

```
/*
 * Polynomial.cpp
 *
 * Created on: Dec 28, 2009
 * Author: koscum
 */

#include "Polynomial.h"
#include <algorithm>

namespace GF
{
    Polynomial::Polynomial():
        order(-1), coefficients()
    {
        // Empty body;
    }

    Polynomial::Polynomial(int c):
        order(0), coefficients(1, c)
    {
        // Empty body;
    }

    Polynomial::Polynomial(int c, int o):
        order(o), coefficients(std::max(0, o + 1), 0)
    {
        if (order < 0)
        {
            throw Error(Error::ORDER_LESS_THAN_ZERO);
        }

        coefficients[o] = c;

        Truncate();
    }

    Polynomial::Polynomial(const int* c, int o):
        order(o), coefficients(std::max(0, o + 1), 0)
    {
        if (order < 0)
        {
            throw Error(Error::ORDER_LESS_THAN_ZERO);
        }

        for (int i = 0; i <= order; i++)
        {
            coefficients[i] = c[i];
        }

        Truncate();
    }

    Polynomial::Polynomial(const std::vector<int>& c):
        order(c.size() - 1), coefficients(c)
    {
        Truncate();
    }
}
```

```
}

Polynomial::Polynomial(const Polynomial& p)
{
    Copy(p);
}

Polynomial::~~Polynomial()
{
    Free();
}

Polynomial& Polynomial::Add(const Polynomial& p) const
{
    return Polynomial::AddSub((*this), p, 0, true);
}

Polynomial& Polynomial::Sub(const Polynomial& p) const
{
    return Polynomial::AddSub((*this), p, 0, false);
}

Polynomial& Polynomial::Mul(const Polynomial& p) const
{
    return Polynomial::MulPow((*this), p, 0, true);
}

Polynomial& Polynomial::Div(const Polynomial& p) const
{
    return Polynomial::DivMod((*this), p, 0, true);
}

Polynomial& Polynomial::Mod(const Polynomial& p) const
{
    return Polynomial::DivMod((*this), p, 0, false);
}

Polynomial& Polynomial::Pow(const Polynomial& p) const
{
    return Polynomial::MulPow((*this), p, 0, false);
}

Polynomial& Polynomial::Add(const Polynomial& p, const int b) const
{
    return Polynomial::AddSub((*this), p, b, true);
}

Polynomial& Polynomial::Sub(const Polynomial& p, const int b) const
{
    return Polynomial::AddSub((*this), p, b, false);
}

Polynomial& Polynomial::Mul(const Polynomial& p, const int b) const
{
    return Polynomial::MulPow((*this), p, b, true);
}
```

```

Polynomial& Polynomial::Div(const Polynomial& p, const int b) const
{
    return Polynomial::DivMod((*this), p, b, true);
}

Polynomial& Polynomial::Mod(const Polynomial& p, const int b) const
{
    return Polynomial::DivMod((*this), p, b, false);
}

Polynomial& Polynomial::Pow(const Polynomial& p, const int b) const
{
    return Polynomial::MulPow((*this), p, b, false);
}

Polynomial& Polynomial::AddSub(const Polynomial& a, const Polynomial& b, const int base,
const bool mode)
{
    Polynomial polynomialA = Polynomial(a);
    Polynomial polynomialB = Polynomial(b);

    if (base != 0)
    {
        polynomialA = polynomialA.Normalise(base);
        polynomialB = polynomialB.Normalise(base);
    }

    Polynomial* result;

    if ((polynomialA == Polynomial::ZERO) && (mode == true))
    {
        result = new Polynomial(polynomialB);
    }
    else if (polynomialB == Polynomial::ZERO)
    {
        result = new Polynomial(polynomialA);
    }
    else
    {
        int resultOrder;
        std::vector<int> resultCoefficients;

        resultOrder = std::max(polynomialA.GetOrder(), polynomialB.GetOrder());
        resultCoefficients = std::vector<int>(resultOrder + 1, 0);

        for (int i = 0; i <= resultOrder; i++)
        {
            if (mode == true)
            {
                resultCoefficients[i] = polynomialA[i] + polynomialB[i];
            }
            else
            {
                resultCoefficients[i] = polynomialA[i] - polynomialB[i];
            }
        }
    }
}

