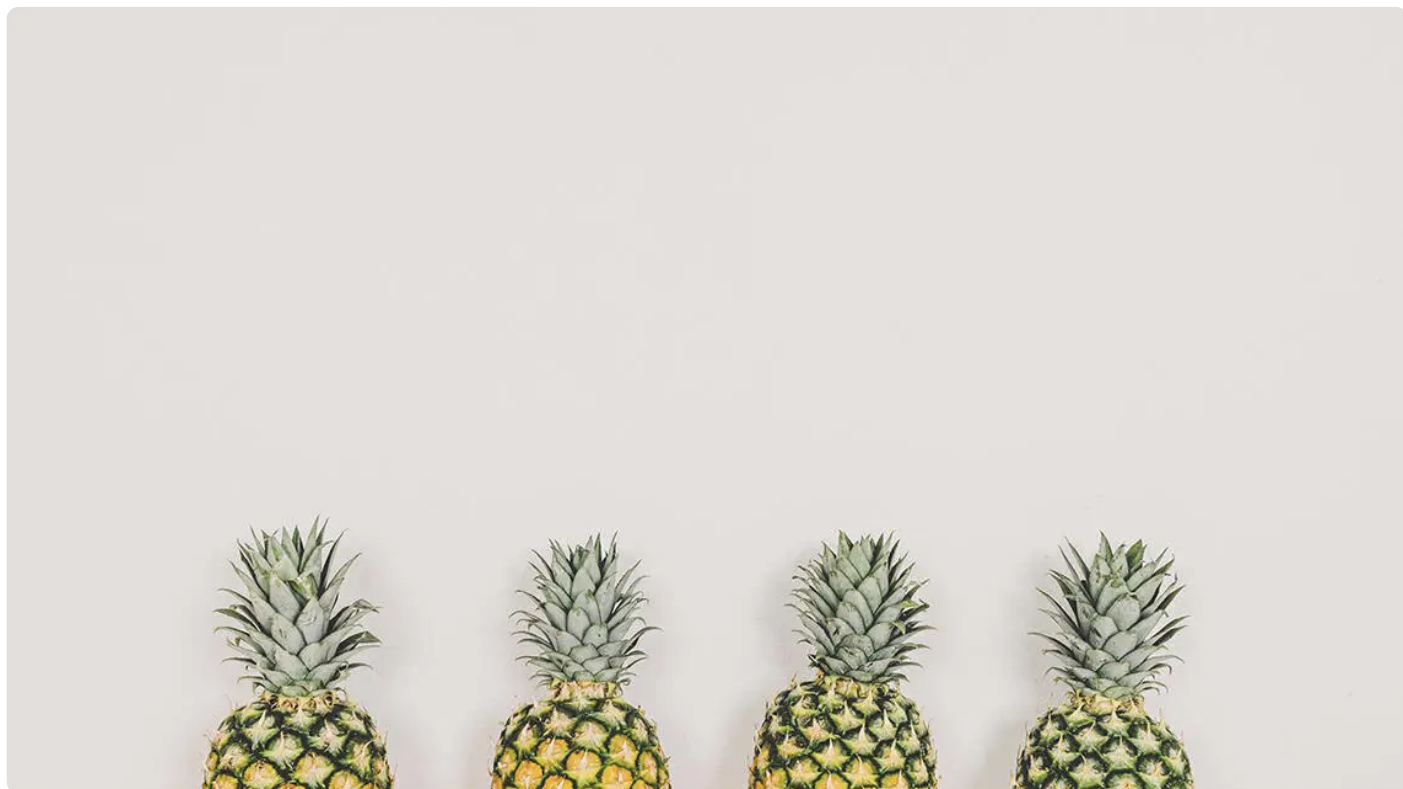


08 | 栈：如何实现数据的后进先出？

2023-03-01 王健伟 来自北京

《快速上手C++数据结构与算法》

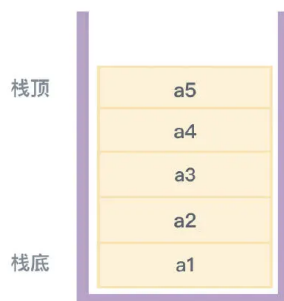


你好，我是王健伟。

从“链表”毕业之后，我们就要进入“栈”的学习了。作为一种耳熟能详的数据结构，“栈”到底是个什么东西呢？

还记得我们之前提到过的线性表吗？没错，栈仍旧是一种线性表。不过它只能在一端进行插入和删除操作，先入栈的数据只能后出来，而后入栈的数据只能先出来。所以栈具有先进后出或者后进先出的特性。通常来说，我们可以把栈理解为一种**受限的线性表**。

