大作业说明文档

Author: Shuang Luo

Number: 2014010300

1. 选择的题目：表达式计算器（ExpressionCalculator）
2. 程序核心算法：

程序中定义了“简单表达式(SimpleExpression)”的概念。

程序中的“简单表达式(SimpleExpression)”指的是“只含基本运算符（+、-、\*、/、^、!）和操作数，不含函数和括号的表达式”。程序以考虑运算优先级为基本出发点，将表达式从内层到外层层层解析，直到获得“简单表达式”，最后再解析“简单表达式”获得最终结果。

算法流程如下：

1. 取代常量
2. 格式化表达式（程序允许用户输入空格，因此需要对表达式进行去空格处理）

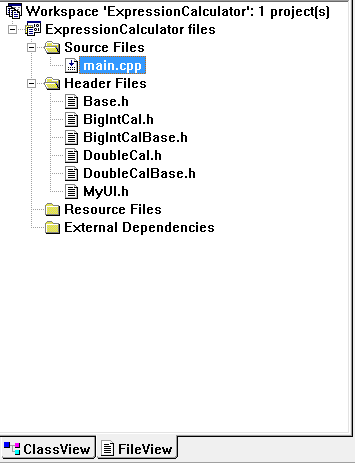
2、判断表达式是否是简单表达式，是则直接解析，输出结果，结束运算；否则进入以下步骤：

3、计算最内层括号里面的内容（最内层括号里面的内容是“简单表达式”）

4、判断最内层括号是函数括号还是一般括号，若是前者则计算并以值代之，若是后者则直接去掉，返回步骤2。

另外，程序中还实现了异常的捕捉和抛出、处理相关功能，有较完善的提示功能。

3. 项目结构：



说明：

Main.cpp(主程序文件)

Base.h(头文件，包含一些基本的函数和常量)

BigIntCalBase.h(头文件，包含一些大数运算中的一些辅助函数)

BigIntCal.h(头文件，包含一些大数运算中的核心函数)

DoubleCalBase.h(头文件，包含一些浮点运算中的一些辅助函数)

DoubleCal.h(头文件，包含一些浮点运算中的核心函数)

MyUI.h()(头文件，程序用户界面相关)

3. 关键代码：

（1）主程序：

#include"MyUI.h"

void main()

{

ShowDocumentation();

while(1)

{

ShowNavigation();

int flag=0;

setbuf(stdin,NULL);

scanf("%d",&flag);

if(flag==0)break;

else if(flag==1)

{

DoubleCalculation();

}

else if(flag==2)

{

BigIntCalculation();

}

else if(flag==3)

{

PreferenceSettings();

}

else if(flag==4)

{

ShowHistory();

}

else

{

printf("\n请按导航输入菜单!\n\n");

}

}

}

（2）相关子程序：

大数运算核心程序文件：

#ifndef DoubleCal

#define DoubleCal

#include"DoubleCalBase.h"

#include<math.h>

#define PI 3.1415926

#define E 2.71828

//全局变量：原始表达式

char OriginalExpression[500];

//全局变量：伴随解析过程的表达式

char Expression[500];

//全局变量：精度

int DoubleCalPrecision=3;

//全局变量：弧度标志

bool IsRad=true;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*本头文件中包含的所有函数声明如下\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//得到表达式

void GetExpression();

//格式化（去掉表达式中的空格）

void Format(char\* expression);

//把表达式中的常量替换成数

void ReplaceConstSym(char\* expression);

//解析字符串形式的整数:start和end分别为str中需要解析的起始位置和终止位置

long ParseInt(char\* str,int start,int end);

//解析字符串形式的小数:start和end分别为str中需要解析的起始位置和终止位置

double ParseDouble(char\* str,int start,int end);

//计算表达式中的简单函数，并用值替换原来的函数

void SolveFun(char\* expression,int LeftBracketIndex,int RightBracketIndex);

//判断表达式指定段是否是只含运算符和操作数，不含函数和括号的简单表达式

bool IsSimpleExpression(char\* expression,int start,int end);

