VC ++ 6.0 编译器编译期存储器分配模型（内存布局）

作者：合肥工业大学 杨铸
网址：www.200yi.com 电邮：fairyfar@msn.com
2008.04.04 清明节
Scripted by FairyFar.

1. 关于本文

我们编写代码，我们编译代码，我们调试代码，满怀期待地运行程序，又眼睁睁地看着程序崩溃……。“编译器背着我们干了什么”我们又知道多少？形形色色的编译错误又意味着什么？有些看似简单的问题，答案却往往令我们大吃一惊（甚至大惊失色，大汗淋漓，不知所措。）。

闲言短叙，本文试图探讨 VC ++ 6.0 编译器编译期存储器布局情况，文中所有案例适用环境：VC ++ 6.0 Enterprise Edition 英文版，所有设置保持默认值；普通 32 位 PC 机。

2. 内存区域划分

本节内容来自网络，作者不详。该部分内容概要地说明了编译期内存划分几大区域，现摘抄如下：

一个由 C/C++编译的程序占用的内存分为以下几个部分：

1)、栈区（Stack）：由编译器(Compiler)自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

2)、堆区（Heap）：一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3)、全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全局变量和初始化的静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

4)、常量区：常量都存于此区，常量字符串就是放在这里的。程序结束后由系统释放。

5)、程序代码区：存放函数体的二进制代码。

3. 测试案例（源码与反汇编对照）

3.1 测试案例源码与反汇编对照

为了能够形象地说明内存布局模型，先来看一段 Win32 Console Application 代码（表 3.1），其中，加粗文字（行最左端为行标号）为 C 源代码，未加粗文字（行最左端为地址）为反汇编后的指令代码。看上去比较零乱，不过一定要耐住性子，后面的文字将基于此。

表 3.1 测试源码与反汇编对照

```
1:    #include <malloc.h>
2:
3:    int no_init_g1;                    //未初始化的全局变量 1
4:    int no_init_g2;                    //未初始化的全局变量 2
5:    int init_g1=111;                   //初始化的全局变量 1
6:    int init_g2=222;                   //初始化的全局变量 2
7:    int no_init_array_g1[10];          //未初始化的全局数组 1
8:    int no_init_array_g2[10];          //未初始化的全局数组 2
9:    int init_array_g1[10]={0};         //初始化的全局数组 1
10:   int init_array_g2[10]={1};         //初始化的全局数组 2
11:   static int no_init_static;         //未初始化的静态量
```

```
12:    static int init_static=3;                //初始化的静态量
13:
14:    void main(void)
15:    {
004106D0   push        ebp
004106D1   mov         ebp,esp
004106D3   sub         esp,0F8h
004106D9   push        ebx
004106DA   push        esi
004106DB   push        edi
004106DC   lea         edi,[ebp-0F8h]
004106E2   mov         ecx,3Eh
004106E7   mov         eax,0CCCCCCCCh
004106EC   rep stos    dword ptr [edi]
16:        int no_init_l1;                      //未初始化的局部变量 1
17:        int no_init_l2;                      //未初始化的局部变量 2
18:        int init_l1=111;                     //初始化的局部变量 1
004106EE   mov         dword ptr [ebp-0Ch],6Fh
19:        int init_l2=222;                     //初始化的局部变量 2
004106F5   mov         dword ptr [ebp-10h],0DEh
20:        int no_init_array_l1[10];            //未初始化的局部数组 1
21:        int no_init_array_l2[10];            //未初始化的局部数组 2
22:        int init_array_l1[10]={0};           //初始化的局部数组 1
004106FC   mov         dword ptr [ebp-88h],0
00410706   mov         ecx,9
0041070B   xor         eax,eax
0041070D   lea         edi,[ebp-84h]
00410713   rep stos    dword ptr [edi]
23:        int init_array_l2[10]={1};           //初始化的局部数组 2
00410715   mov         dword ptr [ebp-0B0h],1
0041071F   mov         ecx,9
00410724   xor         eax,eax
00410726   lea         edi,[ebp-0ACh]
0041072C   rep stos    dword ptr [edi]
24:        char *char1="123456";                //123456 在常量区，char1 在栈上。
0041072E   mov         dword ptr [ebp-0B4h],offset string "123456" (00426ee8)
25:        char *char2=(char *)malloc(20);      //分配得来得 20 字节的区域在堆区
00410738   push        14h
0041073A   call        malloc (004049b0)
0041073F   add         esp,4
00410742   mov         dword ptr [ebp-0B8h],eax
26:    }
00410748   pop         edi
00410749   pop         esi
0041074A   pop         ebx
0041074B   add         esp,0F8h
00410751   cmp         ebp,esp
00410753   call        __chkesp (00410690)
00410758   mov         esp,ebp
0041075A   pop         ebp
0041075B   ret
```

