Programs Knowledge

2012-10-18

C/C++之指標 (pointer)，參考 (reference) 觀念整理與常見問題 (轉貼)

100206 0 C / C++

C/C++之指標 (pointer)，參考 (reference) 觀念整理與常見問題 (轉貼)

轉貼至http://sandwichc-life.blogspot.com/2007/10/cc-pointer-reference.html#two

C/C++之指標 (pointer)，參考 (reference) 觀念整理與常見問題

這篇文章是由我舊的blog轉貼過來的
文中某些小細節稍作修改

前 言
這是以前替人代班教課時寫的一些東西
重新整理後放上來，一方面當作自己的備忘錄 (自己最看得懂的還是自己寫的東西)
另一方面如果有人有這方面的問題，希望此文能對你們也有一點點幫助。

很多程式員說：學C/C++而不會使用指標，相當於沒學過C/C++。
本文針對C/C++中，指標與參考的常見問題或錯誤，做了一番整理，但求能達到拋磚引玉之效。如有疏漏或錯誤之處，尚請不吝告知指教。
目錄

請用「空白」區分關鍵字

贊助商連結

PDFmyURL

1. 何謂指標 (pointer)? 何謂參考 (reference)?
我們先談指標 (pointer)。指標，其實也只是一個變數，只是這個變數的意義是：指向某個儲存位址。很玄嗎?
一點也不。下面這張圖就可以輕易的看出指標為何物。

圖 中，a, b, c, d, p1, p2都是一般的變數，儲存在記憶體 (memory) 中。其中，p1變數所記載的值是變數a的記憶體 (memory) 位址，而p2則記載著b的記憶體位址，像這樣的狀況，我們就稱p1是一個指向a的指標，相同的，p2是一個指向b的指標。
在C/C++中，我們用下面的式子來表示這個關係：

```
int *p1 = &a;
int *p2 = &b;
```

其中的&，稱為address of (取址)。即，p1 = address of a，p2 = address of b。
另一個符號*，代表的意義是指標。

```
int *p1
```

要由後往前閱讀來瞭解它的意義：p1 is a pointer points to an integer。因此，

```
int *p1 = &a;
```

這整行，我們可以看成：p1 is a pointer points to integer variable a，即：p1是一個指標，指向整數變數a。
且讓我們暫時打住指標的討論，轉頭看看參考 (reference)。
參考，可以想像成是一個變數或物件的別名 (alias)。通常，當函式 (function) 的參數 (parameter) 在函式中會被修改，而且要把這個修改結果讓呼叫函式的部份繼續使用，我們會用參考來當參數傳入函式中。
讓我們看看下面的例子：

```
void swap(int &a, int &b){
int tmp = a;
a = b;
b = tmp;
}
```

當其他程式呼叫此交換程式時，只要直接寫swap(x, y)就能交換x與y的值。在這裡，a和b為x與y的別名，即：a就是x，b就是y，如同美國國父就是華盛頓一樣。a和b不是x和y的複製品，任何做用在a與b上的動作都會反應在x與y上面，反之亦然。
指標和參考之所以難懂，有很大一部份的原因是符號上的陌生所致。加上&既能用於取址又能用於參考，容易造成初學者的混淆。下面我們提供幾個建議來幫助各位看懂這些符號。

- 把int *p視為 int* p。
  把int和*連在一起看，當作是一種型態叫做 "指向整數之指標"，要比int *p自然得多。同樣的方式也可以套在char* p或void* p等。但要注意的是下面的狀況：

int* p, q;

不要把這行誤解成p, q都是指向int之指標，事實上，q只是一個int變數。上面這行相當於

int *p, q;

或

int *p; int q;

如果p, q都要宣告成指向int之指標，應寫成：

int *p, *q

或者干脆分兩行寫：

int* p;
int* q;

- 若&前面有資料型態 (ex: int &)，則為參考，&前面有等號 (ex: int* p = &a)，則為取址。
  由於&同時具有多種意義，因此容易造成混淆。這裡列出的這個方法，可以幫助弄清楚每個&的意義。

2. call by value? call by address (或call by pointer)? call by reference? -- swap(int* a, int* b) v.s. swap (int &a, int &b)
JAVA中的reference與C++的reference意義上並不相同，卻使用同一個字，這也是reference容易造成混淆的原因。在此，我們暫不考慮JAVA中reference的觀念 (關於java中reference的觀念，請參考Reference in JAVA -- 淺談java的指標)，純粹把主題放在C/C++上。
呼叫副函式時，call by value, address, 或reference是三種不同的參數傳遞方式。其意義如下：

