**RPN CALCULATOR**

**/\* Calc.c \*/**

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <assert.h>

#include "dynamicArray.h"

struct DynArr

{

TYPE \*data; /\* pointer to the data array \*/

int size; /\* Number of elements in the array \*/

int capacity; /\* capacity ofthe array \*/

};

/\* param: s the string

param: num a pointer to double

returns: true (1) if s is a number else 0 or false.

postcondition: if it is a number, num will hold

the value of the number

\*/

int isNumber(char \*s, double \*num)

{

char \*end;

double returnNum;

if(strcmp(s, "0") == 0)

{

\*num = 0;

return 1;

}

else

{

returnNum = strtod(s, &end);

/\* If there's anythin in end, it's bad \*/

if((returnNum != 0.0) && (strcmp(end, "") == 0))

{

\*num = returnNum;

return 1;

}

}

return 0; //if got here, it was not a number

}

/\* param: stack the stack being manipulated

pre: the stack contains at least two elements

post: the top two elements are popped and

their sum is pushed back onto the stack.

\*/

void add (struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 2){

printf("-- ERROR: INVALID USE OF '+' -- \n");

printf("-- REASON: '+' NEEDS TWO DIGITS -- \n");

return;

}

double a;

double b;

a = topDynArr(stack);

popDynArr(stack);

b = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, a + b);

}

/\* param: stack the stack being manipulated

pre: the stack contains at least two elements

post: the top two elements are popped and

their difference is pushed back onto the stack.

\*/

void subtract(struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 2){

printf("-- ERROR: INVALID USE OF '-' -- \n");

printf("-- REASON: '-' NEEDS TWO DIGITS -- \n");

return;

}

double a;

double b;

a = topDynArr(stack);

popDynArr(stack);

b = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, b - a);

}

/\* param: stack the stack being manipulated

pre: the stack contains at least two elements

post: the top two elements are popped and

their quotient is pushed back onto the stack.

\*/

void divide(struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 2){

printf("-- ERROR: INVALID USE OF '/' -- \n");

printf("-- REASON: '/' NEEDS TWO DIGITS -- \n");

return;

}

double a;

double b;

a = topDynArr(stack);

popDynArr(stack);

b = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, b/a);

}

void multiply(struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 2){

printf("-- ERROR: INVALID USE OF 'x' -- \n");

printf("-- REASON: 'x' NEEDS TWO DIGITS -- \n");

return;

}

double a;

double b;

double c;

a = topDynArr(stack);

popDynArr(stack);

b = topDynArr(stack);

popDynArr(stack);

c= a\*b;

pushDynArr(stack, c);

}

void power(struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 2){

printf("-- ERROR: INVALID USE OF '^' -- \n");

printf("-- REASON: 'x' NEEDS TWO DIGITS -- \n");

return;

}

double a;

double b;

a = topDynArr(stack);

popDynArr(stack);

b = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, pow(b,a));

}

void square(struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF '^2' -- \n");

printf("-- REASON: NOTHING TO '^2' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, pow(a,2));

}

void cube(struct DynArr \*stack)

{

/\* Checking for correct number of digts \*/

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF '^3' -- \n");

printf("-- REASON: NOTHING TO '^3' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, pow(a,3));

}

void absolute(struct DynArr \*stack)

{

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF 'abs' -- \n");

printf("-- REASON: NOTHING TO 'abs' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, abs(a));

}

void squarert(struct DynArr \*stack)

{

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF 'sqrt' -- \n");

printf("-- REASON: NOTHING TO 'sqrt' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, sqrt(a));

}

void exponential(struct DynArr \*stack){

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF 'exp' -- \n");

printf("-- REASON: NOTHING TO 'exp' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, exp(a));

}

void natualLog(struct DynArr \*stack){

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF 'ln' -- \n");

printf("-- REASON: NOTHING TO 'ln' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, log(a));

}

void tenLog(struct DynArr \*stack)

{

if(sizeDynArr(stack) < 1){

printf("-- ERROR: INVALID USE OF 'log' -- \n");

printf("-- REASON: NOTHING TO 'log' -- \n");

return;

}

double a;

a = topDynArr(stack);

popDynArr(stack);

pushDynArr(stack, log10(a));

}

double calculate(int numInputTokens, char \*\*inputString)

{

int i;

double result = 0.0;

char \*s;

struct DynArr \*stack;

double num;

//set up the stack

stack = createDynArr(20);

// start at 1 to skip the name of the calculator calc

for(i=1;i < numInputTokens;i++)

{

s = inputString[i];

// (1) Check if the string s is in the list of operators.