//对简单表达式片段中的二元运算符，得到左操作数

double GetLeftOperand(char\* expression,int index\_of\_operator);

//对简单表达式片段中的二元运算符，得到右操作数

double GetRightOperand(char\* expression,int index\_of\_operator);

//解析简单表达式（只含运算符和操作数，不含函数和括号）

void AnalyzeSimpleExpression(char\* expression,int start,int end);

//解析表达式

void AnalyzeExpression(char\* expression);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*函数的实现\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//得到表达式

void GetExpression()

{

//清空缓冲区

setbuf(stdin,NULL);

gets(Expression);

StrCpy(OriginalExpression,Expression);

Format(Expression);

}

//格式化（去掉表达式中的空格）

void Format(char\* expression)

{

int len=StrLen(expression);

for(int i=0;i<len;i++)

{

if(expression[i]==' ')

{

RemoveCharAt(expression,i);

i--;

len--;

}

}

}

//把表达式中的常量替换成数，支持的常量：E,PI

void ReplaceConstSym(char\* expression)

{

int len=StrLen(expression);

for(int i=0;i<len;i++)

{

if(expression[i]=='E')

{

expression[i]=' ';

InsertString(expression,i,"2.71828");

Format(expression);

}

if(expression[i]=='P'&&expression[i+1]=='I')

{

expression[i]=' ';

expression[i+1]=' ';

InsertString(expression,i,"3.1415926");

Format(expression);

}

len=StrLen(expression);

}

}

//解析字符串形式的整数:start和end分别为str中需要解析的起始位置和终止位置

long ParseInt(char\* str,int start,int end)

{

long result=0;

int weight=1;

if(str[start]=='-')

{

for(int i=end;i>start;i--)

{

int digit=ParseDigit(str[i]);

if(digit<0)

{

Error=invalid\_expression;

ThrowException();

break;

}

else

{

result+=digit\*weight;

weight\*=10;

}

}

return (-1)\*result;

}

else

{

for(int i=end;i>=start;i--)

{

int digit=ParseDigit(str[i]);

if(digit<0)

{

Error=invalid\_expression;

ThrowException();

break;

}

else

{

result+=digit\*weight;

weight\*=10;

}

}

return result;

}

}

//解析字符串形式的小数:start和end分别为str中需要解析的起始位置和终止位置

double ParseDouble(char\* str,int start,int end)

{

long Integer=0;

double Decimal=0;

double result=0;

int IndexOfDot=start;

for(int index=start;index<=end;index++)

{

if(str[index]=='.')

{

IndexOfDot=index;

break;

}

}

if(IndexOfDot==start)

{

Integer=ParseInt(str,start,end);

result=Integer+Decimal;

}

else

{

int i=0;

int int\_weight=1;

double double\_weight=0.1;

if(str[start]=='-')

{

for(i=IndexOfDot-1;i>start;i--)

{

int digit=ParseDigit(str[i]);

if(digit<0)

{

Error=invalid\_expression;

ThrowException();

break;

}

else

{

Integer+=digit\*int\_weight;

int\_weight\*=10;

}

}

for(i=IndexOfDot+1;i<=end;i++)

{

int digit=ParseDigit(str[i]);

if(digit<0)

{

Error=invalid\_expression;

ThrowException();

break;

}

else

{

Decimal+=double\_weight\*digit;

double\_weight\*=0.1;

}

}

result=(-1)\*(Integer+Decimal);

}

else

{

for(i=IndexOfDot-1;i>=start;i--)

{

int digit=ParseDigit(str[i]);

if(digit<0)

{

Error=invalid\_expression;

ThrowException();

break;

}

else

{

Integer+=digit\*int\_weight;

int\_weight\*=10;

}

}

for(i=IndexOfDot+1;i<=end;i++)

{

int digit=ParseDigit(str[i]);

if(digit<0)

{

Error=invalid\_expression;

ThrowException();

break;

}

else

{

Decimal+=double\_weight\*digit;

double\_weight\*=0.1;

}

}

result=Integer+Decimal;

