

FUSHAR'S BLOG

[MY APPS](#) [ARCHIVES](#) [CONTACT](#) [ABOUT](#)

Follow me on:



3 Ways to Define Comparison Functions in C++

August 8, 2011 · by Ashar Fuadi · [22 Comments](#)

One of the reasons that programming in C++ is superior to Pascal is the existence of STL (Standard Template Library) that contains many useful containers and functions.

There are many C++ STL functions that require the underlying parameter to have an ordering, such as `sort()`. Obviously, you can only sort a collection of objects if you can tell whether an object must come before another, so it is important to learn how to define an ordering in a class.

There are also many C++ STL containers that require the underlying type to have an ordering, such as `set<T>` and `priority_queue<T>`.

This post describes how to define an ordering of a class so that it can be used in C++ STL containers or functions that require ordering. As a C++ programmer you definitely need to know these methods.

How to Define Ordering?

In a nutshell, defining an ordering of a class `T` means that for all two objects `a` and `b` of type `T`, we can always determine whether `a` must precede `b` in the ordering. The definition is then implemented in a function that returns a boolean denoting the precedence. Specifically, we would like to implement a function of the form `f(x, y)` that takes two parameters of the same type, and returns whether `x` must come before `y`.

Strict Weak Ordering

Almost all functions/containers (except for example `priority_queue<T>`) require the ordering to satisfy the standard mathematical definition of a *strict weak ordering*, or else the behavior of the functions/containers will be undefined.

Let `f(x, y)` be a comparison function. `f` is in strict weak ordering if it satisfies three invariants:

- **Irreflexivity**
`f(x, x) = false`.
- **Antisymmetry**
If `f(x, y)` then `!f(y, x)`.
- **Transitivity**
If `f(x, y)` and `f(y, z)` then `f(x, z)`.
- **Transitivity of Equivalence**
Let `equal(x, y) = !f(x, y) && !f(y, x)`. If `equal(x, y)` and `equal(y, z)` then `equal(x, z)`.

Don't worry, if your comparison function more or less means "less than", it will almost surely satisfy those invariants.

The Three Ways

There are (at least) three ways to define an ordering in C++. For simplicity, in this post, “STL container” means “STL container that requires its underlying type to have an ordering”, and “STL function” means “STL container that requires its parameters to have an ordering”.

1. Define operator<()

This method can be used if you want objects of a custom class to be able to be sorted naturally. By naturally I mean it is default way to sort objects of this class. For example, you have a class **Edge** defined like this.

```
struct Edge
{
    int from, to, weight;
};
```

Because you are implementing Kruskal's Algorithm, you want to sort edges of your graph in decreasing order of their weight. Specifically, an edge **a** must precede another edge **b** if **a**'s weight is greater than **b**'s. Define **operator<()**, like this.

```
struct Edge
{
    int from, to, weight;
    bool operator<(Edge other) const
    {
        return weight > other.weight;
    }
};
```

More specifically, you define a function with prototype:

```
bool operator<(T other) const
```

that returns **true** if ***this** (the current object) must precede **other**, and **false** otherwise. Note that the **const** keyword is required; it means that you cannot modify member variables of the current object.

If you don't like this syntax, i.e., a single parameter when you actually are comparing two objects, you can use this alternative syntax instead.

```
struct Edge
{
    int from, to, weight;
    friend bool operator<(Edge a, Edge b)
    {
        return a.weight > b.weight;
    }
};
```

This way it is much clearer that you are comparing **a** and **b**, not ***this** and **other**. Note also that friend function is like static function; it cannot access member variables.

An example of classes that implement the natural ordering is STL's `pair<T1, T2>`. Two objects of type `pair<T1, T2>` will be compared according to their first keys, or if they are equal, compared according to their second keys.

All built-in types also implements natural ordering (implemented by the compiler). For example, an int **a** comes before an int **b** if **a** is less than **b**.

