AP CSP Create Task

```
#include <string>
#include <stdio.h>
#include <string.h>
#include <vector>
#include <iostream>
#include <algorithm>
#include <fstream>
#include <stdlib.h>
#include <math.h>
#include <iomanip>
using namespace std;

ifstream fin("input.in");
ofstream fout("output.out");

struct expression
{
    bool type; // true for product, false for sum

    double power;

    double coefficient;

    vector <struct expression> inside;

    // returns if expression evaluates to 0
    bool is_zero() const
    {
        if(coefficient == 0)
        {
            return true;
        }
        return false;
    }

    // returns if function is a constant
    bool is_constant() const
    {
        if(coefficient != 0 and power == 0)
        {
            return true;
        }
        return false;
    }

    // used by the sort functions, it just needs to put identical things next to each other
    bool operator< (const struct expression other) const
    {
        if(inside.size() == other.inside.size()) // on this case, we want to separate expressions that are not equal from each other, but is_equal isn't defined...
        {
            return power < other.power;
        }

        return inside.size() < other.inside.size();
    }
};

// derivative functions
struct expression product_rule(struct expression in);
struct expression power_rule(struct expression in);
struct expression chain_rule(struct expression in);
struct expression derivative(struct expression in);
struct expression derivative_inside(struct expression in);

// input functions
int read_exp(void);
struct expression read_in_expression(void);

// comparison functions
bool is_equal(struct expression a, struct expression b);
bool by_power(struct expression a, struct expression b);

// output functions
void output_expression(struct expression in);
struct expression clean_up(struct expression in);

// because-I-needed-it function
int exp(int base, int power);

int main(void)
{
    // =   <-- because some buttons on my keyboard don't work (for copying)

    int n;
    fin >> n; // takes in a number of expressions

    for(int i = 0; i < n; i++)
    {
        struct expression test;

        test = read_in_expression();

        test = clean_up(test); // just to make sure final answer is in a simpler form, which starts by making sure the input is in simplest form

        struct expression ans = derivative(test);

        output_expression(ans);
        fout << endl;
    }
}

struct expression product_rule(struct expression in)
{
    struct expression ans;

    if(in.type != true)
    {
        cout << "product rule was given a sum" << endl;
        exit(2);
    }

    ans.type = false; // sum of the products, it's a holder for the result
    ans.coefficient = ans.power = 1; // just defaults

    struct expression terms; // this is the stuff multiplied together
    terms.inside = in.inside;
    terms.coefficient = terms.power = 1;
    terms.type = true;

    for(int i = 0; i < terms.inside.size(); i++) // applying the generalized product rule
    {
        struct expression temp;
        temp = terms.inside[i];

        terms.inside[i] = derivative(temp);

        ans.inside.push_back(terms);

        terms.inside[i] = temp;
    }

    return ans;
}

struct expression power_rule(struct expression in)
{
    struct expression ans;
    if(in.power == 0) // it is a constant so just set everything to 0
    {
        ans.power = 0;
        ans.coefficient = 0;
        return ans;
    }

    ans.coefficient = in.coefficient * in.power; // apply the power rule
    ans.power = in.power - 1;
    ans.type = in.type;
    ans.inside = in.inside;

    return ans;
}

struct expression chain_rule(struct expression in)
{
    struct expression ans;
    ans.type = true; // product of f'(g(x)) and g'(x), so we need a new expression to contain them

    ans.inside.push_back(derivative_inside(in));

    in = power_rule(in); // only affects outer layer, f'(g(x))

    ans.inside.push_back(in);

    ans.coefficient = 1;
    ans.power = 1;

    return ans;
}

struct expression derivative(struct expression in)
{
    if(in.inside.size() == 0)
    {
        return power_rule(in);
    }
    else
    {
        return chain_rule(in);
    }
}

struct expression derivative_inside(struct expression in)
{
    struct expression ans;
    if(in.inside.size() == 0)
    {
        ans.power = 0;
        ans.coefficient = 0;
        ans.type = false;
        return ans;
    }
    if(in.type == false) // added together
    {
        ans.type = false; // sum of the individual derivatives
        ans.coefficient = 1;
        ans.power = 1;
        for(int i = 0; i < in.inside.size(); i++)
        {
            ans.inside.push_back(derivative(in.inside[i]));
        }
        return ans;
    }
    if(in.type == true) // multiplied together
    {
        // product rule...
