

# 数据结构：2/8 进制转换器 实验报告

毛子恒 李臻 张梓靖

2020 年 10 月 17 日

## 小组成员

班级：2019211309	姓名：毛子恒	学号：2019211397	分工：代码 文档
班级：2019211310	姓名：李臻	学号：2019211458	分工：测试 文档
班级：2019211308	姓名：张梓靖	学号：2019211379	分工：文档

## 目录

1	需求分析	2
2	概要设计	3
3	详细设计	5
4	调试分析报告	7
5	用户使用说明	7
6	测试结果	8

# 1 需求分析

## 1.1 题目描述

输入一串二进制数，输出其八进制表示。

## 1.2 输入描述

程序从标准输入中读入数据。输入一个 0/1 串表示二进制数，以“#”表示输入结束。

输入满足二进制串的长度  $LEN \leq 10^6$

## 1.3 输出描述

程序向标准输出中输出结果。

输出分为三种情况：

1. 输入合法，程序正常运行结束。此时输出一行一个八进制数。
2. 输入不合法，即输入中包含除了 0/1、空白字符、# 以外的其他字符，此时输出一行一个字符串“Please check your input.”（不带引号）。
3. 程序发生运行时错误，比如内存分配失败。此时程序没有输出。

## 1.4 样例输入输出

### 1.4.1 样例输入输出 1

【输入】

110#

【输出】

6

### 1.4.2 样例输入输出 2

【输入】

010111111#

【输出】

277

### 1.4.3 样例输入输出 3

【输入】

0110001010#

【输出】

612

#### 1.4.4 样例输入输出 4

【输入】

```
110100011001000000010100010110010010000101110011011111111010000001011000101111100010100011100#
```

【输出】

```
15062005054441346777201305742434
```

#### 1.4.5 样例输入输出 5

【输入】

```
112000#
```

【输出】

```
Please check your input.
```

### 1.5 程序功能

程序将输入的二进制串转化为八进制并且输出。

## 2 概要设计

### 2.1 问题解决的思路

使用栈模拟操作过程。首先将所有二进制位依次入栈  $s_1$ 。之后每次从  $s_1$  栈顶取出三个二进制位，将其转化为一个八进制位后入栈  $s_2$ ，最后把  $s_2$  中元素依次出栈并输出。

此题中栈实现了初始化、判空、入栈、出栈、获取栈顶元素、释放空间这六种操作。

### 2.2 栈的定义

```
1 //数据对象
2 typedef char ElemType;
3
4 typedef struct stack
5 {
6     ElemType * top;
7     ElemType * base;
8     int stacksize;
9 } Stack;
10
11 /*
12  * 操作：初始化栈，分配空间
13  * 后件：s指向一个空栈
14  */
15 void initStack(Stack * s);
16
```

```
17  /*
18  * 操作：判断栈是否为空
19  * 前件：s是一个栈
20  * 后件：如果该栈为空，返回true；否则返回false
21  */
22  bool isEmpty(Stack s);
23
24  /*
25  * 操作：将数据元素入栈
26  * 前件：s指向一个栈
27  * 后件：如果入栈成功，item成为栈顶元素；如果入栈之前该栈已满，则重新分配空间
28  */
29  void pushStack(Stack * s, ElemType item);
30
31  /*
32  * 操作：获取栈顶元素
33  * 前件：s是一个栈
34  * 后件：如果该栈不为空，返回栈顶元素
35  */
36  ElemType getStackTop(Stack s);
37
38  /*
39  * 操作：栈顶元素出栈
40  * 前件：s指向一个栈
41  * 后件：如果该栈不为空，栈顶元素出栈，返回这个出栈的元素
42  */
43  ElemType popStack(Stack * s);
44
45  /*
46  * 操作：释放栈空间
47  * 前件：s指向一个栈
48  * 后件：释放该栈的空间
49  */
50  void destroyStack(Stack * s);
```