// (1a) If it is, perform corresponding operations.

// (1b) Otherwise, check if s is a number.

// (1b - I) If s is not a number, produce an error.

// (1b - II) If s is a number, push it onto the stack

if(strcmp(s, "+") == 0)

add(stack);

else if(strcmp(s,"-") == 0)

subtract(stack);

else if(strcmp(s, "/") == 0)

divide(stack);

else if(strcmp(s, "x") == 0)

/\* multiply function \*/

multiply(stack);

else if(strcmp(s, "^") == 0)

/\* power function \*/

power(stack);

else if(strcmp(s, "^2") == 0)

/\* square function \*/

square(stack);

else if(strcmp(s, "^3") == 0)

/\* cube function \*/

cube(stack);

else if(strcmp(s, "abs") == 0)

/\* absolute function \*/

absolute(stack);

else if(strcmp(s, "sqrt") == 0)

/\* sqaurert function \*/

squarert(stack);

else if(strcmp(s, "exp") == 0)

/\* exponential function \*/

exponential(stack);

else if(strcmp(s, "ln") == 0)

/\* naturalLog function \*/

natualLog(stack);

else if(strcmp(s, "log") == 0)

/\* log function \*/

tenLog(stack);

// (when s is not an operator)

// deal with special values ("pi" and "e")

else if(isNumber(s, &num))

pushDynArr(stack, num);

else if(strcmp(s, "pi") == 0)

pushDynArr(stack, 3.14159265);

else if(strcmp(s, "e") == 0)

pushDynArr(stack, 2.7182818);

} //end for

/\* (1) Check if everything looks OK and produce an error if needed.

\* (2) Store the final value in result and print it out.

\*/

if(sizeDynArr(stack) == 1){ //

printf("Answer = %f\n", topDynArr(stack));

return result;

}

else {

printf("ERROR! Trailing numbers! \n");

return 0;

}

}

int main(int argc , char\*\* argv)

{

// assume each argument is contained in the argv array

// argc-1 determines the number of operands + operators

if (argc == 1)

return 0;

calculate(argc,argv);

return 0;

}

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

Dynamic Array Functions

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

/\* Initialize (including allocation of data array) dynamic array.

param: v pointer to the dynamic array

param: cap capacity of the dynamic array

pre: v is not null

post: internal data array can hold cap elements

post: v->data is not null

\*/

void initDynArr(DynArr \*v, int capacity)

{

assert(capacity > 0);

assert(v!= 0);

v->data = (TYPE \*) malloc(sizeof(TYPE) \* capacity);

assert(v->data != 0);

v->size = 0;

v->capacity = capacity;

}

/\* Allocate and initialize dynamic array.

param: cap desired capacity for the dyn array

pre: none

post: none

ret: a non-null pointer to a dynArr of cap capacity

and 0 elements in it.

\*/

DynArr\* createDynArr(int cap)

{

assert(cap > 0);

DynArr \*r = (DynArr \*)malloc(sizeof( DynArr));

assert(r != 0);

initDynArr(r,cap);

return r;

}

/\* Deallocate data array in dynamic array.

param: v pointer to the dynamic array

pre: none

post: d.data points to null

post: size and capacity are 0

post: the memory used by v->data is freed

\*/

void freeDynArr(DynArr \*v)

{

if(v->data != 0)

{

free(v->data); /\* free the space on the heap \*/

v->data = 0; /\* make it point to null \*/

}

v->size = 0;

v->capacity = 0;

