



Chapter 2

Stacks



Chapter 2 Stacks

2.1 Stack Specifications



2.2 Implementation of Stacks

2.3 Application of Stacks

2.4 Application: Bracket Matching

2.5 Abstract Data Types and Their Implementations

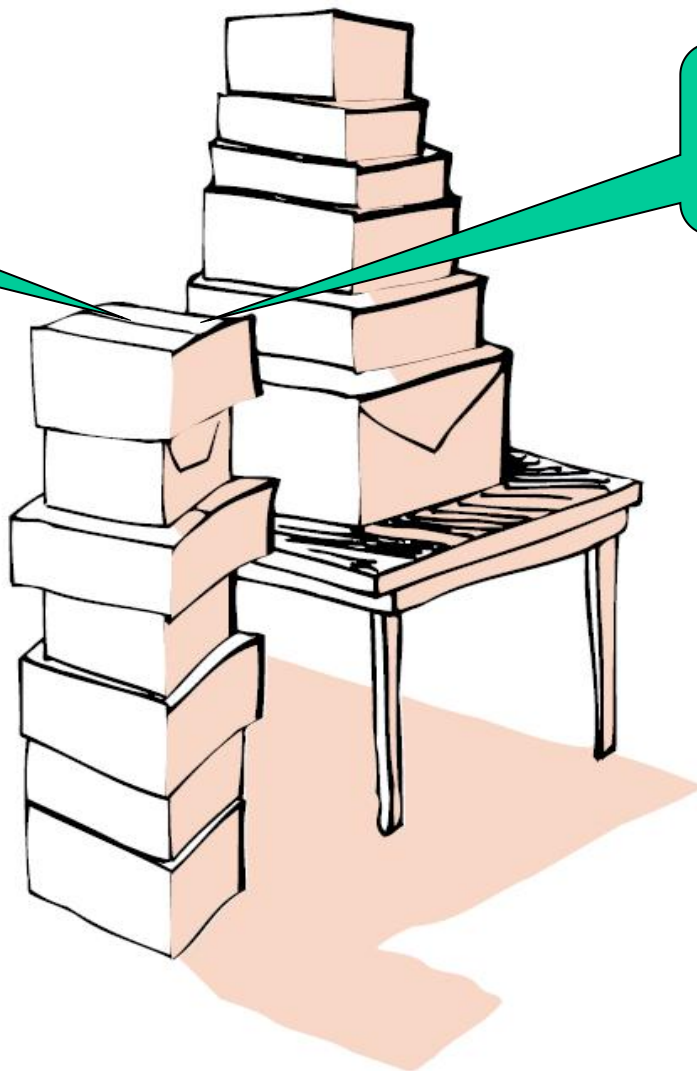
2.1 Stack Specifications

- A ***stack*** is a version of a list ($\mathbf{a_1, a_2, a_3, \dots a_n}$) that is particularly useful in applications involving reversing, such as the problem of Section 2.1.1. In a stack data structure, all insertions and deletions of entries are made at one end, called the ***top*** of the stack. A helpful analogy (see Figure 2.1) is to think of a stack of trays or of plates sitting on the counter in a busy cafeteria. 
- Throughout the lunch hour, customers take trays off the **top** of the stack, and employees place returned trays back on top of the stack. The tray most recently put on the stack is the first one taken off. The **bottom** tray is the first one put on, and the last one to be used. 

