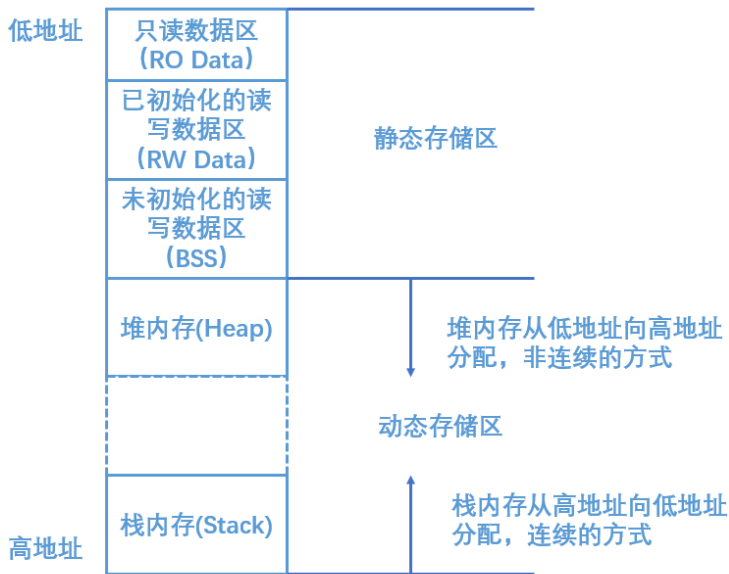


C 语言程序需要载入内存才可以运行，其不同的数据保存在不同的区域。所使用的内存可以分成两类：一类是**静态存储区**，另一类是**动态存储区**。C 语言程序的存储区如下图所示：



# 1 静态存储区

静态存储区分为三类：**只读数据区 (RO Data)**、**已初始化读写数据区 (RW Data)**、**未初始化读写数据区 (BSS)**。这三类存储区都是在程序的编译-连接阶段确定的，且运行过程中是不会变化的，只有当程序退出的时候，静态存储区的内存才会被系统回收。

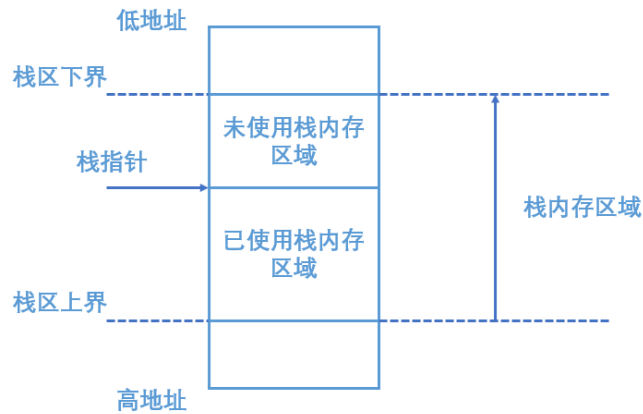
# 2 动态存储区

动态存储区主要分为两类：一类是**栈 (Stack) 内存区域**，栈内存是由编译器管理的；另一类是**堆 (Heap) 内存区域**，堆内存由程序调用具体的库函数来分配的。它们都是程序运行过程中动态分配的。

## 2.1 栈内存区域

### 2.1.1 栈的相关概念

栈内存的使用很大的程度上依赖于处理器的硬件机制。在处理器中，有一个寄存器来表示当前栈指针的位置。通常在内存中分配一块区域，这块区域的上界（高内存地址）和下界（低内存地址）之间是可用的栈内存区域。栈内存如下图所示：



目前常见的体系结构和编译系统中，**栈大多都是向下增长的**。在初始阶段，栈指针是指向栈区间的上界，随着栈使用量的增加，栈指针的值向低地址移动，即栈指针的值将变小。下面来看一段程序：

```

F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题\stack.c - Notepad++
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?
stack.c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int a = 1, b = 2, c = 3;
6
7      printf("a = %d, &a = %#x \n", a, (unsigned int)&a);
8      printf("b = %d, &b = %#x \n", b, (unsigned int)&b);
9      printf("c = %d, &c = %#x \n", c, (unsigned int)&c);
10
11     return 0;
12 }

```

程序运行结果为：

```

C:\WINDOWS\system32\cmd.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>stack.exe
a = 1, &a = 0x61ff0c
b = 2, &b = 0x61ff08
c = 3, &c = 0x61ff04
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>

