动态内存开辟

1 动态内存函数介绍

1.1 malloc()和free()函数

**void** **\*malloc(** **size\_t** *size* **);//传入需要开辟的空间字节数，返回空间的地址（指针void \*）或者返回NULL（空间不够）。**

#include<stdio.h>

#include<stdlib.h>

#include <malloc.h>

int main()

{

//动态内存开辟

//1.申请内存

int\* p = (int\*)malloc(40);

//判断是否开辟成功，失败返回NULL

if (p == NULL)

{

perror("main");

return 0;

}

//2.使用内存

int i = 0;

for (i = 0; i < 10; i++)

{

\*(p + i) = i;

}

for (i = 0; i < 10; i++)

{

printf("%d ", \*(p + i));

}

printf("\n");

//3.释放内存 置空指针

free(p);

p = NULL;

return 0;

}

Void free(void \* ptr);//内存释放函数，但只能是堆的动态内存空间进行释放，不能释放栈区的内存。



1.2 calloc()函数

**void** **\*calloc(** **size\_t** *num***,** **size\_t** *size* **);//开辟空间，两个参数，元素个数，元素大小，并默认将其中的元素初始化为0（特点），返回空间地址或NULL指针。**

#include<stdio.h>

#include<stdlib.h>

#include <malloc.h>

int main()

{

//1.申请内存

//int\* p = (int\*)malloc(40);

int\* p = (int\*)calloc(10, 4);

int i = 0;

//2.是否开辟成功

if (p == NULL)

{

perror("main");

return 0;

}

//3.内存使用

for (i = 0; i < 10; i++)

{

printf("%d\n", \*(p + i));

}

//4.内存释放及置零

free(p);

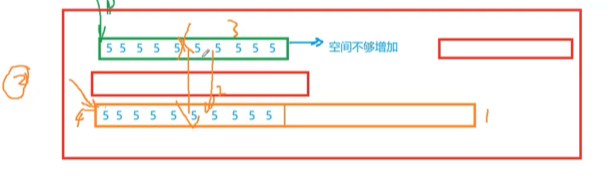
p = NULL;

return 0;

}

1.3 realloc()函数

**void** **\*realloc(** **void** **\****memblock***,** **size\_t** *size* **);//重新开辟空间函数，参数：***memblock***表示先前内存的地址，***size***表示开辟空间的字节大小（总共的大小），返回开辟空间的地址由（分情况）新指针接收。它使得内存管理更加灵活。**



#include<stdio.h>

#include<stdlib.h>

#include <malloc.h>

int main()

{

//1.申请内存

//int\* p = (int\*)malloc(40);

int\* p = (int\*)calloc(10, 4);

int i = 0;

//2.是否开辟成功

if (p == NULL)

{

perror("main");

return 0;

}

//需要再多开辟10个整型，则总共的字节数为20\*sizeof(int)

int\* q = (int\*)realloc(p, 20 \* sizeof(int));

if (q == NULL)

{

perror("main");

return 0;

}

p = q;

//3.内存使用

for (i = 0; i < 20; i++)

{

printf("%d\n", \*(p + i));

}

//4.内存释放及置零

free(p);

p = NULL;

return 0;

}

2 动态内存开辟常见错误

2.1 对NULL指针的解引用

int\* p = (int\*)malloc(10000000000000000000);

\*p = 0;

开辟空间过大，开辟失败，则返回空指针，不能对空指针（或者野指针进行解引用操作）；解决：内存开辟过后，一定记得判断内存是否开辟成功。

2.2 对动态开辟的空间的越界访问

#include<stdio.h>

#include<stdlib.h>

#include <malloc.h>

int main()

{

int\* p = (int\*)malloc(100\*sizeof(int));

if (p == NULL)

{

perror("main");

return 1;

}

int i = 0;

for (i = 0; i < 120; i++)

{

\*(p + i) = i;

}

for (i = 0; i < 120; i++)

{

printf("%d ", \*(p + i));

}

free(p);

p = NULL;

return 0;

}

对空间的使用量大于开辟的空间，存在内存越界访问危险，注意不能超出开辟的空间大小。

2.3 使用free()释放非动态开辟的空间

char str[20] = "hello world";

char\* p = str;

free(p);

p = NULL;

局部变量或函数是在栈区开辟空间的，并非动态内存，不能用free()函数释放，注意只有动态开辟的空间才可用free()函数释放。

2.4 使用free()函数释放动态内存的一部分

#include<stdio.h>

#include<stdlib.h>

#include <malloc.h>

int main()

{

int\* p = (int\*)malloc(100 \* sizeof(int));

int i = 0;

if (p == NULL)

{

perror("main");

return 1;

}

for (i = 0; i < 10; i++)

{

\*p++ = i;