}

/\* Deallocate data array and the dynamic array ure.

param: v pointer to the dynamic array

pre: none

post: the memory used by v->data is freed

post: the memory used by d is freed

\*/

void deleteDynArr(DynArr \*v)

{

assert (v!= 0);

freeDynArr(v);

free(v);

}

/\* Resizes the underlying array to be the size cap

param: v pointer to the dynamic array

param: cap the new desired capacity

pre: v is not null

post: v has capacity newCap

\*/

void \_dynArrSetCapacity(DynArr \*v, int newCap)

{

int i;

TYPE \*oldData;

int oldSize = v->size;

oldData = v->data;

printf("========Resizing========\n");

/\* Create a new dyn array with larger underlying array \*/

initDynArr(v, newCap);

for(i = 0; i < oldSize; i++){

v->data[i] = oldData[i];

}

v->size = oldSize;

/\* Free the original data \*/

free(oldData);

}

/\* Get the size of the dynamic array

param: v pointer to the dynamic array

pre: v is not null

post: none

ret: the size of the dynamic array

\*/

int sizeDynArr(DynArr \*v)

{

assert(v!=0);

return v->size;

}

/\* Adds an element to the end of the dynamic array

param: v pointer to the dynamic array

param: val the value to add to the end of the dynamic array

pre: the dynArry is not null

post: size increases by 1

post: if reached capacity, capacity is doubled

POST: VAL IS IN THE LAST UTILIZED POSITION IN THE ARRAY

\*/

void addDynArr(DynArr \*v, TYPE val)

{

assert(v != 0);

/\* Check to see if a resize is necessary \*/

if(v->size >= v->capacity)

\_dynArrSetCapacity(v, 2 \* v->capacity);

v->data[v->size] = val;

v->size++;

}

/\* Get an element from the dynamic array from a specified position

param: v pointer to the dynamic array

param: pos integer index to get the element from

pre: v is not null

pre: v is not empty

pre: pos < size of the dyn array and >= 0

post: no changes to the dyn Array

ret: value stored at index pos

\*/

TYPE getDynArr(DynArr \*v, int pos)

{

assert(v!=0);

assert(pos < v->size);

assert(pos >= 0);

return v->data[pos];

}

/\* Put an item into the dynamic array at the specified location,

overwriting the element that was there

param: v pointer to the dynamic array

param: pos the index to put the value into

param: val the value to insert

pre: v is not null

pre: v is not empty

pre: pos >= 0 and pos < size of the array

post: index pos contains new value, val

\*/

void putDynArr(DynArr \*v, int pos, TYPE val)

{

assert(v!=0);

assert(pos < v->size);

assert(pos >= 0);

v->data[pos] = val;

assert(v->data[pos] = val);

}

/\* Swap two specified elements in the dynamic array

param: v pointer to the dynamic array

param: i,j the elements to be swapped

pre: v is not null

pre: v is not empty

pre: i, j >= 0 and i,j < size of the dynamic array

post: index i now holds the value at j and index j now holds the value at i

\*/

void swapDynArr(DynArr \*v, int i, int j)

{

TYPE temp;

assert(v!=0);

assert(i < v->size);

assert(j < v->size);

assert(i >= 0);

assert(j >= 0);

temp = v->data[i];

v->data[i] = v->data[j];

v->data[j] = temp;

assert(v->data[i] =j);

assert(v->data[j]=i);

}

/\* Remove the element at the specified location from the array,

shifts other elements back one to fill the gap

param: v pointer to the dynamic array

param: idx location of element to remove

pre: v is not null

pre: v is not empty

pre: idx < size and idx >= 0

post: the element at idx is removed

post: the elements past idx are moved back one

\*/

void removeAtDynArr(DynArr \*v, int idx)

{

int i;

assert(v!= 0);

assert(idx < v->size);

assert(idx >= 0);

//Move all elements up

for(i = idx; i < v->size-1; i++){

v->data[i] = v->data[i+1];

}

v->size--;

