國立屏東大學 資訊工程學系 C++程式設計入門教材

4. 變數、常數與資料型態

C++語言使用變數（Variable）來保存程式相關的資料內容，例如使用者所輸入的資料或是執行過程中暫時或最終的運算結果。一個變數的內容可於程式執行時視需要加以改變，但其所屬型態不能變更。C++語言也提供常數（Constant）來存放特定的資料內容，但常數的內容一經定義就不可以被改變。我們已經在前一章中，使用變數來進行了幾個簡單的程式開發，例如在BMI計算程式中的weight與height，以及在門號違約金計算程式中的contractDays與subsidy等。本章將進一步為讀者詳細說明變數與常數的意義，以及它們在C++語言中的使用方式，具體的內容包含資料型態、變數與常數的宣告、初始值的設定、以及輸入與輸出等主題。

4.1 變數

變數（Variable）指是在程式執行時，用於儲存特定資料型態（Data Type）的「值（Value）」的記憶體空間。此處的「值」是指資料內容，可以在執行過程中視需要改變其內容，但其所屬的「資料型態」不可改變。在我們正式開始介紹變數前，讓我們先回顧一下，上一章IPO程式設計模型的BMI計算程式與門號違約金計算程式範例，讓我們學到了哪些關於變數的知識與概念：

- 變數其實就是一塊記憶體空間，可以在程式執行期間保存特定「資料型態」的「值（Value）」（也就是「資料內容」）。
- 每個變數都必須有一個用以識別的變數名稱（Variable Name）以便在程式碼中用以存取其值。
- 依據程式設計的需要，變數的值在執行期間可以被改變，但其資料型態不可改變。
- C++語言規定所有變數在初次使用前，都必須進行變數宣告（Variable Declaration）— 包含了其變數名稱以及所屬資料型態的宣告。

本節後續將進一步提供更完整的介紹，包含變數宣告、變數的命名規則與記憶體空間等相關主題。

4.1.1 變數宣告（Variable Declaration）

C++語言規定所有變數在初次使用前，都必須先進行「變數宣告（Variable Declaration）」以便在程式執行時，依據其宣告的內容完成其所需的記憶體空間配置。因此，您必須先學會如何使用C++語言來進行變數的宣告，才能夠在程式中使用變數；能夠在程式中使用變數，也才能夠讓程式完成您想要執行的任務。現在，讓我們來學習如何在C++語言中完成變數的宣告？

與C語言不同的是，C++語言允許我們在任意位置進行變數宣告，只要在第一次使用該變數前有完成宣告即可！

變數宣告敘述的語法定義

C++語言使用變數宣告敘述（Variable Declaration Statement）來完成變數宣告，其語法如下：

變數宣告敘述（Variable Declaration Statement）語法定義

資料型態 變數名稱 [=初始值]?[,變數名稱 [=初始值]?]*;

資訊補給站：語法定義符號

在上面的語法定義中，方括號[ ]代表了選擇性（意即可有可無的部份）的語法單元，其後若接續星號 * 則表示該語法單元可使用0次或多次；問號 ? 表示使用0次或1次（也就是可以省略，但至多使用一次）。另外還有此處還未使用到的加號 +，代表的是該語法單元可以使用1次或多次（也就是至少必須使用一次）。後續本書將繼續使用此種表示法做為語法的說明。

在此處的變數宣告敘述語法定義裡，包含了以下三項內容定義：

- 變數名稱（Variable Name）：用以識別的名稱。關於變數的命名規則，將在3-1-2節提供詳細的說明。
- 資料型態（Data Type）：用以定義變數值的範圍，例如被宣告為整數的變數，其值不允許包含小數。關於C++語言所提供的資料型態，除了已經在第2章中介紹過的float與int，其它更完整的介紹請參考本章3-2節。
- 初始值（Initial Value）：用以定義變數值的初始內容。我們將在3-1-3節提供詳細的說明。

注意：變數宣告敘述必須使用分號；結尾

在開始說明如何使用變數宣告敘述前，我們要先提醒讀者所有的「宣告敘述（Statement）」都必須使用分號「;」做為結尾，變數宣告敘述也不例外。請不要忘記在每一行變數宣告敘述後，加上一個分號。

本章後續將依據變數宣告敘述的語法，為讀者詳細說明如何進行變數宣告，但在本小節中將先把選擇性語法單元的部份加以忽略，先行就精簡版的變數宣告敘述加以說明。

精簡變數宣告敘述的語法定義

精簡版的語法定義，略去了原語法中所有可以省略的部份（意即將所有使用方括號包裹的語法單元都已經被省略），是最簡單的變數宣告敘述，只要註明變數的名稱及其資料型態，再加上一個分號「;」就完成了。請參考以下的定義：

精簡版變數宣告敘述語法定義（略去選擇性語法單元）

資料型態 變數名稱;

下列的例子是取自於上一章IPO程式設計模型的BMI計算程式，它們都是符合這種精簡的變數宣告敘述 — 每一行程式敘述各自宣告了一個變數：

```
float weight;  // 體重
float height;  // 身高
float BMI;   // 身體質量指數
```

這三個敘述皆符合精簡版的變數宣告敘述語法定義，都是以一個「資料型態」開頭，其後再接一個「變數名稱」，最後以分號結尾。

程式設計小技巧：為變數宣告提供註解 另外，我們還在這三個宣告敘述以分號結束後，在後面又加上了以「//」開頭的註解，來補充說明這些變數的意義。關於註解的部份並不屬於變數宣告的語法規範，但卻是筆者建議您可以維持這樣的習慣 — 儘可能在宣告變數時，以註解提供變數的補充說明。如此一來，您所撰寫的程式將比較容易維護。

分隔語法單元的「白色空白（Whitespace）」

還有一點要特別注意的是，在C++語言的語法規則中，我們必須使用一個或多個「空白鍵、Tab鍵」、「Enter鍵」或混合使用它們，來將各個語法單元加以分隔。我們將這種分隔方式稱為「白色空白（Whitespace）」，又將這些用以分隔的鍵稱為「白色空白字元（Whitespace Character）」例如在上述的例子中，我們使用了一個「空白鍵」將「float」與「weight」加以分隔[1)]。依照這樣的規定，以下的寫法都是正確的宣告：

```
float weight;  // 使用一個空白鍵分隔
float   height;  // 使用一個Tab鍵分隔
float
BMI;   // 使用一個Enter鍵分隔
```

當然，如果您想要使用更多個分隔字元也是可以的，請參考以下的例子：

```
float      weight;  // 使用多個空白鍵分隔
float       height;  // 使用兩個Tab鍵分隔
float


          BMI;   // 混合使用Enter鍵與空白鍵分隔
```

縮排（Indentation）：程式碼排版風格

白色空白不但可以用在語法單元間的分隔，也常常應用在敘述中的第一個語法單元之前。例如以下的例子：

```
    float weight;  // 在float前使用多個空白鍵
  float height;    // 在float前使用一個Tab鍵
```

相信讀者一定會有一個疑問：「為何要在前面在白色空白呢」？其實這就是所謂的「縮排（Indentation）」又稱為「程式碼排版風格（Coding Style）」透過在每行程式敘述前加入適當的白色空白，將有助於程式碼的閱讀，並可反映出程式碼的結構，是專業程式設計師必備的技能之一[2)]。以下的兩個例子都是在第2章中的BMI計算程式，但使用不同的縮排方法：

全部切齊

```
#include <iostream>
using namespace std;
int main()
{
float weight;
float height;
float BMI;
cout << "請輸入您的體重(公斤):";
cin >> weight;
cout << "請輸入您的身高(公尺):";
cin >> height;
BMI = weight / ( height * height
);
cout << "BMI值為" << BMI << endl;
return 0;
}
```

適當地縮排

```
#include <iostream>
using namespace std;
int main()
{
float weight;
float height;
float BMI;

cout << "請輸入您的體重(公斤):";
cin >> weight;
cout << "請輸入您的身高(公尺):";
cin >> height;

BMI = weight / ( height * height
);

cout << " BMI值為" << BMI << endl;
return 0;
}
```

其中左側的程式碼完全沒有在每行敘述前加入白色空白，右側的部份則適當地使用白色空白將程式碼對齊，不但在閱讀上比較清爽，同時在main()函式內的程式碼透過內縮對齊也反映了它們是屬於函式內部的程式結構。

其實C++語言只規定敘述要以分號做為結尾，但並沒有規定要寫在獨立的一行，把多個敘述寫在同一行，或是將一個敘述分寫在多行裡都是可以的。雖然在語法上允許我們這麼做，但是筆者並不鼓勵您將多個敘述寫在同一行，或是將一個敘述分散到多行（除非一行寫不下的時候），因為這樣一來會將程式碼的結構破壞殆盡，例如以下的例子：

```
#include <iostream>
using namespace std;
int main(){ float weight; float height; float BMI;
cout << "請輸入您的體重(公斤): "; cin >> weight; cout <<
"請輸入您的身高(公尺): ";cin >> height; BMI = weight
/ ( height * height ); cout << "您的BMI值為"<<
BMI << endl;return 0;
}
```

您還認得出來這是哪個程式嗎？沒錯，它仍然是我們熟悉的那個BMI計算程式，但是看起來實在不怎麼友善！事實上，在我們進行程式碼的編譯時，編譯器會先將程式碼中的白色空白移除，然後才進行編譯的動作。因此上面這個排版很不自然的程式仍然可以正常的執行！只是您真的能接受這種雜亂無章的編排方式嗎？這種編排方式，不但其他人看不懂您所寫的程式碼，可能連您自己都看不太懂！

其實，程式碼縮排方式並沒有硬性的規定，但是儘量將每個敘述寫在獨立的一行，並適當地使用白色空白

來將程式碼對齊，是比較好的程式撰寫習慣。

使用一行敘述宣告多個相同型態的變數

除了前述的精簡版變數宣告敘述外，依據原語法定義，多個相同型態的變數還可以寫在同一個宣告敘述中，請參考以下的語法定義：

精簡版變數宣告敘述語法定義（僅略去部份的選擇性語法單元）

資料型態 變數名稱 [,變數名稱]*;

上述的語法定義僅將部份的選擇性語法單元加以省略（也就是僅省略了原語法中關於初始值的部份）。與精簡版的變數宣告敘述相同，我們仍然使用「資料型態」作為開頭，並在其後再接一個「變數名稱」；但不同的是，您可以在變數名稱的後面，視需要再宣告0個或多個變數名稱。具體來說，在語法單元 [,變數名稱]* 中的 * 表示 [,變數名稱] 可以使用0次或多次。用比較白話的說法，此語法定義也可以解讀為「可以使用單一的變數宣告敘述，同時宣告多個變數，不過必須在連續的兩個變數名稱中，使用逗號 , 加以分隔」。以下的例子同時宣告了三個float型態的變數，分別是weight、height與BMI。

```
float weight, height, BMI;
```

以語法來看，其開頭處的float就是語法定義中的「資料型態」部份，至於weight就是後面所接的「變數名稱」，但我們在其後再重覆使用了兩次 [,變數名稱]* 語法規則，也就是「,height」與「,BMI」最後以分號做為此宣告敘述的結尾。

4.1.2 變數命名規則

變數名稱（VariableName）又被稱為識別字（Identifier）是用以區分在程式中不同的變數。為了識別起見，每一個變數必須擁有獨一無二的變數名稱，其命名規則如下：

1. 只能使用英文大小寫字母、數字與底線（Underscore） _
2. 不能使用數字開頭；
3. 不能與C++語言的關鍵字（keyword）相同。

前兩項規則相當容易理解，至於第三項中所謂的關鍵字（Keyword）是在程式語言中具有（事先賦予的）特定意義的文字串組合。由於每個關鍵字都具有事先定義好的意義與用途，因此我們在宣告變數時，變數名稱不能與任何一個關鍵字相同。

<nowiki>C++</nowiki>的關鍵字

以ANSI/ISO C++（或稱為C++89） 為例，共有以下74個關鍵字：

and	and_eq	asm	auto	bitand	bitor	bool	break	case	catch
char	class	compl	const	const-cast	continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export	extern	false	float	for	friend
goto	if	inline	int	long	mutable	namespace	new	not	not_eq

operator	or	or_eq	private	protected	public	register	reinterpret-cast	return	short
signed	sizeof	static	static_cast	struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned	using	virtual	void	volatile
wchar_t	while	xor	xor_eq						

Tab. 1: ANSI/ISO <nowiki

更新版本的C++□則有以下84個保留字：

alignas	alignof	and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch	char	char16_t
char32_t	class	compl	const	constexpr	const_cast	continue
decltype	default	delete	do	double	dynamic_cast	else
enum	explicit	export	extern	false	float	for
friend	goto	if	inline	int	long	mutable
namespace	new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_assert	static_cast
struct	switch	template	this	thread_local	throw	true
try	typedef	typeid	typename	union	unsigned	
using	virtual	void	volatile	wchar_t	while	xor
xor_eq						

Tab. 2: ANSI/ISO <nowiki

正確與錯誤的變數命名範例

依據前述變數命名的三項規則，以下的變數名稱使用了不在規則中允許的字元或符號（僅能使用英文大小寫字母、數字與底線），當然都是錯誤的命名：

```
int cert#, someone@taipei;  //使用了不允許的特殊字元
int average-score;          //使用了不允許的連字號(也就是減號)
int miles/per-second;       //使用了不允許的斜線字元與連字號
float ratio!, question?       //使用了不允許的驚嘆號與問號
```

至於數字在使用上也必須注意（不可以用於變數名稱的開頭），以下是錯誤的例子：

```
float 386PC, 486PC;        //使用了數字開頭
int 1stScore, 2ndScore;     //使用了數字開頭
```

除此之外，還要注意不要使用C++的關鍵字做為變數名稱，例如以下的例子都是錯誤的命名：

```
int namespace;        //使用了<nowiki>C++</nowiki>的關鍵字namespace
float auto;            //使用了<nowiki>C++</nowiki>的關鍵字auto
```

看完了上述的錯誤命名的例子之後，讓我們看看一些正確的變數命名範例。以下的變數名稱僅使用了大小

寫英文字母、數字與底線所組成，同時也都沒有以數字開頭，或是使用了C++的關鍵字，所以當然都是正確的：

```
int A, B, C;        // 使用大寫英文字母所組成的變數名稱
int x, y, z;         // 使用小寫英文字母所組成的變數名稱
int Wig, xYz;      // 混合使用大小寫英文字母所組成的變數名稱
int H224, i386;     // 混合使用英文字母與數字所組成的變數名稱
float w_1, w_2;    // 混合英文、數字與底線所組成的變數名稱
```

