勇敢牛牛 不怕程设

指针篇

> *都说C语言指针最难，我看未必……难的是助教那阴间的题目和测试点* - 窝鬟斐虾

所以……到底什么是指针？

指针就是一个变量！只不过这个变量有那么一丢丢的特殊，他存放的是一个**地址**，指向了内存当中的某个位置，就好像是你平常使用的`int`是房子，而`int*`是房子的门牌号一样。

还记得scanf吗？

当大家初学`scanf`的时候，可能也会疑惑为什么要使用`&`符号，现在学习了指针相信大家也明白了，因为函数的传值并不是传送了本体。例如以下情况：

```
void TryToMessAroundWithYourVariables(int a) {
    a = 1000;
}

int main() {
    int a = 5;
    TryToMessAroundWithYourVariables(a);
    printf("%d", a);
}
```

你会发现，尽管我们在函数内对`a`设置成了`1000`，但是最终输出仍然是`5`，因为在调用函数的时候实际上是将`int`复制了一份传递给了函数，函数内部的`a`和外部的`a`本质上已经不是同一个东西了。所以你再怎么去改变函数内的`a`，为对外面的`a`也毫无影响……
所以怎么办呢？如果我们没办法改动`a`的值，我们直接**掏他老窝！**
我们不再传入`a`，而是传入`&a`，这样，传入的值就是准确地指向`a`真正的位置，当我们在函数内使用`*a = 1000`的时候，就相当于我们顺着地址直接到了`a`的家门口，强行给他更改成了`1000`，这样函数就成功改动了`a`的值。

注意野指针！
当你题目中出现**SIGSEGV**的时候，如果排查了所有的数组越界，不妨排查一下有没有访问**野指针**的情况，野指针就是指向的位置不属于程序的范围或者完全随机，这种访问是危险且不允许的。
为了避免野指针访问，我们一般把指针初始化为`NULL`。

指针……动态数组**

从实际上来说，数组**和我们之前学习的内容相比没有任何特殊之处**，数组的名称其实就是一个**指针**，只不过这个指针你**不能改变它的指向的位置**。

```
int a[10];
```

其中`a`这个量就是一个指针，他指向了数组第一个元素。
`a[offset]`其实真正的意思是：从`a`指向的位置开始，向后偏移`offset`位，再解引用。现在明白为什么数组第一位是`0`下标了吧？因为没有`offset`，没有偏移就是第一个元素嘛！
所以像这种写法：

```
int a = 4;
(&a)[0] = 4;
```

~~其实是允许的但是这样写容易被打……而且还被windows拦截成了病毒~~
现在理解了数组的本质，我们便可以开始实现**动态数组**

```
int n;
scanf("%d", &n),
int array[n];
// NONONONONONONONONONONONO!!!!!JESUS CHRIST JUST NOO!!!
```

像这种写法是不允许的，那么有没有办法在运行过程中间创建数组呢？
sure!

```
int* array = (int*)malloc(sizeof(int) * arrayLength);
```

这样的写法就能动态申请一个长度为`arrayLength`的`int`数组，根据前面的知识，你能理解它吗？~~不理解就多看几遍~~

函数指针 ~~委托嘛！~~

```
void (*function)();
```

像这种写法，可以声明一个名称为`function`的函数指针，该指针指向的类型为返回值为`void`，参数列表为`()`的函数。其实这个和`qsort`里的比较规则函数很像！你可以和前面的类比。
函数名称就是一个函数指针，只不过不允许修改
你也可以把函数指针理解成一个数组，只不过……指向的是一堆指令的集合。

巧用指针避免传值和复制

如果有人写过一些需要传巨大大小的参数时，有可能会出现一种奇怪的bug。如果你使用DEBUG模式调试，会出现：

```
void Function(DataType type) <<== SIGSEGV ERROR
{
    //...
}
```

