Project 1: A Simple Calculator

Contents

§1. Requirements Analysis 需求分析

§1.1. 参数解析

Example (execute with arguments).

```
$./calculator 2 + 3
2 + 3 = 5
$./calculator 2 - 3
2 - 3 = -1
$./calculator 2 x 3
2 x 3 = 6
$./calculator 2 / 3
2 / 3 = 0.666667
```

第一种参数解析方式是直接执行程序，后面跟着两个操作数和一个运算符，程序输出表达式运算结果。 对此我们可以直接使用 argc 和 argv 来解析参数，然后根据参数的个数和内容来判断执行的操作。

Example (execute in interactive mode).

```
$./calculator
2 + 3
2 + 3 = 5
2 - 3
2 - 3 = -1
2 x 3
2 x 3 = 6
2 / 3
2 / 3 = 0.666667
```

第二种参数解析方式是交互式输入，程序输出表达式运算结果。这种情况我们可能需要使用 scanf/fgets/getline 等库函数来解析输入，具体选择哪个就需要深入考察各个函数的实现细节和适用场景。

§1.1.1. scanf()函数

作为我们在 C 中最常用的读取输入内容的函数, C99 标准以来的 scanf()函数原型如下：

```
int scanf( const char *restrict format, ... );
```

可变参数为需要读取的变量地址，返回值为成功读取的参数的个数。在大多数情况下，scanf()函数可以按格式读入需要的各种数据类型，但对于本次 project 中可能需要读取长串的数字类型，scanf()函数还适用吗？例如我们在下面的代码中约定输入缓冲为一个6字符的字符串，如果用户输入的表达式超过6个字符，scanf()函数会怎么处理？

```
#include <stdio.h>
int main() {
    char* str = (char*)malloc(6 * sizeof(char));
    scanf("%s", str);
    for (int i = 0; i < 10; i++) { /* 这里打印10个字符是为便于观察缓冲区之外的内容 */
        if (str[i] == '\0') {
            printf("\\0"); /* 可视化字符串结束符 */
        } else {
            printf("%c", str[i]);
        }
    }
    free(str);
    return 0;
}
```

Observation 1.1.1.1 (Result of the code above).

```
> 123456
< 1234560000
> 1234567
< 1234567000 # extra character '7' was written out of the buffer after scanf()
```

我们由上述输入可以看到，虽然我们在动态分配内存时限制了字符串长度为6，但是`scanf()`函数仍然会读取缓冲区之外的内容，这可能会导致一些不可预知的内存错误。在这个示例中因为内存对齐和变量较少等原因没有导致严重的错误，然而在 project 可能是致命的，它会导致我们无法意识到某处的内存是什么时候被更改了。因此，我们需要考虑使用其他函数来代替`scanf()`函数。

Note (scanf()的底层实现).
为了深入研究 scanf()的运作机制，我从 llvm/llvm-project/blob/main/libc/src/stdio/scanf_core/string_converter.cpp#L22 处找到了核心函数[1]。

```
  // string_converter.cpp
  int convert_string(Reader *reader, const FormatSection &to_conv) {
  // %s "Matches a sequence of non-white-space characters"

  // %c "Matches a sequence of characters of exactly the number specified by
the
  // field width (1 if no field width is present in the directive)"

  // %[ "Matches a nonempty sequence of characters from a set of expected
  // characters (the scanset)."
  size_t max_width = 0;
  if (to_conv.max_width > 0) {
    max_width = to_conv.max_width;
  } else {
    if (to_conv.conv_name == 'c') {
      max_width = 1;
    } else {
      max_width = cpp::numeric_limits<size_t>::max();
    }
  }

  char *output = reinterpret_cast<char *>(to_conv.output_ptr);

  char cur_char = reader->getc();
  size_t i = 0;
  for (; i < max_width && cur_char != '\0'; ++i) {
    // If this is %s and we've hit a space, or if this is %[] and we've found
    // something not in the scanset.
    if ((to_conv.conv_name == 's' && internal::isspace(cur_char)) ||
        (to_conv.conv_name == '[' && !to_conv.scan_set.test(cur_char))) {
      break;
    }
    // if the NO_WRITE flag is not set, write to the output.
    if ((to_conv.flags & NO_WRITE) == 0)
      output[i] = cur_char;
    cur_char = reader->getc();
  }

  /* ... */
  return READ_OK;
}
```

[1]This is typically an implementation of scanf() in LLVM libc. The actual implementation may vary with different compilers.

在初始化 max_width 时，因为我们的字符串没有被初始化，所以 max_width 会被设置为 cpp::numeric_limits<size_t>::max()，这会导致 for 循环中的 i < max_width 条件永远为真，从而导致 for 循环一直读取字符直到遇到\0 为止，同时被处理字符串目标 to_conv.scan_set 被写入 reader 中读出的多余字符。这就是为什么 scanf()会读取缓冲区之外的内容的原因。

scanf()函数对字符串的处理提示我们虽然有些函数看似简单，但是底层实现可能会有很多细节需要考虑。在 C/C++编程中，我们时刻要警惕这些“理所当然”的方法，因为它们可能会导致一些不可预知的错误。

§1.1.2. fgets()函数

C99 标准以来的 fgets()函数原型如下：

```
char* fgets( char* restrict str, int count, FILE* restrict stream );
```

str 为读入的缓冲字符串，count 为读入的最大字符数，stream 为读入的文件流。fgets()函数会读取 count-1 个字符，然后在末尾加上\0，返回 str。当需要从标准输入流读取时，stream 可以为 stdin（注意 stdin 也是 FILE* 文件描述符指针类型）。fgets()函数可以避免 scanf()函数的问题，因为它会读取指定长度的字符，不会读取缓冲区之外的内容。不过 fgets()会读取换行符，所以我们需要在读取后去掉换行符。

```
#include <string.h>

char str[6];
fgets(str, 6, stdin);

/* remove '\n' */
char* n_pos = strchr(str, '\n');
if (n_pos != NULL) {
    *n_pos = '\0';
}
// perform other operations ...
```

§1.2. 高精度数值库的需求

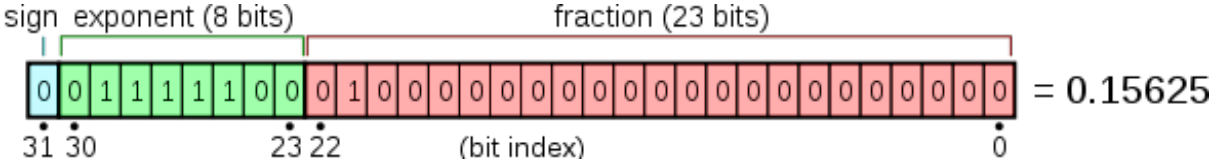
Example (deal with big numbers).

```
$./calculator 987654321.0 + 0.123456789
987654321.0 + 0.123456789 = 987654321.12345789
$./calculator 987654321 * 123456789
987654321 * 123456789 = 121932631112635269
$./calculator 1.0e200 * 1.0e200
1.0e200 * 1.0e200 = 1.0e400
```

从这些输入输出样例可以看出。我们要处理的数值精度可能会非常大，尤其是涉及到乘法运算时，我们可能需要处理10^{400}数量级的数值。我们需要考察现有的数据类型是否能够满足这个需求，如果不能，我们需要设计一个高精度数值库来处理这个问题。

§1.2.1. IEEE 754 浮点数标准

回顾Computer Organization课程中学习的浮点数在计算机中的表示，我们知道浮点数是由符号位(sign bit)、指数位(exponent bit)和尾数位(fraction bit)组成的[3]，数学记法为 $(-1)^s \times M \times 2^E$. 其中$s$为符号位，$M$为尾数，$E$为指数。 例如单精度浮点数0.15625的表示为：



在这里显然符号位为 0，将0.15625分解为二进制表示$(1.01)_2 \times 2^{-3}$，我们可以得到尾数位为$1 + 0.01(i.e.01000000000000000000000)$，指数位需加上偏置值127，即$-3 + 127 = 124(i.e.01111100)$。因此0.15625的 IEEE 754 表示为0_01111100_01000000000000000000000。

§1.2.2. 特殊浮点位

Definition 1.2.2.1.
IEEE 754 标准定义了一些特殊的浮点数 [3]:

- 无穷大(**inf**): 指数位E全为 1，尾数位M全为 0
  - 根据符号位s，正无穷大为$+inf$，负无穷大为$-inf$
  - 如字面意思，无穷大用来表示超过浮点数范围的数值。例如对于 C 中的 `double` 类型，最大值为$1.7976931348623157 \times 10^{308}$，如果我们计算$1.0 \times 309$，则会得到$+inf$。
- **NaN**: 指数位E全为 1，尾数位M不全为 0
  - NaN 有两种类型，一种是$QuietNaN$，另一种是$SignalingNaN$
  - 当计算结果无法表示时，例如计算$0/0$，则会得到NaN
  - 对两个NaN进行比较，结果为 false，因为NaN不等于任何数值，包括自己
  - 确定一个数是否为NaN，可以使用 `isnan()`函数
- 非规格化数: 指数位E全为 0，尾数位M不为 0，此时指数为$E = 1 - bias$，$bias$为偏置值。
  - 非规格化数用来表示接近 0 的数值，例如对于单精度浮点数，$2^{-126} \times 2^{-23} = 2^{-149}$，这个数值小于$2^{-126}$，因此无法用规格化数表示。这时我们可以使用非规格化数来表示。最小的非规格化数为$2^{-126} \times 2^{-23} = 2^{-149}$

