一、动态分配内存的概述

在数组一章中，介绍过数组的长度是预先定义好的，在整个程序中**固定不变**，但是在实际的编程中，往往会发生这种情况，即所需的**内存空间取决于实际输入的数据**，而无法预先确定 。为了解决上述问题，C语言提供了一些**内存管理函数**，这些内存管理函数可以按需要**动态的分配**内存空间，也可把不再使用的空间回收再次利用。

动态分配内存就是在堆区开辟空间

二、静态分配、动态分配

静态分配

1、 在程序编译或运行过程中，按事先规定大小分配内存空间的分配方式。int a [10]

2、 必须事先知道所需空间的大小。

3、 分配在栈区或全局变量区，一般以数组的形式。

4、 按计划分配。

动态分配

1、在程序运行过程中，根据需要大小自由分配所需空间。

2、按需分配。

3、分配在堆区，一般使用特定的函数进行分配。

三、动态分配函数

3.1 malloc

```
#include <stdlib.h>
void *malloc(unsigned int size);
功能：在堆区开辟指定长度的空间，并且空间是连续的
参数：
 size：要开辟的空间的大小
返回值：
 成功：开辟好的空间的首地址
 失败：NULL
```

注意

1、在调用malloc之后，一定要判断一下，是否申请内存成功。

2、如果多次malloc申请的内存，第1次和第2次申请的内存不一定是连续的

3、使用malloc开辟空间需要保存开辟好的空间的首地址，但是由于不确定空间用于做什么，所以本身返回值类型为void *，所以在调用函数时根据接收者的类型对其进行强制类

型转换

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

char *fun()
{
 //char ch[100] = "hello world";

 //静态全局区的空间只要开辟好，除非程序结束，否则不会释放，所以
 //如果是临时使用，不建议使用静态全局区的空间
  //static char ch[100] = "hello world";

  //堆区开辟空间，手动申请手动释放，更加灵活
  //使用malloc函数的时候一般要进行强转
  char *str = (char *)malloc(100 * sizeof(char));
  str[0] = 'h';
  str[1] = 'e';
  str[2] = 'l';
  str[3] = 'l';
  str[4] = 'o';
  str[5] = '\0';

  return str;
}

int main(int argc, char *argv[])
{
  char *p;
  p = fun();
  printf("p = %s\n", p);

  return 0;
}
```

执行结果

```
Starting C:\Users
\debug\01_malloc.
p = hello
C:\Users\lzx\Desk
```

3.2 free

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

```
void free(void *ptr)
功能：释放堆区的空间
参数：
 ptr：开辟后使用完毕的堆区的空间的首地址
返回值：
 无
```

注意：

free函数只能释放堆区的空间，其他区域的空间无法使用free

free释放空间必须释放malloc或者calloc或者realloc的返回值对应的空间，不能说只释放一部分

free(p); 注意当free后，因为没有给p赋值，所以p还是指向原先动态申请的内存。但是内存已经不能再用了，p变成野指针了，所以一般为了放置野指针，会free完毕之后对p赋为NULL。

一块动态申请的内存只能free一次，不能多次free

