Appendix F

string 樣版類別

這個附錄大多數的內容都比較偏向技術性質。然而，如果您只想知道 string 樣版類別的功能，可以將精神集中在各個 string 成員函數的描述即可。

string 類別是根據樣版定義：

```
template<class charT, class traits = char_traits<charT>,
                 class Allocator = allocator<charT> >
class basic_string {...};
```

此處 charT 表示儲存在字串中的型態。traits 參數代表一個類別，定義要表示字串之型態所必備的性質。例如，它應有 length() 成員函數，可以回傳以 charT 陣列表示之字串的長度。這種陣列的尾端會用值 charT(0) 表示，這是 null 字元的一般表示法（運算式 charT(0) 是將 0 型態轉換為型態 charT。它只是 0，就如同型態 char，或是更一般的說法是由 charT 建構函數產生的物件）。這類別也包含比較值的成員函數等等。Allocator 參數表示處理字串記憶體配置的類別。預設的 allocator<charT> 樣版以標準的方式使用 new 和 delete。

有四個已經定義的特定化：

```
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

這些特定化再利用下面的特定化：

```
char_traits<char>
allocator<char>
char_traits<char16_t>
allocator<char_16>
char_traits<char_32>
allocator<char_32>
char_traits<wchar_t>
allocator<wchar_t>
```

您可以定義 traits 類別和使用 basic_string 樣版，而產生不同於 char 或 wchar_t 的 string 類別。

13 種型態和 1 個常數

basic_string 樣版定義可以用於定義成員函數的數種型態：

```
typedef traits                                  traits_type;
typedef typename                                traits::char_type  value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type     difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference     const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
```

注意 traits 是樣版參數，會對應至某一型態，如 char_traits<char>；這會用 typedef 將此特殊型態定義為 traits_type。表示法

```
typedef typename traits::char_type          value_type;
```

意思是 char_type 是定義在 traits 類別中的型態名稱。關鍵字 typename 是用來告訴編譯程式，運算式 traits::char_type 是一種型態。對於字串特定化，例如，value_type 會是 char。

size_type 的用法就像是 size_of，不一樣的地方是它用儲存型態回傳字串的大小。對於字串特定化，會以 char 型態計算，此時 size_type 與 size_of 相同。這是無號型態。

difference_type 用來測量字串的兩個元素之間的距離，同樣是以單一元素的大小為單位。通常它是以 size_type 為基礎的有號版本型態。

對於 char 特定化，pointer 的型態是 char *，而 reference 是型態 char &。若您產生自己設計之型態的特定化，這些型態（pointer 和 reference）會參考具有相同性質的類別，就如同基本的指標和 reference。

要使標準樣版函式庫（STL）演算法處理字串，此樣版要先定義一些迭代器型態：

```
typedef (models random access iterator) iterator;
typedef (models random access iterator) const_iterator;
typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

這樣版定義一個靜態常數：

```
static const size_type npos = -1;
```

因為 size_type 是無號數，指定 -1 實際上是指定最大的可能無號數至 npos。此值相當於最大的可能陣列索引值加 1。

資料訊息，建構函數等

建構函數可用其結果描述之。因為類別的私有部分與實作無關，這些結果應可描述為公有介面中的可用資訊。表 F.1 列出數個成員函數，其回傳值都可用來描述建構函數及其他成員函數的結果。注意有許多術語都來自 STL。

表 F.1 一些 string 資料成員函數

成員函數	回傳值
begin()	指向字串第一個字元的迭代器（亦可用在 const 版本，這回傳 const 迭代器）。
cbegin()	指向字串第一個字元的 const_iterator（C++11）
end()	指向 past-the-end 值的迭代器（亦可用在 const 版本）。
cend()	指向 past-the-end 值的 const_iterator（C++11）
rbegin()	指向 past-the-end 值的反向迭代器（亦可用在 const 版本）。
crbegin()	指向 past-the-end 值的反向 const_iterator（C++11）
rend()	指向第一個字元的反向迭代器（亦可用在 const 版本）。
crend()	指向第一個字元的反向 const_iterator（C++11）
size()	字串中的元素個數，等於 begin() 至 end() 的距離。
length()	與 size() 相同。
capacity()	在字串中，配置的元素個數。這個可以比實際字元個數還要來的大。capacity()-size() 的數值表示在需要配置更多記憶體空間以前，還能夠加入字串內的字元個數。
max_size()	字串所容許的最大長度。
data()	型態 const charT* 的指標，指向陣列的第一個元素，此陣列的前 size() 個元素，等於由 *this 控制之字串的對應元素。在 string 物件本身修改後，這指標不一定有效。