```

可见，变量的存储是从高地址往低地址的方向存储。

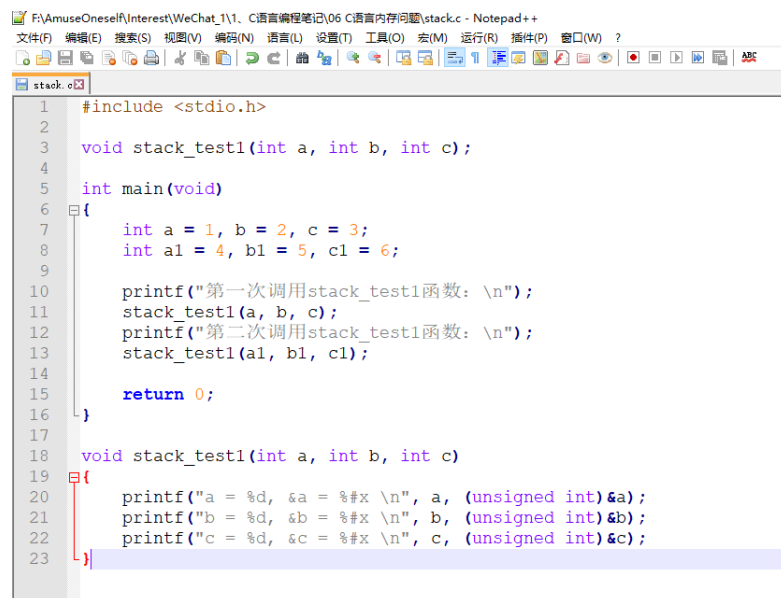
栈有一个重要的特性：先放入的数据最后才能取出，后放入的数据优先能取出，即**先进**

后出（First In Last Out）原则。放入数据常被称为入栈或压栈（Push），取出数据被称为出栈或弹出（Pop）。在运用过程中，栈内存可能出现满栈和空栈两种情况，这是由处理器的体系结构决定的。

栈（Stack）可以存放函数参数、局部变量、局部数组等作用范围在函数内部的数据，它的用途就是完成函数的调用。

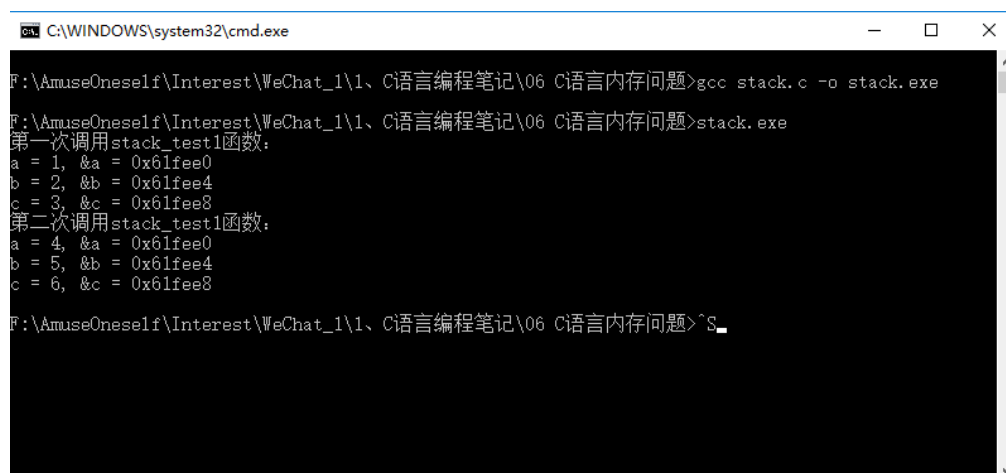
### 2.1.1 需要知道的关于栈的问题：

（1）函数在调用完成之后，栈指针将回到函数进入之前的位置。下面的程序通过两次调用同一个函数印证了这一点：



```
1 #include <stdio.h>
2
3 void stack_test1(int a, int b, int c);
4
5 int main(void)
6 {
7     int a = 1, b = 2, c = 3;
8     int a1 = 4, b1 = 5, c1 = 6;
9
10    printf("第一次调用stack_test1函数: \n");
11    stack_test1(a, b, c);
12    printf("第二次调用stack_test1函数: \n");
13    stack_test1(a1, b1, c1);
14
15    return 0;
16 }
17
18 void stack_test1(int a, int b, int c)
19 {
20     printf("a = %d, &a = %p \n", a, (unsigned int)&a);
21     printf("b = %d, &b = %p \n", b, (unsigned int)&b);
22     printf("c = %d, &c = %p \n", c, (unsigned int)&c);
23 }
```