雖然依據變數的命名規則，這些變數名稱都是正確的，但都不是很好的命名選擇，因為這些變數名稱並不具備任何意義，無法從中得到關於變數的用途或意義的提示。雖然變數的命名必須要滿足其命名規則（因為一定要滿足，不然不能通過編譯），但是更重要是為變數賦與「有意義」的名稱；換句話說，變數名稱的做用不單是用以識別不同的變數，更重要的是要能夠傳達該變數在程式中的用途或意義。為了做到這一點，讀者們應該適當地使用大小寫的英文字母、數字與底線（underscore）為變數命名，以增進程式碼的可讀性。

可讀性（Readability）

所謂的可讀性（Readability）係指程式碼容易被理解的程度，可讀性高的程式碼，閱讀起來就像行雲流水，頃刻之間就可以完全瞭解程式的內容；相反地，可讀性低的程式碼，不但讓人難以理解，有時候甚至是原始的程式創作者自己也無法理解程式的內容。為了增進程式的可讀性，我們建議讀者視變數在程式中的用途，選擇具有意義的英文詞彙為變數命名。為了讓程式碼的可讀性提升，有時我們甚至會使用一個以上的英文單字為變數命名，此時可以適當地調整大小寫或加上底線，例如下面是正確且具有意義的變數名稱：

```
float weight, height, BMI;  // 分別代表體重、身高與BMI值的變數

// 我們可以使用底線連接多個英文單字，以形成更具意義的變數名稱
int student_id;       // 使用底線連接student與id代表學生的學號
float student_score;  // 使用底線連接student與score代表學生的成績

// 我們也可以適當的使用大小寫英文字母，取代使用底線來連接多個單字
int studentID;       // 使用大小寫來區分連接起來的student與ID
float studentScore;  // 使用大小寫來區分連接起來的student與Score
```

除此之外，也可以適當地使用數字的諧音，來精簡表示特定的英文或其它含義。請參考下面的例子：

```
// 使用數字4來代替英文的for
float interest4loan;   // 此變數名稱表示interest for loan，意即貸款的利率
float interest4saving; // 此變數名稱表示interest for saving，意即存款的利率
float discount4group; // 此變數名稱表示discount 4 group，意即團體的折扣
float discount4kids;  // 此變數名稱表示discount 4 kids，意即孩童的折扣

// 使用數字2來代替英文的to
int time2destination;  // 此變數名稱表示time to destination，
```

```
        // 意即到達目的地的時間
int amount2pay;      // 此變數名稱表示amount to pay，意即應支付的金額
```

大小寫敏感（Case Sensitive）

除了前述的三項變數命名的規則之外，還必須特別注意的是，C++語言是對大小寫敏感（Case Sensitive）的程式語言，意即大小寫會被視為不同的字元，因此以下的程式碼所宣告的8個變數名稱都是正確的，而且會被C++語言視為是「不相同」的變數：

```
int day, Day, dAy, daY, DAy, DaY, dAY, DAY;
```

儘管您應該不會像這樣宣告一些非常容易混淆的變數名稱，但仍有可能在程式中發生一些錯誤，請參考以下的例子：

```
float Weight; // 宣告一個float型態的變數Weight

cin >> weight;  // 使用cin取得使用者輸入時，誤將變數名稱寫為「w」eight
```

在上面的程式碼片段中，出現了「寫錯變數名稱」的錯誤！不小心將「Weight」寫成了「weight」與！由於C++語言對於大小寫敏感的關係，在程式碼中的「Weight」與「weight」將會被視為是兩個不同的變數。對於編譯器而言，「Weight」是正確的變數名稱（有事先使用變數宣告敘述宣告），但是用以儲存使用者輸入的「Weight」變數，因為沒有事先的宣告，所以是不正確的使用（無法通過編譯）。像這樣的錯誤其實很容易發生，例如在程式中不小心將「x」寫成「X」，或是把「v」寫成「V」等。

要避免這樣的問題，除了小心謹慎地使用變數名稱外，最好的方式是維持一定的變數命名習慣。舉例來說，如果您固定全部使用小寫字母為變數命名，那麼在使用變數時您自然也會全部使用小寫字母，那麼將「weight」誤寫成「Weight」的機會就會大幅地降地。目前業界已經有一些通用的變數命名方式，我們將其稱之為「命名慣例（Naming Convention）」，依據這些慣例來為變數命名除了有助於提升程式的可讀性外，也能夠減少犯錯機會。

命名慣例（Naming Convention）

目前有一些用於變數命名的規則可參考，例如著名的「駝峰式命名法（CamelCase）」[2]與「匈牙利命名法（Hungarian notation）」[3]等方法。駝峰式命名法是當前主流的變數命名方法，其命名方法是直接使用英文為變數命名，其名稱即為變數的意義或用途，因此具備良好的可讀性。

採用駝峰式命名法命名方法時，直接使用英文為變數命名；若使用到兩個或兩個以上的英文單字時，每個英文單字除首字母外一律以小寫表示，且單字與單字間直接連接（不須空白），但從第二個單字開始，每個單字的首字母必須使用大寫。至於第一個單字的首字母，則依其使用大寫或小寫字母，可將駝峰式命名法再細分為「大寫式駝峰式命名法（UpperCamelCase）」與「小寫式駝峰式命名法（lowerCamelCase）」兩類。例如以下的幾個名稱皆屬於大寫式駝峰式命名法：

```
Number
UserInputNumber
```

```
MaxNumber
StudentIdentifier
FulltimeStudent
BestScore
CourseTime
CamelCase
UpperCamelCase
```

至於小寫式駝峰式命名法，請參考下列的例子：

```
amy
userName
happyStory
setData
getUserInput
lowerCamelCase
```

目前大部份的C++語言程式設計師，都是採用小寫式駝峰式命名法為變數命名（意即使用有意義的英文單字來為變數命名，原則上所有單字皆使用小寫英文字母組成，但從第二個單字開始，每個單字的第一個字母必須使用大寫），本書後續的程式範例，也將繼續使用小寫式駝峰式命名法為變數命名。

標準識別字（Standard Identifier）

本節最後為讀者介紹標準識別字（Standard Identifier），雖然它們完全符合變數命名的各項規定，但仍不建議讀者使用。所謂的標準識別字是一組在特定標頭檔（Header Files）或命名空間（Namespace）中預先定義好的常數、變數或函式名稱，例如我們已經在程式中使用過的cin、cout與endl等，事實上它們是定義在std這個Namespace中的名稱，您可以試著宣告以下的變數：

int cin=5; int cout=10; int endl=20;

相信我，這樣的程式碼是正確的！但如此一來，以後在您的程式碼中的cin、cout與endl，倒底是代表整數值或是其原本定義的輸入、輸出與換行的功能？像這樣的例子就充份說明了什麼是標準識別字 — 雖然不是C++語言的關鍵字，但仍不鼓勵您使用這些名稱來做為您所宣告的變數名稱，因為可能會和其原本的意義不同，對程式碼的意義帶來不必要的錯誤解讀。關於標準識別字還有一些其它的例子，包含NULL、EOF、min、max、open、close、sin、cos、pow與log等，請儘量別使用這些名稱為您的變數命名。

4.1.3 變數初始值宣告

變數就是在程式執行時，用以暫時儲存特定型態的值的一塊記憶體空間；在程式執行的過程中，一個變數所儲存的值可以被改變，這也正是「變數（Variable）」一詞的由來 — vary（變化）加上able（具有…能力）。我們在宣告一個變數時，除了要指定其變數名稱以及型態外，還可以給定一個「初始值（Initial Value）」— 變數一開始所儲存的值。請先回顧以下的變數宣告敘述語法：

變數宣告敘述（Variable Declaration Statement）語法定義

資料型態 變數名稱 [=初始值]?[,變數名稱 [=初始值]?]*;

在上述的語法規定中，每個變數除了宣告其所屬的型態以及其變數名稱外，還可以選擇性地接上「=初始值」；不論是在一個變數宣告敘述中的第一個變數宣告，或是後續其它的變數宣告都可以擁有這個選項 — 給定初始值，請參考以下的例子：

```
float weight=65.5, height=1.72;     // 宣告兩個變數並且都給予初始數值。
int contractDays=100;             // 宣告一個變數並給予初始值。
int productID, price=0, amount=12; // 宣告三個變數並給予其中兩個變數初始值
```

上述的變數宣告是和C語言完全相同的，但C++語言還提供了新的變數初始值給定的方法：

C++語言新提供的變數初始值給定方式
資料型態 變數名稱(初始值);
資料型態 變數名稱={初始值};
資料型態 變數名稱{初始值};

其中最後一種宣告的方法，是在C++11的標準才開始提供的，所以在編譯時必須要使用「-std=gnu++11」的選項才能順利的編譯。請參考下面的例子：

```
#include <iostream>
using namespace std;

int main()
{
  int i = 3;
  int j (4);
  int k = {5};
  int l {6};

  cout << "i=" << i << endl;
  cout << "j=" << j << endl;
  cout << "k=" << k << endl;
  cout << "l=" << l << endl;

  return 0;
}
```

上述這個程式的編譯與執行畫面如下：

```
[03:39 user@ws example]$ <nowiki>C++</nowiki> -std=gnu++11 newVarInitial.cpp
```

```
[03:39 user@ws example]$ ./a.out
i=3
j=4
k=5
l=6
[03:39 user@ws example]$
```

新的宣告方式，還提供了型態安全的檢查，我們將上面的程式修改如下：

```
#include <iostream>
using namespace std;

int main()
{
  int i = 3.5;
  int j (4.5);
  int k = {5.5};
  int l {6.5};
  char x {332};

  cout << "i=" << i << endl;
  cout << "j=" << j << endl;
  cout << "k=" << k << endl;
  cout << "l=" << l << endl;

  return 0;
}
```

其編譯結果如下：

```
[03:52 user@ws example]$ <nowiki>C++</nowiki> -std=gnu++11
newVarInitial2.cpp
newVarInitial2.cpp: In function ‘int main()’:
newVarInitial2.cpp:8:15: warning: narrowing conversion of ‘5.5e+0’ from
‘double’ to ‘int’ inside { } [-Wnarrowing]
newVarInitial2.cpp:9:13: warning: narrowing conversion of ‘6.5e+0’ from
‘double’ to ‘int’ inside { } [-Wnarrowing]
newVarInitial2.cpp:10:14: warning: narrowing conversion of ‘332’ from ‘int’
to ‘char’ inside { } [-Wnarrowing]
newVarInitial2.cpp:10:14: warning: overflow in implicit constant conversion
[-Woverflow]
[03:53 user@ws example]$ ./a.out
i=3
j=4
k=5
l=6
```

```
[03:53 user@ws example]$
```

注意到了嗎？新的{}方式還會進行型態安全的檢查。我們將{}這種宣告初始值的方法稱為List Initialization。當{}內沒有提供初始值時，編譯器會以0做為其初始值。

最後C++還提供了一種新的宣告方式：

```
auto valName = value;
auto valName (value);
auto valName = {value};
auto valName {value};
```

使用auto是讓編譯器視變數的初始值，自動決定適切的資料型態。

4.1.4 變數記憶體配置

在程式碼中使用變數宣告敘述所宣告的變數，在執行時會依所宣告的資料型態在記憶體中配置適當的位置。截至目前為止，本書已經介紹了兩種資料型態，分別是浮點數float與整數int，而它們都是使用連續的4個位元組（Byte）的記憶體空間。關於資料型態的更多細節，請參考本章3-2節。請考慮以下的變數宣告：

```
float weight, height, BMI;
```

此行變數宣告敘述，在程式執行時，會依據其宣告內容在記憶體中配置三個浮點數型態的計憶體空間，並將其分別命名為weight、height與BMI，後續在程式中使用這些變數時（不論是讀取或改變其值），就是對這些記憶體位置內的值進行操作。figure 1顯示了這三個變數可能的記憶體配置圖：

Fig. 1: 變數記憶體配置圖。

在figure 1中的每個矩形就代表了分配給這些變數的記憶體空間，我們將變數名稱標示在其所分配到的記憶體空間左側，並在其上方標示了該空間的起始位址[3]；由於C++語言為float（浮點數）型態的變數，所配置的是連續的4個位元組的記憶體空間[4]，因此以figure 1左上角的weight變數為例，其所配置的記憶體空間是從0x7ffff69a3232開始（標示於上方）至0x7ffff69a3235的連續4個位元組。由於變數所配置到的記憶體位置，必須等到程式執行時才能得知，如果我們想要存取變數weight的值時，並不是在程式中指定要存取0x7ffff69a3232至0x7ffff69a3235這連續4個位元組的記憶體空間（因為在撰寫程式時，我們還無法得知其記憶體位址），而是以該變數的名稱，也就是weight來進行存取。可是程式執行時，又要如何才能得知weight變數所在的記憶體位址呢？關於此點，其實是透過「符號表（Symbol Table）」來查詢的。

符號表（Symbol Table）

符號表是在程式編譯或執行時所建立的一個表格[5]，是為了管理在程式中用以識別的變數名稱與其所分配到的記憶體空間，其內容包含有變數的名稱、型態以及其所分配到的記憶體空間位址等資訊。以我們前述的weight、height與BMI這3個變數為例，在執行時其符號表可能會有如table 3的內容[6]：

符號(Symbol)	資料型態(Data Type)	記憶體位址(Memory Address)
weight	float	0x7ffff69a3232
height	float	0x7ffff69a3255
BMI	float	0x7ffff69a32f0

Tab. 3: 程式執行時的符號表內容

其中在記憶體位址的部份，符號表僅保存其起始位址，至於其結束的位址可以由同樣保存在符號表中的資料型態來決定。例如weight在符號表中記載了其型態為float，因此其所配置到的空間就是從符號表中所記載的0x7ffff69a3232開始，一直到0x7ffff69a3235的連續4個位元組（由於float資料型態佔用了連續4個位元組的記憶體空間）。請注意，此處所使用的記憶體位址僅供參考，其真實的位址必須在程式執行時才能得知。

Address-Of運算子

如果讀者對於某個變數在執行時，其所配置到的記憶體位址有興趣，則可以使用「位址（Address-Of）運算子，&符號」來取得變數的記憶體位址（更具體來說，是變數所配置到的起始位址）。只要在變數名稱前加上位址運算子，就可以取得其所配置的記憶體位址。請參考Example 1的addressOf.cpp程式，其中宣告了兩個變數，並使用&（Address-Of運算子）取得它們所配置到的記憶體位址：