}

}

return result;

}

//对简单表达式片段中的二元运算符，得到左操作数

double GetLeftOperand(char\* expression,int index\_of\_operator)

{

return ParseDouble(expression,GetStartIndexOfLeftOperand(expression,index\_of\_operator),index\_of\_operator-1);

}

//对简单表达式片段中的二元运算符，得到右操作数

double GetRightOperand(char\* expression,int index\_of\_operator)

{

return ParseDouble(expression,index\_of\_operator+1,GetEndIndexOfRightOperand(expression,index\_of\_operator));

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*计算简单表达式中的函数值，并用值替换原来的函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*支持的函数:sin(),cos(),tg(),ctg(),sinh(),cosh(),tanh(),coth(),abs(),sign(),sqrt(),ln()\*/

void SolveFun(char\* expression,int LeftBracketIndex,int RightBracketIndex)

{

bool isFunRecognized=true;

int fun\_start=LeftBracketIndex-1;

int fun\_end=RightBracketIndex;

double para=ParseDouble(expression,LeftBracketIndex+1,RightBracketIndex-1);

double value=0;

//找到函数名起始下标

for(int i=fun\_start;i>=0;i--)

{

if(expression[i]>'a'&&expression[i]<'z')

{

fun\_start=i;

}

else break;

}

//sin

if(expression[fun\_start]=='s'&&expression[fun\_start+1]=='i'&&expression[fun\_start+2]=='n')

{

if(!IsRad)

{

para=para\*3.141592653589793/180;

}

value=sin(para);

}

//cos

else if(expression[fun\_start]=='c'&&expression[fun\_start+1]=='o'&&expression[fun\_start+2]=='s')

{

if(!IsRad)

{

para=para\*3.141592653589793/180;

}

value=cos(para);

}

//tg

else if(expression[fun\_start]=='t'&&expression[fun\_start+1]=='g')

{

if(!IsRad)

{

para=para\*3.141592653589793/180;

}

value=tan(para);

}

//ctg

else if(expression[fun\_start]=='c'&&expression[fun\_start+1]=='t'&&expression[fun\_start+2]=='g')

{

if(!IsRad)

{

para=para\*3.141592653589793/180;

}

if(para==0)

{

Error=data\_overflow;

ThrowException();

}

else value=1/tan(para);

}

//cosh

else if(expression[fun\_start]=='c'&&expression[fun\_start+1]=='o'&&expression[fun\_start+2]=='s'&&expression[fun\_start+3]=='h')

{

value=cosh(para);

}

//sinh

else if(expression[fun\_start]=='s'&&expression[fun\_start+1]=='i'&&expression[fun\_start+2]=='n'&&expression[fun\_start+3]=='h')

{

value=sinh(para);

}

//tanh

else if(expression[fun\_start]=='t'&&expression[fun\_start+1]=='a'&&expression[fun\_start+2]=='n'&&expression[fun\_start+3]=='h')

{

value=tanh(para);

}

//coth

else if(expression[fun\_start]=='c'&&expression[fun\_start+1]=='o'&&expression[fun\_start+2]=='t'&&expression[fun\_start+3]=='h')

{

if(para==0)

{

Error=data\_overflow;

ThrowException();

}

else value=cosh(para)/sinh(para);

}

//abs

else if(expression[fun\_start]=='a'&&expression[fun\_start+1]=='b'&&expression[fun\_start+2]=='s'&&expression[fun\_start+3]=='s')

{

value=fabs(para);

}

//sign

else if(expression[fun\_start]=='s'&&expression[fun\_start+1]=='i'&&expression[fun\_start+2]=='g'&&expression[fun\_start+3]=='n')

{

if(para>0)

{

value=1;

}

else if(para==0)

{

value=0;

}

else

{

value=-1;

}

}

//sqrt

else if(expression[fun\_start]=='s'&&expression[fun\_start+1]=='q'&&expression[fun\_start+2]=='r'&&expression[fun\_start+3]=='t')

{

if(para<0)

{

Error=logic\_error;

ThrowException();

