Calculator: an Expression Evaluator Class

Enrico Bertolazzi

Dipartimento di Ingegneria Meccanica e Strutturale
Università degli Studi di Trento
via Mesiano 77, I – 38050 Trento, Italia
Enrico.Bertolazzi@ing.unitn.it

Release 4.1

Abstract

This document presents a C++ class for run-time evaluation of simple symbolic expressions. This is particularly suitable to facilitates the input process from free format files. The C++ compiler must support templates, namespaces and exceptions.

1 Including the class Calculator<...>

The software consists of a single header file `calc.hh`. There is no `.a` or `.so` files to be linked. The `calc.hh` facilities are available by the following inclusion statement:

```
# include "calc.hh"
using namespace calc_load ;
```

2 Instantiating an expression evaluator.

The instantiation process of an expression evaluator requires the specification of its template type. The statement

```
Calculator<double> ee ;
```

instantiates the object `ee` which in this case is an expression evaluator operating on `double`s.

3 Parsing a string

The object `ee` can now be used to parse simple symbolic expressions input as strings,

```
bool err = ee . parse("1+sin(3.5)/4") ;
```

the boolean variable `err` is set *true* when a parsing error is found. The function `get_value()` returns the value resulting from the parsing process:

```
double res = ee . get_value() ;
```

When an error is produced, an error report explaining what went wrong can be print out by the class method `report_error`, as for example in the following piece of source:

```
ee . report_error(cout) ;
```

The input string to the method `parse` can contain more than one expression. Individual expressions must be separated by the end-of-statement separator "`;`".

```
err = ee . parse("a=1+sin(3) ; b=1-sin(3) ; sqrt(a*b)") ;
```

In the case of a multiple expression parsing, the class method `get_value()` returns only the last evaluated value. In the previous example this is `a*b`. Notice also that the assignement symbol "`=`" makes possible the memorization of intermediate expression values in other variables.

4 Operators

The expression evaluator uses a fixed number of operators, that are listed in order of precedence as follows

- `#` the rest of the string is a comment (and clearly ignored);
- `+`,`-` binary addition and subtraction, e.g. `10+2; 3-4.2`;
- `*`,`/` binary multiplication and division, e.g. `2.3*4.9; 2/4`;
- `^` power, e.g. `10^4` (results 10000);
- `+`,`-` unary `+` and `-`, e.g. `+120; 12+-12`;
- `(`,`)` parenthesis are use to change operator precedence; for example the expression `12-(2-2)` evaluates to `12` while `12-2-2` evaluates to `8`;
- `;` expression separator;
- `=` assignement operator.

5 Symbolic Constants

Two symbolic constants are available whose value is assigned by default:

e	2.71828182845904523536
pi	3.14159265358979323846

They can be used in symbolic expressions like the following one:

```
e + sin(pi*0.5) ;
```

6 Predefined Functions

In the previous section we used the function `sin`. There are a number of predefined functions which can be used in symbolic expressions. The following table lists them.

`abs(x)`	absolute value of `x`
`pos(x)`	positive part of `x`
`neg(x)`	negative part of `x`
`cos(x)`	cosine of `x`
`sin(x)`	sine of `x`
`tan(x)`	tangent of `x`
`asin(x)`	arcsin of `x`
`acos(x)`	arccos of `x`
`atan(x)`	arctan of `x`
`cosh(x)`	hyperbolic cosine of `x`
`sinh(x)`	hyperbolic sine of `x`
`tanh(x)`	hyperbolic tangent of `x`
`exp(x)`	exponential of `x`
`log(x)`	natural logarithm of `x`
`log10(x)`	base 10 logarithm of `x`
`sqrt(x)`	square root of of `x`
`ceil(x)`	least integer over `x`
`floor(x)`	great integer under `x`
`max(x,y)`	maximum of `{x,y}`
`min(x,y)`	minimum of `{x,y}`
`atan2(x,y)`	arctan of `y/x`
`pow(x,y)`	power x^y

7 Defining new functions

A new function can be introduced into the expression evaluator by defining it as static and then passing the evaluator its name and address pointer by using the two evaluator facilities `set_unary_fun` and `set_binary_fun`. The following example illustrates the mechanism. Let us first define the two static functions:

```
static double power2(double const a)
{ return a*a ; }

static double add(double const a, double const b)
{ return a+b ; }
```

Then let us add `power2` and `add` to the current expression evaluator as follows:

```
ee . set_unary_fun("power2",power2) ;
ee . set_binary_fun("add",add) ;
```

These new functions can now be invoked in symbolic expressions as the predefined ones:

```
err = ee . parse("power2(add(2,e))") ;
```

The expression evaluator is capable of handling only unary and binary functions, i.e. functions with one or two arguments.

8 Defining new variables

New variables can be introduced into the expression evaluator by using the method `set` or the assignement operator. For example, the following piece of source code defines the new variable `abc` and initialize it to the value `1/3`

```
err = ee . parse("abc = 1/3") ;
ee . set("abc",1.0/3.0) ;
```

The first statements uses the parse method and the assignement operator `=` of the expression evaluator. The parse method evaluates the expression on the right of `=` and then assigns the parsing result to the variable on the right. If the variable should not exist it would be created and assigned.

The second statement creates – if needed – and assigns directly the variable. Once created and initilized, the variable can be used in the next operations; for example

```
err = ee . parse("zz = abc*sin(3)/(1+abc)") ;
```

In this case the new variable `zz` is also created. A variable is a string which always begins with a letter and may be followed by any sequence of alphanumeric characters, such as numbers, letters or the underscore symbols like `_`.

The `exist` return true if its argument is a defined variable, as in

```
bool ex1 = ee . exist("abc") ;
bool ex2 = ee . exist("pippo") ;
```

In this case `ex1` is set to true and `ex2` to false. It is possible to get out the value of a variable,

```
double val1 = ee . get("abc") ;
double val2 = ee . get("pippo") ;
```

The value of `val1` is 0.333333 while `val2` is null because the variable `pippo` does not exist.

9 Parsing a file

The expression evaluator can be used to parse a complete file. The parsing process proceeds by reading the file one line at a time and parsing it. Use the method

```
ee . parse_file("filename", true) ;
```

The boolean `true` in the second entry asks the expression evaluator for proceeding in verbose mode, that is for printing out on `cerr` input errors when detected. If the flag was set to `false`, reading would proceed silently and errors ignored.

For example, consider the following input file:

```
# this is a comment line
gamma = 1.4
# set left state
rin = 1 # density
vin = 0 # velocity
pin = 1 # pressure
ein = rin*vin*vin/2+pin/(gamma-1)

# set right state
rout = 0.125
vout = 0
pout = 0.1
eout = rin*vin*vin/2+pout/(gamma-1)
```

If a program needs as input parameters `rin`, `vin`, `ein`, `rout`, `vout`, `eout` the following piece of code

```
ee . parse_file("file.data", true) ;
double rin  = ee . get("rin") ;
double vin  = ee . get("vin") ;
double ein  = ee . get("ein") ;
double rout = ee . get("rout") ;
double vout = ee . get("vout") ;
double eout = ee . get("eout") ;
```

does the work. The advantages of using expression evaluators in reading input files are multiples:

- a free input format is easily usable;
- comments can be added everywhere therein;
- simple computations may be inserted as part of an input file.