Example 1

```
#include <iostream>
using namespace std;

int main()
{
   float weight, height; // 此處使用了一行敘述來宣告兩個變數

   cout << "變數weight所配置到的記憶體位址為: " << &weight << endl;
   cout << "變數height所配置到的記憶體位址為: " << &height << endl;
   return 0;
}
```

Example 1的addressOf.cpp之執行結果如下（注意，此結果僅供參考，實際輸出的記憶體位址將會有所差異）：

```
變數weight所配置到的記憶體位址為: 0x7fff5104cafc
```

```
變數height所配置到的記憶體位址為: 0x7fff5104caf8
```

4.2 常數

除了變數之外，在C++語言的程式碼中，還可以宣告常數（Constant）。常數就如同變數一樣，都擁有名稱、型態與內容值，不過一旦給定初始值後，就不允許變更其數值內容。

4.2.1 常數宣告

C++語言的常數宣告（Constant Declaration）的語法如下：

常數宣告敘述（Constant Declaration Statement）語法定義

const 資料型態 常數名稱 = 數值 [, 常數名稱 = 數值]*;

從上面的語法可以得知，其實constant declaration（常數宣告）的語法與變數宣告非常相似，不過必須在最前面加上const這個關鍵字，並且所有常數的宣告都必須給定數值（Value）[7]。在常數名稱方面，其命名規則與變數的命名規則一致，請自行參考3-1-2的說明。接下來，請參考下面的程式碼片段：

```
const int a=100;
const int b=3,c=5;
...
a=200;  // 改變一個常數的值
...
```

上面的程式碼正確地宣告了三個整數常數a、b與c，但在後續的程式碼中卻又改變了其中一個常數的數值！這樣會導致編譯時的錯誤，您會得到（error: read-only variable is not assignable（錯誤：唯讀的變數不可以指定數值）」的錯誤訊息，因為一個常數一但被宣告後，其值是不允許被改變的。

4.2.2 常數定義

除了使用前述的常數宣告方法外，我們還可以使用（#define）這個前置處理器指令（Preprocessor Directive）[8]來定義常數，其語法如下：

常數定義（Constant Definition）語法

#define 常數名稱 數值

與多個常數可以在一個宣告中同時宣告不同，常數定義一次僅能定義一個常數，且在定義時不需要使用等號（=），也不需要在結尾處的分號（；）。依照上面的語法，我們可以定義一個常數 PI（其值為3.1415926：

```
#define  PI  3.1415926
```

或是定義一個名為size的常數，其值為10：

```
#define  size  10
```

其實常數定義並不是幫我們產生一個常數，而是幫我們以代換的方式，將程式碼中所出現的特定文字串組合改以指定的內容代替。因為在C++語言中，所有以井字號（#）開頭的指令，都是在編譯時由編譯器先啟動一個前置處理器（Preprocessor）來負責加以處理的。包含了我們已經使用過的「#include<標頭檔>」指令，其實是在編譯前，先將指定的標頭檔案內容載入到程式碼中。至於此處所介紹的「#define」也是一個前置處理器指令（Preprocessor Directive），同樣是由編譯器內的前置處理器負責處理，以下面的程式碼為例：

```
#define  PI   3.1415926

int main()
{
   int radius=5;
   float area;
   area = PI * radius * radius;
   …
}
```

在編譯時，編譯器會先啟動前置處理器，依據其中第一行所定義的「#define PI 3.1425926」掃描尋找所有在程式碼中出現PI的地方，並將其改以3.1415926進行代換；完成這個代換後，編譯器才會展開真正的編譯工作。因此，上述的程式碼經過前置處理器處理後，會將以下的程式內容送交給編譯器進行編譯的工作：

```
int main()
{
   int radius=5;
   float area;
   area = 3.1415926 * radius * radius;
   …
}
```

關於前置處理指令更詳細的說明，可參閱本書第X章。

4.3 資料型態

一個資料型態包含了一組特定的資料內容的集合以及一組可以對其進行的操作。例如在數學領域裡，「整

數」就是一種資料型態，它是一個包含了在序列{…, -3, -2, -1, 0, 1, 2, 3, …}中的所有正整數、零以及負整數的無窮集合，我們可以對整數的資料內容進行包含加、減、乘、除在內的相關操作。不過要注意的是，因為受限於有限的記憶體空間，在電腦系統裡的資料型態只能是有限的集合。C++語言提供多種資料型態，包含基本內建資料型態（Primitive Built-In Data Type）與使用者自定資料型態（User-Defined Data Type）兩類。本章僅就基本內建資料型態進行說明，使用者自定資料型態請參閱本書後續章節。

C++語言所提供的基本內建型態，可分為以下幾種：

- 整數型態（Integer）
- 浮點數型態（Floating Point）
- 字元型態（Character）
- 布林型態（Boolean）
- 無值型態（Valueless or Void）

我們可以視需求在程式中宣告這些型態的變數來加以使用，本節後續將針對這些型態逐一加以介紹。

4.3.1 整數型態

顧名思義，整數型態（Integer type）就是用以表示整數的資料型態。在C++語言中的整數型態，是以Integer的前三個字母（int）表示。事實上，我們已經在本書中使用過這個int整數型態，在這個小節中我們將提供更完整的說明。請先回顧我們在第3章所介紹過的Example 5的compensaton-1.cpp程式，以下的程式碼宣告了一些整數型態的變數：

```
int contractDays;          // 合約總日數
int contractRemainingDays; // 合約剩餘日數
int monthlyFeeDiscount;    // 每月月租費優惠金額
int subsidy;               // 手機補貼款
int compensation;          // 違約金
```

這些使用int整數型態所宣告的變數，在程式執行的過程中會佔用連續的4個位元組（Byte）的記憶體空間，也就是32個位元（Bit） （雖然int變數的值（Value）在程式執行的過程中可以有所變化，但其值必須符合int整數型態的範圍（Range）。

對於一個資料型態而言，其所謂的範圍係指一個該型態變數之數值的上限與下限，換句話說就是該型態的變數所能表達的最大值與最小值，可由該型態所佔用的記憶體大小加以計算。以4個位元組的int整數為例，其範圍是由在記憶體中連續的32個位元的排列組合所決定的，其中最左邊的位元代表正負號，稱為符號位元（Sign Bit），以0代表非負（Non-Negative）的整數，也就是正整數或0；1則代表負整數（Negative）。此外，讀者必須特別注意的是，不論是正整數或負整數，其值一律以2補數（2’s Complement）表示。依據2補數的運算方式，32位元的int整數其可表達的最大正整數在記憶體中的內容為：

0111 1111 1111 1111 1111 1111 1111 1111

其值為+2,147,483,647，至於最小的整數則為：

1000 0000 0000 0000 0000 0000 0000 0000

其值-2,147,483,648。因此一個int整數型態的變數，其值是介於其最大值2,147,483,647與最小值-2,147,483,648之間。

2補數

2補數是用來表達帶有符號的二進制整數值(也就是有正、負號的整數)的一種方法，關於其原理及計算方式，可以參考Wikipedia關於2補數的條目。此處簡單說明如下：「將最左側的位元視為符號位元，用來代表正整數與負整數，其中0代表正整數、1代表負整數。正整數的數值就是扣除符號位元後，由剩下的位元所組成二進位數值；負整數的部份，同樣扣除符號位元後，將剩下的位元進行逐位元的NOT運算(意即每個位元由0變1、由1變0)後，再加1即為其值。例如一個使用8位元表達的整數，扣除最左側的位元做為正、負號後，其所能表達的最大正整數值即為01111111，意即$+2^7-1=127$；而使用2補數所能表達的最小負整數是10000000，其中最左側的位元為1表示此數為負號，其值之計算可先減一後進行逐位元的NOT運算，意即NOT(10000000-1)=NOT(01111111)=10000000=128，因此2補數的10000000即為十進制的-128。

型態修飾字（Type Modifier）

C++語言有一些可以配合int型態共同使用的關鍵字，我們將其稱為「型態修飾字（Type Modifier）」，例如在int前面加上一個「unsigned」關鍵字，就表示不使用最左邊的符號位元來表達正負號，而是直接使用完整的32個位元做為其數值內容，因此其可表達的範圍將會擴大，不過卻沒有辦法表示負數。因此，一個unsigned int整數可以表達的最大整數即為：

1111 1111 1111 1111 1111 1111 1111 1111

也就是$2^{32}-1=4,294,967,295$；由於不使用符號位元，因此unsigned int可表達的最小數值就是0。除了unsigned以外，其實C++語言還有提供「signed」關鍵字，用以表示要使用符號位元，因此「signed int」就表示使用符號位元的int整數型態。不過符號位元原本預設就會使用，因此不用特別使用signed修飾字「int整數本來就會使用符號位元，所以通常不會在宣告int整數變數時使用signed修飾字。

除了unsigned與signed修飾字外，整數int型態還可以搭配「short」與「long」兩個型態修飾字，將其表達空間加以調整。具體來說，使用short與long來修飾int整數時，會分別將int型態的記憶體大小減半與加倍。理想上，如果int是32位元，那麼short int則變成了16位元的整數，而long int則是64位元的整數。如果64位元的int整數仍無法滿足您程式的需求，C++語言還允許我們可以使用兩次long來修飾int整數，以得到128位元的int整數，也就是可以宣告為「long long int」型態[9]！當然，您也可以再搭配unsigned修飾字一起使用，來得到更大的數值範圍（不過使用unsigned的情況下就無法表示負數）。

配合型態修飾字的使用， C++語言一共有以下6種整數型態：

- short int（短整數
- int（整數
- long int（長整數
- long long int（倍長整數
- unsigned short int（無符號短整數
- unsigned int（無符號整數
- unsigned long int（無符號長整數
- unsigned long long int（無符號倍長整數

注意：型態也可以使用縮寫

我們在宣告singed、unsigned、short或long的整數變數時，還可以將int省略。下面彙整了所有可以縮寫的整數型態：

完整的型態表達	縮寫的型態表達
signed short int	short int或short
signed int	int
signed long int	long int或long
signed long long int	long long int或long long
short int	short
long int	long
long long int	long long
unsigned short int	unsigned short
unsigned long int	unsigned long
unsigned long long int	unsigned long long

2補數與正負數

如本節前述，整數是使用2補數來表示，因此一個使用了n個位元的int整數，其值可表達的範圍可依下列公式計算：

$-(2^{n-1}) \sim +(2^{n-1}-1)$

以32位元的int整數為例，其值可表達的範圍為$-(2^{31}) \sim +(2^{31}-1)$，也就是 -2,147,483,648 ~ +2,147,483,647。但是若使用了unsigned修飾字，則其值可表達的範圍就變成了：

$0 \sim +(2^{n}-1)$

以16位元的unsigned short int為例，其值可表達的範圍可計算為0 至$2^{16}-1$，也就是0至65,535。table 4彙整了C++語言所提供的各種int整數型態，以及其理想上可表達的數值範圍與記憶體大小：

資料型態	記憶體大小(位元)	數值範圍（最小值～最大值）
short int	16	$-(2^{15}) \sim (2^{15}-1)$ = -32,768 ~ 32767
int	32	$-(2^{31}) \sim (2^{31}-1)$ = -2,147,483,648 ~ 2,147,483,647
long int	64	$-(2^{63}) \sim (2^{63}-1)$ = –9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
long long int	128	$-(2^{127}) \sim (2^{127}-1)$= -170,141,183,460,469,231,731,687,303,715,884,105,728 ~ 170,141,183,460,469,231,731,687,303,715,884,105,727
unsigned short int	16	$0 \sim (2^{16}-1)$ = 0 ~ 65535
unsigned int	32	$0 \sim (2^{32}-1)$ = 0 ~ 4,294,967,295
unsigned long int	64	$0 \sim (2^{64}-1)$ = 0 ~ 18,446,744,073,709,551,615
unsigned long long int	128	$0 \sim (2^{128}-1)$= 0 ~ 340,282,366,920,938,463,463,374,607,431,768,211,455

Tab. 4: 整數型態記憶體大小與數值範圍

在此要特別提醒讀者注意，雖然在理想上，short與long分別可為int整數型態縮減一半或加倍其記憶體空間，但是由於實作上的限制，目前的作業系統都沒有完全依照這樣做，甚至在Windows系統與Linux/Mac OS作業系統上，還有不同的做法。換句話說，部份整數型態的記憶體大小配置在不同系統上並不一致。舉例來說，雖然在理想上，long long int會是一個配置有128個位元（也就是16個位元組）的整數，但不論在Windows，Linux或Mac OS系統上，long long int都僅配置到64個位元。table 5與table 6分別針對Windows系統與Linux/Mac OS系統上，列示了不同整數型態實際的記憶體配置與數值範圍：

資料型態	記憶體大小(位元)	數值範圍（最小值～最大值）
short int	16	$-(2^{15}) \sim (2^{15}-1)$ ， -32,768 ~ 32767
int	32	$-(2^{31}) \sim (2^{31}-1)$ ， -2,147,483,648 ~ 2,147,483,647
long int	32	$-(2^{31}) \sim (2^{31}-1)$ ， -2,147,483,648 ~ 2,147,483,647
long long int	64	$-(2^{63}) \sim (2^{63}-1)$ = –9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
unsigned short int	16	$0 \sim (2^{16}-1)$ = 0 ~ 65535
unsigned int	32	$0 \sim (2^{32}-1)$ = 0 ~ 4,294,967,295
unsigned long int	32	$0 \sim (2^{32}-1)$ = 0 ~ 4,294,967,295
unsigned long long int	64	$0 \sim (2^{64}-1)$ = 0 ~ 18,446,744,073,709,551,615

Tab. 5: Windows系統的整數型態記憶體大小與數值範圍

資料型態	記憶體大小(位元)	數值範圍（最小值～最大值）
short int	16	$-(2^{15}) \sim (2^{15}-1)$ ， -32,768 ~ 32767
int	32	$-(2^{31}) \sim (2^{31}-1)$ ， -2,147,483,648 ~ 2,147,483,647
long int	64	$-(2^{63}) \sim (2^{63}-1)$ = –9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
long long int	64	$-(2^{63}) \sim (2^{63}-1)$ = –9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
unsigned short int	16	$0 \sim (2^{16}-1)$ = 0 ~ 65535
unsigned int	32	$0 \sim (2^{32}-1)$ = 0 ~ 4,294,967,295
unsigned long int	64	$0 \sim (2^{64}-1)$ = 0 ~ 18,446,744,073,709,551,615
unsigned long long int	64	$0 \sim (2^{64}-1)$ = 0 ~ 18,446,744,073,709,551,615

