

数据结构课程设计

实验报告

**2022/2023(1)**



实验题目 大整数的运算

学生姓名 温家伟

学生学号 202103151422

学生班级 大数据分析 2101

任课教师 蒋莉

提交日期 2022-11-30

目录

[1.大型实验的内容 1](#_Toc27714)

[1.1 问题描述 1](#_Toc2691)

[1.2 基本要求 1](#_Toc3925)

[1.3 实验提示 1](#_Toc5517)

[2.程序的设计思路与整体流程 2](#_Toc27277)

[2.1 大整数的实现逻辑 2](#_Toc902)

[2.1.1List的模拟实现 2](#_Toc17371)

[2.1.2Vector的模拟实现 5](#_Toc10128)

[2.1.3Bignum的模拟实现 7](#_Toc31495)

[2.2 界面的设计 11](#_Toc32464)

[2.2.1Widget 11](#_Toc19225)

[2.2.2Button 12](#_Toc10436)

[2.2.3Table 12](#_Toc11235)

[2.2.4Window 13](#_Toc25365)

[2.3 管理器的设计 15](#_Toc32003)

[3.调试分析 15](#_Toc11547)

[3.1 技术难点分析 15](#_Toc11739)

[3.1.1界面的设计 15](#_Toc6261)

[3.1.2迭代器的模拟 15](#_Toc29255)

[3.2 调试错误分析 16](#_Toc14174)

[3.2.1宽字符的显示问题 16](#_Toc21028)

[3.2.2减法操作的一些问题 16](#_Toc24228)

[3.2.3屏幕闪烁严重的问题 17](#_Toc4574)

[3.2.4输入窗口弹两次（甚至更多次）的问题 17](#_Toc24689)

[4. 测试与运行截图 17](#_Toc9377)

[5. 实验总结 26](#_Toc20040)

[6. 附录（源代码） 27](#_Toc7393)

[6.1 main.cpp 27](#_Toc985)

[6.2 window.h 27](#_Toc30295)

[6.3 window.cpp 28](#_Toc2466)

[6.4 Table.h 33](#_Toc11423)

[6.5 Table.cpp 34](#_Toc5499)

[6.6 Button.h 37](#_Toc27681)

[6.7 Button.cpp 38](#_Toc22252)

[6.8 Widget.h 40](#_Toc1190)

[6.9 Widget.cpp 40](#_Toc26020)

[6.10 Bignum.hpp 41](#_Toc27411)

[6.11 \_\_reverse\_iterator.hpp 55](#_Toc27836)

[6.12 Vector.hpp 56](#_Toc20564)

[6.13 List.hpp 61](#_Toc20614)

[6.14 Manager.h 67](#_Toc24814)

[6.15 Manager.cpp 68](#_Toc1776)

[6.16 Algorithm.hpp 70](#_Toc12461)

**1.大型实验的内容**

**1.1问题描述**

密码学分为两类密码：对称密码和非对称密码。对称密码主要用于数据的加/解密，而非对称密码则主要用于认证、数字签名等场合。非对称密码在加密和解密时，是把加密的数据当作一个大的正整数来处理，这样就涉及到大整数的加、减、乘、除和指数运算等，同时，还需要对大整数进行输出。请采用相应的数据结构实现大整数的加、减、乘、除和指数运算，以及大整数的输入和输出。

**1.2基本要求**

（1）要求采用链表来实现大整数的存储和运算，不允许使用标准模板类的链表类(list)和函数。同时要求可以从键盘输入大整数，也可以文件输入大整数，大整数可以输出至显示器，也可以输出至文件。大整数的存储、运算和显示，可以同时支持二进制和十进制，但至少要支持十进制。大整数输出显示时，必须能清楚地表达出整数的位数。测试时，各种情况都需要测试，并附上测试截图；要求测试例子要比较详尽，各种极限情况也要考虑到，测试的输出信息要详细易懂，表明各个功能的执行正确；

（2）要求大整数的长度可以不受限制，即大整数的十进制位数不受限制，可以为十几位的整数，也可以为 500 多位的整数，甚至更长；大整数的运算和显示时，只需要考虑正的大整数。如果可能的话，请以秒为单位显示每次大整数运算的时间；

（3）要求采用类的设计思路，不允许出现类以外的函数定义，但允许友元函数。主函数中只能出现类的成员函数的调用，不允许出现对其它函数的调用；

（4）要求采用多文件方式：.h 文件存储类的声明，.cpp 文件存储类的实现，主函数 main 存储在另外一个单独的 cpp 文件中。如果采用类模板，则类的声明和实现都放在.h 文件中；

（5）不强制要求采用类模板，也不要求采用可视化窗口，要求源程序中有相应注释；

（6）建议采用 Visual C++ 6.0 及以上版本进行调试。

**1.3实验提示**

（1）大整数的加减运算可以分解为普通整数的运算来实现；而大整数的乘、除和指数运算，

可以分解为大整数的加减运算；

（2）大整数的加、减、乘、除和指数运算，一般是在求两大整数在取余操作下的加、减、乘、除和指数运算，即分别求 (a + b) mod n, (a - b) mod n, (a \* b) mod n, (a / b) mod n 和(a ^ b)mod n。其中 a ^ b 是求 a 的 b 次方，而 n 称之为模数。说明：取余操作(即 mod 操作)是计算相除之后所得的余数。不同于除法运算的是，取余操作得到的是余数，而不是除数。如 7 mod 5 = 2。模数 n 的设定，可以为 2^m 或 10^m，m 允许每次计算时从键盘输入。模数 n 的取值一般为2^512(相当于十进制 150 位左右), 2^1024(相当于十进制 200~300位), 2^2048(相当于十进制 300~500 位)。为了测试，模数 n 也可以为 2^256, 2^128等值；

（3）需要设计主要类有：链表类和大整数类。链表类用于处理链表的相关操作，包括缺省构造函数、拷贝构造函数、赋值函数、析构函数、链表的创建、插入、删除和显示等；而大整数类则用于处理大整数的各种运算和显示等。

**2.程序的设计思路与整体流程**

本次实验使用主体上分为三个部分，即大整数的实现逻辑、界面的设计和大整数的管理。

程序设计思路图

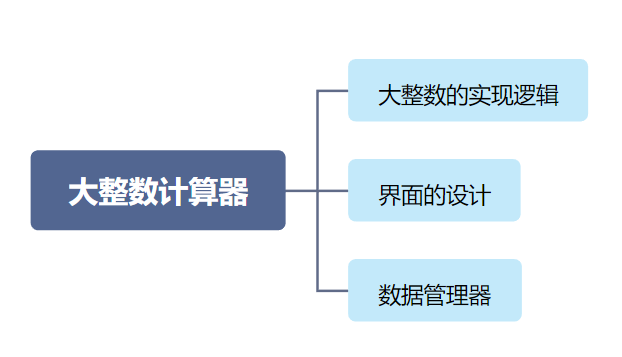


图2-1

大整数的实现逻辑包括Bignum.hpp几个文件，其中Vector和List分别模拟实现了STL库中的vector和list。然后，大整数类Bignum可由线性容器适配而成（仿照STL中stack由deque适配）。Vector和List两个类分别实现了迭代器，以此统一它们在Bignum类中的行为。

界面的设计包括Widget.h、Widget.cpp、Button.h、Button.cpp、Window.h、Window.cpp几个文件。Widget是抽象部件基类，Button按钮类和Window窗口类都是继承自Widget类。

数据管理器写在Manager.h和Manager.cpp两个文件中，主要用于管理数据读写文件，以及大整数数据的增删查改等。

**2.1大整数的实现逻辑**

**2.1.1 List的模拟实现**

List的实现模拟了STL中的list,是带头双向循环链表。并且封装了迭代器以模拟指针的行为，由于链表是按需申请的空间，不连续，都是一块一块的，所以不必考虑迭代器失效的问题。

list迭代器不同于string和vector迭代器，string类和vector类储存的数据是连续的，它们的迭代器类似指针，但是list类存储的数据是随机的，不能简单的用指针加减来进行访问。

为了使list满足迭代器的要求，我们对list结点进行封装，对结点指针的各种运算符操作进行重载。

模板参数列表当中为什么有三个模板参数？

template<class T, class Ref, class Ptr>

在list的模拟实现当中，我们typedef了两个迭代器类型，普通迭代器和const迭代器。

typedef \_\_list\_iterator<T, Ref, Ptr> iterator;

typedef \_\_list\_iterator<T, const T&, const T\*> const\_iterator;

迭代器类的模板参数列表当中的Ref和Ptr分别代表的是引用类型和指针类型。

当我们使用普通迭代器时，编译器就会实例化出一个普通迭代器对象；当我们使用const迭代器时，编译器就会实例化出一个const迭代器对象。其实，这很好的体现了代码的复用，当写好普通迭代器，然后再实现const迭代器的时候，我们首先想到的就是复制粘贴一份，然后再改出一份适于const类型的迭代器，但加入3个模板参数就可以很好地解决这个问题——普通类型模板传参T&、T\*，const类型模板传参const T&、const T\*。

template<class T>

struct list\_node

{

T \_data;

list\_node<T>\* \_next;

list\_node<T>\* \_prev;

// 节点的构造函数

list\_node(const T& x = T()):\_data(x), \_next(nullptr), \_prev(nullptr);

};

template<class T, class Ref, class Ptr>

struct \_\_list\_iterator

{

typedef list\_node<T> Node;

typedef \_\_list\_iterator<T, Ref, Ptr> iterator;

typedef T value\_type;

typedef Ptr pointer;

typedef Ref reference;

Node\* \_node;

// 迭代器的构造函数

\_\_list\_iterator(Node\* node):\_node(node);

// 重载!=

bool operator!=(const iterator& it) const;

// 重载==

bool operator==(const iterator& it) const;

// 重载解引用操作符

Ref operator\*();

// 重载->操作符

Ptr operator->();

// 重载+操作符

iterator operator+(int pos);

// 重载前置++操作符

iterator& operator++();

// 重载后置++操作符

iterator operator++(int);

// 重载前置--操作符

iterator& operator--();

// 重载后置--操作符

iterator operator--(int);

};

template<class T>

class List

{

typedef list\_node<T> Node;

public:

typedef \_\_list\_iterator<T, T&, T\*> iterator;

typedef \_\_list\_iterator<T, const T&, const T\*> const\_iterator;

typedef \_\_reverse\_iterator<iterator, T&, T\*> reverse\_iterator;

typedef \_\_reverse\_iterator<const\_iterator, const T&, const T\*> const\_reverse\_iterator;

// const迭代器的头

const\_iterator begin() const;

// const迭代器的尾

const\_iterator end() const;

// 迭代器的头

iterator begin();

// 迭代器的尾

iterator end();

// 反向迭代器的头

reverse\_iterator rbegin();

// const反向迭代器的头

const\_reverse\_iterator rbegin() const;

// 反向迭代器的尾

reverse\_iterator rend();

// const反向迭代器的尾

const\_reverse\_iterator rend() const;

// 容器的第一个元素

T& first();

// 链表的构造函数

List();

// 链表的拷贝构造函数

List(const List<T>& lt);

// 链表的赋值运算符重载函数

List<T>& operator=(List<T> lt);

// 析构函数

~List();

// 清理函数

void clear();

// 容器的大小

size\_t size() const;

// 尾插

void push\_back(const T& x);

// 头插

void push\_front(const T& x);

// 任意位置的插入函数

iterator insert(iterator pos, const T& x);

// 尾删

void pop\_back();

// 头删

void pop\_front();

// 任意位置的删除函数

iterator erase(iterator pos);

// 交换函数

void swap(List<T>& lt);

private:

Node\* \_head;

};

