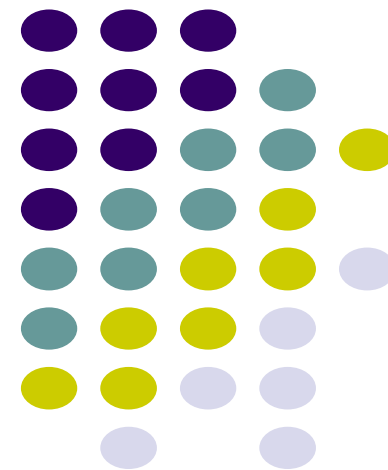


L3: 栈

吉林大学计算机学院
谷方明

fmgu2002@sina.com





学习目标

- 掌握栈的定义、特性和操作；
- 掌握栈的两种存储方式及实现；
- 应用栈解题；
- 掌握算术表达式的求解；
- 理解数据结构的封装（衔接）



栈的定义

□ 例

- ✓ 摞盘子
- ✓ 子弹夹

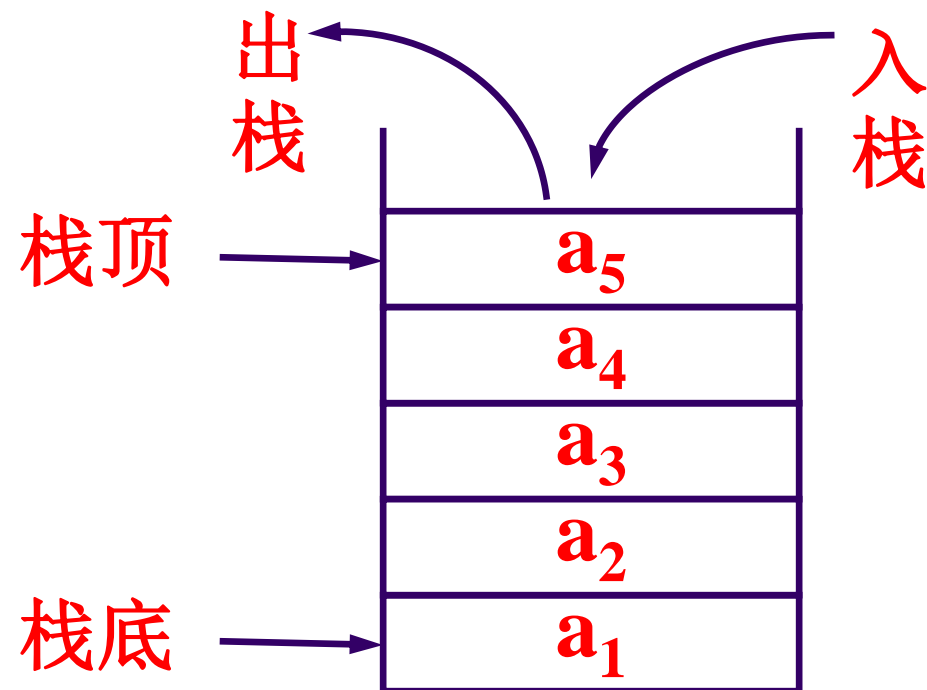


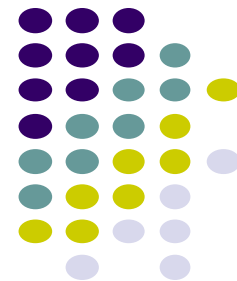
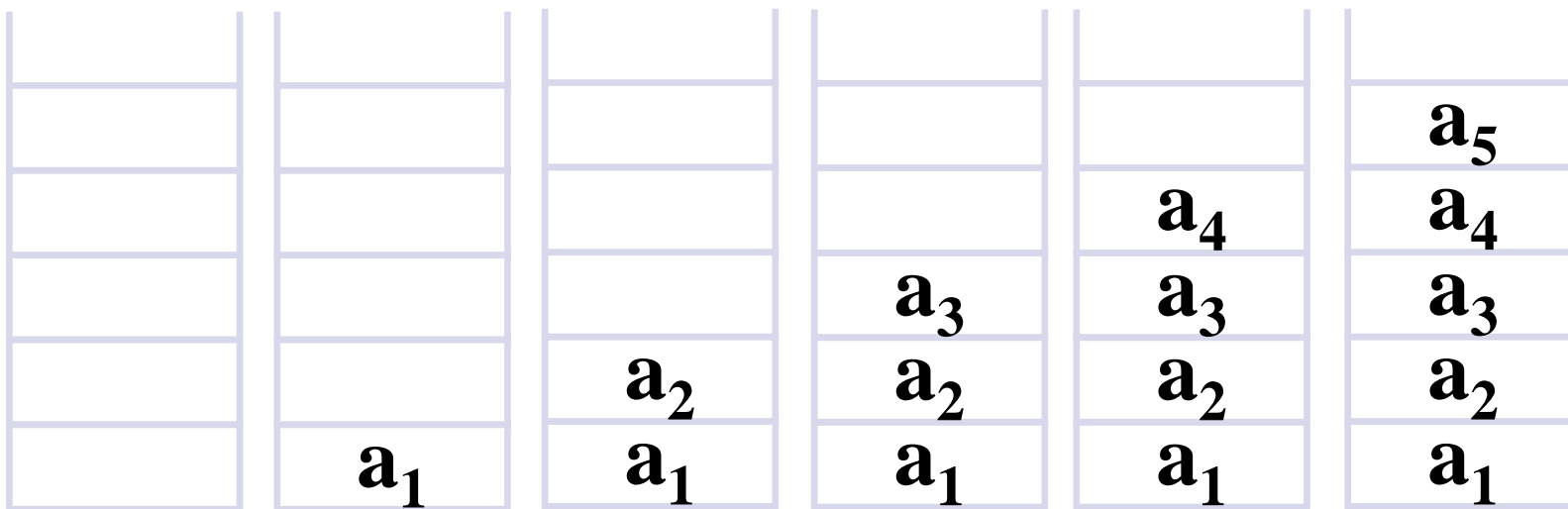
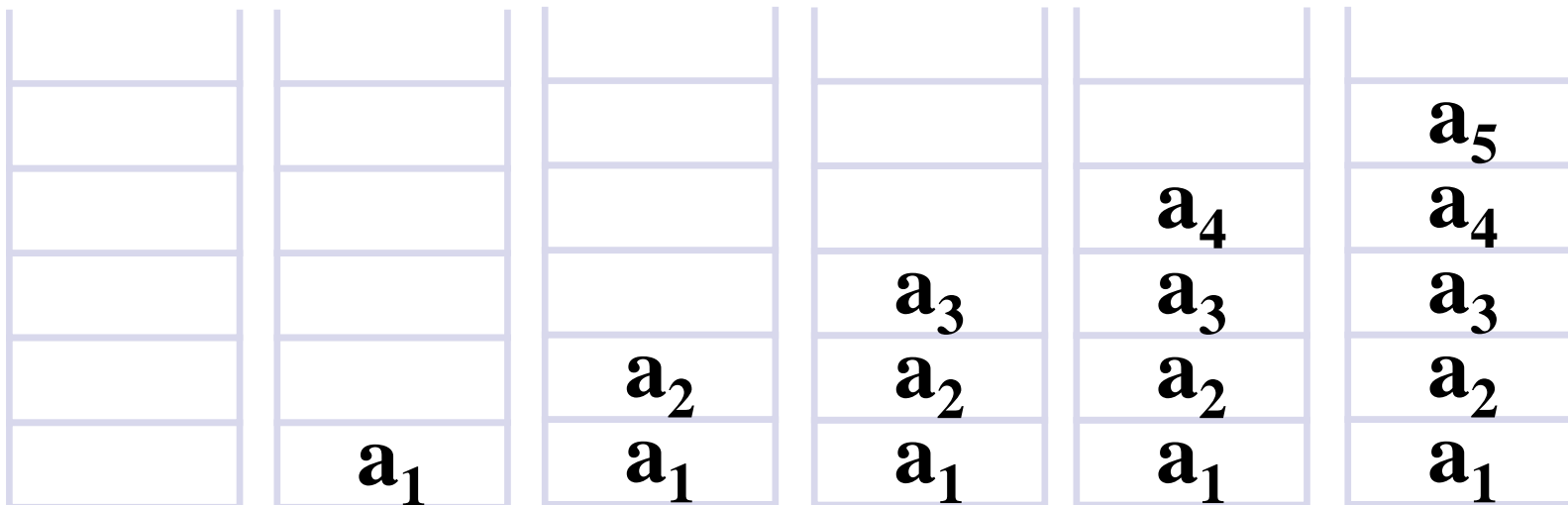
- 栈（**stack**）是一种操作受限的线性表，只允许在表的**同一端**进行插入和删除操作。