Tab. 6: Linux/Mac OS系統的整數型態記憶體大小與數值範圍

從table 5中，讀者可以發現Windows系統的long int其實與int，以及unsigned long int與unsigned int，一樣都是32位元；但在Linux/Mac OS系統上，table 6則顯示了long int與long long int ，以及unsigned long int與unsigned long long int，都是64位元。

記憶體大小

我們現在已經知道同一個整數型態在不同系統上，其可以表達的數值範圍可能並不相同。因此同一個程式，如果在不同系統上執行時，有可能發生數值資料不正確的問題。為了保險起見，建議讀者在不同系統上開發C++程式時，最好先確定各種型態所佔用的記憶體大小為何，再進行相關的設計。為了得知一個型態的記憶體大小，我們可以使用sizeof運算子[10]來進行，sizeof可以幫助我們了解特定資料型態所佔用的記憶體大小，例如想透過sizeof了解int佔多大的記憶體空間，僅需以sizeof(int)即可得到答案（請注意sizeof所傳回的答案是以位元組為單位，如果您需要以位元為單位則必須再乘以8，也就是sizeof(int)*8[11]）。 請參考Example 2的intMemSize.cpp，下面的程式使用sizeof將各種整數型態所佔用的記憶體空間大小加以輸出：

Example 2

```
#include<iostream>
using namespace std;

int main()
{
      cout << "The size of a short int is ";
      cout << sizeof(short int) << " bytes." << endl;
      cout << "The size of an int is ";
      cout << sizeof(int) << " bytes." << endl;
      cout << "The size of a long int is ";
      cout << sizeof(long int) << " bytes." << endl;
      cout << "The size of a long long int is ";
      cout << sizeof(long long int) << " bytes." << endl;

      cout << "The size of an unsigned short int is ";
      cout << sizeof(unsigned short int) << " bytes." << endl;
      cout << "The size of an unsigned int is ";
      cout << sizeof(unsigned int) << " bytes." << endl;
      cout << "The size of an unsigned long int is ";
      cout << sizeof(unsigned long int) << " bytes." << endl;
      cout << "The size of an unsigned long long int is ";
      cout << sizeof(unsigned long long int) << " bytes." << endl;
}
```

這個程式在不同的系統上執行時，將會有不同的結果（如同table 5與table 6所示），其中在Windows系統上執行的結果如下：

```
The size of a short int is 2 bytes.
The size of an int is 4 bytes.
The size of a long int is 4 bytes.
The size of a long long int is 8 bytes.
The size of an unsigned short int is 2 bytes.
The size of an unsigned int is 4 bytes.
The size of an unsigned long int is 4 bytes.
The size of an unsigned long long int is 8 bytes.
```

但是同一個程式若是在Linux或Mac OS系統上執行時，其結果如下：

```
The size of a short int is 2 bytes.
The size of an int is 4 bytes.
The size of a long int is 8 bytes.
The size of a long long int is 8 bytes.
The size of an unsigned short int is 2 bytes.
The size of an unsigned int is 4 bytes.
The size of an unsigned long int is 8 bytes.
```

```
The size of an unsigned long long int is 8 bytes.
```

從上述的執行結果可以確認int型態的記憶體大小為4個位元組（也就是32個位元□□short int則將記憶體大小減半（變成了2個位元組）。不過Windows系統與Linux/Mac OS系統的差別在於Windows系統的long int型態仍然是佔用4個位元組，而Linux/Mac OS系統則是使用8個位元組做為long int型態。至於long long int型態，則不論Windows或Linux/Mac OS都是使用8個位元組。除了不同作業系統可能會有佔用不同大小的整數型態外，在一些較舊的系統上，整數型態所佔用的記憶體間亦可能會有所不同。因此建議讀者可以先在您的開發環境上執行Example 2的intMemSize.cpp程式，先確認各種整數所佔用的記憶體大小為何，再開始您的程式開發。

數值範圍

如果想要知道您所使用的系統上，各種整數資料型態的最小值與最大值，可以使用在climits標頭檔中的Macro□巨集）定義。在climits中總共定義了12個相關的macro定義，請參考table 7□

巨集(Marco)	說明
SHRT_MIN	short int型態的最小值
SHRT_MAX	short int型態的最大值
INT_MIN	int型態的最小值
INT_MAX	int型態的最大值
LONG_MIN	long int型態的最小值
LONG_MAX	long int型態的最大值
LLONG_MIN	long long int型態的最小值
LLONG_MAX	long long int型態的最大值
USHRT_MAX	unsigned short int型態的最大值
UINT_MAX	unsigned int型態的最大值
ULONG_MAX	unsigned long int型態的最大值
ULLONG_MAX	unsigned long long int型態的最大值

Tab. 7: climits標頭檔定義的整數型態相關Macro

你可以使用上述12個macro□印出各個整數資料型態的最大值與最小值，請參考下面這個程式：

Example 3

```
#include <iostream >
#include <climits>
using namespace std;

int main()
{
   cout << "The value of a short int is between ";
   cout << SHRT_MIN << " and " << SHRT_MAX << "." << endl;
   cout << "The value of an int is between ";
   cout << INT_MIN << " and " << INT_MAX << "." << endl;
   cout << "The value of a long int is between ";
   cout << LONG_MIN << " and " << LONG_MAX << "." << endl;
```

```
    cout << "The value of a long long int is between ";
    cout << LLONG_MIN << " and " << LLONG_MAX << "." << endl;

    cout << "The value of an unsigned short int is between ";
    cout << 0 << " and " << USHRT_MAX << "." << endl;
    cout << "The value of an unsigned int is between ";
    cout << 0 << " and " << UINT_MAX << "." << endl;
    cout << "The value of an unsigned long int is between ";
    cout << 0 << " and " << ULONG_MAX << "." << endl;
    cout << "The value of an unsigned long long int is between ";
    cout << 0 << " and " << ULLONG_MAX << "." << endl;
}
```

同樣地，這個程式在不同的作業系統上的執行結果並不相同，以Windows系統為例，其執行結果如下：

```
The value of a short int is between -32768 and 32767.
The value of an int is between -2147483648 and 2147483647.
The value of a long int is between -2147483648 and 2147483647.
The value of a long long int is between -9223372036854775808 and
9223372036854775807.
The value of an unsigned short int is between 0 and 65535.
The value of an unsigned int is between 0 and 4294967295.
The value of an unsigned long int is between 0 and 4294967295.
The value of an unsigned long long int is between 0 and
18446744073709551615.
```

至於在Linux/Mac OS系統上的執行結果如下：

```
The value of a short int is between -32768 and 32767.
The value of an int is between -2147483648 and 2147483647.
The value of a long int is between -9223372036854775808 and
9223372036854775807.
The value of a long long int is between -9223372036854775808 and
9223372036854775807.
The value of an unsigned short int is between 0 and 65535.
The value of an unsigned int is between 0 and 4294967295.
The value of an unsigned long int is between 0 and 18446744073709551615.
The value of an unsigned long long int is between 0 and
18446744073709551615.
```

上述的結果與table 5與table 6一致，但仍建議您在您的作業系統上實際執行一遍range.cpp程式，以確認在您的開發環境中各種整數型態可表達的範圍。

資訊補給站：使用來自C語言的標頭檔

由於C++語言是以C語言為基礎的程式語言，不但其基本的語法與C語言相同，甚至C語言所支援的函式與定義等，都仍然可以在C++語言中繼續使用。在使用C語言進行程式設計時，如果要使用特定的函式或定義時，必須先將其標頭檔（Header File）以「#include」加以載入。在C++語言中，如果要使用來自C語言的函式或定義時，也

必須將相關的標頭檔載入，不過要特別提醒讀者注意的是C++語言在標頭檔方面做了以下的變化：

1. C語言的標頭檔副檔名為.h但C++語言取消了副檔名。
2. 如果一個標題檔是沿用自C語言時C++語言會在其檔名前增加了一個小寫的c字母，以標明該標題檔是沿用自C語言。

舉例來說，在C語言中原本就有limits.h標題檔，用以定義整數型態的極值（包含最大值與最小值）；沿用到C++語言時，該標題檔檔名就變成了climits前面多了一個c後面少了副檔名）。

固定記憶體大小的整數

從C++ 11開始C++語言提供了固定記憶體大小的整數型態Fixed Width Integer Type可以解決整數型態存在著「在不同作業系統上長度不一的問題」。table 8是定義在cstdint標頭檔 中的固定記憶體大小的整數型態定義，以及可用於查詢其數值範圍的巨集：

型態	最大值(巨集)	最小值(巨集)	說明
int8_t	INT8_MAX	INT8_MIN	如同signed int型態，但準確使用8、16、32與64個位元的記憶體空間；其最左邊的位元為符號位元，並使用2補數表示負數。
int16_t	INT16_MAX	INT16_MIN	
int32_t	INT32_MAX	INT32_MIN	
int64_t	INT64_MAX	INT64_MIN	
uint8_t	UINT8_MAX	0	如同unsigned int型態（不使用符號位元的整數型態），但準確使用8、16、32與64個位元的記憶體空間。
uint16_t	UINT16_MAX	0	
uint32_t	UINT32_MAX	0	
uint64_t	UINT64_MAX	0	

Tab. 8: 定義在cstdint標題檔中的固定記憶體大小整數型態與數值範圍(自<nowiki

從table 8可得知C++語言（自C++11開始）提供了intX_t與「uintX_t做為使用X位元的signed與unsigned整數型態，其中X可為8、16、32與64位元。其中signed的「intX_t的數值範圍是介於INTX_MIN與INTX_MAX之間，而unsigned的「uintX_t的數值範圍則是介於0到UINTX_MAX之間。

下列的Example 4印出了這些型態所使用的記憶體空間（以位元組為單位）：

Example 4

```
#include <iostream >
#include <cstdint>
using namespace std;

int main()
{
    cout << "The size of an int8_t is ";
        cout << sizeof(int8_t) << " byte." << endl;
```

```
        cout << "The size of an int16_t is ";
        cout << sizeof(int16_t) << " bytes." << endl;
        cout << "The size of an int32_t is ";
        cout << sizeof(int32_t) << " bytes." << endl;
        cout << "The size of an int64_t is ";
        cout << sizeof(int64_t) << " bytes." << endl;

        cout << "The size of an uint8_t is ";
        cout << sizeof(uint8_t) << " byte." << endl;
        cout << "The size of an uint16_t is ";
        cout << sizeof(uint16_t) << " bytes." << endl;
        cout << "The size of an uint32_t is ";
        cout << sizeof(uint32_t) << " bytes." << endl;
        cout << "The size of an uint64_t is ";
        cout << sizeof(uint64_t) << " bytes." << endl;
}
```

請特別注意上面的fixedWidthInt.cpp程式，必須使用#include <cstdint>將所需的標頭檔載入。這個程式不論在Windows或Linxu/Mac OS作業系統上，都是相同的執行結果：

```
The size of an int8_t is 1 byte.
The size of an int16_t is 2 bytes.
The size of an int32_t is 4 bytes.
The size of an int64_t is 8 bytes.
The size of an uint8_t is 1 byte.
The size of an uint16_t is 2 bytes.
The size of an uint32_t is 4 bytes.
The size of an uint64_t is 8 bytes.
```

只要使用這組固定記憶體大小的整數來進行變數宣告，就不用擔心在不同作業系統上可能發生的數值範圍不同的問題。

接下來的Example 5使用了定義在cstdint標頭檔中的巨集，來印出各個固定計憶體大小的整數型態的數值範圍：

Example 5

```
#include <iostream >
#include <cstdint>
using namespace std;

int main()
{
   cout << "The value of an int8_t is between ";
   cout << INT8_MIN << " and " << INT8_MAX << "." << endl;
   cout << "The value of an int16_t is between ";
   cout << INT16_MIN << " and " << INT16_MAX << "." << endl;
```

```
    cout << "The value of an int32_t is between ";
    cout << INT32_MIN << " and " << INT32_MAX << "." << endl;
    cout << "The value of a long long int is between ";
    cout << INT64_MIN << " and " << INT64_MAX << "." << endl;

    cout << "The value of an uint8_t is between ";
    cout << 0 << " and " << UINT8_MAX << "." << endl;
    cout << "The value of an uint16_t is between ";
    cout << 0 << " and " << UINT16_MAX << "." << endl;
    cout << "The value of an uint32_t is between ";
    cout << 0 << " and " << UINT32_MAX << "." << endl;
    cout << "The value of an uint64_t is between ";
    cout << 0 << " and " << UINT64_MAX << "." << endl;
}
```

此程式在Windows系統與Linux/Mac OS系統上的執行結果相同，請參考以下的執行結果：

```
The value of an int8_t is between -128 and 127.
The value of an int16_t is between -32768 and 32767.
The value of an int32_t is between -2147483648 and 2147483647.
The value of a long long int is between -9223372036854775808 and
9223372036854775807.
The value of an uint8_t is between 0 and 255.
The value of an uint16_t is between 0 and 65535.
The value of an uint32_t is between 0 and 4294967295.
The value of an uint64_t is between 18446744073709551615.
```

128位元的整數型態

除了C++自C++11後開始支援的int8_t~int64_t與uint8_t~uint64_t之外，GNU Compiler Collection （GCC）也提供了使用符號位元與不使用符號位元（也就是signed與unsigned）的128位元整數型態，__int128[12] 以及 unsigned __int128。以下程式顯示了__int128與unsigned __int128型態所佔用的記憶體大小：

Example 6

```
#include <iostream >
using namespace std;

int main()
{
    cout << "The size of a __int128 is ";
    cout << sizeof( __int128) << " bytes." << endl;
    cout << "The size of an unsigned __int128 is ";
    cout << sizeof(unsigned __int128) << " bytes." << endl;
```

```
}
```

此程式執行結果如下（其所顯示的數值單位為位元組，再乘以8之後即為位元）：

```
The size of a _ _int128 is 16 bytes.
The size of an unsigned _ _int128 is 16 bytes.
```

數值表達

除了宣告變數為整數型態外，我們也可以直接在程式碼中使用整數數值，我們將在此說明各種整數型態的數值表示方法，其中依所使用的進位系統可分成十進制（Decimal）、二進制（Binary）、八進制（Octal）與十六進制（Hexadecimal）等四種表示法：

- 十進制（Decimal）: 除正負號外，以數字0到9組成，除了數值0之外，不可以0開頭，例如：0, 34, -99393皆屬之。
- 二進制（Binary）: 除正負號外，僅由數字0與1組成，必須以0b（零b）或0B開頭，例如（0b0, 0B101, 0b111皆屬之。
- 八進制（Octal）: 除正負號外，僅由數字0到7組成，必須以0（零）開頭，例如：00, 034, 07777皆屬之。
- 十六進制（Hexadecimal）: 除正負號外，由數字0到9以及字母（大小寫皆可）a到f組成，必須以0x或0X開頭，例如（0xf, 0xff, 0X34A5, 0X3F2B01皆屬之。

