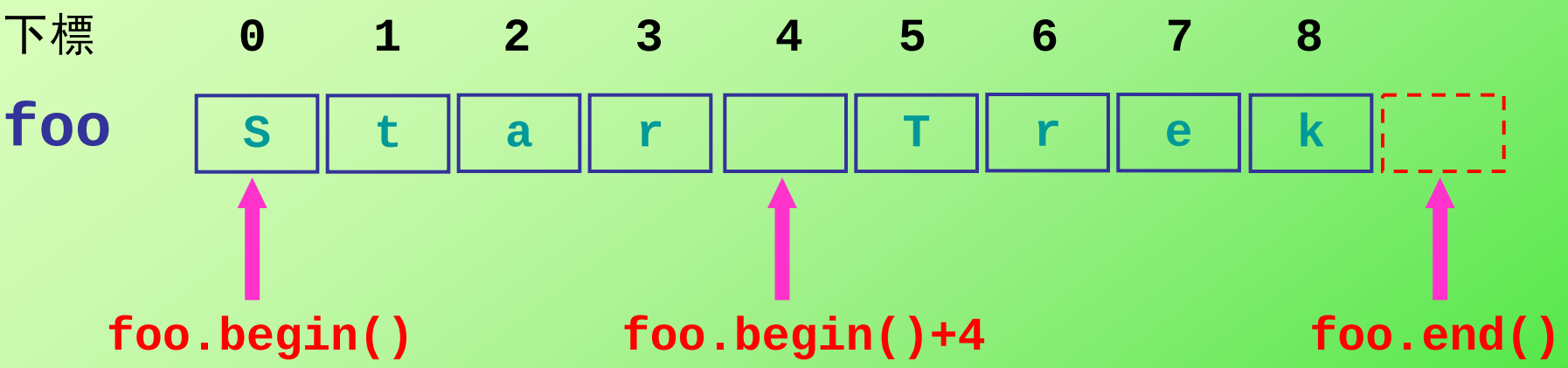
```
string foo = "Star Trek" ;
string::iterator iter    ;   // 定義 c++ 字串迭代器
for( iter = foo.begin() ; iter != foo.end() ; ++iter){
    if ( *iter >= 'a' && *iter <= 'z')
        *iter = *iter - 'a' + 'A' ;
}
cout << foo << endl ; // 輸出：STAR TREK
```

❖ 以上的 iter, foo.begin(), foo.end() 皆為 C++ 字串迭代器

iterator

C++字串與字串迭代器（二）

- 迭代器作用類似指標



- 以上的 `foo.begin()` 指向 `foo` 字串的第一個字元、`foo.end()` 指向 `foo` 字串末尾字元後的下一個位置

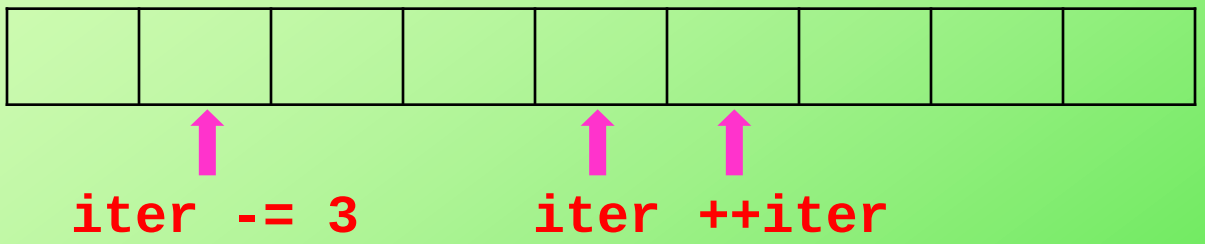
- 迭代器的型別為 `string::iterator`

```
string::iterator  iter ;  // 產生一迭代器物件
```

C++字串與字串迭代器（三）

- 迭代器物件可以仿照指標方式，透過覆載的 +,-, ++,--,+=,-= 等運算子自由移動所指向的字元

```
++iter    ;    // 往後移動一個字元
iter -= 3 ;    // 往前移動三個字元
```



