**C++中堆和栈的完全解析**

博客园 Likwo: <https://www.cnblogs.com/likwo/archive/2010/12/20/1911026.html>

**内存分配方面：**

堆： 操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删 除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码 中的delete语句才能正确的释放本内存空间。我们常说的内存泄露，最常见的就是堆泄露（还有资源泄露），它是指程序在运行中出现泄露，如果程序被关闭掉的话，操作系统会帮助释放泄露的内存。

栈：在函数调用时第一个进栈的主函数中的下一条指令（函数调用语句的下一条可执行语句）的地址然后是函数 的各个参数，在大多数的C编译器中，参数是由右往左入栈，然后是函数中的局部变量。

**一、预备知识—程序的内存分配**

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，呵呵。

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。 - 程序结束后有系统释放

4、文字常量区 —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。

有些说法，把3，4合在一起，也有的把3分成自由存储区（malloc/free）和全局/静态存储区。

这与编译器和操作系统有关。

**二、例子程序**

这是一个前辈写的，非常详细

//main.cpp

int a = 0; 全局初始化区

char \*p1; 全局未初始化区

main()

{

int b; 栈

char s[] = "abc"; 栈 //更正：abc 分配在静态存储区，不是栈上

char \*p2; 栈

char \*p3 = "123456"; 123456\0在常量区，p3在栈上。

static int c =0； 全局（静态）初始化区

p1 = (char \*)malloc(10);

p2 = (char \*)malloc(20);

分配得来得10和20字节的区域就在堆区。

strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方。

}

**二、堆和栈的理论知识**

**2.1申请方式**

stack:

由系统自动分配。 例如，声明在函数中一个局部变量 int b; 系统自动在栈中为b开辟空间

heap:

需要程序员自己申请，并指明大小，在c中malloc函数

如p1 = (char \*)malloc(10);

在C++中用new运算符

如p2 = (char \*)malloc(10);

但是注意p1、p2本身是在栈中的。

**2.2**申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，

会 遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内 存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大 小，系统会自动的将多余的那部分重新放入空闲链表中。

**2.3申请大小的限制**

栈：在Windows下,栈是向低地址扩展的数据结构，是 一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个 编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

**2.4申请效率的比较：**

栈由系统自动分配，速度较快。但程序员是无法控制的。

堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便.

另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一快内存，虽然用起来最不方便。但是速度快，也最灵活

**2.5堆和栈中的存储内容**

栈： 在函数调用时，第一个进栈的是主函数的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

**2.6存取效率的比较**

char s1[] = "aaaaaaaaaaaaaaa";

char \*s2 = "bbbbbbbbbbbbbbbbb";

aaaaaaaaaaa是在运行时刻赋值的；

而bbbbbbbbbbb是在编译时就确定的；

但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

比如：

＃i nclude

void main()

{

char a = 1;

char c[] = "1234567890";

char \*p ="1234567890";

a = c[1];

a = p[1];

return;

}

对应的汇编代码

10: a = c[1];

00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]

0040106A 88 4D FC mov byte ptr [ebp-4],cl

11: a = p[1];

0040106D 8B 55 EC mov edx,dword ptr [ebp-14h]

00401070 8A 42 01 mov al,byte ptr [edx+1]

00401073 88 45 FC mov byte ptr [ebp-4],al

第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到edx中，在根据edx读取字符，显然慢了。

**2.7小结：**

堆和栈的区别可以用如下的比喻来看出：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

堆和栈的区别主要分：

操作系统方面的堆和栈，如上面说的那些，不多说了。

还有就是数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第1个元素有最高的优先权；栈实际上就是满足先进后出的性质的数学或数据结构。

虽然堆栈，堆栈的说法是连起来叫，但是他们还是有很大区别的，连着叫只是由于历史的原因。

**2.8 补充知识：**

堆(heap)和栈(stack)是C/C++编程不可避免会碰到的两个基本概念。首先，这两个概念都可以在讲数据结构的书中找到，他们都是基本的数据结构，虽然栈更为简单一些。

在具体的C/C++编程框架中，这两个概念并不是并行的。对底层机器代码的研究可以揭示，栈是机器系统提供的数据结构，而堆则是C/C++函数库提供的。

具 体地说，现代计算机(串行执行机制)，都直接在代码底层支持栈的数据结构。这体现在，有专门的寄存器指向栈所在的地址，有专门的机器指令完成数据入栈出栈 的操作。这种机制的特点是效率高，支持的数据有限，一般是整数，指针，浮点数等系统直接支持的数据类型，并不直接支持其他的数据结构。因为栈的这种特点， 对栈的使用在程序中是非常频繁的。对子程序的调用就是直接利用栈完成的。机器的call指令里隐含了把返回地址推入栈，然后跳转至子程序地址的操作，而子 程序中的ret指令则隐含从堆栈中弹出返回地址并跳转之的操作。C/C++中的自动变量是直接利用栈的例子，这也就是为什么当函数返回时，该函数的自动变 量自动失效的原因(因为堆栈恢复了调用前的状态)。

和栈不同，堆的数据结构并不是由系统(无论是机器系统还是操作系统)支持的，而是由函数 库提供的。基本的malloc/realloc/free函数维护了一套内部的堆数据结构。当程序使用这些函数去获得新的内存空间时，这套函数首先试图从 内部堆中寻找可用的内存空间，如果没有可以使用的内存空间，则试图利用系统调用来动态增加程序数据段的内存大小，新分配得到的空间首先被组织进内部堆中 去，然后再以适当的形式返回给调用者。当程序释放分配的内存空间时，这片内存空间被返回内部堆结构中，可能会被适当的处理(比如和其他空闲空间合并成更大 的空闲空间)，以更适合下一次内存分配申请。这套复杂的分配机制实际上相当于一个内存分配的缓冲池(Cache)，使用这套机制有如下若干原因：

