目录

下面有几段代码都是有问题的：

1.

2.

3.

4.

---

free

C语言提供了一个函数free，专门是用来做动态内存的释放和回收的，函数原型如下：

```
void free (void* ptr);
```

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。

malloc

```
void* malloc (size_t size);
```

这个函数向内存申请一块连续可用的空间（size的单位是字节），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(在VS2022上malloc会象征性的给你一个地址，当你访问这个地址的时候就会造成越界访问)

```
int main()
{
    int* p = NULL;
    p = malloc(3 * sizeof(int));
    if (p == NULL)
        perror("malloc");
    else
    {
        //......
    }
    free(p);
    p = NULL;
    return 0;
}
```

calloc

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

函数的功能是为 num 个大小为 size（单位也为字节） 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

```
int main()
{
    int* p = NULL;
    p = calloc(3, 4);
    if (p == NULL)
        perror("calloc");
    else
    {
        for (int j = 0; j < 3; j++)
        {
            printf("%d ",*(p+j));
        }
    }
    free(p);
    p = NULL;
    return 0;
}
```

realloc

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，那为了合理的分配内存，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。

```
//函数原型如下：
void* realloc (void* ptr, size_t size);
```

ptr 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在调整内存空间的是存在两种情况：

情况1：原有空间之后有足够大的空间

这时直接在后面开辟空间，并返回起始地址（就是传过来的那个）。

情况2：原有空间之后没有足够大的空间

这时编译器会在内存的另一块空间开辟size大小的地址，并将原空间的数据拷贝到新空间，返回新空间的起始地址。

常见的动态内存的错误

1.对NULL指针的解引用操作

```
void test()
{
    int *p = (int *)malloc(INT_MAX/4);
    *p = 20;//如果p的值是NULL，就会有问题
    free(p);
}
```

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.对非动态开辟内存使用free释放

```
void test()
{
    int a = 10;
    int *p = &a;
    free(p);//ok?
}
```

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
}
```

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
}
```

6.动态开辟内存忘记释放（内存泄漏）

```
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漏。
切记：动态开辟的空间一定要释放，并且正确释放

下面有几段代码都是有问题的：

1.

```
void GetMemory(char* p)
{
    p = (char*)malloc(100);
}
void Test(void)
{
    char* str = NULL;
    GetMemory(str);//这里str并没有改变，p是在函数内部创建的变量，在出函数的时候就会被销
```

```
毁，
                    //这里应该传二级指针。
    strcpy(str, "hello world");
    printf(str);
}
main()
{
    Test();
    return 0;
}
```

2.

```
char* GetMemory(void)
{
    char p[] = "hello world";
    return p;
}
void Test(void)
{
    char* str = NULL;
    str = GetMemory();//在这个函数被调用之后，在函数内部创建的串"hello world"的空间就
会被操作系
                        //统回收，最后打印出：烫烫烫......
    printf(str);//这里打印是没有问题的!
}

int main()
{
    Test();

    return 0;
}
```

3.

```
void GetMemory(char** p, int num)
{
    *p = (char*)malloc(num);
```

```
}
void Test(void)
{
    char* str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
        //在使用完了空间后没有free!
}

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

4.

```
void Test(void)
{
    char* str = (char*)malloc(100);
    strcpy(str, "hello");
    free(str);
    if (str != NULL)//这里str已经被free掉了，按理来说就不能使用了
    {
        strcpy(str, "world");//编译器会报警告：C6001使用未初始化的内存“str”
        printf(str);
    }
}
int main()
{
    Test();
    return 0;
}
```

本期博客到这里就结束了，如果有什么错误，欢迎指出，如果对你有帮助，请点个赞，谢谢!