**直接列出考古題1(精華完整48題含解答)**

**1. 物件導向的三大特性**

封裝、繼承、多型為物件導向三大基礎。  
此三者具有次序性，沒有封裝就不可能有繼承、沒有繼承就不可能有多型。

**封裝(Encapsulation):**  
就是把不必要的資訊隱藏(Information Hiding)起來,只把必要的操作開放出去。譬如開車加速,我們只要知道踩油門就好,不需知道細節(內部零件間怎麼協同完成這件事),這樣的好處是物件間或模組間的藕合力(Coupling)低,若今天要完成一件事的細節變了,呼叫者可以完全不受影響,藕合力愈低,單元測試愈好寫啊…當你呼叫一個物件的get,就要小心也許你正在破壞他的封裝!

**繼承(Inheritance):**  
我們都知道就是父子關係(IS-A relationship),子類別會繼承父類的方法和屬性,繼承本身並沒有問題,但常會見到誤用的情形!就是開發者忘了IS-A的概念,為了reuse,造成了功能型的繼承!若要reuse且非IS-A關係,應用Composition來達成,把共同的部份移至composed class,把要做的事delegate另一個物件完成!

**多型(Polymorphism):**  
延申自繼承(Inheritance)或介面(Interface),指的就是不同型態的物件,定義相同的操作介面,由於被呼叫者(Callee)有著相同介面,呼叫者並不用指定特定型別,只需針對介面進行操作,實際執行的物件則在runtime決定,藉此增加程式碼的彈性。例如: 超載 (Overloading)與重寫 (Overriding)相關，和上面的繼承更是息息相關。一個物件如果你看待的方式不同，就可以發揮不同的功用。這在現實生活中很合理，一個碗可以拿來當容器，也可以拿來盛土種花，當然要倒過來當蓋子也行，甚至，你要拿來砸碎當發聲的樂器也不犯法。因此同一個物件，在不同的場合，不同的使用情境，就應該合理運作展現應有的效果，這也就是多型的精神。

**2.Overloading 和 Overriding 的區別。Overloading 的方法是否可以改變返回值的類型?**

Overloading 發生在同一類別裡，Overriding 發生在繼承關係間。  
Overloading 為靜態連結，Overriding 為動態連結。  
可改變返回值的型態，前提是參數也要變，參數相同傳回值不同是不被允許的。

**3.write your own strcmp**

如果字串比較結果一樣，便會傳回 0  
若字串不同則：  
字串1 > 字串2的話，會傳回大於 0 的值 1  
字串1 < 字串2的話，會傳回小於 0 的值 -1



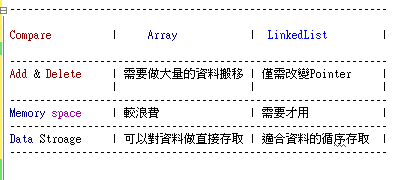
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | int ownstrcmp(char a[], char b[])  {    int i = 0;    while( a[i] == b[i] )    {      if( a[i] == '\0' )        return 0;      ++i;    }    return ( a[i] < b[i]) ? -1 : 1;  } |

**4. what is different between mutex and semaphore?**

Mutex是一把鑰匙，一個人拿了就可進入一個房間，出來的時候把鑰匙交給隊列的第一個。一般的用法是用於串行化對critical section代碼的訪問，保證這段代碼不會被並行的運行。 (A mutex is really a semaphore with value 1.)  
Semaphore是一件可以容納N人的房間，如果人不滿就可以進去，如果人滿了，就要等待有人出來。對於N=1的情況，稱為binary semaphore。一般的用法是，用於限制對於某一資源的同時訪問。

**5. Compare array(陣列) and Linked list(鏈節串列)**

**陣列(array)是什麼 ?**  
1. 一種結構性的資料儲存空間，其同一陣列裡的資料性質呈一致性  
2. 元素與元素之間的記憶位置是相鄰的  
3. 通常我們利用一個變數來代表整體的資料。  
舉例而言，我們可以把陣列想成一群鳥窩，而陣列裡的變數個數代表鳥窩的數目，例如：麻雀[20]，我們可以想成麻雀的窩總共有二十間，每一間住著一間麻雀，假如我們想要知道第三間麻雀的名字，只要把麻雀[2]的值取出，便可知道住在第三間麻雀的姓名。C語言的陣列索引一定是從0的開始的。

**Linked List (鏈節串列)是什麼 ?**  
1. 一種結構將許多一樣的東西串起來，而這些一樣的東西叫做結點 (Node)，然後透過Pointer 連起來。  
2. 在操作上比陣列彈性許多，像:加入/刪除，你可以在任何你想要的地方 加入Node和刪除Node。  
3. 有需要的時候才動態配至記憶體空間 。  
4. 可以取代陣列儲存格式(堆疊和佇列)。  
[](https://3.bp.blogspot.com/-x_D7VLqFImc/Wl2dHKMmuWI/AAAAAAAAI5s/iI3sDA8MbNAK1pIf7r0jxVkcDeVGdr23ACLcBGAs/s1600/arlit.png)

**6.Compare stack and queue**

佇列(Queue)是用先進先出的方式處理物件的集合，例如到銀行排隊，先排的人先處理；  
而堆疊(Stack )是後進先出的集合，例如玩撲克牌排遊戲時，發牌時是從整疊的最上一張拿取。

**7.write a function that can calculate 1\*2+2\*3+…..+(n-1)\*n**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int nc(int n)  {  int sum = 0;  for(int i = 2; i <= n; i++){    sum = sum + i\*(i-1);  }  return sum;  } |

類似的, 改成乘法  
寫一個 function 可傳入正整數參數 N，回傳 1 + 2 + 3 + … + N 的和



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int Fab (int n) {     if (n <= 0)        printf("N should larger than 0 ! \n");     else if( n == 1)        return 1;     else        return n + Fab(n-1);  } |

**8. Explain Static and volatile**

**Static:**  
(1) 修飾檔案中的global variable：  
使這個變數只有在本檔案中才可以被使用，相同專案中的其他檔案看不到它的存在。 (補：放在function前也有一樣的作用。)

(2) 修飾function中的local variable：  
此變數一旦經過初始化就會一直存在直到程式結束，跳出function時它也會保持當下的值，(ex. 可用來計算同一個function被呼叫的次數。) 只會被初始化一次，並且只有進入function中才看得到這個變數 !!

(3) 修飾class中的member variable和 function：  
variable：會使同一個class的所有實體共用同一個member variable，或者說這個member variable在同一個class的所有實體擁有相同的值。 一樣只會初始化一次，甚至不需要實體就可呼叫。  
function：static member function不屬於任何一個實體，也是不需要實體就可呼叫，但它只能操作static member variables而已。

他們都透過 :: 運算子來呼叫，表示屬於某一個class但不屬於任何實體。 ex. A::x  
也可以透過實體用 . 運算子呼叫，但觀念上比較不好！

**Volatile:**  
被volatile修飾的變數代表它的值有可能因為編譯器不知道的因素修改，所以告訴編譯器不要對它涉及的地方做最佳化，並在每次操作它的時候都去讀取該變數實體位址上最新的值，而不是讀取CPU暫存器上的值，一般的變數可能因為剛剛讀取過而放在CPU暫存器上使動作變快。

例子：  
(1) 硬體暫存器，如狀態暫存器。  
(2) 多執行緒所共用的全域變數。  
(3) 中斷服務函式 (Interrupt Service Rountie，ISR)所使用的全域變數。

**Volatile陷阱, 想想底下範例是否有問題?**([參考](http://adrianhuang.blogspot.tw/2011/08/cvolatile.html))  
假設有個函數



|  |  |
| --- | --- |
| 1  2  3 | int square(volatile int \*ptr) {     return \*ptr \* \*ptr;  } |

其問題在於square函式的平方算式，\*ptr\*\*ptr，此指令代表到ptr位址讀取其內容。  
然而，ptr位址可能儲存硬體暫存器，這些暫存器內容會隨時間而改變 (例如: 狀態暫存器)，有可能第一次讀取的時候為4, 下一次讀取為5, 導致計算出來的值不正確。 則編譯器可能會把它解讀成下列code



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int square(volatile int \*ptr) {     int a,b;     a = \*ptr;     b = \*ptr;     return a \* b;  } |

實際上產生的可能不是平方值，所以改成下列方式宣告一local變數才不會出錯



|  |  |
| --- | --- |
| 1  2  3  4  5 | int square(volatile int \*ptr) {     int a;     a = \*ptr;     return a \* a;  } |

**C/C++中的volatile使用時機?**  
不知各位對volatile(揮發性的)這個字陌不陌生? 我相信大家在一些程式或多或少都看過這個字眼, 但是究竟要在何種場合用它呢? 當然一定是有需要, C/C++才會有這個保留字, 否則只是增加programmer的困擾而已

