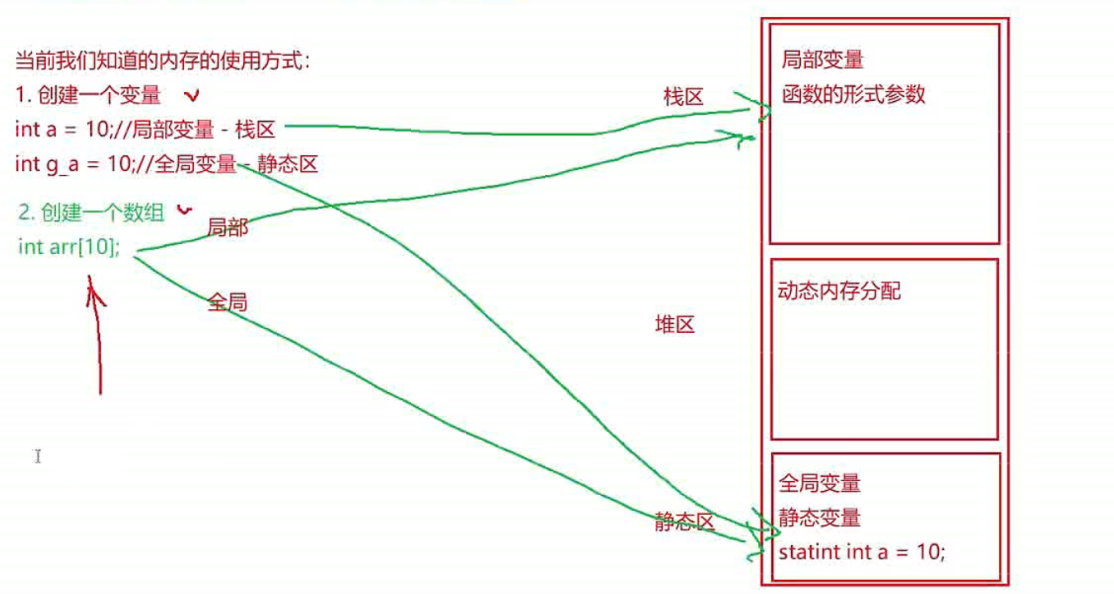
动态内存管理

笔记本:	My Notebook		
创建时间:	2023/10/29 22:01	**更新时间:**	2023/11/2 19:50
作者:	lwb		
URL:	mk:@MSITStore:D:\bo\c语言学习\MSDN\vccore.chm::/html/_crt_malloc.htm		

1.当前我们知道的内存的使用方式：
1.创建一个变量
int a = 10; // 局部变量 - 栈区
int g_a = 10；//全局变量 - 静态区
2.创建一个数组



上面的开辟空间的方式有两个特点
1.空间开辟大小是固定的
2.数组在申明的时候，必须指定数组的长度，它所需要的内存在编译时分配。
但是对于空间的需求，不仅仅是上述的情况。有时候我们需要的空间大小在程序运行的时候才能知道
那数组编译时开辟空间的方式就不能够满足了，所以需要动态开辟内存

2.动态内存函数的介绍
2.1 malloc和free
void *malloc(size_t *size*);
这个函数向内存申请一块**连续可用**的空间，并返回指向这块空间的指针。

- 如果开辟成功，则返回一个指向开辟好空间的指针。
- 如果开辟失败，则返回一个NULL指针，因此malloc的返回值一定要做检查。
- 返回值的类型是 `void*` ，所以malloc函数并不知道开辟空间的类型，具体在使用的时候使用者自己
- 来决定。
- 如果参数 `size` 为0，malloc的行为是标准是未定义的，取决于编译器。

void free(void *_memblock_);
free函数用来释放动态开辟的内存。

- 如果参数 ptr 指向的空间不是动态开辟的，那free函数的行为是未定义的。
- 如果参数 ptr 是NULL指针，则函数什么事都不做。

```
#include <stdio.h>
int main()
{
//代码1
int num = 0;
scanf("%d", &num);
int arr[num] = {0};
//代码2
int* ptr = NULL;
ptr = (int*)malloc(num*sizeof(int));
if(NULL != ptr)//判断ptr指针是否为空
{
int i = 0;
for(i=0; i<num; i++)
{
*(ptr+i) = 0;
}
}
free(ptr);//释放ptr所指向的动态内存后
ptr = NULL;//是否有必要？free空间后ptr指向的地址还是原值，所以还是需要将其指向的空间指向0，即赋值为空指针
return 0;
}
```

2.2 calloc
void *calloc(size_t _num_, size_t _size_);

- 函数的功能是为 num 个大小为 size 的元素开辟一块空间，并且把空间的每个字节初始化为0。
- 与函数 malloc 的区别只在于 calloc 会在返回地址之前把申请的空间的每个字节初始化为全0。