}

free(p);

p = NULL;

return 0;

}

p++使得p指针的指向始终在改变，当我们对p指针进行释放时，已经不是对整个开辟的内存进行释放了，而只是开辟内存的一部分，注意释放时一定要是开辟内存的首地址。

2.5 对同一块动态内存多次释放

对同一块动态开辟的空间进行多次释放，不要多次使用free()函数释放同一块动态开辟空间。

2.6 动态开辟的内存忘记释放了（内存泄漏）

#include<stdio.h>

#include<stdlib.h>

#include <malloc.h>

void test()

{

int\* p = (int\*)malloc(100 \* sizeof(int));

if (p == NULL)

{

perror("test");

return;

}

}

int main()

{

test();

return 0;

}

指针p是一个局部变量，当函数结束后，p指针销毁，则我们找不到动态开辟的空间了，动态开辟的空间只有通过free()函数手动释放，或者程序结束释放，其余情况不会自动销毁。这样，如果我们不释放内存，则会导致内存不可寻找，从而导致内存的白白流失，内存泄漏。注意，申请动态内存之后要及时的释放。

综上所述，我们在进行动态内存开辟时需要注意其中的每一个步骤细节，即内存申请、内存判断、内存使用、内存释放、指针置空。

**几个例子：**

（1）#include<stdio.h>

#include<stdlib.h>

#include <string.h>

void GetMemory(char\* p)

{

p = malloc(100);//p为局部变量，没有进行动态内存释放，导致内存泄漏

}

void test()

{

char\* str = NULL;

GetMemory(str);//str为传值调用，不会改变str的指向，所以str仍为NULL

strcpy(str, "hello world");//不能对NULL指针进行操作

printf(str);

}

int main()

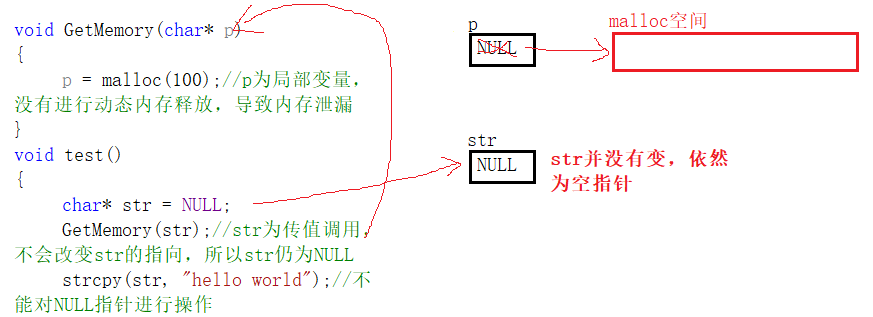
{

test();

return 0;

}

内存分析：



**修改：**

void GetMemory(char\*\* p)//指针的地址二级指针接收

{

\*p = malloc(100);//\*p为一级指针

}

void test()

{

char\* str = NULL;

GetMemory(&str);//str为传址调用，可以改变str的值

strcpy(str, "hello world");

printf(str);

free(str);//对动态开辟空间进行释放

str = NULL;

}

int main()

{

test();

return 0;

}

（2）#include<stdio.h>

#include<stdlib.h>

#include <string.h>

char\* GetMemory()

{

char p[] = "hello world";

return p;

}//GetMemory()函数是在栈区开辟的，函数结束，相应内存释放

void test()

{

char\* str = NULL;

str = GetMemory();//虽然str得到一个地址，但地址的空间已经销毁，没有意义

printf(str);

}

int main()

{

test();

return 0;

}

（3）int\* f2(void)

