TinyExpr++

C++ formula parsing and evaluation library

```
f(x) 🖋
te_parser tep;
tep.set_variables_and_functions({
                                   /* Returns the p-level of a study if:
                                       p-level < 5% AND
/* This will compile the expressio
                                       number of observations was at least 30.
if (tep.compile(expression))
                                       Otherwise, NaN is returned. */
   x = 3; y = 4;
   const double r = tep.evaluate(
                                   IF(// Review the results from the analysis
   std::cout << "Result:\n\t" <<
                                       AND(P LEVEL < .05, N OBS >= 30),
                                       // ...and return the p-level if acceptable
else
                                       P LEVEL,
                                       // or NaN if not
    /* Show the user where the err
                                       NAN)
   std::cout << "\t " << std::set
       std::setw(tep.get_last_error_position()) <<
       "\tError near here\n";
   LU TURY LIVING IN WEST CHELSE
```

The Comprehensive
User Reference &
Programming Manual

Blake Madden

$\mathbf{Tiny}\mathbf{Expr} + +$

 $User\ Reference\ \&\ Programming\ Manual$

Blake Madden

$\mathsf{TinyExpr}++$

User Reference & Programming Manual Copyright @ March 24, 2024 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.

Table of contents

O.	verview	1
	Features	1
Ι	User Guide	3
1	Usage	5
2	Operators	7
	Compatibility Note	9
3	Functions	11
	Compatibility Note	16
4	Constants	17
5	Comments	19
II	Developer Guide	21
6	Building	23
	Requirements	23
7	Usage	2 5
	Data Types	25
	Functions	26
	Error Handling	26
	Example	28
8	Custom Extensions	29
	Binding to Custom Variables	29
	Binding to Custom Functions	30
	Binding to Custom Classes	31

Table of contents

9	Handling Unknown Variables	33
10	Non-US Formatted Formulas	37
11	How it Works	39
	Grammar	40
12	Compile-time Options	41
	TE_FLOAT	41
	TE_LONG_DOUBLE	41
	TE_BITWISE_OPERATORS	41
	TE_BRACKETS_AS_PARENS	41
	TE_NO_BOOKKEEPING	42
	TE_POW_FROM_RIGHT	42
	C++20 Features	42
13	Embedded Programming	43
	Performance	43
	Volatility	43
	Floating-point Numbers	44
	Exception Handling	45
	Virtual Functions	45
II	I Appendix	47
Re	eferences	49
In	dex	51

List of Tables

2.1	Operators	8
3.1	Math Functions	11
3.2	Engineering Functions	14
3.3	Statistical Functions	16
3.4	Logic Functions	16
4.1	Math Constants	17
4.2	Logical Constants	17
4.3	Number Formats	17

vi List of Tables

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.
- Can be configured to use double, long double, or float for its calculations.
- 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.

2 Features

Part I User Guide

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)
```

Logical checks can also be nested, creating a "case"-like statement:

- (1) First logical check failed, so now check another scenario and return false if it meets our criteria.
- (2) Neither scenario passed; return NaN because the values are in an unaccounted for scenario.

The same can be accomplished using the IFS() function:

```
IFS(AND(smartMeter1.power > 1900, sensor1.temperature < 52), TRUE,
   AND(smartMeter1.power < 0), FALSE,
   AND(smartMeter1.power < 300, sensor1.temperature > 55), FALSE)

