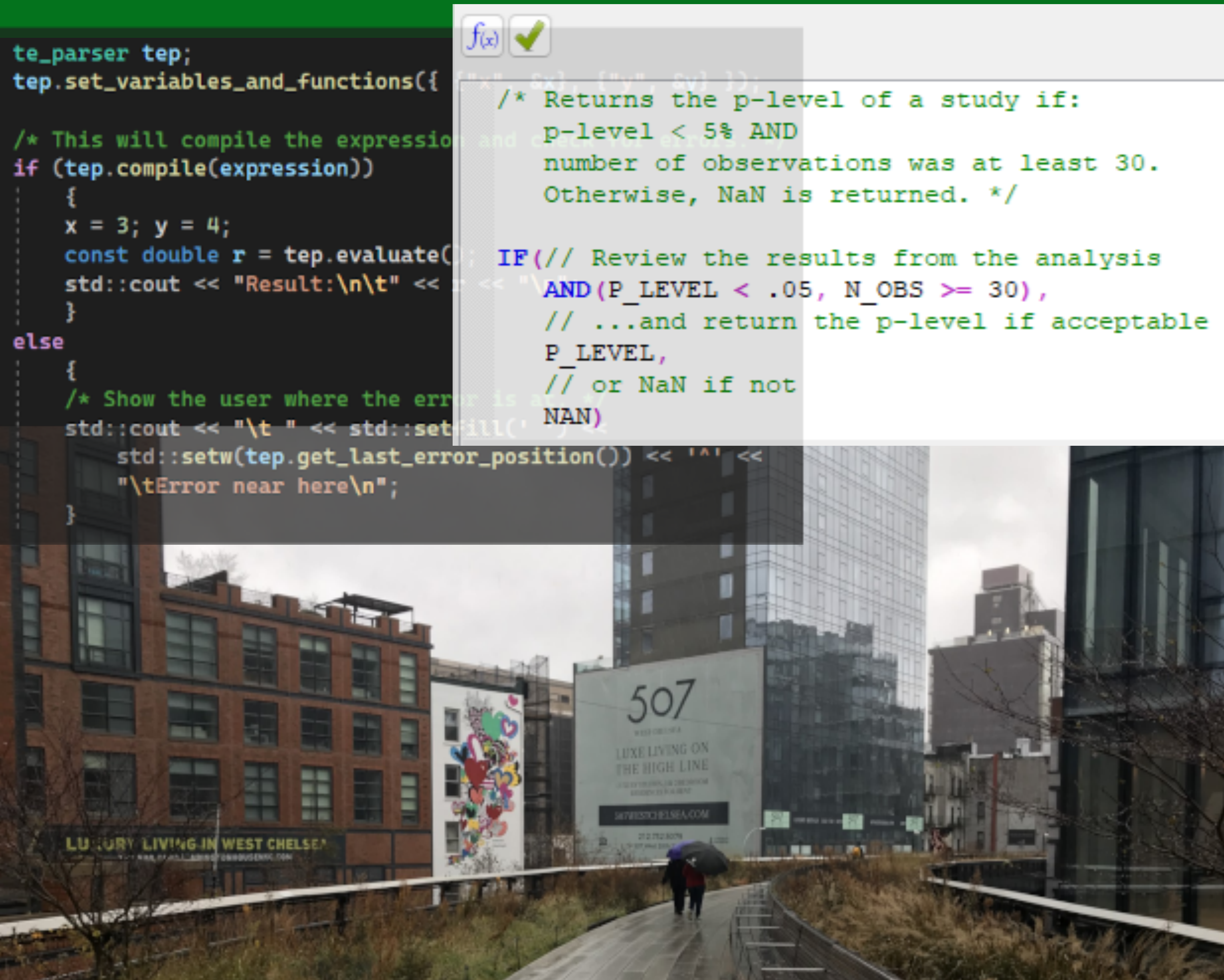
TinyExpr++

C++ formula parsing and evaluation system



The Comprehensive User Reference & Programming Manual

Blake Madden

TinyExpr++

User Reference & Programming Manual

Blake Madden

TinyExpr++

User Reference & Programming Manual

Copyright © 2023 Blake Madden

Some rights reserved.

Published in the United States

This book is distributed under a Creative Commons Attribution-NonCommercial-Sharealike 4.0 License.

That means you are free:

- **To Share** – copy and redistribute the material in any medium or format.
- **To Adapt** – remix, transform, and build upon the material.

The licensor cannot revoke these freedoms as long as you follow the license terms:

- **Attribution** – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** – You may not use the material for commercial purposes.
- **Share Alike** – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions —You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Contents

List of Tables

Chapter 1

Overview

This is the programming manual for *TinyExpr++*, the C++ version of the *TinyExpr* (Winkle) formula parsing library. (This manual includes documentation from *TinyExpr* by Lewis Van Winkle.)

TinyExpr++ is a small parser and evaluation library for solving math expressions from C++. It's open-source, free, easy-to-use, and self-contained in a single source and header file pair.