成員函數	回傳值
c_str()	型態 const charT* 的指標，指向陣列的第一個元素，此陣列的前 size() 個元素，等於由 *this 控制之字串的對應元素，而且對於 charT 型態，下一個元素是 charT(0) 字元（字串結束符號）。在 string 物件本身修改後，這指標不一定有效。
get_allocator()	用於配置 string 物件之記憶體的 allocator 物件的副本。

要注意 begin()，rend()，data()，和 c_str() 之間的差異。這些都與字串的第一個字元相關，但是方式不同。begin() 和 rend() 成員函數回傳迭代器，這是一般化的指標，如第 16 章 STL 討論中的描述。尤其是 begin() 回傳往前之迭代器的模式，而 rend() 是反向迭代器的副本。兩個都是參考 string 物件管理的實際字串（因為 string 類別使用動態記憶體配置，實際的 string 內容不一定在物件中，所以我們用**管理**（manage）一詞來描述物件和字串之間的關係）。您可以將回傳迭代器的成員函數用在以迭代器為基礎的 STL 演算法上。例如，您可用 STL 的 reverse() 函數反轉字串的內容：

```
string word;
cin >> word;
reverse(word.begin(), word.end());
```

而 data() 和 c_str() 則回傳一般的指標。這回傳的指標，指向儲存字串字元之陣列的第一個元素。這陣列可以是，但不一定是 string 物件管理之原始字串的副本（string 物件使用的內部表示法可以是陣列，但不是一定的）。因為回傳的指標可能指向原始的資料，這是 const，所以不能用來修改資料。而且這指標在字串修改後不一定是有效的，這意思是它們可能指向原始的資料。data() 和 c_str() 之間的差異是，c_str() 所指的陣列以 null 字元（或同義字元）作結束，而 data() 只是確保實際的字串字元是存在的。例如，c_str() 可以當作期待 C 格式字串之函數的引數：

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

同樣的，data() 和 size() 可以用在引數為陣列元素之指標，和表示要處理元素個數的函數中：

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

一種 C++ 實作方式是選擇以動態配置之 C 格式的字串表示 string 物件的字串，並將往前迭代器實作為 char * 指標。此時，這實作可以使 begin()，data()，和 c_str() 都回傳相同的指標。這只是合理的回傳三種不同資料物件的 reference。

對於 basic_string 樣版類別，C++11 有 11 個建構函數，C++98 有 6 個建構函數，與 1 個解構函數：

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const basic_string& str);
basic_string(const basic_string& str, const Allocator&);
basic_string(const basic_string& str, size_type pos,
    size_type n = npos, const Allocator& a = Allocator());
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string&& str, const Allocator&);
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
    const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
~basic_string();
```

注意，大部份的建構函數都有下列含有一個引數的格式：

```
basic_string(const basic_string& str, size_type pos = 0,
size_type n = npos, const Allocator& a = Allocator());
```

C++11 以上列的第二、第三及第四項取代此運算函數。此讓 C++98 版本的編程更有效率。真正新加入的是移動運算函數 (move constructor)，這是帶有 initializer_list 的運算函數。請參閱第 18 章「檢視新的 C++標準」。

注意，大部份的運算函數有一參數，如下列所示：

```
const Allocator& a = Allocator()
```

Allocator 是管理記憶體之 allocator 類別的樣版參數名稱。Allocator() 是此類別的預設建構函數。因此，這建構函數在預設上是使用 allocator 物件的預設版本，但是您可以選擇使用其他版本的 allocator 物件。以下的章節個別地說明這些建構函數。

預設建構函數

下面是預設建構函數的原型：

```
explicit basic_string(const Allocator& a = Allocator());
```

通常您可以採用 allocator 類別的預設引數，並用此建構函數產生空字串：

