链接地址：<http://blog.csdn.net/jjzhoujun2010/article/details/6856164>

堆栈（stack）的显著特点是后进先出（Last-In First-Out, LIFO），其实现的方法有三种可选方案：静态数组、动态分配的数组、动态分配的链式结构。

　　静态数组：特点是要求结构的长度固定，而且长度在编译时候就得确定。其优点是结构简单，实现起来方便而不容易出错。而缺点就是不够灵活以及固定长度不容易控制，适用于知道明确长度的场合。

　　动态数组：特点是长度可以在运行时候才确定以及可以更改原来数组的长度。优点是灵活，缺点是由此会增加程序的复杂性。

　　链式结构：特点是无长度上线，需要的时候再申请分配内存空间，可最大程度上实现灵活性。缺点是链式结构的链接字段需要消耗一定的内存，在链式结构中访问一个特定元素的效率不如数组。

　　首先先确定一个堆栈接口的头文件，里面包含了各个方案下的函数原型，放在一起是为了实现程序的模块化以及便于修改。然后再接着分别介绍各个方案的具体实施方法。

　　堆栈接口stack.h文件代码：

[cpp] [view plain](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [copy](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [print?](http://blog.csdn.net/jjzhoujun2010/article/details/6856164)

/\*

\*\* 堆栈模块的接口 stack.h

\*/

#include<stdlib.h>

#define STACK\_TYPE int /\* 堆栈所存储的值的数据类型 \*/

/\*

\*\* 函数原型：create\_stack

\*\* 创建堆栈，参数指定堆栈可以保存多少个元素。

\*\* 注意：此函数只适用于动态分配数组形式的堆栈。

\*/

void create\_stack(size\_t size);

/\*

\*\* 函数原型：destroy\_stack

\*\* 销毁一个堆栈，释放堆栈所适用的内存。

\*\* 注意：此函数只适用于动态分配数组和链式结构的堆栈。

\*/

void destroy\_stack(void);

/\*

\*\* 函数原型：push

\*\* 将一个新值压入堆栈中，参数是被压入的值。

\*/

void push(STACK\_TYPE value);

/\*

\*\* 函数原型：pop

\*\* 弹出堆栈中栈顶的一个值，并丢弃。

\*/

void pop(void);

/\*

\*\* 函数原型：top

\*\* 返回堆栈顶部元素的值，但不改变堆栈结构。

\*/

STACK\_TYPE top(void);

/\*

\*\* 函数原型：is\_empty

\*\* 如果堆栈为空，返回TRUE,否则返回FALSE。

\*/

int is\_empty(void);

/\*

\*\* 函数原型：is\_full

\*\* 如果堆栈为满，返回TRUE,否则返回FALSE。

\*/

int is\_full(void);

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

/\*

\*\* 堆栈模块的接口 stack.h

\*/

#include<stdlib.h>

#define STACK\_TYPE int /\* 堆栈所存储的值的数据类型 \*/

/\*

\*\* 函数原型：create\_stack

\*\* 创建堆栈，参数指定堆栈可以保存多少个元素。

\*\* 注意：此函数只适用于动态分配数组形式的堆栈。

\*/

void create\_stack(size\_t size);

/\*

\*\* 函数原型：destroy\_stack

\*\* 销毁一个堆栈，释放堆栈所适用的内存。

\*\* 注意：此函数只适用于动态分配数组和链式结构的堆栈。

\*/

void destroy\_stack(void);

/\*

\*\* 函数原型：push

\*\* 将一个新值压入堆栈中，参数是被压入的值。

\*/

void push(STACK\_TYPE value);

/\*

\*\* 函数原型：pop

\*\* 弹出堆栈中栈顶的一个值，并丢弃。

\*/

void pop(void);

/\*

\*\* 函数原型：top

\*\* 返回堆栈顶部元素的值，但不改变堆栈结构。

\*/

STACK\_TYPE top(void);

/\*

\*\* 函数原型：is\_empty

\*\* 如果堆栈为空，返回TRUE,否则返回FALSE。

\*/

int is\_empty(void);

/\*

\*\* 函数原型：is\_full

\*\* 如果堆栈为满，返回TRUE,否则返回FALSE。

\*/

int is\_full(void);

