

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

#include "GF.h"
#include <cmath>

namespace GF
{
    Inverse::Inverse():
        add(-1), mul(-1)
    {
        // Empty body;
    }

    GF::GF(int c)
    {
        if (IsPrimePower(c))
        {
            Initialize(GF::BaseOf(c), GF::PowerOf(c), GF::FindIrreduciblePolynomial(GF::
BaseOf(c), GF::PowerOf(c)));
        }
        else
        {
            throw Error(Error::BASE_NOT_PRIME);
        }
    }

    GF::GF(int b, int p)
    {
        Initialize(b, p, GF::FindIrreduciblePolynomial(b, p));
    }

    GF::GF(int c, const Polynomial& i)
    {
        if (IsPrimePower(c))
        {
            Initialize(GF::BaseOf(c), GF::PowerOf(c), i);
        }
        else
        {
            throw Error(Error::BASE_NOT_PRIME);
        }
    }

    GF::GF(int b, int p, const Polynomial& i)
    {
        Initialize(b, p, i);
    }

    GF::GF(const GF& gf)
    {
        Copy(gf);
    }
}
```

```

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

Polynomial& GF::Add(const Polynomial& a, const Polynomial& b) const
{
    Polynomial* result;

    result = new Polynomial;

    if ((*this).IsExtended())
    {
        (*result) = a.Add(b, base).Mod(irreduciblePolynomial, base);
    }
    else
    {
        (*result) = a.Add(b, base);
    }

    return (*result);
}

Polynomial& GF::Sub(const Polynomial& a, const Polynomial& b) const
{
    Polynomial* result;

    result = new Polynomial;

    if ((*this).IsExtended())
    {
        (*result) = a.Add(GetAddInverse(GetIndexof(b)), base).Mod(irreduciblePolynomial,
base);
    }
    else
    {
        (*result) = a.Sub(b, base);
    }

    return (*result);
}

Polynomial& GF::Mul(const Polynomial& a, const Polynomial& b) const
{
    Polynomial* result;

    result = new Polynomial;

    if ((*this).IsExtended())
    {
        (*result) = a.Mul(b, base).Mod(irreduciblePolynomial, base);
    }
    else
    {
        (*result) = a.Mul(b, base);
    }
}

```

```

    return (*result);
}

Polynomial& GF::Div(const Polynomial& a, const Polynomial& b) const
{
    Polynomial* result;

    result = new Polynomial;

    if ((*this).IsExtended())
    {
        (*result) = a.Mul(GetMulInverse(GetIndexof(b)), base).Mod(irreduciblePolynomial,
base);
    }
    else
    {
        (*result) = a.Div(b, base);
    }

    return (*result);
}

Polynomial& GF::Mod(const Polynomial& a, const Polynomial& b) const
{
    Polynomial* result;

    result = new Polynomial;

    if ((*this).IsExtended())
    {
        (*result) = Sub(a, Mul(Div(a, b), b));
    }
    else
    {
        (*result) = a.Mod(b, base);
    }

    return (*result);
}

Polynomial& GF::Pow(const Polynomial& a, const Polynomial& b) const
{
    if (b.GetOrder() != 0)
    {
        throw Error(Error::ORDER_NOT_ZERO);
    }

    Polynomial* result;

    result = new Polynomial(Polynomial::ZERO);

    if ((*this).IsExtended())
    {
        Polynomial power = b;

        if (power == Polynomial::ZERO)

```

```
{
    (*result) = (Polynomial::ONE);
}
else
{
    (*result) = a;
}

while (power > Polynomial::ONE)
{
    (*result) = Mul((*result), a);

    power -= Polynomial::ONE;
}
}
else
{
    (*result) = a.Pow(b, base);
}
return (*result);
}

int GF::GetCharasteristic() const
{
    return (int) pow(base, power);
}

int GF::GetBase() const
{
    return base;
}

int GF::GetPower() const
{
    return power;
}

GF& GF::GetBaseField() const
{
    GF* result;

    result = new GF(base);

    return (*result);
}

const Polynomial& GF::GetAlpha() const
{
    return elements[alpha];
}

const Polynomial& GF::GetIrreduciblePolynomial() const
{
    return irreduciblePolynomial;
}
```

