#include <iostream>

#include <string>

#include <fstream>

#include <cstdlib>

#include <map>

#include <vector>

#include <bitset>

using namespace std;

struct Node

{

Node();

unsigned char ch;

int times;

Node\* left;

Node\* right;

};

Node::Node()

{

ch=0;

times=0;

left=NULL;

right=NULL;

}

void scanFile(fstream& input\_file,map<unsigned char,int>& content)

{

char ch;

unsigned char key;

while(input\_file.get(ch)){

key = reinterpret\_cast<unsigned char&>(ch);

if(content.find(key)==content.end())

content[key]=1;

else

content[key]++;

}

}

Node\* minNode(vector<Node>& save\_node,int& pos,vector<Node>& help\_save\_node,int& help\_pos,int& length)

{

//cout<<"pos:"<<pos<<' '<<"help\_pos:"<<help\_pos<<endl;

if(pos==length)

{

help\_pos++;

return &help\_save\_node[help\_pos-1];

}

int next=help\_save\_node[help\_pos].times;

if(next==0)

{

pos++;

return &save\_node[pos-1];

}

else

{

if(save\_node[pos].times<=help\_save\_node[help\_pos].times)

{

pos++;

return &save\_node[pos-1];

}

else

{

help\_pos++;

return &help\_save\_node[help\_pos-1];

}

}

}

Node\* huffmanTree(vector<Node>& save\_node,vector<Node>& help\_save\_node,int& length)

{

int pos=0;

int help\_pos=0;

for(int i=length;i<2\*length-1;i++)

{

Node\* minNode1=minNode(save\_node,pos,help\_save\_node,help\_pos,length);

Node\* minNode2=minNode(save\_node,pos,help\_save\_node,help\_pos,length);

save\_node[i].left=minNode1;

save\_node[i].right=minNode2;

save\_node[i].ch=0;

save\_node[i].times=minNode1->times+minNode2->times;

//cout<<"min1"<<' '<<minNode1->times<<' '<<minNode1->ch<<endl;

//cout<<"min2"<<' '<<minNode2->times<<' '<<minNode2->ch<<endl;

cout<<"node->ch"<<' '<<&save\_node[i]<<' '<<"ASCII:"<<int(save\_node[i].ch)<<' '<<"times:"<<save\_node[i].times<<' '<<i<<endl;

cout<<"node->left"<<' '<<save\_node[i].left<<' '<<"ASCII:"<<int(save\_node[i].left->ch)<<' '<<"times:"<<save\_node[i].left->times<<endl;

cout<<"node->right"<<' '<<save\_node[i].right<<' '<<"ASCII:"<<int(save\_node[i].right->ch)<<' '<<"times:"<<save\_node[i].right->times<<endl<<endl;

help\_save\_node[i-length]=save\_node[i];

}

Node \*root=&save\_node[2\*length-2];

return root;

}

void compressTree(fstream& output\_file,Node\* node,map<unsigned char,string>& chToBits,string bits,string& tree,string& judge)

{

cout<<"node"<<' '<<node<<' '<<"ASCII:"<<int(node->ch)<<' '<<node->times<<endl;

if(node->left==NULL)

{

//cout<<"leaf"<<endl;

chToBits[node->ch]=bits;

tree+=node->ch;

judge+="l";

int size=bits.size();

bits=bits.substr(0,size-1);

return;

}

else

{

tree+=node->ch;

judge+="n";

bits+="0";

//cout<<"node->left"<<' '<<node->left<<' '<<node->left->ch<<' '<<node->left->times<<endl;

compressTree(output\_file,node->left,chToBits,bits,tree,judge);

int size=bits.size();

bits=bits.substr(0,size-1);

bits+="1";

//cout<<"node->right"<<' '<<node->right<<' '<<node->right->ch<<' '<<node->right->times<<endl;

compressTree(output\_file,node->right,chToBits,bits,tree,judge);

size=bits.size();

if(size>0)

{

bits=bits.substr(0,size-1);

}

return;

}

}

void compress(string inputFilename,string outputFilename)

{

fstream input\_file(inputFilename, ios::in | ios::binary);

fstream output\_file(outputFilename, ios::out | ios::binary);

input\_file.seekg(0, ios::end);

if (input\_file.tellg() == 0) {

input\_file.close();

output\_file.close();

return;