术语

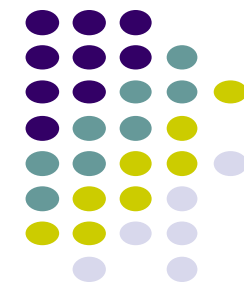
- 栈顶：进行插入、删除的一端；
- 栈底：另一端；
- 空栈：栈中无元素时。
- 插入：入栈、进栈、压栈
- 删除：出栈、退栈、弹栈

[例] 线性表
(a_1, a_2, \dots, a_5),
入栈出栈情况





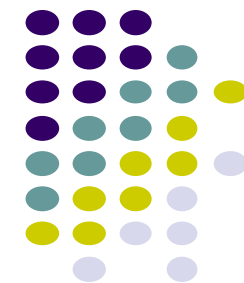
栈的特性

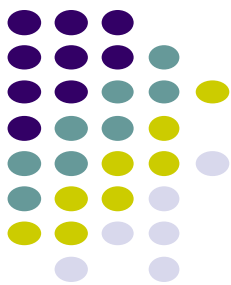


□ 后进先出(**Last In First Out, LIFO**)。栈也称作后进先出表。

栈的基本操作

1. 压栈 **push**
2. 弹栈 **pop**
3. 取栈顶元素 **peek**
4. 栈初始化
5. 判栈空
6. 判栈满
7. 清空栈





栈的顺序存储

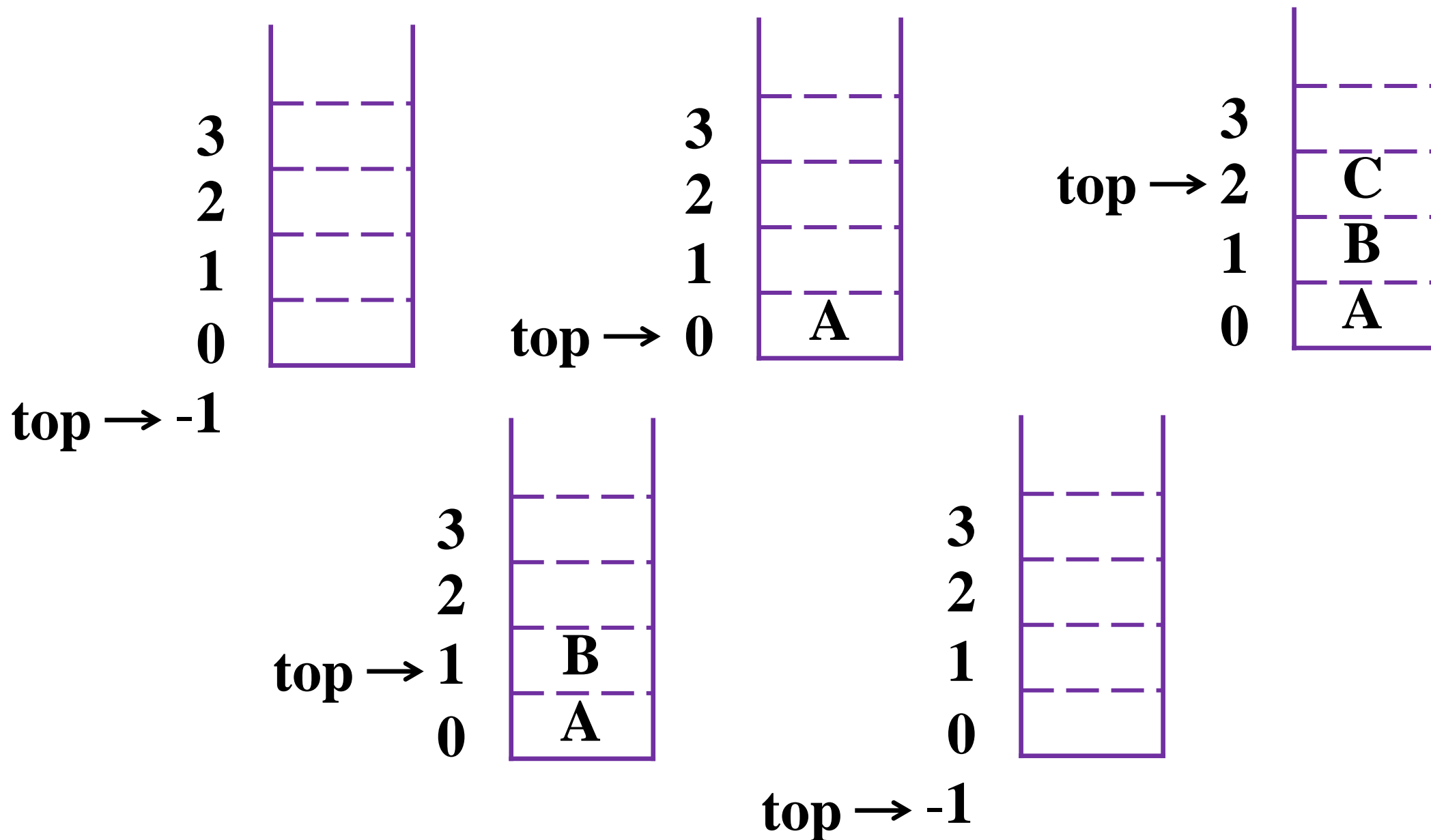
- 按顺序存储方式存放栈元素，称为顺序栈。
- 栈中的元素数（规模）必须不超过数组的规模。

