核心思想：谁申请的，谁释放；

对于文件：谁打开的谁关闭

heap上的内存需要释放的原因：虽然 在程序结束之后，操作系统会把 heap释放掉，但是，这是对一些 小型的程序而言可以的，因为 它的运行 可能消耗的 内存不是很多，但是：如果是 大型的服务器程序，那程序运行很长的 时间，这时 如果 你heap上数据 不清除 服务器一定会在某个时间上奔溃！

我觉得程序员思想应该是：不能说 系统会帮我们做，我们就不做了！而是，如果我们可以做，我们就去做！！

malloc alloc realloc

返回值 均是一个 void *va

```
1. malloc (int)   int表示申请的内存的大小
2. calloc (size_t nmemb, size_t size) 申请 nmemeb * size 大小的内存 (size表示单位数据大小， nmemb表示要多少个此单位大小的内存)
3. realloc (void* ptr, size_t size)  对 ptr重新分配内存 (ptr 必须是 malloc或者calloc的返回值) , 如果，ptr起始的地址空间中，内存不足 size，则会重新找一片 内存空间，分配好size内存，然后返回 新的ptr值

1. free (void* ptr) 释放 **ptr** 空间
```

$\color{red}{注意free之后， 按道理，p是不能够再去对之前对应的那片内存进行操作的，因为很有可能那片内存分给了别人！ **但是,由于p指针，本身是栈上的一个数据，其只有程序结束时，才会被释放，如果，你在free之后，任然使用了p，那么编译器并不会报错，但十分危险，因为你很可能修改别人的内容（地址已分配给别人了）}$

但是注意 $\color{blue}{而如果 p指向 栈上的变量 a，那么在 free之后，对应的 变量a和指针p，都不可以再进行任何调用，否则直接报错！！！ （不向上面说的 如果 你真的只是 p申请内存，然后在p内存中进行操作，然后free掉后，你再调用它，程序也不报错，但这里不行！！！）}$

$\color{green}{而且，其实返回指向栈数据的指针的操作就是错的！！本质上还是因为：栈是随着函数结束后被销毁的，而如果是堆指针，或static 变量 const变量的指针，那是可以返回的！！ （再或者 利用传入传出参数！！！ （调用函数和被调用函数，指针都在双方函数内部存在！！））}$

```
    int* p = (int*)malloc(sizeof(int));
    int a = 10;
    *p = a;
    printf("%p------>:%d\n", p, *p);

    free(p);

    *p = 123;                               // 报错
    a=123;                                  // 报错
```

```
    printf("%p------>:%d\n", p, *p);     // 报错
```

```
    int* p = (int*)malloc(sizeof(int));

    if (p == 0) {
        free(p);
        printf("No space for malloc\n");
    }

    *p = 10;
    printf("%p----->:%d\n", p, *p);
    free(p);

    // 不报错！！
    *p = 123;
    printf("%p----->:%d\n", p, *p);
```

```
int* p = (int*)malloc(sizeof(int));

if (p == 0) {
    free(p);
    printf("No space for malloc\n");
}

*p = 10;
printf("%p----->:%d\n", p, *p);
free(p);

*p = 123;
printf("%p----->:%d\n", p, *p);
```

```
000001A227EC4C50----->:10
000001A227EC4C50----->:123
```

p任然可以去修改那片地址里的内容，这很危险

```
int* getptr() {

    int* p;
    int a = 10;
    p = &a;

    printf("%p------>:\n",&a);

    a = 20;
    printf("%p------>:%d\n", p, *p);

    return p;
}

void testptr() {
    int* p = getptr();
    *p = 40;

    printf("test %p------>:%d\n", p,*p);
```

}这段C/C++代码到底有没有问题？因为按道理来说，是不可以返回指向栈数据的指针的，因为函数结束后，栈上内存会被释放。但是，为什么在代码测试的时候，testptr中，函数运行正常，p的解引用和p本身输出（p地址结果和在getptr函数中p的地址一致）都是正常的呢？

看下 GPT给出的回答：

这段代码存在严重的问题，即函数 `getptr` 返回一个指向局部变量的指针，这是未定义行为。在C/C++中，局部变量的生命周期在函数返回时结束，返回指向局部变量的指针会导致访问已经被释放的栈内存。

虽然在某些情况下，这段代码在测试中可能表现正常，但这是因为未定义行为有时可能看起来像是"工作正常"，但实际上这是偶然的。具体来说，这里发生的情况可能是：

1. **未定义行为的影响**：未定义行为意味着程序的行为是不可预测的，可能在不同的编译器、不同的优化级别、甚至在不同的运行条件下表现不同。在某些情况下，返回的指针仍然指向有效的数据，但这只是运气好而已。
2. **栈内存未被覆盖**：在你的特定测试中，函数 `getptr` 返回后，栈内存没有立即被覆盖，因此 `testptr` 中对指针 `p` 的解引用看起来是有效的。这种情况非常脆弱，稍微改变代码，增加其他函数调用，或改变编译器设置，都可能导致程序崩溃或行为异常。
3. **调试器行为**：如果使用调试器运行代码，调试器有时会保持栈帧信息不变，以便于调试。这可能使代码表现得比实际运行时更稳定。

