sorting a vector of structs [duplicate]

This question already has an answer here:
Sorting a vector of custom objects *12 answers*

I have a `vector<data> info` where `data` is defined as:

```
struct data{
    string word;
    int number;
};
```

I need to sort `info` by the length of the word strings. Is there a quick and simple way to do it?

c++ sorting vector

edited Jan 17 '12 at 23:47 — greatwolf — 15.3k 9 52 93

asked Feb 3 '11 at 22:49 — calccrypto — 3,028 15 52 83

marked as duplicate by Walter, Sebastian, Kon, SingerOfTheFall, Eric Brown Sep 13 '13 at 6:00

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

close this message

1 If you think your question was solved, mark one solution as accepted. – Murilo Vasconcelos Feb 4 '11 at 1:40

sorry. i didnt have time to check for the past few hours – calccrypto Feb 4 '11 at 3:18

4 Answers

Use a comparison function:

```
bool compareByLength(const data &a, const data &b)
{
    return a.word.size() < b.word.size();
}
```

and then use `std::sort` in the header `#include <algorithm>` :

```
std::sort(info.begin(), info.end(), compareByLength);
```

edited Dec 14 '15 at 20:06 — TryToSolveItSimple — 422 8 13

answered Feb 3 '11 at 22:52 — Oliver Charlesworth — 212k 25 436 573

What if I wish to sort the vector in a lexicographic manner according to the string filed ? (I'm using C++11 if it matters). Is there a way to approach it other than defining a comparison function /use lambda and instead use the integral operator< of std::string ? Below is my solution using lambda: sort(info.begin(),info.end(), [](const data& d1, const data& d2) { return (d1.word.compare(d2.word) < 0); }); – Guy Avraham Jul 6 '17 at 6:31

Just make a comparison function/functor:

```
bool my_cmp(const data& a, const data& b)
{
    // smallest comes first
    return a.word.size() < b.word.size();
}

std::sort(info.begin(), info.end(), my_cmp);
```

Or provide an `bool operator<(const data& a) const` in your `data` class:

```
struct data {
    string word;
    int number;

    bool operator<(const data& a) const
    {
        return word.size() < a.word.size();
    }
};
```

```
    int number;
};

bool operator<(const data& a, const data& b)
{
    return a.word.size() < b.word.size();
}
```

and just call `std::sort()` :

```
std::sort(info.begin(), info.end());
```

edited Feb 3 '11 at 23:26

answered Feb 3 '11 at 22:52

Murilo Vasconcelos
3,628 13 25

Op< should be a non-member and number should probably be considered in op< so other algorithms, such as std::unique, behave as expected when used with the default std::less; otherwise spot on. – Fred Nurk Feb 3 '11 at 23:06

Why `operator<()` should be non-member? – Murilo Vasconcelos Feb 3 '11 at 23:09

@MuriloVasconcelos: So implicit conversions apply to the left-hand side. – Fred Nurk Feb 3 '11 at 23:16

1 IMHO, you shouldn't use operator overloading to wrap behaviour that isn't immediately intuitive. In this situation, it doesn't really make any sense to say that `data a` is "less than" `data b` if its string member is shorter, so I wouldn't use `operator<` to express that idea. – Oliver Charlesworth Feb 3 '11 at 23:26

2 In this case I agree with you and is why I write about the "function-way" first and then I explained the other ways for learning purposes. – Murilo Vasconcelos Feb 3 '11 at 23:31

|

As others have mentioned, you could use a comparison function, but you can also overload the < operator and the default `less<T>` functor will work as well:

```
struct data {
    string word;
    int number;
    bool operator < (const data& rhs) const {
        return word.size() < rhs.word.size();
    }
};
```

Then it's just:

```
std::sort(info.begin(), info.end());
```

Edit

As James McNellis pointed out, `sort` does not actually use the `less<T>` functor by default. However, the rest of the statement that the `less<T>` functor will work as well is still correct, which means that if you wanted to put `struct data` s into a `std::map` or `std::set` this would still work, but the other answers which provide a comparison function would need additional code to work with either.

edited Feb 3 '11 at 23:08

answered Feb 3 '11 at 22:56

user470379
4,286 9 18

Interestingly, while `std::map` and `std::set` default to using `std::less<T>` , `std::sort` and the rest of the sorting functions default to using `operator<` . You'll only notice a difference if you specialize `std::less` to do something other than what `operator<` does. – James McNellis Feb 3 '11 at 23:00

When I said "you'll only notice a difference if...," I was wrong. You'll also notice a difference if you have a container of pointers, e.g. `std::vector<int*> v; v.insert(new int); v.insert(new int); std::sort(v.begin(), v.end());` , since the behavior is undefined if you compare unrelated pointers using `<` . That said, why you'd want to sort a container of pointers by the pointer value and not the value of the pointed-to object, I don't know. – James McNellis Feb 3 '11 at 23:57

Yes: you can sort using a custom comparison function:

```
std::sort(info.begin(), info.end(), my_custom_comparison);
```

`my_custom_comparison` needs to be a function or a class with an `operator()` overload (a functor) that takes two `data` objects and returns a `bool` indicating whether the first is ordered prior to the second (i.e., `first < second`). Alternatively, you can overload `operator<` for your class type `data` ; `operator<` is the default ordering used by `std::sort` .

Either way, the comparison function must yield a strict weak ordering of the elements.