堆和栈的理解和区别，C语言堆和栈完全攻略

在计算机领域，堆栈绝对是一个不容忽视的概念，并且在编写 C 语言程序的时候也会频繁用到。但对大多数 C 语言初学者来说，堆栈却是一个很模糊的概念。“堆栈：一种数据结构，一个在程序运行时用于存放的地方”，相信这可能是很多初学者共同的认识，这也是大部分教科书对“堆栈”的解释。

很显然，用这么简单的概括来解释“堆栈”是不合适的。要深刻认识堆和栈的概念与区别，还必须从如下两方面说起。

数据结构的堆和栈

在数据结构中，栈是一种可以实现“先进后出”（或者称为“后进先出”）的存储结构。假设给定栈 S=（a_0，a_1，…，a_{n-1}），则称 a_0 为栈底，a_{n-1} 为栈顶。进栈则按照 a_0，a_1，…，a_{n-1} 的顺序进行进栈；而出栈的顺序则需要反过来，按照“后存放的先取，先存放的后取”的原则进行，则 a_{n-1} 先退出栈，然后 a_{n-2} 才能够退出，最后再退出 a_0。

在实际编程中，可以通过两种方式来实现：使用数组的形式来实现栈，这种栈也称为静态栈；使用链表的形式来实现栈，这种栈也称为动态栈。

相对于栈的“先进后出”特性，堆则是一种经过排序的树形数据结构，常用来实现优先队列等。假设有一个集合 K={k_0，k_1，…，k_{n-1}}，把它的所有元素按完全二叉树的顺序存放在一个数组中，并且满足：

$$k_i \leqslant k_{2i+1} \text{ 且 } k_i \leqslant k_{2i+2} \text{（或者 } k_i \geqslant k_{2i+1} \text{ 且 } k_i \geqslant k_{2i+2}\text{）（}i=0,\ 1,\ \cdots,\ (n-2)/2\text{）}$$

则称这个集合 K 为最小堆（或者最大堆）。

由此可见，堆是一种特殊的完全二叉树。其中，节点是从左到右填满的，并且最后一层的树叶都在最左边（即如果一个节点没有左儿子，那么它一定没有右儿子）；每个节点的值都小于（或者都大于）其子节点的值。

内存分配中的堆和栈

在 C 语言中，内存分配方式不外乎有如下三种形式：

1. 从静态存储区域分配：它是由编译器自动分配和释放的，即内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在，直到整个程序运行结束时才被释放，如全局变量与 static 变量。
2. 在栈上分配：它同样也是由编译器自动分配和释放的，即在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元将被自动释放。需要注意的是，栈内存分配运算内置于处理器的指令集中，它的运行效率一般很高，但是分配的内存容量有限。
3. 从堆上分配：也被称为动态内存分配，它是由程序员手动完成申请和释放的。即程序在运行的时候由程序员使用内存分配函数（如 malloc 函数）来申请任意多少的内存，使用完之后再由程序员自己负责使用内存释放函数（如 free 函数）来释放内存。也就是说，动态内存的整个生存期是由程序员自己决定的，使用非常灵活。需要注意的是，如果在堆上分配了内存空间，就必须及时释放它，否则将会导致运行的程序出现内存泄漏等错误。

由此可见，内存分配的堆栈与数据结构中所阐述的堆栈有着本质的区别，这一点千万不要混淆。同样，在内存分配中的堆和栈也存在着很大的区别，也不要混淆这两者的概念。为了加深理解，看下面一段示例代码：

```
01.  #include <stdio.h>
02.  #include <malloc.h>
03.  int main(void)
04.  {
05.      /*在栈上分配*/
06.      int  i1=0;
07.      int  i2=0;
08.      int  i3=0;
09.      int  i4=0;
10.      printf("栈：向下\n");
11.      printf("i1=0x%08x\n",&i1);
12.      printf("i2=0x%08x\n",&i2);
13.      printf("i3=0x%08x\n",&i3);
14.      printf("i4=0x%08x\n\n",&i4);
15.       printf("--------------------\n\n");
16.      /*在堆上分配*/
17.      char  *p1 = (char *)malloc(4);
18.      char  *p2 = (char *)malloc(4);
19.      char  *p3 = (char *)malloc(4);
20.      char  *p4 = (char *)malloc(4);
21.      printf("p1=0x%08x\n",p1);
22.      printf("p2=0x%08x\n",p2);
23.      printf("p3=0x%08x\n",p3);
24.      printf("p4=0x%08x\n",p4);
25.      printf("堆：向上\n\n");
26.      /*释放堆内存*/
27.      free(p1);
28.      p1=NULL;
29.      free(p2);
```

```
30.     p2=NULL;
31.     free(p3);
32.     p3=NULL;
33.     free(p4);
34.     p4=NULL;
35.     return 0;
36. }
```

该示例代码主要演示了在内存分配中的堆和栈的区别，其运行结果为：

栈：向下
i1=0x0060fefc
i2=0x0060fef8
i3=0x0060fef4
i4=0x0060fef0

---

p1=0x00bd14e0
p2=0x00bd3148
p3=0x00bd3158
p4=0x00bd3168
堆：向上

从运行结果中不难发现，内存中的栈区主要用于分配局部变量空间，处于相对较高的地址，其栈地址是向下增长的；而堆区则主要用于分配程序员申请的内存空间，堆地址是向上增长的。

