作者: cppOrz (cppOrz) 看板: C_and_CPP
標題: [心得] C++ 與 C 的特性及區別
時間: Sat Aug 27 01:55:27 2005

無聊寫的，適合晚上睡不著的人催眠用。保證對學習沒幫助，對了解 C++ 是什麼，有一定的混淆作用；大家沒事看看就好，不用在意。

◎貓抓老鼠－－沒有適合所有人的編程語言

常常見到許多人在問「我應該學習什麼語言？」。類似這樣的問題，與其說是「見仁見智」，不如說是「貓抓老鼠」。俗話說：「會抓老鼠的貓，就是好貓。」對使用者而言，究竟何種編程語言是最合適的，端視其個人的需求及能力。要是始終拿不住耗子，這隻貓就算再名貴，再漂亮，也沒什麼意義。

當然，反過來說，如果學不好某種語言，也不必太過灰心，這或許表示您應該嘗試著轉往另外一片更適合自己的天空發展（另一片天空，可能是換養另一隻貓，也可能是換抓不同的老鼠，甚至可能是不抓老鼠改行養老鼠）。但千萬莫要因自己的挫折經驗，就拼命攻擊抵毀它，尤其是當「這隻貓」早已經被整個地球上業界頂尖的高手，和無數職業編程人員及業餘玩家，證明了「它絕對是個好樣的」，實用價值無可取代時，那些私心的言論，只不過暴露了批評者本身的偏狹。

◎其他主流語言與 C/C++ 的差異

在討論 C++ 和 C 的區別之前，或許先從「旁觀」者的角度，看看它們「相同」或「相似」的部份。此處主要的參照體是選擇一般通用型的編程語言。

一、實際運作的觀點

首先，從實際運作的觀點，C 及 C++ 都是循傳統的方式，透過編譯器和連結器，直接產生原生的機器碼（Machine Code 或 Native Code），而新一代的編程語言，有很多（例如 Java, C# 等）是先透過翻譯轉成 bytecode，然後再由虛擬機器（Virtual Machine）來執行。

雖然很多人認為 Java、C# 等語言依賴虛擬機器執行的方式，效率不佳，不過客觀的說，其實這種技術在某種意味上是比較先進的觀念，它最重要的優勢顯示在移植性方面。至於效率的問題多半出在各平台間的差異太大，而實作技術則顯然尚未完全成熟。（但這是可以克服的）

可能已經有人開始著急了。「照這樣說，C/C++ 不是落伍了嗎？」其實並沒有。本質上來看，兩者是一樣的。因為大可以把 C++ Complier 當成虛擬機，只是它不是由一家公司或少數特定人士所規範的，而且絕大多數的平台（機器和作業系統）上，都是支持 C/C++ 的。而像 J2SE,.NET 這些架構則是 Sun 或 MS 所制定的。

（甚至可以這樣認為：C/C++ 的虛擬機器是很多不同廠商、組織各自實作的，只是它們儘量遵循 ISO ANSI C/C++ 的標準，而 JVM 又或 CLI 這些東西，雖說也是開放的，但實則操縱在 Sun 和 MS 手中。）

實際上，C/C++ 與 Java, C# 等最大的分別，並不是體現在虛擬機器的觀念或作法上，而是體現在應用層面。光學會 C/C++ 語言，甚至它們的標準程式庫後，通常幹不了什麼有用的事。一個 C/C++ 程式人員，至少得熟悉一種 GUI 框架、一種 IPC 框架及一種 Database 框架，才大致可以說能處理大部份的應用問題。

當然，不是說用 Java, C# 就不必學會這些東西，只是這些功能有很多都已經成為該語言（框架）標準的一部份，在學習語言的時候，通常就會順便學到應用的架構。但在 C/C++ 中，所謂的「標準程式庫」，卻只規範了最最基本的 I/O，檔案處理，和常用的基礎演算法等等，其他都必須仰賴第三方或特定廠商的程式庫的支援，而這些東西則沒有所謂的標準，又常常受限於特定的平台環境，在取捨上比較不易。

