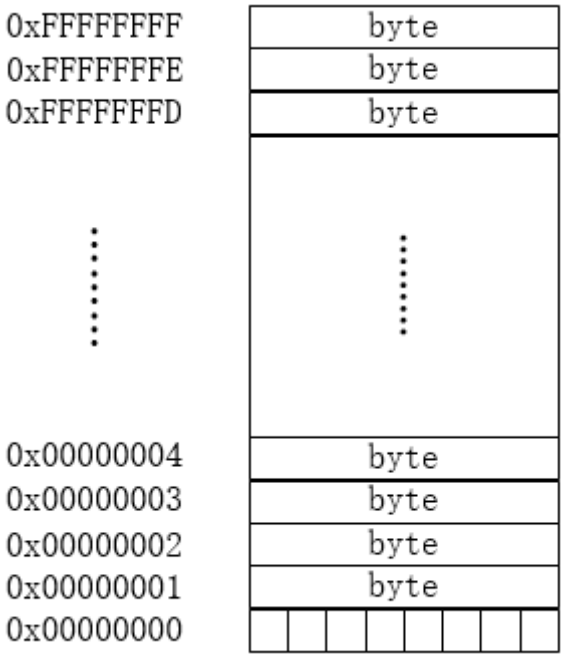
1 什么是指针

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

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。

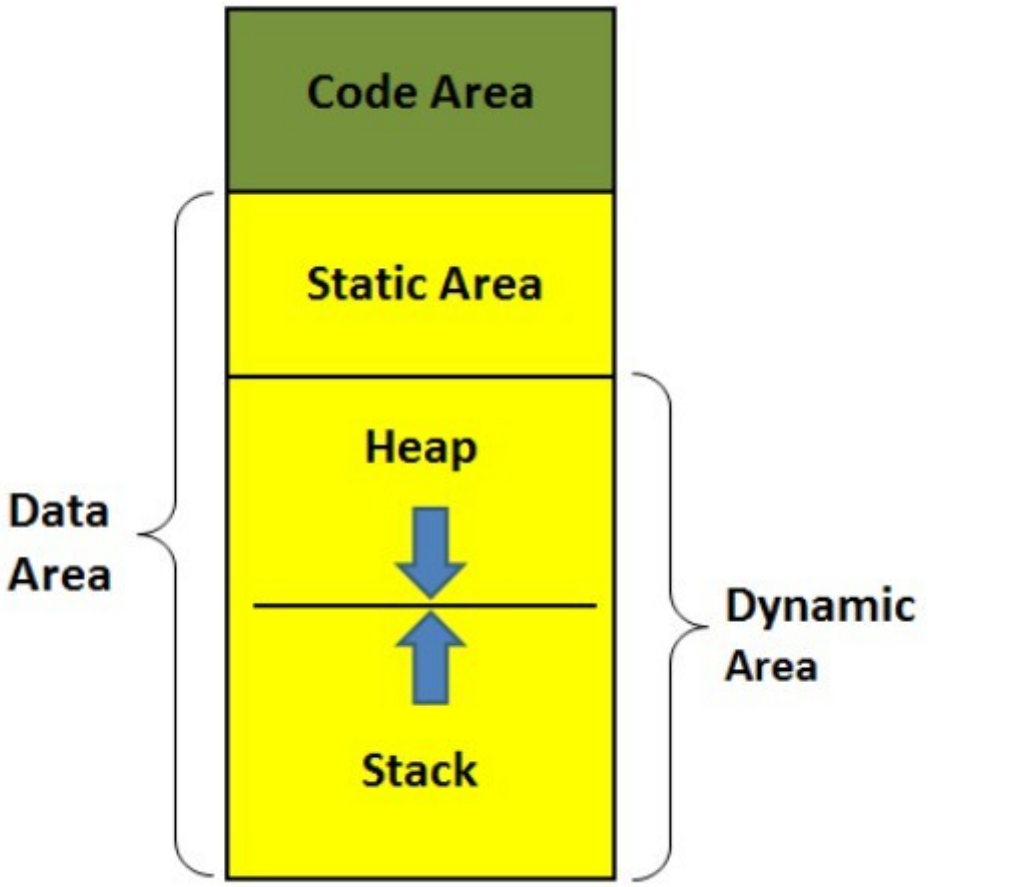
3.1 变量概念：

- 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。

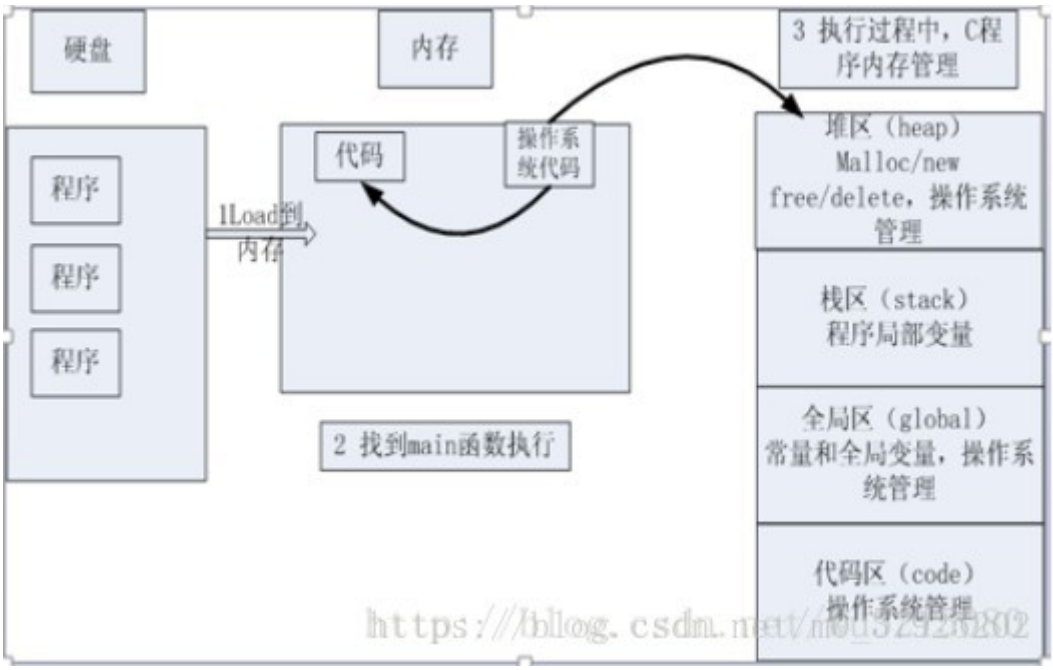
