BÁO CÁO ĐỒ ÁN 1

BIỂU DIỄN VÀ TÍNH TOÁN SỐ HỌC TRÊN MÁY TÍINH



Nhóm Big-O | Kiến trúc máy tính và Hợp ngữ | 27 Tháng Ba 2018

MỤC LỤC

[THÀNH VIÊN 3](#_Toc510124592)

[PHẦN SỐ NGUYÊN 3](#_Toc510124593)

[PHẦN SỐ THỰC 3](#_Toc510124594)

[KHAI BÁO 3](#_Toc510124595)

[HÀM NHẬP 4](#_Toc510124596)

[HÀM CHECKX10 6](#_Toc510124597)

[HÀM CONVERTBFD 7](#_Toc510124598)

[HÀM DIV2 8](#_Toc510124599)

[HÀM CONVERTAFD 8](#_Toc510124600)

[HÀM NORMALIZEBIN 9](#_Toc510124601)

[HÀM CONVERTTOQFLOAT 10](#_Toc510124602)

[HÀM SETBIT 12](#_Toc510124603)

[HÀM XUẤT 13](#_Toc510124604)

[HÀM GETTYPEOFQFLOAT: 15](#_Toc510124605)

[HÀM CHECKBITSCOPE 16](#_Toc510124606)

[HÀM GETBIT 16](#_Toc510124607)

[LỚP STRFLOAT 17](#_Toc510124608)

[HÀM KHỞI TẠO 18](#_Toc510124609)

[HÀM CHUYỂN XÂU SANG STRFLOAT 18](#_Toc510124610)

[HÀM ISNEGATIVE 18](#_Toc510124611)

[HÀM NORMALIZE 18](#_Toc510124612)

[HÀM THỰC HIỆN PHÉP CỘNG 19](#_Toc510124613)

[HÀM BALANCELENGTH 21](#_Toc510124614)

[HÀM THỰC HIỆN PHÉP GÁN 22](#_Toc510124615)

[HÀM THỰC HIỆN PHÉP NHÂN 22](#_Toc510124616)

[HÀM THỰC HIỆN PHÉP CHIA 23](#_Toc510124617)

[HÀM NHẬP XUẤT 24](#_Toc510124618)

[HÀM CHUYỂN ĐỔI BIỂU DIỄN SỐ 24](#_Toc510124619)

[HÀM BINTODEC 24](#_Toc510124620)

[HÀM DECTOBIN 25](#_Toc510124621)

[ĐÁNH GIÁ 25](#_Toc510124622)

THÀNH VIÊN

|  |  |  |
| --- | --- | --- |
| MSSV | Họ và tên | Công việc |
| 1612224 | Hồ Minh Huấn | Lập trình phần QInt |
| 1612904 | Mai Nguyễn Anh Vũ | Lập trình nhập, xuất, chuyển đổi trong QFloat |
| 1612098 | Lâm Cương Đạt | Lập trình các operator QFloat và Demo |

PHẦN SỐ NGUYÊN

Số nguyên lớn 128 bit **QInt** được biễu diễn bằng 1 mảng 4 số nguyên không dấu 32 bit.

Trước tiên, ta khai báo một lớp phụ trợ **StrInt** nhằm giúp cho quá trình nhập xuất **QInt** dễ dàng hơn. Lớp **StrInt** chủ yếu hỗ trợ việc chuyển đổi 1 chuỗi thành các bit của **QInt** do việc nhập xuất số nguyên lớn chỉ có thể thực hiện thông qua chuỗi.

Lớp StrInt

Ta khai báo lớp **StrInt** như sau:

class StrInt{

private:

bool m\_negative;

std::string m\_numberString;

static const unsigned int MAX\_RESERVED\_LENGTH = 100;

StrInt divTwo(size\_t length);

void normalize();

public:

StrInt();

StrInt(int number);

friend std::ostream& operator<<(std::ostream& os, const StrInt& number);

friend std::istream& operator>>(std::istream& is, StrInt& number);

unsigned short lastDigit();

bool isEven();

bool isZero();

bool isNegative();

void toNegative();

void toPositive();

StrInt Half();

StrInt Double();

StrInt& operator++();

};

std::istream& operator>>(std::istream& is, StrInt& number);

std::ostream& operator<<(std::ostream& os, const StrInt& number);

Lớp **StrInt** bao gồm các hàm:

* **lastDigit()**: lấy chữ số cuối cùng của số nguyên
* **isEven()**: kiểm tra số chẵn
* **isZero()**: kiểm tra số nguyên bằng 0
* **isNegative()**: kiểm tra số âm
* **toNegative()**: chuyển 1 số thành số âm
* **toPositive()**: chuyển 1 số thành số dương
* **Half()**: chia 2 số nguyên – có hàm hỗ trợ **divTwo()**
* **Double()**: nhân 2 số nguyên
* **normalize()**: chuẩn hóa số nguyên, như loại bỏ các số 0 thừa và chỉnh dấu cho phù hợp

Ngoài ra, lớp **StrInt** còn overload lại **operator++** để cộng thêm 1 vào số nguyên, các **operator<<** và **operator>>** để dùng cho nhập xuất.

Cụ thể các hàm trong lớp **StrInt** được cài đặt như sau:

StrInt StrInt::divTwo(size\_t length){

// Base case

unsigned short lastDigit = m\_numberString[length - 1] - '0';

if (length == 1){

return StrInt(m\_negative ? -lastDigit / 2 : lastDigit / 2);

}

// Recursive case

bool odd = (m\_numberString[length - 2] - '0') % 2 == 1;

StrInt number = divTwo(length - 1);

if (number.m\_numberString[0] == '0'){

number.m\_numberString.clear();

number.m\_negative = m\_negative;

}

if (odd){

number.m\_numberString.push\_back(lastDigit / 2 + 5 + '0');

}

else{

number.m\_numberString.push\_back(lastDigit / 2 + '0');

}

return number;

}

void StrInt::normalize(){

if (m\_numberString == "0"){

m\_negative = false;

}

}

StrInt::StrInt() : StrInt(0){}

StrInt::StrInt(int number){

m\_numberString.reserve(MAX\_RESERVED\_LENGTH);

if (number == 0){

m\_negative = false;

m\_numberString.push\_back('0');

return;

}

std::string temp;

if (m\_negative = number < 0 ? true : false){

number = -number;

}

while (number){

temp.push\_back((number % 10) + '0');

number /= 10;

}

for (int i = temp.length() - 1; i >= 0; i--)

m\_numberString.push\_back(temp[i]);

}

std::ostream& operator<<(std::ostream& os, const StrInt& number){

os << (number.m\_negative ? "-" : "") << number.m\_numberString;

return os;

}

std::istream& operator>>(std::istream& is, StrInt& number){

std::string buffer;

is >> buffer;

for (size\_t i = 1; i < buffer.length(); i++){

if (buffer[i] < '0' || buffer[i] > '9'){

return is;

}

}

if (buffer.length() < 1 || ((buffer[0] == '-' || buffer[0] == '+') && buffer.length() == 1)){

return is;

}

number.m\_negative = buffer[0] == '-';

number.m\_numberString.clear();

for (size\_t i = (buffer[0] == '-' || buffer[0] == '+') ? 1 : 0; i < buffer.length(); i++)

number.m\_numberString.push\_back(buffer[i]);

number.normalize();

return is;

}

unsigned short StrInt::lastDigit(){

return m\_numberString[m\_numberString.length() - 1] - '0';

}

bool StrInt::isEven(){

return lastDigit() % 2 == 0;

}

bool StrInt::isZero(){

return m\_numberString == "0";

}

bool StrInt::isNegative(){

return m\_negative;

}

void StrInt::toNegative(){

if (m\_numberString != "0"){

m\_negative = true;

}

}

void StrInt::toPositive(){

m\_negative = false;

}

StrInt StrInt::Half(){

return divTwo(m\_numberString.length());

}

StrInt StrInt::Double(){

StrInt doub;

doub.m\_numberString.clear();

std::string temp;

unsigned short accumulator = 0;

for (int i = m\_numberString.length() - 1; i >= 0; i--){

unsigned short digit = m\_numberString[i] - '0';

digit = digit \* 2 + accumulator;

accumulator = digit / 10;

digit %= 10;

temp.push\_back(digit + '0');

}

if (accumulator > 0){

temp.push\_back(accumulator + '0');

}

for (int i = temp.length() - 1; i >= 0; i--){

doub.m\_numberString.push\_back(temp[i]);

}

doub.m\_negative = m\_negative;

doub.normalize();

return doub;