}

input\_file.seekg(0, ios::beg);

map<unsigned char,int>content;

scanFile(input\_file,content);

int length=content.size();

cout << length << endl;

vector<Node> save\_node(2\*length-1);

vector<Node> help\_save\_node(length-1);

int pos=0;

int size=0;

for (map<unsigned char, int>::iterator it = content.begin(); it != content.end(); it++) {

save\_node[pos].ch = (\*it).first;

save\_node[pos].times = (\*it).second;

save\_node[pos].left = NULL;

save\_node[pos].right = NULL;

size += save\_node[pos].times;

cout<<"save\_node"<<' '<<&save\_node[pos]<<' '<<"ASCII:"<<int(save\_node[pos].ch)<<' '<<"times:"<<save\_node[pos].times<<"pos"<<pos<<endl;

pos++;

}

//create a huffman tree

Node\* root= huffmanTree(save\_node,help\_save\_node,length);

for (vector<Node>::iterator it = help\_save\_node.begin(); it != help\_save\_node.end(); it++) {

cout<<"help\_save\_node"<<' '<<"ASCII:"<<int(it->ch)<<' '<<"times:"<<(it->times)<<endl;

}

cout<<endl;

//write head

map<unsigned char,string>chToBits;

string bits="";

string tree="";

string judge="";

compressTree(output\_file,root,chToBits,bits,tree,judge);

for (map<unsigned char,string>::iterator it = chToBits.begin(); it != chToBits.end(); it++) {

cout<<(\*it).first<<':'<<(\*it).second<<endl;

}

length=tree.size();

output\_file.write(reinterpret\_cast<char\*>(&size), sizeof(int));

output\_file.write(reinterpret\_cast<char\*>(&length), sizeof(int));

output\_file.write(tree.c\_str(), length);

output\_file.write(judge.c\_str(), length);

//write body

char temp;

unsigned char key;

bits="";

string now;

input\_file.close();

input\_file.open(inputFilename.c\_str(), ios::in | ios::binary);

cout<<"READY"<<endl;

while(input\_file.get(temp))

{

key = reinterpret\_cast<unsigned char&>(temp);

bits+=chToBits[key];

while(bits.length()>=8)

{

bitset<8>now(bits.substr(0,8));

if(bits.length()==8)

bits="";

else

bits=bits.substr(8);

unsigned char ch=(unsigned char)(now.to\_ulong());

output\_file.write(reinterpret\_cast<char\*>(&ch),sizeof(char));

}

}

if(bits!="")

{

bits+="00000000";

bitset<8>now(bits.substr(0,8));

unsigned char ch=(unsigned char)(now.to\_ulong());

output\_file.write(reinterpret\_cast<char\*>(&ch),sizeof(char));

}

input\_file.close();

output\_file.close();

}

void decompressTree(string tree,string judge,Node\* node,vector<Node>& save\_tree,int& vec\_pos)

{

int length=tree.size();

//cout<<"decompressTree "<<length<<endl;

unsigned char ch=tree[0];

save\_tree[vec\_pos].ch=ch;

int vec\_now=vec\_pos;

//cout<<"vec\_now "<<vec\_now<<endl;

vec\_pos++;

//cout<<"node "<<node<<endl;

if(length==1)

{

return;

}

length-=1;

//cout<<"length "<<length<<endl;

tree=tree.substr(1);

judge=judge.substr(1);

int pos=0;

int not\_leaf=0;

int leaf=0;

bool find=false;

bool contin=true;

string leftTree,rightTree,leftJudge,rightJudge;

while(pos<length&&!find)

{

unsigned char now=tree[pos];

//cout<<"now:"<<int(now)<<endl;

//cout<<"judge:"<<judge<<endl;

//cout<<"bool:"<<(now==(unsigned char)(0))<<(judge[pos]=='n')<<endl;

if(tree.size()==2)

{

leftTree=tree[0];

rightTree=tree[1];

break;

}

if(now==(unsigned char)(0)&&judge[pos]=='n')

{

not\_leaf+=1;

}

else

{

leaf+=1;

}

pos++;

//cout<<"contin"<<pos<<' '<<tree.size()<<endl;

//cout<<"not\_leaf:"<<not\_leaf<<"leaf:"<<leaf<<endl;

if(pos==tree.size())