有兩個場合(I/O & multithread program), 供各位參考! 請大家check自己的程式中(尤其是第2個場合), 若有的話請記得加上volatile

1. I/O, 假設有一程式片斷如下



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | U8 \*pPort;  U8 i, j, k;    pPort = (U8 \*)0x800000;    i = \*pPort;  j = \*pPort;  k = \*pPort; |

以上的i, j, k很有可能被compiler最佳化而導致產生  
i = j = k = \*pPort;  
的code, 也就是說只從pPort讀取一次, 而產生 i = j = k 的結果, 但是原本的程式的目的是要從同一個I/O port讀取3次的值給不同的變數, i, j, k的值很可能不同(例如從此 I/O port 讀取溫度), 因此i = j = k的結果不是我們所要的, 那怎麼辦?



|  |  |
| --- | --- |
| 1  2  3  4 | 用volatile, 將  U8 \*pPort;  改為  volatile U8 \*pPort; |

告訴compiler, pPort變數具有揮發性的特性, 所以與它有關的程式碼請不要作最佳化動作. 因而  
i = \*pPort;  
j = \*pPort;  
k = \*pPort;  
此三列程式所產生的code, 會真正地從pPort讀取三次, 從而產生正確的結果

2. Global variables in Multithread program  
=> 這是在撰寫multithread program時最容易被忽略的一部份  
=> 此原因所造成的bug通常相當難解決(因為不穩定)

假設有以下程式片斷, thread 1 & thread 2 共用一個global var: gData



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | thread 1: thread 2:    ... ....  int gData;  extern int gData;    while (1)  {      int i, j, k;      gData = rand();      ....      for (i = 0; i < 1000; i++)      {       .....       /\* A \*/       j = gData;       ....       ....      }  } |

在thread 2的for loop中, 聰明的compiler看到gData的值, 每次都重新從memory load 到register, 實在沒效率, 因此會產生如下的code(注意,tmp也可以更進一步的用register取代):



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | tmp = gData;  for (i = 0; i < 1000; i++)  {      /\* A \*/      j = tmp;      ....  } |

也就是gData只讀取一次, 這下子問題來了, 說明如下:  
. thread 2在執行for loop到j = gData 的前一列(A)的時候(假設此時gData=tmp=5), 被切換到thread 1執行  
. 在thread 1的while loop中透過gData = rand(), 對gData做了修改(假設改為1), 再切換回thread 2執行  
. 繼續執行 j = gData, 產生j = 5的結果  
. 但是正確的結果應該是 j = 1 怎麼辦 => **也是用volatile**,



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 在thread 1中, 將  int gData;  改為  volatile int gData;    在thread 2中, 將  extern int gData;  改為  extern volatile int gData; |

類似有:  
Q: C語言的volatile的含義是什麼。使用時會對編譯器有什麼暗示。  
A:  
那麼這道題非常重要！！嵌入式系統程式師經常同硬體、中斷、RTOS等等打交道，所用這些都要求volatile變數。  
從詞面上講，volatile的意思是易變的，也就是說，在程式運行過程中，有一些變數可能會被莫名其妙的改變，而優化器為了節約時間，有時候不會重讀這個變數的真實值，而是去讀在寄存器的備份，這樣的話，這個變數的真實值反而被優化器給“優 化”掉了，用時髦的詞說就是被“和諧”了。

如果使用了這個修飾詞，就是通知編譯器別犯懶，老老實實去重新讀一遍！可能我說的太“通俗”了，那麼我引用一下 “大師”的標準解釋：  
volatile的本意是“易變的” 。  
由於訪問寄存器的速度要快過RAM,所以編譯器一般都會作減少存取外部RAM的優化，但有可能會讀髒資料。當要求使用volatile 聲明的變數的值的時候，系統總是重新從它所在的記憶體讀取資料，即使它前面的指令剛剛從該處讀取過資料。而且讀取的資料立刻被保存。

精確地說就是，優化器在用到這個變數時必須每次都小心地重新讀取這個變數的值，而不是使用保存在寄存器裏的備份。  
下面是volatile變數的幾個例子：  
1). 並行設備的硬體寄存器（如：狀態寄存器）  
2). 一個中斷服務副程式中會訪問到的非自動變數(Non-automatic variables)  
3). 多執行緒(Thread)應用中被幾個任務共用的變數

**9. Explain process and thread**

**Program：**  
放在二次儲存裝置中，尚沒有被Load到記憶體的一堆Code稱之為「程式」。 (也就是還是死的)

**Process：**  
已經被Load到記憶體中，任何一行Code隨時會被CPU執行，且其宣告的在記憶體的變數的值會隨著需求而不斷變動。稱之為「程序」。 (也就是活的Program) => 恐龍本第三章  
一個多工作業系統(Multitasking Operating System)可以同時運行多個Process, 然而一個CPU一次只能做一件事情，但CPU的數量永遠少於運行中的Process數，因此每個Process使用的時間需要被排程(Scheduling) => 恐龍本第五章  
又每個Process間在記憶體中，如果擺放的方式不當，就會在記憶體中產生很多沒辦法用到的碎片，因此MemoryManagement是一個問題 => 恐龍本第八章  
另外，每個Process所需要的記憶體總合，也可能大於實體記憶體，因此需要另外用二次儲存裝置充當**虛擬記憶體(Virtual Memory)**，但是二次儲存裝置的速度肯定很慢，因此如何做到對虛擬記憶體最小的依賴，盡量避免Page Fault(電腦在主記憶體中找不到資料，而要去二次記憶體找，就稱為Page Fault), 防止Thrashing的發生(因為Virtual Memory演算法不當，造成幾乎每次存取都要依賴二次記憶體，就是Thrashing)，以達到效能最佳化，也是個學問 => 第九章

**Thread ：**  
在同一個Process底下，有許多自己的分身，就是Thread，中文又翻成執行緒。以往一個Process一次只能做一件事情，因此要一面輸入文字，一面計算字數，這種事情是不可能的。但是有了Thread之後，可以在同一個Process底下，讓輸入文字是一個Thread，計算文字又是另外一個Thread，對CPU來說兩個都是類似一個Process，因此兩個可以同時做。  
又一個Process底下有數個Thread，而一個Process的Global Variable可以讓它的所有Thread共享，也就是所有Thread都**可以存取同一個Process的Global Variable**。而每個Thread自己也有自己的專屬Variable。 => 恐龍本第四章  
但是，如果有兩個Thread要存取同一個Global Variable，有可能發生問題，也就是說可能會存取到錯的值(例如兩個Thread同時要對一個Variable做加減，最後那個答案可能會是錯的)，這就是Synchronization問題 =>恐龍本第六章  
又，每一個Thread之間可能會互搶資源，而造成死結(Deadlock)，只要以下四個條件都滿足就有死結。(1)這個資源不能同時給兩個人用 (2)有一個人拿了一個資源，又想拿別人的資源 (3)如果一個人占了茅坑不拉屎，占用資源很久，仍不能趕他走 (4)A等B，B等C，C等D，D又等A 等成一圈。 要解決這種狀況有 Avoid(預防) 或 避免(Prevent)兩種方式，破除以上四種其中一種即可。=> 恐龍本第七章

**10. 以下程式碼印出來為何?**

printf(“size of BYTE = %d\n \  
size of float = %d\n \  
size of unsigned int = %d\n \  
size of int = %d\n \  
size of double = %d\n \  
size of unsigned char = %d\n \  
size of char = %d\n”  
,sizeof(BYTE)  
,sizeof(float)  
,sizeof(unsigned int)  
,sizeof(int)  
,sizeof(double)  
,sizeof(unsigned char)  
,sizeof(char));



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | size of BYTE = 1  size of float = 4  size of unsigned int = 4  size of int = 4  size of double = 8  size of unsigned char = 1  size of char = 1 |

補充: C語言所定義的資料型別如下

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 型別 | 符號位元 | 位元長度 | 表示方法 | 數值範圍 |
| 整數 | 有 | 16或32 | int | -2147483648 ~ 2147483647 |
| 8 | char | -128 ~ 127 |
| 16 | short | -32768 ~ 32767 |
| 32 | long | -2147483648 ~ 2147483647 |
| 64 | long long |  |
| 無 | 16或32 | unsigned int | 0 ~ 4294967295 |
| 8 | unsigned char | 0 ~ 255 |
| 16 | unsigned short | 0 ~ 65535 |
| 32 | unsigned long | 0 ~ 4294967295 |
| 64 | unsigned long long |  |
| 浮點數 | 有 | 32 | float | 10^-38~10^38 |
| 64 | double | 10^-308~10^308 |
| 字元 | 有 | 8 | char | -128 ~ 127 |

**11.請試著寫出下面代碼的輸出:**

view plaincopy to clipboardprint?

