浙江大學

Project 3: 表达式计算

实验报告

课程名称：	C语言程序设计专题
学　　院：	数学科学学院
专　　业：	数学与应用数学（强基计划）
姓　　名：	黄文翀
学　　号：	3200100006
指导老师：	翁恺

Project 3: 表达式计算 实验报告

黄文翀，3200100006

浙江大学 强基数学2001班

摘　要： 表达式计算是任何一个高级语言需要实现的基本功能，对于一个复杂的语言系统，其运算符通常多达几十种，运算优先级通常分为十余级。这要求编译器能够对表达式做出正确的分析，并且根据优先级计算出正确的结果。本文从四则运算以及取模、小括号几种常用运算出发，研究表达式计算的实现。本文最后进行拓展，给出多种运算符、多种优先级的复杂情况下，表达式计算的实现方法，供读者参考。

关键词： 计算机；C语言；递归；栈；

1　实验基本要求

实现一个表达式计算工具，支持加、减、乘、整除、取模、小括号。所有的运算都是整数运算，并且运算中间值都不会超过`int`的表示范围。

2　基本思路与过程描述

2.1　基本思路

首先，不考虑小括号，那么题目要求我们支持的运算只有两个优先级。因此，基本方法是，延缓一次计算，如果下一个运算符的优先级与上一次相同或更低，则执行延缓的那一次计算；否则先执行本次计算，上一个运算继续延缓。现在考虑小括号，这也不是什么困难的问题，递归计算即可，遇到左括号则进入递归，遇到右括号则跳出递归，并将计算结果返回给上一层递归，这样对于上一层递归而言，一对括号内不管是什么东西，最后它得到的都是一个运算结果，至于运算过程是由它的下一层递归处理的。

2.2　过程描述

基于上述基本思路，可以很清晰地设计出一个程序，但马上遇到了一个问题，就是减号与负号的判定问题。由于减号与符号共用一个符号，但减号是双目运算，而符号是单目运算，因此必须进行特判。特判方法很简单，当出现符号`-`时，如果它的前一个字符是数字或者`)`，那么它就是减号，否则它就是负号。这样就需要一个额外的变量对前一个字符进行存储。这个问题是本次实验唯一的细节问题，解决此问题后程序即直接通过测试。

3　代码解释

3.1　单次运算

```
int calc(int a, int b, char op){
    switch (op){
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
        case '%': return a % b;
    }
}
```

这是进行单次运算的函数，无需解释

3.2 判断优先级

```
int lower(char op1, char op2){
    return (op1 == '+' || op1 == '-') && (op2 == '*' || op2 == '/' || op2 == '%');
}
```

此函数用于判断运算符op1的优先级是否比op2更低。由于优先级只有两个，所以op1优先级更低的情况只能是：op1是加减，op2是乘除模。

3.3 主递归函数

```
int work(){
    int res = 0, lastnum = 0;
    char lastop = '\0', op = '\0';
    static char lastch = '\0';
    char ch = getchar();
    while(!isspace(ch)){
        int x = 0, needcalc = 0;
        if(isdigit(ch) || ch == '-' && !isdigit(lastch) && lastch != ')'){
            int fg = 1;
            if(ch == '-') fg = -1, lastch = ch, ch = getchar();
            while(isdigit(ch)) x = x * 10 + (ch - '0'), lastch = ch, ch = getchar();
            x *= fg;
            needcalc = 1;
        } else {
            if(ch == '(') x = work(), needcalc = 1;
            else if(ch == ')') break;
            else {
                if(!lastop) lastop = ch;
                else op = ch;
            }
            lastch = ch, ch = getchar();
        }

        if(needcalc){
            if(!lastop) res = x;
            else if(!op) lastnum = x;
            else{
                if(lower(lastop, op)){
                    lastnum = calc(lastnum, x, op);
                } else {
                    res = calc(res, lastnum, lastop);
                    lastnum = x;
                    lastop = op;
                }
                op = '\0';
            }
        }
    }
    if(lastop) res = calc(res, lastnum, lastop);
    return res;
}
```