如果我们把栈比成一个类似木桶这样的容器，栈有两端，把允许进行插入和删除操作的一端称为**栈顶**（top）也就是桶口，或者称为线性表的表尾，而另一端称为**栈底**（bottom）也就是桶底，或者称为线性表的表头。不包含任何数据的栈，叫做空栈（空线性表）。



整体结构理清之后，我们再说相关的操作。向栈中插入元素，可以叫做**入栈**或进栈，从栈中删除元素，就叫**出栈**。除了能够在表尾插入和删除数据外，对于栈这种数据结构，在任何其他位置插入和删除数据都不应该被允许。你只能提供符合这个规定的操作接口，否则实现的就不是栈了。

栈也被称为**后进先出（Last In First Out: LIFO）的线性表**，这意味着最后放入到栈里的数据（插入数据）只能最先被从栈中拿出来（删除数据）。

其实，我们生活中满足栈这种后进先出的情形非常多，比如往抽屉里放东西，先放进去的肯定会被堆到最里面，所以只能最后取出，而最后放入的物品往往在最前面或者最上面，所以会被最先取出。

如果用示意图表示用栈存取数据的过程，就会像图 1 一样：

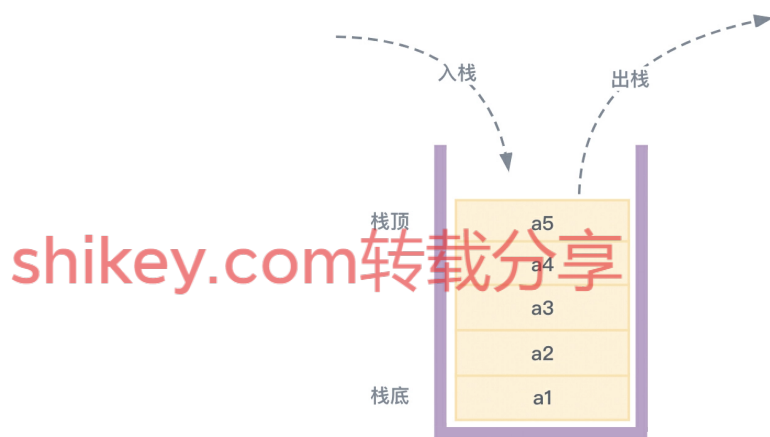


图1 栈存取数据示意图

在图 1 中，如果分别将数据 a1、a2、a3、a4、a5 存入栈中，那么在将数据出栈的时候，顺序就应该是 a5、a4、a3、a2、a1（与入栈顺序正好相反）。

我们刚刚说过，栈是**受限的线性表**，比如因为只能在栈顶进行元素的插入和删除操作，所以也无法指定插入和删除操作的位置，所以，栈所支持的操作，可以理解为线性表操作的**子集**，一般包括栈的创建、入栈（增加数据）、出栈（删除数据）、获取栈顶元素（查找数据）、判断栈是否为空或者是否已满等操作。