■ 例如一叠书或一叠盘子。

You **take** one
on the top

You **put** one
on the top



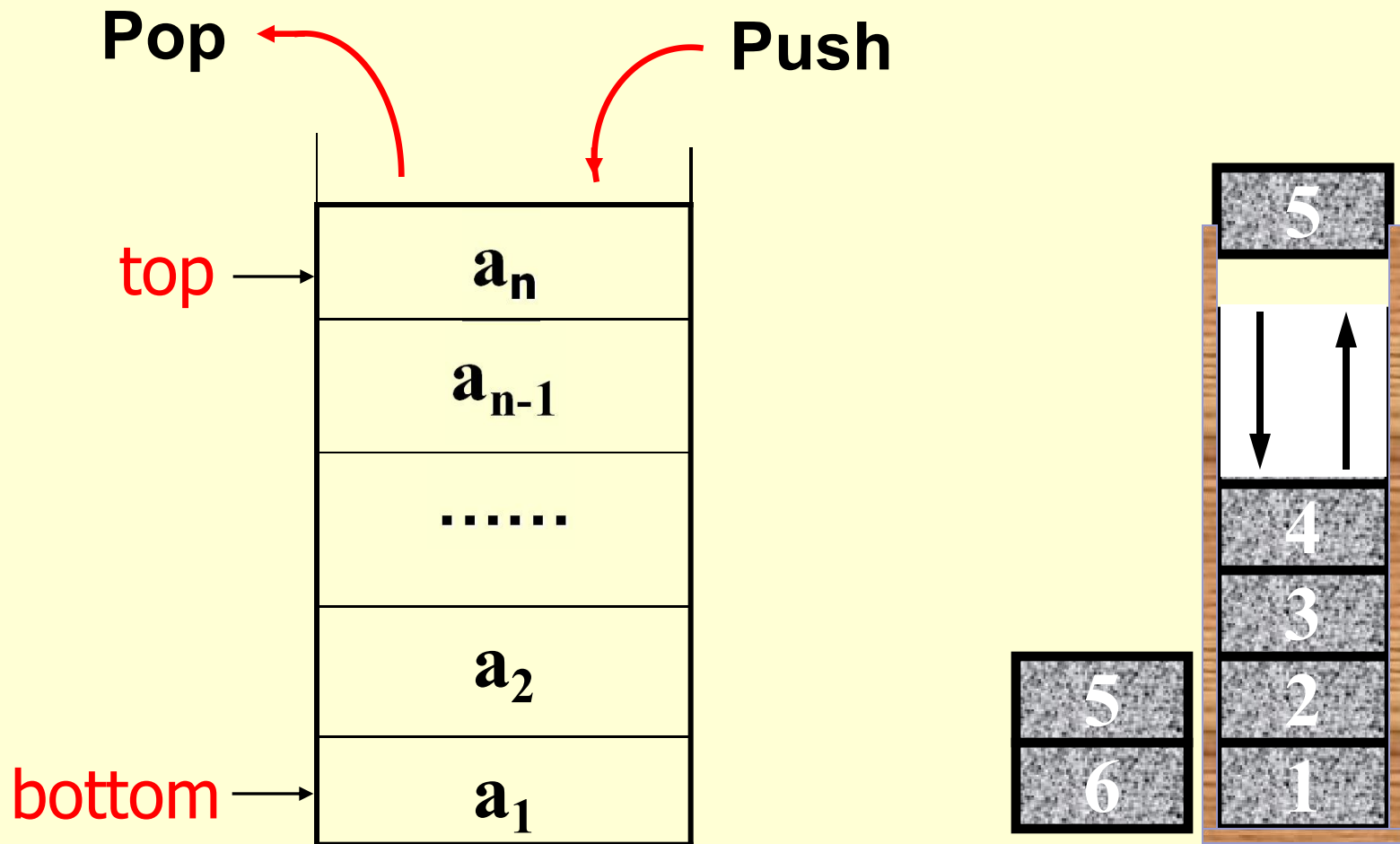



Figure 2.1 Pushing and popping a stack

2.1 Stack Specifications

- When we add an item to a stack, we say that we push it onto the stack, and when we remove an item, we say that we pop it from the stack.
- See Figure 2.2. Note that the last item pushed onto a stack is always the first that will be popped from the stack. This property is called *last in, first out*, or **LIFO** for short. 

