# 高效C語言技巧

其他 · 發表 2019-01-12

原文:
https://blog.csdn.net/wangliang888888/article/details/51302375

一段完美的程式碼不僅在於找到一個給定的問題的解決方案，但在它的簡單性，有效性，緊湊性和效率（記憶體）。設計的程式碼比實際執行更難。因此，每一個程式設計師當用C語言開發時，都應該保持這些基本的東西在頭腦中。本文向你介紹規範你的C程式碼的幾種方法。

1、在可能的情況下使用typedef替代macro.當然有時候你無法避免macro,但是typedef更好。

typedef int* INT_PTR; INT_PTR a ,b;

# define INT_PTR int*; INT_PTR a ,b;

在這個巨集定義中，a是一個指向整數的指標，而b是隻有一個整數宣告。使用typedef a和b都是整數的指標。

2、在一個邏輯條件語句中常數項永遠在左側。

int x = 4; if （x = 1）

{

x = x + 2;

printf（"%d",x）； // Output is 3 }

int x = 4;

if （1 = x）

{

x = x + 2;

printf（"%d",x）； // Compilation error

}

使用"="賦值運算子，替代"=="相等運算子，這是個常見的輸入錯誤。常數項放在左側，將產生一個編譯時錯誤，讓你輕鬆捕獲你的錯誤。注："="是賦值運算子。b = 1會設定變數b等於值1。"=="相等運算子。如果左側等於右側，返回true,否則返回false。

3、確保宣告和定義是靜態的，除非您希望從不同的檔案中呼叫該函式。

在同一檔案函式對其他函式可見，才稱之為靜態函式。它限制其他訪問內部函式，如果我們希望從外界隱藏該函式。現在我們並不需要為內部函式建立標頭檔案，其他看不到該函式。靜態宣告一個函式的優點包括：

（1）兩個或兩個以上具有相同名稱的靜態函式，可用於在不同的檔案。

（2）編譯消耗減少，因為沒有外部符號處理。

4、節約記憶體（記憶體對齊和填充的概念）

struct { char c; int i;

short s;}str_1;

struct { char c; shorts; inti;}str_2;

假設一個字元需要1個位元組，short佔用2個位元組和int需要4位元組的記憶體。起初，我們會認為上面定義的結構是相同的，因此佔據相同數量的記憶體。然而，而str_1佔用12個位元組，第二個結構只需要8個位元組？這怎麼可能呢？

請注意，在第一個結構，3個不同的4個位元組被分配到三種資料型別，而在第二個結構的前4個自己char和short可以被採用，int可以採納在第二個的4個位元組邊界（一共8個位元組）

5、使用無符號整數，而不是整數的，如果你知道的值將永遠是否定的。

有些處理器可以處理無符號的整數，比有符號整數的運算速度要快。（這也是很好的實踐，幫助self-documenting程式碼）

6、switch-case語句。在程式中經常會使用switch-case語句，每一個由機器語言實現的測試和跳轉僅僅是為了決定下一步要做什麼，就浪費了處理器時間。為了提高速度，可以把具體的情況按照它們發生的相對頻率排序。即把最可能發生的情況放在第一，最不可能發生的情況放在最後，這樣會減少平均的程式碼執行時間。

當switch語句中的case標號很多時，為了減少比較的次數，明智的做法是把大switch語句轉為巢狀switch語句。把發生頻率高的case標號放在一個switch語句中，並且是巢狀switch語句的最外層，發生相對頻率相對低的case標號放在另一個switch語句中。比如，下面的程式段把相對發生頻率低的情況放在預設的case標號內。

```
pMsg=ReceiveMessage();
switch (pMsg->type)
{
case FREQUENT_MSG1:
handleFrequentMsg();
break;
case FREQUENT_MSG2:
handleFrequentMsg2();
break;
......
case FREQUENT_MSGn:
handleFrequentMsgn();
break;
default: //巢狀部分用來處理不經常發生的訊息
switch (pMsg->type)
{
case INFREQUENT_MSG1:
handleInfrequentMsg1();
break;
case INFREQUENT_MSG2:
handleInfrequentMsg2();
break;
......
case INFREQUENT_MSGm:
handleInfrequentMsgm();
break;
}
}
```

如果switch中每一種情況下都有很多的工作要做，那麼把整個switch語句用一個指向函式指標的表來替換會更加有效，比如下面的switch語句，有三種情況：

```
enum MsgType{Msg1, Msg2, Msg3}
switch (ReceiveMessage())
{
```

```
case Msg1;.....

case Msg2;.....

case Msg3;.....

}
```

