Expression

In recitations, we showed an example of designing a class hierarchy for different kinds of nodes in an **abstract syntax tree** (AST). In this problem, we will support more functionalities:

- A new kind of node called `Variable`, which represents the varaible `x`. With this supported, our AST can be used to represent unary functions, instead of only arithmetic expressions with constants.
- Evaluation of the function at a certain point $x = x_0$.
- Evaluation of the derivative of the function at a certain point $x = x_0$.

The operations that need to be supported are declared in this base class:

```
class NodeBase {
 public:
  // Make any of these functions virtual, or pure virtual, if necessary.
  NodeBase() = default;
  double eval(double x) const; // Evaluate f(x)
  double derivative(double x) const; // Evaluate df(x)/dx
  std::string rep() const; // Returns the parenthesized representation of the function.
  ~NodeBase() = default;
};
```

In recitations, we used only one class `BinaryOperator` to represent four different kinds of binary operators. Such design may not be convenient for this problem, because the ways of evaluating the function and its derivative differ to a greater extent between different operators. Therefore, we separate them into four classes:

```
class BinaryOperator : public NodeBase {
 protected:
  Expr m_left;
  Expr m_right;
 public:
  BinaryOperator(const Expr &left, const Expr &right)
      : m_left{left}, m_right{right} {}
};

class PlusOp : public BinaryOperator {
  using BinaryOperator::BinaryOperator;
  // Add functions if necessary.
};

class MinusOp : public BinaryOperator {
  using BinaryOperator::BinaryOperator;
  // Add functions if necessary.
};

class MultiplyOp : public BinaryOperator {
  using BinaryOperator::BinaryOperator;
  // Add functions if necessary.
};
```

```
class DivideOp : public BinaryOperator {
  using BinaryOperator::BinaryOperator;
  // Add functions if necessary.
};
```

By declaring `using BinaryOperator::BinaryOperator;`, we are **inheriting** the constructor from `BinaryOperator`. For example, the class `PlusOp` now has a constructor like this:

```
class PlusOp : public BinaryOperator {
 public:
  PlusOp(const Expr &left, const Expr &right) : BinaryOperator(left, right) {}
};
```

and the same for `MinusOp`, `MultiplyOp` and `DivideOp`.

To support the variable `x`, we define a new class `Variable`:

```
class Variable : public NodeBase {
  // Add functions if necessary.
};
```

Then we define a `static` member of `Expr` as follows.

```
class Expr {
  std::shared_ptr<NodeBase> m_ptr;
  Expr(std::shared_ptr<NodeBase> ptr) : m_ptr{ptr} {}

 public:
  Expr(double value);

  static const Expr x;
  // Other members ...
};

// After the definition of `Variable`:
const Expr Expr::x{std::make_shared<Variable>()};
```

Note that `Expr` also has a non-`explicit` constructor that receives a `double` and creates a `Constant` node:

```
Expr::Expr(double value) : m_ptr{std::make_shared<Constant>(value)} {}
```

With all of these, and the overloaded arithmetic operators defined, we can create functions in a very convenient manner:

```
auto &x = Expr::x;
auto f = x * x + 2 * x + 1; // x^2 + 2x + 1
std::cout << f.rep() << std::endl; // (((x) * (x)) + ((2.000000) * (x))) + (1.000000)
std::cout << f.eval(3) << std::endl; // 16
std::cout << f.derivative(3) << std::endl; // 8
auto g = f / (-x * x + x - 1); // (x^2 + 2x + 1)/(-x^2 + x - 1)
std::cout << g.eval(3) << std::endl; // -2.28571
std::cout << g.derivative(3) << std::endl; // 0.489796
```

Requirements for `rep()`

Note: This is a very simple and naive way to convert an expression to a string, because we don't want to set barriers on this part. You can search for some algorithms to make the expression as simplified as possible, but it may not pass the OJ tests.

Any operand of the unary operators (`+`, `-`) or the binary operators (`+`, `-`, `*`, `/`) should be surrounded by a pair of parentheses. Correct examples:

- $2 + 3$: `(2.000000) + (3.000000)`
- $2x + 3$: `((2.000000) * (x)) + (3.000000)`
- $2 \cdot (-x) + 3$: `((2.000000) * (-(x))) + (3.000000)`

There should be a space before and after each binary operator. For example, `(expr1) + (expr2)` instead of `(expr1)+(expr2)`.

To convert a floating-point number to `std::string`, just use `std::to_string` and you will be free of troubles caused by precisions and floating-point errors.

Note: If the floating-point value of a `Constant` node is negative, it is **not** treated as a unary minus sign applied to its absolute value. For example,

```
Expr e(-2);
std::cout << e.rep() << std::endl;
```

The output should be `-2.000000` instead of `-(2.000000)`.

`example.cpp` contains these simple tests.

Requirements for `Expr`

From the user's perspective, only `Expr` and its arithmetic operators are **interfaces**. Anything else in your code is **implementation details**, which user code will not rely on. In other words, you are free to implement these things in any way, as long as the interfaces of `Expr` meet our requirements.

- `Expr` is copy-constructible, copy-assignable, move-constructible, move-assignable and destructible. These operations should just perform the corresponding operations on the member `m_ptr`, and let the corresponding function of `std::shared_ptr` handle everything. The move operations should be `noexcept`.
- `Expr` is constructible from a `double` value, and this constructor is non-`explicit`.

- Let `e` be `const Expr` and `x0` be `double`, then the subtree rooted at `e` represents a function. `e.eval(x0)` returns the value of this function at $x = x_0$. `e.derivative(x0)` returns the value of the derivative of this function at $x = x_0$. `e.rep()` returns a `std::string` representing this function.
- Let `e1` and `e2` be two objects of type `const Expr`. The following expressions return an object of type `Expr`, which creates a new node corresponding to the operators.
  - `-e1`
  - `+e1`
  - `e1 + e2`
  - `e1 - e2`
  - `e1 * e2`
  - `e1 / e2`
- `Expr::x` is an object of type `const Expr`, which represents the variable `x`.

Use `compile_test.cpp` to check whether your code compiles.

Submission

Submit `expr.hpp` or its contents to OJ.

Thinking

Why do we use `std::shared_ptr` instead of `std::unique_ptr`?

Can you add support for more functionalities, e.g. the elementary functions $\sin(e)$, $\cos(e)$, exponential expressions $e_1^{e_2}$, ...? More variables? Print expressions to $\LaTeX$?