1.1 内存分区，堆区

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。

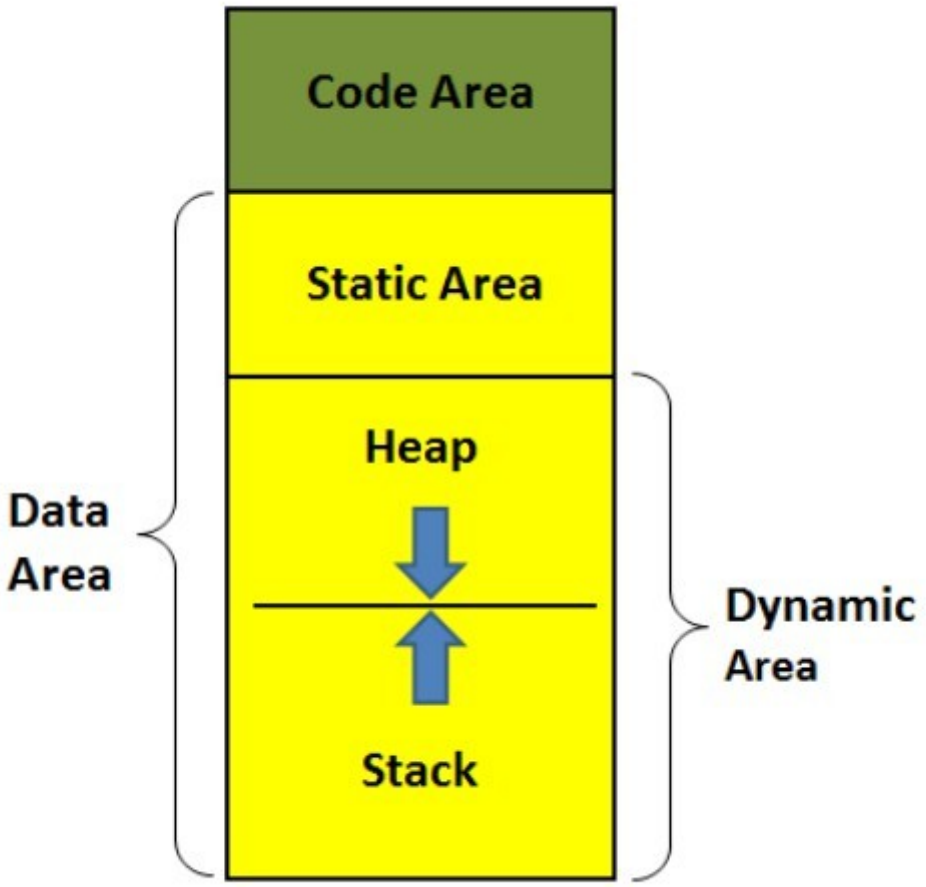
1.1.1 变量概念：

- 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。

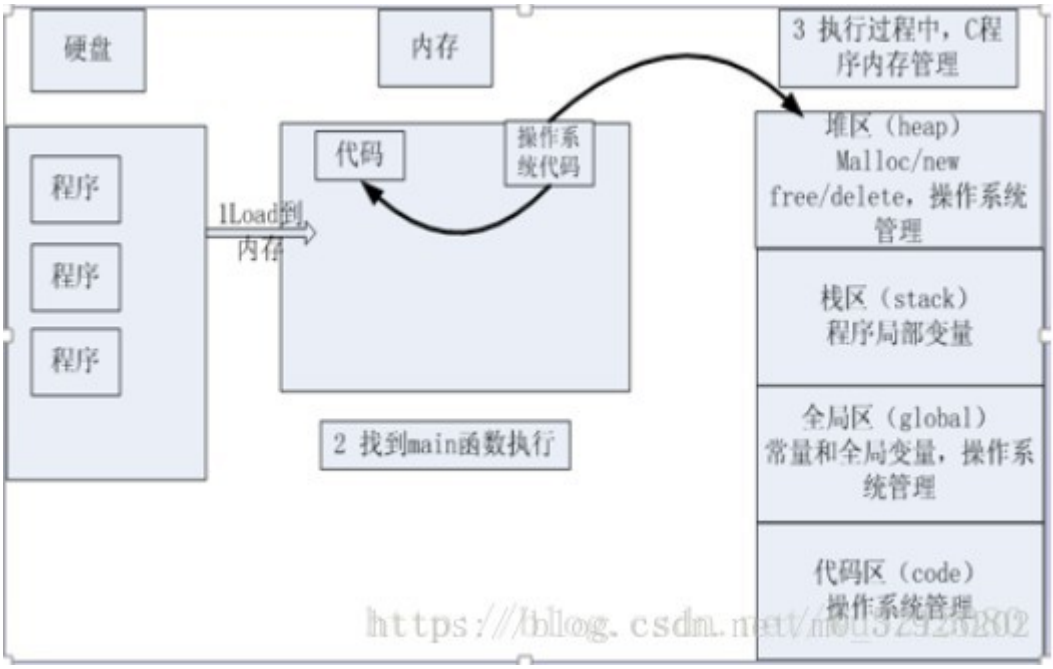
1.1.2 extern关键字：

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）。



程序执行流程：



1.1.3 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。

1.1.4 静态区

静态区存放程序中所有的全局变量和静态变量。

1.1.5 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();
}
```

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();
}
```

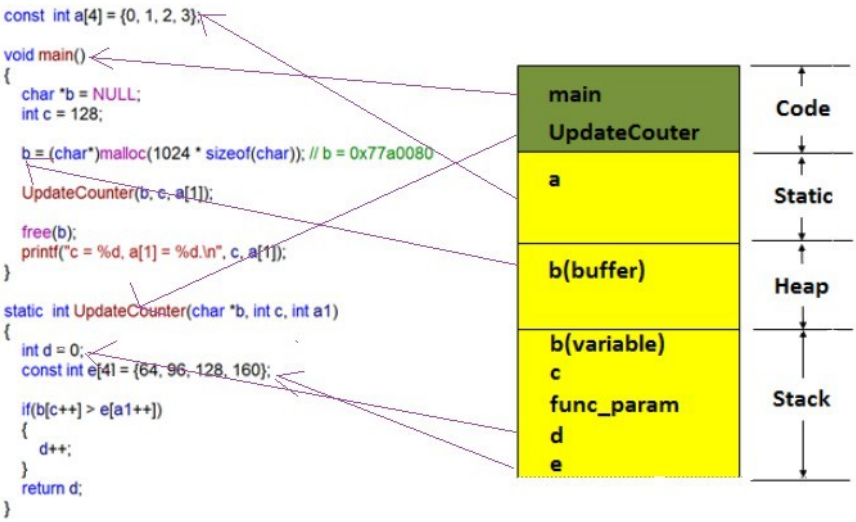
这样就解决了刚才栈溢出问题。堆的容量有多大?理论上讲，它可以使用除了系统占用内存空间之外的所有空间。实际上比这要小些，比如我们平时会打开诸如QQ、浏览器之类的软件，但这在一般情况下足够用了。不能将一个栈变量的地址通过函数的返回值返回，如果我们需要返回一个函数内定义的变量的地址该怎么办？可以这样做：

```
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。
```

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；

1.1.6 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;
}
```

备注：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。

1.2 指针间接传值详解

1.2.1 值传递与指针传递

值传递：

形参是实参的拷贝，改变形参的值并不会影响外部实参的值。从被调用函数的角度来说，值传递是单向的（实参->形参），参数的值只能传入， 不能传出。当函数内部需要修改参数，并且不希望这个改变影响调用者时，采用值传递。

指针传递：

形参为指向实参地址的指针，当对形参的指向操作时，就相当于对实参本身进行的操作

引用传递：

形参相当于是实参的“别名”，对形参的操作其实就是对实参的操作，在引用传递过程中，被调函数的形式参数虽然也作为局部变量在栈

中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过

栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