- 迭代器也可以使用參照運算子取得字元資料

```
string foo = "abc"  ;
string::itreator iter = foo.begin() ;
cout << *iter << endl ;      // 列印 'a'
```

C++字串與字串迭代器（四）

- 若是字串為常數，則要定義迭代器為常數迭代器

```
const string foo = ”Ship in a bottle” ;
string::const_iterator  iter ;

// 列印 foo 字串 字元間以空白分開
for( iter = foo.begin() ; iter != foo.end() ; ++iter){
    cout << *iter << ’ ’ ;
}
```

C++字串的產生（一）

■ 產生 C++ 字串方式

使用方式	作 用
`string foo ;`	空字串
`string foo = "" ;`	空字串
`string foo = "Star Trek" ;`	字串包含 'S','t' 等九個字元
`string foo("Star Trek") ;`	同上
`string foo( 5 , 'a' ) ;`	五個'a'字元的字串
`string bar = "Star Trek" ;` `string foo = bar ;`	複製 bar 給 foo

❖ 錯誤的字串產生方式

```
string foo = 'a' ;  // 須改成 string foo = "a"
string bar = 20  ;  // 無法由整數轉成字串
```

C++字串的產生（二）

- 由傳統字串內複製子字串成一C++字串

```
char bar[] = "Picard"   ;

// 將傳統字串 bar 由下標值 2 起 連續 4 個字元複製給 foo
// foo = "card"
string foo(bar + 2 , 4) ;
```

C++字串的產生（三）

- 複製某 C++ 字串 bar 由下標 p 後的 s 個字元

```
string bar = "The best of both worlds" ;
string foo( bar.begin() + p , bar.begin() + p + s ) ;
```

- 複製某 C++ 字串 bar 由下標 p 後的所有字元

```
// string::npos 為無號整數，其值為 2^32-1
string foo1( bar , p , string::npos ) ;

string foo2( bar.begin()+p , bar.end() ) ;
```

C++字串長度

- C++字串長度：使用 **length()** 或 **size()** 取得

```
string foo = "data" ;

// 皆輸出 4
cout << "字串長度 :" << foo.size()   << endl ;
cout << "字串長度 :" << foo.length() << endl ;
```

❖ C++字串並不儲存空字元

下標運算子（一）

- C++字串內的字元可以使用下標運算子取得

```
string foo = "borg" ;
for ( int i = 0 ; i < foo.length() ; ++i )
   cout << foo[i] << '-' ;
```

- 若要將字母轉成大寫

```
string foo = "picard" ;
for ( int i = 0 ; i < foo.length() ; ++i )
   foo[i] = foo[i] - 'a' + 'A' ;
```

❖ C++ 字串的下標範圍為 0 ～ length()-1

下標運算子（二）

- 兩種不同的下標運算子

```
int i , j ;
string foo[3] = { "I" , "am" , "borg" } ;
for ( i = 0 ; i < 3 ; ++i ) {
    for ( j = 0 ; j < foo[i].size() ; ++j ) {
        if ( foo[i][j] >= 'a' && foo[i][j] <= 'z')
            foo[i][j] = foo[i][j] - 'a' + 'A' ;
    }
}
```

- `foo[i]` : `foo` 陣列的第 `i+1` 個字串
- `foo[i].size()` : `foo` 陣列的第 `i+1` 個字串的長度
- `foo[i][j]` : `foo` 陣列的第 `i+1` 個字串的第 `j+1` 個字元

檢驗下標：at 成員函式

- at 成員函式作用如同下標運算子，但其會檢查輸入的下標運算子是否在字元範圍內，若下標超出範圍則程式立即終結

```
string foo = ”picard” ;
for ( int i = 0 ; i < foo.length() ; ++i )
   foo.at(i) = foo.at(i) – ’a’ + ’A’ ;
```

C++字串的指定（一）

- 字元、傳統字串、或者是另一個 C++ 字串都可以利用指定運算子將資料複製到另一個 C++ 字串

```
string   foo , bar ;
foo = 'I'          ;   // 字元的指定
bar = "Borg"       ;   // 傳統字串的指定
bar = foo          ;   // 同型別字串的指定
```