int main()  
{  
char strAry[] = “This is string”;  
char \*aryPtr = strAry;  
int \*intPtr = (int\*)strAry;

printf(“\t[Line01] strAry=%s\n”, strAry);  
printf(“\t[Line02] aryPtr=%s\n”, aryPtr);  
//printf(“\t[LineX] \*aryPtr=%s\n”, \*aryPtr); // Segment fault  
printf(“\t[Line03] sizeof(aryPtr)=%d\n”, sizeof(aryPtr));  
printf(“\t[Line04] sizeof(\*aryPtr)=%d\n”, sizeof(\*aryPtr));  
printf(“\t[Line05] \*aryPtr=’%c’\n”, \*aryPtr);  
printf(“\t[Line06] \*aryPtr+1=’%c’\n”, \*aryPtr+1);  
printf(“\t[Line07] \*(aryPtr+1)=’%c’\n”, \*(aryPtr+1));  
printf(“\t[Line08] sizeof(intPtr)=%d\n”, sizeof(intPtr));  
printf(“\t[Line09] sizeof(\*intPtr)=%d\n”, sizeof(\*intPtr));  
printf(“\t[Line10] intPtr=%s\n”, intPtr);  
//printf(“\t[LineX] \*intPtr=%s\n”, \*intPtr); // Segment fault  
printf(“\t[Line11] \*intPtr=’%c’\n”, \*intPtr);  
printf(“\t[Line12] \*intPtr+1=’%c’\n”, \*intPtr+1);  
printf(“\t[Line13] \*(intPtr+1)=’%c’\n”, \*(intPtr+1));  
return 0;  
}

Sol:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | [Line01] strAry=This is string  ### 字串陣列 char[] ="..." 會自動加上 NULL 到結尾.  [Line02] aryPtr=This is string  ### 同上, 只是把 aryPtr 指標指向 strAry 的位置. strAry 本身也是個指標.  [Line03] sizeof(aryPtr)=4       ### 指標的大小根據系統是 32bit (4byte) 或是 64bit(8bypte) 有所不同.  [Line04] sizeof(\*aryPtr)=1      ### char 的大小為 1 byte.  [Line05] \*aryPtr='T'            ### 指向字串中第一個字元 'T'  [Line06] \*aryPtr+1='U'          ### char 'T' + 1=char 'U'. -&gt; ASCII 'T'=84. 84+1=85=ASCII 'U'.  [Line07] \*(aryPtr+1)='h'        ### 將 aryPtr 指標移動一個 char 的位置 (1 個 byte 的距離), 即是字串的第二個字元 'h'.  [Line08] sizeof(intPtr)=4       ### 同 Line03  [Line09] sizeof(\*intPtr)=4      ### int 類型的大小為 4 byte.  [Line10] intPtr=This is string  ### 雖然用 int\* 指定 pointer 類型, 但是在 printf 使用 '%s', 故還是打印出字串出來.  [Line11] \*intPtr='T'            ### 指向字串中第一個字元 'T'.  [Line12] \*intPtr+1='U'          ### 同 Line6  [Line13] \*(intPtr+1)=' '        ### 因為 指標類型為 int, 故移動一個位置為 4 byte, 所以指向第 0+4 =4 位置上的字元, 即字串的第五個字元 (從 0 開始). |

同樣類型題目, 程式碼輸出為何?  
int a[5] ={1,2,3,4,5};  
int \*p = (int \*)(&a+1);  
ask: the value of \*(a+1), (\*p-1)?

\*(a+1) : 2  
(\*p-1) : ????????? (undefined behavior)

這題要考你 a+1 和 &a+1 的區別,  
第一種的+1一次是跳一個int的大小(i.e., 4bytes in 32-bits cpu)  
第二種的+1一次是跳一整個array的大小 (i.e., 4\*5=20 bytes)

更清楚的[說明在這](http://stackoverflow.com/questions/2989370/why-do-a1-and-a1-give-different-results-when-a-is-an-int-array" \t "_blank) 可以看到: a 和 &a 的type不同, 而當要 +1 時，要加的是 type的大小  
所以\*(a+1)是2，(\*p-1)是garbage (unknown value)

同樣類型題目, 請用變數 a 給出下面的定義:  
a)一個整型數 (An integer)  
b)一個指向整數的指標 (A pointer to an integer)  
c)一個指向指標的指標，它指向的指標是指向一個整型數 (A pointer to a pointer to an integer)  
d)一個有10個整數型的陣列 (An array of 10 integers)  
e)一個有10個指標的陣列，該指標是指向一個整數型的 (An array of 10 pointers to integers)  
f)一個指向有10個整數型陣列的指標 (A pointer to an array of 10 integers)  
g)一個指向函數的指標，該函數有一個整數型參數並返回一個整數 (A pointer to a function that takes an integer as an argument and returns an integer)  
h)一個有10個指標的陣列，該指標指向一個函數，該函數有一個整數型參數並返回一個整數 (An array of ten pointers to functions that take an integer argument and return an integer)



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 答案是︰  a) int a;             // An integer  b) int \*a;            // A pointer to an integer  c) int \*\*a;           // A pointer to a pointer to an integer  d) int a[10];         // An array of 10 integers  e) int \*a[10];        // An array of 10 pointers to integers  f) int (\*a)[10];      // A pointer to an array of 10 integers  g) int (\*a)(int);     // A pointer to a function a that takes an integer argument and returns an integer  h) int (\*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer |

**12. Write a function which can return the bit content form**

嵌入式系統總是要用戶對變量或暫存器進行位操作。  
給定一個整型變量a，寫兩段程式碼，  
第一個設置a 的bit 3，第二個清除a 的bit 3。  
在以上兩個操作中，要保持其它位不變。



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #define BIT3 (0x1 << 3)  static int a;    void set\_bit3(void)  {      a |= BIT3;  }    void clear\_bit3(void)  {      a &= ~BIT3;  }    void get\_bit3(void)  {      int mask = 1 << 3; int masked\_n = a & mask; int a = masked\_n >>3;  } |

若是題目要求寫成#define, 則方式如下:



|  |  |
| --- | --- |
| 1  2  3  4 | #define SET\_BIT(x, n) ( (x) |= (1<< (n)) )  #define CLR\_BIT(x, n) ( (x) &= (~(1<< (n))))  #define CHK\_BIT(x, n) ( ((x) & (1<< (n)))!=0 ) ==> 檢查是否為 1  #define FLIP\_BIT(x, n) ( (x) ^= (1<< (n)) ) ==> 指的是原先某個位元，從1變成0，從0變成1 |

類似的還有,  
存取固定的記憶體位置 (Accessing fixed memory locations) 嵌入式系統經常具有要求程式員去存取某特定的記憶體位置的特點。  
在某工程中，要求設定一個絕對位址為 0x67a9 的整數型變數的值 為0xaa55。  
編譯器是一個純粹的ANSI編譯器。寫程式碼去完成這一任務。這一問題測試你是否知道為了存取一絕對位址把一個整數型強製轉型 (typecast) 為一指標是合法的。  
典型的類似程式碼如下︰



|  |  |
| --- | --- |
| 1  2  3  4 | int \*ptr;    ptr = (int \*)0x67a9;  \*ptr = 0xaa55; |

一個較艱澀的方法是︰



|  |  |
| --- | --- |
| 1 | \*(int \* const)(0x67a9) = 0xaa55; |

即使你的品味更接近第二種方案，但我建議你在面試時使用第一種方案。

**13. Explain Const**

C 語言中const用來定義常量。const定義的變量在定義時要初始化，否則將會是一個隨機值，而且在定義後其值不能被改變。const在指標(pointer)使用中可以表明指標指到data是const or 指標本身是const or 兩者都是:

1. int \*a ;                      /\* non-const pointer, non-const data \*/
2. const int a;                  /\* const data，a是一個常整型數\*/
3. int const a;                  /\* const data ，a是一個常整型數\*/
4. const int \*a;                 /\* non-const pointer, const data ,  a 是一個指向常整型數的指針（也就是，整型數是不可修改的，但指針可以） \*/
5. int \* const a;                /\* const pointer, non-const data, a 是一個指向整型數的常指針（也就是說，指針指向的整型數是可以修改的，但指針是不可修改的） \*/
6. int const \* a const;    /\* const pointer, const data, a 是一個指向常整型數的常指針（也就是說，指針指向的整型數是不可修改的，同時指針也是不可修改的） \*/
7. const int \* const a ;    /\* const pointer, const data, a 是一個指向常整型數的常指針（也就是說，指針指向的整型數是不可修改的，同時指針也是不可修改的） \*/

相同類型:  
Q: C語言的const的含義是什麼。在定義常量時，為什麼推薦使用const，而不是#define。  
A:  
const修飾詞可以將一個變數修飾為“唯讀”，這個就能稱 為常量麼？姑且認為可以。  
回到題目中，const是唯讀的意思，它限定一個變數不允許被改變，誰都不能改！  
既然是修飾變數，那麼變數的類型就可以豐富多彩，int 啊，char啊，只要C認識的都可以；  
但是#define就不可以了，在預處理階段缺乏類型檢測機制，有可能會出錯。  
還有就是變數可以 extern，但是#define就不可以。