}

StrInt& StrInt::operator++(){

std::string temp;

unsigned short accumulator = m\_negative ? -1 : 1;

for (int i = m\_numberString.length() - 1; i >= 0; i--){

char c = m\_numberString[i] + accumulator;

if (c < '0'){

c = '9';

}

else if (c > '9'){

c = '0';

}

else{

accumulator = 0;

}

temp.push\_back(c);

}

if (temp.back() == '0' && accumulator == 0){

temp.pop\_back();

}

m\_numberString.clear();

if (accumulator == 1){

m\_numberString.push\_back('1');

}

for (int i = temp.length() - 1; i >= 0; i--){

m\_numberString.push\_back(temp[i]);

}

normalize();

return \*this;

}

Lớp QInt

Sau đó, ta tiến hành khai báo lớp QInt với các hằng số cần thiết:

* **BIN\_PER\_HEX**: số bit để biểu diễn một giá trị thập lục phân từ 0 – F
* **HEX\_PER\_UINT**: số chữ số thập lục phân trong 1 số nguyên 32 bit không dấu
* **MAX\_HEX**: giá trị tối đa của 1 chữ số thập lục phân
* **MAX\_UINT**: giá trị tối đa của 1 số nguyên không dấu 32 bit
* **BINARY & HEX**: hệ cơ số nhị phân và thập lục phân
* **UINT\_NUM**: số lượng số nguyên 32 bit không dấu trong QInt
* **SIZE\_OF\_QINT**: kích thước theo bit của QInt

struct QInt{

QInt();

QInt(const QInt& q);

QInt& operator=(const QInt& q);

static const unsigned short BIN\_PER\_HEX = 4;

static const unsigned short HEX\_PER\_UINT = 8;

static const unsigned short MAX\_HEX = 15;

static const uint32\_t MAX\_UINT = 0xffffffff;

static const unsigned short BINARY = 2;

static const unsigned short HEX = 16;

static const unsigned short UINT\_NUM = 4;

static const unsigned short SIZE\_OF\_QINT = 128;

uint32\_t m\_binary[UINT\_NUM];

};

Và các hàm cơ bản cho nhập xuất và tính toán trên QInt:

bool GetBit(QInt q, unsigned short position);

void SetBit(QInt& q, unsigned short position, bool on);

void ScanQInt(QInt& q);

void PrintQInt(QInt q);

std::string DecToBin(QInt q);

QInt BinToDec(const std::string& bits);

std::string BinToHex(const std::string& bits);

std::string DecToHex(QInt q);

QInt HexToDec(const std::string& hexs);

QInt operator+(QInt q, QInt m);

QInt operator-(QInt q, QInt m);

QInt operator\*(QInt q, QInt m);

QInt operator/(QInt q, QInt m);

QInt operator&(QInt q, QInt m);

QInt operator|(QInt q, QInt m);

QInt operator^(QInt q, QInt m);

QInt operator~(QInt q);

QInt operator<<(QInt q, const int n);

QInt operator>>(QInt q, const int n);

Trong đó bao gồm các hàm lấy bit, nhập xuất, chuyển đổi qua lại giữa các hệ cơ số, các phép toán cộng trừ nhân chia, and or xor not, các phép dịch số học trái phải.

Các hàm trên được cài đặt cụ thể như sau:

Nhóm hàm khởi tạo

QInt::QInt(){

for (int i = 0; i < 4; i++){

m\_binary[i] = 0;

}

}

QInt::QInt(const QInt& x){

for (int i = 0; i < 4; i++)

m\_binary[i] = x.m\_binary[i];

}

QInt& QInt::operator=(const QInt& x){

for (int i = 0; i < 4; i++)

m\_binary[i] = x.m\_binary[i];

return \*this;

}

Operator-

Đổi dấu QInt. Thực hiện not các phần tử **uint32\_t**, sau đó cộng 1 vào và kiểm tra tràn số ở mỗi phần tử.

QInt operator-(QInt q){

QInt result;

for (int i = 0; i < 4; i++){

result.m\_binary[i] = ~q.m\_binary[i];

}

bool overflow = true;

for (int i = 0; i < 4 && overflow; i++){

overflow = result.m\_binary[i] == 0xffffffff;

result.m\_binary[i]++;

}

return result;

}

Lấy và bật/tắt bit

// Lấy bit ở vị trí position

bool GetBit(QInt q, unsigned short position){

unsigned short block = position / 32; // Block data cần chọn

unsigned short i = position % 32; // Vị trí cần lấy trong block data

return (q.m\_binary[block] & (1 << i)) != 0;

}

// Đặt bit ở vị trí position, đặt = 1 nếu on = true, nếu không đặt = 0

void SetBit(QInt& q, unsigned short position, bool on){

unsigned short block = position / 32;

unsigned short i = position % 32;

if (on){

q.m\_binary[block] |= (1 << i);

}

else{

q.m\_binary[block] &= (~(1 << i));

}

}

Nhập xuất

Nhập xuất thông qua lớp phụ trợ **StrInt**

* **Nhập:** Nhập vào **StrInt**, thực hiện chia 2 liên tục cho đến khi bằng 0, nếu chẵn thì thêm 0 vào cuối **QInt**, ngược lại thêm 1.
* **Xuất:** Bắt đầu từ **StrInt** bằng 0, đi từ trái sang phải các bit trong **QInt**, nếu bit 1 thì cộng 1 vào **StrInt**, sau đó nhân 2 **StrInt**. Cuối cùng xuất ra **StrInt**.

// Đọc vào QInt dưới dạng số thập phân từ stdin

void ScanQInt(QInt& q){

// Đọc vào StrInt

StrInt temp;

std::cin >> temp;

bool negative = temp.isNegative(); // Kiểm tra âm

unsigned short shift = 0;

// Chia 2 cho đến khi = 0

while (!temp.isZero()){

// Nếu vượt quá số bit có, thông báo số lớn

if (shift >= QInt::SIZE\_OF\_QINT){

std::cout << "Too big" << std::endl;

return;

}

// Nếu số chẵn, set bit = 0, nếu lẻ set bit = 1

unsigned short block = shift / 32;

unsigned short pos = shift % 32;

if (!temp.isEven()){

q.m\_binary[block] |= 1 << pos;

}

shift++;

temp = temp.Half();

}

// Nếu âm, đổi dấu

if (negative){

q = -q;

}

}

// In QInt ra stdout dưới dạng thập phân

void PrintQInt(QInt q){

bool negative = (q.m\_binary[3] & 1 << 31) != 0;

// Nếu âm, đổi thành dương

if (negative){

q = -q;

}

// Tạo biến StrInt chứa giá trị thập phân

StrInt result;

// Đi từ trái qua, nhân 2 tuần tự, gặp bit 1 thì cộng thêm 1

for (short i = 127; i >= 0; i--){

unsigned short block = i / 32;

unsigned short pos = i % 32;

result = result.Double();

uint32\_t isOn = q.m\_binary[block] & (1 << pos);

if (isOn != 0)

++result;

}

// Nếu âm, đổi StrInt thành âm

if (negative){

result.toNegative();

}

std::cout << result << std::endl;

}

Chuyển đổi cơ số

* Thập phân sang nhị phân: bỏ qua các bit liên tục trùng nhau ở đầu thành 1 bit duy nhất, từ đó thêm các bit còn lại.

// Chuyển thập phân sang nhị phân

std::string DecToBin(QInt q){

std::string result;

// Lấy liên tục các bit từ trái qua thêm vào chuỗi

unsigned short prevBit = GetBit(q, 127);

unsigned short curBit = GetBit(q, 126);

short pos = 126;

while (curBit == prevBit && pos >= 0){

prevBit = curBit;

pos--;

curBit = GetBit(q, pos);

}

pos++;

while (pos >= 0){

result.push\_back(GetBit(q, pos--) + '0');

}

return result;

}

* Nhị phân sang thập phân: thêm bit từ cuối chuỗi, các bit còn lại được thêm dựa vào bit đầu của chuỗi (Cách này sẽ chuyển thành các số bù 2).

// Chuyển từ nhị phân sang thập phân

QInt BinToDec(const std::string& bits){

QInt result;

short length = bits.length();

// Set bit theo chuỗi nhị phân

for (short i = 0; i < length; i++){

SetBit(result, length - 1 - i, bits[i] == '1');

}

// Set các bit đầu dựa trên bit đầu của chuỗi nhị phân

for (short i = length; i < QInt::SIZE\_OF\_QINT; i++){

SetBit(result, i, bits[0] == '1');

}

return result;

}

* Nhị phân sang thập lục phân: Đi từ cuối dãy bit lên, gom nhóm 4 bit thành 1 chữ số thập lục phân, nếu chưa đủ 4 bit thêm dựa theo bit đầu.