3.2 内存布局图

对于该案例，以下几幅图形象地说明了第 2 节提到的内存 5 大区域。需要注意的是，图中各区域的起始地址不是绝对的，不同的编译环境可能不完全相同，这里给出的只是一个典型示例。需要注意的是，不同区域地址编址方向也不同。

main()例程的栈（Stack）

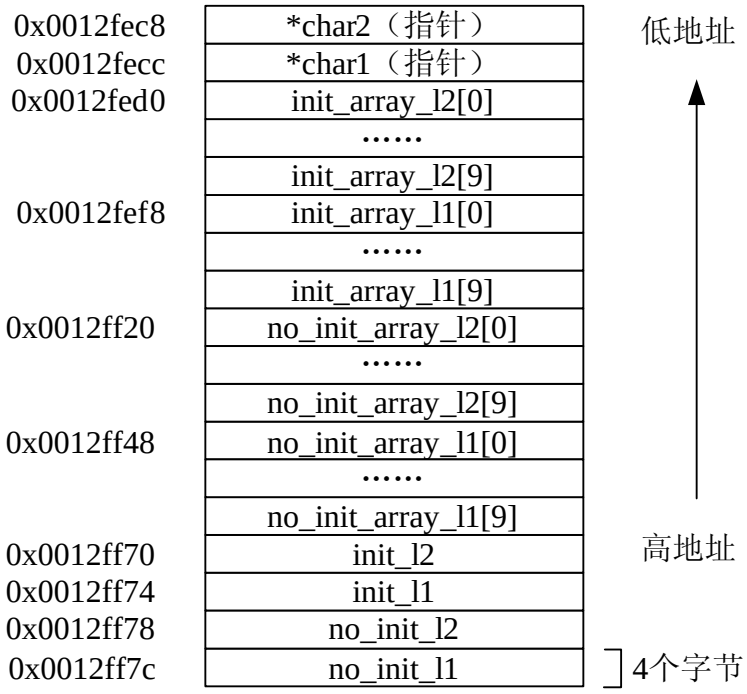

图3.1

全局区（静态区） 未初始化部分

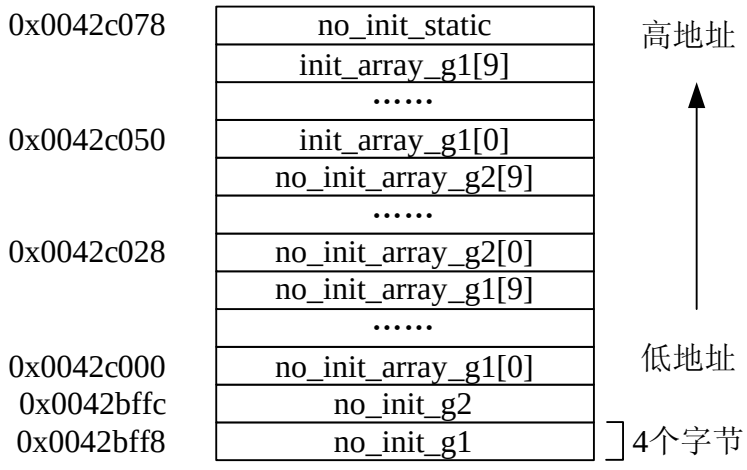

图3.2a

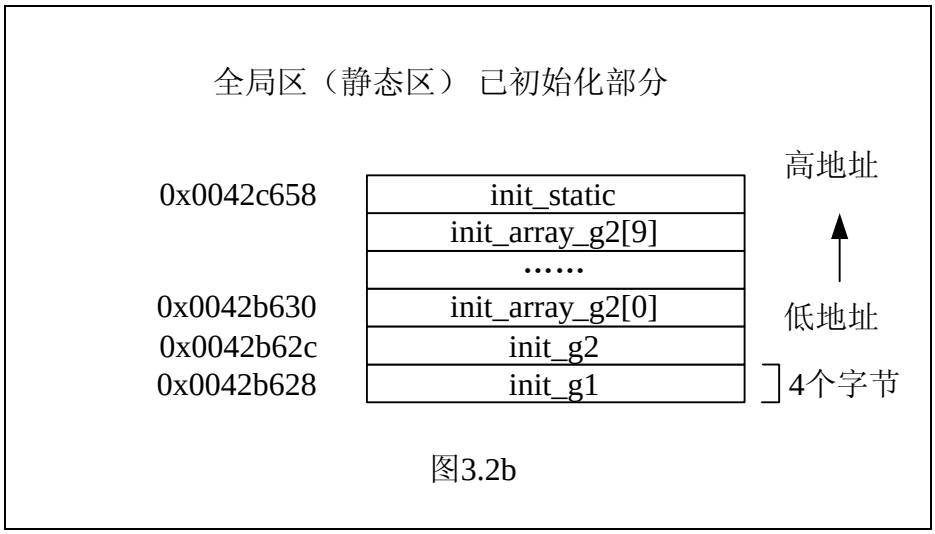

