# 西安微易码巨大数技术文档

目录

[西安微易码巨大数技术文档 1](#_Toc495075719)

[一、 项目概述 2](#_Toc495075720)

[1.1项目背景 2](#_Toc495075721)

[1.2项目核心技术 2](#_Toc495075722)

[1.2.1一万进制 2](#_Toc495075723)

[1.1.2微易码补码 4](#_Toc495075724)

[1.3开发环境 8](#_Toc495075725)

[1.3.1开发工具 8](#_Toc495075726)

[1.3.2操作系统 8](#_Toc495075727)

[1.3.3开发语言 8](#_Toc495075728)

[二、项目功能 8](#_Toc495075729)

[2.1提供对存储在文件中的巨大数的读取功能； 8](#_Toc495075730)

[2.2提供对巨大数的显示和输出功能； 8](#_Toc495075731)

[2.3提供将巨大数写入文件中的功能； 8](#_Toc495075732)

[2.4提供对两个巨大数进行四则运算的功能； 8](#_Toc495075733)

[三、工具使用说明 9](#_Toc495075734)

[3.1提供函数 9](#_Toc495075735)

[四、操作方法说明 9](#_Toc495075736)

[4.1准备巨大数 9](#_Toc495075737)

[4.2巨大数运算 9](#_Toc495075738)

[4.2.1加减法的实现 9](#_Toc495075739)

[4.2.2乘法的实现 9](#_Toc495075740)

[4.3巨大数的显示 9](#_Toc495075741)

## 项目概述

### 1.1项目背景

当前计算机对于数据的处理是存在一定限度的，用与计算处理各种数据，c语言提供了许多整数类型，一般情况系使用int类型即可，但是要满足特定任务和机器要求的时候，数据超出处理范围时就需要使用另外的方法对这些数据进行处理，虽然float类型（可表示的数据范围是）和double类型（可表示的数据范围-）也可以使用，但是float的有效位数只有7位，double类型的有效位数也只有15位，处理数据不够精确，数据的位数越多，丢失的信息量也越大。因此我们提供一种对位数超出表示范围的数据的处理工具—也就是这个巨大数的处理工具。

### 1.2项目核心技术

该项目的核心技术有两个，一个是一万进制，另一个是微易码补码；

#### 1.2.1一万进制

对于数据的处理，是要在数据存储的基础上进行的，也就是说想要解决巨大数问题，首先要处理巨大数的存储和表示；根据项目背景的分析可知对于数字的存储首先想到的就是使用字符串的方式，但是使用这种方法的弊端就是存在很多次的类型转换，所以在处理时我们采用的是万进制。

万进制的原理同二进制十进制是一样的，二进制是逢二进一，同理万进制就是逢万进一。

例如：（p = 10000）；

I.195698445564

该数据可表示为：1956×P² + 9844×P¹ + 5564×Pº；

II.156023001589728

该数据可表示为：156×P³ + 230×P² + 158×P¹ + 9728×Pº；

III.10002300005

该数据可表示为：100×P² + 230×P¹ + 5×Pº；

##### 1.2.1.1万进制的优点

1> **存储效率高**

将巨大数当作字符串处理时，会使用字符型数组来存储它，也就是说将巨大数中的每一位数字视作一个字符存储进数组，每个字节存储一个数字字符；若是使用一万进制的话，每次将存储四个字节，数组中的每个元素取值为0~9999，和使用字符串的效率相同。

2> **计算效率高**

同样若将巨大数当作字符串处理，在每次运算的过程中都要进行一次类型转换，每一次也只能处理一位数字，处理的效率极低，直接导致运算的效率低下，而在万进制的情况下，每次可同时处理四位数字，数组中存储的都是整形量数字，也不需要进行类型转换，大大地提高了效率。

3> **进位时位数处理合适**

之所以使用万进制，而不用更高的进制位，是因为在涉及乘法的计算中依然要是数据保持在int类型可以表示的范围之内，倘若使用更高的进位，那么在计算的过程中就会超出表示范围，而万进制则是处理该情况合适进制（例：若使用十万进制，当出现99999×99999 = 9999800001，但是很明显9999800001已经超出了int类型的表示范围，因此一万进制是最合适的进位）。

在进行加法运算时，对应位置进行加和之后都会有进位的可能，为保证所有数据都不遗失，设置进位的问题：在本位出存储的数据是该数字本身对10000取余的结果，再进位处存储该数字本身对10000取整的结果。