```
string bean;
wstring theory;
```

在呼叫建構函數之後，下面的關係就成立了：

- data() 成員函數回傳非 null 的指標，這指標可以加上 0。
- size() 回傳 0。
- capacity() 的回傳值未規定。

假設將 data() 的回傳值設給指標 str，然後第一個條件表示 str + 0 是有效的。

使用 C 格式字串的建構函數

這個建構函數是用 C 格式字串初始化 string 物件，更一般的說法是，可以用 charT 的陣列值初始化 charT 特定化：

```
basic_string(const charT * s, const Allocator& a = Allocator( ));
```

為了決定要複製多少個字元，此建構函數對 s 所指的陣列，使用 traits::length() 成員函數（指標 s 不應為 null 指標）。例如，下一敘述使用指定的字元字串，初始 toast 物件。

```
string toast("Here's looking at you, kid.");
```

對於型態 char，traits::length() 成員函數會用 null 字元，來決定需複製的字元數。

在呼叫建構函數之後，下面的關係就成立了：

- data() 成員函數回傳的指標指向陣列 s 之副本的第一個元素。
- size() 成員函數回傳的值等於 traits::length()。
- capacity() 的回傳值至少與 size() 一樣大。

使用部份 C 格式字串的建構函數

這個建構函數是使用部分 C 格式字串初始化 string 物件，更一般的說法是，可以用部分的 charT 陣列值初始化 charT 特定化：

```
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

這建構函數從 s 所指的陣列，複製共 n 個字元至要建構的物件中。注意，若 s 的字元少於 n，不會停止複製。若 n 超過 s 的長度，則建構函數會將字串之後的記憶體內容，視為它們是儲存型態 charT 的資料。

這建構函數要求 s 不是空指標，而且 n < npos（npos 是靜態的類別常數，等於字串最大的可能元素數）。若 n 等於 npos，則建構函數會丟出 out_of_range 異常（因為 n 的型態是 size_type，npos 是最大的 size_type 值，n 不能大於 npos）。在呼叫建構函數之後，下面的關係就成立了：

- data() 成員函數回傳的指標，指向陣列 s 之副本的第一個元素。
- size() 成員函數回傳值 n。
- capacity() 成員函數的回傳值，至少與 size() 一樣大。

使用 Lvalue Reference 的建構函數

複製建構函數看起來像這樣：

```
basic_string(const basic_string& str);
```

此建構函數使用 string 引會初始一新的 string 物件：

```
string mel("I'm ok!");
string ida(mel);
```

此處 ida 會取得 mel 掌管之字串的副本。

```
basic_string(const basic_string& str, const Allocator&);
```

下一個建構函數需要指定一額外的 allocator。

- data() 成員函數回傳的指標，指向從字串 str.data()的第一個元素所配置的陣列。
- size() 成員函數回傳 str.size()的值。
- capacity() 成員函數的回傳值，至少與 size() 一樣大。

下一個建構函數需要您設定一些項目：

```
basic_string(const basic_string& str, size_type pos = 0, size_type n = npos,
    const Allocator& a = Allocator());
```

選擇性的第二個引數 pos，指定要複製原始字串的起始位置：

```
string att("Telephone home.");
string et(att, 4);
```

位置編號從 0 開始，所以位置 4 是 p 字元。因此 et 會初始化為 "Telephone home."。

選擇性的第三個引數 n，指定要複製的最大字元數。所以

```
string att("Telephone home.");
string pt(att, 4, 5);
```

會將 pt 初始化為字串 "Telephone"。但是這建構函數不會超過原始字串的尾端；例如：

```
string pt(att, 4, 200);
```

在複製句點之後就會停止。因此，這建構函數實際上會複製的字元數，等於 n 和 str.size() - pos 兩者中較小者。

這建構函數的條件是 pos <= str.size()，也就是說，要複製的最初位置是在原始字串內，若非如此，則會丟出 out_of_range() 的異常。否則，copy_len 會代表 n 和 str.size() - pos 兩者中較小者，在呼叫建構函數之後，下面的關係就成立了：

- data() 成員函數回傳的指標，指向從字串 str 的 pos 位置開始複製 copy_len 個元素的副本。
- size() 成員函數回傳 copy_len。
- capacity() 成員函數的回傳值至少與 size() 一樣大。