这是什么问题！？指着函数的名字出错？？？
其实这种情况大多是因为你传入了过大的值导致了函数**爆栈**。当函数进行调用的时候，会将函数参数列表里所有的参数压入一个调用栈当中。如果你传入的参数太大，这个栈就**BIM BIM BOOM!** 爆炸啦！
所以我们要避免这种情况，可以巧妙地使用指针，因为虽然参数本体很大，但是指针却一般就4个字节，就避免啦！
例如我们要传入之前所说的`struct`：

```
typedef struct Matrix{
    long long base[500][500];
} Matrix;

void FunctionBoom(Matrix matrix) {
    //...
}

void FunctionGood(Matrix* matrix) {
    //...
}

int main() {
    Matrix matrix;
    FunctionBoom(matrix); //危险，传入的参数太大，很容易爆栈
    FunctionGood(&matrix); //安全，传入的参数只是一个指针，不会爆栈
}
```

但是要注意，如果使用指针，你也就默许了可以在函数内部更改参数的值，所以应该倍加小心或者使用`const`指针。

指针常量和常量指针

记住就行了，指针常量就是说指针的**指向位置不能改变**，但是可以改变指向的值
常量指针就是允许改变指针的指向，但是**不能改变指向的值**。
常量指针：`const int* a`
指针常量：`int* const a`
~~杂交指针~~：`const int* const a`，又不允许改变指向位置也不允许改变指向的值。

字符串操作

---

1. 字符串本质上是用字符数组来存的，但是**字符串一定要以'\0'结尾**！
2. 字符串的一些基本操作：

遍历

```
char s[100];
int len=strlen(s);
for(int i=0;i<len;i++){
    //
}

//注意不要写成以下代码！！！
for(int i=0;i<strlen(s);i++){

}
//这种循环方式每次都会调用一次strlen函数，大大增加循环的运行时间
```

