第三章：指针

概述

- 指针的实质是变量，但用途不同，指针变量存的是地址
- 指针的出现是对间接寻址的封装
- 高级语言JAVA，C#没有指针，语言本身封装了指针
- 指针使用三部曲：**定义**指针变量、为指针变量**赋值**（不赋值存随机数字）、**解引用** (*p=555)

符号问题

- 定义指针变量时，*位置，加不加空格，无差别，编译器忽略空格

示例

```
int *p1,p2;
等同于 int* p1,p2;//p1指针，p2整数
int *p1,*p2;//两个都是指针
```

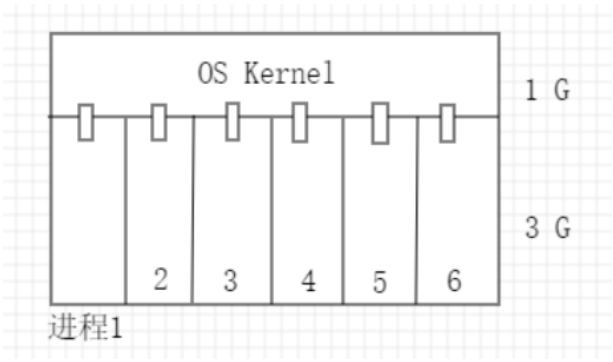
- 变量做左值：表示变量对应的内存空间
- 变量做右值：表示变量对应内存空间存的数字

示例

```
a=3,b=5;
a=b;//a的空间存上5
b=a;//a的空间内的值3存到b的空间里
```

野指针（指针指向位置不确定）

- 原因：指针变量定义后未初始化、也没赋值，对指针解引用
- **野指针三种情况：**
  - **指向OS不允许访问的地址**（如：内核空间）→触发段错误
    - 每个进程都运行在虚拟地址，认为自己有全部的空间，每个进程可以寻址的空间里有一段属于os，不可访问



- **指向可用、但没特别意义的空间**（如：曾使用过但已经不用的栈空间）→程序运行不会出错，但其实有问题
- **指向可用、但被其它程序使用的空间**→变量被改变，一般会导致数据损害、程序崩溃（危害最大）

- 指针变量若是局部变量，分配到的地址是栈上一次使用时被赋予的值（栈使用完不擦除）
- **避免野指针（平时不用保证其为NULL，真正要用时与可用地址绑定，用完再赋为NULL）**
  - **定义指针时，初始化为NULL**
  - **指针解引用之前，先判断是不是NULL（可能中间隔了很多行代码，忘记指针是否赋值，先判断更为保险）**
  - **指针使用之前，将其绑定给一个可用的地址空间（不确定可以访问，就不要解引用）**
  - **指针用完后，赋值为NULL**
- NULL是什么？实质是0，指针指向NULL，就是让指针指向0地址处

NULL的定义

```
#ifdef _cplusplus /*有时候C和C++混合使用，\
C++编译环境中预先定义了一个宏，程序中用该宏判定编译环境 */
#define NULL 0
#else
#define NULL (void*)0/*C语言中NULL为强制转成void*类型的0，本质是0，但不是数字，是内存
#endif

//定义不同的原因：C语言中会做严格的类型检查
int *p;
p=0;错误
p=(int*)0正确
p=(void*)0正确
```

  - 为什么指向0地址处？
    - 0地址默认是特殊地址，放野指针

- 0地址在一般os中不可访问，这是野指针错误中最好的结果（触发段错误，但不会出大问题）
- **一般判断野指针写成 if (NULL! =p)，避免需要==，但写成=，编译器不报错，但很难检查**

const关键字：修饰变量，表示变量为常量

- **修饰指针的四种形式**

```
const int *p;//const修饰int，p本身不是const,*p（p指向的变量）是const
int const *p;//const修饰*p，p本身不是const,*p（p指向的变量）是const
int * const p;//const修饰p，p是const，*p（p指向的变量）不是const
const int * const p;//p本身是const，*p（p指向的变量）也是const
```

  - 理解指针变量涉及指针变量p，p指向的变量（*p），一个const只能修饰一个变量，关键在于搞清楚const修饰谁
  - const往后看紧挨着p，说明p是const，没有紧挨，说明p指向变量为const
- const修饰的变量可以改吗?
  - const使用更多是传递一种信息，告诉他人没必要修改，但gcc环境下，const修饰的变量可以改，C语言没有严格要求