Features

- C++17 with no dependencies.
- Single source file and header file.
- Simple and fast.
- Implements standard operator precedence.
- Implements logical and comparison operators.
- Exposes standard C math functions (`sin`, `sqrt`, `ln`, etc.), as well as some *Excel*-like functions (e.g., `AVERAGE()` and `IF()`).
- Can add custom functions and variables easily.
- Can add a custom handler to resolve unknown variables.
- Can bind constants at eval-time.
- Supports variadic functions (taking between 1-7 arguments).
- Case insensitive.
- Supports non-US formulas (e.g., `POW(2,2; 2)` instead of `POW(2.2, 2)`).
- Supports C and C++ style comments within math expressions.
- Released under the zlib license - free for nearly any use.
- Easy to use and integrate with your code.
- Thread-safe; parser is in a self-contained object.

Part I

User Guide

Chapter 2

Usage

TinyExpr++ is a formula-solving library which accepts math and logic expressions such as:

```
ABS(((5+2) / (ABS(-2))) * -9 + 2) - 5^2
```

Applications using *TinyExpr++* may provide context-specific variables that you can use in your expressions. For example, in a spreadsheet application, values representing cells such as `C1` and `D2` may be available. This would enable the use of expressions such as:

```
SUM(C1, C2, C3, D1, D2, D3)
```

As another example, in a statistical program, the values `N_OBS` and `P_LEVEL` may be available. This would make an expression such as this possible:

```
IF(AND(P_LEVEL < .05, N_OBS >= 30),
   P_LEVEL,
   NAN)
```

Please consult your application's documentation for which custom variables and functions it may provide for its formulas.

Chapter 3

Operators

The following operators are supported within math expressions:

Table 3.1: Operators

Operator	Description
*	Multiplication.
/	Division.
%	Modulus: Divides two values and returns the remainder.
+	Addition.
-	Subtraction.
^	Exponentiation. The number in front of ^ is the base, the number after it is the power to raise it to.
**	Exponentiation. (This is an alias for ^)
=	Equals.
<	Less than.
>	Greater than.
<>	Not equal to.
!=	Not equal to. (This is an alias for <>)
>=	Greater than or equal to.
<=	Less than or equal to.
&	Logical conjunction (AND).
|	Logical alternative (OR).
( )	Groups sub-expressions, overriding the order of operations.

For operators, the order of precedence is:

Operator	Description
( )	Instructions in parentheses are executed first.
^	Exponentiation.
*, /, and %	Multiplication, division, and modulus.
+ and -	Addition and subtraction.

For example, the following:

$$5 + 5 + 5/2$$

Will yield `12.5`. `5/2` is executed first, then added to the other fives. However, by using parentheses:

$$(5+5+5)/2$$

You can override it so that the additions happen first (resulting in `15`), followed by the division (finally yielding `7.5`). Likewise, `(2+5)^2` will yield `49` (`7` squared), while `2+5^2` will yield `27` (`5` squared, plus `2`).

Compatability Note

The `%` character acts as a modulus operator in *TinyExpr++*, which is different from most spreadsheet programs. In programs such as *LibreOffice Calc* and *Excel*, `%` is used to convert a number to a percentage. For example, `=20%` would yield `0.20` in Excel. In *TinyExpr++*, however, `20%` will result in a syntax error as it is expecting a binary (modulus) operation.

Chapter 4

Functions

The following built-in functions are available:

Table 4.1: Math Functions

Function	Description
ABS(Number)	Absolute value of *Number*.
ACOS(Number)	Returns the arccosine, or inverse cosine, of *Number*. The arccosine is the angle whose cosine is number. The returned angle is given in radians in the range `0` (zero) to `PI`.
ASIN(Number)	Returns the arcsine, or inverse sine function, of *Number*, where -1 <= *Number* <= 1. The arcsine is the angle whose sine is *Number*. The returned angle is given in radians where -pi/2 <= angle <= pi/2.
ATAN(x)	Returns the principal value of the arc tangent of x, expressed in radians..
ATAN2(y, x)	Returns the principal value of the arc tangent of y,x, expressed in radians.
BITLSHIFT(Number, ShiftAmount)	Returns *Number* left shifted by the specified number (*ShiftAmount*) of bits.
BITRSHIFT(Number, ShiftAmount)	Returns *Number* right shifted by the specified number (*ShiftAmount*) of bits.
CEIL(Number)	Smallest integer not less than *Number*. `CEIL(-3.2)` = -3 `CEIL(3.2)` = 4
CLAMP(Number, Start, End)	Constrains *Number* within the range of *Start* and *End*.
COMBIN(Number, NumberChosen)	Returns the number of combinations for a given number (*NumberChosen*) of items from *Number* of items. Note that for combinations, order of items is not important.
COS(Number)	Cosine of the angle *Number* in radians.
COSH(Number)	Hyperbolic cosine of *Number*.
COT(Number)	Cotangent of *Number*.
EXP(Number)	Euler to the power of *Number*.
FAC(Number)	Returns the factorial of *Number*. The factorial of *Number* is equal to 1*2*3*...* *Number*

