Comprehensive Guide on Creating Custom Comparators in C++

Comparators in C++ allow you to define custom sorting orders for data structures such as `priority_queue`, `sort()`, and associative containers like `set` and `map`.

---

1. Understanding Comparators

A **comparator** is a function (or functor/lambda) that defines how two elements should be ordered in a container.
For two elements `a` and `b`:

- If `a` should come **before** `b`, return `true`.
- If `b` should come **before** `a`, return `false`.

---

2. Types of Comparators

1. **Function Pointer**
2. **Lambda Function**
3. **Functor (Function Object)**
4. **Operator Overloading (For Classes)**

---

3. Comparators in Sorting (`sort()`)

The `sort()` function in C++ uses a comparator to arrange elements in ascending or descending order.

Example 1: Sorting in Descending Order

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool cmp(int a, int b) {
    return a > b;  // Sort in descending order
}

int main() {
    vector<int> arr = {5, 3, 9, 1, 7};

    sort(arr.begin(), arr.end(), cmp);

    for (int num : arr) cout << num << " "; // Output: 9 7 5 3 1
}
```

> ☑ `sort()` by default sorts in **ascending** order. The comparator `cmp()` changes it to **descending**.

4. Comparators in `priority_queue`

By default, `priority_queue` in C++ is a **max-heap** (largest element at the top). If you need a **min-heap** or custom ordering, you must define a comparator.

Example 2: Min-Heap Using a Lambda Function

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

int main() {
    auto cmp = [](int a, int b) { return a > b; }; // Min-heap

    priority_queue<int, vector<int>, decltype(cmp)> pq(cmp);

    pq.push(5);
    pq.push(1);
    pq.push(8);
    pq.push(3);

    while (!pq.empty()) {
        cout << pq.top() << " ";  // Output: 1 3 5 8
        pq.pop();
    }
}
```

> ☑ The lambda function ensures the smallest element is always at the top.

5. Comparators for `pair<int, string>`

Example 3: Custom Sorting in a Priority Queue

- If two elements have the same integer value, sort them lexicographically.
- Otherwise, sort by integer value in **descending order**.

```
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

int main() {
    auto cmp = [](const pair<int, string>& a, const pair<int, string>& b) {
        if (a.first == b.first)
```

```
            return a.second > b.second;  // Lexicographical order
        return a.first < b.first;       // Higher number first
    };

    priority_queue<pair<int, string>, vector<pair<int, string>>, decltype(cmp)>
pq(cmp);

    pq.push({2, "banana"});
    pq.push({1, "apple"});
    pq.push({2, "apple"});
    pq.push({3, "date"});

    while (!pq.empty()) {
        cout << pq.top().first << " " << pq.top().second << endl;
        pq.pop();
    }

    return 0;
}
```

Output:

```
3 date
2 apple
2 banana
1 apple
```

☑ Highest numbers come first, and in case of ties, words are sorted alphabetically.

---

6. Comparators for `set` and `map`

By default, `set` and `map` store elements in **ascending order**. Custom comparators can change this behavior.

Example 4: Storing Pairs in a `set` with Custom Order

```
#include <iostream>
#include <set>

using namespace std;

// Comparator for set
struct CustomCompare {
    bool operator()(const pair<int, string>& a, const pair<int, string>& b) const
{
        if (a.first == b.first)
            return a.second < b.second;  // Lexicographically smaller first
        return a.first > b.first;       // Larger number first
    }
```

```
};

int main() {
    set<pair<int, string>, CustomCompare> s;

    s.insert({3, "apple"});
    s.insert({2, "banana"});
    s.insert({2, "cherry"});
    s.insert({1, "date"});

    for (auto p : s)
        cout << p.first << " " << p.second << endl;

    return 0;
}
```

Output:

```
3 apple
2 banana
2 cherry
1 date
```

☑ The set is now ordered in descending order of `int`, with ties resolved lexicographically.

---

7. Comparators for Sorting Custom Classes

Example 5: Sorting a Vector of Custom Objects

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Student {
public:
    string name;
    int marks;

    Student(string n, int m) : name(n), marks(m) {}
};

// Comparator for sorting students
bool cmp(const Student& a, const Student& b) {
    if (a.marks == b.marks)
        return a.name < b.name; // Sort lexicographically for same marks
    return a.marks > b.marks;   // Higher marks first
```

```
}

int main() {
    vector<Student> students = {{"Alice", 85}, {"Bob", 92}, {"Charlie", 85},
{"Dave", 95}};

    sort(students.begin(), students.end(), cmp);

    for (const auto& s : students)
        cout << s.name << " " << s.marks << endl;

    return 0;
}
```

Output:

```
Dave 95
Bob 92
Alice 85
Charlie 85
```

> ☑ Sorting prioritizes higher marks, with lexicographical order as a tiebreaker.

---

8. Comparators Using `greater<>` (Built-in)

If you want to **reverse the default order**, you can use `greater<>` from `<functional>`.

Example 6: Min-Heap Using `greater<>`

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional>

using namespace std;

int main() {
    priority_queue<int, vector<int>, greater<int>> pq; // Min-heap

    pq.push(5);
    pq.push(1);
    pq.push(8);
    pq.push(3);

    while (!pq.empty()) {
        cout << pq.top() << " ";  // Output: 1 3 5 8
        pq.pop();
```

```
    }
}
```

☑ `greater<>` makes `priority_queue` behave like a min-heap.

---

9. Summary

Use Case	Default Order	Custom Comparator
`sort()`	Ascending	`bool cmp(a, b) { return a > b; }`
`priority_queue<T>`	Max-Heap	`priority_queue<T, vector<T>, decltype(cmp)>`
`set<T>`	Ascending	`set<T, CustomCompare>`
`map<K, V>`	Ascending by Key	`map<K, V, CustomCompare>`

---

10. Key Takeaways

☑ Use **lambda functions** for short, inline comparators.
☑ Use **functors** (`struct` with `operator()`) for reusable comparators.
☑ Use **min-heaps** (`greater<>` or custom comparator) when needed.
☑ Always **check `first` before `second`** when dealing with `pair<>`.

Let me know if you need more clarifications! 🚀