**2.1.2 Vector的模拟实现**

Vector同样模拟的是STL中的vector，由于他的迭代器就是原生指针，其他逻辑也都比较简单，此处便不展开赘述，详细的实现逻辑见源代码。

class Vector

{

public:

typedef T\* iterator;// 普通迭代器

typedef const T\* const\_iterator;// const迭代器

typedef \_\_reverse\_iterator<iterator, T&, T\*> reverse\_iterator;// 反向迭代器

typedef \_\_reverse\_iterator<const\_iterator, const T&, const T\*> const\_reverse\_iterator;// const反向迭代器

// 无参构造

Vector():\_start(nullptr), \_finish(nullptr), \_endofstorage(nullptr) ;

// 拷贝构造

Vector(const Vector<T>& v):\_start(nullptr), \_finish(nullptr), \_endofstorage(nullptr);

// 赋值运算符重载函数

Vector<T>& operator=(const Vector<T>& v);

// 析构函数

~Vector();

// 迭代器的头

iterator begin();

// const迭代器的头

const\_iterator begin() const;

// 迭代器的尾

iterator end();

// const迭代器的尾

const\_iterator end() const;

// 反向迭代器的头

reverse\_iterator rbegin();

// const反向迭代器的头

const\_reverse\_iterator rbegin() const;

// 反向迭代器的尾

reverse\_iterator rend();

// const反向迭代器的尾

const\_reverse\_iterator rend() const;

// 容器大小

size\_t size()const;

// 容器容量

size\_t capacity()const;

// 提前开空间

void reserve(size\_t n);

// 开空间并初始化

void resize(size\_t n, const T& val = T());

// 删除

iterator erase(iterator pos);

// 尾插

void push\_back(const T& x);

// 尾删

void pop\_back();

// 容器的第一个元素

T& first();

// 重载下标访问操作符

T& operator[](size\_t pos);

// 重载下标访问操作符的const版本

const T& operator[](size\_t pos) const;

// 插入

void insert(iterator pos, const T& x);

private:

iterator \_start;

iterator \_finish;

iterator \_endofstorage;

};

**2.1.3 Bignum的模拟实现**

因为用到了模板，而模板不支持分离编译，所以把.h和.cpp放到一个文件中，即.hpp文件。

template<class Container = List<int>>

//考虑用容器适配器封装，顺序容器的迭代器全部实现以此统一成员函数的实现逻辑

class Bignum

{

public:

//正向迭代器的头

typename Container::iterator begin();

//反向迭代器的头

typename Container::reverse\_iterator rbegin();

//正向迭代器的尾

typename Container::iterator end();

//反向迭代器的尾

typename Container::reverse\_iterator rend();

//大整数的长度

size\_t length()const;

//输入

void push\_back(int val);

//重载>

bool operator>(const Bignum& right)const;

//重载>=

bool operator>=(const Bignum& right)const;

//重载<

bool operator<(const Bignum& right)const;

//重载<=

bool operator<=(const Bignum& right)const;

//重载==

bool operator==(const Bignum& right)const;

//重载!=

bool operator!=(const Bignum& right)const;

//重载+

Bignum operator+(const Bignum& right)const;

//重载+=

Bignum& operator+=(const Bignum& right);

//重载-

Bignum operator-(const Bignum& right);

Bignum operator-(const Bignum& right)const;

//重载-=

Bignum& operator-=(const Bignum& right);

//重载\*

Bignum operator\*(const Bignum& right);

//重载\*=

Bignum& operator\*=(const Bignum& right);

//重载/

Bignum operator/(const Bignum& right);

//重载/=

Bignum& operator/=(const Bignum& right);

//重载%

Bignum operator%(const Bignum& right);

//重载%=

Bignum& operator%=(const Bignum& right);

//重载^

Bignum operator^(const Bignum& right);

//重载^=

Bignum& operator^=(const Bignum& right);

friend std::wstringstream& operator>>(std::wstringstream& in, Bignum& right);

//重载<<

friend ostream& operator<<(ostream& out, const Bignum& right);

// 显示函数

void display(ostream& out)const;

// 大整数转wstring

wstring Bignum2wstring();

wstring Bignum\_wstring()const;

// 大整数转string

string Bignum\_string()const;

//2进制转10进制

void bin2dec();

//10进制转2进制

void dec2bin();

private:

//工具函数，实现一位数乘大整数

Bignum \_miniMul(int num, const Bignum& bignum);

//工具函数，计算平方

Bignum square(Bignum& right);

Container \_con; //适配大整数

};

在Bignum这个类中，只有一个数据成员\_con,他的类型为Container，是用模板实现的，即template<class Container = List<int>> \_con用来存放大整数。这里就是模仿了STL中的适配器，（就像stack由deque适配出），模板给了缺省值List<int>,即符合了题目的用链表实现的要求。同时，在使用这个类的时候也可显示的传入其他参数。经测试，大多线性容器都可以，除了用自己实现的Vector<int>之外，也同样支持STL中的vector<int>、list<int>和deque<int>。

成员函数重点介绍+-\*/%^六个函数，其他有些代码较短，比较简单，详细实现逻辑见源代码。

重载二元算数运算符+ - \* / % ^ 时，这里特别说明，虽然用^来表示次方运算改变了^的原意（按位异或），但是map里也有重载[]的情况，而且确实^的优先级也比其他5个运算符高，此次实验中也不会用到按位异或的操作，^也符合用户的输入习惯，所以此处用^来表示次方运算。

+= -= \*= /= %= ^= 几个运算符分别复用对应的运算符即可。

下面是几个运算符的具体实现：

加法操作：模拟竖式加法，从后往前用两个迭代器遍历，carry表示进位。由此有三种情况，左边迭代器走到尾：ret插入右操作数+进位；右边迭代器走到尾：ret插入左操作数+进位；左右迭代器都没有走到尾：ret插入左操作数+右操作数+进位。

比如86043+5582：在图2-2中，定义两个反向迭代器，分别指在3和2的位置上。第一次相加进位是0；第二次相加时进位是1，所以第三次相加时就加上刚才相加的进位1。当迭代器分别走到6和5的位置，完成相加后，下一次相加由于右操作数上已经没有数字了，所以ret插入左操作数+进位。

加法示例图

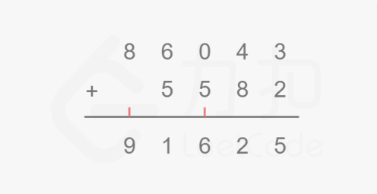


图2-2

减法操作：左等于右：直接返回0；左大于右：右的每一个数字乘以 -1，然后模拟竖式减法，borrow表示借位；左小于右：右减左，然后给第一个数字乘上 -1。

乘法操作模拟了竖式乘法，即个位乘以另一个数，加上十位乘以另一个数再乘十……，其中，一位数与大整数相乘单独写一个迷你乘法当作工具函数以供调用。加法操作复用了前面写好的加等运算符。

比如1234\*567：图2-3中，用迭代器反向遍历567，即可把乘法操作转化为下面的三个迷你乘法和两个加法操作。其中，加法得到的和存在ret里面，这里复用了前面写过的+=操作符；而迷你乘法专为本函数来写，所以设置为工具函数放在private里面，具体的实现思路为：

Bignum \_miniMul(int num, const Bignum& bignum);

一个int类型的整数乘以Bignum，即用迭代器遍历Bignum,用每一位乘以num，然后再维护一个carry来存放进位。

乘法示例图

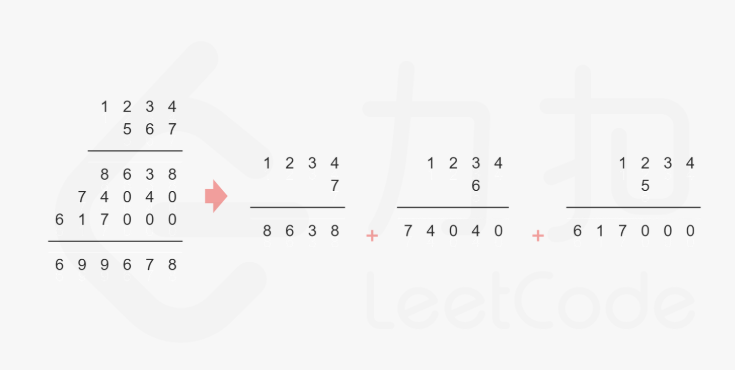


图2-3

除法操作模拟了竖式除法，先用迭代器定位出一个大整数类型的temp对象，每次商一个数，商好之后就减去已经除掉的部分，迭代器后移，以表示下一位数被拉下来。循环此过程即得到答案，最后记得把0去掉即可。

比如7375428413/125473：图2-4中，temp对象先取前六位737542（和125473一样的位数），然后从9开始，循环往下乘125473（9\*125473、8\*125473、7\*125473……）当得数小于737542的时候就跳出循环。而本例中5\*125473=627365，小于737542，跳出循环。这时，用737542减去刚才的得数627365，得到110187，这时再把8插入，即1101878。如此循环往复，直至求出答案。

除法示例图

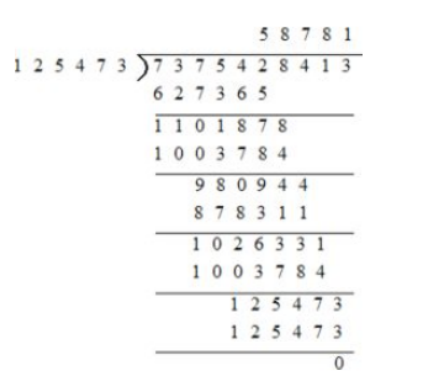


图2-4

%复用了除法，由数学关系式“返回值 = 左 - (左 / 右) \*右”得出，这个比较简单。

次方用了递归的算法，比如要算2的100次方，先算2的平方，再算2的平方的平方一直这样算下去直到逼近100.此时应该已经算出来2的64次方，那么接下来就只需算2的36次方，递归调用自己即可。

**2.2界面的设计**

界面设计用到了继承和多态的知识。界面用到三个类：按钮类、表格类和窗口类，而它们都有统一的一部分，那就是形状都是矩形。所以不妨先设计一个抽象部件基类，然后这三个类再去继承他们。这样可以大大减少代码的重复，为后续的维护工作提供了方便。

界面类关系示意图

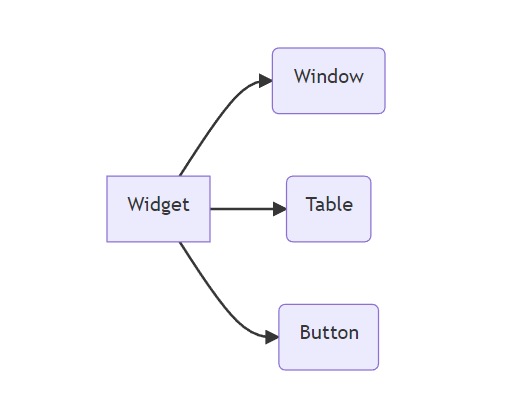


图2-5

**2.2.1 Widget**

using std::wstring;

// 部件抽象基类

class Widget

{

public:

Widget(const int& x = 0, const int& y = 0, const int& width = 100, const int& height = 60);

virtual void show() const = 0; // 纯虚函数，显示函数用多态来实现

protected:

int \_x; // X 坐标

int \_y; // Y 坐标

int \_width; // 宽度

int \_height; // 高度

};

现在的部件类就是一个长方形，然后显示函数设计为纯虚函数，以便之后的子类可以调到各自对应的show()函数。

**2.2.2 Button**

class Button :public Widget

{

public:

Button(const int& x, const int& y, const int& width,

const int& height, const wstring& text);

void show() const;

bool state(const ExMessage& msg) const;

const wstring& getText() const;

void setText(const std::wstring& text);

bool isIn(const ExMessage& msg) const;

private:

wstring \_text; // 按钮文本

};