**14. 下面的代碼輸出是什麽，為什麽？**



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | void foo(void)  {     unsigned int a = 6;     int b = -20;     (a+b > 6) ? puts("> 6") : puts("<= 6");  } |

這無符號整型問題的答案是輸出是 "> 6"。  
原因是當表達式中存在有符號類型和無符號類型時所有的操作數都自動轉換為無符號類型。  
因此-20變成了一個非常大的正整數，所以該表達式計算出的結果大於6。  
  
另一個奇怪確合法的語法  
C語言允許一些令人震驚的結構，下面的結構是合法的嗎，如果是，它做些什麼 ?

int a = 5, b = 7, c;  
c = a+++b;  
不管你相不相信，上面的例子是完全合法的。  
問題是編譯器如何處理它 ? 根據最處理原則，編譯器應當能處理儘可能所有合法的用法。  
因此，上面的程式碼被處理成︰  
c = a++ + b;

因此，這段程式碼持行後  
a = 6, b = 7, c = 12

**15.String 跟 StringBuffer差別。**

String 類型和StringBuffer的主要性能區別：  
String是不可變的對象, 因此在每次對 String 類型進行改變的時候，都會生成一個新的 String 對象，然後將指針指向新的 String 對象，所以經常改變內容的字符串最好不要用 String ，因?每次生成對象都會對系統性能產生影響，特別當內存中無引用對象多了以後， JVM 的 GC 就會開始工作，性能就會降低。

使用 StringBuffer 類時，每次都會對 StringBuffer 對象本身進行操作，而不是生成新的對象並改變對象引用。所以多數情況下推薦使用 StringBuffer ，特別是字符串對象經常改變的情況下。  
在某些特別情況下， String 對象的字符串拼接其實是被 JVM 解釋成了 StringBuffer 對象的拼接，所以這些時候 String 對象的速度並不會比 StringBuffer 對象慢，例如：  
String S1 = “This is only a” + “ simple” + “ test”;  
StringBuffer Sb = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);

生成 String S1 對象的速度並不比 StringBuffer慢。其實在 JVM 裏，自動做了如下轉換：  
String S1 = “This is only a” + “ simple” + “test”;  
JVM直接把上述語句當作：  
String S1 = “This is only a simple test”;  
所以速度很快。但要注意的是，如果拼接的字符串來自另外的String對象的話，JVM就不會自動轉換了，速度也就沒那麼快了，例如：

String S2 = “This is only a”;  
String S3 = “ simple”;  
String S4 = “ test”;  
String S1 = S2 +S3 + S4;

這時候，JVM 會規規矩矩的按照原來的方式去做。  
在大部分情況下，StringBuffer > String。

**16. 解釋並舉例多型、封裝、繼承。**

如問題1

**17. 作業系統名詞解釋**

**Context switch:**  
A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU from one process or thread to another.

**中斷（Interrupt）：**  
屬於非同步發生的事件（event），在任何時間都可能發生且 與處理器（processer）在在執行的東西毫無關連，通常它由輸出入裝置（I/O devices），處理計時器，或時序（timers）產生

**陷阱（EXception or Trap）：**  
屬於同步狀態，通常由執行某一特別的指令。陷阱（Trap或exception）可以藉由執行相同資料及狀態下的程式而重複產生。其例有無效記憶體存取或除以零。

**輪詢（polling）：**  
中斷發生時，CPU做輪詢的動作，去查詢所有I/O裝置看誰需要服務。

**interrupt發生後，OS的處理程序** (在monitor area內會存放interrupt vector及各種ISR):  
— Step：  
–1 暫停目前process的執行，並保存當時執行狀況  
–2 根據interrupt ID查詢interrupt vector，取出對應的Interrupt Service Routine(ISR)起始位址  
–3 Jump to ISR的initial address，執行該ISR  
–4 ISR complete  
–5 OS恢復原先中斷前的process執行

Interrupt的種類：  
–1. External interrupt(HW) : CPU以外的周邊元件所發出的, eg. I/O complete、I/O error、machine check  
–2. Internal interrupt(HW) : CPU本身所引發的, eg. stack overflow、illegal command(非法指令執行)、divided by zero(除以0)…  
–3. Software interrupt : 當user program執行時，若需要OS提供服務，則發出此類中斷通知OS執行對應的service routine, eg. system call、trap

**Interrupt與Trap之比較**:  
Interrupt：Hardware generated interrupt, eg. I/O device發出”I/O complete”中斷  
Trap：Software generated interrupt 用途：user program需要OS提供Service時發出, Catch up arithematic error, eg. Divide-by-Zero

**DMA（Direct Memory Access）:**  
高速地將 I/O 資料傳送到記憶體，而不被CPU干涉。  
— Used for high-speed I/O devices able to transmit information at close to memory speeds.  
— Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.  
— Only on interrupt is generated per block， rather than the one interrupt per byte.

**Call Back Function :**  
簡單的說，如果你使用了某個function，那麼你就是『call』了一個function。如果系統或是函式是要求你給一個function pointer，這個function pointer指到一個實際的函式(多半這個函式是你自己寫的)。然後它會在適當的時間呼叫此function，則此function就是所謂的 callback function。因為這個function是被『callback』了。

**18. 請說明 C/C++ 字串格式化內容與指標使用:**

scanf ( " %d,%f" ,&a ,&b)  
=====================================================  
%c 字元 輸入一字元到指定位址  
%s 字串指標 輸入一字串到指定位址內  
%d 整數 輸入一整數值到指定變數位址內  
%i 整數 輸入十進位,八進位,十六進位整數值到指定變數位址內  
%o 整數 輸入八進位值到指定變數位址內  
%u 整數 輸入一無正負號十進位整數到指定變數位址  
%x 整數 輸入十六進位值到指定變數位址內  
%e 浮點數 輸入一浮點數到指定變數位址內,可用指示  
%f 浮點數 輸入一浮點數到指定變數位址內  
=====================================================

指標:  
& : 用來取得變數位址  
\* :用來取得指表所指變數的內含值及定義指標  
int a, \*p;  
p = &a;

**19. 預處理器 (Preprocessor)**

用預處理指令#define 聲明一個常數，用以表示1年中有多少秒 (忽略閏年問題):



|  |  |
| --- | --- |
| 1 | #define SECONDS\_PER\_YEAR (60 \* 60 \* 24 \* 365)UL |

- 懂得預處理器將為你計算常數表達式的值，因此，直接寫出你是如何計算一年中有多少秒會比直接計算出實際的值更清晰。  
- 意識到這個表達式將使一個16位元的機器產生整數型溢位 - 因此要用到長整型符號L，告訴編譯器這個常數是的長整型數。  
- 如果你在你的表達式中用到UL (表示無符號長整型) ，那麼你有了一個好的起點。記住，第一印象很重要。

寫一個“標準”巨集MIN ，這個巨集輸入兩個參數並返回較小的一個:



|  |  |
| --- | --- |
| 1 | #define MIN(A，B)  ( (A)  &lt;= (B) ? (A) : (B)) |

這個測試是為下面的目的而設的︰  
∙ 標識#define在巨集中應用的基本知識。這是很重要的，因為在行內(inline)運算子變為標準C的一部分之前，巨集是方便產生行內程式碼的唯一方法，對於嵌入式系統來說，為了能達到要求的性能，行內程式碼經常是必須的方法。  
∙ 三元運算子的知識。這個運算子存在C語言中的原因是它使得編譯器能產生比if-then-else更優化的程式碼，了解這個用法是很重要的。  
∙ 懂得在巨集中小心地把參數用括號括起來

**預處理器許多標識的意義?**  
\_\_FILE\_\_ ：此標誌會在预编译时会替换成当前的源文件名  
\_\_LINE\_\_ ：此標誌會在预编译时会替换成当前的行号  
\_\_FUNCTION\_\_：此標誌會在预编译时会替换成当前的函数名称  
defined() : 檢查某的巨集定義是否被定義了，通常與 #if 合用，像是 #if defined(OS) …  
#error : 在編譯時，輸出錯誤訊息，警告使用者某些錯誤，並且不會真的進行編譯，在巨集處理階段就會停止。  
就是在編譯前程式設計師就預測在某種情況下會發生錯誤, 所以compiler看到它就會停掉啦, 舉例如下:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #if defined(BUILD\_TYPE\_NORMAL)              # define DEBUG(x) do {;} while (0) /\* paranoid-style null code \*/              #elif defined(BUILD\_TYPE\_DEBUG)              # define DEBUG(x) \_debug\_trace x /\* e.g. DEBUG((\_debug\_trace args)) \*/              #else              # error "Please specify build type in the Makefile"              #endif |