②
```

- (1) First logical check failed, so now check another scenario and return false if it meets our criteria.
- (2) Neither scenario passed; return NaN because the values are in an unaccounted for scenario.

Note

Expressions can optionally begin with an =, the same as spreadsheet programs. For example:

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

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

Operators

The following operators are supported within math expressions:

Table 2.1: Operators

Description
Multiplication.
Division.
Modulus: Divides two values and returns the
remainder.
Addition.
Subtraction.
Either exponentiation (the default) or bitwise XOR.
For 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 ^)
Less than.
Greater than.
Greater than or equal to.
Less than or equal to.
Equals.
Equals. (This is an alias for $=$)
Not equal to.
Not equal to. (This is an alias for $\langle \rangle$)
Either logical (the default) or bitwise conjunction
(AND).
Logical conjunction (AND).
Either Logical (the default) or bitwise alternative
(OR).
Logical alternative (OR).
Groups sub-expressions, overriding the order of
operations.
Bitwise NOT.
Bitwise left shift.
Bitwise right shift.
Bitwise left rotate. (Only available if compiled as
C++20.)
Bitwise right rotate. (Only available if compiled as
C++20.)
Bitwise right shift. Bitwise left rotate. (Only available if compiled $C++20$.) Bitwise right rotate. (Only available if compile

? Tip

Define TE_BITWISE_OPERATORS to enable bitwise behavior for &, |, and ^.

Chapter 2. Operators

For operators, the order of precedence is:

Operator	Description
()	Instructions in parentheses are executed first. (If
	TE_BRACKETS_AS_PARENS is defined, then [] are
	treated the same way.)
+, -, ~	Positive or negative sign for a value, and bitwise
	NOT.
^	Exponentiation.
*, /, and %	Multiplication, division, and modulus.
+ and -	Addition and subtraction.
<<, >>, <<<, >>>	Bitwise left and right shift and rotation.
<, >, >=, <=	Relational comparisons.
= and $!=$	Equality comparisons.
&&	Logical conjunction (AND).
	Logical alternative (OR).
&	Bitwise AND (if TE_BITWISE_OPERATORS is
	defined).
^	Bitwise XOR (if TE_BITWISE_OPERATORS is defined).
	Bitwise OR (if TE_BITWISE_OPERATORS is defined).

A Warning

Defining TE_FLOAT will disable all bitwise functions and operators.

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).

Compatibility 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.

10 Compatibility Note

Functions

The following built-in functions are available:

Table 3.1: Math Functions

Function	Description
ABS(Number)	Absolute value of <i>Number</i> .
ACOS(Number)	Returns the arccosine, or inverse cosine, of <i>Number</i> .
	The arccosine is the angle whose cosine is number.
	The returned angle is given in radians in the range
	O (zero) to PI.
ASIN(Number)	Returns the arcsine, or inverse sine function, of
	Number, where $-1 \le Number \le 1$. The arcsine is
	the angle whose sine is <i>Number</i> . The returned angle
ATDANI ()	is given in radians where $-pi/2 \ll -pi/2$.
ATAN(x)	Returns the principal value of the arc tangent of x ,
ATAN2(y, x)	expressed in radians Returns the principal value of the arc tangent of
A1A1V2(y, x)	y,x, expressed in radians.
CEIL(Number)	Smallest integer not less than Number.
CDID(IVIIIDEI)	CEIL $(-3.2) = -3$
	CEIL(3.2) = 4
CLAMP(Number, Start, End)	Constrains <i>Number</i> within the range of <i>Start</i> and
ozimi (ramoor, saara, ziaa)	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 <i>Number</i> .
COT(Number)	Cotangent of Number.
EXP(Number)	Euler to the power of <i>Number</i> .
EVEN(Number)	Returns <i>Number</i> rounded up to the nearest even integer.
	Values are always rounded away from zero (e.g.,
	EVEN(-3) = -4).
FAC(Number)	Returns the factorial of <i>Number</i> . 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
ISEVEN(Number)	Returns true if <i>Number</i> is even, false if odd.
ISODD(Number)	Returns true if <i>Number</i> is odd, false if even.
LN(Number)	Natural logarithm of <i>Number</i> (base Euler).
LOG10(Number)	Common logarithm of Number (base 10).
MIN(Number1, Number2,)	Returns the smallest value from a specified range of values.
MAX(Number1, Number2,)	Returns the largest value from a specified range of values.
MAXINT()	Returns the largest integer that the parser can
V	process.
MOD(Number, Divisor)	Returns the remainder after <i>Number</i> is divided by
, , , , , ,	Divisor. The result has the same sign as divisor.
NAN	Returns an invalid value (i.e., Not-a-number).
NCR(Number, NumberChosen)	Alias for COMBIN().
NPR(Number, NumberChosen)	Alias for PERMUT().
ODD(Number)	Returns Number rounded up to the nearest odd
	integer.
	Values are always rounded away from zero (e.g.,
	$\mathtt{ODD}(-4) = -5).$
PERMUT(Number, NumberChosen)	Returns the number of permutations for a given
	number $(Number Chosen)$ of items that can be
	selected <i>Number</i> 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 <i>Base</i> to any power. For fractional exponents,
DOWED/D E ()	Base must be greater than 0.
POWER(Base, Exponent)	Alias for POW().
RAND()	Generates a random floating point number within
DOLIND(N. 1. N. D)	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(-11.0) = 12 ROUND(1.5, 0) = 2
	ROUND(1.5, 0) = 2 ROUND(1.55, 1) = 1.6
	ROUND(3.1415, 3) = 3.142
	ROUND(3.1415, 3) = 3.142 ROUND(-50.55, -2) = -100
SIGN(Number)	Returns the sign of $Number$. Returns 1 if $Number$ is
STOTY(Trumber)	positive, zero (0) if <i>Number</i> is 0, and -1 if <i>Number</i>
	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 <i>Number</i> .
(- :	

Function	Description
TRUNC(Number)	Discards the fractional part of <i>Number</i> . TRUNC(-3.2) = -3 TRUNC(3.2) = 3

Table 3.2: Engineering Functions

Function	Description