■ 作为一种抽象数据类型，常用的栈运算有：

运算	含义
----	----

MakeNull(S) 使**S**成为一个空栈。


Top(S) 这是一个函数，函数值为**S**中的栈顶元素。

Pop(S) 从栈**S**中删除栈顶元素，简称为抛栈。


Push(S,x) 在**S**的栈顶插入元素**x**，简称为将元素**x**入栈。

Empty(S) 这是一个函数。当**S**为空栈时，函数值为**true**，否则函数值为**false**。

- 栈有两种存储表示方式：顺序表示和链式表示。它们可以分别用**数组实现**和**指针实现**。
- 本章介绍顺序表示。

- In our program we shall rely on the ***standard template library*** of C++ (usually *library* called the **STL**) to provide a **class** that implements stacks. 

#include <stack>

- we have used not only the methods **push()**, *initialization* **top()**, and **pop()** of the stack called numbers, but we have made crucial use of the implicit initialization of numbers as an **empty** stack. 
- 另外可以使用C++的模板参数(**template parameter**)指定栈元素的不同数值类型，例如：
stack<double>numbers;
stack<int>numbers;

■ 使用C++ **STL**(Standard Template Library) 中栈的实现:

#include <stack>

- 有了以上的声明就可以直接初始化一个空栈对象并且对它进行*push*、*pop*、*top*以及*empty*等操作，另外可以使用C++的模板参数(*template parameter*)指定栈元素的不同数值类型，例如：

stack<double>numbers;

stack<int>numbers;

第一个例子:表的逆置 (First Example: Reversing a List)

```
#include <stack>
```

```
int main( )
```

```
/* Precondition : The user supplies an integer n and n decimal numbers.
```

```
Postcondition: The numbers are printed in reverse order.
```

```
Uses: The STL class tack and its methods */
```

```
{ int n;  
  double item;  
  stack<double> numbers; // declares and initializes a stack of numbers  
  cout << " Type in an integer n followed by n decimal numbers." <<  
    endl  
    << " The numbers will be printed in reverse order."<< endl;  
  cin >> n;  
  for (int i = 0; i < n; i++) {  
    cin >> item;  
    numbers.push(item);  
  }  
  cout << endl << endl;  
  while (!numbers.empty( )) {  
    cout << numbers.top( ) << " ";  
    numbers.pop( );  
  }  
  cout << endl;  
}
```

C++概念回顾

■ 信息隐藏

C++的STL为常用的数据结构(包括栈)提供了不同的实现方式。实现方式不同则性能不同。客户程序的代码不应该依赖这些数据结构的实现。客户程序可以通过第二个可选的模板参数选用所期望的实现方式。

■ 本书的大部分内容可以看作是对C++标准模板库实现方式的介绍。

- **Constructor**

- **Entry Types, Generics**

- **Error Processing**

- **Specication for Methods**

Constructor

- The *constructor* is a function with the same name as the corresponding class and no return type. It is invoked automatically whenever an object of the class is declared.

Stack :: Stack();

Pre: None.

Post: The Stack exists and is initialized to be empty.

Entry Types, Generics

- For generality, we use `Stack_entry` for the type of entries in a `Stack`.
- A client can specify this type with a definition such as
`typedef int Stack_entry;`
- or
`typedef char Stack_entry;`
- The ability to use the same underlying data structure and operations for different entry types is called **generics**. **`typedef`** provides simple generics. Starting in Chapter 6, we shall use C++ **templates** for greater generality.

Error Processing

- We shall use a single **enumerated type** called `Error_code` to report errors from all of our programs and functions.
- The enumerated type `Error_code` is part of the utility package in Appendix C(P.678). Stacks use three values of an `Error_code`:

success, overflow, underflow

- Later, we shall use several further values of an `Error_code`.

Specication for Methods

Error_code Stack :: pop();

Pre: None.

Post: If the Stack is not empty, the top of the Stack is removed. If the Stack is empty, an Error_code of underflow is returned and the Stack is left unchanged.

Error_code Stack :: push(**const** Stack_entry &item);

Pre: None.