1. 系统调用可能不支持任意大小的内存分配。有些系统的系统调用只支持固定大小及其倍数的内存请求(按页分配)；这样的话对于大量的小内存分类来说会造成浪费。

2. 系统调用申请内存可能是代价昂贵的。系统调用可能涉及用户态和核心态的转换。

3. 没有管理的内存分配在大量复杂内存的分配释放操作下很容易造成内存碎片。

**堆和栈的对比**

从 以上知识可知，栈是系统提供的功能，特点是快速高效，缺点是有限制，数据不灵活；而堆是函数库提供的功能，特点是灵活方便，数据适应面广泛，但是效率有一 定降低。栈是系统数据结构，对于进程/线程是唯一的；堆是函数库内部数据结构，不一定唯一。不同堆分配的内存逻辑上无法互相操作。栈空间分静态分配和动态 分配两种。静态分配是编译器完成的，比如自动变量(auto)的分配。动态分配由alloca函数完成。栈的动态分配无需释放(是自动的)，也就没有释放 函数。为可移植的程序起见，栈的动态分配操作是不被鼓励的！堆空间的分配总是动态的，虽然程序结束时所有的数据空间都会被释放回系统，但是精确的申请内存 /释放内存匹配是良好程序的基本要素。

堆和栈究竟有什么区别？

主要的区别由以下几点：

1、管理方式不同；

2、空间大小不同；

3、能否产生碎片不同；

4、生长方向不同；

5、分配方式不同；

6、分配效率不同；

管理方式：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生memory leak。

空间大小：一般来讲在32位系统下，堆内存可以达到4G的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在VC6下面，默认的栈空间大小是1M（好像是，记不清楚了）。当然，我们可以修改：

打开工程，依次操作菜单如下：Project->Setting->Link，在Category 中选中Output，然后在Reserve中设定堆栈的最大值和commit。

注意：reserve最小值为4Byte；commit是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。

碎片问题：对于堆来讲，频繁的new/delete势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题， 因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的 可以参考数据结构，这里我们就不再一一讨论了。

生长方向：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。

分配方式：堆都是动态分配的，没有静态分配的堆。栈有2种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由alloca函数进行分配，但是栈的动态分配和堆是不同的，他的动态分配是由编译器进行释放，无需我们手工实现。

分配效率：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比 较高。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆 内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分 到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

从这里我们可以看到，堆和栈相比，由于大量new/delete的使用，容易造成大量的内存碎片；由于没有专门的系统支持，效率很低；由于可能引发用户态 和核心态的切换，内存的申请，代价变得更加昂贵。所以栈在程序中是应用最广泛的，就算是函数的调用也利用栈去完成，函数调用过程中的参数，返回地址， EBP和局部变量都采用栈的方式存放。所以，我们推荐大家尽量用栈，而不是用堆。

虽然栈有如此众多的好处，但是由于和堆相比不是那么灵活，有时候分配大量的内存空间，还是用堆好一些。

无论是堆还是栈，都要防止越界现象的发生（除非你是故意使其越界），因为越界的结果要么是程序崩溃，要么是摧毁程序的堆、栈结构，产生以想不到的结果,就 算是在你的程序运行过程中，没有发生上面的问题，你还是要小心，说不定什么时候就崩掉，那时候debug可是相当困难的：）

对了，还有一件事，如果有人把堆栈合起来说，那它的意思是栈，可不是堆，呵呵，清楚了？

栈到底是什么东西

MSDN中关于Thread Stack Size有这么一段话：

Platform SDK: DLLs, Processes, and Threads

Thread Stack Size

Each new thread receives its own stack space, consisting of both committed and reserved memory. The system will commit one page blocks from the reserved stack memory as needed, until the stack cannot grow any farther.

The default size for committed and reserved memory is specified in the executable file header. The default reserved memory is one megabyte. To specify a different default stack size, use the STACKSIZE statement in the module definition (.DEF) file. Your linker may also support a command-line option for setting the stack size. For more information, see the documentation included with your linker.

Threads that call the C run-time library or the Windows API must allow sufficient stack space for use of these functions. Do not lower the reserved stack size below 64 KB.

To increase the amount of stack space which is to be initially committed for a thread, specify the value in the dwStackSize parameter of the CreateThread or CreateRemoteThread function. This value is rounded to the nearest page. The call to create the thread fails if there is not enough memory to commit or reserve the number of bytes requested. If dwStackSize is smaller than the default reserve size, the new thread uses the default reserve size. If dwStackSize is larger than the default reserve size, the reserve size is rounded up to the nearest multiple of 1 MB.

Windows Server 2003 and Windows XP: If the dwCreationFlags parameter of CreateThread or CreateRemoteThread is STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION, the dwStackSize parameter specifies the amount of stack space which is to be initially reserved for the thread.

The stack is freed when the thread terminates.

Reference:

http://blog.csdn.net/fondiyass007/archive/2007/06/13/1650683.aspx

http://blog.csdn.net/szs1860/archive/2007/05/12/1606192.aspx

（转）http://blog.csdn.net/jsjwql/archive/2007/09/10/1779516.aspx