CS205 · C/C++ Programming

Project1 Report: A "Simple" Calculator

赖海斌
12211612

摘要：在本次 Project 中，我初步用 C 实现了一个简单的计算器，可以简单地完成项目需求。在计算器的输入有效数字小于 8 位时，我阅读了 **IEEE754 标准**，使用 double 的加减法进行运算。在计算器输入数字大于 8 位时，我**借鉴了 Postgresql 数据库**中 numeric 的实现方法，使用**数组**进行精确计算。我注意到**内存泄露**的问题，并构建了一个内存控制检测系统，可以减少我的内存泄露。另外，我学习了解了 **OpenMP 与 CUDA** 编程上，各自写了一个乘法方法，可惜由于身体原因没能测试性能。除此之外，我了解了大整数乘法等计算器在现代的应用，比如在密码学中的加密，科学计算等。我也了解了算法，比如 **Karatsuba** 乘法算法。

关键词：IEEE754 Standard; Postgresql Source Code; Memory Leak; OpenMP; CUDA;

Contents

Part 1: 需求分析 2
1.1 Project 需求分析： 2
1.2 想学的知识需求分析： 3
Part 2: 环境配置/编译 3
2.1 环境及相关库配置 3
2.2 编译配置 5
Part 3: 功能演示 5
Part 4: 需求实现：路上的困难 8
4.1 输入 x 还是 * ? 9
4.2 Mode2 的输入 10
4.3 IEEE754: 深入了解浮点数 10
4.4 如何精确打印浮点数? 12
4.5 Numeric Implementation: 仿 Postgresql 的 numeric.c 实现的大数计算器 14
4.6 Karatsuba 大整数乘法探究 17
Part 5: 内存泄露：Java 最高兴的一集 18
Part 6: 并行计算 21
6.1 应用 OpenMP 21
6.2 GPU 编程：将浮点计算移植到 CUDA 架构 22
6.3 性能测试 23
Part 7: 致敬传奇 23
7.1 快速平方根算法背后的秘密 23
Part 8: 库的学习 24
8.1 大整数乘法 GMP 数学库 24
8.2 与 PBC 密码学库 25
Part 9: 结语 25
Appendix 25
Acknowledgement 25

Part 1: 需求分析

经过上学期数据库（H）的腥风血雨，这学期 C/C++又来八级狂风。但我一直相信，风是让羽翼丰满的。因此需求分析分为：Project 的需求分析，**我想学的知识的需求分析**。

1.1 Project 需求分析:

- 基本需求是处理输入字符串，加减乘除，大数运算，和指数运算。初看会觉得很简单，但是其实里边涵盖了 C 中的字符，数据类型的了解，要应用条件与分支语句，初步探索指针和 C 特有的内存泄露。除此之外，还有一些算法设计。
- 本质：了解 C 的基本语法，C 中对字符串及各种数据类型的处理，学习 C 指针，学习用 C 写简单的算法。还有就是了解精确计算、科学计算以及大整数乘法的应用，这些让我去了解了数值计算领域。

1.2 想学的知识需求分析：

李卓钊导师希望我能学习 C++跟 CUDA 编程，充分吸收这方面的知识，在大三做创新实践能够有所应用。另外这学期超算比赛进了决赛，我还想多学一点 OpenMP 的知识。所以，在本次的计算器中我初步**尝试 CUDA 和 OpenMP** 的使用，并且移植到 GPU 上进行测试。

Part 2: 环境配置/编译

2.1 环境及相关库配置

本次 Project 使用的编译及测试环境如下：

硬件使用南科大科学与工程计算中心的计算工作站一台，拥有 2CPU 128 核，总内存约为 500GB。工作站共装配三台 NVIDIA A100 GPU。详见表 2.1，表 2.2。

CPU Architecture	x86_64
Model name	AMD EPYC 7773X 64-Core Processor
CPU family	25
Thread(s) per core	1
CPU(s)	128
Core(s) per socket	64
CPU max MHz	3527.7339
CPU min MHz	1500.0000

表 2.1 硬件 CPU 环境

GPU name	A100-SXM4-80GB
GPU amount	3
CUDA Version	12.4
Driver Version	550.54.14
Memory	98304MiB

表 2.2 硬件 GPU 环境

服务器内操作系统为最新版 Ubuntu 23.04，gcc 编译器版本为 12.3.0，CUDA 版本为 12.4，CUDA 编译器 nvcc 版本 11.8。详细软件配置见表 2.3。

Software	Version
Operating System	Ubuntu 23.04
OS Kernel	Linux H3C1 6.2.0-39-generic
gcc	(Ubuntu 12.3.0-1ubuntu1~23.04) 12.3.0
nvcc: NVIDIA (R) Cuda compiler driver	release 11.8, V11.8.89
Vim	Vi IMproved 9.0 (2022 Jun 28)
VScode Server	1.87.1
Math Library	ISO C99
btop	V1.2.13
nvitop	1.3.2

表 2.3 软件环境

本次 Project 使用了以下几个 C 语言库，下表 2.4 展示了他们的版本和用途。

Library	Version	Usage
stdio.h	跟随 gcc	使用标准输入输出函数
stdlib.h	跟随 gcc	使用 malloc,free 函数
string.h	跟随 gcc	判断字符串长度，比较
unistd.h	跟随 gcc	打开本地 txt 文件进行计算
math.h	跟随 gcc	-lm 调用本地 C99 数学库。仅作为计算器计算指数使用
omp.h	跟随 gcc	使用 OpenMP 实现乘法的并行计算
gmp.h	6.3.0	快速计算大整数乘法库。在本次 Project 中作为与我写的乘法库的比较，也可作为计算器的辅助功能。
sodium.h	1.0.18	一个大数计算、随机数生成及多种加解密算法生成库。在本次 Project 中仅作为我学习大数乘除法在密码学中的应用，对计算器没有过多作用。
pbc.h	0.5.14	基于 GMP 数学库的密码学库，在本次 Project 中仅作为全同态加密方面的了解学习而使用，对计算器没有过多作用。

表 2.4 相关库配置

这几个额外安装的库只是作为测试使用，当然也可以作为计算器的一个插件。具体安装都比较简单：

在 Linux 中安装 GMP 库可以使用如下命令（测试计算器性能用，可有可无）：

```
wget https://gmplib.org/download/gmp/gmp-6.3.0.tar.xz
tar -xvzf gmp-6.3.0.tar.xz
cd gmp-6.3.0
./configure
make                # 编译
make check          # 检验
sudo make install   # 安装
```

在 Linux 中安装 Sodium 库（仅作为我学习，老师不必安装）：

```
wget https://download.libsodium.org/libsodium/releases/libsodium-1.0.18.tar.gz
tar -xvzf libsodium-1.0.18.tar.gz
cd libsodium-1.0.18
./configure
make                # 编译
make check          # 检验
sudo make install   # 安装
```

在 Linux 中安装 PBC 库（仅作为我学习，老师不必安装）：

```
wget https://crypto.stanford.edu/pbc/files/pbc-0.5.14.tar.gz
tar -xvzf pbc-0.5.14.tar.gz
cd pbc-0.5.14
./configure
make                # 编译
make check          # 检验
sudo make install   # 安装
```

```
cd /etc/ld.so.conf.d
sudo vi libpbc.conf # 编译好 pbc 库后，加入 libpbc.so.1 的路径
cd /etc/ld.so.conf.d && vim libpbc.conf  # 将 /usr/local/lib 输入到 libpbc.conf 文件中
sudo ldconfig # 更新 cache
```

遇到库无法安装的问题，可能是缺少相应的软件，尝试检查 g++,m4,flex 等软件。

2.2 编译配置

注意：由于我是在 linux 上进行开发和编译，虽然后来成功移植到我的 Windows 电脑上，如果编译出现问题，可能的原因是我的计算器中使用的函数在 Windows MinGW 上可能不是标准 C 库的一部分。

计算器标准版编译：

```
gcc src/haibin_calculator.c -o haibinCalculator  -lm
```

全部使用了我自己写的功能，可以做到普通加减乘除，大整数加减法和乘法，指数运算。

更多编译选项：

```
-DUSE_LONG_INPUT      使用 getline 读取输入，这样输入可以更长
-lgmp                 使用 GMP 大整数库
-DACTIVATE_MESSAGE    编译时启动消息
-lm                   启动数学库，用于指数运算 pow 函数
-fopenmp              启动 OpenMP 多线程
```

Part 3: 功能演示

1. 小数字加减乘除

当输入的两个数字有效位数小于 8 位时，此时计算器会将数字转换为 double 进行运算。Double 的精度在 8.85×10^{-15}，满足我们的精度需求。同时，在计算除法时，除不尽的结果将保留 8 位小数。

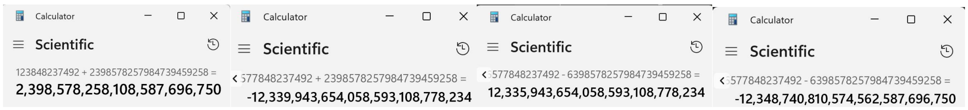
```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 2 + 3
2 + 3 = 5
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 2 - 3
2 - 3 = -1
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 2 x 3
2 x 3 = 6
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 2 / 3
2 / 3 = 0.66666667
```

图 3.1 基本加减乘除

2. 大整数加减法，乘法

当两个数字的有效位数超过 8 位且没有小数和指数 e 时，计算器将自动转换为大整数模式。计算器将数字转换为 char*并进行类似于数组的加减法，从而实现大整数的加减法和乘法。大整数除法的实现过于复杂，由于时间紧凑，我就花更多时间放在了自己更想探索的地方，没有实现除法。