}

else value=sqrt(para);

}

//ln

else if(expression[fun\_start]=='l'&&expression[fun\_start+1]=='n')

{

if(para<=0)

{

Error=logic\_error;

ThrowException();

}

else value=log(para);

}

else

{

isFunRecognized=false;

Error=unsupported\_function;

ThrowException();

}

if(isFunRecognized&&!CatchException())

{

char buffer[50];

sprintf\_by\_precision(buffer,value,DoubleCalPrecision);

for(int k=fun\_start;k<=RightBracketIndex;k++)

{

expression[k]=' ';

}

InsertString(expression,fun\_start,buffer);

Format(expression);

}

}

//解析简单表达式（只含运算符和操作数，不含函数和括号）

void AnalyzeSimpleExpression(char\* expression,int start,int end)

{

char buffer[50];

int new\_start=start;

int new\_end=end;

for(int i=start;i<=new\_end;i++)

{

if(IsOperator(expression,i))

{

if(expression[i]=='!')

{

int start\_index\_of\_operand=GetStartIndexOfLeftOperand(expression,i);

int end\_index\_of\_operand=i-1;

double n=ParseInt(expression,start\_index\_of\_operand,end\_index\_of\_operand);

if(n<0)

{

Error=logic\_error;

ThrowException();

printf("The operand of \'!\' must be none-negative!");

}

else

{

double value=1;

while(n>1)

{

value\*=n;

n--;

}

sprintf\_by\_precision(buffer,value,DoubleCalPrecision);

for(int j=start\_index\_of\_operand;j<=i;j++)

{

expression[j]=' ';

}

InsertString(expression,start\_index\_of\_operand,buffer);

Format(expression);

/\*\*DEBUG

\*\*printf("expression=%s\n",expression);

\*\*/

i=start;

new\_end=new\_end+StrLen(buffer)-(i-start\_index\_of\_operand+1);

}

}

}

}

for(i=start;i<=new\_end;i++)

{

if(IsOperator(expression,i))

{

if(expression[i]=='^')

{

int start\_index\_of\_left\_operand=GetStartIndexOfLeftOperand(expression,i);

int end\_index\_of\_right\_operand=GetEndIndexOfRightOperand(expression,i);

double LeftOperand=GetLeftOperand(expression,i);

double RightOperand=GetRightOperand(expression,i);

double value=pow(LeftOperand,RightOperand);

sprintf\_by\_precision(buffer,value,DoubleCalPrecision);

if(LeftOperand==0,RightOperand<0)

{

Error=logic\_error;

ThrowException();

}

if(!CatchException())

{

for(int j=start\_index\_of\_left\_operand;j<=end\_index\_of\_right\_operand;j++)

{

expression[j]=' ';

}

InsertString(expression,start\_index\_of\_left\_operand,buffer);

Format(expression);

printf("expression=%s\n",expression);

i=start;

new\_end=new\_end+StrLen(buffer)-(end\_index\_of\_right\_operand-start\_index\_of\_left\_operand+1);

}

}

}

}

for(i=start;i<=new\_end;i++)

{

if(IsOperator(expression,i))

{

if(expression[i]=='\*'||expression[i]=='/')

{

int start\_index\_of\_left\_operand=GetStartIndexOfLeftOperand(expression,i);

int end\_index\_of\_right\_operand=GetEndIndexOfRightOperand(expression,i);

double LeftOperand=GetLeftOperand(expression,i);

double RightOperand=GetRightOperand(expression,i);

double value=0;

if(expression[i]=='\*')

{

value=LeftOperand\*RightOperand;

}

if(expression[i]=='/')

{

if(RightOperand==0)

{

Error=logic\_error;

ThrowException();

printf("Zero can not be a divider!\n");

}

else value=LeftOperand/RightOperand;

}

/\*\*DEBUG

printf("LeftOperand=%lf\n",LeftOperand);

printf("RightOperand=%lf\n",RightOperand);

printf("value=%lf\n",value);

\*/

sprintf\_by\_precision(buffer,value,DoubleCalPrecision);

if(!CatchException())