{

contin=false;

}

if(leaf==(not\_leaf+1))

{

leftTree=tree.substr(0,pos);

if(contin)

rightTree=tree.substr(pos);

leftJudge=judge.substr(0,pos);

if(contin)

rightJudge=judge.substr(pos);

//cout<<"leftTree "<<leftTree<<"rightTree "<<rightTree<<endl;

if(pos==1)

{

find=(rightJudge[rightJudge.size()-2]=='l')&&(rightJudge[rightJudge.size()-1]=='l');

//cout<<"bool "<<(rightJudge[rightJudge.size()-2]=='l')<<(rightJudge[rightJudge.size()-1]=='l')<<endl;

}

else if(pos==tree.size()-1)

{

find=(leftJudge[leftJudge.size()-2]=='l')&&(leftJudge[leftJudge.size()-1]=='l');

//cout<<"bool "<<(leftJudge[leftJudge.size()-2]=='l')<<(leftJudge[leftJudge.size()-1]=='l');

}

else

{

find=(leftJudge[leftJudge.size()-2]=='l')&&(leftJudge[leftJudge.size()-1]=='l')

&&(rightJudge[rightJudge.size()-2]=='l')&&(rightJudge[rightJudge.size()-1]=='l');

//cout<<"bool "<<(leftJudge[leftJudge.size()-2]=='l')<<(leftJudge[leftJudge.size()-1]=='l')

// <<(rightJudge[rightJudge.size()-2]=='l')<<(rightJudge[rightJudge.size()-1]=='l');

}

}

}

//cout<<"not\_leaf:"<<not\_leaf<<"leaf:"<<leaf<<endl;

//cout<<endl<<vec\_now<<" BEFORE LEFT"<<save\_tree[vec\_now].left<<' '<<save\_tree[vec\_now].right<<endl;

save\_tree[vec\_now].left=&(save\_tree[vec\_pos]);

//cout<<endl<<"LEFT "<<save\_tree[vec\_now].left<<endl;

node->left=save\_tree[vec\_now].left;

//cout<<"left "<<&(save\_tree[vec\_pos])<<endl;

decompressTree(leftTree,leftJudge,node->left,save\_tree,vec\_pos);

if(contin)

{

//cout<<endl<<vec\_now<<" BEFORE RIGHT"<<save\_tree[vec\_now].left<<' '<<save\_tree[vec\_now].right<<endl;

save\_tree[vec\_now].right=&(save\_tree[vec\_pos]);

//cout<<endl<<"RIGHT "<<save\_tree[vec\_now].right<<endl;

node->right=save\_tree[vec\_now].right;

//cout<<"right "<<&(save\_tree[vec\_pos])<<endl;

decompressTree(rightTree,rightJudge,node->right,save\_tree,vec\_pos);

}

}

void preorder(const vector<Node>& save\_tree,Node\* node,map<string,unsigned char>& bitsToCh,string& bits)

{

//cout<<"ok"<<endl;

if(node->left == NULL)

{

bitsToCh[bits]=node->ch;

int length=bits.size();

bits=bits.substr(0,length-1);

return;

}

else

{

bits+="0";

preorder(save\_tree,node->left,bitsToCh,bits);

bits+="1";

preorder(save\_tree,node->right,bitsToCh,bits);

int length=bits.size();

if(length>0)

bits=bits.substr(0,length-1);

}

}

void decompress(string inputFilename,string outputFilename)

{

fstream input\_file(inputFilename, ios::in | ios::binary);

fstream output\_file(outputFilename, ios::out | ios::binary);

//read head

input\_file.seekg(0, ios::end);

if (input\_file.tellg() == 0) {

input\_file.close();

output\_file.close();

return;

}

input\_file.seekg(0, ios::beg);

map<unsigned char,int>content;

int size,length;

string tree,judge;

int pos=0;

char ch;

unsigned char key;

input\_file.read(reinterpret\_cast<char\*>(&size),sizeof(int));

input\_file.read(reinterpret\_cast<char\*>(&length), sizeof(int));

while(pos<length)

{

input\_file.get(ch);

cout<<ch;

key = reinterpret\_cast<unsigned char&>(ch);

tree+=key;

pos++;

}

pos=0;

while(pos<length)

{

input\_file.get(ch);

