第二次习题课

习题讲解

谢强

中国科学技术大学信息学院

2021 年 11 月 23 日

1 函数相关
- 参数传入
- 函数的递归调用

2 指针
- 指针前瞻
- 地址和溢出

C 语言的传参方式

在 C 语言中，理论上只存在一种传参方式：传值引用，例如：

传值

```
1 void minus(int x){
2     x=-x;
3 }
4 //调用
5 int a=3;
6 minus(a);//minus(3);
```

上面的写法中，在调用 minus(a) 时，相当于将 a 的值复制了一份给了新的变量 x，它们是完全无关的变量，因此最后 a 的值不会变。

传指针

我们常说的传指针等，其实本质上也是传值引用。

传值

```
1 void minus(int *x){
2     *x=-*x;
3     x=-x;
4 }
5 minus(&a);
```

这里的 x 是一个指针变量，里面存的值是地址；在 64 位下，可以认为是一个 longlong 整数。调用 minus(&a) 的时候，新开了一个 x 变量，存入了 a 的地址，因此在第一句的时候，我们直接修改了 a 所在的内存。

从快速幂说起

大家最早接触递归这个概念，可能是从高中讲递归公式说起的。

$$a^n = \begin{cases} a^{n-1} \times a & n > 0 \\ 1 & n = 0 \end{cases} \tag{1}$$

我们发现，事实上递归的重点在于，求解同一个问题，仅仅只是换了一个参数。因此，我们可以自己调用自己，因为问题的形式是一样的，只是参数不一样。

递归式快速幂

这里有一个简单的优化

$$a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & n \equiv 0 mod 2 \\ a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}} \times a & n \equiv 1 mod 2 \\ 1 & n = 0 \end{cases} \quad (2)$$

这就是递归式的快速幂。之前实验课上讲过非递归的，相当于按位加权；递归式的相当于秦九韶展开。
同样地，这里的递归也满足上面的性质。

快速幂代码

递归式快速幂代码实现

```
1 int quick_pow(int a,int n){
2     if(!n) return 1;
3     int ans=pow(a,n/2);
4     if(n&1) return ans*ans*a;
5     return ans*aans;
6 }
```

这就是根据上面的递推公式得出的代码。

误区

需要认识到的：

- 递归不一定存在一个线性减小的变量
- 递归甚至不一定存在一个减小的变量
- 递归不等价于解决递推公式的求值问题

递归的真正含义，是解决一套**形式相同、参数不同、出口可达**的子问题。
类似于数学归纳法，只要每一步都正确、出口也正确，那么整个递归过程就正确。

二进制转换

递归转二进制

```
1 void dec2bin(int a){
2     if(!a) return;
3     dec2bin(a/2);
4     printf("%d",a&1);
5 }
```

之前有同学这么写，原理在于：二进制转换是从高位到低位的，因此我们可以总结递归过程：

1. 输出除了最低位以外的二进制
2. 输出最低位

同样发现，第一个子任务和主任务的形式相同，仅仅是参数不同。

Eculid 辗转相除法

递归实现

```
1 void gcd(int a,int b){
2     if(!b) return a;
3     return gcd(b,a%b);
4 }
```

同样地，满足形式相同、参数不同原则。辗转的理由在于，递归出口可达性。

概览

程序的虚拟内存空间是一个线性结构，或者可以类比成是一个很大的字节数组。
是数组，那么就有下标，也就是常说的**地址 (address)**
下标可以用变量来存，地址也可以用变量存，这样的变量就叫做**指针 (pointer)**
(广义来说，存下标的变量都可以叫指针，但是是逻辑上的；课程上的指针一般指专有名词)
指针的大小由机器的配置决定，常说的 32 位和 64 位。

指针的定义

然而，内存显然不能只用字节来划分。
比如 char 型的信息，一个占一字节；但是 int 型的，一个占 4 字节。
因此，我们要对指针进行类型的划分；不同类型的指针，移动的单位不同，也最好不要混用。

指针的定义

```
1 char *a;
2 int *b;
3 short (*c)[10];
```

指针的使用

指针初始值

```
1 char ta;
2 char *a=&ta;
3 int tb;
4 char *b=&tb
```

就像我们不能随便拿一个值来当数组下标一样，指针也应该有个初始值。取得变量的地址需要用 `&` 符号。

指针的使用

指针取值

```
1 char a='3';
2 char *pa=&a;
3 *pa='1';
```

用 * 号来访问地址所指向的内存

指针的运算

主要是加和减

```
1 int *a;
2 char *b;
3 short (*c)[10];
4 a+1 b+1 c+1
```

在内存中，a 加了 4，b 加了 1，c 加了 20

指针和数组

用数组为指针赋值

```
1 int a[10]={0,1,2,3};
2 int *p=a;//*p=&a[0]
3 p[3]=4;
4 //x[a]=*(x+a)
```