Function	Description
FACT(Number)	Alias for `FAC()`
FLOOR(Number)	Returns the largest integer not greater than *Number*. `FLOOR(-3.2)` = -4 `FLOOR(3.2)` = 3
LN(Number)	Natural logarithm of *Number* (base Euler).
LOG10(Number)	Common logarithm of *Number* (base 10).
MIN(Value1, Value2, …)	Returns the lowest value from a specified range of values.
MAX(Value1, Value2, …)	Returns the highest value from a specified range of values.
MOD(Number, Divisor)	Returns the remainder after *Number* is divided by *Divisor*. The result has the same sign as divisor.
NCR(Number, NumberChosen)	Alias for `COMBIN()`.
NPR(Number, NumberChosen)	Alias for `PERMUT()`.
PERMUT(Number, NumberChosen)	Returns the number of permutations for a given number (*NumberChosen*) of items that can be selected *Number* of items. A permutation is any set of items where order is important. (This differs from combinations, where order is not important).
POW(Base, Exponent)	Raises *Base* to any power. For fractional exponents, *Base* must be greater than 0.
POWER(Base, Exponent)	Alias for `POW()`.
RAND()	Generates a random floating point number within the range of `0` and `1`.
ROUND(Number, NumDigits)	*Number* rounded to *NumDigits* decimal places. If *NumDigits* is negative, then *Number* is rounded to the left of the decimal point. (*NumDigits* is optional and defaults to zero.) `ROUND(-11.6, 0)` = 12 `ROUND(-11.6)` = 12 `ROUND(1.5, 0)` = 2 `ROUND(1.55, 1)` = 1.6 `ROUND(3.1415, 3)` = 3.142 `ROUND(-50.55, -2)` = -100
SIGN(Number)	Returns the sign of *Number*. Returns `1` if *Number* is positive, zero (`0`) if *Number* is `0`, and `-1` if *Number* is negative.
SIN(Number)	Sine of the angle *Number* in radians.
SINH(Number)	Hyperbolic sine of *Number*.
SQRT(Number)	Square root of *Number*.
TAN(Number)	Tangent of *Number*.
TGAMMA(Number)	Returns the gamma function of *Number*.
TRUNC(Number)	Discards the fractional part of *Number*. `TRUNC(-3.2)` = -3 `TRUNC(3.2)` = 3

Table 4.2: Statistical Functions

Function	Description
AVGERAGE(Value1, Value2,...)	Returns the mean of a specified range of values.
SUM(Value1, Value2,...)	Returns the sum of a specified range of values.

Table 4.3: Logic Functions

Function	Description
AND(Value1, Value2, ...)	Returns true if all conditions are true.
IF(Condition, ValueIfTrue, ValueIfFalse)	If *Condition* is true (non-zero), then *ValueIfTrue* is returned; otherwise, *ValueIfFalse* is returned.
NOT(Value)	Returns the logical negation of *Value.*
OR(Value1, Value2, ...)	Returns true if any condition is true.

Chapter 5

Constants

The following mathematical and logical constants are available:

Table 5.1: Math Constants

Constant	Value
E	Euler's number (`2.71828182845904523536`)
NAN	NaN (Not-a-Number)
PI	pi (`3.14159265358979323846`)

Table 5.2: Logical Constants

Constant	Value
TRUE	`1`
FALSE	`0`

The following number formats are supported:

Table 5.3: Number Formats

Format	Example
Scientific notation	`1e3` for `1000`

Chapter 6

Comments

Comments can be embedded within an expression to clarify its intent. C/C++ style comments are supported, which provide:

- multi-line comments (text within a pair of `/*` and `*/`).
- single line comments (everything after a `//` until the end of the current line).

For example, assuming that the variables `P_LEVEL` and `N_OBS` have been defined within the parser, an expression such as this could be used:

```
/* Returns the p-level of a study if:
   p-level < 5% AND
   number of observations was at least 30.
   Otherwise, NaN is returned. */

IF(// Review the results from the analysis
   AND(P_LEVEL < .05, N_OBS >= 30),
   // ...and return the p-level if acceptable
   P_LEVEL,
   // or NaN if not
   NAN)
```

Part II

Developer Guide

Chapter 7

Building

TinyExpr++ is self-contained in two files: `tinyexpr.cpp` and `tinyexpr.h`. To use *TinyExpr++*, add those two files to your project.

The API documentation can be built using the following:

```
doxygen docs/Doxyfile
```

Requirements

TinyExpr++ must be compiled as C++17.

MSVC, GCC, and Clang compilers are supported.

Chapter 8

Usage

TinyExpr++'s `te_parser` class defines these functions:

```
double evaluate(const std::string_view expression);
double get_result();
bool success();
int64_t get_last_error_position();
std::string get_last_error_message();
set_variables_and_functions(const std::set<te_variable>& vars);
std::set<te_variable>& get_variables_and_functions();
add_variable_or_function(const te_variable& var);
set_unknown_symbol_resolver(te_usr_variant_type usr);
get_decimal_separator();
set_decimal_separator();
get_list_separator();
set_list_separator();
```

`evaluate()` takes an expression and immediately returns the result. If there is a parse error, then it returns NaN (which can be verified by using `std::isnan()`). (`success()` will also return false.)