考虑特殊浮点位的存在，一个给定精度的浮点数的表示范围会更加有限。比如对于 `float` 类型，规格数最大值为$1..1 \times 2^{127} \approx 3.4 \times 10^{38}$ (指数位不能为全 1)， 而非规格化数最小值为$0..01 \times 2^{-23} \approx 1.4 \times 10^{-45}$。

除了范围，我们还需考察现有浮点数的精度，在此我们首先定义一个浮点数的精度(Precision)为

Definition 1.2.2.2 (Precision of float number).
一个浮点数的精度为其尾数位上对应到十进制的有效位数。

一个单精度的浮点数底数的有效位数有23位，由于$2 \times 10^{23} \approx 10^{6.92}$，因此单精度浮点数的精度应该有7位。这也是我们在 C 中将这样的字面量赋值给 `float` 类型时，小数点后面的位数会被截断的原因。

```
float f1 = 0.123456780f; // f = 0.1234568
float f2 = 0.123456789f; // f = 0.1234568
```

以下为 Wikipedia 上对单精度浮点数与双精度浮点数的表示范围与精度的总结：

Level	Width	Range at full precision	Precision[a]
Single precision	32 bits	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	Approximately 7 decimal digits
Double precision	64 bits	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	Approximately 16 decimal digits

经过上述论述，现有的数值类型无论是 `float` 还是 `double` 从精度上来说，不可能存的下$1e400$这种数量级的数值，而且从精度上来说，也不可能支持$987654321.0 + 0.123456789 = 987654321.12345789$这种至少需要18位精度的计算。对此我们可以选择直接使用更高精度的 `long double` 类型，然而这种方案的缺陷是存储任何精度的数值都会占用更多的内存，而且 `long double` 的精度实际上也是有限的，这样的设计原则并不优雅，为此从头实现一个能够指定任意精度的数值类型是一个更好的选择。

§1.3. 多位乘法运算的效率优化

一个朴素的实现多位乘法的方法是仿照竖式乘法，即将两个数的每一位相乘，然后将结果相加。这种方法的时间复杂度为$O(n^2)$，其中n为两个数的位数。这也是大多数硬件底层实现的乘法运算的方法。如果我们要计算极高精度的两个数的乘法，这种方法的效率会非常低,对于我们要设计一个较高响应速度的程序的目标是相悖的。因此我们需要考虑一些优化方法，并且通过实际测试来验证这些方法的效率。

§1.3.1. Karatsuba 算法 [1]

Karatsuba 算法是对朴素竖式除法进行分治操作的优化，具体的操作步骤如下:

考虑两个十进制数x, y, 均有n个数位，任取 $0 < m < n$, 记为

$$x = x_1 \cdot 10^m + x_0$$

$$y = y_1 \cdot 10^m + y_0$$

$$x \cdot y = z_2 \cdot 10^2 m + z_1 \cdot 10^m + z_0$$

其中 $x_0, y_0, z_0, z_1 < 10^m$, 可知

$$\begin{aligned} z_2 &= x_1 \\ z_1 &= x_1 \cdot y_0 + x_0 \cdot y_1 \\ z_0 &= x_0 \cdot y_0 \end{aligned}$$

经过代入，我们可以得到

$$z_1 = (x_1 + x_2)(y_1 + y_2) - z_2 - z_0$$

上面的示例展示了 Karatsuba 算法的核心思想，即将长度为n的大数乘法问题分解为三个长度为$\lceil \frac{n}{2} \rceil$的大数乘法问题，然后通过递归的方式解决这些子问题。 以$T(n)$ 表示该算法的运行时间，则有$T(n) = 3T(\lceil \frac{n}{2} \rceil) + O(n)$，根据主定理，我们可以得到 Karatsuba 算法的时间复杂度为$\Theta\left(n^{\log_2 3}\right) \approx \Theta(n^{1.585})$。 [2]

[2]Master theorem is used to analyze the time complexity of divide-and-conquer algorithms. For more details, see OI-wiki#Master-Theorem.

Warning (Cons of Karatsuba).
虽然 Karatsuba 算法在一定程度上提高了多位乘法的效率，但是在实际应用中，由于递归的开销，Karatsuba 算法的效率并不一定比朴素的多位乘法高。因此在实际应用中，我们需要根据具体情况来选择是否使用 Karatsuba 算法。

§1.3.2. Fast Fourier Transorm 算法 [2]

在 ACM-OI 竞赛中一种常用的处理大数乘法的思想是离散傅里叶变换(DFT)。DFT 是一种将一个离散序列转换为另一个离散序列的方法，它在信号处理、图像处理等领域有着广泛的应用，但同时它也是我们处理多项式乘法的利器。

在大数乘法中，我们可以将两个大数看作是两个多项式，如 n 位的数 A 和 B 可以记作

$$A(x) = a_{n-1}x^{n-1} + ... + a_1x + a_0$$

$$B(x) = b_{n-1}x^{n-1} + ... + b_1x + b_0$$

其相乘结果即为 $C(x) = A(x) \cdot B(x)$，当然仅仅如此等效我们仍需要至少计算 n^2 次乘法运算。如果我们能像求两个长度为 n 的向量内积一样求两个多项式的乘积，那么就能大大减少计算量。在思考如何进行这种变换之前，我们先来看看多项式的表示。

§1.3.2.1. 多项式系数表示与点值表示

Definition 1.3.2.1.1.
一个多项式 $f(x)$ 的系数表示为

$$A(x) = a_{n-1}x^{n-1} + ... + a_1x + a_0$$

也可记作 $f(x) = (a_{n-1}, a_{n-2}, ..., a_1, a_0)$，即系数的有序排列

而我们有代数基本定理[3]，可以不加证明地得出，如果我们在复数域上考察多项式 $f(x)$ 的 n 个根 $x_0, x_1, ..., x_{n-1}$，则 $f(x)$ 可以唯一地表示为

$$f(x) = a_{n-1}(x - x_0)(x - x_1)...(x - x_{n-1})$$

即 $f(x)$ 可以记作

$$\{(x_0, f(x_0)), (x_1, f(x_1)), ..., (x_n, f(x_n))\}$$

这种表示称为多项式的点值表示。在这种表示下，假设两个多项式 $f(x), g(x)$ 分别表示为

$$\tau(f) = \{(x_0, f(x_0)), (x_1, f(x_1)), ..., (x_n, f(x_n))\}$$
$$\tau(g) = \{(x_0, g(x_0)), (x_1, g(x_1)), ..., (x_n, g(x_m))\}$$

则 $f(x) \cdot g(x)$ 的点值表示为

$$\tau(f) \cdot \tau(g) = \{(x_0, f(x_0) \cdot g(x_0)), (x_1, f(x_1) \cdot g(x_1)), ..., (x_n, f(x_n) \cdot g(x_m))\}$$

这样就使得多项式的乘法运算变得非常简单，只需要 $O(n)$ 次乘法运算即可得到结果。

[3]see https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra

§1.3.2.2. 快速傅里叶变换(正向)

在理解傅里叶变换之前可能需要了解单位根与虚数的相关知识，此部分内容请移步附录 Appendix A，我们直接给出典型的傅里叶变换的定义：

Corollary 1.3.2.2.1.
一个关于时间 t 的连续信号 $f(t)$ 在频率为 ω 下的傅里叶变换为

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt$$

类似的，我们可以定义一个离散序列 $\{x_n\}_{n=0}^{N-1}$ 的傅里叶变换为

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k\frac{n}{N}}$$

这就是离散形式的傅里叶变换(Discrete Fourier Transorm)，其中 k 为频率，N 为序列长度。

对于一个多项式 $f(x) = \sum_{i=0}^{n-1} a_i x^i \in X_{n-1}$，假设 $n = 2^s, s \in \mathbb{N}$，我们将多项式的奇偶项分开，即

$$f(x) = \sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{\frac{n}{2}-1} a_{2i}x^{2i} + x\sum_{i=0}^{\frac{n}{2}-1} a_{2i+1}x^{2i+1} = f_1(x^2) + xf_2(x^2)$$

带入单位根 $x = \omega_n^k (k < \frac{n}{2})$ 可以将 $f(x)$ 变换为

$$\begin{aligned} f\left(\omega_n^k\right) &= f_1\left(\omega_n^{2k}\right) + \omega_n^k f_2\left(\omega_n^{2k}\right) \\ &= f_1\left(\omega_{\frac{n}{2}}^k\right) + \omega_n^k f_2\left(\omega_{\frac{n}{2}}^k\right) \end{aligned}$$