- 但以下並非指定

```
string foo = 'I' ;   // 需改成 string foo = "I" ;
```

C++字串的指定（二）

- 字元資料的指定也可以利用 **assign** 函式

```
string foo , bar = "Startrek Enterprise" ;


foo.assign( 5 , 'Q' ) ;        // foo = QQQQQ
foo.assign( "Borg" ) ;         // foo = Borg
foo.assign( "Borg" , 3 ) ;     // foo = Bor

// foo = Enterprise
foo.assign( bar , 9 , string::npos ) ;

// foo = Startrek
foo.assign( bar.begin() , bar.begin() + 8 ) ;
```

由C++字串轉成傳統字串

- 使用 **c_str()** 可以由 **C++** 字串轉換成傳統字串

```
string foo = "11001001"          ;
int    no  = atoi( foo.c_str() ) ;
```

- 以上的 **atoi** 函式是用來將一傳統字串的數字字串參數，轉成一整數回傳，此函式在 **stdlib.h** (或 **cstdlib**)標頭檔內

C++字串邏輯運算子

- C++字串之間可以使用 == , != , < , <= , > , >= 等邏輯運算子來比較字串的「大小」

- 字串的大小是由字串的第一個字元起依字元在上的順序一一作比較

- 常見的字元大小在 ASCII 表內

  阿拉伯數字 < 英文大寫字母 < 英文小寫字母

比較大小：compare

- 也可以使用 compare 成員函式比較兩字串

```
string foo( ”the inner light” ) ;
string bar( ”The Inner Light” ) ;
cout << foo.compare( bar ) << endl ;
```

- 以上如果

  foo > bar 則回傳大於零的整數

  foo < bar 則回傳小於零的整數

  foo == bar 則回傳整數 0

字串合成（一）

- **operator+** 可將 C++ 字串與字元、傳統字串或另一個 C++ 字串連接在一起，合成後函式回傳一 C++ 字串

```
string foo ;
foo = 'I' + " Borg" ;                  // 錯誤
foo = 'I' + string( " Borg" );         // 正確

// 錯誤
foo = "You too will " + "be " + "assimilated" ;
// 正確
foo = string( "You too will " ) + "be " +
      "assimilated" ;
```

字串合成（二）

- 可利用 **operator+=** 將字元、傳統字串或另一個 C++ 字串加於字串之後

```
string foo = "Sector " ;
foo += "001"           ;
```

- 傳統字串不能相加在一起

```
string foo = "You too will "  ;
foo += "be " + "assimilated" ;  // 錯誤
```

C++ 沒有傳統字串的加法運算子，應該為

```
foo += "be " + string("assimilated") ;
```

附加字串：append

- append 作用如同 operator+=，但更多變化

```
string DNA = "AAA" , DNA2 = "GCGC" ;
char DNA3[] = "TCTC" ;

DNA.append( DNA2 )                          ;   // AAAGCGC
DNA.append( DNA2.begin() , DNA2.end() );   // AAAGCGC
DNA.append( DNA2 , 1 , 3 )                  ;   // AAACGC


DNA.append( DNA3 )      ;  // AAATCTC
DNA.append( DNA3 , 3 ) ;  // AAATCT
DNA.append( 3 , 'C' )  ;  // AAACCC
```

插入字串：insert （一）

- C++字串的插入：

```
string DNA = "AAA" , DNA2 = "GCGC" ;
```

➢ 在 DNA 字串下標 2 位置之前，插入整串 DNA2

```
DNA.insert( 2 , DNA2 ) ;                    // DNA = AAGCGCA
```

➢ 在 DNA 字串下標 2 位置之前，插入 DNA2 由下標 1 開始的連續 3 個字元

```
DNA,insert( 2 , DNA2 , 1 , 3 ) ;    // DNA = AACGCA
```

➢ 在 DNA 字串的第二個字元前，插入整串 DNA2 字串

```
DNA,insert( DNA.begin() + 1 ,          // DNA = AGCGCAA
            DNA2.begin() , DNA2.end() ) ;
```

插入字串：insert（二）

- 傳統字串的插入：

```
string DNA = "AAA"   ;
char DNA3[] = "TCTC" ;
```

➢ 在 DNA 字串下標 1 位置之前，插入整串 DNA3

```
DNA.insert( 1 , DNA3 ) ;          // DNA = ATCTCAA
```