使用 Rvalue Reference 的建構函數 (C++11)

C++11 加入了移動語意到字串類別。在第 18 章有加以描述，此呼叫是額外加入的移動建構函數，它使用 rvalue reference 取代 lvalue reference。

```
basic_string(basic_string&& str) noexcept;
```

當實際引數是暫存的物件時，此建構函數將被呼叫。

```
string one("din"); // C-style string constructor
string two(one); // copy constructor - one is an lvalue
string three(one+two); // move constructor, sum is an rvalue
```

如同第十八章所描述的，string three 經由 operator+()建立物件的關係，而不是複製物件，並且讓原先的物件破壞掉。

第二個 rvalue 建構函數允許您額外加入特定的整 allocator。

```
basic_string(const basic_string&& str, const Allocator&);
```

在呼叫這兩者之一的建構函數之後，下面的關係將成立了：

- data() 成員函數回傳指向一已配置好複製陣列的指標，並經由 str.data()指向此陣列的第一元素。
- size() 成員函數回傳 n。
- capacity() 成員函數的回傳值，至少與 size() 一樣大。

使用 n 份字元的建構函數

這個會使用一個字元 n 個複本之建構函數，產生的 string 物件是由 n 個連續相同，且有 c 值的字元組成：

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

這建構函數的條件是 n < npos。若 n 等於 npos，則建構函數會丟出 out_of_range() 的異常。在呼叫建構函數之後，下面的關係將成立了：

- data() 成員函數回傳的指標，指向具有 n 個元素字串的第一個元素，且均設為 c。
- size() 成員函數回傳 str.size()之值。
- capacity() 成員函數的回傳值，至少與 size() 一樣大。

使用區間的建構函數

最後一個會使用區間的建構函數，使用以 STL 格式之迭代器定義的區間：

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
                const Allocator& a = Allocator());
```

迭代器 begin 指向要複製之原始資料的開始處，end 指向要複製之最後位置的後面一個。

您可以將此格式用在陣列，字串或是 STL 收納器上；

```
char cole[40] = "Old King Cole was a merry old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
    input.push_back(ch);
string str_input(input.begin(), input.end());
```

第一個用法，InputIterator 會求得型態 const char *。在第二個用法中，InputIterator 會求得型態 vector<char>::iterator。

在呼叫建構函數之後，下面的關係就成立了：

- data() 成員函數回傳的指標，指向從區間 [begin, end) 複製元素所形成之字串的第一個元素。
- size() 成員函數回傳 begin 和 end 之間的距離（這距離的測度單位等於迭代器提領時其內含資料型態的大小）。
- capacity() 成員函數的回傳值，至少與 size() 一樣大。

使用初始列的建構函數 (C++11)

這一建構函數取一 initializer_list<charT> 當做參數:

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

您可用它與大括號內的字元串列

```
string slow({'s', 'n', 'a', 'i', 'l'});
```

這不是最方便的初始字串的方法，但可使其字串介面和 STL 相似。initializer_list 類別有 begin() 和 end() 成員函數，使用此建構函數和使用區間的建構函數具有相同的效果。

```
basic_string(il.begin(), il.end(), a);
```

記憶體的其他功能

有數個處理記憶體的成員函數，例如，清除記憶體的內容，改變字串大小，調整字串容量。表 F.2 是一些與記憶體相關的成員函數。

表 F.2 與記憶體相關的成員函數

成員函數	處理結果
void resize(size_type n)	若 n > npos，則丟出 out_of_range 異常。其他則將字串大小改成 n，若 n < size()，則截去字串；若 n > size()，則補上 charT(0) 字元。
void resize(size_type n, charT c)	若 n > npos，則丟出 out_of_range 異常。其他，則將字串大小改成 n，若 n < size()，則截去字串；若 n > size()，則補上字元 c。

成員函數	處理結果
void reserve(size_type res_arg = 0)	將 capacity() 指定為大於等於 res_arg。因為這會重新配置字串，而使之前字串的 reference，迭代器，和指標都變成無效。
void shring_to_fit()	非繫結性的要求減少 capacity() 為 size()。
void clear()	移除字串的所有字元。
bool empty() const	若 size() == 0 則回傳 true。

字串存取

存取個別字元有 4 個成員函數，兩個使用 [] 運算子，兩個用 at() 成員函數：

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

第一個 operator[]() 成員函數是用陣列表示法，存取字串的個別元素，這可用來讀取或修改值。第二個 operator[]() 可以處理 const 物件，只能用來讀取值：