栈的顺序存储（顺序栈）

所谓顺序栈，就是顺序存储（用一段连续的内存空间依次存储）栈中的数据。前面我们在学习顺序表的时候就提出过 2 种保存数据的方案同样适合顺序栈。

1. 通过为一维数组**静态**分配内存的方式来保存数据。
2. 通过为一维数组**动态**分配内存的方式来保存数据。

为了顺序栈中数据存满时可以对栈进行扩容，在这里，我会采用第 2 种保存数据的方案来编写顺序栈的实现代码。

此外，为了考虑到元素存取的便利性，将数组下标为 0 的一端作为栈底最合适。

顺序栈的类定义、初始化和释放操作

我们还是和之前的讲解一样，先说类定义、初始化以及释放操作。

 复制代码


```
1 #define InitSize 10    //动态数组的初始尺寸
2 #define IncSize  5     //当动态数组存满数据后每次扩容所能多保存的数据元素数量
3
4 template <typename T> //T代表数组中元素的类型
5 class SeqStack
6 {
7 public:
8     SeqStack(int length = InitSize);    //构造函数，参数可以有默认值
9     ~SeqStack();                       //析构函数
10
11 public:
12     bool Push(const T& e); //入栈（增加数据）
13     bool Pop(T& e);        //出栈（删除数据），也就是删除栈顶数据
14     bool GetTop(T& e);     //读取栈顶元素，但该元素并没有出栈而是依旧在栈中
15
```

```

16     void DispList();                //输出顺序栈中的所有元素
17     int ListLength();               //获取顺序栈的长度（实际拥有的元素数量）
18
19     bool IsEmpty();                 //判断顺序栈是否为空
20     bool IsFull();                  //判断顺序栈是否已满
21
22 private:
23     void IncreaseSize();           //当顺序栈存满数据后可以调用此函数为顺序栈扩容
24
25 private:
26     T* m_data;                      //存放顺序栈中的元素
27     int m_maxsize;                  //动态数组最大容量
28     int m_top;                      //栈顶指针(用作数组下标)，指向栈顶元素，该值为-1表示空栈
29 };
30
31 //通过构造函数对顺序栈进行初始化
32 template <typename T>
33 SeqStack<T>::SeqStack(int length)
34 {
35     m_data = new T[length];        //为一维数组动态分配内存，该值和算法空间复杂度无关，空间复杂度一
36     m_maxsize = length;             //顺序栈最多可以存储m_maxsize个数据元素
37     m_top = -1;                     //空栈
38 }
39
40 //通过析构函数对顺序栈进行资源释放
41 template <typename T>
42 SeqStack<T>::~SeqStack()
43 {
44     delete[] m_data;
45 }

```

在 main 主函数中，加入代码创建一个初始大小为 10 的顺序栈对象。

 复制代码

```
1 SeqStack<int> seqobj(10);
```

shikey.com转载分享

在利用上面的代码创建 seqobj 对象之后，顺序栈看起来就会是图 2 的样子，此时是一个空栈：

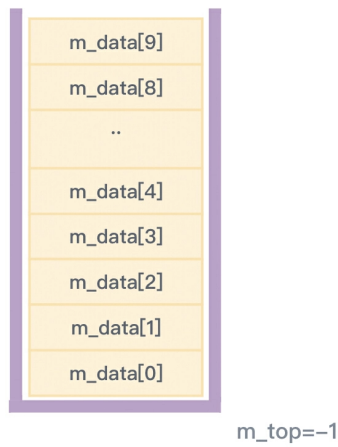


图2 能保存10个元素的顺序栈（空栈）

顺序栈的入栈、出栈、获取栈顶元素、显示元素等操作


首先是入栈以及扩容的相关操作。在阅读代码的同时，不要忘了在心里思考一下算法对应的时间复杂度。

复制代码

```
1 //入栈（增加数据），通常时间复杂度为O(1)，但一旦需要扩容，时间复杂度就会变成O(n)了
2 template <typename T>
3 bool SeqStack<T>::Push(const T& e)
4 {
5     if (IsFull() == true)
6     {
7         //cout << "顺序栈已满，不能再进行入栈操作了!" << endl;
8         //return false;
9         IncreaseSize(); //扩容
10    }
11
12    m_top++;           //栈顶指针向后走
13    m_data[m_top] = e; //本行和上一行可以合并写成一行代码: m_data[++m_top] = e;
14    return true;
15 }
16
17 //当顺序栈存满数据后可以调用此函数为顺序栈扩容，时间复杂度为O(n)
18 template<class T>
19 void SeqStack<T>::IncreaseSize()
20 {
21     T* p = m_data;
22     m_data = new T[m_maxsize + IncSize]; //重新为顺序栈分配更大的内存空间
23     for (int i = 0; i <= m_top; i++)
24     {
25         m_data[i] = p[i];                //将数据复制到新区域
26     }
```

```
27     m_maxsize = m_maxsize + IncSize;    //顺序栈最大长度增加IncSize
28     delete[] p;                          //释放原来的内存空间
29 }
```