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 123848237492 + 2398578257984739459258
123848237492 + 2398578257984739459258 = 2398578258108587696750
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator -12342342223165777848237492 + 2398578257984739459258
-12342342223165777848237492 + 2398578257984739459258 = -12339943654058593108778234
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 12342342223165777848237492 - 6398578257984739459258
12342342223165777848237492 - 6398578257984739459258 = 12335943654058593108778234
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator -12342342223165777848237492 - 6398578257984739459258
-12342342223165777848237492 - 6398578257984739459258 = -12348740810574562587696750
```




```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 987654321987654321987654321987654321987654321987654321
987654321987654321987654321 + 987654321987654321987654321987654321987654321987654321987654321987654321
987654321987654321987654321987654321987654321987654321987654321987654321987654321 + 987654321987654321987654321987
654321987654321987654321987654321 = 987654322975308643975308643975308643975308643975308643975308643975308642
```

[1]组图 3. 2 大整数加减及校验

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 987654321 x 987654321
987654321 x 987654321 = 975461057789971041
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 987654321 x -987654321
987654321 x -987654321 = -975461057789971041
```

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator 987654321987654321987654321987654321987654321987654321
987654321987654321987654321 x 987654321987654321987654321987654321987654321987654321987654321987654321
987654321987654321987654321987654321987654321987654321987654321987654321987654321 x 987654321987654321987654321987
654321987654321987654321987654321 = 9754610597408931595063252592717573590371894588026215585680536583334857571234567971479957383825636386171315388516994390862673393208352395554031397899710
41
```

组图 3. 3 大整数乘法

3. 指数运算

当两个数字都是指数时，将进行指数运算。对于指数乘法，在 e 前的参数采用 double 运算，在 e 后的参数使用 double 相加减。除法类似，核心基于 double 的除法。对于加减法，先将两个数字的指数大小保持一致，e 前的参数采用大整数加减法。

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./haibinCalculator
1e200 x 1e200
1e200 x 1e200 = 1.00e400
1e200 + 1e200
1e200 + 1e200 = 2.00e200
```

4. 提高性能：CUDA 和 OpenMP

我们的大整数乘法是用大整数加法和移位实现的，计算复杂度为 O(N · M), M,N 为两个数字的位数。幸运的是，我们可以把乘法过程并行化。我们的实际演示将在后面的。

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ ./GPUCalculator
Enter first number: 987654321
Enter second number: 987654321
Product: 975461057789971041
```

CUDA

```
(base) asc648@H3C1:~/haibin/Back_up/CPP_backup_2024031005/CPPLearning$ ./big
Enter first number: 123
Enter second number: 1
Product: 123
```

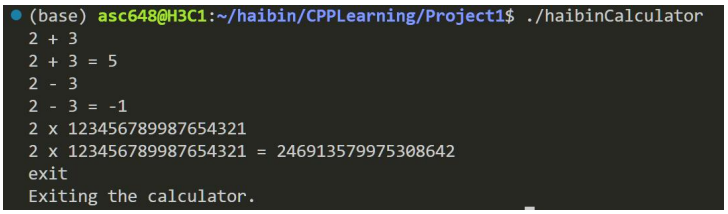
OpenMP

[1] 由于大整数加减实现时采用的是指针的形式，因此只要内存足够，就可以对超长位数进行加减。这里由于微软计算器已经超限，就没有验证数值。

5. Human Computer Interaction：人机交互

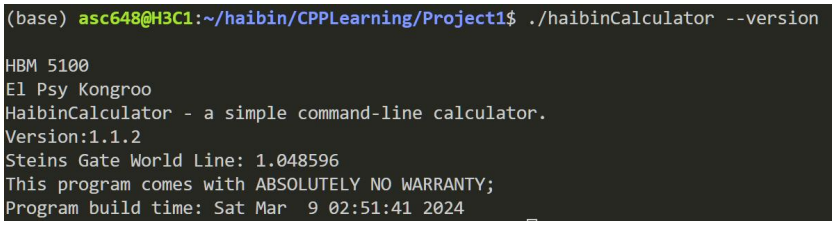
Mode2：等待输入模式

如果我们的计算器在调用时没有输入任何参数，计算器将进入另一种模式，它将等待用户的输入并进行计算。计算的效果与直接调用模式相同。

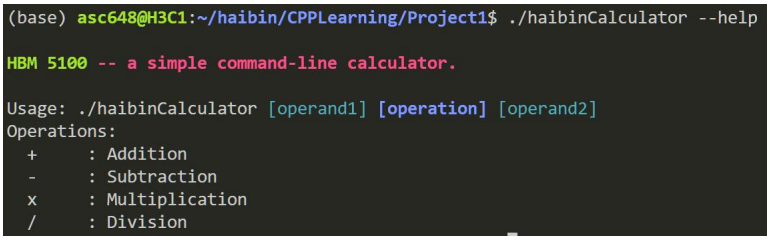


-v / --version

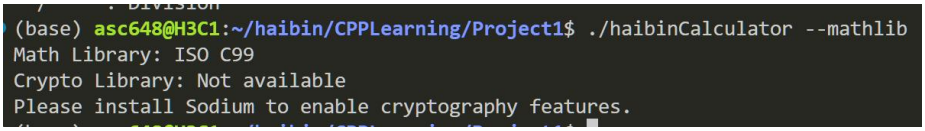
打印计算器版本，显示计算器编译时间。打印 El Psy Kongroo 和世界线常数（设定）。



-h / --help 打印五彩斑斓的帮助

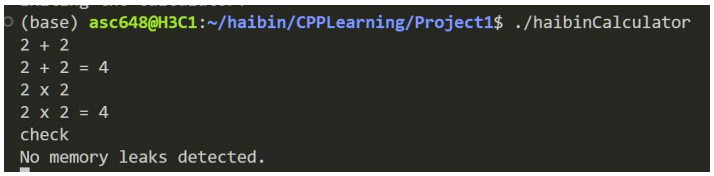


-l / --mathlib 打印计算器使用的数学库



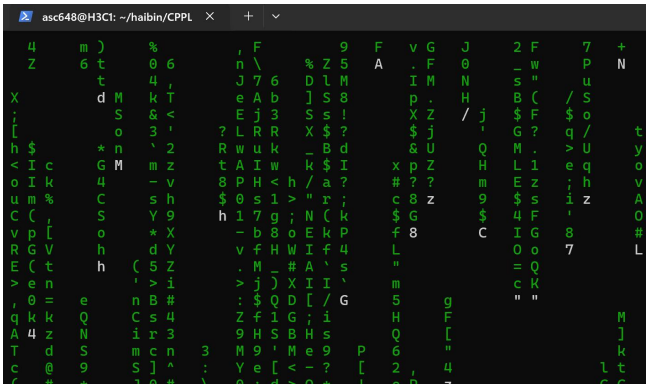
cls / clear 调用 system(“clear”) 清空屏幕

check 检测计算器内存泄露



cmatrix 调用一个好玩的软件包(sudo apt install cmatrix)，变成黑客帝国，ctrl-c 退出[2]。

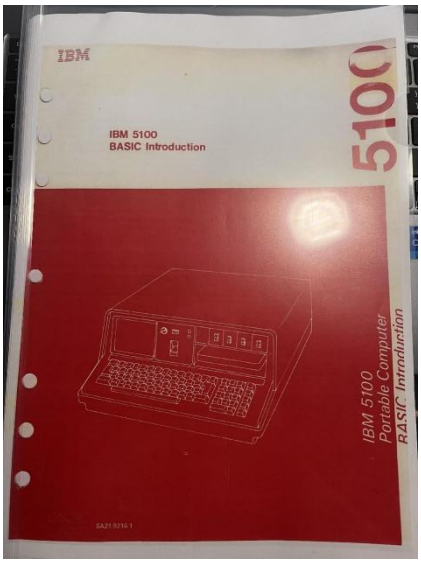
[2] 需要安装 cmatrix 软件包。做这个单纯是觉得计算器应该要点好玩的设定。

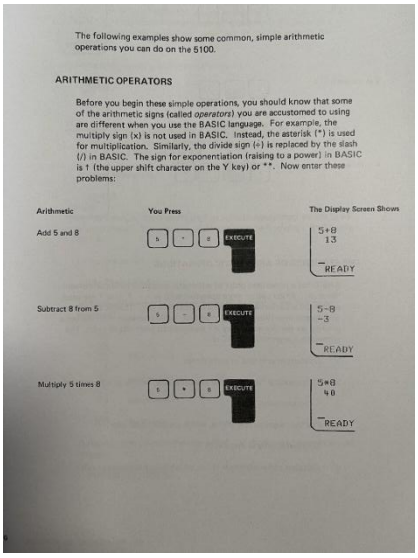


6. 背景设定（纯中二病，可以跳过）

在做 Project 时我刚好在看《命运石之门》，在这部剧里有一台重要的计算机：IBM5100。IBM5100 是世界上第一款移动式计算机，可以进行一些简单运算。曾经有个都市传说，在 2001 年一个叫 John Titor 的人声称自己来自 2036 年，那个时代人类的计算机因为千年虫问题而崩溃。而他穿越时空寻找 IBM5100，试图靠这台机器找寻崩溃的 bug 的秘密。

我很喜欢这个设定，并且，我刚好淘到了一本 IBM5100 的操作手册。上面也有计算指令。我决定复刻 5100 上的几个指令，于是就把我的计算器叫做：Hai Bin Machine 5100，简称 HBM5100。

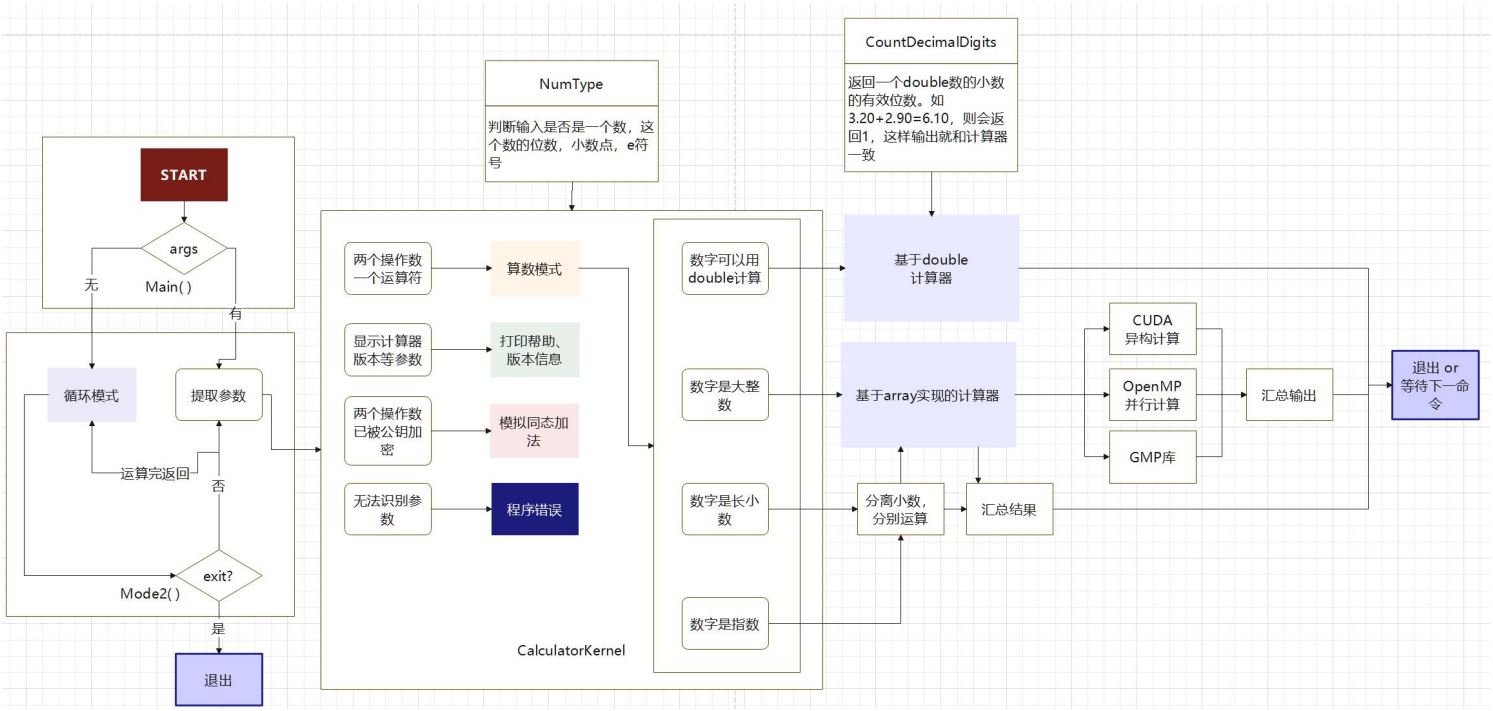




更为惊奇的是，在 Project 中的 Calculator bc 文档，是用 CERN 的 text2html 生成的，而 CERN，欧洲核子研究组织，正是《命运石之门》中的最大反派！因此，我们的这个设定天经地义，这一切都是命运石之门的选择！

Part 4: 需求实现：路上的困难

本次 Project 项目结构如下，从 main 函数启动后，根据参数判断模式，判断执行命令。如果是计算命令，就会凭借数字类型进行具体计算。



4.1 输入 x 还是 * ?

我的 Project 开始于对输入的处理。main 函数会根据空格分隔出几个字符串，随后计算器将判断输入的字符串数量。