`get_result()` can be called anytime afterwards to retrieve the result from `evaluate()`.

`set_variables_and_functions()`, `get_variables_and_functions()`, and `add_variable_or_function()` are used to add custom variables and functions to the parser.

`set_unknown_symbol_resolver()` is used to provide a custom function to resolve unknown symbols in an expression. (Refer to ch. 11 for further details.)

`get_decimal_separator()`/`set_decimal_separator()` and `get_list_separator()`/`set_list_separator()` can be used to parse non-US formatted formulas.

Example:

```
te_parser tep;

// Returns 10, error position is set to te_parser::npos (i.e., no error).
double result = tep.evaluate("(5+5)");
// Returns NaN, error position is set to 3.
double result2 = tep.evaluate("(5+5");
```

You can also provide `set_variables_and_functions()` a list of constants, bound variables, and function pointers/lambdas. `evaluate()` will then evaluate expressions using these variables and functions.

Error Handling

TinyExpr++ will throw exceptions when:

- An illegal character is specified in a custom function or variable name.
- An illegal character is provided as a list or decimal separator.
- The same character is provided as both the list and decimal separator.

It is recommended to wrap the following functions in `try`/`catch` blocks to handle these exceptions:

- `compile()`
- `evaluate()`
- `set_variables_and_functions()`
- `add_variable_or_function()`
- `set_decimal_separator()`
- `set_list_separator()`

Syntax and calculation errors are trapped within calls to `compile()` and `evaluate()`. Error information can be retrieved afterwards by calling the following:

- `success()`: returns whether the last parse was successful or not.
- `get_last_error_position()`: returns the 0-based index of where in the expression the parse failed (useful for syntax errors). If there was no parse error, then this will return `te_parser::npos`.
- `get_last_error_message()`: returns a more detailed message for some calculation errors (e.g., division by zero).

Example:

```cpp
#include "tinyexpr.h"
#include <iostream>

double x{ 0 }, y{ 0 };
// Store variable names and pointers.
te_parser tep;
tep.set_variables_and_functions({{"x", &x}, {"y", &y}});

// Compile the expression with variables.
auto result = tep.evaluate("sqrt(x^2+y^2)");

if (tep.success())
    {
    x = 3; y = 4;
    // Will use the previously used expression, returns 5.
    const double h1 = tep.evaluate();

    x = 5; y = 12;
    // Returns 13.
    const double h2 = tep.evaluate();
    }
else
    {
    std::cout << "Parse error at " <<
        std::to_string(tep.get_last_error_position()) << "\n";
    }
```

Along with positional and message information, the return value of a parse can also indicate failure. When an evaluation fails, the parser will return NaN (i.e., `std::numeric_limits<double>::quiet_NaN()`) as the result. NaN values can be verified using the standard function `std::isnan()`.

Chapter 9

Examples

The following are examples demonstrating how to use *TinyExpr++*.

Example 1

```
include "tinyexpr.h"
#include <iostream>

int main(int argc, char *argv[])
    {
    te_parser tep;
    const char *c = "sqrt(5^2+7^2+11^2+(8-2)^2)";
    double r = tep.evaluate(c);
    std::cout << "The expression:\n\t" <<
        c << "\nevaluates to:\n\t" << r << "\n";
    return EXIT_SUCCESS;
    }
```

Example 2: Binding Custom Variables

```cpp
#include "tinyexpr.h"
#include <iostream>
#include <iomanip>

int main(int argc, char* argv[])
    {
    if (argc < 2)
        {
        std::cout << "Usage: example \"expression\"\n";
        return EXIT_SUCCESS;
        }

    const char* expression = argv[1];
    std::cout << "Evaluating:\n\t" << expression << "\n";

    /* This shows an example where the variables
       x and y are bound at eval-time. */
    double x{ 0 }, y{ 0 };
    // Store variable names and pointers.
    te_parser tep;
    tep.set_variables_and_functions({ {"x", &x}, {"y", &y} });

    /* This will compile the expression and check for errors. */
    if (tep.compile(expression))
        {
        /* The variables can be changed here, and eval can be called as many
           times as you like. This is fairly efficient because the parsing has
           already been done. */
        x = 3; y = 4;
        const double r = tep.evaluate();
        std::cout << "Result:\n\t" << r << "\n";
        }
    else
        {
        /* Show the user where the error is at. */
        std::cout << "\t " << std::setfill(' ') <<
            std::setw(tep.get_last_error_position()) << '^' <<
            "\tError near here\n";
        }

    return EXIT_SUCCESS;
    }
```

