```
/*--------------------------------------------------------------------------*\
 |                                                                          |
 |  CALCULATOR                                                              |
 |                                                                          |
 |  date         : 2000, Aug 25                                             |
 |  release      : 3.4                                                      |
 |  release_date : 1999, May 22                                             |
 |  file         : calc.hh                                                  |
 |  author(s)    : Enrico Bertolazzi                                        |
 |  email        : enrico.bertolazzi@ing.unitn.it                           |
 |                                                                          |
 |  This program is free software; you can redistribute it and/or modify    |
 |  it under the terms of the GNU General Public License as published by    |
 |  the Free Software Foundation; either version 2, or (at your option)     |
 |  any later version.                                                      |
 |                                                                          |
 |  This program is distributed in the hope that it will be useful,         |
 |  but WITHOUT ANY WARRANTY; without even the implied warranty of          |
 |  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the           |
 |  GNU General Public License for more details.                            |
 |                                                                          |
 |  You should have received a copy of the GNU General Public License       |
 |  along with this program; if not, write to the Free Software             |
 |  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.               |
 |                                                                          |
 |  Copyright (C) 1999                                                      |
 |                                                                          |
 |            ___    ____  ___   _   _        ___    ____  ___   _   _      |
 |           /   \  /     /   \  \  /        /   \  /     /   \  \  /       |
 |          /____/ /__   /____/   \/        /____/ /__   /____/   \/        |
 |         /   \  /     /  \      /        /   \  /     /  \      /         |
 |        /____/ /____ /    \    /        /____/ /____ /    \    /          |
 |                                                                          |
 |      Enrico Bertolazzi                                                   |
 |      Dipartimento di Ingegneria Meccanica e Strutturale                  |
 |      Universita` degli Studi di Trento                                   |
 |      Via Mesiano 77, I-38050 Trento, Italy                               |
 |                                                                          |
\*--------------------------------------------------------------------------*/


/*
 * This code is inspired by the ``Expression Evaluator'' of
 * Mark Morley (c) 1992
 * morley@camosun.bc.ca
 *      Mark Morley
 *      3889 Mildred Street
 *      Victoria, BC  Canada
 *      V8Z 7G1
 *      (604) 479-7861
 */


# ifndef CALCULATOR_HH
# define CALCULATOR_HH

# ifdef DEBUG_CALCULATOR
# define CALCULATOR_DEBUG(A) { cout << A << endl ; }
# else
# define CALCULATOR_DEBUG(A)
# endif

# ifdef __DECCXX
  # include <math.h>
# else
  # include <cmath>
# endif

// standard includes I/O
# include <iostream>
# include <iomanip>
```

```
// STL lib
# include <string>
# include <map>

namespace calc {

  using namespace std ;

  template <typename T>
  struct CalculatorFun {

    static T my_cos(T x) { return cos(x) ; }
    static T my_sin(T x) { return sin(x) ; }
    static T my_tan(T x) { return tan(x) ; }

    static T my_acos(T x) { return acos(x) ; }
    static T my_asin(T x) { return asin(x) ; }
    static T my_atan(T x) { return atan(x) ; }

    static T my_cosh(T x) { return cosh(x) ; }
    static T my_sinh(T x) { return sinh(x) ; }
    static T my_tanh(T x) { return tanh(x) ; }

    static T my_exp(T x)   { return exp(x) ; }
    static T my_log(T x)   { return log(x) ; }
    static T my_log10(T x) { return log10(x) ; }
    static T my_sqrt(T x)  { return sqrt(x) ; }