同样的，将单位根在复数域旋转 $\frac{2\pi}{n}$ 可以得到

$$\begin{aligned} f\left(\omega_n^{k+\frac{n}{2}}\right) &= f_1\left(\omega_n^{2k+n}\right) + \omega_n^{k+\frac{n}{2}} f_2\left(\omega_n^{2k+n}\right) \\ &= f_1\left(\omega_{\frac{n}{2}}^k\right) - \omega_n^k f_2\left(\omega_{\frac{n}{2}}^k\right) \end{aligned}$$

因此问题被转化为求 $f_1\left(\omega_{\frac{n}{2}}^k\right), f_2\left(\omega_{\frac{n}{2}}^k\right)$，这样就可以在 $\mathrm{O}(\log n)$ 的时间完成将多项式 $f(x)$ 转换为点值表示 $f\left(\omega_n^k\right)$ 的过程。

§1.3.2.3. 快速傅里叶逆变换

在得到多项式的点值表示后，我们执行$O(n)$复杂度的内积运算，但结果仍然是一个点值表示的多项式。为了得到多项式的系数表示，也就是将乘积转换为我们熟悉的十进制表示，我们需要考虑 FFT 的逆变换。

如果将离散傅里叶变换视作一个线性算子，一个离散傅里叶变换可以表示为

$$\begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix}$$

这个矩阵是一个 Vandermonde 矩阵，它的逆矩阵可以通过 Vandermonde 矩阵的性质[4]得到，即对应每一个元素的逆，再除以 N 即可得到逆变换的结果。

根据欧拉公式，对于 ω_k^{-1} 我们有 $\omega_k^{-1} = \cos(2\pi\frac{k}{N}) - i\sin(2\pi\frac{k}{N})$，也就是 $e^{\frac{-2\pi i}{k}}$，这样对于实现逆变换就有了清晰的思路。

§2. Play with program 项目功能演示

> **Note.**
> 推荐编译命令:
>
> ```
> gcc submission_ver.c -lm -Wall
> ```

1.基础用法：无任何参数，以交互模式与默认精度启动，打印“minibc“的 banner 并输出提示符引导用户输入 二元表达式、数学函数调用以及回显数字，输入 `q`, `quit`, `Ctrl-D` 退出程序。

Example (Basic Interactive Usage).

```
$./minibc
                    _       _ __
              ____ ___  (_)___  (_) /_  _____
             / __ `__ \/ / __ \/ / __ \/ ___/
            / / / / / / / / / / / /_/ / /__
           /_/ /_/ /_/_/_/ /_/_/_.___/\___/

====== Interactive Mode (type "quit" or 'q' to leave) ======
> 987654321 + 0.123456789
987654321.123456789
> 1e200 x 1E200
1E400
> 1 / 99
0.0101
> -1 - 0
-1
> q
=========================================================
Free the memory and exit...
=========================================================
Bye!
```

[4]This is a lemma about Vandermonde matrix, see https://en.wikipedia.org/wiki/Vandermonde_matrix

2. 带参命令用法：直接在执行参数加上二元表达式会以“表达式 = 结果”的格式打印。
目前这种模式不支持函数调用，因为会与 shell 对括号()的解析发生冲突。

Example (Command Line Usage).

```
$./minibc 987654321 + 0.123456789
987654321 + 0.123456789 = 987654321.123456789
$./minibc 1e200 x 1E200
1e200 x 1E200 = 1E400
$./minibc 1 / 99
1 / 99 = 0.0101
$./minibc -1 - 0
-1 - 0 = -1
```

3. 指定计算精度：加上“-p “可以以指定精度启动，精度为负数或其他不合法格式会输出提醒。

Example (Interactive start with given presion).

```
$./minibc -p 5
                    _       _ __
              ____ ___  (_)___  (_) /_  _____
             / __ `__ \/ / __ \/ / __ \/ ___/
            / / / / / / / / / / / /_/ / /__
           /_/ /_/ /_/_/_/ /_/_/_.___/\___/

====== Interactive Mode (type "quit" or 'q' to leave) ======
> 12345 x 54321
 [NOTICE]: /home/zaddle/code/CPP/project1/src/math.c:598: at function
'Exn_normalize'
  └──<Math> Precision loss for current precision = 5, increase to 9 at least.
670590000
```

4. 用户友好的错误输出：当用户输入了不合法的表达式，都会引发相应的详细错误输出，
包括发生错误的代码文件名、函数名、行数及错误原因，详细错误类型可参考以下例子。

Example (Some invalid input).

```
> 0721$
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:478: at function
'__Exn_fromstr'
  └──<Math Exception> Cannot convert 'str' to math type 'ExtendedNumber': not a
digit
        0721$
            ^


 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1874: at function
'MathExpr_build'
  └──<Input Exception> Invalid argument 'opr': The echo number has an error
> 34 ^ 8
```

```
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1852: at function
'MathExpr_build'
  └──<Math Exception> Unrecognized operation '^'
> 7.8E26.3
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:552: at function 'atoExn'
  └──<Math Exception> Cannot convert 'str' to math type 'ExtendedNumber': no
decimal point in an exponent
        7.8E26.3
              ^~

 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1874: at function
'MathExpr_build'
  └──<Input Exception> Invalid argument 'opr': The echo number has an error
> 1 / 0
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1383: at function
'Exn_div'
  └──<Math Exception> Division by zero: The divisor can't be zero
> sqrt(-1)
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1629: at function
'MFunc_ckarg'
  └──<Math Exception> Square root of negative number: -1
> fact(7, 5)
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1646: at function
'MFunc_ckarg'
  └──<Function Exception> Argument mismatch in function 'fact': the function
should have 1 arguments but received 2 arguments.
> div(9,4
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1912: at function
'MathExpr_build'
  └──<Input Exception> Invalid argument 'expr': The function call needs a right
parenthesis
> sin(15)
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:2035: at function
'MathExpr_build'
  └──<Function Exception> Cannot invoke function 'sin': no such math function
> fact()
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1646: at function
'MFunc_ckarg'
  └──<Function Exception> Argument mismatch in function 'fact': the function
should have 1 arguments but received 0 arguments.
```

5. 完善的内存管理：Valgrind[5] 是一个开源的一款用于内存调试、内存泄漏检测以及性能分析的软件开发工具，对于 C/C++的程序开发非常有帮助，下面是使用 Valgrind 检测程序运行中的内存泄漏的一个示例，经多次检验程序在正常退出后不会发生任何内存泄漏。

Example (Memory check).

```
$valgrind -s --leak-check=full ./minibc
==71909== Memcheck, a memory error detector
```

[5]Valgrind homepage: https://valgrind.org/

```
==71909== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==71909== Using Valgrind-3.24.0 and LibVEX; rerun with -h for copyright info
==71909== Command: ./minibc
==71909==
                      _       _ __
              ____ ___  (_)___  (_) /_  _____
             / __ `__ \/ / __ \/ / __ \/ ___/
            / / / / / / / / / / / /_/ / /__
           /_/ /_/ /_/_/_/ /_/_/_.___/\___/

====== Interactive Mode (type "quit" or 'q' to leave) ======
> 1 / 99
0.0101
> 1.39g
 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:478: at function
'__Exn_fromstr'
  └──<Math Exception> Cannot convert 'str' to math type 'ExtendedNumber': not a
digit
        1.39g
            ^

 [WARNING]: /home/zaddle/code/CPP/project1/src/math.c:1874: at function
'MathExpr_build'
  └──<Input Exception> Invalid argument 'opr': The echo number has an error
> q
=======================================================
Free the memory and exit...
=======================================================
Bye!
==71909==
==71909== HEAP SUMMARY:
==71909==     in use at exit: 0 bytes in 0 blocks
==71909==   total heap usage: 81 allocs, 81 frees, 3,220,032 bytes allocated
==71909==
==71909== All heap blocks were freed -- no leaks are possible
==71909==
==71909== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

§3. Module Implementation 模块实现

§3.1. 基础建设

§3.1.1. 预备知识: 宏编程

宏操作是一种在预处理阶段进行的文本替换操作，它可以帮助我们简化代码，提高代码的可读性。在 C 语言中，我们可以使用 `#define` 关键字来定义宏，例如最常用的方法是定义一个常量，例如

```
#define E 2.71828
```

除了这些基础的技巧，我们还可组合 `#`、`##` 等文本操作符来实现更丰富的功能。例如我们可以定义一个宏来实现变量名的输出

```
#define PRINT_VAR(var) printf(#var " = %d\n", var)
```

操作符会将后面的参数转换为字符串，因此 PRINT_VAR(x) 会被展开为 printf("x = %d\n", x)。而连接两个字符串的操作符 ## 可以帮助我们实现更复杂的宏操作，例如我们可以定义一个宏来实现函数名的前缀修饰

```
#define FUNC_PREFIX(func) my_ ## func
```

而 ## 操作符会将两个参数先分割，然后强制以文本的形式连接在一起，比如 FUNC_PREFIX(add) 会被展开为 my_add。

如果仅仅如此，宏操作的功能就太过于简单了，C99 标准后引入的__VA_ARGS__操作符可以帮助我们实现可变参数的宏操作,这在我们熟悉的 printf 函数中就有广泛的应用。例如我们可以定义一个宏来接受一个格式化字符串和可变参数，然后输出到标准输出流