```
// pass by value
 void swap(int a,int b){
     int temp = a;
     a = b;
     b = temp;
 }
 // pass by address
 void swap1(int* a,int *b){
     int temp = *a;
     *a = *b;
    *b = temp;
 }
 // pass by reference
 void swap2(int& a,int& b){
     int temp = a;
     a = b;
     b = temp;
 }
 // pass by value ?
 void swap3(int* a,int *b){
     int* temp = a;
     a = b;
     b = temp;
 }
```

值传递

这个比较简单，假设实参a 原本指向地址 0x16FF212，代表0x16FF212这个地址的值是3。在swap函数中，实参a将值拷贝给形参a，形参a此时也在内存中拥有地址，假设形参地址=0x16DD111，值为3，在所有的函数体内的操作，都是对形参 0x16DD111这个地址的操作，所以并不会影响实际参数的值。

地址传递

这个对于理不清指针是什么的同学来说比较难。在这里我们习惯把指针写成int* a，int* b而不是int *a，int * b。我们可以这样理解：指针是一种特殊的数据类型，若 int c = 5;int* a = &c;则a是一个指针变量，

它的值是c的地址！星号“*”是一个取值操作，和号“&”是一个取址操作。所以此时单纯看a和b都是一个整数，它们表示地址，进行取值操作之后就可以得到相应地址的值。函数接受两个类型为指针的变量，实际接受的是a和b，即两个地址。所以现在分析函数体：

```
1 int temp = *a;//取出地址a的值，并赋值给整型变量temp
2 *a = *b;       //取出地址b的值，并将这个值赋给地址a指向的值
3 *b = temp;     //将temp的值赋给地址b所指向的值
```

因此，我们看到，由于函数传入的是地址，而函数体内又对地址进行取值和赋值操作，所以相对应的地址的值发生了改变。但是地址并没有实际改变，从函数的输出来看，a的地址并不会改变。在C语言中，函数在运行的时候会对每个变量分配内存地址，分配之后只要变量不被销毁，这个地址不能改变。&a = &b；是无法编译通过的。

引用传递

这个理解起来更简单，我们这样理解引用，引用是变量的一个别名，调用这个别名和调用这个变量是完全一样的。所以swap2的结果可以解释。值得注意的是，由于引用时别名，所以引用并不是一种数据类型，内存并不会给它单独分配内存，而是直接调用它所引用的变量。这个与地址传递也就是指针是不一样的（也就是说一个指针虽然指向一个变量，但是这个指针变量在内存中是有地址分配的），下面代码进行验证。

1.3 宏函数与普通函数

定义:

为了提高程序的效率，我们用define来定义一个函数，这样在频繁调用的时候就不会有函数调用的开销了，这就是宏函数，但宏定义只是把S(a,b)简单地替换成a*b， 这就是宏函数

注意事项

> **宏函数 需要加小括号 修饰，保证运算完整新**

宏函数比普通函数在一定程序上执行效率高，省去函数入栈，出栈时间上的开销

> **优点: 以时间换空间**

老的C语言程序员中有一种倾向，就是把很短的执行频繁的计算写成宏，而不是定义为函数，宏可以避免函数调用的开销。实际上，即使是在C语言刚诞生时(那时的机器非常慢，函数调用的开销也特别大)，这个论据也是很脆弱的，

1.4 二级指针

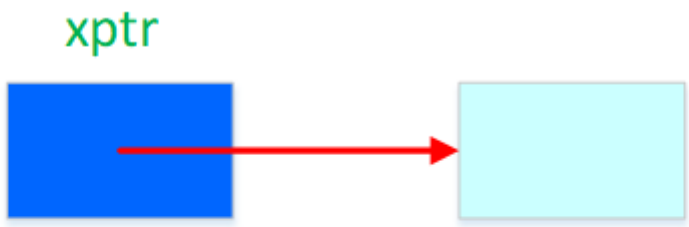
1.4.1一级指针

如下图所示，整型指针xptr指向变量x的地址。

