C++ STL排序 `sort()` 知识点总结

1. 头文件

`sort` 函数包含在 `<algorithm>` 头文件中。

```
#include <algorithm>
```

2. 函数原型与基本语法

`sort` 对给定区间内的元素进行排序。

- **函数原型**：`void sort(RandomAccessIterator first, RandomAccessIterator last);`
- **参数**：
  - `first`：指向要排序序列起始位置的迭代器（包含）。
  - `last`：指向要排序序列结束位置的下一位的迭代器（不包含）。
  - 即：排序区间为 **左闭右开 [first, last)**。

3. 默认行为

- **排序规则**：默认按照 **升序**（从小到大）排列。
- **比较依据**：使用元素类型自带的 `<` 运算符进行比较。
- **适用类型**：
  - 基本数据类型（`int`, `double`, `char` 等）。
  - C++内置的扩展类型（`string`，按字典序(ASCII码表的顺序)排序）。
  - 自定义结构体类型（需要配合自定义比较函数）。

4. 算法复杂度

- **时间复杂度**：
  - 平均情况：$O(N \log N)$
  - 最坏情况：$O(N \log N)$
- **空间复杂度**：
  - $O(\log N)$

5. 底层实现原理 (了解)

`sort` 并非简单的快速排序，而是一种混合排序算法：

1. **快速排序**： 作为主要排序手段，不断的将数据拆分分片。
2. **堆排序**： 当快速排序的递归深度过深（超过 $\log N$ 级别）时，自动切换为堆排序。
3. **插入排序**：当数据片段非常小（少于 16 个元素）时，切换为插入排序。因为对于极小数组，插入排序比快排和堆排更高效且常数更小。

6. 代码示例

示例 1：数组排序

```
int arr[] = {4, 2, 5, 1, 3};
// 排序整个数组，arr 是首地址，arr+5 是尾地址 + 1
sort(arr, arr + 5);
// 结果: 1, 2, 3, 4, 5
```

示例 2：Vector 排序

```
vector<int> v = {10, 30, 20};
sort(v.begin(), v.end());
// 结果: 10, 20, 30
```

示例 3：局部排序

```
vector<int> v = {5, 1, 9, 3, 7};
// 只排序前3个元素 (下标0到2)
sort(v.begin(), v.begin() + 3);
// 结果: 1, 5, 9, 3, 7 (后两个元素位置未动)
```

7. 自定义比较函数

`sort` 的第三个参数接收一个比较器，其中存在一个内置比较器`greater<T>()`用于降序排序。

```
sort(v.begin(), v.end(), greater<T>()); // T 为 vec 中储存的数据类型
```

而在这里如果我们需要自定义比较的方法的话，我们就需要传递一个**比较函数。**

- **函数参数**：必须接受两个跟你要排序的同类型的参数（建议使用 `const&`以提高效率，不加也可以，但是因为加上`&`是传递原件，能节省复印所需要的时间），并返回 `bool` 类型。
- **返回值含义**：
  - 返回 `true`：表示第一个参数应该排在第二个参数**前面。**
  - 返回 `false`：表示第一个参数应该排在第二个参数**后面**，或者两者相等。

8. 比较函数的写法模板

假设有一个结构体 `Student`：

```
struct Student {
    string name;
    int score;
    int id;
};
```

我们需要定义一个普通的函数（写在 `main` 函数外面）：

```
// 比较函数：cmp
bool cmp(const Student& a, const Student& b) {
    // 排序逻辑写在这里
    return a.score < b.score; // 当 a 应该排在前面时，需要返回 true，我们这里按照分数的升序排序
}
```

9. 常用排序逻辑

A. 升序排列 (从小到大)

逻辑：当 `a` 的值 **小于** `b` 的值时，`a` 在前，返回 `true`。

```
bool cmp(const Student& a, const Student& b) {
    return a.score < b.score;
}
```

B. 降序排列 (从大到小)

逻辑：当 `a` 的值 **大于** `b` 的值时，`a` 在前，返回 `true`。

```
bool cmp(const Student& a, const Student& b) {
    return a.score > b.score;
}
```

C. 多条件排序 (重点)

这是结构体排序最常见的场景。例如：**先按分数从高到低排序；如果分数相同，则按学号从小到大排序。**

```
bool cmp(const Student& a, const Student& b) {
    if (a.score > b.score) {
        // 第一关键字：分数不同，按分数大的在前排序
        return true;
    } else if(a.score < b.score){
        // 第一关键字：分数不同，按分数大的在前排序，a 更小所以应该排在后面，返回false
        return false;
    }else{
        // 第二关键字：分数相同，按学号升序（小的在前）
        return a.id < b.id;
    }
}
```

10. 完整代码示例

```
#include <bits/stdc++.h>
using namespace std;

struct Student {
    string name;
    int score;
    int id;
```

```
};

// 自定义比较函数：先按分数降序，分数相同按ID升序
bool cmp(const Student& a, const Student& b) {
    if (a.score > b.score) {
        // 第一关键字：分数不同，按分数大的在前排序
        return true;
    } else if(a.score < b.score){
        // 第一关键字：分数不同，按分数大的在前排序，a 更小所以应该排在后面，返回false
        return false;
    }else{ // 分数相同
        // 第二关键字：分数相同，按学号升序（小的在前）
        return a.id < b.id;
    }
}

int main() {
    vector<Student> students; // 为了方便，没有输入测试数据，你可以自行添加

    // 调用 sort，传入函数名 cmp 作为第三个参数
    sort(students.begin(), students.end(), cmp);

    // 输出结果
    for (const auto& s : students) {
        cout << s.name; // 输出的顺序已经按分数从高到低排序；如果分数相同，则按学号从小到大
排序
    }

    return 0;
}
```

11. 注意事项

在编写比较函数时，必须满足严格弱序原则。最常见的错误是使用 `<=` 或 `>=`。

- **错误写法**:

  ```
  bool cmp(const Student& a, const Student& b) {
      return a.score >= b.score; // 错误!
  }
  ```

  为什么? ：因为如果 `a.score` 等于 `b.score`，`cmp(a, b)` 返回 `true`，且 `cmp(b, a)` 也返回 `true`。这会让 `sort` 认为 a **必须**在 b 前面，同时 b **也必须**在 a 前面，导致逻辑矛盾，可能引发**死循环**或**程序崩溃**。

- **正确的方法**:
  当两个元素相等时，比较函数**必须**返回 `false`。两个都返回 `false` 函数就会认为 **a 等于 b** (谁在前无所谓，视为相等)。

- **正确写法**:

  ```
  bool cmp(const Student& a, const Student& b) {
      return a.score > b.score; // 正确
  }
  ```