Understanding comparators?

When we are building associative containers like set, multiset, map, multimap etc, we need to have a way to order the things. While the basic datatypes have relational operators like <, >, <=, >= etc, we don't have a way to compare every data that the user throws at us. So, we ask the user to tell us a way to compare the items he is giving us. This custom 'way' of how to compare things and decide what comes first and what comes second is what needs to be defined as `Comparator Structure or Function`.

Since we are developing containers, we bother mainly about `Comparator structure`. Before knowing, how we deal with the comparators, let's first understand how comparators are defined. Please note that here we are talking about comparator.

The syntax:

```
template<class T>
struct Comparator {
    bool operator() (const T& a, const T& b);
};
```

Before we get to operator overloading and the structure, note that the function return type is `bool`. The containers need to be coded in a way that if the return value of the comparator is `false`, it order the element `b` before `a`. Else, `a` is put before `b`. For example, if we are developing a `set` and the return value of comparing the new node with root node is `true` when passed in the order of `(root, node)`, then the node must be moved to the right of the tree because root should be appear before the new node when traversing in `inorder`.

Now to the confusing part. As you might be able to understand, we are overloading `()` in this code. This is so that we can use `()` operator with the object of `Comparator` structure. An example usage is as follows:

```
int a=10, b=5;
Comparator<int> cmp;
cmp(a, b);
```

This usage is similar to overloading `[]`. But in `[]`, it can take only one argument. Here, `()` is taking two arguments.

A valid question is why do we need to overload `()` operator in a structure and use it's object for comparision. Why not just create something like `bool comparator(const T& a, const T& b);`?

Yes, we can and that's what `Comparator functions` are but the containers take input related to datatype of elements only through templates. There are ways you can pass custom comparators in different ways but the ideal way is to pass them through template arguments. And the template arguments, or for that matter any type of arguments, must be of some type of variable. They need to point to some memory in the system. Functions are in memory(code section) but we can't point to them directly. We need to use function pointers, which is much more mind-bending than what we are dealing with right now(we also have the trendy lambda, but with all that's going on, you wouldn't want to know about it right now). Other way is to pass the function in some kind of structure and call the function from there so that we satisfy the point of passing `data types` as template arguments.

Please note that this function to be called can have any name actually. For example, the usage of comparators can be:

```
template<class T>
struct Comparator{
    bool compare(const T& a, const T& b);
};

/*************Somewhere in container code*****************/
int a=10, b=5;
Comparator<int> cmp;
cmp.compare(a, b);
```

Then, why are we having to overload `()` and having to deal with it. It's just convention that people adhered to so that the container can understand how to use the comparator. The container needn't bother with what function to call in the structure. Also, another name of these type of structures is `functors`. Because in the syntax shown previously we are using `object` as a `function`.

Hopefully, that covers your doubts about why the comparator need to be declared the way we are doing. Now, if there is anyone confused with the sample provided here about the comment on overloading operators <, > etc. One should understand that comparision is a way of relating different elements with each other. So, we need relational operators to compare. Also, it is to be noted that we needn't do any of this relational operator overloading. It is on the user using our container to do so. If he doesn't provide a way to perform relational operation, then our default comparator structures will fail and he will have to pass his own comparator function.

For example, suppose the user created a custom string class but didn't overload relational operators. And he wants to use our custom set container. We have defined a default comparator which uses `<` operator for comparision. But, since the user hasn't provided any code for operator `<`, the code won't compile. So, the user could either just overload operator `<` or he could create a custom comparator. In either case, he has the flexibility of using any kind of logic he wants. Like he could compare strings lexicographically or the size. Whatever way he chooses, the set container doesn't know and doesn't care. All it requires is for the user to adhere to the convention that is to be followed. If s/he doesn't, container with the help of compiler shows the middle finger.

Understanding CustomSet

What we are doing with `CustomSet` is just creating a BST based `set` container. You can view the full code here.

First, we declared a templated `Node` class. That doesn't need explanation. Next, we are defining two comparators `Less` and `Great`.

Code for `Less`:

```
template <class T> struct Less {
    bool operator()(const T &lhs, const T &rhs) {
        // Datatype T must have operator< defined. Otherwise, this fails.
        return lhs < rhs;
    }
};
```

Code for `Great`:

```
template <class T> struct Greater {
    bool operator()(const T &lhs, const T &rhs) {
        // Datatype T must have operator> defined. Otherwise, this fails.
        return lhs > rhs;
    }
};
```

As being said in the comment above, these comparator functions fail when operators `<` or `>` are not defined. And we don't need to handle this case because it is the responsibility of the user to do that.

I believe the code is self-explanatory now with all that you have read above.

Onto declaring our `CustomSet`.

```
template<class T, class Compare = Less<T>>
class CustomSet {
    private:
        Node<T> *gRoot;
        Comparator cmp;

        Node<T> *_insert(Node<T>* root); //explained later
        void _inorder(Node<T>* root); //doesn't need explanation
    public:
        Node() : gRoot(NULL), cmp(Comparator()) {}
        void insert(); //doesn't need explanation
        void inorderPrint(); //doesn't need explanation
};
```

The first thing we need to talk about is the second template argument `Compare`. It is the argument that takes the `Comparator structure` that either the user provides or the default one(`Less` in this case). Just like a variable is maintained for the custom datatype provided by the user(`T` in our case), we also maintain a variable for the `Comparator structure`. The variable is `cmp`. It is initialized to an object in the constructor of `CustomSet` class.

Now, let's look at `_insert()` code to understand how comparator is used.

```
Node<T> *_insert(Node<T> *root, Node<T> *node) {
        if (root == NULL) {
            root = node;
            return root;
        }

        // cmp(a, b) checks whether a should come before b in the order.
        if (cmp(root->key, node->key))
            root->right = _insert(root->right, node);

        // cmp(b, a) to check if b should come before a
        else if (cmp(node->key, root->key))
            root->left = _insert(root->left, node);

        // will reach here when both are equal. u can insert the node in either
        // left or right if multiset. if normal set, directly return.
        else
            root->left = _insert(root->left, node);

        return root;
    }
```

In the code above, `cmp(root->key, node->key)` calls the `()` overloaded operator in `Compare` structure as it is called on `cmp`. `root->key`, `node->key` will be parameters. So, the call will be `cmp.operator()(root->key, node->key)` and final result will be same as `root->key < node->key`.

One might ask why not put that equation `root->key < node->key` directly as we know that that is the way a set needs to ordered. This question assumes two things. 1) User gives types that can be comparable with `<` operator and 2) That the user only wants to do that kind of ordering.

What if the user wants to order it in descending order. Then, he could create a `Comparator structure` similar to `Great` as described above. Now, we needn't change our container code to `root->key > node->key` everytime user wants that way, we can just ask him the way he wants to compare and compare it that way.

TL;DR

They key things to remember when developing a container with custom comparators.

1. Declare a template argument that takes a comparator structure. Eg:`template<class T, class Compare = Less<T>>`. Here, `Compare` is our comparator structure which takes `Less` as default type of structure.
2. Create a variable for `Compare` in the container class. Eg: `Compare cmp;`
3. Initialize `cmp` in the constructor. Eg: `Container() : cmp(Comparator()) {}`
4. Wherever you have comparision logic, replace it with custom comparator. Eg: replace `if(a < b)` with `if(cmp(a, b))`.
5. Don't forget to define the default comparator. You can copy paste the code of `Less` as defined above. It is the same code used by C++ STl
6. If the user doesn't provide atleast one of `operator<` or custom comparator, feel free to make the his/her life miserable.

HOPEFULLY, this clears your doubts.