其次，就是相应的出栈、读取栈顶元素代码。

 复制代码

```
1  //出栈（删除数据），也就是删除栈顶数据，时间复杂度为O(1)
2  template <typename T>
3  bool SeqStack<T>::Pop(T& e)
4  {
5      if (IsEmpty() == true)
6      {
7          cout << "当前顺序栈为空，不能进行出栈操作!" << endl;
8          return false;
9      }
10
11     e = m_data[m_top]; //栈顶元素值返回到e中。有的实现版本不会在Pop()成员函数中返回栈顶元素，
12     m_top--;           //本行和上一行可以合并写成一行代码：e = m_data[m_top--];
13     return true;
14 }
15
16 //读取栈顶元素，但该元素并没有出栈而是依旧在栈顶中，因此m_top值不会发生改变，时间复杂度为O(1)
17 template <typename T>
18 bool SeqStack<T>::GetTop(T& e)
19 {
20     if (IsEmpty() == true)
21     {
22         cout << "当前顺序栈为空，不能读取栈顶元素!" << endl;
23         return false;
24     }
25
26     e = m_data[m_top]; //栈顶元素返回到e中。
27     return true;
28 }
```

shikey.com转载分享

最后，是一些顺序栈的常用操作，比如输出所有元素、获取长度、判断是否为空、是否已满。

 复制代码

```
1  //输出顺序栈中的所有元素，时间复杂度为O(n)
2  template<class T>
3  void SeqStack<T>::DispList()
```


```

4 {
5     //按照从栈顶到栈底的顺序来显示数据
6     for (int i = m_top; i >= 0; --i)
7     {
8         cout << m_data[i] << " "; //每个数据之间以空格分隔
9     }
10    cout << endl; //换行
11 }
12
13 //获取顺序栈的长度（实际拥有的元素数量），时间复杂度为O(1)
14 template<class T>
15 int SeqStack<T>::ListLength()
16 {
17     return m_top+1;
18 }
19
20 //判断顺序栈是否为空，时间复杂度为O(1)
21 template<class T>
22 bool SeqStack<T>::IsEmpty()
23 {
24     if (m_top == -1)
25     {
26         return true;
27     }
28     return false;
29 }
30
31 //判断顺序栈是否已满，时间复杂度为O(1)
32 template<class T>
33 bool SeqStack<T>::IsFull()
34 {
35     if (m_top >= m_maxsize - 1)
36     {
37         return true;
38     }
39     return false;
40 }

```

shikey.com转载分享

在 main 主函数中，继续增加下面的代码。

 复制代码

```

1 seqobj.Push(150);
2 seqobj.Push(200);
3 seqobj.Push(300);
4 seqobj.Push(400);
5 seqobj.DispList();

```

```

6  int eval = 0;
7  seqobj.Pop(eval);
8  seqobj.Pop(eval);
9  cout << "-----" << endl;
10 seqobj.DispList();

```

执行结果如下：

```
400 300 200 150
```

```
-----
```

```
200 150
```

从结果可以看到，程序是先把 150、200、300、400 进行了入栈操作，此时顺序栈中的数据如图 3(a) 所示，最先进去的 150 是在最下面的。然后又将栈顶的两个元素出栈，此时顺序栈中的数据如图 3(b) 所示。



图3 先入栈4个元素再出栈2个元素的顺序栈示意图

shikey.com转载分享

在图 3 中，先入栈了 4 个元素，而后再出栈了 2 个元素，出栈时虽然只修改了 m_top 栈顶指针，并没有把对应的元素值 (m_data[2]和 m_data[3]) 抹除，但这些值已经没有用处了，下次再进行入栈操作的时候会被入栈的新值覆盖。你可以在 mian 主函数中继续添加下面的代码并跟踪调试看看。


```
1 seaobi.Push(8100):
```

在有的实现栈的范例代码中，会让 `m_top` 的初始值等于 0（指向 0 的位置），那么判断栈是否为空的代码（`IsEmpty` 函数）也就是判断 `m_top` 是否等于 0，而判断栈满（`IsFull` 函数）的条件也应该变成 `if (m_top >= m_maxsize)`。这种实现方式，实际就是让 `m_top` 代表下一个可以放入栈中的元素的下标，当数据入栈（`Push` 函数）时，代码行 `m_top++`; 和代码行 `m_data[m_top] = e;` 的执行就需要互换顺序，而当数据出栈（`Pop` 函数）时，代码行 `e = m_data[m_top];` 和代码行 `m_top--`; 的执行也需要互换顺序。