A class that has natural ordering defined can be used directly in STL functions:

```
vector<Edge> v;
sort(v.begin(), v.end());
```

We can also use this class as an underlying type of STL containers:

```
priority_queue<Edge> pq;
set<Edge> s;
```

2. Define a custom comparison function

Use this method if you are comparing built-in types, you cannot modify the class you are comparing, or you want to define another ordering besides its natural ordering.

Basically, a comparison function is just a function that takes two parameter of the same type and returns a boolean:

```
bool name(T a, T b)
```

For example, you want to sort a vector<int> **data** in ascending order of their occurrences in a text. The number of occurrences of a number is available in vector<int> **occurrences**. Define a function, say, with name **cmp**:

```
bool cmp(int a, int b)
{
    return occurrences[a] < occurrences[b];
}
```

Now you can sort data by specifying the comparison function to use as the additional argument:

```
sort(data.begin(), data.end(), cmp);
```

Always consult STL reference on where to place the comparison function in an STL function.

3. Define operator()

You can use comparison function for STL containers by passing them as the first argument of the constructor, and specifying the function type as the additional template argument. For example:

```
set<int, bool (*)(int, int)> s(cmp);
```

You may use this option, but it may be too confusing for most users. You may want to use the third method, by utilizing *functors*.

A *functor*, or a function object, is an object that can behave like a function. This is done by defining **operator()** of the class. In this case, implement **operator()** as a comparison function:

```
vector<int> occurrences;
struct cmp
{
    bool operator()(int a, int b)
    {
        return occurrences[a] < occurrences[b];
    }
};
```

Now, you pass this class as a template argument to STL containers. The details vary between containers, consult STL reference this.

```
set<int, cmp> s;
priority_queue<int, vector<int>, cmp> pq;
```

STL also has some built-in functor classes such as `less<T>`, the default comparison class, and `greater<T>`, the negation of `less<T>`.

A functor can be used as an ordinary function by instantiating the class. For this purpose, the simplest way is to append `()` after the class name. So, for example, if you want to sort a `vector<int>` `data` in descending order (that's the negation of ascending order), you may use:

```
sort(data.begin(), data.end(), greater<int>());
```

Bonus Tips

const T&

If your class structure is large, then it is better that the parameters in your comparison function use `const` keyword and reference operator `&`. Example:

```
bool cmp(const Edge& a, const Edge& b)
```

The advantage is that it ensures that you are unable to modify the contents of `a` and `b`, and they are passed as reference only, not the whole objects.

sort() v.s. stable_sort()

It is possible that for two objects `a` and `b` of type `T`, neither `f(a, b)` nor `f(b, a)` returns true. In this case, `a` and `b` is said to be *equivalent* under comparison function `f`. If you sort a `vector<T>` `data` with `sort()`, then for each two equivalent elements their final relative order are undefined; `a` may comes before `b` or vice-versa.

If you want to preserve the relative order of equivalent elements, use `stable_sort()`. The final order of equivalent elements will be the same as their initial order before being sorted.

References

- [SGI's STL Programmer's Guide](#)



Like 35

Tweet

Related posts:

- [Polyglot: Write Once, Compile Anywhere](#)
- [Write C++ without Semicolons!](#)
- [Solving Linear Recurrence for Programming Contest](#)
- [Add Epsilon Before Casting Double to Integer!](#)
- [Solving Linear Recurrence for Programming Contest – Examples](#)

filed under: [tips](#) · tagged: [c++](#)



About Ashar Fuadi

Ashar Fuadi is a competitive programmer from University of Indonesia. He loves to code, especially for TopCoder SRM, Codeforces, and ICPC.

Follow Ashar on [Google+](#) and [Twitter](#).

Comments

fushar says:

August 11, 2011 at 03:50

Thanks to **tomek** for pointing out that an ordinary function can be used as a comparer in an STL container.



[Reply](#)

Kemal Maulana says:

August 11, 2011 at 05:29

nice post shar!