- 以整数栈为例：

```
int stackArray [MaxStackSize];
```

```
int top = -1;
```

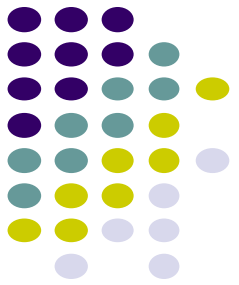

顺序栈操作情况



```
void push(int x){  
    stackArray[++top]=x;  
}
```

```
int pop(){  
    return stackArray[top--];  
}
```

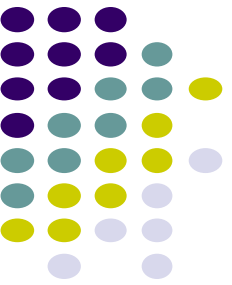
```
int peek(){  
    return stackArray[top];  
}
```



```
bool empty(){  
    return top==-1;  
}
```

```
bool full(){  
    return top==MaxStackSize-1;  
}
```

```
void clear(){  
    top=-1;  
}
```



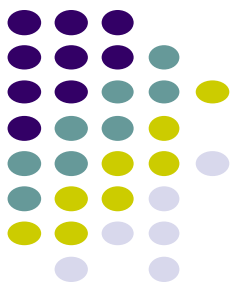
测试



```
int main(){
    int n,i;
    scanf("%d",&n);

    for(i=1;i<=n;i++) push(i);

    for(i=1;i<=n;i++) printf("%d ",pop());
    printf("\n");
}
```



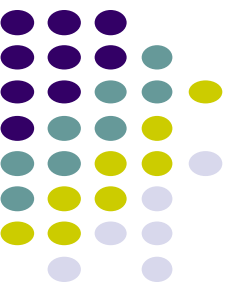
简化的风险

□ 安全的做法

- ✓ push操作前，判断栈满
- ✓ pop和peek操作前，判断栈空

□ 简化的前提是确信栈的操作都合法

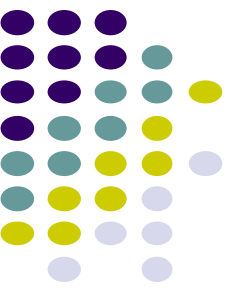
- ✓ 空间足够大，确保不会溢出
- ✓ 不会发生空栈弹栈和读取



栈的链接存储

- 用链接存储实现栈，要为每个元素分配一个额外的指针空间，指向后继结点。也称为链栈。
- 以整数栈为例：

```
struct Node{  
    int data;  
    Node* next;  
};  
Node* top=NULL;
```



压栈算法ADL描述

算法 **Push** (A, item)

// 向栈顶指针为**top**的链式栈中压入一个元素**item**

P1. [创建新结点]

$s \leftarrow \text{AVAIL.}$ //为新结点申请空间

$\text{data}(s) \leftarrow \text{item.}$ $\text{next}(s) \leftarrow \text{top.}$

P2. [更新栈顶指针]

$\text{top} \leftarrow s.$ ■



```
void push( int x )  
{  
    Node* p = new Node;  
    // Node* p = (Node *)malloc(sizeof(Node*))  
    p->data = x, p->next = top;  
    top = p;  
}
```




弹栈算法ADL描述

算法 **Pop** (A. item)

//从栈顶指针为top的链式栈中弹出栈顶元素，并存放在变量item中

P1. [栈空?]