//cout<<ch;

key = reinterpret\_cast<unsigned char&>(ch);

judge+=key;

pos++;

}

Node root;

vector<Node>save\_tree(length);

pos=0;

decompressTree(tree,judge,&root,save\_tree,pos);

map<string,unsigned char>bitsToCh;

string bits="";

preorder(save\_tree,&(save\_tree[0]),bitsToCh,bits);

//write

int i=0;

//for (map<string,unsigned char>::iterator it = bitsToCh.begin(); it != bitsToCh.end(); it++) {

// cout<<(\*it).first<<':'<<(\*it).second<<endl;

// cout<<i<<endl;

// i++;

//}

string now="";

string buffer="";

pos=0;

cout<<"START"<<endl;

while(true)

{

//cout<<buffer<<endl;

if(buffer.size()==0)

{

unsigned char ch;

input\_file.read(reinterpret\_cast<char\*>(&ch),sizeof(char));

bitset<8>get(ch);

buffer+=get.to\_string();

}

if(bitsToCh.find(now)!=bitsToCh.end())

{

output\_file.write(reinterpret\_cast<char\*>(&(bitsToCh[now])),sizeof(char));

now="";

pos++;

if(pos==size)

{

break;

}

}

else

{

now+=buffer[0];

cout<<now<<endl;

if(buffer.size()==1)

{

buffer="";

}

else

{

buffer=buffer.substr(1);

}

}

}

input\_file.close();

output\_file.close();

}

void useage(string prog)

{

cerr << "Useage: " << endl

<< " " << prog << "[-d] input\_file output\_file" << endl;

exit(2);

}

int main(int argc, char\* argv[])

{

int i;

string inputFilename, outputFilename;

bool isDecompress = false;

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

{

if (argv[i] == string("-d")) isDecompress = true;

else if (inputFilename == "") inputFilename = argv[i];

else if (outputFilename == "") outputFilename = argv[i];

else useage(argv[0]);

}

if (outputFilename == "") useage(argv[0]);

if (isDecompress) {

decompress(inputFilename, outputFilename);

}

else {

compress(inputFilename, outputFilename);

}

return 0;

}

#include <iostream>

#include <string>

#include <fstream>

#include <cstdlib>

#include <map>

#include <vector>

#include <bitset>

using namespace std;

struct Node

{

Node();

unsigned char ch;

int times;

Node\* left;

Node\* right;

};

//initialize

Node::Node()

{

ch=0;

times=0;

left=NULL;

right=NULL;

}

void scanFile(fstream& input\_file,map<unsigned char,int>& content)

{

char ch;

unsigned char key;

while(input\_file.get(ch))

{

key = reinterpret\_cast<unsigned char&>(ch);

if(content.find(key)==content.end())

{

content[key]=1;

}

else

{

content[key]++;

}

}

}

Node\* minNode(vector<Node>& save\_node,int& pos,vector<Node>& help\_save\_node,int& help\_pos,int& length)

{

//find Node with minest times number

if(pos==length)

{

help\_pos++;

return &help\_save\_node[help\_pos-1];

}

int next=help\_save\_node[help\_pos].times;

if(next==0)

{

pos++;

return &save\_node[pos-1];

}

else

{

if(save\_node[pos].times<=help\_save\_node[help\_pos].times)

{

pos++;

return &save\_node[pos-1];

}

else

{

help\_pos++;

return &help\_save\_node[help\_pos-1];

}

}

}

Node\* huffmanTree(vector<Node>& save\_node,vector<Node>& help\_save\_node,int& length)

{

int pos=0;

int help\_pos=0;

for(int i=length;i<2\*length-1;i++)

{

Node\* minNode1=minNode(save\_node,pos,help\_save\_node,help\_pos,length);

Node\* minNode2=minNode(save\_node,pos,help\_save\_node,help\_pos,length);

save\_node[i].left=minNode1;

save\_node[i].right=minNode2;

save\_node[i].ch=0;

save\_node[i].times=minNode1->times+minNode2->times;

help\_save\_node[i-length]=save\_node[i];

}

//root node of huffman tree

Node \*root=&save\_node[2\*length-2];

return root;

}

void compressTree(fstream& output\_file,Node\* node,map<unsigned char,string>& chToBits,string bits,string& tree,string& judge)