二、型別系統的觀點

C/C++ 語言都是採用傳統的靜態型別系統（static type system），而許多新語言，為了便利物件導向特性的運作，是採用基於單根繼承的泛化型別系統，例如 Object Pascal, Java, C# 都是如此。

靜態型別系統的特性，就是不強制改變使用者自訂型別（UDT: User-defined Type）的記憶體佈局，並且允許在 stack 中配置 UDT 變量（也就是「物件」，但由於在 C 語言中，沒有真正物件導向的觀念，因此以「變量」來指稱）。此外，在靜態型別系統中，「型別」和「變量」之間，是壁壘分明的，你無法在編譯期產生變量，也不可能在執行期產生新的「型別」。

相對的，基於單根繼承的泛化型別系統，例如在 Delphi 的 VCL 架構中，所有的 VCL 元件，都繼承自 TObject，這就使得某些特殊的功能，例如以 ClassName 取得物件的實際型別資訊，就很容易實現。Java 和 C#等也都是如此。某些語言甚至內建 MetaClass 的特性，型別本身也可以當作變量，在執行期建立新的、

或修改既有的型別，這些都是根源於泛化型別系統的基礎。相形之下，在靜態型別系統中，很多特殊的功能，語言本身不直接支持，就必須自己去實現，或仰賴函式庫。

當然，靜態型別系統的最大優勢，就是執行期的效率。這也就是 C/C++的「零成本」原則：「使用者不該為他沒有用到的功能，付出執行期的效率代價」。因為不是每一件事情都得靠泛化型別系統的多態性來解決，並且解決的辦法也不應該只有一種（該語言所限制住的那一種）。

三、哲學的觀點

簡單的說，C/C++ 的設計哲學是把程式人員視為「成人」。它認為程式人員知道自己在幹什麼，而不是把程式人員當成「小孩」甚至「犯人」，需要特別的保護，甚至預設程式人員一定會犯某種錯誤，所以它儘量給予最大的自由及彈性，而不是強迫的限制或規範。

例如，包括內建型別，使用者自訂型別，和指標在內，它不強迫你一定要將變量（物件、陣列或指標）初始化，不強迫你檢查陣列的範圍，不強迫指標一定要指向合法的位址，它甚至允許你在各型別之間任意轉換。

又例如，C/C++它並不內建垃圾回收器（GC: Garbage Collection），它認為唯有程式人員自己，才能決定何時方是歸還動態申請記憶體的最適當時機，而不會在背後監視著一舉一動，幫忙收破爛。

當然，如果只是因為「自由」和「彈性」，而要付出高昂的管理和維護的代價，那是不值得的。C/C++ 相對於其他語言，顯得較為「寬鬆」，主要還是基於效率方面的考量。很多基於物件導向特性的新語言，雖然增加了安全和提供某些狀況下的便利性，然而一旦面對陌生或特異的問題，既有的工具和規範，無法直接套用時，過多的限制或「預設立場」，就很可能反變成了累贅。

從這個角度，也可以說，C/C++（其實主要指 C++）並不認為存在著某種最完善的方案，可以解決所有「應用層次」的問題，因此並不在語言層次去規範這些問題應該怎麼解決，而是把解決方案交給應用層（程式庫）去負責。語言本身只提供各種抽象的設計機制（介面），讓程式庫的使用能儘量與語言系統的風格一致。

◎ 偉大的 C 語言

就筆者個人的認知，C 絕對稱得上是一個偉大的語言。它最偉大之處，在於語

言本身，良好地對映了 Von Neumann 所提出的現代計算機的模型（主要是：二進位制、序列執行，以及將程式與資料都儲存在機器裏）。C 語言的指標（pointer），對記憶體操縱的簡潔、自由、及靈活性，就充份體現了這一特色。透過 C 語言，使用者可以較為直覺地運用抽象的數學觀念，來編寫程式，而不必直接面對晦澀的機器指令。

由於與機器模型之間的高度映射關係，以及語言本身的精鍊，相較於機器語言，C 除了具備高度的移植性，在效能方面的表現也相當突出，大部份的情況下，幾乎不遜於機器語言多少。很多大型的系統，除了少部份的核心代碼使用機器語言之外，絕大部份都是以 C 語言編寫的。

