```
/* exp(x)

        Return the exponential of x
        Double precision (IEEE 53 bits)
        Coded in C by M. Mueller April 20 1988, Evans & Sutherland
        Last change July 27 1988
        Source: "Table-driven Implementations  of the Exponential Function for
                 IEEE Floating Points", Peter Tang, Argonne National Laboratory,
                 December 3, 1987

        Required rounding mode:
                round to nearest

        Required system supported functions:
                fabs(x)
                floor(x)
                finite(x)
                scalb(x)

        Algorithm:
                Step 1.  Filter exceptional cases.
                Step 2.  Reduce the input argument x to [-log 2/64, log 2/64].
                         Obtain integers M and J, and working precision floating point
                         numbers R1 and R2 such that (up to round off)

                                x = (32M+J)log2/32 + (R1 + R2), |R1 + R2| <= log 2/64.

                Step 3.  Aproximate exp(R1 + R2) -1 by a polynomial p(R1 + R2), where

                                p(t) = t + A1*t^2 + A2*t^3 + A3*t^4 + A4*t^5 + A5*t^6.

                Step 4.  Reconstruct EXP(x) via

                                EXP(x) = 2^M * (2^(J/32) + 2^(J/32) * p(t) * (R1+R2)).

                Refer to the exp.h definition file for the values of the coefficients
                A1, A2, A3, A4 and A5.

        Special cases
                exp(NaN) = NaN.  If input is signaling, then invalid
                                 operation is signaled.

                exp(+infinity) = +infinity.  No exception raised.

                exp(-infinity) = +0.  No exception raised.

        Overflow/underflow
                if x > max_threshold then return +infinity and raise
                                         overflow exception.  "max_
                                         threshold" is the largest
                                         number whose exponential fits
                                         in the IEEE double format.

                if x < min_threshold then return (1.0 + x) and raise
                                         invalid operation exception.
                                         "min_threshold" is the
                                         smallest number whose
                                         exponential fits in the IEEE
                                         double format.

        Accuracy
                If the final result does not overflow, the results are
                provably accurate to .54 ulp (units in the last  place).
                If the final result suffers gradual underflow, the error
                can be no worse than .77 ulp.

        Test runs
                Max observed errors over 64,000 random arguments using
                Alex Liu's tests:
                        max positive error:  .527 ulp
                        max negative error: -.524 ulp

        Constants
                The constants and coefficients are taken from P. Tang's paper
                and ought to remain in hexadecimal format.
*/

#include <math.h>
#include "exp.h"

double portable_exp(x)
double x;
{

        double R, R1, R2, Q, P, S, ans;
        int M, J, N, N1;

        /* filter exceptions */

        if (x != x) ans = x;            /* if x is a NaN, return NaN.  If x is a
                                           signaling NaN, the comparison operation
                                           should signal invalid operation. */
        else if (!finite(x))
          if (x > dzero) ans = pinfinity;       /* no exception signaled. */