Post: If the Stack is not full, item is added to the top of the Stack. If the Stack is full, an Error_code of overflow is returned and the Stack is left unchanged.

2.2 Implementation of Stacks

```
const int maxstack = 10; // small value for testing
class Stack {
public:
    Stack( );
    bool empty( ) const;
    Error_code pop( );
    Error_code top(Stack_entry &item) const;
    Error_code push(const Stack_entry &item);
private:
    int count;
    Stack_entry entry[maxstack];
};
```

注意：

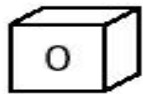
1. 构造函数**Stack()**与类名相同且无返回值类型；
2. 返回值类型**Error_code()**有三个枚举值**success**、**overflow**以及**underflow**以便实现出错处理。

顺序栈的类说明

- 用一个数组来存放栈的元素并用一个计数器记录栈中实际的元素个数。
- 请注意**private**的作用它使客户程序只能访问**push()**、**pop()**和**top()**等从而实现“封装”

```
const int maxstack = 10; // small value for testing
class Stack {
public:
    Stack( );
    bool empty( ) const;
    Error_code pop( );
    Error_code top(Stack_entry &item) const;
    Error_code push(const Stack_entry &item);
private:
    int count;
    Stack_entry entry[maxstack];
};
```

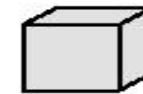
count



entry



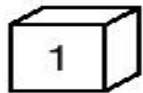
...



[maxstack - 1]

(a) Stack is empty.

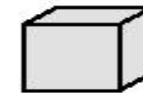
count



entry



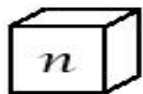
...



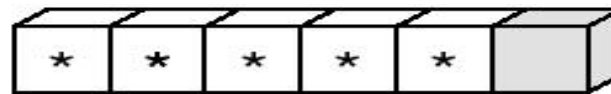
[maxstack - 1]

(b) Push the first entry.

count



entry



...



[maxstack - 1]

(c) n items on the stack

push()的实现

```
Error_code Stack :: push(const Stack_entry &item)
```

```
/* Pre: None.
```

```
   Post: If the Stack is not full,item is added to the top  
         of the Stack . If the Stack is full, an Error_code of  
         overflow is returned and the Stack is left  
         unchanged. */
```

```
{ Error_code outcome = success;  
  if (count >= maxstack)  
      outcome = overflow;  
  else  
      entry[count++] = item;  
  return outcome;  
}
```

pop()的实现

```
Error_code Stack :: pop( )
```

```
/* Pre: None.
```

Post: If the Stack is not empty, the top of the Stack is removed. If the Stack is empty, an Error_code of underflow is returned. */

```
{
```

```
    Error_code outcome = success;
```

```
    if (count == 0)
```

```
        outcome = underflow;
```

```
    else --count;
```

```
    return outcome;
```

```
}
```

top()的实现

```
Error_code Stack :: top(Stack_entry &item) const  
/* Pre: None.
```

Post: If the Stack is not empty, the top of the Stack is returned in item . If the Stack is empty an Error_code of underflow is returned. */

```
{  
    Error_code outcome = success;  
    if (count == 0)  
        outcome = underflow;  
    else  
        item = entry[count - 1];  
    return outcome;  
}
```

栈判空及初始化的实现

```
bool Stack :: empty( ) const
```

```
/* Pre: None.
```

```
   Post: If the Stack is empty, true is returned. Otherwise false is returned. */
```

```
{  bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}
```

```
Stack :: Stack( )
```

```
/* Pre: None.
```

```
   Post: The stack is initialized to be empty. */
```

```
{  count = 0; }
```

2.3 Application of Stacks

- 编译器检查程序的语法错误
 - 检查是否满足匹配原则, 例如括号匹配
- 利用栈实现函数调用
 - 在介绍递归时会详细介绍
- 表达式求值
 - 例如后缀表达式的计算
- 其它
 - 行编辑器