當預處理打#error指令，它會報告字符串作為錯誤信息，並停止編譯;究竟該錯誤信息是這樣依賴於編譯器。

#warning : 在編譯時，輸出警告訊息，警告使用者某些注意事項，但是不會中止編譯，仍然會繼續編譯出目的檔。



|  |  |
| --- | --- |
| 1 | #warning "Do not use ABC, which is deprecated. Use XYZ instead." |

#pragma : 用來告知編譯器某些特殊指示，例如不要輸出錯誤訊息，抑制警告訊息，或者加上記憶體漏洞檢查機制等。

**想要預處理器處理成字符串, 怎麼做?**  
1. 固定參數, 可以使用#號 ：



|  |  |
| --- | --- |
| 1 | #define MONCK(ARGTERM) printf("The term " #ARGTERM " is a string/n") |

若使用 MONCK(A to B);  
則输出：The term A to B is a string

2. 可變參數, 用三個點（...）來表示，且配合用\_\_VA\_ARGS\_\_來展開：



|  |  |
| --- | --- |
| 1 | #define err(...) fprintf(stderr,\_\_VA\_ARGS\_\_) |

若使用 err("%s %d/n","The error code: ",48); => 就是 fprintf(stderr,"%s %d/n","The error code ",48); 一樣意思  
或是另一種與法如:



|  |  |
| --- | --- |
| 1 | #define ABC(format, arg…)   printf(format, ##arg); |

**20. 找出程式錯誤:**

(1) 一個中斷服務次程序(ISR)的程式碼  
中斷是嵌入式系統中重要的組成部分，這導致了很多編譯開發商提供一種擴展 - 讓標準C支持中斷。具代表的事實是，產生了一個新的關鍵字 \_\_interrupt。  
下面的程式碼就使用了\_\_interrupt關鍵字去定義了一個中斷服務次程序(ISR)，請評論一下這段程式碼的。



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \_\_interrupt double compute\_area(double radius)  {  double area = PI \* radius \* radius;  printf("\nArea = %f"， area);  return area;  } |

這個函數有太多的錯誤了︰  
∙ ISR 不能返回一個值。如果你不懂這個，那麼你不會被雇用的。  
∙ ISR 不能傳遞參數。如果你沒有看到這一點，你被雇用的機會等同第一項。  
∙ 在許多的處理器/編譯器中，浮點一般都是不可重入的。有些處理器/編譯器需要讓多餘的暫存器入棧(PUSH入堆疊)，有些處理器/編譯器就是不允許在ISR中做浮點運算。此外，ISR應該是短而有效率的，在ISR中做浮點運算是不明智的。  
∙ 與第三點一脈相承，printf()經常有重入和性能上的問題。如果你丟掉了第三和第四點，我不會太為難你的。但如果你能得到後兩點，那麼你的被雇用前景越來越光明了。

(2) 程式碼片斷



|  |  |
| --- | --- |
| 1  2  3 | unsigned int zero = 0;  unsigned int compzero = 0xFFFF;  /\*1's complement of zero \*/ |

對于一個int型不是16位的處理器為說，上面的程式碼是不正確的。  
應編寫如下︰



|  |  |
| --- | --- |
| 1 | unsigned int compzero = ~0; |

**21. 如何在C中初始化一個字元陣列。**

最簡單的方法是  
char array[];  
但是在初始化上好像還欠缺點什麼，以下：  
char array[5]={'1','2','3','4','5'};  
或者  
char array[5]={"12345"};  
或者  
char array[2][10]={"China","Beijing"};  
也許更符合“初始化”的意思。

**22. 如何在 C 中為一個陣列分配空間。**

一種是棧的形式：  
char array[5]； 意思是分配給陣列array一個5個位元組的空間。

一種是堆的形式：  
char \*array;  
array=(char \*)malloc(5);  
//C++: array=new char[5];

**23. 如何初始化一個指標陣列。**

指向陣列的指標：  
char (\*array)[5];含義是一個指向存放5個字元的陣列的指標。  
存放指標的陣列：  
char \*array[5];含義是一個陣列中存放了5個指向字元型資料的指標。  
按照題意，我理解為初始化一個存放指標的陣列，char \*array[2]={"China","Beijing"}；  
其含義是初始化了一個有兩個指向字元型資料的指標的陣列，這兩個指標分別指向字串"China"和"Beijing"。

**24. 如何定義一個有10個元素的整數型指標陣列。**

int \*array[10];

**25. s[10]的另外一種表達方式是什麼。**

前面說過了，陣列和指標其實是資料存在形態的兩種表現形式，如果說對於陣列s[]，我們知道\*s=s[0]，那麼s[10]的另一種表達方式就是：  
\*(s+10)

**26. GCC3.2.2版本中支援哪幾種編程語言**

支援 C, C++, Java, Obj-C, Ada, Fortran, Pascal, Modula-3等語言

**27. 要使用CHAR\_BIT需要包含哪個頭檔**

limits.h

**28. 對(-1.2345)取整是多少？**

其實不同的取整函數可能有不同的結果，不過這個數沒有太大的爭議，答案是-1。

**29. 如何讓局部變數具有全局生命期。**

即用static修飾就可以了，但是只是生命期延長，範圍並沒有擴大，除非把這個變數定義在函數體外的靜態區，不過那樣就變成Global變數了

**30. C 中的常量字串應在何時定義？**

據我理解，有兩種情況，一種是預處理階段，用#define定義；還有就是使用const修飾詞，不過const修飾的是一個變數，其含義是“唯讀”，稱之為常量並不準確，但是確實可以用操作變數的方法當常量用。所以還是第一種比較靠譜。

**31. 如何在兩個.c檔中引用對方的變數**

最簡單最直接的方法是為變數添加extern修飾詞，當然這個變數必須是Global變數  
(還有一種就是利用函數調用來進行變量的間接引用，比如這個C檔中的一個函數引用另外一個C中的函數，將變數通過實參的形式傳遞過去。  
不過題目既然說是引用，那麼還是用第一個答案好了。)

**32. 使用malloc之前需要做什麼準備工作**

首先要知道malloc的用途，簡單的說就是動態的分配一段空間，返回這段空間的頭指針。  
實際的準備工作可以這麼分：需要這段空間的指標是否存在，若不存在，則定義一個指標用來被賦值，還要清楚要返回一個什麼類型的指標，分配的空間是否合理；如果指標已經存在，那麼在重新將新的空間頭位址賦值給這個指標之前，要先判斷指標是否為NULL，如果不是要free一下，否則原來的空間就會被浪費，或者出錯，free之後就按照前一種情形考慮就可以了。

**33. realloc函數在使用上要注意什麼問題。**

據我的初步理解，這個函數的作用是重新分配空間大小，返回的頭指標不變，只是改變空間大小。  
既然是改變，就有變大、變小和為什麼改變的問題。變大，要注意不能大到記憶體溢出；變小，那變小的那部分空間會被徵用，原有資料不再存在；為什麼改變，如果是想重新挪作他用，還是先free 了吧。

**34. strtok 函數在使用上要注意什麼問題。**

這個函數的作用是分割字串，但是要分割的字串不能是常量，這是要注意的。strtok的原形是char \*strtok(char \*string, char \*delim);  
比如先定義一個字串：char array[]="part1,part2";  
我們將","作為分隔符號，先用pt=strtok(array,",");，得到的結果print出來就是"part1"，

那後面的呢，要寫成pt=strtok(NULL,",");  
注意，要用NULL，如果被分割的字串會被分成N段，那從第二次開始就一直要用NULL。  
總結起 來，需要注意的是：被分割的字串和分隔符號都要使用變數；除第一次使用指向字串的指標外，之後的都要使用NULL；

**35. gets函數在使用上要注意什麼問題。**

這是一個鍵盤輸入函數，將輸入字串的頭位址返回。  
輸入完一個字串後，這個字串可能依然存在於這個標準輸入流之中，當再次使用gets的時候，也許會把上次輸入的東西讀出來，所以應該在使用之後用 fflush(stdin);處理一下，將輸入流清空。  
最後也還是要注意溢出的問題。

**36. a+++++b 所表示的是什麼意思？有什麼問題？**

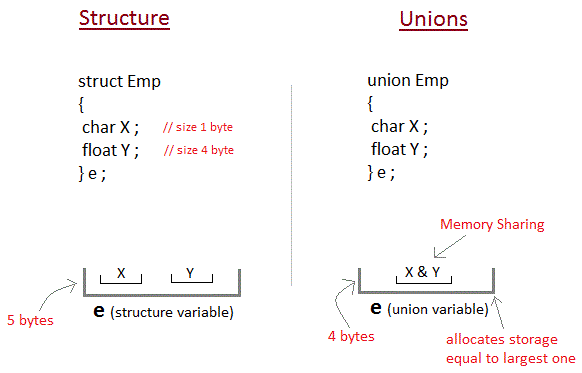
這個其實並沒有語法錯誤，按照C對運算符等級的劃分，++的優先順序大於+，那麼這句話會被編譯器看做：(a++)+ (++b)，這回明白了吧。有什麼問題，語法上沒有問題，有的是道德上的問題！作為一個優秀的程式師，我們要力求語句的合法性和可讀性，如果寫這句的人是 在一個team裏，那麼他基本會被打的半死……最後討論一下結果：假設a之前的值是3，b是4，那麼運行完這個變態語句後，a的值是4，b是5，語句的結 果是8。