    static T my_ceil(T x)  { return ceil(x) ; }
    static T my_floor(T x) { return floor(x) ; }

    static T my_atan2(T x, T y)  { return atan2(x,y) ; }
    static T my_pow(T x, T y)    { return pow(x,y) ; }

  public:

    typedef T (*Func1)(T) ;
    typedef T (*Func2)(T,T) ;

    typedef map<string,Func1>  map_fun1 ;
    typedef map<string,Func2>  map_fun2 ;
    typedef map<string,T>      map_real ;

    map_fun1 unary_fun ;
    map_fun2 binary_fun ;
    map_real variables ;

    void init() {
      unary_fun["cos"]    = my_cos ;
      unary_fun["sin"]    = my_sin ;
      unary_fun["tan"]    = my_tan ;

      unary_fun["acos"]   = my_acos ;
      unary_fun["asin"]   = my_asin ;
      unary_fun["atan"]   = my_atan ;

      unary_fun["cosh"]   = my_cosh ;
      unary_fun["sinh"]   = my_sinh ;
      unary_fun["tanh"]   = my_tanh ;

      unary_fun["exp"]    = my_exp ;
      unary_fun["log"]    = my_log ;
      unary_fun["log10"]  = my_log10 ;
      unary_fun["sqrt"]   = my_sqrt ;
      unary_fun["ceil"]   = my_ceil ;
      unary_fun["floor"]  = my_floor ;

      binary_fun["atan2"] = my_atan2 ;
      binary_fun["pow"]   = my_pow ;

      // predefined variables
      variables["pi"] = 3.14159265358979323846 ;
      variables["e"]  = 2.71828182845904523536 ;
    }
```

```
} ;


template <typename T_type, typename U_type>
class Calculator : public CalculatorFun<T_type> {

  typedef T_type               value_type ;
  typedef value_type*          pointer;
  typedef const value_type*    const_pointer ;
  typedef value_type&          reference ;
  typedef const value_type&    const_reference ;

  typedef U_type               index_type ;
  typedef index_type*          index_pointer;
  typedef const index_type*    index_const_pointer ;
  typedef index_type&          index_reference ;
  typedef const index_type&    index_const_reference ;

  typedef CalculatorFun<T_type>     CALCULATOR_FUN ;
  typedef Calculator<T_type,U_type> CALCULATOR ;

  typedef typename CALCULATOR_FUN::Func1 Func1 ;
  typedef typename CALCULATOR_FUN::Func2 Func2 ;

  typedef typename CALCULATOR_FUN::map_fun1  map_fun1 ;
  typedef typename CALCULATOR_FUN::map_fun2  map_fun2 ;
  typedef typename CALCULATOR_FUN::map_real  map_real ;

  typedef typename map_fun1::iterator       map_fun1_iterator ;
  typedef typename map_fun1::const_iterator map_fun1_const_iterator ;
  typedef typename map_fun2::iterator       map_fun2_iterator ;
  typedef typename map_fun2::const_iterator map_fun2_const_iterator ;
  typedef typename map_real::iterator       map_real_iterator ;
  typedef typename map_real::const_iterator map_real_const_iterator ;

  typedef enum {
    Number, Variable, Parameter,
    Plus, Minus, Times, Divide, Power,
    OpenPar, ClosePar,
    Assign, Comma, Unrecognized,
    EndOfExpression, EndOfString
  } Token_Type ;

  typedef enum {
    No_Error,
    Expected_OpenPar, Expected_ClosePar, Expected_Comma,
    Unknown_Variable, Bad_Position
  } ErrorCode ;

  static value_type max(value_type const a, value_type const b) { return a > b ? a : b ; }
  static value_type min(value_type const a, value_type const b) { return a < b ? a : b ; }

private:
  // error handling
  ErrorCode  error_found ;

  // token management
  Token_Type tt ; // token type
  string     tv ; // token value

  // internal use
  char const * string_in ;
  char const * ptr ;

  char const & get(void)       { return *ptr++ ; }
  char const & see(void) const { return *ptr ; }
  index_type pos(void)   const { return index_type(ptr - string_in) ; }

  Calculator(const CALCULATOR &) ;

public:
```

```
  Calculator(void) ;

  value_type parse(char const * const str, bool & err) ;

  value_type parse(string const & s, bool & err)
  { return parse( s. c_str(), err ) ; }

  void parse(istream &, ostream &) ;

  void report_error(ostream& s) ;

  value_type get(char const * const s, bool & exist) ;

  value_type get(string const & s, bool & exist)
  { return get(s . c_str(), exist) ; }

  void set(char const * const s, const_reference val)
  { variables[s] = val ; }

  void set(string const & s, const_reference val)
  { set(s . c_str(), val) ; }

  void print(ostream & s) const ;

private:

  value_type G0(void) ;
  value_type G1(void) ;
  value_type G2(void) ;
  value_type G3(void) ;
  value_type G4(void) ;
  value_type G5(void) ;
  void Next_Token(void) ;