以現在的眼光，雖然 C 語言不是大多數應用領域的首選（當然，還是有不少領域是非常 prefer C 語言的），但透過 C 語言的學習，對於理解程式在機器中實際的運作情形，有莫大的幫助，也可以說是理解程式的基礎。任何人若想成為編程高手，精通 C 語言，可以說是起碼的條件。在整個資訊科學領域中，C 語言更是佔有極其關鍵、無法磨滅的歷史性地位。

◎從 C 到 C++

雖然其實筆者是很想下「偉大的 C++」這樣的標題，但實際上如果不是承襲了 C 語言的精髓，C++ 是不可能有今天的成就的。另一方面，C++的某些不盡人意之處（例如語法的過於複雜），也是因為承襲了 C 語言的特點才造成的。

究竟 C++ 和 C 有什麼不同呢？本來，在 ANSI C99 的標準以前（C89），C++ 至少有 95% 甚至可以說 99% 是兼容於 C 語言的，因此可以說 C 語言是 C++ 的一個子集。但在 C99 之後，某些 C 語言新的特性，特別是動態長度的 Array，使得這種大體上的兼容性被破壞了，也就是說，把 C 當成 C++ 的子集，這樣的說法可能要有所保留了。如果將來，C 和 C++ 再度出現某些重大的分歧，也不是什麼令人意外的事情。

一、強化「型別安全」——對型別系統的全面改進

許多涉及語法細節之處就略過了。在此只提出一個較重要的部份，是關於 C++ 與 C 的根本不同之處：

```
int *v = ...;
void *p = v;
int *p2 = p; // 合法的 C 程式碼，但在 C++ 中不合法
```

簡單的說，C++ 不允許 void * 隱式轉換為任意型別 T 的指標。但在 C 語言中，這是合法的。

C++ 禁止上述操作的理由，是為了強化「型別安全」。程式中一旦使用 void *，就等於自動放棄了編譯器對型別的自動檢查與核對動作，也就是放棄了型別安全。而明知不好，C++ 仍然支援 void * 這種用法的原因，主要是為了兼容於 C，但由於 void * 隱式換為任意型別的 T *，這種用法實在太危險，所以在 C++ 中被禁止了。

理想的 C++ 程式，是不應該出現 void * 這種用法的。C++ 之父 B.S.就曾指出，除了低階程式之外，應該儘量避免使用 void *，如果非得用 void * 不可，通常代表你的設計出了某些問題。

仔細觀察，C++ 的每一項基礎設施，都有提升型別安全的意味在其中。例如：

1. 引入 bool 型別，避免混淆。（主要問題在函式 overload 時）
2. 鼓勵以 0 而非自行定義的 NULL 巨集等代表空指標。（B.S.大和另一位 Herb Sutter 大，在 2003 年底提出新增加 nullptr 關鍵字，但不曉得 C++03 是否有通過）。
3. 引入 const，讓「常數性」成為與型別不可分割的一部份，除了提升安全，讓編譯器承擔檢核的責任之外，也有助於代碼的優化。（因此後來 C 語言也跟進採用。）
4. 引入 const, inline 等用法，減少非必要巨集的使用。（因為展開巨集是預處理器的動作，沒有通過編譯器，也就沒有型別安全可言）。
5. 引入 reference 機制，簡化指標的語法，並有效減少指標（尤其是兩層以上的複雜指標）的使用。
6. 引入 new 和 delete，取代 malloc 和 free，把動態記憶體配置的工作，提升至語言層級，減少強制轉型的使用（另一主要目的是為了配合 operator overloading，提升介面的一致性）。
7. 引入新的 static_cast, const_cast 等關鍵字，鼓勵儘量減少強制轉型的使用。
8. 引入 function/operator overloading 機制，讓同名函式及各種運算子，可依據不同的操作型別，實現不同的動作。強調「型別」也是函式具名的一部份，達成介面一致性，並使 UDT 能像內建型別的操作一樣自然。

這些每一個小地方，都可以看出 C++ 為了強化「型別安全」，所付出的用心和努力，雖然除了禁止 void * 的隱式轉型之外，基本上沒有限制 C++ 使用者延用舊的 C 語言的舊式習慣寫法，但筆者認為，了解型別系統的特性，並隨時意識