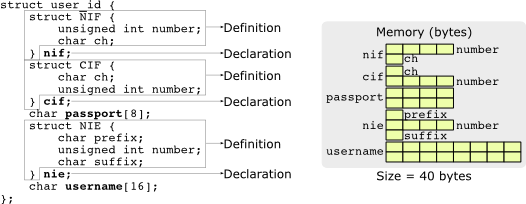
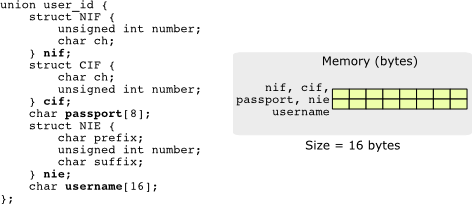
**37. 如何定義Bool變數的TRUE和FALSE的值**

把TURE和FALSE給定義了，使用#define就可以：  
#define TURE 1  
#define FALSE 0  
如果有一個變數需要定義成bool型的，舉個例子：bool a=TURE;就可以了。

**38. explain "struct" and "union"**

struct → 所佔記憶體空間為 member 相加  
union → 所佔記憶體空間由最大size member決定

note: 所以union的member同一時間只會最多出現一個  
[](http://4.bp.blogspot.com/-6bD_hsEUoXY/VNCJ_wzTQdI/AAAAAAAAF3c/Ru9dazSJmTQ/s1600/su.png)

**更詳細的圖表比較:**  
**struct:**  
假設integers 佔用4個bytes並且characters為1個bytes，則structure需要將40個bytes存儲在memory 中，如圖所示。 但是在所有的領域中，只有一個包含information，其餘都是空的。 這意味著data structure只會佔用佔用空間的12.5％和40％之間。 超過一半的內存將被浪費。  
[](https://4.bp.blogspot.com/-jNk2CAkzT5Y/WsNGSoEcYdI/AAAAAAAAJtw/g1Z8CEvaLa84GWMRnDLg7FkeKDVurA4fgCLcBGAs/s1600/CDataStructuresUserStruct_en.png)  
**union:**  
定義“union”時，只保留最大字段的空間。 data存儲在相同的memory位置和所選字段的結構中。 這種結構不會儲存哪個區域正在使用的地方。 如果需要此information，程序員必須將其存儲在additional data structure中。 對“union”中數據的訪問完全按照結構執行。  
[](https://2.bp.blogspot.com/-6x3p9D-vWvI/WsNGSrqM2NI/AAAAAAAAJt0/zI3OgV6WPAA_DFu7AxD86dNXUev4KiNbgCLcBGAs/s1600/CDataStructuresUnion_en.png)

**39. explain lvalue and rvalue**

我們要先知道lvalue和rvalue，指的是一個表達式(expression)而非物件(object)。兩者的差別在於，lvalue指明了一個續存物件(persistent object)或function，例如++a,obj, \*ptr, arr[index]等都是lvalue；而rvalue是沒有名字的，可能是暫時物件(temporary object)，函數傳回的non-reference value，或者就只是一個字面常數值，例如a++,42,int(3.14)。重要的差別在於，lvalue所指的東西是有名字的、能續存下去的物件或函數，而rvalue所指的東西是沒有名字的、暫時性的、在表達式之後就會消失的東西。

一個判斷表達式是lvalue或rvalue的方法是：能不能對該表達式取址(address of, &運算子)?如果可以，則是lvaule，反之則是rvalue。因為，lvalue指的東西是有名字的，在表達式結束後不會消失，取其位址是安全的；而rvalue指的東西在表達式結束後就會消失，如果我們取它的位址，在表達式結束後，rvalue所指的東西就會蒸發，該位址上會有什麼值就不得而知了，這是非常危險的動作，所以編譯器是不給過的。

**40. write a code that check the input is a multiple of 3 or not without using division or mod**



|  |  |
| --- | --- |
| 1  2  3  4  5 | while n > 0:      n -= 3  while n < 0:      n += 3  return n == 0 |

**41. 不使用暫存變數交換兩個變數 ( Swap two variables without using a temporary variable )**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 第一種方式是使用 XOR 運算：  a = a ^ b  b = a ^ b  a = a ^ b    第二種是使用加法與減法：  a = a + b  b = a - b  a = a - b    第三種是使用乘法與除法：  a = a \* b  b = a / b  a = a / b    這三種方式作用都相同，但是第二種與第三種都可能會有溢位（overflow）的問題，所以最佳的解法是使用第一種 XOR 運算。 |

**42. 寫出一個簡單 makefile 編譯 main.c 與 foo.c, 也可清除main.o foo.o 檔**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | main: main.o foo.o      gcc main.o foo.o -o main  main.o: main.c      gcc main.c -c  foo.o: foo.c      gcc foo.c -c  clean:      rm -rf main.o foo.o |

**43. 用 C 實作 Stack的 pop()與 push() :**

在程式語言中，要實現堆疊有許多種方法，最簡單的方式就是使用陣列模擬。 ([參考](http://program-lover.blogspot.tw/2008/05/stack.html))  
即定義一個大小為 MAX\_SIZE 的陣列，並使用一個變數 top 來表示堆疊的頂端。  
以下是 C 以陣列實現的堆疊原始碼，可以參考看看：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55 | #include <stdio.h>  #include <stdlib.h>    #define MAX\_SIZE 10    void push(int);  void pop(void);    int stack[MAX\_SIZE], top = 0;    int main(void)  {      push(0);      push(1);      push(2);      pop();      pop();      push(3);      push(4);      pop();      pop();      pop();      push(5);      push(6);      push(7);      pop();      pop();      pop();        system("pause");  }    void push(int data)  {      if (top == MAX\_SIZE)      {          printf("The stack is full!\n");      }      else      {          stack[top++] = data;      }  }    void pop(void)  {      if (top == 0)      {          printf("The stack is empty!\n");      }      else      {          printf("%d\n", stack[--top]);      }  } |

**44. 什麼是 extern C 與 extern ?**

extern "C" 是C++特有的組合關鍵字，在C裡並沒有這個的組合，僅有extern這個關鍵字!  
**extern "C" :**

為什麼C++會需要這樣的關鍵字組呢? 原因是C++它有一個複載(overloading)的功能，也就是說同樣的函式名稱可以有多個定義只要參數簽名不同即可。比如說C++裡可以有以下的二個宣告  
bar(int i, int j);  
bar(double i, double j);  
這二個函式都是同樣的名字叫bar，僅參數型式不同。然而在C語言裡是不被允許的! C++是如何處理這同名的函式呢? 其實他在編譯時會偷偷的把這二個函式名變成不同的名字，舉例來說bar(int i, int j)可能會被改成\_bar\_int\_int(每種compiler產生不太一樣)，而另一個則被改成\_bar\_double\_double。這技術稱Mangling。

問題來了! 當我們希望C++不要偷換函式名時該怎麼辦? 於是就有了extern "C" 這個關鍵字組出現了。這個字組就是請C++不要自己又偷天換日，請它保留原名。所以當我們宣告一個函式如下時:  
extern "C" bar(int i, int j);

編譯器就不會把bar變成\_bar\_double\_double。

實際使用的注意事項:  
1. 當C++使用C的函式庫(library)時，C++不能直接套用C的header檔。因為他會把header裡的宣告給mangleing了。所以他必須使用如下:

extern "C"  
{  
#include "C\_LIB.h" //C\_LIB 是C語言所製告出來的。  
}

2. 相反的，在C語言的編譯器裡若要使用由C++所製告出來的C函式庫，那麼也不能直接的使用C++的header檔。因為此header檔必然存在extern "C" 這個關鍵字組，而這字組C語言是不認識的。所以必需要把C++的header檔裡的extern "C" { } 移除後才可以讓C編譯器使用。

**extern :**

一般是用在外部變數(或稱為全域變數)上的  
\* 宣告在函數內部的變數為內部變數，又稱為自動變數

主要功能是可以讓不同檔案間可以共用同一個變數  
\* 變數宣告可以多次，宣告其存在  
\* 變數定義只可以一次，讓程式為其配置空間

file1.cpp  
---------  
int a; /\* 變數a的定義兼宣告 \*/

file2.cpp  
---------  
extern int a;  
/\* 變數a的外部宣告 \*/  
/\* 表示a在別的檔案被定義了 \*/

**45. 什麼是 inline ?**

在函式前面加入關鍵字 inline , 此函式就可以「建議」編譯器將之設定為 「行內函式」(Inline function)  
有什麼優點呢?  
如果編譯器評估效能有改善而建議被採納，則該函式會自動在呼叫點展開為程式碼，行內函式建議可以直接定義於表頭檔案中



|  |  |
| --- | --- |
| 1  2  3 | inline int pow2(int num) {     return num\*num;  } |

