C++ 链表（一）：讲义

一、 引言

1.1 问题引入

我们来回顾一下最熟悉的数据结构——数组。如果想记录一排学生的队伍，用数组 `string names[50];` 来存，非常方便。数组的特点是什么？—— 内存中，所有元素都是"肩并着肩"挨在一起的，位置**连续且固定**。

这带来了好处：可以通过下标 `names[i]` 瞬间访问到任何一个学生，这叫**随机访问**。

但也带来了麻烦：

- **插入**：如果一个新同学想插到队伍中间，那么他之后的所有同学都必须**依次往后挪一个位置**，工程浩大。
- **删除**：如果一个同学离队，他之后的所有同学又得**依次往前填补空位**。

对于一个需要频繁增删成员的动态队伍来说，数组这种结构，显然不够灵活。

1.2 价值阐述

有没有一种数据结构，能像一个灵活的队伍，轻松地让成员加入或离开，而不需要大规模地挪动呢？这就是今天要学习的——**链表**。

链表是计算机科学中最基础也最重要的数据结构之一。它在内存中的存储方式，解决了数组在插入和删除操作上的低效问题。掌握链表，将能够：

- 理解一种全新的、非连续的数据组织方式。
- 高效地处理动态变化的数据集合。
- 为学习更高级的数据结构（如哈希表、树、图等）打下理论基础。

二、 核心思想

2.1 思路铺垫

想象一下，排好队的学生解散后，在大厅里随意站着，他们在大厅中的位置是**不连续**的。我们还能还原出原来的队伍顺序吗？

如果只知道他们的名字，显然不能。但如果在解散前，我们让每个同学都记住一件事——**我后面站的是谁**，情况就完全不同了。

只要我们还知道**谁是第一个人（起点）**，我们就能：

1. 找到第一个人。
2. 问他："你后面是谁？" -> 从而找到第二个人。
3. 再问第二个人："你后面是谁？" -> 从而找到第三个人。

   ...
   就这样，像一条线索，一个接一个，把整个队伍重新"串"起来。

这种数据在内存中可以分散存储，仅依靠一个"指向下一个元素"的信息来维持逻辑顺序的数据结构，就是**链表**。

2.2 链表的组成

一个链表由若干个**节点（Node）**串联而成。每个节点，就像队伍里的一个同学，包含两个核心信息：

1. **数据域 (Data)**：节点本身存储的数据。例如，同学的编号、姓名等。
2. **指针域 (Next)**：一个“指针”或“线索信息”，用于存储**下一个节点的位置信息。**

此外，我们还需要一个单独的变量 `head`，来记录整个链表的**头节点（起点）**。

2.3 步骤拆解：用 `Next` 数组模拟链表

在程序中，如何表示“某人后面是谁”这个信息呢？一种非常直观的方式是使用一个`Next`**数组**，这种方法也被称为**静态链表。**

- **规则**：`Next[i] = j`，其含义是：编号为 `i` 的节点，它的下一个节点是编号为 `j` 的节点。
- **结束标记**：队伍的最后一个人后面没人了。我们需要一个特殊值来表示“链表的末尾”，通常使用 `0` 或 `-1`（因为节点编号通常从1开始）。

案例【例15-10】：
有4名同学，原始队列顺序是 `1 -> 4 -> 3 -> 2`。已知1号是头节点。
根据“我后面是谁”的规则，我们可以构建出 `Next` 数组：

- 1号后面是4号 => `Next[1] = 4;`
- 4号后面是3号 => `Next[4] = 3;`
- 3号后面是2号 => `Next[3] = 2;`
- 2号后面没人了 => `Next[2] = 0;` (使用0作为结束标记)

这样，我们就用一个简单的数组，成功地在计算机中描述了这个手牵手的链式关系。

三、 实例：遍历链表

遍历是指从头节点开始，沿着“Next”指针，依次访问链表中的每一个节点，直到末尾。

3.1 例15-10 还原排队顺序

- **已知信息**：
  - 头节点 `head = 1`。
  - `Next` 数组：`Next[1]=4`，`Next[4]=3`，`Next[3]=2`，`Next[2]=0`。

3.2 分步演示

1. **【起点】**：创建一个“当前位置”变量 `current`，让它从头节点出发。`current = head;` (即 `current = 1`)。
2. **【第1步】**：
   - **判断**：`current` 不是终点 (`1 != 0`)。
   - **访问**：输出当前位置的编号 `1`。
   - **前进**：查询 `Next[1]`，得到 `4`。更新 `current` 的值为 `4`。
3. **【第2步】**：查询 `Next[4]`，得到 `3`。更新 `current` 的值为 `3`。(其他略)
4. **【第3步】**：查询 `Next[3]`，得到 `2`。更新 `current` 的值为 `2`。(其他略)