例如：15789—>存储在本位的是 5789，存储在进位的是 0001；

#### 1.1.2微易码补码

在巨大数处理过程中，为了避免对负数操作过程中引起的复杂运算，引用到微易码补码，与反码的原理相似（反码表示法规定：正数的反码与其原码相同；负数的反码是对其原码逐位取反，但符号位除外）而对于微易码补码，则是将负数的每一位与9取反，同样符号位是除外的，同时将符号位也可以参与到数值的运算中。

x≥0 -> 0: x

x<0 -> 1:(9999 ……9999 - |x|)

1表示负号，0表示正号；

正数： 25689156484984 负数：-25689156484984

微易码补码：0：25 6891 5648 4984 微易码补码：1：9974 3218 4351 5415

微易码补码：1：（9999-25）（9999-6891）（9999-5648）（9999-4984）

**经过推导有如下规律：**

**微易码补码转换方式**

举例如下：

|  |  |  |
| --- | --- | --- |
| **一、无进位类型** | | |
| a+b | 36 + 325 | 11 - 23 |
| a的微易码补码 | 0 ：0036 | 0 ：0011 |
| b的微易码补码 | 0 ：0325 | 1 ：9976 |
| 进位 | 0 | 0 |
| 运算结果 | X ：0361 | X ：9987 |
| 正确结果 | 0 ：0361 | 1 ：9987 |
| 结果值 | 361 | -12 |

由上表可看出在计算过程中，主要的问题就是符号位的确定，经过多次的推导又有如下结论：符号位的确定用两个数的符号位和进位按位异或，如上述例子中：

36+325 符号位就是0^0^0 = 0；

11-23 符号位就是0^1^0 = 1；

|  |  |  |
| --- | --- | --- |
| **二、有进位** | | |
| a+b | -25 - 26 | 846 - 198 |
| a的微易码补码 | 1 ：9974 | 0：0846 |
| b的微易码补码 | 1 ：9973 | 1：9801 |
| 进位 | 1 | 1 |
| 符号位 | 1^1^1 = 1 | 0^1^1= 0 |
| 运算结果 | 1 ：19947(-52) | 0 ：10647（-9352） |
| 正确结果 | -51 | 648 |
| 结果值 | 错误 | 错误 |

由上表可得，在有进位时，微易码补码在现有的设定情况下一下会出现错误，但是若是将进位端的结果加到最后运算的值上，即对9947再加上进位就是9948，转换过后就是-51，将这个规律运用到无进位的计算中，也是正确的。如果这样计算的话，就忽略了一种重要的情况将会出现过饱和（在部分运算时，会出现有进位时将进位加到最后结果上的时候，结果反而出现错误，反而按照之前的无进位计算方法得到的答案是正确的）。

过饱和的情况如下：

|  |  |  |
| --- | --- | --- |
| a+b | 8000+3000 | 9999+1111 |
| a的微易码补码 | 0：8000 | 0：9999 |
| b的微易码补码 | 0：3000 | 0：1111 |
| 进位 | 1 | 1 |
| 符号位 | 0^0^1 = 1 | 0^0^1 = 1 |
| 运算结果 | 1：11000 | 1：11110 |
| 正确结果 | 0：11000 | 0：11110 |
| 结果值 | 错误 | 错误 |

在这种情况下，如果把进位加到最后的结果上，就会出现错误。所以，就要将它视作无进位，实现这种方式的方法就是在原本数据的前面加上辅助位：正数前面补上0000，负数前面补上9999。

举例如下：

|  |  |  |
| --- | --- | --- |
| a+b | 4948 - 2645 | 10000-9999 |
| a的微易码补码 | 0 ：0000 4948 | 0 ：0000 0001 0000 |
| b的微易码补码 | 1 ：9999 7354 | 1 ：9999 9999 0000 |
| 结果的微易码补码 | 0 ：10000 2302 | 0 ：10000 0000 0000 |
| 进位 | 1 | 1 |
| 符号位 | 0^1^1 = 0 | 0^1^1 = 0 |
| 最终的微易码补码 | 0 ：0000 2303 | 0 ：0000 0000 0001 |
| 正确的微易码补码 | 0 ：0000 2303 | 0 ：0000 0000 0001 |
| 结果正确数值 | 2303 | 1 |