　　一、静态数组堆栈

　　在静态数组堆栈中，STACK\_SIZE表示堆栈所能存储的元素的最大值，用top\_element作为数组下标来表示堆栈里面的元素，当top\_element == -1的时候表示堆栈为空；当top\_element == STACK\_SIZE - 1的时候表示堆栈为满。push的时候top\_element加1，top\_element == 0时表示第一个堆栈元素；pop的时候top\_element减1。

　　a\_stack.c 源代码如下：

[cpp] [view plain](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [copy](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [print?](http://blog.csdn.net/jjzhoujun2010/article/details/6856164)

/\*

\*\*

\*\* 静态数组实现堆栈程序 a\_stack.c ，数组长度由#define确定

\*/

#include"stack.h"

#include<assert.h>

#include<stdio.h>

#define STACK\_SIZE 100 /\* 堆栈最大容纳元素数量 \*/

/\*

\*\* 存储堆栈中的数组和一个指向堆栈顶部元素的指针

\*/

static STACK\_TYPE stack[STACK\_SIZE];

static int top\_element = -1;

/\* push \*/

void push(STACK\_TYPE value)

{

    assert(!is\_full()); /\* 压入堆栈之前先判断是否堆栈已满\*/

    top\_element += 1;

    stack[top\_element] = value;

}

/\* pop \*/

void pop(void)

{

    assert(!is\_empty()); /\* 弹出堆栈之前先判断是否堆栈已空 \*/

    top\_element -= 1;

}

/\* top \*/

STACK\_TYPE top(void)

{

    assert(!is\_empty());

    return stack[top\_element];

}

/\* is\_empty \*/

int is\_empty(void)

{

    return top\_element == -1;

}

/\* is\_full \*/

int is\_full(void)

{

    return top\_element == STACK\_SIZE - 1;

}

/\*

\*\* 定义一个print函数，用来打印堆栈里面的元素。

\*/

void print(void)

{

    int i;

    i = top\_element;

    printf("打印出静态数组堆栈里面的值: ");

    if(i == -1)

        printf("这是个空堆栈\n");

    while(i!= -1)

        printf("%d ",stack[i--]);

    printf("\n");

}

int main(void)

{

    print();

    push(10); push(9); push(7); push(6); push(5);

    push(4);  push(3); push(2); push(1); push(0);

    printf("push压入数值后：\n");

    print();

    printf("\n");

    pop();

    pop();

    printf("经过pop弹出几个元素后的堆栈元素:\n");

    print();

    printf("\n");

    printf("top()调用出来的值: %d\n",top());

    return 1;

}

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

/\*

\*\*

\*\* 静态数组实现堆栈程序 a\_stack.c ，数组长度由#define确定

\*/

#include"stack.h"

#include<assert.h>

#include<stdio.h>

#define STACK\_SIZE 100 /\* 堆栈最大容纳元素数量 \*/

/\*

\*\* 存储堆栈中的数组和一个指向堆栈顶部元素的指针

\*/

static STACK\_TYPE stack[STACK\_SIZE];

static int top\_element = -1;

/\* push \*/

void push(STACK\_TYPE value)

{

assert(!is\_full()); /\* 压入堆栈之前先判断是否堆栈已满\*/

top\_element += 1;

stack[top\_element] = value;

}

/\* pop \*/

void pop(void)

{

assert(!is\_empty()); /\* 弹出堆栈之前先判断是否堆栈已空 \*/

top\_element -= 1;

}

/\* top \*/

STACK\_TYPE top(void)

{

assert(!is\_empty());

return stack[top\_element];

}

/\* is\_empty \*/

int is\_empty(void)

{

return top\_element == -1;

}

/\* is\_full \*/

int is\_full(void)

{

return top\_element == STACK\_SIZE - 1;

}

/\*

\*\* 定义一个print函数，用来打印堆栈里面的元素。

\*/

void print(void)

{

int i;

i = top\_element;

printf("打印出静态数组堆栈里面的值: ");

if(i == -1)

printf("这是个空堆栈\n");

while(i!= -1)

printf("%d ",stack[i--]);

printf("\n");