按钮类继承自部件基类，按钮文本即按钮名称，提示用户进行相应操作。

show函数用以绘制按钮和按钮文字；

state函数用以判断按钮三种状态：鼠标悬浮在按钮上，鼠标点击按钮，其他；

getText函数用以获取按钮文本；

setText函数用以获取按钮文本；

isIn函数用以判断鼠标是否在按钮的矩形框内；

**2.2.3 Table**

// 表格类

class Table :

public Widget

{

public:

Table(const Manager& manager, const int& x = 0, const int& y = 0, const int& width = 100, const int& height = 60);

void show() const; // 显示表格

void pageUp(); // 上一页

void pageDown(); // 下一页

void showEditTable(const wchar\_t\* searchTerms); // 显示编辑表格

const int& getSearchIndex() const; // 获取搜索索引

private:

size\_t curIndex; // 当前大整数索引

int searchIndex; // 搜索索引

const Manager& manager; // 管理器引用

};

表格类也是继承自部件基类，用于显示从文件中读取的数据。

curIndex和searchIndex主要用于管理大整数及表格的绘制中的数学逻辑，manager用于管理大整数，依托一个表达式结构体（后面会提到）。

**2.2.4 Window**

class Window :public Widget

{

public:

// 窗口状态标识，每个状态代表了一个界面

enum WindowState { mainWindow, viewBigums };

#define BACKGROUND\_IMAGE L"background.jpg" // 背景图片名的常量

Window(const int& width = 600, const int& height = 400);

void show() const; // 显示窗口

void messageLoop(); // 消息循环

void close(); // 关闭窗口

void showMainWindow(); // 显示主窗口

void showViewBignums(); // 显示查看大整数窗口

Bignum<List<int>> wstring2Bignum(wstring str); // wstring转大整数

private:

// 主窗口按钮

Button\* mainWindow\_leftnum; // 左操作数

Button\* mainWindow\_rightnum; // 右操作数

Button\* mainWindow\_sign; // 操作符

Button\* mainWindow\_exit; // 退出程序

Button\* mainWindow\_RWfile; // 读写文件

// 查看大整数窗口按钮

Button\* RWfile\_back; // 返回

Button\* RWfile\_pageUp; // 上一页

Button\* RWfile\_pageDown; // 下一页

WindowState state; // 窗口状态

Table\* table; // 大数据表格

Manager \_manager; // 管理器

Bignum<List<int>> \_left; // 左操作数

Bignum<List<int>> \_right; // 右操作数

Bignum<List<int>> \_ret; // 结果

int \_base = 0; // 进制

// C++11的新特性，支持在声明时给缺省值

};

数据成员已经给出了详细的注释，成员函数见名知意。

构造函数完成了新建窗口、绘制表格，读取文件、设置绘图样式、创建按钮、显示主界面、进制选择等一系列操作。

消息循环函数的流程图如下：

消息循环函数的流程图

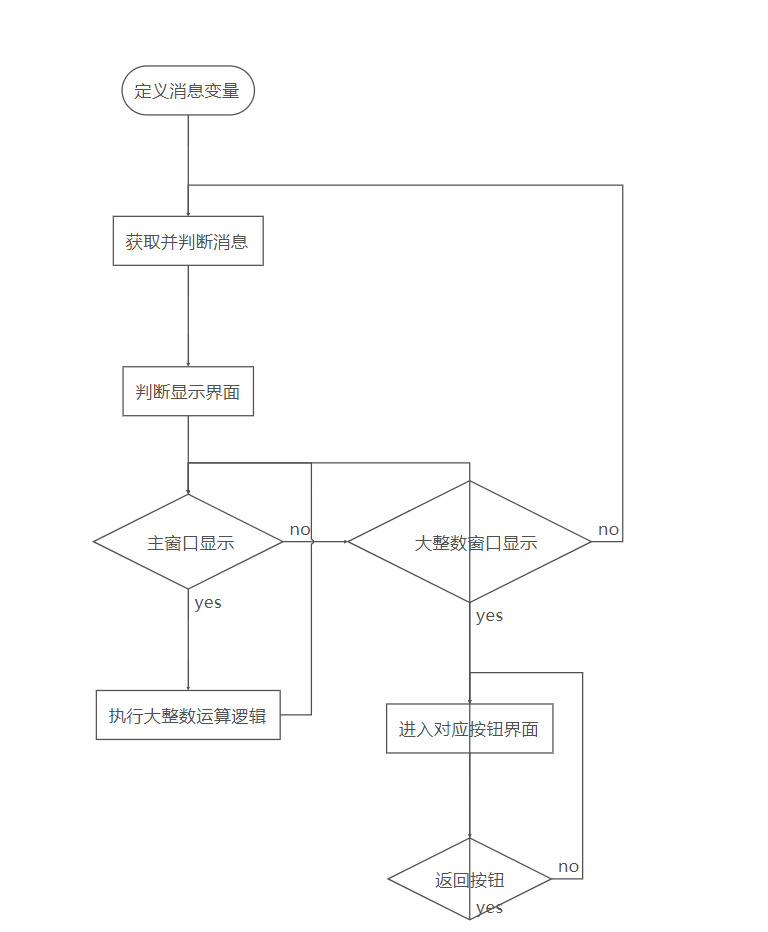


图2-6

**2.3管理器的设计**

struct Expression // 表达式结构体

{

Bignum<List<int>> \_left; // 左操作数

Bignum<List<int>> \_right; // 右操作数

Bignum<List<int>> \_result; // 结果

char \_sign; // 操作符

};

class Manager

{

public:

bool read(const std::string& fileName); // 读取文件到数据

void write(const std::string& fileName, string wstr) const; // 写入数据到文件

void addBignum(const Expression& bignum); // 添加大整数

const Expression& getBignums(const int& index) const; // 获取大整数数据

void setBignum(const int& index, const Expression& bignum); // 修改大整数

void deleteBignum(const int& index); // 删除大整数

const size\_t size() const; // 大整数数量

Bignum<List<int>> string2Bignum(string str); // string转大整数

private:

vector<Expression> \_bignums; // 大整数数组

};

首先定义一个表达式结构体Expression，成员有左右操作数、运算符和结果。

然后管理器类的数据成员由表达式数组构成。在成员函数方面，读写都是文本文件的读写，我把两个文件分开，一个用于读取，另一个用于写入。其他函数都见名知意，或是一些辅助性质的函数，此处不再展开。

**3.调试分析**

**3.1技术难点分析**

**3.1.1界面的设计**

早在写C++课设的时候就用到了easyX图形库，但那时最初的使用明显还不够纯熟：大量重复的代码，杂乱无章的函数调用关系，以及那时还没有现在统一的面向对象的思想。C++的面向对象的思想，就是对客观事物的抽象：即先描述再组织。实现界面类的时候同样如此，定义一个部件抽象基类，再派生出按钮类、表格类和窗口类三个子类。纯虚函数的定义，也能让子类在合适的时候调用到对应的方法。

**3.1.2迭代器的模拟**

可以说，迭代器的设计模式，是STL中的精华。本次实验模拟了迭代器。其中Vector的迭代器就是原生指针，而List的节点不连续，它的迭代器就需要进一步的封装。但在外部看来使用List的迭代器就像是在使用指针一样，这是因为它重载了指针的两个运算符\*和->。

**3.2调试错误分析**

**3.2.1宽字符的显示问题**

因为easy X库中用的是宽字符（串），不经转换直接显示会有乱码的现象。所以应当写出两者转换的函数来对应easy库中的字符。

乱码错误示意图



图3-1

**3.2.2减法操作的一些问题**

在写次方操作、取模操作的时候，总是会遇到一些莫名的错误，排查后才发现，原来是减法操作就写的有问题，比如43438-40239，先执行乘-1的操作。即43438+（-4 0 -2 -3 -9），得到03209，但在用借位转换的时候，等于0的情况也走了else,直接导致了错误。所以应该大于0、等于0、小于0三种情况分开。

减法单步调试过程图

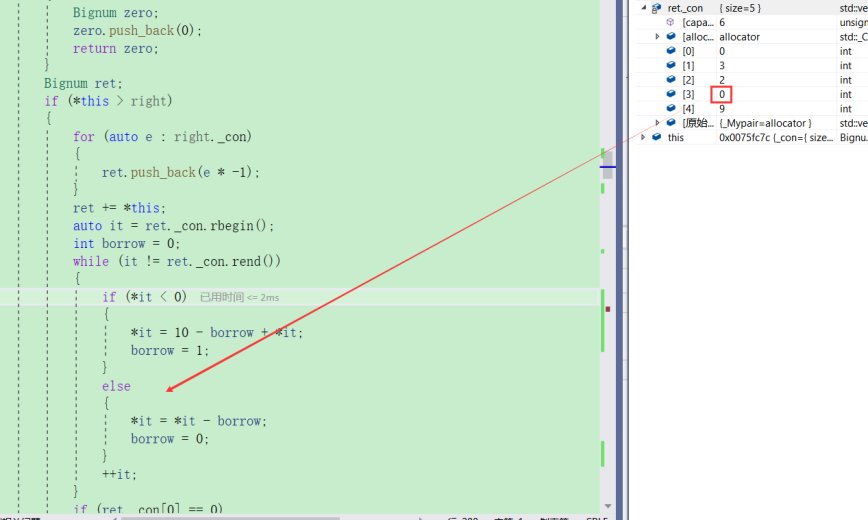


图3-2

**3.2.3屏幕闪烁严重的问题**

这个showMainWindow()函数原本写在了循环里，直接导致了每次接受鼠标消息后，都要执行这个函数，这就意味着这个函数随着鼠标的移动而被不断执行，屏幕也不断刷新，而把它放到构造函数里就可以解决这个问题。构造函数只在窗口类初始化地时候执行一次，很好地完成了showMainWindow()函数的功能，也就不会出现屏幕闪烁严重的问题了。

showMainWindow()函数位置图



图3-3

**3.2.4输入窗口弹两次（甚至更多次）的问题**

if (mainWindow\_leftnum->state(msg) && msg.message != WM\_LBUTTONUP)

这里的代码原本没有加&&后的内容，输入窗口会弹出很多次。这是因为在新的窗口创建之前，循环执行了好多次。在电脑上创建 InputBox 窗口的时间内，假设循环执行了四次，因此鼠标的消息队列有了四个鼠标事件，并且左键都是按下的状态。而系统的确在这个时间内产生了很多鼠标消息，而有些消息是有可能左键同时按下的，比如当消息 WM\_MOUSEMOVE 发生时，的确有可能鼠标左键按下为真。如果有这样多个消息产生，那么 InpuBox 就会弹出多次，所以应该检测是什么消息。