我們還可以在數值後面加上L（或l）、U（或u）強制該數值為long型態或是unsinged型態，兩者也可以混用以表示unsigned long型態，例如（13L, 376l, 0374L, 0x3ab3L, 0xffffffUL, 03273LU等皆屬之。

4.3.2 浮點數型態

浮點數型態（Floating Type）就是用以表示小數的資料型態，這也是我們在前面的章節內容中已經使用過的資料型態（C++語言提供三種浮點數型態，分別是float（double與long double（其中float型態適用於對小數的精確度不特別要求的情況（例如體重計算至小數點後兩位、學期成績計算至小數點後一位等情況），而double則用在重視小數精確度的場合（例如台幣對美金的匯率、工程或科學方面的應用等），至於long double（則提供更進一步的精確度。table 9先行將C++語言所提供的三種浮點數型態及其所佔之記憶體空間大小與數值範圍加以彙整，後續將在本小節後續內容中進行詳細的說明：

型態	意義	記憶體大小(位元)	數值範圍		精確度（有效位數）
			最大值	最小值	
float	單精確度浮點數 (Single-Precision Floating Point)	32	1.17549×10^{-38}	3.40282×10^{38}	6位
double	倍精確度浮點數 (Double-Precision Floating Point)	64	2.22507×10^{-308}	1.79769×10^{308}	15位
long double	擴充精確度浮點數 (Extended-Precision Floating Point)	128	3.3621×10^{-4932}	$1.79769 \times 10^{10308}$	19位

Tab. 9: C++語言所提供的浮點數資料型態與其記憶體大小和數值範圍

記憶體大小

如table 9所示（float（double與long double型態分別使用了32、64與128個位元（也就是4、8與16個位元組）的記憶體空間。如同我們在介紹整數時一樣（Example 7同樣使用sizeof運算子來將浮點數型態所佔用的記憶體大小印出：

Example 7

```
#include <iostream>
using namespace std;

int main()
{
        cout << "The size of a float is "
             << sizeof(float) << " bytes." << endl;
        cout << "The size of a double is "
             << sizeof(double) << " bytes." << endl;
        cout << "The size of a long double is "
             << sizeof(long double) << " bytes." << endl;
}
```

注意：過長的程式碼可以換行後再繼續

當你所撰寫的程式碼超過了文字編輯軟體一行可以顯示的字數上限時，雖然程式仍能正確地編譯與執行，但你也可以使用Enter來適度地換行，讓程式碼可以在同一個畫面中閱讀。例如在Example 7中的floatingMemSize.cpp中的第5行到第10行間，共有三個cout的敘述超過了一行，因此我們選擇在換行後繼續其內容。但是請不用擔心，這並不會在編譯時造成錯誤。事實上C++語言的程式碼是以分號「；」做為一行程式碼的結束。所以，那怕你用了很多行才寫完一個程式的敘述，只有在其分號出現時，才算是一個C++語言的敘述結束。

Example 7的執行結果如下：

```
The size of a float is 4 bytes.
The size of a double is 8 bytes.
The size of a long double is 16 bytes.
```

此處的結果顯示的是float、double與long double分別使用了4、8與16個位元組（也就是32、64與128個位元）的記憶體空間。讀者要注意的是，此執行結果在Windows與Linux/Mac OS系統上都相同，但是這不代表在所有的系統上C++語言的浮點數型態所佔用的記憶體大小永遠都會一致，尤其是在一些比較舊或是嵌入式系統上。所以仍建議讀者將Example 7的floatingMemSize.cpp程式實際在你的開發環境上執行一次，以確定其記憶體大小究竟是多少。

IEEE 754-1985標準

C++語言所提供的三種浮點數的型態，其實都是參考自IEEE 754-1985標準[13] 所加以實作的[14]。配合這個標準，在實作上C++語言將一個浮點數以下列方式表達：

$sign \:\:\: Mantissa \times base^{exponent}$

其中$sign$、$mantissa$、$base$與$exponent$分別表示：

- $sign$（正負符號）：用以決定此數值為正數或負數。
- $mantissa$（尾數）：又稱為significand（有效數）
- $base$（基底）：定義指數的基底，以2表示二進制、10表示十進制或16表示十六進制，在絕大多數的實作上，都是以二進制做為基底。
- $exponent$（指數）：指數值，其值可為正或負的整數。

舉例來說，一個浮點數123.84923可以使用這種方式表達為：

$123.84923 = + \:\:\: 1.2384923 \times 10^{2}$

其中這個浮點數的$sign$(符號)為「+」、$mantissa$(尾數)為1.2384923、$base$(基底)為10以及$exponent$(指數)為2。讓我們再看看另一個例子：

$-0.0012384923 = - \:\:\: 1.2384923 \times 10^{-3}$

其中這個浮點數的$sign$為「-」、$mantissa$與$base$仍為1.2384923與10，至於其$exponent$則為-3。這兩個例子主要的目的是幫助你瞭解C++語言用以表達浮點數的方法，所以都使用我們比較熟悉的十進制(以10做為基底)，因此也可以被稱做科學記號表示法(Scientific Notation)。不過電腦系統在實作上，浮點數都是使用2做為基底(電腦比較熟悉二進制啊～)，此處的兩個例子只是為了幫助讀者起見，才選用較易理解的十進制（以就是以10作為基底）來進行示範。

cfloat標頭檔

由於不同作業平台上的C++語言對於浮點數的實作存在著差異，因此其數值範圍與精確度皆不盡相同，如果需要準確的資訊，可以參考table 10所列示的「cfloat」標頭檔[15]中的相關巨集（Macro）。其中以FLT開頭的為float相關的巨集定義、DBL開頭的為double相關定義，以及LDBL開頭的為long double的相關定義：

Macro（巨集）	意義
FLT_RADIX	此巨集定義了C++語言用以做為基底（Base）的數值，其中以2表示二進制、10表示十進制以及16表示十六進制。由於效能的考量，在大部份的作業平台上，FLT_RADIX的值為2（意即採用二進制）。
FLT_MANT_DIG	在採用FLT_RADIX所定義的進制做為基底時，此巨集定義了尾數（Mantissa）所能使用的位數。如FLT_RADIX的值為2，則此定義為尾數以二進制表達時的可使用的位元數，此數值直接影響了浮點數的精確程度。
DBL_MANT_DIG	
LDBL_MANT_DIG	
FLT_DIG	定義了在採用十進制做為基底（Base）時，尾數（Mantissa）所能使用的位數。雖然絕大部份的實作都是以二進制做為基底，但由於我們人類所慣用的進制為十進制，因此這組巨集所提供的是等價的十進制數值。
DBL_DIG	
LDBL_DIG	
FLT_MIN_EXP	在採用FLT_RADIX所定義的進制做為基底（Base）時，此巨集定義了指數（Exponent）的最小值。這個數值為一個負的整數，並且直接影響了浮點數所能呈現的小數點後的位數。
DBL_MIN_EXP	
LDBL_MIN_EXP	
FLT_MAX_EXP	在採用FLT_RADIX所定義的進制做為基底（Base）時，此巨集定義了指數（Exponent）的最大值。這個數值為一個正的整數，並且直接影響了浮點數所能呈現的小數點前最多的位數。
DBL_MAX_EXP	
LDBL_MAX_EXP	
FLT_MIN_10_EXP	在採用十進制做為基底（Base）時，此巨集定義了指數（Exponent）的最小值。這個數值為一個負的整數，並且直接影響了浮點數所能呈現的小數點後最多的位數。這組巨集所提供的是便利我們使用的等價之十進制數值。
DBL_MIN_10_EXP	
LDBL_MIN_10_EXP	

Macro（巨集）	意義
FLT_MAX_10_EXP	在採用十進制做為基底（Base）時，此巨集定義了指數（Exponent）的最大值。這個數值為一個正的整數，並且直接影響了浮點數所能呈現的小數點前最多的位數。這組巨集所提供的是便利我們使用的等價之十進制數值。
DBL_MAX_10_EXP	
LDBL_MAX_10_EXP	
FLT_MIN	這組巨集所定義的是浮點數所能表示的最小值。
DBL_MIN	
LDBL_MIN	
FLT_MAX	這組巨集所定義的是浮點數所能表示的最大值。
DBL_MAX	
LDBL_MAX	
FLT_EPSILON	這組巨集所定義的是浮點數的誤差，其值為數值1以及使用浮點數所能表達的一個大於1的最小值之差。此值直接影響了數字的精確性。
DBL_EPSILON	
LDBL_EPSILON	
FLT_ROUNDS	當精確度不足而需要進位時，此巨集定義了所採用的方法，其可能的數值為：
	-1：未知（意即沒有一定的處理方法）
	0：強制捨去
	1：強制進位
	2：強制為正無窮大
	3：強制為負無窮大

Tab. 10: 定義在cfloat標頭檔中與浮點數相關的重要巨集

在table 10中的巨集是與平台相關（Platform-Dependent）[16]的定義，讀者可以透過這些巨集取得在您的作業平台上的浮點數之數值範圍及精確度的詳細等資訊。現在，讓我們透過Example 8的floatingMacros.cpp程式，實際執行以取得在您的作業平台上的浮點數數值範圍與精確度等資訊。

Example 8

```
#include <iostream>
#include <cfloat>
using namespace std;

int main()
{
  cout << "FLT_RADIX = " << FLT_RADIX << endl;
  cout << "FLT_MANT_DIG = " << FLT_MANT_DIG << endl;
  cout << "DBL_MANT_DIG = " << DBL_MANT_DIG << endl;
  cout << "LDBL_MANT_DIG = " << LDBL_MANT_DIG << endl;
  cout << "FLT_DIG = " << FLT_DIG << endl;
  cout << "DBL_DIG = " << DBL_DIG << endl;
  cout << "LDBL_DIG = " << LDBL_DIG << endl;
  cout << "FLT_MIN_EXP = " << FLT_MIN_EXP << endl;
  cout << "DBL_MIN_EXP = " << DBL_MIN_EXP << endl;
  cout << "LDBL_MIN_EXP = " << LDBL_MIN_EXP << endl;
```

```
  cout << "FLT_MAX_EXP = " << FLT_MAX_EXP << endl;
  cout << "DBL_MAX_EXP = " << DBL_MAX_EXP << endl;
  cout << "LDBL_MAX_EXP = " << LDBL_MAX_EXP << endl;
  cout << "FLT_MIN_10_EXP = " << FLT_MIN_10_EXP << endl;
  cout << "DBL_MIN_10_EXP = " << DBL_MIN_10_EXP << endl;
  cout << "LDBL_MIN_10_EXP = " << LDBL_MIN_10_EXP << endl;
  cout << "FLT_MAX_10_EXP = " << FLT_MAX_10_EXP << endl;
  cout << "DBL_MAX_10_EXP = " << DBL_MAX_10_EXP << endl;
  cout << "LDBL_MAX_10_EXP = " << LDBL_MAX_10_EXP << endl;
  cout << "FLT_MIN = " << FLT_MIN << endl;
  cout << "DBL_MIN = " << DBL_MIN << endl;
  cout << "LDBL_MIN = " << LDBL_MIN << endl;
  cout << "FLT_MAX = " << FLT_MAX << endl;
  cout << "DBL_MAX = " << DBL_MAX << endl;
  cout << "LDBL_MAX = " << LDBL_MAX << endl;
  cout << "FLT_EPSILON = " << FLT_EPSILON << endl;
  cout << "DBL_EPSILON = " << DBL_EPSILON << endl;
  cout << "LDBL_EPSILON = " << LDBL_EPSILON << endl;
  cout << "FLT_ROUNDS = " << FLT_ROUNDS << endl;
}
```

這個程式的執行結果如下：

```
FLT_RADIX = 2
FLT_MANT_DIG = 24
DBL_MANT_DIG = 53
LDBL_MANT_DIG = 64
FLT_DIG = 6
DBL_DIG = 15
LDBL_DIG = 18
FLT_MIN_EXP = -125
DBL_MIN_EXP = -1021
LDBL_MIN_EXP = -16381
FLT_MAX_EXP = 128
DBL_MAX_EXP = 1024
LDBL_MAX_EXP = 16384
FLT_MIN_10_EXP = -37
DBL_MIN_10_EXP = -307
LDBL_MIN_10_EXP = -4931
FLT_MAX_10_EXP = 38
DBL_MAX_10_EXP = 308
LDBL_MAX_10_EXP = 4932
FLT_MIN = 1.17549e-38
DBL_MIN = 2.22507e-308
LDBL_MIN = 3.3621e-4932
FLT_MAX = 3.40282e+38
DBL_MAX = 1.79769e+308
LDBL_MAX = 1.18973e+4932
FLT_EPSILON = 1.19209e-07
DBL_EPSILON = 2.22045e-16
```

```
LDBL_EPSILON = 1.0842e-19
FLT_ROUNDS = 1
```

資訊補給站：科學記號表示法（Scientific Notation）

請注意，在Example 8的執行結果中，小數值的部份都是以科學記號表示法（Scientific Notation）來顯示。所謂的科學記號表示法是將數值改寫為介於1至10之間的實數a與一個10的N次方的乘積，也就是$a \times 10^N$的形式。C++語言在使用科學記號表示法時，則是使用「ae$\pm N$」的形式，其中a為介於1至10的實數，N則為整數（可為正或負數）。例如在Example 8執行結果中的FLT_EPSILON，就是被顯示為1.19209e-07，其中a與N分別為1.19209與 -7，代表1.19209×10^{-7}，也就等於0.000000119209。又比方123.4567可以表示為1.234567e+2，也就是1.234567×10^2。

<nowiki>C++</nowiki>語言的浮點數實作

Example 8的執行結果，提供了我們進一步去瞭解C++語言浮點數型態的機會。首先FTL_RADIX巨集的值為2，就表示在實作上是以二進制為基底（當然這是為了效能的考量，在絕大多數的平台上都是如此）。至於「FLT_MANT_DIG」、「DBL_MANT_DIG」與「LDBL_MANT_DIG」巨集，則分別表示了float、double與long double等型態的尾數（Mantissa）所佔用的記憶體空間為24、53與64個位元。考慮到每個浮點數都有一個用以表示正或負數的符號位元（Sign Bit）以0代表正數，以1代表負數），因此尾數實際使用的記憶體空間還必須扣除掉符號位元所佔用的空間。利用型態所佔用的空間並扣除掉尾數所佔用的記憶體空間後，就可以得出其指數（Exponent）所佔用的空間；以32位元的float型態為例，扣除掉24個尾數位元（定義在FLT_MANT_DIG，其中包含一個符號位元）後，就可以得到其指數佔用了8個位元。依據此計算方式，figure 2顯示了C++語言浮點數的記憶體配置情形：