// Chuyển từ nhị phân sang thập lục phân

std::string BinToHex(const std::string& bits){

// Mảng thập lục phân cơ bản

char hex[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

std::string result;

std::string temp;

unsigned short pos = 0;

unsigned short total = 0;

unsigned short multiplier = 1; // Biến nhân

while (pos < bits.length()){

// Chạy từ trái qua phải, gặp bit 1 thì cộng thêm 2^multiplier vào total

// multiplier \* 2 mỗi lần chạy

total = total + (bits[bits.length() - 1 - pos] - '0') \* multiplier;

pos++;

multiplier \*= QInt::BINARY;

// Nếu đã đủ số 8 bit -> chuyển total thành thập lục phân = mảng đã tạo

if (pos % QInt::BIN\_PER\_HEX == 0){

temp.push\_back(hex[total]);

total = 0;

multiplier = 1;

}

}

// Nếu chưa đủ bit cho 1 thập lục phân, add thêm các bit = bit đầu

while (pos % QInt::BIN\_PER\_HEX != 0){

total = total + (bits[0] - '0') \* multiplier;

pos++;

multiplier \*= 2;

if (pos % QInt::BIN\_PER\_HEX == 0){

temp.push\_back(hex[total]);

}

}

for (short i = temp.length() - 1; i >= 0; i--){

result.push\_back(temp[i]);

}

return result;

}

* Thập phân sang thập lục phân: Thực hiện qua trung gian chuỗi nhị phân.

// Chuyển thập phân sang thập lục phân

std::string DecToHex(QInt q){

return BinToHex(DecToBin(q));

}

* Thập lục phân sang thập phân: Gom nhóm 8 chữ số thập lục phân thành 1 số nguyên không dấu 32 bit. Nếu thiếu, không thêm dựa trên bit đầu mà chỉ để 0 nên chỉ chuyển thành số dương.

/ Chuyển thập lục phân sang thập phân

QInt HexToDec(const std::string& hexs){

QInt result;

uint32\_t multiplier = 1; // biến nhân 16^x

for(unsigned short pos = 0; pos < hexs.length(); pos++){

// chạy từ phải qua, tính đủ số thập lục phân cho 1 block data

// rồi reset lại

char c = hexs[hexs.length() - 1 - pos];

unsigned short val = c < 'A' ? (c - '0') : (10 + c - 'A');

unsigned short block = pos / QInt::HEX\_PER\_UINT;

result.m\_binary[block] = result.m\_binary[block] + (val \* multiplier);

multiplier \*= QInt::HEX;

}

return result;

}

Cộng trừ nhân chia

* Cộng: Cộng từng phần tử với nhau, nếu có tràn số thì lần cộng kế cộng thêm 1.

// Phép cộng

QInt operator+(QInt q, QInt m){

QInt result;

bool overflow = false;

for (unsigned short i = 0; i < 4; i++){

// Cộng bình thường các block data

// Nếu có overflow từ trước, cộng thêm 1

result.m\_binary[i] = q.m\_binary[i] + m.m\_binary[i] + (overflow ? 1 : 0);

overflow = (0xffffffff - q.m\_binary[i] < m.m\_binary[i]) || (0xffffffff - q.m\_binary[i] == m.m\_binary[i] && overflow);

}

return result;

}

* Trừ: Cộng số đầu với số thứ 2 đổi dấu

// Phép trừ

QInt operator-(QInt q, QInt m){

return q + (-m);

}

* Nhân: Nhân bằng thuật toán Booth

// Phép nhân thuật toán booth

QInt operator\*(QInt q, QInt m){

QInt a;

unsigned short temp = 0;

unsigned short n = QInt::SIZE\_OF\_QINT;

while (n > 0){

unsigned short lastQ = GetBit(m, 0);

if (lastQ == 1 && temp == 0){

a = a - q;

}

else if (lastQ == 0 && temp == 1){

a = a + q;

}

unsigned short lastA = GetBit(a, 0);

a = a >> 1;

m = m >> 1;

SetBit(m, 127, lastA == 1);

temp = lastQ;

n--;

}

return m;

}

* Chia: Chia theo thuật toán số không dấu, nếu trái dấu thì đổi dấu thương

// Phép chia

// Chia không dấu

// Nếu 2 số khác dấu

// Thì đổi dấu thương

QInt operator/(QInt q, QInt m){

QInt a;

unsigned short n = QInt::SIZE\_OF\_QINT;

bool sameSign = (GetBit(q, 127) ^ GetBit(m, 127)) == 0;

if (GetBit(q, 127) == 1){

q = -q;

}

if (GetBit(m, 127) == 1){

m = -m;

}

while (n > 0){

unsigned short first = GetBit(q, 127);

a = a << 1;

q = q << 1;

SetBit(a, 0, first == 1);

a = a - m;

if (GetBit(a, 127) == 1){

a = a + m;

}

else{

SetBit(q, 0, true);

}

n--;

}

if (!sameSign){

return -q;

}

return q;

}

And Or Xor Not

Thực hiện And Or Xor Not với từng cặp phần tử tương ứng

// Phép AND

QInt operator&(QInt q, QInt m){

QInt result;

for (unsigned short i = 0; i < 4; i++){

result.m\_binary[i] = (q.m\_binary[i] & m.m\_binary[i]);

}

return result;

}

// Phép OR

QInt operator|(QInt q, QInt m){

QInt result;

for (unsigned short i = 0; i < 4; i++){

result.m\_binary[i] = (q.m\_binary[i] | m.m\_binary[i]);

}

return result;

}

// Phép XOR

QInt operator^(QInt q, QInt m){

QInt result;

for (unsigned short i = 0; i < 4; i++){

result.m\_binary[i] = (q.m\_binary[i] ^ m.m\_binary[i]);

}

return result;

}

// Phép NOT

QInt operator~(QInt q){

QInt result;

for (unsigned short i = 0; i < 4; i++){

result.m\_binary[i] = ~q.m\_binary[i];

}

return result;

}

Dịch số học trái phải

Nếu lượng bit dịch ít hơn hoặc bằng 32 bit, dịch từng phần tử, lưu lại phần bit bị mất do dịch ở mỗi phần tử, sau khi dịch phần tử kế tiếp thì thêm vào sau.

Nếu lượng bit dịch nhiều hơn 32 bit, chia thành các phần 32 bit hoặc nhỏ hơn rồi tiến hành dịch nhiều lần.

// Dịch arithmetic trái

QInt operator<<(QInt q, const int n){

unsigned short sum = 0;

while (sum < n){

unsigned short k;

if (n - sum > 32){

k = 32;

}

else{

k = n - sum;

}

sum += k;

uint32\_t current = 0;

uint32\_t next = 0;

for (unsigned short i = 0; i < 4; i++){

next = 0 | (q.m\_binary[i] >> (32 - k));

q.m\_binary[i] = (q.m\_binary[i] << k) | current;

current = next;

}

}

return q;

}

// Dịch arithmetic phải

QInt operator>>(QInt q, const int n){

unsigned short sum = 0;

bool negative = GetBit(q, 127) == 1;

while (sum < n){

unsigned short k;

if (n - sum > 32){

k = 32;

}

else{

k = n - sum;

}

sum += k;

uint32\_t current = 0;

uint32\_t next = 0;

for (short i = 3; i >= 0; i--){

next = 0 | (q.m\_binary[i] << (32 - k));

q.m\_binary[i] = (q.m\_binary[i] >> k) | current;

current = next;

}

if (negative){

q.m\_binary[3] |= 0xffffffff << (32 - k);

}

}

return q;

}

PHẦN SỐ THỰC

KHAI BÁO

Số thực được biểu diễn theo chuẩn IEEE754 với 128 bit.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **S** | **E** | | **Significand** | | |
| 1 bit | | | 15 bit | | 112 bit |

– **S**: dấu (Sign) – 0: dương, 1: âm

– **Exponent**: phần số mũ (lưu dưới dạng số biased)

– **Significand**: phần định trị

• **Ngầm định bắt đầu là** 1 + phần trị ~ (1 + 23) bits

• **Dạng chuẩn**: **+/-**1.xxx…x2×2**yyy…y**2

Chúng ta **khai báo một lớp đối tượng QFloat** có cấu trúc như sau:

class QFloat

{

int m\_el[4]; //m\_el[0]: bit 0 - 31

//m\_el[1]: bit 32 - 63

//m\_el[2]: bit 64 - 95

//m\_el[3]: bit 96 – 127

}

Trong đó: m\_el là một mảng kiểu int gồm 4 phần tử. Mỗi phần tử chịu trách nhiệm lưu 32 bit lần lượt từ trái qua phải (m\_el[0]: bit 0 → 31, m\_el[1]: bit 32 → 63, m\_el[2]: bit 64 → 95, m\_el[3]: bit 96 → 127).