- call by value
  假設函式A呼叫函式B(p, q)，則B中的p和q是「複製」自函式A所傳入的參數，B中對p, q所做的任何運算都不會影響到A中的p和q，因為B執行完後，並不會把複製的p, q存回到A中。這種參數傳遞方式，我們稱之為call by value。
  以swap這個常見的函式為例，若swap寫成下面的樣子：

```
void swap(int a, int b){
int tmp = a;
a = b;
b = tmp;
}
```

則呼叫

```
swap(x, y)
```

後，x和y的值並不會有變化。

- call by address (或call by pointer)
  利用指標來做參數傳遞，這種方法骨子裡仍是call by value，只不過call by value的value，其資料型態為指標罷了。我們同樣看看用call by address來寫swap交換兩個integer的例子。

```
void swap(int* a, int* b){
int tmp = *a;
*a = *b;
*b = tmp;
}
```

呼 叫swap時，要寫成swap(&x, &y)。呼叫swap時，x的指標 (x的儲存位置) 與y的指標 (y的儲存位置) 會被複製一份到swap中，然後把該位置內所記載的值做更換。swap結束後，&x (address of x) 和&y (address of y) 依然沒變，只是address of x所記錄之變數值與address of y所記錄之變數值交換了。因為&x 和&y 其實是利用call by value在傳，因此，call by address其實骨子裡就是call by value。

- call by reference
  這是C++才加進來的東西，C本身並沒有call by reference。call by reference基本上是把參數做個別名 (alias)，以下面的swap為例：

```
swap(int &a, int &amp;b){
int tmp = a;
a = b;
b = tmp;
}
```

未來使用時，只要呼叫swap(x, y)，就可以讓x和y的值交換。在這個例子中，a 就是 x, b 就是 y。這個觀念在上一節已經提過，在此不再贅述。

3. pointer to pointer, reference to pointer (int** v.s. int*&)
當 我們用call by pointer (或address) 來傳遞參數時，被呼叫的函式複製一份pointer的值過去。但是，當我們想在函式內改變pointer的值 (而非pointer所指向之變數的值)，而且改變的效果要能在函式外看得到時，call by pointer就不足夠用了。此時應該用的是"call by pointer to pointer"或"call by reference to pointer"。我們先看下面的例子：

```
int g_int = 0;
void changePtr(int* pInt){
pInt = &g_int;
}
void main(){
int localInt = 1;
int* localPInt = &localInt;
changePtr(localPInt);
printf("%d\n", *localPInt);
}
```

在這個例子中，印出來的數字仍然會是localInt的1，因為changPtr中的pInt是由localPInt「複製」過去的，對pInt做改變並不會反應到localPInt身上。
我們先用pointer to pointer對localPInt做改變，請看下例。

```
int g_int = 0;
void changePtr(int** pInt){
*pInt = &g_int;
}
void main(){
int localInt = 1;
int* localPInt = &localInt;
changePtr(&localPInt);
printf("%d\n", *localPInt);
}
```

本 例中，印出來的數字會是g_int的0。changePtr函式中的pInt是由&localPInt複製所得，因此對pInt做改變並不會影響 main中的&localPInt (資料型態：pointer to pointer to integer)。但在changePtr函式中我們改變的對象是pInt所指向的內容，因此這項改變在main中會顯示出來。

同樣的功能，我們也可改用reference to pointer來完成。但同樣切記，reference是C++才有的功能，因此reference to pointer也只能在支援C++的環境中使用。

```
int g_int = 0;
void changePtr(int* &refPInt){
refPInt = &g_int;
}
void main(){
int localInt = 1;
int* localPInt = &localInt;
changePtr(localPInt);
printf("%d\n", *localPInt);
}
```

這 段程式印出來的數字會是0。因為在changePtr中，我們宣告的參數型態為int* &，即：reference to pointer to integer。因此，main中的localPInt與changePtr函式中的refPInt其實是「同一件東西」。
另一種 常見的混淆是pointer array (指標陣列) 與pointer to pointers，因為兩種都可以寫成**的型式。如，int** 可能是pointer to pointer to integer，也可能是integer pointer array。但pointer array的觀念相對來講要簡單且直觀許多，這裡我們就暫不花篇幅敘述。常見的例子：main(int argc, char** argv)其實應該是main(int argc, char* argv[])。
4. function pointer
變數的指標指向變數的位址，同樣的，function pointer (函式指標) 也是指向函式的位址的指標。
函式指標的加入，讓C/C++的符號更複雜，也使更多人望之而卻步。在說明函式指標的用途前，我們先直接由語法來看看函式指標該怎麼宣告、怎麼理解。
假設有個函式長成下面的樣子：

```
void func1(int int1, char char1);
```