```
#define MYPRINTF(format, ...) printf(format, ##__VA_ARGS__)
```

在这里加入 ## 操作符是为了处理可变参数为空的情况，这样我们就可以使用 MYPRINTF("Hello, World!\n") 来输出 Hello, World! 到标准输出流。

其他在 C 标准库中定义的宏操作符还有__FILE__、__LINE__、__FUNCTION__等，它们分别对应当前运行代码所在的文件名、代码所在的行数以及代码所处的函数名，可以帮助我们输出调试信息，例如

```
#define DEBUG_PRINT(fmt, ...) fprintf(stderr, "%s:%d:%s(): " fmt, __FILE__, __LINE__, __FUNCTION__, ##__VA_ARGS__)
```

__attribute__ 是 GCC 的一个扩展[6]，它可以帮助我们在函数定义时添加一些属性，这些扩展属性常用的有

- __attribute__((noreturn)): 表示函数不会返回，例如 exit() 函数
- __attribute__((aligned(n))): 表示函数的内存对齐方式，例如 __attribute__((aligned(16))) 表示函数的内存对齐方式为 16 字节，在定义结构体时也可以使用这个属性
- __attribute__((packed)): 表示函数的内存对齐方式为 1 字节，这样可以减少内存的使用，但是会增加内存访问的时间
- __attribute__((constructor)): 表示函数在程序启动时会被调用，可以用来初始化一些全局变量
- __attribute__((destructor)): 表示函数在程序结束时会被调用，可以用来释放一些资源
- __attribute__((always_inline)): 与常用的 inline 函数修饰符相比，这个函数扩展表示函数会被强制内联，可以保证减少函数调用的开销，但是会增加代码的体积

§3.1.2. 异常处理

一个好的项目离不开优秀的异常处理与反馈机制，我们应该预料到用户的一些不合法输入，以及程序内部的一些错误，尤其是对于表达式解析以及多层计算过程中可能出现的错误，我们应该给出友好的提示信息，帮助用户更好地理解程序的运行情况。

一个初等的思想是，编写大量的 C 函数来处理各种异常情况，然后在主函数中调用这些函数，例如

```
void handle_divide_by_zero(char* msg) {
    printf("Error: divide by zero!");
    printf("Details: %s", msg);
}
```

[6]Refernce: https://gcc.gnu.org/onlinedocs/gcc/index.html#toc-Extensions-to-the-C-Language-Family

```
// ...

int main() {
    // ...
    if (b == 0) {
        handle_divide_by_zero("divend cannot be zero!");
        return -1;
    }
    // ...
}
```

但如果我们想往错误输出中加入更多的参数来帮助用户理解错误发生的位置，函数的调用就会变得非常繁琐。这时我们联想到了上面提到的宏操作符__VA_ARGS__，通过传入格式化字符串和可变参数，再加上__FILE__、__LINE__、__FUNCTION__，我们可以实现一个更加通用的异常处理函数，例如

```
/// @file: error.h
#include <stdio.h>
#define ___ERR_OUT(fmt, ...) do { \
    fprintf(stderr, "%s:%d:%s(): " fmt, __FILE__, __LINE__, __FUNCTION__,
##__VA_ARGS__); \
    fprintf(stderr, "\r\n"); \
} while(0)

#define __WARNING(fmt, ...) do { \
    ___ERR_OUT("[WARNING]: ") \
    ___ERR_OUT(fmt, ##__VA_ARGS__); \
} while(0)

// ...

#define MATH_URECG_OP(__op) do { \
    ___WARNING("Unrecognized operator: %s", __op);
```

其中不带下划线的宏是对外暴露的宏。我们顺便还将各种异常的等级进行划分，例如 WARNING、ERROR、NOTICE 等，这样我们就可以根据异常的等级来决定程序的运行状态，例如 WARNING 可以继续运行，ERROR 可以终止程序，NOTICE 提示用户一些不严重的问题。

不止如此，实际上如今的 Linux 上的常用程序中，颜色输出也是一种用户友好的处理方式，在此前的学习中，我了解到 UNIX 系统中的终端输出是可以通过控制字符来实现颜色输出的，比如知名的 ANSI 控制字符

- \x1b[0m: 重置颜色
- \x1b[31m: 红色
- \x1b[32m: 绿色
- \x1b[33m: 黄色
- \x1b[34m: 蓝色
- …

以对应颜色的 ANSI 字符开始的字符串序列在终端的前景色都会显示为指定颜色，如果想要取消着色，则使用\x1b[0m 结束。这些控制字符可以预先在头文件定义为宏，然后在输出提示信息的方法中使用即可。

§3.2. 任意精度数值类-ExtendedNumber

§3.2.1. 基本结构体定义

如何设计一个可以任意指定精度的数值类型？我们很自然地联想到二进制数在计算机中的存储，也就是维护一个数组 digit，使数组模仿 little-endian 的寻址方式从内存低位到内存高位存储 0 或 1，负数采用 two's completement 的形式。但是这种实现方式过于繁琐，首先在 C 中最小的数据类型 unsigned char 都需要占用 1byte，每一位只存 0/1 未免有些太浪费内存；其次二进制的表达方式对人类并不友好，在实现后续计算很容易造成可读性下降。

我们可以参考 gnu-bc 中/h/number.h 中对 bc 计算器中使用的高精度数结构体的字段分配：

```
typedef enum {PLUS, MINUS} sign;

typedef struct bc_struct *bc_num;

typedef struct bc_struct
    {
      sign  n_sign;
      int   n_len;  /* The number of digits before the decimal point. */
      int   n_scale;  /* The number of digits after the decimal point. */
      int   n_refs;     /* The number of pointers to this number. */
      bc_num n_next;  /* Linked list for available list. */
      char *n_ptr;  /* The pointer to the actual storage.
         If NULL, n_value points to the inside of
         another number (bc_multiply...) and should
         not be "freed." */
      char *n_value;  /* The number. Not zero char terminated.
         May not point to the same place as n_ptr as
         in the case of leading zeros generated. */
    } bc_struct;
```

在这里面，结构体 bc_struct 使用枚举类型 sign 作为数值的符号位表示，同时维护 n_len 作为数值中小数点之前数字的位数，n_scale 为小数点之后数字的位数，n_refs 为引用计数，n_ptr 作为实际存储十进制位的数组指针，为了便于在函数中引用传参，在开头 typedef 了 bc_struct 的指针类型为 bc_num。

这启发我们可以采用类似的设计，当然为了支持任意精度我们可以改进一下字段

```
typedef struct {
    char* digits;   /* The pointer to valid digits array of the number*/
    int length;     /* The length of the number */
    int decimal;    /* The refernce position of the decimal point */
    int sign;       /* The sign of the number */
    int precision;   /* The precision of the number */
    Exnerr error;   /* The error code of the number */
} Exn_struct, *Exn;
```

decimal 的引入是为减少占位零的存储，因为如果要存下 1e200 这样的数值而需要至少 201bytes 的空间过于浪费，所以我们只需要维护数值的有效数字位和数量级即可。precision 是这个数值可能的最大有效数字位数，也是一开始申请动态内存的依据。 而 error 字段用于标识数值在计算中出现的错误，便于在函数返回时对返回值进行检查再判断后续动作。

§3.2.2. 字符串解析

有了基本的结构体，我们就要考虑如何从用户输入字符串解析出高精度数值，首先需要确定怎样的数值表达是合法的，我们可以用下列正则表达式来表示一个合法的数值

```
[+-]?([0-9]+(\.[0-9]+)?|[0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?
```

然后按照正则表达式的状态转移来编写一个解析函数，在这里我们选择先寻找`E/e`的位置，如果没有找到，我们便按照`[+-]?([0-9]+(\.[0-9]+)?|[0-9]*\.[0-9]+)`的正则表达式来解析数值，如果找到了`E/e`，我们便按`[+-]?[0-9]+`的正则表达式来解析指数部分。我们可以使用`strchr()`函数来寻找`E/e`的位置， 然后将底数和指数部分分开，分别解析。

```
NO_EXPORT Exn __Exn_fromstr( const char* str, int prec )
{
    if (str == NULL || str[0] == '\0') return NULL;
    Exn num = Exn_create(prec);
    if (num == NULL) return NULL;

    int len = strlen(str);
    int digit_count = 0;
    bool has_decimal = false;
    bool has_digit = false;
    int pos = 0;

    /* skip the leading spaces */
    while (pos < len && isspace(str[pos])) {
        pos++;
    }

    /* extract the sign */
    if (str[pos] == '-') {
        num->sign = -1;
        pos++;
    } else {
        num->sign = 1;
        if (str[pos] == '+') {
            pos++;
        }
    }

    /* skip the leading zeros */
    while (pos < len && str[pos] == '0') {
        pos++;
    }