}

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

Stack Interface Functions

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

/\* Returns boolean (encoded in an int) demonstrating whether or not the

dynamic array stack has an item on it.

param: v pointer to the dynamic array

pre: the dynArr is not null

post: none

ret: 1 if empty, otherwise 0

\*/

int isEmptyDynArr(DynArr \*v)

{

assert(v!= 0);

return !(v->size);

}

/\* Push an element onto the top of the stack

param: v pointer to the dynamic array

param: val the value to push onto the stack

pre: v is not null

post: size increases by 1

if reached capacity, capacity is doubled

val is on the top of the stack

\*/

void pushDynArr(DynArr \*v, TYPE val)

{

assert(v!=0);

addDynArr(v, val);

}

/\* Returns the element at the top of the stack

param: v pointer to the dynamic array

pre: v is not null

pre: v is not empty

post: no changes to the stack

\*/

TYPE topDynArr(DynArr \*v)

{

assert(v!=0);

assert(!isEmptyDynArr(v));

return v->data[v->size-1];

}

/\* Removes the element on top of the stack

param: v pointer to the dynamic array

pre: v is not null

pre: v is not empty

post: size is decremented by 1

the top has been removed

\*/

void popDynArr(DynArr \*v)

{

assert(v!=0);

assert(! isEmptyDynArr(v));

v->size--;

}

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

Bag Interface Functions

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

/\* Returns boolean (encoded as an int) demonstrating whether or not

the specified value is in the collection

true = 1

false = 0

param: v pointer to the dynamic array

param: val the value to look for in the bag

pre: v is not null

pre: v is not empty

post: no changes to the bag

\*/

int containsDynArr(DynArr \*v, TYPE val)

{

int i = 0;

assert(v!=0);

assert(!isEmptyDynArr(v));

for(i = 0; i < sizeDynArr(v); i++)

if(EQ(v->data[i], val) )

return 1;

return 0;

}

/\* Removes the first occurrence of the specified value from the collection

if it occurs

param: v pointer to the dynamic array

param: val the value to remove from the array

pre: v is not null

pre: v is not empty

post: val has been removed

post: size of the bag is reduced by 1

\*/

void removeDynArr(DynArr \*v, TYPE val)

{

int i = 0;

assert(v!=0);

assert(!isEmptyDynArr(v));

assert(containsDynArr(v,val));

for(i = 0; i < sizeDynArr(v); i++)

if(EQ(v->data[i], val))

{

removeAtDynArr(v,i);

break;

}

}

/\* Utility function for debugging \*/

void \_printDynArr(struct DynArr \*da)

{

int i;

for(i=0; i < da->size; i++)

printf("DA[%d] == %f\n", i, da->data[i]);

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* dynamicArray.h : Dynamic Array implementation. \*/**

#include<math.h>

#ifndef DYNAMIC\_ARRAY\_INCLUDED

#define DYNAMIC\_ARRAY\_INCLUDED 1

# ifndef TYPE

# define TYPE double

# define TYPE\_SIZE sizeof(double)

# endif

# ifndef EQ

# define EQ(A, B) (fabs(A - B) < 10e-7)

# endif

typedef struct DynArr DynArr;

/\* Dynamic Array Functions \*/

DynArr \*createDynArr(int cap);

void deleteDynArr(DynArr \*v);

int sizeDynArr(DynArr \*v);

void addDynArr(DynArr \*v, TYPE val);

TYPE getDynArr(DynArr \*v, int pos);

void putDynArr(DynArr \*v, int pos, TYPE val);

void swapDynArr(DynArr \*v, int i, int j);

void removeAtDynArr(DynArr \*v, int idx);

/\* Stack interface. \*/

int isEmptyDynArr(DynArr \*v);

void pushDynArr(DynArr \*v, TYPE val);

TYPE topDynArr(DynArr \*v);

void popDynArr(DynArr \*v);

/\* Bag Interface \*/

int containsDynArr(DynArr \*v, TYPE val);

void removeDynArr(DynArr \*v, TYPE val);