```
1  int main(int argc, char **argv) {
2
3      switch (argc)
4      {
5      // Mode 1: input like 3 + 5
6      case 4:
7          char *operand1 = argv[1];
8          char *operator = argv[2];
9          char *operand2 = argv[3];
10         CalculatorKernel(operand1, operand2, operator);
```

图 4.1 main 输入判断

但是当我输入 3 * 5 时，我发现 argc 变量并不是我想的 4 个。我随后向肖翊成同学提问，他暑假看了 B 站上的 C++网课，但是他也不了解这件事。随后我们进行了测试，发现”*”号会读取当前文件夹下所有的文件作为字符串输入。原来，代表乘法的 ’*’ 号 在 linux 系统中会被识别为通配符。

```
(base) jaredan@Jaredan:~/SUSTech/CPP/projects/2023Spring/project1_impl$ ./calculator 3 * 5
argc = 5
argv[0] --> ./calculator
argv[1] --> 3
argv[2] --> calculator
argv[3] --> calculator.c
argv[4] --> 5
argv[argc] = (nil)
```

图 4.2 判断 * 输入

在经过讨论后，并且在老师的同意下，我认为使用 x 号相比使用*号然后做各种处理会更好：

第一，只有输入“3 ‘*’ 5”才能被专门识别为乘号，但是为了乘号多打两个单引号很麻烦。如果单纯忽略，只读取最前面的 3 和最后的 5，也感觉非常不妥。第二，这个问题跟安全有关：不同的用户有不同的目录，如果有人依靠我的计算器输入读取了他文件夹下的所有文件，说不定别人就能通过攻击我的计算器查看到他的文件。

我后来也去查找如何修改这一配置，我们可以去修改 shell 的配置文件，进入~/.bashrc 文件，并用以下命令禁止通配符展开。然而这样的修改变动太大，为了一个计算器得不偿失。因此最终我使用了”x”作为乘号使用。

```
1.      set -o noglob
```

4.2 Mode2 的输入

在 mode2 下，我们要等待用户输入，然后执行相应的计算指令。

```c
char *expression = NULL;  // 初始时将表达式指针设为 NULL
    size_t expression_size = 0;

    while (1) {
        // 使用 getline 动态获取用户输入的表达式
        ssize_t chars_read = getline(&expression, &expression_size, stdin);

```

我一开始用的是 getline 函数，这个函数在 stdio.h 中。但是，当我把这个函数移植到 Windows 的 minGW 时，它显示 Windows 上编译器没有这个函数。我随后用了 fget 函数。

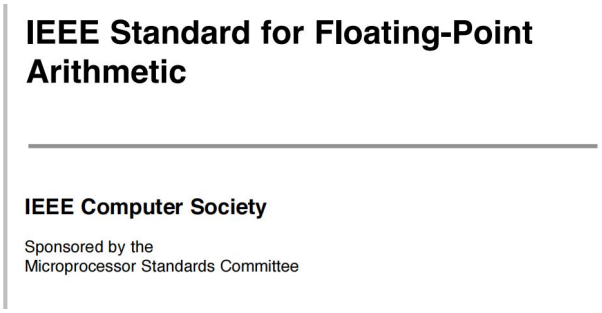
```c
#ifdef USE_LONG_INPUT
char *expression = NULL;  // 初始时将表达式指针设为 NULL
size_t expression_size = 0;
#else
int max_size = 2048;
// 声明一个字符串
char expression[max_size];
#endif

while (1) {

    #ifdef USE_LONG_INPUT
    // 使用 getline 动态获取用户输入的表达式
    size_t chars_read = getline(&expression, &expression_size, stdin);
    #else
    fgets(expression,max_size,stdin);
    #endif
```

采用 fget 函数的代价便是会有长度输入的限制，不过这也让我好奇不同函数他们是怎么处理输入的。

简书上有一篇博客 C 中读入一行字符串的方法 讲的挺好的，它对比了 scanf，gets 和 fget 三种函数的区别。scanf 是非常邪恶的，我们给他长度为 6 的 char 数组，然后输入 8 个 char，结果它会在数组后继续填写数字。也就是说，会覆盖我们的内存。而 gets 的原理和 scanf 相同，只有 fget 函数才能保证内存上的安全，并且能读 stdin 和文件。

4.3 IEEE754: 深入了解浮点数

我刚开始实现计算器时，发现 matlab 和我的 CASIO 计算器在正常情况下用的都是 double 进行计算。Double 真的够精确吗？抱着这个疑惑，我去阅读了 double 的标准：IEEE754[3]。



在 IEEE754 标准中，正如我们学习的那样，double 是使用 1 位表示正负

位	宽度	用法
63	1	存储符号，其中 0 表示正值，1 表示负值
62 到 52	11	存储指数，偏移量为 1023
51 到 0	52	存储尾数

在此之外，我还发现了一些关于浮点数里有趣的事情。比如浮点数里的 NaNs 是用 1111...11 表示的，当我们使用 0/0 时，会出现这个结果。我想跟除法的运算性质有关。不过，跟 Infinities 比起来，无限这个数的 N-1 bits of Significand 是 0。另一个有趣的点是浮点数是区分正零和负零的，跟我们在数字逻辑里的二进制表示整数的方法并不一致。

Encodings of $\pm 2^{k+1-N}\, n$ into Binary Fields :

Number Type	Sign Bit	K+1 bit Exponent	Nth bit	N-1 bits of Significand
NaNs:	?	binary 111...111	1	binary 1xxx...xxx
SNaNs:	?	binary 111...111	1	nonzero binary 0xxx...xxx
Infinities:	±	binary 111...111	1	0
Normals:	±	$k-1+2^K$	1	nonnegative $n - 2^{N-1} < 2^{N-1}$
Subnormals:	±	0	0	positive $n < 2^{N-1}$
Zeros:	±	0	0	0

[3]

这里还有几个有趣的运算，比如 1 的无穷次方不是 1 而最好是 NaN，但是 0 的 0 次方却是 1 而不是 NaN。为了引入这个诡异的 Not a Number，标准对待它就像数据库对待 NULL 一样谨慎。

Differences of opinion persist about whether certain functions should be INVALID or defined by convention at internal discontinuities; a few examples are ...

$1.0^{**}\infty = (-1.0)^{**}\infty = 1.0$?	(NaN is better.)
ATAN2(0.0, 0.0) = 0.0 or $+\pi$ or $-\pi$?	(NaN is worse.)
ATAN2($+\infty$, $+\infty$) = $\pi/4$?	(NaN is worse.)
SIGNUM(0.0) = 0.0 or +1.0 or -1.0 or NaN ?	(0.0 is best.)
SGN(0.0) = 0 .	(Standard BASIC)
SIGN(+0.0) = SIGN(-0.0) = +1.0 .	(Fortran Standard)
CopySign(1.0, $\pm$0.0) = $\pm$1.0 respectively.	(IEEE 754/854)

看过标准后，我们这里主要处理的，就是 0/0。在 8 位有效数字下，最大的两者相乘也是 16 位，我们就可以都用 double 表示。

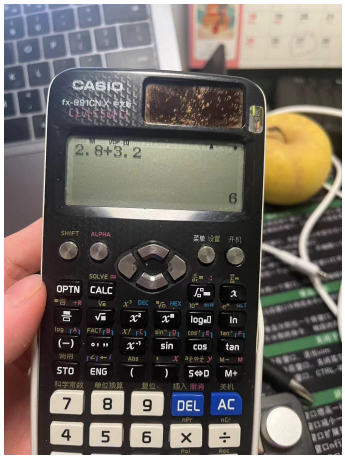
[3] 表格内容来自文章[4]。