Example 3: Calling a Free Function

```
#include "tinyexpr.h"
#include <iostream>
#include <iomanip>

/* An example of calling a free function. */
double my_sum(double a, double b)
    {
    std::cout << "Called C function with " <<
        a << " and " << b << ".\n";
    return a + b;
    }

int main(int argc, char *argv[])
    {
    const char *expression = "mysum(5, 6)";
    std::cout << "Evaluating:\n\t" << expression << "\n";

    te_parser tep;
    tep.set_variables_and_functions({{"mysum", my_sum}});

    if (tep.compile(expression))
        {
        const double r = tep.evaluate();
        std::cout << "Result:\n\t" << r << "\n";
        }
    else
        {
        /* Show the user where the error is at. */
        std::cout << "\t " << std::setfill(' ') <<
            std::setw(tep.get_last_error_position()) << '^' <<
            "\tError near here\n";
        }

    return EXIT_SUCCESS;
    }
```

Example 4: Non-US Formatted Formulas

```
#include "tinyexpr.h"
#include <iostream>
#include <iomanip>
#include <locale>
#include <clocale>

int main(int argc, char *argv[])
    {
    /* Set locale to German.
       This string is platform dependent. The following works on Windows,
       consult your platform's documentation for more details.*/
    setlocale(LC_ALL, "de-DE");
    std::locale::global(std::locale("de-DE"));

    /* After setting your locale to German, functions like strtod() will fail
       with values like "3.14" because it expects "3,14" instead.
       To fix this, we will tell the parser to use "," as the decimal separator
       and ";" as list argument separator.*/

    const char *expression = "pow(2,2; 2)"; // Instead of "pow(2.2, 2)"
    std::cout << "Evaluating:\n\t" << expression << "\n";

    te_parser tep;
    tep.set_decimal_separator(',');
    tep.set_list_separator(';');

    if (tep.compile(expression))
        {
        const double r = tep.evaluate();
        std::cout << "Result:\n\t" << r << "\n";
        }
    else
        {
        /* Show the user where the error is at. */
        std::cout << "\t " << std::setfill(' ') <<
            std::setw(tep.get_last_error_position()) << '^' <<
            "\tError near here\n";
        }

    return EXIT_SUCCESS;
    }
```

Example 5: Binding to Custom Classes

A class derived from `te_expr` can be bound to custom functions. This enables you to have full access to an object (via these functions) when parsing an expression.

The following demonstrates creating a `te_expr`-derived class which contains an array of values:

```
class te_expr_array : public te_expr
    {
public:
    explicit te_expr_array(const te_variable_flags type) noexcept :
        te_expr(type) {}
    std::array<double, 5> m_data = { 5, 6, 7, 8, 9 };
    };
```

Next, create two functions that can accept this object and perform actions on it. (Note that proper error handling is not shown for brevity.):

```
// Returns the value of a cell from the object's data.
double cell(const te_expr* context, double a)
    {
    auto* c = dynamic_cast<const te_expr_array*>(context);
    return static_cast<double>(c->m_data[static_cast<size_t>(a)]);
    }

// Returns the max value of the object's data.
double cell_max(const te_expr* context)
    {
    auto* c = dynamic_cast<const te_expr_array*>(context);
    return static_cast<double>(
        *std::max_element(c->m_data.cbegin(), c->m_data.cend()));
    }
```

Finally, create an instance of the class and connect the custom functions to it, while also adding them to the parser:

```
te_expr_array teArray{ TE_DEFAULT };

te_parser tep;
tep.set_variables_and_functions(
    {
        {"cell", cell, TE_DEFAULT, &teArray},
        {"cellmax", cell_max, TE_DEFAULT, &teArray}
    });

// Change the object's data and evaluate their summation
// (will be 30).
teArray.m_data = { 6, 7, 8, 5, 4 };
auto result = tep.evaluate("SUM(CELL 0, CELL 1, CELL 2, CELL 3, CELL 4)");

// Call the other function, getting the object's max value
// (will be 8).
res = tep.evaluate("CellMax()");
```

Chapter 10

Custom Extensions

Binding to Custom Functions

TinyExpr++ can also call custom functions. Here is a short example:

```
double my_sum(double a, double b)
    {
    /* Example function that adds two numbers together. */
    return a + b;
    }

te_parser tep;
tep.set_variables_and_functions(
{
    { "mysum", my_sum } // function pointer
});

const double r = tep.evaluate("mysum(5, 6)");
// will be 11
```

Here is an example of using a lambda:

```
te_parser tep;
tep.set_variables_and_functions({
    { "mysum",
        [](double a, double b) noexcept
            { return a + b; } }
    });

const double r = tep.evaluate("mysum(5, 6)");
// will be 11
```

Binding to Custom Classes

A class derived from `te_expr` can be bound to custom functions. This enables you to have full access to an object (via these functions) when parsing an expression.