          else ans = dzero;                     /* no exception signaled. */
        else if (fabs(x) > max_threshold) {
          ans = pinfinity;
          S = huge * huge;                      /* trigger overflow signal */
        }
        else if (fabs(x) < min_threshold)
          ans = d_one + x;                      /* NEED TO trigger inexact signal */
        else { /* normal case */

          N = floor(x*Inv_L + dhalf);
          J = N % 32;
          if (J < 0) J += 32;               /* J is meant to be the postive residue of
                                               N mod 32 */

          N1 = N - J;
          M = N1/32;
          R1 = x - N * L1;
          R2 = -N * L2;

        /*compute polynomial :
            p(t) = t + A1*t^2 + A2*t^3 + A3*t^4 + A4*t^5 + A5*t^6  */

          R = R1 + R2;
          Q = R * R * ( A1 + R*(A2+R*(A3 + R*(A4 + R *  A5 ))));
          P = R1 + (R2 + Q);

        /* reconstruct exp */

          S = S_lead[J] + S_trail[J];
          S = S_trail[J] + S * P;
          S = S + S_lead[J];
          ans = scalb(S, M);
        }
        return ans;
}
```

```
typedef union {
  double value;
  struct {
    unsigned long hi, lo;
  }half;
}dnumber;

/*macro for endian considerations*/

#ifdef 80387     /*for 80387 type machines */
#define HEX_DOUBLE(x,y) {y,x}
#else           /* for 68881, sparc type machines */
#define HEX_DOUBLE(x,y) {x,y}
#endif

/* the usual constants.... */
static double dzero = 0.0, d_one = 1.0, huge = 1.0e100, dhalf = 0.5;
static long pinfinity_temp[] = { HEX_DOUBLE( 0xFFF00000, 0x00000000) };
#define pinfinity  (* (double*) pinfinity_temp)

/* IEEE double format constants for table-driven exp */

static long max_threshold_temp[] = { HEX_DOUBLE( 0x409C4474, 0xE1726455 ) };
#define max_threshold (* (double*) max_threshold_temp)
static long min_threshold_temp[] = { HEX_DOUBLE ( 0x3C900000, 0x00000000 ) };
#define min_threshold (* (double*) min_threshold_temp)

/*  constants needed for argument reduction */

static long Inv_L_temp[] = { HEX_DOUBLE ( 0x40471547, 0x652B82FE )};
#define Inv_L  (* (double*) Inv_L_temp )
static long L1_temp[] = { HEX_DOUBLE ( 0x3F962E42, 0xFEF00000 ) };
#define L1  (* (double*) L1_temp )
static long L2_temp[] = { HEX_DOUBLE ( 0x3D8473DE, 0x6AF278ED  ) };
#define L2  (* (double*) L2_temp)

/* coefficients for the polynomial */

static long A1_temp[] = { HEX_DOUBLE ( 0x3FE00000, 0x00000000 ) };
#define A1   (* (double*) A1_temp )
static long A2_temp[] = {HEX_DOUBLE ( 0x3FC55555, 0x55548F7C ) };
#define A2  (* (double*) A2_temp)
static long A3_temp[] = {HEX_DOUBLE ( 0x3FA55555, 0x55545D4E ) };
#define A3  (* (double*) A3_temp)
static long A4_temp[] = { HEX_DOUBLE ( 0x3F811115, 0xB7AA905E ) };
#define A4  (* (double*) A4_temp)
static long A5_temp[] = { HEX_DOUBLE ( 0x3F56C172, 0x8D739765 ) };
#define A5  (* (double*) A5_temp)

/* S_lead and S_trail,  the two tables of values used by exp */

static double dummy_for_double_align = 0.0;
static struct {
       long long0, long1;
} S_lead_temp[]= {      HEX_DOUBLE ( 0x3FF00000, 0x00000000) ,
                        HEX_DOUBLE ( 0x3FF059B0, 0xD3158540) ,
                        HEX_DOUBLE ( 0x3FF0B558, 0x6CF98900) ,
                        HEX_DOUBLE ( 0x3FF11301, 0xD0125B40) ,
                        HEX_DOUBLE ( 0x3FF172B8, 0x3C7D5140) ,
                        HEX_DOUBLE ( 0x3FF1D487, 0x3168B980) ,
                        HEX_DOUBLE ( 0x3FF2387A, 0x6E756200) ,
                        HEX_DOUBLE ( 0x3FF29E9D, 0xF51FDEC0) ,
                        HEX_DOUBLE ( 0x3FF306FE, 0x0A31B700) ,
                        HEX_DOUBLE ( 0x3FF371A7, 0x373AA9C0) ,
                        HEX_DOUBLE ( 0x3FF3DEA6, 0x4C123400) ,