插入字符串
我们当然可以进行遍历，按照字符进行操作，但是我更提倡使用c库里的一些函数，不容易出错

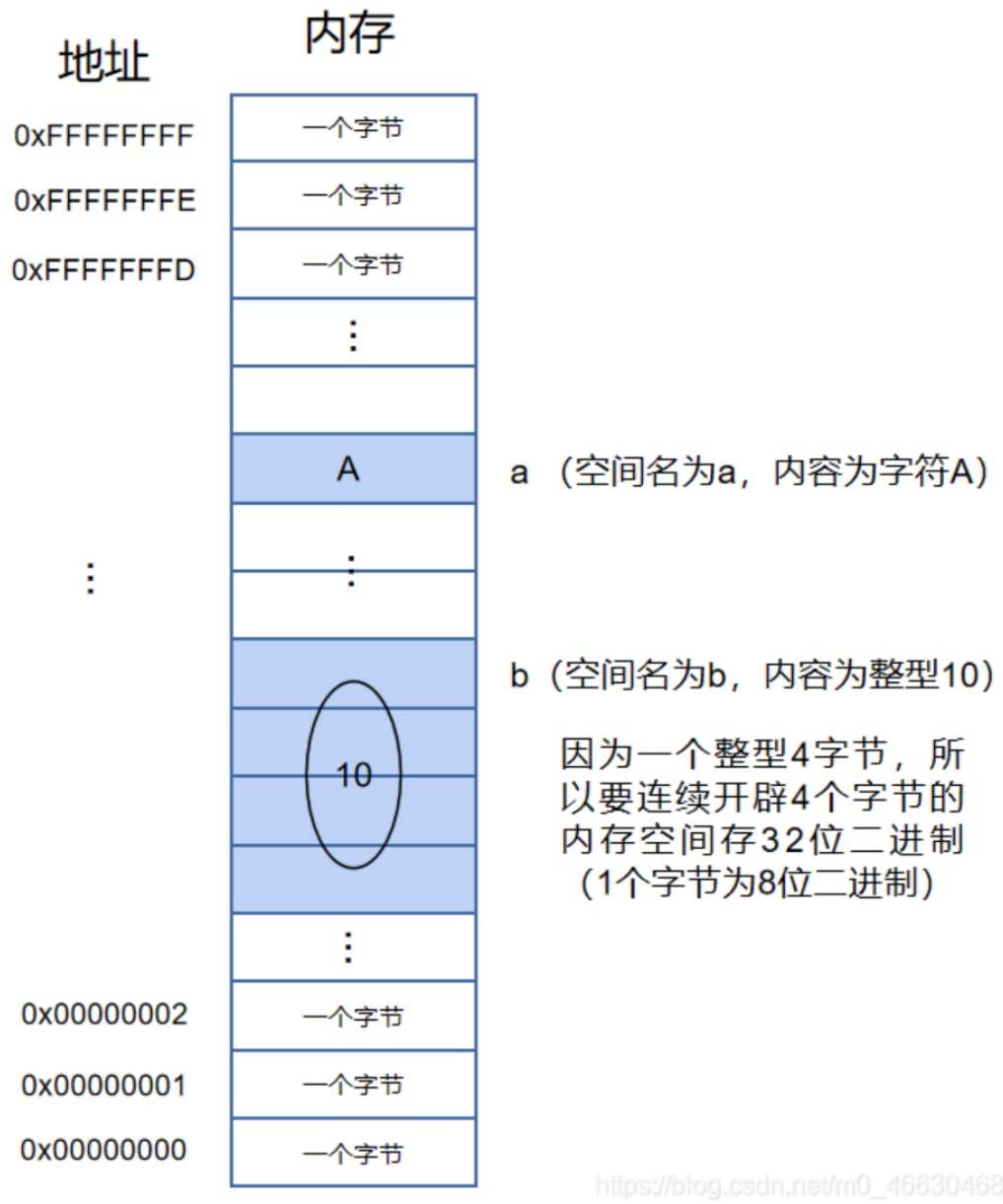
```
char a[100],b[100];
//将b插入到a的第n个字符后面
strcat(b,a+n);
strcpy(a+n,b);
//将a的第n位后面的字符拼接到b字符串后面，将b复制覆盖a的第n位后的字符
```

其余的一些常用操作我不进行过多举例，相信大家平常多少有存过一些模板。

指针和数组的一些关系

指针也是一种变量类型，不过这个变量存的不是整数，也不是浮点数，存的是地址。（*嗯，学了计组以后，理解会更深刻的*）

- C语言中定义变量都是在内存中定义的，定义变量的本质是开辟空间。
  比如我们`char a=A, int b=10`，那么系统会开辟一块属于a的内存空间，把A存到这个空间内，b也同理。
- 为了有效的使用内存，把内存划分成一个个小的内存单元，每个内存单元的大小是1个字节。
- 为了能够有效的访问即快速找到内存的每个单元，就给内存单元进行了编号，这些编号被称为该内存单元的地址。
- 以下图较为形象的展示了上述内容



https://blog.csdn.net/m0_46630468

c语言内的地址就是32位的数字，例如0x32ff4466之类的，也就是占4个字节。因为指针存的是地址，地址的长度是不变的，所以任何数据类型的指针都是4个字节的长度。

因为是指针涉及对底层的内存地址进行操作，所以指针访问的速度是极快的，这是c和c++运行速度比其他一些高级语言快的重要原因之一。同时对内存里面的内容直接进行各种修改，这几乎是一种不受到任何限制的权力。

没有足够的能力，获得过大的权力就容易失控，所以尽量少用指针（我的程设老师的劝诫）！

指针和数组之间其实有着紧密的联系。

数组本质上是开辟了一段连续的内存空间。我们通过下标进行访问，实际上就是在对指针做一种

偏移操作。

指针数组和数组指针，指针函数和函数指针，不知道大家有没有搞清楚呢，后者可以不用掌握，前者的区别简单来说如下。

- 指针数组是一个**数组**，数组里面的元素存的是指针。
- 数组指针是一个**指针**，这个指针指向一块连续的内存空间，有多大取决于你数组开辟的长度。

这里我以数组和数组指针的关系为例进行说明。

```
int a[5]={1,2,3,4,5};
int (*p)[5]=NULL;
p=&a;//a代表数组首元素的地址，&a代表整个数组的地址

int i;
for(i=0;i<5;i++)
{
    printf("%d ",*(*p+i));//遍历输出数组的值
}
```