➢ 在 DNA 字串下標 1 之前，插入 DNA3 字串前 3 個字元

```
DNA,insert( 1 , DNA3 , 3 ) ;      // DNA = ATCTAA
```

➢ 在 DNA 字串下標 1 位置之前，插入 3 個'C'字元

```
DNA,insert( 1 , 3 , 'C' ) ;       // DNA = ACCCAA
```

➢ 在 DNA 的起始字元前，插入 1 個'C'字元

```
DNA,insert( DNA.begin() , 'C' ) ; // DNA = CAAA
```

插入字串：insert（三）

- 所有的 **insert** 函式都回傳字串本身，因此可以在一個式子重複使用

```
string DNA = "TTT"   ;

// DNA = TCAATT
DNA.insert( 1 , 2 , 'A' ).insert( 1 , 1 , 'C' ) ;

// 列印 : CATTCAATT
cout << DNA.insert( 0 , "CAT" ) << endl ;
```

❖ 字串的插入須要耗費相當的時間，應避免經常性的使用

子字串

- 子字串：

  由原字串內某段連續的字元所構成的字串。子字串最少包含一個字元，最長可以等於原字串

```
string foo = ”The Lord of The Rings” ;
```

❖ 劃底線的部分皆可為子字串

substring

子字串的取出：substr

- **substr(p,s)：**

  代表字串在下標 **p** 開始後的連續 **s** 個字元所構成的子字串。若 **s** 被省略則代表下標 **p** 之後的所有字元所構成的子字串

```
string foo = "Red Alert !!!" ;
cout << foo.substr(4,5) << endl ; // 列印： Alert
cout << foo.substr(4)   << endl ; // 列印： Alert !!!
```

❖ **substr** 回傳子字串的複製字串，無法當作指定運算子的左值使用

```
string foo = "Red Alert !!!"  ;
foo.substr(4,5) = "ALERT" ;        // 錯誤
```

子字串的搜尋：find（一）

- **find(str)**：

  由字串起始尋找 **str** 子字串所在的下標位置

```
string foo = "Number One" , bar = "Riker" ;
cout << foo.find( 'u' )   << endl ;  // 列印： 1
cout << foo.find( "One" ) << endl ;  // 列印： 7
cout << foo.find( bar )   << endl ;  // 列印：數值
                                     // string::npos
```

  以上的 **str** 型別可為字元、傳統字串或C++字串

- ❖ 所有的找尋函式都回傳無號整數型別的下標位置
- ❖ 若搜尋的子字串不在字串內，則回傳 **string::npos** 之值

子字串的搜尋：find（二）

- **find(str , p)：**

  由字串的第 p 個字元下標起始尋找 str 子字串所在的下標位置

```
string foo = "11001001" , bar = "100" ;
cout << foo.find( '1'   , 3 ) << endl ; // 列印： 4
cout << foo.find( "100" , 3 ) << endl ; // 列印： 4
cout << foo.find( bar   , 3 ) << endl ; // 列印： 4
```

子字串的搜尋：find（三）

- **find(str , p , n)：**

  由字串的第 p 個字元下標起始找尋由 str 字串的前 n 個字元所構成的子字串的下標位置

```
char   bar[] = "011"      ;
string foo   = "11001001" ;

// 列印： 3
cout << foo.find( bar , 1 , 2 ) << endl ;

// 列印： string::npos 的值
cout << foo.find( bar , 1 , 3 ) << endl ;
```

另類搜尋（一）

- C++字串另提供五種其他形式的子字串搜尋

成員函式名稱	作　用
rfind	由字串末尾往前找尋子字串的位置
find_first_of	找尋第一個出現於子字串的字元位置
find_last_of	找尋最後一個出現於子字串的字元位置
find_first_not_of	找尋第一個未出現於子字串的字元位置
find_last_not_of	找尋最後一個未出現於子字串的字元位置