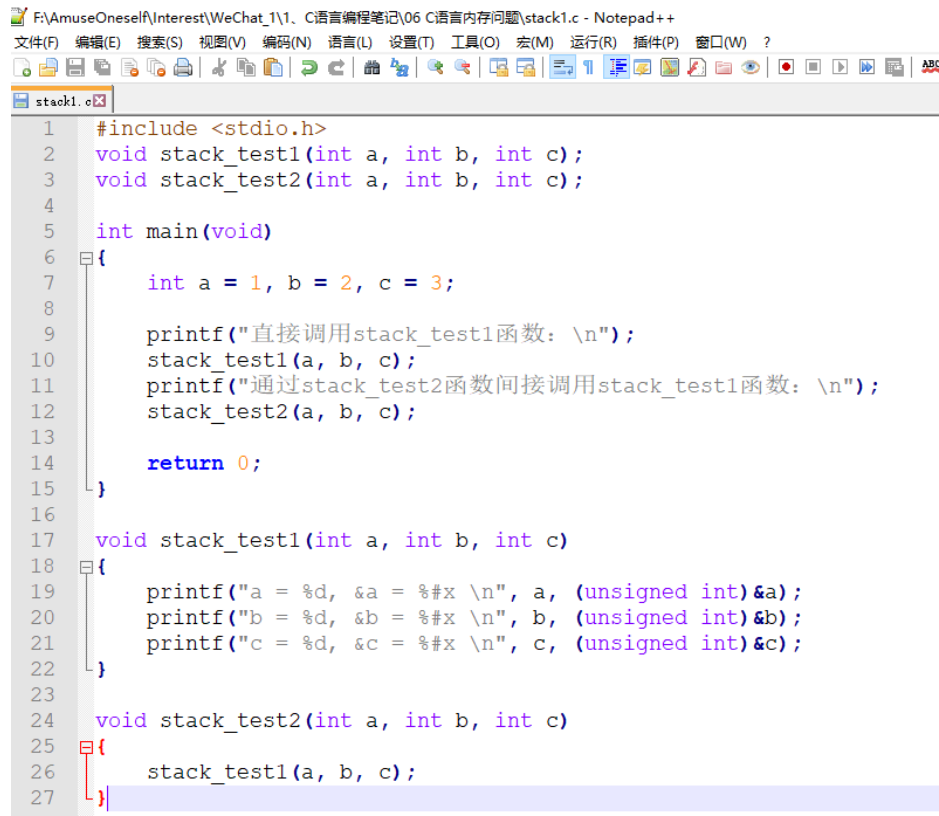
程序运行结果：



```
C:\WINDOWS\system32\cmd.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>gcc stack.c -o stack.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>stack.exe
第一次调用stack_test1函数:
a = 1, &a = 0x61fee0
b = 2, &b = 0x61fee4
c = 3, &c = 0x61fee8
第二次调用stack_test1函数:
a = 4, &a = 0x61fee0
b = 5, &b = 0x61fee4
c = 6, &c = 0x61fee8
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>^S_
```

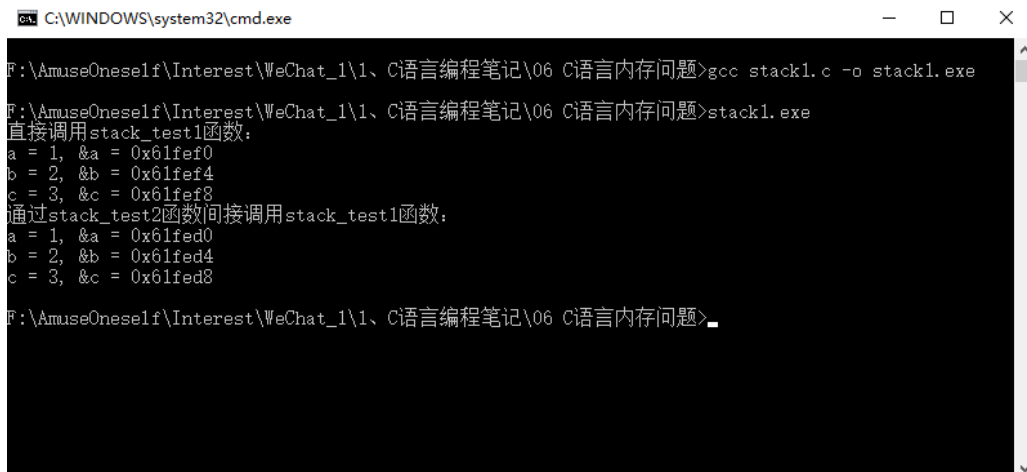
可见，两次调用中函数参数使用的栈内存是相同的，即第一次调用函数完成之后，栈指针将回到函数进入之前的位置。