由指针引起的const变量被修改

```
const int a = 5;//变量a是const，理论上不可以改
int *p;
p=&a;//p指向a的地址
*p=6;//a地址处赋值为6

/* 原理：gcc中，const通过编译时执行检查实现，把const类型常量a放在data段
编译器不认为p是const类型，可通过指针p修改a*/
```

深入理解数组

- 编译器角度：数组变量也是变量，变量的本质是地址
- 内存角度：数组变量=一次分配多个变量，且多个变量在内存中依次相连
- 符号角度：int a[10]

	左值含义	右值含义

a	不可作左值（因为a代表整个数组所有空间，而数组操作要一个个进行）	**数组首元素首地址，等同于&a[0]**
a[0]	数组首元素对应的空间，连续四字节	数组首元素的值
&a	不可作左值（因为它是常量，每次执行程序就分配好）	**整个数组首地址**
&a[0]	数组首元素对应内存空间	**数组首元素首地址**

- 数组两种访问方式：数组、指针， a[3] 相当于 *(a+3)
- sizeof（a），数组名不作左值，也不作右值，返回数组占用内存空间大小
- 常用如下方式表示数组元素个数（改动数组大小，下面的代码自动适应，不用修改）：int a[47]; int b=sizeof(a)/sizeof(a[0]);
- 区分 a 和 &a

> 示例

```
int *p;int a[5];
p=a 正确，首元素首地址
p=&a 错误，数组首地址
```

指针与强制类型转换

- 变量数据类型的本质：变量的存储方式
  - int、char、short 属于整型，存储方式相同，彼此兼容
  - float、double 存储方式不同，彼此不兼容，和整型更不兼容
  - 另：printf会按照本身数据类型存入数据，按照format对应的类型解析对应的内存空间提取数据
- 指针变量数据类型
  - 不同类型指针都是按照地址的方式解析，32位系统中，只要是指针都占4字节
  - 强制类型转换主要的目的是为了骗过编译器
  - 指针变量的强制类型转换

> 示例

```
int *-> char* //类型兼容，但int范围比char范围大，在char可表示范围内互转不会出错，超出会出错
int *-> float* //解析方式不兼容，转换后访问一定会出错
```

指针与函数

- 指针与函数传参
  - 普通变量作为形参：实参与形参在独立的内存空间，值相等，但是是不同变量（传值调用，相当于实参做右值、形参做左值）
  - 数组作为形参：传递整个数组的首地址（传址调用），[]可有可无，因为根本不会传递数组长度，若想传入数组大小，需要加一个参数

    示例

    ```
    void func1(int a[])//int a[]等同于int*a
    {
      sizeof(a);
    }

    int main(void)
    {
      int a[20];
      func(a);//返回的是4，直接调用返回的是整个数组占用空间
    }
    ```

  - 指针作为形参：等同于数组作为形参
  - 结构体变量作为形参：结构体传参类似传普通变量；结构体很大，直接传结构体效率会低，一般传结构体指针
  - 传值调用与传址调用
    - 传值调用：实参的数据传给形参，操作的是形参，实参值不改变
    - 传址调用：实参地址传给形参，操作的是实参，实参值改变
- 函数的本质
  - 函数是一台加工机器，形参列表或全局变量是输入部分，返回值是输出部分
    - 全局变量传参（尽量少用）、形参列表传参都有使用
    - C语言中全局变量尽量少用，形参传参更多，可以实现模块化编程
    - 函数参数太多，打包成结构体，传结构体的指针进去（linux内核中常用）

- 函数名：整个函数代码段的首地址，实际上是指针常量

- 输入型参数和输出型参数
  - 如何看出函数参数做输入还是输出?
    - 传参为普通变量（非指针）→输入型参数
    - 传参为指针→输入或输出
      - 函数内部只需要读取该参数，不需要修改，在前面加const→输入型参数

        示例

        ```
        void func(const int *p);//程序中不可改变指针指向的内容
        int main(void)
        {
          int *p="linux";
          func(p);//合适，"linux" 字符串存在代码段，本身不可改变，同时函数声明中也暗示p指向的内容不可改变
        }
        ```

      - 传参为指针，且前面无const→输出型参数
  - 函数向外部返回多个值的方法
    - 参数用作返回值，返回值返回0或者负数，用来表示程序执行结果对还是错（典型的linux风格函数）

其它

- 在32位编译器下，使用%p打印指针变量，显示32位的地址（16进制）；64位编译器下，使用%p打印指针变量，显示64位的地址（16进制）
- nil表示指针不指向任何一个对象，NULL是一个宏