- 後四種是用來找尋字元位置，其中前兩種是找尋字元出現在子字串內的位置，後兩種則是用來找尋字元不在子字串內的位置。

另類搜尋（二）

- 使用範例

```
string DNA = "ATGCGCTA" ;

cout << DNA.find( "GC" )  << endl ;  // 2

cout << DNA.rfind( "GC" ) << endl ;  // 4

cout << DNA.find_first_of( "CG" ) << endl ;   // 2

cout << DNA.find_last_of( "GC" )  << endl ;   // 5

cout << DNA.find_first_not_of( "AGC" ) << endl ;   // 1

cout << DNA.find_last_not_of( "AGC" )  << endl ;   // 6
```

搜尋與複製子字串

- 取得中括號內子字串

```
string lsymbol = "[" , rsymbol = "]" ;
string picard = "I am Locutus of [Borg]. "
                "Resistance is futile." ;
int i = picard.find_first_of( lsymbol ) ;
int j = picard.find_first_of( rsymbol , i+1 ) ;
cout << picard.substr( i+1 , j-(i+1) ) << endl ;
                                    // 列印：Borg
```

- 取得數字所構成的子字串

```
string num = "0123456789" ;
string data = "Captain, the Borg "
              "have entered Sector 001" ;
int i = data.find_first_of( num ) ;
int j = data.find_first_not_of( num , i+1 ) ;
cout << data.substr( i, j-i ) << endl ;    // 列印：001
```

子字串的取代：replace（一）

- **replace(p , n , ...)：**

  取代字串由下標 **p** 起連續 **n** 個字元所構成的子字串

```
string foo = "Number One" , bar = "12345" ;
foo.replace( 7 , 3 , bar ) ;              // (1)
foo.replace( 7 , 3 , bar , 0 , 1 ) ;      // (2)
foo.replace( 7 , 3 , "123" ) ;            // (3)
foo.replace( 7 , 3 , "123" , 2) ;         // (4)
foo.replace( 7 , 3 , 5 , '1' ) ;          // (5)
```

(1) "Number 12345" (2) "Number 1" (3) "Number 123"

(4) "Number 12" (5) "Number 11111"

子字串的取代：replace（二）

- **replace(a , b , ...)：**

  取代由迭代器在[a ,b)範圍之間所構成的子字串

```
string a = "Number One" , b = "12345" ;
a.replace(a.begin()+7, a.begin()+10, b) ;              // (1)
a.replace(a.begin()+7, a.begin()+10, b.substr(0,1));// (2)
a.replace(a.begin()+7, a.begin()+10, "123") ;          // (3)
a.replace(a.begin()+7, a.begin()+10, "123", 2) ;       // (4)
a.replace(a.begin()+7, a.begin()+10, 5, '1');          // (5)
```

(1) "Number 12345"　(2) "Number 1"　(3) "Number 123"

(4) "Number 12"　(5) "Number 11111"

先搜尋後取代子字串

- 將 **foo** 字串的所有子字串 **a** 取代成字串 **b**

```
void substitute( string& foo , const string& a ,
                 const string& b ) {
    int i = 0 ;
    while( 1 ) {
        i = foo.find( a , i ) ;
        if ( i == string::npos ) return ;
        foo.replace( i , a.size() , b ) ;
        i += b.length()                 ;
    }
}
```

也可以使用

```
string foo = "Warp 13" ;
foo.replace( foo.find( "13" ) , 2 , "9.9" ) ;
```

子字串的刪除：erase

- **erase**：將指定範圍的字元去除

```
foo.erase() ;           // 清除所有字元
foo.erase( 3 ) ;        // 清除下標 3 之後的所有字元
foo.erase( 3 , 4 ) ;    // 清除下標 3 之後的 4 個字元
```

也可以使用迭代器

```
string::iterator i = foo.begin() ;
foo.erase( i+3 ) ;                  // 清除在下標 3 的字元
foo.erase( i+3 , foo.end() ) ;      // 清除下標 3 後的所有字元
foo.erase( i+3 , i+7 ) ;            // 清除下標 3 後的 4 個字元
```

C++字串的輸入與輸出

- 使用覆載的 **operator<<** 與 **operator>>**

- **getline**：讀入一整行到C++字串

```
string line ;
while( 1 ) {
    getline( cin , line ) ;
    cout << line << endl  ;
}
```