共享栈


除了刚才我们提到的常用操作外，这里再补充一个“共享栈”，也就是**两个顺序栈**共享存储空间。

为什么会提出这个概念呢？之前我们提到的顺序栈，一个比较大的缺点是保存数据的空间初始尺寸不好确定，如果太大，就会浪费空间，如果太小，那么存满数据后再入栈新数据就需要扩容，而扩容就又需要开辟一整块更大的新区域并将原有数据复制到新区域，操作起来比较耗费性能。

不过，我们可以设想一下。假设有两个相同数据类型的顺序栈，如果分别为他们开辟了保存数据的空间，那是不是就可能出现，第一个栈的数据已经存满了而另一个栈中还有很多存储空间的情形呢？那么，如果开辟出来一块保存数据的空间后，让这两个栈同时使用，也就是共享这块空间，是不是也许就能达到最大限度利用这块空间、减少浪费的目的呢？这就是共享栈的含义。

具体如何做到呢？我们还是先从代码层理解一下。

```
1 //共享栈
2 template <typename T> //T代表数组中元素的类型
3 class ShareStack
4 {
5 public:
6     ShareStack(int length = InitSize)    //构造函数，参数可以有默认值
7     {
```

 复制代码

```

8     m_data = new T[length]; //为一维数组动态分配内存
9     m_maxsize = length;     //共享栈最多可以存储m_maxsize个数据元素
10    m_top1 = -1;             //顺序栈1的栈顶指针为-1, 表示空栈
11    m_top2 = length;         //顺序栈2的栈顶指针为length, 表示空栈
12 }
13 ~ShareStack()               //析构函数
14 {
15     delete[] m_data;
16 }
17
18 public:
19     bool IsFull()             //判断共享栈是否已满
20     {
21         if (m_top1 + 1 == m_top2)
22         {
23             return true;
24         }
25         return false;
26     }
27
28     bool Push(int stackNum, const T& e) //入栈 (增加数据), 参数stackNum用于标识栈1还是栈2
29     {
30         if (IsFull() == true)
31         {
32             //共享栈满了, 你也可以自行增加代码来支持动态增加共享栈的容量, 这里简单处理, 直接返回false
33             cout << "共享栈已满, 不能再进行入栈操作了!" << endl;
34             return false;
35         }
36         if (stackNum == 1)
37         {
38             //要入的是顺序栈1
39             m_top1++;           //栈顶指针向后走
40             m_data[m_top1] = e;
41         }
42         else
43         {
44             //要入的是顺序栈2
45             m_top2--;
46             m_data[m_top2] = e;
47         }
48         return true;
49     }
50
51     bool Pop(int stackNum, T& e) //出栈 (删除数据), 也就是删除栈顶数据
52     {
53         if (stackNum == 1)
54         {
55             //要从顺序栈1出栈
56             if (m_top1 == -1)

```

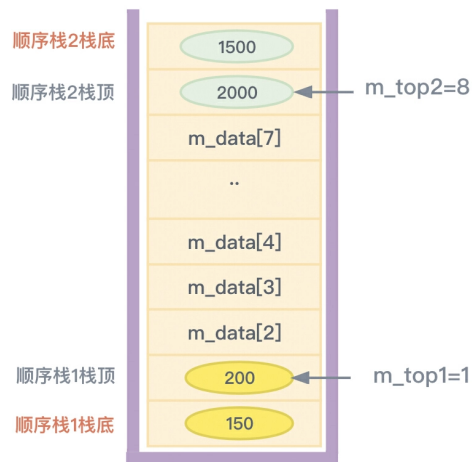
```

57     {
58         cout << "当前顺序栈1为空，不能进行出栈操作!" << endl;
59         return false;
60     }
61     e = m_data[m_top1]; //栈顶元素值返回到e中
62     m_top1--;
63 }
64 else
65 {
66     //要从顺序栈2出栈
67     if (m_top2 == m_maxsize)
68     {
69         cout << "当前顺序栈2为空，不能进行出栈操作!" << endl;
70         return false;
71     }
72     e = m_data[m_top2];
73     m_top2++;
74 }
75 return true;
76 }
77
78 private:
79     T*   m_data;                //存放共享栈中的元素
80     int  m_maxsize;            //动态数组最大容量
81     int  m_top1;               //顺序栈1的栈顶指针
82     int  m_top2;               //顺序栈2的栈顶指针
83 };

