面向对象程序设计
实验1 整数

55240425

屈熙宸

- 任务一、
- 设计一个类封装（参考C语言）6种位运算(函数接口封装)，给出调用测试。
  基于位运算模拟10进制的+、-、*3个运算，给出调用测试。

程序源码+运行结果

```h
//bitOperations.h

#include <iostream>

class BitOperations {
public:
    int bitAnd(int a, int b);

    int bitOr(int a, int b) ;

    int bitXor(int a, int b);

    int bitNot(int a) ;

    int bitLeftShift(int a, int n) ;

    int bitRightShift(int a, int n);

};
```

```cpp
//bitoperations.cpp

#include "bitOperations.h"
#include <iostream>

int BitOperations::bitAnd(int a, int b) {
    return a & b;
}

int BitOperations::bitOr(int a, int b) {
    return a | b;
}

int BitOperations::bitXor(int a, int b) {
    return a ^ b;
}

int BitOperations::bitNot(int a) {
    return ~a;
}

int BitOperations::bitLeftShift(int a, int n) {
    return a << n;
}

int BitOperations::bitRightShift(int a, int n) {
    return a >> n;
}
```

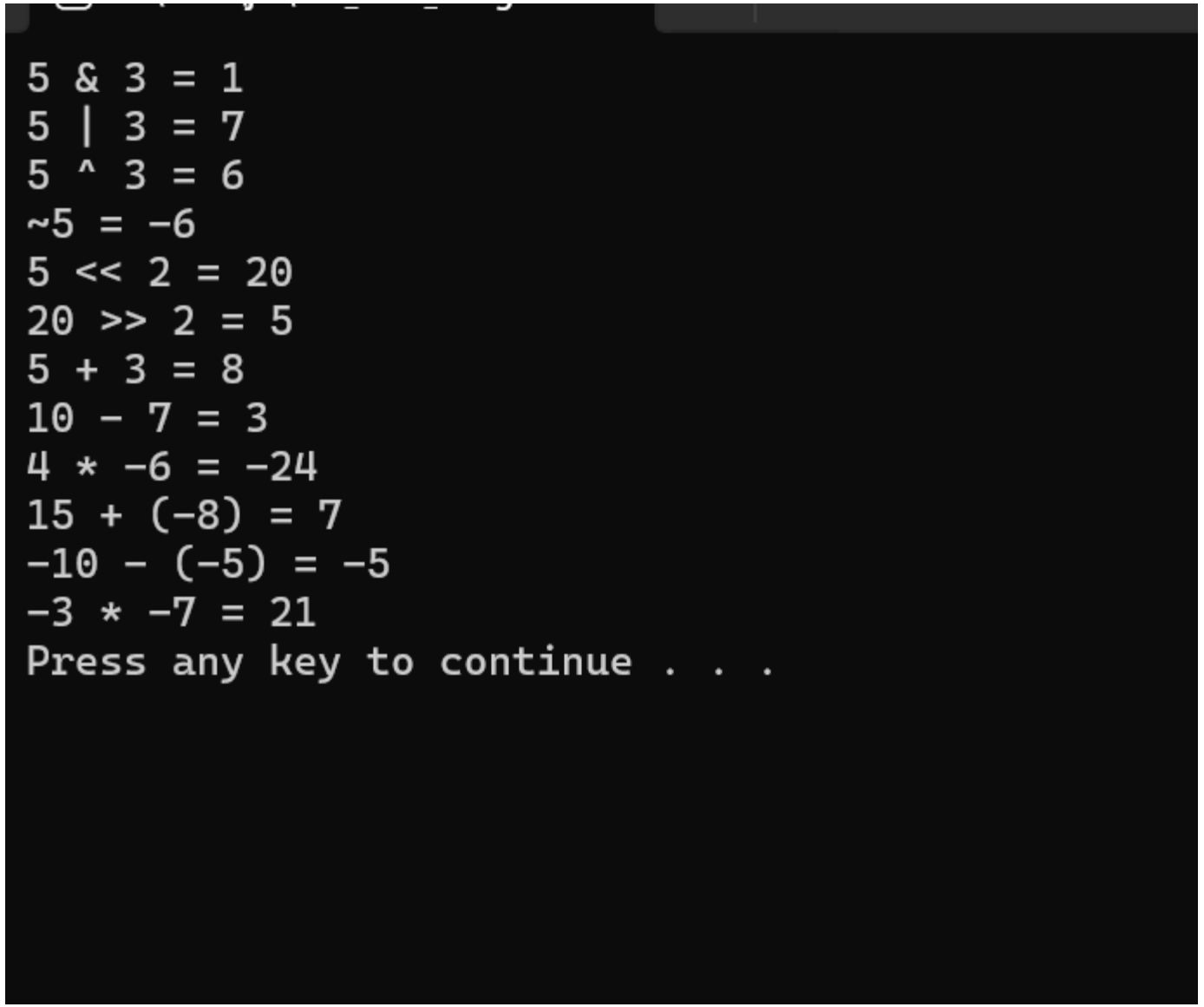
test01 > include > C CalcOperation.h > ...

```h
//CalcOperations.h

class CalcOperations {
    public:
        // 位运算加法
        int add(int a, int b) ;

        // 位运算减法
        int subtract(int a, int b) ;

        // 位运算乘法
        int multiply(int a, int b) ;
    };
```

