Assignment 4 Pointers, Arrays and C-style Strings

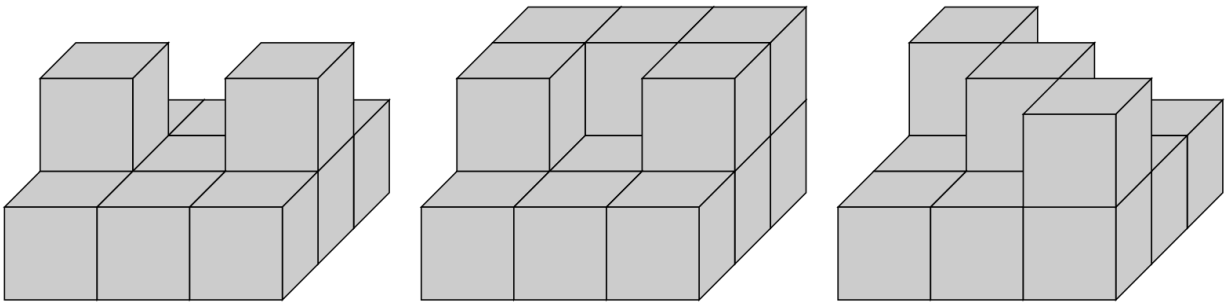
> 本次作业一共有三道题目，总分为十分，其中前两道为必做题，分数共计十分；第三道为选做题，可用于补救前两道题的错误，分数为两分，附加分数不会溢出总分。

> 为鼓励大家使用数组和指针，在本次作业中部分容器将无法使用。每道题的提示部分说明了本题无法使用哪些容器。

1. 三维模型的表面积

题目描述

一个三维模型建立在规则的 n * m 网格上，并且网格均由 1 * 1 正方形构成。在每个网格上都有一个长方体，长方体由若干个 1 * 1 * 1 的立方体搭建而成（所有长方体的底部都在同一平面上）。几个典型的模型如下图所示：



现在给出每个网格上长方体的高度，即每个网格上的长方体由多少个立方体搭建而成，要求这个三维模型的表面积是多少。

输入格式

第 1 行包含 2 个正整数 n m，为模型的长与宽。

接下来 n 行，每行 m 个数字 hnm， hnm描述第n行第m列这个网格的长方体的高度。

输出格式

输出一个非负整数，为三维模型的表面积。

数据范围

```
1 < m, n, h < 100
```

提示

本题不能使用的容器有：vector、queue、map、stack。鼓励大家使用新学的数组（array）。

建议在source/1_area/main.cpp的基础上继续往下写。

样例

输入

```
3 3
1 1 1
2 1 2
1 1 1
```

输出

```
38
```

2. 正则表达式匹配器

题目描述

正则表达式（Regular Expression）是一种描述指定格式字符串的方式，在计算机科学中有很多应用。 比如，Linux中用于查找文件里符合正则表达式的行的grep命令、文本编辑器以及浏览器中的find功能以及 Perl程序语言，均内置了正则表达式匹配器。

我们说正则表达式regex匹配一个字符串str，是指str中存在一个连续子串（可能是空串），符合regex描述的指定格式。

在本题中，你需要实现一个正则表达式匹配器，接收一个正则表达式和一个字符串，返回该字符串是否匹配该正则表达式。

正则表达式的表示形式

正则表达式有非常多的变体，下面给出本题中的正则表达式的表示形式。（注意，下文中我们使用'c'表示字符c，使用"abc"表示字符串abc，使用""表示空串）

一个正则表达式由几种构造形式拼接形成，不同的构造形式表示不同的匹配规则。

基本构造有两种：

c —— 匹配字符'c'，即寻找目标字符串中是否存在字符'c'。比如a 匹配 "a"， b 匹配 "aaabaaba"，但是b不匹配"aaaccc"。

. —— 匹配任意单个字符，即寻找目标字符串中是否存在任意单个字符。比如.匹配 "a"， .匹配"cpp"，但是.不匹配""。

正则表达式的拼接：我们可以拼接构造形式以获得新的正则表达式。比如，ac匹配字符串"aaaaacaaaaa"， 1.3匹配字符串"11345"。

高级构造有三种：

^ —— 匹配目标字符串的开头，^之后往往会拼接一个正则表达式。比如，^123 匹配 "123456"， ^1.3 匹配 "113"，但是^123 不匹配 "1123"。

$ —— 匹配目标字符串的结尾，$之前往往会拼接一个正则表达式。比如，123$ 匹配 "456123"，但是 123$ 不匹配 "1233"。

* —— 匹配在它之前的一个字符（可能是一般字符c，也可能是构造.）0或1或多次。比如，ba*c 匹配 "baaaaaaaaaaac"， ba*c 匹配 "bc"， a.*匹配任意以a开头的字符串。

