动态内存分配

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
int arr[10]; // 存放方式同上
	

为什么存在动态内存分配

我们已经掌握的内存开辟方式有：

```
int val = 20 ; //在栈空间开辟四个字节
```

```
char arr[10] = {0}; // 在栈空间开辟10个直接的连续空间
```

但是上述的开辟空间有两个特点：

1. 开间开辟大小是固定的。
2. 数组在声明的时候必须**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，有时候我们需要的空间大小只有在程序运行的时候才能知道，这时数组在编译时开辟空间的方式就不能满足此时的需求，这时需要动态开辟内存。

动态内存函数的介绍

`malloc` 以及 `free` 函数 (有借有还)

`malloc` ----申请空间地址

所需头文件 `stdlib.h` 以及 `malloc.h`

函数原型

`void * malloc( size_t size);`

例如

```
int main(void)
{
    int *p = (int *)malloc(10*sizeof(int));
    return 0;
}
```

注：内存开辟时，如遇内存空间不足等情况，`malloc`函数将会返回`NULL`型指针。

注：如果参数`size`为`0`，`malloc`的行为标准未定义的，取决于编译器。

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <errno.h>
int main(void)
{
    int *p = (int *)malloc(1e100*sizeof(int));
    if (p == NULL)
    {
        printf("%s\n",strerror(errno));
    }
    return 0;
}
```

free ——回收释放空间

`free` ---- 释放地址

所需头文件 `stdlib.h` 以及 `malloc.h`

函数原型

`void * free( void *memblock);`

示例:

```
int main(void)
{
    int *p = (int *)malloc(10*sizeof(int));
    free(p);
    p = NULL;   // 失去定位
    return 0;
}
```

注：当程序生命周期结束后，所申请的动态内存也会释放，此时的指针仍可以定位到这一段内存。

`calloc`函数

`calloc`函数会将所开辟的内存空间初始化为`0`。

所需头文件 `stdlib.h` 以及 `malloc.h`

函数原型

`void * calloc( size_t num, size_t size);`