**4.测试与运行截图**

首先是运行后选择进制，非2非10会提醒再次输入。

我们先选择十进制进行测试。

进制选择示意图



图4-1

测试加法：123456789+987654321

加法测试示意图1

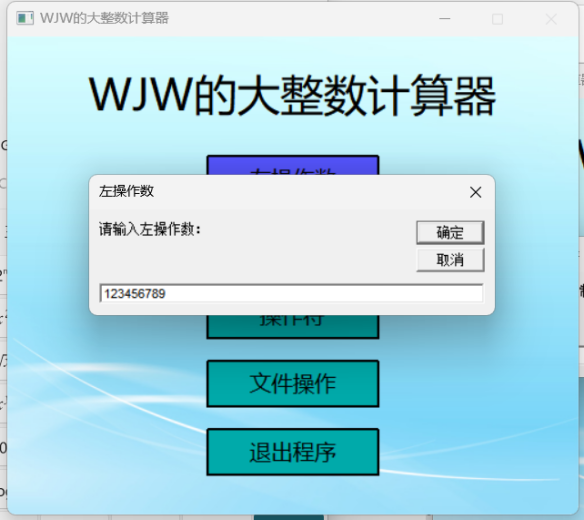


图4-2

加法测试示意图2

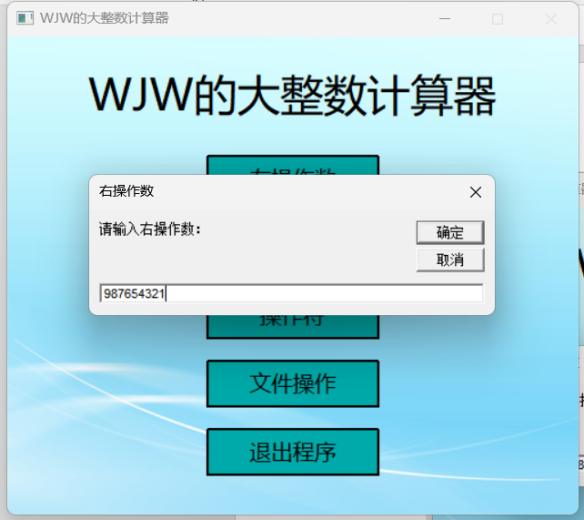


图4-3

加法测试示意图3

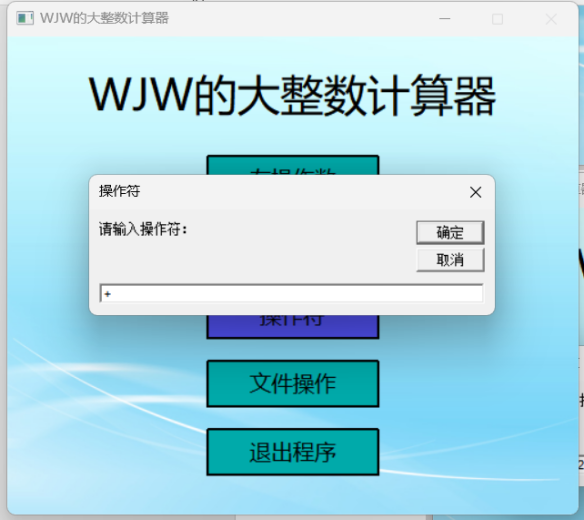


图4-4

加法测试示意图4



图4-5

加法测试示意图5



图4-6

测试减法：7186759-5100352

减法测试示意图



图4-7

测试乘法：999999999\*999999999

乘法测试示意图

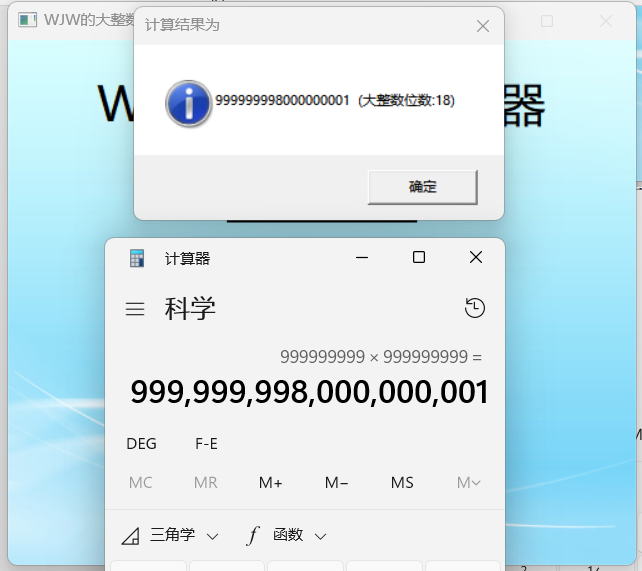


图4-8

测试除法：71807595100352\*2022

除法测试示意图



图4-9

测试取模：123456789%10

取模测试示意图

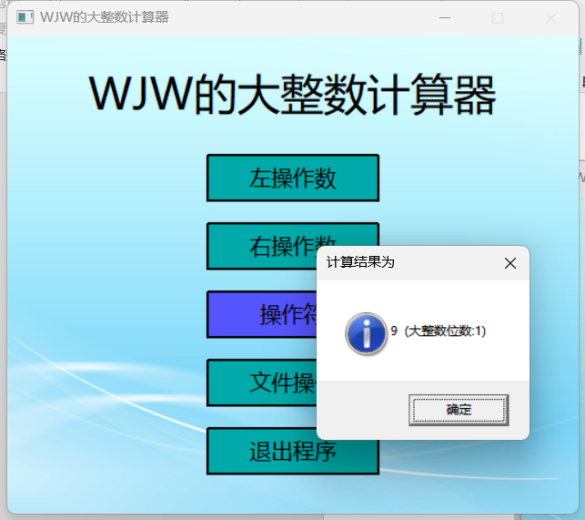


图4-10

测试次方：2^10000

次方测试示意图1

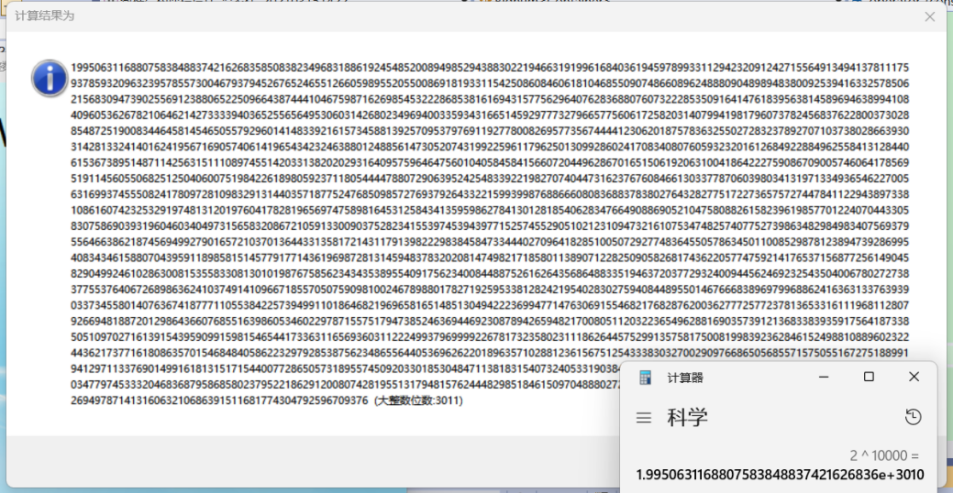


图4-11

次方测试示意图2



图4-12

运行过后我们可以看到答案都很好的存在了file.txt中：

文件测试示意图

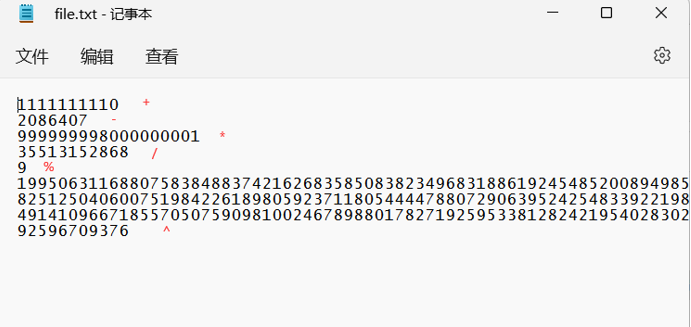


图4-13

图4-14到4-17是从文件中读取大整数的演示：

文件读取示意图1



图4-14

文件读取示意图2



图4-15

文件读取示意图3



图4-16

文件读取示意图4

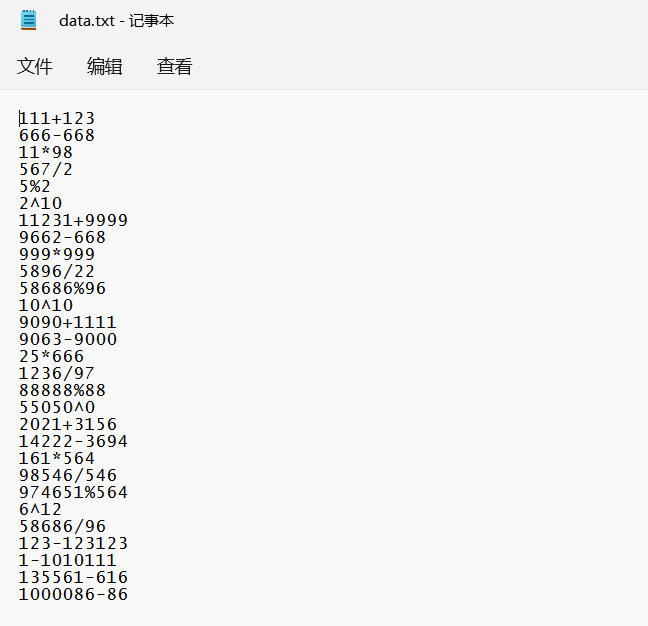


图4-17

下面是二进制计算的测试：

二进制测试示意图



图4-18

测试加法：1010101+1111111

二进制加法测试示意图

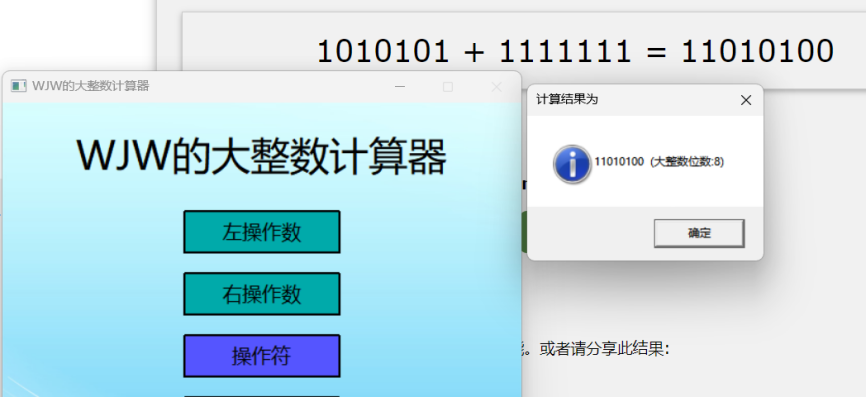


图4-19

测试减法：1001001001-1010101

二进制减法测试示意图

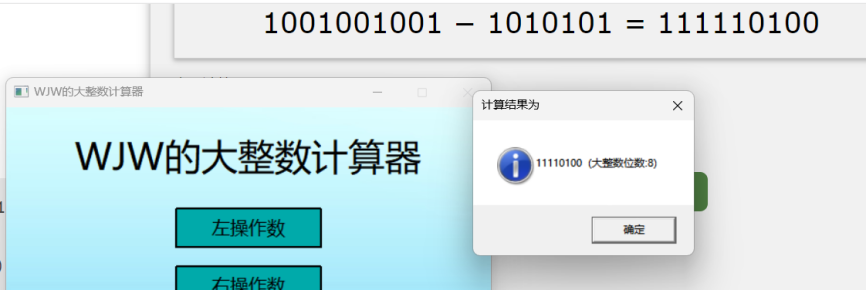


图4-20

测试乘法：111111111\*111111111

二进制乘法测试示意图

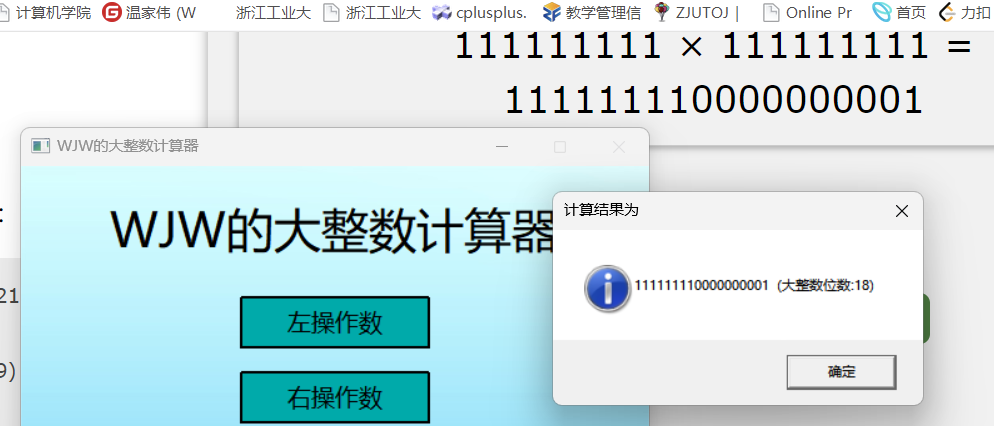


图4-21

测试除法：1001010010/10010

二进制除法测试示意图

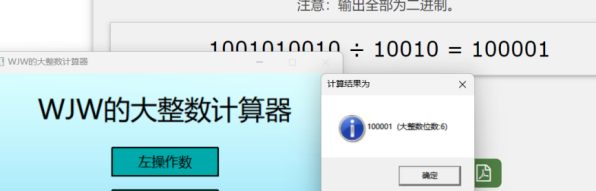


图4-22

测试取模：1000%11（8%3）

二进制取模测试示意图



图4-23

测试次方：10^111111(2^63)