```
1264        case '/':
1265        // dog shit!
1266        // 有效精度：8.8818e-16 (double 有效精度)
1267
1268            if(double_operand2 == 0){
1269                ErrorDivideByZero();
1270                return;
1271            }
1272
1273            result = double_operand1 / double_operand2;
1274
1275            int effect_num_of_result = CountDecimalDigits(result);
1276            int effect_num = Min(DOUBLE_DIVIDE_PRECISON,effect_num_of_result);
```

在此以后，面对小运算，我们就可以“偷个懒”，用计算机自带的 double 计算完成运算。但是，怎么打印出来，成为了我下一个问题。

4.4 如何精确打印浮点数?

是的，浮点数其实并不是很好打印。0.1+0.2 = 0.300000004，这本质上涉及到进制的问题。我们在 10 进制下十分之一是一个有限小数，但是 2 进制下十分之一和五分之一都是无限不循环小数，而用 double 相加后，自然会产生一个误差。但是，如果我直接打印，0.1+0.2=0.30000004 很明显是不对的。并且，如果我们直接使用 printf(“%s”,Number)，我们得到的是保留小数点后 6 位的 double 输出。比如 2.8+3.2 = 6，但是我们的浮点数计算器会输出 6.000000。



那么这应该怎么做呢？我设计了一个方法 CountDecimalDigits，使用三指针判断一个数的有效位数。这个方法将返回 double 的小数里最后一个不是 0 的数的位置。在处理过程中，我会将 double 临时转换成 char 数组，然后在数组内进行操作。使用了这个方法后，就可以打印浮点数了。

```
528  *@brief Count the decimal digits of a double number
529  */
530  int CountDecimalDigits(double num) {
531      int int_count = 0;
532      int isDecimal = 0;
533
534      if (num < 0) {
535          num *= -1;
536      }
537
538      int max_count = 0;
539
540      char result_in_char_format[64] = "";
541      sprintf(result_in_char_format,"%.16f", num);
542
543      //解法：三指针
544      // check num in deciaml strlen(result_in_char_format)
545      for (int i = 0; i < 18; i++) {
546          if(isDecimal){
547              if(result_in_char_format[i] != '0'){
548                  max_count = i;
549              }
550          }
551          if(result_in_char_format[i] == '.'){
552              isDecimal = 1;
553              int_count = i;
554              max_count = i;
555          }
556      }
557
558      return max_count-int_count;
559  }
```

不过，假如我们算出来的数字与我们的 double 表示方法不一致，printf 是如何快速转换和打印浮点数的呢？就比如说 2/3=0.66666667，假如保留不同的位数，输出的结尾都会四舍五入一个 7，这是怎么做到的呢？我去看了文章[5]《如何快速打印浮点数》，大概发现了一些答案。

Input: output base B and positive floating-point number $v = f \times b^e$ of precision $p > 0$

Output: $V = 0.d_1 d_2 \ldots d_n \times B^k$, where $d_1, \ldots, d_n$ are base-B digits and n is the smallest integer such that:

(1) $\frac{v^- + v}{2} < V < \frac{v + v^+}{2}$, *i.e.*, V would round to v when input, regardless of how the input rounding algorithm breaks ties, and

(2) $|V - v| \leq \frac{B^{k-n}}{2}$, *i.e.*, V is correctly rounded.

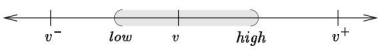
Procedure:

1. Determine v^- and v^+, the floating-point predecessor and successor of v, respectively.

$$v^- = \begin{cases} v - b^e & \text{if } e = \text{min. exp. or } f \neq b^{p-1} \\ v - b^{e-1} & \text{if } e > \text{min. exp. and } f = b^{p-1} \end{cases}$$

$$v^+ = v + b^e$$

Let $low = \frac{v^- + v}{2}$ and $high = \frac{v + v^+}{2}$. All numbers between low and $high$ round to v, regardless of how the input rounding algorithm breaks ties.



2. Find the smallest integer k such that $high \leq B^k$; *i.e.*, $k = \lceil \log_B high \rceil$. k is used to scale v appropriately.

3. Let $q_0 = \frac{v}{B^k}$. Generate digits as follows:

$$\begin{array}{ll} d_1 = \lfloor q_0 \times B \rfloor & q_1 = \{q_0 \times B\} \\ d_2 = \lfloor q_1 \times B \rfloor & q_2 = \{q_1 \times B\} \\ \vdots & \vdots \end{array}$$

4. Stop at the smallest n for which

(1) $0.d_1 \ldots d_n \times B^k > low$, *i.e.*, the number output at point n would round up to v, or

(2) $0.d_1 \ldots d_{n-1}[d_n+1]^1 \times B^k < high$, *i.e.*, incrementing digit d_n would make the number round down to v.

If condition (1) is true and condition (2) is false, return $0.d_1 \ldots d_n \times B^k$.

If condition (2) is true and condition (1) is false, return $0.d_1 \ldots d_{n-1}[d_n+1] \times B^k$.

If both conditions are true, return the number closer to v. If the two are equidistant from v, use some strategy to break the tie (*e.g.*, round up).

在这篇文章里作者提出了一个打印算法，它的操作有点像数学里的牛顿法，每一位都尽可能贪心地逐步逼近我们的真实值，直到用完 double 里的所有可表示位。文章还进行了有效性的数学证明与伪代码，我就看了最后的数学证明就溜了。

这项知识很有意思，因为在 java 的时候，我从没有考虑计算机的底层会怎么实现这些

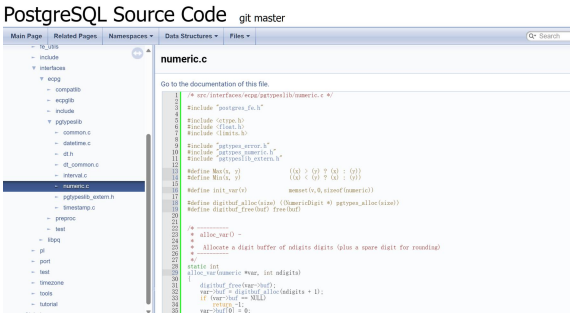
打印，我们所度过的每个平凡的算法，也许就是连续发生的奇迹。

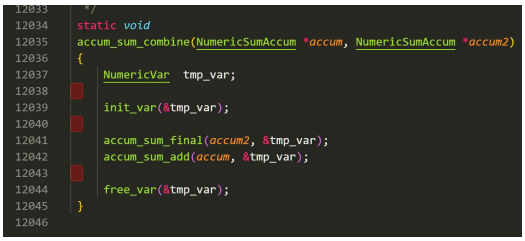
4.5 Numeric Implementation: 仿 Postgresql 的 numeric.c 实现的大数计算器

但是，使用 double 的话，987654321 x 987654321 是不可能算出精确结果的。为了计算这个大数，我必须采用新的方法。

但是新的是什么呢？我在构思是否可以用数组实现。然后这时，于老师提示我在数据库里，刚好有人实现了这样的精确大数计算。

于是，我去查找了 Postgresql 的 source code，专门去研究了它 numeric.c 的实现。结果啊，不看不知道，一看吓一跳，这 numeric.c 的加减乘除，还有类型转换，Postgresql 竟然写了有 12000 行！