{

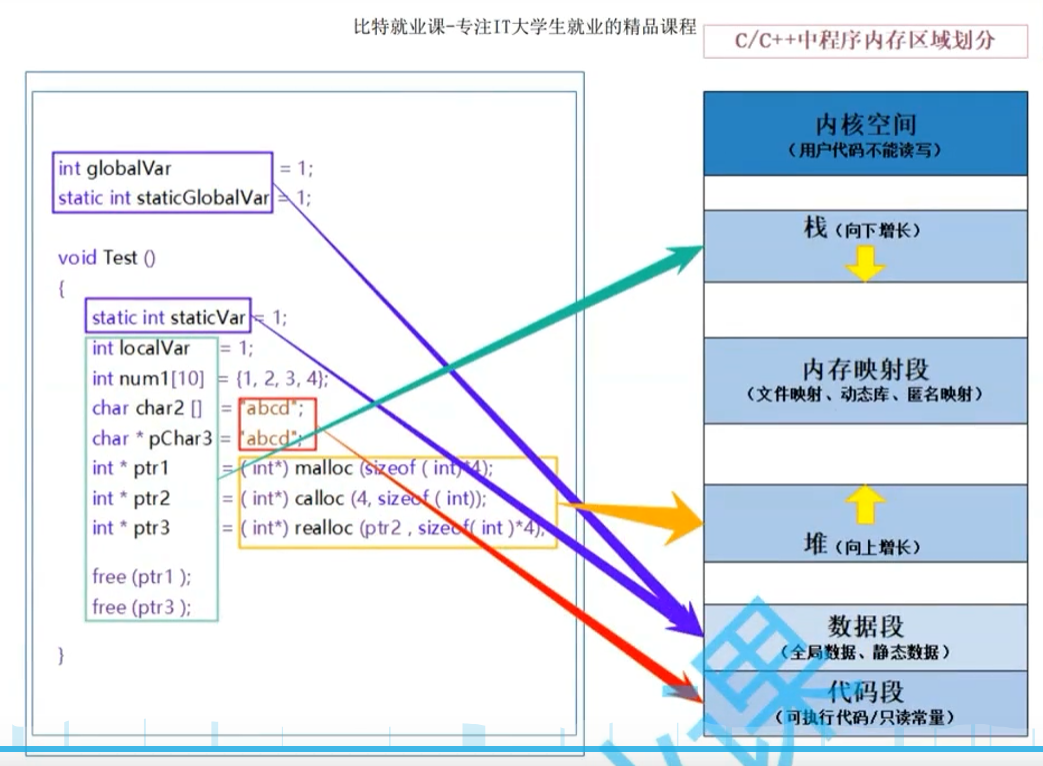
int\* ptr;

\*ptr = 10;//ptr没有赋初值，野指针，不能进行解引用操作

return ptr;

}

3 C/C++中程序内存区域划



堆区（动态内存开辟）、栈区（局部变量、函数）、静态区（数据段，全局变量和静态变量）、代码段（可执行二进制码，只读常量），内核空间（用户不能修改），内存映射段（文件映射，动态库，匿名映射）。

4 柔性数组

C99中，结构体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。例：

struct S

{

int n;

int arr[];

};

struct S1

{

int n1;

int arr1[0];

};

**柔性数组的特点：**

（1）结构体中的柔性数组成员前面必须至少有一个其他成员；

（2）sizeof返回的这种结构体类型大小不包括柔性数组的内存；

#include<stdio.h>

struct S

{

int n;

int arr[];

};

int main()

{

printf("%d\n", sizeof(struct S));

return 0;

}

（3）包含柔性数组成员的结构体用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体的大小，以适应柔性数组的预期大小。

#include<stdio.h>

#include<stdlib.h>

struct S

{

int n;

int arr[];//柔性数组

};

int main()

{

//假设arr[]数组预期大小为10个元素

struct S\* ps = (struct S\*)malloc(sizeof(struct S) + 10 \* sizeof(int));

//包含柔性数组的结构体内存动态开辟

if (ps == NULL)

{

perror("main1");

return 1;

}

ps->n = 10;

int i = 0;

for (i = 0; i < 10; i++)

{

ps->arr[i] = i;

}

printf("%d\n", ps->n);

for (i = 0; i < 10; i++)

{

printf("%d ", ps->arr[i]);

}

printf("\n");

//内存不够，重开辟 柔性数组预期总共20个

struct S\* ptr = (struct S\*)realloc(ps, sizeof(struct S) + 20 \* sizeof(int));

if (ptr == NULL)

{

printf("main2");

return 1;

}

ps = ptr;

for (i = 10; i < 20; i++)

{

ps->arr[i] = i;

}

printf("开辟过后\n");

printf("%d\n", ps->n);

for (i = 0; i < 20; i++)

{

printf("%d ", ps->arr[i]);

}

free(ps);

ps = NULL;

return 0;

}



以指针实现柔性数组功能：

#include<stdio.h>

#include<stdlib.h>

struct S

{

int n;

int\* arr;

};

int main()

{

struct S\* ps = (struct S\*)malloc(sizeof(struct S));

if (ps == NULL)

{

perror("main1");

return 1;

}

ps->n = 10;

ps->arr = (int\*)malloc(10 \* sizeof(int));

int i = 0;

for (i = 0; i < 10; i++)

{

ps->arr[i] = i;

}

printf("%d\n", ps->n);

for (i = 0; i < 10; i++)

{

printf("%d ", ps->arr[i]);

}

printf("\n");

//内存不够，重开辟

int\* ptr = (int\*)realloc(ps->arr, 20 \* sizeof(int));

if (ptr == NULL)

{

printf("main2");

return 1;

}

ps->arr = ptr;

for (i = 10; i < 20; i++)

{

ps->arr[i] = i;

}

printf("开辟过后\n");

printf("%d\n", ps->n);

for (i = 0; i < 20; i++)

