C语言入门（*CS100*）Part Four

> *-- By Yiming Li*

> 本文章的内容参考至*Stephen Prata*的《C Primer Plus》

> 本内容适用于*CS*专业学生

指针-（内存地址）

指针的基本知识

- 指针变量的定义：数据类型 * 变量名，其中数据类型要和指向变量的类型保持一致。

```
int a = 0;
int* pi = &a;

double b = 20;
double* p2 = &b;
```

- 指针的作用之查询变量名 方式：* 变量名;其中*表示解引用运算符，表示通过对应的内存地址来获取对应的数据
- 存储/修改数据：*p = 200;同样的将*写在左边表示存储数据
- 注意：
    1. 指针变量占用的大小，跟数据类型无关：32位：4个字节，64位：8个字节。
    2. 给指针变量赋值的时候，不能把一个数值赋给指针变量int* p = 500;,这里的500都没有内存分配，在内存分配了的才能够这么写。

指针的作用

在函数中修改外部数据

```
#include <stdio.h>
void swap(int num1,int num2);
int main()
{
    int a = 10;
    int b = 20;
    printf("调用前 %d,%d",a,b);//10 20
    swap(a,b);
    printf("调用后 %d,%d",a,b);//10 20
    return 0;
}
void swap(int num1,int num2)
{
    int temp = num1;
    num1 = num2;
```

```
    num2 = temp;
}
```

- 你会发现根本就没有进行交换：这是为什么？因为和之前的传递的`int arr[]`不一样这个地方传递的不是地址而是数值，因此在`num1`,`num2`中交换的数据是不会改变`a`,`b`的数值的。
- 怎么进行修改呢？

```
#include <stdio.h>
void swap(int num1,int num2);
int main()
{
    int a = 10;
    int b = 20;
    printf("调用前 %d,%d",a,b);//10 20
    swap(&a,&b);
    printf("调用后 %d,%d",a,b);//20 10
    return 0;
}
void swap(int* p1,int* p2)
{
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
    // 相当于直接修改的就是相同地址的数据
}
```

- 一个小细节：
  - 函数中的变量的生命周期和函数相关，函数结束了，变量也会消失，此时在其他函数中，就无法通过指针使用了，**如果不想函数中的变量被回收，可以在变量前面添加`static`关键字**

```
#include <stdio.h>
int *method();
int main()
{
    // 调用method函数并且使用其中的变量a
    int *p = method();
    printf("%d\n", *p); // 如果没有static不能够打印的，因为method函数结束之后，该函数中的所有的变量都会随之消失
    return 0;
}
int *method()
{
    static int a = 10;//地址静态
    return &a;
}
```

函数返回多个值

```c
// 获取多个“返回值”
#include <stdio.h>
void FIND_MAX_MIN(int arr[], int len, int *max, int *min);
int main()
{
    int arr[] = {1, 2, 3, 45, 101, 0, -2};
    int len = sizeof(arr) / sizeof(int);
    int max = arr[0];
    int min = arr[0];
    FIND_MAX_MIN(arr, len, &max, &min);
    printf("max = %d,\tmin = %d", max, min);
}
void FIND_MAX_MIN(int arr[], int len, int *max, int *min)
{
    for (int i = 1; i < len; i++)
    {
        if (arr[i] > *max)
            *max = arr[i];
        else if (arr[i] < *min)
            *min = arr[i];
    }
}
```

- 其实我个人偏向于理解为`int* max = &max`表示`max`变量的地址，传入的也是这个地址。`*max`表示的就是改变的`max`这个地址的变量的内容。

指针的高级用法

指针的计算

- 复习：指针的类型到底是什么意思？指针就是内存地址的意思

```c
int a = 10;
int *p = &a;//0x01
p + 1 //0x05
```

- 为什么是`0x05`因为`int`的字节是4个，所以走一步，其中一步的步长为4.
- 观察下面程序:

```c
int a = 10;
int *p = &a;
p + '1';
```

- 在这个地方虽然`char`类型的占位为1，但是在这个地方你要注意会隐式转换成为`int 49`所以会加上的为`49 * sizeof(int)`
- 有意义的：指针跟整数进行加减操作（每次移动N个步长）