```
12033   */
12034  static void
12035  accum_sum_combine(NumericSumAccum *accum, NumericSumAccum *accum2)
12036  {
12037      NumericVar  tmp_var;
12038
12039      init_var(&tmp_var);
12040
12041      accum_sum_final(accum2, &tmp_var);
12042      accum_sum_add(accum, &tmp_var);
12043
12044      free_var(&tmp_var);
12045  }
12046
```

我将代码复制了下来，顺便把零零散散各种定义，结构体，宏文件也复制下来，看了三天三夜.....单单是这一个类，我估计总代码量就已经超过 2 万行了。比起我自己的“计算器”，那是真的简单又幼稚啊，“朝菌不知晦朔，蟪蛄不知春秋”，可能说的就是如此吧。

但是幸运的是，我找到了 add_var 的实现。仔细研究后，我大概明白了它的工作原理。

```
8379  static void
8380  add_var(const NumericVar *var1, const NumericVar *var2, NumericVar *result)
8381  {
8382      /*
8383       * Decide on the signs of the two variables what to do
8384       */
8385      if (var1->sign == NUMERIC_POS)
8386      {
8387          if (var2->sign == NUMERIC_POS)
8388          {
8389              /*
8390               * Both are positive result = +(ABS(var1) + ABS(var2))
8391               */
8392              add_abs(var1, var2, result);
8393              result->sign = NUMERIC_POS;
8394          }
8395          else
8396          {
8397              /*
8398               * var1 is positive, var2 is negative Must compare absolute values
8399               */
8400              switch (cmp_abs(var1, var2))
8401              {
8402                  case 0:
8403                      /* ----------
8404                       * ABS(var1) == ABS(var2)
8405                       * result = ZERO
8406                       * ----------
8407                       */
```

在这里，Postgresql 首先先不管符号，实现了一个 add_abs 和 sub_abs,在这两个基础方法内考虑两个正数相加，以及大数剪小数。之后，add 和 sub 分别会考虑正负性，在输出时加上正负号。另外，在 Postgresql 内所有类型的传输都是依靠结构体进行完成。

```
11403     i1 = res_rscale + var1->weight + 1;
11404     i2 = res_rscale + var2->weight + 1;
11405     for (i = res_ndigits - 1; i >= 0; i--)
11406     {
11407         i1--;
11408         i2--;
11409         if (i1 >= 0 && i1 < var1ndigits)
11410             carry += var1digits[i1];
11411         if (i2 >= 0 && i2 < var2ndigits)
11412             carry += var2digits[i2];
11413
11414         if (carry >= NBASE)
11415         {
11416             res_digits[i] = carry - NBASE;
11417             carry = 1;
11418         }
11419         else
11420         {
11421             res_digits[i] = carry;
11422             carry = 0;
11423         }
11424     }
11425
```

Postgresql 里实现的 add_abs

```
681     for(i= maxlen-1; i>=0; i--){
682         int digit1 = 0;
683         if(ptr1 >= 0){
684             digit1 = number1[ptr1] - '0';
685             ptr1--;
686         }
687         int digit2 = 0;
688         if(ptr2 >= 0){
689             digit2 = number2[ptr2] - '0';
690             ptr2--;
691         }
692         int sum = digit1 + digit2 + carry;
693         result[i] = (sum % 10) + '0'; // 将个位数添加到结果字符串中
694         carry = sum / 10; // 计算进位
695     }
696
697     /*
698     * 血泪的教训：内存分配与泄露，上了C的第一课了属于是
699     */
700     if (carry > 0) {
701         for(int i=maxlen; i>=0; i--){
702             result[i+1] = result[i];
703         }
704         result[0] = carry + '0'; // 如果最后还有进位，加入结果中
705
706         return result;
707     }
```

我自己实现的 add_abs

因此，我依照 Postgresql 的方式，顺利完成了大整数加法和减法的设计。但是乘法怎么做呢？我继续去读了源代码。

具体的思想便是，它会将每个数字单独相乘，然后将结果像矩阵一样竖着相加。

$$\text{Let} \quad T = \begin{bmatrix} a[1]\ b[2] & & & & \\ b[2]\ a[2]\ b[3] & & & & \\ b[3]\ a[3]\ b[4] & & & & \\ b[4]\ a[4] & \dots & & & \\ \dots & \dots & b[n] & & \\ b[n]\ a[n] & & & & \end{bmatrix}$$

No b [j] = 0 .

Let q [1] := 0 and

$$q[j] := b[j]^2 > 0 \quad \text{for} \quad 1 \le j \le n .$$

```
8728     for (i1 = Min(var1ndigits - 1, res_ndigits - 3); i1 >= 0; i1--)
8729     {
8730         NumericDigit var1digit = var1digits[i1];
8731
8732         if (var1digit == 0)
8733             continue;
8734
8735         /* Time to normalize? */
8736         maxdig += var1digit;
8737         if (maxdig > (INT_MAX - INT_MAX / NBASE) / (NBASE - 1))
8738         {
8739             /* Yes, do it */
8740             carry = 0;
8741             for (i = res_ndigits - 1; i >= 0; i--)
8742             {
8743                 newdig = dig[i] + carry;
8744                 if (newdig >= NBASE)
8745                 {
8746                     carry = newdig / NBASE;
8747                     newdig -= carry * NBASE;
8748                 }
8749                 else
8750                     carry = 0;
8751                 dig[i] = newdig;
8752             }
8753             Assert(carry == 0);
8754             /* Reset maxdig to indicate new worst-case */
8755             maxdig = 1 + var1digit;
8756         }
```

```
/*
 * Add the appropriate multiple of var2 into the accumulator.
 *
 * As above, digits of var2 can be ignored if they don't contribut
 * so we only include digits for which i1+i2+2 < res_ndigits.
 *
 * This inner loop is the performance bottleneck for multiplicatio
 * so we want to keep it simple enough so that it can be
 * auto-vectorized.  Accordingly, process the digits left-to-right
 * even though schoolbook multiplication would suggest right-to-le
 * Since we aren't propagating carries in this loop, the order doe
 * not matter.
 */
{
    int         i2limit = Min(var2ndigits, res_ndigits - i1 - 2);
    int        *dig_i1_2 = &dig[i1 + 2];

    for (i2 = 0; i2 < i2limit; i2++)
        dig_i1_2[i2] += var1digit * var2digits[i2];
}
```

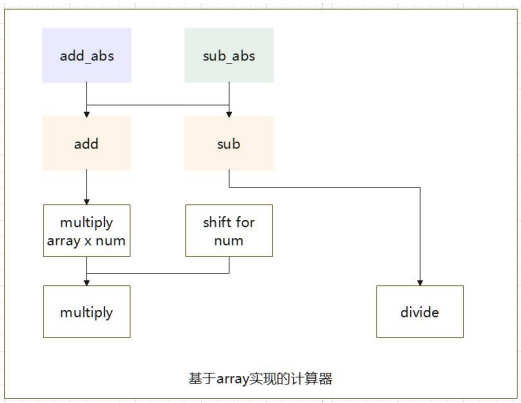
这里边它的计算复杂度是 O(n^2)，使用了两个 for 循环。在第二个 for 循环里，它将我们的矩阵逐步相加。不过这个 A[i] += B * C[i]的公式我可太熟悉了，这不就是我们的并行计算里最标准的并行化算法吗！看来如果想优化，我们可以从这里入手。当然，我猜测考虑到通信成本的问题，在这个第二层的 for 循环搞并行化一定是得不偿失的。

大概看完代码后，我实现乘法决定这么实现：对于数 A x B，我们先用 B 中的一位 var_digit 分别去乘上 A，也就是用一个数乘一个数组（multiply array x num），然后我们通过 var_digit 所在的位数，将我们的结果进行移位（shift），最后把我们的结果加到我们的 result 数组上。这样的操作是和 Postgresql 里的操作基本一致的。

```
943   /*
944   * Multiply an array of char and a number
945   * EX: "987654321" x "9" = "8888888889"
946   */
947   char* Multiply_array_and_num(char* number, char num, char* number_copy){
948       if(num == '0'){return "0";}
949       if(num == '1'){return number;}
950
951       int num_of_num = num - '0';
952
953       for(int i = 0; i < num_of_num; i++){
954           number_copy = Add_abs(number_copy,number,number_copy);
955       }
956       return number_copy;
957   }
```

```
961   char* Shifting_num_for_n_digit(char* number, int num, char* result,int number_length){
962       if(num==0){
963           return number;
964       }
965       int len = number_length;
966
967       for(int i = 0; i < len; i++){
968           result[i] = number[i];
969       }
970       // 全部填0
971       for(int i = len; i < len + num; i++){
972           result[i] = '0';
973       }
974       result[len + num] = '\0';
975       return result;
976   }
```

这样，我们就通过 array，实现了大整数的乘法。这里我画了一张实现的图，它阐述了各个方法之间的关系。当然，除法我没有实现。因为我暂时没有想明白，小数点应该怎么处理。



基于array实现的计算器

而 Postgresql 里的 div_var，更是让我都没有看明白。我，肖翊成和马国恒同学组成了对这个除法问题的研究小组，结果三天三夜又过去了，我们没有取得任何阶段性进展，我随即结束了这个项目的研究。

大整数的加减法跟 double 的加减法是不一样的。同样是一个加号，两者在最后的实现操作完全不同。这让我想起在上学期的离散数学里老师讲的群环域的知识。Double 所表示的数跟加法运算构成阿贝尔群，大整数所表示的数跟加法构成另一个阿贝尔群，这两个群或者说集合的元素与 Cardinality 都不相同，我们是不能直接因为在草稿纸上脑子一动，就理所应当地认为他们的加法是一样的。同样对于乘法也是一样的，一个是有限域，一个是整数域，我们不能简单地一视同仁。