5. **【第4步】**：
    - **判断**：`current` 不是终点（`2 != 0`）。
    - **访问**：输出当前位置的编号 `2`。
    - **前进**：查询 `Next[2]`，得到 `0`。更新 `current` 的值为 `0`。
6. **【终点】**：
    - **判断**：`current` 是终点（`0 == 0`）。循环条件不满足，**循环结束**。

最终，我们按 `1 4 3 2` 的正确顺序输出了整个队伍。

四、 链表的遍历代码实现

4.1 `while` 循环实现

`while` 循环非常直观地体现了遍历的逻辑。

```
// 假设Next数组已构建好，head = 1
int current = head;          // 1. 从头节点出发
while (current != 0) {       // 2. 只要还没到终点
    cout << current << " "; // 3. 访问当前节点
    current = Next[current];  // 4. 移动到下一个节点
}
```

4.2 `for` 循环的实现

我们也可以用 `for` 循环来表达同样的过程，这种写法更紧凑，并将控制逻辑集中在一起。

```
// for (初始化; 循环条件; 循环后操作)
for (int current = head; current != 0; current = Next[current]) {
    cout << current << " ";
}
```

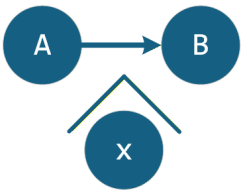
- **初始化**：`int current = head`（从起点开始）
- **循环条件**：`current != 0`（何时停止）
- **循环后操作**：`current = Next[current]`（如何去下一个节点）

这两种写法本质上完全一样，都实现了链表的遍历。

五、 核心操作一：插入

5.1 先连接新节点的后继，再修改前驱的后继

实例场景：
假设我们有一条 `A -> B` 的链路，现在新节点 `X` 想要插入在 `A` 和 `B` 之间，形成 `A -> X -> B` 的新逻辑顺序。



这个过程需要修改节点的后继关系。正确的步骤分为两步，且**顺序很重要**：

1. **第一步：设置新节点X的后继**
   - **动作**：将新节点 `X` 的后继指针，指向节点 `A` 原本的后继 `B`。
   - **原因**：节点 `A` 的后继信息 `Next[A]` 是我们找到节点 `B` 的唯一信息。如果先修改 `Next[A]`，`B` 的位置信息就会丢失，导致链表后半部分失联，所以要先设置新节点X的后继。
   - **代码实现**：

     ```
     Next[X] = Next[A];
     ```

2. **第二步：修改前驱节点A的后继**
   - **动作**：现在 `X` 已经连接到了链表的后半部分，可以放心地修改 `A` 的后继指针，使其指向新插入的节点 `X`。
   - **代码实现**：

     ```
     Next[A] = X;
     ```

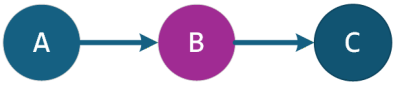
总结：`Next[X] = Next[A];` 和 `Next[A] = X;` 这两行代码，必须严格按照此顺序执行，以确保链表的完整性。

因为如果先把节点A的后继更新B。那么节点A原本的后继是B的这一关键信息就彻底丢失了。接下来需要设置新节点 X 的后继为 B，此时 B 节点就无法再找到了，因为如果节点 B 去查询节点 A 的后继，得到的结果变成了 B 自己，所以会导致无法执行 `Next[X] = Next[A];` 这一步。

六、 删除

6.1 核心思想：修改前驱节点的后继，直接指向被删节点的后继

实例：
假设我们有一条 `A -> B -> C` 的链路，现在需要从删除节点 `B`，形成 `A -> C` 的新顺序。



这个过程只需要一步关键操作：

1. **修改A的后继**
   - **动作**：将节点 `A` 的后继指针，直接指向 `B` 的后继节点 `C`。
   - **分析**：节点 `A` 如何知道 `C` 的位置？通过 `B` 的后继信息 `Next[B]` 即可获得。
   - **代码实现**：

     ```
     Next[A] = Next[B];
     ```

- 执行完这行代码后，节点 `A` 绕过了 `B` 直接与 `C` 连接，节点 `B` 从逻辑上被成功删除，因为再也没有可能访问到节点 `B` 了。

6.2 如何找到前驱节点?