```
string word("tack");
cout << word[0];    // display the t
word[3] = 't';      // overwrite the k with a t
const ward("garlic");
cout << ward[2];    // display the r
```

成員函數 at() 提供類似的存取，不一樣的地方是索引值，以函數引數的方式傳入：

```
string word("tack");
cout << word.at(0);    // display the t
```

差異處是（除了語法外）成員函數 at() 提供邊界檢查，若 pos >= size() 就會丟出 out_of_range 異常。注意 pos 的型態是 size_type，這是無號型態，因此 pos 不可能是負值。operator[]() 成員函數不作邊界檢查，所以若 pos >= size() 其結果並未定義，除了 const 版本會在 pos == size() 時，回傳空 null 字元。

因此，您可以在安全性（使用 at() 並檢查異常）和執行速度（使用陣列表示法）之間作一選擇。

還有一個函數會回傳新字串，它是原始字串的子字串：

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

它回傳的字串是從位置 pos 開始，複製字串的 n 個字元或是至字串結束，看哪一種情況先發生而定。例如，下面的敘述會將 pet 初始化為 "donkey"：

```
string message("Maybe the donkey will learn to sing.");
string pet(message.substr(10, 6));
```

C++11 加了以下四個擷取成員函數：

```
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
```

front() 成員函數擷取字串的第一個元素。動作有如 operator[](0)。而 back() 成員函數擷取字串最後一個元素，動作有如 operator[](size() - 1)。

基本指定

C++11 有五個多載的指定成員函數，而原先的 C++98 只有前三個：

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(const basic_string&& str) nonexcept; //C++11
basic_string& operator=(initializer_list<chart>);            //C++11
```

第一個是將一個 string 物件指定給另一個，第二個是將 C-格式字串指定給 string 物件，第三個是指定單一字元至 string 物件。第四個是使用搬移語意法，指定 rvalue 的 string 物件指定給一 string 物件。第五個允許使用初始化串列的指定方式。因此，下面都是可能的操作：

```
string name("George Wash");
string pres, veep, source, join, awkward;
pres = name;
veep = "Road Runner";
source = 'X';
join = name + source;
awkard = {'C', 'l', 'o', 'u', 's', 'e', 'a', 'u'};
```

字串搜尋

string 類別提供 6 個搜尋函數，每個都有 4 個原型。以下會簡短描述之。

find() 家族

下面是 C++11 所提供 find() 的原型：

```
size_type find (const basic_string& str, size_type pos = 0) const;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (charT c, size_type pos = 0) const;
```

第一個成員回傳子字串 str 在呼叫物件中，第一次出現的位置，每次搜尋都從位置 pos 開始。若找不到子字串，這成員函數回傳 npos。

以下的程式碼會在一個較長的字串中，尋找子字串 "hat" 的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);           // sets loc1 to 1
size_type loc2 = longer.find(shorter, loc1 + 1); // sets loc2 to 16
```

因為第二個搜尋從位置 2 開始（That 的 a），它找到 hat 第一次出現之處，是靠近字串結束處。要檢查是否失敗，可使用 string::npos 值：

```
if (loc1 == string::npos)
    cout << "Not found\n";
```

第二個成員函數執行同樣的工作，不一樣的地方是子字串，是用字元陣列，而非 string 物件：

```
size_type loc3 = longer.find("is");              // sets loc3 to 5
```

第三個成員函數的功能與第二個相同，差別是它只用字串 s 的前 n 個字元。這結果與使用 basic_string(const charT* s, size_type n) 建構函數，並將結果物件當作第一個 find() 形式的 string 引數相同。例如，下面的敘述會搜尋子字串 "fun"：

```
size_type loc4 = longer.find("funds", 3);        // sets loc4 to 10
```

第四個成員函數和第一個相同，差別是它的子字串是用單一字元，而不是 string 物件：