```
8857    /*
8858     * If the divisor has just one or two digits, delegate to div_var_int(),
8859     * which uses fast short division.
8860     *
8861     * Similarly, on platforms with 128-bit integer support, delegate to
8862     * div_var_int64() for divisors with three or four digits.
8863     */
8864    if (var2ndigits <= 2)
8865    {
8866        int         idivisor;
8867        int         idivisor_weight;
8868
8869        idivisor = var2->digits[0];
8870        idivisor_weight = var2->weight;
8871        if (var2ndigits == 2)
8872        {
8873            idivisor = idivisor * NBASE + var2->digits[1];
8874            idivisor_weight--;
8875        }
8876        if (var2->sign == NUMERIC_NEG)
8877            idivisor = -idivisor;
8878
8879        div_var_int(var1, idivisor, idivisor_weight, result, rscale, round);
8880        return;
```

Postgresql 里除法的实现内容

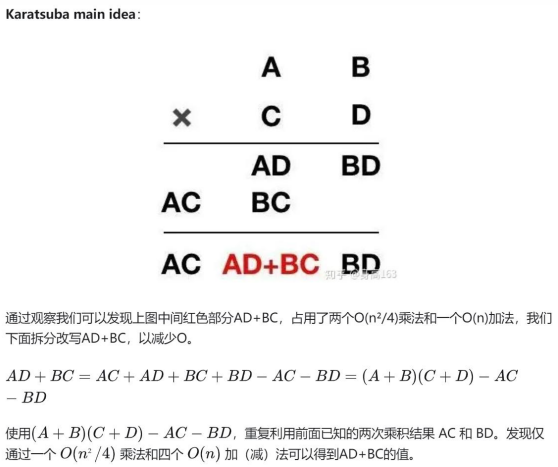
4.6 Karatsuba 大整数乘法探究

有没有比相加然后移位更快的算法？我专门上网找了一下，还真有。知乎的这篇 Karatsuba 大数乘法算法 - 知乎 (zhihu.com)，就介绍了多种乘法运算。我们刚刚使用的，仅仅是小学模拟乘法，列竖式相加。而其中使用 FFT 的 Schönhage–Strassen algorithm，复杂度可以说是已经很少很少！仅仅 O(n logn loglogn)！

大数乘法主要算法：

1. **小学模拟乘法**：最简单的乘法竖式手算累加型；
2. **分治乘法**：最简单的是Karatsuba乘法，一般化以后有Toom-Cook乘法；
3. **快速傅里叶变换FFT**：（为了避免精度问题，可以改用快速数论变换FNTT），时间复杂度O(N lgN lglgN)。具体可参照Schönhage–Strassen algorithm；
4. **中国剩余定理**：把每个数分解到一些互素的模上，然后每个同余方程对应乘起来就行；

至于 Karatsuba 算法，这算是我目前能够理解的算法了。它的核心要义在于，我们以往的列竖式相加，我们要分别计算 A x D 和 B x C，随后再相加。而他想到了讲这个数变成 (A+B)(C+D)-AC-BD，这样就能用上两边已经运算了的数字，减少了运算次数。假如一个 4 位数乘 4 位数，我们用小学生办法算要 16 次乘法，而他只要 9 次，这加快了我们的速度。



但是，这个算法有个很严重的问题，就是他需要计算对数和指数，这需要新的实现以及更多的功夫，因此，我最终决定不采用它这个算法。但是，这也让我开了眼界，让我意识到刚刚我所想的有限域与整域什么的，确实乘法可能比我想象的要复杂的多。

```c
double karatsuba_multiply(double x, double y) {
    if (x < 10 || y < 10) {
        return x * y;
    }

    // 计算 n 和 n2
    int n = fmax(log10(x) + 1, log10(y) + 1);
    int n2 = n / 2;

    // 拆分 x 和 y
    int a = x / pow(10, n2);
    int b = x - a * pow(10, n2);
    int c = y / pow(10, n2);
    int d = y - c * pow(10, n2);

    // 递归计算乘积
    double ac = karatsuba_multiply(a, c);
    double bd = karatsuba_multiply(b, d);
    double ad_plus_bc = karatsuba_multiply(a + b, c + d) - ac - bd;

    // 计算结果
    double result = ac * pow(10, 2 * n2) + ad_plus_bc * pow(10, n2) + bd;
    return result;
}
```

Karatsuba 算法实现（Generated by GPT）

Part 5: 内存泄露：Java 最高兴的一集

啊！终于来到了最痛苦的部分。我在这里花费了 4 天 4 夜，也依旧没有解决完这个问题。这是我有一天看完何泽安学长的 C/C++作业的时候发现的：内存泄露！每当我们在方法里 malloc 时，如果我们不 free 掉这部分内存，那么这部分内存会一直被视为被程序占用，一直到这个程序终止，这样的话，我们的计算器会随着时间的发展，逐步吃掉我们整个计算机的内存！

有什么办法可以检测这个问题吗？我找到了一个软件 valgrind，它是一个虚拟 CPU，在这个虚拟 CPU 上会运行我们的代码，然后检测有没有内存从此消逝了。

```
exit
Exiting the calculator.
==517688==
==517688== HEAP SUMMARY:
==517688==     in use at exit: 232 bytes in 3 blocks
==517688==   total heap usage: 5 allocs, 2 frees, 2,280 bytes allocated
==517688==
==517688== 120 bytes in 1 blocks are possibly lost in loss record 3 of 3
==517688==    at 0x4843828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==517688==    by 0x4A2515D: getdelim (iogetdelim.c:65)
==517688==    by 0x10B804: Mode2 (in /home/asc648/haibin/CPPLearning/Project1/haibinCalculator)
==517688==    by 0x10BB0E: main (in /home/asc648/haibin/CPPLearning/Project1/haibinCalculator)
==517688==
==517688== LEAK SUMMARY:
==517688==    definitely lost: 0 bytes in 0 blocks
==517688==    indirectly lost: 0 bytes in 0 blocks
==517688==      possibly lost: 120 bytes in 1 blocks
==517688==    still reachable: 112 bytes in 2 blocks
==517688==         suppressed: 0 bytes in 0 blocks
==517688== Reachable blocks (those to which a pointer was found) are not shown.
==517688== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==517688==
==517688== For lists of detected and suppressed errors, rerun with: -s
==517688== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

我们用 valgrind ./haibinCalcluator 12345 x 123456，去启动我们的计算器。结果发现，我们的计算器光荣地漏掉了 120byte，可能漏掉了 112byte。

我第一次考虑到这个问题时，前面的方法已经实现完成了，我此时只好去重构。而重构的方法也就借鉴了 Postgresql 里的操作：将代表 result 的字符串也输入进来，在 result 上进行操作，而不是在方法内自己 malloc 一个，不然的话返回的时候是无法 free 掉它的。真是伤脑筋！我瞬间感受到 **C/C++的弱点：内存不安全**！我又一想到可怕的 Windows 和 Postgresql 里有几千万行代码，说不定哪行代码就没有实现内存释放 free，就会导致 Windows 越用越卡！

重构花去了我很多的时间去调试和修改，写了一个寒假的 java 后，突然遇到这样的情况，我顿时有一种在屎山里遨游的感觉，我顿时懂得了 Oracle 程序员的心理，太狗屎了！

```
char* Add_abs(char* number1, char* number2, char* result) {
    int len1 = strlen(number1);
    int len2 = strlen(number2);
    int maxlen = Max(len1,len2);
    int carry = 0;
```

此时一位明君启发了我。他在他的博客里写明了一种管理内存分配的方法：我们每次要分配一个内存的时候，我们就往一个链表里塞上我们分配内存的指针。在释放时我们将指针从链表中移除。https://blog.csdn.net/weixin_43308899/article/details/135122404

```
143 typedef struct Allocation {
144     void* address;
145     size_t size;
146     struct Allocation* next;
147 } Allocation;
148
149 Allocation* allocations = NULL;
```

```
228 void add_allocation(void* p, size_t size){
229     Allocation* newAlloc = (Allocation*)malloc(sizeof(Allocation));
230     newAlloc->address = p;
231     newAlloc->size = size;
232     newAlloc->next = allocations;
233     allocations = newAlloc;
234 }
235
236 // 函数来移除内存分配记录
237 // https://blog.csdn.net/weixin_43308899/article/details/135122404
238 void remove_allocation(void* p) {
239     Allocation **ptr = &allocations;
240
241     // 栈移到下一个指针
242     while (*ptr) {
243         Allocation* entry = *ptr;
244         if (entry->address == p) {
245             *ptr = entry->next;
246             free(entry);
247             return;
248         }
249         ptr = &entry->next;
250     }
251 }
```

```c
void* Calculator_malloc(size_t size){
    void* p = malloc(size);
    add_allocation(p, size);
    return p;
}

// 自定义的 free 函数
void Calculator_free(void* p) {
    remove_allocation(p);
    free(p);
}
```

通过这样的方法，我们就可以动态控制我们之前分配的指针。为了方便，我自己写了一个 memcheck 和 clearMem 的函数，他们可以随时监控内存泄露和清空已经分配的内存。

```c
// made by haibin Lai
void clear_allocations() {
    Allocation* current = allocations;
    while (current) {
        Allocation* temp = current;
        current = current->next;
        free(temp->address);
        free(temp);
    }
    allocations = NULL;
}
```

```c
void check_for_leaks() {
    Allocation* current = allocations;
    if (current == NULL) {
        printf("No memory leaks detected.\n");
    } else {
        printf("Memory leaks detected:\n");
        while (current) {
            printf("Leaked memory at address %p, size %zu
            current = current->next;
        }
    }
}
```

