Lab 7 - 高精度实数

本次Lab将通过链表存储并实现高精度实数的打印与计算。

我们将定义名为decimal的数据结构，存储"没有位数与精度限制"的高精度实数（可以是有限小数，也可以是整数）。

decimal内部将通过单向链表存储实数的有效数字（每一位有效数字都将单独存储在一个节点中），同时会存储该数的指数项（用于表示小数点的位置）以及符号位。

为了能够方便地打印、计算、比较高精度实数，我们将需要实现单向链表的反转操作。

最终，利用链表的反转操作，我们将实现高精度实数的加法、减法以及打印输出。

本次实验内容包括：

- Task 1 - 定义digit与decimal
  - 构建数据类型：`struct s_digit`, `struct s_decimal`
  - 使用 `typedef` 简化指针表述
- Task 2 - 内存分配与释放函数
  - 内存分配函数 `new_decimal`, `new_digit`
  - 内存释放函数 `free_decimal`, `free_digits`
- Task 3 - 如何得到一个decimal
  - 解析函数 `new_decimal_by_string` 将一个合规字符串转化为高精度实数
  - 调试用的打印函数 `debug_print_decimal` 用于观察得到的decimal的结构
- Task 4 - 链表反转函数 `_reverse_digits`
- Task 5 - 打印输出函数 `print_decimal`
- Task 6 - 无符号加法 `_add_positive` & 无符号减法 `_sub_positive`
- Task 7 - 比较绝对值大小 `_compare_abs`
- Task 8 - 有符号加减法 `add`, `sub`
- Homework
  - 完成上述所有函数实现，并进行测试
  - 撰写一份实验报告
    - 阐述decimal与digit的关系
    - 阐述是否可以进一步优化decimal和digit空间使用率，分析可行思路与利弊
    - 思考是否可能通过递归函数print_digit，在不逆置链表的情况下实现打印输出，简述设计思路

代码框架说明

本次实验课所有代码都将在lab7.c文件中实现与测试。

lab7.c文件中包含了代码框架，提供了每个task需要完成的函数原型，以及一些测试用例。

在逐步推进实验的过程中，可以选中多行代码，并使用 `Ctrl[Command]+/` 快捷键**批量取消注释**后续部分的代码。

Task 1 - 让我们定义digit和decimal

```c
typedef struct s_digit
{
    char data; // 0-9
    struct s_digit *next;
} *digit;

typedef struct s_decimal
{
    // e.g. -3.14 is represented as -314*10^{-2}
    // is_negative = 1, [4 (head) -> 1 -> 3], e = -2
    char is_negative; // 1 for negative, 0 for positive
    digit head;       // linked list starting from the **LOWEST** digit
    int e;            // exponent of 10
} *decimal;
```

为什么我们需要高精度实数?

为什么0.1+0.2=0.30000000000000004而1.1+2.2=3.3000000000000003？ - Johan约翰的回答 - 知乎

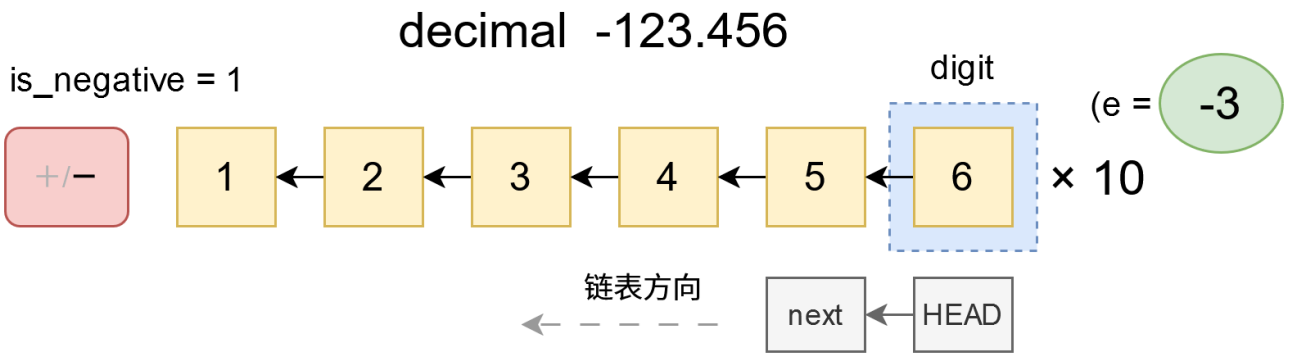
> 注：这也是一般不能直接用==来判断两个普通浮点数是否相等的原因。

高精度实数的 存储方式 与 定义

由于十进制的小数在转化为二进制存储时会产生误差，因此我们只要定义一个数据结构，通过十进制的方式存储实数，就可以避免误差的产生。

我们认为，一个高精度实数由以下几个部分组成：

- 符号位（0代表正数，1代表负数）
- 有效数字链表（每一位有效数字都将单独存储在一个节点中）
- 指数项（$\times 10^{?}$，间接表示了小数点的位置）



