切換導航 Farseerfc的小窩

- 繁體
- <u>简体</u>
- **English** 日本語

- **≜** About
- **≜** Links
- About
- <u>Links</u>
- \Box
- □ Import
- □ Life
- <u>□ Tech</u>

- □ Import
- 🗅 <u>Life</u>
- <u>Tech</u>
- Q

搜索

- 搜索
- # <u>歸檔</u>
- *y*

C++ Tricks 2.5 I386平臺的邊界對齊(Align)

2.5 I386平臺的邊界對齊(Align)

首先提問,既然I386上sizeof(int)==4、 sizeof(char)==1,那麼如下結構(struct)A的sizeof是 多少?

struct A{int i;char c;};

答案是sizeof(A)==8……1+5=8?

呵呵,這就是I386上的邊界對齊問題。我們知道,I386上有整整4GB的地址空間,不過並不是每一個字節上都可以放置任何東西的。由於內存總線帶寬等等的技術原因,很多體系結構都要求內存中的變量被放置於某一個邊界的地址上。如果違反這個要求,重則導致停機出錯,輕則減慢運行速度。對於I386平臺而言,類型爲T的變量必須放置在sizeof(T)的整數倍的地址上,char可以隨便放置,short必須放在2的整數倍的地址上,int必須放在4的整數倍的地址上,double必須放在8的整數倍的地址上。如

果違反邊界對齊要求,從內存中讀取數據必須進行 兩次,然後將獨到的兩半數據拼接起來,這會嚴重 影響效率。

由於邊界對齊問題的要求,在計算struct的sizeof的時候,編譯器必須算入額外的字節填充,以保證每一個變量都能自然對齊。比如如下聲明的struct:

```
struct WASTE
```

{

```
char c1;
int i;
char c2;
}
```

實際上相當於聲明瞭這樣一個結構:

```
struct WASTE
```

```
char c1;
```

```
char_filling1 [3];//三個字節填充,保證下一個int的
對齊
int i;
char c2;
char _filling2 [3];//又三個字節填充
}
值得注意的是尾部的3個字節填充,這是爲了可以在
一個數組中聲明WASTE變量,並且每一個都自然對
齊。因爲有了這些填充, 所以
sizeof(WASTE)==12。這是一種浪費,因爲只要我
們重新安排變量的聲明,就可以減少sizeof:
struct WASTE
int i;
char c1,c2;
}
```

像這樣的安排,sizeof就減少到8,只有2個字節的額外填充。爲了與彙編代碼相兼容,C語言語法規定,編譯器無權擅自安排結構體內變量的佈局順序,必須從左向右逐一排列。所以,妥當安排成員順序以避免內存空間的浪費,就成了我們程序員的責任之一。一般的,總是將結構體的成員按照其sizeof從大到小排列,double在最前,char在最後,這樣總可以將結構的字節填充降至最小。

C++繼承了C語言關於結構體佈局的規定,所以以上的佈局準則也適用於C++的class的成員變量。C++進一步擴展了佈局規定,同一訪問區段(private、public、protected)中的變量,編譯器無權重新排列,不過編譯器有權排列訪問區段的前後順序。基於這個規則,C++中有的程序員建議給每一個成員變量放在單獨區段,在每一個成員聲明之前都加上private:、public:、protected:標誌,這可以最大限度的利用編譯器的決策優勢。

在棧中按順序分配的變量,其邊界也受到對齊要求 的限制。與在結構中不同的是,棧中的變量還必須 保證其後續變量無論是何種類型都可以自由對齊, 所以在棧中的變量通常都有平臺相關的對齊最小 值。在MSVC編譯器上,這個最小值可以由宏 _INTSIZEOF(T)查詢:

#define _INTSIZEOF(T) ((sizeof(T) + sizeof(int) 1) & ~(sizeof(int) - 1))

_INTSIZEOF(T)會將sizeof(T)進位到sizeof(int)的整 數倍。

由於在棧中分配變量使用_INTSIZEOF而不是 sizeof,在棧上連續分配多個小變量(sizeof小於int 的變量)會造成內存浪費,不如使用結構(struct)或數 組。也就是說:

char c1,c2,c3,c4;//使用16字節

char c[4];//使用4字節

當然,使用數組的方法在訪問數組變量(比如c[1])時有一次額外的指針運算和提領(dereference)操作,這會有執行效率的損失。這又是一種空間(內存佔用)vs時間(執行效率)的折中,需要程序員自己根據情況權衡利弊。

sizeof的大小可能比我們預期的大,也可能比我們預

期的小。對於空類:

class Empty {};

在通常情況下,sizeof(Empty)至少為1。這是因為 C++語法規定,對於任何實體類型的兩個變量,都必 須具有不同的地址。為了符合語法要求,編譯器會 給Empty加入1字節的填充。所以sizeof()的值不可 能出現0的情況。可是對於以下的類聲明:

class A:public Empty{vitual ~A(){}};

sizeof(A)有可能是6,也有可能是5,也有可能是4!必不可少的四個字節是一個指向虛函數表的指針。一個可能有的字節是Empty的大小,這是是因爲編譯器在特定情況下會將Empty視作一個"空基類",從而實施"空基類優化",省掉那毫無作用的一字節填充。另一個字節是A的一字節填充,因爲從語法上講,A沒有成員聲明,理應有1字節填充,而從語義上講,編譯器給A的聲明加入了一個指向虛函數表的指針,從而A就不再是一個"空類",是否實施這個優化,要看編譯器作者對語法措詞的理解。也就是說,sizeof也會出現4+1+1=4的情況。具體要看編譯器有沒有實施"空基類優化"和"含虛函數表的空類優

結構和類的空間中可能有填充的字節,這意味着填充字節中可能有數值,雖然這數值並不影響結構的 邏輯狀態,但是它也可能不知不覺中影響到你。比如說,你手頭正好有一組依賴於底層硬件(比如多處理器)的函數,他們在操縱連續字節時比手動編碼要快很多,而你想充分利用這種硬件優勢:

bool BitCompare(void* begin,void* end,void*
another);

這個函數將區間[begin,end)之間的字節與another開始的字節相比較,如果有一位不同就返回false,否則返回true。

比如你想將這個函數用於你自己的類的operator==中,這樣可以利用硬件加快速度。不過你在動手前要充分考慮,你的class是否真的要比較每一位。如果在類的成員中存在編譯器填充的字節數,那麼應用以上的函數就是不正確的,因爲填充的字節中可以有不同的值。爲了保證你可以用Bitwise Compare,你必須確保填充的字節中的值也是相同的。這不僅要求你在類的構造函數中初始化類的每

一bit而不是每一個成員,也要求你在複製初始化和 複製賦值函數中也同時保證bitwise copy語義,而 不是編譯器默認產生的memberwise語義。當然, 你可能通過與BitCompare一同提供的BitCopy來完 成這個艱鉅的任務。