CS240 - Systems Programming in C
Exam #2
2005 Mar 29

"You think you know when you can learn,
are more sure when you can write,
even more when you can teach,
but certain when you can program."
– *Epigrams in Programming*, Alan J. Perlis

Instructions: Read carefully through the entire exam first, and plan your time accordingly. Note the relative weights of each segment, as a percentage of the total exam score.

This exam is **closed book, closed notes**. You may *not* refer to any books or other materials during the exam. Calculators or other electronic devices are also prohibited unless explicitly stated otherwise.

Hats should be removed, or worn brim backwards, to ensure that the proctors have an unobstructed view of your eyes.

Write your answers on this exam. You may use both sides of the page.

When you are done, present your completed exam and your student identification to the instructor or proctors at the head table. If leaving before the exam period is concluded, please leave as quietly as possible as a courtesy to your neighbors.

Name:

Student Number:

Signature:

1. (A to F - 15%) Write an `atof()` function, which takes a character string as input and returns an equivalent floating point number. You may assume the string contains no leading white space, no scientific notation or exponents, and that the value will fit into a `double`. There may be an optional leading minus sign. You may use any of the standard library functions prototyped in `ctype.h`, (although this is not required), but *NOT* any of the other standard library functions.

```
#include<ctype.h>
/**
 * atof - Given a character string, returns the equivalent double float.
 * Parameters: char *s - A string of characters.  We assume that the
 *        pointer is not NULL, and the number spelled out in the array
 *        will fit into a 'double' type variable.
 *        There may or may not be a leading minus sign.
 *        There may or may not be decimal points.
 * Returns: Floating point value.  If anything goes wrong,
 *        (non-digits, multiple decimal points, etc.,) return 0.0.
 */
double atof(char *s)
{

  double result = 0.0;
  int power = -1; /* power < 0 means no decimal point.   */
  int sign = 1;   /* sign == 1 for positive, -1 for neg. */

  /* Check for leading negative sign.          */
  if ('-' == *s) { sign = -1; s++; }
  while (*s) /* Loop until null terminator.    */
    { /* Check for plain digits.               */
      if (isdigit(*s))
             {
                 result = result * 10.0 + (*s - '0');
                 if (power >= 0) power++;
             }
      /* Check for a single decimal point.     */
      else if ( ('.' == *s) && (power < 0) )
             {   power = 0;   }
      else   {   return 0.0;  }  /* Else bail. */
      s++;
    }
  /* Divide by 10 for each decimal place.      */
  while (0 < power--) { result /= 10.0; }
  return sign * result;
}
```

2. (Stack of Floats - 20%) Given this struct

```
struct node
{
  double content;
  struct node *next;
};
```

implement `push()` and `pop()` functions for a dynamically-allocated stack of floating point values. Recall that the *stack* data structure is last-in, first-out. A `push()` puts a new element at the top of the stack, *pushing* all of the previous elements down further. A `pop()` *pops* the top element off of the stack, and the element beneath becomes the new top.

```
/**
 * push - Pushes an element onto our floating-point stack.
 * Parameters: payload - the element to store on top of the stack.
 *             top - a pointer to the top of the old stack.
 * Returns:    A pointer to the top of the new stack.
 * Notes:      Returns pointer to old stack if it could not
 *             allocate memory for a new element.
 */
struct node *push(double payload, struct node *top)
{

  struct node *newtop =
    (struct node *)malloc(sizeof(struct node));

  if (newtop)
    {
      newtop->content = payload;
      newtop->next    = top;
    }
  return newtop;
}
```

```
/**
 * pop - Pops an element off of our floating-point stack.
 * Parameters: payload - A pointer to the location we should pop
 *             the top element into.
 *             top - a pointer to the top of the old stack.
 * Returns:    A pointer to the top of the new stack.
 * Notes:      Responsible for deallocating the memory for the
 *             top element after it is off the stack.
 *             Does nothing if either parameter is NULL.
 */
struct node *pop(double *payload, struct node *top)
{

  struct node *oldtop = top;
  if (payload && top)
    {
      *payload = top->content;
      top = top->next;
      free(oldtop);
    }
  return top;
}
```

3. (Calculator - 15%)

   Using the functions from the last two questions (you may assume that they work properly, even if you suspect yours do not) implement a command-line, stack-based Reverse Polish calculator.

   The calculator will take floating point operators and operands separated by spaces. The two operators are defined as follows:

   - `+`: Pops two numbers off of the stack, adds them together, and pushes the result on the stack.
   - `x`: Pops two numbers off of the stack, multiplies them together, and pushes the result on the stack.

   For example:

   ```
   romulus> calculator 1.2 3.4 +
   ```

   adds together 1.2 and 3.4 to produce

   ```
   4.600000
   ```

   ```
   romulus> calculator 1.2 3.4 + -5.6 x
   ```

   adds together 1.2 and 3.4, and multiplies the result by -5.6.

