C++ 数据结构

C++ 提供了多种数据结构，既有基础的如数组、结构体、类等，也有高级的 STL 容器如 `vector`、`map` 和 `unordered_map` 等。下面详细介绍 C++ 中常用的数据结构及其特点和用法。

1. 数组（Array）

数组是最基础的数据结构，用于存储一组相同类型的数据。

特点：

固定大小，一旦声明，大小不能改变。

直接访问元素，时间复杂度为 O(1)。

适合处理大小已知、元素类型相同的集合。

实例

```
int arr[5] = {1, 2, 3, 4, 5};
cout << arr[0]; // 输出第一个元素
```

优缺点：

优点：访问速度快，内存紧凑。

缺点：大小固定，无法动态扩展，不适合处理大小不确定的数据集。

2. 结构体（Struct）

结构体允许将不同类型的数据组合在一起，形成一种自定义的数据类型。

特点：

可以包含不同类型的成员变量。

提供了对数据的基本封装，但功能有限。

示例：

实例

```
struct Person {
    string name;
    int age;
};
Person p = {"Alice", 25};
cout << p.name << endl; // 输出 Alice
```

3. 类（Class）

类是 C++ 中用于面向对象编程的核心结构，允许定义成员变量和成员函数。与 `struct` 类似，但功能更强大，支持继承、封装、多态等特性。

特点：

可以包含成员变量、成员函数、构造函数、析构函数。

支持面向对象特性，如封装、继承、多态。

实例

```
class Person {
private:
    string name;
    int age;
public:
    Person(string n, int a) : name(n), age(a) {}
    void printInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
Person p("Bob", 30);
p.printInfo(); // 输出：Name: Bob, Age: 30
```

4. 链表（Linked List）

链表是一种动态数据结构，由一系列节点组成，每个节点包含数据和指向下一个节点的指针。

特点：

动态调整大小，不需要提前定义容量。

插入和删除操作效率高，时间复杂度为 O(1)（在链表头部或尾部操作）。

线性查找，时间复杂度为 O(n)。

实例（单向链表）

```
struct Node {
    int data;
    Node* next;
};
Node* head = nullptr;
Node* newNode = new Node{10, nullptr};
head = newNode; // 插入新节点
```

优缺点：

优点：动态大小，适合频繁插入和删除的场景。

缺点：随机访问效率低，不如数组直接访问快。

5. 栈（Stack）

栈是一种后进先出（LIFO, Last In First Out）的数据结构，常用于递归、深度优先搜索等场景。

特点：

只允许在栈顶进行插入和删除操作。

时间复杂度为 O(1)。

实例

```
stack<int> s;
s.push(1);
s.push(2);
cout << s.top(); // 输出 2
s.pop();
```

优缺点：

优点：操作简单，效率高。

缺点：只能在栈顶操作，访问其他元素需要弹出栈顶元素。

6. 队列（Queue）

队列是一种先进先出（FIFO, First In First Out）的数据结构，常用于广度优先搜索、任务调度等场景。

特点：

插入操作在队尾进行，删除操作在队头进行。

时间复杂度为 O(1)。

实例

```
queue<int> q;
q.push(1);
q.push(2);
cout << q.front(); // 输出 1
q.pop();
```

优缺点：

优点：适合按顺序处理数据的场景，如任务调度。

缺点：无法随机访问元素。

7. 双端队列（Deque）

双端队列允许在两端进行插入和删除操作，是栈和队列的结合体。

特点：

允许在两端进行插入和删除。

时间复杂度为 O(1)。

实例

```
deque<int> dq;
dq.push_back(1);
dq.push_front(2);
cout << dq.front(); // 输出 2
dq.pop_front();
```

优缺点：

优点：灵活的双向操作。

缺点：空间占用较大，适合需要在两端频繁操作的场景。

8. 哈希表（Hash Table）

哈希表是一种通过键值对存储数据的数据结构，支持快速查找、插入和删除操作。C++ 中的 `unordered_map` 是哈希表的实现。

特点：

使用哈希函数快速定位元素，时间复杂度为 O(1)。

不保证元素的顺序。

实例

```cpp
unordered_map<string, int> hashTable;
hashTable["apple"] = 10;
cout << hashTable["apple"]; // 输出 10
```

优缺点:

优点：查找、插入、删除操作效率高。

缺点：无法保证元素顺序，哈希冲突时性能会下降。

9. 映射（Map）

`map` 是一种有序的键值对容器，底层实现是红黑树。与 `unordered_map` 不同，它保证键的顺序，查找、插入和删除的时间复杂度为 O(log n)。

特点:

保证元素按键的顺序排列。

使用二叉搜索树实现。

实例

```cpp
map<string, int> myMap;
myMap["apple"] = 10;
cout << myMap["apple"]; // 输出 10
```

优缺点:

优点：元素有序，适合需要按顺序处理数据的场景。

缺点：操作效率比 `unordered_map` 略低。

10. 集合（Set）

`set` 是一种用于存储唯一元素的有序集合，底层同样使用红黑树实现。它保证元素不重复且有序。

特点:

保证元素的唯一性。

元素自动按升序排列。

时间复杂度为 O(log n)。

实例

```cpp
set<int> s;
s.insert(1);
s.insert(2);
cout << *s.begin(); // 输出 1
```

优缺点:

优点：自动排序和唯一性保证。

缺点：插入和删除的效率不如无序集合。

11. 动态数组（Vector）

`vector` 是 C++ 标准库提供的动态数组实现，可以动态扩展容量，支持随机访问。

特点:

动态调整大小。

支持随机访问，时间复杂度为 O(1)。

当容量不足时，动态扩展，时间复杂度为摊销 O(1)。

实例

```
vector<int> v;
v.push_back(1);
v.push_back(2);
cout << v[0]; // 输出 1
```

优缺点:

优点：支持随机访问，动态扩展。

缺点：插入和删除中间元素的效率较低。