图3.2b

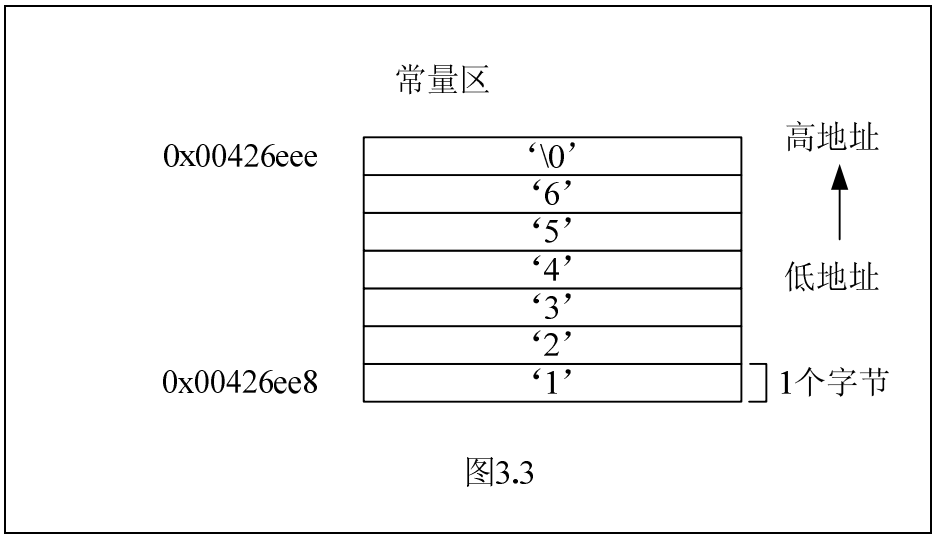

图3.3

图3.4

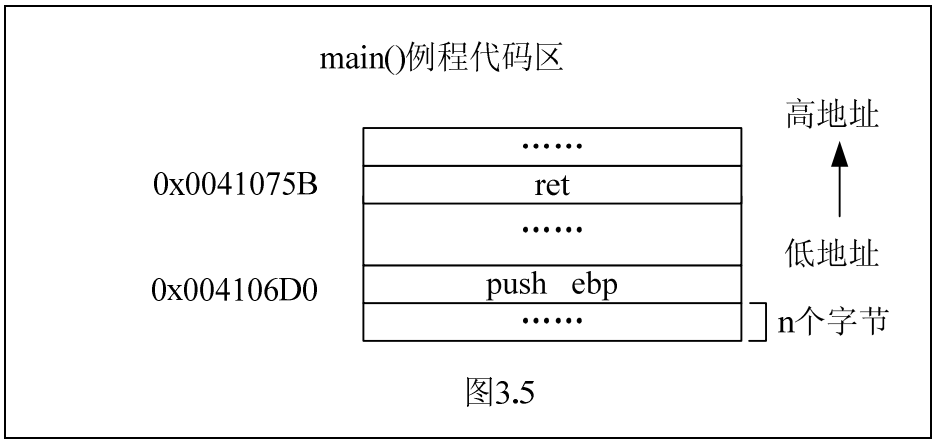

图3.5

4. 应用

通过对第 3 节案例的理解，我们将对一些现象予以解释。

4.1 变量初始化

1）局部变量将被分配在栈中，如果不初始化，则为不可预料值。编译时，编译器将抛出一个编号为 C4700 警告错误（local variable ’变量名’ used without having been initialized）。