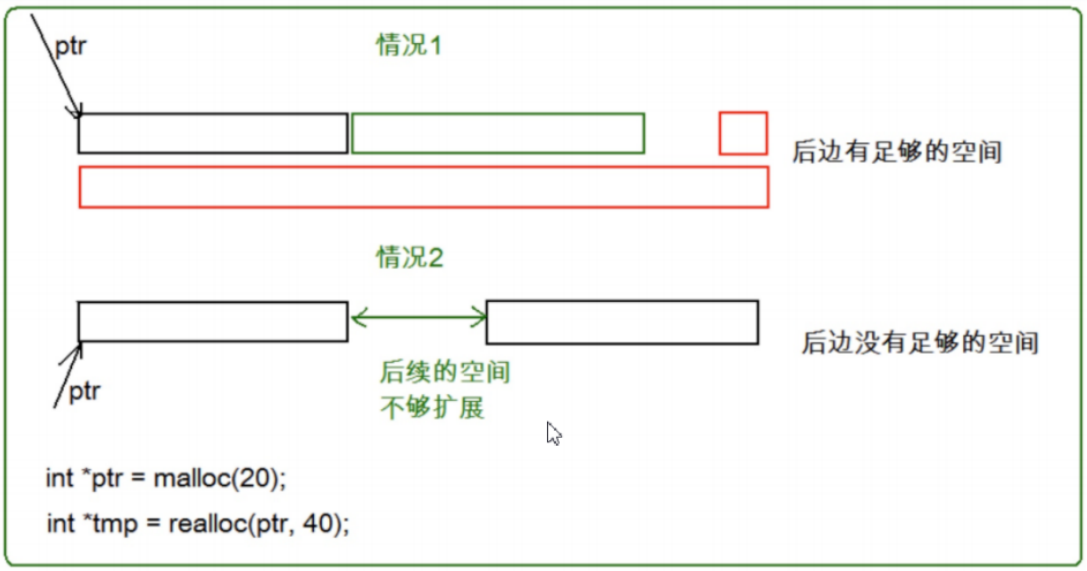
如果我们对申请的内存空间的内容要求初始化，那么可以使用calloc函数来完成任务

2.3 realloc
void *realloc(void *_memblock_, size_t _size_);

- realloc函数的出现让动态内存管理更加灵活。
- 有时会我们发现过去申请的空间太小了，有时候我们又会觉得申请的空间过大了，那为了合理的时候内存，我们一定会对内存的大小做灵活的调整。那 realloc 函数就可以做到对动态开辟内存大小的调整。
- *memblock*是要调整的内存地址
- size 调整之后新大小
- 返回值为调整之后的内存起始位置。
- 这个函数调整原内存空间大小的基础上，还会将原来内存中的数据移动到 新 的空间。

realloc在调整内存空间的时候存在两种情况：

- 1.原有空间之后有足够大的空间，直接追加空间，原来空间的数据不发生变化
- 2.原有空间之后没有足够大的空间，扩展的方法是：在堆空间上寻找另一个合适大小的连续空间来使用，这样函数返回的是一个新的内存地址。



3.常见的动态内存错误

3.1对NULL指针的解引用操作

```
void test()
{
    int *p = (int *)malloc(INT_MAX/4); // malloc开辟空间可能会失败，需要先判断
    *p = 20;//如果p的值是NULL，就会有问题
    free(p);
}
```

3.2对动态开辟空间的越界访问

```
void test()
{
    int i = 0;
    int *p = (int *)malloc(10*sizeof(int));
    if(NULL == p)
    {
        exit(EXIT_FAILURE);
    }
    for(i=0; i<=10; i++)
    {
        *(p+i) = i;//当i是10的时候越界访问
    }
    free(p);
}
```

3.3对非动态开辟内存使用free释放

```
void test()
{
    int a = 10;
    int *p = &a;
    free(p);//ok?no
}
```

3.4使用free释放一块动态开辟内存的一部分

```
void test()
{
    int *p = (int *)malloc(100);
    p++;
    free(p);//p不再指向动态内存的起始位置
}
```