二进制次方测试示意图1

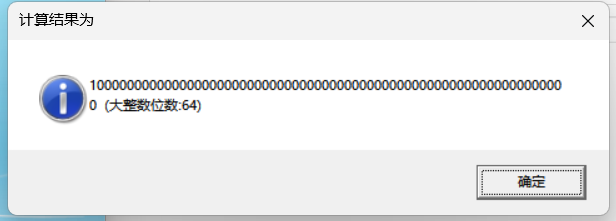
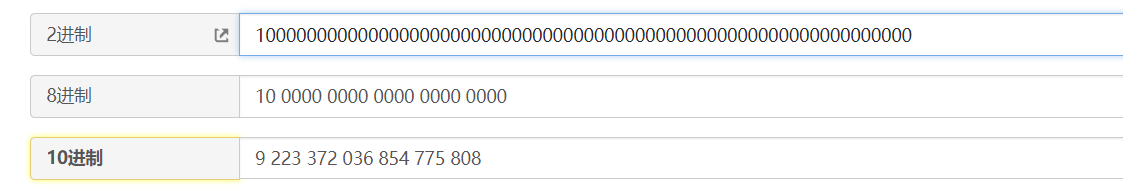


图4-24

二进制次方测试示意图2



二进制次方测试示意图3

图4-25



图4-26

可以看到，新增了一些文件内容，即二进制的测试样例：

二进制文件测试示意图

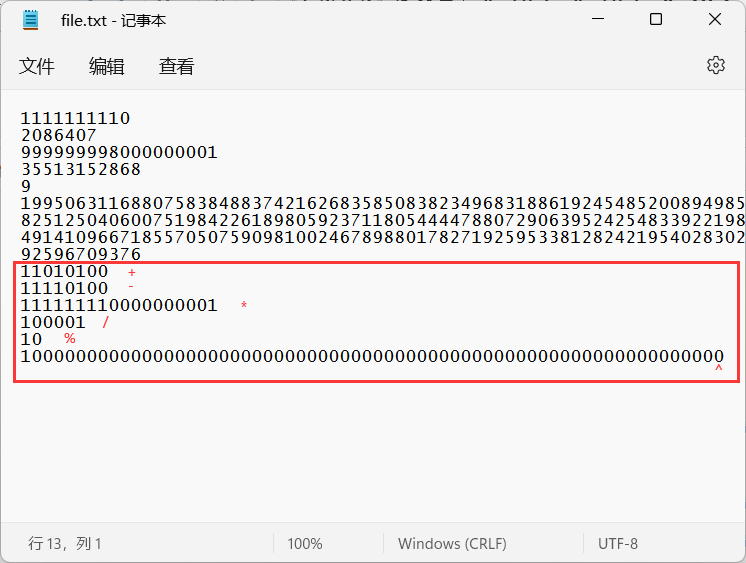


图4-27

退出界面示意图



图4-28

**5.实验总结**

这次在写课设前就想着把学过的C++知识都用进来，实验中包含了STL中的容器，迭代器，容器适配器的自定义实现。其实，在暑假就接触过这一部分的内容了，读了侯捷老师的《STL源码剖析》，当首次接触到这一部分的技术时，个人内心是非常震撼的：容器能对外完全屏蔽内部的复杂实现逻辑，迭代器又将不同的容器的行为统一起来，适配器用泛型编程做到了代码的复用。仿函数很好的替代了函数指针，以及我在见到仿函数之前完全没有想过可以重载operator()。并且在界面的设计上也用了继承和多态的知识，先写一个抽象部件基类，然后窗口类，和按钮类再去继承它，这些都是我在C++的课程设计中没有做到的。同时，这也算是我第一次按照面向对象的思想完完整整地写出的一份代码。都说软件工程要做到高内聚，低耦合。在这次实验中，写大整数的运算和设计界面是我分开进行的，但在最后合并地过程中并没有那么复杂，这就是面向对象的妙处！同时，在这次实验中也暴露了我的很多问题。比如，不改变数据成员的成员函数不加const，这就导致常对象无法调用它，编程的细节习惯还是要从平时注意起来。学了C++一年多了，终于才感觉摸到了C++的一些门道，看着我的那本厚厚的《C++ Primer Plus》，也在感慨，自己还有很多要学习的知识。

除此之外，本次课设还有一些不满意的地方，比如次方运算速度太慢了（2^30000大约要花6分钟，而我用python验证的时候确是秒出)，首先看到这个要求的时候我还想的是一项一项的乘，这是一个不能用的算法，但后来想到了可以逐渐逼近的递归算法，确实优化了一些但还是算的很慢，这还是我要继续改进的地方。

**6.附录（源代码）**

**6.1 main.cpp**

#include "Window.h"

int main()

{

Window window(502, 420);

window.messageLoop();

return 0;

}

**6.2 window.h**

#pragma once

#include "Table.h"

#include "Widget.h"

#include "Button.h"

#include "Manager.h"

#include <Windows.h>

#include <time.h>

#include "List.hpp"

class Window :public Widget

{

public:

// 窗口状态标识，每个状态代表了一个界面

enum WindowState { mainWindow, viewBigums, edit };

#define BACKGROUND\_IMAGE L"background.jpg" // 背景图片名的常量

Window(const int& width = 600, const int& height = 400);

void show() const; // 显示窗口

void messageLoop(); // 消息循环

void close(); // 关闭窗口

void showMainWindow(); // 显示主窗口

void showViewBignums(); // 显示查看大整数窗口

Bignum<List<int>> wstring2Bignum(wstring str); // wstring转大整数

private:

// 主窗口按钮

Button\* mainWindow\_leftnum; // 左操作数

Button\* mainWindow\_rightnum; // 右操作数

Button\* mainWindow\_sign; // 操作符

Button\* mainWindow\_exit; // 退出程序

Button\* mainWindow\_RWfile; // 读写文件

// 查看大整数窗口按钮

Button\* RWfile\_back; // 返回

Button\* RWfile\_pageUp; // 上一页

Button\* RWfile\_pageDown; // 下一页

WindowState state; // 窗口状态

Table\* table; // 大数据表格

Manager \_manager; // 管理器

Bignum<List<int>> \_left; // 左操作数

Bignum<List<int>> \_right; // 右操作数

Bignum<List<int>> \_ret; // 结果

int \_base = 0; // 进制

// C++11的新特性，支持在声明时给缺省值

};

**6.3 window.cpp**

#include "Window.h"

// 宽字符串转大整数

Bignum<List<int>> Window::wstring2Bignum(wstring str)

{

Bignum<List<int>> ret;

for (auto e : str)

{

ret.push\_back(e - '0');

}

return ret;

}

Window::Window(const int& width, const int& height) : Widget(0, 0, width, height)

{

show();

// 创建表格

table = new Table(\_manager, 10, 10, 502 - 20, 420 - 60);

if (!\_manager.read("data.txt"))

{

MessageBox(GetHWnd(), L"文件打开失败，无法对其进行操作！", L"错误", MB\_OK | MB\_ICONERROR);

exit(-1);

}

// 设置绘图样式

LOGFONT f; // 字的属性结构体

gettextstyle(&f); // 获取当前文字样式

f.lfQuality = DEFAULT\_QUALITY; // 指定文字的输出质量，指定输出质量不重要

settextstyle(&f); // 设置当前文字样式

settextcolor(BLACK); // 设置当前文字颜色

setbkmode(TRANSPARENT); // 设置当前设备图案填充和文字输出时的背景模式，背景是透明的

setlinecolor(BLACK); // 设置当前设备画线颜色

// 创建按钮

mainWindow\_leftnum = new Button((502 - 150) / 2, 105, 150, 40, L"左操作数");

mainWindow\_rightnum = new Button((502 - 150) / 2, 165, 150, 40, L"右操作数");

mainWindow\_sign = new Button((502 - 150) / 2, 225, 150, 40, L"操作符");

mainWindow\_RWfile = new Button((502 - 150) / 2, 285, 150, 40, L"文件操作");

mainWindow\_exit = new Button((502 - 150) / 2, 345, 150, 40, L"退出程序");

// 显示主界面

showMainWindow();

RWfile\_pageUp = new Button(10, 380, 100, 30, L"上一页");

RWfile\_pageDown = new Button(250, 380, 100, 30, L"下一页");

RWfile\_back = new Button(380, 380, 100, 30, L"返回");

// 进制选择

while (\_base != 2 && \_base != 10)

// 当输入进制不为2或10时，持续输入

{

wchar\_t base[100];

// 以对话框形式获取用户输入

if (InputBox(base, 100, L"请选择进制(2/10)：", L"进制选择", NULL, 0, 0, false))

{

std::wstringstream format(base);

format >> \_base;

}

}

}

void Window::messageLoop()

{

ExMessage msg; //保存鼠标消息的结构体

// 开启消息循环

while (true)

{

// 获取并判断消息

msg = getmessage();

//showMainWindow();

// 放到构造函数里去了，不然闪烁太严重

//https://tieba.baidu.com/p/2242135590

//解决InputBox弹两次的问题 && msg.message != WM\_LBUTTONUP

// 判断显示界面

if (state == WindowState::mainWindow) // 主窗口显示

{

// 输入左操作数

if (mainWindow\_leftnum->state(msg) && msg.message != WM\_LBUTTONUP)

{

wchar\_t leftnum[999];

if (InputBox(leftnum, 999, L"请输入左操作数：", L"左操作数", NULL, 0, 0, false))

{

wstring temp(leftnum);

\_left = wstring2Bignum(temp);

}

}

// 输入右操作数

else if (mainWindow\_rightnum->state(msg) && msg.message != WM\_LBUTTONUP)

{

wchar\_t rightnum[999];

if (InputBox(rightnum, 999, L"请输入右操作数：", L"右操作数", NULL, 0, 0, false))

{

wstring temp(rightnum);

\_right = wstring2Bignum(temp);

}

}

// 输入操作符

else if (mainWindow\_sign->state(msg) && msg.message != WM\_LBUTTONUP)

{

wchar\_t sign[10];

if (InputBox(sign, 10, L"请输入操作符：", L"操作符", NULL, 0, 0, false))

{

time\_t start\_time;

time(&start\_time);

if (\_base == 2)

{

\_left.bin2dec();

\_right.bin2dec();

}

if (sign[0] == '+')

{

\_ret = \_left + \_right;

}

else if (sign[0] == '-')

{

\_ret = \_left - \_right;

}

else if (sign[0] == '\*')

{

\_ret = \_left \* \_right;

}

else if (sign[0] == '/')

{

\_ret = \_left / \_right;

}

else if (sign[0] == '%')

{

\_ret = \_left % \_right;

}

else if (sign[0] == '^')

{

\_ret = \_left ^ \_right;

}

if (\_base == 2)

{

\_ret.dec2bin();

}

time\_t end\_time;

time(&end\_time);

wstring arr\_bignum = \_ret.Bignum2wstring();

wstring temp = to\_wstring(int(\_ret.length()));

arr\_bignum.append(wstring(L" (大整数位数:"));

arr\_bignum.append(temp);

arr\_bignum.append(wstring(L")"));