```

从代码中可以看到，既然是两个顺序栈共享同一块内存空间，那么就需要引入两个栈顶指针（m_top1、m_top2）来分别标识这两个顺序栈的栈顶位置。顺序栈 1 的栈底位置在最下面，而顺序栈 2 的栈底位置在最上面。

同时，注意阅读判断共享栈是否已满的代码（IsFull）以及入栈和出栈（Push、Pop）的代码。如果对顺序栈 1 进行入栈操作，则 m_top1 要递增，数据要从下向上存储。如果对顺序栈 2 进行入栈操作，则 m_top2 要递减，数据从上向下存储。这样的话，从逻辑上看，实现的是两个栈，但这两个栈又是共享着同一块物理内存的，从而提高内存利用率。把它转换成示意图，就会是图 4 的样子：



极客时间

图4 共享栈存储数据示意图

一般来讲，对两个顺序栈的存储空间需求正好相反时，将这两个顺序栈放在一起当作共享栈使用意义才比较大。这里的需求相反，指的就是当一个栈进行入栈操作时，另一个栈同时在进行出栈操作，也就是此消彼长的情况。所以，你也可以把共享栈看成是一种顺序栈的实现技巧。

可以在 main 主函数中增加下面的代码来测试共享栈。

复制代码

```
1 ShareStack<int> sharesobj(10);
2 sharesobj.Push(1, 150);
3 sharesobj.Push(2, 200);
4 int eval2;
5 sharesobj.Pop(1, eval2);
6 sharesobj.Pop(1, eval2);
```


栈的链式存储（链式栈 / 链栈）

所谓链式栈，就是链式存储方式来实现的栈。回忆前面讲过单链表的插入操作 ListInsert 方法，其第一个参数用于指定元素要插入的位置，如果把该参数值设置为 1，就是链式栈的入栈操作。对于单链表的删除操作 ListDelete 方法，其参数用于指定要删除的元素位置，如果把该参数值也设置为 1，就是链式栈的出栈操作。

可以看到，链式栈其实就是一个单链表，只不过人为的规定只能在单链表的第一个位置进行插入（入栈）和删除（出栈）操作，即链表头这一端是栈顶。

链式栈的实现代码和单链表的实现代码非常类似，可以把链式栈理解成受限的单链表。在讲解单链表时，可以带头结点也可以不带头节点，但对于链式栈来讲，考虑到只在链表头位置插入数据，所以链式栈一般不需要带头节点。

看一看链式栈的类定义以及常用操作。

 复制代码

```
1 //链式栈中每个节点的定义
2 template <typename T> //T代表数据元素的类型
3 struct StackNode
4 {
5     T data; //数据域，存放数据元素
6     StackNode<T>* next; //指针域，指向下一个同类型（和本节点类型相同）节点
7 };
8
9 //链式栈的定义
10 template <typename T>
11 class LinkStack
12 {
13 public:
14     LinkStack(); //构造函数
15     ~LinkStack(); //析构函数
16
17 public:
18     bool Push(const T& e); //入栈元素e
19     bool Pop(T& e); //出栈（删除数据），也就是删除栈顶数据
20     bool GetTop(T& e); //读取栈顶元素，但该元素并没有出栈而是依旧在栈中
21
22     void Displist(); //输出链式栈中的所有元素
23     int ListLength(); //获取链式栈的长度
24     bool Empty(); //判断链式栈是否为空
25
26 private:
27     StackNode<T>* m_top; //栈顶指针
28     int m_length; //链式栈当前长度
29 };
30
31 //通过构造函数对链式栈进行初始化
32 template <typename T>
33 LinkStack<T>::LinkStack()
34 {
35     m_top = nullptr;
36     m_length = 0;
37 }
```

```
1 //入栈元素e, 时间复杂度为O(1)
2 template <typename T>
3 bool LinkStack<T>::Push(const T& e)
4 {
5     StackNode<T>* node = new StackNode<T>;
6     node->data = e;
7     node->next = m_top;
8     m_top = node;
9     m_length++;
10    return true;
11 }
12
13 //出栈 (删除数据), 也就是删除栈顶数据, 时间复杂度为O(1)
14 template <typename T>
15 bool LinkStack<T>::Pop(T& e)
16 {
17     if (Empty() == true) //链式栈为空
18         return false;
19
20     StackNode<T>* p_willdel = m_top;
21     m_top = m_top->next;
22     m_length--;
23     e = p_willdel->data;
24     delete p_willdel;
25     return true;
26 }
27
28 //读取栈顶元素, 但该元素并没有出栈而是依旧在栈中
29 template <typename T>
30 bool LinkStack<T>::GetTop(T& e)
31 {
32     if (Empty() == true) //链式栈为空
33         return false;
34
35     e = m_top->data;
36     return true;
37 }
```