BITAND(Number1, Number2)	Returns a bitwise 'AND' of two (integral) numbers. (Both numbers must be positive and cannot exceed (2^48)-1.)
${\bf BITLROTATE8} ({\bf Number}, {\bf RotateAmount})$	Returns Number left rotated left to the most significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 8-bit integer.
${\bf BITLROTATE16} ({\bf Number,\ RotateAmount})$	(Only available if compiled as C++20.) Returns Number left rotated left to the most significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 16-bit integer.
${\bf BITLROTATE 32 (Number,Rotate Amount)}$	(Only available if compiled as C++20.) Returns Number left rotated left to the most significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 32-bit integer.
${\bf BITLROTATE 64 (Number,Rotate Amount)}$	(Only available if compiled as C++20.) Returns Number left rotated left to the most significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 64-bit integer.
${\bf BITLROTATE (Number,\ Rotate Amount)}$	(Only available if compiled as C++20.) Returns <i>Number</i> left rotated left to the most significant bit by the specified number (<i>RotateAmount</i>) of bits.
${\bf BITLSHIFT (Number,ShiftAmount)}$	Numbers are rotated as either a 32- or 64-bit integer (depending on what is supported by the compiler). (Only available if compiled as C++20.) Returns Number left shifted by the specified number (ShiftAmount) of bits. Numbers cannot exceed (2^48)-1 and ShiftAmount
BITNOT8(Number)	cannot exceed 53 (or 63, if 64-bit is supported). Returns a bitwise 'NOT' of a 8-bit integer.
BITNOT16(Number)	Returns a bitwise 'NOT' of a 16-bit integer.
BITNOT32(Number)	Returns a bitwise 'NOT' of a 32-bit integer.
BITNOT64(Number)	Returns a bitwise 'NOT' of a 64-bit integer (if 64-bit integers are supported).
BITNOT(Number)	Returns a bitwise 'NOT' of a 32- or 64-bit integer (depending on whether 64-bit is supported).
BITOR(Number1, Number2)	Returns a bitwise 'OR' of two (integral) numbers. (Both numbers must be positive and cannot exceed (2^48)-1.)
${\bf BITRROTATE8 (Number,\ RotateAmount)}$	Returns Number right rotated right to the least significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 8-bit integer. (Only available if compiled as C++20.)

Chapter 3. Functions 15

Function	Description
BITRROTATE16(Number, RotateAmount)	Returns Number right rotated right to the least significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 16-bit integer. (Only available if compiled as C++20.)
${\bf BITRROTATE 32 (Number,\ Rotate Amount)}$	Returns Number right rotated right to the least significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 32-bit integer. (Only available if compiled as C++20.)
${\bf BITRROTATE 64 (Number,\ Rotate Amount)}$	Returns Number right rotated right to the least significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as an unsigned 64-bit integer. (Only available if compiled as C++20.)
${\bf BITRROTATE(Number,RotateAmount)}$	Returns Number right rotated right to the least significant bit by the specified number (RotateAmount) of bits. Numbers are rotated as either a 32- or 64-bit integer (depending on what is supported by the compiler).
${\bf BITRSHIFT(Number,ShiftAmount)}$	(Only available if compiled as C++20.) Returns <i>Number</i> right shifted by the specified number (<i>ShiftAmount</i>) of bits. Numbers cannot exceed (2^48)-1 and <i>ShiftAmount</i> cannot exceed 53 (or 63, if 64-bit is supported).
BITXOR(Number1, Number2)	Returns a bitwise 'XOR' of two (integral) numbers. (Both numbers must be positive and cannot exceed (2^48)-1.)
${\bf SUPPORTS32BIT()}$	Returns true if 32-bit integers are supported. This will affect the supported range of values for bitwise operations.
SUPPORTS64BIT()	Returns true if 64-bit integers are supported. This will affect the supported range of values for bitwise operations.

▲ Warning Defining TE_FLOAT will disable all bitwise functions and operators.

16 Compatibility Note

Table 3.3: 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 3.4: Logic Functions

Function	Description
AND(Value1, Value2,)	Returns true if all conditions are true.
IF(Condition, ValueIfTrue, ValueIfFalse)	If <i>Condition</i> is true (non-zero), then <i>ValueIfTrue</i> is returned; otherwise, <i>ValueIfFalse</i> is returned. Note that multiple IF commands can be nested to create a "case" statement.
IFS(Condition1, Value1, Condition2, Value2,)	Checks up to three conditions, returning the value corresponding to the first met condition. This is shorthand for multiple nested IF commands, providing better readability. Will accept 1–3 condition/value pairs. NaN will be returned if all conditions are false.
NOT(Value) OR(Value1, Value2,)	Returns the logical negation of <i>Value</i> . Returns true if any condition is true.



The first argument to any logic function must be valid (i.e., not NaN). If the first argument evaluates to NaN, then NaN will be returned. Any subsequent arguments that evaluate to NaN will be ignored.

Compatibility Note

BITNOT will call either BITNOT32 or BITNOT64, depending on whether 64-bit integers are supported. This differs from Excel, which only works with 16-bit integers. To match the behavior of Excel, explicitly call BITNOT16.

Constants

The following mathematical and logical constants are available:

Table 4.1: Math Constants

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

A Warning

If TE_FLOAT is defined, then some precision for pi and Euler's number will be lost.

Table 4.2: Logical Constants

Constant	Value
TRUE	1
FALSE	0

The following number formats are supported:

Table 4.3: Number Formats

Format	Example
Decimal	5754615 or 3.14
Hexadecimal	0x57CEF7, which yields 5754615
Scientific notation	1e3, which yields 1000

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)
```

