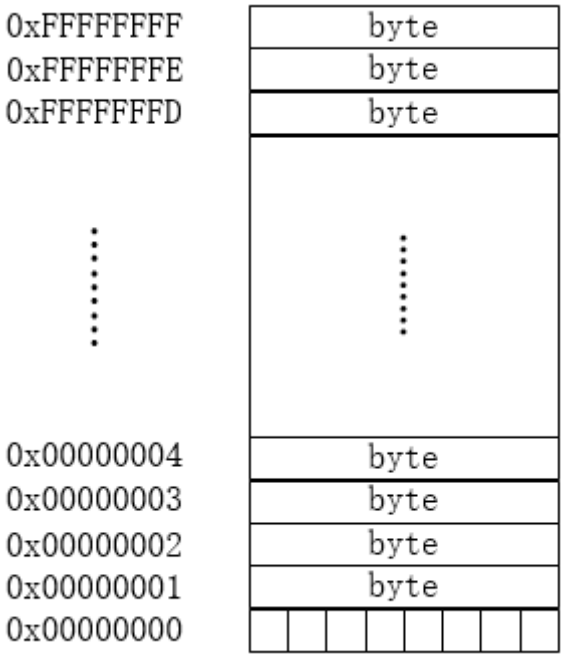
1 什么是指针

1.1.1、什么是指针

C语言里，变量存放在内存中，而**内存其实就是一组有序字节组成的数组**，每个字节有唯一的内存地址。CPU 通过内存寻址对存储在内存中的某个指定数据对象的地址进行定位。这里，数据对象是指存储在内存中的一个指定数据类型的数值或字符串，它们都有一个自己的地址，而指针便是保存这个地址的变量。也就是说：**指针是一种保存变量地址的变量。**

前面已经提到内存其实就是一组有序字节组成的数组，数组中，每个字节大大小固定，都是 8bit。对这些连续的字节从 0 开始进行编号，每个字节都有唯一的一个编号，这个编号就是内存地址。示意如下图：



这是一个 4GB 的内存，可以存放 2^32 个字节的数据。左侧的连续的十六进制编号就是内存地址，每个内存地址对应一个字节的内存空间。而指针变量保存的就是这个编号，也即内存地址。

1.1.2为什么要使用指针

在C语言中，指针的使用非常广泛，因为使用指针往往可以生成更高效、更紧凑的代码。总的来说，使用指针有如下好处：

1）指针的使用使得不同区域的代码可以轻易的共享内存数据，这样可以使程序更为快速高效；

2）C语言中一些复杂的数据结构往往需要使用指针来构建，如链表、二叉树等；

3）C语言是传值调用，而有些操作传值调用是无法完成的，如通过被调函数修改调用函数的对象，但是这种操作可以由指针来完成，而且并不违背传值调用。

1.内存分配函数

1.1.2 指针表达式

```
char ch = 'a';
char *cp = &ch;
```

现在有了两个变量，一个ch（字符类型变量），一个cp（指向字符类型的指针变量）。

> **这个比较好理解**

接下来我们一起看看这个

```
ch
&ch
cp
&cp
*cp
*cp+1
*(cp+1)
++cp
cp++
*++cp
*cp++
++*cp
(*cp)++
++*++cp
++*cp++
```

左值(L-value) 和 右值(R-value):

> **左值**就是能够出现在赋值符号左边的东西，
>
> **右值**就是那些可以出现在赋值符号右边的东西了。

1. 较为准确的定义：**左值**指的是如果一个表达式可以引用到某一个对象，并且这个对象是一块内存空间且可以被检查和存储，那么这个表达式就可以做为一个左值。**右值**指的是引用了一个存储在某个内存地址里的数据。从上面的两个定义可以看出，
2. **左值其实要引用一个对象**，而一个对象在我们的程序中又肯定有一个名字或者可以通过一个名字访问到，所以左值又可以归纳为：左值表示程序中必须有一个特定的名字引用到这个值。
3. 而**右值引用的是地址里的内容**，所以相反右值又可以归纳为：右值表示程序中没有一个特定的名字引用到这个值除了用地址。