`res`、`lastnum`、`lastop`是用来进行延缓计算的。程序读取到一个符号时，先进行判断，如果尚未存储延缓计算符号，则将这个符号存储为`lastop`，否则存储为`op`。程序读取到一个数字时，也进行判断，如果尚未存储延缓计算符号，则将其存储为`res`；如果已经存储了延缓计算符号，但没有存储当前运算符号，说明读取到的是延缓符号`lastop`紧跟着的数字，根据延缓法则应当将其存储为`lastnum`；如果两个运算符都有，那么如果延缓符号的优先级更低，先进行本次运算，结果给`lastnum`，而延缓运算符继续延缓，否则先进行延缓运算，结

果存储为`res`，同时更新`lastnum`与`lastop`，将当前运算进行延缓。递归函数最后应该进行判断，将没有处理的延缓运算处理掉，并将最后的结果`res`返回。

另外，当读取到`(`时，进入下一层递归，将下一层递归的运算结果作为读取的一个数字，进行上述处理。读取到`)`时应当结束本次递归的计算，并将结果返回。

4 更加复杂的表达式计算

借助“延缓计算”的思路，我们完全可以实现一个更加复杂的表达式计算工具。由于具有多个优先级，延缓的运算符将不止一个，当读取到一个新的运算时，从后往前依次将延缓运算的优先级与当前运算进行比较，将不低于当前运算的延缓运算进行运算，然后再将当前运算加入延缓运算的存储区中。

上述思路中，“从后往前”这个词很重要，也就是说，越后出现的延缓运算，应当越先与当前运算进行比较。这也就明示我们需要使用一种数据结构——栈。我们需要两个栈，一个用于存储延缓运算的符号，一个用于存储延缓运算的中间值。

接下来首先介绍栈在C语言中的实现，然后用一个具体的例子实现复杂的表达式计算。

4.1 栈在C语言中的实现

4.1.1 栈的概述

栈是一种“后进先出”的数据结构。当加入一个新数据时，它会进入到栈的顶端，而弹出时将优先弹出栈顶端的元素。所以我们需要实现`push`、`top`、`pop`三个函数，分别用于压入新元素、访问顶端元素、弹出顶端元素。

4.1.2 定义栈结构

```
struct stack{
    int val;
    struct stack* next;
};
```

我们采用链表的方式实现栈，以上是其结构定义。

4.1.3 push函数

```
struct stack* push(struct stack* st, int x){
    struct stack* p = (struct stack*)malloc(sizeof(struct stack));
    p -> val = x;
    p -> next = st;
    return p;
}
```

新建节点，存储`x`值，并将`next`指针指向原来的栈顶，这个新节点就是栈的新顶，将这个新顶返回。

4.1.4 top函数

```
int top(struct stack *st){
    return st -> val;
}
```

直接返回栈顶节点存的值即可。

4.1.5 pop函数

```
struct stack* pop(struct stack* st){
    struct stack* p = st -> next;
    free(st);
    return p;
}
```

用指针`p`指向栈顶的`next`所指节点，将栈的原顶释放，`p`指向的节点就是栈的新顶，将新顶返回。

4.2 复杂表达式计算实例

4.2.1 运算与优先级列表

优先级	运算符
1	()
2	-(负号)、~(按位取反)
3	*、/、%
4	+、-(减号)
5	<(左移)、>(右移)
6	&(按位与)
7	^(按位异或)
8	|(按位或)

由于仅出于演示目的，这里不对C语言中的所有运算符进行支持，只考虑以上几种运算。注意这里为了方便，将左移与右移的运算符简化为了单个符号`<`、`>`，注意与C语言中的表达方法`<<`、`>>`进行区别。

4.2.2 单次运算

```
int calc(int a, int b, char op){
    switch(op){
        case '*': return a * b;
        case '/': return a / b;
        case '%': return a % b;
        case '+': return a + b;
        case '-': return a - b;
        case '<': return a << b;
        case '>': return a >> b;
        case '&': return a & b;
        case '^': return a ^ b;
        case '|': return a | b;
    }
}
```