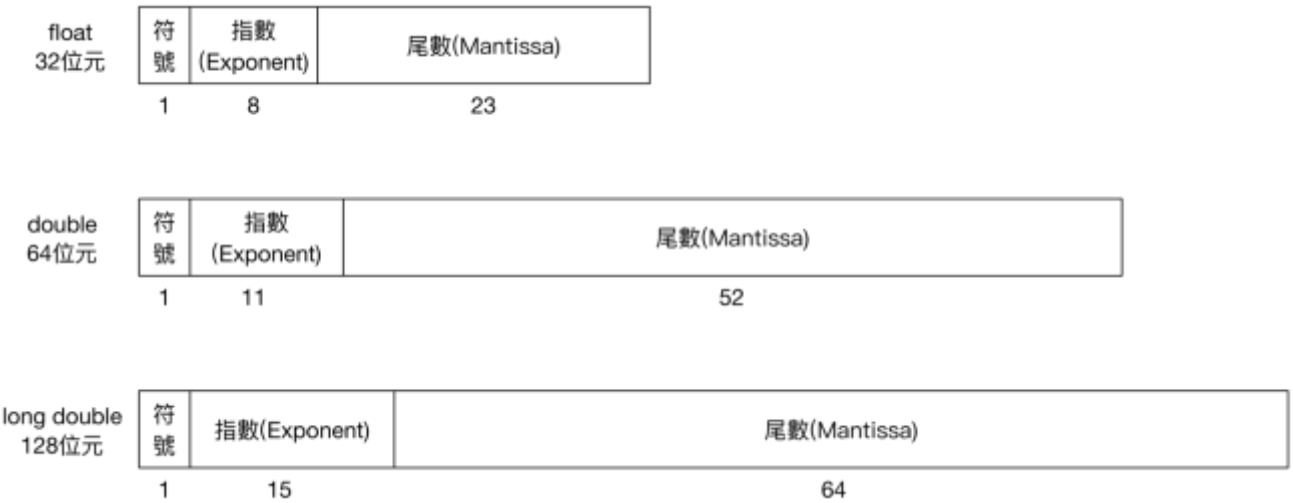

Fig. 2: 浮點數實作的記憶體位元配置

此處要注意的是，雖然long double型態是佔用了128個位元，不過在實作上（包含GNU Compiler Collection在內）通常都只使用其中的80個位元。

數值範圍

依據figure 2的記憶體配置，就可以分別計算出尾數與指數可表達的數值範圍，並進一步得出浮點數的最大值與最小值。不過在cfloat標頭檔中已定義有可以顯示極值的巨集，包含FLT_MIN、FLT_MAX、DBL_MIN、DBL_MAX、LDBL_MIN與LDBL_MAX等，請讀者自行參考Example 8的執行結果，您應該會發現它們的數值與我們在table 9中所列示的一致。但是要注意的是，其極值所顯示的是最大與最小的正值（Maximum and Minimum Positive Value），結合了符號位元與最大的正值後（FLT_MAX、DBL_MAX與LDBL_MAX），浮點數可以表示的最大與最小值如table 11所示：

型態	最小值	最大值
float	-3.40282×10^{38}	$+3.40282 \times 10^{38}$
double	-1.79769×10^{308}	$+1.79769 \times 10^{308}$
long double	$-1.18973 \times 10^{4932}$	$+1.18973 \times 10^{4932}$

Tab. 11: 浮點數的數值範圍

當然，對於浮點數來說，最重要的倒不是其最大值與最小值的範圍，而是其小數點後可以顯示的範圍；不論正或負值（也就是不考慮符號位元的話），浮點數可表達的最小正數值為：

型態	最小正值（Minimum Positive Value）
float	1.17549×10^{-38}
double	2.22507×10^{-308}
long double	3.3621×10^{-4932}

Tab. 12: 浮點數最小的正值

table 12的數值是取自於cfloat標頭檔中的FLT_MIN、DBL_MIN與LDBL_MIN等巨集。要注意的是，此處所謂的最小的正值（Minimum Positive Value）是暫不考慮符號位元的結果，比較像是說明小數點後可以表達的範圍。但是要特別注意的是，這裡的「最小正值」雖然表示的是一個浮點數可以表達到多小的數值，但這並不代表能夠精確地表達。事實上，一個浮點數能夠精確表達到小數點後幾位，其實是要由尾數來決定的。

精確度（有效位數）

浮點數的精確度（Precision）又稱為有效位數）是指數值中有多少位數是精確的，而不是指小數點後有多少位數的是精準的！舉例來說，123.456789在7位精準度的情況下，是指它可以精準到123.4567，而不是指它可以精準到小數點後7位。本節後續將討論C++的浮點數可以提供多少位數的精準度，不過這個主題涉及到二進制的小數表達方法，為了幫助讀者理解並簡化我們的討論，本節將以十進制做為討論的基礎。

如前述，浮點數其實會以figure 2的方式，儲存在記憶體中，其中的尾數部份完全決定了該數值的精確度（Precision也就是有效位數）。讓我們以一個簡單的例子來說明浮點數的精確度該如何計算：

> 請考慮一個使用float型態儲存的浮點數123.456789，依照本小節前面的說明，此數值必須先表達為 $+1.23456789 \times 10^2$，然後分別使用0、2與123456789做為符號位元（0表示正數）、指數（Exponent）以及尾數（Mantissa）的值。如figure 2所示，一個32位元的float型態，其指數與尾數分別佔用了8與2個位元，因此我們將使用這些位元來存放123.456789的指數與尾數部份，請參考figure 3。

Fig. 3: 浮點數123.456789的記憶體配置

此處主要的問題在於數值123456789無法使用23個位元來表達，事實上23個位元最多可以表達的數值為2^{23}-1=8388607，比起123456789還少了兩個位數。因此123.456789並無法使用float型態來精確地表達。當然，如果將這個數值改以double型態來表達時，就沒有精確度的問題，因為double使用52個位元來表達尾數，其最大可表達的數字遠遠超過123456789所需的位數。

有鑑於此，不同的浮點數型態可以提供的精確度是取決於其尾數可以表達的數值範圍，舉例來說，float型態的尾數為23位元，換句話說它可以使用23個位元來表達一個二進制的數值，轉換為十進制時，其可以表達的最大位數可計算為$\lfloor \log 2^{23} \rfloor = \lfloor 6.92368990027 \rfloor = 6$，也就是6個位數的精確度。至於double與long double則可以計算為$\lfloor \log 2^{52} \rfloor = \lfloor 15.6535597745 \rfloor = 15$ 與$\lfloor \log 2^{64} \rfloor = \lfloor 19.2659197225 \rfloor = 19$，也就是分別可提供15位與19位數的精確度。最後還是要在提醒讀者，此處的討論使用的是十進制而非實際使用的二進制，因此在一些細節上並不完全正確，但已經足夠做為觀念的演譯了。有興趣的讀者，可以自行參考二進制的小數表達方式，對浮點數的精確度進行更深入的探討。

數值表達

浮點數數值的表達有兩種方式:

- 十進制（Decimal）
  1. 除正負號外，以數字0到9以及一個小數點組成。
  2. 例如：0.0, 34.3948, 3.1415926, -99.393皆屬之。
- 科學記號表示法（Scientific Notation）
  1. 由一個十進制的數值與指數所組成。
  2. 在十進制的數值前可包含一個正負號，且在數值中可包含一個小數點。
  3. 指數的部份是表示10的若干次方，以一個E或e後接次方數表達。
  4. 在E或e的後面可接一個正負號，表示該次方數為正或負。例如：345E0代表345.0，3.45e+1代表34.5，以及3.45E-5表示0.000345。

另外，C++語言默認的浮點數型態為double，如果您要特別強制一個數值之型態為float或long double，可以在數值後接上一個F或L（大小寫皆可）。例如：3.45L, 3.45f等皆屬之。

4.3.3 字元型態

所謂的字元型態（CharacterType）就是用以表示文字、符號等資料，包含大小寫的英文字母、阿拉伯數字、空白、換行（也就是Enter）以及!@#$%^&*(){}_+-\|~'"等符號皆屬之，我們都將其統稱為「字元」。

char型態

C++語言提供「char」型態，來儲存與操作字元資料。下面這段程式碼宣告了一個char型態的變數，並將其初始值設定為字元A：

```
char c;  // 宣告一個char字元變數c
c ='A';  // 設定變數c的值為字元A
```

要特別注意的是，在C++語言的程式中，我們必須使用一組單引號「'」來將字元值包裹起來。因此上面這段程式碼必須使用'A'來表示大寫的英文字母A，如果沒有使用單引號來標明的話，C++語言的編譯器將無法理解它究竟是一個字元還是一個變數名稱，請參考下面這段程式碼：

```
char A;   // 宣告一個char字元變數A
char c;  // 宣告一個char字元變數c
c =A;   // 設定c的值為字元A? 還是讓變數c等於變數A的內容？
```

想想看，如果您是C++語言的編譯器，您會怎麼做？此處的「c=A」到底是設定變數c的值為字元A，還是讓變數c等於變數A的內容？答案當然是後者！您必須明確的使用單引號來標明字元值，編譯器才能正確地解讀。所以千萬不要忘記在設定字元值時，前後一定要加上單引號！

記憶體大小與字元編碼

現在讓我們來看看一個char型態的字元變數，在記憶體中將會佔用多少空間？請參考下列程式碼來檢查char型態所佔用的記憶體大小：

```
cout << sizeof(char) << endl;
```

試試看自己動手寫出可以測試上面這行程式碼的程式！執行看看結果是什麼？沒錯，它所輸出的結果將會是1，代表一個char型態的值會佔用1個位元組的記憶體空間，也就是8個位元。因此char型態的值可以有2^8=256種排列組合，從00000000到11111111（也就是十進制的0到255）。為了能夠在電腦系統內表達不同的字元，最直接的方式就是使用編號的方式，為每個字元給定一個獨一無二的數值號碼，例如字元A為1號、字元B為2號、…，依此類推。事實上，我們把這種方式稱之為「字元編碼（Character Encoding）」。以8位元的char型態來說，不論採用的編碼方式為何，最多都不能為超過256個字元提供不重覆的編碼。

ASCII字元編碼

目前在絕大部份的電腦系統中（不論是桌上型的Windows、Linux與Mac系統，或是手機或平板電腦的Android或iOS，乃至於嵌入式系統等都包含在內），字元都是採用名為ASCII的字元編碼方式來進行編碼。ASCII是American Standard Code for Information Interchange（美國標準資訊交換碼）的縮寫，採用整數的編碼值為電腦系統上的每個字元指定了一個介於0-127之間的整數值，其中小於32的數值代表的是控制字元，主要是用於通訊控制或字元控制的用途，請參考table 13的ASCII字元編碼表（其中Dec與Hex分別表示數值為十進制與十六進制）：

數值		字元	意義
Dec	Hex		
0	0x00	NUL	Null（意即「空」或「無」）
1	0x01	SOH	Start of Header（表頭開始）
2	0x02	STX	Start of Text（文章開始）

數值		字元	意義
Dec	Hex		
3	0x03	ETX	End of Text（文章結束）
4	0x04	EOT	End of Transmission（終止傳輸）
5	0x05	ENQ	Enquiry（查詢）
6	0x06	ACK	Acknowledgement（確認）
7	0x07	BEL	Bell（鈴響）
8	0x08	BS	Backspace（倒退鍵）
9	0x09	HT	Horizontal Tab（水平定位tab）
10	0x0a	LF	Line Feed（換行）
11	0x0b	VT	Vertical Tab（垂直定位tab）
12	0x0c	FF	Form Feed（表格換行）
13	0x0d	CR	Carriage Return（游標復位）
14	0x0e	SO	Shift Out（轉換退出）
15	0x0f	SI	Shift In（轉換進入）
16	0x10	DLE	Data Link Escape（資料聯結跳脫）
17	0x11	DC1	XON Device Control 1（設備控制1）
18	0x12	DC2	Device Control 2（設備控制2）
19	0x13	DC3	XOFF Device Control 3（設備控制3）
20	0x14	DC4	Device Control 4（設備控制4）
21	0x15	NAK	Negative Acknowledgement（否決確認）
22	0x16	SYN	Synchronous Idle（同步空檔）
23	0x17	ETB	End of Transmission Block（結束傳輸阻擋）
24	0x18	CAN	Cancel（取消）
25	0x19	EM	End of Medium（媒體結束）
26	0x1a	SUB	Substitute（替代）
27	0x1b	ESC	Escape（跳脫）
28	0x1c	FS	File Separator（檔案分隔）
29	0x1d	GS	Group Separator（群組分隔）
30	0x1e	RS	Request to Send（傳送要求）
31	0x1f	US	Unit Separator（單位分隔）

Tab. 13: ASCII字元編碼表中的控制字元

至於數值介於32至127之間的則是一般的可見字元，請參考table 14（其中Dec與Hex分別表示數值為十進制與十六進制）：

數值		字元	數值		字元	數值		字元	數值		字元	
DEC	HEX		DEC	HEX		DEC	HEX		DEC	HEX		
32	0x20	Space	56	0x38	8	80	0x50	P	104	0x68	h	
33	0x21	!	57	0x39	9	81	0x51	Q	105	0x69	i	
34	0x22	“	58	0x3a	:	82	0x52	R	106	0x6a	j	
35	0x23	#	59	0x3b	;	83	0x53	S	107	0x6b	k	
36	0x24	$	60	0x3c	<	84	0x54	T	108	0x6c	l	