输入格式

第一行是一个正则表示式，第二行是整数n，接下来n行每行是一个字符串。

输出格式

输出为n行，第i行代表第i个输入的字符串能否被输入的正则表达式匹配。若能被匹配，则在第i行输出`matched`，否则输出`unmatched`。

数据范围

0 < 输入的正则表达式的长度 < 20。

测试用例中输入的正则表达式满足的约束有：

1. 正则表达式的构造c的范围是（'A'-'Z','a'-'z','0'-'9'）。
2. 如果表达式中存在^，则^一定在表达式的开头。
3. 如果表达式中存在$，则$一定在表达式的结尾。
4. 如果表达式中存在*，则*之前的字符只可能是一般字符c或构造.。即不会出现^*、a**等情况。

0 < 输入字符串的长度 < 1000。

字符串中的字符范围与构造c相同。

提示

本题不能使用容器（string, vector、queue、map、stack）以及标准库的regex，鼓励大家使用c-style string以及char*。

本题适合使用递归解决，我们在源文件./source/2_regex/main.cpp给出了三个函数的声明， 同学们可以通过正确地实现这三个函数解决本题：

```
// match : 检查text中是否存在子串匹配regexp描述的格式
bool match(char *regexp, char *text);

// matchstart : 检查text的开头子串是否匹配regexp
bool matchstart(char *regexp, char *text);

// matchstar : 检查text的开头子串是否匹配表达式 c*regexp
bool matchstar(int c, char *regexp, char *text);
```

其中，match是完成正则表达式匹配功能的函数，matchstart和matchstar是需要用到的两个辅助函数。

1. match检查text是否存在子串能够匹配regexp。针对某个子串的匹配通过调用matchstart来检查。
2. matchstart检查传入的text的开头子串是否匹配regexp。我们需要分情况讨论regexp，以执行不同的后续匹配操作。

- 比如说，（在regexp[0] != '\0'的情况下）regexp[1] == '*'，那么regexp 可以被分解为 c*subRegexp。这时通过调用matchstar，检查text的开头子串是否匹配表达式 c*subRegexp。
- 再比如说，regexp[0] == 'a'并且text[0] == 'a'，说明text[0]成功匹配regexp[0]， 那么我们就递归调用matchstart检查text剩余的字符串的开头子串是否匹配regexp剩余的正则表达式。

3. matchstar检查text的开头子串是否匹配表达式 c*subRegexp。设 text = text1 + text2， 我们需要在text的开头子串text1匹配c*的情况下，通过调用matchstart检查text的剩余部分text2的开头子串是否匹配subRegexp。如果成立，表示text匹配 c*subRegexp。

关于使用c-style string读取输入数据的示例：

```
#include <iostream>

int main() {
    char text[1000]; // 声明c-style string
    std::cin >> text; // 读取输入数据，效果和使用string读取相同
}
```

样例

输入

```
^b*123
5
b123
123
bb123
ab123
b12345
```

输出

```
matched
matched
matched
unmatched
matched
```

3. 基于寄存器模型的ASM

在assignment3，我们尝试实现了基于stack machine的ASM。这种ASM将数据存放在栈上，通过对栈进行pop和push操作来完成计算功能。

我们在课上学习c++ abstract machine时，存储数据的结构是内存。它更贴近另一种计算模型——register machine，它的组成部分是寄存器和内存。 寄存器通常有多个，用于存放数据和对这些数据做计算。 内存可以被看作是一个很长的字节数组，其中存放了数据，每个字节的数组索引即是它的地址。寄存器中能够存放的数据远小于内存的大小。

Register machine通过读取指令，对寄存器和内存进行相应的操作，来完成计算任务。

题目描述

本题中，我们要实现的ASM包含内存和3个寄存器。

内存是一个长度为512的字节(uint8_t)数组（用 M[]表示），使用大端法存储。（关于uint8_t、uint16_t和大端法的介绍，可以参考文档最后的"补充概念"）

3个寄存器是%r1，%r2和%r3，它们分别可以存储16bit的数据（一个字节的长度是8bit）。每个寄存器又可以拆成两个更小的寄存器，每个小寄存器存储1个字节。比如，%r1可以拆成%r11和%r12，%r11存储%r1的第一个字节，%r12存储%r1的第二个字节。同理，%r2可以拆成%r21和%r22，%r3可以拆成%r31和%r32。