```
int *xptr;
int x=10;

xptr = &x;

12345
```



源码：

```
#include <stdio.h>
int main()
{
    int *xptr = NULL;
    int x = 10;

    xptr = &x;

    printf("x = %d, *xptr = %d\n", x, *xptr);
    printf("&x = %p, xptr = %p\n", &x, xptr);

    return 0;
}

1234567891011121314
```

运行结果：

```
x = 10, *xptr = 10
&x = 0x7fff2a042414, xptr = 0x7fff2a042414
```

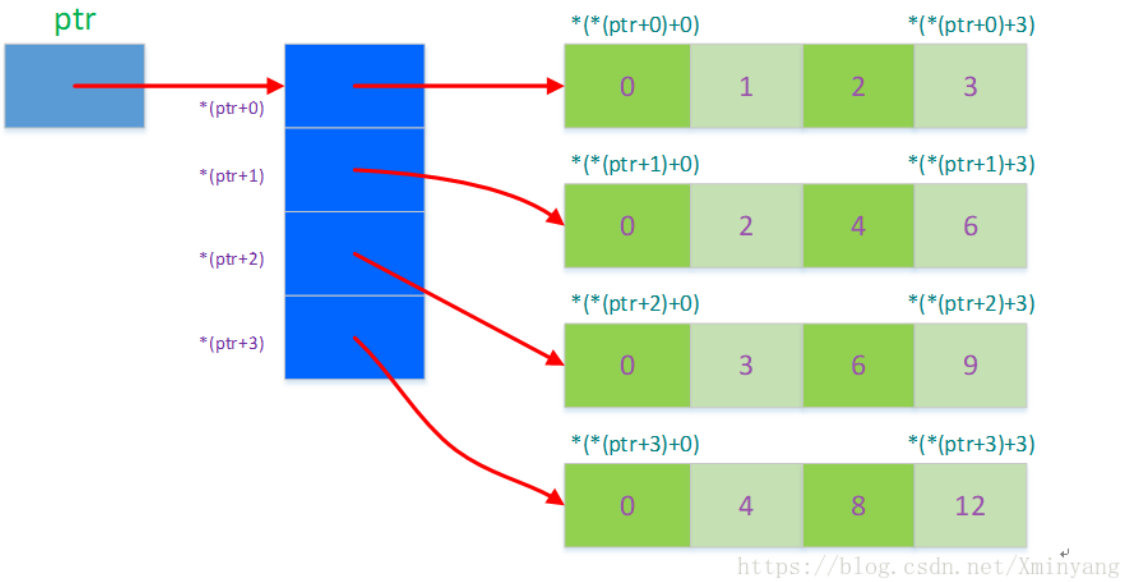

1.4.2 二级指针

1.4.3 实现方法一

如下图所示，先为二级整型指针ptr分配空间，然后赋值。

```
int  **ptr=NULL;
int num=4, size=4, i,j;

ptr = (int **)malloc(num*sizeof(int*));
for(i=0; i<num; ++i)
{
    *(ptr+i) = (int *)malloc(size*sizeof(int));
    *(*(ptr+i) +j)=(i+1)*j;
}
```



源码:

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

int main()
{
    int **ptr = NULL;
    int num=4, size=4, i, j;

    ptr = (int **)malloc(num * sizeof(int *));
    for(i=0; i<num; ++i)
    {
        *(ptr+i) = (int *)malloc(size * sizeof(int));
        for(j=0; j<size; ++j)
        *(*(ptr+i)+j) = (i+1)*j;
    }
    for(i=0; i<num; ++i)
    {
        for(j=0; j<size; ++j)
        {
            printf("(%d, %d) -> %d\t", i, j, *(*(ptr+i)+j));
        }
        printf("\n");
    }
    return 0;
}
```