src > CalcOperation.cpp > ...

```cpp
int CalcOperations::add(int a, int b) {
    BitOperations bitOps;
    while (b != 0) {
        int sum = bitOps.bitXor(a, b);        // 无进位相加结果
        int carry = bitOps.bitLeftShift(bitOps.bitAnd(a, b), 1); // 进位值
        a = sum;
        b = carry;
    }
    return a;
}

int CalcOperations::subtract(int a, int b) {
    BitOperations bitOps;
    return add(a, add(bitOps.bitNot(b), 1)); // a + (-b)
}

int CalcOperations::multiply(int a, int b) {
    BitOperations bitOps;
    bool negative = bitOps.bitXor(a < 0, b < 0); // 确定符号
    a = a < 0 ? add(bitOps.bitNot(a), 1) : a;
    b = b < 0 ? add(bitOps.bitNot(b), 1) : b;

    int result = 0;
    while (b != 0) {
        if (bitOps.bitAnd(b, 1))
            result = add(result, a);
        a = bitOps.bitLeftShift(a, 1);
        b = bitOps.bitRightShift(b, 1);
    }
    return negative ? add(bitOps.bitNot(result), 1) : result;
}
```

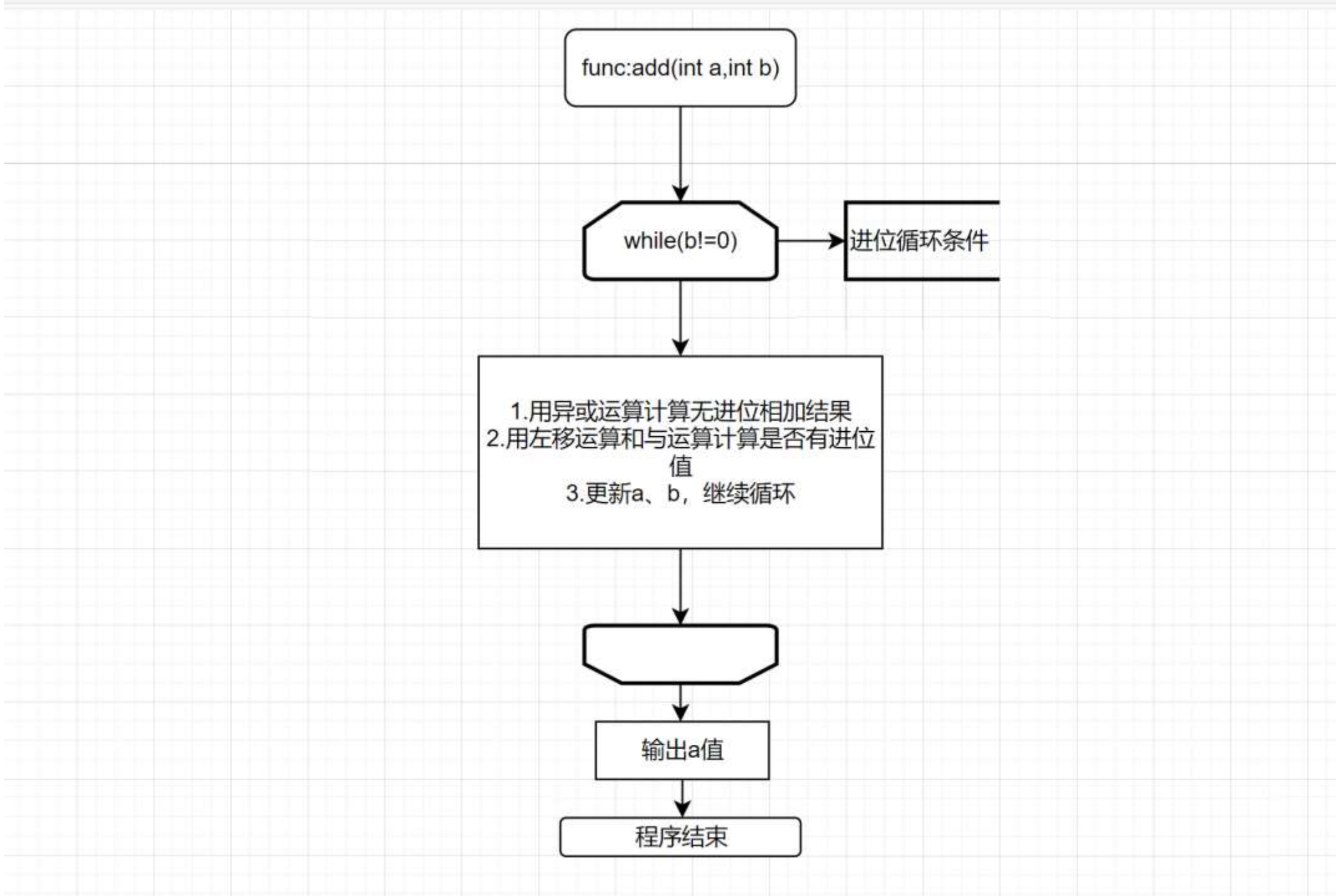
```cpp
#include <iostream>
#include "CalcOperations.h"
#include "bitOperations.h"

int main()
{

    // 测试位运算
    BitOperations bitOps;

    // 按位与
    int andResult = bitOps.bitAnd(5, 3);
    std::cout << "5 & 3 = " << andResult << std::endl;