```
size_type loc5 = longer.find('a');               // sets loc5 to 2
```

rfind() 家族

下面是 rfind() 成員函數的原型：

```
size_type rfind(const basic_string& str,
                size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const noexcept;
```

這些成員函數的運作與其對比的 find() 類似，不一樣的地方是，它們從位置 pos 或是之前，找出字串或字元最後一次出現的位置。若找不到子字串，則此成員函數回傳 npos。

以下的程式碼會在一個較長的字串中，尋找子字串 "hat" 的位置，而尋找的位置是由長字串的最後面開始：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);           // sets loc1 to 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // sets loc2 to 1
```

find_first_of() 家族

下面是 find_first_of() 成員函數的原型：

```
size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const noexcept;
```

這些成員函數與其對應的 find() 類似，但不是找到整個子字串的完全匹配，而是找尋子字串中任一字元的第一次匹配的地方。

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter); // sets loc1 to 10
size_type loc2 = longer.find_first_of("fat"); // sets loc2 to 2
```

fluke 的 5 個字元中，任一個字元的第一次出現之處，是 longer 物件之 funny 的 f。fat 的任一字元第一次出現在 longer 物件之 That 的 a。

find_last_of() 家族

下面是 find_last_of() 成員函數的原型：

```
size_type find_last_of (const basic_string& str,
                        size_type pos = npos) const noexcept;
size_type find_last_of (const charT* s, size_type pos, size_type n) const;
size_type find_last_of (const charT* s, size_type pos = npos) const;
size_type find_last_of (charT c, size_type pos = npos) const noexcept;
```

這些成員函數與其對應的 rfind() 類似，但不是找到整個子字串的完全匹配，而是找尋子字串中任一字元的最後一次匹配的地方。

以下的程式碼會在一個較長的字串中，尋找最後一個有出現 "hat" 之任一字元的位置：

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter);  // sets loc1 to 18
size_type loc2 = longer.find_last_of("any");    // sets loc2 to 17
```

hat 的任一字元最後一次出現在 longer 中，是 hat 的 t。any 的任一字元最後一次出現之處是 longer 之 hat 的 a。

find_first_not_of() 家族

find_first_not_of() 成員函數有這些原型：

```
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const;
size_type find_first_not_of(const charT* s, size_type pos,
                             size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

這些成員函數的運作就像是 find_first_of() 成員函數，差別是它們找尋不屬於子字串的任何字元第一次出現的地方。

以下的程式碼會在一個較長的字串中，尋找最前面不在 "This" 之任一字元的位置：

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter);    // sets loc1 to 2
size_type loc2 = longer.find_first_not_of("Thatch");   // sets loc2 to 4
```

longer 物件中，That 的 a 是沒有出現在 This 中的第一個字元。longer 字串中，第一個空白是第一個不在 Thatch 的字元。

find_last_not_of() 家族

find_last_not_of() 成員函數有這些原型：

```
size_type find_last_not_of (const basic_string& str,
                            size_type pos = npos) const;
size_type find_last_not_of (const charT* s, size_type pos,
                            size_type n) const;
size_type find_last_not_of (const charT* s, size_type pos = npos) const;
size_type find_last_not_of (charT c, size_type pos = npos) const;
```

這些成員函數的運作就像是 find_last_of() 成員函數，差別是它們找尋不屬於子字串的任何字元最後一次出現的地方。

以下的程式碼會在一個較長的字串中，由尋找最後一個不在 "That." 之任一字元的位置：

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter);     // sets loc1 to 15
size_type loc2 = longer.find_last_not_of(shorter, 10); // sets loc2 to 10
```

在 longer 中，最後一個空白是最後一個不在 shorter 中的字元。在 longer 字串中，f 是在位置 10，沒有出現在 shorter 中的最後一個字元。

比較成員函數和函數

string 類別提供成員函數和函數，用來比較兩個字串。首先是這些成員函數的原型：

