

第六讲

Vector, Grid, Stack, Queue

薛浩

2023 年 4 月 6 日

www.stickmind.com

- 话题 1：编程基础** 初学编程的新手，一般应该熟练使用函数和库处理字符串相关的编程任务。
- 话题 2：抽象数据类型的使用** 在尝试实现抽象数据类型之前，应该先熟练使用这些工具解决问题。
- 话题 3：递归和算法分析** 递归是一种强有力的思想，一旦掌握就可以解决很多看起来非常难的问题。
- 话题 4：类和内存管理** 使用 C++ 实现数据抽象之前，应先学习 C++ 的内存机制。
- 话题 5：常见数据结构和算法** 在熟练使用抽象数据类型解决常见问题之后，学习如何实现它们是一件很自然的事情。

话题 2: 抽象数据类型的使用

在尝试实现抽象数据类型之前，应该先熟练使用这些工具解决问题。

- Vecotr、Grid、Stack、Queue
- Map、Set、Lexicon

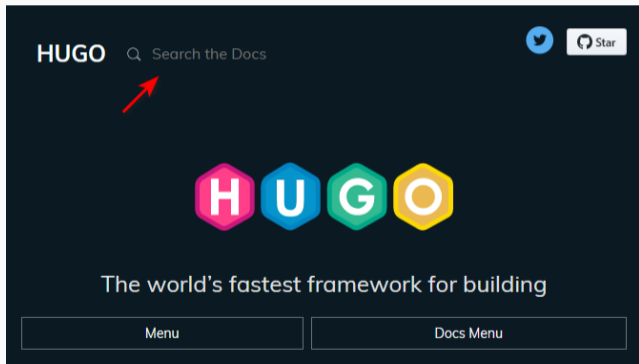


Figure 1: 搜索引擎

如何使用计算机对实际问题建模？

1. 复习：字符串
2. 容器类概述
3. Vector 类
4. Grid 类
5. Stack 类
6. Queue 类

复习：字符串

复习：字符串

在 C++ 中，字符串是一个抽象数据类型，支持更丰富的操作接口。在函数传递过程中，如果不使用引用参数，函数将会创建一份字符串拷贝。

对 string 进行遍历，使用传统的 for 循环结构较为复杂，还有可能会发生越界问题。新标准提供了更现代的基于范围的 for 循环：

```
for(const char & ch: str) {  
    // ...  
}
```

新的语法虽然更简洁，但却无法逆向或跳跃遍历，在具体编程场景中需要根据情况进行选择。

容器类概述

斯坦福库提供了很多现成的抽象数据类型，这些类可以用于存储其他对象，所以称作**容器** (container)。



对于容器的使用，有以下一些通用的准则需要注意：

- 这些类表示**抽象数据类型** (abstract data types)，其细节是隐藏的
- 除了 Lexicon 外，每个类的使用都需要类型参数
- 声明这些类型的变量时，总会调用**构造器**
- 一旦变量离开声明的范围，这些对象占用的内存将被释放
- 将一个变量赋值给另一个变量时，将会得到完整的对象结构拷贝
- 为了避免拷贝，可以使用引用传递

容器类的实现用到了**模板类** (Template Class) 技术, 和模板函数类似, 该技术让复用整个类的代码称为可能。

与普通类型声明变量不同的是, 容器类需要提供一个类型参数才能确定容器中元素的类型。例如, `Vector<int>` 表示一个存储整型的容器; `Grid<char>` 表示一个存储二维字符数组的容器。

嵌套类型也是允许的, 例如, 可以像下面这样声明一个表示数独的容器:

```
Vector<Vector<int>> sudoku;
```

Vector 类

Vector<type> 类的构造器

```
Vector<type> vec;
```

Initializes an empty vector of the specified element type.

```
Vector<type> vec(n);
```

Initializes a vector with `n` elements all set to the default value of the type.

```
Vector<type> vec(n, value);
```

Initializes a vector with `n` elements all set to `value`.

```
Vector<type> vec{value1, value2, ...};
```

Initializes a vector with the given `value list`.

Vector<type> 类常用接口

vec.size() Returns the number of elements in the vector.
vec.isEmpty() Returns true if the vector is empty.
vec.get(i) <i>or</i> vec[i] Returns the i^{th} element of the vector.
vec.set(i, value) <i>or</i> vec[i] = value; Sets the i^{th} element of the vector to value .
vec.add(value) <i>or</i> vec += value; Adds a new element to the end of the vector.
vec.insert(index, value) Inserts the value before the specified index position.
vec.remove(index) Removes the element at the specified index.
vec.clear() Removes all elements from the vector.

readEntireFile

```
/*
 * Function: readEntireFile
 * Usage: readEntireFile(is, lines);
 * -----
 * Reads the entire contents of the specified input stream
 * into the string vector lines.
 */

void readEntireFile(istream & is, Vector<string> & lines) {
    lines.clear();
    string line;
    while (getline(is, line)) { lines.add(line); }
}
```

Grid 类

Grid<type> 类常用接口

Grid<type> grid(nrows, ncols);