数组的名称，是一个数组元素型的地址，指向数组首地址。
最后一行的等式非常重要。根据最后一行等式，我们可以将 `p[3]` 写成 `3[p]`

指针的读法

形式化表述

```
1 char a[10][20][30]; >>> char aa[20][30];
2    char (*pa)[20][30]=&aa; pa=a[0];
3 int (*b)[10];  >>> int bb[10]; b=&bb;
4 int *(c[10]);  >>> int *cc; c[0]=cc;
```

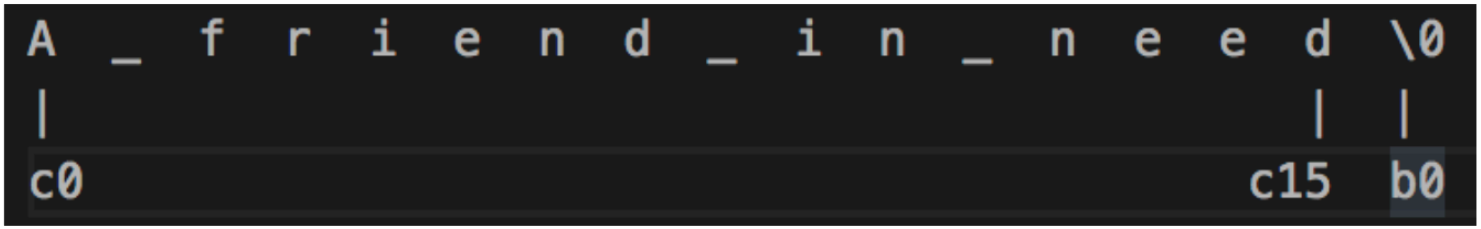
根据优先级，由内而外消解。
只判断是否正确，不要太死磕含义。
最重要的是判断 `sizeof`

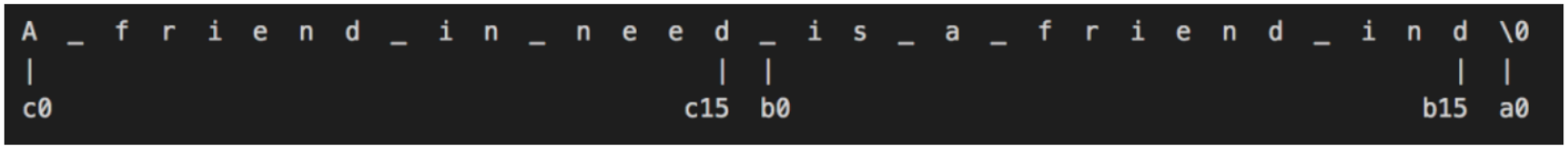
形式化表述

```
1 char a[16];
2 char b[16];
3 char c[16];
4 scanf("%s",c);//不用加&
5 gets(b);
6 gets(a);//只传入地址
7 puts(c);
```

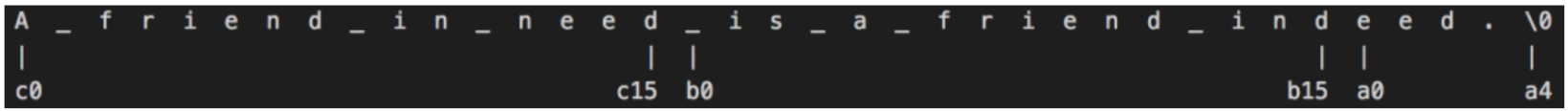
都输入长度为 16 的字符，会导致溢出，输出整个字符串

图示





道理和上面一样，但是从 b0 开始写入，导致第一次输入的\0 被覆盖了。



同样的，从 a0 输入，导致第二次的\0 又被覆盖了。

sizeof

这里需要有一个概念就是 c 语言判断完整的字符串一定是**从某个地址开始，直到读到\0，才认为字符串结束**。而且默认情况下，这是唯一标准，**而不是所谓的根据什么“字符串长度”来判断结束。**
为什么呢？同样的，包括 printf %s、puts 等等输出方式，同样把输入的参数当作单纯的地址来看待，并不知道它是一个数组，更不知道它的数组长度。因此，\0 成了唯一的结束判据。

此时我们 puts(c),就是从 c0 开始读取，读到第一个\0 为止，也就是输出从 c0 到 a4 的一整句话。

这里可能就有人有疑问了，sizeof(c)这个函数凭什么能知道 c 的长度呢？难道 c 在这里不是作为一个地址输入到函数里的吗？这里需要科普到是，**sizeof()不是一个函数**。

sizeof()是一个保留的 c 关键字，相当于 if(),while()等等。作为关键字，事实上就是编译器在分析程序语法的时候分析的东西，而能知道变量占了多少空间的，恰恰就是关键字。

感谢

Thank You for Your Attention!