```

```

        if (base != 0)
        {
            result = new Polynomial(Polynomial(resultCoefficients).Normalise(base));
        }
        else
        {
            result = new Polynomial(resultCoefficients);
        }
    }

    return (*result);
}

Polynomial& Polynomial::MulPow(const Polynomial& a, const Polynomial& b, const int base,
const bool mode)
{
    Polynomial polynomialA = Polynomial(a);
    Polynomial polynomialB = Polynomial(b);

    if (base != 0)
    {
        polynomialA = polynomialA.Normalise(base);
        polynomialB = polynomialB.Normalise(base);
    }

    Polynomial* result;

    if ((polynomialB.GetOrder() != 0) && (mode == false))
    {
        throw Error(Error::ORDER_NOT_ZERO);
    }
    else if (((polynomialA == Polynomial::ZERO) || (polynomialB == Polynomial::ZERO)) &&
(mode == true))
    {
        result = new Polynomial(Polynomial::ZERO);
    }
    else if ((polynomialA == Polynomial::ONE) && (mode == true))
    {
        result = new Polynomial(polynomialB);
    }
    else if (((polynomialA == Polynomial::ONE) || (polynomialA == Polynomial::ZERO)) && (
mode == false))
    {
        result = new Polynomial(polynomialA);
    }
    else if ((polynomialB == Polynomial::ZERO) && (mode == false))
    {
        result = new Polynomial(Polynomial::ONE);
    }
    else if (polynomialB == Polynomial::ONE)
    {
        result = new Polynomial(polynomialA);
    }
    else
    {
        int mulCount;
        int resultOrder;

```

```

std::vector<int> resultCoefficients;

if (mode == true)
{
    mulCount = 1;
    resultOrder = polynomialA.GetOrder() + polynomialB.GetOrder();
}
else
{
    mulCount = polynomialB[0];
    resultOrder = polynomialA.GetOrder() * mulCount;

    polynomialB = polynomialA;
}

resultCoefficients = std::vector<int>(resultOrder + 1, 0);

for (int i = 0; i <= polynomialA.GetOrder(); i++)
{
    resultCoefficients[i] = polynomialA[i];
}

int i = 1;
do
{
    std::vector<int> temp = std::vector<int>(resultOrder + 1, 0);

    for (int j = 0; j <= (polynomialA.GetOrder() * i); j++)
    {
        for (int k = 0; k <= polynomialB.GetOrder(); k++)
        {
            temp[j + k] += resultCoefficients[j] * polynomialB[k];
        }
    }
    resultCoefficients = temp;

    mulCount--;
    i++;
}
while (mulCount > 1);

if (base != 0)
{
    result = new Polynomial(Polynomial(resultCoefficients).Normalise(base));
}
else
{
    result = new Polynomial(resultCoefficients);
}
}

return (*result);
}

```

```

Polynomial& Polynomial::DivMod(const Polynomial& a, const Polynomial& b, const int base,
const bool mode)
{

```

```

Polynomial polynomialA = Polynomial(a);
Polynomial polynomialB = Polynomial(b);

if (base != 0)
{
    polynomialA = polynomialA.Normalise(base);
    polynomialB = polynomialB.Normalise(base);
}

Polynomial* divResult;
Polynomial* remainder;

if (polynomialB == Polynomial::ZERO)
{
    throw Error(Error::DIVISION_BY_ZERO);
}
else if (polynomialB == Polynomial::ONE)
{
    divResult = new Polynomial(polynomialA);
    remainder = new Polynomial(Polynomial::ZERO);
}
else if (polynomialA.GetOrder() < polynomialB.GetOrder())
{
    divResult = new Polynomial(Polynomial::ZERO);
    remainder = new Polynomial(polynomialA);
}
else if (polynomialA == polynomialB)
{
    divResult = new Polynomial(Polynomial::ONE);
    remainder = new Polynomial(Polynomial::ZERO);
}
else if (((polynomialA[polynomialA.GetOrder()] % polynomialB[polynomialB.GetOrder()])
!= 0) && (base == 0))
{
    divResult = new Polynomial(Polynomial::ZERO);
    remainder = new Polynomial(polynomialA);
}
else
{
    int i;
    int divResultOrder;
    std::vector<int> divResultCoefficients;

    divResultOrder = polynomialA.GetOrder() - polynomialB.GetOrder();
    divResultCoefficients = std::vector<int>(divResultOrder + 1, 0);

    remainder = new Polynomial(polynomialA);

    if (base == 0)
    {
        do
        {
            i = (*remainder).GetOrder() - polynomialB.GetOrder();
            if (i >= 0)
            {
                divResultCoefficients[i] = (*remainder)[(*remainder).GetOrder()] /
polynomialB[polynomialB.GetOrder()];