- ① Comment that can span multiple lines.
- (2) Single line comment.

Part II Developer Guide

Building

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

The API documentation can be built using the following:

doxygen docs/Doxyfile

Requirements

TinyExpr++ must be compiled as C++17. (Some additional features require C++20.) MSVC, GCC, and Clang compilers are supported.

24 Requirements

Usage

Data Types

The following data types and constants are used throughout TinyExpr++:

te_parser: The main class for defining an evaluation engine and solving expressions.

te_type: The data type for variables, function parameters, and results. By default this is double, but can be float (when TE_FLOAT is defined) or long double (when TE_LONG_DOUBLE is defined).

If long double is in use, then various bitwise functions (BITNOT(), BITRROTATE()) may be able to support 64-bit integers. This will depend on the compiler used to build the library and can be verified by calling te_parser::supports_64bit() in your code or SUPPORTS64BIT() in an expression at runtime.

te_variable: A user-defined variable or function that can be embedded into a te_parser. (Refer to custom variables.)

te_expr: The base class for a compiled expression. Classes that derive from this can be bound to custom functions. In turn, this allows for connecting more complex, class-based objects to the parser. (Refer to custom classes.)

te_parser::te_nan: An invalid numeric value. This should be returned from user-defined functions to signal a failed calculation.

te_parser::npos: An invalid position. This is returned from te_parser::get_last_error_position() when no error occurred.

te_usr_noop: A no-op function. When passed to te_parser::set_unknown_symbol_resolver(), will disable unknown symbol resolution. (Refer to ch. 9.)

26 Functions

Functions

The primary interface to TinyExpr++ is the class te_parser. A te_parser is a self-contained parser, which stores its own user-defined variables & functions, separators, and state information.

te_parser provides these functions:

```
te_type evaluate(const std::string_view expression);
                                                                                          (1)
te_type get_result();
                                                                                          (2)
bool success();
int64_t get_last_error_position();
std::string get_last_error_message();
set_variables_and_functions(const std::set<te_variable>& vars);
                                                                                          (3)
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);
                                                                                          4
get_decimal_separator();
set_decimal_separator();
get_list_separator();
set_list_separator();
```

- (1) 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.)
- (2) get_result() can be called anytime afterwards to retrieve the result from evaluate().
- (3) set_variables_and_functions(), get_variables_and_functions(), and add_variable_or_function() are used to add custom variables and functions to the parser.
- (4) set_unknown_symbol_resolver() is used to provide a custom function to resolve unknown symbols in an expression. (Refer to ch. 9 for further details.)
- (5) 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).
auto result = tep.evaluate("(5+5)");

// Returns NaN, error position is set to 3.
auto 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.