(2) 在函数调用的过程中，每增加一个层次，栈空间就会被压入更多的内容，下面的程序验证了这一点：



```
1 #include <stdio.h>
2 void stack_test1(int a, int b, int c);
3 void stack_test2(int a, int b, int c);
4
5 int main(void)
6 {
7     int a = 1, b = 2, c = 3;
8
9     printf("直接调用stack_test1函数: \n");
10    stack_test1(a, b, c);
11    printf("通过stack_test2函数间接调用stack_test1函数: \n");
12    stack_test2(a, b, c);
13
14    return 0;
15 }
16
17 void stack_test1(int a, int b, int c)
18 {
19     printf("a = %d, &a = %#x \n", a, (unsigned int)&a);
20     printf("b = %d, &b = %#x \n", b, (unsigned int)&b);
21     printf("c = %d, &c = %#x \n", c, (unsigned int)&c);
22 }
23
24 void stack_test2(int a, int b, int c)
25 {
26     stack_test1(a, b, c);
27 }
```

程序运行结果：



```
C:\WINDOWS\system32\cmd.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>gcc stack1.c -o stack1.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>stack1.exe
直接调用stack_test1函数:
a = 1, &a = 0x61fef0
b = 2, &b = 0x61fef4
c = 3, &c = 0x61fef8
通过stack_test2函数间接调用stack_test1函数:
a = 1, &a = 0x61fed0
b = 2, &b = 0x61fed4
c = 3, &c = 0x61fed8
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>
```

可见, 在程序中两次调用 `stack_test1` 函数, 第一次是直接调用, 第二次是通过 `stack_test2` 函数间接调用。从运行结果来看, 通过 `stack_test2` 函数间接调用 `stack_test1` 函数的栈指针的值变小了, 说明是由于栈中压入了更多的内容。

(3) 函数调用结束后, 函数栈上的内容不能被其他函数使用。例如, 下面是一种错误的用法:

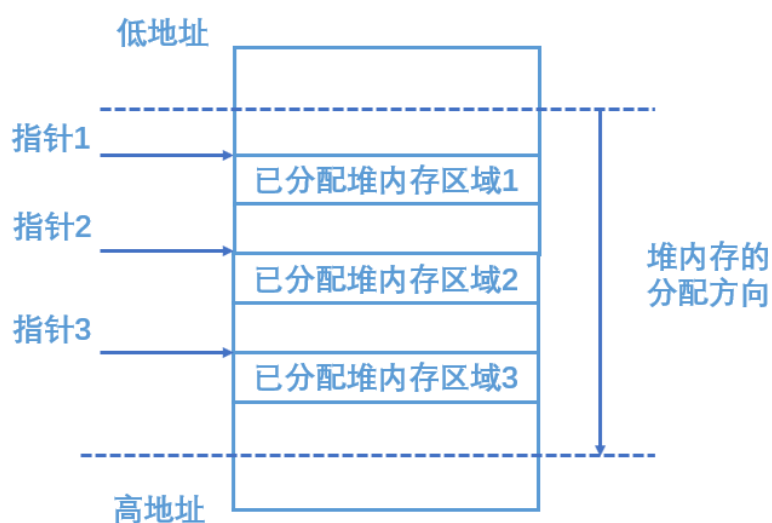
```
int *stack_test3(void)
{
    int a;
    /* ..... */
    return (&a);
}
```

`return(&a)` 将自动变量 `a` 的值返回, 这种写法不会发生编译错误 (又可能出现警告), 但是其逻辑是不正确的。此时, 调用者可以得到 `stack_test3` 运行时 `a` 的地址, 但是由于变量 `a` 是建立在栈上, 函数退出后, 栈区域已经释放, 这个地址已经指向无效的内存, 因此不应该再被程序使用。

## 2.2 堆内存区域

### 2.2.1 堆的相关概念

在一般的编译系统中, 堆内存的分配方向和栈内存是相反的。栈内存利用的是处理器的硬件机制, 而堆内存的处理使用的是库函数。堆内存的分配形式如下图:



可见，堆内存与栈内存的区别：栈内存只有一个入口点，就是栈指针，栈内存压栈和出栈都只能通过栈指针及其偏移量；而堆内存有多个入口点，每次分配得到的指针就是访问内存的入口，每个分配内存区域都可以被单独释放。

当频繁的分配和释放内存的过程中，将会出现如下情况：在两块已经分配的内存之间可能出现较小的未分配的内存区域，这些内存理论上可以被使用。但是由于它们的空间较小，不够连续内存的分配，因此当分配内存的时候，它们经常不能被使用。这种较小的内存就是内存碎片。

## 2.2.2 关于堆空间的使用及其一些问题：

### (1) 库文件：stdlib.h

实现堆内存分配和释放的 4 个主要函数为：

```
/* 分配内存空间 */
void *malloc(size_t size);
/* 释放内存空间 */
void free(void *ptr);
/* 分配内存空间 */
void *calloc(size_t nmemb, size_t size);
/* 重新分配内存空间 */
void *realloc(void *ptr, size_t size);
```

## (2) malloc 和 free 的简单应用

```
//malloc和free的简单应用
void heap_test1(void)
{
    int *pa;

    pa = (int*)malloc(sizeof(int));
    if ( NULL != pa )
    {
        *pa = 0x1234;
        printf("pa = %#x, *pa = %x\n", (unsigned int)pa, *pa);
        free(pa);
    }

    return;
}
```

在 malloc 分配完内存之后，可以用得到的指针值是否为 NULL 来判断内存是否分配成功。按照 C 语言内存分配规则，如果内存分配成功，返回的是内存的地址；如果内存分配不成功，将返回 NULL (0x0)，表示一个无效的地址。

(3) malloc 在分配内存的时候，是从低地址至高地址方向。但是，先分配的内存地址不一定比后分配的内存地址小。下面的程序验证了这一点：

```
//后分配内存地址反而更小
void heap_test2(void)
{
    void *pa;
    void *pb;
    void *pc;
    void *pd;
    pa = (int*)malloc(1024);
    printf("pa = %#x \n", (unsigned int)pa);
    pb = (int*)malloc(1024);
    printf("pb = %#x \n", (unsigned int)pb);
    pc = (int*)malloc(1024);
    printf("pc = %#x \n", (unsigned int)pc);
    free(pb);
    pd = (int*)malloc(1024);
    printf("pd = %#x \n", (unsigned int)pd);

    free(pa);
    free(pc);
    free(pd);

    return;
}
```

程序运行结果：

```
C:\WINDOWS\system32\cmd.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>gcc heap.c -o heap.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>heap.exe
pa = 0x652178
pb = 0x653588
pc = 0x653990
pd = 0x653588
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>
```

可见，在该程序中，首先 3 次分配 1024 字节的堆上内存，然后再将第二次分配的内存释放，再次分配内存时，将利用了这一块空间。

(4) calloc()和 malloc()很类似，主要区别是 calloc()可以将分配好的内存区域的初始值全部设置为 0，以下程序验证了这一点：

```
//calloc和malloc的主要区别
void heap_test3(void)
{
    unsigned int *pa;
    int i;

    pa = (unsigned int*)calloc(sizeof(unsigned int), 5);
    if ( NULL != pa )
    {
        printf("<< calloc pa = %#x >>\n", (unsigned int)pa);
        for ( i = 0; i < 5; i++ )
        {
            printf("pa[%d] = %d \n", i, pa[i]);
        }
        free(pa);
    }

    return;
}
```

程序运行结果：



```
选择C:\WINDOWS\system32\cmd.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>gcc heap.c -o heap.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>heap.exe
<< calloc pa = 0x1f2178 >>
pa[0] = 0
pa[1] = 0
pa[2] = 0
pa[3] = 0
pa[4] = 0
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>
```

除此之外, calloc()和 malloc()另外一个不同之处在于参数的个数, malloc 只有一个参数, 即要分配的内存字节数; calloc 有两个参数, 第一个是分配单元的大小, 第二个是要分配的数目。从本质上, calloc 使用两个参数和 malloc 使用一个并没有区别。