- 无意义的：指针和整数乘除操作；指针和指针进行加减

指向不明的指针

- 野指针：指针指向的空间未分配
- 悬空指针：指针指向的空间已分配，但是被释放了

```
#include <stdio.h>
int main()
{
    int a = 10;
    int *p1 = &a;
    printf("%p\n",p1);
    printf("%d\n",*p1);

    //p2野指针
    int *p2 = p1+10;
    printf("%p\n",p2);
    printf("%d\n",*p2);
    return 0;
}
```

- 其实后面的内容是完全可以输出的，但是注意的是，这个地方的内容完全不属于本程序，因为在这个程序中没有分配这个内存空间（应用：外挂）

```
#include <stdio.h>
int main()
{
    int a = 10;
    int *p1 = &a;
    printf("%p\n",p1);
    printf("%d\n",*p1);

    //p2悬空指针
    int *p2 = method();
    return 0;
}
int* method()
{
    int num = 10;
    int *p = &num;
    return p;
}
```

- 同样的这里获得的内容，也不是真正的内容，不要随意修改。

void类型的指针

- **void类型的指针的特点：无法获得数据，无法计算，但是可以接受任何地址**
- 举个例子：

```
int a = 10;
short b = 10;

int *p1 = &a;
short *p2 = &b;

char *p3 = p1; //出现问题，不能够赋值给另外的指针
//但是你可以这么写：
void * p3 = p1;
void * p4 = p2;
// 虽然p3可以接受p1，但是不能够从p3中获得数据
printf("%d\n",*p3);//报错
printf("%p\n",p3+1);//报错
```

- 作用：

```
int main()
{
    swap(&c,&b);
}
void swap(void * p1,void * p2,int len)
{
    char* pc1 = p1;
    char* pc2 = p2;
    for (int i =0;i<len;i++)
    {
        temp = *pc1;
        *pc1 = *pc2;
        *pc2 = temp;

        pc1++;
        pc2++;
    }
}
```

- 这样通过我们进行单字节的交换进行 数字的交换
- 小细节：
- 在C语言中，`int* pc1 = p1` 和 `char* pc1 = p1` 之间的主要区别在于指针的类型和指针运算的行为。以下是详细的解释：

    1. 指针类型

        - **`int* pc1 = p1;`**：
        - `pc1` 是一个指向 `int` 类型的指针。

- 这意味着 `pc1` 假设它所指向的内存地址存储的是 `int` 类型的数据。
- `char* pc1 = p1;`:
- `pc1` 是一个指向 `char` 类型的指针。
- 这意味着 `pc1` 假设它所指向的内存地址存储的是 `char` 类型的数据。

2. 指针运算

- `int* pc1 = p1;`:
- 当你对 `int*` 类型的指针进行递增或递减操作时，指针会移动 `sizeof(int)` 个字节。
- 例如，`pc1++` 会使 `pc1` 移动 4 个字节（假设 `int` 占 4 字节）。
- `char* pc1 = p1;`:
- 当你对 `char*` 类型的指针进行递增或递减操作时，指针会移动 `sizeof(char)` 个字节。
- 例如，`pc1++` 会使 `pc1` 移动 1 个字节（因为 `char` 通常占 1 字节）。

3. 内存访问

- `int* pc1 = p1;`:
- 通过 `int*` 类型的指针访问内存时，编译器会假设该内存地址存储的是 `int` 类型的数据，并按 `int` 的大小进行读取和写入。
- `char* pc1 = p1;`:
- 通过 `char*` 类型的指针访问内存时，编译器会假设该内存地址存储的是 `char` 类型的数据，并按 `char` 的大小进行读取和写入。