MessageBox(GetHWnd(), arr\_bignum.c\_str(), L"计算结果为", MB\_OK | MB\_ICONINFORMATION);

time\_t time = end\_time - start\_time;

wstring arr\_second = to\_wstring(int(time));

arr\_second.append(wstring(L"秒"));

MessageBox(GetHWnd(), arr\_second.c\_str(), L"计算所用时间", MB\_OK | MB\_ICONINFORMATION);

\_manager.write("file.txt",\_ret.Bignum\_string());

}

}

else if (mainWindow\_RWfile->state(msg))

{

showViewBignums();

}

// 退出程序

else if (mainWindow\_exit->state(msg))

{

MessageBox(GetHWnd(), L"感谢使用！", L"退出", MB\_OK | MB\_ICONINFORMATION);

return;

}

}

else if (state == Window::viewBigums)

{

if (RWfile\_back->state(msg) && msg.message != WM\_LBUTTONUP)

{

showMainWindow();

}

else if (RWfile\_pageUp->state(msg) && msg.message != WM\_LBUTTONUP)

{

table->pageUp();

}

else if (RWfile\_pageDown->state(msg) && msg.message != WM\_LBUTTONUP)

{

table->pageDown();

}

}

}

}

void Window::showMainWindow()

{

state = WindowState::mainWindow;

cleardevice();

// 加载背景图片

loadimage(NULL, BACKGROUND\_IMAGE, 502, 420);

// 绘制提示文字

RECT rect = { 0, 0, \_width, 100 };

settextstyle(50, 0, L"微软雅黑");

drawtext(L"WJW的大整数计算器", &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

}

void Window::showViewBignums()

{

state = WindowState::viewBigums;

cleardevice();

// 加载背景图片

loadimage(NULL, BACKGROUND\_IMAGE, 502, 420);

// 显示控件

table->show();

RWfile\_pageUp->show();

RWfile\_pageDown->show();

RWfile\_back->show();

}

void Window::show() const

{

// 新建窗口

SetWindowText(initgraph(\_width, \_height, EW\_NOCLOSE), L"WJW的大整数计算器");

setbkcolor(WHITE);

cleardevice();

}

void Window::close()

{

closegraph();

}

**6.4 Table.h**

#pragma once

#include "Widget.h"

#include "Manager.h"

#include <sstream>

// 表格类

class Table :

public Widget

{

public:

Table(const Manager& manager, const int& x = 0, const int& y = 0, const int& width = 100, const int& height = 60);

void show() const; // 显示表格

void pageUp(); // 上一页

void pageDown(); // 下一页

void showEditTable(const wchar\_t\* searchTerms); // 显示编辑表格

const int& getSearchIndex() const; // 获取搜索索引

private:

size\_t curIndex; // 当前大整数索引

int searchIndex; // 搜索索引

const Manager& manager; // 管理器引用

};

**6.5 Table.cpp**

#include "Table.h"

Table::Table(const Manager& manager, const int& x, const int& y

, const int& width, const int& height)

: Widget(x, y, width, height)

, manager(manager)

{

curIndex = 0;

searchIndex = -1;

}

void Table::show() const

{

// 设置绘图样式

setfillcolor(WHITE);

// 绘制表格

fillrectangle(\_x, \_y, \_x + \_width, \_y + \_height);

// 画竖线

for (int i = 130; i <= 370; i += 120)

{

line(i, 10, i, 10 + 360);

}

// 画横线

for (int j = 40; j <= 340; j += 30)

{

line(10, j, 10 + 482, j);

}

RECT rect;

// 绘制表头

wchar\_t header[5][5] = { L"左操作数",L"操作符", L"右操作数", L"结果"}; // 表头数据

for (int i = 10, j = 0; i < 490 && j < 5; i += 120, j++)

{

rect = { i, 10, i + 120, 10 + 30 };

drawtext(header[j], &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

}

// 输出大整数数据

size\_t i = curIndex;

for (int j = 40; j < 370; j += 30)

{

if (i < manager.size()) // 在不超出索引范围的情况下才读取数据

{

rect = { 10, j, 10 + 120, j + 30 };

drawtext(manager.getBignums(i).\_left.Bignum\_wstring().c\_str(), &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

rect = { 130, j, 130 + 120, j + 30 };

drawtext(manager.getBignums(i).\_sign, &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

rect = { 250, j, 250 + 120, j + 30 };

drawtext(manager.getBignums(i).\_right.Bignum\_wstring().c\_str(), &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

rect = { 370, j, 370 + 120, j + 30 };

drawtext(manager.getBignums(i).\_result.Bignum\_wstring().c\_str(), &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

i++;

}

else

{

break;

}

}

// 绘制页数提示

std::wstringstream format;

setbkmode(OPAQUE);

format << L"第" << (curIndex + 11) / 11 << L"页" << L"/" << L"共" << (manager.size() + 11) / 11 << L"页";

outtextxy(127, 383, format.str().c\_str());

setbkmode(TRANSPARENT);

}

void Table::pageUp()

{

// 输出大整数数据

curIndex -= 11;

if (curIndex > manager.size()) // 读取到第一页停止操作

{

curIndex += 11;

return;

}

show();

}

void Table::pageDown()

{

// 输出大整数数据

curIndex += 11;

if (curIndex > manager.size()) // 读取到最后一页停止操作

{

curIndex -= 11;

return;

}

show();

}

void Table::showEditTable(const wchar\_t\* searchTerms)

{

// 设置绘图样式

setfillcolor(WHITE);

// 绘制表格

fillrectangle(\_x, \_y, \_x + \_width, \_y + 60);

// 画竖线

for (int i = 130; i <= 370; i += 120)

{

line(i, 10, i, 10 + 60);

}

// 画横线

line(10, 40, 490, 40);

RECT rect;

// 绘制表头

wchar\_t header[4][3] = { L"学号", L"姓名", L"班级", L"总分" }; // 表头数据

for (int i = 10, j = 0; i < 490 && j < 4; i += 120, j++)

{

rect = { i, 10, i + 120, 10 + 30 };

drawtext(header[j], &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

}

}

const int& Table::getSearchIndex() const

{

return searchIndex;

}

**6.6 Button.h**

#pragma once

#include "Widget.h"

#include "Vector.hpp"

#include "List.hpp"

class Button :public Widget

{

public:

Button(const int& x, const int& y, const int& width,

const int& height, const wstring& text);

void show() const;

bool state(const ExMessage& msg) const;

const wstring& getText() const;

void setText(const std::wstring& text);

bool isIn(const ExMessage& msg) const;

private:

wstring \_text; // 按钮文本

};

**6.7 Button.cpp**

#include "Button.h"

Button::Button(const int& x

, const int& y

, const int& width

, const int& height

, const wstring& text)

: Widget(x, y, width, height)

, \_text(text)

{

}

void Button::show() const

{

// 设置样式

setlinestyle(PS\_SOLID, 2);

setfillcolor(CYAN);

settextstyle(25, 0, L"微软雅黑");

// 绘制按钮

fillrectangle(\_x, \_y, \_x + \_width, \_y + \_height);

// 绘制文本

RECT rect = { \_x, \_y, \_x + \_width, \_y + \_height };

drawtext(\_text.c\_str(), &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

}

bool Button::state(const ExMessage& msg) const

{

if (msg.message == WM\_MOUSEMOVE && isIn(msg)) // 按钮悬浮

{

// 设置样式

setlinestyle(PS\_SOLID, 2);

setfillcolor(LIGHTBLUE);

settextstyle(25, 0, L"微软雅黑");

// 绘制按钮

fillrectangle(\_x, \_y, \_x + \_width, \_y + \_height);

// 绘制文本

RECT rect = { \_x, \_y, \_x + \_width, \_y + \_height };

drawtext(\_text.c\_str(), &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

return false;

}

else if ((msg.message == WM\_LBUTTONDOWN || msg.message == WM\_LBUTTONUP) && isIn(msg)) // 按钮被点击

{

// 设置样式

setlinestyle(PS\_SOLID, 2);

setfillcolor(LIGHTBLUE);

settextstyle(25, 0, L"微软雅黑");

// 绘制按钮

fillrectangle(\_x, \_y, \_x + \_width, \_y + \_height);

// 绘制文本

RECT rect = { \_x, \_y, \_x + \_width, \_y + \_height };

drawtext(\_text.c\_str(), &rect, DT\_CENTER | DT\_VCENTER | DT\_SINGLELINE);

return true;

}

else // 恢复按钮原来的状态

{

show();

return false;

}

}

const wstring& Button::getText() const

{

return \_text;

}

void Button::setText(const std::wstring& text)

{

this->\_text = text;

}

bool Button::isIn(const ExMessage& msg) const

{

if (msg.x >= this->\_x && msg.x <= this->\_x + \_width

&& msg.y >= this->\_y && msg.y <= this->\_y + \_height)

{

return true;

}

return false;

}

**6.8 Widget.h**

#pragma once

#include <graphics.h>

#include <time.h>

#include <conio.h>

#include <string>

#include "Vector.hpp"

#include "List.hpp"

using std::wstring;

// 部件抽象基类

class Widget

{

public:

Widget(const int& x = 0, const int& y = 0, const int& width = 100, const int& height = 60);

virtual void show() const = 0; // 纯虚函数，显示函数用多态来实现

protected:

int \_x; // X 坐标

int \_y; // Y 坐标

int \_width; // 宽度

int \_height; // 高度

};

**6.9 Widget.cpp**

#include "Widget.h"

Widget::Widget(const int& x, const int& y, const int& width, const int& height)

: \_x(x)

, \_y(y)

, \_width(width)

, \_height(height)

{

}

**6.10 Bignum.hpp**

#pragma once

#include <list>

#include <vector>

#include <deque>

#include <string>

#include <iostream>

#include "Vector.hpp"

#include "List.hpp"

#include "Algorithm.hpp"

using namespace std;

template<class Container = List<int>>

//考虑用容器适配器封装，顺序容器的迭代器全部实现以此统一成员函数的实现逻辑

class Bignum

{

public:

//正向迭代器的头

typename Container::iterator begin()

{

return \_con.begin();

}

//反向迭代器的头

typename Container::reverse\_iterator rbegin()

{

return \_con.rbegin();

}

//正向迭代器的尾

typename Container::iterator end()

{

return \_con.end();

}

//反向迭代器的尾

typename Container::reverse\_iterator rend()

{

return \_con.rend();

}

//大整数的长度

size\_t length()const

{

return \_con.size();

}

//输入

void push\_back(int val)

{

\_con.push\_back(val);

}

//重载>

bool operator>(const Bignum& right)const

{

//先比较长度，长度大的数值大

if (length() > right.length())

{

return true;

}

else if (length() < right.length())

{

return false;

}

//再逐个比较，相等就接着比

else

{

auto it = \_con.begin();

auto right\_it = right.\_con.begin();

while (it != \_con.end())

{

if (\*it < \*right\_it)

{

return false;

}

else if (\*it > \*right\_it)

{

return true;

}

else

{

++it;

++right\_it;

}

}

return false;//循环结束的话说明两者相等

}

}

//重载>=

bool operator>=(const Bignum& right)const

{

return (\*this > right) || (\*this == right);

}

//重载<

bool operator<(const Bignum& right)const

{

return !(\*this >= right);

}

//重载<=

bool operator<=(const Bignum& right)const

{

return (\*this < right) || (\*this == right);

}

//重载==

bool operator==(const Bignum& right)const

{

if (length() != right.length())

{

return false;

}

else

{

auto it\_con = \_con.begin();

auto it\_right = right.\_con.begin();

while (it\_con != \_con.end())

{

if (\*it\_con != \*it\_right)

{

return false;

}

else

{

++it\_con;