```

```

        (*remainder) = (*remainder) - (Polynomial(divResultCoefficients[i], i)
* polynomialB);
    }
}
while (i > 0);
}
else
{
    do
    {
        i = (*remainder).GetOrder() - polynomialB.GetOrder();
        if (i >= 0)
        {
            divResultCoefficients[i] = Polynomial::ModuoDiv((*remainder)[(*
remainder).GetOrder()], polynomialB[polynomialB.GetOrder()], base);
            (*remainder) = (*remainder).Sub(Polynomial(divResultCoefficients[i], i
).Mul(polynomialB, base), base);
        }
    }
    while (i > 0);
}

divResult = new Polynomial(divResultCoefficients);
}

if (mode == true)
{
    return (*divResult);
}
else
{
    return (*remainder);
}
}

int Polynomial::ModuoDiv(const int a, const int b, const int base)
{
    int result = 0;

    for (int i = 1; i < base; i++)
    {
        if (((b * i) % base) == a)
        {
            result = i;
            break;
        }
    }

    return result;
}

Polynomial& operator +(const Polynomial& a, const Polynomial& b)
{
    return a.Add(b);
}

Polynomial& operator -(const Polynomial& a, const Polynomial& b)

```

```
{
    return a.Sub(b);
}

Polynomial& operator *(const Polynomial& a, const Polynomial& b)
{
    return a.Mul(b);
}

Polynomial& operator /(const Polynomial& a, const Polynomial& b)
{
    return a.Div(b);
}

Polynomial& operator %(const Polynomial& a, const Polynomial& b)
{
    return a.Mod(b);
}

Polynomial& operator ^(const Polynomial& a, const Polynomial& b)
{
    return a.Pow(b);
}

Polynomial& Polynomial::operator =(const Polynomial& p)
{
    Copy(p);

    return (*this);
}

Polynomial& Polynomial::operator +=(const Polynomial& p)
{
    (*this) = (*this) + p;

    return (*this);
}

Polynomial& Polynomial::operator -=(const Polynomial& p)
{
    (*this) = (*this) - p;

    return (*this);
}

Polynomial& Polynomial::operator *=(const Polynomial& p)
{
    (*this) = (*this) * p;

    return (*this);
}

Polynomial& Polynomial::operator /=(const Polynomial& p)
{
    (*this) = (*this) / p;

    return (*this);
}
```



```
}

Polynomial& Polynomial::operator %=(const Polynomial& p)
{
    (*this) = (*this) % p;

    return (*this);
}

Polynomial& Polynomial::operator ^=(const Polynomial& p)
{
    (*this) = (*this) ^ p;

    return (*this);
}

bool operator ==(const Polynomial& polynomialA, const Polynomial& polynomialB)
{
    if (polynomialA.GetOrder() != polynomialB.GetOrder())
    {
        return false;
    }
    else
    {
        for (int i = 0; i <= polynomialA.GetOrder(); i++)
        {
            if (polynomialA[i] != polynomialB[i])
            {
                return false;
            }
        }

        return true;
    }
}

bool operator !=(const Polynomial& polynomialA, const Polynomial& polynomialB)
{
    return !(polynomialA == polynomialB);
}

bool operator <(const Polynomial& polynomialA, const Polynomial& polynomialB)
{
    if (polynomialA.GetOrder() < polynomialB.GetOrder())
    {
        return true;
    }
    else if (polynomialA.GetOrder() == polynomialB.GetOrder())
    {
        for (int i = polynomialA.GetOrder(); i >= 0; i--)
        {
            if (polynomialA[i] > polynomialB[i])
            {
                return false;
            }
            else if (polynomialA[i] < polynomialB[i])
            {