HÀM NHẬP

Để nhập vào một đối tượng QFloat, ta xây dựng cho lớp QFloat một hàm nhập từ bàn phím với kiểu dữ liệu ban đầu là string và số nhập vào ở dạng cơ số 10 hoặc cơ số 2. Đối với mỗi dạng cơ số sẽ xử lý khác nhau.

class QFloat

{

void ScanQFloat(int base);//Đọc số QFloat, với base là cơ số.

}

Chúng ta đi cài đặt **hàm ScanQFloat** như sau:

void QFloat::ScanQFloat(int base)

{

//Nếu là ở dạng cơ số 10 (base = 10) thì chuyển về dạng nhị phân với MAX\_VALUE\_EXP số sau dấu phẩy. Chuẩn hóa dạng nhị phân và chuyển về QFloat.

int N\_exp = 0; //Lưu số lượng chữ số đứng trước dấu phẩy, phục vụ cho việc tính toán dời dấu phẩy về đằng trước.

string tmp; //Lưu xâu được nhập vào từ bàn phím.

cin >> tmp;

if (base == 10)

N\_exp = convertToBin(tmp); //Đổi về cơ số 2.

normalizeBin(tmp); //Chuẩn hóa ở dạng cơ số 2.

this->convertToQFloat(tmp, N\_exp); //Chuyển về dạng QFloat.

}

Nếu là ở dạng cơ số 10 (base = 10) thì chuyển về dạng nhị phân với MAX\_VALUE\_EXP số sau dấu phẩy. Sau đó chuẩn hóa dạng nhị phân và chuyển về QFloat.

Ta xét đến **hàm chuyển về cơ số 2** (**convertToBin**).

int convertToBin(string &a)

{

int hx10 = checkx10(a); //Kiểm tra xem a có dạng x10^

string bfd = ""; //Lưu số trước dấu phẩy.

string afd = "0."; //Lưu số sau dấu phẩy.

int i;

//Tách số trước dấu phẩy.

for (i = 0; i < a.length() && a[i] != '.'; i++)

bfd.push\_back(a[i]);

//Tách số sau dấu phẩy.

i++; //Nhảy qua "."

for (i = i; i < a.length(); i++)

afd.push\_back(a[i]);

afd.push\_back('0');//Tránh trường hợp "0." nên cần thêm "0" vào sau cùng (không thay đổi giá trị của số)

//Nếu a có dạng "x10^" thì chuyển theo từng trường hợp (>0 hoặc <0)

if (hx10 > 0) {

for (i = 1; i <= hx10; i++)

{

bfd.push\_back(afd[2]); //afd[2] bỏ qua "0."

afd.erase(2, 1);

afd.push\_back('0'); //Thêm 0 vào để luôn có số sau dấu phẩy.

}

}

else

{

for (i = 1; i <= hx10; i++)

{

afd.insert(2, 1, bfd[bfd.length() - 1]); //Chuyển số cuối từ bfd sang làm số đầu của afd.

bfd.erase(bfd.length() - 1, 1);

if (bfd.length() == 0 || bfd[bfd.length() - 1] == '-') bfd.push\_back('0'); //Xét trường hợp bfd rỗng hoặc bfd="-".

}

}

convertBFD(bfd); //Chuyển đổi bfd

int Nexp = bfd.length() - 1 - (bfd[0] == '-'); //Số lượng số có thể chuyển sang cho afd với điều kiện bfd chứ 1 chữ số.

convertAFD(afd, Nexp);

a = bfd + afd;

return Nexp;

}

Mở đầu hàm, chúng ta cài đặt một **hàm (checkx10())** bổ trợ cho phép kiểm tra xâu có dạng biểu diễn “chuẩn khoa học” (scientific notation). Nếu là đúng thì **hàm checkx10()** xóa phần đuôi này. Kết quả của phần mũ (x10^x) sẽ được lưu vào hx10.

Sau đó tách xâu ban đầu (a) thành 2 phần riêng biệt: trước và sau dấu phẩy vì mỗi phần có cách chuyển qua nhị phân khác nhau. Phần trước dấu phẩy được lưu vào xâu bfd, phần sau được lưu vào afd. Khi tách cần chú ý đến trường hợp xâu đã nhập không có phần sau dấu phẩy.

Chúng ta sẽ dựa vào giá trị của hx10 để chuyển vị trí của các số trong 2 xâu afd và bfd để làm cho giá trị của hx10 = 0, nghĩa là chuyển về dạng “biểu diễn thông thường”.

Gọi hàm chuyển đổi cho bfd. Rồi lưu lại số lượng chữ số mà bfd có thể chuyển cho afd sao cho bfd chỉ còn 1 số (Nexp). Gọi hàm chuyển đổi afd.

Sau khi chuyển xong thì xâu a được gán bằng tổng của bfd và afd. Trả về giá trị “mũ” của xâu a. (giá trị “mũ” được hiểu là lượng số có thể chuyển từ phần nguyên sang phần thập phân).

Chúng ta đi vào chi tiết từng hàm được gọi trong hàm converToBin.

HÀM CHECKX10

Xét **hàm checkx10**:

int checkx10(string &a)

{

int posx10; //Ví trí xuất hiện kí tự: 'x'

for (posx10 = a.length() - 1; posx10 >= 0; posx10--) //Chạy từ cuối về kiểm tra.

if (a[posx10] == 'x') break;

//Không tìm thấy thì thoát.

if (posx10 == -1) return 0;

a.erase(posx10 + 1, 3); //Xóa chuỗi "10^" từ (vị trí ‘x’ +1) đến (vị trí ‘x’ +3)

int sign = 1; //Lưu dấu của giá trị mũ

if (a[posx10 + 1] == '-') {//Kiểm tra mũ âm

sign = -1;

a.erase(posx10 + 1, 1); //Xóa mũ âm.

}

int vx10 = 0; //Lưu giá trị của mũ.

while (a.length() > posx10 + 1) //Xóa khi còn kí tự 'x'

{

vx10 = vx10 \* 10 + a[posx10 + 1] - '0'; //Tính giá trị của mũ

a.erase(posx10 + 1, 1);

}

//Xóa dấu 'x'

a.erase(a.length() - 1, 1);

//Trả về giá trị kèm với dấu.

return vx10\*sign;

}

Hàm kiểm tra dãy số nhập vào có dạng xxxxx\*10^x hay không? Trả về giá trị “mũ”.

Biến posx10 chịu trách nhiệm lưu vị trí xuất hiện kí tự ‘x’ trong xâu. Nếu posx10= -1 thì xâu a không có dạng “biểu diễn khoa học”, trả về 0. Ngược lại ta đi vào xử lý chi tiết.

Ta xóa các kí tự thừa sau ‘x’: chuỗi “10^”. Rồi kiểm tra xem mũ có âm hay không? Nếu có thì xóa kí tự ‘-‘. Ta thực hiện tính giá trị của mũ bằng cách lấy (đọc và xóa) kí tự liền sau kí tự ‘x’ cho tới khi độ dài của xâu bằng (vị trí của ‘x’+1). Ta xóa luôn kí tự ‘x’ và trả về giá trị của mũ.

HÀM CONVERTBFD

Xét **hàm ConvertBFD**:

void convertBFD(string &bfd)

{

string res = ""; //Lưu tạm giá trị phép dư số khi chia cho 2.

string tmp = bfd; //Lưu tạm giá trị của bfd.

int j = 0;

int sign = 0; //Lưu giá trị dấu

if (tmp[0] == '-') { //Kiểm tra dấu

sign = -1;

tmp.erase(0, 1); //Xóa dấu

}

//Thực hiện phép chia 2, lưu lại số dư trong từng phép chia.

while (tmp.length() != 0 && tmp[0]!='0') {

if ((tmp[tmp.length() - 1] - '0') % 2 == 1) res.push\_back('1');

else res.push\_back('0');

div2(tmp);

}

if (res.length() == 0) res.push\_back('0');

//Thêm dấu cho kết quả ở dạng nhị phân.

if (sign == -1) res.push\_back('-');

//Chuyển đổi về đúng dạng nhị phân bằng cách đảo chuỗi.

reverse(res.begin(),res.end());

//Gán lại kết quả.

bfd = res;

}

Hàm này chia giá trị của xâu cho 2 để chuyển đổi.