++it\_right;

}

}

}

return true;

}

//重载!=

bool operator!=(const Bignum& right)const

{

return !(\*this == right);

}

//重载+

Bignum operator+(const Bignum& right)const

{

int carry = 0; // 进位

Bignum ret;

auto it\_con = \_con.rbegin();

auto it\_right = right.\_con.rbegin();

while (it\_con != \_con.rend() || it\_right != right.\_con.rend())

{

if (it\_con == \_con.rend())

{

ret.push\_back((\*it\_right + carry) % 10);

carry = (\*it\_right + carry) / 10;

it\_right++;

}

else if (it\_right == right.\_con.rend())

{

ret.push\_back((\*it\_con + carry) % 10);

carry = (\*it\_con + carry) / 10;

it\_con++;

}

else

{

ret.push\_back((\*it\_con + \*it\_right + carry) % 10);

carry = (\*it\_con + \*it\_right + carry) / 10;

it\_con++;

it\_right++;

}

}

if (carry)

{

ret.push\_back(carry);

}

Reverse(ret.begin(), ret.end());

return ret;

}

//重载+=

Bignum& operator+=(const Bignum& right)

{

\*this = \*this + right;

return \*this;

}

//重载-

Bignum operator-(const Bignum& right)

{

Bignum ret;

if (\*this == right)

{

Bignum zero;

zero.push\_back(0);

return zero;

}

else if (\*this > right)

{

for (auto e : right.\_con)

{

ret.push\_back(e \* -1);

}

ret += \*this;

while (\*ret.\_con.begin() == 0)

{

ret.\_con.erase(ret.begin());

}

auto it = ret.\_con.rbegin();

int borrow = 0;

while (it != ret.\_con.rend())

{

if (\*it < 0)

{

\*it = 10 - borrow + \*it;

borrow = 1;

}

else if (\*it > 0)

{

\*it = \*it - borrow;

borrow = 0;

}

else

{

if (borrow)

{

\*it = 10 - borrow + \*it;

}

}

++it;

}

while (\*ret.\_con.begin() == 0)

{

ret.\_con.erase(ret.begin());

}

}

else

{

ret = right - \*this;

\*ret.\_con.begin() \*= -1;

}

return ret;

}

Bignum operator-(const Bignum& right)const

{

Bignum ret;

if (\*this == right)

{

Bignum zero;

zero.push\_back(0);

return zero;

}

else if (\*this > right)

{

for (auto e : right.\_con)

{

ret.push\_back(e \* -1);

}

ret += \*this;

while (\*ret.\_con.begin() == 0)

{

ret.\_con.erase(ret.begin());

}

auto it = ret.\_con.rbegin();

int borrow = 0;

while (it != ret.\_con.rend())

{

if (\*it < 0)

{

\*it = 10 - borrow + \*it;

borrow = 1;

}

else if (\*it > 0)

{

\*it = \*it - borrow;

borrow = 0;

}

else

{

if (borrow)

{

\*it = 10 - borrow + \*it;

}

}

++it;

}

while (\*ret.\_con.begin() == 0)

{

ret.\_con.erase(ret.begin());

}

}

else

{

ret = right - \*this;

\*ret.\_con.begin() \*= -1;

}

return ret;

}

//重载-=

Bignum& operator-=(const Bignum& right)

{

\*this = \*this - right;

return \*this;

}

//重载\*

Bignum operator\*(const Bignum& right)

{

Bignum ret;

auto it = right.\_con.rbegin();

int N = 0;

while (it != right.\_con.rend())

{

//调用工具函数

Bignum temp = (\_miniMul(\*it, \*this));

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

{

temp.push\_back(0);

}

N++;

ret += temp;

++it;

}

return ret;

}

//重载\*=

Bignum& operator\*=(const Bignum& right)

{

\*this = \*this \* right;

return \*this;

}

//重载/

Bignum operator/(const Bignum& right)

{

Bignum ret;

Bignum zero;

zero.push\_back(0);

if (\*this < right)

{

ret.push\_back(0);

}

else if (\*this == right)

{

ret.push\_back(1);

}

else

{

//模拟竖式除法

Bignum temp;

auto it = \_con.begin();

for (size\_t i = 0; i < right.length() - 1; ++i)

{

temp.push\_back(\*it);

++it;

}

temp.push\_back(\*it);

while (it != \_con.end())

{

for (int i = 9; i >= 0; --i)

{

if (\_miniMul(i, right) <= temp)

{

{

ret.push\_back(i);

}

temp -= \_miniMul(i, right);

if (it + 1 != \_con.end())

{

if (temp == zero)

{

temp.\_con.erase(temp.begin());

}

temp.push\_back(\*(it + 1));

}

break;

}

}

++it;

}

}

if (\*ret.begin() == 0 && ret.\_con.size() != 1)

{

ret.\_con.erase(ret.begin());

}

return ret;

}

//重载/=

Bignum& operator/=(const Bignum& right)

{

\*this = \*this / right;

return \*this;

}

//重载%

Bignum operator%(const Bignum& right)

{

Bignum ret;

ret = \*this - (\*this / right) \* right;

return ret;

}

//重载%=

Bignum& operator%=(const Bignum& right)

{

\*this = \*this % right;

return \*this;

}

//重载^

Bignum operator^(const Bignum& right)

{

Bignum zero;

zero.push\_back(0);

Bignum one;

one.push\_back(1);

Bignum two;

two.push\_back(2);

if (right == zero)

{

return one;

}

else if (right == one)

{

return \*this;

}

else if (right == two)

{

return square(\*this);

}

Bignum ret = \*this;

Bignum pow;

pow.push\_back(1);

while (pow \* two < right)

{

ret = square(ret);

pow \*= two;

}

ret \*= ((\*this) ^ (right - pow));

return ret;

}

//重载^=

Bignum& operator^=(const Bignum& right)

{

\*this = \*this ^ right;

return \*this;

}

friend std::wstringstream& operator>>(std::wstringstream& in, Bignum& right)

{

char temp = 0;

while (true)

{

temp = getchar(); //神奇的getchar

if (temp == ' ' || temp == '\n')

{

break;

}

right.push\_back(temp - '0');

}

return in;

}

//重载<<

friend ostream& operator<<(ostream& out, const Bignum& right)

{

for (auto e : right.\_con)

//这里又犯了低级错误

//for(auto e : \_con)

//友元函数已不属于这个类了，只是能访问到这个类的数据而已

//所以直接\_con里面是没有数据的

{

out << e;

}

return out;

}

// 显示函数

void display(ostream& out)const

{

for (auto e : \_con)

{

out << e;

}

out << endl;

}

// 大整数转wstring

wstring Bignum2wstring()

{

wstring ret;

auto it = \_con.begin();

if (\*it < 0)

{

\*it \*= -1;

ret.push\_back('-');

}

for (auto e : \_con)

{

ret.push\_back(e + '0');

}

return ret;

}

wstring Bignum\_wstring()const

{

wstring ret;

auto it = \_con.begin();

if (\*it < 0)

{

ret.push\_back('-');

ret.push\_back(\*it \* (-1) + '0');

}

for (auto e : \_con)

{

if (e >= 0)

ret.push\_back(e + '0');

}

return ret;

}

// 大整数转string

string Bignum\_string()const

{

string ret;

auto it = \_con.begin();

if (\*it < 0)

{

ret.push\_back('-');

}

for (auto e : \_con)

{

ret.push\_back(e + '0');

}

return ret;

}

//2进制转10进制

void bin2dec()

{

Bignum ret;

ret.push\_back(0);

Bignum pow;

pow.push\_back(0);

Bignum one;

one.push\_back(1);

Bignum two;

two.push\_back(2);

auto it = \_con.rbegin();

while (it != \_con.rend())

{

ret += \_miniMul(\*it, two ^ pow);

pow += one;

it++;

}

\*this = ret;

}

//10进制转2进制

void dec2bin()

{

Bignum ret;

Bignum zero;

zero.push\_back(0);

Bignum two;

two.push\_back(2);

while (\*this != zero)

{

Bignum temp = \*this % two;

ret.push\_back(temp.\_con.first());

\*this /= two;

}

Reverse(ret.begin(), ret.end());

\*this = ret;

}

private:

//工具函数，实现一位数乘大整数

Bignum \_miniMul(int num, const Bignum& bignum)

{

Bignum ret;

Bignum zero;

zero.push\_back(0);

if (num == 0)

{

return zero;

}

int carry = 0;//进位

//这里出现过忘记用反向迭代器，直接用语法糖遍历的错误

auto it = bignum.\_con.rbegin();

while (it != bignum.\_con.rend())

{

ret.push\_back((\*it \* num + carry) % 10);

carry = (\*it \* num + carry) / 10;

++it;

}

if (carry)

{

ret.push\_back(carry);

}

Reverse(ret.begin(), ret.end());

return ret;

}

//工具函数，计算平方

Bignum square(Bignum& right)

{

return right \* right;

}

Container \_con; //适配大整数

};

**6.11 \_\_reverse\_iterator.hpp**

#pragma once

template<class iterator, class Ref, class Ptr>

struct \_\_reverse\_iterator

{

typedef \_\_reverse\_iterator<iterator, Ref, Ptr> riterator;

iterator \_cur;

// 利用正向迭代器实现反向迭代器的构造函数

\_\_reverse\_iterator(iterator it)

: \_cur(it)

{

}

// 复用正向迭代器的操作实现反向迭代器

// 正向迭代器的begin

riterator operator++()

{

\_cur--;

return \*this;

}

riterator operator++(int)

{

riterator tmp(\*this);

\_cur--;

return tmp;

}

riterator operator--()

{

\_cur++;

return \*this;

}

riterator operator--(int)

{

riterator tmp(\*this);

\_cur++;

return tmp;

}

Ref operator\*()

{

// 因为正向迭代器的begin是反向的end，正向的end是反向的begin，stl中的设计是保持对称的

// 而正向的end是\_finish，是指向最后一个元素的下一个位置

// 所以方向迭代器取元素时要先减减再解引用

iterator tmp = \_cur;

tmp--;

return \*tmp;

}

Ptr operator->()

{

return &(operator\*());

}

bool operator!=(const riterator& it)

{

return \_cur != it.\_cur;

}

bool operator==(const riterator& it)

{

return \_cur == it.\_cur;

}

};

**6.12 Vector.hpp**

#pragma once

#include<iostream>

#include<cassert>

#include "List.hpp"

#include "\_\_reverse\_iterator.hpp"

using namespace std;

template<class T>

class Vector

{

public:

typedef T\* iterator;//普通迭代器

typedef const T\* const\_iterator;//const迭代器

typedef \_\_reverse\_iterator<iterator, T&, T\*> reverse\_iterator;//反向迭代器

typedef \_\_reverse\_iterator<const\_iterator, const T&, const T\*> const\_reverse\_iterator;//const反向迭代器

// 无参构造

Vector()

:\_start(nullptr)

, \_finish(nullptr)

, \_endofstorage(nullptr)

{

}

// 拷贝构造

Vector(const Vector<T>& v)

:\_start(nullptr)

, \_finish(nullptr)

, \_endofstorage(nullptr)

{

\_start = new T[v.capacity()]; //开辟一块和容器v大小相同的空间

for (size\_t i = 0; i < v.size(); i++) //将容器v当中的数据一个个拷贝过来

{

\_start[i] = v[i];

}

\_finish = \_start + v.size(); // 容器有效数据的尾

\_endofstorage = \_start + v.capacity(); // 整个容器的尾

}

// 赋值运算符重载函数

Vector<T>& operator=(const Vector<T>& v)

{

if (this != &v)

{

delete[]\_start; // 释放原空间

\_start = new T[v.capacity()]; // 开辟新空间

for (size\_t i = 0; i < v.size(); i++) // 拷贝数据

{

\_start[i] = v[i];