37	0x25	%	61	0x3d	=	85	0x55	U	109	0x6d	m	
38	0x26	&	62	0x3e	>	86	0x56	V	110	0x6e	n	
39	0x27	'	63	0x3f	?	87	0x57	W	111	0x6f	o	
40	0x28	(	64	0x40	@	88	0x58	X	112	0x70	p	
41	0x29	)	65	0x41	A	89	0x59	Y	113	0x71	q	
42	0x2a	*	66	0x42	B	90	0x5a	Z	114	0x72	r	
43	0x2b	+	67	0x43	C	91	0x5b	[	115	0x73	s	
44	0x2c	,	68	0x44	D	92	0x5c	\	116	0x74	t	
45	0x2d	-	69	0x45	E	93	0x5d	]	117	0x75	u	
46	0x2e	.	70	0x46	F	94	0x5e			118	0x76	v
47	0x2f	/	71	0x47	G	95	0x5f	_	119	0x77	w	
48	0x30	0	72	0x48	H	96	0x60	`	120	0x78	x	
49	0x31	1	73	0x49	I	97	0x61	a	121	0x79	y	
50	0x32	2	74	0x4a	J	98	0x62	b	122	0x7a	z	
51	0x33	3	75	0x4b	K	99	0x63	c	123	0x7b	{	
52	0x34	4	76	0x4c	L	100	0x64	d	124	0x7c		
53	0x35	5	77	0x4d	M	101	0x65	e	125	0x7d	}	
54	0x36	6	78	0x4e	N	102	0x66	f	126	0x7e	~	
55	0x37	7	79	0x4f	O	103	0x67	g	127	0x7f	DEL	

Tab. 14: ASCII字元編碼表中的可見字元

從table 13與table 14中，可得知ASCII總共使用了0至127，共128個數值來為字元編碼；因此一個以ASCII編碼的字元需要使用7個位元來儲存其值（因為 7個位元可以表達二進制的0000000到1111111，或是十進制的0到127，總共2^7=128種可能的排列組合）。雖然ASCII只使用了7個位元所能表達的128種可能的排列組合來為字元編碼，但其已經涵蓋了英文的大小寫字母、阿拉伯數字、空白、換行以及一些特殊符號與控制用途的字元，是目前電腦系統使用最廣的編碼方式，幾乎沒有電腦系統不支援ASCII編碼！只要是使用ASCII編碼方式的字元資料，可以廣泛地應用在各種與字元資料相關的應用之上。

要注意的是，雖然ASCII是使用7個位元來表示一個字元，但由於系統存取記憶體是以位元組（也就是8個位元）為單位，因此實作上C++語言只能選擇大於等於7、且最為接近8的倍數的位元數做為char型態，這也就是char型態佔用8個位元的原因。

中文字元該如何處理？

聰明的讀者應該已經發現，佔8個位元記憶體空間的char型態，是無法用來表達中文字元的！因為就算把全部的8個位元都拿來使用，最多也只能使用從00000000到11111111（也就是從從0到255），來表達256個不同的字元 – 這遠遠不及所需要表達的中文字元數目。事實上，除了中文字元以外，世界上還有其它許多語言的文字也需要由電腦來進行處理，因此C++還提供了「寬字元型態(Wide Character Type)□□用以表達需要更多記憶體空間才能處理的多國語言文字字元。這個新的型態是wchar_t型態，依不同電腦系統的實作，其所佔的記憶體空間為2或4個位元組(也就是16或32個位元)，其所可以表達的字數則是2^{16}到2^{32}，也就是高達65,536與4,294,967,296個字元 – 這應該已經遠遠足夠了。

字元的表達

基本上，C++語言針對char型態的字元值有以下兩種表達方法：

- 字元值：以一對單引號' '將字元放置其中，例如'A'、'4'、'p'、'&'皆屬之。
- 整數值：對應於ASCII字元編碼的整數值，例如65與97皆屬之。

以下的程式碼宣告兩個char型態的變數c1與c2，並將它們的初始值都設定為英文大寫字母A，只不過分別是使用字元值與整數值加以設定：

```
char c1= 'A';  // 宣告一個char字元變數c1，並將字元A做為其初始值
char c2 =65;  // 宣告一個char字元變數c2，並將整數65做為其初始值
```

我們不但可以整數值來做為字元，甚至可以使用不同的數字系統，例如八進制或十六進制。請參考下面的例子：

```
char c3= 081;   // 宣告一個char變數c3，並將八進制的081做為其初始值
char c4 =0x41;  // 宣告一個char變數c4，並將十六進制的0x41做為其初始值
```

此處宣告了兩個char型態的變數c3與c4，並將其初始值分別設定八進制的081與十六進制的0x41，也就都是十進制的65），所以這兩個變數的初始值都是字元A，關於整數的八進制與十六進制的表達方法，請參考本章整數型態小節的說明。

以下的程式碼使用cout將上述所宣告的字元變數c1、c2、c3與c4加以輸出：

```
cout << c1 << c2 << c3 << c4 << endl;
```

其執行結果如下：

```
AAAA
```

由於這些變數在宣告時，其所給定的都是對應於ASCII編碼的字元A，不論其初始值給定的方式為何），因此它們的輸出結果都是A。

跳脫序列

C++語言還提供一種被稱為「跳脫序列（Escape Sequence）」的字元值表達方法，從字面上來看就是使用一組字元或數值，但讓它們跳脫原本的意義。要注意的是，使用時與字元值一樣，必須使用一組單引號將跳脫序列包含在內。在單引號內所包裹的內容，是先以一個反斜線「\」開頭，再接上一個或多個特定的字元或數值。例如反斜線與字元n所組合出的'\n'就是一個例子，它代表了要讓游標換行的意思。請參考下面的這段程式碼，它宣告了一個名為newLine的字元變數，並使用一組跳脫序列'\n'做為其初始值：

```

```

```
char newLine= '\n';
```

所以後續我們就可以使用newLine這個字元變數來進行換行，例如下面的程式碼：

```
cout << newLine;
```

當然，您也可以直接使用'\n'來做為換行：

```
cout << '\n';
```

跳脫序列除了可以用來做為字元變數的值以外，也可以混合在字串當中。例如以下的程式碼，會先印出一個"Hello World!"字串後，然後再加以換行：

```
cout << "Hello World!\n";
```

除了'\n'以外， C++語言還提供了一些跳脫序列，我們將其彙整於table 15。

Escape Sequence（跳脫序列）	**意義**
\a	alert（警示），也就是以電腦系統的蜂鳴器發生警示音。
\b	blackspace（倒退鍵），其作用為讓游標倒退一格。
\f	form feed（跳頁），讓游標跳至下一頁的第一個位置。
\n	new line（換新行），讓游標跳至下一行。
\r	carriage return（歸位鍵），讓游標回到同一行的第一個位置。
\t	horizontal tab（水平定位），讓游標跳至右側的下一個定位點。
\v	vertical tab（垂直定位），讓游標跳至下方的下一個定位點。
\\	backslash（反斜線），因為escape sequence是以反斜線開頭，所以若要輸出的字元就是反斜線時，必須使用兩個反斜線代表。
\?	?，輸出問號。
\'	'，輸出單引號。
\”	，輸出雙引號。

Tab. 15: Escape Sequence彙整

除了table 15所彙整的跳脫序列外，我們也可以在一組單引號內，以反斜線開頭再接上一個整數值（不論是使用八進制、十進制或十六進制皆可，但不支援二進制），來設定字元的ASCII編碼值。請參考以下的程式碼：

```
char c1= '\65';  // 宣告一個char變數cA並將十進制的65做為其初始值
char c2 ='\045'; // 宣告一個char變數cB並將八進制的45做為其初始值
char c3 ='\x41'; // 宣告一個char變數cC並將十六進制的0x41做為其初始值
cout << c1 << c2 << c3 << endl;
```

請讀者參考table 14的內容，想想看此程式片段的執行結果為何？此處要提醒讀者使用這種在一組單引號內，以反斜線開頭再接上一個整數值的宣告方法，其中在十六進制的部份並不需要使用0開頭，直接使用x開頭即可，例如'\x41'即可，不需要、也不允許寫成'\0x41'。

將字元視為整數

如前述，char型態的字元其實儲存的是ASCII編碼的整數值，因此可以將char型態視為是整數型態，只不過是個僅佔1個位元組（8個位元）的整數！我們已經在本節前面的內容中，學到了可以將ASCII的整數編碼值指定給字元變數，例如：

```
char c=65;
```

此處將字元c的定義為ASCII編碼值65，也就是大寫的英文字母A。請再參考以下的程式碼：

```
int i='A';
```

沒錯！您一點都沒看錯！這行變數宣告也是正確的！在上面的程式碼中，令字元A為int整數變數的值，其實就是將A的ASCII編碼值設定為變數i的初始值，也就是等同於「int i=65;」。您可以試著使用cout來將i的值印出，看看結果如何？

事實上，char型態除了可以做為ASCII的字元外，也可以直接做為整數使用，甚至還可以進行整數的運算。考慮以下的宣告：

```
int i='A' +3; // 令int整數變數i為字元A的ASCII編碼值再加上3
int j='N'-'G'; // 令int整數變數j為字元N與字元G的間距
```

在此處宣告的i數值為字元A的ASCII值加上3，也就是68；至於j的數值則是由字元'N'減去字元'G'，也就是ASCII值的78減71，結果等於7。從這些例子中可得知，char型態其實就是整數型態，只不過是一個僅佔1個位元組（也就是8個位元）的整數，其值對應到經由ASCII編碼後的字元。

有些程式設計師習慣使用char型態做為「較短」的整數，也就是一個佔8個位元的整數。回想一下整數的型態：預設的int型態是32位元，如果冠以short修飾字就可以將其記憶體大小縮減為16位元。所以將char型態做為8位元的整數之用，就好比是「short short int」 — 「短短整數」一樣。在C++11還沒有支援int8_t之前，這是一種相當常見的做法，尤其是當我們所要處理整數值較小時，實在不需要佔用過多的記憶體空間。例如，用以表示「棒球比賽進行時，一個打者的好壞球數」的整數變數strike與ball，其可能的數值分別是介於0至3與0至4的整數，若採用32位元的整數去存放似乎太浪費記憶體空間了。

若是將char視為是整數型態，還可以加上signed與unsigned修飾字，來使用或不使用符號位元。table 16彙整了其加上修飾字後的型態變化與數值範圍：

資料型態	記憶體大小（位元）	數值範圍（最小值～最大值）
signed char	8	$-(2^7) \sim (2^7-1)$ □ -128 ~ 127
char	8	$-(2^7) \sim (2^7-1)$ □ -128 ~ 127
unsigned char	8	$0 \sim (2^8-1) = 0 \sim 255$

Tab. 16: 以char型態做為整數的記憶體大小與數值範圍

以下以棒球比賽的成績記錄軟體為例，介紹一些適合使用char做為整數的例子：

```
char strike, ball, out; // 宣告strike□ball與out
// 做為記錄棒球比賽的好壞球與出局數的變數

char inning;      // 宣告inning變數代表棒球比賽所進行到的局數
char numPlayer;  // 宣告numPlayer代表棒球比賽的球員數
char hit, run;     // 宣告hit與run代表安打數與得分數
```

這些變數都有共通的特性：都是整數，而且都是數值範圍有限的整數。上面這些變數當然也可以改用佔了32位元的int型態來宣告，不過如此一來就會浪費3倍的記憶體空間。不過從C++11開始，筆者更建議您使用int8_t來代替char型態的整數使用，畢竟使用char宣告的整數，在可讀性上還是差了一些。

寬字元wchar_t型態

由於ASCII僅使用了7個位元來進行編碼，所以其可以表達的排列組合相當有限。換句話說，這世界上還有許許多多的文字與符號還沒有被收錄，為了解決這個問題，不同的系統先後採用了一些不同的編碼來解決這個問題，目前以Unicode編碼是最為普遍、且最廣為被接受的方式□Unicode一般中譯為萬國碼或統一碼，其存在就像是一套字典一樣，收錄了全世界所有可能被使用到的各種語言的文字符號，且仍不斷地持續增修中。每一個被收錄在Unicode中的字元符號使用16位元（也就是2個位元組）加以表示。由於16位元可表達的編碼空間為216=65,536，尚不足以涵蓋全世界各種語言文字符號的需求；因此Unicode更進一步依據使用的頻率，將所有的字元符號區分為17個平面□Plane□□以包含更多的可能性。簡單來說，您可以將Unicode想像為一套17冊的字典，每一冊（也就是每一個平面）都可以收錄65,536個字元符號，所以總計可表達17×65,536=1,114,112種排列組合。

不過Unicode並不是電腦系統上實際使用的編碼方式，目前在大部份的系統上是使用UTF-8來為Unicode字元編碼□UTF-8(8-bit Unicode Transformation Format)是以8位元為基礎的Unicode可變長度編碼方式，在實作上以8個位元（也就是1個位元組）為基礎，依編碼需要使用不同的位元數目來表達文字符號，但其位元數必須是8的倍數。換言之□UTF-8以8個位元所組成的一個位元組為基礎，視需要使用1至6個位元組為每個Unicode的文字與符號進行編碼；其中使用一個位元組的編碼與ASCII相容，所以大部份的ASCII文字資料都可以相容於UTF-8□至於其它的文字與符號則使用2個至6個位元組進行編碼[17)]，以繁體中文為例□UTF-8使用三個位元組進行編碼。

除了UTF-8之外，目前亦有UTF-16(16-bit Unicode Transformation Format)被不同的系統使用中，例如Windows系統就是預設使用UTF-16□與UTF-8類似□UTF-16採用16個位元做為基礎，任何字元所使用的位元數必須為16的倍數。為了因應UTF-8與UTF-16等各種需要多位元組的編碼方法□C++語言除了佔用1個位元組的char型態以外，還支援了佔用多個位元組的wchar_t型態，其命名是取自於Wide Character Type之意，也就是寬字元的意思。在Linxu與Mac OS系統上□wchar_t型態佔了4個位元組（也就是32個位元），可以配合UTF-8用以儲存一個Unicode字元；但是在Windows系統上□wchar_t的型態則僅佔了2個位元組（16

個位元），不能夠完整做為一個Unicode字元，而是配合UTF-16（16-bit Unicode Transformation Format）做為一個字元的組成部份。換句話說，在Linux/Mac OS系統上，一個wchar_t型態的字元可以做為一個Unicode的字元；但在Windows系統上，則是使用多個wchar_t型態的字元才能表達一個Unicode的字元。

類似於整數型態在不同系統上所佔用的記憶體大小不同的問題，從C++ 11起，C++語言還提供了char16_t與char32_t這兩種同樣用於處理多位元組的編碼需求的型態，更重要的是它們不論在哪個作業系統上，其所佔用的記憶體大小是固定的（分別是16個與32個位元）。這些寬字元在處理中文(或其它語言)的字元時，有時必須要一次使用多個寬字元才行，所以關於wchar_t（char16_t與char32_t等寬字元的使用，以及Unicode（UTF-8與UTF-16更多的細節，我們將在本書第XX章寬字元(目前還未完成)加以說明；屆時我們也將針對C++語言如何取得與輸入中文的字元與字串等主題加以探討。在此之前，本書後續的範例如有需要字元型態之處，一律都先使用char型態。