The following demonstrates creating a `te_expr`-derived class which contains an array of values:

```
class te_expr_array : public te_expr
    {
public:
    explicit te_expr_array(const te_variable_flags type) noexcept :
        te_expr(type) {}
    std::array<double, 5> m_data = { 5, 6, 7, 8, 9 };
    };
```

Next, create two functions that can accept this object and perform actions on it. (Note that proper error handling is not included for brevity.):

```
// Returns the value of a cell from the object's data.
double cell(const te_expr* context, double a)
    {
    auto* c = dynamic_cast<const te_expr_array*>(context);
    return static_cast<double>(c->m_data[static_cast<size_t>(a)]);
    }

// Returns the max value of the object's data.
double cell_max(const te_expr* context)
    {
    auto* c = dynamic_cast<const te_expr_array*>(context);
    return static_cast<double>(
        *std::max_element(c->m_data.cbegin(), c->m_data.cend()));
    }
```

Finally, create an instance of the class and connect the custom functions to it, while also adding them to the parser:

```
te_expr_array teArray{ TE_DEFAULT };

te_parser tep;
tep.set_variables_and_functions(
    {
        {"cell", cell, TE_DEFAULT, &teArray},
        {"cellmax", cell_max, TE_DEFAULT, &teArray}
    });

// change the object's data and evaluate their summation
// (will be 30)
teArray.m_data = { 6, 7, 8, 5, 4 };
auto result = tep.evaluate("SUM(CELL 0, CELL 1, CELL 2, CELL 3, CELL 4)");

// call the other function, getting the object's max value
// (will be 8)
res = tep.evaluate("CellMax()");
```

Chapter 11

Handling Unknown Variables

Although it is possible to add custom variables to the parser, there may be times when you can't anticipate the exact variable names that a user will enter. For example, a user could enter variables representing fiscal years in the format of `FY[year]`. From there, they would like to perform operation such as getting the range between them. In this situation, their expression may be something like this:

```
ABS(FY1999 - FY2009)
```

Here, one could expect the variables `FY1999` and `FY2009` to be treated as `1999` and `2009`, respectively. By subtracting them (and then taking the absolute value of the result), this should yield `10`. The problem is that you would need to set up custom variables prior for `FY1999` and `FY2009`. Even worse, to handle any additional years the user may enter, you would need to create custom variables for every year possible.

Rather than doing that, you can allow the parser to not recognize these variables as usual. At this point, it will fall back to a user-defined function that you provide to resolve it. This is called an "unknown symbol resolver" (USR), and this function will:

- Receive a `std::string_view` of the symbol (i.e., variable name) that the parser failed to recognize
- Determine how to resolve the name
  - If it can resolve it, return a numeric value for the symbol
  - Otherwise, either return `te_parser::te_nan` (i.e., NaN) or throw an exception

When your function resolves a symbol, then its name and numeric value will be added to the parser. Any future evaluations will recognize this name and return the value you previously resolved it to.

> **Tip**
> To change the value for a variable that was resolved previously, use `te_parser::set_constant()`.

This function is set up in the parser by passing it to `te_parser::set_unknown_symbol_resolver()` and can take one of the following signatures:

```
double callback(std::string_view);
double callback(std::string_view, std::string&);
```

The first version will accept the unknown symbol and either return a resolved value or `te_parser::te_nan`. The second version is the same, except that it also accepts a string reference to write a custom message to. (This message can later be retrieved by calling `te_parser::get_last_error_message()`.)

Note
If your USR throws a `std::runtime_exception` with an error message in it, then that message will also be available through `te_parser::get_last_error_message()`.

`te_parser::set_unknown_symbol_resolver()` can accept either a function pointer or a lambda. Here is a simple example using a function:

```
double ResolveResolutionSymbols(std::string_view str)
    {
    // Note that this is case sensitive for brevity.
    return (str == "RES" || str == "RESOLUTION") ?
        96 : te_parser::te_nan;
    }
```

This can be connected as such:

```
te_parser tep;
tep.set_unknown_symbol_resolver(ResolveResolutionSymbols);

// Will resolve to 288, and "RESOLUTION" will be added as a
// variable to the parser with a value of 96.
// Also, beccause TinyExpr++ is case insensitive,
// "resolution" will also be seen as 96 once "RESOLUTION"
// was resolved.
tep.evaluate("RESOLUTION * 3");
```

Warning
Although TinyExpr++ is case insensitive, it is your USR's responsibility to process case-insensitively when resolving names. Once a name has been resolved, then the parser will recognize it case-insensitively in future evaluations.