3.1.2 extern关键字：

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
```

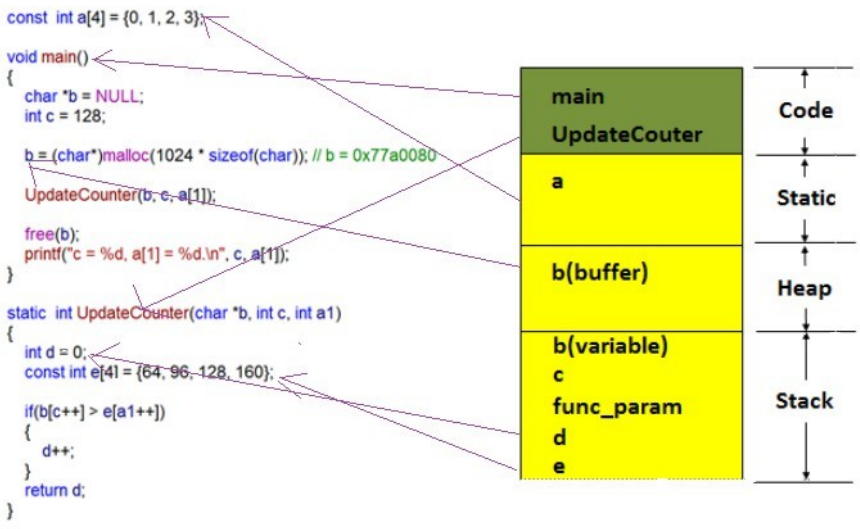
这样就解决了刚才栈溢出问题。堆的容量有多大?理论上讲，它可以使用除了系统占用内存空间之外的所有空间。实际上比这要小些，比如我们平时会打开诸如QQ、浏览器之类的软件，但这在一般情况下足够用了。不能将一个栈变量的地址通过函数的返回值返回，如果我们需要返回一个函数内定义的变量的地址该怎么办？可以这样做：

```
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。