```
int compare(const basic_string& str) const;

int compare(size_type pos1, size_type n1,
            const basic_string& str) const;
int compare(size_type pos1, size_type n1,
            const basic_string& str,
            size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1,
            const charT* s, size_type n2 = npos) const;
```

這些成員函數利用 traits::compare()，這是定義用於字串之特定字元型態的成員函數。第一個成員函數是根據 traits::compare() 所提供的排序方法，若第一個字串在第二個字串之前，則回傳值小於 0；若兩個字串相等，則回傳 0；若第一個字串在第二個字串之後，則回傳值大於 0。若兩個字串至較短字串結束處都完全相同，則較短的字串會在較長的字串之前。

以下的範例將字串 s1 與 s3 作比較，以及將 s1 與 s2 作比較：

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3); // a13 is < 0
int a12 = s1.compare(s2); // a12 is> 0
```

第二個成員函數類似於第一個，差別是它是從第一個字串的位置 pos1 開始比較 n1 個字元。

以下的範例將對字串 s1 前四個字元與 s2 作比較：

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2); // a2 is 0
```

第三個成員函數類似於第一個，差別是它從第一個字串的位置 pos1 開始，取 n1 個字元與從第二個字串的位置 pos2 開始，取 n2 個字元來作比較。例如，下面的敘述是比較 stout 中的 out 和 about 中的 out：

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3);  // a3 is 0
```

第四個成員函數類似於第一個，不一樣的地方是，它的第二個字串是用字串陣列，而非 string 物件。

第五個成員函數類似於第三個，不一樣的地方是，它的第二個字串是用字串陣列，而非 string 物件。

非成員的比較函數是多載的關係運算子：

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

每個運算子都是多載，所以它們可以比較 string 物件與 string 物件，string 物件與字串陣列，字串陣列與 string 物件。它們是以 compare() 成員函數定義之，所以在字串比較的表示法上，它們提供更便利的方式。

字串修飾元

string 類別提供數個成員函數可以修改字串。大部分有許多的多載版本，所以可以用於 string 物件，字串陣列，個別字元，和迭代器區間。

附加和新增的成員函數

您可以用多載的 += 運算子或是 append() 成員函數，將一個字串附加至另一個字串。若結果大於最大的字串大小，則會丟出 length_error 異常。+= 運算子可以附加 string 物件，字串陣列，或是個別字元至另一個字串。

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

append() 也可以附加 string 物件，字串陣列，或是個別字元至另一個字串。此外，還可以附加部分的 string 物件，只要指定起始位置及要附加的字元數，或是指定區間即可。您可以指定字串的字元數而附加部分的字串。附加字元的版本可以指定要複製該字元多少次。以下是各種 append() 成員函數的原型：

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c);   // append n copies of c
void push_back(chart c);                      // append 1 copy of c
```

一些範例如下：

```
string test("The");
test.append("ory"); // test is "Theory"
test.append(3,'!'); // test is "Theory!!!"
```

多載的函數 operator+() 可作字串連結。這多載的函數不會修改字串，相反的是產生新字串，其內容是這第二個字串附加至第一個字串。加法函數不是成員函數，而且您可以將 string 物件加至 string 物件，字串陣列加至 string 物件，string 物件加至字串陣列，字元加至 string 物件，以及 string 物件加至字元。下面是一些範例：

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce";  // st3 is "reduce"
string st4 = 't' + st2;    // st4 is "train"
string st5 = st1 + st2;    // st5 is "redrain"
```

再談指定成員函數

除了基本的指定運算子之外，string 類別提供 assign() 成員函數，這可以指定整個字串，或是部分字串，或是一串相同字元至 string 物件。以下是各種 assign() 成員函數的原型：

```
basic_string& assign(const basic_string& str);
basic string& assign(basic_string&& str) noexcept; // C++11
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
```

```
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c); // assign n copies of c
template<class InputIterator>
basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>); // C++11
```

一些範例如下：

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5);   // test is "et tu"
test.assign(6, '#");        // test is "######"
```

以 rvalue reference（C++11 新加入的）的 assign()成員函數，允許使用移動語意方式。第二個 assign()成員函數允許將 initializer_list 指定給 string 字串物件。

插入成員函數

insert() 成員函數可以在 string 物件中插入 string 物件，字串陣列，字元，或是數個字元。這個成員函數類似於 append()，差別是它們有另一個引數表示要在何處插入新內容。這個引數可以是位置或是迭代器。此內容會置於插入點之前。這成員函數的數個版本，會回傳結果字串的 reference。若 pos1 超過目的字串的尾端，或是 pos2 超過要插入之字串的尾端，則此成員函數會丟出 out_of_range 異常。若結果字串大於最大的長度，則會丟出 length_error 異常。以下是各種 insert() 成員函數的原型：