} ;


template <typename T_type, typename U_type>
inline
ostream &
operator << (ostream & s, Calculator<T_type,U_type> const & pc) {
  pc . print(s) ;
  return s ;
}

template <typename T_type, typename U_type>
inline
Calculator<T_type,U_type>::Calculator(void) : CalculatorFun<value_type>() {

  CalculatorFun<value_type>::init() ;

  binary_fun["max"] = max ;
  binary_fun["min"] = min ;

}

template <typename T_type, typename U_type>
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::parse(char const * const str, bool & err) {
  string_in = ptr = str ;
  error_found = No_Error ;
  value_type res = 0 ;
  do { Next_Token() ; } while ( tt == EndOfExpression ) ;
  while ( tt != EndOfString && error_found == No_Error ) {
    res = G0() ;
    while ( tt == EndOfExpression ) Next_Token() ;
  }
  err = error_found != No_Error ;
  return res ;
}

template <typename T_type, typename U_type>
void
```

```cpp
Calculator<T_type,U_type>::parse(istream & in, ostream & out) {
  bool err ;
  string line ;
  while ( in . good() ) {
    getline( in, line) ;
    parse(line,err) ;
    if ( err ) report_error(out) ;
  }
}

template <typename T_type, typename U_type>
void
Calculator<T_type,U_type>::report_error(ostream& s) {
  switch (error_found) {
  case No_Error:
    s << "No error found" << endl ;
    return ;

  case Expected_OpenPar:
    s << "expect ``('' found ``" << tv << "''" << endl ;
    break;

  case Expected_ClosePar:
    s << "expect ``)'' found ``" << tv << "''" << endl ;
    break;

  case Expected_Comma:
    s << "expect ``,'' found ``" << tv << "''" << endl ;
    break;

  case Unknown_Variable:
    s << "unknown variable: " << tv << endl ;
    break ;

  case Bad_Position:
    s << "bad position for token: ``" << tv << "''" << endl ;
    break;
  }
  s << "            : " << string_in << endl << "            : "  ;
  for ( index_type i=pos() ; i > 1 ; --i) s << '-' ;
  s << '^' << endl ;
}

template <typename T_type, typename U_type>
inline
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::get(char const * const s, bool & exist) {
  map_real_const_iterator ii = variables . find(s) ;
  exist = ii != variables . end() ;
  if ( exist ) return ii -> second ;
  else         return 0 ;
}

template <typename T_type, typename U_type>
void
Calculator<T_type,U_type>::print(ostream & s) const {
  map_real_const_iterator ii ;
  map_fun1_const_iterator f1 ;
  map_fun2_const_iterator f2 ;

  s << endl << "UNARY FUNCTIONS" << endl ;
  for ( f1 = unary_fun . begin() ; f1 != unary_fun . end() ; ++f1 )
    s << f1 -> first << ", " ;

  s << endl << endl << "BINARY FUNCTIONS" << endl ;
  for ( f2 = binary_fun . begin() ; f2 != binary_fun . end() ; ++f2 )
    s << f2 -> first << ", " ;

  s << endl << endl << "VARIABLES" << endl ;
  for ( ii = variables . begin() ; ii != variables . end() ; ++ii )
    s << ii -> first << " = " << ii -> second << endl ;

  s << "END LIST" << endl ;
```

```
}

template <typename T_type, typename U_type>
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::G0() {

  if ( tt == EndOfExpression || tt == EndOfString ) return 0 ;

  char const * const bf_ptr = ptr ; // save pointer
  string     name  = tv ;
  Token_Type token = tt ;

  if ( tt == Variable ) {
    Next_Token() ;
    if ( tt == Assign ) {  // handle assign
      Next_Token() ;       // eat assign
      return variables[name] = G1() ; // assign value
    }
  }

  ptr = bf_ptr ; // restore pointer
  tv  = name ;
  tt  = token ;

  CALCULATOR_DEBUG( "G0: " << tv )
  return G1() ; // handle immediate evaluation
}

// handle binary + and -
template <typename T_type, typename U_type>
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::G1() {
  CALCULATOR_DEBUG( ">>G1: " << tv )
  value_type lval = G2() ;
  if ( error_found == No_Error ) {
    switch ( tt ) {
    case Plus:  Next_Token() ; return lval + G1() ;
    case Minus: Next_Token() ; return lval - G1() ;
    default: break ;
    }
  }
  return lval ;
}

// handles * and /
template <typename T_type, typename U_type>
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::G2() {
  CALCULATOR_DEBUG( ">>>>G2: " << tv )
  value_type lval = G3() ;
  if ( error_found == No_Error ) {
    switch ( tt ) {
    case Times:  Next_Token() ; return lval * G2() ;
    case Divide: Next_Token() ; return lval / G2() ;
    default: break ;
    }