    /* This is an zero  */
    if (pos == len || pos == len - 1 && str[pos] == '.') {
        Exn_cpy(&num, Exn_zero);
        return num;
    }

    /* extract the digits */
    for (; pos < len; pos++) {

        if (str[pos] == '.') { /* encounter a decimal point */
            if (has_decimal) {
                num->error = EXTENDED_NUM_INVALID_INPUT;
                MATH_CVT_FAIL("str", "ExtendedNumber", "too many decimal points");
                WAVY(str, pos, 1, ANSI_PURPLE);
```

```
                return num;
            }
            has_decimal = true;
            num->decimal = digit_count;
        } else if (isdigit(str[pos])) { /* encounter a digit */
            has_digit = true;
            if (digit_count < prec) {
                num->digits[digit_count++] = str[pos];
            } else {
                digit_count++;
            }
        } else {
            num->error = EXTENDED_NUM_INVALID_INPUT;
            MATH_CVT_FAIL("str", "ExtendedNumber", "not a digit");
            WAVY(str, pos, len - pos, ANSI_RED);
            return num;
        }
    }

    if (!has_digit) {
        Exn_cpy(&num, Exn_zero);
        return num;
    }

    if (!has_decimal) {
        num->decimal = digit_count;
    }

    if (digit_count > prec) MATH_PREC_LOSE(prec, digit_count);

    num->length = digit_count;
    Exn_normalize(&num);
    return num;
}
```

在这里我们主要通用的解析字符串的函数为：

```
Exn atoExn( const char* str, int prec )
{
    if (str == NULL || str[0] == '\0') return NULL;

    int len = strlen(str);
    int e_pos = -1; /* pointer to find 'e' or 'E' symbol */

    for (int i = 0; i < len; i++) {
        if (str[i] == 'e' || str[i] == 'E') {
            e_pos = i;
            break;
        }
    }

    if (e_pos == -1) return __Exn_fromstr(str, prec);

    /* extract the base */
    char* base_str = (char*)malloc(e_pos + 1);
    if (base_str == NULL) {
        MEM_ALLOC_FAIL(&base_str, "The base string allocation failed");
```

```
        return NULL;
    }
    strncpy(base_str, str, e_pos);
    base_str[e_pos] = '\0';
    Exn base = __Exn_fromstr(base_str, prec);
    free(base_str);

    if (!base || base->error != EXTENDED_NUM_OK) {
        return base;
    }

    /* extract the exponent */
    int exponent = 0;
    int exp_sign = 1;
    int exp_start = e_pos + 1;

    if (exp_start < len) {
        if (str[exp_start] == '+') {
            exp_start++;
        } else if (str[exp_start] == '-') {
            exp_sign = -1;
            exp_start++;
        }

        for (int i = exp_start; i < len; i++) {
            if (isdigit(str[i])) {
                exponent = exponent * 10 + Ch2val(str[i]);
            } else {
                base->error = EXTENDED_NUM_INVALID_INPUT;
                // .. some error handling
                WAVY(str, i, len - i, ANSI_CYAN);
                return base;
            }
        }
    }

    exponent *= exp_sign;
    base->decimal += exponent;
    Exn_normalize(&base);
    return base;
}
```

> **Note** (Precision loss case).
> 如果用户指定的精度小于实际数值的精度，我们无法将所有数值存储下来，这里选择的是将多余的数值截断掉，然后提示用户至少需要的精度。

§3.2.3. 数值的加法与减法

基于 two's-completement 的计算机存储的二进制数是很容易进行加法与减法的，无论符号位是正数还是负数，直接将两个数相加按位相加即可，然后根据符号位的值来判断结果的符号位。然而我们的自建类 Exn 并不是相应的 ten's-completement 的模式，因此在进行加法与减法时，我们需要考虑符号位的影响。

一种可行的方案是(假设为高精度加法, 减法为加法的逆操作):

- 如果 a 与 b 的 sign bit 相同，则直接忽略符号位进行加法，然后再将结果的符号位设置为 a 或 b 的符号位
- 如果 a 与 b 的 sign bit 不同，则需要判断绝对值的大小，较大的数减去较小的数，然后将结果的符号位设置为较大的数的符号位

在 gnu/bc 中也是类似的实现思路，我们首先需要定义 cmp 函数来比较两个数的大小，然后根据大小关系来决定加法的操作。对于两个高精度数值的比较实现较为朴素，在此不再赘述。

§3.3. 多位乘法优化

我们准备了两种优化方案来实现高精度数值的乘法，一种是 Karatsuba 算法，另一种是快速傅里叶变换算法。另外包括朴素多位乘法的实现。在这里分别测试这三种算法的效率。

远程 SSH 测试环境为南科大计算服务器 [7]，系统配置及 GXX 版本信息如下：

Item	Details
System	Ubuntu 20.04.2 LTS
OS Kernel	5.4.0-205-generic
gcc-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
g++-version	(Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
CPU	Intel(R) Xeon(R) E5-2680 v4 CPU @ 2.30GHz
CPU-cores	4 cores
CPU-cache-size	35840 KB
Memory	251.77GB

测试框架为 Google Tests-benchmark 框架 [8]，测试代码如下：

```
#include "math.h"
#include <benchmark/benchmark.h>
#include <iostream>
using namespace std;

static void mul_plain_speed(benchmark::State &state) {
    int size = state.range(0);
    for (auto _ : state) {
        state.PauseTiming();
        Exn num1 = Exn_rand(size);
        Exn num2 = Exn_rand(size);
        int len = num1->length + num2->length;
        int sign = num1->sign * num2->sign;
        state.ResumeTiming();
        Exn result = __Exn_mul_pl(num1, num2, len, sign);
        state.PauseTiming();
        if (result == NULL) {
            fprintf(stderr, "Failed to multiply extended numbers\n");
            return;
        }
        Exn_release(&num1);
```

[7] 原本为本学期计算机视觉课程给学生分配的服务器，~~没想到正好能上传 project 代码跑测试。~~

[8] see details in doc https://google.github.io/benchmark/

```
        Exn_release(&num2);
        Exn_release(&result);

        benchmark::ClobberMemory();
    }
    state.SetComplexityN(size);
}

BENCHMARK(mul_plain_speed)
    ->RangeMultiplier(8)
    ->Range(8, 1<<16)
    ->Unit(benchmark::kMicrosecond)
    ->Complexity(benchmark::oNSquared);

static void mul_karatsuba_speed(benchmark::State &state) {
    int size = state.range(0);
    for (auto _ : state) {
        state.PauseTiming();
        Exn num1 = Exn_rand(size);
        Exn num2 = Exn_rand(size);
        int len = num1->length + num2->length;
        int sign = num1->sign * num2->sign;
        state.ResumeTiming();
        Exn result = __Exn_mul_karatsuba(num1, num2, len, sign);
        state.PauseTiming();
        if (result == NULL) {
            fprintf(stderr, "Failed to multiply extended numbers\n");
            return;
        }
        Exn_release(&num1);
        Exn_release(&num2);
        Exn_release(&result);

        benchmark::ClobberMemory();
    }
    state.SetComplexityN(size);
}

BENCHMARK(mul_karatsuba_speed)
    ->RangeMultiplier(8)
    ->Range(8, 1<<16)
    ->Unit(benchmark::kMicrosecond)
    ->Complexity(benchmark::oNLogN);