    // 按位或
    int orResult = bitOps.bitOr(5, 3);
    std::cout << "5 | 3 = " << orResult << std::endl;

    // 按位异或
    int xorResult = bitOps.bitXor(5, 3);
    std::cout << "5 ^ 3 = " << xorResult << std::endl;

    // 按位取反
    int notResult = bitOps.bitNot(5);
    std::cout << "~5 = " << notResult << std::endl;

    // 左移位
    int leftShiftResult = bitOps.bitLeftShift(5, 2);
    std::cout << "5 << 2 = " << leftShiftResult << std::endl;

    // 右移位
    int rightShiftResult = bitOps.bitRightShift(20, 2);
    std::cout << "20 >> 2 = " << rightShiftResult << std::endl;

    //测试十进制
    CalcOperations calcOps;

    // 测试加法
    int addResult = calcOps.add(5, 3);
    std::cout << "5 + 3 = " << addResult << std::endl;

    // 测试减法
    int subtractResult = calcOps.subtract(10, 7);
    std::cout << "10 - 7 = " << subtractResult << std::endl;

    // 测试乘法
    int multiplyResult = calcOps.multiply(4, -6);
    std::cout << "4 * -6 = " << multiplyResult << std::endl;

    // 测试更多案例
    std::cout << "15 + (-8) = " << calcOps.add(15, -8) << std::endl;
    std::cout << "-10 - (-5) = " << calcOps.subtract(-10, -5) << std::endl;
    std::cout << "-3 * -7 = " << calcOps.multiply(-3, -7) << std::endl;

    return 0;
}
```