Chapter 7. Usage 27

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:

```
#include "tinyexpr.h"
#include <iostream>
te_type 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;
    //\ \mbox{Will} use the previously used expression, returns 5.
    const auto h1 = tep.evaluate();
    x = 5; y = 12;
    // Returns 13.
    const auto h2 = tep.evaluate();
else
    {
    std::cout << "Parse error at " <<
        std::to_string(tep.get_last_error_position()) << "\n";</pre>
```

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<te_type>::quiet_NaN()) as the result. NaN values can be verified using the standard function std::isnan().

28 Example

Example

The following is a short example demonstrating how to use $\mathit{TinyExpr}++.$

Custom Extensions

Binding to Custom Variables

Along with the built-in functions and constants, you can create custom variables for use in expressions:

```
#include "tinyexpr.h"
#include <iostream>
#include <iomanip>
int main(int argc, char* argv[])
   if (argc < 2)
       std::cout << "Usage: example \"expression\"\n";</pre>
       return EXIT_SUCCESS;
   const char* expression = argv[1];
   te_type x{ 0 }, y{ 0 }; // x and y are bound at eval-time.
   // Store variable names and pointers.
   te_parser tep;
   tep.set_variables_and_functions({ {"x", &x}, {"y", &y} });
   if (tep.compile(expression)) // Compile the expression and check for errors.
       /* The variables can be changed here, and evaluate can be called multiple
          times. This is efficient because the parsing has already been done.*/
       x = 3; y = 4;
        const auto 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 here\n";</pre>
   return EXIT_SUCCESS;
```

Binding to Custom Functions

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

Here is an example of using a lambda:

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<te_type, 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.):

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

Note

Valid variable and function names consist of a letter or underscore followed by any combination of: letters a-z or A-Z, digits 0-9, periods, and underscores.

Handling Unknown Variables

Although it is possible to add 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:

```
te_type callback(std::string_view);
te_type 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:

```
te_type 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");
```

A 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 te_type.
    [](std::string_view str) -> te_type
    {
    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 be 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().

Also, once a variable has been resolved by your USR, it will be added as a custom variable with the resolved value assigned to it. With subsequent evaluations, the previously unrecognized symbols you resolved will be remembered. This means that it will not to be resolved again and will result in the value that your USR returned from before.

If you prefer to not have resolved symbols added to the parser, then pass false to the second parameter to set_unknown_symbol_resolver(). This will force the same unknown symbols to be resolved again with every call to compile(expression) or evaluate(expression). (The version of evaluate() which does not take any parameters will not force calling the USR again, see below.) This can be useful for when your USR's symbol resolutions are dynamic and may change with each call.

For example, say that an end user will enter variables that start with "STRESS," but you are uncertain what the full name will be. Additionally, you want to increase the values of these variables with every evaluation. The following demonstrates this:

```
te_parser tep;
tep.set_unknown_symbol_resolver(
    [](std::string_view str) -> te_type
   static te_type stressLevel{ 3 };
   if (std::strncmp(str.data(), "STRESS", 6) == 0)
        { return stressLevel++; }
   else
        { return te_parser::te_nan; }
   },
    // purge resolved variables after each evaluation
   false);
// Initial resolution of STRESS_LEVEL will be 3.
tep.evaluate("STRESS_LEVEL");
// Because STRESS_LEVEL wasn't kept as a variable (with a value of 3)
// in the parser, then subsequent evaluations will require
// resolving it again:
tep.evaluate("STRESS_LEVEL"); // Will be 4.
tep.evaluate("STRESS LEVEL"); // 5
tep.evaluate("STRESS_LEVEL"); // 6
```

As a final note, if you are not keeping resolved variables, your USR will only be called again if you call evaluate or compile with an expression. Because the original expression gets optimized, evaluate() it will not re-evaluate any variables. Calling evaluate with the original expression will force a re-compilation and in turn call the USR again.

Non-US Formatted Formulas

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

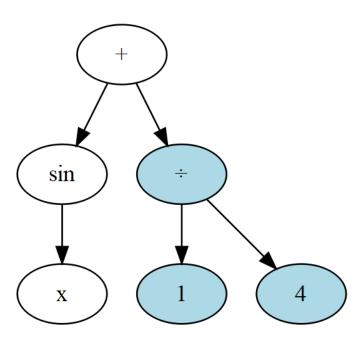
```
#include "tinyexpr.h"
#include <iostream>
#include <iomanip>
#include <locale>
#include <clocale>
int main(int argc, char *argv[])
   /* Set locale to German.
       This string is platform dependent. The following works on Windows,
       consult your platform's documentation for more details.*/
   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";</pre>
   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";</pre>
   return EXIT_SUCCESS;
```

This produces the output:

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

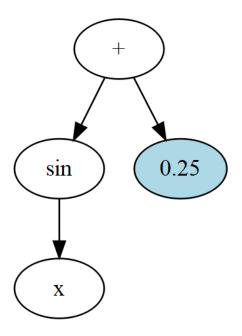
How it Works

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:

40 Grammar



Grammar

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

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

Compile-time Options

TE_FLOAT

double is the default data type used for the parser's variable types, parameters, and return types. Compile with TE_FLOAT defined to use float instead.

Refer to floating-point numbers for more information.

A Warning

This flag will also disable all bitwise functions and operators.

TE_LONG_DOUBLE

Compile with TE_LONG_DOUBLE defined to use long double as the default data type.

Depending on the compiler, this may provide support for handling uint64_t values. Call te_parser::supports_64bit() (or SUPPORTS64BIT() in an expression at runtime) to confirm this.

TE_BITWISE_OPERATORS

By default, the operators &, |, and ^ represent logical AND, logical OR, and exponentiation. If TE_BITWISE_OPERATORS is defined, then they will represent bitwise AND, OR, and XOR.

TE_BRACKETS_AS_PARENS

Define this flag to treat [] pairs the same as (). This can be useful if using the parser for a debugger's watch window. For example, an expression such as [rsp+1000] would be evaluated as 1000 being added with the variable rsp. In this context, rsp could represent a memory address in the debugger.

42 TE NO BOOKKEEPING

TE_NO_BOOKKEEPING

By default, the parser will keep track of all functions and variables used in the last expression it evaluated. From this, the presence of a function or variable in the expression can be verified via is_function_used() and is_variable_used().

Turning this option off can provide a small optimization, as it will result in less heap allocations and search operations. Defining TE_NO_BOOKKEEPING will disable this feature.

TE POW FROM RIGHT

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

$$a^b^c == (a^b)^c \text{ and } -a^b == (-a)^b$$

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

C++20 Features

If compiling as C++20 (and TE_FLOAT is not defined), then the following functions and operators will be available:

- BITLROTATE8
- BITLROTATE16
- BITLROTATE32
- BITLROTATE64
- BITLROTATE
- BITRROTATE8
- BITRROTATE16
- BITRROTATE32
- BITRROTATE64
- BITRROTATE
- <<< (unsigned 64-bit left rotation)
- >>> (unsigned 64-bit right rotation)

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 or does hard calculations (e.g., exponentiation). 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
$\overline{\operatorname{sqrt}(a^{1.5+a}2.5)}$	1,707 ns	58.25 ns	29% slower
a+5	535 ns	$0.67 \mathrm{\ ns}$	798% slower
(1/(a+1)+2/(a+2)+3/(a+3))	3,388 ns	$3.941~\mathrm{ns}$	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), case insensitivity, and bookkeeping operations.

Refer to compile-time options for flags that can provide optimization.

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:

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.

Floating-point Numbers

double is the default data type used for the parser's variable types, parameters, and return types. For embedded environments that require float, compile with TE_FLOAT defined to use float instead.

When using this option, it is recommended to use the helper typedef te_type. This will map to either float or double (depending on whether TE_FLOAT is defined). By defining your functions and variables with te_type, you won't need to replace double and float if needing to change this compiler flag.

For example, a custom function would be written as such:

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 may 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 "Binding to Custom Classes" for an example). (te_expr defines a virtual destructor that may be implicitly optimized to final if no derived classes are defined.)

Virtual Functions

Part III

Appendix

References

Alexander, Michael, and Dick Kusleika. $Microsoft\ Excel\ 365\ Bible.$ Wiley, 2022. Winkle, Lewis Van. $C\ Math\ Evaluation\ Library:\ TinyExpr.\ 2016,\ https://codeplea.com/tinyexpr.$

50 References

Index

C	Н
case statements, 5, 16	hexadecimal, 17
classes binding to custom, 31	L
derived, 45 comments, 19	localization separators, 37
options, 42	O
requirements, 23 constants, 17	operators, 7
D	R
data types 64-bit integer, 16, 25	results
float vs. double, 44 long double, 25	evaluating failed, 27
E	S
embedded systems, 43	scientific notation, 17
exceptions, 26, 45	U
F	unknown symbols resolving, 33
functions binding to custom, 30	V
engineering, 14 logical, 16	variables
math, 11 statistical, 16	binding to custom, 29 unknown, see unknown symbols
virtual, 45	volatile, 44