IF top = NULL

THEN (PRINT“栈空无法弹出” . RETURN.)

P2. [出栈]

item←data(top). q←next(top).

AVAIL←top. // 释放栈顶结点的空间

top←q. █



```
int pop(){  
    int x = top->data;  
    Node* p = top; top = top->next; delete p;  
    return x;  
}
```

```
int peek(){  
    return top->data;  
}
```

```
bool empty(){  
    return top==NULL;  
}
```

```
void clear(){  
    Node * p = top;  
    while(p){  
        top = p->next;  
        delete p;  
        p = top;  
    }  
}
```



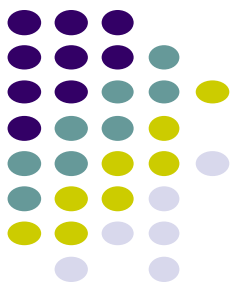


顺序栈与链式栈的比较

□ 空间复杂度

- ✓ 顺序栈初始必须申请固定的空间，当栈不满时，必然造成空间的浪费；
- ✓ 链式栈所需空间是根据需要随时申请的，其代价是为每个元素提供空间以存储next指针域。

□ 在时间复杂性上，对于针对栈顶的基本操作（压入、弹出和栈顶元素存取），顺序栈和链式栈的渐进时间复杂度均为 $O(1)$ 。



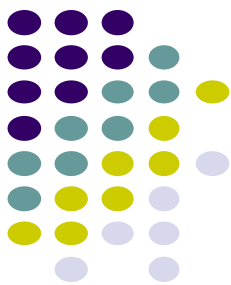
设计决策

□ 一般来讲

- ✓ 顺序栈易于实现
- ✓ 链式栈能更合理的利用空间

□ 设计决策:

- ✓ 空间是否够用
- ✓ 使用方式（）
 - 解题 => 顺序栈
 - 写成容器 => 链式栈



应用1：括号匹配

- 输入一字符串，判断括号是否匹配：“(” 与 “)”.
- 匹配输出 “**yes**”，否则输出 “**no**”.



括号匹配增强版(bracket.cpp)

- 高级语言程序设计中的各种括号应该匹配，例如：“(”与“)”匹配、“[”与“]”匹配、“{”与“}”匹配等。
- 输入一字符串，判断括号是否匹配；匹配输出“**yes**”，否则输出“**no**”。

测试样例



输入	输出
}	no
[(])	no
{ ()	no
{a=(b*c}+free()]	no
({ })	yes



栈的引入

- 遇到闭括号时，应考察其与最近未匹配的开括号是否匹配；
用来进行匹配的开括号是最后输入的，符合栈的后进先出特性，因此用栈来存放开括号，模拟匹配过程。
- ✓ 若匹配，则将匹配的开括号从栈顶删除，继续考察下一闭括号；若不匹配，说明输入的括号不配对。
- ✓ 注意：输入结束后，栈中还有多余的开括号，匹配失败。



应用2: 算术表达式求值

- 表达式求值是程序设计语言编译中的一个基本问题。其实现方法是栈的一个典型应用实例。
- 表达式是由操作数 (**operand**)、运算符 (**operator**)和界限符 (**delimiter**) 组成的。
- 其中操作数可以是常数，也可以是变量或常量的标识符；运算符是算术运算符 (**+, -, *, /**)；界限符为左右括号和标识表达式结束的结束符。



例：算术表达式

□ $1 \times (2 / 3 + 4 - 5)$

□ 运算规则：

- (1) 先计算括号内，后计算括号外；
- (2) 无括号或同层括号内，先进行乘除运算，后进行加减运算，即乘除运算的优先级高于加减运算的优先级；
- (3) 同一优先级运算，从左向右依次进行。