Constructs a grid with the specified dimensions.

grid.numRows()

Returns the number of rows in the grid.

grid.numCols()

Returns the number of columns in the grid.

grid[i][j]

Selects the element in the i^{th} row and j^{th} column.

resize(nrows, ncols)

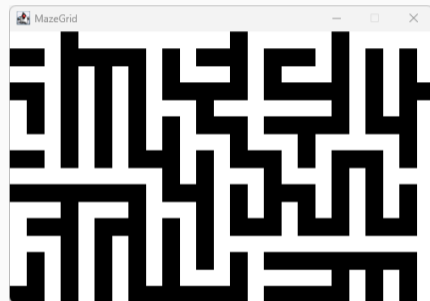
Changes the dimensions of the grid and clears any previous contents.

inBounds(row, col)

Returns `true` if the specified row and column position is within the grid.

练习: MazeGrid

```
1  ---@-----@-----@-----
2  @@-@@@@@-@-@@@-@@@-@-@-@-@-@-
3  ---@-@-@-@-@-@-@-@-@-@-@-@-
4  @@-@-@-@-@@@@@-@@@@@-@-@@@
5  -@-@-@-@-@-@-----@-@-@-@-
6  -@-@-@-@-@-@@@-@@@@@-@@@-@-
7  ---@-@-@-@-@-@-----@-----
8  @@-@-@-@@@-@-@@@-@-@@@-@-@-
9  -----@-@-----@-@-----
10 @@@@@@@@@-@@@-@-@@@-@-@-@-@-
11 ---@---@---@-@-@-@-@-@-@-@-
12 -@@@-@@@-@-@-@@@-@@@-@@@-@-
13 ---@-@-@-@-@-@-----@-----
14 -@-@-@-@-@-@-@-@@@@@@@@@@@-
15 -@-@-@-@-@-@-----@-@-@-@-
16 @@-@-@-@@@@@@@@@-@@@@@-@-@-
17 |
```



Stack 类

Stack 容器在处理元素方式上采用**后进先出** (last-in-first-out) 的原则。基本的操作有入栈 (push) 添加元素到栈顶和出栈 (pop) 从栈顶删除元素。

生活中最常见的场景是餐盘的存储和使用。放在上面的盘子，总是比底部较早放进去的盘子，优先被使用。



Stack<type> 类常用接口

stack.size ()

Returns the number of values pushed onto the stack.

stack.isEmpty ()

Returns `true` if the stack is empty.

stack.push (value)

Pushes a new value onto the stack.

stack.pop ()

Removes and returns the top value from the stack.

stack.peek ()

Returns the top value from the stack without removing it.

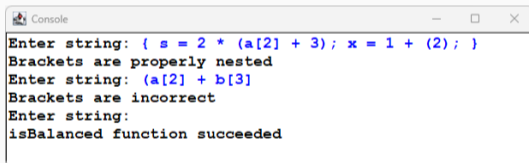
stack.clear ()

Removes all values from the stack.

练习: isBalanced

括号匹配问题在开发环境中很常见。我们可以利用栈的行为，检测一段代码中的括号是否平衡。

```
{ s = 2 * (a[2] + 3); x = (1 + (2)); }
```



```
Console
Enter string: { s = 2 * (a[2] + 3); x = 1 + (2); }
Brackets are properly nested
Enter string: (a[2] + b[3]
Brackets are incorrect
Enter string:
isBalanced function succeeded
```

Queue 类

Queue<type> 类常用接口

queue.size ()

Returns the number of values in the queue.

queue.isEmpty ()

Returns **true** if the queue is empty.

queue.enqueue (value)

Adds a new value to the end of the queue (which is called its *tail*).

queue.dequeue ()

Removes and returns the value at the front of the queue (which is called its *head*).

queue.peek ()

Returns the value at the head of the queue without removing it.

queue.clear ()

Removes all values from the queue.

练习: Mystery

以下代码输出是什么:

```
Queue<int> queue{1, 2, 3, 4, 5, 6};

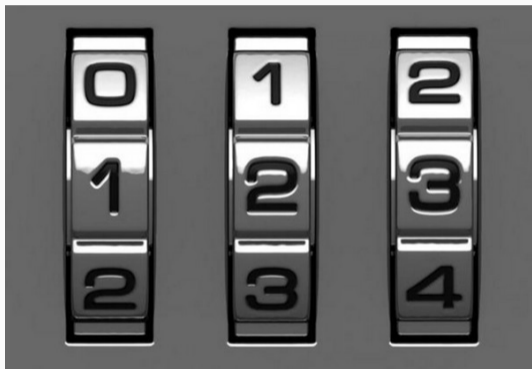
for (int i = 0; i < queue.size(); i++) {
    cout << queue.dequeue() << " ";
}

cout << queue.toString() << " size " << queue.size() << endl;
```

选择:

- A. 1 2 3 4 5 6 {} size 0
- B. 1 2 3 {4, 5, 6} size 3
- C. 1 2 3 4 5 6 {1, 2, 3, 4, 5, 6} size 6

练习: Unlock



如何使用计算机对实际问题建模？

问题?