著「型別安全」，是掌握良好 C++ 編程風格的最重要觀念。

二、在「思維方法」上的差異

程式語言處理的不外乎資料結構及演算法，STL 的發明人也說過：「程式基於精確的數學。」前面提過，C 語言偉大之處，就是它十分良好地對映到機器模型，免除了直接使用機器語言的晦澀。

也就是說，C 程式人員不必去操心 register 管理、記憶體定址等等極度低階的細節問題。其所思考的，多半像是「我應該用什麼演算法，把某幾段特定記憶體內的資料取出來，經過怎樣的運算後，再存到特定的記憶體區段去.....。」這種把運算和存取操作的細部具體動作，轉換為抽象的數學思考的流程，本質上仍然是非常貼近機器模型的。而這樣的風格，不僅反映在 C 程式碼上，更多半根深蒂固地植入 C 程式人員的思維方式內。

隨著資訊科學的發展，愈來愈多的應用問題，需要利用編寫程式來處理；人們發現，大部份應用程式所使用的演算法和資料結構，是極為有限的。另一方面，編寫程式語言的常用技巧，卻已經累積地相當成熟了，程式人員需要付出更多心力的，不再是某個典型的演算法或資料結構，應該如何實現，如何處理；而在於，如何將問題的本身，適當地轉換為程式語言。

因此，一種讓程式語言能夠以「貼近待解決的問題」的方式來思考，而不再只是侷現於「貼近機器模型」的思想，就應運而生。簡單地說，它就是起源於 70 年代（甚至更早），在 80~90 年代開始快速發展，直至今日，雖不再新鮮，卻仍屬方興未艾的「物件導向」的觀念。

由於物件導向（OO: Object-Orientd）的觀念是如此氾濫，甚至已經上升到哲學的層次，幾乎沒有一個比較新的語言（80 年代以後），不支援它的特性，所以這裏也就不多介紹了。只是要指出一點， C++ 也好，或其他支援物件導向特性的編程語言也好，它們與 C 語言最大的分別，並不在語法或功能的區別上，而是在於看待問題的基本思考方式，也就是所謂「思維方法」上的差異。

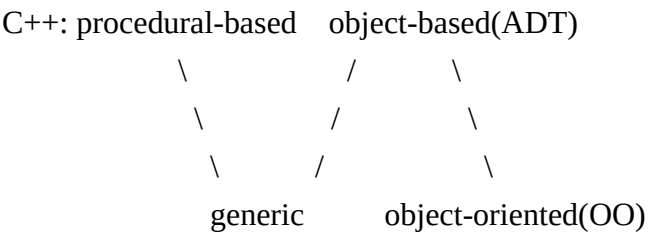
三、multi-paradigm

C++ 和 C 語言，在觀念上最大的不同之處，就是，C++ 是支持 multi-paradigm 的編程語言。如下面所示，C 語言及傳統的 Pascal 語言，是所謂 procedual-based 的編程語言，而 Java, C# 等較新的語言，則是 object-oriented 的編程語言（OOPL）。

至於 C++，它實際上是個支援 multi-paradigm 的編程語言，因為它不僅保留了 C 的程序導向的編程，更重要的是它沒有沒有為了要支援 OO，而破壞基於 C 語言的靜態型別系統，因此它提供的 ADT（abstract data type）機制，與繼承和執行期繫結等 OO 特性的機制是互相獨立的。這使得 C++ 在 OO 的執行期多型之外，罕有地提供了強大的編譯期多型的機制，也就是一般稱為「泛型編程」的技術。

procedural-based（eg: C, Pascal...）

object-oriented（eg: Objective C, Object Pascal, Java, C#...）



由上面的簡單示意圖可看出，泛型（generic）的編譯期多型的特性，不止對應在 ADT 上，也可以直接對應到程序導向的編程，例如 C++ 標準程式庫所提供的泛型演算法，就大部份是以函式而不是 class 來呈現的，實際上，整個 C++ Standard Library，除了 I/O 的部份，幾乎完全沒有用到 OO 的執行期多型的特性（更多的是 ADT 和 template）。