1.1.3如何定义一个指针

1. int p; //这是一个普通的整型变量
2. int *p; //首先从P 处开始,先与*结合,所以说明P 是一个指针,然后再与int 结合,说明指针所指向的内容的类型为int 型.所以P是一个返回整型数据的指针
3. int p[3]; //首先从P 处开始,先与[]结合,说明P 是一个数组,然后与int 结合,说明数组里的元素是整型的,所以P 是一个由整型数据组成的数组
4. int *p[3]; //首先从P 处开始,先与[]结合,因为其优先级比*高,所以P 是一个数组,然后再与*结合,说明数组里的元素是指针类型,然后再与int 结合,说明指针所指向的内容的类型是整型的,所以P 是一个由返回整型数据的指针所组成的数组
5. int (*p)[3]; //首先从P 处开始,先与*结合,说明P 是一个指针然后再与[]结合(与"()"这步可以忽略,只是为了改变优先级),说明指针所指向的内容是一个数组,然后再与int 结合,说明数组里的元素是整型的.所以P 是一个指向由整型数据组成的数组的指针

1. int **p; //首先从P 开始,先与*结合,说是P 是一个指针,然后再与*结合,说明指针所指向的元素是指针,然后再与int 结合,说明该指针所指向的元素是整型数据.由于二级指针以及更高级的指针极少用在复杂

的类型中,所以后面更复杂的类型我们就不考虑多级指针了,最多只考虑一级指针.

2. int p(int); //从P 处起,先与()结合,说明P 是一个函数,然后进入()里分析,说明该函数有一个整型变量的参数,然后再与外面的int 结合,说明函数的返回值是一个整型数据
3. Int (*p)(int); //从P 处开始,先与指针结合,说明P 是一个指针,然后与()结合,说明指针指向的是一个函数,然后再与()里的int 结合,说明函数有一个int 型的参数,再与最外层的int 结合,说明函数的返回类型是整型,所以P 是一个指向有一个整型参数且返回类型为整型的函数的指针
4. int *(*p(int))[3]; //可以先跳过,不看这个类型,过于复杂从P 开始,先与()结合,说明P 是一个函数,然后进入()里面,与int 结合,说明函数有一个整型变量参数,然后再与外面的*结合,说明函数返回的是一个指针,,然后到最外面一层,先与[]结合,说明返回的指针指向的是一个数组,然后再与*结合,说明数组里的元素是指针,然后再与int 结合,说明指针指向的内容是整型数据.所以P 是一个参数为一个整数据且返回一个指向由整型指针变量组成的数组的指针变量的函数.

说到这里也就差不多了,我们的任务也就这么多,理解了这几个类型,其它的类型对我们来说也是小菜了,不过我们一般不会用太复杂的类型,那样会大大减小程序的可读性,请慎用,这上面的几种类型已经足够我们用了.

运算符优先级

```
1.::{作用域}
2.(){函数调用，类型构造：type（exp）},[]{下标},.{成员选择},->{成员选择}
3.++{后置},--{后置},typeid{类型id},explicit_cast{四种类型转换}
4.++{前置},--{前置},~{取反},!{逻辑非},-{一元负},+{一元正},\*{指针指向值},&{取地址},(){老式类型转换},sizeof{对象大小}
5.sizeof{类型或参数包的大小},new{分配内存},delete{释放内存},noexcept{能否抛出异常}
6.->\*{指向成员中的指针},.\*{指向成员中的指针}
7.*,/,%
8.+,-
9.<<,>>
10.<,>,<=,>=
11.==,!=
12.&
```

1.1.4 指针自身类型

从语法的角度看，你只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。让我们看看例一中各个指针的类型：

```
(1)int * ptr;//指针的类型是int *

(2)char * ptr;//指针的类型是char *

(3)int ** ptr;//指针的类型是int **

(4)int( *  ptr)[3];//指针的类型是int(*)[3]

(5)int *  ( * ptr)[4];//指针的类型是int*(*)[4]
```

怎么样？找出指针的类型的方法是不是很简单？

1.1.5 指针所指向的类型 （重要）

当你通过指针来访问指针所指向的内存区时，**指针所指向的类型决定了编译器将把那片内存区里的内容**当做什么来看待。
从语法上看，你只须把指针声明语句中的指针名字和名字左边的指针声明符*去掉，剩下的就是指针所指向的类型。例如：

```
(1)int * ptr; //指针所指向的类型是int

(2)char * ptr; //指针所指向的的类型是char

(3)int * * ptr; //指针所指向的的类型是int*

(4)int( * ptr )[3]; //指针所指向的的类型是int()[3]

(5)int * (  * ptr)[4]; //指针所指向的的类型是int *()[4]
```