Trước tiên, chúng ta kiểm tra xem xâu bfd có chưa kí ‘-‘ biểu hiện cho số âm hay không. Nếu có thì xóa ‘-‘ và gán biến sign = -1 (lưu giá trị âm dương).

Thực hiện phép chia bằng hàm div2(). Trong quá trình chia thì kiểm tra số dư bằng cách kiểm tra số cuối cùng là chẵn hay lẻ. Rồi gán số dư vào sau cùng chuỗi res (lưu kết quả chuyển đổi).

Chuỗi res sẽ được thêm dấu ‘-‘ nếu là số âm ban đầu. Rồi đảo ngược chuỗi res lại cho đúng với cách biểu diễn nhị phân từ phải qua trái. Trường hợp res rỗng do bfd = “0” thì gán res = “0”.

Gán giá trị bfd = res. Vì bfd chính là biến lưu kết quả sau khi thoát ra khỏi hàm.

HÀM DIV2

Xét **hàm div2**:

void div2(string &bfd)

{

int p = 2,j=0;

string tmp = "";

for (int k = 0; k < bfd.length(); k++) //Thực hiện phép chia từ trái sang phải.

{

j = j \* 10 + bfd[k] - '0'; //Lấy số thứ k trong chuỗi.

tmp.push\_back((j / p) + '0'); //Lưu kết quả vào sau tmp.

j %= p;

}

while (tmp.length() > 1 && tmp[0] == '0') tmp.erase(0, 1);

bfd = tmp;

}

Biến p = 2 lưu số chia. j là biến lưu giá trị các số được trích ra trong xâu bfd. Duyệt xâu bfd từ trái sang phải, thực hiện phép chia tự nhiên. Số dư sẽ được lưu sau của tmp.

Khi duyệt xong xâu thì xóa những số 0 ở đầu xâu kết quả (tmp). Sau đó gán bfd = tmp.

HÀM CONVERTAFD

Xét **hàm convertAFD**:

void convertAFD(string &afd, int Nexp) //Nexp: lưu số lượng số có thể chuyển từ bfd sang cho afd.

{

int j;

string res = "."; //Lưu giá trị số sau dấu phẩy phần thập phân ở dạng cơ số 2. Độ dài tối đa là 112 bit nhưng do chuyển từ bfd sang Nexp số nên chiều dài của res= 112-Nexp

while (res.length() - 1 < LENGTH\_OF\_AFTER\_POINT\_BASE\_2 - Nexp) { //-1 vì có '.'

//Nhân afd với 2.

int s = 0; // lưu số nhớ trong phép nhân.

j = 0; //Lưu phần trích của số để nhân.

for (int i = afd.length() - 1; i >= 0; i--)

if (afd[i] != '.')

{

j = (afd[i] - '0') \* 2 + s;

if (j > 9) s = 1;

else s = 0;

j %= 10;

afd[i] = j + '0';

}

res.push\_back(afd[0]); //Thêm vào sau kết quả của phần nguyên afd sau khi nhân.

afd.erase(0, 1); //Bỏ phần nguyên của afd.

afd.insert(0, 1, '0'); //Thêm '0' vào phần nguyên của afd.

}

afd = res;

}

Hàm thực hiện chuyển về dạng nhị phân của những số sau dấu phẩy bằng cách nhân 2 rồi lấy phần nguyên. Biến j lưu giá trị số trích từ xâu afd. Biến res lưu kết quả chuyển đổi.

Ta thực hiện nhân cho 2 đến khi res chứa 112 – Nexp số. Vì ta có thể chuyển từ bfd sang cho afd Nexp số. Sau mỗi lần nhân 2, ta lấy giá trị phần nguyên ở afd (chắc chắn chỉ bao gồm ‘1’ hoặc ‘0’). Lấy (đọc và xóa) xong, chúng ta thêm ‘0’ và đầu để thực hiện phép nhân 2 cho chính xác.

Gán afd = res.

HÀM NORMALIZEBIN

Xét **hàm normalizeBin**:

void normalizeBin(string &a)

{

string bfd = "", afd = "."; //bfd: lưu số trước dấu phẩy

//afd: lưu số sau dấu phẩy

int i;

//Tách thành 2 phần: Trước dấu phẩy và sau dấu phẩy.

for (i = 0; i < a.length() && a[i] != '.'; i++) bfd.push\_back(a[i]); //Tách trước dấu phẩy.

for (i = i + 1; i < a.length(); i++) afd.push\_back(a[i]); //Tách sau dấu phẩy.

while (bfd.length() > 1) //Chuyển về dạng x.xxxxx tức là bfd có 1 kí tự duy nhất.

{

if (bfd.length() == 2 && bfd[0] == '-') break; //Dừng khi bfd="-x"

afd.insert(1, 1, bfd[bfd.length()-1]); //Thêm số vào sau dấu phẩy. (Thêm vào đầu afd)

bfd.erase(bfd.length() - 1); //Xóa số trước dấu phẩy. (Xóa cuối bfd).

}

//Làm cho số lượng số sau dấu phẩy bằng LENGTH\_OF\_AFTER\_POINT\_BASE\_2 (=112 là số bit biểu diễn giá trị)

while (afd.length() - 1 < LENGTH\_OF\_AFTER\_POINT\_BASE\_2) afd.push\_back('0'); //Thêm cho đủ

while (afd.length() - 1 > LENGTH\_OF\_AFTER\_POINT\_BASE\_2) afd.erase(afd.length() - 1, 1); //Nếu dư thì xóa

a = bfd + afd; //Nối lại để thành chuỗi hoàn chỉnh.

}

Hàm chuẩn hóa cơ số 2 thực hiện chuyển dấu phẩy qua bên trái sao cho xâu a có dạng x.xxxxx và sau dấu phẩy có LENGTH\_OF\_AFTER\_POINT\_BASE\_2 (112) số.

Trước tiên, hàm normalizeBin tách thành 2 phần bfd và afd lưu lần lượt các số trước và sau dấu phẩy trong xâu a. Sau đó thực hiện lấy (đọc và xóa) từng số ở cuối xâu bfd thêm vào đầu xâu afd cho tới khi xâu bfd có 1 chữ số hoặc ở dạng “-x”.

Sau đó thêm vào afd một số lượng số ‘0’ cho đủ 112 số sau dấu phẩy. Nếu afd có nhiều hơn 112 số thì xóa. Rồi nối lại thành xâu a.

HÀM CONVERTTOQFLOAT  
Xét **hàm convertToQFloat**:

void QFloat::convertToQFloat(string a, int N\_Exp)

{

int i;

//Dấu

this->SetBit(0, a[0] == '-'); //Nếu a[0]=='-' thì đặt bit đầu là 1, ngược lại là 0.

if (a[0] == '-') a.erase(0, 1); //Xóa kí tự đầu tiên của a nếu là '-'.

//Chia ra theo loại biểu diễn nhị phân để chuyển đổi.

if (a[0] == '1') //Dãy nhị phân a ở dạng 1.xxxx với số mũ là N\_Exp.

{

if (N\_Exp > MAX\_VALUE\_EXP) //Infinitite //Lớn hơn phạm vi mà QFloat có thể biểu diễn ở dạng nhị phân. (số mũ > MAX\_VALUE\_EXP)

{

for (i = 1; i < 16; i++) this->SetBit(i, 1); //Đặt tất cả các bit mũ =1.

for (i = 16; i < 128; i++) this->SetBit(i, 0); //Đặt tất cả các bit giá trị = 0.

}

if (N\_Exp <= MIN\_VALUE\_EXP) //NaN //Số mũ nhỏ hơn hoặc bằng MIN\_VALUE\_EXP báo số lỗi.

{

NaN:

for (i = 1; i < 16; i++) this->SetBit(i, 1); // Đặt bit mũ =1.

// Đặt bit giá trị = 0 và bit thứ 16 = 1

for (i = 17; i < 128; i++) this->SetBit(i, 0);

this->SetBit(17, 1);

}

else

{ //Normalized

Normalized:

// Chuyển số mũ về dạng Bias bằng cách cộng cho 2^14-1.

