It's back! **Take the 2018 Developer Survey today »**

How to create std::set of structures

I need to create stl::set of structures. I write

```
stl::set <Point> mySet; //Point - name of structure
```

But then I try to add a structure instance to mySet

```
Point myPoint;
mySet.insert(myPoint);
```

There are several compilation errors (error C2784, error C2676). Somebody can give an advice?

> 1>C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\xfunctional(125): error C2784: bool std::operator <(const std::vector<_Ty,_Ax> &,const std::vector<_Ty,_Ax> &): failed to bring the argument to a template "const std::vector<_Ty,_Ax> &" from"const Point"
>
> 1>C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\include\xfunctional(125): error C2676: binary "<": "const Point "does not define this operator or a conversion to a type acceptable to the integrated operator

c++ stl set

edited Feb 2 '17 at 16:48 — WhozCraig — 47.6k 8 51 97

asked Jan 14 '17 at 9:25 — Dmitrii — 59 6

Well, its `std::set` . `C2784` and `C2676` don't make any sense. We need the actual message. Also, did you make sure `struct Point` implements `operator<` ? `std::set` requires it. – DeiDei Jan 14 '17 at 9:28

1 Define an 'operator<' for your 'Point' struct. – void Jan 14 '17 at 9:40

1 Answer

The **`std::set`** template provides an associative container that contains a sorted set of unique objects. The key words there is **sorted** and **unique**. To support sorting, a number of possibilities ensue, but ultimately the all must lead to a conforming with *strict weak ordering*.

The second template argument to `std::set` is a *comparison* type. The default, `std::less<Key>` , is supplied by the standard library, where `Key` is the type of object you're storing in your container (in your case, `Point`). That default simply generates a comparison using any allowable available `operator <` supporting the key type. Which means one way or another, if you're using the default comparator (`std::less<Point>` in your case), then your class must suppose operations like this:

```
Point pt1(args);
Point pt2(args);

if (pt1 < pt2)  // <<=== this operation
    dosomething();
```

Multiple methods for doing this appear below:

Provide a member `operator <`

By far the easiest method to accomplish this is to provide a member `operator <` for your `Point` class. In doing so `pt1 < pt2` becomes valid and `std::less<Point>` is then happy. Assuming your class is a traditional x,y point, it would look like this:

```
struct Point
{
    int x,y;

    // compare for order.
    bool operator <(const Point& pt) const
    {
        return (x < pt.x) || ((!(pt.x < x)) && (y < pt.y));
    }
};
```

Provide a Custom Comparator Type

Another method would be to provide a custom comparator type rather than relying on `std::less<Point>`. The biggest advantage in this is the ability to define several that can mean different things, and use them in containers or algorithms as appropriately needed.

```
struct CmpPoint
{
    bool operator()(const Point& lhs, const Point& rhs) const
    {
        return (lhs.x < rhs.x) || ((!(rhs.x < lhs.x)) && (lhs.y < rhs.y));
    }
};
```

With that, you can now declare your `std::set` like this:

```
std::set<Point,CmpPoint> mySet;
```

Something to consider with this approach: The type is not part of `Point`, so any access to private member variables or functions has to be accounted for via friending in come capacity.

Provide a free-function `operator <`

Another less common mechanism is simply provide a global free-function that provides `operator <`. This is NOT a member function. In doing this, once again, the default `std::less<Point>` will result in valid code.

```
bool operator <(const Point& lhs, const Point& rhs)
{
    return (lhs.x < rhs.x) || ((!(rhs.x < lhs.x)) && (lhs.y < rhs.y));
}
```

This may seem a mix of both the custom comparator and the member operator, and indeed many of the pros and cons of each come along. Ex: like the member `operator <`, you can just use the default `std::less<Point>`. Like the custom comparator, this is a non-class function, so access to private members must be provided via friending or accessors.

Summary

For your needs, I'd go with the simple approach; just make a member `operator <`. Chances are you'll always want to order your `Point`s in that fashion. If not, go with the custom comparator. In *either* case make *sure* you honor strict weak ordering.

answered Jan 14 '17 at 10:13

WhozCraig
47.6k 8 51 97

Join Stack Overflow to learn, share knowledge, and build your career. Email Sign Up OR SIGN IN WITH Google Facebook