数据结构与算法 — Project 1 实验报告

廖海涛, SID 24344064

2025 年 10 月 14 日

1. 程序功能

一个一元稀疏多项式简单计算器和算数表达式求值计算器，界面为 CLI。支持的功能有

- `help` 显示帮助。
- `expr <expression>` 计算分式表达式。支持解析 $+, -, \times, \div, \text{power}$ 双目运算和 $+, -$ 的单目运算。
- `poly new <name>` 交互式创建多项式。
- `poly list` 列出已保存的多项式。
- `poly show <name> [-l, --latex]` 显示多项式，可选 LaTeX 格式，默认输出整数序列。
- `poly eval <name> <x>` 计算 $P(x)$。
- `poly deriv <name>` 输出 $P(x)$ 的导数。
- `poly add <A> <B>` 显示 $A(x) + B(x)$ 的结果。
- `poly sub <A> <B>` 显示 $A - B$ 的结果。
- `poly mul <A> <B>` 显示 $A \times B$ 的结果。
- `exit` 退出程序。

2. 功能演示

2.1. 算数表达式求值计算器

```
> expr 3*(7-2)
结果 = 15/1   (= 15)

> expr 8
结果 = 8/1   (= 8)

> expr 88 - 1 * 5
结果 = 83/1   (= 83)

> expr 1024 / 4 * 8
结果 = 2048/1   (= 2048)

> expr 1024 / (4 * 8)
结果 = 32/1   (= 32)
```

```
> expr (20 + 2) * (6 / 2)
结果 = 66/1   (= 66)

> expr 3 - 3 - 3
结果 = -3/1   (= -3)

> expr 8 / (9 - 9)
错误: division by zero

> expr (6 + 2 * (3 + 6 * (6 + 6)))
结果 = 156/1   (= 156)

> expr (((6+6) * 6 + 3) * 2 + 6) * 2
结果 = 312/1   (= 312)

> expr -2^(-3)
结果 = -1/8   (= -0.125)

> expr 2^(-1 - 1)^3
结果 = 1/256   (= 0.00390625)

> poly eval p61 3
P(3) = 39

> poly deriv p31 -l
LaTeX 格式: $6x^{5} + 5x^{4} + 4x^{3} + 3x^{2} + 2x + 1$
```

2.2. 一元稀疏多项式简单计算器

```
> poly new p11
输入项数量以及各项(系数 指数), 例如:
3  2 2  -1 1  5 0
表示 3 个项: 2x^2 - 1x + 5
> 3 2 1 5 8 -3.1 11
多项式 'p11' 已保存。

> poly add p11 p12 -l
add(p11, p12) = $-3.1x^{11} + 11x^{9} + 2x + 7$

> poly sub p21 p22 -l
```

```
sub(p21, p22) = $-7.8x^{15} - 1.2x^{9} - x + 12x^{-3}$

> poly add p31 p32 -l
add(p31, p32) = $x^{6} + x^{5} + x^{2} + x + 1$

> poly add p41 p42 -l
add(p41, p42) = $0$

> poly add p51 p52 -l
add(p51, p52) = $x^{200} + 2x^{100} + x$

> poly add p61 p62 -l
add(p61, p62) = $x^{3} + x^{2} + x$
```

2.3. 程序运行界面

```
                Polynomial & Expression
                    Calculator CLI


> help
可用命令:
  help                          显示帮助
  expr <expression>             计算分式四则表达式
  poly new <name>               交互式创建多项式
  poly list                     列出已保存的多项式
  poly show <name>              显示多项式
  poly eval <name> <x>          计算 P(x)
  poly deriv <name>             输出导数
  poly add <A> <B>              显示 A+B 的结果
  poly sub <A> <B>              显示 A-B 的结果
  poly mul <A> <B>              显示 A×B 的结果
  exit                          退出程序

> exit
再见!
```

3. 关键代码说明

要实现此程序，一共有如下关键要点：

1. 实现分数类，以给出精确计算结果。
2. 实现栈，用于表达式求值的运算符和运算数的堆栈。
3. 实现表达式求值函数，主要函数逻辑是逐个处理运算符，并用栈结构保证算符优先级。
4. 实现链表，并支持特殊操作，用以存储稀疏一元多项式。
5. 解析命令，实现 CLI。

3.1. 分数类实现

expression.hpp

```hpp
#pragma once

#include <string>
#include <iostream>

using i64 = long long;

struct Fraction {
    i64 numerator;
    i64 denominator;

    Fraction(i64 num = 0, i64 denom = 1);
    void normalize();

    friend Fraction operator+(const Fraction &a, const Fraction &b);
    Fraction& operator+=(const Fraction &other);
    Fraction operator-() const;
    friend Fraction operator-(const Fraction &a, const Fraction &b);
    Fraction& operator-=(const Fraction &other);
    friend Fraction operator*(const Fraction &a, const Fraction &b);
    Fraction& operator*=(const Fraction &other);
    friend Fraction operator/(const Fraction &a, const Fraction &b);
    Fraction& operator/=(const Fraction &other);
    friend Fraction operator^(const Fraction &base, int exponent);
    Fraction& operator^=(int exponent);
};
```

```
27
28 Fraction expression_evaluate(const std::string &expr);
29 std::ostream& operator<<(std::ostream &os, const Fraction &value);
```