运行结果：

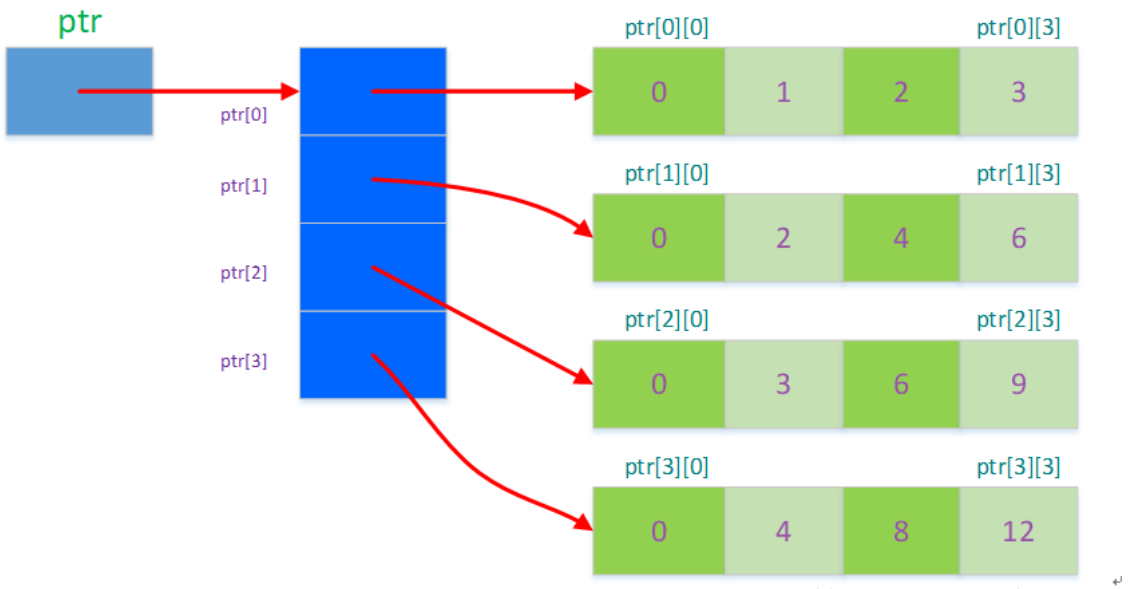
```
(0, 0) -> 0     (0, 1) -> 1     (0, 2) -> 2     (0, 3) -> 3
(1, 0) -> 0     (1, 1) -> 2     (1, 2) -> 4     (1, 3) -> 6
(2, 0) -> 0     (2, 1) -> 3     (2, 2) -> 6     (2, 3) -> 9
(3, 0) -> 0     (3, 1) -> 4     (3, 2) -> 8     (3, 3) -> 12
```


1.2.3 实现方法二

如下图所示，先为二级整型指针ptr分配空间，然后赋值。
与实现方法一的不同之处，在于使用数组形式就行相关操作。

```
int  **ptr=NULL;
int num=4, size=4, i;

ptr = (int **)malloc(num*sizeof(int*));
for(i=0; i<num; ++i)
{
    ptr[i]= (int *)malloc(size*sizeof(int));
    ptr[i][j]=(i+1)*j;
}
```



源码：

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int **ptr = NULL;
    int num=4, size=4, i, j;

    ptr = (int **)malloc(num * sizeof(int *));
    for(i=0; i<num; ++i)
    {
        ptr[i] = (int *)malloc(size * sizeof(int));
```

```
        for(j=0; j<size; ++j)
        {
            ptr[i][j] = (i+1)*j;
        }
    }
    for(i=0; i<num; ++i)
    {
        for(j=0; j<size; ++j)
        {
            printf("[%d, %d] -> %d\t", i, j, ptr[i][j]);

        }
        printf("\n");
    }

    return 0;
}
```

运行结果：