在指针的算术运算中，指针所指向的类型有很大的作用。**他决定我们昨天所讲的步长**

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对C 越来越熟悉时，你会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。

每遇到一个指针，都应该问问：这个指针的类型是什么？指针指的类型是什么？该指针指向了哪里？（重点注意）

1.1.6 指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的，以单元为单位。例如：

```
    char a[20] = {'a', 'b', 'c', 'd','e'};
    int *ptr=(int *)a; //强制类型转换并不会改变a 的类型
    ptr++;
    printf("char %c\n", *a);
    printf("char %c\n", *ptr)
```

在上例中，指针ptr 的类型是int*,它指向的类型是int，它被初始化为指向整型变量a。接下来的第3句中，指针ptr被加了1，编译器是这样处理的：它把指针ptr 的值加上了sizeof(int),

int 占4 个字节。由于地址是用字节做单位的，故ptr 所指向的地址由原来的变量a 的地址向高地址方向增加了4 个字节。由于char 类型的长度是一个字节，所以，原来ptr 是指向数组a 的第0 号单元开始的四个字节，此时指向了数组a 中从第4 号单元开始的四个字节。我们可以用一个指针和一个循环来遍历一个数组，看例子：

```
    int array[4]={0};
    int *ptr=array;
    //printf("ptr : 0x%x\n",  ptr );
    for(int i=0;i<4;i++)
    {
        (*ptr)++;
        ptr++;
```

```
    }
    for (int i = 0; i < 4; ++i) {
        printf("value: %d\n", *(array+i));
    }
    //printf("ptr : 0x%x\n",  ptr );
    //printf("ptr : %d\n",  *(ptr-1) );
    return 0;
}
```

这个例子将整型数组中各个单元的值加1。由于每次循环都将指针ptr加1 个单元，所以每次循环都能访问数组的下一个单元。

```
    char a[20]="You_are_a_girl";
    int *ptr=(int *)a;
    printf("* ptr addr %p\n",  ptr);
    ptr+=1;
    printf("* ptr addr %p\n",  ptr);
    printf("* ptr=%c\n", *ptr);
```

在这个例子中，ptr 被加上了1，编译器是这样处理的：将指针ptr 的值加上1 乘sizeof(int)， 由于地址的单位是字节，故现在的ptr 所指向的地址比起加4 后的ptr 所指向的地址来说，向高地址方向移动了4 个字节。

> 假设 加的不是1 而是 加的是 5 呢   ptr+=5;

没加5 前的ptr 指向数组a 的第0 号单元开始的四个字节，加5 后，ptr 已经指向了数组a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。如果上例中，ptr 是被减去5，那么处理过程大同小异，只不过ptr 的值是被减去5 乘sizeof(int)，新的ptr 指向的地址将比原来的ptr 所指向的地址向低地址方向移动了20 个字节。

下面请 再举一个例子

```
    char a[20]="You_are_a_girl";
    char *p=a;
    char **ptr=&p;

    printf("ptr=0x%x\n",ptr);
    printf("p=%c\n",*p);
    printf("**ptr=%c \n",**ptr);
    printf("beafor ptr= 0x%x\n", ptr);
    ptr++;

    printf("after ptr= 0x%x\n", ptr);
    printf("ptr= %c\n",**ptr);
```

- **误区一**、输出答案为Y 和o
  误解:ptr 是一个char 的二级指针,当执行ptr++;时,会使指针加一个sizeof(char),所以输出如上结果,这个可能只是少部分人的结果.
- **误区二、**输出答案为Y 和a误解:ptr 指向的是一个char *类型,当执行ptr++;时,会使指针加一个sizeof(char *)(有可能会有人认为这个值为1,那就会得到误区一的答案,这个值应该是4,参考前面内容), 即&p+4; 那进行一次取值运算不就指向数组中的第五个元素了吗?那输出的结果不就是数组中第五个元素了吗?答案是否定的.
- **正解:** ptr 的类型是char **,指向的类型是一个char * 类型,该指向的地址就是p的地址(&p),当执行ptr++;时,会使指针加一个sizeof(char*),即&p+4;那*(&p+4)指向哪呢,这个你去问上帝吧,或者他会告诉

你在哪?所以最后的输出会是一个随机的值,或许是一个非法操作.

