C++内存与堆栈机制

在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

**栈（堆栈）**，就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是局部变量、函数参数等。

**堆**，就是那些由new分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。

**自由存储区**，就是那些由malloc等分配的内存块，他和堆是十分相似的，  
不过它是用free来结束自己的生命的。

**全局/静态存储区**，全局变量和静态变量被分配到同一块内存中，在以前的  
C语言中，全局变量又分为初始化的和未初始化的（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与静态变量在相邻的另一块区域，同时未被初始化的对象存储区可以通过void\*来访问和操纵，程序结束后由系统自行释放），在C++里面没有这个区分了，他们共同占用同一块内存区。

**常量存储区**，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改（当然，你要通过非正当手段也可以修改，而且方法很多）

**明确区分堆与栈**

我们举一个例子：

void f() { int\* p=new int[5]; }

这条短短的一句话就包含了堆与栈，看到new，我们首先就应该想到，我们

分配了一块堆内存，那么指针p呢？他分配的是一块栈内存，所以这句话的意思

就是：在栈内存中存放了一个指向一块堆内存的指针p。在程序会先确定在堆中

分配内存的大小，然后调用operator new分配内存，然后返回这块内存的首地

址，放入栈中。

**堆和栈究竟有什么区别？**

**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。

**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：）

例子：

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  
char \*p2; //栈  
char \*p3 = “123456”; //123456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。  
}

C++反向迭代器

以string为例：

string body = “hello world”;

string::iterator iter = body.begin();

string::iterator iter1 = body.end() - 1;

cout << \*iter << endl;//输出h

cout << \*iter1 << endl;//输出d

cout << "倒序输出：" << endl;

for (string::reverse\_iterator it = body.rbegin(); it != body.rend(); it++)

cout << \*it << endl;//倒序输出“hello world”

三元表达式解析器

一道leetcode上的vip题，实现并记录一下。

题目：

Given a string representing arbitrarily nested ternary expressions, calculate the result of the expression. You can always assume that the given expression is valid and only consists of digits 0-9, ?, :, T and F (T and Frepresent True and False respectively).

Note:

The length of the given string is ≤ 10000.

Each number will contain only one digit.

The conditional expressions group right-to-left (as usual in most languages).

The condition will always be either T or F. That is, the condition will never be a digit.

The result of the expression will always evaluate to either a digit 0-9, T or F.

Input：“T?2:3”

Output：“2”

Input：“F?1:T?4:5”

Output：“4”

Input：“T?T?F:5:3”

Output：“F”

stack<char> body;

string input = "";

bool flag = true;

cin >> input;

string::iterator iter = input.begin();

while (iter != input.end()) {

body.push(\*iter);

iter++;

}

input = "";

while (!body.empty()) {

if (body.top() != '?') {

input += body.top();

body.pop();

}

else {

body.pop();//先将问号弹出

if (body.top() == 'T') {

body.pop();//弹出符号T

for (string::reverse\_iterator it = input.rbegin(); it != input.rend(); it++) {

if (\*it == ':' && flag) {

flag = false;

it += 2;

if (it > input.rbegin())

break;

continue;

}

body.push(\*it);

}

}

else {

body.pop();//弹出符号F

for (string::reverse\_iterator it = input.rbegin(); it != input.rend(); it++) {

if (\*it == ':' && flag) {

flag = false;

continue;

}

if (!flag) {

body.push(\*it);

}

}

}

flag = true;

input = "";

}

}

cout << input << endl;

C++中的0、NULL、nullptr

本质上讲：

1) 0是int型的字面值常量

2) NULL 是预处理变量，定义在 cstdlib 中，其值是0

3) nullptr 是 nullptr\_t 类型的字面值。

cstdlib 中 NULL 的定义

#ifdef \_\_cplusplus

#define NULL 0

#else

#define NULL ((void \*)0)

#endif

之所以这样定义，是因为在C语言中，允许void\*类型隐式转换为任意指针类型，而C++中不允许这样的强制类型转换，但是可以为任意类型的指针赋0值，因此，在C++中将NULL定义为0。

为了避免“野指针”（即指针在首次使用之前没有进行初始化）的出现，我们声明一个指针后最好马上对其进行初始化操作。如果暂时不明确该指针指向哪个变量，则需要赋予NULL值。除了NULL之外，C++11新标准中又引入了nullptr来声明一个“空指针”，之所以引入nullptr，是因为NULL在C++中不完全兼容C，C中NULL的定义为：

#define NULL ((void \*)0)

也就是说NULL实质上是一个void\*指针。NULL在C++中不完全兼容C的原因和C++的函数重载机制有关。如果C++让NULL也支持void \*的隐式类型转换，这样编译器就不知道应该调用哪一个函数。例子如下：

void Func(char \*);

void Func(int);

int main()

{

Func(NULL);

}

如果C++让NULL也支持void\*的隐式类型转换，这样编译器就不知道该调用哪个函数，C++把NULL定义为0，解决了函数重载后的函数匹配问题，但是又引入了另一个问题，同样是这一段代码，由于我们经常使用NULL表示空指针，所以从程序员的角度来看，Func（NULL）应该调用的是Func（char \*）但实际上NULL的值是0，所以调用了Func（int）。nullptr关键字真是为了解决这个问题而引入的。另外我们还有注意到NULL只是一个宏定义，而nullptr是一个C++关键字。

nullptr的使用：nullptr关键字用于标识指针，是std::nullptr\_t类型的（constexpr）变量。它可以转换成任何指针类型和bool布尔类型（主要是为了兼容普通指针可以作为条件判断语句的写法），但是不能被转换为整数：

char \*p1 = nullptr; // 正确

int \*p2 = nullptr; // 正确

bool b = nullptr; // 正确. if(b)判断为false

int a = nullptr; // 错误