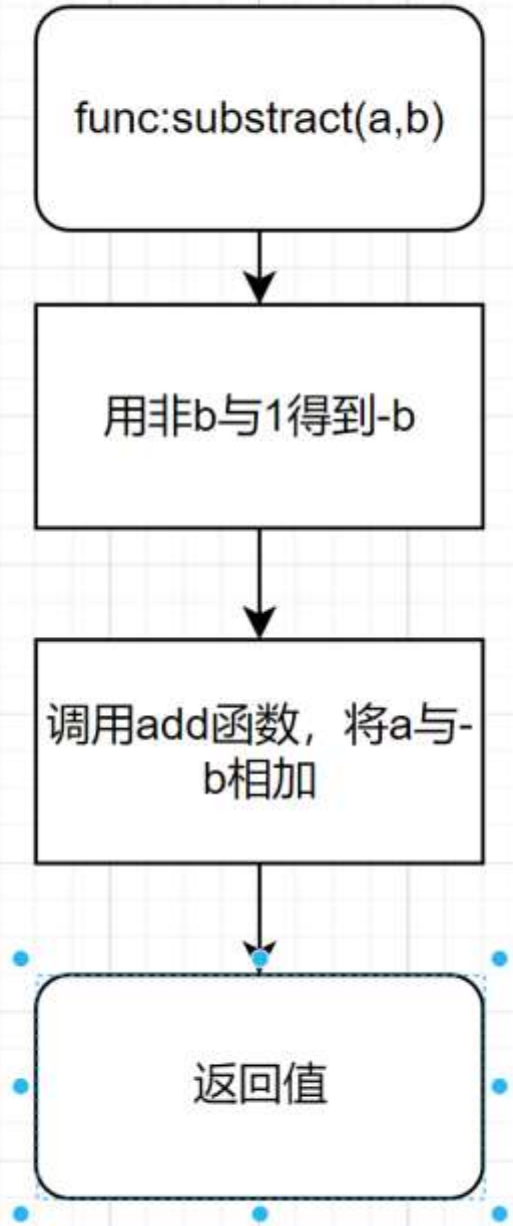

5 & 3 = 1
5 | 3 = 7
5 ^ 3 = 6
~5 = -6
5 << 2 = 20
20 >> 2 = 5
5 + 3 = 8
10 - 7 = 3
4 * -6 = -24
15 + (-8) = 7
-10 - (-5) = -5
-3 * -7 = 21
Press any key to continue . . .