4.3.4 布林型態

除了整數、浮點數與字元以外，C++還提供了布林型態（Boolean Type） — bool，供我們使用。布林型態的值是來自於邏輯運算（Logic Expression），其運算結果只有兩種可能的值（true（真）或false（假），代表著某種情況、情境或是狀態、條件的「正確」與「錯誤」、「成立」與「不成立」、「真」與「偽」等「正面的」或「負面的」兩種可能。由於19世紀著名的英國數學家George Boole（對於邏輯運算有極大的貢獻，因此在電腦科學領域又將邏輯運算稱為布林運算（Boolean Expression），其值也稱為布林值（Boolean Value），這也正是C++語言所提供的布林型態bool的名稱由來。

請參考以下的布林型態變數宣告：

```
bool b1;
bool b2=true;
bool b3=false;
```

在上述的宣告中，我們以（bool）做為變數b1（b2與b3的型態，其中b2與b3還分別給定了true與false的初始值。

在此要提醒讀者注意的是，過去C語言並沒有提供布林型態，而是直接以整數值0做為false，並將其它所有非0的整數值視為true（不論正的或負的整數都視為true），由於C++語言是源自於C語言，所以bool型態的值，仍然可以沿用C語言的做法 — 使用非0的整數值與0，來分別表示true與false（不過做為一個C++語言的程式設計師，您應該要儘量避免使用整數值來代替布林值，應該直接以關鍵字（true）與「false（代替整數值，以提升程式的可讀性。以下程式碼示範了整數值與布林值間的關係：

```
bool b1=true;   // 宣告b1為布林型態的變數，其初始值為true
bool b2=false;  // 宣告b2為布林型態的變數，其初始值為true
cout << "b1=" << b << " b2="<< b2 << endl; // 印出變數b1與b2的內容
```

讀者可能以為上述程式碼的執行結果會輸出（b1=true b2=false），但其實使用cout來輸出布林值時，其輸出的將會是該布林值所對應的整數值。所以上述的程式碼將輸出以下的執行結果：

```
b1=1 b2=0
```

這顯示了雖然C++將非零的整數值視為true，並將整數0視為false，但在實際儲存布林值時並不是使用任意一個非零的整數做為true，而是使用整數值1做為true，並以整數值0做為false。

接下來，我們將b1與b2這兩個布林變數的值加以改變，並再次將它們的值印出：

```
b1= 5;      // 改變b1的值為整數值5
b2= -100;  // 改變b2的值為整數值-100
cout << "b1=" << b << " b2="<< b2 << endl; // 印出變數b1與b2的內容
```

由於5與-100皆為非零的整數值，代表的都是true的意思，因此b1與b2的值都會儲存為1（代表它們都是true），所以其執行結果為：

```
b1=1 b2=1
```

我們也可以將布林值指定給整數（當然true與false是分別使用1與0表示），請參考以下的例子：

```
int x= true;   // 宣告整數變數x，並將布林值true做為x的初始值
int y= false;  // 宣告整數變數y，並將布林值false做為y的初始值
cout << "x=" << x << " y="<< y << endl; // 印出變數x與y的內容
```

此例的執行結果如下：

```
x=1 y=0
```

最後，請讀者猜猜看，一個bool布林型態的變數會佔多少記憶體空間？除了撰寫一個使用「sizeof(bool)」的程式之外，您也可以自己推論一下合理的記憶體大小是多少？筆者在此給您兩個提示：

1. 電腦系統實際存取記憶體的最小單位為8個位元，也就是一個位元組；
2. 要使用多少位元才能表示true與false兩種可能的情況？解答附於本章末，請自行參考。

4.4 本章內容回顧

以下我們為讀者彙整了本章的學習重點：

- 變數（Variable）：在程式執行期間，用以儲存資料內容的記憶體空間，且該空間內所存放的數值內容可以視需要改變其值。
- 常數（Constant）：與變數類似，同樣是在程式執行期間，用以儲存資料內容的記憶體空間，但其內容值一旦給定後就不得再變更。
- 變數宣告（Variable Declaration）：C++語言要求所有變數，在首次使用前必須先加以宣告，就其名稱（Name）與資料型態（Data Type）進行定義，還可以在宣告的同時給定其值（稱為初始值），請參考以下的例子：
    1. 使用一行宣告敘述來宣告一個變數（包含或不包含初始值）：

```
int x;   // 宣告一個int整數變數x
```

```
int y=5; // 宣告一個int整數變數y，並給定初始值為5
```

1. 使用一行宣告敘述來宣告多個變數（包含或不包含初始值）：

```
int x,y;   // 宣告兩個int整數變數x與y
int i,j=5,k; // 宣告三個int整數變數i，j與k，其中j的初始值設為5
```

- 變數命名規則：
  1. 只能使用英文大小寫字母、數字與底線 _ ；
  2. 不能使用數字開頭；
  3. 不能與C++語言的關鍵字（keyword）相同。
- 關鍵字（Keyword）：在程式語言中具有特定意義的文字符號組合，C++的關鍵子列表請參考表3-1。
- 可讀性（Readability）：係指程式碼容易被理解的程度。增進可讀性的方式之一，就是使用固定的規則與易於理解的名稱來為變數命名。
- 大小寫敏感（Case Sensitive）：C++語言是對大小寫敏感的程式語言，意即在程式碼中所出現英文大小寫字母（例如A與a）會被視為不同的字元。
- 命名慣例（Naming Convention）：程式設計所使用的變數命名慣例，依據慣例來為變數命名除了有助於提升程式的可讀性外，也能夠減少犯錯機會。
- 駝峰式命名法（CamelCase）：目前業界普遍使用的命名慣例之一，直接使用英文為變數命名，因此變數名稱即為變數的意義或用途，因此具備良好的可讀性。若使用到兩個或兩個以上的英文單字時，每個英文單字除首字母外一律以小寫表示，且單字與單字間直接連接（不須空白），但從第二個單字開始，每個單字的首字母必須使用大寫。依據第一個單字的首字母，若是使用大寫字母，則稱之為「大寫式駝峰式命名法（UpperCamelCase）」，使用小寫則稱為「小寫式駝峰式命名法（lowerCamelCase）」。
- 標準識別字（Standard Identifier）：一組在特定標頭檔（Header Files）或命名空間（Namespace）中預先定義好的常數、變數或函式名稱。雖然完全符合變數命名的規則，但為避免造成認知上的混淆，仍不建議讀者使用。例如定義在std命名空間中的cin、cout與endl等，已經具備事實上的標準意義，不建議做為變數名稱之用。
- 常數宣告（Constant Declaration）：與變數一樣，在首次使用前必須先加以宣告。使用（const）關鍵字開頭，接著與變數宣告語法一樣，要定義其常數名稱與資料型態；但與變數宣告不同的是，常數必須在宣告的同時給定其值。請參考下面的例子：

```
const float pi=3.1415;   // 宣告一個float浮點數常數pi，其值為3.1415
```

- 常數定義（Constant Definition）：使用（#define）前置處理指令，來定義常數的名稱與其數值。請參考以下的例子：

```
#define  PI  3.1415  // 定義一個常數PI，其值為3.1415
```

- 資料型態（Data Type）：一個資料型態包含了一組特定的資料內容的集合以及一組可以對其進行的操作。
- int整數型態（int型態是使用二補數表示的整數值，可以使用short或long修飾字來將記憶體空間減半或倍增，包含int、short int、long int與long long int都是C++語言所支援的整數型態。這些型態還可

以使用signed與unsigned來標明使用與不使用符號位元（Signed Bit）。要特別注意的是，這些型態所佔用的記憶體大小，是與平台相關的，意即同樣的型態在不同系統裡所佔用的記憶體大小不一定相同。

- sizeof運算：令T為一資料型態，使用sizeof(T)可以求得型態T所佔用的記憶體大小。要注意的是其單位為位元組（Byte），而不是位元（Bit）。
- Address of運算：在程式執行時，每個變數會配置到一個特定記憶體位置。我們可以使用「位址（Address-Of）運算子 — &符號，來取得變數的記憶體位址（具體來說，是變數所配置到的記憶體位置的起始位址）。例如&x即為變數x所配置的記憶體起始位址。
- 固定記憶體大小的整數（Fixed Width Integer Type）：從C++ 11開始，C++語言提供了固定記憶體大小的整數型態，可以解決整數型態存在著「在不同作業系統上長度不一的問題」。相關型態包含intX_t與uintX_t（其中X為記憶體大小（可以使用的值包含8、16、32、64），至於在int前的u則代表unsigned之意。有些編譯器，還進一步提供了128位元的整數型態，例如GCC就提供了__int128與unsigned __int128兩種128位元的整數型態。
- 整數的數值表達方法：C++的整數可以表達為十進制（Decimal）、二進制（Binary）、八進制（Octal）與十六進制（Hexadecimal）等四種方法，其中二進制的數字使用0b或0B開頭、八進制以0開頭，至於十六進制則使用0x或0X開頭。整數數值後面也可以再加上L（或l）、U（或u）強制該數值為long型態或是unsinged型態，或者兩者也可以混用以表示unsigned long型態。
- 浮點數型態：參考自IEEE754-1985標準，C++語言提供了float、double與longdouble等三種浮點數，分別佔用了32、64與128位元的記憶體大小。對於一個浮點數的數值來說，C++語言預設的型態為double。
- 浮點數的數值表達方法：C++允許我們使用十進制或是科學記號表示法（Scientific Notation）來表達一個浮點數的數值。其中科學記號表示法是將一個浮點數分成一個十進制的數值與指數所組成，在十進制的數值前可包含一個正負號，並且在數值中可包含一個小數點。至於指數的部份是表示10的若干次方，以一個E或e後接次方數表達（可以為負數）。
- 字元型態：C++的char字元型態使用一個位元組，來表示對應使用ASCII編碼的字元符號。
- ASCII字元編碼： ASCII是American Standard Code for Information Interchange（美國標準資訊交換碼）的縮寫，是目前所有資訊系統廣泛使用的文字編碼方式。ASCII使用7個位元所能表達的0至127之間的整數值，來為電腦系統上的每個字元指定了一個唯一的值。其中小於32的數值代表的是控制字元，主要是用於通訊控制或字元控制的用途；至於大於等於32的數值代表的是包含阿拉伯數字、英文大小寫字母，以及各式可見的字元符號。
- 字元表達：C++可以使用一組單引號' '將字元放置其中，例如'A'、'4'、'p'、'&'皆屬之。此外，由於一個字元在記憶體中是以ASCII編碼的整數數值加以表達，因此字元也可以被視為是一個整數，並且直接做為一個整數使用。
- 跳脫序列（Escape Sequence）：使用一個反斜線「\」開頭，再接上一個或多個字元或數值，並賦予它們不同於原本的意義，例如\n代表換行之意。不過在使用時，仍需以一組單引號加以包裹。
- 寬字元（Wide Character）：不同於char型態僅使用一個位元組，寬字元使用多個位元組來表達一個字元。目前C++所提供的寬字元型態為wchar_t。
- 布林型態（Boolean Type）：C++語言還提供了bool布林型態，其值是來自於邏輯運算（Logic Expression）的運算結果，只能夠有兩種可能的值（true（假）或false（假），代表著某種情況、情境或是狀態、條件的正面或負面的兩種可能。

雖然只需要1個位元(以0及1分別代表false與true)即可，但C++語言的bool型態仍佔了1個Byte的記憶體空間，因為這是其最小存取記憶體的單位。

---

[1] 試想如果不加以分隔：float weight;（不就變成了（floatweight;）：這樣怎麼知道你是在宣告一個型態為（float）的「weight（變數呢？還是在寫一行（floatweight;）敘述呢？當然：floatweight;）敘述是錯誤的寫法。

[2] 專業的軟體開發團隊也會規範程式碼的縮排方式，以便團隊開發出的程式碼能具有一致性的風格。

[3] 此處使用的是以0x開頭的16進制的數字，同時出現在圖3-1中的記憶體位址僅供參考，其實際的位址於程式執行時才能得知。

[4] 在本書上一章IPO程式設計模型中，我們已經介紹過兩種不同的資料型態，分別是float（浮點數）與int（整數），其中float是使用4個位元組的記憶體空間，至於其它資料型態與其所佔之記憶體空間等細節，將於本章後續第3-3節為你說明。

[5] 依所使用的程式語言及其編譯技術而定，有些時候是在編譯階段產生符號表以將變數名稱轉譯為其對應的記憶體位址，有時則是將符號表嵌入在目的檔或可執行檔中，在執行階段才進行位址的轉譯。
[6] 此處所列示的符號表內容僅供參考，且其內容經過大幅度地簡化，其中所顯示的記憶體位址只是假設的，真正執行時其內容與此表格並不相同。
[7] 由於變數的內容可以在程式執行過程中被改變，所以一開始所給定的數值被稱為「初始值」，以便與後續變動後的不同數值做區隔；但常數在宣告時所設定的數值內容並不允許被改變，因此其數值內容並不存在著「初始」與「後續」的差異，我們將其稱為「數值」。
[8] 前置處理器指令是在程式被編譯前，先執行的指令操作，詳細說明可參考本書第X章。
[9] 或許你會推測C++語言也提供「short short int」的型態，來將int整數的記憶體大小縮減四倍，將32位元的int整數縮減為8位元的整數！不過請特別注意「C++語言並不允許這樣做！如果你真的需要8位元的整數，可以使用char型態做為替代，請參考本章char 字元型態小節
[10] 關於運算子「Operator」請參考本書第X章。
[11] C++語言使用星號*做為乘法的運算符號，更多細節請參考本書第X章。
[12] 請注意「_ _int128」型態名稱是以兩個底線開頭(中間不必使用空白分開)，後面再接上「int128」而成。
[13] 請參考IEEE Computer Society「IEEE Standard for Floating-Point Arithmetic」IEEE 754-2008 「Revision of IEEE 754-1985」「29 August 2008」
[14] 不過由於不同作業系統在浮點數的實作上存在著部份差異，因此其數值範圍與精確度亦有所不同。
[15] 沒錯「cfloat標頭檔也是來自於C語言的float.h標頭檔。
[17] 儘管原始的UTF-8規劃使用1至6個位元組來表達一個Unicode字元，但在2003年11月所公佈的RFC 3629重新規範為最多使用4個位元組，亦即不能超出原始Unicode編碼空間U+0000到U+10FFFF」

From:
https://junwu.nptu.edu.tw/dokuwiki/ - **Jun Wu的教學網頁**
國立屏東大學資訊工程學系
CSIE, NPTU
Total: 192972

Permanent link:
https://junwu.nptu.edu.tw/dokuwiki/doku.php?id=cppbook:ch-varconsttype

Last update: **2024/02/29 06:35**