C Language Samples

All of these were created as fast coding exercises under a very tight time constraint; minimal error checking.

Sample A

```
/* This function returns true if the argument
// is a palindrome string.  For example:
// aba a abba
// Parameters:
//   input C string
// Return values:
//   -1 for error, 0 for false, 1 for true
*/
int is_palindrome(char *str)
{
  // check for null string
  if ( str == 0)
    return -1;

  // indices for front and back of string
  int fi = 0;
  int bi = str.len() - 1;

  // loop through string, characters should match
  // at each check
  while ( fi < bi ) {
    if ( str[fi] == str[bi] );

    else
      return 0;
    fi++;
    bi--;
  } // while

  return 1;
} // is_palindrome
```

Sample B

```
/*
Background:
A human interface has a button that records a speed when pushed.  We
need a debounce.
```

```
Terminology:
Signal A. State of the button - Boolean (i.e. bool)
Signal B. Speed of the vehicle - Analog (i.e. float)

Requirement:
Create a helper function to tell the calling code when A went from 0
to 1 and A has been stable at 1 for at least 3 samples. At that time,
also tell the calling code what B was when A first went from 0 to 1.

No need to sample A or B. They are provided as a pair at a specific
frequency. In other words, function will be called periodically with a
new value for A and B.

Examples:
   A: 0 0 0 0 1 0 0 0 1 1 1 1 1 0 0
   B: 1 1 2 2 3 3 4 5 5 5 6 6 7 7 8
noise on A:   ^
A is stable:              ^
B to return:          ^

   A: 0 0 0 0 1 1 0 0 1 1 1 0 1 0 0
   B: 4 3 2 1 0 0 0 0 0 0 1 1 1 2 2
noise on A:   ^ ^             ^
A is stable:              ^
B to return:          ^
*/

int Aprev;
int count = 0;

// container for A and B
typedef struct values {
  int Abutton;
  float Bspeed;
} myVal;

// Debounce button_helper function.
// Parameters: A is button state, B is speed
// Returns: both A and B when A is detected as button press
values button_helper(int A, float B)
{
  // first indication A is pressed… maybe… start counting
  if ( A == 1 && Aprev == 0 ) {
    count++;
    myVal.Bspeed = B;
  }

  // continue counting for depressed A button
  if ( A == 1 )
    if ( count )
      count++;
```

```
  // check for button reset
  if ( Aprev == 0 )
    count = 0;

  // confirmed button press
  if ( count >= 3 ) {
    myVal.Abutton = A;
    count = 0;
  }

  // always store current value as previous before leaving
  Aprev = A;

  return myVal;

} // button_helper
```

Sample C

```
/*
This is a program that takes a string arithmetic equation and returns
an integer evaluating the equation.
Must support 2 operators: + and *
Negative numbers are not required. We only need to support integers.
Console interface not provided for this project.

For example:
   Input:  "1+2+3"      Output:  6
   Input:  "1+2*3"      Output:  7
   Input:  "1*2+3*4"    Output:  14
   Input:  "10*100"     Output:  1000
   Input:  "10+100*2"   Output:  210
   Input:  "7*8*9+7+8+9"Output:  528

We'll not use library functions atoi, itoa or std::to_string.
*/

#include <stdio.h>
#include <string.h>

int calc(char * s);
int ctoi(char * snum);
int evaluate(int * num, int num_size, int * ops);

int main()
{
  printf("Expression Evaluator\n");

  // define some input string expressions here to test
```

```
  int result;

  result = calc("19*2+3");
  printf("result: %d\n", result);

  result = calc("1+2+3*4+5*6*7+8");
  printf("result: %d\n", result);

  return 0;
} // main

// asci table values for multiply/plus operators
// also bounds for digit characters
#define ASCII_M  42
#define ASCII_P  43
#define ASCII_0  48
#define ASCII_9  57

#define MAX_NUMS    10
#define MAX_DIGITS  10

/* entry point to the evaluation
// Parameters:  string expression
*/ Returns:     integer result
int calc(char* s)
{
  // this will hold the final value of the expression for return
  int val = 0;

  // let's parse everything out first and store it, then math away
  // might be a way to do this stack style
  int num[MAX_NUMS];
  int ops[MAX_NUMS-1];

  // temp holder for individual number string
  char snum[MAX_DIGITS];

  // indices for expression, snum, ops, and nums
  int ei = 0;
  int sni = 0;
  int oi = 0;
  int ni = 0;

  // walk through each operand/operator in input string s
  // assume no string errors for now
  // eventually we'll hit the last number; terminate and process after
  while (s[ei] != 0) {

    // if we find a multiplier, end of number so convert and save it
    if (s[ei] == ASCII_M) {
      // terminate the number string and convert to integer
      snum[sni] = 0;
```

```
      num[ni] = ctoi(snum);

      // save multiply op to ops stack, then increment both index
      ops[oi] = ASCII_M;
      oi++;
      ni++;
      // reset temp stuff
      sni = 0;
    }
    // case that we hit addition plus sign, also convert + save
    else if (s[ei] == ASCII_P) {
      // terminate the number string and convert to integer
      snum[sni] = 0;
      num[ni] = ctoi(snum);

      // save plus op to ops stack, then increment both index
      ops[oi] = ASCII_P;
      oi++;
      ni++;
      // reset temp stuff
      sni = 0;
    }
    // if no operator, then continue pulling digits
    else {
      snum[sni] = s[ei];
      sni++;
    }

    // increment main expression string index
    ei++;

  } //while

  //now, get the last number out and store it
  snum[sni] = 0;
  num[ni] = ctoi(snum);
  ni++;

  // do the math
  val = evaluate(num, ni, ops);

  return val;
} //calc

/* this function converts the character digits to an int
// Parameters:  string to convert
*/ Returns:     integer equivalent
int ctoi(char* snum) {
  int len = strlen(snum);

  int i = 0;
  int col = 1;
```

```
  int sum = 0;
  int intval;

  // start at tail of number string
  // convert by subtracting ascii zero value from digit,
  // then multiply by decimal column factor
  for ( i=len-1; i>=0; i--) {
    intval = (int) snum[i] - ASCII_0;
    sum += intval * col;
    col *= 10;
  }

  return sum;
} // ctoi

/* this function does the math on the stack of operands and operators
// it takes as arguments the stack of converted integer numbers,
// the size of that stack, and the operator stack
// Parameters:  pointer to int stack
//              size of stack
//              pointer to operator stack
*/ Returns:     computation answer
int evaluate(int * num, int num_size, int * ops) {

  // indices for num and ops
  int ni,oi;
  // temporary multiplication value storage
  int mstore = 1;
  // final return value
  int val = 0;

  // do multiplication first, store intermediate vals with the nums
  for (oi=0; ops[oi] == ASCII_M || ops[oi] == ASCII_P; oi++) {

    // when multiply, store result of X*Y,
    // clear one of the two operands in the numbers stack,
    // keep going until no more multiplies,
    // then save result to numbers stack
    while (ops[oi] == ASCII_M) {
      ni = oi;
      mstore *= num[ni];
      num[ni] = 0;
      oi++;
    } //while
    ni = oi;
    mstore *= num[ni];
    num[ni] = mstore;

    //reset for the next multiplied set
    mstore = 1;
  } //for
```

```
  // now do the final add, since multiplies are all done
  for (ni=0; ni < num_size; ni++) {
    val += num[ni];
  }

  return val;

} //evaluate
```