指针

指针指向一块地址,指针类型的值可以是**指向对象**如`(int *)`的指针,**函数指针**如`int (*)(int)`

或某个类型的**空指针**.

对象指针

指针的声明方式如下

```
int *p; // 在某个类型后加*标记p是一个指针类型
```

如果在一条语句定义**多个变量**会有差异

```
int *p1,p2; // p1是指向int类型的指针,但是p2只是一个普通的int对象
int *p3,*p4; // p3 p4都为指向int类型的指针
```

获取对象的地址

指针用于存放某个对象的地址,想要获得某个对象的地址,需要通过**内建运算符(C++自带就有的运算符)&**获得

```
int x = 1;
std::cout << &x; // 会输出一个16进制数,为x的地址
int *p = &x; // 把x的地址存到指针p中
```

通过指针访问对象

指针虽然指向了一个对象,但是只是指向了它的地址,想要通过地址获得对应的变量,需要用内建运算符*来获取,这个操作也叫做**解引用**

```
int x = 1;
int *p = &x;
std::cout <<  p; // 直接输出p输出的是x的地址
std::cout << *p; // 输出x的值1
```

解引用获得的不仅仅是对象的值,解引用后的指针就是指针指向地址的对象本身.

```
int x = 1;
int *p = &x;
*p = 2;
std::cout << "x = " << x << " *p = " << *p;
```

输出结果

```
x = 2 *p = 2
```

指针初始化

```
int *p;
```

这个语句中,定义了一个`int`类型的指针`p`但是并没有进行初始化,此时的p的指向是不确定,对于这种指向不确定的指针也叫做**野指针**,对**野指针解引用是未定义行为**

```
int *p = 0; // 使指针p不指向任何对象
```

在一些地方也会看见

```
int *p = NULL; // 来标识p不指向任何对象
```

但是其实NULL是一个宏,在标准库中可以看到这样一条语句

```
#define NULL 0
```

所以上述两种设置为空指针的写法完全等价,但是在**C++11**引入了一个新的字面量`nullptr`用来表示空指针

```
int *p = nullptr;
```

对空指针解引用也是未定义行为

指针的赋值

指针虽然可以通过解引用的方式来标识其它对象,但是指针自己本身也是一个对象,所以可以被赋值,或者被二级指针(指向指针的指针)指向自己.

```
int a = 1, b = 0;
int *p1 = &a, *p2 = &b; // 分别让P1 ,p2 指向 a,b的地址
p1 = p2; // 把p2的值赋值给p1也就是说 p1此时不再指向a的地址,而是跟p2一起共同指向b
(*p1)++; // 使得b的值增加1
std::cout << *p2; // 虽然修改的是*p1,但是p1,p2都是指向b所以*p2为1
```

指向指针的指针

```
int x = 0;
int *p = &x;
int **p2 = &p; // &p获得的p自己本身的地址,而不是p指向的x的地址,**代表这是一个指向int*的指针,也就是指向int指针的指针
(**p2)++;   // *p2得到的值是p,此时p的值只是x的地址,**p2两次解引用相当于*p,也就是得到变量x
std::cout << x; // 输出结果为1
```

指向结构体的指针

指向结构体的指针与普通指针并无差异,只是多了一种用内建运算符`->`访问成员的方式,假设有以下结构体A

```
struct A {
    int a;
};
```

```
A test;
test.a = 1;
//定义一个结构体变量test
A *p = &test; //定义一个指向A的指针p,指向test的地址
```

如果此时我想通过`p`来获得`test.a`有两种方式

```
std::cout << (*p).a << "\n"; //解引用p得到test,再.a得到test.a
```

```
std::cout << p->a << "\n";
//p->a这个表达式完全等价(*p).a;与第一种方式解释完全相同
```

对指针的理解

当我们输出一个指针指向的地址的时候,他是一个字节的16进制数,但如果我指向一个4字节的`int`类型时,输出地址时却还是一个字节,那么剩下的三字节去哪了?根据我目前的理解是,所谓指针就是**类型+地址**,指针指向的地址其实是对象的首地址,当你解引用时,由于你的指针类型是**int***,所以他会根据`int`的大小,把从首地址开始算,连续取四个字节的内存一起拿出来,从而得到完整的`int`对象,如果是`long long`他就会从他指向的首地址开始算取出八字节.所以在一些代码中可以看到譬如从一种指针的类型强制转换到另外一种指针的类型后再解引用,指针指向的对象首地址从来没变过,转换的只是解引用需要取出几个字节的大小