```
int main(void)
{
    int* p = (int*)calloc(10,sizeof(int));
    if (p == NULL)
    {
        printf("%s\n",strerror(errno));
    }
    else
    {
        int i = 0;
        for (i = 0;i < 10; i++)
        {
            printf("%d ", *(p+i));
        }
    }
    free(p);
    p = NULL;
    return 0;
}
```

`realloc`函数

- `realloc`函数的出现让动态内存管理更加灵活
- 有时我们发现曾经申请的空间太小了，有时又太大了，为了合理管理内存，我们需要对内存的大小进行灵活的调整。那`realloc`函数就可以调整动态开辟的内存大小。

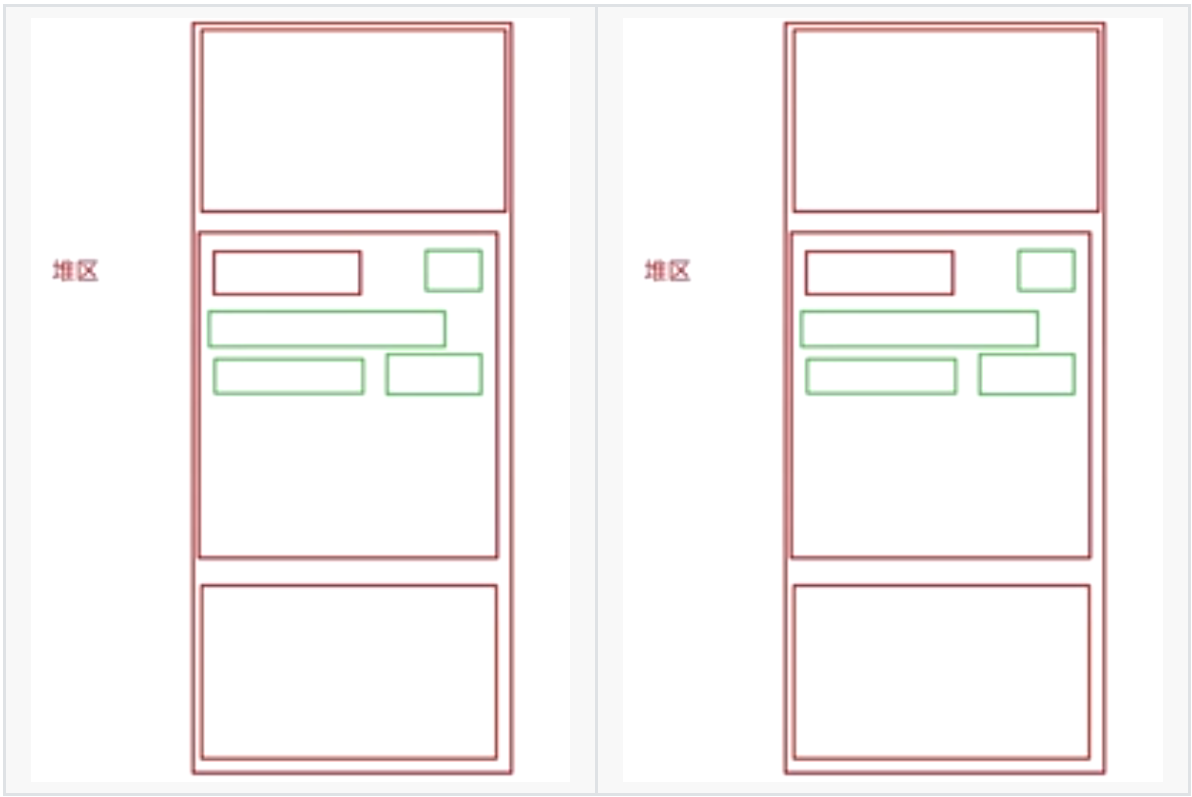
所需头文件 `stdlib.h` 以及 `malloc.h`

函数原型

> `void * realloc( void * ptr, size_t size);`
>
> `ptr` 是要调整的内存地址
>
> `size` 为调整之后的内存大小

> 返回值为调整之后内存的初始位置
>
> 函数在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`在调整内存空间时有如下两种情况
>
> - 情况1：原内存空间之后仍有足够大的空间，则直接追加内存，后返回地址`p`
> - 情况2：原内存空间之后没有足够大的空间，将重新开辟一块空间，将旧内存空间拷贝并释放，后返回新的地址`p`
> - 情况3：追加空间过大，无法继续开辟内存时，NULL指针覆盖指针`p`，原内存中的数据会丢失。

注：所以要用一个新的指针`ptr`接收新开辟内存。



常见的内存错误

1. NULL指针的解引用操作

当`malloc`失败了，`p`就被赋值为`NULL`，此时对指针解引用就会出现错误。

```
int main(void)
{
    int *p = (int *)malloc(40);
    // 万一malloc失败了，p就被赋值为NULL
    int i = 0;
    for (i = 0;i < 5; i++)
    {
        *(p+i) = i;     // *p = 0;  //err
    }

    return 0;
}
```

2. 对动态开辟内存的越界访问

访问动态内存之外的部分时，会出现错误。

```
int *p = malloc(5*sizeof(int));
if (p == NULL)
{
    return 0;
}
else
{
    int i = 0;
    for (i;i<10;i++)
    {
        *(p + i) = i;
        printf("%d ",*(p + i));
    }
}
free(p);
p = NULL;
return 0;
```

3. 对非动态开辟内存进行释放

对非堆区变量指针的释放。

```
int a = 10;
int *p = &a;
free(p);
p = NULL;
```

4. 使用`free`释放动态内存中的一部分

```
int *p = (int *)malloc(10*sizeof(int))
if (p == NULL)
{
    return 0;
}
int i = 0;
for (i = 0; i < 10; i++)
{
    *p++ = i;
}
free(p);
p = NULL;
```

5. 使用`free`释放动态内存多次

- 谁申请，谁回收
- `p = NULL`

```
int *p = (int *)malloc(10*sizeof(int))
if (p == NULL)
{
    return 0;
}
free(p);
....
free(p);
```

6. 动态开辟内存忘记释放(内存泄露)

```
while (1)
{
    malloc(1);
}
```

经典面试题

1

```
void GetMemory(char *p)
{
    p = (char *)malloc(100);
}

void Test(void)
{
    char *str = NULL;
    GetMemory(str);
    strcpy(str,"hello world");
    printf(str);

}

int main()
{
    Test();
    return 0;
}
```

- 运行代码时，程序会崩溃
- 程序存在内存泄露的问题

p是GetMemory函数的形参，只在函数内部有效，当GetMemory函数执行完毕，动态开辟的内存尚未释放并且无法访问，故会造成内存泄露。