The following is an example using a lambda and demonstrates the fiscal-year-variables scenario mentioned earlier:

```
te_parser tep;

// Create a handler for undefined tokens that will recognize
// dynamic strings like "FY2004" or "FY1997" and convert them to 2004 and 1997.
tep.set_unknown_symbol_resolver(
    // Handler should except a string (which will be the unrecognized token)
    // and return a double.
    [](std::string_view str) -> double
    {
    const std::regex re{ "FY([0-9]{4})",
        std::regex_constants::icase | std::regex_constants::ECMAScript };
    std::smatch matches;
    std::string var{ str };
    if (std::regex_search(var.cbegin(), var.cend(), matches, re))
        {
        // Unrecognized token is something like "FY1982," so extract "1982"
        // from that and return 1982 as a number. At this point, the variable
        // "FY1982" will added to the parser and set to 1982. All future
        // evaluations will see this as 1982 (unless set_constant() is called
        // to change it).
        if (matches.size() > 1)
            { return std::atol(matches[1].str().c_str()); }
        else
            { return te_parser::te_nan; }
        }
    // Can't resolve what this token is, so return NaN.
    else
        { return te_parser::te_nan; }
    });

// Calculate the range between to fiscal years (will be 10):
tep.evaluate("ABS(FY1999-FY2009)")
```

By default, the parser's USR is a no-op and will not process anything. If you had provided a USR but then need to turn off this feature, then pass a no-op lambda (e.g., `[]{}`) or no-op object (`te_usr_noop{}`) to `set_unknown_symbol_resolver()`.

Chapter 12

Non-US Formatted Formulas

TinyExpr++ supports other locales and non-US formatted formulas. Here is an example:

```
#include "tinyexpr.h"
#include <iostream>
#include <iomanip>
#include <locale>
#include <clocale>

int main(int argc, char *argv[])
    {
    /* Set locale to German.
       This string is platform dependent. The following works on Windows,
       consult your platform's documentation for more details.*/
    setlocale(LC_ALL, "de-DE");
    std::locale::global(std::locale("de-DE"));

    /* After setting your locale to German, functions like strtod() will fail
       with values like "3.14" because it expects "3,14" instead.
       To fix this, we will tell the parser to use "," as the decimal separator
       and ";" as the list argument separator.*/
    const char* expression = "pow(2,2; 2)"; // instead of "pow(2.2, 2)"
    std::cout << "Evaluating:\n\t" << expression << "\n";

    te_parser tep;
    tep.set_decimal_separator(',');
    tep.set_list_separator(';');
    const auto result = tep.evaluate(expression);

    if (tep.success())
        { std::cout << "Result:\n\t" << result << "\n"; }
    else /* Show the user where the error is at. */
        {
        std::cout << "\t " << std::setfill(' ') <<
            std::setw(tep.get_last_error_position()) << "^\tError here\n";
        }

    return EXIT_SUCCESS;
    }
```

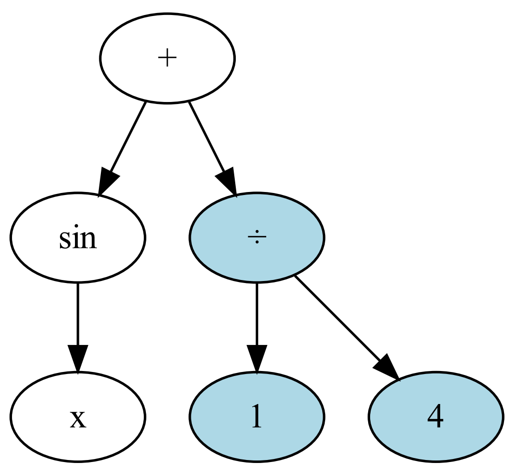
This produces the output:

```
$ Evaluating:
    pow(2,2; 2)
  Result:
    4,840000
```

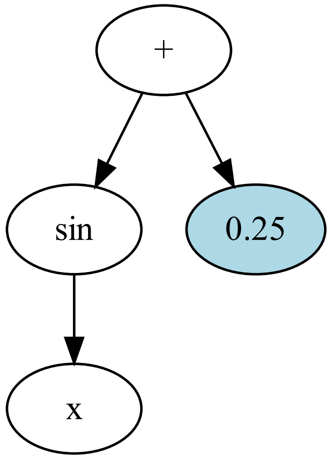
Chapter 13

How it Works

`te_parser::evaluate()` uses a simple recursive descent parser to compile your expression into a syntax tree. For example, the expression `"sin x + 1/4"` parses as:



`te_parser::evaluate()` also automatically prunes constant branches. In this example, the compiled expression returned by `compile()` would become:



Grammar

TinyExpr++ parses the following grammar (from lowest-to-highest operator precedence):

```
<list>      =    <expr> {(",", ";" [dependent on locale]) <expr>}
<expr>      =    <term> {("&" | "|") <term>}
<expr>      =    <term> {("<>" | "!=" | "=" | "<") | "<=") | ">" | ">=") <term>}
<expr>      =    <term> {("<<" | ">>") <term>}
<expr>      =    <term> {("+" | "-") <term>}
<term>      =    <factor> {("*" | "/" | "%") <factor>}
<factor>    =    <power> {("^" | "**") <power>}
<power>     =    {("-" | "+")} <base>
<base>      =    <constant>
               | <variable>
               | <function-0> {"(" ")"}
               | <function-1> <power>
               | <function-X> "(" <expr> {"," <expr>} ")"
               | "(" <list> ")"
```

In addition, whitespace between tokens is ignored.