```
//使用free函数释放空间
free(p);
//防止野指针
p = NULL;
```

3.3 calloc

```
#include <stdlib.h>
void * calloc(size_t  nmemb,size_t  size);
功能：在堆区申请指定大小的空间
参数：
 nmemb：要申请的空间的块数
 size：每块的字节数
返回值：
 成功：申请空间的首地址
 失败：NULL
```

注意：

malloc和calloc函数都是用来申请内存的。

区别：

1) 函数的名字不一样

2) 参数的个数不一样

3) malloc申请的内存，内存中存放的内容是随机的，不确定的，而calloc函数申请的内存中的内容为0

例如：

char *p=(char *)calloc(3,100);

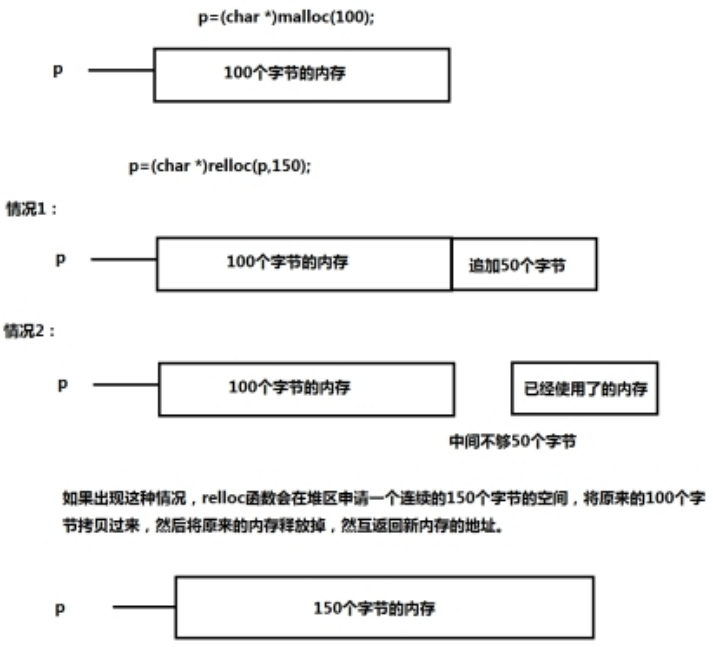
在堆中申请了3块，每块大小为100个字节，即300个字节连续的区域。

3.4 realloc

```
#include <stdlib.h>
void* realloc(void *s,unsigned int newsize);
功能：在原本申请好的堆区空间的基础上重新申请内存，新的空间大小为函数的第二个参数
 如果原本申请好的空间的后面不足以增加指定的大小，系统会重新找一个足够大的位
 置开辟指定的空间，然后将原本空间中的数据拷贝过来，然后释放原本的空间
 如果newsize比原先的内存小，则会释放原先内存的后面的存储空间，
 只留前面的newsize个字节
参数：
 s：原本开辟好的空间的首地址
  newsize：重新开辟的空间的大小
返回值：
 新的空间的首地址
```

增加空间：

```
char *p;
p=(char *)malloc(100)
//想在100个字节后面追加50个字节
p=(char *)realloc(p,150);//p指向的内存的新的大小为150个字节
```



减少空间：

```
char *p;
p=(char *)malloc(100)
//想重新申请内存,新的大小为50个字节
```

```
p=(char *)realloc(p,50);//p指向的内存的新的大小为50个字节,100个字节的后50个字节的存储空间就被释放了
```

注意:malloc calloc relloc 动态申请的内存，只有在free或程序结束的时候才释放。

四、内存泄漏

内存泄露的概念：

申请的内存，首地址丢了，找不了，再也没法使用了，也没法释放了，这块内存就被泄露了。

内存泄漏案例1：

```
int main()
{
 char *p;
 p=(char *)malloc(100);
 //接下来，可以用p指向的内存了

 p="hello world";//p指向别的地方了，保存字符串常量的首地址

 //从此以后，再也找不到你申请的100个字节了。则动态申请的100个字节就被泄露了

  return 0;
}
```

内存泄漏案例2：

```
void fun()
{
 char *p;
 p=(char *)malloc(100);
 //接下来，可以用p指向的内存了
 ...
}

int main()
{
  //每调用一次fun泄露100个字节
  fun();
  fun();
```

```
 return 0;
}
```

解决方式1：

```
void fun()
{
 char *p;
 p=(char *)malloc(100);
 //接下来，可以用p指向的内存了
 ...
 free(p);
}

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

解决方式2：

```
char * fun()
{
 char *p;
 p=(char *)malloc(100);
 //接下来，可以用p指向的内存了
 ...
 return p;
}

int main()
{
 char *q;
 q=fun();
 //可以通过q使用 ， 动态申请的100个字节的内存了
 //记得释放
 free(q);
 //防止野指针
 q = NULL;
```

```
19
20   return 0;
21  }
```

总结：申请的内存，一定不要把首地址给丢了，在不用的时候一定要释放内存。