```
const Polynomial& GF::GetElement(int i) const
{
    if (i > (int) pow(base, power))
    {
        return Polynomial::ZERO;
    }

    return elements[i];
}

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

bool GF::IsBase() const
{
    if (power == 1)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool GF::IsExtended() const
{
    if (power == 1)
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool GF::IsPrimitive() const
{
    if (alpha == base)
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool GF::IsIrreducible(const Polynomial& irrPoly, int base, int power)
{
    std::vector<Polynomial> p;

    if (irrPoly.GetOrder() < power)
```

```
{
    return false;
}

p = GF::GenerateAllPolynomials(base, power);

if (power == 1)
{
    for (int i = 2; i < base; i++)
    {
        if (irrPoly.Mod(p[i]) == Polynomial::ZERO)
        {
            return false;
        }
    }
}
else
{
    for (int i = base; i < (int) p.size(); i++)
    {
        if (irrPoly.Mod(p[i], base) == Polynomial::ZERO)
        {
            return false;
        }
    }
}

return true;
}

bool GF::IsPrime(int a)
{
    if (a == 2)
    {
        return true;
    }

    if ((a % 2) == 0)
    {
        return false;
    }

    for (int i = 3; i < (a / 2); i += 2)
    {
        if ((a % i) == 0)
        {
            return false;
        }
    }

    return true;
}

bool GF::IsPrimePower(int a)
{
    if (GF::IsPrime(a))
    {

```

```

        return true;
    }
    else
    {
        for (int i = 2; i <= sqrt(a); i++)
        {
            if (GF::IsPrime(i) == true)
            {
                for (int j = 1; j == j; j++)
                {
                    if (((int) pow(i, j)) == a)
                    {
                        return true;
                    }
                    else if (((int) pow(i, j)) > a)
                    {
                        break;
                    }
                }
            }
        }
    }

    return false;
}

int GF::BaseOf(int a)
{
    if (GF::IsPrime(a))
    {
        return a;
    }
    else
    {
        if (GF::IsPrimePower(a) == true)
        {
            for (int i = 2; i <= sqrt(a); i++)
            {
                if (GF::IsPrime(i) == true)
                {
                    for (int j = 1; j == j; j++)
                    {
                        if (((int) pow(i, j)) == a)
                        {
                            return i;
                        }
                        else if (((int) pow(i, j)) > a)
                        {
                            break;
                        }
                    }
                }
            }
        }
    }

    return -1;
}

```

```

}

int GF::PowerOf(int a)
{
    if (GF::IsPrime(a))
    {
        return 1;
    }
    else
    {
        if (GF::IsPrimePower(a) == true)
        {
            for (int i = 2; i <= sqrt(a); i++)
            {
                if (GF::IsPrime(i) == true)
                {
                    for (int j = 1; j == j; j++)
                    {
                        if (((int) pow(i, j)) == a)
                        {
                            return j;
                        }
                        else if (((int) pow(i, j)) > a)
                        {
                            break;
                        }
                    }
                }
            }
        }
    }

    return -1;
}

const Polynomial& GF::FindIrreduciblePolynomial(int base, int power)
{
    Polynomial* result;
    std::vector<Polynomial> p;

    p = GF::GenerateAllPolynomials(base, power + 1);

    for (int i = 2; i < (int) p.size(); i++)
    {
        if (p[i].GetOrder() != power)
        {
            continue;
        }

        if (GF::IsIrreducible(p[i], base, power) == true)
        {
            result = new Polynomial(p[i]);
            return (*result);
        }
    }

    result = new Polynomial(Polynomial::ZERO);
}

```



```

    return (*result);
}

void GF::Initialize(int b, int p, const Polynomial& i)
{
    if (!GF::IsPrime(b))
    {
        throw Error(Error::BASE_NOT_PRIME);
    }
    else if (!GF::IsIrreducible(i, b, p))
    {
        throw Error(Error::POLYNOMIAL_NOT_IRREDUCIBLE);
    }
    else
    {
        base = b;
        power = p;
        irreduciblePolynomial = i;

        elements = GF::GenerateAllPolynomials(b, p);

        FindAlpha();

        GenerateAlphaTable();
        GenerateInversionsTable();
    }
}

void GF::Free()
{
    while (alphaTable.size() != 0)
    {
        alphaTable.pop_back();
    }

    while (inversionsTable.size() != 0)
    {
        inversionsTable.pop_back();
    }

    while (elements.size() != 0)
    {
        elements.pop_back();
    }
}

void GF::Copy(const GF& gf)
{
    Free();

    base = gf.base;
    power = gf.power;
    alpha = gf.alpha;
    irreduciblePolynomial = gf.irreduciblePolynomial;
    alphaTable = std::vector<int>(gf.alphaTable);
    inversionsTable = std::vector<Inverse>(gf.inversionsTable);
    elements = std::vector<Polynomial>(gf.elements);
}

```