}

int main(void)

{

print();

push(10); push(9); push(7); push(6); push(5);

push(4); push(3); push(2); push(1); push(0);

printf("push压入数值后：\n");

print();

printf("\n");

pop();

pop();

printf("经过pop弹出几个元素后的堆栈元素:\n");

print();

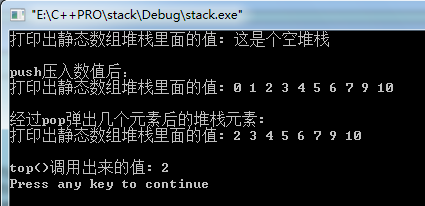
printf("\n");

printf("top()调用出来的值: %d\n",top());

return 1;

}

　　结果如下图：



　　二、动态数组堆栈

　　头文件还是用stack.h，改动的并不是很多，增加了stack\_size变量取代STACK\_SIZE来保存堆栈的长度，数组由一个指针来代替，在全局变量下缺省为0。

　　create\_stack函数首先检查堆栈是否已经创建，然后才分配所需数量的内存并检查分配是否成功。destroy\_stack函数首先检查堆栈是否存在，已经释放内存之后把长度和指针变量重新设置为零。is\_empty 和 is\_full 函数中添加了一条断言，防止任何堆栈函数在堆栈被创建之前就被调用。

　　d\_stack.c源代码如下：

[cpp] [view plain](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [copy](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [print?](http://blog.csdn.net/jjzhoujun2010/article/details/6856164)

/\*

\*\* 动态分配数组实现的堆栈程序 d\_stack.c

\*\* 堆栈的长度在创建堆栈的函数被调用时候给出，该函数必须在任何其他操作堆栈的函数被调用之前条用。

\*/

#include"stack.h"

#include<stdio.h>

#include<malloc.h>

#include<assert.h>

/\*

\*\* 用于存储堆栈元素的数组和指向堆栈顶部元素的指针

\*/

static STACK\_TYPE \*stack;

static int        stack\_size;

static int        top\_element = -1;

/\* create\_stack \*/

void create\_stack(size\_t size)

{

    assert(stack\_size == 0);

    stack\_size = size;

    stack = (STACK\_TYPE \*)malloc(stack\_size \* sizeof(STACK\_TYPE));

    if(stack == NULL)

        perror("malloc分配失败");

}

/\* destroy \*/

void destroy\_stack(void)

{

    assert(stack\_size > 0);

    stack\_size = 0;

    free(stack);

    stack = NULL;

}

/\* push \*/

void push(STACK\_TYPE value)

{

    assert(!is\_full());

    top\_element += 1;

    stack[top\_element] = value;

}

/\* pop \*/

void pop(void)

{

    assert(!is\_empty());

    top\_element -= 1;

}

/\* top \*/

STACK\_TYPE top(void)

{

    assert(!is\_empty());

    return stack[top\_element];

}

/\* is\_empty \*/

int is\_empty(void)

{

    assert(stack\_size > 0);

    return top\_element == -1;

}

/\* is\_full \*/

int is\_full(void)

{

    assert(stack\_size > 0);

    return top\_element == stack\_size - 1;

}

/\*

\*\* 定义一个print函数，用来打印堆栈里面的元素。

\*/

void print(void)

{

    int i;

    i = top\_element;

    printf("打印出动态数组堆栈里面的值: ");

    if(i == -1)

        printf("这是个空堆栈\n");

    while(i!= -1)

        printf("%d ",stack[i--]);

    printf("\n");

}

int main(void)

{

    create\_stack(50);

    print();

    push(10); push(9); push(7); push(6); push(5);

    push(4);  push(3); push(2); push(1); push(0);

    printf("push压入数值后：\n");

    print();

    printf("\n");

    pop();

    pop();

    printf("经过pop弹出几个元素后的堆栈元素:\n");

    print();

    printf("\n");

    printf("top()调用出来的值: %d\n",top());

    destroy\_stack();

    return 1;

}

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

/\*

\*\* 动态分配数组实现的堆栈程序 d\_stack.c

\*\* 堆栈的长度在创建堆栈的函数被调用时候给出，该函数必须在任何其他操作堆栈的函数被调用之前条用。

\*/

#include"stack.h"

