```
struct SStack
{
    void * data[1024]
    int m_size;  栈的大小
}
```

栈顶在数组的首地址还是尾地址？

栈顶设计在尾地址！原因数组尾部进行频繁插入删除效率会比头部高

```
8  struct SStack
9  {
10     void * data[MAX]; //数组
11
12     //栈的元素个数
13     int m_Size;
14 };
15
16 typedef void * seqStack;
17
18 //初始化栈
19
20 //入栈
21
22 //出栈
23
24 //获取栈顶元素
25
26 //栈的大小
27
28 //判断栈是否为空
29
30 //销毁栈
```

void*是无类型的指针(这样使用是为了给用户提供可以储存任意数据类型的栈)

> 当任意类型的指针赋值给void*
>
> void* 会自动转换成对应的数据类型
>
> 任何类型的指针都可以直接赋值给它，无需进行强制类型转换
>
> 但这并不意味着，void * 也可以无需强制类型转换地赋给其它类型的指针。因为"无类型"可以包容"有类型"，而"有类型"则不能包容"无类型"。

需要注意的是： void **指针进行算法操作**，即下列操作都是不合法的：

```
void * pvoid;
pvoid++; //ANSI：错误
pvoid += 1; //ANSI：错误
//ANSI标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。
//例如：
int *pint;
pint++; //ANSI：正确
```

```
void * pvoid;
((char *)pvoid)++; //ANSI：错误；GNU：正确
(char *)pvoid += 1; //ANSI：错误；GNU：正确
```

初始化栈

```
//初始化栈
seqStack init_SeqStack()
{
    struct SStack * stack = malloc(sizeof(struct SStack));

    if (stack == NULL)
    {
        return NULL;
    }

    //清空数组中的每个元素
    memset(stack->data, 0, sizeof(void*)*MAX);

    stack->m_Size = 0;

    return stack;
}
```

入栈

```
//入栈
void push_SeqStack( seqStack stack, void * data)
{
    if (stack == NULL)
    {
        return;
    }
    if (data == NULL)
    {
        return;
    }

    //判断是否已经栈满 ,如果满了 不可以再入栈了
    struct SStack * myStack = stack;
    if (myStack->m_Size == MAX)
    {
        return;
    }

    myStack->data[myStack->m_Size] = data; //入栈  尾插

    myStack->m_Size++; //更新栈大小
}
```

出栈

```
//出栈
void pop_SeqStack(seqStack stack)
{
    if (stack == NULL)
    {
        return;
    }

    //如果是空栈 不执行出栈
    struct SStack * myStack = stack;
    if (myStack->m_Size <= 0)
    {
        return;
    }
```

```
    //执行出栈
    myStack->data[myStack->m_Size - 1] = NULL;
    //更新栈的大小
    myStack->m_Size--;

}
```

获取栈顶元素

```
//获取栈顶元素
void * top_SeqStack(seqStack stack)
{

    if (stack == NULL)
    {
        return NULL;
    }

    struct SStack * myStack = stack;

    //如果是空栈   返回 NULL
    if (myStack->m_Size == 0)
    {
        return NULL;
    }

    return myStack->data[myStack->m_Size - 1];


}
```

栈的大小

```
//栈的大小
int size_SeqStack(seqStack stack)
{
    if (stack == NULL)
    {
        return -1;
    }

    struct SStack * myStack = stack;

    return myStack->m_Size;
}
```

判断栈是否为空

```
int isEmpty_SeqStack(seqStack stack)
{
    if (stack == NULL)
    {
        return -1; //真
    }

    struct SStack * myStack = stack;
    if (myStack->m_Size <= 0)
    {
        return 1; //真
    }

    return  0; //返回假 代表不是空栈
}
```

销毁栈

```
//销毁栈
void destroy_SeqStack(seqStack stack)
{

    if (stack == NULL)
    {
        return;
    }

    free(stack);
    stack = NULL;
}
```

好的，通过上述的步骤，你可以将栈这一数据结构封装分文件编写，然后呢，我们以后再去使用栈就可以直接引用头文件，非常方便。

封装好的栈

SeqStack.h

```
#pragma once
#ifndef SEQSTACK_H
#define SEQSTACK_H

#include <stdio.h>
#include <stdlib.h>

//数组去模拟栈的顺序存储
#define MAX_SIZE 1024
#define SEQSTACK_TRUE 1
#define SEQSTACK_FALSE 0
```

```
typedef struct SEQSTACK {
    void* data[MAX_SIZE];
    int size;
}SeqStack;

//初始化栈
SeqStack* Init_SeqStack();
//入栈
void Push_SeqStack(SeqStack* stack, void* data);
//返回栈顶元素
void* Top_SeqStack(SeqStack* stack);
//出栈
void Pop_SeqStack(SeqStack* stack);
//判断是否为空
int IsEmpty(SeqStack* stack);
//返回栈中元素的个数
int Size_SeqStack(SeqStack* stack);
//清空栈
void Clear_SeqStack(SeqStack* stack);
//销毁
void FreeSpace_SeqStack(SeqStack* stack);


#endif // !SEQSTACK_H
```

SeqStack.c

```
#include "SeqStack.h"


//初始化栈
SeqStack* Init_SeqStack()
{
    SeqStack* stack = (SeqStack*)malloc(sizeof(SeqStack));
    for (int i = 0; i < MAX_SIZE; i++) {
        stack->data[i] = NULL;
    }
    stack->size = 0;
    return stack;
}
//入栈
void Push_SeqStack(SeqStack* stack, void* data)
{
    if (stack == NULL) {
        return;
    }
    if (stack->size == MAX_SIZE) {
        return;
    }
    if (data == NULL) {
        return;
    }

    stack->data[stack->size] = data;
    stack->size++;
}
//返回栈顶元素
```

```
void* Top_SeqStack(SeqStack* stack)
{
    if (stack == NULL) {
        return NULL;
    }
    if (stack->size == 0) {
        return NULL;
    }
    return stack->data[stack->size-1];
}
//出栈
void Pop_SeqStack(SeqStack* stack)
{
    if (stack == NULL) {
        return;
    }
    if (stack->size == 0) {
        return;
    }
    stack->data[stack->size - 1] = NULL;
    stack->size--;
}
//判断是否为空
int IsEmpty(SeqStack* stack)
{
    if (stack == NULL) {
        return -1;
    }
    if (stack->size == 0) {
        return
            SEQSTACK_TRUE;
    }
    return SEQSTACK_FALSE;
}
//返回栈中元素的个数
int Size_SeqStack(SeqStack* stack)
{
    return stack->size;
}
//清空栈
void Clear_SeqStack(SeqStack* stack)
{
    if (stack == NULL) {
        return;
    }
    for (int i = 0; i < stack->size; i++) {
        stack->data[i] = NULL;
    }
    stack->size = 0;
}
//销毁
void FreeSpace_SeqStack(SeqStack* stack)
{
    if (stack == NULL) {
        return;
    }
    free(stack);
}
```