```
}

void GF::Out(std::ostream& out)
{

}

int GF::GetIndexOf(const Polynomial& p) const
{
    Polynomial temp;

    temp = p.Normalise(base).Mod(irreduciblePolynomial, base);

    for (int i = 0; i < (int) elements.size(); i++)
    {
        if (temp == elements[i])
        {
            return i;
        }
    }

    return -1;
}

const Polynomial& GF::GetAddInverse(int i) const
{
    if (i != 0)
    {
        return elements[inversionsTable[i].add];
    }
    else
    {
        return Polynomial::ZERO;
    }
}

const Polynomial& GF::GetMulInverse(int i) const
{
    if (i != 0)
    {
        return elements[inversionsTable[i].mul];
    }
    else
    {
        return Polynomial::ONE;
    }
}

void GF::GenerateInversionsTable()
{
    inversionsTable = std::vector<Inverse>(((int) pow(base, power)) + 1);

    inversionsTable[0].add = 0;
    inversionsTable[0].mul = -1;

    for (int i = 1; i < (int) pow(base, power); i++)
    {
```

```

    bool addmul[2] = {true, true};

    for (int j = 1; j < (int) pow(base, power); j++)
    {
        if (inversionsTable[i].add == -1)
        {
            addmul[0] = false;

            if (Add(elements[i], elements[j]) == Polynomial::ZERO)
            {
                addmul[0] = true;
                inversionsTable[i].add = j;
                inversionsTable[j].add = i;
            }
        }

        if (inversionsTable[i].mul == -1)
        {
            addmul[1] = false;

            if (Mul(elements[i], elements[j]) == Polynomial::ONE)
            {
                addmul[1] = true;
                inversionsTable[i].mul = j;
                inversionsTable[j].mul = i;
            }
        }

        if ((addmul[0] == true) && (addmul[1] == true))
        {
            break;
        }
    }
}

void GF::FindAlpha()
{
    if ((base == 2) && (power == 1))
    {
        alpha = 1;
        return;
    }

    for (int i = 2; i < (int) pow(base, power); i++)
    {
        Polynomial temp = (*this)[i];

        for (int j = 2; j < (int) pow(base, power); j++)
        {
            temp = Mul(temp, (*this)[j]);

            if (temp == Polynomial::ONE)
            {
                if (j == ((int) pow(base, power) - 1))
                {
                    alpha = i;
                }
            }
        }
    }
}

```

```

        return;
    }
    else
    {
        break;
    }
}
}
}

void GF::GenerateAlphaTable()
{
    Polynomial temp = elements[alpha];

    alphaTable = std::vector<int>(((int) pow(base, power)) + 1);

    alphaTable[0] = -1;
    alphaTable[1] = 0;
    alphaTable[alpha] = 1;

    for (int i = 2; i < ((int) pow(base, power)) - 1; i++)
    {
        temp = temp * (*this)[alpha];

        for (int j = 2; j < (int) pow(base, power); j++)
        {
            if (j == alpha)
            {
                continue;
            }
            else if (temp == (*this)[j])
            {
                alphaTable[j] = i;
                break;
            }
        }
    }
}

std::vector<Polynomial>& GF::GenerateAllPolynomials(int base, int powerLimit)
{
    int a;
    std::vector<int> ploynomialCoefficients;
    std::vector<Polynomial>* polynomials;

    polynomials = new std::vector<Polynomial>;

    ploynomialCoefficients = std::vector<int>(powerLimit + 1, 0);

    a = 0;
    while ((*polynomials).size() != ((unsigned int) pow(base, powerLimit)))
    {
        int temp;

        temp = a;
    }
}

```

```

    for (int i = powerLimit; i >= 0; i--)
    {
        ploynomialCoefficients[i] = temp / ((int) pow(base, i));
        temp %= (int) pow(base, i);
    }

    (*polynomials).push_back(Polynomial(ploynomialCoefficients));

    a++;
}

return (*polynomials);
}

std::ostream& operator <<(std::ostream& o, const GF& gf)
{
    o << "GF(" << (int) pow(gf.GetBase(), gf.GetPower()) << ", " << gf.
GetIrreduciblePolynomial() << "):" << std::endl;

    for (int i = 0; i < gf.GetCharasteristic(); i++)
    {
        if (i == gf.alpha)
        {
            o << "GEN -->";
        }
        o << "\t" << gf[i] << std::endl;
    }

    return o;
}
}

```