Valid variable names consist of a letter followed by any combination of: letters `a` through `z` or `A` through `Z`, digits `0` through `9`, periods, and underscores. Constants can be integers, decimal numbers, or in scientific notation (e.g., `1e3` for `1000`). A leading zero is not required (e.g., `.5` for `0.5`).

Chapter 14

Compile-time Options

TE_POW_FROM_RIGHT

By default, *TinyExpr++* does exponentiation from left to right. For example:

`a^b^c == (a^b)^c` and `-a^b == (-a)^b`

This is by design; it's the way that spreadsheets do it (e.g., *LibreOffice Calc*, *Excel*, *Google Sheets*).

If you would rather have exponentiation work from right to left, you need to define `TE_POW_FROM_RIGHT` when compiling. With `TE_POW_FROM_RIGHT` defined, the behavior is:

`a^b^c == a^(b^c)` and `-a^b == -(a^b)`

That will match how many scripting languages do it (e.g., Python, Ruby).

Note that symbols can be defined by passing them to your compiler's command line (or in a Cmake configuration) as such: `-DTE_POW_FROM_RIGHT`

Chapter 15

Embedded Programming

The following section discusses topics related to using *TinyExpr++* in an embedded environment.

Performance

TinyExpr++ is fairly fast compared to compiled C when the expression is short, when the expression does hard calculations (e.g., exponentiation), and when some of the work can be simplified by `evaluate()`. *TinyExpr++* is slower compared to C when the expression is long and involves only basic arithmetic.

Here are some example benchmarks:

Expression	*TinyExpr++*	Native C	Comparison
sqrt(a$^{1.5+a}$2.5)	1,707 ns	58.25 ns	29% slower
a+5	535 ns	0.67 ns	798% slower
a+(5*2)	0.73 ns	969 ns	1,327% slower
(a+5)*2	0.66 ns	980 ns	1,484% slower
(1/(a+1)+2/(a+2)+3/(a+3))	3,388 ns	3.941 ns	859% slower

Note that *TinyExpr++* is slower compared to *TinyExpr* because of additional type safety checks (e.g., the use of `std::variant` instead of unions).

Volatility

If needing to use a `te_parser` object as `volatile` (e.g., accessing it in a system interrupt), then you will need to do the following.

First, declare your `te_parser` as a non-`volatile` object outside of the interrupt function (e.g., globally):

```
te_parser tep;
```

Then, in your interrupt function, create a `volatile` reference to it:

```
void OnInterrupt()
    {
    volatile te_parser& vTep = tep;
    }
```

Functions in `te_parser` which have `volatile` overloads can then be called directly:

```
void OnInterrupt()
    {
    volatile te_parser& vTep = tep;
    vTep.set_list_separator(',');
    vTep.set_decimal_separator('.');
    }
```

The following functions in the `te_parser` class have `volatile` overloads:

- `get_result()`
- `success()`
- `get_last_error_position()`
- `get_decimal_separator()`
- `set_decimal_separator()`
- `get_list_separator()`
- `set_list_separator()`

For any other functions, use `const_cast<>` to remove the parser reference's volatility:

```
void OnInterrupt()
    {
    volatile te_parser& vTep = tep;

    // Use 'const_cast<te_parser&>(vTep)' to access
    // non-volatile functions.
    const_cast<te_parser&>(vTep).set_variables_and_functions(
         { {"STRESS_L", 10.1},
           {"P_LEVEL", .5} });
    const_cast<te_parser&>(vTep).compile(("STRESS_L*P_LEVEL"));

    if (vTep.success())
        {
        auto res = vTep.get_result();
        // Do something else...
        }
    }
```

Note that it is required to make the initial declaration of your `te_parser` non-`volatile`; otherwise, the `const_cast<>` to the `volatile` reference will cause undefined behavior.

Exception Handling

TinyExpr++ requires exception handling, although it does attempt to minimize the use of exceptions (e.g., `noexcept` is used extensively). Syntax errors will be reported without the use of exceptions; issues such as division by zero or arithmetic overflows, however, will internally use exceptions. The parser will trap these exceptions and return NaN (not a number) as the result.

Exceptions can also be thrown when defining custom functions or variables which do not follow the proper naming convention. (Function and variable names must only contain the characters `a-z`, `A-Z`, `0-9`, `.`, and `_`, and must begin with a letter.)

Finally, specifying an illegal character for a list or decimal separator will also throw.

The following functions in `te_parser` can throw and should be wrapped in `try/catch` blocks:

- `compile()`
- `evaluate()`
- `set_variables_and_functions()`
- `add_variable_or_function()`
- `set_decimal_separator()`
- `set_list_separator()`

The caught `std::runtime_error` exception will provide a description of the error in its `what()` method.

Virtual Functions

TinyExpr++ does not use virtual functions or derived classes, unless you create a custom class derived from `te_expr` yourself (refer to Example 5). (`te_expr` defines a virtual destructor that may be implicitly optimized to `final` if no derived classes are defined.)

Part III

Appendix

References

Winkle, Lewis Van. C Math Evaluation Library: TinyExpr. 2016, codeplea.com/tinyexpr.

Index