二级指针和多级指针

- 指向指针的指针叫做多级指针
- 指针的数据类型和指向空间中的数据类型是一致的。
- 例如我们以二级指针为例： `int * * 指针名`

```
#include <stdio.h>
int main()
{
    int a = 10, b = 20;
    int *p = &a;
    int **pp = &p; // 定义二级指针
    *pp = &b;      // 注意这里是一次解引用，得到的是指针p的值，也是a的地址（注意不是a的数值），如果想获得a的数值要进行两次的解引用**p
    printf("%p\n", &a);
    printf("%p\n", &b);
    printf("%p\n", p);
    printf("%d\n", **pp);
    printf("%d\n", *p);
```

```
}
/*
0000006AB17FFCD4
0000006AB17FFCD0
0000006AB17FFCD0
20
20
*/
```

数组指针

- 指向数组的指针叫做数组指针

```
#include <stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50};
    int len = sizeof(arr) / sizeof(int);
    // 获取首地址
    int *p1 = arr;
    int *p2 = &arr[0];
    printf("%p\t%p\n", p1, p2);
    printf("%d\n", *p1);
    printf("%d\n", *(p1 + 1));//printf("%d\n", *(++p1));
}
```

```
00000038F9DFF9D0        00000038F9DFF9D0
10
20
```

```
//通过遍历的方式来打印整个数组
for (int i = 0; i < len; i++)
{
    printf("%d\n", *(p1++));
}
return 0;
```

- 注意这里要分别是写++p还是p++，如果是写++p表示的就是先往后移一位然后再输出对应的地址中的数据，但是如果是p++就是先输出原来的地址中的内容，然后再相加一位，这样的话就可以保证第一位的输出。

数组指针的细节

- arr参与计算的话，会退化成第一个元素的指针

```
int arr[] = {1,2,3};
int *p = arr;
int *pp = arr+1;//指向第二个元素2
```

- 特殊情况下不会退化：
    1. 进行sizeof()的运算的时候，不会退化，arr还是整体
    2. 如果写&arr的时候，arr不会退化，表示一个整体，而不是一个指针
- 举个例子：

```
#include<stdio.h>
int main()
{
    int arr[] = {1,2,3,4,5,6,7,8,9,10};
    printf("%zu\n",sizeof(arr));
    printf("%p\n",arr);
    printf("%p\n",&arr);
    printf("%p\n",arr+1);
    printf("%p\n",&arr+1);
    return 0;
}
```

- 在这个例子中，我们发现，如果是只是对地址进行输出，则两种取地址的方式的结果都是相同的
- 如果是进行计算的话，arr进行计算arr会被看做是首地址，步长则是首地址元素所对应的长度,在本题中为int:4
- &arr表示的就是整个的数组，不会进行退化，所以当我们输入&arr+1的时候就表示地址增加的长的是数据类型*数组的长度 = 40,也就是说这样会直接跳到数组的最后的结尾

遍历二维数组

- 概念：把多个小数组放到一个大的数组中

```
数组类型 arr[m][n] = //m表示的是有多少个小数组，n表示的是没一个数组的长度
{
    {1,2,3,4},
    {1,2,3,4}
}

int arr1[] = {1,2,3,4,5};
int arr2[] = {6,7,8,9,10};
int * arr[2] = {arr1,arr2};//二维数组，里面存了两个小数组的地址
```

```
#include <stdio.h>
int main()
{
```

```
    int arr1[] = {1, 2, 3};
    int arr2[] = {1, 2, 3, 4, 5};
    int arr3[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *arr[] = {arr1, arr2, arr3};
    for (int i = 1; i < 3;i++)
    {
        int len = sizeof(arr[i]) / sizeof(int);
        for (int j = 0; j < len;j++)
        {
            printf("%d\n", arr[i][j]);
        }
        printf("\n");
    }
}
```

- 注意这样的结果是错误的，原因：在`int *arr[] = {arr1,arr2,arr3}`的时候这里的`arr1`等就已经退化成为了地址了（指针了）所以在计算`sizeof(arr1)`的时候的答案就是`8bit`.

```
#include <stdio.h>
int main()
{
    int arr1[] = {1, 2, 3};
    int arr2[] = {1, 2, 3, 4, 5};
    int arr3[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *arr[] = {arr1, arr2, arr3};
    int len[] = {sizeof(arr1) / 4, sizeof(arr2) / 4, sizeof(arr3) / 4};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < len[i]; j++)
        {
            printf("%d", arr[i][j]);
        }
        printf("\n");
    }
}
```