{

if(node->left==NULL)//leaf node

{

chToBits[node->ch]=bits;

tree+=node->ch;

judge+="l";//means this node is leaf

int size=bits.size();

bits=bits.substr(0,size-1);

return;

}

else//not leaf node

{

tree+=node->ch;

judge+="n";//means this node is not leaf

bits+="0";

compressTree(output\_file,node->left,chToBits,bits,tree,judge);

int size=bits.size();

bits=bits.substr(0,size-1);

bits+="1";

compressTree(output\_file,node->right,chToBits,bits,tree,judge);

size=bits.size();

if(size>0)

{

bits=bits.substr(0,size-1);

}

return;

}

}

void compress(string inputFilename,string outputFilename)

{

fstream input\_file(inputFilename, ios::in | ios::binary);

fstream output\_file(outputFilename, ios::out | ios::binary);

input\_file.seekg(0, ios::end);

//empty?

if (input\_file.tellg() == 0)

{

input\_file.close();

output\_file.close();

return;

}

input\_file.seekg(0, ios::beg);

//scan file and analyse

map<unsigned char,int>content;

scanFile(input\_file,content);

//save tree node

int length=content.size();

vector<Node> save\_node(2\*length-1);

vector<Node> help\_save\_node(length-1);

int pos=0;

int size=0;

for (map<unsigned char, int>::iterator it = content.begin(); it != content.end(); it++)

{

save\_node[pos].ch = (\*it).first;

save\_node[pos].times = (\*it).second;

save\_node[pos].left = NULL;

save\_node[pos].right = NULL;

size += save\_node[pos].times;

pos++;

}

//create a huffman tree

Node\* root= huffmanTree(save\_node,help\_save\_node,length);

//write head

map<unsigned char,string>chToBits;

string bits="";

string tree="";

string judge="";

compressTree(output\_file,root,chToBits,bits,tree,judge);

length=tree.size();

output\_file.write(reinterpret\_cast<char\*>(&size), sizeof(int));

output\_file.write(reinterpret\_cast<char\*>(&length), sizeof(int));

output\_file.write(tree.c\_str(), length);

output\_file.write(judge.c\_str(), length);

//write body

char temp;

unsigned char key;

bits="";

string now;

input\_file.close();

input\_file.open(inputFilename.c\_str(), ios::in | ios::binary);

while(input\_file.get(temp))

{

key = reinterpret\_cast<unsigned char&>(temp);

bits+=chToBits[key];

while(bits.length()>=8)

{

bitset<8>now(bits.substr(0,8));

if(bits.length()==8)

{

bits="";

}

else

{

bits=bits.substr(8);

}

unsigned char ch=(unsigned char)(now.to\_ulong());

output\_file.write(reinterpret\_cast<char\*>(&ch),sizeof(char));

}

}

if(bits!="")

{

bits+="00000000";

bitset<8>now(bits.substr(0,8));

unsigned char ch=(unsigned char)(now.to\_ulong());

output\_file.write(reinterpret\_cast<char\*>(&ch),sizeof(char));

}

input\_file.close();

output\_file.close();

}

void decompressTree(string tree,string judge,Node\* node,vector<Node>& save\_tree,int& vec\_pos)

{

int length=tree.size();

unsigned char ch=tree[0];

save\_tree[vec\_pos].ch=ch;

int vec\_now=vec\_pos;

vec\_pos++;

if(length==1)//leaf

{

return;

}

length-=1;

tree=tree.substr(1);

judge=judge.substr(1);

int pos=0;

int not\_leaf=0;

int leaf=0;

bool find=false;

bool contin=true;

string leftTree,rightTree,leftJudge,rightJudge;

while(pos<length&&!find)//find subtree

{

unsigned char now=tree[pos];

if(tree.size()==2)//leaf

{

leftTree=tree[0];

rightTree=tree[1];

break;

}

if(now==(unsigned char)(0)&&judge[pos]=='n')//not leaf

{

not\_leaf+=1;

}

else

{

leaf+=1;

}

pos++;

if(pos==tree.size())

{

contin=false;

}

if(leaf==(not\_leaf+1))

{

leftTree=tree.substr(0,pos);

if(contin)

{

rightTree=tree.substr(pos);

}

leftJudge=judge.substr(0,pos);

if(contin)

{

rightJudge=judge.substr(pos);