寄存器参考实现方式:

```
typedef uint8_t Register[2];  // 给类型uint8_t[2]起个别名为Register

// Register r; 等价于 uint8_t r[2];
// 两种方式声明的r在后续的使用中没有任何区别

void setH(Register& r, uint8_t n) {
    r[0] = n;
}

void setL(Register& r, uint8_t n) {
    r[1] = n;
}

void setR(Register& r, uint16_t n) { // 使用大端法存储n
    setH(r, n / 256);
    setL(r, n % 256);
}

uint8_t readH(const Register& r) { return r[0]; }
uint8_t readL(const Register& r) { return r[1]; }

// 可以看到，readR是setR的逆操作。
// 但是，与setR不同的是，我们首先需要将r[0]转为uint16_t，然后再做乘法。
// 因为r[0]的类型是uint8_t，r[0] * 256执行的是uint8_t上的乘法，其结果的类型还是
uint8_t,
// 因此在数学上的结果是(r[0] * 256) % (2 ^ 8)，执行mod操作是为了保证结果在uint8_t能表
```

```
示的数的范围内。
// 但如果使用了uint16_t(r[0]) * 256, uint16_6(r[0])的类型是uint16_t,
// uint16_6(r[0]) * 256 执行的是uint16_t上的乘法，其结果的类型是uint16_t,
// 以保证后续的加法操作也是uint16_t类型的。整个过程中不会有值的溢出。
uint16_t readR(const Register& r) {
    return uint16_t(r[0]) * 256 +
           r[1];
}

int main () {
    Register r1, r2, r3; // 声明3个寄存器
    setR(r1, 500); // 设置r1寄存器的值
    cout << readR(r1) << endl; // 读r1寄存器的值 : 500
    cout << unsigned(readH(r1)) << endl; // 读r1寄存器中第一个字节的值 : 1
    cout << unsigned(readL(r1)) << endl; // 读r1寄存器中第二个字节的值 : 244
}
```

ASM执行的指令的结构为 `op S D`，op表示具体执行的操作，S和D是操作数。

操作数：

类型	语法	表示的值	例子
立即数	$num	num	$100 => 100
寄存器	%register_name	该寄存器中的值	%r1 => %r1的值
内存	imm	M[imm]	123 => M[123]（从0开始计数，地址为123的字节）
内存	(%register_name)	M[%register_name]	(%r2) => M[%r2]，取出寄存器%r2中的值n，把该值当作地址，返回M[n]。
内存	imm(%register_name)	M[imm + %register_name]	100(%r3) => M[100 + %r3的值], 取出寄存器%r3中的值n，返回M[100+n]

其中操作数D的类型只可能是寄存器或内存。内存类型的操作数中如果包含寄存器（上述最后两条规则的情况），则寄存器只可能是%r1或%r2或%r3。

指令

类型	语法	执行效果	例子
移动	mov2 S D	复制S的2个字节到从D开始的2个字节	mov2 $1234 %r1 => %r1 = 1234
移动	mov1 S D	复制S的1个字节到从D开始的1个字节	mov1 100(%r22) %r12 => %r12 = M[100 + %r22]

类型	语法	执行效果	例子
加法	add2 S D	将从S的2个字节与从D开始的2个字节按照16bit无符号整数相加，把结果存入D中	mov2 $10 %r2 => %r2 = 10 mov2 $20 %r3 => %r3 = 20 add2 %r2 %r3 => %r3 = %r2 + %r3
加法	add1 S D	将从S的1个字节与从D开始的1个字节按照8bit无符号整数相加，把结果存入从D开始的1个字节中	add1 $100 102 => M[102] = M[102] + 100

其中，在mov2和add2中，若S或D的类型是寄存器，则只可能是%r1、%r2、%r3（即2字节的寄存器）。 在mov1和add1中，若S或D的类型是寄存器，则只可能是%r11、%r12、%r21、%r22、%r31、%r32（即1字节的寄存器）。

输入格式

第一行：3个数字x y z，表示初始时寄存器%r1，%r2，%r3的值。

第二行：512个数字a0 a1 ... a511（ai和a(i+1)之间用空格隔开），表示地址从0...511的字节数据。

第三行：1个数字n。

接下来n行：每行一条ASM指令。

输出格式

依次执行每条ASM指令。

1. 若访问内存时发现地址address非法（mov1或add1时address >= 512，以及mov2或add2时address >= 511）， 输出`segmentation fault`，并终止执行该条以及之后的ASM指令，**不用**输出寄存器和内存的状态。

2. 若n条ASM指令全部执行结束，输出执行完n条指令后的寄存器和内存状态。
   第一行输出%r1，%r2，%r3的值，中间用空格隔开。
   第二行依次输出地址从0...511的内存数据，中间用空格隔开。