{

for(int j=start\_index\_of\_left\_operand;j<=end\_index\_of\_right\_operand;j++)

{

expression[j]=' ';

}

InsertString(expression,start\_index\_of\_left\_operand,buffer);

Format(expression);

i=start;

new\_end=new\_end+StrLen(buffer)-(end\_index\_of\_right\_operand-start\_index\_of\_left\_operand+1);

}

}

}

}

for(i=start;i<=new\_end;i++)

{

if(IsOperator(expression,i))

{

if(expression[i]=='+'||expression[i]=='-')

{

int start\_index\_of\_left\_operand=GetStartIndexOfLeftOperand(expression,i);

int end\_index\_of\_right\_operand=GetEndIndexOfRightOperand(expression,i);

double LeftOperand=GetLeftOperand(expression,i);

double RightOperand=GetRightOperand(expression,i);

double value=0;

if(expression[i]=='+')

{

value=LeftOperand+RightOperand;

}

if(expression[i]=='-')

{

value=LeftOperand-RightOperand;

}

/\*\*DEBUG

printf("LeftOperand=%lf\n",LeftOperand);

printf("RightOperand=%lf\n",RightOperand);

printf("value=%lf\n",value);

\*/

sprintf\_by\_precision(buffer,value,DoubleCalPrecision);

for(int j=start\_index\_of\_left\_operand;j<=end\_index\_of\_right\_operand;j++)

{

expression[j]=' ';

}

InsertString(expression,start\_index\_of\_left\_operand,buffer);

Format(expression);

//printf("expression=%s\n",expression);

i=start;

new\_end=new\_end+StrLen(buffer)-(end\_index\_of\_right\_operand-start\_index\_of\_left\_operand+1);

}

}

}

}

//解析表达式

void AnalyzeExpression(char\* expression)

{

Format(expression);

ReplaceConstSym(expression);

toLowercase(expression);

while(!IsSimpleExpression(expression,0,StrLen(expression)-1))

{

//括号不匹配

if(GetIndexOfInnermostLeftBracket(expression)\*GetIndexOfInnermostRightBracket(expression)<0)

{

Error=unmatched\_bracket;

ThrowException();

break;

}

//表达式非法

if(GetIndexOfInnermostLeftBracket(expression)<0&&GetIndexOfInnermostRightBracket(expression)<0)

{

Error=invalid\_expression;

ThrowException();

printf("It seems that you\'ve made some mistakes.Please check it!\n");

break;

}

AnalyzeSimpleExpression(expression,GetIndexOfInnermostLeftBracket(expression)+1,GetIndexOfInnermostRightBracket(expression)-1);

if(!IsSimpleExpression(expression,GetIndexOfInnermostLeftBracket(expression)+1,GetIndexOfInnermostRightBracket(expression)-1))

{

Error=invalid\_expression;

ThrowException();

break;

}

if(IsBracketOfFun(expression,GetIndexOfInnermostLeftBracket(expression),GetIndexOfInnermostRightBracket(expression)))

{

SolveFun(expression,GetIndexOfInnermostLeftBracket(expression),GetIndexOfInnermostRightBracket(expression));

if(CatchException())break;

}

else

{

expression[GetIndexOfInnermostLeftBracket(expression)]=' ';

expression[GetIndexOfInnermostRightBracket(expression)]=' ';

Format(expression);

}

}

if(!CatchException())

{

AnalyzeSimpleExpression(expression,0,StrLen(expression)-1);

}

else

{

printf("表达式有误,解析失败!\n\n");;

}

}

#endif

大数运算核心程序文件：

#ifndef BigIntCal

#define BigIntCal

#include<stdlib.h>

#include"BigIntCalBase.h"

/\*-----------------------------------------------------

定义大数结构：用一字符串Num表示大数,

整型数组bitSet将其按位存储(第一位为符号位,0为正,1为负),

bool型isP标记其正负,int型bits存储其位数(不包括符号位)

------------------------------------------------------\*/

struct BigInt

{

char\* Num;

int\* bitSet;

bool isP;

int bits;