[Reply](#)

adekroisa says:

September 18, 2011 at 12:46

Thanks for information.. 😊



[Reply](#)

Jerry Wong says:

June 10, 2012 at 17:12

im not quite sure for the following part,why you need a greaterthan operator, "weight > other.weight", to define the lessthan operator??

"struct Edge

{

int from, to, weight;

bool operator other.weight;

}

};"



[Reply](#)

fushar says:

June 11, 2012 at 23:22

Because in Kruskal's Algorithm, we want to sort the edges in decreasing order of their weights, so we can just reverse the operator sign 😊



Mohamed Aboutabl says:

January 27, 2013 at 01:36

Ashar,

Jazak Allah Khair. This is indeed an excellent tutorial on using comparators. I will point my students to your page to learn this subject.



[Reply](#)

smit hinsu says:

February 13, 2013 at 00:47

I liked this tutorial .

Why stl needs a function which will return false when a is equal to b .

I understand that this is requirement and we need to follow it strictly , otherwise behaviour of the sort function would be unpredictable.

I want to know what internal implementation causes this ?



[Reply](#)

lisa says:

May 15, 2013 at 05:50



so you have to define occurrences as a global variable, right? it cannot be passed to cmp as a parameter.

I am writing dijkstra algorithm in which the vertex in priority_queue Q are sorted according to their distance to the source. When writing the comparing logic in operator(), it seems the distance[] has to be defined as global, according to your post. but it looks ugly. is there any more elegant way? Thanks.

[Reply](#)

Ashar Fuadi says:

May 15, 2013 at 09:38



@ smit: Please read http://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings for more information.

@ lisa: You can use a class that implements operator() to accomplish this. This class should also store the distance array. For example: <http://ideone.com/z2g36S>

[Reply](#)

AyushSharma says:

October 13, 2013 at 13:55



I have declared a class and in that class I have three vectors of pairs. I need to sort these three arrays based on different comparators. Is there a way to declare these 3 comparator functions?

[Reply](#)

Ashar Fuadi says:

October 20, 2013 at 10:40



@ AyushSharma: Sorry for the late reply. I still don't understand your question clearly. Are you sorting the arrays, or the objects of the class?

[Reply](#)

cyberaxe says:

October 17, 2014 at 12:02



Thanks!

[Reply](#)

aymos says:

December 15, 2014 at 02:45



Great post! 😊

[Reply](#)

Yogeshwar says:

December 25, 2014 at 13:07

Thanks for the article.



[Reply](#)

Dan says:

March 14, 2015 at 01:43

Really informative!



[Reply](#)

Pragadeesh says:

June 7, 2015 at 17:12

Thanks! You made it clear for me 😊



[Reply](#)

Bob says:

August 31, 2016 at 15:01

Note that `std::priority_queue` *does* require a strict weak ordering. See 23.6.4 Class template `priority_queue` in the C++11 standard for example.



[Reply](#)

Piyush Agarwal says:

March 10, 2017 at 13:40

Thanks a lot for this blog! This page keeps helping me every now and then.



[Reply](#)

crisp says:

March 16, 2017 at 10:47

Can I add `const T&` for all three ways? i.e

`bool operator < (const Edge& other)`

`bool operator () (const Edge& p, const Edge& q)`

`bool cmp (const Edge& p, const Edge& q)`



[Reply](#)

Nidhi Kesarwani says:

April 5, 2018 at 12:00

Thank you so much Asar.. I had a vague Idea of the compare function. I was always confused .. but you made it easy.



[Reply](#)

Malcolm says:

October 9, 2018 at 15:17

can you add the method for using lambda functions?



[Reply](#)

Trackbacks

New Learnings: Custom Comparator with STL, const int * const ptr, reference to pointer to const, Operator Overloading – 108 Foundation says:

August 25, 2016 at 17:28

[...] Adding a custom comparator with STL: <http://fusharblog.com/3-ways-to-define-comparison-functions-in-cpp/> [...]

[Reply](#)

Speak Your Mind

 Name * Email * Website