我们应该知道的是数组名可以代表两种含义：①数组首元素地址②整个数组

显然在给数组指针赋值时，用到了②这个含义，取地址后代表了整个数组的地址，赋给了数组指针p，此处虽然地址值和光写一个a时所代表的值一样，但其意义却不一样。

```
int a[2][3]={1,2,3,4,5,6};
int (*p)[3]=NULL;
p=a;

int i,j;
for(i=0;i<2;i++)
{
    for(j=0;j<3;j++)
        {
            printf("%d ",*(*(p+i)+j));//遍历输出
        }
}
```

tips：其实二维数组本质上还是一维数组，`a[2][3]`本质上还是开辟6个int的连续空间，不过是把每3个连续的元素分成了一组，便于通过下标访问。所以`*a[0]`本质上是第一个元素，`*a[1]`本质上是第4个元素。

如果感到烧脑，一定运行该程序，一步步调试，慢慢分析，即看清楚p里存的是什么，*p取出存的东西又是什么，对地址进行加减又意味着着什么。

快排篇

> *学了快排，什么都用快排，谁还用冒泡啊。* - 万能福虾

让我们首先看看快速排序的函数原型：

```
void qsort(
    void *_Base,
    size_t _NumOfElements,
    size_t _SizeOfElements,
    int (__cdecl *_PtFuncCompare)(const void *, const void *)
)
```

怎么样，是不是有点不像人话？我给你翻译一下

```
void qsort(
    void *排序对象的数组指针
    size_t 排序对象的个数
    size_t 一个排序对象的大小
    int (__cdecl *用于比较大小的规则)(const void *, const void *)
)
```

其中一般最重要的就是最后一个**规则**，它决定了你排序的正确性和结果，而前面几个，相信大家一般也都会输入了。

书写排序规则

排序规则的原型要求你的定义必须与格式**完全一致**（函数名和形参名不必相同）

```
int comparator(const void* objectA, const void* objectB);
```

为什么必须要求是`const void*`呢？因为C本身不知道你想要对什么类型进行排序，所以只能指向一个不知道是什么东西的地址，也就是`void*`，为了禁止你在排序的过程中更改它，所以必须是`const`。
一般来说对于一个`int`排序，`comparator`函数定义如：

```
int comparator(const void* objectA, const void* objectB) {
    int* integerA = (const int*) objectA;
    int* integerB = (const int*) objectB;
    return (*integerA) < (*integerB) ? -1 : 1;
}
```

这个排序规则可以对一个数组进行升序排序

- 最后返回的`-1`和`1`怎么理解
  你可以把每次比较的时候传入的参数左边的是-1，右边的是1，比较规则返回的就是哪个排在前面

  ```
  int comparator(const void* objectA, const void* objectB) {
      int* integerA = (const int*) objectA;
      int* integerB = (const int*) objectB;
      return (*integerA) < (*integerB) ? 1 : -1;
  }
  ```

  比如如果我这么写，就是问程序：*A和B进行比较的话，A比B小吗？如果A小就把B排在前边，否则把A排在前边。*
  整理一下，其实也就是**降序排序**，把大的排在前边