上述微易码补码的方法及简述，都是构思者的经验所得，由于数学理论功底的不足，无法给出具体的数学理论证明，希望有高手可以指点一二。

综上所述，可以给出设置存储巨大数结构体：

typedef struct HUGE\_NUMBER{

char symbol； //存储从文件中读取到的巨大数的符号

int \*data； //存储巨大数的数字部分

int dataCount；//存储巨大数数字部分数组元素的个数

int power； //存储小数权重

} HUGE\_NUMBER；

例：

巨大数：-152 ,4565 ,7812

实际存储：2187, 5434, 9847, 9999

Symbol：**-**

dataCount：4；

dataCount的算法是：

（11（数字的个数） + 3 ） / 4 + 1；

例：

巨大数：152 ,4565 ,7812

实际存储：7812， 4565， 152， 0

Symbol：**+**

dataCount：4；

dataCount的算法是：

（11（数字的个数） + 3 ） / 4 + 1；

数据存储情况如下：

打开存储巨大数的文件之后，对读到的首个字符进行判断，再将其结果存储到定义的结构体的Symbol成员中，在读取整个文件的长度，并对文件中的数据进行二次判断（是否是小数），运用dataCount的算法得到数组元素的个数，申请空间；将数字部分四个一组的存放到数组中（从高位开始存储），如果是小数的话，将小数点在文件中的位置存放在power处。

### 1.3开发环境

#### 1.3.1开发工具

DEV-C++；

#### 1.3.2操作系统

Windows 10；

#### 1.3.3开发语言

C语言；

## 二、项目功能

### 2.1提供对存储在文件中的巨大数的读取功能；

对存储在文件中的数据进行读取，并将读取到的所有数据都存储到文件头部定义的结构体成员中，作为支持以后运算的基础；

### 2.2提供对巨大数的显示和输出功能；

在对巨大数进行操作之后，将得到的新的巨大数（区分整数和小数）显示在屏幕上；

### 2.3提供将巨大数写入文件中的功能；

在得到的新的巨大数显示在屏幕上的同时，将得到的答案存入一个文件；

### 2.4提供对两个巨大数进行四则运算的功能；

在有了巨大数的基础上，对读取到的巨大数进行四则运算；

## 三、工具使用说明

### 3.1提供函数

## 四、操作方法说明

### 4.1准备巨大数

在.c文件的同级目录下建立一个文本文件，用来存储巨大数，然后根据所存储的巨大数的类型选择调用巨大数的读取函数（判断文件中的数据是小数，负数还是正数，再根据具体情况对数据进行分类处理），在读取文件的过程中将数据的部分存储到文件头部定义的HUGE\_NUM结构体中，将该结构体视作一个巨大数，在之后的运算中直接对结构体实例的内容进行更改即可。

### 4.2巨大数运算（具体代码实现）

#### 4.2.1加减法的实现

1.调用initArray（huge\_number \*num， FILE \*fp）函数Udine巨大数结构体成员进行初始化；

2.调用readArray（huge\_number \*num， FILE \*fp）函数，按照上文中描述的读取思路以及一万进制的存储方式将文件中的巨大数存储在巨大数数组中；

3.调用PreArray（huge\_number \*num1， huge\_number \*num2）函数按照上文的分析过程将巨大数转化为其微易码补码的形式；

4.调用AddHugeNumber（huge\_number \*num1， huge\_number \*num2， huge\_number \*result）将已经读取并且存储的两个目标巨大数的对应位相加，然后根据进位结果判断是否需要增加空间，调用ResultNumber（huge\_number \*result）函数申请新的空间，将结果的巨大数数值存储到新的数组中；

5.减法收下你调用Opposite（huge\_number \*num）对减数求相反数，然后在使用加法的方法来计算即可（减一个数即加上它的相反数）；

6.调用WriteToFile（huge\_number \*num， FILE \*resultFile）函数将计算的数值结果写入文件。

#### 4.2.2乘法的实现

乘法的手工过程如下：

乘数1：

0 1 2 3 4 5 …… n-2 n-1 n

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| hug[0] | hug[1] | hug[2] | hug[3] | hug[4] | hug[5] | …… | hug[n-2] | hug[n-1] | hug[n] |

乘数2：

0 1 2 3 4 5 …… n-2 n-1 n

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| hug2[0] | hug2[1] | hug2[2] | hug2[3] | hug2[4] | hug2[5] | …… | hug2[n-2] | hug2[n-1] | hug2[n] |

结果：

0 1 2 3 4 5 6 …… n-2 n-1 n n+1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| res[0] | res[1] | res[2] | res[3] | res[4] | res[5] | res[6] | …… | res[n-2] | res[n-1] | res[n] | res[n+1] |

乘数1 =

按照上述步骤，准备两个巨大数然后调用所需要的运算函数，运算结束时将得到的答案存储到准备好的结构体的实例中。

### 4.3巨大数的显示

根据运算后的结果进行判断（小数，整数和负数），然后调用不同的显示函数，对结果的数据进行显示。