shikey.com转载分享

```
1 //输出链式栈中的所有元素, 时间复杂度为O(n)
2 template<class T>
3 void LinkStack<T>::DispList()
4 {
5     if (Empty() == true) //链式栈为空
6         return;
```

```

7
8     StackNode<T>* p = m_top;
9     while (p != nullptr)
10    {
11        cout << p->data << " "; //每个数据之间以空格分隔
12        p = p->next;
13    }
14    cout << endl; //换行
15 }
16
17 //获取链式栈的长度，时间复杂度为O(1)
18 template<class T>
19 int LinkStack<T>::ListLength()
20 {
21     return m_length;
22 }
23
24 //判断链式栈是否为空，时间复杂度为O(1)
25 template<class T>
26 bool LinkStack<T>::Empty()
27 {
28     if (m_top == nullptr) //链式栈为空
29     {
30         return true;
31     }
32     return false;
33 }
34
35 //通过析构函数对链式栈进行资源释放
36 template <typename T>
37 LinkStack<T>::~~LinkStack()
38 {
39     T tmpnousevalue = {0};
40     while (Pop(tmpnousevalue) == true) {} //把栈顶元素删光，while循环也就退出了，此时也就是
41 }

```

在 main 主函数中，可以加入如下代码进行测试：

 复制代码

```

1 LinkStack<int> slinkobj;
2 slinkobj.Push(12);
3 slinkobj.Push(24);
4 slinkobj.Push(48);
5 slinkobj.Push(100);
6 slinkobj.DispList();

```

```
7
8 int eval3 = 0;
9 slinkobj.Pop(eval3);
10 slinkobj.DispList();
```

与顺序栈相比，链式栈没有长度限制，不存在内存空间的浪费问题。但对于数据的入栈和出栈这些需要对数据进行定位的操作，顺序栈更加方便，而链式栈中的每个数据节点都需要额外的指针域以指向下一个数据节点，这会略微降低数据的存储效率，当然也会多占用一些内存。所以，如果要存储的数据数量无法提前预估，一般考虑使用链式栈，而如果数据的数量比较固定，可以考虑使用顺序栈。

在前面的讲解中，我们实现了顺序表（以数组的方式），又实现了各种各样的链表。在这节课，我们也通过数组的方式实现了顺序栈，通过单链表实现了链式栈。所以，把顺序栈看成是**功能受限的数组**，或把链式栈看成是功能受限的单链表，都是没有问题的。

可能你会认为，既然已经存在数组和单链表，直接使用不是更方便，为什么又创造出功能受限的栈来呢？你可以理解为，因为功能受限，所以使用起来也更加简单，错用误用的概率比数组、单链表等更低。

栈有很多应用，比如很多人都知道的在函数调用期间需要用栈来保存临时的参数信息、函数内局部变量信息、函数调用返回地址信息等。网上也有很多小例子演示栈的简单应用，比如利用栈来进行括号匹配的检验，利用栈来计算表达式结果等。有兴趣的话，你可以通过搜索引擎自行搜索了解。在后续的课程中，我也会用栈来实现诸如树的非递归遍历、记录节点路径信息等操作，相信那时你会对栈的应用有更深刻的理解。

小结

这节课，我们讲解了栈这种常用的数据结构，分别用代码实现了顺序栈（包括共享栈）和链栈，同时针对栈的应用举了几个小例子。

栈的应用场合其实很多，上面也只是列举了几个范例进行简单演示，更多的案例需要你在实践中学习和体会。在实际应用中，有时为达到简化计算步骤的目的，还可以根据算法的需要引入两个甚至更多的栈来参与计算，每个栈都有各自的目的和分工，这都很平常，也不必觉得奇