(5) realloc 的应用。realloc 函数具有两个参数, 一个是指向内存的地址指针, 另一个是重新分配内存的大小, 而返回值是指向所分配内存的指针。基本应用代码如下:

```
void heap_test4(void)
{
    int *pa;
    int i;

    pa = (int*)malloc(sizeof(int)*6);
    if ( NULL != pa ){
        for ( i = 0; i < 6; i++ ){
            *(pa + i) = i;
        }
        for ( i = 0; i < 6; i++ ){
            printf("pa[%d] = %d \n", i, pa[i]);
        }
    }
    printf("realloc重新分配内存\n");
    pa = (int*)realloc(pa, sizeof(int)*10);
    if ( NULL != pa ){
        for ( i = 0; i < 10; i++ ){
            printf("pa[%d] = %d\n", i, pa[i]);
        }
        free(pa);
    }

    return;
}
```

程序运行结果：

```
C:\WINDOWS\system32\cmd.exe
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>heap.exe
pa[0] = 0
pa[1] = 1
pa[2] = 2
pa[3] = 3
pa[4] = 4
pa[5] = 5
realloc重新分配内存
pa[0] = 0
pa[1] = 1
pa[2] = 2
pa[3] = 3
pa[4] = 4
pa[5] = 5
pa[6] = 0
pa[7] = 0
pa[8] = 0
pa[9] = 0
F:\AmuseOneself\Interest\WeChat_1\1、C语言编程笔记\06 C语言内存问题>
```

除此之外，realloc 还具有两种功能：一是当指针为 NULL 的时候，作为 malloc 使用，分配内存；二是当重新分配内存大小为 0 的时候，作为 free 使用，释放内存。

(6) 再堆内存的管理上，容易出现以下几个问题：

- 开辟的内存没有释放，造成内存泄漏  
内存泄漏的例子：

```
//内存泄漏例子
void heap_test6(void)
{
    char *pa;
    pa = (char*)malloc(sizeof(char)*20);
    /* ..... */

    return;
}
```

在函数 heap\_test6 中，使用 malloc 开辟了 20 个字节的内存区域，但是使用结束后该函数没有释放这块区域，也没有通过任何返回值或者参数的手段将这块内存区域的地址告诉其它函数。此时，这 20 个字节的内存不会被任何程序释放，因此再调用该函数的时候，就会导致内存泄漏。

- 野指针被使用或者释放

野指针是一个已经被释放的内存指针，它指向的位置已经被 free 或者 realloc 释放了，此时再使用该指针，就会导致程序的错误。野指针例子：

```

//野指针例子
void heap_test6(void)
{
    char *pa;
    pa = (char*)malloc(sizeof(char)*20);
    /* ..... */
    free(pa);
    /* ..... */
    printf("pa = %s \n",pa); //野指针被使用

    return;
}

```

在此程序中，调用 free 函数已经释放了 pa 指针，但后面还在继续使用 pa，这就是一个错误的程序。

- 非法释放指针

1) 非法释放静态存储区的内存，示例如下：

```

const char ro_data[] = "ro_data";
char rw_data[] = "rw_data";
char bss_data[100];

void heap_test7(void)
{
    /* ..... */
    /* 错误释放只读数据区指针 */
    free(ro_data);
    /* 错误释放已初始化读写数据区指针 */
    free(rw_data);
    /* 错误释放未初始化读写数据区指针 */
    free(bss_data);

    /* 错误释放代码区指针 */
    free(heap_test7);
    /* ..... */
    return;
}

```

2) 非法释放栈上的内存，示例如下：

```

void heap_test8(void)
{
    char a[20];
    int b;

    /* 错误释放栈上内存 */

    /* ..... */
    free(a);
    free(&b);
    /* ..... */
    return;
}

```

### 3) 非法释放堆上内存，示例如下：

```
void heap_test9(void)
{
    char *pa;
    /* ..... */
    pa = (char*)malloc(sizeof(char)*20);
    free(pa);
    free(pa); //错误释放堆内存
    /* ..... */
    return;
}
```

第一次释放之后，该地址已经变成了未被分配的堆上的内存了，free 函数不能释放未分配的堆内存。

```
void heap_test10(void)
{
    char *pa;
    char *pb;
    /* ..... */
    pa = (char*)malloc(sizeof(char)*20);
    pb = pa++;
    free(pb); //错误释放堆内存
    /* ..... */
    return;
}
```

释放内存 pb 是非法的内存释放，由于这个指针并不是从 malloc 分配出来的，而是一个中间的指针值。

以上是对于 C 语言内存的一些总结笔记，如有错误的欢迎指出，谢谢！想阅读更多分享欢迎扫描以下二维码关注我的微信公众号，也可搜索 zhengnian-2018 关注。