Task 2 - 内存分配与释放函数

内存分配：为结构体分配新的内存空间，初始化该结构体，并返回指向该内存空间的指针：

```
// Allocate memory for a new decimal
// Note: the decimal has no digit yet, we consider it as NaN
decimal new_decimal()
{
    decimal d = (decimal)malloc(sizeof(struct s_decimal));
    d->is_negative = 0;
    d->head = NULL;
    d->e = 0;
    return d;
}

// Allocate memory for a new digit
digit new_digit(char data, digit next)
{
    digit d = (digit)malloc(sizeof(struct s_digit));
    d->data = data;
    d->next = next;
    return d;
}
```

内存释放：释放链表中所有节点的内存空间，以及释放decimal结构体的内存空间：

```
// Free the memory of a linked list of digits
void free_digits(digit head)
{
    digit cur = head;
    while (cur != NULL)
    {
        digit next = cur->next;
        free(cur);
        cur = next;
    }
}

// Free the memory of a decimal
void free_decimal(decimal d)
{
    free_digits(d->head);
    free(d);
}
```

Task 3 - 该你了：如何得到一个decimal

在Task 3中，我们将实现一个简单的转换函数`decimal new_decimal_by_string(char *s)`，并在main函数中进行测试。

该函数的功能是将传入的字符串s转化为一个新的高精度实数，并返回指向该数的decimal类型指针。

```
// Parse a string to set the decimal
decimal new_decimal_by_string(char *s)
{
    decimal d = new_decimal();

    // TODO: Task 3

    return d;
}
```

注意：

- 我们假设输入的字符串是合法的，即不会出现空字符、非数字字符、多个小数点等情况。
- 有效数字上（即链表末尾）的0可以保留，例如`0.00123456`可能在链表末尾出现3个0节点，目前我们暂时不需要对这些节点进行删除。

以下函数已在代码框架中提供，可用于查看decimal的内部结构：

```
void debug_print_decimal(decimal d)
{
    printf("is_negative: %d, e: %d\n", d->is_negative, d->e);
    printf("digits: [head]");
    for (digit cur = d->head; cur != NULL; cur = cur->next)
        printf("%d->", cur->data);
    printf("NULL\n");
}
```

正常情况下，字符串`-12.3456`应该被转化为结构如下的decimal：

```
is_negative: 1, e: -4
digits: [head]6->5->4->3->2->1->NULL
```

下面是一些可用的测试用例，这些测试用例都可能用于测试new_decimal_by_string函数：

```
char *test_parser[] = {
    "123456",
    "12.3456",
    "12345.6",
    "0.123456",
    "0.00123456",
    "123456.0",
    "-123456",
    "-12.3456",
    "-12345.6",
    "-0.123456",
    "-0.00123456",
    "-123456.0",
    ".123456",
    "-.123456",
    "123456.",
    "-123456.",
};
```

Task 4 - 链表反转

为了能够顺序打印decimal，在Task 4中，我们将首先实现一个链表反转函数`digit _reverse_digits(digit head, int *len_output)`，并在main函数中进行测试。

该函数的功能是反转以head为首节点的链表，返回反转后链表的首节点地址（即原先的尾节点地址），同时将该链表的长度（节点数量）存储在len_output指向的变量中。

```
// Reverse the linked list of digits, returning the new head
digit _reverse_digits(digit head, int *len_output)
{
    // TODO: Task 4
}
```

正常情况下，将-12.3456得到的decimal中的链表进行反转，并将head指向反转函数返回的新的头节点，能够得到如下结构的decimal:

```
Original:
is_negative: 1, e: -4
digits: [head]6->5->4->3->2->1->NULL

Reversed:
is_negative: 1, e: -4
digits: [head]1->2->3->4->5->6->NULL

Length: 6
```

Task 5 - 真正的打印输出

在Task 5中，我们将正式实现一个打印函数`void print_decimal(decimal d)`，并对其进行测试。

该函数的功能是打印高精度实数d的值。

例如:

```
is_negative: 1, e: -6
digits: [head]6->5->4->3->2->1->NULL
```

结构如上的decimal应被打印为-0.123456。

提示:

- 打印之前，需要先将链表反转（因为我们需要由最高位至最低位依次输出）。打印之后，还需要将链表转回来，所以要记得存好翻转后得到的新的"头"节点。
- 翻转的过程中，我们可以一并得到链表的长度。
- 如果 **指数项的相反数 大于等于 链表长度**，意味着小数点将出现在有效数字的最左侧。如：$123456 \times 10^{-8} = 0.00123456$，此时我们需要首先打印**0.**以及有效数字前相应需要补足的**0**。
- 如果 **指数项的相反数 小于 链表长度，且 指数项 小于0**，意味着小数点将出现在有效数字中间。如：$123456 \times 10^{-4} = 12.3456$。在依次打印有效数字时，我们需要注意在合适的位置打印小数点。
- 如果 **指数项 大于等于0**，意味着小数点出现在有效数字最右侧（此时该实数不存在小数部分）。如：$123456 \times 10^{2} = 12345600$，我们需要按需补足末尾的0。此处不会遇到这种情况