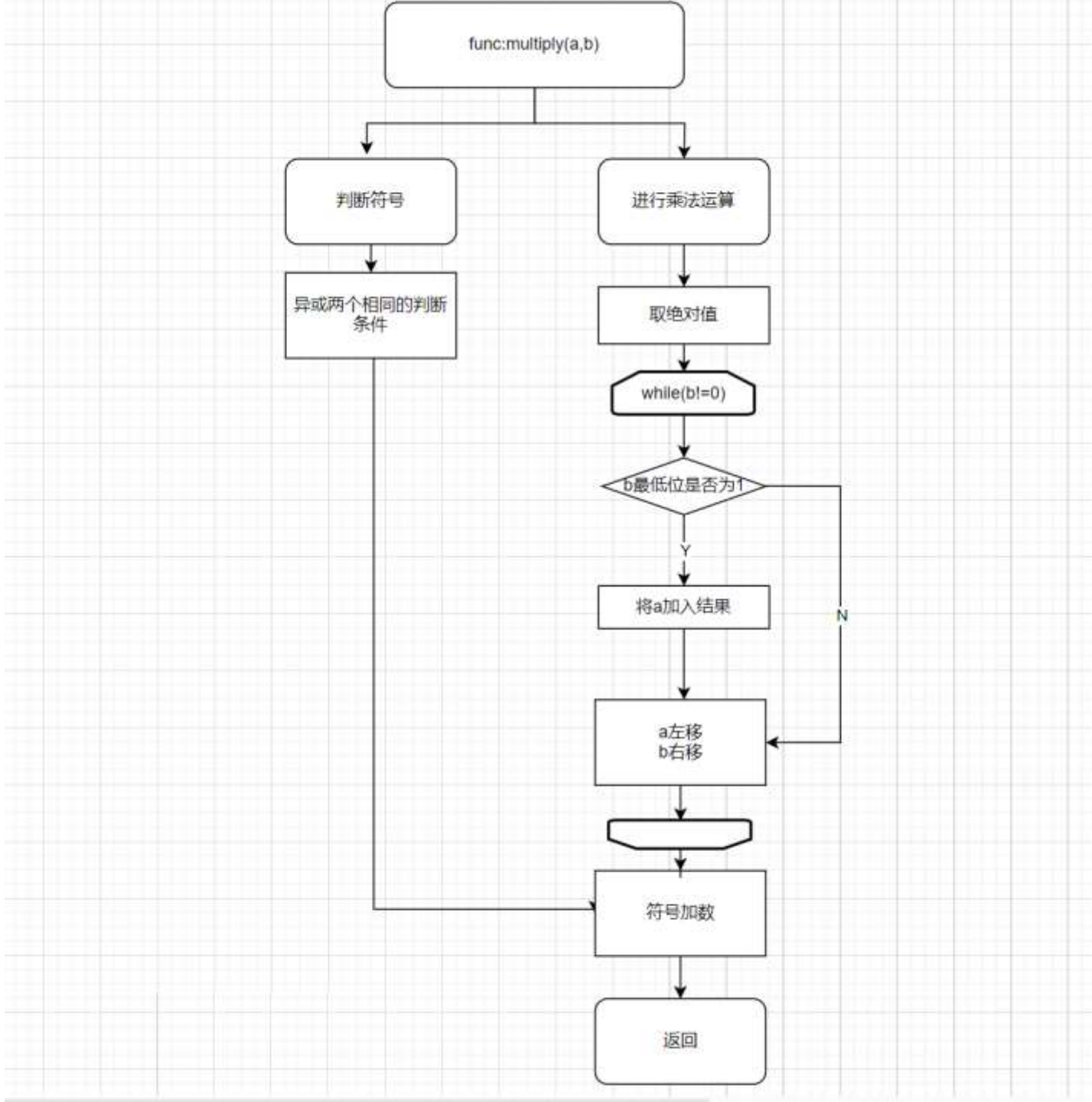
1.分析题目要求

- 1.首先将二进制的六种位运算用函数实现并且封装在bitOperations类中
- 2.利用bitOperations类中的六种位运算函数实现十进制中+，-，*，并且封装进calcOperations类中
- 3.在主程序中模拟并检验结果

2.流程图


func:add(int a,int b)
while(b!=0)
进位循环条件
1.用异或运算计算无进位相加结果
2.用左移运算和与运算计算是否有进位值
3.更新a、b，继续循环
输出a值
程序结束


func:substract(a,b)
用非b与1得到-b
调用add函数，将a与-b相加
返回值


func:multiply(a,b)
判断符号
异或两个相同的判断条件
进行乘法运算
取绝对值
while(b!=0)
b最低位是否为1
Y
N
将a加入结果
a左移
b右移
符号加数
返回

3.分析难点

- 难点在于乘法的实现
- 解决：1.将符号和数位分开
- 2.对于数位：将被乘数按乘数的每一位权重展开相加
  - 检查数位是否为一，为一则乘被乘数。
  - b右移一位判断下一位，a左移一位加权

4.分析

- 1.优点：
  - 基本实现了所要求的功能

- 2.缺点：
  - 范围太小，可能会导致溢出
  - 没有溢出报错提醒

5.收获

- 更深刻的认识到二进制乘法的流程
- 对类封装有进一步认知

- 任务二:
  - 实现一个整数中所有数字累加和的计算。（可以和（1）独立project）
    给出1000！的计算，可以采用任意方法（注意1000！将溢出）。

- 程序源码+运行结果

```h
// sumOfDigits.h
#include <iostream>
#include <string>
#include <vector>

// 计算整数中所有数字的累加和
int sumOfDigits(int number) ;
std::vector<int> factorial(int n);
```

```cpp
// sumOfDigits.cpp

#include <iostream>
#include <string>
#include <vector>
#include "sumOfDigits.h"

int sumOfDigits(int number) {
    int sum = 0;
    number = std::abs(number); // 确保处理正数
    while (number > 0) {
        sum += number % 10; // 提取最后一位数字并累加
        number /= 10;       // 去掉最后一位数字
    }
    return sum;
}

std::vector<int> factorial(int n) {
    std::vector<int> result(1, 1);
    for (int i = 2; i <= n; ++i) {
        int carry = 0;
        for (auto& digit : result) {
            int product = digit * i + carry;
            digit = product % 10;
            carry = product / 10;
        }
        while (carry) {
            result.push_back(carry % 10);
            carry /= 10;
        }
    }
    return result;
}
```