3.5对同一块动态内存多次释放

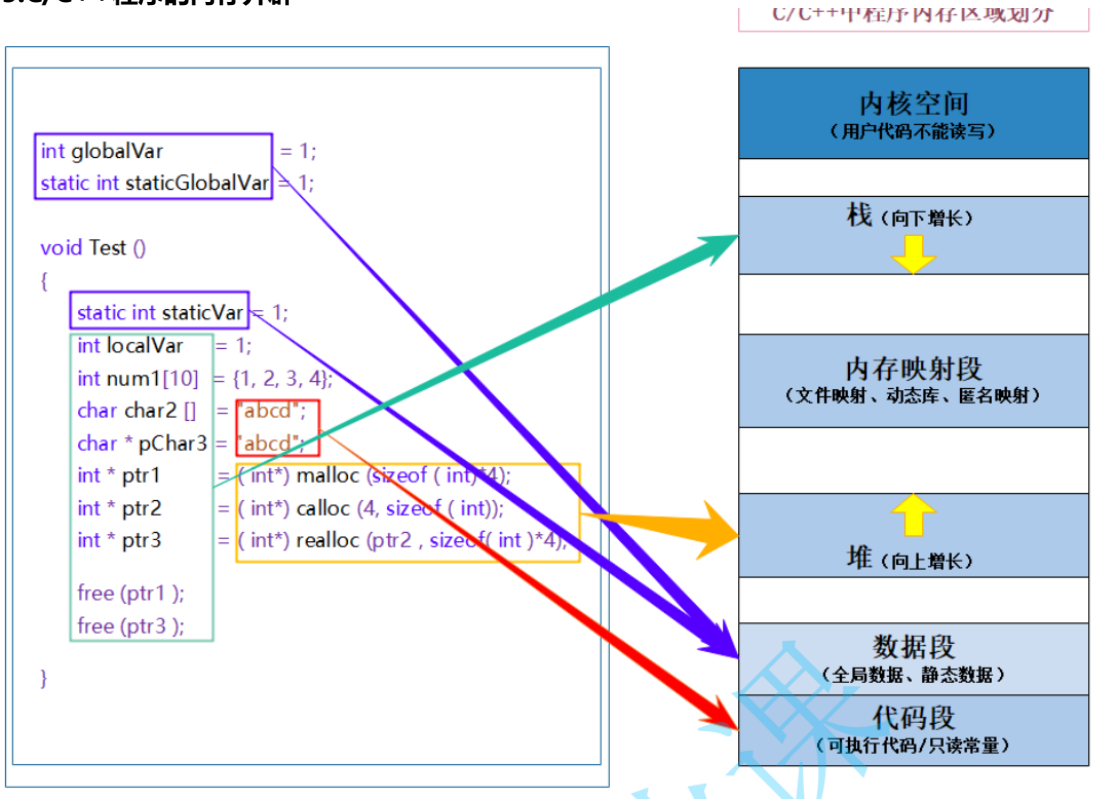
```
void test()
{
    int *p = (int *)malloc(100);
    free(p);
    free(p);//重复释放
}
```

3.6动态开辟内存忘记释放(内存泄露)

```
void test()
{
    int *p = (int *)malloc(100);
    if(NULL != p)
    {
        *p = 20;
    }
}
int main()
{
    test();
    while(1);
}
```

忘记释放不再使用的动态开辟的空间会造成内存泄露
切记：动态开辟的空间一定要释放，并且正确释放

5.C/C++程序的内存开辟



C/C++程序内存分配的几个区域：

- 栈区：在执行函数时，函数内局部变量的存储单元都可以在栈区创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。栈区主要存放运行函数而分配的局部变量、函数参数、返回数据、返回地址等。
- 堆区：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收，分配方式类似于链表。

- 数据段(静态区)（static）存放全局变量，静态数据。程序结束后由操作系统释放
- 代码段：存放函数体（类成员函数和全局函数）的二进制代码

普通的局部变量是在栈区分配空间的，栈区的特点是在上面创建的变量出了作用域就销毁。
但是被static修饰的变量存放在数据段（静态区），数据段的特点是在上面创建的变量，直到程序结束后才销毁，所以生命周期边长。

柔性数组
C99中，结构中的最后一个元素允许是未知大小的数组，这就叫做柔性数组成员

```
typedef struct st_type
{
    int i;
    int a[0];//柔性数组成员
}type_a;
```

特点：

- 结构中的柔性数组前面必须至少有一个其他成员
- sizeof返回的这种结构大小不包含柔性数组的内存
- 包含柔性数组成员的结构用malloc()函数进行内存的动态分配，并且分配的内存应该大于结构的大小，以适应柔性数组的预期

柔性数组的使用

```
//代码1
int i = 0;
type_a *p = (type_a*)malloc(sizeof(type_a)+100*sizeof(int));
//业务处理
p->i = 100;
for(i=0; i<100; i++)
{
    p->a[i] = i;
}
free(p);
```

上述的type_a结构也可以设计为：

```
//代码2
typedef struct st_type
{
    int i;
    int *p_a;
}type_a;
type_a *p = (type_a *)malloc(sizeof(type_a));
p->i = 100;
p->p_a = (int *)malloc(p->i*sizeof(int));
//业务处理
for(i=0; i<100; i++)
{
    p->p_a[i] = i;
}
//释放空间
free(p->p_a);
p->p_a = NULL;
free(p);
p = NULL;
```

上述代码1和代码2可以完成同样的功能，但是方法1的实现有两个好处
第一个好处：方便内存释放
如果我们的代码是在一个给别人用的函数中，你在里面做了二次内存分配，并把整个结构体返回给
用户。用户调用free可以释放结构体，但是用户并不知道这个结构体内的成员也需要free，所以你不能指望用户来发现这个事。所以，如果我们把结构体的内存以及其成员要的内存一次性分配好了，并返回给用户一个结构体指针，用户做一次free就可以把所有的内存也给释放掉。
第二个好处：这样有利于访问速度

连续的内存有益于提高访问数据，也有利于减少内存碎片