N\_Exp += MAX\_VALUE\_EXP;

for (i = 15; i > 0; i--) {

this->SetBit(i, N\_Exp % 2);

N\_Exp /= 2;

}

//Gán những bit sau dấu phẩy trong a vào QFloat.

int j = 2;

for (i = 16; i < 128; i++)

this->SetBit(i, a[j++] == '1');

}

}

else

{ //Denormalized

//Chuyển từ dạng 0.xxx về 1.xxx với số mũ > -(2^14-1) = -16382 = MIN\_VALUE\_EXP

// hoặc dạng 0.xxxx với số mũ = MIN\_VALUE\_EXP

// Nếu không thì dừng lại.

while (a[0] == '0' && N\_Exp > MIN\_VALUE\_EXP)

{

N\_Exp--;

a.erase(0, 2);

a.insert(1, 1, '.');

a.push\_back('0');

}

//Nếu ở dạng 0.xxx với số mũ MIN\_VALUE\_EXP hoặc 1.xx với số mũ > MIN\_VALUE\_EXP. Ngược lại thì nằm ngoài phạm vi biểu diễn.

if ((a[0] == '0' && N\_Exp == MIN\_VALUE\_EXP) || (a[0] == '1' && N\_Exp > MIN\_VALUE\_EXP)) goto Normalized;

else goto NaN;

}

}

Hàm convertToQFloat thực hiện chuyển dãy nhị phân a với số mũ là N\_exp về dạng QFloat.

Bước đầu tiên là kiểm tra dấu. Nếu âm thì bật bit đầu tiên của m\_el[0] lên.

Sau đó kiểm tra xem xâu a biểu diễn số như thế nào:

* Số 0: khi tất cả bit mũ, bit giá trị đều bằng 0.
* Vô cùng: khi a có dạng 1.xxxx và số mũ vượt quá MAX\_VALUE\_EXP (số mũ tối đa biểu diễn được). Số vô cùng được biểu diễn dưới dạng: tất cả bit mũ bằng 1 và tất bit giá trị bằng 0.
* Số lỗi (NaN): khi a có dạng 1.xxxx và số mũ nhỏ hơn hoặc bằng MIN\_VALUE\_EXP (số mũ nhỏ nhất có thể biểu diễn được). Số lỗi được biểu diễn dưới dạng: tất bit mũ bằng 1 và bit giá có 1 bit khác 0.
* Số dạng chuẩn: khi a có dạng 1.xxxx và số mũ trong nửa khoảng (MIN\_VALUE\_EXP,MAX\_VALUE\_EXP].
* Số dạng không chuẩn: khi a có dạng 0.xxxx và số mũ = MIN\_VALUE\_EXP.

Đối với số dạng chuẩn: Ta thực hiện chuyển số mũ về dạng Bias bằng cách cộng cho 2^14-1 (MAX\_VALUE\_EXP), gán vào từng bit mũ của N\_exp và những bit sau dấu phẩy trong a vào QFloat bằng hàm SetBit(x,y) (x là vị trí bit cần thay đổi, y = true thì bật, y = false thì tắt).

Đối với số dạng không chuẩn: Ta thực hiện chuyển từ dạng 0.xxx về 1.xxx với số mũ > MIN\_VALUE\_EXP hoặc dạng 0.xxxx với số mũ = MIN\_VALUE\_EXP. Nếu ở dạng 0.xxx với số mũ MIN\_VALUE\_EXP hoặc 1.xx với số mũ > MIN\_VALUE\_EXP, thì quay lên chuyển đổi như số dạng chuẩn. Ngược lại nằm ngoài phạm vi biểu diễn, thì chuyển thành số lỗi.

HÀM SETBIT

Xét **hàm SetBit**:

//Đặt giá trị của bit.

void QFloat::SetBit(unsigned short position, bool on) {

unsigned short block = position / 32;

unsigned short i = position % 32;

if (on) {

m\_el[block] |= (1 << i);

}

else {

m\_el[block] &= (~(1 << i));

}

}

Biến position lưu vị trí cần thay đổi trong dãy bit lưu ở m\_el. Biến block cho biết phần tử thứ mấy của m\_el đang giữ bit thứ position. Biến i cho biết vị trí của (bit thứ position) trong m\_el[block]. Vì mỗi phần tử của m\_el lưu 32 bit nên ta dễ dàng tính được block và i theo công thức:

* block=position / 32.
* i=position % 32.

HÀM XUẤT

Xét **hàm PrintQFloat**:

//In số QFloat ở dạng cơ số 10.

void QFloat::PrintQFloat()

{

//Lấy loại của QFloat.

int exp = GetTypeOfQFloat();

StrFloat value("0.0"); //Lưu giá trị của các bit sau dấu phẩy ở dạng thập phân.

//Kiểm tra loại của QFloat, trả về các giá trị:

//0: Số 0 // Number 0

//1: Dạng chuẩn // Normalized

//2: Dạng không chuẩn // Denormalized

//3: Vô cùng // Infinite

//4: Lỗi // NaN

if (exp == 0) cout << 0;

else

if (exp == 3) cout << (CheckBitScope(0, 0, 1) == true ? "-" : "") << "Infinite" << endl;

else

if (exp == 4) cout << "NaN" << endl;

else

{

int sign = CheckBitScope(0, 0, 1); //0: (+); 1:(-)

int base2 = 1; //Biến lưu giá trị tạm để tính 2^(x).

StrFloat base2after("1"); //Biến lưu giá trị tạm để tính 2^(-x)

int Vexp = 0; //Lưu giá trị của mũ

int i;

//Tính giá trị của mũ theo số bít lưu trong QFloat

for (i = 15; i > 0; i--) {

if (GetBit(i)) Vexp += base2;

base2 \*= 2;

}

//Chuyển lại theo số Bias bằng cách trừ đi lượng MAX\_VALUE\_EXP

Vexp -= (MAX\_VALUE\_EXP);

//Chuyển đối giá trị sau dấu phẩy về dạng thập phân với 35 số sau dấu phẩy.

for (i = 16; i < 128; i++) {

base2after /= 2; //Tính 2^(-x) x=i-15

if (GetBit(i))

value += base2after; //Nếu giá trị của bit = 1 thì cộng dồn vào giá trị sau dấu phẩy.

}

if (exp == 1) value++;

//In giá trị

if (sign) cout << '-'; //Kiểm tra số âm.

if (Vexp > 0) for (i = 0; i < Vexp; i++) value = value \* 2;

else

if (Vexp < 0) for (i = 0; i < -Vexp; i++) value = value / 2;

cout << value << endl;

}

}

Hàm PrintQFloat in ra số được lưu trữ trong QFloat ở dạng cơ số 10.

Đầu tiên, biến exp sẽ lưu loại của QFloat bằng cách nhận giá trị trả ra của hàm GetTypeOfQFloat(), làm tiền đề thực hiện các bước xử lý sau. Ta tạo một kiểu dữ liệu mới là StrFloat để lưu số thập phân ở dạng xâu sau khi chuyển đổi. Biến value có nhiệm vụ cụ thể hóa kiểu dữ liệu mới.

Nếu biết exp có giá trị:

* 0: là số 0. Ta chỉ việc in ra số 0 rồi thoát khỏi hàm.
* 3: là số vô cùng. Ta in ra “Infinite”.
* 4: là số lỗi. Ta in ra “NaN” – not a number.
* Ngoài ra, ta xét dấu được lưu ở bit đầu, chuyển đổi phần mũ (đang được lưu ở dạng nhị phân số bias) và chuyển đổi phần trị lưu trong QFloat (từ bit 16 đến bit 127) về dạng thập phân thông qua kiểu StrFloat và các hàm được cài trong StrFloat để thực hiện các phép tính cộng, chia trên xâu. Nếu exp=1 thì thực hiện cộng thêm 1 vào value. Rồi xuất.

HÀM GETTYPEOFQFLOAT:

Xét **hàm GetTypeOfQFloat**:

//Kiểm tra loại của QFloat, trả về các giá trị:

//0: Số 0 // Number 0

//1: Dạng chuẩn // Normalized

//2: Dạng không chuẩn // Denormalized

//3: Vô cùng // Infinite

//4: Lỗi // NaN

int QFloat::GetTypeOfQFloat() //

{

int exp = 1;

if (CheckBitScope(1, 15, 1)) //Kiểm tra bit dấu (bit từ 1 đến 15) có bằng 1. -> đúng thì kiểm tra loại Infinite hoặc NaN.

{

if (CheckBitScope(16, 127, 0)) exp = 3; // "Infinite";

else exp = 4; // "NaN";

}

else

if (CheckBitScope(1, 15, 0)) //Kiểm tra bit dấu (bit từ 1 đến 15) có bằng 0. -> Đúng thì kiểm tra số 0 hoặc dạng không chuẩn.