為了提高執行速度，用下面這段程式碼來替換這個上面的switch語句

```
/*準備工作*/

int handleMsg1(void);

int handleMsg2(void);

int handleMsg3(void);

/*建立一個函式指標陣列*/

int (*MsgFunction [])()=
{handleMsg1, handleMsg2, handleMsg3};

/*用下面這行更有效的程式碼來替換switch語句*/

status=MsgFunction[ReceiveMessage()]();
```

7、全域性變數。使用全域性變數比向函式傳遞引數更加有效率，這樣做去除了函式呼叫前引數入棧和函式完成後引數出棧的需要。當然，使用全域性變數會對程式有一些負作用。使用全域性變數比函式傳遞引數更加有效率。這樣做去除了函式呼叫引數入棧和函式完成後引數出棧所需要的時間。然而決定使用全域性變數會影響程式的模組化和重入，故要慎重使用。

8、嵌入式系統程式設計應避免使用標準庫例程，因為很多大的庫例程設法處理所有可能的情況，所以佔用了龐大的記憶體空間，因而應儘可能地減少使用標準庫例程。

9、Inline函式。在C++中，關鍵字Inline可以被加入到任何函式的宣告中。這個關鍵字請求編譯器用函式內部的程式碼替換所有對於指出的函式的呼叫。這樣做在兩個方面快於函式呼叫。這樣做在兩個方面快於函式呼叫：第一，省去了呼叫指令需要的執行時間；第二，省去了傳遞變元和傳遞過程需要的時間。但是使用這種方法在優化程式速度的同時，程式長度變大了，因此需要更多的ROM。使用這種優化在Inline函式頻繁呼叫並且只包含幾行程式碼的時候是最有效的。

10、用指標代替陣列。在許多種情況下，可以用指標運算代替陣列索引，這樣做常常能產生又快又短的程式碼。與陣列索引相比，指標一般能使程式碼速度更快，佔用空間更少。使用多維陣列時差異更明顯。下面的程式碼作用是相同的，但是效率不一樣。

```
陣列索引                    指標運算

For(;;){                    p=array

A=array[t++];               for(;;){

                              a=*(p++);
```

.....  .....

}

指標方法的優點是，array的地址每次裝入地址p後，在每次迴圈中只需對p增量操作。在陣列索引方法中，每次迴圈中都必須進行基於t值求陣列下標的複雜運算。

11、不定義不使用的返回值。function函式定義並不知道函式返回值是否被使用，假如返回值從來不會被用到，應該使用void來明確宣告函式不返回任何值。

12、手動編寫彙編。在嵌入式軟體開發中，一些軟體模組最好用匯編語言來寫，這可以使程式更加有效。雖然C/C++編譯器對程式碼進行了優化，但是適當的使用內聯彙編指令可以有效的提高整個系統執行的效率。

13、使用暫存器變數。在宣告區域性變數的時候可以使用register關鍵字。這就使得編譯器把變數放入一個多用途的暫存器中，而不是在堆疊中，合理使用這種方法可以提高執行速度。函式呼叫越是頻繁，越是可能提高程式碼的速度。

14、使用增量和減量操作符。在使用到加一和減一操作時儘量使用增量和減量操作符，因為增量符語句比賦值語句更快，原因在於對大多數CPU來說，對記憶體字的增、減量操作不必明顯地使用取記憶體和寫記憶體的指令，比如下面這條語句：
x=x+1;

模仿大多數微機組合語言為例，產生的程式碼類似於：

move A,x ;把x從記憶體取出存入累加A

add A,1 ;累加器A加1

store x ;把新值存回x

如果使用增量操作符，生成的程式碼下：

incr x ;x加1

顯然，不用取指令和存指令，增、減量操作執行的速度加快，同時長度也縮短了。

標籤：

R語言技巧：讀csv格式的檔案
高精度乘法（高精乘高精）（C語言實現）
怎樣將kux格式轉換mp4？高效簡單的技巧你要懂
求解二叉樹的深度（高度）_C語言
讓你提高效率的 Linux 技巧
高效率的Word技巧趕快學起來，快速節省工作時間不再加班！
C語言已死（連載1）——趣味、通俗、實用的計算機達人成長之路之C語言高階技巧篇

首頁
Python教學