1.1.7指针总结:

一个指针ptr 加(减)一个整数n 后，

> 1. 结果是一个新的指针ptr_new，ptr_new 的类型和ptr_old 的类型相同，
> 2. ptr_new 所指向的类型和ptr_old所指向的类型也相同。
> 3. ptr_new 的值将比ptr_old 的值增加(减少)了n 乘sizeof(ptrold 所指向的类型)个字节。

就是说，ptr_new 所指向的内存区将比ptr_old 所指向的内存区向高(低)地址方向移动了n 乘 sizeof(ptr_old 所指向的类型)个字节。

指针和指针进行加减：两个指针不能进行加法运算，这是非法操作，因为进行加法后，得到的结果指向一个不知所向的地方，而且毫无意义。

对自身的指针可以进行加减1操作， 一般用在数组方面，不多说了。

2 内存分配

C语言的标准内存分配函数：malloc，calloc，realloc，free等。

区别：

malloc与calloc的区别为1个size与n个size大小内存的区别：

使用方式

> malloc调用形式为(类型) **malloc(size)**：在内存的动态存储区中分配一块长度为“size”字节的连续区域，返回该区域的首地址。
>
> calloc调用形式为(类型) **calloc(n，size)**：在内存的动态存储区中分配n块长度为“size”字节的连续区域，返回首地址。
>
> realloc调用形式为(类型) **realloc(*ptr，size)**：将ptr内存大小增大到size。
>
> free的调用形式为(类型) **free(void *ptr)**：释放ptr所指向的一块内存空间。

2.1.1 共同点就是：

- 都为了分配存储空间，
- 它们返回的是 void * 类型，也就是说如果我们要为int或者其他类型的数据分配空间必须显式强制转换；

2.1.2 不同点是：

- malloc一个形参，因此如果是数组，必须由我们计算需要的字节总数作为形参传递
  用**malloc只分配空间不初始化**，也就是依然保留着这段内存里的数据，
- calloc 2个形参 ,因此如果是数组，需要传递个数和数据类型
  而calloc则进行了初始化，**calloc分配的空间全部初始化为0**，这样就避免了可能的一些数据错误。

3 内存管理机制

内存资源是非常有限的。尤其对于移动端开发者来说，硬件资源的限制使得其在程序设计中首要考虑的问题就是如何有效地管理内存资源。本文是作者在学习C语言内存管理的过程中做的一个总结。