};

//从数字格式的字符串构建一个大数

BigInt CreatBigInt(char\* str\_BigInt)

{

BigInt new\_BigInt;

int len=StrLen(str\_BigInt);

new\_BigInt.Num=(char\*)malloc(sizeof(char)\*len);

StrCpy(new\_BigInt.Num,str\_BigInt);

new\_BigInt.isP=str\_BigInt[0]=='-'?false:true;

new\_BigInt.bits=new\_BigInt.isP?len:len-1;

new\_BigInt.bitSet=(int\*)malloc(sizeof(int)\*(new\_BigInt.bits+1));

new\_BigInt.bitSet[0]=new\_BigInt.isP?0:1;

if(new\_BigInt.isP)

{

for(int i=1;i<new\_BigInt.bits+1;i++)

{

new\_BigInt.bitSet[i]=ParseDigit(str\_BigInt[i-1]);

}

}

else

{

for(int i=1;i<new\_BigInt.bits+1;i++)

{

new\_BigInt.bitSet[i]=ParseDigit(str\_BigInt[i]);

}

}

return new\_BigInt;

}

//从表达式指定段解析出一个大数

BigInt ParseBigInt(char\* expression,int start,int end)

{

char\* str\_BigInt=(char\*)malloc(sizeof(char)\*(end-start+1));

for(int i=start,j=0;i<=end;i++,j++)

{

str\_BigInt[j]=expression[i];

}

return CreatBigInt(str\_BigInt);

}

//大数克隆函数

BigInt CloneBigInt(BigInt bigInt)

{

return CreatBigInt(bigInt.Num);

}

//取相反数

BigInt BigInt\_opp(BigInt bigInt)

{

if(bigInt.isP)

{

char\* Num=(char\*)malloc(sizeof(char)\*(bigInt.bits+1));

Num[0]='-';

for(int i=1;i<bigInt.bits+1;i++)

{

Num[i]=bigInt.Num[i-1];

}

return CreatBigInt(Num);

}

else

{

return ParseBigInt(bigInt.Num,1,StrLen(bigInt.Num));

}

}

//大数的绝对值函数

BigInt BigInt\_abs(BigInt bigInt)

{

if(bigInt.isP)

{

return CloneBigInt(bigInt);

}

else

{

return CloneBigInt(BigInt\_opp(bigInt));

}

}

//大数的绝对值大小比较|BigInt1|>=<|BigInt2|各返回1,0,-1

int BigIntCmpAbs(BigInt BigInt1,BigInt BigInt2)

{

if(BigInt1.bits>BigInt2.bits)

{

return 1;

}

else if(BigInt1.bits<BigInt2.bits)

{

return -1;

}

else

{

int bits=BigInt1.bits;

int\* a=(int\*)malloc(sizeof(int)\*(bits+1));

int\* b=BigInt2.bitSet;

for(int i=bits;i>=0;i--)

{

a[i]=BigInt1.bitSet[i];

}

bool equal=true;

for(int j=bits;j>0;j--)

{

if(a[j]!=b[j])

{

equal=false;

break;

}

}

if(equal)

{

return 0;

}

else

{

bool borrow=false;

for(int k=bits;k>0;k--)

{

if(borrow)

{

a[k]=a[k]>b[k]?(borrow=false,a[k]-b[k]-1):(a[k]+10-b[k]);

}

else

{

a[k]=a[k]<b[k]?(borrow=true,a[k]+10-b[k]):(a[k]-b[k]);

}

}

if(borrow)

{

return -1;

}

else

{

return 1;

}

}

}

}

//大数的大小比较BigInt1>=<BigInt2各返回1,0,-1

int BigIntCmp(BigInt BigInt1,BigInt BigInt2)

{

if(BigInt1.isP&&(!BigInt2.isP))

{

return 1;

}

else if((!BigInt1.isP)&&BigInt2.isP)

{

return -1;

}

else

{

return BigInt1.isP?BigIntCmpAbs(BigInt1,BigInt2):BigIntCmpAbs(BigInt2,BigInt1);