其他 C++ 字串函式

- **bool empty()**：檢查字串是否為空字串

- **void resize(size_type n , char c)**：

  調整字串長度為 **n**，若字元數增加則補上字元**c**，若字元數減少則去除多的字元

- **void swap(string& a , string& b)**：

  對調 **a , b** 兩字串

❖ 前兩種為字串類別內的成員函式，最後一種則為一般函式

C++字串陣列與指標

- C++字串陣列

```
string foo[]  = { "I" , "am" , "borg" } ;
string foo[3] = { "I" , "am" , "borg" } ;
```

- 字串指標

```
string *foo ;
foo = new string( "borg" ) ;
```

- 字串指標陣列

```
string *bar[3] ;
bar[0] = new string( "I" ) ;
bar[1] = new string( "am" ) ;
bar[2] = new string( "borg" ) ;
string *bar[2]= { new string("I") , new string("borg") };
```

使用上

```
for( int i = 0 ; i < 2 ; ++i )  cout << *bar[i] << endl ;
```

範例一：有用的字串運算子

- 覆載 operator* , *= , - , -= 等

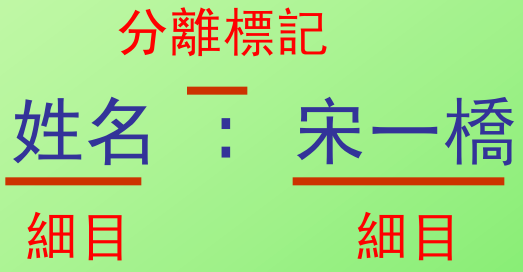
```
string  foo = "windy" , bar = "dy" ;

foo = foo * 2 ;    // WindyWindy
foo *= 2      ;    // WindyWindy

foo = foo – bar ; // Win
foo -= bar      ; // Win
```

程式

輸出

範例二：拆解字串

- 讀入一筆記錄(record)，將其細目(field)資料取出，細目間以分離標記隔開



程式 輸出

範例三：字串內的數字和

- 分析輸入的字串行，將字串內的整數總和算出來

輸入：`Voyager, 20 to be beamed up. Voyager, 3 more to be beamed up.`

輸出：`23`

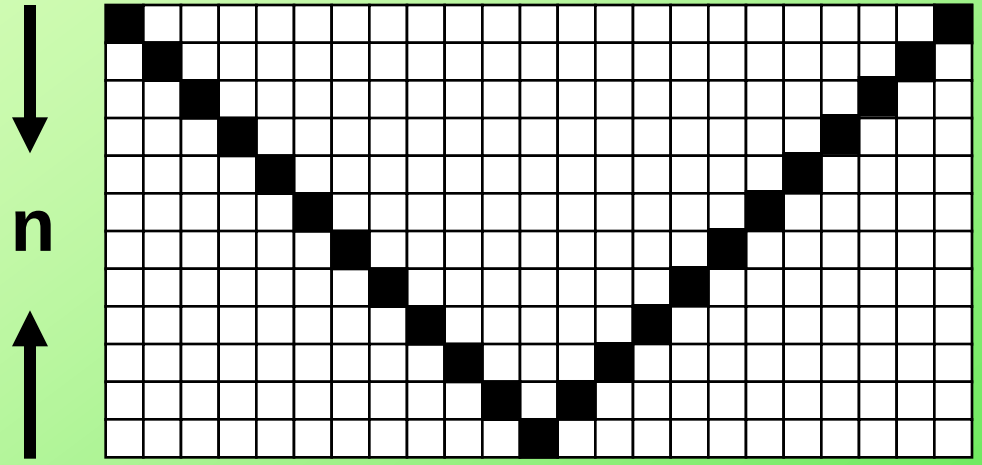
❖ 數字字串轉數字也可使用字串串流物件來處理

```
int no ;
istringstream num ;
num.str( ”-12” )  ;
num >> no         ;
```

程式　輸出

範例四：點矩陣函數圖形

- 目標：以點矩陣的方式畫出圖形於螢幕上
- 方法：將螢幕上的每一列用C++字串表示。
  每一列的行數即為字串的長度



螢幕

```
string line[n] ;
```

$f(x) = |X|$

程式　輸出