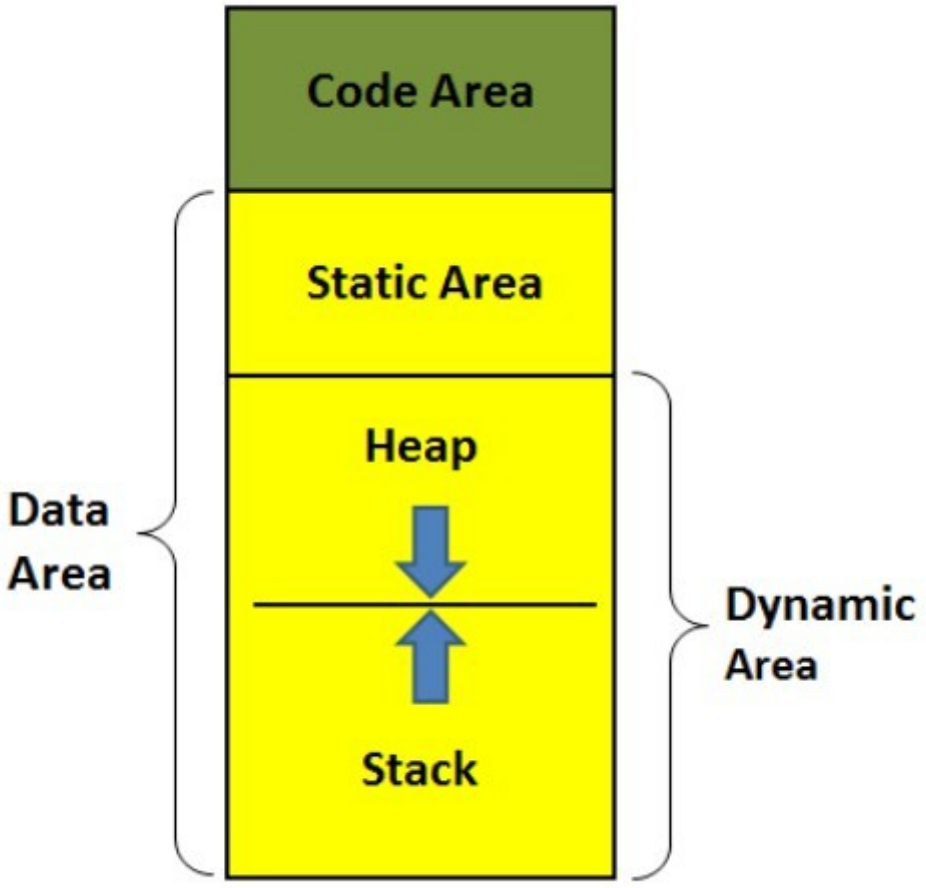
3.1 变量概念：

- 全局变量（外部变量）：出现在代码块{}之外的变量就是全局变量。
- 局部变量（自动变量）：一般情况下，代码块{}内部定义的变量就是自动变量，也可使用auto显示定义。
- 静态变量：是指内存位置在程序执行期间一直不改变的变量，用关键字static修饰。
  代码块内部的静态变量只能被这个代码块内部访问，代码块外部的静态变量只能被定义这个变量的文件访问。

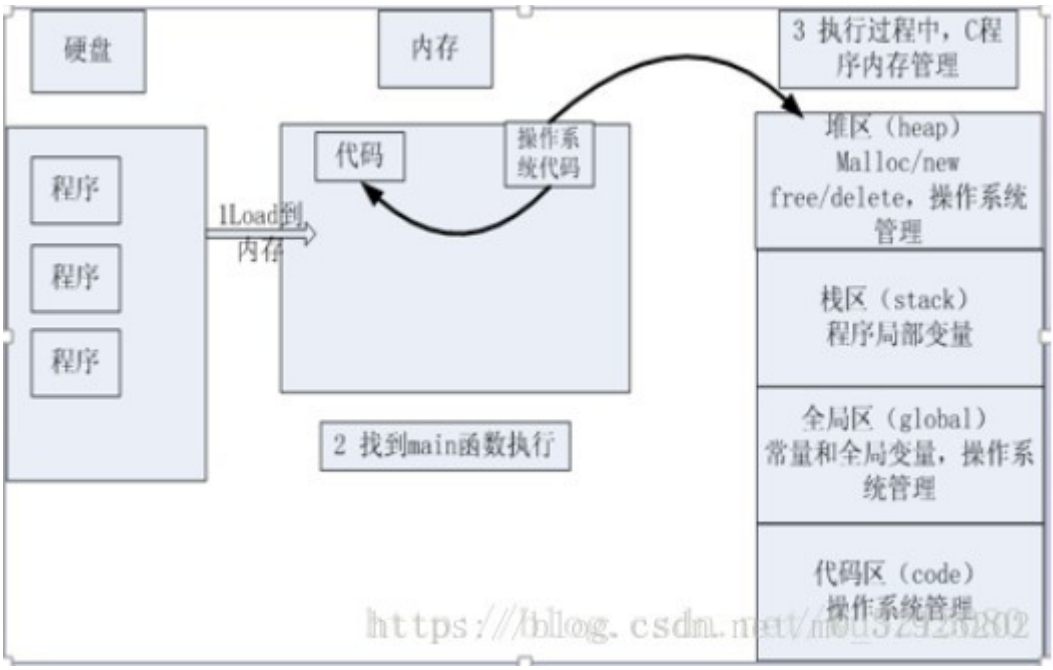
3.1.2 extern关键字：

1、引用同一个文件中的变量；
2、引用另一个文件中的变量；
3、引用另一个文件中的函数。

注意：C语言中函数默认都是全局的，可以使用static关键字将函数声明为静态函数（只能被定义这个函数的文件访问的函数）。



3.1.4 程序执行流程：



代码区：

程序被操作系统加载到内存的时候，所有的可执行代码（程序代码指令、常量字符串等）都加载到代码区，这块内存在程序运行期间是不变的。代码区是平行的，里面装的就是一堆指令，在程序运行期间是不能改变的。函数也是代码的一部分，故函数都被放在代码区，包括main函数。