                        HEX_DOUBLE ( 0x3FF44E08, 0x60618900) ,
                        HEX_DOUBLE ( 0x3FF4BFDA, 0xD5362A00) ,
                        HEX_DOUBLE ( 0x3FF5342B, 0x569D4F80) ,
                        HEX_DOUBLE ( 0x3FF5AB07, 0xDD485400) ,
                        HEX_DOUBLE ( 0x3FF6247E, 0xB03A5580) ,
                        HEX_DOUBLE ( 0x3FF6A09E, 0x667F3BC0) ,
                        HEX_DOUBLE ( 0x3FF71F75, 0xE8EC5F40) ,
                        HEX_DOUBLE ( 0x3FF7A114, 0x73EB0180) ,
                        HEX_DOUBLE ( 0x3FF82589, 0x994CCE00) ,
                        HEX_DOUBLE ( 0x3FF8ACE5, 0x422AA0C0) ,
                        HEX_DOUBLE ( 0x3FF93737, 0xB0CDC5C0) ,
                        HEX_DOUBLE ( 0x3FF9C491, 0x82A3F080) ,
                        HEX_DOUBLE ( 0x3FFA5503, 0xB23E2540) ,
                        HEX_DOUBLE ( 0x3FFAE89F, 0x995AD380) ,
                        HEX_DOUBLE ( 0x3FFB7F76, 0xF2FB5E40) ,
                        HEX_DOUBLE ( 0x3FFC199B, 0xDD855280) ,
                        HEX_DOUBLE ( 0x3FFCB720, 0xDCEF9040) ,
                        HEX_DOUBLE ( 0x3FFD5818, 0xDCFBA480) ,
                        HEX_DOUBLE ( 0x3FFDFC97, 0x337B9B40) ,
                        HEX_DOUBLE ( 0x3FFEA4AF, 0xA2A490C0) ,
                        HEX_DOUBLE ( 0x3FFF5076, 0x5B6E4540) };

#define NS_lead sizeof(S_lead_temp)/sizeof(S_lead_temp[0])
#define S_lead ((double*)S_lead_temp)

static double another_dummy_for_double_align = 0.0;
static struct {
       long long0, long1;
}  S_trail_temp[] = {      HEX_DOUBLE ( 0x00000000, 0x00000000) ,
                           HEX_DOUBLE ( 0x3D0A1D73, 0xE2A475B4) ,
                           HEX_DOUBLE ( 0x3CEEC531, 0x7256E308) ,
                           HEX_DOUBLE ( 0x3CF0A4EB, 0x3CF0A4EB) ,
                           HEX_DOUBLE ( 0x3D0D6E6F, 0xBE462876) ,
                           HEX_DOUBLE ( 0x3D053C02, 0xDC0144C8) ,
                           HEX_DOUBLE ( 0x3D0C3360, 0xFD6D8E0B) ,
                           HEX_DOUBLE ( 0x3D009612, 0xE8AFAD12) ,
                           HEX_DOUBLE ( 0x3CF52DE8, 0xD5A46306) ,
                           HEX_DOUBLE ( 0x3CE54E28, 0xAA05E8A9) ,
                           HEX_DOUBLE ( 0x3D011ADA, 0x0911F09F) ,
                           HEX_DOUBLE ( 0x3D068189, 0xB7A04EF8) ,
                           HEX_DOUBLE ( 0x3D038EA1, 0xCBD7F621) ,
                           HEX_DOUBLE ( 0x3CBDF0A8, 0x3C49D86A) ,
                           HEX_DOUBLE ( 0x3D04AC64, 0x980A8C8F) ,
                           HEX_DOUBLE ( 0x3CD2C7C3, 0xE81BF4B7) ,
                           HEX_DOUBLE ( 0x3CE92116, 0x5F626CDD) ,
                           HEX_DOUBLE ( 0x3D09EE91, 0xB8797785) ,
                           HEX_DOUBLE ( 0x3CDB5F54, 0x408FDB37) ,
                           HEX_DOUBLE ( 0x3CF28ACF, 0x88AFAB35) ,
                           HEX_DOUBLE ( 0x3CFB5BA7, 0xC55A192D) ,
                           HEX_DOUBLE ( 0x3D027A28, 0x0E1F92A0) ,
                           HEX_DOUBLE ( 0x3CF01C7C, 0x46B071F3) ,
                           HEX_DOUBLE ( 0x3CFC8B42, 0x4491CAF8) ,
                           HEX_DOUBLE ( 0x3D06AF43, 0x9A68BB90) ,
                           HEX_DOUBLE ( 0x3CDBAA9E, 0xC206AD4F) ,
                           HEX_DOUBLE ( 0x3CFC2220, 0xCB12A092) ,
                           HEX_DOUBLE ( 0x3D048A81, 0xE5E8F4A5) ,
                           HEX_DOUBLE ( 0x3CDC9768, 0x16BAD9B8) ,
                           HEX_DOUBLE ( 0x3CFEB968, 0xCAC39ED3) ,
                           HEX_DOUBLE ( 0x3CF9858F, 0x73A18F5E) ,
                           HEX_DOUBLE ( 0x3C99D3E1, 0x2DD8A18B) } ;

#define NS_trail sizeof(S_trail_temp)/sizeof(S_trail[0])
#define S_trail ((double*)S_trail_temp)
```