Task 6 - 无符号加减法

在实现有符号decimal的加减法之前，我们首先需要实现无符号加减法。

（将无符号加减法与比大小相结合就能够实现有符号的加减法）

_add_positive函数将会考虑a与b的有效数字与指数项（即只考虑a与b的绝对值），将其相加后存储在c中。

提示：

- 在a和b的链表表头处（即最低位有效数字处）补0，并同时调节指数项，使得a和b的小数部分相互对齐。补足的0以及调节过的指数项在加法计算完成后会需要恢复至开始时的状态。
- 小数对齐后，就可以从低位开始逐渐按数位实现加法了。需要注意进位项carry的存在。计算得到的数位可以依次插入至c的链表头部（这种实现方法在计算完成后需要反转c的链表），或是插入至c的链表尾部（可以思考一下该如何实现？）
- 当a或b任意一方的高位有效数字消耗完毕，而另一方仍存在有效数字时，需要将剩余的有效数字继续加至c中（注意进位项可能仍然存在）
- 如果a与b的有效数字都消耗完毕，但进位项仍然非0，则将进位项单独添加至c中。
- c的计算结果中有效数字的最高位可能存在多个0，此时在翻转c的链表之前可以先删去头部的0。（注意，如果链表只剩下一个0节点时，c的值为0，此时不能继续删除最后的0节点。）若是链表只剩下一个0节点，此时还需要对其进行归一化，将指数项与符号位置0（因为我们不区分正0与负0的区别，也不允许0存在多位有效数字）
- 最终，反转c的链表完成后，不要忘记恢复(删去)开始时为了对齐a与b在最低位有效数字上添加的0，并恢复指数项。

_subtract_positive与_add_positive的实现思路类似，需要将进位项carry修改为借位项borrow。

另外需要注意的是，减法实现时有效数字的按位计算可能产生类似321-319=002的情况，在计算完成后、反转链表前，需要删除链表头部的0。

Task 7 - 比较绝对值大小

在实现有符号decimal的加减法之前，我们还需要实现一个比较绝对值大小的函数。

_compare_abs函数将会比较a与b的绝对值大小，返回值-1、0、1分别表示$|a| < |b|$、$|a| = |b|$、$|a| > |b|$。

提示：

- 反转链表，从而从高位有效数字开始比较。（与此同时，记下链表长度）
- 移除最高位上多余的0
- 根据链表长度和指数项的关系，判断两数整数部分有效数字位数，如果不同则直接返回结果
- 如果整数部分位数相同，此时两数链表头部在最高位有效数字上已经对齐，可以直接从高位有效数字开始比较，直至找到不同的节点，返回比较结果
- 如果两数共有部分的所有有效数字都相同，但有一数的最低位有效数字未消耗完毕，则该数更大
- 反之两数相等。
- 注意：在任何情况下return之前，都不要忘记再次反转链表恢复原状。

Task 8 - 有符号加减法

在Task 8中，我们将实现有符号decimal的加减法。

add函数将会计算a+b的值，存储至新的高精度实数中并返回其指针。

sub函数将会计算a-b的值，存储至新的高精度实数中并返回其指针。

提示：

- 首先比较a与b的同异号情况。同号加法与异号减法都可以通过调用无符号加法实现计算。
- 异号加法与同号减法可以通过比较绝对值大小，调整参数顺序，再通过调用无符号减法计算结果。

`lab7.c`中提供了一些测试用例，可以用于测试`add`与`sub`函数。（计算结果的打印输出可能与测试用例中的字符串不同，但计算结果应该是正确的）

Homework 作业

提交代码`lab7.c`（根据给定的代码框架）：

- 完成上述所有函数代码实现（Task3-8），并进行测试（课件PPT会在课后上传至Canvas）

提交实验报告`report.pdf`，报告中需要对以下问题进行回答：

- 阐述`decimal`与`digit`的关系
- 阐述是否可以进一步优化`decimal`和`digit`空间使用率，分析可行思路与利弊
- 思考是否可能通过递归函数`print_digit`，在不逆置链表的情况下实现打印输出，简述设计思路

将代码与实验报告压缩为`zip`文件后上传。

缺交、漏交、迟交会导致作业分数会受到一定的影响。

因此在提交前，请检查提交文件的正确性与完整性，确保提交的代码可以通过所有测试用例并正常工作、报告回答了上述问题并且能够正常显示。