经过我们的一番爆改之后，我们的计算器基本上不会出现内存泄露了。这里边很神奇，我今天（3/10）早上还是内存泄露的，结果下午改完了函数，就没有出现泄露了。之前的泄露主要是会固定有 112byte 的泄露。我至今没有找到泄露的原因。但是这次的学习让我认识到，提早规划好内存分配问题，才能及时止损。这给我的 C 编程一个很大的教训。

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ valgrind ./haibinCalculator 213456 x 123456
==771375== Memcheck, a memory error detector
==771375== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==771375== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==771375== Command: ./haibinCalculator 213456 x 123456
==771375==
213456 x 123456 = 26352423936
==771375==
==771375== HEAP SUMMARY:
==771375==     in use at exit: 0 bytes in 0 blocks
==771375==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==771375==
==771375== All heap blocks were freed -- no leaks are possible
==771375==
==771375== For lists of detected and suppressed errors, rerun with: -s
==771375== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
(base) asc648@H3C1:~/haibin/CPPLearning/Project1$ valgrind ./haibinCalculator 213456 + 2423453123456
==773311== Memcheck, a memory error detector
==773311== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==773311== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==773311== Command: ./haibinCalculator 213456 + 2423453123456
==773311==
==773311== Conditional jump or move depends on uninitialised value(s)
==773311==    at 0x4849CB8: strlen (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==773311==    by 0x48EAF57: puts (ioputs.c:35)
==773311==    by 0x10A5B8: Add_using_array (in /home/asc648/haibin/CPPLearning/Project1/haibinCalculator)
==773311==    by 0x10B070: CalculateBigInterger (in /home/asc648/haibin/CPPLearning/Project1/haibinCalculator)
==773311==    by 0x10B41E: CalculatorKernel (in /home/asc648/haibin/CPPLearning/Project1/haibinCalculator)
==773311==    by 0x10B7B5: main (in /home/asc648/haibin/CPPLearning/Project1/haibinCalculator)
==773311==
213456 + 2423453123456 = 2423453336912
==773311==
==773311== HEAP SUMMARY:
==773311==     in use at exit: 0 bytes in 0 blocks
==773311==   total heap usage: 3 allocs, 3 frees, 1,063 bytes allocated
==773311==
==773311== All heap blocks were freed -- no leaks are possible
==773311==
==773311== Use --track-origins=yes to see where uninitialised values come from
==773311== For lists of detected and suppressed errors, rerun with: -s
==773311== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
213456 x 24234531234567 = 5173006099204239936
==774196==
==774196== HEAP SUMMARY:
==774196==     in use at exit: 0 bytes in 0 blocks
==774196==   total heap usage: 55 allocs, 55 frees, 2,073 bytes allocated
==774196==
==774196== All heap blocks were freed -- no leaks are possible
==774196==
==774196== Use --track-origins=yes to see where uninitialised values come from
==774196== For lists of detected and suppressed errors, rerun with: -s
==774196== ERROR SUMMARY: 70 errors from 7 contexts (suppressed: 0 from 0)
```

```
==777740==
987654321 x 987654321 = 975461057789971041
987654321 x 987654321
987654321 x 987654321 = 975461057789971041
exit
Exiting the calculator.
==777740==
==777740== HEAP SUMMARY:
==777740==     in use at exit: 0 bytes in 0 blocks
==777740==   total heap usage: 79 allocs, 79 frees, 3,684 bytes allocated
==777740==
==777740== All heap blocks were freed -- no leaks are possible
==777740==
==777740== Use --track-origins=yes to see where uninitialised values come from
==777740== For lists of detected and suppressed errors, rerun with: -s
==777740== ERROR SUMMARY: 180 errors from 7 contexts (suppressed: 0 from 0)
```

当然，就算是内存泄露了，也没什么问题，我们用 htop 或者 btop（btop 的 UI 特别好看！强烈推荐）查看我们的计算器占用内存：

```
523595 root      20   0  2636  1024  1024 S   0.0  0.0  0:00.00 fusermount3 -o rw,nosuid,nodev,fsname=portal,au
523700 yangyk    20   0  2732  1024  1024 S   0.0  0.0  0:00.00 sh /home/yangyk/.vscode-server/bin/74f6148eb9ea
709762 asc648    20   0  8216  1024  1024 S   0.0  0.0  0:00.00 sleep 180
709764 asc648    20   0  8216  1024  1024 S   0.0  0.0  0:00.00 sleep 180
709912 asc648    20   0  3900  1024  1024 S   0.0  0.0  0:00.00 ./haibinCalculator
  3956 root      20   0  2380     0     0 S   0.0  0.0  0:00.46 tini -g -- /startup/docker-entrypoint.sh neo4j
  3971 root      20   0  2616     0     0 S   0.0  0.0  0:00.02 /bin/sh -c python3 app.py
```

我们发现，它的占用非常之少，我们甚至可以不管这个内存的泄露哈哈，但是为了后续矩阵能够乘过去，我们还是要保持严谨。

Part 6: 并行计算

6.1 应用 OpenMP

我简单使用了一下 OpenMp 来完成乘法的并行化。在这里，我学习了 omp 的基本语法。下面是我的一些理解：

omp parallel for: 这段 for 循环可以在多核上并行完成

shared: 在循环时，本质上每个核会自己有一份参数的 copy。那么为了保证 copy 的一致性，我们就要将它们设为 shared。其实按照道理 default 状态下 result 和 carry 就都是共享状态了。如果想要每个 for 循环的参数都是独立的，我们要加 private。

reduction: 规约。我们的计算当中，如果想汇总各个计算核心的结果，我们就要加一个 reduction 操作。

```
626  void multiply(char *num1, char *num2, char *result) {
627      int len1 = 0, len2 = 0;
628      while (num1[len1]) len1++;
629      while (num2[len2]) len2++;
630
631      int len_result = len1 + len2;
632      int carry = 0;
633
634      #pragma omp parallel for shared(result, carry)
635      for (int i = 0; i < len_result; i++) {
636          int temp = carry;
637          #pragma omp parallel for reduction(+:temp)
638          for (int j = 0; j <= i; j++) {
639              if (j < len1 && i - j < len2) {
640                  temp += (num1[len1 - j - 1] - '0') * (num2[len2 - (i - j) - 1] - '0');
641              }
642          }
643          result[len_result - i - 1] = temp % 10 + '0';
```

```
        result[len_result - i - 1] = temp % 10 + '0';
        carry = temp / 10;
    }

    result[len_result] = '\0';

    // Remove leading zeros
    int start = 0;
    while (result[start] == '0' && result[start] != '\0') {
        start++;
    }
    if (start > 0) {
        for (int i = start; i <= len_result; i++) {
            result[i - start] = result[i];
        }
    }
}
```

去年我参加了一个鲲鹏 HPC 的训练营，在 MPI 中，类似的这样的并行参数有像 barrier,Bcast,Scatter。其实 OpenMP 的思想跟 MPI 的差不多，一个是核与核之间的通信，另一个是节点与节点之间的通信。OpenMP 的通信更加的细腻。

这次的计算器算是我初步了解 OpenMP 吧，简单学习一下它的思想。我相信在接下来的 Matrix 相关的制作里，我会对这个更加得心应手。

6.2 GPU 编程：将浮点计算移植到 CUDA 架构

在 CUDA 编程中，我们的计算器主要包含两个部分：CPU 上的函数和 GPU 的核函数。

GPU 的核函数用 global 表示入口，然后在内部写下函数。只要满足 A = B + C*D 这种矩阵形式，nvcc 在编译时将自动把算法分配到各个核中去执行。

```
__global__ void multiply(char *num1, char *num2, char *result, int len1, int len2) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < len1 + len2) {
        int temp = 0;
        for (int j = 0; j <= idx; j++) {
            if (j < len1 && idx - j < len2) {
                temp += (num1[len1 - j - 1] - '0') * (num2[len2 - (idx - j) - 1] - '0');
            }
        }
        result[idx] = temp % 10 + '0';
    }
}
```

而此时 CPU 由于计算的任务主要有 CUDA 负责了，CPU 任务主要就是将我们的参数搬运到 GPU 的内从中去。我们用 cudaMalloc 函数就能完成了。

之后，我们再用 NVCC 编译器去编译我们的 haibinCalculator.cu 文件，就能获得我们的大整数乘法计算器了。

```
1.      nvcc -o GPUCalculator src/haibin_calculator_cuda.cu
```

```
    char padded_num1[MAX_SIZE], padded_num2[MAX_SIZE];
    for (int i = 0; i < MAX_SIZE; i++) {
        padded_num1[i] = (i < len1) ? num1[len1 - i - 1] : '0';
        padded_num2[i] = (i < len2) ? num2[len2 - i - 1] : '0';
    }

    cudaMalloc((void **)&d_num1, MAX_SIZE * sizeof(char));
    cudaMalloc((void **)&d_num2, MAX_SIZE * sizeof(char));
    cudaMalloc((void **)&d_result, (len1 + len2) * sizeof(char));

    cudaMemcpy(d_num1, padded_num1, MAX_SIZE * sizeof(char), cudaMemcpyHostToDevice);
    cudaMemcpy(d_num2, padded_num2, MAX_SIZE * sizeof(char), cudaMemcpyHostToDevice);

    int num_blocks = (len1 + len2 + BLOCK_SIZE - 1) / BLOCK_SIZE;
    multiply<<<num_blocks, BLOCK_SIZE>>>(d_num1, d_num2, d_result, len1, len2);

    cudaMemcpy(result, d_result, (len1 + len2) * sizeof(char), cudaMemcpyDeviceToHost);
```