分数需要重载最基本的四则运算，`normalize` 则用于将分数化为最简分数，实现上，需要用欧几里得算法求解 gcd。

nomalize()

```
1 void Fraction::normalize() {
2     if (denominator < 0) {
3         denominator = -denominator;
4         numerator = -numerator;
5     }
6     auto d = std::gcd(numerator, denominator);
7     numerator /= d;
8     denominator /= d;
9 }
```

此外还需支持乘方运算，为了保证结果仍是有理数，这里限制了指数只能为整数。特别的，定义 $0^0 = 1$。

power

```
1  Fraction operator^(const Fraction &base, int exponent) {
2      if (exponent == 0) {
3          return Fraction(1, 1);
4      }
5      if (base.numerator == 0 && exponent < 0) {
6          throw std::runtime_error("zero cannot be raised to negative power");
7      }
8      Fraction result(1, 1);
9      Fraction factor = base;
10     if (exponent < 0) {
11         factor = Fraction(base.denominator, base.numerator);
12         exponent = -exponent;
13     }
14     while (exponent) {
15         if (exponent & 1) {
16             result *= factor;
17         }
18         if (exponent > 1) {
19             factor *= factor;
20         }
```

```
        exponent >>= 1;
    }
    return result;
}
```

3.2. 栈实现

栈需要支持基本的查询和修改操作。对于内存管理，在容量不足时，加倍申请空间，保证时间复杂度为线性。

stack.hpp

```hpp
#pragma once

#include <cstddef>

template <typename T>
class Stack {
    public:
    Stack();
    explicit Stack(std::size_t initial_capacity); // initialize with given
        capacity
    Stack(const Stack &other); // copy constructor
    Stack(Stack &&other) noexcept; // move constructor
    ~Stack();

    Stack &operator=(const Stack &other);
    Stack &operator=(Stack &&other) noexcept;

    void push(const T &value);
    T pop();
    const T &top() const;
    bool empty() const;
    std::size_t size() const;
    void clear();

    private:
    static constexpr std::size_t DEFAULT_CAPACITY = 8;

    T *data_;
    std::size_t size_;
    std::size_t capacity_;
```

```
30
31      void ensure_capacity(std::size_t min_capacity);
32  };
```

这里展示 ensure_capacity 的实现。

ensure_capacity

```
1   template <typename T>
2   void Stack<T>::ensure_capacity(std::size_t min_capacity) {
3       if (capacity_ >= min_capacity) {
4           return;
5       }
6       std::size_t new_capacity = capacity_ == 0 ? DEFAULT_CAPACITY : capacity_;
7       while (new_capacity < min_capacity) {
8           new_capacity *= 2;
9       }
10      T *new_data = allocate<T>(new_capacity);
11      for (std::size_t i = 0; i < size_; ++i) {
12          new_data[i] = data_[i];
13      }
14      delete[] data_;
15      data_ = new_data;
16      capacity_ = new_capacity;
17  }
```

3.3. 表达式求值函数实现

表达式求值的流程如下。不断将数和算符压栈。如果当前压入栈的算符比栈顶算符的优先级低，则需要先执行栈顶的算符。如果优先级相等并且此算符是右结合（如乘方），则也需要先执行栈顶的算符。

对于单目运算符，将 $-x$ 视为 $0-x$，归约为双目运算符。为了保证运算符优先级，形如 3×-2 的表达式将被判定为非法表达式，要求写作 $3\times(-2)$。