上述删除操作看似简单，但它有一个前提：我们必须知道待删除节点 `B` 的**前驱节点**是 `A`。

在单向链表中，每个节点只存储了其后继节点的信息，并不知道其前驱是谁。这意味着，如果只给定删除节点 B，我们无法直接执行 `Next[A] = Next[B]`，因为我们不知道 `A` 在什么地方。

为了找到 `A`，我们只能从链表的**头节点** `head` 开始，沿着后继指针逐个遍历，直到找到一个节点 `p`，满足 `Next[p] == B`。这个节点 `p` 就是我们需要的 `A`。这个查找过程，在最坏情况下需要遍历整个链表，时间复杂度为 O(N)。

这使得单向链表的删除操作，在其核心逻辑之外，增加了一个效率不高的查找步骤。

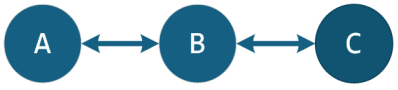
七、 双向链表

为了解决单向链表寻找前驱节点难的问题，我们对链表结构进行了一次升级。

7.1 记录前驱与后继的双向连接

定义：双向链表的每个节点，不再只有一个后继指针，而是拥有**两个指针**：

1. `next` **指针**：指向下一个节点。
2. `prev` **指针**：指向**前一个**节点。



通过这种方式，每个节点都同时掌握了其前驱和后继的信息，数据在逻辑上形成了双向的链接。

7.2 双向链表的优势

- **高效查找前驱**：对于任意一个节点 `B`，找到其前驱 `A` 不再需要遍历，只需一步 `A = B->prev` 即可，时间复杂度为 **O(1)**。
- **实现了双向遍历**：我们既可以从头节点 `head` 开始，通过 `next` 指针正向遍历链表；因为是双向的，我们自然还会定义并储存一个尾节点 `tail`，那么就可以通过尾节点 `tail` 的 `prev` 指针反向遍历。
- **高效的删除**：由于可以瞬间找到前驱和后继，双向链表的删除操作也变得极其高效。

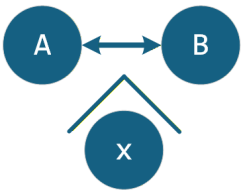
双向链表删除节点B的逻辑：

1. 让B的前驱A，跳过B，直接指向B的后继C：`A->next = C` 即 `B->prev->next = B->next`。
2. 让B的后继C，回头跳过B，直接指向B的前驱A：`C->prev = A` 即 `B->next->prev = B->prev`。

完成这两步，节点 `B` 就从双向连接中删除了，整个过程的时间复杂度为 **O(1)**。

7.3 双向链表的插入逻辑

双向链表的插入操作同样高效，其核心在于要同时维护 `next` 和 `prev` 两个方向的链接。假设我们需要在节点 `A` 和其后继节点 `B` 之间，插入一个新节点 `X`，目标是形成 `A <--> X <--> B` 的新结构。



步骤一：将新节点 `X` 链接到其前驱和后继

首先设置新节点 `X` 自己的 `prev` 和 `next` 指针，让它正确地指向其前驱和后继。

1. **设置 `X` 的 `next` 指针**：
   - **动作**：将 `X` 的 `next` 指针指向 `A` 原本的后继节点 `B`。
   - **代码实现**：`X->next = A->next;`
2. **设置 `X` 的 `prev` 指针**：
   - **动作**：将 `X` 的 `prev` 指针指向它的前驱节点 `A`。
   - **代码实现**：`X->prev = A;`

完成这两步后，新节点 `X` 已经正确地连接到了链表中，但原有的 `A` 和 `B` 之间的链接还没有更新到 `X`。

步骤二：更新原有节点，使其指向新节点 `X`

接下来修改 `A` 和 `B` 的指针，让它们转而指向新节点 `X`。

3. **更新后继节点 `B` 的 `prev` 指针**：
   - **动作**：将 `B` 的 `prev` 指针从指向 `A` 改为指向 `X`。
   - **代码实现**：`A->next->prev = X;` (即 `B->prev = X`)
   - **注意**：这一步必须在第4步之前执行，因为第4步会修改 `A->next` 的指向。
4. **更新前驱节点 `A` 的 `next` 指针**：
   - **动作**：将 `A` 的 `next` 指针从指向 `B` 改为指向 `X`。
   - **代码实现**：`A->next = X;`

整个过程因为不涉及数据移动，时间复杂度同样为 **O(1)**。

八、 总结

- 链表的核心是用一个指针（`Next`信息）将物理上不连续的数据，在逻辑上串联起来。
- 遍历链表需要两个关键信息：**头节点** `head` 和 `Next` **关系网**。
- 遍历的三要素是：**起点（`current=head`）**、**终点判断（`current!=0`）** 和 **前进方式（`current=Next[current]`）**。
- 链表的核心优势在于其**插入**和**删除**操作的高效性，这源于它通过修改指针（后继/前驱关系）而非移动数据来改变逻辑顺序。
- **插入操作**的关键在于**"先连接新节点的后继，再修改前驱节点的后继"**，以确保链表的连续性。
- **单向链表**的删除操作存在一个瓶颈：**寻找待删节点的前驱节点是低效的 (O(N))**。
- **双向链表**通过增加一个 `prev` 指针，以微小的空间开销，换来了**O(1)的前驱查找能力**，使得插入和删除操作在各种情况下都极为高效，是实际应用中更常用、更强大的链表形态。