表 4.1 代码测试了局部变量未初始化的情况。

表 4.1 未初始化局部变量测试

```
#include <iostream.h>
void main(void)
{
    int no_init_l1;                    //未初始化的局部变量 1
    cout<<no_init_l1<<endl;
}
```

该测试的一个典型输出结果为：-858993460，同时，编译时编译器抛出了一条警告错误。

2）全局变量如果不初始化，则默认为 0，编译时编译器不提示“变量未初始化”。

表 4.2 代码测试了全局变量未初始化的情况。

表 4.2 未初始化全局变量测试

```
#include <iostream.h>
int no_init_g1;                    //未初始化的全局变量 1
void main(void)
{
    cout<<no_init_g1<<endl;
}
```

该测试的输出结果为：0。

3）全局变量初始化为 0 与不初始化效果一样。请留意表 3.1 第 9 行代码，即

```
int init_array_g1[10]={0};         //初始化的全局数组 1
```

我们原来的意图是将 init_array_g1[]第 0 个元素初始化为 0（其它元素也将被初始化为 0），然而从该案例的内存布局来看，init_array_g1[]被安排在 “全局区（静态区）未初始化区域（图 3.2a 所示）”。

因此，对于全局变量初始化为 0 和不初始化是一样的。对于本案例有，

```
int init_array_g1[10]={0};         //初始化的全局数组 1
```

等效于：

```
int init_array_g1[10];             //初始化的全局数组 1
```

当然，出于谨慎，我们还是建议在使用全局变量前对其初始化。

4.2 变量初始化对代码空间的影响

本小节仍然讨论变量初始化问题，但出于重视，我们将其独立成小节。先来看两个测试案例。

案例 1：建立 Win32 Console Application 工程，工程名：Test1，代码如表 4.3。

表 4.3 工程 Test1

```
int init_array_g1[10000000]={0};   //初始化的全局数组 1
void main(void)
{
}
```

编译成Debug 版本，察看 Debug 目录下的 Test1.exe 可执行文件大小，典型大小约 184KB

（约 0.18MB）。

案例 2：建立 Win32 Console Application 工程，工程名：Test2，代码如表 4.3。

表 4.4 工程 Test2

```
int init_array_g1[10000000]={1};          //初始化的全局数组 1
void main(void)
{
}
```

编译成 Debug 版本，察看 Debug 目录下的 Test2.exe 可执行文件大小，典型大小约 46MB。

两个案例唯一区别不过在于是用 0 还是 1 初始化 init_array_g1[]数组第 0 个元素。生成的可执行文件大小却天壤之别。

上面已经说过，对于全局变量初始化为 0 与不初始化效果一样。因此，这里的 Test1 案例并没有对全局变量初始化。

那么全局变量初始化与不初始化对代码空间又有什么影响呢？

我们知道，运行于基于冯·诺依曼体系结构系统上的程序，数据和程序是一起存储的，区别于哈佛体系的数据和程序分立存储结构。因此，编译时，编译器会将全局变量的初始化数据捆绑到最终生成的程序文件中，而对于未初始化的全局变量只是为其分配（指示）了存储位置，不会将大量的 0 捆绑到程序中。

现在再来看以上两个案例。Test1 实质上没有初始化全局变量，编译时编译器只是为 init_array_g1[]指出了将要使用的内存位置，而不发生数据邦定。Test2 则不同，它将 init_array_g1[0]初始化为 1，其它元素全部初始化为 0，因此，编译器将把 init_array_g1[]数组的 10000000 个元素的初始化数据全部捆绑到最终的可执行文件中，导致编译后的文件十分庞大。

4.3 关于堆和栈

由于历史原因，我们习惯把堆和栈合在一起称呼（堆栈），然而，在这里我们要严格区分堆和栈的概念。

例程中声明的局部变量被分配在栈中，而栈的大小是相当有限的（一、两个兆），庞大的数组可能使栈不够用，造成运行期栈溢出（Overflow）错误（注意：不是编译期错误），而堆的大小主要取决于系统可用内存和虚存的多少。下面来看几个例子：

案例 3 代码如表 4.5 所示。

表 4.5 案例 3：栈溢出测试

```
void main(void)
{
    int init_array_l1[1000000]={1};      //初始化的局部数组 1
}
```