{

printf("%d ", ps->arr[i]);

}

free(ps->arr);//内部动态空间先释放

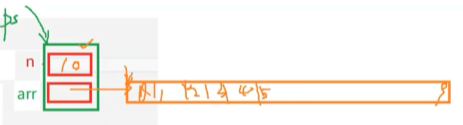
ps->arr = NULL;

free(ps);//外部动态内存后释放

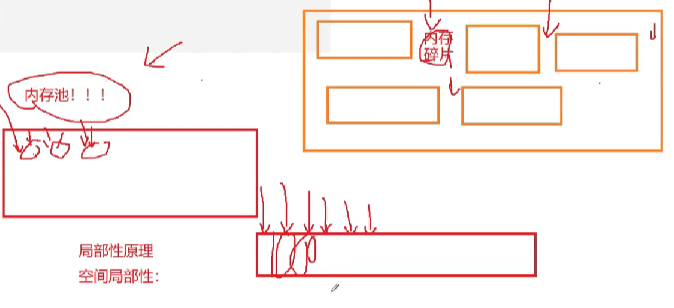
ps = NULL;

return 0;

}

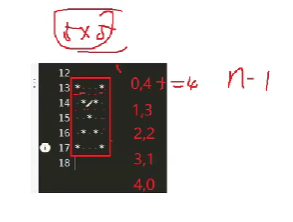


以指针实现柔性数组的缺点：（1）需要多次开辟和释放空间，容易出错。（2）两段开辟的空间不连续，容易形成较多内存碎片，也不利于程序运行效率。



**在线OJ刷题：**

例1：循环输入n打印X的\*号。



分析：对角线是\*号，其余位置都为空格，整体是一个矩形。

#include<stdio.h>

int main()

{

int i = 0;

int j = 0;

int n = 0;

while (scanf("%d", &n) != EOF)

{

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

if (i == j)

{

printf("\*");

}

else if (i + j == n - 1)

{

printf("\*");

}

else

{

printf(" ");

}

}

printf("\n");

}

}

return 0;

}

例2：考官打七个分数，打多组，键盘输入，去掉一个最高分和一个最低分取平均值打印换行。**一次循环解决所有计算**。

#include<stdio.h>

int main()

{

int count = 0;

int i = 0;

int j = 0;

float score[3][7] = { 0 };

float max[3] = { 0 };

float min[3] = { 100,100,100 };

float sum[3] = { 0 };

//输入多少组 3组

scanf("%d", &count);

//输入每组的分数

for (i = 0; i < count; i++)

{

for (j = 0; j < 7; j++)

{

scanf("%f", &score[i][j]);

sum[i] += score[i][j];//求和

if (score[i][j] > max[i])

{

max[i] = score[i][j];//找最大值

}

if (score[i][j] < min[i])

{

min[i] = score[i][j];//找最小值

}

}

}

for (i = 0; i < count; i++)

{

printf("%.2f\n", (sum[i] - max[i] - min[i])/5);//和去掉最大值和最小值求平均值打印

}

return 0;

}

例3：输入年份和月份，打印该年的该月有多少天。

#include<stdio.h>

int main()

{

int y = 0;

int m = 0;

int arr[13] = { 0,31,28,31,30,31,30,31,31,30,31,30,31 };//下标即为月份对应的天数

int day = 0;

while (scanf("%d %d", &y, &m) != EOF)

{

day = arr[m];

if ((y % 4 == 0 && y % 100 != 0) || y % 400 == 0)

{

if (m == 2)

{

day++;//闰年的2月加一

}

}

printf("%d\n", day);

}

return 0;

}

例4：实现一串升序数列中插入一个数组，使得插入后的数列仍为升序。

#include<stdio.h>

//实现一串升序数列中插入一个数组，使得插入后的数列仍为升序

int main()

{

int n = 0;

int arr[51] = { 0 };

int input = 0;

int i = 0;

int label = 0;//插入位置标号

//输入序列个数

scanf("%d", &n);

//输入序列

for (i = 0; i < n; i++)

{

scanf("%d", &arr[i]);

}

//输入插入的数

scanf("%d", &input);

//查找插入数的位置

for (i = 0; i < n - 1; i++)

{

//1.最左边 边界条件

if (input <= arr[0])

{

label = 0;

}

//2.最右边

if (input >= arr[n - 1])

{

label = n;

}

//3.中间某个位置

if (input >= arr[i] && input <= arr[i + 1])

{

label = i + 1;

}

}

//插入数

for (i = n; i > label; i--)

{

arr[i] = arr[i - 1];

}

arr[label] = input;

//打印插入后的数组

for (i = 0; i < n + 1; i++)

{

printf("%d ", arr[i]);

}

printf("\n");

return 0;

}

**5 方向选择**