}

}

/\*------------

大数的四则运算

-------------\*/

//两个大数的绝对值之和

BigInt Add\_abs(BigInt BigInt1,BigInt BigInt2)

{

BigInt bigInt1=BigIntCmpAbs(BigInt1,BigInt2)?BigInt\_abs(BigInt1):BigInt\_abs(BigInt2);

BigInt bigInt2=BigIntCmpAbs(BigInt1,BigInt2)?BigInt\_abs(BigInt2):BigInt\_abs(BigInt1);

int bits\_max=bigInt1.bits;

int bits\_min=bigInt2.bits;

int\* a=bigInt1.bitSet;

int\* b=bigInt2.bitSet;

int\* c=(int\*)malloc(sizeof(int)\*(bits\_max+1));

for(int s=bits\_max;s>=0;s--)

{

c[s]=0;

}

for(int i=bits\_max,j=bits\_min;j>0;i--,j--)

{

c[i]=b[j];

}

int carry=0;

for(int k=bits\_max;k>=0;k--)

{

c[k]+=a[k]+carry;

carry=c[k]/10;

c[k]=c[k]%10;

}

char\* Num=(char\*)malloc(sizeof(char)\*(bits\_max+1));

for(int m=0;m<bits\_max+1;m++)

{

Num[m]=toChar(c[m]);

}

TrimRedundantZero(Num);

return CreatBigInt(Num);

}

//两个大数的绝对值之差(大减小)

BigInt Minus\_abs(BigInt BigInt1,BigInt BigInt2)

{

BigInt bigInt1=BigIntCmpAbs(BigInt1,BigInt2)?BigInt\_abs(BigInt1):BigInt\_abs(BigInt2);

BigInt bigInt2=BigIntCmpAbs(BigInt1,BigInt2)?BigInt\_abs(BigInt2):BigInt\_abs(BigInt1);

int bits\_max=bigInt1.bits;

int bits\_min=bigInt2.bits;

int\* a=bigInt1.bitSet;

int\* b=bigInt2.bitSet;

int\* c=(int\*)malloc(sizeof(int)\*(bits\_max+1));

for(int s=bits\_max;s>=0;s--)

{

c[s]=0;

}

for(int i=bits\_max,j=bits\_min;j>0;i--,j--)

{

c[i]=b[j];

}

bool borrow=false;

for(int k=bits\_max;k>0;k--)

{

if(borrow)

{

a[k]=a[k]>c[k]?(borrow=false,a[k]+10-c[k]):(a[k]-c[k]);

}

else

{

a[k]=a[k]<c[k]?(borrow=true,a[k]+10-c[k]):(a[k]-c[k]);

}

}

char\* Num=(char\*)malloc(sizeof(char)\*(bits\_max));

for(int m=0;m<bits\_max;m++)

{

Num[m]=toChar(a[m+1]);

}

TrimRedundantZero(Num);

return CreatBigInt(Num);

}

//加

BigInt Add(BigInt BigInt1,BigInt BigInt2)

{

if(BigInt1.isP&&BigInt2.isP)

{

return Add\_abs(BigInt1,BigInt2);

}

else if(BigInt1.isP&&(!BigInt2.isP))

{

return BigIntCmpAbs(BigInt1,BigInt2)>0?Minus\_abs(BigInt1,BigInt2):BigInt\_opp(Minus\_abs(BigInt1,BigInt2));

}

else if((!BigInt1.isP)&&BigInt2.isP)

{

return BigIntCmpAbs(BigInt1,BigInt2)>0?BigInt\_opp(Minus\_abs(BigInt2,BigInt1)):Minus\_abs(BigInt2,BigInt1);

}

else

{

return BigInt\_opp(Add\_abs(BigInt1,BigInt2));

}

}

//减

BigInt Minus(BigInt BigInt1,BigInt BigInt2)

{

if(BigInt1.isP&&BigInt2.isP)