#endif

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\*Odd\_Even.C; RPN functions \*/**

#include<stdio.h>

#define MAXOP 100 /\* maximum size of the operand of operator \*/

#define NUMBER '0' /\* signal that a number was found \*/

int getOp(char []); /\* takes a character string as an input \*/

void push(double);

double pop(void);

double asciiToFloat(char []);

/\* reverse polish calculator \*/

int main()

{

int type;

double op2;

char s[MAXOP];

printf("Please enter Polish Notation or ^d to quit.\n");

while ((type = getOp(s)) != EOF) {

switch (type) {

case NUMBER:

push(asciiToFloat(s));

break;

case '+':

push(pop() + pop());

break;

case '\*':

push(pop() \* pop());

break;

case '-':

op2 = pop();

push(pop() - op2);

break;

case '/':

op2 = pop();

if (op2 != 0.0)

push(pop() / op2);

else

printf("error : zero divisor!\n");

break;

case '\n':

printf("\t%.2f\n", pop()); /\* this will print the last item on the stack \*/

// printf("Neither a digit or a decimal: %d 0x%x.\n", c, c); /\* re-prompt the user for further calculation \*/

break;

default:

printf("error: unknown command %s.\n", s);

break;

}

}

return 0;

}

#include <ctype.h>

/\* asciiToFloat: this will take ASCII input and convert it to a double \*/

double asciiToFloat(char s[])

{

double val;

double power;

int i;

int sign;

for (i = 0; isspace(s[i]); i++) /\* gets rid of any whitespace \*/

;

sign = (s[i] == '-') ? -1 : 1; /\* sets the sign of the input \*/

if (s[i] == '+' || s[i] == '-') /\* excludes operands, plus and minus \*/

i++;

for (val = 0.0; isdigit(s[i]); i++)

val = 10.0 \* val + (s[i] - '0');

if (s[i] = '.')

i++;

for (power = 1.0; isdigit(s[i]); i++) {

val = 10.0 \* val + (s[i] - '0');

power \*= 10.0;

}

return sign \* val / power;

}

#define MAXVAL 100 /\* maximum depth of value stack \*/

int sp = 0; /\* next free stack position \*/

double val[MAXVAL]; /\* value stack \*/

/\* push: push f onto value stack \*/

void push(double f)

{

if (sp < MAXVAL) {

val[sp++] = f; /\* take the input from the user and add it to the stack \*/

// printf("The value of the stack position is %d\n", sp);

}

else

printf("error: stack full, cant push %g\n", f);

}

/\* pop: pop and return the top value from the stack \*/

double pop(void)

{

if (sp > 0)

return val[--sp];

else {

printf("error: stack empty\n");

return 0.0;

}

}

#include <ctype.h>

int getch(void);

void ungetch(int);

/\* getOp: get next operator or numeric operand \*/

int getOp(char s[])

{

int i;

int c;

while ((s[0] = c = getch()) == ' ' || c == '\t')

;

s[1] = '\0';

// if (!isdigit(c) && c != '.') {

// printf("Neither a digit or a decimal: %d 0x%x.\n", c, c);

// return c; /\* neither a digit nor a decimal \*/

// }

i = 0;

if (isdigit(c)) /\* grab the integer \*/

while (isdigit(s[++i] = c = getch()))

;

if (c == '.') /\* grab the fraction \*/

while (isdigit(s[++i] = c = getch()))

;

s[i] = '\0';

if (c != EOF)

ungetch(c);

return NUMBER;

}

#define BUFSIZE 100

char buf[BUFSIZE]; /\* buffer for ungetch \*/

int bufp = 0; /\* next free position in buffer \*/

/\* getch: get a number that may or may not have been pushed back \*/

int getch(void)

{

return (bufp > 0) ? buf[--bufp] : getchar();

}

/\* ungetch: if we read past the number, we can push it back onto input buffer \*/

void ungetch(int c)

{

if (bufp >= BUFSIZE)

printf("ungetch: to many characters.\n");

else

buf[bufp++] = c;

}

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++