编译该代码，没有编译期错误。执行时却发生了运行期错误（提示 Stack Overflow），因为栈空间不够用。

案例 4，把案例 3 代码改一下，数组定义为全局变量，如表 4.6 所示。

表 4.6 案例 4

```
int init_array_g1[1000000]={1};          //初始化的全局数组 1
void main(void)
{
}
```

编译该代码，没有编译期错误，也不发生运行期错误。因为全局变量不是分配在栈中的（注意：也不在堆中），能用多大空间取决于系统可用内存和虚存的多少。

对于案例 3 的问题还有一种方法可以解决：动态申请内存空间。

动态申请的内存空间是在运行期分配的，一旦申请成功，将分配在堆中，因此，大小也

是取决于系统可用内存和虚存的多少。

案例 5， 把案例 3 代码用另一种方法改一下，如表 4.7 所示。

表 4.6 案例 5：动态申请内存空间解决庞大数据存储问题

```
#include <malloc.h>
void main(void)
{
    //分配得来得空间在堆中
    int *init_array_11=(int *)malloc(1000000);
}
```

啰嗦一下：案例 5 中的内存空间在堆中。还有一点不同于案例 4：案例 4 的内存空间是在编译期分配的，而案例 5 的内存空间是在运行期分配的，有可能分配不到空间。

4.4 地址递减编址方式

或许其它资料中已经描述了“地址递减”编址方式分配内存的概念，所谓“地址递减”是指编译器编译程序时，按变量声明先后，从可分配内存中从高地址向低地址分配内存。什么意思？还是先来看一个例子。

案例 6 是一个有逻辑错误的程序（表 4.7 所示），不妨称其为“变态”程序。那么它是如何 BT 的呢？

表 4.7 “变态”的程序

```
void main(void)
{
    int a=0, b=0;                   //定义两个局部变量
    int i, array[10];               //i 是循环控制量
    for (i=1; i<=10; ++i)
        array[i]=1;                 //将 array[]数组全部置 1
}
```

这个程序没有编译期错误，但却是一个死循环程序。我们想知道的是：它为什么是个死循环，而不是其它什么错误？通过以上文字对内存布局的介绍，我们已经可以很容易解释之。

仿照第 3 节内容可以画出内存布局示意图（如图 4.1 所示，图中起始地址只是一个典型情况。）。

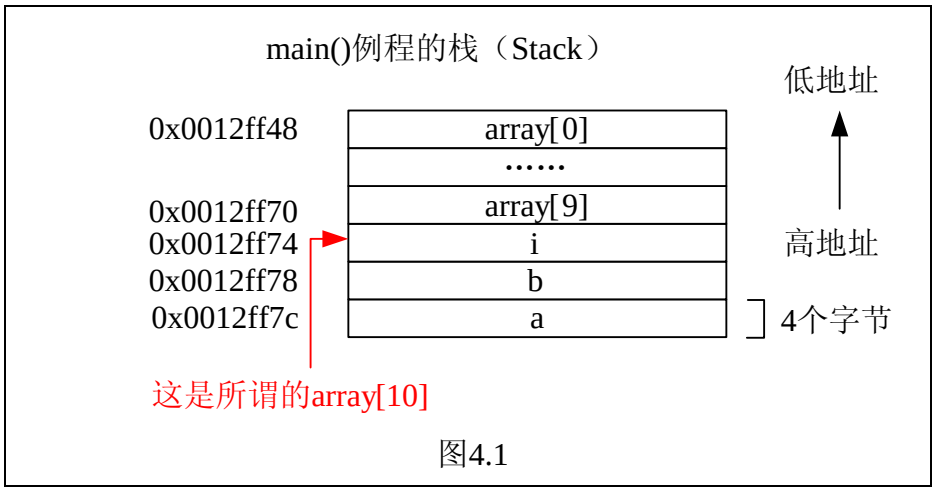

图4.1

注意，程序中引用了 array[10]——数组下标越界（VC ++ 6.0 编译器可以检查出显式的下标越界，但是不检查隐式的下标越界。），循环内部会将所谓的 array[10]置 1，而从图 4.1 可知，array[10]实质上就是 i，导致程序最终死循环也就理所当然了。

一切变得明朗起来，我们不仅解释了程序中的问题，同时还明白了“地址递减”编址方式并不神秘，它原来就是我们前面提到的栈内存区的编址方式。