  }
  return lval ;
}

// handles ^ operator
template <typename T_type, typename U_type>
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::G3() {
  CALCULATOR_DEBUG( ">>>>>>>>G3: " << tv )
  value_type lval = G4() ;
  if ( error_found == No_Error && tt == Power )
    { Next_Token() ; return pow( lval, G4() ) ; }
  return lval ;
}

// handles any unary + or - signs
template <typename T_type, typename U_type>
```

```
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::G4() {
  CALCULATOR_DEBUG( ">>>>>>>>>>G4: " ≪ tv )
  value_type val = 0 ;
  if ( error_found == No_Error ) {
    switch ( tt ) {
    case Plus:  Next_Token() ; break ;
    case Minus: Next_Token() ; return val = - G5() ;
    default: break ;
    }
    val = G5() ;
  }
  return val ;
}

// handles numbers, variables, functions and parentesis
template <typename T_type, typename U_type>
typename Calculator<T_type,U_type>::value_type
Calculator<T_type,U_type>::G5() {
  CALCULATOR_DEBUG( ">>>>>>>>>>>>G5: " ≪ tv )
  value_type val = 0 ;
  if ( error_found ≠ No_Error ) return val ;

  if ( tt == OpenPar ) { // handle ( ... )

    Next_Token() ; // eat (
    val = G0() ;
    if ( tt ≠ ClosePar )
      { error_found = Expected_ClosePar ; goto error_found ; }
      Next_Token() ; // eat )

  } else if ( tt == Number ) {

    val = atof(tv.c_str()) ;
    Next_Token() ; // eat number

  } else if ( tt == Variable ) {

    map_real_iterator iv = variables  . find(tv) ;
    map_fun1_iterator f1 = unary_fun  . find(tv) ;
    map_fun2_iterator f2 = binary_fun . find(tv) ;

    if ( iv ≠ variables . end() ) {
      val = iv → second ;
      Next_Token() ; // eat token
    } else if ( f1 ≠ unary_fun . end() ) {
      Next_Token() ; // expect (

      if ( tt ≠ OpenPar )
        { error_found = Expected_OpenPar ; goto error_found ; }

      Next_Token() ; // eat (
      val = G0() ;

      if ( tt ≠ ClosePar )
        { error_found = Expected_ClosePar ; goto error_found ; }

      val = f1 → second(val) ;
      Next_Token() ; // eat )

    } else if ( f2 ≠ binary_fun . end() ) {

      Next_Token() ; // expect (

      if ( tt ≠ OpenPar )
        { error_found = Expected_OpenPar ; goto error_found ; }

      Next_Token() ; // eat (
      value_type v1 = G0() ;

      if ( tt ≠ Comma )
        { error_found = Expected_Comma ; goto error_found ; }
```

```
      Next_Token() ; // eat ,
      value_type v2 = GO() ;

      if ( tt != ClosePar )
        { error_found = Expected_ClosePar ; goto error_found ; }

      val = f2 -> second(v1,v2) ;
      Next_Token() ; // eat )

    } else {
      error_found = Unknown_Variable ;
    }
  } else {
    error_found = Bad_Position ;
  }

 error_found:
  return val ;
}

template <typename T_type, typename U_type>
void
Calculator<T_type,U_type>::Next_Token() { // eat separators
  tt = EndOfExpression ;
  tv = "" ;

  // EAT SEPARATORS
  while ( see() == '\t' || see() == ' ' || see() == '\n' ) get() ;

  // SKIP COMMENTS
  if ( see() == '#' ) {
    while ( see() != '\n' && see() != '\0' ) get() ;
    return ;
  }

  if ( isalpha(see()) ) {
    do { tv += get() ; } while ( isalnum(see()) || see() == '_' ) ;
    tt = Variable ;
  } else if ( isdigit(see()) ) {
    tt = Number ;
    do { tv += get() ; } while ( isdigit(see()) ) ;
    if ( see() == '.' ) {
      do { tv += get() ; } while ( isdigit(see()) ) ;
    }
    if ( see() == 'e' || see() == 'E' ) {
      tv += get() ;
      if ( see() == '+' || see() == '-' ) tv += get() ;
      if ( isdigit(see()) ) {
        do { tv += get() ; } while ( isdigit(see()) ) ;
      } else {
        tt = Unrecognized ;
      }
    }
  } else {
    tv = see() ;
    switch (see()) {
    case '+' : tt = Plus            ; break ;
    case '-' : tt = Minus           ; break ;
    case '*' : tt = Times           ; break ;
    case '/' : tt = Divide          ; break ;
    case '^' : tt = Power           ; break ;
    case '(' : tt = OpenPar         ; break ;
    case ')' : tt = ClosePar        ; break ;
    case '=' : tt = Assign          ; break ;
    case ',' : tt = Comma           ; break ;
    case '\0': tt = EndOfString     ; break ;
    case ';' : tt = EndOfExpression ; break ;
    default  : tt = Unrecognized    ; break ;
    }
    get() ;
  }
  CALCULATOR_DEBUG( "get token: " << tv )
}
```

```
} // end namespace

# endif

// end of file: calc.hh
```