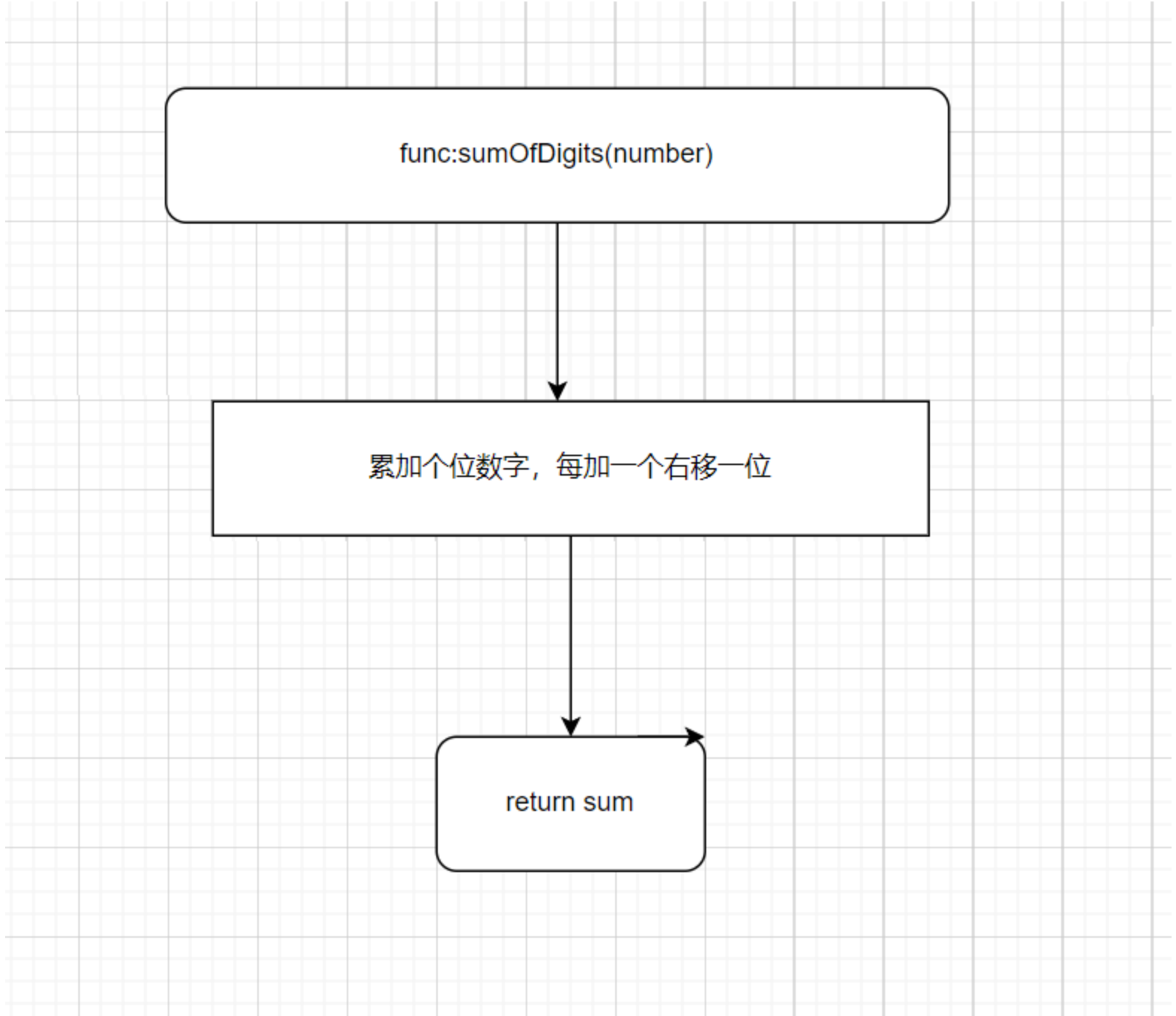
```cpp
#include <iostream>
#include <vector>
#include "sumOfDigits.h"

int main() {
    int number = 12345;
    std::cout << "数字 " << number << " 的各位数之和为: " << sumOfDigits(number) << std::endl;

    number = -9876;
    std::cout << "数字 " << number << " 的各位数之和为:  " << sumOfDigits(number) << std::endl;

// 计算1000!
    int n = 1000;
    std::vector<int> result = factorial(n);
    int sum = 0;
    for (int digit : result) {
        sum += sumOfDigits(digit);
    }

    std::cout << "1000! 的各位数之和为:  " << sum << std::endl;

    return 0;
}
```