evaluate_expression()

```
1   Fraction expression_evaluate(const std::string &expr) {
2       std::string input = remove_spaces(expr);
3       if (input.empty()) {
4           throw std::runtime_error("expression is empty");
5       }
6
7       Stack<Fraction> values;
```

```
    Stack<char> operators;

    std::size_t index = 0;
    while (index < input.size()) {
        char ch = input[index];

        if (std::isdigit(static_cast<unsigned char>(ch))) {
            Fraction number{parse_integer(input, index), 1};
            values.push(number);
            continue;
        }

        if (ch == '(') {
            operators.push('(');
            ++index;
            continue;
        } else if (ch == ')') {
            ++index;
            collapse(values, operators);
            continue;
        } else {
            if (is_unary(ch, input, index)) {
                int count = 0;
                while (index < input.size() && (input[index] == '+' || input[
                    index] == '-')) {
                    if (input[index] == '-') {
                        ++count;
                    }
                    if (index > 0 && (input[index - 1] == '*' || input[index -
                        1] == '/' || input[index - 1] == '^')) {
                        throw std::runtime_error("invalid use of unary
                            operator after '*', '/' or '^'");
                    }
                    ++index;
                }
                values.push(Fraction(0, 1));
                ch = (count % 2 == 0) ? '+' : '-';
                --index; // adjust for the upcoming ++index
            }

            process_operator(values, operators, ch);
```

```
            ++index;
        }
    }

    while (!operators.empty()) {
        char op = operators.pop();
        if (op == '(' || op == ')') {
            throw std::runtime_error("mismatched parentheses");
        }
        apply_operator(values, op);
    }

    if (values.size() != 1) {
        throw std::runtime_error("malformed expression");
    }
    return values.pop();
}
```

3.4. 链表和多项式实现

先给出多项式需要支持的基本操作。元素用有序链表存储。链表首不为无意义节点。关键需要实现的函数是 `addTerm()`。

polynomial.hpp

```
#pragma once

struct PolyTerm {
    double coefficient;
    int exponent;
    PolyTerm *next;
};

struct Polynomial {
    Polynomial();
    Polynomial(const Polynomial& other);
    Polynomial(Polynomial&& other) noexcept;
    ~Polynomial();

    Polynomial& operator=(Polynomial other) noexcept; // copy-swap
    friend void swap(Polynomial& a, Polynomial& b) noexcept;

```

```
    friend Polynomial operator+(const Polynomial &a, const Polynomial &b);
    Polynomial& operator+=(const Polynomial &other);
    Polynomial operator-() const;
    friend Polynomial operator-(const Polynomial &a, const Polynomial &b);
    Polynomial& operator-= (const Polynomial &other);
    friend Polynomial operator*(const Polynomial &a, const Polynomial &b);
    Polynomial& operator*=(const Polynomial &other);

    double evaluate(double x) const;
    Polynomial derivative() const;
    void addTerm(double coefficient, int exponent);

    void print() const;
    void printLaTeX() const;

    private:
    PolyTerm *head;
    // PolyTerms in descending order of exponent
};

Polynomial createPoly();
```

`addTerm()` 函数的功能是，支持插入一个单项式，保持链表的有序性。

addTerm

```
void insert_term(PolyTerm *&head, double coefficient, int exponent) {
    if (is_zero(coefficient)) {
        return;
    }

    PolyTerm **current = &head; // points to the pointer to the current node
    while (*current && (*current)->exponent > exponent) {
        current = &((*current)->next);
    }

    if (*current && (*current)->exponent == exponent) {
        (*current)->coefficient += coefficient;
        if (is_zero((*current)->coefficient)) {
            PolyTerm *to_delete = *current;
            *current = (*current)->next;
            delete to_delete;
        }
```

```
18          return;
19      }
20
21      PolyTerm *node = new PolyTerm{coefficient, exponent, *current};
22      *current = node;
23  }
```

3.4. CLI 命令解析

首先实现 `trim()` 函数，将命令首尾的空白字符去掉。然后找到第一个单词作为命令（不区分大小写），剩下的作为参数。

split_command()

```
1  std::pair<std::string, std::string> split_command(const std::string &line) {
2      ...
3      auto pos = trimmed.find_first_of(" \t");
4      if (pos == std::string::npos) {
5          std::string cmd = trimmed;
6          std::transform(cmd.begin(), cmd.end(), cmd.begin(), [](unsigned char c
               ) { return static_cast<char>(std::tolower(c)); });
7          return {cmd, ""};
8      }
9      std::string cmd = trimmed.substr(0, pos);
10     std::transform(cmd.begin(), cmd.end(), cmd.begin(), [](unsigned char c) {
           return static_cast<char>(std::tolower(c)); });
11     std::string rest = trim(trimmed.substr(pos + 1));
12     return {cmd, rest};
13 }
```

其次为了保存多项式，使用 `std::unordered_map` 保存 context。

```
1  struct CLIContext {
2      std::unordered_map<std::string, Polynomial> polynomials;
3  };
```

最后根据解析出的命令执行对应功能即可。

4. 程序运行方式说明

代码文件目录如下：

```
Calculator
  report.pdf
  statement.pdf

  include
     expression.hpp
     polynomial.hpp
     stack.hpp

  src
     expression.cpp
     main.cpp
     polynomial.cpp
     stack.cpp

  test
     expression.in
     polynomial.in
     polynomial.out
     test_expression.cpp
     test_polynomial.cpp
```

项目基于 C++20，使用的 GCC 的版本为 `version 15.1.0 (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders, r4)`。使用的编译命令如下

```
g++ -std=c++20 -I include src/*.cpp -o cli.exe
```