**46. 請說明 call by value, call by address, call by reference**

在C語言裡裡，傳遞參數的2種方式，分別是Call by value、Call by pointer。而在C++裡多了一個Call by reference的方法。Call by reference的作用和目的和Call by pointer是一樣的，都是想要指回原本的變數並且可以修改。不過Call by reference寫起來更簡單。 (或看[另一文章介紹](http://eeepage.info/call-by/))

//call by value  
傳值call by value，傳值的意思顧名思義就是只把『值』複製給對方  
對方拿到了你的值以後做任何改變都不會影響到原本的值



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(){     int a = 10;     plus(a);  }  int plus(int a){     //兩個a位於不同記憶體空間     return a++;  } |

//call by address  
傳指標就是傳address過去，說到底仍然也是call by value，只是那個value是指標本身，複製的內容也是指標本身，只不過那個值Value剛好就是位址address  
也稱作call by value of pointer, 所以對方修改address裡的數值也從時修改了傳送方原本的值 ([參考](http://wp.mlab.tw/?p=176))



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(){     int a = 10;     plus(&a); // a = 11  }  void plus(int\* a){     //傳入a的記憶體位置，function中的a pointer指向main中的a變數     (\*a)++;  } |

//call by reference  
傳參考是C++才有的東西，C語言是沒有的唷，可以說call by reference是call by address的進化版，  
因為傳址的指標它的內容為指向的位址，但他本身仍然有記憶體位址，但是傳參考是不會有的



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int main(){     int a = 10;     plus(a); // a = 11;  }  void plus(int &a){     //作用與call by address相同，寫法更簡潔 但僅限C++ only     a++;  } |

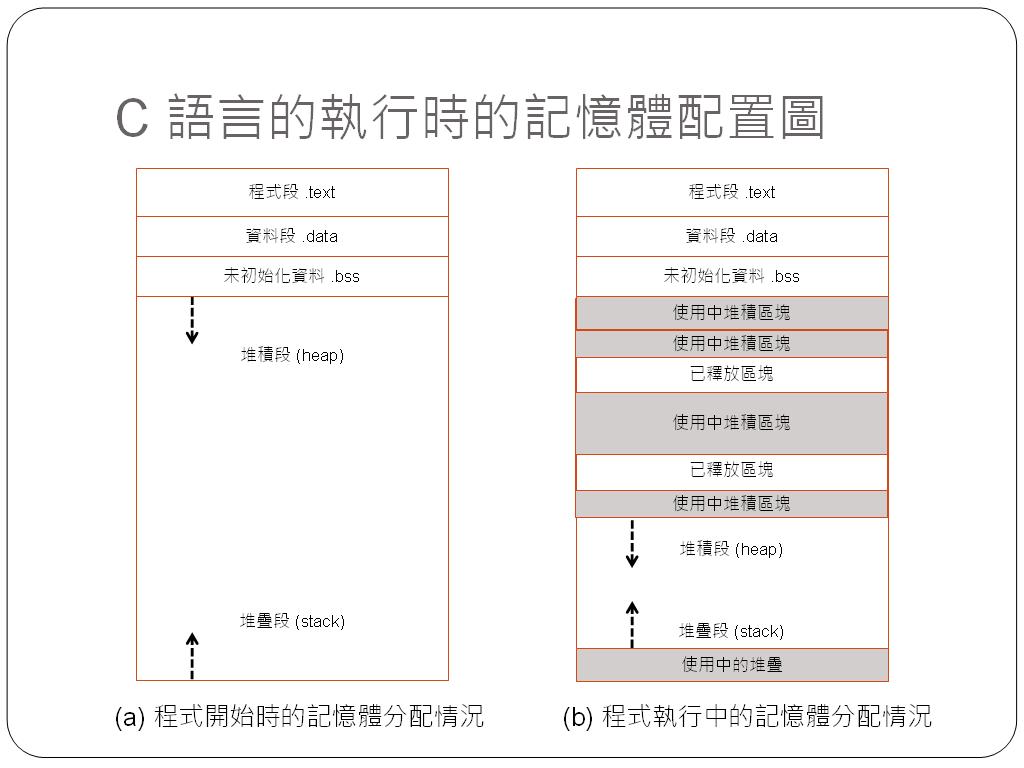
**47. 學習心得: 函式指標 (pointer to function)**

你會學習到：  
1. 函式在記憶體中的情況  
2. 如何宣告一個函式指標  
3. 函式的宣告  
4. 使用 typedef 來定義一個函式指標的類型

------------------------ 1. 函式在記憶體中的情況 ------------------------

在真正的開始學習函式指標 ( pointer to function ) 之前，我們要先弄懂到底什麼是函式以及函式在記憶體分布的情形。

當一個程式碼檔案 ( .c , .cpp ...) 被執行的時候，會產生出一個程序 ( process )，此時才會開始佔用記憶體及進行各種運算。先前我們定義的指標都是針對一個資料型態，如：int \* 這是一個指向 int 資料類型的指標。然而，Ｃ為什麼沒有為 function 也定義出一個資料型別？因為不同的 function 依照建立的方式不同而有不同的 type ，所以沒辦法替每一種 function 都定義出一個通用的型別。

實際上，function 和一般的資料儲存的狀況不太一樣，他儲存的是一堆的指令。  
我們可以參考下面的圖片：  
[](https://1.bp.blogspot.com/-8kOXtwC1x7A/WV9qXO4N25I/AAAAAAAAIE4/y2bCKOZc_noJ3SNi5mTqIBydhRn8dyJiwCLcBGAs/s1600/CEnvironment.jpg)

當一個 process 產生的時候，會有一塊自己可以使用的記憶體，並且這個記憶體分成許多不同的區塊，儲存不同的東西。一般來說，我們使用的區域變數是儲存在 stack 區段。程式碼則是儲存在 .text 區。因此，雖然和一般的資料型式不同，但是程式碼也是會佔用記憶體的！理所當然，他也會有自己的位址。

我們可以利用反組譯的方式來查看：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include<stdio.h>    void say\_hello();    void say\_hello(){  printf("Hello\n");  return;  }    int main(){  say\_hello();  return 0;  } |

編譯方式：gcc -g -o test test.c  
在這裡我使用 gdb 來輔助



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | (gdb) print say\_hello  $1 = {void ()} 0x40052d <say\_hello>  (gdb) disassemble main  Dump of assembler code for function main:     0x000000000040053e <+0>: push   %rbp     0x000000000040053f <+1>: mov    %rsp,%rbp     0x0000000000400542 <+4>: mov    $0x0,%eax     0x0000000000400547 <+9>: callq  0x40052d <say\_hello>     0x000000000040054c <+14>: mov    $0x0,%eax     0x0000000000400551 <+19>: pop    %rbp     0x0000000000400552 <+20>: retq  End of assembler dump.  (gdb) disassemble say\_hello  Dump of assembler code for function say\_hello:     0x000000000040052d <+0>: push   %rbp     0x000000000040052e <+1>: mov    %rsp,%rbp     0x0000000000400531 <+4>: mov    $0x4005e4,%edi     0x0000000000400536 <+9>: callq  0x400410 <puts@plt>     0x000000000040053b <+14>: nop     0x000000000040053c <+15>: pop    %rbp     0x000000000040053d <+16>: retq  End of assembler dump. |

第 1 行 顯示 function name 的位址，接著我們到第15行可以發現，function 第 1 行指令的位址和function name的位址是一樣的，所以我們可以得知：其實function name的位址就是function第一行指令的位址。接著，在第 8 行中，我們可以看到main呼叫函式也是利用函式第 1 行指令的位址。

由此可知，其實在function在記憶體中也是有特定儲存的位址。  
如果我們在linux的環境中使用 objdump 這個指令可以看得更清楚，他會指名在.text區  
Disassembly of section .text:  
.....  
.....  
000000000040052d :

------------------------ 2. 如何宣告一個函式指標 ------------------------  
宣告方式： return\_type (\*func\_pointer)( parameter list );  
如下



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include<stdio.h>    int add(int n1, int n2){  return n1 + n2;  }    int main(){  int (\*fptr)(int, int);  fptr = add;    int num1 = 10, num2 = 30;  printf("num1 + num2 is : %d\n", fptr(num1, num2));    return 0;  } |

第8行宣告一個指標名為 fptr ，指向 int (int, int)  
換句話說，fptr的type : int (\*)(int, int)

我們可以用以下兩種方式將已知的function, assign 給 function pointer：

1. fptr = &func\_name;  
2. fptr = func\_name;

使用的時候也有兩種方法可以使用：

1. (\*fptr)(num1, num2);  
2. fptr(num1, num2);

每次讀到這一段的時候，總是有個疑惑：為什麼兩種方法都可以？  
甚至，你編譯下面這一段 code 也可以執行：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #include<stdio.h>    void print\_hello(){  printf("Hello world\n");  return;  }    int main(){  print\_hello();  (&\*print\_hello)();  (\*&\*print\_hello)();  (\*\*&print\_hello)();  (&\*\*&print\_hello)();  (&\*&\*print\_hello)();  return 0;  } |

Why?