2.3 Application of Stacks

- 算术表达式的三种表示形式
 1. 前缀表达式(Polish notation)
 2. 中缀表达式(for person use)
 3. 后缀表达式(Reverse Polish Notation)
- 三种表达式之间的转换：按运算的优先次序全部加上括号，逐个括号写成另一种表示式

(括号 \rightarrow * , \rightarrow +, -)

2.4 Applications: Balancing Symbols

- 检查一个输入的文本文件中的以下括号是否匹配{ }, ()以及[]——“最近匹配原则”，例：
{0[0]}
- 逐个输入文本中的字符将遇到的每一个开括号看作是未匹配的字符并保存起来直到遇到一个能与之匹配的闭括号时才将其删除当输入终止时检查是否仍保存着未匹配的括号
- 由于括号的检查次序与输入次序相反因此使用Stack保存括号



Check if parenthesis **()**, brackets **[]**, and braces **{ }** are balanced.

Algorithm {

Make an empty stack S;

while (read in a character c) {

if (c is an opening symbol)

 Push(c, S);

else if (c is a closing symbol) {

if (S is empty) { ERROR; **exit**; }

else { /* stack is okay */

if (Top(S) doesn't match c) { ERROR; **exit**; }

else Pop(S);

 } /* end else-stack is okay */

} /* end else-if-closing symbol */

} /* end while-loop */

if (S is not empty) ERROR;

}

$T(N) = O(N)$
where N is the length
of the expression.
This is an
on-line algorithm.

2.5 Abstract Data Types and Their Implementations

- 关于“抽象”或称为“归纳”
- ADT的定义
- 对数据说明的求精

基本定义

■ 类型

类型是一个集合，集合的元素称为类型的值(value)

■ 原子类型

其值不能在分的类型，称为原子类型。如
`int, float, char`。

■ 构造类型

用原子类型或其他类型建立的新的类型，称为构造类型。

关于“抽象”或称为“归纳”

- 考察用“数组+计数器”取代栈操作的情形
- 缺陷
 - 程序员需要花费额外的时间考虑操作的实现细节而不是如何使用这些操作
- 改进方法
 - 对相似的事物抽象/归纳出共同的概念
- 抽象的步骤
 - 识别并表达数据之间的逻辑关系/逻辑结构
 - 考虑用什么方法实现逻辑结构才能得到良好的性能和效率

ADT的定义(Abstract Stacks)

DEFINITION A *sequence of length* 0 is empty. A *sequence of length* $n \geq 1$ of elements from a set T is an ordered pair (S_{n-1}, t) where S_{n-1} is a sequence of length $n - 1$ of elements from T , and t is an element of T .

DEFINITION A *stack* of elements of type T is a finite sequence of elements of T , together with the following operations:

1. *Create* the stack, leaving it empty.
2. Test whether the stack is *Empty*.
3. *Push* a new entry onto the top of the stack, provided the stack is not full.
4. *Pop* the entry off the top of the stack, provided the stack is not empty.
5. Retrieve the *Top* entry from the stack, provided the stack is not empty.

ADT的定义

- 定义一个ADT的具体工作
 - 描述数据成员之间的相互逻辑关系如栈中数据成员之间的相互逻辑关系是“序列”
 - 说明可以对数据成员实现的操作如栈的5个基本操作

对数据说明的求精

- **abstract**层确定数据之间的相互关系以及所需什么操作，但不考虑数据如何存储以及操作如何实现。
- **data structure**层详细地分析各个方法的行为并选择合适的实现方式。
- **implementation**层决定如何在机器的内存中表示数据结构。
- **application**层解决应用中的所有细节问题。

精对数据说明的求精(续)

- 概念层 **conceptual** = *abstract* 层
+ *data structures* 层
——“分析”
- 算法层 **algorithmic** = *data structures* 层
+ *implementation* 层
——“设计”
- 程序设计层 **programming** = *implementation* 层
+ *application* 层
——“实现”