```
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  void insert(iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>); // C++11
```

例如，下面的程式碼將字串 "former " 插在 "The banker." 的 b 之前：

```
string st3("The banker.");
st3.insert(4, "former ");
```

接下來的程式碼，將字串 " waltzed"（不會包括！，這是第 9 個字元）加在 "The former banker." 的句點之前：

```
st3.insert(st3.size() - 1, " waltzed!", 8);
```

清除成員函數

erase() 成員函數會移除字串中的字元。原型如下：

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator position);
iterator erase(const_iterator first, iterator last);
void pop_back();
```

第一種格式從位置 pos 開始移除之後的 n 個字元，或是至字串尾端，視何者先發生而定。第二個格式移除迭代器 position 所指的單一字元，並回傳指向下一個元素的迭代器，或是若沒有其他元素，則回傳 end()。第三個移除區間 [first, last) 中的字元；也就是包括 first 至 last，但不包括 last。這成員函數回傳最後被清除之元素的後面元素的迭代器。

取代成員函數

各種 replace() 成員函數會標示要被取代的字串以及取代物。要被取代的部分用起始位置和字元數表示，或是用迭代器區間表示。取代物可以是 string 物件，字串陣列，或是特定的字元重複數次。取代物若為 string 物件和陣列，則可以再指定特定的部分，使用位置和字元數，或是迭代器區間表示。以下是各種 replace() 成員函數的原型：

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const charT* s, size_type n);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                      size_type n, charT c);
template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2,
                        InputIterator j1, InputIterator j2);
basic_string& replace(const_iteraor i1, const_iteator i2,
initializer)list<charT> il);
```

範例如下：

```
string test("Take a right turn at Main Street.");
test.replace(7,5,"left"); // replace right with left
```

請注意，我們可以 replace() 中，使用 find() 來尋找位置：

```
string s1 = "old";
string s2 = "mature";
string s3 = "The old man and the sea";
string::size_type pos = s3.find(s1);
if (pos != string::npos)
    s3.replace(pos, s1.size(), s2);
```

這個範例會以 mature 取代 old。

其它的修改成員函數：copy() 和 swap()

copy() 成員函數會複製 string 物件，或是部分字串至目的字串陣列：

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

此處 s 指向目的陣列，n 表示要複製的字元數，而 pos 表示 string 物件要複製的開始處。複製會處理 n 個字元或是直到 string 物件的最後一個字元，看哪一個先發生而定。這函數回傳複製的字元數。這成員函數不會附加 null 字元，而且由程式設計師檢查陣列是否夠大可以儲存此副本。

copy() 成員函數不會附加 null 字元，也不會檢查目的地陣列是否夠大。

成員函數 swap() 以固定時間的演算法，交換兩個 string 物件的內容：

```
void swap(basic_string& str);
```

輸出和輸入

string 類別多載 << 運算子以顯示 string 物件。它回傳 istream 物件的 reference，所以可以串接輸出敘述：

```
string claim("The string class has many features.");
    cout << claim << endl;
```

string 類別多載 >> 運算子，所以可以將輸入讀入字串：

```
string who;
cin >> who;
```

輸入會因檔案終點，讀入字串所接受的最大字元數，或是遇到正常空白字元而結束（正常空白的定義會視字元集和 charT 所代表的型態而定）。

有兩個 getline() 函數。第一個的原型如下：

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
              basic_string<charT,traits,Allocator>& str, charT delim);
```

它從輸入串流 is 讀入字元至字串 str，直到遇見分隔字元 delim，或是達到字串的最大長度，或是遇到檔案終點。這會讀入 delim 字元（從輸入串流中移除），但是不會儲存。第二個版本沒有第三個引數，並用換行字元取代 delim：

```
string str1, str2;
getline(cin, str1); // read to end-of-line
getline(cin, str2, '.'); // read to period
```