{

if (CheckBitScope(16, 127, 0)) exp = 0; //Kiểm tra bit giá trị (bit từ 16 đến 127) có bằng 0. -> Đúng là số 0, sai là dạng không chuẩn.

else exp = 2;

}

return exp;

}

Hàm GetTypeOfQFloat sẽ trả về các giá trị tương ứng với loại của QFloat như sau:

* 0: là số 0.
* 1: dạng chuẩn.
* 2: dạng không chuẩn.
* 3: vô cùng (Infinite)
* 4: số lỗi (NaN).

Bằng cách kiểm tra dãy bit mũ và bit giá trị được lưu trong m\_el của QFloat thông qua hàm CheckBitScope(). Mỗi loại số điều được nói rõ ràng về cách lưu trong hàm convertToQFloat() bên trên.

HÀM CHECKBITSCOPE

Xét **hàm CheckBitScope**:

//Kiểm tra dãy bit từ l đến r sao cho mỗi bit bằng sd (standard).

bool QFloat::CheckBitScope(int l, int r, int sd)

{

for (int i = l; i <= r; i++)

if (GetBit(i) != sd) return false;

return true;

}

Hàm này kiểm tra dãy bit bắt đầu từ l đến r được lưu trong m\_el của QFloat. Nếu tất cả bit đều bằng giá trị sd thì trả về true, ngược lại trả về false.

Ta đọc từng giá trị của bit thông qua hàm GetBit.

HÀM GETBIT

Xét **hàm GetBit**:

//Lấy giá trị của bit.

bool QFloat::GetBit(unsigned short position) {

unsigned short block = position / 32; //Tính xem vị trí position nằm ở ô nào trong m\_el.

unsigned short i = position % 32; //Tính bit thứ mấy trong block.

// Giải thích: Vì mỗi m\_el[i] lưu 32 bit và có tất cả 128 bit nên ta dùng hai phép trên để xác định vị trí tương đối của bit cần lấy trong m\_el

return (m\_el[block] & (1 << i)) != 0;

}

Giống như đã giải thích trên hàm SetBit. Ở đây ta chỉ thay phép gán lại bit bằng phép đọc bit và trả về kết quả.

LỚP STRFLOAT

Ta xây dựng một lớp đối tượng mới đáp ứng nhu cầu lưu trữ và tính toán số thập phân trong xâu. Đó chính là lớp StrFloat.

const int AFTER\_DOT = 35; //Số chữ số tối đa sau dấu phẩy ở cơ số 10: 2^-112 ~ 10^-35

class StrFloat //Lưu số Float ở cơ số 10.

{

string m\_float; //Lưu dãy số.

bool m\_negative; //Lưu đấu. m\_negative=true (âm), =false (dương)

public:

StrFloat(); //Khởi tạo giá trị ban đầu.

StrFloat(string p); //Chuyển một chuỗi vào số.

~StrFloat();

bool isNegative(); //Kiểm tra số âm.

void normalize(); //Chuẩn hóa: loại bỏ "-" trong chuỗi, số lượng số sau dấu phẩy = AFTER\_DOT, xóa số 0 ở đầu, kiểm tra trường hợp "-0".

StrFloat operator+(const StrFloat &p);

StrFloat& operator=(StrFloat &p);

StrFloat& operator/(int p);

StrFloat& operator\*(int p);

StrFloat operator/=(int p);

StrFloat operator+=(const StrFloat& p);

StrFloat operator++(int x);

friend istream& operator>>(istream& is, StrFloat& p);

friend ostream& operator<<(ostream& os, const StrFloat &p);

};

Lớp StrFloat có 2 thuộc tính:

* m\_float: có kiểu string để lưu dãy số ở dạng thập phân.
* m\_negative: có kiểu bool để lưu dấu. True là âm, false là dương.

Lớp StrFloat được trang bị các hàm:

* StrFloat(): Khởi tạo giá trị ban đầu của các thuộc tính.
* StrFloat(string p): chuyển từ xâu p sang StrFloat.
* isNegative(): Hàm kiểm tra số có âm hay không.
* normalize(): Hàm chuẩn hóa: loại bỏ ‘-‘ trong xâu, số lượng số sau dấu phẩy = AFTER\_DOT, xóa số 0 ở đầu, kiểm tra trường hợp "-0".
* Các hàm quá tải toán tử: +, \*, /, >>, <<.

Ta đi vào chi tiết từng hàm.

HÀM KHỞI TẠO

StrFloat::StrFloat()

{

m\_float = "";

m\_negative = false;

}

HÀM CHUYỂN XÂU SANG STRFLOAT

StrFloat::StrFloat(string p)

{

m\_float = p;

normalize(); //Chuẩn hóa m\_float.

}

Sau khi gán xâu p vào m\_float thì ta đi chuẩn hóa m\_float.

HÀM ISNEGATIVE

bool StrFloat::isNegative()

{

return (m\_negative);

}

HÀM NORMALIZE

Xét **hàm normalize**:

void StrFloat::normalize()

{

//Kiểm tra số âm.

if (m\_float[0] == '-') {

m\_negative = true;

m\_float.erase(0, 1); //Loại bỏ dấu.

}

else m\_negative = false;

//Giữ số lượng số sau dấu phẩy bằng AFTER\_DOT = 35.

int afd = 0; //Lưu số lượng số sau dấu phẩy.

//Đếm số lượng số sau dấu phẩy -> afd.

int i = m\_float.length() - 1;

while (i > -1 && m\_float[i] != '.') {

i--;

afd++;

}

if (i == -1) { //Nếu như i==-1 nghĩa là số không tồn tại số sau dấu phẩy. Thêm số '0' để đủ AFTER\_DOT số sau dấu phẩy.

m\_float.push\_back('.'); //Thêm '.'

for (int j = 1; j <= AFTER\_DOT; j++) m\_float.push\_back('0'); //Thêm 0.

}

else

if (afd < AFTER\_DOT) //Thêm '0'.

for (int j = afd + 1; j <= AFTER\_DOT; j++) m\_float.push\_back('0');

else

while (afd > AFTER\_DOT) { //Xóa bớt số để cho đủ AFTER\_ DOT.

afd--;

m\_float.erase(m\_float.length() - 1, 1);

}

//Xóa số 0 ở đâu cho đến khi gặp "0.0...."

while (m\_float[0] == '0' && m\_float[1] != '.') m\_float.erase(0, 1);

//Kiểm tra trường hợp "-0".

for (i = 0; i < m\_float.length(); i++)

if (m\_float[i] != '0' && m\_float[i] != '.') return;

//Nếu là trường hợp "-0" thì bỏ dấu.

m\_negative = false;

}

Hàm normalize thực hiện chuẩn hóa số như sau:

* Nếu số âm thì loại bỏ dấu và gán m\_negative=true.
* Kiểm tra số lượng số sau dấu phẩy và đưa số lượng đó bằng AFTER\_DOT bằng cách thêm ‘0’ hoặc xóa bớt.
* Loại bỏ số "0" ở đầu.
* Kiểm tra trường hợp “-0”.

HÀM THỰC HIỆN PHÉP CỘNG

Ta quá tải các toán tử cộng: +, += và ++ (hậu tố).

StrFloat StrFloat::operator+(const StrFloat & p)

{

StrFloat tmp; //Lưu kết quả trả về.

string a = m\_float, b = p.m\_float;

//Cân bằng số lượng số giữa a và b ở 2 phần: trước và sau dấu phẩy.

balanceLength(a, b);

//Thực hiện phép cộng

int i, j = 0;

for (i = a.length() - 1; i >= 0; i--)

if (a[i] != '.') //Nếu kí tự khác "." thì thực hiện cộng.

{

int sp = a[i] + b[i] - '0' - '0' + j;

if (sp > 9) j = 1;

else j = 0;

sp %= 10;

tmp.m\_float.push\_back(sp + '0');

}

else tmp.m\_float.push\_back('.');

while (j > 0)

{

tmp.m\_float.push\_back(j % 10 + '0');

j = j / 10;

}

reverse(tmp.m\_float.begin(), tmp.m\_float.end());//Đảo ngược chuỗi cho đúng thứ tự.

tmp.normalize(); //Chuẩn hóa kết quả.

return tmp;

}

Biến tmp lưu kết quả tính. Ta dùng 2 biến tạm a và b để lưu dãy số trong 2 m\_float của 2 toán hạng.

Trước khi thực hiện phép cộng, ta cần phải cân bằng số chữ số trước và sau dấu phẩy ở cả a và b thông qua hàm balanceLength().

Ta thực hiện phép cộng bình thường. Ta đảo ngược xâu được lưu trong m\_float của tmp. Chuẩn hóa tmp rồi trả về kết quả.