- 利用指针来遍历二维数组：

```
如果是按照
arr[3][5] =
{
    {1,2,3,4},
    {2,3,4,5}
};这样来定义的话：
数据类型 * 指针名 = arr
int (*p)[5] = arr;
```

- 但是这个地方要注意arr+1表示的就是加上一个数组arr的总长度，不在是表示一维数组的时候arr+1表示的是加上一个字节的长度。
- 可以这么记忆：就是在使用arr+1的时候表示的是移动内部一个元素的距离。

```
int (*p)[5] = arr;
for (int i = 0;i<3;i++)
{
    for(int j=0;j<5;j++)
    {
        printf("%d ",*(*p+j));
    }
    p++;
}
```

- *p表示的是内部的arr1，而在这里arr1+1表示的就是单个数字的移动长度。然后再取值获得数值，这样就可以遍历整个二维数组了
- p++就表示的就是移动一行，移动的长度为sizeof(int(N))

```
如果是`int* arr = {arr1,arr2};`来定义的话:
数据类型 * 指针名称 = arr;
  int*  *  p      = arr;
```c
#include <stdio.h>
int main()
{
    int arr1[] = {1, 2, 3};
    int arr2[] = {1, 2, 3, 4, 5};
    int arr3[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *arr[] = {arr1, arr2, arr3};
    int len[] = {3, 5, 9};
    int **p = arr;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < len[i]; j++)
        {
            printf("%d ", *(*p + j));
        }
        p++;
        printf("\n");
    }
}
```

- 强调一下：数组指针和指针数组
  - 数组指针：数组的指针，用来方便操作数组的内容int *p = arr;步长为4``int (*p)[5] = &arr(arr);步长为20
  - 指针数组：存放指针的数组int* arr = {arr1,arr2};``int *p[5](一定要和上面的区别开来)

函数指针
```

格式：返回值类型（*指针名）（形参列表）

```
#include <stdio.h>
void method1();
int method2(int num1, int num2);
int main()
{
    void (*p1)() = method1;
    int (*p2)(int, int) = method2;
}
void method1()
{
    printf("method1\n");
}
int method2(int num1, int num2)
{
    return num1 + num2;
}
```

- 指针定义的小技巧:直接将函数定义复制到前面，然后将函数名改为(*指针名)，并去掉后面的传入参数的名称，就写完了

```
int method2(int num1,int num2)
int (*p1)  (int     ,int     ) = method2

//函数指针数组
int (*arr[4])(int, int) = {add,····};
//这里注意第一个int表示返回值为int，(*arr[4])表示的是arr是一个长度为4的数组，其中每一个元素都是一个指向函数的指针，这些函数接受两个int类型的参数
```

作用

- 利用函数指针来调用函数

```
#include <stdio.h>
void method1();
int method2(int num1, int num2);
int main()
{
    void (*p1)() = method1;
    int (*p2)(int, int) = method2;
    p1();
    printf("%d\n",p2(10,20));
}
void method1()
{
    printf("method1\n");
```

```
}
int method2(int num1, int num2)
{
    return num1 + num2;
}
```

```
#include<stdio.h>
int add(int num1, int num2);
int minus(int num1, int num2);
int multiply(int num1, int num2);
int devide(int num1, int num2);
int main()
{
    /*
        定义加减乘除四个函数
        用户键盘输入三个数
        前两个表示参与计算的数字
        第三个表示调用的函数
        1：加法
        2：减法
        3：乘法
        4：除法
    */
    //键盘录入三个数字
    int num1, num2, flag;
    scanf("%d %d %d", &num1, &num2, &flag);
    //定义一个数组来装四个函数的指针
    int (*arr[4])(int, int) = {add, minus, multiply, devide};
    printf("%d\n", (arr[flag - 1])(num1, num2));
}
int add(int num1,int num2)
{
    return num1 + num2;
}
int minus(int num1,int num2)
{
    return num1 - num2;
}
int multiply(int num1,int num2)
{
    return num1 * num2;
}
int devide(int num1,int num2)
{
    return num1 / num2;
}
```