}

if(pos==1)

{

find=(rightJudge[rightJudge.size()-2]=='l')&&(rightJudge[rightJudge.size()-1]=='l');

}

else if(pos==tree.size()-1)

{

find=(leftJudge[leftJudge.size()-2]=='l')&&(leftJudge[leftJudge.size()-1]=='l');

}

else

{

find=(leftJudge[leftJudge.size()-2]=='l')&&(leftJudge[leftJudge.size()-1]=='l')

&&(rightJudge[rightJudge.size()-2]=='l')&&(rightJudge[rightJudge.size()-1]=='l');

}

}

}

save\_tree[vec\_now].left=&(save\_tree[vec\_pos]);

node->left=save\_tree[vec\_now].left;

decompressTree(leftTree,leftJudge,node->left,save\_tree,vec\_pos);

if(contin)

{

save\_tree[vec\_now].right=&(save\_tree[vec\_pos]);

node->right=save\_tree[vec\_now].right;

decompressTree(rightTree,rightJudge,node->right,save\_tree,vec\_pos);

}

}

void preorder(const vector<Node>& save\_tree,Node\* node,map<string,unsigned char>& bitsToCh,string& bits)

{

if(node->left == NULL)

{

bitsToCh[bits]=node->ch;

int length=bits.size();

bits=bits.substr(0,length-1);

return;

}

else

{

bits+="0";

preorder(save\_tree,node->left,bitsToCh,bits);

bits+="1";

preorder(save\_tree,node->right,bitsToCh,bits);

int length=bits.size();

if(length>0)

bits=bits.substr(0,length-1);

}

}

void decompress(string inputFilename,string outputFilename)

{

fstream input\_file(inputFilename, ios::in | ios::binary);

fstream output\_file(outputFilename, ios::out | ios::binary);

//empty?

input\_file.seekg(0, ios::end);

if (input\_file.tellg() == 0) {

input\_file.close();

output\_file.close();

return;

}

input\_file.seekg(0, ios::beg);

//read head

map<unsigned char,int>content;

int size,length;

string tree,judge;

int pos=0;

char ch;

unsigned char key;

input\_file.read(reinterpret\_cast<char\*>(&size),sizeof(int));

input\_file.read(reinterpret\_cast<char\*>(&length), sizeof(int));

while(pos<length)

{

input\_file.get(ch);

key = reinterpret\_cast<unsigned char&>(ch);

tree+=key;

pos++;

}

pos=0;

while(pos<length)

{

input\_file.get(ch);

key = reinterpret\_cast<unsigned char&>(ch);

judge+=key;

pos++;

}

Node root;

vector<Node>save\_tree(length);

pos=0;

//recreate the huffman tree

decompressTree(tree,judge,&root,save\_tree,pos);

map<string,unsigned char>bitsToCh;

string bits="";

preorder(save\_tree,&(save\_tree[0]),bitsToCh,bits);

//write

string now="";

string buffer="";

pos=0;

while(true)

{

if(buffer.size()==0)

{

unsigned char ch;

input\_file.read(reinterpret\_cast<char\*>(&ch),sizeof(char));

bitset<8>get(ch);

buffer+=get.to\_string();

}

if(bitsToCh.find(now)!=bitsToCh.end())//find

{

output\_file.write(reinterpret\_cast<char\*>(&(bitsToCh[now])),sizeof(char));

now="";

pos++;

if(pos==size)

{

break;

}

}

else//not find

{

now+=buffer[0];

if(buffer.size()==1)

{

buffer="";

}

else

{

buffer=buffer.substr(1);

}

}

}

input\_file.close();

output\_file.close();

}

void useage(string prog)

{

cerr << "Useage: " << endl

<< " " << prog << "[-d] input\_file output\_file" << endl;

exit(2);

}

int main(int argc, char\* argv[])

{

int i;

string inputFilename, outputFilename;

bool isDecompress = false;

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

{

if (argv[i] == string("-d")) isDecompress = true;

else if (inputFilename == "") inputFilename = argv[i];

else if (outputFilename == "") outputFilename = argv[i];

else useage(argv[0]);

}

if (outputFilename == "") useage(argv[0]);

if (isDecompress) {

decompress(inputFilename, outputFilename);

}

else {

compress(inputFilename, outputFilename);

}

return 0;

}