{

return BigIntCmpAbs(BigInt1,BigInt2)>0?Minus\_abs(BigInt1,BigInt2):BigInt\_opp(Minus\_abs(BigInt1,BigInt2));

}

else if(BigInt1.isP&&(!BigInt2.isP))

{

return Add\_abs(BigInt1,BigInt2);

}

else if((!BigInt1.isP)&&BigInt2.isP)

{

return BigInt\_opp(Add\_abs(BigInt1,BigInt2));

}

else

{

return Add(BigInt1,BigInt\_opp(BigInt2));

}

}

//大数的1

BigInt BigInt\_0()

{

char zero[]="0";

return CreatBigInt(zero);

}

//大数的1

BigInt BigInt\_1()

{

char one[]="1";

return CreatBigInt(one);

}

//比较取大者

BigInt Max(BigInt BigInt1,BigInt BigInt2)

{

return BigIntCmp(BigInt1,BigInt2)?BigInt1:BigInt2;

}

//比较取小者

BigInt Min(BigInt BigInt1,BigInt BigInt2)

{

return BigIntCmp(BigInt1,BigInt2)?BigInt2:BigInt1;

}

//绝对值相乘

BigInt Multiply\_abs(BigInt BigInt1,BigInt BigInt2)

{

BigInt max=Max(BigInt\_abs(BigInt1),BigInt\_abs(BigInt2));

BigInt min=Min(BigInt\_abs(BigInt1),BigInt\_abs(BigInt2));

int bits=max.bits+min.bits;

int\* a=max.bitSet;

int\* b=min.bitSet;

int\* c=(int\*)malloc(sizeof(int)\*bits);

int\* d=(int\*)malloc(sizeof(int)\*(max.bits+1));

for(int m=0;m<bits;m++)c[m]=0;

for(int i=min.bits,k=bits-1;i>0;i--,k--)

{

for(int j=max.bits,carry=0;j>0;j--)

{

d[j]=a[j]\*b[i]+carry;

carry=d[j]/10;

d[j]%=10;

}

d[0]=carry;

int car=0;

for(j=max.bits;j>=0;j--)

{

int index=k-max.bits+j;

c[index]+=d[j];

car=c[index]/10;

c[index]%=10;

}

//c[k-max.bits+j]+=car;

}

char\* Num=(char\*)malloc(sizeof(char)\*bits);

for(int s=0;s<bits;s++)

{

//DEBUG

char ch=toChar(c[s]);

if(ch!=0)Num[s]=ch;

else printf("toChar(int) error!");

}

return CreatBigInt(Num);

}

//乘

BigInt Multiply(BigInt BigInt1,BigInt BigInt2)

{

return (BigInt1.isP&&BigInt2.isP)||((!BigInt1.isP)&&(!BigInt2.isP))?Multiply\_abs(BigInt1,BigInt2):BigInt\_opp(Multiply\_abs(BigInt1,BigInt2));

}

//除(只考虑整除)【还未实现】

BigInt Devide(BigInt BigInt1,BigInt BigInt2)

{

if(BigIntCmpAbs(BigInt1,BigInt2)==0)

{

return BigInt\_1();

}

else if(BigIntCmpAbs(BigInt1,BigInt2)<0)

{

return BigInt\_0();

}

else

{

return BigInt1;

}

}

//乘方

BigInt Pow(BigInt BigInt1,BigInt BigInt2)

{

BigInt bigInt1=BigInt1;

BigInt bigInt2=BigInt2;

while(BigIntCmp(bigInt2,BigInt\_1())>0)

{

bigInt1=Multiply(bigInt1,bigInt1);

bigInt2=Minus(bigInt2,BigInt\_1());

}

return bigInt1;

}

//阶乘

BigInt Fact(BigInt bigInt)

{

BigInt bigint=bigInt;

for(BigInt i=Minus(bigInt,BigInt\_1());BigIntCmp(i,BigInt\_1());Minus(i,BigInt\_1()))

{

bigint=Multiply(bigint,i);

}

return bigint;

}

//输出

void BigIntOut(BigInt bigInt)

{

printf("%s",bigInt.Num);

}

#endif