我們想宣告一個能指向func1的指標，則寫成下面這樣：

```
void (*funcPtr1)(int, char);
```

這樣的寫法應理解成：funcPtr1是一個函數指標，它指向的函數接受int與char兩個參數並回傳void。如果今天有另一個函式長成

```
void func2(int int2, char char2);
```

則funcPtr1也能指向func2。
指標指向的方法，寫成下面這樣：

```
funcPtr1 = &func1;
```

取址符號省略亦可，效果相同：

```
funcPtr1 = func1;
```

若欲在宣告時就直接給予初值，則寫成下面這樣：

```
void (*funcPtr1)(int, char) = &func1;   //&亦可省略
```

stdlib.h中提供的qsort函式是函式指標最常見的應用之一。此函式之prototype長得如下：

```
void qsort(void* base, size_t n, size_t size, int (*cmp)(const void*, const void*));
```

其中的int (*cmp)(const void*, const void*) 就使用到函式指標。
函式指標常見的使用時機是multithread時。函數指標負責把函數傳進建立執行緒的API中。
另外，callback function也是常使用函式指標的地方。所謂callback function即：發生某事件時，自動執行某些動作。在event driven的環境中，便時常使用callback function來實現此機制。
事實上，函式指標還能讓C語言實作polymorphism。但礙於篇幅，在此不再詳述。
5. void ** (*d) (int &, char **(*)(char *, char **))....如何看懂複雜的宣告…
在這裡，我們介紹兩種方式來看懂複雜的宣告。第一種要判斷的是：常數與指標混合使用時，到底const修飾的是指標還是指標所指的變數? 第二種是面對如標題所示這種複雜的宣告時，我們要怎麼讀懂它。
5.1 常數與指標的讀法

```
const double *ptr;
double *const ptr;
double const* ptr;
const double *const ptr;
```

以上幾個宣告，到底const修飾的對象是指標，還是指標所指向的變數呢?
其實，關鍵在於：*與const的前後關係!
當*在const之前，則是常數指標，反之則為常數變數。因此，

```
const double *ptr;   // ptr指向常數變數
double *const ptr;   // ptr是常數指標
```

double const* ptr; // ptr指向常數變數
const double *const ptr; // 指向常數變數的常數指標

事實上，在The C++ Programming Language中有提到一個簡單的要訣：由右向左讀!!讓我們用這個要訣再來試一次。

const double *ptr; // ptr is a pointer points to double, which is a constant
double *const ptr; // ptr is a constant pointer points to double
double const* ptr; // ptr is a pointer points to constant double
const double *const ptr; // ptr is a constant pointer points to double, which is a constant

結果完全相同 :-)
5.2 複雜宣告的讀法 void ** (*d) (int &, char **(*)(char *, char **)).....
其實閱讀C/C++中複雜的宣告有點像是讀英文的長句子，看多了，自然知道句子是怎麼構造出來的。
但對於句子還不熟的人，難免得藉助文法來拆解一個句子。關於C語言複雜宣告的解析文法，最令我印象深刻的，莫過於印度工程師Vikram的"The right-left rule"。他是這麼說的：
「從最內層的括號讀起，變數名稱，然後往右，遇到括號就往左。當括號內的東西都解讀完畢了，就跳出括號繼續未完成的部份，重覆上面的步驟直到解讀完畢。」
舉個例子：void ** (*d) (int &, char*)依下面方式解讀：
1. 最內層括號的讀起，變數名稱: d
2. 往右直到碰到) : (空白)
3. 往左直到碰到(:是一個函數指標
4. 跳出括號，往右，碰到(int &, char*): 此函式接受兩個參數：第一個參數是reference to integer，第二個參數是character pointer。
5. 往左遇上void **: 此函式回傳的型態為pointer to pointer to void。
==> d是一個函式指標，指向的函式接受int&和char*兩個參數並回傳void**的型態。
如何，是不是好懂很多了呢?
標題中的void ** (*d) (int &, char **(*)(char *, char **))其實和上面的例子幾乎一樣，只是函式的第二個參數又是一個函式指標，接受char*和char**兩個參數並回傳char**的型態。

回首頁

© 2020 點部落 Ver. 2020.3.14.1 

電魔小鋪有限公司 製作、維運；登豐數位科技 提供資安檢測

PDFmyURL.com - convert URLs, web pages or even full websites to PDF online. Easy API for developers!