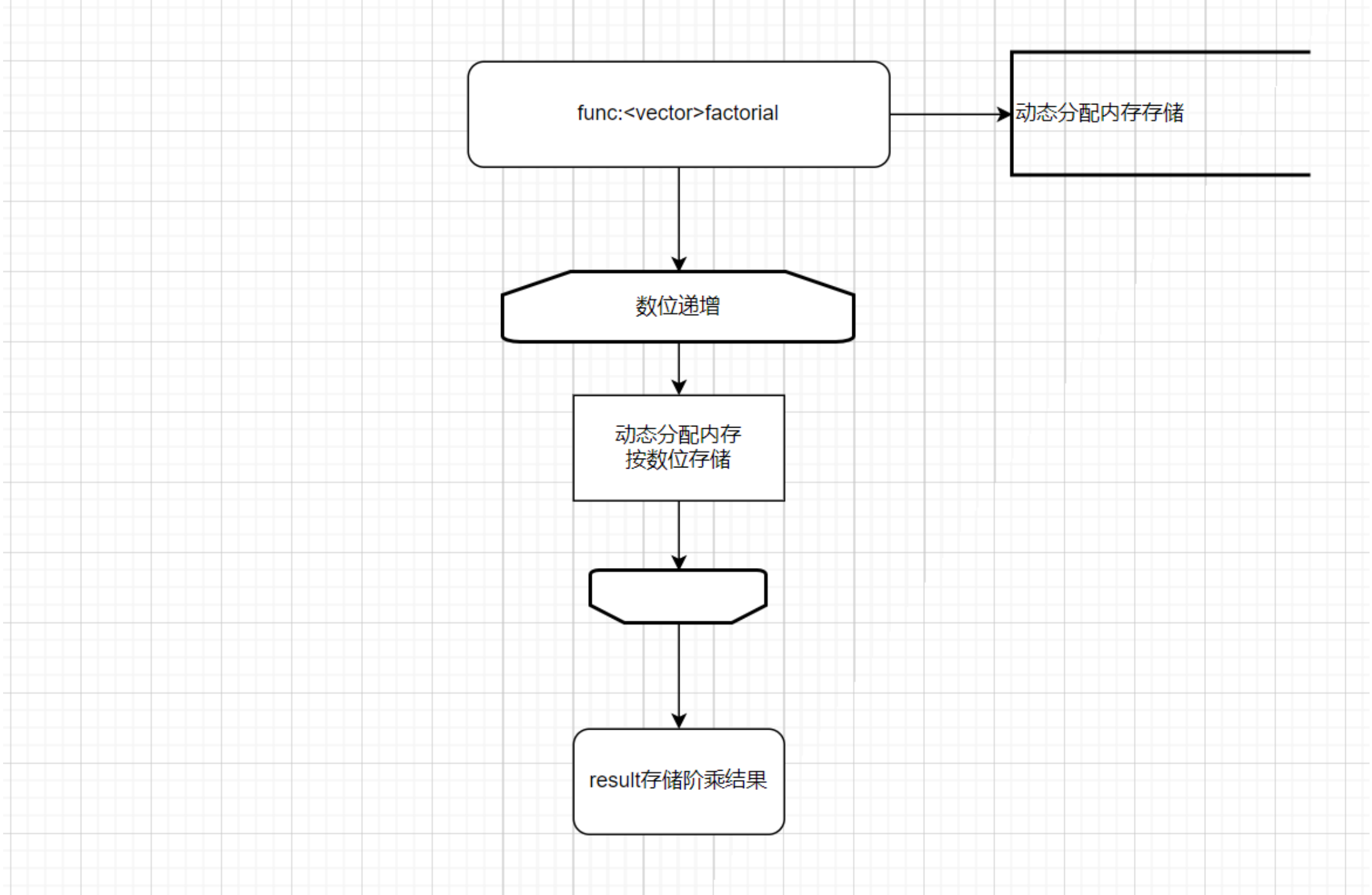
```
PS D:\code\JLU\JLU_OOP_Assignments\test01\output>
数字 12345 的各位数之和为: 15
数字 -9876 的各位数之和为:  30
1000! 的各位数之和为:  10539
```

1.分析题目要求

- 1.封装各位数字相加函数
- 2.用各种方法处理严重溢出的数（1000！）并计算其各位数字相加和

2.流程图


func:sumOfDigits(number)
累加个位数字，每加一个右移一位
return sum


func:<vector>factorial
动态分配内存存储
数位递增
动态分配内存
按数位存储
result存储阶乘结果

3.分析难点

- 难点在于1000！阶乘溢出的问题

解决：

使用vector容器动态分配内存

4.分析

- 1.优点：
  - 程序简单，易于维护

- 2.缺点：
  - 时间复杂度较高
  - 未使用简化算法

5.收获

- 学会了vector容器的使用
- 了解对于溢出问题的解决方法