## 2.3 主程序的流程

1. 初始化栈
2. 输入，元素入 s1 栈
3. 每次从 s1 中出栈三个元素，计算对应的八进制位并入 s2 栈，重复执行直到 s1 为空
4. 元素出 s2 栈并输出
5. 释放空间

## 2.4 各程序模块之间的层次关系

模块调用关系图如图 1。

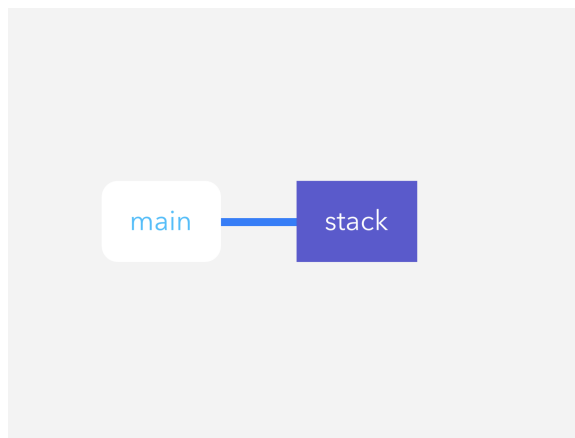


图 1: 模块的调用关系

### 3 详细设计

#### 3.1 栈的实现

栈设计中基本操作的伪代码算法如下:

```
1 void Stack(Stack * s) // 初始化栈
2 {
3     给*s分配内存
4     if (*s内存分配失败) // 空间分配失败
5         异常退出
6     s->top <- s->base
7     s->stacksize <- STACKINCREASESIZE // 初始栈空间为STACKINCREASESIZE
8 }
9
10 bool isStackEmpty(Stack s) // 判断栈是否为空
11 {
12     if (s栈为空) 返回1
13     else 返回0
14 }
15
16 void pushStack(Stack * s, ElemType item) // 将数据元素入栈
17 {
18     if (s栈满)
19     {
20         分配给s更多的空间
21         if (s空间分配失败)
22             异常退出
23         s->top <- s->base + s->stacksize
24         s->stacksize <- s->stacksize + STACKINCREASESIZE
25     }
26     item入栈
27 }
28
29 ElemType getStackTop(Stack s) // 获取栈顶元素
```

```
30 {
31     if (栈为空)
32         返回异常
33     返回 *(s.top - 1) // 返回栈顶元素
34 }
35
36 ElemType popStack(Stack * s) // 栈顶元素出栈
37 {
38     if (栈为空)
39         返回异常
40     返回 *(--s->top) // 返回栈顶元素，并且top减1
41 }
42
43 void destroyStack(Stack * s) // 释放栈空间
44 {
45     释放s->base
46     s->base <- NULL
47     s->top <- NULL
48     s->stacksize <- 0
49 }
```