要確保初值化(initialization)或是assignment的正確性，取決於  
1.數值 2.型別  
舉例來說：  
int func(int, int);  
int (\*fptr)(int, int);

type of fptr is : int (\*)(int, int);  
type of func is : int (int, int);  
type if &func is : int (\*)(int , int);

所以 fptr = &func 很合理，型別正確並且數值也正確（&func數值和func一樣）  
可是 fptr = func 型別不一樣。 所以在這裡其實做了implicit conversion 將function name ( i.e. function designator ) 轉成函式指標使用

至於，使用的方式有兩種一種用deference(i.e use \* operator)，一種不用  
我曾經在一本書上看到這樣的解釋：

在使用function的時候，fun()，其中()稱作 function-call operator  
function-call operator 只允許 pointer to function使用。

所以一般我們在使用func\_name(); 其實會做implicit conversion將func\_name轉型, 所以我們這樣寫其實也可以執行(&func\_name)();

以上就是 function pointer 的基本操作。

------------------------ 3. 函式的宣告 ------------------------

不知道大家有沒有想過一個問題：

我們在宣告變數的時候，都是依循這樣的形式 : type var\_name;  
為什麼宣告function是 : return\_type func\_name( parameter list );

其實我們的看法應該是這樣 func\_name( parameter list) 這一整個和var\_name對照  
所以，我們在gdb中檢查 func\_name( parameter list) 這整個東西的type會和return type是一樣的。

在gdb中要查看變數的type可以使用：ptype var\_name / whatis var\_name

利用先前的例子：



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | (gdb) ptype say\_hello  type = void ()  (gdb) ptype say\_hello()  type = void  (gdb) ptype &say\_hello  type = void (\*)() |

------------------- 4. 使用 typedef 來定義一個函式指標的類型 ---------------

每次我們要宣告一個 function pointer 假如都要照之前那樣寫，對大多數人來說其實不太容易看。更甚者，牽扯到一堆轉型的時候更讓人頭暈目眩。

因此我們利用typedef來定義一個function pointer的type



|  |  |
| --- | --- |
| 1  2  3  4  5 | int (\*ptr)(int, int);  // declare a pointer to function : ptr                         // and its type is : int (\*)(int, int)    typedef int (\*func\_t)(int, int);  這個時候，func\_t 就是一個 int(\*)(int, int)的型別了 |

這個語法可能讓人感到confuse，因為以前我們定義的方式很單純：  
typedef int bool;  
在這裡我會這樣看 (\*func\_t)(int, int)是一個東西，然後藉由  
typedef int (\*func\_t)(int, int);來間接定義func\_t

**48. 程式設計C++: Struct與Class有什麼不同?**

C++中的struct對C中的struct進行了擴充，它已經不再只是一個包含不同數據類型的數據結構了，它已經獲取了太多的功能。  
struct能包含成員函數嗎？ 能！  
struct能繼承嗎？ 能！！  
struct能實現多態嗎？ 能！！！

整理如下:

|  |  |  |
| --- | --- | --- |
|  | **Struct** | **Class** |
| 預設修飾詞 | public | private |
| 其他修飾詞 | 無 | protected、public |
| 參照方式 | value type | reference type |
| 繼承 | 不支援 | 支援 |
| 記憶體形式 | stack | heap |
| 初始化 | 無 | 有 |
| 自訂函式 | 無 | 有 |

既然這些它都能實現，那它和class還能有什麼區別？  
最本質的一個區別就是默認的訪問控制：  
默認的繼承訪問權限  
struct是public的，class是private的。  
你可以寫如下的代碼：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | struct A  {      char a;  }；    struct B : A  {      char b;  }； |

這個時候B是public繼承A的。  
如果都將上面的struct改成class，那麼B是private繼承A的。這就是默認的繼承訪問權限。  
所以我們在平時寫class繼承的時候，通常會這樣寫：



|  |  |
| --- | --- |
| 1 | class B : public A |

當然，到底默認是public繼承還是private繼承，取決於子類而不是基類。  
我的意思是，struct可以繼承class，同樣class也可以繼承struct，那麼默認的繼承訪問權限是看子類到底是用的struct還是class。如下：



|  |  |
| --- | --- |
| 1  2 | struct A{}；class B : A{}; //private繼承  struct C : B{}； //public繼承 |

struct作為數據結構的實現體，它默認的數據訪問控制是public的，而class作為對象的實現體，它默認的成員變量訪問控制是private的! 我依舊強調struct是一種數據結構的實現體，雖然它是可以像class一樣的用。我依舊將struct里的變量叫數據，class內的變量叫成員，雖然它們並無區別。

到底是用struct還是class，完全看個人的喜好，你可以將程序里所有的class全部替換成struct，它依舊可以很正常的運行。但我給出的最好建議，還是：  
- 當你覺得你要做的更像是一種數據結構的話，那麼用struct  
- 如果你要做的更像是一種對象的話，那麼用class。  
當然，我在這裡還要強調一點的就是，對於訪問控制，應該在程序里明確的指出，而不是依靠默認，這是一個良好的習慣，也讓你的代碼更具可讀性。

說到這裡，很多了解的人或許都認為這個話題可以結束了，因為他們知道struct和class的「唯一」區別就是訪問控制。很多文獻上也確實只提到這一個區別。但我上面卻沒有用「唯一」，而是說的「最本質」，那是因為，它們確實還有另一個區別，雖然那個區別我們平時可能很少涉及。

那就是：「class」這個關鍵字還用於定義模板參數，就像「typename」。但關鍵字「struct」不用於定義模板參數。這一點在Stanley B.Lippman寫的Inside the C++ Object Model有過說明。

問題討論到這裡，基本上應該可以結束了。但有人曾說過，他還發現過其他的「區別」，那麼，讓我們來看看，這到底是不是又一個區別。還是上面所說的，C++中的struct是對C中的struct的擴充，既然是擴充，那麼它就要兼容過去C中struct應有的所有特性。例如你可以這樣寫：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | struct A //定義一個struct  {    char c1;    int n2;    double db3;  };    A a={'p', 7, 3.1415926}; //定義時直接賦值 |

**也就是說struct可以在定義的時候用{}賦初值。那麼問題來了，class行不行呢？**  
將上面的struct改成class，試試看。報錯！  
噢~於是那人跳出來說，他又找到了一個區別。我們仔細看看，這真的又是一個區別嗎？

**你試著向上面的struct中加入一個構造函數（或虛函數），你會發現什麼？**  
對，struct也不能用{}賦初值了  
的確，以{}的方式來賦初值，只是用一個初始化列表來對數據進行按順序的初始化，如上面如果寫成A a={'p',7};則c1,n2被初始化，而db3沒有。這樣簡單的copy操作，只能發生在簡單的數據結構上，而不應該放在對象上。加入一個構造函數或是一個虛函數會使struct更體現出一種對象的特性，而使此{}操作不再有效。

事實上，是因為加入這樣的函數，使得類的內部結構發生了變化。而加入一個普通的成員函數呢？你會發現{}依舊可用。其實你可以將普通的函數理解成對數據結構的一種算法，這並不打破它數據結構的特性。

那麼，看到這裡，我們發現即使是struct想用{}來賦初值，它也必須滿足很多的約束條件，這些條件實際上就是讓struct更體現出一種數據機構而不是類的特性。

**那為什麼我們在上面僅僅將struct改成class，{}就不能用了呢？**  
其實問題恰巧是我們之前所講的——訪問控制！  
你看看，我們忘記了什麼？對，將struct改成class的時候，訪問控制由public變為private了，那當然就不能用{}來賦初值了。加上一個public，你會發現，class也是能用{}的，和struct毫無區別！！！

做個總結，從上面的區別，  
我們可以看出，**struct更適合看成是一個數據結構的實現體，class更適合看成是一個對象的實現體。**

本篇文章尚有頁面: [1](http://eeepage.info/interview-c/) [2](http://eeepage.info/interview-c/2/) [3](http://eeepage.info/interview-c/3/) [4](http://eeepage.info/interview-c/4/) [5](http://eeepage.info/interview-c/5/) 6 [7](http://eeepage.info/interview-c/7/) [8](http://eeepage.info/interview-c/8/) [9](http://eeepage.info/interview-c/9/) [10](http://eeepage.info/interview-c/10/)