Polynomial

In this task, you will design and implement a class `Polynomial` representing a real polynomial of the following form:

$$P(x)=\sum_{i=0}^{n}a_ix^i,\quad a_i,x\in\mathbb{R}$$

where $a_0,\cdots,a_n$ are the coefficients and $a_n\neq 0$. The **degree** of a polynomial is the highest degree of the terms with non-zero coefficients, denoted by

$$\deg(P)=n.$$

In particular, the following corner cases are defined.

- $a_0x^0=a_0$ even if $x=0$.
- **A polynomial has at least one coefficient.** When $n=0$, the polynomial becomes a constant function $P(x)=a_0$ whose degree is 0, even if $a_0=0$.

There are a number of operations that can be performed on polynomials, including addition, subtraction, multiplication, etc. To ensure that the coefficient of the highest-order term is always non-zero, we may need a function `adjust` to remove the unnecessary trailing zeros.

```cpp
class Polynomial {
 private:
  // `m_coeffs` stores the coefficients.
  // Note: This is not the unique correct implementation.
  // For example, you may separate the constant term from others,
  // and store the constant term using another variable.
  std::vector<double> m_coeffs;

  static auto isZero(double x) {
    static constexpr auto eps = 1e-8;
    return x < eps && -x < eps;
  }

  // Remove trailing zeros, to ensure that the coefficient of the term with
  // the highest order is non-zero.
  // Note that a polynomial should have at least one term, which is the
  // constant. It should not be removed even if it is zero.
  // If m_coeffs is empty, adjust() should also insert a zero into m_coeffs.
  void adjust() {
    // YOUR CODE HERE
  }

  // Other members ...
};
```

Considering potential floating-point errors, it is suggested to use the function `isZero(x)` to determine whether `x` is zero, which is effectively $|x|<\epsilon$ for some very small $\epsilon>0$.

The coefficients, of course, should be stored in a sequential container, and `std::vector` is a perfect choice.

Constructors and copy control

The class `Polynomial` should meet the following requirements.

- Default-constructible. A `Polynomial` can be default-initialized to the constant function $P(x) = 0$.
- Copyable, movable and destructible. A `Polynomial` should have a copy constructor, a move constructor, a copy assignment operator, a move assignment operator and a destructor. These functions should perform the corresponding operations directly on the member `m_coeffs`. Before starting to define them, think about the following questions:
  - Will the compiler generate these functions for you if you do not define them?
  - What are the behaviors of the compiler-generated functions? Do they meet the requirements?

  Note: Direct moving of the member `m_coeffs` will possibly make the moved-from polynomial have no coefficients (its `m_coeffs` will possibly be empty), which is not in a valid state if we require all `Polynomial`s to have at least one coefficient. But considering that this is not the concentration of this problem, we only require that the moved-from object can be safely assigned to or destroyed, which means that the compiler-generated move operations suffice. The move operations of `Polynomial` are not required to be `noexcept` here. You are free to choose any reasonable design.
- Constructible from a pair of *InputIterators*. This function has been implemented for you, as long as your `adjust()` is correct.

  ```
  template <typename InputIterator>
  Polynomial(InputIterator begin, InputIterator end)
      : m_coeffs(begin, end) { adjust(); }
  ```

  This allows the following ways of constructing a `Polynomial`:

  ```
  double a[] = {1, 2, -1, 3.5};
  std::vector b{2.5, 3.3, 0.0};
  std::list c{2.7, 1.828, 3.2}; // not a vector, but have iterators
  std::deque<double> d; // empty
  Polynomial p1(a, a + 4); // 1 + 2x - x^2 + 3.5*x^3
  Polynomial p2(b.begin(), b.end()); // 2.5 + 3.3x, the trailing zero removed
  Polynomial p3(c.begin(), c.end()); // 2.7 + 1.828x + 3.2*x^2
  Polynomial p4(d.begin(), d.end()); // 0
  ```

  If the iterator range `[begin, end)` is empty, i.e. `begin == end`, this constructor initializes the `Polynomial` to be `P(x)=0`.
- Constructible from a `const std::vector<double>`, which contains the coefficients.

  ```
  std::vector a{2.5, 3.3, 4.2, 0.0, 0.0};
  std::vector<double> b; // empty
  Polynomial p1(a); // 2.5 + 3.3x + 4.2*x^2.
  Polynomial p2(b); // 0
  Polynomial p3(std::move(a)); // Move it, instead of copying it.
  ```

Note that if the argument passed in is a non-`const` rvalue, it should be moved instead of being copied. Moreover, implicit conversion from a `std::vector<double>` to `Polynomial` through this constructor should be disabled.

Basic operations

Let `p` be of type `Polynomial` and `cp` be of type `const Polynomial`. The following operations should be supported.

- Both `p.deg()` and `cp.deg()` return the degree of the polynomial. Any reasonable integral return type is allowed, but it should not be too small.
- For an integer `i`, both `p[i]` and `cp[i]` return the coefficient of the term x^i as a read-only number. Any reasonable return type is allowed. The behavior is undefined if `i` is not in the valid range `[0, deg()]`.
- For an integer `i` and a real number `c`, `p.setCoeff(i, c)` sets the coefficient of the term x^i to `c`. The behavior is undefined if `i < 0`. If `i > deg()`, it has the same effect as adding a monomial cx^i to it, which means that after this operation,
  - `p.deg()` becomes `i`.
  - `p[j]` is zero for every `j` in `(d, i]`, where `d` is the degree of `p` before this operation.

  Note that allowing the user to modify the coefficients may result in trailing zeros, which your code must handle correctly. This is why we don't allow `p[i]` to return the non-`const` reference to the `i`-th coefficient.

  `std::vector<T>::resize` may help you.
- `operator==` and `operator!=` should be defined. Two polynomials `a` and `b` are equal if and only if `a - b` is the zero constant function $P(x) = 0$. Equivalently, `a == b` if and only if
  - `a.deg() == b.deg()`, and
  - `isZero(a[i] - b[i])` holds for every integer `i` in `[0, a.deg()]`.
- For a number `x0` (suppose it is a `double`), both `p(x0)` and `cp(x0)` return the value of the polynomial at $x = x_0$, i.e. $P\left(x_0\right)$.

Arithmetic operations

The arithmetic operators (as well as the corresponding compound assignment operators) that need to be supported are as follows.

- `-a` (the unary negation operator), `a + b`, `a += b`, `a - b`, `a -= b`.
- `a * b`, `a *= b`.

Derivatives and integrals

For a polynomial `p`,

- `p.derivative()` should return the derivative of `p`, i.e. $\dfrac{\mathrm{d}p(x)}{\mathrm{d}x}$. Note that this should still be a polynomial.
- `p.integral()` should return the integral of `p`, defined as

  $$\int_0^x p(t)\mathrm{d}t.$$

  Note that this should still be a polynomial.

Notes

For each member function, should it be `const`, or non-`const`, or having the "`const` vs non-`const` overloads"? Think about these questions carefully before coding.

Regarding the floating-point errors: we will not test this on purpose, but such problems might show up accidentally and cause unexpected results. To avoid such risks, it is suggested to always use `isZero(x)` and `isZero(a - b)`, instead of directly comparing floating-point numbers.

Compile-time requirements test

We have provided a compile-time test `compile_test.cpp` for you.