}

\_finish = \_start + v.size(); // 更新\_finish

\_endofstorage = \_start + v.capacity(); // 更新\_capacity

}

return \*this;

}

// 析构函数

~Vector()

{

delete[] \_start;

}

// 迭代器的头

iterator begin()

{

return \_start;

}

// const迭代器的头

const\_iterator begin() const

{

return \_start;

}

// 迭代器的尾

iterator end()

{

return \_finish;

}

// const迭代器的尾

const\_iterator end() const

{

return \_finish;

}

// 反向迭代器的头

reverse\_iterator rbegin()

{

return reverse\_iterator(end());

}

// const反向迭代器的头

const\_reverse\_iterator rbegin() const

{

return const\_reverse\_iterator(end());

}

// 反向迭代器的尾

reverse\_iterator rend()

{

return reverse\_iterator(begin());

}

// const反向迭代器的尾

const\_reverse\_iterator rend() const

{

return const\_reverse\_iterator(begin());

}

// 容器大小

size\_t size()const

{

return \_finish - \_start;

}

// 容器容量

size\_t capacity()const

{

return \_endofstorage - \_start;

}

// 提前开空间

void reserve(size\_t n)

{

size\_t sz = size();

if (n > capacity())

{

T\* temp = new T[n];//开新空间

//把原有的数据拷入

if (\_start)

{

memcpy(temp, \_start, sizeof(T) \* size());

delete[] \_start;

}

\_start = temp;

}

\_finish = \_start + sz;

\_endofstorage = \_start + n;

}

// 开空间并初始化

void resize(size\_t n, const T& val = T())

{

//开空间

if (n > capacity())

{

reserve(n);

}

//初始化

if (n > size())

{

while (\_finish < \_start + n)

{

\*\_finish = val;

++\_finish;

}

}

else

{

\_finish = \_start + n;

}

}

// 删除

iterator erase(iterator pos)

{

assert(pos >= \_start && pos < \_finish);

iterator cur = pos + 1;

while (cur < \_finish)

{

\*(cur - 1) = \*cur;

cur++;

}

\_finish--;

return pos;//更新pos位置

}

// 尾插

void push\_back(const T& x)

{

//满了的话就先开空间

if (\_endofstorage == \_finish)

{

size\_t newCapacity = capacity() == 0 ? 4 : 2 \* capacity();

reserve(newCapacity);

}

\*\_finish = x;

++\_finish;

}

// 尾删

void pop\_back()

{

if (\_finish > \_start)

{

--\_finish;

}

}

// 容器的第一个元素

T& first()

{

return \_start[0];

}

// 重载下标访问操作符

T& operator[](size\_t pos)

{

assert(pos < size());

return \_start[pos];

}

// 重载下标访问操作符的const版本

const T& operator[](size\_t pos) const

{

assert(pos < size());

return \_start[pos];

}

// 插入

void insert(iterator pos, const T& x)

{

//检查越界

assert(pos >= \_start && pos <= \_finish);

//满了就扩容

if (\_finish == \_endofstorage)

{

//扩容后pos就失效了，需要更新

size\_t n = pos - \_start;

size\_t newCapacity = capacity() == 0 ? 4 : 2 \* capacity();

reserve(newCapacity);

pos = \_start + n;

}

//挪动数据

iterator end = \_finish - 1;

while (end >= pos)

{

\*(end + 1) = \*end;

--end;

}

\*pos = x;

++\_finish;

}

private:

iterator \_start;

iterator \_finish;

iterator \_endofstorage;

};

**6.13 List.hpp**

#pragma once

#include "\_\_reverse\_iterator.hpp"

template<class T>

struct list\_node

{

T \_data;

list\_node<T>\* \_next;

list\_node<T>\* \_prev;

// 节点的构造函数

list\_node(const T& x = T())

:\_data(x)

, \_next(nullptr)

, \_prev(nullptr)

{

}

};

template<class T, class Ref, class Ptr>

struct \_\_list\_iterator

{

typedef list\_node<T> Node;

typedef \_\_list\_iterator<T, Ref, Ptr> iterator;

typedef T value\_type;

typedef Ptr pointer;

typedef Ref reference;

Node\* \_node;

// 迭代器的构造函数

\_\_list\_iterator(Node\* node)

:\_node(node)

{

}

// 重载!=

bool operator!=(const iterator& it) const

{

return \_node != it.\_node;

}

// 重载==

bool operator==(const iterator& it) const

{

return \_node == it.\_node;

}

// 重载解引用操作符

Ref operator\*()

{

return \_node->\_data;

}

// 重载->操作符

Ptr operator->()

{

return &(operator\*());

}

// 重载+操作符

iterator operator+(int pos)

{

Node\* tmp = this->\_node;

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

{

tmp = tmp->\_next;

}

iterator ret(tmp);

return ret;

}

// 重载前置++操作符

iterator& operator++()

{

\_node = \_node->\_next;

return \*this;

}

// 重载后置++操作符

iterator operator++(int)

{

iterator tmp(\*this);

\_node = \_node->\_next;

return tmp;

}

// 重载前置--操作符

iterator& operator--()

{

\_node = \_node->\_prev;

return \*this;

}

// 重载后置--操作符

iterator operator--(int)

{

iterator tmp(\*this);

\_node = \_node->\_prev;

return tmp;

}

};

template<class T>

class List

{

typedef list\_node<T> Node;

public:

typedef \_\_list\_iterator<T, T&, T\*> iterator;

typedef \_\_list\_iterator<T, const T&, const T\*> const\_iterator;

typedef \_\_reverse\_iterator<iterator, T&, T\*> reverse\_iterator;

typedef \_\_reverse\_iterator<const\_iterator, const T&, const T\*> const\_reverse\_iterator;

// const迭代器的头

const\_iterator begin() const

{

return const\_iterator(\_head->\_next);

}

// const迭代器的尾

const\_iterator end() const

{

return const\_iterator(\_head);

}

// 迭代器的头

iterator begin()

{

return iterator(\_head->\_next);

}

// 迭代器的尾

iterator end()

{

return iterator(\_head);

}

// 反向迭代器的头

reverse\_iterator rbegin()

{

return reverse\_iterator(end());

}

// const反向迭代器的头

const\_reverse\_iterator rbegin() const

{

return const\_reverse\_iterator(end());

}

// 反向迭代器的尾

reverse\_iterator rend()

{

return reverse\_iterator(begin());

}

// const反向迭代器的尾

const\_reverse\_iterator rend() const

{

return const\_reverse\_iterator(begin());

}

// 容器的第一个元素

T& first()

{

return \*begin();

}

// 链表的构造函数

List()

{

\_head = new Node;

\_head->\_next = \_head;

\_head->\_prev = \_head;

}

// 链表的拷贝构造函数

List(const List<T>& lt)

{

\_head = new Node();

\_head->\_next = \_head;

\_head->\_prev = \_head;

for (auto e : lt)

{

push\_back(e);

}

}

// 链表的赋值运算符重载函数

List<T>& operator=(List<T> lt)

{

swap(lt);

return \*this;

}

// 析构函数

~List()

{

clear();

delete \_head;

\_head = nullptr;

}

// 清理函数

void clear()

{

iterator it = begin();

while (it != end())

{

it = erase(it);

}

}

// 容器的大小

size\_t size() const

{

size\_t sz = 0;

auto it = begin();

while (it != end())

{

sz++;

it++;

}

return sz;

}

// 尾插

void push\_back(const T& x)

{

insert(end(), x);

}

// 头插

void push\_front(const T& x)

{

insert(begin(), x);

}

// 任意位置的插入函数

iterator insert(iterator pos, const T& x)

{

Node\* cur = pos.\_node;

Node\* prev = cur->\_prev;

Node\* newnode = new Node(x);

prev->\_next = newnode;

newnode->\_prev = prev;

newnode->\_next = cur;

cur->\_prev = newnode;

return iterator(newnode);

}

// 尾删

void pop\_back()

{

erase(--end());

}

// 头删

void pop\_front()

{

erase(begin());

}

// 任意位置的删除函数

iterator erase(iterator pos)

{

assert(pos != end());

Node\* cur = pos.\_node;

Node\* prev = cur->\_prev;

Node\* next = cur->\_next;

prev->\_next = next;

next->\_prev = prev;

delete cur;

return iterator(next);

}

// 交换函数

void swap(List<T>& lt)

{

std::swap(\_head, lt.\_head);

}

private:

Node\* \_head;

};

**6.14 Manager.h**

#pragma once

#include "Bignum.hpp"

#include <string>

#include <vector>

#include "Vector.hpp"

#include <fstream>

#include <iostream>

struct Expression // 表达式结构体

{

Bignum<List<int>> \_left; // 左操作数

Bignum<List<int>> \_right; // 右操作数

Bignum<List<int>> \_result; // 结果

char \_sign; // 操作符

};

class Manager

{

public:

bool read(const std::string& fileName); // 读取文件到数据

void write(const std::string& fileName, string wstr) const; // 写入数据到文件

void addBignum(const Expression& bignum); // 添加大整数

const Expression& getBignums(const int& index) const; // 获取大整数数据

void setBignum(const int& index, const Expression& bignum); // 修改大整数

void deleteBignum(const int& index); // 删除大整数

const size\_t size() const; // 大整数数量

Bignum<List<int>> string2Bignum(string str); // string转大整数

private:

Vector<Expression> \_bignums; // 大整数数组

};

**6.15 Manager.cpp**

#include "Manager.h"

#include <algorithm>

bool Manager::read(const std::string& fileName)

{

string arr[100];

ifstream infile(fileName, ios::in | ios::\_Nocreate);

int i = 0;

while (infile >> arr[i])

{

string left;

string right;

char sign;

for (auto e : arr[i])

{

if (e > '9' || e < '0')

{

sign = e;

left = string(arr[i].begin(), arr[i].begin() + arr[i].find(sign));

right = string(arr[i].begin() + arr[i].find(sign)+1, arr[i].end());

Expression temp;

temp.\_left = string2Bignum(left);

temp.\_right = string2Bignum(right);

temp.\_sign = sign;

if (temp.\_sign == '+')

{

temp.\_result = temp.\_left + temp.\_right;

}

else if (temp.\_sign == '-')

{

temp.\_result = temp.\_left - temp.\_right;

}

else if (temp.\_sign == '\*')

{

temp.\_result = temp.\_left \* temp.\_right;

}

else if (temp.\_sign == '/')

{

temp.\_result = temp.\_left / temp.\_right;

}

else if (temp.\_sign == '%')

{

temp.\_result = temp.\_left % temp.\_right;

}

else if (temp.\_sign == '^')

{

temp.\_result = temp.\_left ^ temp.\_right;

}

\_bignums.push\_back(temp);

}

}

++i;

}

infile.close();

return true;

}

void Manager::write(const std::string& fileName, string wstr) const

{

ofstream outfile(fileName, ios::out | ios::\_Nocreate | ios::app);

// 写入数据

outfile << wstr.c\_str() << endl;

outfile.close();

}

void Manager::setBignum(const int& index, const Expression& bignum)

{

\_bignums[index] = bignum;

}

void Manager::deleteBignum(const int& index)

{

\_bignums.erase(\_bignums.end() + index);

}

void Manager::addBignum(const Expression& bignum)

{

\_bignums.push\_back(bignum);

}

const Expression& Manager::getBignums(const int& index) const

{

return \_bignums[index];

}

const size\_t Manager::size() const

{

return \_bignums.size();

}

Bignum<List<int>> Manager::string2Bignum(string str)

{

Bignum<List<int>> ret;

for (auto e : str)

{

ret.push\_back(e - '0');

}

return ret;

}

**6.16 Algorithm.hpp**

#pragma once

template <class BidirectionalIterator>

void Reverse(BidirectionalIterator first, BidirectionalIterator last)

{

while ((first != last) && (first != --last))

{

std::iter\_swap(first, last);

++first;

}

}