内存分配中的栈与堆主要存在如下区别。

1) 分配与释放方式

栈内存是由编译器自动分配与释放的，它有两种分配方式：静态分配和动态分配。

- 静态分配是由编译器自动完成的，如局部变量的分配（即在一个函数中声明一个 int 类型的变量i 时，编译器就会自动开辟一块内存以存放变量 i）。与此同时，其生存周期也只在函数的运行过程中，在运行后就释放，并不可以再次访问。
- 动态分配由 alloca 函数进行分配，但是栈的动态分配与堆是不同的，它的动态分配是由编译器进行释放，无需任何手工实现。值得注意的是，虽然用 alloca 函数可以实现栈内存的动态分配，但 alloca 函数的可移植性很差，而且在没有传统堆栈的机器上很难实现。因此，不宜使用于广泛移植的程序中。当然，完全可以使用 C99 中的变长数组来替代 alloca 函数。

而堆内存则不相同，它完全是由程序员手动申请与释放的，程序在运行的时候由程序员使用内存分

配函数（如 malloc 函数）来申请任意多少的内存，使用完再由程序员自己负责使用内存释放函数（如 free 函数）释放内存，如下面的代码所示：

```
/*分配堆内存*/
char  *p1 = (char *)malloc(4);
… …
/*释放堆内存*/
free(p1);
p1=NULL;
```

对栈内存的自动释放而言，虽然堆上的数据只要程序员不释放空间就可以一直访问，但是，如果一旦忘记了释放堆内存，那么将会造成内存泄漏，导致程序出现致命的潜在错误。

2) 分配的碎片问题

对堆来说，频繁分配和释放（malloc / free）不同大小的堆空间势必会造成内存空间的不连续，从而造成大量碎片，导致程序效率降低；而对栈来讲，则不会存在这个问题。

3) 分配的效率

大家都知道，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，例如，分配专门的寄存器存放栈的地址，压栈出栈都有专门的执行指令，这就决定了栈的效率比较高。一般而言，只要栈的剩余空间大于所申请空间，系统就将为程序提供内存，否则将报异常提示栈溢出。

而堆则不同，它是由 C/C++ 函数库提供的，它的机制也相当复杂。例如，为了分配一块堆内存，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节点，然后将该节点从空闲节点链表中删除，并将该节点的空间分配给程序。而对于大多数系统，会在这块内存空间的首地址处记录本次分配的大小，这样，代码中的 delete 语句才能正确释放本内存空间。另外，由于找到的堆节点的大小不一定正好等于申请的大小，系统会自动将多余的那部分重新放入空闲链表中。很显然，堆的分配效率比栈要低得多。

4) 申请的大小限制

由于操作系统是用链表来存储空闲内存地址（内存区域不连续）的，同时链表的遍历方向是由低地址向高地址进行的。因此，堆内存的申请大小受限于计算机系统中有效的虚拟内存。

而栈则不同，它是一块连续的内存区域，其地址的增长方向是向下进行的，向内存地址减小的方向增长。由此可见，栈顶的地址和栈的最大容量一般都是由系统预先规定好的，如果申请的空间超过栈的剩余空间时，将会提示溢出错误。由此可见，相对于堆，能够从栈中获得的空间相对较小。

5) 存储的内容

对栈而言，一般用于存放函数的参数与局部变量等。例如，在函数调用时，第一个进栈的是（主函数中的）调用处的下一条指令（即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 C 编译器中，参数是由右往左入栈的，最后是函数中的局部变量（注意 static 变量是不入栈的）。

当本次函数调用结束后，遵循“先进后出”（或者称为“后进先出”）的规则，局部变量先出栈，然后是参数，最后栈顶指针指向最开始保存的地址，也就是主函数中的下一条指令，程序由该点继续运行。下面的示例代码可以清晰反映这种入栈顺序：

```
01.  void f(int i)
02.  {
03.      printf("%d,%d,%d,%d\n", i, i++, i++, i++);
04.  }
05.  int main(void)
06.  {
07.      int i = 1;
08.      f(i);
09.      return 0;
10.  }
```

由于栈的“先进后出”规则，所以程序最后的输出结果是“4，3，2，1”。

对堆而言，具体存储内容由程序员根据需要决定存储数据。

最后介绍一下 C 语言中各类型变量的存储位置和作用域。

- 全局变量。从静态存储区域分配，其作用域是全局作用域，也就是整个程序的生命周期内都可以使用。与此同时，如果程序是由多个源文件构成的，那么全局变量只要在一个文件中定义，就可以在其他所有的文件中使用，但必须在其他文件中通过使用extern关键字来声明该全局变量。
- 全局静态变量。从静态存储区域分配，其生命周期也是与整个程序同在的，从程序开始到结束一直起作用。但是与全局变量不同的是，全局静态变量作用域只在定义它的一个源文件内，其他源文件不能使用。
- 局部变量。从栈上分配，其作用域只是在局部函数内，在定义该变量的函数内，只要出了该函数，该局部变量就不再起作用，该变量的生命周期也只是和该函数同在。
- 局部静态变量。从静态存储区域分配，其在第一次初始化后就一直存在直到程序结束，该变量的特点是其作用域只在定义它的函数内可见，出了该函数就不可见了。