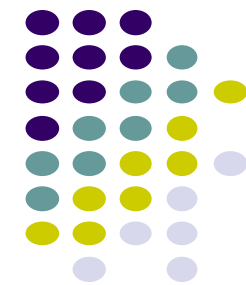
模拟法

- 用计算机模拟人工计算比较复杂。一个算术表达式中,有多少个运算符,原则上就需对表达式进行多少遍扫描,才能完成计算。
- 时间复杂性: $O(n*L)$
 L 是算术表达式的长度, n 是操作符的个数



后缀表达式

- 运算符紧跟在两个操作数之后的表达式称作后缀表达式。波兰逻辑学家**Lukasiewicz**提出，也称逆波兰式。（前缀表达式称为波兰式）
- 例： 后缀表达式 $A B \times C /$
中缀表达式 $A \times B / C$



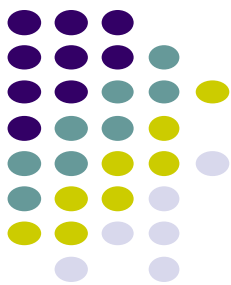
后缀表达式的特点

- 后缀表达式没有括号
- 不存在优先级的差别，计算过程完全按照运算符出现的先后次序进行
- 演示
 - ✓ $1\ 2\ 3\ 4\ +\ -\ *$
 - ✓ $(100)\ (20)\ 3\ 4\ *\ -\ /$



后缀表达式的计算

1. 左起依次读取后缀表达式的一个符号；
2. 若读入的是操作数，则将其压入栈；
3. 若读入的是运算符，则从栈中连续弹出两个元素，进行相应的运算，并将结果压入栈中。
4. 若读入的是结束符，则栈顶元素是计算结果。



中缀表达式转后缀表达式

中缀表达式	后缀表达式
$a+b$	$ab+$
$a+b\times c$	$abc\times+$
$a\times b\times c+c\times d$	$ab\times c\times cd\times+$
$(a+b)\times((c-d)\times e+f)$	$ab+cd-e\times f+\times$

- ❑ 操作数出现的顺序一样
- ❑ 区别在于操作符的顺序，后缀表达式中操作符按计算顺序出现



引入操作符栈，确定出栈顺序

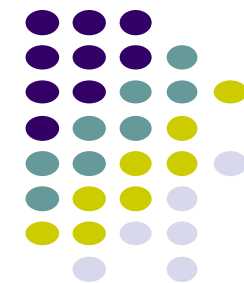
□ 规则1：运算符优先级

当前操作符 $>$ 栈顶操作符，压栈；否则，弹栈

□ 规则2：括号

左括号压栈，右括号弹栈至左括号

□ 规则3：结束符 弹栈至空栈



算术表达式的计算

□ 方法一

- ✓ 中缀转后缀 扫描一遍
- ✓ 计算后缀表达式 扫描一遍
- ✓ $T(L) = 2L$

□ 方法二

- ✓ 边转换、边计算 只需扫描一遍
- ✓ $T(L) = L$

演示（方法二）

$$\square 1 \times (2 / 3 + 4 - 5)$$





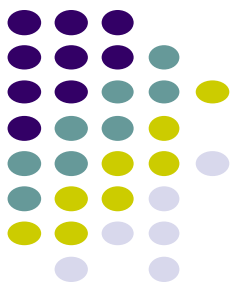
S1. READ(x). // 左起依次读取中缀表达式的 1 个符号

S2. IF x 是操作数 THEN $Q \leftarrow x$.

S3. IF x 是运算符 THEN // 运算符包括括号
IF S 为空 THEN $S \leftarrow x$.
ELSE // S 非空. $top(S)$: 栈顶元素. '(' 优先级最高.
IF x 是 '(' OR 优先级 $x > top(S)$ THEN $S \leftarrow x$
ELSE IF x 是 ')' THEN (
WHILE $top(S) \neq '('$ DO
($d2 \leftarrow Q$. $d1 \leftarrow Q$. $t \leftarrow S$ $Q \leftarrow d1$ $t \leftarrow d2$.)
 $t \leftarrow S$.) // 弹出 '('
ELSE (WHILE (S 非空 AND 优先级 $top(S) \geq x$) DO
($d2 \leftarrow Q$. $d1 \leftarrow Q$. $t \leftarrow S$ $Q \leftarrow d1$ $t \leftarrow d2$.)
 $S \leftarrow x$.)