#include<stdio.h>

#include<malloc.h>

#include<assert.h>

/\*

\*\* 用于存储堆栈元素的数组和指向堆栈顶部元素的指针

\*/

static STACK\_TYPE \*stack;

static int stack\_size;

static int top\_element = -1;

/\* create\_stack \*/

void create\_stack(size\_t size)

{

assert(stack\_size == 0);

stack\_size = size;

stack = (STACK\_TYPE \*)malloc(stack\_size \* sizeof(STACK\_TYPE));

if(stack == NULL)

perror("malloc分配失败");

}

/\* destroy \*/

void destroy\_stack(void)

{

assert(stack\_size > 0);

stack\_size = 0;

free(stack);

stack = NULL;

}

/\* push \*/

void push(STACK\_TYPE value)

{

assert(!is\_full());

top\_element += 1;

stack[top\_element] = value;

}

/\* pop \*/

void pop(void)

{

assert(!is\_empty());

top\_element -= 1;

}

/\* top \*/

STACK\_TYPE top(void)

{

assert(!is\_empty());

return stack[top\_element];

}

/\* is\_empty \*/

int is\_empty(void)

{

assert(stack\_size > 0);

return top\_element == -1;

}

/\* is\_full \*/

int is\_full(void)

{

assert(stack\_size > 0);

return top\_element == stack\_size - 1;

}

/\*

\*\* 定义一个print函数，用来打印堆栈里面的元素。

\*/

void print(void)

{

int i;

i = top\_element;

printf("打印出动态数组堆栈里面的值: ");

if(i == -1)

printf("这是个空堆栈\n");

while(i!= -1)

printf("%d ",stack[i--]);

printf("\n");

}

int main(void)

{

create\_stack(50);

print();

push(10); push(9); push(7); push(6); push(5);

push(4); push(3); push(2); push(1); push(0);

printf("push压入数值后：\n");

print();

printf("\n");

pop();

pop();

printf("经过pop弹出几个元素后的堆栈元素:\n");

print();

printf("\n");

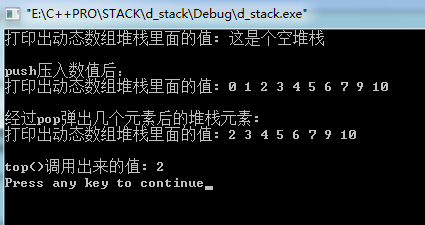
printf("top()调用出来的值: %d\n",top());

destroy\_stack();

return 1;

}

　　结果如下图：



　　三、链式堆栈

　　由于只有堆栈顶部元素才可以被访问，因此适用单链表可以很好实现链式堆栈，而且无长度限制。把一个元素压入堆栈是通过在链表头部添加一个元素实现。弹出一个元素是通过删除链表头部第一个元素实现。由于没有长度限制，故不需要create\_stack函数，需要destroy\_stack进行释放内存以避免内存泄漏。

　　头文件stack.h 不变，l\_stack.c 源代码如下：

[cpp] [view plain](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [copy](http://blog.csdn.net/jjzhoujun2010/article/details/6856164) [print?](http://blog.csdn.net/jjzhoujun2010/article/details/6856164)

/\*

\*\* 单链表实现堆栈，没有长度限制

\*/

#include"stack.h"

#include<stdio.h>

#include<malloc.h>

#include<assert.h>

#define FALSE 0

/\*

\*\* 定义一个结构以存储堆栈元素。

\*/

typedef struct STACK\_NODE

{

    STACK\_TYPE value;

    struct STACK\_NODE \*next;

} StackNode;

/\* 指向堆栈中第一个节点的指针 \*/

static StackNode \*stack;

/\* create\_stack \*/

void create\_stack(size\_t size)

{}

/\* destroy\_stack \*/

void destroy\_stack(void)

{

    while(!is\_empty())

        pop();  /\* 逐个弹出元素，逐个释放节点内存 \*/

}

/\* push \*/

void push(STACK\_TYPE value)