提示

本题不能使用的容器：vector、queue、stack。

由于cin和cout无法直接处理类型为uint8_t的值，我们需要做一步隐式转换。供参考的示例如下所示。

读内存数据并初始化内存：

```
uint8_t M[512]; // 声明内存M
for (size_t i = 0; i < 512; i++) { // 读取输入的内存数据并初始化内存M
    uint16_t tmp;
    std::cin >> tmp;
    M[i] = tmp;
}
```

输出类型为uint8_t的变量的值：

```
uint8_t a = 1;
std::cout << unsigned(a);
```

关于如何解析指令（以mov2 S D为例）：

1. 关于指令的操作的解析，可以看到操作由行为和字长两部分组成，比如mov2是移动2字节。其中，字长决定了我们如何去解析S和D的值（是按1字节解释还是2字节解释），而行为决定了 我们怎么处理S和D的值（加法还是移动）。
2. 关于S的解析。我们可以把S的类型分为两类，一类是立即数，一类是非立即数（寄存器和内存）。 如果是立即数，它的字符串形式为"$num"，忽略$，使用函数stoul("num")将字符串"num"转为无符号整数num。如果是非立即数，首先根据S[0] == "%"判断其是否是寄存器类型。如果是，则按照解析寄存器的方式解析S；如果不是，则按照解析内存的方式解析S。注意，在解析寄存器或内存时， 就需要依赖1中提到的字长。
3. 由于D一定是非立即数类型，所以其解析方式和S的非立即数类型解析相似。

数据范围

0 ≤ x,y,z ≤ 2^16 - 1 (即uint16_t的范围)

对于任意的ai, 0 ≤ ai ≤ 2^8 - 1 (即uint8_t的范围)

mov1/add1 S D中，若S为立即数$num，则0 ≤ num ≤ 2^8 - 1

mov2/add2 S D中，若S为立即数$num，则0 ≤ num ≤ 2^16 - 1

样例

输入

```
256 255 255
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4
mov2 %r1 0
mov2 %r2 2
mov1 %r32 4
mov1 %r31 5
```

输出

```
256 255 255
1 0 0 255 255 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

提交格式

你提交的文件结构应该类似如下形式：

```
<your student number>.zip
|- 1_area
|  |- main.cpp
|
|- 2_regex
|  |- main.cpp
|
|- 3_register
|  |- main.cpp
|
|- cs1604.txt (include it if you use StanfordCppLib)
```

补充概念：

uint8_t和uint16_t

为保证c/c++程序的可移植性，我们可以在声明整数类型时显式地附带上它所占比特数。
比如，int默认是32bit（4字节），但其实其大小在不同的操作系统可能是不同的。
但是，int32_t保证这一定是32bit的整数类型。
而uint8_t和uint16_t是两种无符号（unsigned）整数类型，
无符号整数的名称由来和它对于存储数据的解释有关。
假设一个字节的二进制表示为0b10110011，如果它的类型是uint8_t,
则它的十进制解释为（从低位到高位加权求和）1 * 1 + 1 * 2^1 + 0 * 2^2 + 0 * 2^3 + 1 * 2^4 + 1 * 2^5 + 0 * 2^6 + 1 * 2^7 = 179。
使用这种加权平均的解释意味着其代表的数字一定≥0，所以是“无符号”的。
假设有一个长度是16的由0和1组成的串，我们既可以把它看作两个连续的uint8_t类型的数，也可以用一个uint16_t来解释它。

大端法（Big-Endian）存储和小端法（Little-Endian）存储

在内存中，最基本的数据单位是字节（Byte），一个字节的大小是8bit。
如上文中所说的，碰到内存中两个连续的字节，我们既可以把它解释为两个uint8_t类型的数，也可以把它解释为一个uint16_t的数
（在c和c++中这种解释可以通过强制转换实现）。解释uint8_t没有问题，按照1中所示的加权求和即可。
但是在解释uint16_t时，同样使用加权求和，但是由于存在两个字节，哪个字节中的bit应该被当作高位bit，哪个字节中的bit应该被当作低位bit呢？
就这个问题，产生了两种不同的解释方式—大端法存储和小端法存储。设内存地址100处的字节为0b11111111，内存地址101处的字节为0b00000000。
在大端法中，按照uint16_t对100-101这16个bit的解释为0 * 1 + 0 * 2^1 + 0 * 2^2 + .. + 0 ^ 2^7 + 1 * 2^8 + 1 * 2^9 + ... + 1 * 2^15。

但是在小端法的解释中，会把要解释的byte倒过来（变成字节101在前，字节100在后）再加权求和，所以是1 * 1 + 1 * 2^1 + 1 *2^2 + ... + 1 * 2^7 + 0 * 2^8 + 0 * 2^9 + ... + 0 ^ 2^15。
目前主流的操作系统中，windows采用小端法，macOS采用大端法，linux大端小端都有。