S4. IF x 是 '#' THEN(// '#' : 中缀表达式的结束符
WHILE S 非空 DO ($d2 \leftarrow Q$. $d1 \leftarrow Q$. $t \leftarrow S$. $Q \leftarrow d1$ $t \leftarrow d2$.)
RETURN).

S5. GOTO S1 . ■



栈的应用小结

- 函数调用和返回（**1946图灵**）
 - 递归
 - **undo**功能
 -
-
- 凡数据符合后进先出性的问题，可考虑应用栈



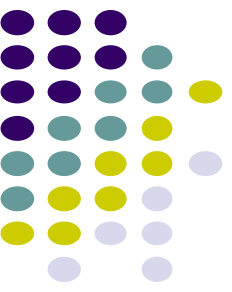
数据结构的封装

□ 数据结构通常使用不需要自行封装

- ✓ 语言提供了数据结构：STL，Java、Python等自带
- ✓ 追求高效时，如**ACM、CSP**认证等，可以纯手写(全局)或优化现有结构(如STL的allocator)

□ 掌握数据结构的封装是必要的

- ✓ 现有的数据结构不存在 或 达不到要求（如图等）
- ✓ 高阶技能和专业课程要求（数据结构及后续课）



STL中关于栈的实现

□ **#include <stack>**

...

using namespace std;

□ **stack<int > s;** //不用声明大小

□ **s.push(x);**

□ **s.top();** //相当于**peek**

□ **s.pop();** //只弹出栈顶，不取值，与**top**合用

□ **s.empty();** //判空

□ **s.size();** //栈的长度

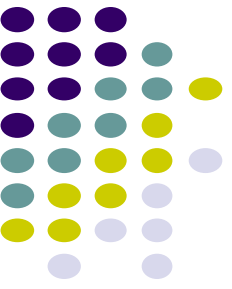


栈数据类型

□ 场景：需要多个相同类型的栈：

□ C-Style (C++99 和非纯c环境)

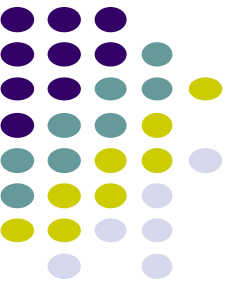
```
struct AStack{  
    int stackArray[MaxStackSize];  
    int top;  
};  
void init(AStack& s){  
    s.top = -1;  
}
```

```
void push(AStack& s,int x){  
    s.stackArray[++s.top]=x;  
}
```

```
int pop(AStack& s){  
    return s.stackArray[s.top--];  
}
```

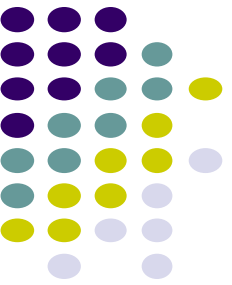
测试



```
int main(){
    int n,i;
    AStack s;
    init(s);

    scanf("%d",&n);
    for(i=1;i<=n;i++) push(s,i);

    for(i=1;i<=n;i++) printf("%d ",pop(s));
    printf("\n");
}
```



栈泛型

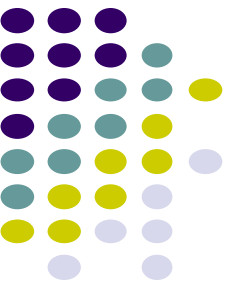
- 场景2：需要多个不同类型的栈
- C-Style (C++99 和非纯c环境)

template<class T>

```
struct AStack{  
    T stackArray[MaxStackSize];  
    int top;  
};
```

template<class T>

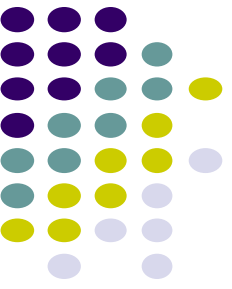
```
void init(AStack<T>& s){  
    s.top = -1;  
}
```



```
template<class T>
void push(AStack<T>& s,T x){
    s.stackArray[++s.top]=x;
}
```

```
template<class T>
T pop(AStack<T>& s){
    return s.stackArray[s.top--];
}
```