{

    StackNode \*new\_node;

    new\_node = (StackNode \*)malloc(sizeof(StackNode));

    if(new\_node == NULL)

        perror("malloc fail");

    new\_node->value = value;

    new\_node->next = stack;  /\* 新元素插入链表头部 \*/

    stack = new\_node;       /\* stack 重新指向链表头部 \*/

}

/\* pop \*/

void pop(void)

{

    StackNode \*first\_node;

    assert(!is\_empty());

    first\_node = stack;

    stack = first\_node->next;

    free(first\_node);

}

/\* top \*/

STACK\_TYPE top(void)

{

    assert(!is\_empty());

    return stack->value;

}

/\* is\_empty \*/

int is\_empty(void)

{

    return stack == NULL;

}

/\* is\_full \*/

int is\_full(void)

{

    return FALSE;

}

/\*

\*\* 定义一个print函数，用来打印堆栈里面的元素。

\*/

void print(void)

{

    StackNode \*p\_node;

    p\_node = stack;

    printf("打印出链式堆栈里面的值: ");

    if(p\_node == NULL)

        printf("堆栈为空\n");

    while(p\_node != NULL)

    {

        printf("%d ", p\_node->value);

        p\_node = p\_node->next;

    }

    printf("\n");

}

int main(void)

{

    print();

    push(10); push(9); push(7); push(6); push(5);

    push(4);  push(3); push(2); push(1); push(0);

    printf("push压入数值后：\n");

    print();

    printf("\n");

    pop();

    pop();

    printf("经过pop弹出几个元素后的堆栈元素:\n");

    print();

    printf("\n");

    printf("top()调用出来的值: %d\n",top());

    destroy\_stack();

    return 1;

}

[http://static.blog.csdn.net/images/save_snippets.png](javascript:;)

/\*

\*\* 单链表实现堆栈，没有长度限制

\*/

#include"stack.h"

#include<stdio.h>

#include<malloc.h>

#include<assert.h>

#define FALSE 0

/\*

\*\* 定义一个结构以存储堆栈元素。

\*/

typedef struct STACK\_NODE

{

STACK\_TYPE value;

struct STACK\_NODE \*next;

} StackNode;

/\* 指向堆栈中第一个节点的指针 \*/

static StackNode \*stack;

/\* create\_stack \*/

void create\_stack(size\_t size)

{}

/\* destroy\_stack \*/

void destroy\_stack(void)

{

while(!is\_empty())

pop(); /\* 逐个弹出元素，逐个释放节点内存 \*/

}

/\* push \*/

void push(STACK\_TYPE value)

{

StackNode \*new\_node;

new\_node = (StackNode \*)malloc(sizeof(StackNode));

if(new\_node == NULL)

perror("malloc fail");

new\_node->value = value;

new\_node->next = stack; /\* 新元素插入链表头部 \*/

stack = new\_node; /\* stack 重新指向链表头部 \*/

}

/\* pop \*/

void pop(void)

{

StackNode \*first\_node;

assert(!is\_empty());

first\_node = stack;

stack = first\_node->next;

free(first\_node);

}

/\* top \*/

STACK\_TYPE top(void)

{

assert(!is\_empty());

return stack->value;

}

/\* is\_empty \*/

int is\_empty(void)

{

return stack == NULL;

}

/\* is\_full \*/

int is\_full(void)

{

return FALSE;

}

/\*

\*\* 定义一个print函数，用来打印堆栈里面的元素。

\*/

void print(void)

{

StackNode \*p\_node;

p\_node = stack;

printf("打印出链式堆栈里面的值: ");

if(p\_node == NULL)

printf("堆栈为空\n");

while(p\_node != NULL)

{

printf("%d ", p\_node->value);

p\_node = p\_node->next;

}

printf("\n");

}

int main(void)

{

print();

push(10); push(9); push(7); push(6); push(5);

push(4); push(3); push(2); push(1); push(0);

printf("push压入数值后：\n");

print();

printf("\n");

pop();

pop();

printf("经过pop弹出几个元素后的堆栈元素:\n");

print();

printf("\n");

printf("top()调用出来的值: %d\n",top());

destroy\_stack();

return 1;

}

　　结果如下图：