static void mul_fft_speed(benchmark::State &state) {
    int size = state.range(0);
    for (auto _ : state) {
        state.PauseTiming();
        Exn num1 = Exn_rand(size);
        Exn num2 = Exn_rand(size);
        int len = num1->length + num2->length;
        int sign = num1->sign * num2->sign;
        state.ResumeTiming();
        Exn result = __Exn_mul_fft(num1, num2, len, sign);
        state.PauseTiming();
        if (result == NULL) {
```

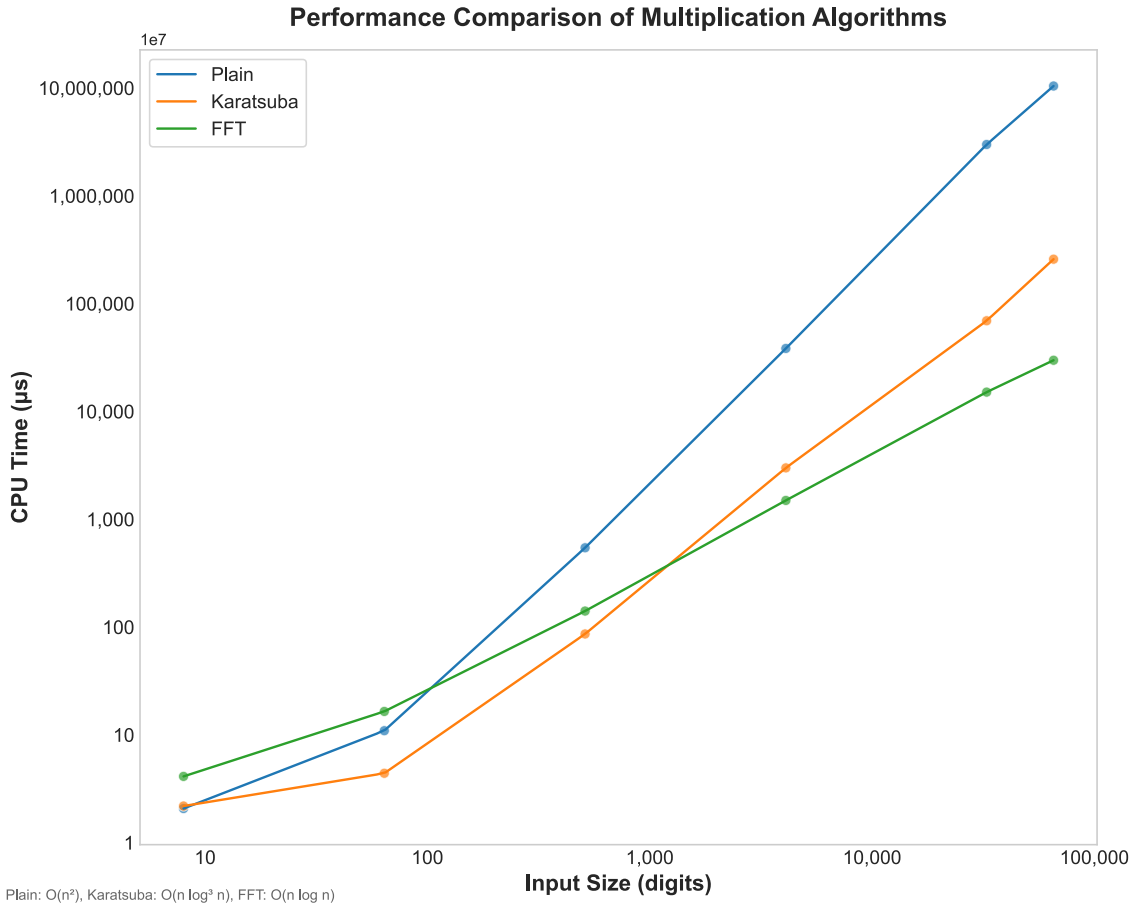
```
            fprintf(stderr, "Failed to multiply extended numbers\n");
            return;
        }
        Exn_release(&num1);
        Exn_release(&num2);
        Exn_release(&result);

        benchmark::ClobberMemory();
    }
    state.SetComplexityN(size);
}

BENCHMARK(mul_fft_speed)
    ->RangeMultiplier(8)
    ->Range(8, 1<<16)
    ->Unit(benchmark::kMicrosecond)
    ->Complexity(benchmark::oNLogN);

BENCHMARK_MAIN();
```

编译命令为:

```
g++ -Wall -O3 -std=c++14 \
-I ./benchmark/include \
-L ./build/src test_mul.cpp math.c \
-lstdc++ -D_GLIBCXX_USE_CXX11_ABI=1 -pthread \
-lbenchmark -lm -o test_mul
```

§3.3.1. 测试结果

测试结果通过`./test_mul --benchmark_format=csv > result.csv`导出到`result.csv`文件中，并使用`python`的`matplotlib`库进行可视化[9]



从三种实现方法的CPU时间曲线来看，在输入规模为100以下的情况下，三种方法的时间复杂度比较接近，但是随着输入规模的增大，FFT算法的时间复杂度增长速度明显低于朴素乘法和Karatsuba算法，虽然没有测试更大规模的输入，但是从曲线的趋势来看，FFT算法的时间复杂度应该是最低的。从Google BenchMark的时间复杂度分析来看，朴素乘法的时间复杂度为$0.5 \cdot O(n^2)$，Karatsuba算法的时间复杂度为$300 \cdot O(n \log n)$，FFT算法的时间复杂度为$5 \cdot O(n \log n)$，这与现有结论是基本一致的。

§3.3.2. 总结

以上结果说明了FFT算法在高精度数值乘法中的优越性，尤其是在大规模输入的情况下，FFT算法的时间复杂度远远低于朴素乘法和Karatsuba算法。因此在实际应用中，我们应该尽量选择FFT算法来实现高精度数值的乘法。当然，在小规模输入时，可以选择朴素乘法来实现，复杂度常数较低，效率更高。算法切换的条件可以确定为输入规模大于某个阈值(比如在这里可以选择100)时，选择FFT算法，否则选择朴素乘法。

§3.4. Newton迭代法构建数学函数

[9]The experiment is preformed eight times, and the average time is taken as the final result.

Newton-Raphson迭代法是一种求解方程近似解的方法，它的基本思想是通过不断迭代来逼近方程的解。对于一个方程$f(x)=0$，我们可以通过迭代的方式来求解x的值，迭代公式为

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

其中$f'(x)$为$f(x)$的导数。这种方法的优点是很容易通过`while`循环实现，当精度满足要求时，我们可以停止迭代。但是这种方法的缺点是当迭代方程比较复杂时，如果初始值选择不当，可能会导致迭代结果不收敛，甚至发散， 非常影响效率。

Newton 迭代法的伪代码展示如下:

```
function Newton-Iteration(func, x)
    while Check-with-precision(x) do
        update x using the iteration function func(x)
    return x
```

§3.4.1. 数值除法

我们在此前已经实现了数值的加法、减法以及乘法，然而对于除法，我们不能再直接进行按位操作，因为除法是一个不可逆的操作。一种选择是通过长除法的方式来实现[10]，但是这种方法的效率非常低，需要反复移位和减法操作，复杂度最终会达到$O(n^2)$。因此我们需要寻找一种更高效的方法来实现除法[4]。

如果我们想求出两个数 n_1, n_2 的商，可以选择先求出 $\frac{1}{n_2}$ 的值再与 n_1 相乘，这样就可以得到商。乘法的效率我们已经通过前面的讨论得到了提高，而且精度不会在计算过程的中间阶段丢失。而求倒数的方法我们可以通过牛顿迭代法来实现，即求解方程 $f(x)=\frac{1}{n_2}-x=0$ 的解。我们可以通过迭代的方式来逼近这个解，迭代公式为

$$\begin{aligned}x_{n+1}&=x_n-\frac{f(x_n)}{f'(x_n)}\\&=x_n(2-n_2x_n)\end{aligned}$$

而初值的选择我们可以结合反函数 $f(x)=\frac{1}{x}$ 的图像来进行选择，假设除数 $n_2=m\cdot 10^n$ ， 则初值为 $x_0=10^{-n}$ 比较好。

§3.4.2. 开方函数

对于开方函数有很多经典的实现，比如Q_rsqrt[5]，它最初是为了快速计算游戏图形学中常用的参数 $d=\frac{1}{\sqrt{x^2+y^2+z^2}}$ 而发明出来的

```
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 level hacking
  i = 0x5f3759df - (i >> 1);                 // what the fuck?
  y = * ( float * ) &i;
```

[10]Typically by GMP https://gmplib.org/manual/Basecase-Division

```
  y = y * (threehalfs - ( x2 * y * y ) );    // 1st iteration
  y = y * (threehalfs - ( x2 * y * y ) );    // 2nd iteration, this can be removed

  return y;
}
```

先忽略 `1.5`, `0x5f3759df` 等 Magic numbers 是怎么来的，我们再来回忆浮点数与长整数在计算机中的存储

$$L = 2^{23} \cdot E + M$$

$$F = \left(1 + \frac{M}{2^{23}}\right) \cdot 2^{E-127}$$

对浮点数表示公式取对数变成，

$$\log_2 F = \log_2\left(1 + \frac{M}{2^{23}}\right) + E - 127$$

根据函数的极限知识 $\lim_{x\to 0^+} \log_2(1+x) = x$, $\lim_{x\to 1^-} \log_2(1+x) = x$ 可以将上面的等式简化为

$$\begin{aligned}\log_2 F &\approx \frac{M}{2^{23}} + E - 127 \\ &= \frac{1}{2^{23}}(M + E \cdot 2^{23}) - 127 \\ &= \frac{L}{2^{23}} - 127\end{aligned}$$

我们可以近似地认为浮点数的二进制表示 L 是数值本身 F 呈对数关系

$$\log_2 F \approx L$$

那么这条语句的作用就可以理解为在取 `y` 的对数赋值给 `i`

