/*
   Cross-Correlation Speed Sensor Code.
*/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <memory.h>

long random(void); /* This should be in the include file but isn't. */

#define WINDOW 100 /* Number of samples to compare. */
#define DIST 1.0 /* Distance between sensors, in meters. */
#define SIZE 1000 /* Number of samples to store. */
#define SAMPLE_INTERVAL 0.01 /* Time between reading sensors, in seconds. */
/* Find delta that yields best match. */

int best_match(int *samp_A, int *samp_B, int size)
{
    int delta = 0;
    double lowest_err = 0; /* Initialize only to eliminate compiler warnings. */
    int best_delta = -1;

    for (delta = 0; delta < size - WINDOW; delta++) {
        double this_err = 0; /* Error for this delta. */
        int i;

        /* Compute error by summing absolute value of differences. */
        for (i = delta; i < delta + WINDOW; i++)
            this_err += abs(samp_A[i] - samp_B[i-delta]);

        /* See if error for this delta is smaller than all previous. */
        if (best_delta == -1 || this_err < lowest_err) {
            lowest_err = this_err;
            best_delta = delta;
        }
    }

    return best_delta;
}
/* Routines below are stubs for interface and timing functions.  
   (That is, they allow the program to be run without having to 
   read real sensors or to sleep.) */

double get_current_time()
{
    struct timespec tp;
    /* clock_gettime is part of posix library. Can substitute time or, since 
       the return value isn’t really used, can comment out completely. */
    clock_gettime(CLOCK_REALTIME, &tp);
    return ((double)tp.tv_sec) + tp.tv_nsec * 1e9;
    /* return 0.0; */
}

/* The sleep_until routine does nothing since we’re not reading from 
   real sensors anyway. If this were an actual RTS, we would need 
   to call a precision sleep routine. (The sleep routine in the C library 
   is too coarse.) */

void sleep_until(double wake_time){return;}

/* Variables for simulating sensor output. */
static int ri_sample_history[SIZE]; /* Simulated sensor A output. */
static int ri_next_sample = -1;
static int ri_delta = SIZE / 17;
#define NOISE_MASK 0x3fff

/* Generate a random sample stream "for" sensor A.
   Samples have a uniform distribution over [0,2147483647] */

int readInterfaceA()
{
  if( ri_next_sample == -1 )
  {
    memset(ri_sample_history,0,sizeof(ri_sample_history));
    ri_next_sample = 0;
  }
  else
  {
    if( ++ri_next_sample == SIZE ) ri_next_sample = 0;
  }
  return (ri_sample_history[ri_next_sample] = random() & 0x3FFF);
}
/* Sample stream "for" sensor B is A’s stream delayed by constant ri_delta,  
   with some non-trivial noise added. */

int readInterfaceB() {
    int index = ri_next_sample - ri_delta;
    if( index < 0 ) index += SIZE;
    return ri_sample_history[ index ] +
        ( random() & NOISE_MASK ) - ( NOISE_MASK >> 1 );
}

/* This just writes to stdout if the count is in the proper range. The  
   count is used to only print out values of interest. (In a real system  
   we might display every value.) */

void display(double value){
    static count=0;
    count++;
    if( count > 700 )printf("Speed %f meters / second.\n",value);
    if( count == 720 )exit(0);
}
void cross_cor() /* Main Loop */
{
  int samp_A[SIZE];  /* Hold values read from sensor A. */
  int samp_B[SIZE];  /* Hold values read from sensor B. */
  double next_sample_time; /* Time at which sensors will be read. */
  int i;

  for(i=0; i<SIZE; i++) samp_A[i]=samp_B[i]=0;  /* Initialize arrays. */
  next_sample_time = get_current_time();      /* Initialize to now. */

  while(1){
    int a = readInterfaceA();       /* Read sensor A. */
    int b = readInterfaceB();       /* Read sensor B. */
    double speed;
    int delta;

    /* Make room for new samples. (There are faster ways of doing this.) */
    for(i=SIZE-1; i>0; i--){samp_A[i] = samp_A[i-1]; samp_B[i] = samp_B[i-1];}

    samp_A[0] = a; samp_B[0] = b;  /* Store new samples. */
    delta = best_match( samp_A, samp_B, SIZE );  /* Samples between sensors. */
    speed = DIST / ( SAMPLE_INTERVAL * delta );  /* Compute speed. */
    display( speed );              /* Display flow speed. */
    next_sample_time += SAMPLE_INTERVAL;  /* Compute when sensors next read. */
    sleep_until( next_sample_time );  /* Returns when time to sample again. */
  }
}
int main(int argv, char **argc)
{
    printf("The correct speed is: %f meters / second.\n",
            DIST / ( ri_delta * SAMPLE_INTERVAL ));
    cross_cor(); /* Never returns. */
    return 0;
}