   ```
   -25.760000
   ```

   You may assume that input is well-formed, that the answer 0.000000 is returned if anything goes wrong, and that upon exit it prints whatever element is on the top of the stack (the final result) using `printf()` and the "`%f`" format specifier.

```
#include<stdio.h>
/* Prototypes for code in previous problems. */
/* (Assume these are already written.)       */
double atof(char *s);
struct node { double content;  struct node *next; };
struct node *push(double payload, struct node *top);
struct node *pop(double *payload, struct node *top);

/**
 * Reverse Polish Calculator
 */
int main(int argc, char **argv)
{


  int arg = 1;
  double oper1 = 0.0, oper2 = 0.0;
  struct node *top = NULL;

  while (arg < argc)
    {
      if (0 == strcmp(argv[arg], "x"))
        {
          top = pop(&oper2, top);
          top = pop(&oper1, top);
          top = push(oper1 * oper2, top);
        }
      else if (0 == strcmp(argv[arg], "+"))
        {
          top = pop(&oper2, top);
          top = pop(&oper1, top);
          top = push(oper1 + oper2, top);
        }
      else
        {
          top = push(atof(argv[arg]), top);
        }
      arg++;
    }
  top = pop(&oper1, top);
  printf("%f\n", oper1);
}
```

4. (Testing - 15%)

   What are the major types of testing and testcases that should be applied to the Reverse Polish Calculator system (Questions 1 - 3)? Give specific examples for each type.

   - Unit Testing of `atof()` function independently from calculator:
     - Positive cases: simple inputs like "`1.2`", "`3.4`", and "`-5.6`".
     - Negative cases: erroneous inputs, like "`1.2.3`", "`+4.5`", "`1.foo`", "`--6.7.`", and "`6.-7`".
     - Boundary cases: "`0`", "`-1`", "`2`", "`3.`", "`0.4`", "`.5`"
   - Unit Testing of stack functions independently from calculator:
     - Positive cases: Try a simple `push()` and `pop()`, see if you get the right answer back.
     - Negative cases: Pass in NULL pointer to `pop()`, try to allocate too many elements.
     - Boundary cases: `push()` first element onto stack, `pop()` last element off stack, `push()` more elements after stack has been empty.
   - Integration testing of whole calculator:
     - Positive cases: Add two numbers; multiply two numbers; add a positive and a negative, two negatives; multiply a positive and a negative, or two negatives. Try a compound expression with addition and multiplication.
     - Negative cases: Ill-formed input. (Improper floats, non-existent operators.)
     - Boundary cases: No arguments; Only one operand; Only one operator; One operand and one operator; Expression with too few operators (stack has multiple operands left); Expression with too many operators (stack runs out of operators.)

5. (PixelMix - 15%)

The Made-Up Notation Key Image (MUNKI) file format specifies that every pixel in the image will contain: Red, Green, and Blue color values ranging from 0 to 127; X and Y offsets ranging from -8 to +7; a Masked flag that is either true or false.

Define a `typedef`'ed struct below that will encode a MUNKI Pixel in less than 4 bytes of space on a Pentium or Sparc machine.

```
typedef struct
{
  unsigned int red    : 7;
  unsigned int green  : 7;
  unsigned int blue   : 7;
    signed int x      : 4;
    signed int y      : 4;
  unsigned int masked : 1;
} Pixel;
```

Write a matching allocator function that takes parameters for all of the components of a MUNKI Pixel, and returns a pointer to a dynamically-allocated Pixel.

```
/**
 * pixAlloc() - returns a pointer to a fresh Pixel.
 * Parameters: No range checks are performed.
 * Returns:    New Pixel, or NULL if one cannot be allocated.
 */
Pixel *pixAlloc(int red, int green, int blue, int x, int y, int masked)
{

  Pixel *p = (Pixel *)malloc(sizeof(Pixel));
  if (NULL != p)
    {
      p->red   = red;   p->green = green;   p->blue    = blue;
      p->x     = x;     p->y     = y;       p->masked = masked;
    }
  return p;
}
```

6. (TreeSort - 20%)

Assuming a working implementation from the last question, define and `typedef` a `PixelNode` structure that contains a pointer to a `Pixel`, and left and right pointers for a binary tree of MUNKI Pixel's.

```
typedef struct node
{
  Pixel *pixel;
  struct node *left;
  struct node *right;
} PixelNode;
```

Assuming an appropriate `pixCompare` function is defined that works the same way as `strcmp()` except with `Pixel`s, write an `addPixel()` function that adds a new `Pixel` into a sorted binary tree, returnng a pointer to the new tree.

```
/* This function already defined for you. */
int pixCompare(Pixel p1, Pixel p2);

/* Add pixel into the sorted binary tree at root. */
PixelNode *addPixel(Pixel *pixel, PixelNode *root)
{

  int cmp = 0;
  if (NULL == pixel) return root;
  if (NULL == root)
    {
      root = (PixelNode *)malloc(sizeof(PixelNode));
      if (NULL != root)
        {
          root->pixel = pixel;
          root->left  = NULL;
          root->right = NULL;
        }
      return root;
    }
  cmp = pixCompare(*pixel, *(root->pixel));
  if (cmp > 0)     {  root->right = addPixel(pixel, root->right);  }
  else if (cmp < 0)  {  root->left = addPixel(pixel, root->left);  }
  else  {  free(root->pixel);   root->pixel = pixel;  }
  return root;
}
```

```
if (exam == DONE) { free(suffering_CS240_student); }
```