```
    // Remove leading zeros
    int start = 0;
    while (result[start] == '0' && start < len1 + len2) {
        start++;
    }
    for (int i = start; i < len1 + len2; i++) {
        result[i - start] = result[i];
    }

    printf("Product: %s\n", result);

    cudaFree(d_num1);
    cudaFree(d_num2);
    cudaFree(d_result);
```

由于这次只是初识 CUDA，我就没有继续做其他复杂的操作。我希望在后续的 Project 里，我能在矩阵运算中发挥这些知识。

6.3 性能测试

Which are better：OpenMP or CUDA

很遗憾，因为最后这几天我都感冒了，我决定放弃这部分的测试。其实根据我自己的猜测，对于一个大整数乘法，其效率开始是 单线程>OpenMP>CUDA， 随后是 OpenMP > 单线程 > CUDA，后面可能是 CUDA > OpenMP > 单线程。这一切应该是跟通信有关。

Part 7: 致敬传奇

7.1 快速平方根算法背后的秘密

这是一个伟大的算法，最早出现在雷神之锤 III 游戏中，它展现了程序员高超的数学能力以及计算机底层知识。

```
/**
 * A Tribute to the legend
 * 致敬传奇平方根倒数算法
 */
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit leve
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be r

    return y;
}
```

我原本很想分享一下这个算法的故事，以及 MathWorks 的创始人跟他的故事，但是身体不佳，只能用 GPT 诉说：

这个算法是一种快速计算浮点数倒数平方根的近似方法，通常被称为"快速反平方根"算法。它的起源可以追溯到 Quake III Arena 游戏的源代码，由程序员 John Carmack 在 1999 年设计。这个算法以其非常高的性能和相对简单的实现而闻名。

下面是对这个算法的简要解释：

1.x2 = number * 0.5F;：将输入的浮点数 number 乘以 0.5，得到其一半的值，并赋给 x2。

2.y = number;: 将输入的浮点数 number 赋给 y。

3.i = * (long *) &y;: 使用类型转换（类型强制转换），将浮点数 y 的内存表示转换为长整型 i。这是一种"浮点数位级别的邪恶黑魔法"，用于获取浮点数的二进制表示。

4.i = 0x5f3759df - (i >> 1);: 这一行进行了一些神秘的操作。通过这个操作，算法试图近似计算输入数的平方根的倒数。这个数 0x5f3759df 是一个经验性选择的常数，通过一些黑魔法的方式，将它减去右移一位后的 i，得到一个近似的倒数平方根的近似值。

5.y = * (float *) &i;: 将上一步计算得到的近似倒数平方根的近似值转换回浮点数，赋给 y。

6.y = y * (threehalfs - (x2 * y * y));: 使用牛顿迭代法，通过进行一次迭代来提高近似值的精度。这个迭代步骤是算法的核心。它通过对牛顿迭代公式进行简化，以提高计算效率。这一行是算法的主要迭代步骤。

7.注释掉的 y = y * (threehalfs - (x2 * y * y)); 行是第二次迭代。在实际使用中，第二次迭代可能会带来更高的精度，但也会增加计算成本。在某些情况下，第二次迭代被注释掉以平衡性能和精度的需求。

这个算法是一种高度优化的近似计算方法，适用于需要快速计算浮点数倒数平方根的场景。然而，它并不总是提供完全精确的结果，因此在一些应用中，更为精确但计算成本更高的方法可能更为合适。

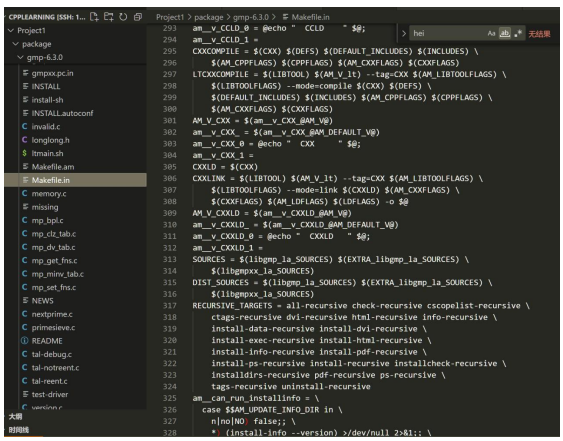
Part 8: 库的学习

8.1 大整数乘法 GMP 数学库

网址：https://gmplib.org/ GMP 大数库是 GUN 项目的一部分，它诞生于 1991 年，作为一个任意精度的大整数运算库，它包含了任意精度的整数、浮点数的各种基础运算操作。它是一个 C 语言库，主要应用于密码学应用和研究、互联网安全应用、代数系统、计算代数研究等。

GMP 库运行速度非常快，有网友测试它算 1000!结果不到一秒钟就已经算好，这还包括了输入输出的时间。

GMP 非常的恐怖，里边的历史日志真的从 1991 年开始记录，一直记录到 2023 年。单单是编译安装我都用了 5 分钟，而里边几百上千个文件，就连 Makefile 都有快千行。我又想到，我的计算器比起 Postgresql 已经是小巫见大巫，而这个 GMP 库的计算器又让 Postgresql 显得何其渺小。真是“哀吾生之须臾，羡长江之无穷。”



既然已经有了这么牛的计算器，那我们为什么还在写计算器呢？我想，可能就跟苏轼说得一样，GMP 库从 0 到这么辉煌，终究还是别人的成果，而我们自己如果想体验这样的状态，没有自己书写过，怎么会有相似的感悟呢？“耳得之而为声，目遇之而成色”，

8.2 与 PBC 密码学库

密码学里有一个好玩的东西是全同态加密，我本来想在我的计算器里进行尝试，结果时间不够，放弃了。希望有机会再体验。

Part 9: 结语

这个简单的计算器，让我感受到 C 语言的不简单。虽然才刚刚开始上手，但是指针，内存管理，多线程，优化等等东西都让我感受到这门语言独有的魅力。我认为语言的学习不是学个语法和单词就结束的，也不是学到分支、循环就结束了。学一门语言，就要学它的优势之处，学它的核心要义，这才是最重要的。

而我的这次体验，让我感受到 C 的基础与重要性。我们现在的各种工具，操作系统，最底层里原来也就是这么简单的事务组合而成。想到这里，我又冒出了那个熟悉的词汇：**敬畏。**这些代码一点一点的累加，才有了今天的高楼大厦。

我也没有把实现一个完美的计算器作为的我目标。因为我知道，仅仅 3 周的时间，我不可能做出像 GMP 库这样这么伟大的数学库，我也不想单纯做一个调包侠，调这个包那个包，因为我的目标是学习语言，而并非一个打败世界的计算器。只要我的需求达到了，我想我的 Project 目的也做到了。

另外，这次我的研究其实涉及的范围挺广的，主要是因为我单纯抱着第一步认识的**兴趣**去研究。为什么我想研究这些？其实也没有什么特别的原因。但是我此时想起了朱光潜《谈美》里的一段话：

“科学和哲学穷到极境，都是要满足求知的欲望。每个哲学家和科学家对于他自己所见到的一点真理（无论它究竟是不是真理）都觉得有趣味，都用一股热忱去欣赏它。“地球绕日运行”，“勾方加股方等于弦方”一类的科学事实，和《密罗斯爱神》或《第九交响曲》一样可以摄魂震魄。科学家去寻求这一类的事实，穷到究竟，也正因为它们可以摄魂震魄。所以科学的活动也还是一种艺术的活动，不但善与美是一体，真与美也并没有隔阂。”

写 C 程序的美感跟 Java 很不一样，但是，我很喜欢。

Appendix

Acknowledgement

肖翊成 Lab member No.2

同班的好朋友，超算队的好队友。虽然这学期没有选择 C/C++这门课，但是为学习和创新实践的需要跟随了我的 lab 和 Project，他的创新实践是 linux 上跑一个自动驾驶的软件。平时就遇到的问题我们也会相互讨论。感谢他在这次 Project 中就 Postgresql 源码分析中的解读，以及听我倾诉这次 Project 的不容易。

马国恒 Lab member No.3

同宿舍的好朋友，航模社的好飞手。这学期跟着我一起旁听了这门课程。感谢他这三周都为我大课占位置，以及在 https://github.com/btmills/calculator 这个人的计算器上与我一起进行了解读。Github 上这个人是一家金融公司的程序员，他写出了他们公司优秀的前端。

杨宇坤

感想杨师兄。感谢他与我数组实现的分享以及大整数乘法中 FFT 方面相关的知识。

Reference:

[1] PostgreSQL Global Development Group. (2024.3). PostgreSQL: Documentation: Main Page. Retrieved from https://doxygen.postgresql.org/

[2] GMP Project. (2023.7). GMP, Arithmetic without limitations: The GNU Multiple Precision Arithmetic Library. Retrieved from https://gmplib.org/

[3] "IEEE Standard for Binary Floating-Point Arithmetic," in ANSI/IEEE Std 754-1985 , vol., no., pp.1-20, 12 Oct. 1985, doi: 10.1109/IEEESTD.1985.82928. keywords: {Digital arithmetic;Standards;binary;Floating-point arithmetic},

[4] IEEE Standard 754 for Binary Floating-Point Arithmetic. (1997.10). Retrieved from https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

[5] Burger, Robert G. and R. Kent Dybvig. “Printing floating-point numbers quickly and accurately.” ACM-SIGPLAN Symposium on Programming Language Design and Implementation (1996).

[6] CUDA C Programming Guide. Retrived from https://docs.nvidia.cn/cuda/cuda-c-programming-guide/contents.html

[7] Using OpenMP with C. Retrived from https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html