User-defined numeric types

Sometimes we need to represent arbitrarily large integers. A `BigInt` class, along with associated arithmetic operators, can represent such indefinite-magnitude integers:

```
namespace bigint
{
class BigInt
{
    // ...
};

BigInt operator+(const BigInt&);
BigInt operator-(const BigInt&);
BigInt operator+(const BigInt&, const BigInt&);
BigInt operator-(const BigInt&, const BigInt&);
BigInt operator*(const BigInt&, const BigInt&);
BigInt operator/(const BigInt&, const BigInt&);
BigInt abs(const BigInt&);
// ...
```

A `BigInt` literal must be able to represent a value larger than would fit in the largest built-in integral type, so we define the suffix using a `raw` UDL operator:

```
namespace literals
{
BigInt operator""_bigint(const char* digits)  // raw literal
{
    BigInt value;
    // ...        (Compute BigInt from digits.)
    return value;
}
}  // close namespace literals

using namespace literals;
}  // close namespace bigint

using namespace bigint::literals;  // Make _bigint literal available.
bigint::BigInt bnval  = 587135094024263344739630005208021231626182814_bigint;
bigint::BigInt bigone = 1_bigint;  // small value, but still has type BigInt
```

The `BigInt` class is appropriate for large integers, but numbers having fractional parts have a different problem: The IEEE standard **`double`** floating-point type cannot represent certain values exactly, e.g., `24692134.03`. When **`double`**s are used to represent values, long summations may eventually produce an error in the hundredths place; i.e., the result will be off by `.01` or more. In financial calculations, where values represent money, errors of just one or two pennies might be unacceptable. For this problem, we turn to decimal fixed-point (rather than binary floating-point) arithmetic.