为了解决这个问题，应该避免返回指向局部变量的指针。你可以考虑以下几种替代方法：

1. **使用动态内存分配**：

c 复制代码

```c
int* getptr() {
    int* p = (int*)malloc(sizeof(int));
    *p = 10;
    printf("%p------>:\n", p);
    *p = 20;
    printf("%p------>:%d\n", p, *p);
    return p;
}

void testptr() {
    int* p = getptr();
    *p = 40;
    printf("test %p------>:%d\n", p, *p);
    free(p);
}
```

2. **通过参数传递指针**：

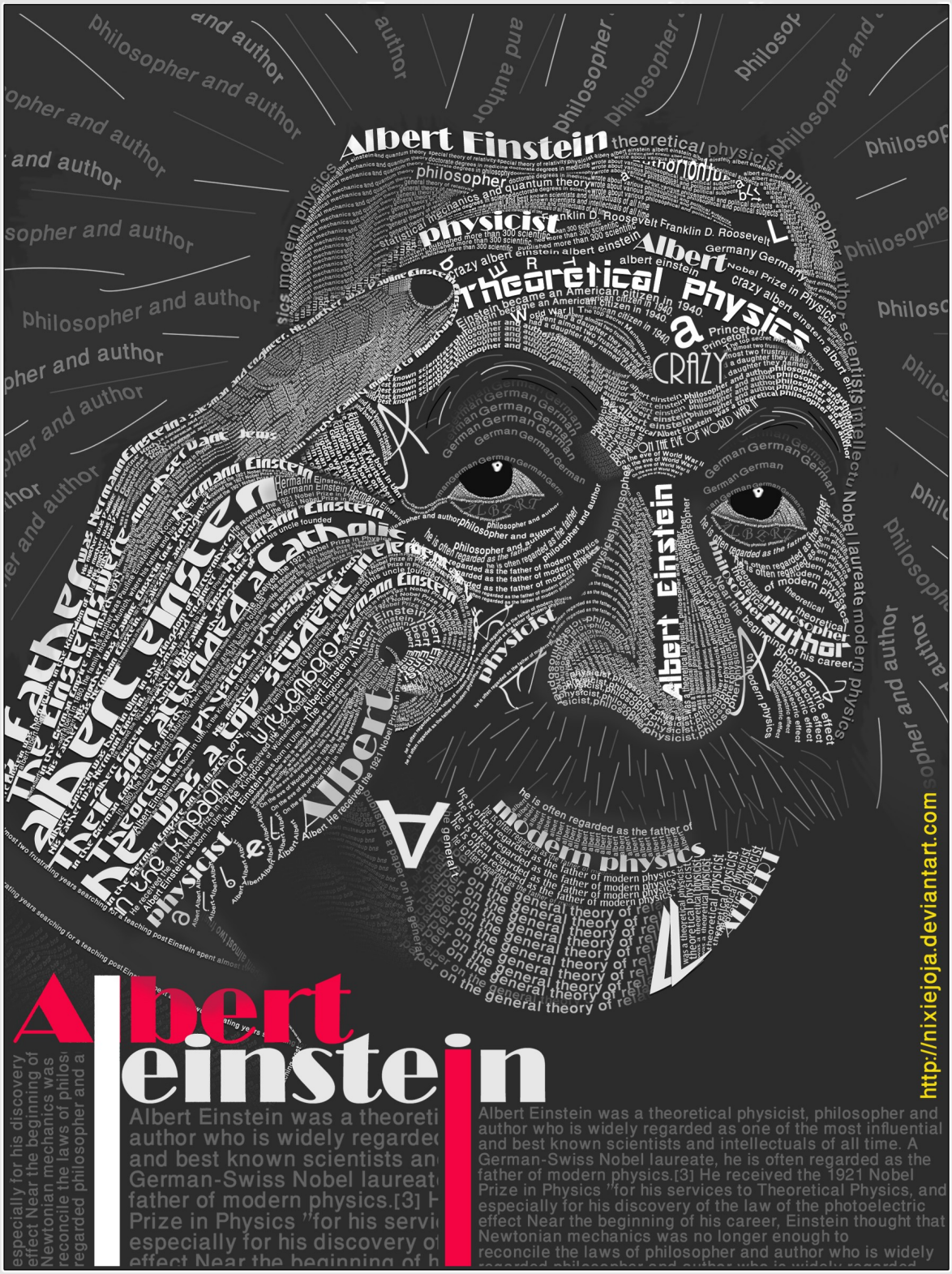
c 复制代码

```c
void getptr(int* p) {
    *p = 10;
    printf("%p------>:\n", p);
    *p = 20;
    printf("%p------>:%d\n", p, *p);
}

void testptr() {
    int a;
    getptr(&a);
    a = 40;
    printf("test %p------>:%d\n", &a, a);
}
```

而又对于：

```
int *p =new int();

return p;
```

这个也是可以的！虽然，对于 **p指针**这个变量本身而言，其属于栈上数据，但是 其指向的内存是 new关键字在堆上开辟的（后序使用 deldete删除），所以可以返回并使用！！！（函数 可以返回 **局部变量**，此时，这里的p就是作为一个单纯的局部变量被返回的）



指针和其解引用

```
char Mychar[] = "1234";
int len = strlen(Mychar);
/*cout << len << endl;*/

cout << "------" << endl;

for (char* i = Mychar;;) {
    printf("%c", *i);
    if (*i == '\0') {
        printf("over");
        break;
    }
    i++;
}
```

Mychar就是一个char *的地址，解引用是一个char字符;而且如果不加break，可以打出非常多的错误字符，其实就是c对于该情况下内存限制不严格导致！！