这里只实现单次的双目运算。单目运算需要进行特判处理。

4.2.3 判断优先级

```
int priority(char op){
    switch(op){
        case '~': return 2;
        case '*': case '/': case '%': return 3;
        case '+': case '-': return 4;
        case '<': case '>': return 5;
        case '&': return 6;
        case '^': return 7;
        case '|': return 8;
    }
}
```

对运算符op，返回其优先级代号，代号较小的运算符优先级高。

4.2.4 进行延缓运算

```
void deal(struct stack** op, struct stack** num, int pri){
    struct stack* opt = *op;
    struct stack* numt = *num;
    while(opt && priority(top(opt)) <= pri){
        char p = top(opt); opt = pop(opt);
        if(p == '~'){
            int a = top(numt); numt = pop(numt);
            numt = push(numt, ~a);
        } else {
            int b = top(numt); numt = pop(numt);
            int a = top(numt); numt = pop(numt);
            numt = push(numt, calc(a, b, p));
        }
    }
    *op = opt;
    *num = numt;
}
```

当读取到新的运算时，要从后往前依次将优先级高于它的延缓运算处理掉。这里将运算符栈与中间值栈的地址传入，再将新运算的优先级代号传入，即可处理掉所有比它优先级高的运算。

4.2.5 主递归函数

```
int work(){
    static char lastch = '\0';
    char ch = getchar();
    struct stack* op = NULL;
    struct stack* num = NULL;
    while(!isspace(ch)){
        if(isdigit(ch) || ch == '-' && !isdigit(lastch) && lastch != ')'){
            int x = 0, fg = 1;
            if(ch == '-') fg = -1, lastch = ch, ch = getchar();
            while(isdigit(ch)) x = x * 10 + (ch - '0'), lastch = ch, ch = getchar();
            num = push(num, x * fg);
        } else {
            if(ch == '(') num = push(num, work());
            else if(ch == ')') break;
            else{
                deal(&op, &num, priority(ch));
                op = push(op, ch);
            }
            lastch = ch, ch = getchar();
        }
    }
    deal(&op, &num, 8);
```

```
        return top(num);
    }
```

结构与简单版本是相似的。只不过由于栈的存在，不需要再做复杂的判断，读取到新运算直接先调用`deal`函数，再将其加入延缓栈即可。最后中间值栈中一定会剩下恰好一个数字，即为运算结果。

4.2.6 测试

由于本程序是向下兼容简单版本的，所以可以提交PTA测试，完美一遍通过。

样例输入一：`30/(10^29)&15+20*2^16|12*23>6+12`

程序输出：`17`

样例输入二：`(30/((5<1)^29)&15+20*-2^16|~12*2+3)>1+(12+2*-6)`

程序输出：`-4`

使用C语言运算上述两个表达式（将符号`>`改为`>>`，`<`改为`<<`），结果相同

5 实验体会与心得

表达式计算是任何一个高级语言需要实现的基本功能，对于一个复杂的语言系统，其运算符通常多达几十种，运算优先级通常分为十余级。这要求编译器能够对表达式做出正确的分析，并且根据优先级计算出正确的结果。本次实验要求实现整数的四则运算以及取模、小括号，让我们对优先级的处理、括号递归处理有了基本的了解。此外，利用栈这个数据结构，就可以把思路推广到更加复杂的表达式计算中，实现多优先级的处理。

我感到最为麻烦的是单目运算的处理，比如负号和按位取反符号，这些都要进行特判，不像双目运算可以用一个函数统一处理。这样想来，C语言中各种各样的运算，甚至还有三目运算符，要全部实现难度是非常之高的。另一件麻烦的事就是样例数据的选择，在运算符多起来之后很难找到合适的长表达式进行测试，一不小心就会超出`int`的范围。如果能够结合高精度处理，实现高精度下的复杂表达式运算，那不管是实用价值还是测试数据的选择都将会方便很多，但这也是比较困难的任务。