```

```

        return true;
    }
}

return false;
}

bool operator >(const Polynomial& polynomialA, const Polynomial& polynomialB)
{
    if ((polynomialA == polynomialB) || (polynomialA < polynomialB))
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool operator <=(const Polynomial& polynomialA, const Polynomial& polynomialB)
{
    if (polynomialA > polynomialB)
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool operator >=(const Polynomial& polynomialA, const Polynomial& polynomialB)
{
    if (polynomialA < polynomialB)
    {
        return false;
    }
    else
    {
        return true;
    }
}

Polynomial& Polynomial::Normalise(int mod) const
{
    int resultOrder;
    int* resultCoefficients;
    int temp;
    Polynomial* result;

    resultOrder = (*this).GetOrder();
    resultCoefficients = new int[resultOrder + 1];
    for (int i = 0; i <= resultOrder; i++)
    {
        temp = (*this)[i] % mod;
        if (temp < 0)

```

```
{
    temp += mod;
}

resultCoefficients[i] = temp;
}

result = new Polynomial(resultCoefficients, resultOrder);

return (*result);
}

void Polynomial::Truncate()
{
    while (order > 0 && coefficients[order] == 0)
    {
        order--;
        coefficients.pop_back();
    }
}

int Polynomial::GetOrder() const
{
    return order;
}

int Polynomial::GetCoefficient(int i) const
{
    if (i < 0)
    {
        throw Error(Error::NEGATIVE_ORDER_OF_COEFFICIENT);
    }
    else if (i > order)
    {
        return 0;
    }

    return coefficients[i];
}

int Polynomial::operator [](int i) const
{
    return (*this).GetCoefficient(i);
}

void Polynomial::Free()
{
    // Empty body;
}

void Polynomial::Copy(const Polynomial& p)
{
    Free();

    order = p.order;

    coefficients = std::vector<int>(p.coefficients);
}
```

```

}

void Polynomial::Out(std::ostream& o)
{
    bool first = true;

    if ((*this).GetOrder() == 0)
    {
        if ((*this)[0] < 0)
        {
            o << " - " << -(*this)[0];
        }
        else
        {
            o << (*this)[0];
        }
    }
    else
    {
        for (int i = 0; i <= (*this).GetOrder(); i++)
        {
            if ((*this)[i] == 0)
            {
                continue;
            }
            else if ((*this)[i] > 0)
            {
                if (!first)
                {
                    o << " + ";
                }
                if ((i == 0) || ((*this)[i] != 1))
                {
                    o << (*this)[i];
                }
            }
            else
            {
                o << " - ";
                if ((i == 0) || ((*this)[i] != -1))
                {
                    o << -(*this)[i];
                }
            }

            if (i > 0)
            {
                o << "x";
                if (i != 1)
                {
                    o << "^" << i;
                }
            }

            if (first)
            {
                first = false;
            }
        }
    }
}

```

```

    }
    }
}

std::ostream& operator <<(std::ostream& o, const Polynomial& p)
{
    bool first = true;

    o << "[";

    if (p.GetOrder() == 0)
    {
        if (p[0] < 0)
        {
            o << " - " << -p[0];
        }
        else
        {
            o << p[0];
        }
    }
    else
    {
        for (int i = 0; i <= p.GetOrder(); i++)
        {
            if (p[i] == 0)
            {
                continue;
            }
            else if (p[i] > 0)
            {
                if (!first)
                {
                    o << " + ";
                }
                if ((i == 0) || (p[i] != 1))
                {
                    o << p[i];
                }
            }
            else
            {
                o << " - ";
                if ((i == 0) || (p[i] != -1))
                {
                    o << -p[i];
                }
            }
        }

        if (i > 0)
        {
            o << "x";
            if (i != 1)
            {
                o << "^" << i;
            }
        }
    }
}

```

```
        }

        if (first)
        {
            first = false;
        }
    }

    o << " ]";

    return o;
}

const Polynomial Polynomial::ZERO = Polynomial(0);
const Polynomial Polynomial::ONE = Polynomial(1);
}
```