```
i = * ( long * ) &y;
```

为什么要繁琐地先取指再强制指针类型转换然后解引用的原因也很好理解， 如果只是简单的`(long)`的强制转换会将二进制值改变从而损失精度， 而仅仅转换指针类型实际上两者指向的是同一个内存数据，这样就能达成上述取对数的目标。

有了对数操作 $\log_2(x)$，我们重新审视我们最开始求平方根的逆的目标：对数的引入会简化开方这一复杂操作为

$$\begin{aligned}\log_2\left(\frac{1}{\sqrt{y}}\right) &= -\frac{1}{2}\log_2(y) \\ &\approx -(i \gg 1)\end{aligned}$$

除以 2 的操作我们可以等价为算术右移操作，在此之前我们在计算过程中忽略的近似引起的偏差值可以通过详细的计算来进行修正[11]，这也就是 `0x5f3759df` 这个 Magic number 的作用。

而下面的两行代码则是通过牛顿迭代法来逼近开方的值，这里的迭代公式为

[11]For detailed derivation, see: https://zhuanlan.zhihu.com/p/445813662

$$x_{n+1} = x_n \left(\frac{3}{2} - y \frac{x_n^2}{2} \right)$$

这样得出的平方根值已经非常接近真实值，而且计算效率也非常高。

虽然 Q_rsqrt 的方法相比如今的硬件针对优化已经意义不大，我们仍可以从中获得一些启发，比如可以使用我们构建好的牛顿迭代法来实现自建函数库中的 sqrt 函数。我们采用相同的“曲线救国”的方式，不直接使用 $f(x) = \sqrt{x}$ 作为目标方程 (因为对应的迭代方程涉及到高精度除法，会引起精度和效率损失)，而是先迭代方程 $f(y) = \frac{1}{\sqrt{y}}$，然后仅需要一次高精度除法就能得到答案。具体实现如下：

```
Exn Exn_sqrt(const Exn num) {
  if (!num || !num->digits) {
    INPUT_ARG_FAIL("operand", "The operand can't be NULL");
    return NULL;
  }
  if (num->error != EXTENDED_NUM_OK) {
    INPUT_ARG_FAIL("operand", "The operand has an error");
    return NULL;
  }
  Exn result = NULL;
  if (Exn_iszero(num)) {
    Exn_cpy(&result, Exn_zero);
    return result;
  }
  int len = num->length * 2 + 1;
  int prec = MAX(num->precision, num->decimal);
  int bound = MAX_DIGITS / 2; /* the bound of the iteration */

  /* select initial value */
  int shift = -num->decimal / 2;
  if (num->decimal % 2 != 0) {
    shift = -num->decimal / 2 - 1;
  }
  result = Exn_shift(Exn_one, shift);
  Exn_newton(__ntiter_sqrt, bound, num, &result);
  Exn_round(&result, len, -1);

  /* invert \frac{1}{\sqrt{x}} */
  Exn inv = Exn_shift(Exn_one, -result->decimal);
  Exn_newton(__ntiter_inv, prec, result, &inv);
    Exn_release(&result);
    Exn_round(&inv, len, 0);
  return inv;
}
```

§3.5. 交互式输入处理

处理命令行参数可以直接用 C 标准库<getopt.h>进行参数解析[12]，然后将对应参数解析为我们想要的值即可。但在交互模式下考虑的就很多了，结合上面的讨论我们最好使用 fgets()函数读取用户交互信息，这个函数会将用户输入的字符按长度全部读出来，包括可能的空格，括号与逗号分隔符 这时就可能面临以下这些输入情况：

[12]Usage: https://www.gnu.org/software/libc/manual/html_node/Getopt.html

- 合法性未知的二元表达式，即`<oprand1> <op> <oprand2>`
- 合法性未知的函数调用表达式，即`<funcname>(arg1, arg2, ...)`，在这里我打算判定的宽松一点，参数字符与左右括号以及逗号分隔符之间可以留任意长的空格，并且保证参数过多或过少都不会引起程序错误
- 合法性未知的回显表达式，即`<echonum>`，这个模式是方便用户查看 ExtendedNum 的解析与格式化功能是否正常
- 'q'或者"quit"，当读到直接回收内存并结束程序
- 不属于上面任何一种的非法表达式，这时候可能要根据一些输入的特征提醒用户可能希望使用的输入表达式

要分别处理好如此多情况，狂堆 if-else 分支肯定不优雅也不符合项目可读性规范。我们希望能够合并一些情况，然后再延迟处理，这样也会使代码相互解耦，扩展性较高。又联系到表达式的解析，我们很自然地想到编译原理中的递归下降的概念，~~不过这个项目并没有实现完整的 Parser 功能，意味着解析不了 nested-expression，这里只是借用一下思想~~，在这里我们定义一个 MathExpression (记作E)，很显然它是一个 non-terminal，在这里我们的一元回显表达式、二元表达式和函数调用表达式都属于E，而唯一的 terminal 为 Extendednumber (记作num) 则 E 的 CFG 文法为

```
E -> num
E -> num ["+"/"-"/"x"/"/"] num
E -> funcname "(" [num] {"," num} ")"
```

然后分别写三个结构体来表示这三种模式，由于 C 中没有继承的概念，我们只能用 `union` 存储子结构体，搭配 `enum` 表示类型来实现多态，这样我们只需要使用 MathExpr 结构体来构建一个表达式，然后在需要求值时使用一个统一的 `eval` 函数来处理即可。

```c
typedef enum {
    MATH_EXPR_BINOP = 0,
    MATH_EXPR_FUNC,
    MATH_EXPR_ECHO
} MExprType;
typedef struct {
    MExprType type; // type of the expression
    union {
        BinOprExpr* binop; // binary operand expression
        MathFunc* func; // math function
        Exn echo; // echo expression
    } expr; // expression
} MathExpr;
```

这样做的好处还有另外一个，就是可以任意增加新的数学函数，只需要在 `build` 和 `eval` 函数中增加对应的分支即可。[13]

§3.6. 其他碎碎念

§3.6.1. 快速幂

快速幂也是 ACM 竞赛中常用的一种技巧[6]，它的基本思想是将指数分解为二进制数，然后通过不断平方来求解幂次方。这样可以将时间复杂度从$O(n)$降低到$O(\log n)$。在这里我们可以用快速幂来实现高精度数值的幂运算，这样可以大大提高效率。快速幂的代码由于本身十分简洁，所以实现起来没有遇到太大的困难。

[13]The best implementation of function invocation is to build a symbol table and use a hash table to store the function pointer, but we don't have too many functions to implement, so we don't need to do that.

```
Exn ksm(Exn num, int exp) {
    Exn result = Exn_shift(Exn_one, 0);
    result->precision = num->precision;
    if (result == NULL) {
        MEM_ALLOC_FAIL(&result, "The extended number allocation failed");
        return NULL;
    }
    Exn a = Exn_shift(num, 0);
    while (exp > 0) {
        Exn tmp = NULL;
        if (exp & 0x1) {
            Exn tmp_ = Exn_mul(result, a);
            Exn_mv(&result, &tmp_);
        }
        tmp = Exn_mul(a, a);
        Exn_mv(&a, &tmp);
        exp >>= 1;
    }
    Exn_release(&a);
    Exn_normalize(&result);
    return result;
}
```

§3.6.2. 上下文命令导航

许多现代的终端程序中都支持的一个方便的功能是上下文命令导航，比如我们在 bash 中按下上/下键就会在光标处切换历史命令，如果想要在这个小的计算器程序的交互模式实现这个功能，就必须了解终端是如何处理各种 IO 流的。

<termios.h>这个标准库[14]提供了一些结构体和函数来控制终端的输入输出，其中最基本的结构体是 struct termios，它包含了终端的所有属性，

```
struct termios
{
    unsigned short c_iflag; /* 输入模式标志*/
    unsigned short c_oflag; /* 输出模式标志*/
    unsigned short c_cflag; /* 控制模式标志*/
    unsigned short c_lflag; /*区域模式标志或本地模式标志或局部模式*/
    unsigned char c_line; /*行控制line discipline */
    unsigned char c_cc[NCC]; /* 控制字符特性*/
};
```

为了实现上下文导航，我们需要对每个字段重新配置

- c_lflag 用于控制本地模式，我们需要禁用 ISIG 标志，这样就可以禁用信号处理，即不会因为按下 Ctrl+C 而中断程序。
- c_iflag 用于控制输入模式，我们需要禁用 ICANON 标志，这样就可以实现非规范模式，即每次输入一个字符就立即返回给程序；同样我们需要禁用 ECHO 标志，这样就可以实现不回显输入字符；最后需要禁用 ICRNL 标志，这样就可以实现不将回车符\n 映射为换行符\r\n。
- c_cc 用于控制特殊字符，我们需要重新配置 c_cc[VMIN]和 c_cc[VTIME]，这两个字段用于控制 read 函数的行为，VMIN 表示最小读取字符数，VTIME 表示读取字符的超时时间，我们需要将 VMIN 设置为 1，VTIME 设置为 0，这样就可以实现每次只读取一个字符，不会等待超时。

[14]Termios manual, see https://www.man7.org/linux/man-pages/man3/termios.3.html

实现整个完整功能需要四步：首先需要保存每次读取的输入到历史列表，然后如果历史不为空，按上下键可以刷新当前输入，然后显示当前选中的历史命令，最后处理正常的输入。

保存历史命令的操作很简单，我们将它独立出来作为一个函数，然后在 main 函数每次读取输入时调用即可。

```
void add_to_history(const char *cmd) {
    if (strlen(cmd) == 0) return;
    if (history_count > 0 && strcmp(history[history_count-1], cmd) == 0) {
        return;
    }
    if (history_count >= MAX_HISTORY) {
        free(history[0]);
        for (int i = 0; i < history_count - 1; i++) {
            history[i] = history[i+1];
        }
        history_count--;
    }

    history[history_count] = strdup(cmd);
    history_count++;
}
```

我们在这里特殊处理了前后输入相同的情况，这样可以避免重复存储相同的历史命令。

如何刷新当前输入呢？我们可以将这个操作分为两个部分，一个是清除当前输入并退格到行首，另一个是将历史命令写入到当前输入中， 清除函数如下：