怪。后面的课程中，我们也会经常用到栈这种数据结构，届时你对栈的应用也会有更深层次的理解。

通过今天这一讲，你会发现，“在某一个事物的进行中需要保存一些数据，而将来又会以相反的顺序取回”这样的场合里，总会有“栈”的身影。

在 STL（标准模板库）中，提供了一个名字叫做 stack 的容器，该容器其实是一个类模板，这个类模板实现的就是栈的功能，所以通常情况下不需要你自己写栈的实现代码，用这个现成的即可。而这节课的内容，就可以让你对栈的理解更深入，从而更好地使用 stack 容器。

课后思考

在这节课的最后，我也给你留了几道思考题，可以想一想。

1. 回忆一下你曾经在哪些场合使用过栈或遇到过对栈的应用，试着列举出一二。
2. 如果你有兴趣可以对 stack 容器的源码做适当的研究，当然，其源码比较繁琐和复杂，需要你有比较好的模板与泛型编程基础才能读懂。
3. 这节课讲的**栈**是一种数据结构。但从操作系统层面，也有栈这个概念，只不过操作系统层面所指的栈和我们这里所讲的栈不是同一个意思。有兴趣了解的话，可以在搜索引擎中输入“堆和栈的区别”这样的关键词来了解操作系统层面的栈指的是什么意思。

欢迎你在留言区和我互动。如果觉得有所收获，也可以把课程分享给更多的同学一起交流学习，我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

shikey.com转载分享

精选留言 (2)



徐曙辉

2023-03-02 来自湖南

1. 操作系统调用栈，浏览器前进后退功能
3. 内存中的堆栈和数据结构堆栈不是一个概念，内存中的堆栈是真实存在的物理区，数据结

构中的堆栈是抽象的数据存储结构。

操作系统给每个线程分配了一块独立的内存空间，这块内存被组织成“栈”这种结构，用来存储函数调用时的临时变量。每进入一个函数，就会将临时变量作为一个栈帧入栈，当被调用函数执行完成，返回之后，将这个函数对应的栈帧出栈，符合先进后出的特性。

从调用函数进入被调用函数，对于数据来说，变化的是作用域。所以只要能保证每进入一个新的函数，都要是一个新的作用域。而要实现这个，用栈就非常方便。在进入被调用函数的时候，分配一段栈空间给这个函数的变量，在函数结束的时候，将栈顶复位，正好回到调用函数的作用域内。

用Go实现的数组栈和链表栈 <https://github.com/xushuhui/algorithm-and-data-structure/tree/master/datastructure/stack>

作者回复: 🤔🤔😄😄



👍 1



编程小白小吴

2023-03-17 来自中国台湾

- 1、操作系统中的调用栈：在操作系统中，每个进程都有一个独立的调用栈，用于存储函数调用的返回地址和参数。当程序执行一个函数时，它的参数和返回地址被压入该进程的调用栈中。当函数返回时，这些参数和返回地址被弹出栈。
- 2、网络协议中的数据包处理：在网络协议中，数据包处理通常采用栈的方式进行，以确保数据包能够正确地路由。数据包处理时，每个网络层都会将自己的头部信息入栈，并在发送时逆序弹出栈，以保证数据包的正确传输。
- 3、数据库系统中的事务处理：在数据库系统中，事务处理可以采用栈来实现。每次进行事务操作时，会将操作入栈，以便回滚时能够按照相反的顺序执行操作。（事务的实现可能会有所不同。除了使用栈来管理事务操作序列外，还可以使用其他数据结构，如链表、数组等来实现事务。此外，不同的数据库系统还可能会采用不同的事务模型和隔离级别，以满足不同的应用需求。）
- 4、操作系统中的中断处理：在操作系统中，中断处理也可以采用栈来实现。当系统发生中断时，将当前程序的状态入栈，然后转到中断处理程序。当中断处理程序执行完毕后，将保存的程序状态弹出栈，恢复原来的程序执行。
- 6、撤销操作：在编辑器中，撤销操作通常也可以使用栈来实现，将每次编辑的操作记录在栈中，撤销时将最近的操作出栈并撤销。

作者回复: 🤔🤔🤔🤔



shikey.com转载分享