常见错误

~~类似于像把参数位置写错位置的就不说了）~~
最最最搞的错误

- 注意看我们前面对`int`进行排序的时候，是不是排序规则里强行转化的目标是`const int*`对吧，你有没有发现，当我们对一个类型进行排序的时候，排序规则里必须要写的是**它的指针**？
  平时可能不会出这种错误，但是如果你要对一堆**指针对象**进行排序的时候，可能就出错了。

```
int* bunchOfPointers[20] = /*...*/;
int compare(const void *a, const void *b) {
    const int** pointerA = (const int**) a;
    const int** pointerB = (const int**) b;
    return /*...*/;
}

qsort(bunchOfPointers, 20, sizeof(int*), compare);
```

  注意在排序规则当中，我转换的对象是`const int**`！如果你转化为`const int*`，那么排序的结果多半是错误的

  这种错误其实最容易出现在字符串指针的排序，当你要对一堆`char*`进行排序的时候，记住，你要转换成的对象是`const char**`二级指针

和结构体的应用（大杀器！）

相信大家有的时候可能会遇到这种情况，比如现在给你一堆瓜，给出他们的**价格，大小等信息**要你按照他们的大小作为第一关键字进行排序，然后再依次输出。

怎么做呢？应该很多人会直接开始设置数组：

```
int melonPrice[100];
int melonSize[100];
```

按第一关键字进行排序……那么就对`melonPrice`排序吧！
现在问题来了，你对`Price`进行排序了，但是`Size`没有呀……而且对应的顺序也就乱了，怎么办？

结构体介绍 申佬版

- C 数组允许定义可存储相同类型数据项的变量，结构是 C 编程中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。
- 结构体中的数据成员可以是基本数据类型（如 int、float、char 等），也可以是其他结构体类型、指针类型等。
- 简单来说，结构体就是自定义一种新的数据类型，这个新的数据类型可以看作其他数据类型的一个集合。

-假设我们要存一本书的相关信息。我们可以这样写一个结构体。

```
struct book{
    char[50] name;//书名
    double price;//价格
    int page_num;//页码
    //...还可以定义更多的属性
}

struct book A[100];//开一个A数组来存我们定义的结构体。
//可以通过以下方法对里面的元素进行访问。
A[0].name=//;
A[0].price=//;
```

结构体的优势在于，他将一个元素具有的某些性质绑定在了一起，我们可以按照这个元素具有的某些性质来对元素进行合理的排序。

结构体介绍 窝鬣虾版

隆重介绍……**结构体**(struct)!
在上面的情况中，我们的问题其实归根结底是两种数据是分离的，并没有某种绑定关系，利用结构体，我们可以很好地进行绑定。

```
struct Melon {
    int price, size;
};
```

其中`Melon`是结构体的名称，在花括号里的是结构体的成员。
现在，我们定义了一个叫`Melon`的结构体，怎么使用它呢？

```
struct Melon melons[100];
```

如果我们要访问某个瓜的属性，就直接

```
melons[0].size = 100;
melons[0].price = 2;
```

如果你要进行排序，现在就容易多了，因为现在`Melon`把两个属性绑定在了一个大结构下。
……
还是有点云里雾里？给你举个把一堆瓜按大小进行升序排序的例子。

```
struct Melon melons[200];
int compare(const void* a, const void* b) {
    const Melon* melonA = (const Melon*) a;
    const Melon* melonB = (const Melon*) b;
    return melonA->size < melonB->size ? -1 : 1;
}
qsort(melons, 200, sizeof(struct Melon), compare);
```

发现了吗？经过这样的包装，你**不再需要考虑数组是否保持对齐**，因为`Melon`这个结构把两个数固定地绑在了一起！
当然，你会发现好像每次用`Melon`这个名称的时候前面都要加一个`struct`挺讨厌的，那么我们可以写一个
`typedef struct Melon Melon;`
或者你在定义的时候直接写：

```
typedef struct Melon{
    int price, size;
} Melon;
```

这样，在你之后使用`Melon`这个名称的时候，前面就不需要加`struct`啦！

补充

如果你有一个`Melon*`指针`ptr`，你可以通过`ptr->price`来访问成员，也可以使用`(*ptr).price`来访问，但是明显前者要简洁一点点

- by 虾 and 申佬