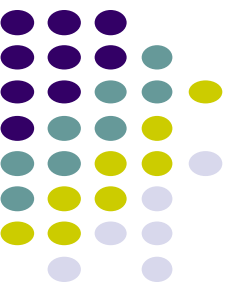
测试



```
int main(){
    int n,i;
    AStack<int> s;
    init(s);

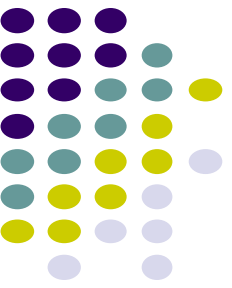
    scanf("%d",&n);
    for(i=1;i<=n;i++) push(s,i);

    for(i=1;i<=n;i++) printf("%d ",pop(s));
    printf("\n");
}
```



面向对象方式

```
class AStack{  
    int stackArray[MaxStackSize];  
    int top;  
  
    public:  
        AStack(){top=-1;}  
        void push(int x){stackArray[++top]=x;}  
        int pop(){return stackArray[top--];}  
        void clear(){top = -1;}  
};
```



struct扩充方式（C++99）

```
struct AStack{  
    int stackArray[MaxStackSize];  
    int top;  
  
    AStack(){top=-1;}  
    void push(int x){stackArray[++top]=x;}  
    int pop(){return stackArray[top--];}  
    int peek(){return stackArray[top];}  
};
```



struct VS class

- **C++对C中的struct进行了扩充（C++99）**
 - ✓ 可包含成员函数
 - ✓ 可继承
 - ✓ 可多态
- **struct 与 class语法类似，意义不同**
 - ✓ 默认权限：struct默认public，class 默认private
 - ✓ 传递方式：传值 VS 传址



封装注意事项：模板类

□ 封装鼓励声明和实现分离

- ✓ 声明放在.h文件中
- ✓ 实现放在.cpp文件中

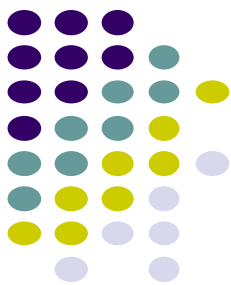
□ 模板类（**template**）的声明和实现都放在头文件中，不能分开成.h 和 .cpp



封装注意事项：重复include

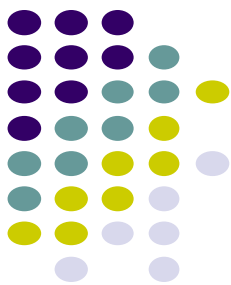
□ 防止重复include同一个头文件

- ✓ //astack.h
- ✓ #ifndef _A_STACK
- ✓ #define _A_STACK
- ✓
- ✓ #endif



数据结构封装小结

- 建议： **C++**方式 或 **struct**扩充方式
- 可以使用**STL**解题，除非题目明确说明不允许 或 题目本身要求实现对应数据结构



总结

- 栈的定义、特性和基本操作；
- 栈的顺序存储方式及实现（顺序栈）；
- 栈的链式存储方式及实现（链栈）；
- 栈的应用
 - ✓ 数据容器满足LIFO
 - ✓ 经典示例：表达式计算
- 数据结构的封装（衔接）
 - ✓ OO、struct



现在开始编程

□ 做中学

- ✓ 理论源于实践；用数据结构求解问题
- ✓ 做什么（实现每种数据结构、在线练习、作业）
- ✓ 心理准备（基础、修炼）

□ 尝试好的编程习惯

- ✓ 想好了再写（建立计算模型）
- ✓ 写好后读一遍（检查小错误和逻辑）
- ✓ 学会调试（输出中间结果，写调试函数）
- ✓ 记下犯过的错误
- ✓ 有一点代码风格（空行、缩进、变量、必要注释）
- ✓ 建立自己的代码



第3-4章 任务

□ 在线测试平台注册

✓ [pintia.cn/学习通](http://pintia.cn/)

□ 慕课

✓ 在线学习/预习 第 3 章 视频

✓ 自学第4章的视频: 4.1 和 4.2 (考、不讲)

□ 作业

✓ P72: 3-2, 3-6, 3-8,

3-11, 3-17, 3-25

✓ 本章结束后一周在线提交