Hàm này chỉ tính với điều kiện 2 toán hạng không âm.

Quả tải toán tử: +=

StrFloat StrFloat::operator+=(const StrFloat & p)

{

(\*this) = (\*this) + p;

return (\*this);

}

Quá tải toán tử: ++

StrFloat StrFloat::operator++(int x)

{

StrFloat p("1.0");

p.normalize();

(\*this) += p;

return (\*this);

}

Đối với toán tử ++, ta cộng thêm vào phần nguyên 1 đơn vị.

HÀM BALANCELENGTH

Xét **hàm balanceLength**:

void balanceLength(string &a, string &b)

{

int i;

//Tínhh số chữ số trước dấu phẩy của a và b.

int bfa = 0, bfb = 0;

for (i = 0; i < a.length() && a[i] != '.'; i++)

bfa++;

for (i = 0; i < b.length() && b[i] != '.'; i++)

bfb++;

//Thêm "0" vào đầu của a và b sao cho số lượng số bằng nhau.

string tmp = "0";

for (bfa = bfa; bfa < bfb; bfa++) a.insert(0, tmp);

for (bfb = bfb; bfb < bfa; bfb++) b.insert(0, tmp);

//Tínhh số chữ số sau dấu phẩy của a và b.

bfa = bfb = 0;

for (i = a.length() - 1; i >= 0 && a[i] != '.'; i--)

bfa++;

for (i = b.length() - 1; i >= 0 && b[i] != '.'; i--)

bfb++;

//Thêm "0" vào đầu của a và b sao cho số lượng số bằng nhau. Trường hợp nếu a và b không có số sau dấu phẩy thì bfa=bfb -> vẫn đúng. //Cần <=AFTER\_DOT?

for (bfa = bfa; bfa < bfb && bfa <= AFTER\_DOT; bfa++) a.push\_back('0');

for (bfb = bfb; bfb < bfa && bfb <= AFTER\_DOT; bfb++) b.push\_back('0');

}

Đầu tiên ta tính số lượng số trước dấu phẩy ở 2 xâu a và b. Rồi sao đó chèn ‘0’ vào đầu a hoặc b sao cho số lượng 2 số nằm bằng max(bfa, bfb). 2 biến bfa và bfb lần lượt lưu số lượng số trước dấu phẩy của a và b.

Sau đó ta thực hiện như vậy cho phần số sau dấu phẩy.

HÀM THỰC HIỆN PHÉP GÁN

StrFloat & StrFloat::operator=(StrFloat & p)

{

m\_float = p.m\_float;

m\_negative = p.m\_negative;

return (\*this);

}

Ta gán từng thuộc tính.

HÀM THỰC HIỆN PHÉP NHÂN

StrFloat & StrFloat::operator\*(int p)

{

//Kiểm tra số nhân là âm thì đổi dấu của kết quả.

if (p < 0) {

p = -p;

m\_negative = !m\_negative;

}

StrFloat res; //Lưu kết quả nhân

int j = 0, s = 0;

for (int i = m\_float.length() - 1; i >= 0; i--)

if (m\_float[i] != '.')

{

j = (m\_float[i] - '0')\*p + s;

s = j / 10;

res.m\_float.push\_back(j % 10 + '0');

}

else res.m\_float.push\_back('.');

//Nếu số dư còn thì thêm vào kết quả.

while (s > 0) {

res.m\_float.push\_back(s % 10 + '0');

s = s / 10;

}

reverse(res.m\_float.begin(), res.m\_float.end());//Đảo ngược xâu kết quả để cho kết quả đúng.

res.normalize(); //Chuẩn hóa kết quả.

(\*this) = res;

return (\*this);

}

Ta kiểm tra xem số nhân vào âm hay dương để thay đổi m\_negative. Sau đó thực hiện phép nhân. Kết quả được lưu tạm vào res. Rồi thực hiện đảo xâu kết quả và chuẩn hóa trước khi gán lại cho (\*this).

HÀM THỰC HIỆN PHÉP CHIA

StrFloat & StrFloat::operator/(int p)

{

if (p == 0) return (\*this);

if (p < 0) { //Kiểm tra số chia có âm hay không. Nếu có thì thay đổi dấu bị chia.

m\_negative = !m\_negative;

p = -p;

}

bool haveDot = false; //Kiểm tra xem có gặp dấu "." chưa.

string tmp = ""; //Lưu kết quả của phép chia.

int j = 0, afd = 0; //afd: số lượng số sau dấu phẩy.

for (int k = 0; k < m\_float.length(); k++) //Thực hiện phép chia từ trái sang phải.

if (m\_float[k] == '.') //Nếu gặp dấu "."

{

haveDot = true;

if (tmp == "") tmp.push\_back('0'); //Trường hợp phần nguyên không chia được cho số chia thì thêm "0" vào trước dấu ".".

tmp.push\_back('.');

}

else

{

j = j \* 10 + m\_float[k] - '0'; //Lấy số thứ k trong chuỗi.

tmp.push\_back((j / p) + '0'); //Thêm kết quả vào sau chuỗi tmp.

j %= p;

afd += haveDot;

}

//Nếu số bị chia nguyên và không chia được cho số chia thì thêm "0.". Nếu còn thừa số dư và vẫn chưa có phần thập phân thì thêm "."

if (j > 0 && !haveDot)

{

if (tmp.length() == 0) tmp.push\_back('0'); //Kiểm tra trường hợp số nguyên < số chia.

tmp.push\_back('.');

}

//Chia cho tới khi phần thập phân có đủ AFTER\_DOT số hoặc số dư = 0.

while (j > 0 && afd < AFTER\_DOT) {

j \*= 10;

tmp.push\_back(j / p + '0');

j %= p;

afd++;

}

//Gán kết quả

m\_float = tmp;

this->normalize();

return (\*this);

}

Hàm thực hiện phép chia số thập phân có dấu phẩy.

Trước tiên, ta kiểm tra số chia có âm hay không để thay đổi dấu của số chia. Biến tmp lưu thương. Thương của phép chia chứa tối đa 35 số ở phần thập phân. Biến haveDot cho biết là thương có dấu phẩy hay chưa.

Ta thực hiện phép chia theo tự nhiên tới khi kết thúc số. Nếu vẫn còn số dư thì chia tiếp cho tới khi có đủ 35 số ở phần thập phân hoặc số dư bằng 0.   
Biến haveDot giúp ta xác định khi nào cần thêm dấu phẩy vào thương.

Sau khi kết thúc phép chia thì gán lại vào m\_float và chuẩn hóa kết quả.

StrFloat StrFloat::operator/=(int p)

{

(\*this) = (\*this) / p;

return (\*this);

}

HÀM NHẬP XUẤT

Quá tải toán tử nhập: >>

istream & operator>>(istream & is, StrFloat & p)

{

is >> p.m\_float;

p.normalize();

return is;

}

Quá tải toán tử xuất: <<

ostream & operator<<(ostream & os, const StrFloat & p)

{

os << p.m\_float;

return os;

}

HÀM CHUYỂN ĐỔI BIỂU DIỄN SỐ

HÀM BINTODEC

Xét **hàm BinToDec**:

// m\_el[0]: 0-31; m\_el[1]:32-63; m\_el[2]:64-95; m\_el[3]:96-127

QFloat QFloat::BinToDec(bool \* bit)

{

//Chuyển từ dãy bit -> QFloat p.

QFloat p;

for (int i = 0; i < 128; i++)

p.SetBit(i, bit[i]);

return p;

}

Ta chuyển từ dãy nhị phân được lưu trong mảng bit có kiểu bool vào QFloat như sau: m\_el[0]: bit thứ 0-31; m\_el[1]: bit thứ 32-63; m\_el[2]: bit thứ 64-95; m\_el[3]: bit thứ 96-127

HÀM DECTOBIN

Xét **hàm DecToBin**:

// 1-128: el[0] -> el[3]

bool \* QFloat::DecToBin()

{

//Chuyển từ el[i] vào mảng p lưu giá trị nhị phân. 0: false; 1: true.

bool \*p;

p = new bool[128];

for (int i = 0; i < 128; i++)

p[i] = this->GetBit(i);

return p;

}

Ta chuyển dãy nhị phân được biểu diễn trong các phần tử của m\_el ra một mảng kiểu bool. Theo thứ tự sau: m\_el[0]: bit thứ 0-31; m\_el[1]: bit thứ 32-63; m\_el[2]: bit thứ 64-95; m\_el[3]: bit thứ 96-127.

ĐÁNH GIÁ