### 3.2 函数的调用关系图

函数的调用关系图如图 2。

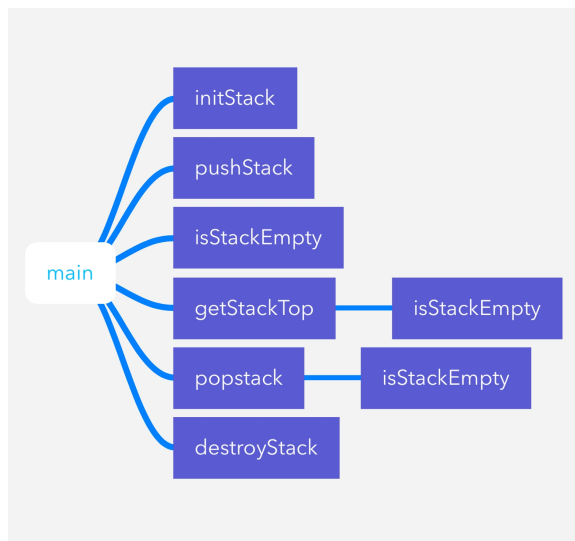


图 2: 函数的调用关系

## 4 调试分析报告

### 4.1 调试过程中遇到的问题和思考

初步实现后，测试样例时发现对于前导零的情况没有处理，遂增加查询栈顶操作，并且在主程序中增加弹出栈顶 0 元素的循环。

之后对串全为 0 的情况增加特判。

对于规模较大的数据测试时发现指针会访问到无效位置，发现是 `realloc` 操作之后没有更新 `top` 指针的位置所致。遂增加更新 `top` 指针的语句。

### 4.2 设计实现的回顾讨论

`Stack` 类型的 `top` 指针指向栈顶元素的下一个位置，实现时有几次没有注意到这个问题而导致错误。

期望输入合法，所以对于更多的不合法输入（比如输入末尾没有 `#`）没有作处理。

由于不清楚测试用数据的范围大小，`STACKINCREASESIZE` 常量的大小取到 100，也就是每次增加 100 个元素的空间。当输入很大时复杂度可能会比较大。

由于主函数对函数的调用足够严密，所以栈的实现没有考虑不符合前件的情况。

### 4.3 算法复杂度分析

`initStack`, `isStackEmpty`, `pushStack`, `getStackTop`, `popStack`, `destroyStack` 函数的时间复杂度均为  $O(1)$ 。

主程序复杂度为  $O(n)$ ，整体时间复杂度为  $O(n)$ 。

整体空间复杂度为  $O(n)$ 。

### 4.4 改进设想的经验和体会

#### 4.4.1 改进 1

程序中的各个栈操作都可以直接改成一个数组的正序/逆序的访问。但是空间浪费很严重。并且不符合题目要求。

#### 4.4.2 改进 2

此题中可以将改为根据输入规模一次性分配足够规模的栈空间，但是由于主程序中读入方式的限制而没有实现。

## 5 用户使用说明

使用 `gcc` 编译生成可执行文件。

```
gcc -o main -std=c11 main.c stack.c
```

执行可执行文件：

```
./main
```

在 Windows cmd 下：

```
main
```

之后通过标准输入输入数据，输入格式参考 1.2 节的输入描述，结果通过标准输出返回。如果输入合法并且程序正常运行结束，主函数返回值为 0。

## 6 测试结果

测试环节分为三个步骤。

### 6.1 测试第一部分

对 1.4 节给出的样例进行测试。

### 6.2 测试第二部分

测试非法输入和边界条件。

【输入】

1.01#

【输出】

Please check your input.

【输入】

0#

【输出】

0

【输入】

00000000000000000000#

【输出】

0

【输入】

0000000010#

【输出】

2

### 6.3 测试第三部分

使用 Python 实现 2/8 进制转换器 (testing/test.py) 如下：

```
1 a = int(input()[:-1], 2)
2 print(oct(a)[2:])
```

将原解法与此解法比对。

测试在 macOS Catalina 10.15.6 下进行。

在  $LEN \leq 10$ ,  $LEN \leq 1000$ ,  $LEN \leq 1000000$  的范围下分别随机生成 1000 组测试数据，分别传入 main 和 test.py，并且比对两程序的输出。



3000 组数据中两程序的输出均相同。

数据生成程序 (testing/data.cpp) 如下：

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int LEN = 1e4;
6
7 int main()
8 {
9     srand(time(0));
10    int n = rand() % LEN + 1;
11    for (int i = 1; i <= n; ++i)
12        printf("%d", rand() % 2);
13    puts("#");
14    return 0;
15 }
```

比对脚本 (testing/chk.sh) 如下：

```
1 for i in {1..100}
2 do
3     sleep 1
4     ./data >in.in
5     ./main <in.in >out.out
6     python ./test.py <in.in >out1.out
7     if ! diff out.out out1.out
8     then
9         break
10    fi
11    echo "Correct"
12 done
```