静态区

静态区存放程序中所有的全局变量和静态变量。

栈区

栈（stack）是一种先进后出的内存结构，所有的自动变量、函数形参都存储在栈中，这个动作由编译器自动完成，我们写程序时不需要考虑。栈区在程序运行期间是可以随时修改的。当一个自动变量超出其作用域时，自动从栈中弹出。

每个线程都有自己专属的栈；
栈的最大尺寸固定，超出则引起栈溢出；
变量离开作用域后栈上的内存会自动释放。

```
int main(int argc, char* argv[])
{
    char array_char[1024*1024*1024] = {0};
    array_char[0] = 'a';
    printf("%s", array_char);
    getchar();
}1234567
```

栈溢出怎么办呢？就该堆出场了。

堆（heap）和栈一样，也是一种在程序运行过程中可以随时修改的内存区域，但没有栈那样先进后出的顺序。更重要的是堆是一个大容器，它的容量要远远大于栈，这可以解决内存溢出困难。一般比较复杂的数据类型都是放在堆中。但是在C语言中，堆内存空间的申请和释放需要手动通过代码来完成。

那堆内存如何使用？

malloc函数用来在堆中分配指定大小的内存，单位为字节（Byte），函数返回void *指针；free负责在堆中释放malloc分配的内存。

```
#include <stdlib.h>
#include<stdio.h>
#include <string.h>

void print_array(char *p, char n)
{
    int i = 0;
    for (i = 0; i < n; i++)
    {
        printf("p[%d] = %d\n", i, p[i]);
    }
}

int main(int argc, char* argv[])
{
    char *p = (char *)malloc(1024 * 1024 * 1024);//在堆中申请了内存
    memset(p, 'a', sizeof(int)* 10);//初始化内存
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        p[i] = i + 65;
    }
    print_array(p, 10);
    free(p);//释放申请的堆内存
    getchar();
}12345678910111213141516171819202122232425 26
```

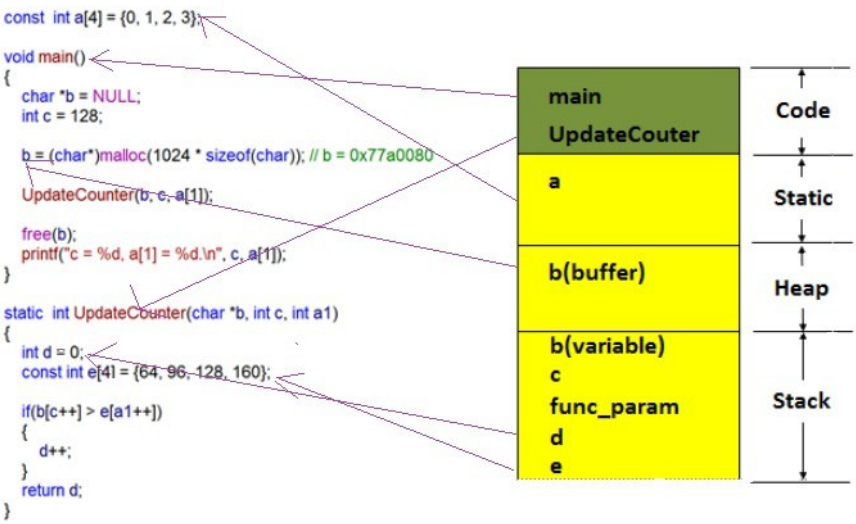
这样就解决了刚才栈溢出问题。堆的容量有多大?理论上讲，它可以使用除了系统占用内存空间之外的所有空间。实际上比这要小些，比如我们平时会打开诸如QQ、浏览器之类的软件，但这在一般情况下足够用了。不能将一个栈变量的地址通过函数的返回值返回，如果我们需要返回一个函数内定义的变量的地址该怎么办？可以这样做：

```
int *getx()
{
    int *p = (int *)malloc(sizeof(int));//申请了一个堆空间
    return p;
}

int main(int argc, char* argv[])
{
    int *pp = getx();
    *pp = 10;
    free(pp);
    return 0;
}
//类似创建链表时，新增一个节点。1234567891011121314
```