此外，或許有人會提出，其實 Java 或 C# 也是支援 generic 編程的，是沒錯，Java 也有類似 C++ 的樣板容器的功能，但實際上是用「代換法」做的，並沒有真正產生新的型別，因此它無法達到 C++ template 那樣可以有型別客製化（特殊化: specialization），或與其他抽象化機制合作（例如繼承、甚至遞迴）的多樣化的能力，並不算真正意義上的編譯期多型。實際上，Java 和 C# 語言所採行的單根繼承的泛化型別系統，早就先天限定它們不適合朝編譯期多型的方向發展，它們比較接近純粹的 OOPL。

C 語言的思考方式偏重於資料運算和記憶體存取的動作，物件導向的思考方式，則是將問題分解成不同的抽象概念（class），讓使用者專注在概念與概念間之的關聯，能從一個整體的大的方向，去關注問題，避免過早陷入細節，見樹而不見林。

同時，良好的設計，是當需求有所改變時，只需要修改、調整部份的模組，就可

以完成工作，不必整體性的翻修，牽一髮而動全身。這也是物件導向設計的重要精神，有一個專門的領域 DPs（Design Patterns），它與特定程式語言無關，就是在研究面對各種問題需求的典型解決方式，現在學物件導向設計一定會接觸到它。

至於，C++「多思維面向」（multi-paradigm）的特性，又是如何影響編程的思考方式呢？

這裏舉個《Modern C++ Design》第七章的例子。Smart Pointer 的發展動機，是為了防止直接操作指標所帶來的危險性，但隨著各種不同的需求，它的實作細節也就有所不同。例如：它能不能與其他容器類（例如標準程式庫中的 vector, list 等）共用，以及使用的細節如何？是否允許取得原始指標？是否對各種操作動作進行檢查，如何檢查？甚至，是否支援多緒程式安全地操作……等等。

如果將各種需求組合都列出清單，再一個一個實作，勢必沒完沒了。最理想的方式，是讓程式員自由選擇各種「需求策略」，讓編譯器自動產生相應的程式碼。這種設計乍看來是遙不可及的理想，但實際上已經做到了。這就是 Loki 函式庫所提供的實作品 class template SmartPtr：

```
template
<
   typename T,
   template <class> class OwnershipPolicy = RefCounted,
   class ConversionPolicy = DisallowConversion,
   template <class> class CheckingPolicy = AssertCheck,
   template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

由於牽涉的選擇項目過多，這裏只解釋 OwnershipPolicy，也就是實際物件擁有權的策略，它預設是 RefCounted，也就是參用計數的規則。但也可以依據需求的不同，選擇其他的擁有權策略，例如：RefCountedMT、DestructiveCopy、DeepCopy、……等等。使用方式如下：

```
class User {...};

typedef SmartPtr<User, RefCounted> UserPtr;
```

如此，UserPtr 就變成類似 boost::shared_ptr<User> 的作用，可以和標準容器合作，而實現 Java、C# 語言常見的功能。又假如：

```
class Manager {...};
typedef SmartPtr<Manager, DestructiveCopy> ManagerPtr;
```

現在，MangerPtr 則和 std::auto_ptr<Manager> 一樣，採取所謂「摧毀式複製」的語義，也就是同時只有一個 ManagerPtr 可以真正操縱同一份 Manager 類型的實體物件。

實際上，SmartPtr 的實現牽涉到 ADT、多重繼承、編譯期多型等等的特性，它應用了一種叫 policy-based 的設計觀念。這與其他程式語言或是 DPs 所標榜的 OO 的特性，或所謂「良好設計」的最終目的，並沒有不同，同樣是將不同的概念獨立分解，再巧妙組合起來。只不過，在 C++ 中，除了傳統 OO 執行期多型的技術之外，還多了強大的編譯期多型的支援，使得不僅「物件」(資料結構和演算法)，可以在執行期被彈性處理，就連「型別」(概念)的本身，在編譯期，也可以自由的選取整合。這對程式碼編寫的簡潔、靈活性和執行效率，都能帶來很大的提升。