        return product_rule(in);
    }
}

int read_exp(void) // just a simple helper function
{
    int exponent;
    fin >> exponent;
    return exponent;
}

struct expression read_in_expression(void)
{
    char in;
    fin >> in;

    int coefficient;

    if((in >= '0' and in <= '9') or in == '-') // it's a number
    {
        fin.putback(in);
        fin >> coefficient;
    }
    else
    {
        coefficient = 1;
        fin.putback(in);
    }

    char x;
    fin >> x;

    int exponent;
    struct expression ans;

    if(x != 'x' and x != '(')
    {
        fin.putback(x);

        exponent = 0; // it's a constant
        ans.power = exponent;
        ans.coefficient = coefficient;

        return ans;
    }

    if(x == 'x')
    {
        // return the expression with an exponent, do nothing, the code after will do that
    }
    if(x == '(')
    {
        // recursively read more expressions

        struct expression temp;
        temp = read_in_expression();

        ans.inside.push_back(temp);

        char next;

        do
        {
            fin >> next;

            if(next != '+' and next != '-' and next !=')' and next != '*')
            {
                cout << "Expected *, +, - or )" << endl;
                exit(-1);
            }

            if(next == '+')
            {
                if(ans.type == true and ans.inside.size() > 1) // conflicts with previous operation
                {
                    cout << "Expected * but got -" << endl;
                }
                temp = read_in_expression();

                ans.type = false;

                ans.inside.push_back(temp);
            }
            if(next == '-')
            {
                if(ans.type == true and ans.inside.size() > 1) // conflicts with previous operation
                {
                    cout << "Expected * but got -" << endl;
                }
                temp = read_in_expression();
                temp.coefficient *= -1;

                ans.type = false;

                ans.inside.push_back(temp);
            }
            if(next == '*')
            {
                if(ans.type == false and ans.inside.size() > 1) // conflicts with previous operation
                {
                    cout << "Expected +/- but got *" << endl;
                }

                temp = read_in_expression();

                ans.type = true;

                ans.inside.push_back(temp);
            }
            if(next == ')')
            {
                break;
            }
        }while(next != ')');
    }

    char carat;
    fin >> carat;

    if(carat != '^')
    {
        fin.putback(carat);

        exponent = 1;
    }
    else
    {
        exponent = read_exp();
    }
    ans.power = exponent;
    ans.coefficient = coefficient;

    return ans;
}


bool is_equal(struct expression a, struct expression b) // only checks if stuff inside is equal
{
    if(a.inside.size() == 0 and b.inside.size() == 0)
    {
        return true;
    }
    if(a.inside.size() != b.inside.size())
    {
        return false;
    }

    sort(a.inside.begin(), a.inside.end(), by_power); // put stuff in order, puts equal stuff together
    sort(b.inside.begin(), b.inside.end(), by_power);

    for(int i = 0; i < a.inside.size(); i++)
    {
        if(a.inside[i].coefficient != b.inside[i].coefficient)
        {
            return false;
        }
        if(a.inside[i].power != b.inside[i].power)
        {
            return false;
        }
        if(is_equal(a.inside[i], b.inside[i]) == false)
        {
            return false;
        }
    }

    return true;
}

bool by_power(struct expression a, struct expression b)
{
    if(a.power == b.power)
    {
        return a.coefficient < b.coefficient;
    }
    return a.power > b.power;
}

void output_expression(struct expression in)
{
    in = clean_up(in);
    sort(in.inside.begin(), in.inside.end(), by_power);
    if(in.coefficient == 0) // should have been cleaned up already? but just in case
    {
        fout << "0";
        return;
    }
    if(in.power == 0)
    {
        fout << in.coefficient;
        return;
    }

    if(in.inside.size() == 0)
    {
        if(in.power == 0)
        {
            fout << in.coefficient;
        }
        else
        {
            if(in.coefficient != 1)
            {
                fout << in.coefficient;
            }
            fout << "x";
            if(in.power != 1)
            {
                fout << "^" << in.power;
            }
        }
        return;
    }

    if(in.coefficient != 1)
    {
        fout << in.coefficient;
    }

    if(in.power == 0) // only care about coefficient
    {
        return;
    }

    if(in.inside.size() > 1)
        fout << "(";

    for(int i = 0; i < in.inside.size(); i++) // recursively print stuff inside
    {
        output_expression(in.inside[i]);
        if(i < in.inside.size() - 1)
        {
            if(in.type)
            {
                fout << " * ";
            }
            else
            {
                fout << " + ";
            }
        }
    }
    if(in.inside.size() > 1)
        fout << ")";

    if(in.power != 1)
    {
        fout << "^" << in.power;
    }
}

struct expression clean_up(struct expression in)
{
    for(int i = 0; i < in.inside.size(); i++)
    {
        clean_up(in.inside[i]); // clean up those first
    }

    if(in.inside.size() == 1) // bring everything up a level
    {
        in.coefficient *= exp(in.inside[0].coefficient, in.power);
        in.power *= in.inside[0].power;

        in.inside = in.inside[0].inside;

        return in;
    }

    if(in.type == true) // product
    {
        // clean constants
        for(int i = 0; i < in.inside.size(); i++)
        {
            if(in.inside[i].is_constant())
            {
                in.coefficient *= exp(in.inside[0].coefficient, in.power);

                in.inside.erase(in.inside.begin() + i);
                i--;
            }
        }

        // could also merge stuff that is the same base

        sort(in.inside.begin(), in.inside.end());

        for(int i = 1; i < in.inside.size(); i++)
        {
            if(is_equal(in.inside[i], in.inside[i - 1])) // they have the same base
            {
                in.inside[i].power += in.inside[i - 1].power;
                in.inside[i].coefficient *= in.inside[i - 1].coefficient;

                in.inside.erase(in.inside.begin() + i - 1);
                i--;
            }
        }

        return in;
    }

    else // sum
    {
        // clean zeroes
        for(int i = 0; i < in.inside.size(); i++)
        {
            if(in.inside[i].is_zero())
            {
                in.inside.erase(in.inside.begin() + i);
                i--;
            }
        }

        // could also merge stuff that is the same base and power

        sort(in.inside.begin(), in.inside.end());

        for(int i = 1; i < in.inside.size(); i++)
        {
            if(is_equal(in.inside[i], in.inside[i - 1]) and in.inside[i].power == in.inside[i - 1].power) // they have the same base and power
            {

                in.inside[i].coefficient += in.inside[i - 1].coefficient;

                in.inside.erase(in.inside.begin() + i - 1);
                i--;
            }
        }

        return in;
    }
}


int exp(int base, int power) // naive exponent function
{
    int ans = 1;
    for(int i = 0; i < power; i++)
    {
        ans *= base;
    }
    return ans;
}
```

PDF document made with CodePrint using Prism