```
void clear_line(int position) {
    printf("\r");
    for (int i = 0; i < position + 2; i++) {
        printf(" ");
    }
    printf("\r");
}
```

写入函数如下：

```
void refresh_line(const char *prompt, char *buffer) {
    printf("\r%s%s", prompt, buffer);
    fflush(stdout);
}
```

对于方向键的处理，我们至少需要读入三个字符，分别是 ESC，[，A 或 B，这样我们就可以判断用户按下的是上键还是下键，相应的移动历史命令的指针刷新当前输入即可。

```
else if (c == 27) {
            char seq[3];

            if (read(STDIN_FILENO, &seq[0], 1) == 0) continue;
            if (read(STDIN_FILENO, &seq[1], 1) == 0) continue;

            if (seq[0] == '[') {
                if (seq[1] == 'A') {
                    if (history_count > 0 && current_history > 0) {
                        current_history--;
                        clear_line(position);
                        strcpy(buffer, history[current_history]);
                        position = strlen(buffer);
```

```
                refresh_line(prompt, buffer);
            }
        } else if (seq[1] == 'B') {
            if (current_history < history_count) {
                current_history++;
                clear_line(position);
                if (current_history == history_count) {
                    buffer[0] = '\0';
                    position = 0;
                } else {
                    strcpy(buffer, history[current_history]);
                    position = strlen(buffer);
                }
                refresh_line(prompt, buffer);
            }
        }
```

这样就基本完成了上下文导航的功能，这个功能对于用户来说是非常方便的，可以快速选择之前的输入，避免重复输入。

§3.6.3. 测试函数注册

这个部分属于在随手写测试函数的时候发现的，如果我们有很多测试函数，每编写一个测试就要在 main 函数中调用一次，而且想要定位 到某个测试函数的位置也不是很方便，比如

```
void test1() {
    // test code
}

void test2() {
    // test code
}

// ...

void testN() {
    // test code
}

int main() {
    test1();
    test2();
    // ...
    testN();
    return 0;
}
```

这时候我们想来一开始讲到的宏技巧和 GNU 扩展，我们可以使用 __attribute__((constructor))来实现函数注册，这样我们只需要在 main 函数中调用一个注册函数即可，而且我们可以通过宏定义来简化注册过程，例如在 utils.h 中的定义

```
typedef struct {
    void (*func)(void);
    const char *name;
} test_entry_t; /* Test Unit Struct */
```

```c
#define MAX_TESTS 100
static test_entry_t test_registry[MAX_TESTS];
static int test_count = 0;
#define TEST(test_name) \
    static void test_##test_name(void); \
    static void __attribute__((constructor)) register_##test_name(void) { \
        register_test(test_##test_name, #test_name); \
    } \
    static void test_##test_name(void)
void register_test(void (*test_func)(void), const char *name)
 {
    if (test_count < MAX_TESTS) {
        test_registry[test_count].func = test_func;
        test_registry[test_count].name = name;
        test_count++;
    }
}

void run_all_tests(void) {
    printf("Running %d tests...\r\n\n", test_count);

    for (int i = 0; i < test_count; i++) {
        printf("Test '%s': ", test_registry[i].name);
        test_registry[i].func();
        printf("\r\n");
    }

    printf("All tests completed.\r\n");
}
```

这样想要编写某个测试函数只需要在函数定义前加上 TEST 宏即可， 然后在 main 函数中加上一句 run_all_tests()测试所有样例。

```c
TEST(some_test)
{
  // test something
}

int main()
{
    run_all_tests();
}
```

§4. Evaluation and Foresee 评估与展望

虽然这只是一个一个功能简单的小型计算器程序，但是在实现过程中涉及到了各种软件开发的基本思想，比如从确定需求到设计基本结构，规范化错误输出，模块化设计，再到往程序中加入一些辅助功能，比如上下文导航， 一开始因为接触到宏编程所以查阅了大量资料学习了一些宏技巧，结果发现可以用在很多模块中，比如测试函数注册，这毫无疑问是一个很好的简化开发流程的技巧，属于意料之外的收获。这个项目也让我对 C 语言的一些特性有了更深的理解，比如指针的使用，内存管理，结构体的设计等等。尤其是 valgrind 的使用，让我意识到内存分配和回收在 C 语言中是头等问题，这个软件在调试过程中帮助我找到了很多潜在的内存泄漏问题， 减少了很多不必要的麻烦。

在这个项目中，我稍微注意了一下代码风格，遵从 google-standard-C-style 进行各种命名，并在函数声明部分加入了尽可能详尽的文档注释。

除此之外，平时在算法课中只是从书本读到中的各种算法，比如快速幂，快速傅里叶变换等等，通过这个项目的实现， 我对这些算法有了更深的理解，也知道了它们在实际应用中的价值，比如 FFT 算法在高精度数值乘法中的应用，Q_rsqrt 中的 bit hacking 技巧等等。

不过一开始的设计有些不够完善，比如在设计高精度数值的结构体时，如果将数值数组反位存储，那么在进行乘法运算时 可以减少很多数组的中间转换，这样可以大大提高效率，这些都会成为经验教训，以后在设计类似的项目时会更加谨慎， 在实现底层积木的时候要考虑到上层的需求，这样才能更好地设计出一个高效的程序。

最后处理交互式输入的部分引入了表达式解析的思想，虽然没有实现完整的 Parser 框架，但是也让我对编译原理中的一些概念有了更深的理解，比如 CFG 文法，递归下降等等，这些都让我在实际例子中对这些较晦涩艰深的理论知识有了更深的认识。

这个项目可以提升的点还有很多，比如可以实现更多的数学函数，比如三角函数，对数函数等等，可以实现更多的高精度数值运算，比如除法，开方等等，可以实现更多的交互式功能，比如变量赋值，函数定义等等，这些都是可以进一步完善的地方。 当然这个项目离现有的开源计算器软件 (gnu-bc 等) 还有很大的差距，这也让我认识自己现有的知识与能力与整个 C/C++领域乃至计算机科学领域的广度与深度之间的鸿沟， 我希望借助此项目的能够正式开启本学期深入 C/C++编程的奇幻旅途， 重新审视编程的目的究竟是什么，怎样才算真正的入门计算机科学。

Bibliography

[1] A. Karatsuba and Yu. Ofman, “Multiplication of many-digital numbers by automatic computers,” *Proceedings of the USSR Academy of Sciences*, vol. 145, 1962, [Online]. Available: http://www.ccas.ru/personal/karatsuba/divcen.pdf

[2] A. Schönhage and V. Strassen, “Schnelle Multiplikation großer Zahlen,” *Computing*, vol. 7, 1971, doi: https://link.springer.com/article/10.1007/BF02242355.

[3] W. Kahan, “IEEE Standard 754 for Binary Floating-Point Arithmetic,” *IEEE Standard*, vol. 754, Jul. 2019, doi: https://doi.org/10.1109/IEEESTD.2019.8766229.

[4] morris821028, “關於高效大數除法的那些事.” [Online]. Available: https://morris821028.github.io/2017/04/09/division/

[5] “Origin of Quake3's Fast InvSqrt().” [Online]. Available: https://www.beyond3d.com/content/articles/8/

[6] “Binary Exponentiation.” [Online]. Available: https://cp-algorithms.com/algebra/binary-exp.html#effective-computation-of-large-exponents-modulo-a-number

§A Root of unity 单位根

§AA 欧拉公式

对任意实数 x 存在

$$e^{ix} = \cos x + i \sin x$$

其中 e 为自然对数的底数，i为虚数单位。有关 i 与复数在 C 标准库中的`<complex.h>`有定义。

§AB 单位根的定义

以单位圆点为起点，单位圆的 n 等分点为终点，在单位圆上可以得到 n 个复数，假设辐角为正且最小的复数为 ω_n，称为 $\boldsymbol{n}$ 次单位根，即

$$\omega_n = \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right)$$

由欧拉公式，有

$$\omega_n^k = \cos\left(\frac{2k\pi}{n}\right) + i\sin\left(\frac{2k\pi}{n}\right)$$

特别地，对 $k = 0$和$k = n$，有

$$\omega_n^0 = \omega_n^n = 1$$

§AC 单位根的性质

$$\omega_{rn}^{rk} = \cos\left(\frac{2rk\pi}{rn}\right) + i\sin\left(\frac{2rk\pi}{rn}\right) = \omega_n^k \qquad r \in \mathbb{N}^+$$

$$\omega_n^{k+\frac{n}{2}} = \omega_n^k\left(\cos\left(\frac{n}{2}\cdot\frac{2\pi}{n}\right) + i\sin\left(\frac{n}{2}\cdot\frac{2\pi}{n}\right)\right) \qquad k \in \mathbb{N}^+$$

Appendix