可以通过函数返回一个堆地址，但记得一定用通过free函数释放申请的堆内存空间。

分析：



main函数和UpdateCounter为代码的一部分，故存放在代码区

数组a默认为全局变量，故存放在静态区

main函数中的”char *b = NULL”定义了自动变量b(variable)，故其存放在栈区

接着malloc向堆申请了部分内存空间，故这段空间在堆区

需要注意以下几点：

栈是从高地址向低地址方向增长；
在C语言中，函数参数的入栈顺序是从右到左，因此UpdateCounter函数的3个参数入栈顺序是a1、c、b；
C语言中形参和实参之间是值传递，UpdateCounter函数里的参数a[1]、c、b与静态区的a[1]、c、b不是同一个；

3.1.5 内存管理的目的

学习内存管理就是为了知道日后怎么样在合适的时候管理我们的内存。那么问题来了？什么时候用堆什么时候用栈呢？一般遵循以下三个原则：

- 如果明确知道数据占用多少内存，那么数据量较小时用栈，较大时用堆；
- 如果不知道数据量大小（可能需要占用较大内存），最好用堆（因为这样保险些）；
- 如果需要动态创建数组，则用堆。

创建动态数组：

```
//动态创建数组
int main()
{
    int i;
    scanf("%d", &i);
    int *array = (int *)malloc(sizeof(int) * i);
    //...//这里对动态创建的数组做其他操作
    free(array);
    return 0;
}12345678910
```

操作系统在管理内存时，最小单位不是字节，而是内存页（32位操作系统的内存页一般是4K）。比如，初次申请1K内存，操作系统会分配1个内存页，也就是4K内存。4K是一个折中的选择，因为：内存页越大，内存浪费越多，但操作系统内存调度效率高，不用频繁分配和释放内存；内存页越小，内存浪费越少，但操作系统内存调度效率低，需要频繁分配和释放内存。

4 异常指针

空悬指针是这样一种指针：指针正常初始化，曾指向过一个正常的对象，但是对象销毁了，该指针未置空，就成了悬空指针。

野指针 是这样一种指针：未初始化的指针，其指针内容为一个垃圾数。 （一般我们定义一个指针时会初始化为NULL或者直接指向所要指向的变量地址，但是如果我们没有指向NULL或者变量地址就对指针进行使用，则指针指向的内存地址是随机的）。存在野指针是一个严重的错误。

```
int main() {
    int *p; // 指针未初始化，此时 p 为野指针
    int *pi = nullptr;

    {
        int i = 6;
        pi = &i; // 此时 pi 指向一个正常的地址
        *pi = 8; // ok
    }

    *pi = 6; // 由于 pi 指向的变量 i 已经销毁，此时 pi 即成了悬空指针

    return 0;
}
```

空指针与NULL指针

4.1.1 什么是空指针

如果 p 是一个指针变量，则 p = 0; p = 0L; p = '\0'; p = 3 - 3; p = 0 * 17; 中的任何一种赋值操作之后（对于 C 来说还可以是 p = (void*)0;）， p 都成为一个空指针，由系统保证**空指针不指向任何实际的对象或者函数。反过来说，任何对象或者函数的地址都不可能是空指针。**（比如这里的**(void*)0就是一个空指针**

4.1.2什么是NULL指针

NULL 是一个标准规定的宏定义，用来表示空指针常量。因此，除了上面的各种赋值方式之外，还可以用 p = NULL; 来使 p 成为一个空指针。**与上一钟情况相似，只不过是空指针的一种例子**

4.1.3为什么通过空指针读写的时候就会出现异常?

NULL指针分配的分区：其范围是从 0x00000000到0x0000FFFF。这段空间是空闲的，对于空闲的空间而言，没有相应的物理存储器与之相对应，所以对这段空间来说，任何读写操作都是会引起异常的。空指针是程序无论在何时都没有物理存储器与之对应的地址。为了保障"无论何时"这个条件，需要人为划分一个空指针的区域，固有上面NULL指针分区。