链表和数组的区别

在C++中，数组和链表是两种最基础的线性数据结构，均用于存储多个相同类型的元素，但二者在内存布局、操作效率、适用场景等方面存在本质差异。数组是C++内置的复合类型，而链表多通过结构体/类自定义实现（STL中提供`std::list`双向链表、`std::forward_list`单向链表），掌握二者区别是合理选择数据结构、优化代码性能的关键，下文将从核心维度详细剖析。

一、核心定义回顾

1. 数组（Array）

由相同类型元素组成的线性集合，元素在内存中连续排列，通过下标可直接访问对应元素。C++中分为静态数组（声明时固定大小）和动态数组（通过`new`分配内存），STL中的`std::vector`是动态数组的封装，兼顾灵活性与访问效率。

```
// 静态数组
int staticArr[5] = {1, 2, 3, 4, 5};
// 动态数组
int* dynamicArr = new int[5];
// STL动态数组（vector）
#include <vector>
std::vector<int> vec = {1, 2, 3};

```

2. 链表（Linked List）

由多个节点串联形成的线性集合，每个节点包含两部分：数据域（存储元素值）和指针域（存储相邻节点的内存地址）。根据指针域数量，分为单向链表（仅存下一个节点地址）、双向链表（存前后两个节点地址）和循环链表（首尾节点相连），核心优势是动态调整大小。

```
// 自定义单向链表节点
struct ListNode {
    int data; // 数据域
    ListNode* next; // 指针域，指向后一个节点
    // 构造函数
    ListNode(int val) : data(val), next(nullptr) {}
};
// 创建链表头节点
ListNode* head = new ListNode(1);
```

```
10
```

二、核心区别详解

1. 内存布局（最本质区别）

- **数组**：元素在内存中**连续分配**，所有元素占用一块连续的内存空间，无需额外存储关联信息。例如int类型数组，每个元素占用4字节，5个元素共占用20字节，内存地址连续递增。这种布局的优势是内存利用率高（无冗余空间），但依赖连续内存块，若内存中无足够大的连续空间，无法创建数组。
- **链表**：节点在内存中**分散分配**，每个节点独立占用一块内存，通过指针域关联形成线性结构。节点的内存地址无需连续，系统可在任意空闲内存区域分配节点。这种布局的劣势是需额外占用内存存储指针（如单向链表每个节点多占用4/8字节指针空间），内存利用率低于数组。

2. 访问效率

- **数组**：支持**随机访问**，通过下标`arr[i]`可直接计算出元素的内存地址（地址=数组起始地址+i×元素大小），访问操作的时间复杂度为O(1)，无论元素数量多少，访问速度均保持稳定，这是数组的核心优势。
- **链表**：不支持随机访问，仅支持顺序访问。若要访问第i个节点，必须从表头（或表尾）开始，依次遍历前i-1个节点，才能定位到目标节点，访问操作的时间复杂度为O(n)，节点数量越多，访问耗时越长。

3. 增删操作效率

增删操作的效率取决于是否需要移动元素，二者差异显著，且需区分“已找到目标位置”和“未找到目标位置”两种场景：

- **数组**：增删元素时，需移动目标位置前后的元素，避免内存空隙或元素覆盖，时间复杂度为O(n)。例如在数组中间插入元素，需将插入位置后的所有元素向后移动一位；删除元素则需将后续元素向前移动一位，操作繁琐且效率低。即便在数组尾部增删（静态数组尾部也无法直接增删），动态数组扩容时仍需拷贝原有元素，效率受影响。
- **链表**：若已找到目标节点，增删操作仅需修改目标节点及相邻节点的指针指向，无需移动其他节点，时间复杂度为O(1)。例如单向链表插入节点，只需让前驱节点的指针指向新节点，新节点的指针指向后继节点即可；删除节点同理，修改前驱节点指针跳过目标节点即可。但如果未找到目标节点，需先遍历定位，整体时间复杂度仍为O(n)。

4. 大小灵活性

- **数组**：静态数组大小固定，声明时必须确定长度，后续无法修改；动态数组（`new`分配或`vector`）虽可扩容，但扩容时需重新申请一块更大的连续内存，将原有元素拷贝至新内存，再释放旧内存，不仅操作繁琐，还会产生额外的时间开销，整体灵活性较差。

- **链表**：大小可动态调整，增删节点时直接分配/释放对应节点的内存，无需提前规划总大小，也无需拷贝元素，灵活性极高，可适配元素数量频繁变化的场景。

5. 代码实现复杂度

- **数组**：语法简单，直接声明即可使用，C++内置支持下标访问、赋值等操作，无需额外实现复杂逻辑，代码实现难度低，适合新手使用。STL的`vector`进一步封装了扩容、增删等操作，使用更便捷。
- **链表**：需自定义节点结构（或使用STL封装类），手动实现遍历、增删、节点释放等操作，尤其双向链表、循环链表的指针操作逻辑更繁琐，还需注意避免内存泄漏（手动分配的节点需手动释放），代码实现复杂度高于数组。STL的`std::list`已封装好所有常用操作，可简化开发。

6. 内存开销

- **数组**：仅占用存储元素所需的内存，无额外内存开销（动态数组的扩容预留空间除外），内存利用率高。
- **链表**：每个节点需额外存储指针（单向链表1个指针，双向链表2个指针），指针占用的内存空间会随节点数量增加而增多，且频繁增删节点可能产生内存碎片，整体内存开销高于数组。

三、适用场景对比

1. 优先使用数组（或`vector`）的场景

- 元素数量固定，或变化频率极低；
- 需要频繁对元素进行随机访问（如查询指定索引的元素）；
- 对内存利用率要求高，需减少额外内存开销；
- 示例场景：存储班级学生成绩、固定长度的配置参数、数组排序与查找等。

2. 优先使用链表（或`std::list`）的场景

- 元素数量不固定，频繁进行增删操作（尤其是中间位置的增删）；
- 无需随机访问元素，仅需顺序遍历；
- 内存中无足够大的连续空间，无法使用数组；
- 示例场景：消息队列、任务调度链表、浏览器历史记录（频繁插入/删除）、LRU缓存淘汰机制等。

四、总结

数组和链表的核心差异源于内存布局的不同：数组以“连续内存”换取高效的随机访问和高内存利用率，适合频繁访问、少增删的场景；链表以“分散内存+额外指针”换取灵活的动态调整和高效的增删操作，适合频繁增删、少访问的场景。

在实际C++开发中，很少直接使用原生数组和自定义链表，更多采用STL封装的`vector`（动态数组）和`std::list`（双向链表），二者兼顾了易用性和性能，可根据具体业务需求灵活选择。若需兼顾随机访问和动态增删，`vector`通常是更优选择（尾部增删效率接近O(1)）；若存在大量中间位置增删，`std::list`更合适。