HAYS
Recruiting experts in Digital Technology
HOW TO:
MASTER YOUR DEV DAEMONS
5 WAYS TO TAKE BACK CONTROL
Read the blog

Iterator Loop vs index loop [duplicate]

Ask Question

> **Possible Duplicate:**
> Why use iterators instead of array indices?

I'm reviewing my knowledge on C++ and I've stumbled upon iterators. One thing I want to know is what makes them so special and I want to know why this:

```
using namespace std;

vector<int> myIntVector;
vector<int>::iterator myIntVectorIterator;

// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

for(myIntVectorIterator = myIntVector.begin();
        myIntVectorIterator != myIntVector.end();
        myIntVectorIterator++)
{
    cout<<*myIntVectorIterator<<" ";
    //Should output 1 4 8
}
```

is better than this:

```
using namespace std;

vector<int> myIntVector;
// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

for(int y=0; y<myIntVector.size(); y++)
{
    cout<<myIntVector[y]<<" ";
    //Should output 1 4 8
}
```

And yes I know that I shouldn't be using the std namespace. I just took this example off of the cprogramming website. So can you please tell me why the latter is worse? What's the big difference?

c++ loops c++11 indexing iterator

edited May 23 '17 at 11:47

CodingMadeEasy
678 1 8 18

marked as duplicate by Yuushi, Tadeusz Kopec, skolima, 0x499602D2, BЈовић Jan 17 '13 at 12:11

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.

1 Please read contrast with indexing on Wikipedia. – Jesse Good Jan 17 '13 at 7:18

8 Answers

The special thing about iterators is that they provide the glue between algorithms and containers. For generic code, the recommendation would be to use a combination of STL algorithms (e.g. `find`, `sort`, `remove`, `copy`) etc. that carries out the computation that you have in mind on your data structure (`vector`, `list`, `map` etc.), and to supply that algorithm with iterators into your container.

Your particular example could be written as a combination of the `for_each` algorithm and the `vector` container (see option 3) below), but it's only one out of four distinct ways to iterate over a std::vector:

1) index-based iteration

```
for (std::size_t i = 0; i != v.size
    // access element as v[i]

    // any code including continue,
}
```

Advantages: familiar to anyone familiar with C-style code, can loop using different strides (e.g. `i += 2`).

Disadvantages: only for sequential random access containers (`vector`, `array`, `deque`), doesn't work for `list`, `forward_list` or the associative containers. Also the loop control is a little verbose (init, check, increment). People need to be aware of the 0-based indexing in C++.

2) iterator-based iteration

```
    // any code including continue,
}
```

Advantages: more generic, works for all containers (even the new unordered associative containers, can also use different strides (e.g. `std::advance(it, 2)`);

Disadvantages: need extra work to get the index of the current element (could be O(N) for list or forward_list). Again, the loop control is a little verbose (init, check, increment).

3) STL for_each algorithm + lambda

```
std::for_each(v.begin(), v.end(),
     // if the current index is ne
     auto i = &elem - &v[0];

     // cannot continue, break or
});
```

Advantages: same as 2) plus small reduction in loop control (no check and increment), this can greatly reduce your bug rate (wrong init, check or increment, off-by-one errors).

Disadvantages: same as explicit iterator-loop plus restricted possibilities for flow control in the loop (cannot use continue, break or return) and no option for different strides (unless you use an iterator adapter that overloads `operator++`).

4) range-for loop

```
for (auto& elem: v) {
     // if the current index is ne
     auto i = &elem - &v[0];

    // any code including continue,
}
```

Advantages: very compact loop control, direct access to the current element.

Disadvantages: extra statement to get the index. Cannot use different strides.

What to use?

For your particular example of iterating over `std::vector` : if you really need the index (e.g. access the previous or next element, printing/logging the index inside the loop etc.) or you need a stride different than 1, then I would go for iterator loop unless the code contained no flow control inside the loop and needed stride 1, in which case I'd go for the STL `for_each` + a lambda.

edited May 23 '17 at 12:26

Community♦
1 1

answered Jan 17 '13 at 8:04

TemplateRex
52.2k 14 118 228

1 Well if iteration is done over only one container I guess using iterators with `next`, `prev`, `advance` functions even in case of need in previous/ next elements and/or different stride would do just fine and possibly will be even more readable. But using several iterators to iterate several containers simultaneously doesn't look very elegant and most likely indexes should be used in this case. – Predelnik May 16 '14 at 13:33

This is a very informative answer! Thank you for laying out the pros and cons of these four different approaches. One question: The index-based iteration uses `i != v.size()` for the test. Is there a reason to use `!=` instead of `<` here? My C instincts tell me to use `i < v.size()` instead. I would expect that either one should work the same, I'm just more used to seeing `<` in a numeric `for` loop. – Michael Geary Sep 21 '17 at 5:02

Using the range loop, wouldn't this require for the container to have the elements in an array like order? Would this still work to get the index with a container which does not store the items in sequential order? – Devolus Nov 14 '17 at 8:05

Iterators make your code more generic.
Every standard library container provides an iterator hence if you change your container class in future the loop wont be affected

But don't all container classes have a size function? If I were to change the original container the latter should still be able to work because the size method doesn't change. – CodingMadeEasy Jan 17 '13 at 7:23

@CodingMadeEasy: in C++03 and earlier, `std::list` had an O(n) `size()` function (to ensure sections of the list - denoted by iterators - could be removed or inserted without needing an O(n) count of their size in order to update the overall container size: either way you win some / lose some). – Tony Delroy Jan 17 '13 at 7:28

1 @CodingMadeEasy: But builtin arrays don't have a size function. – Sebastian Mach Jan 17 '13 at 7:36

4 @CodingMadeEasy But not all containers offer random access. That is, `std::list` doesn't (and can't) have `operator[]` (at least not in any efficient way). – Angew Jan 17 '13 at 7:42

@phresnel I wasn't aware that you could iterate through arrays. I thought they were only for container classes. – CodingMadeEasy Jan 17 '13 at 7:43

Iterators are first choice over `operator[]`. C++11 provides `std::begin()`, `std::end()` functions.

As your code uses just `std::vector`, I can't say there is much difference in both codes, however, `operator []` may not operate as you intend to. For example if you use map, `operator[]` will insert an element if not found.

Also, by using `iterator` your code becomes more portable between containers. You can switch containers from `std::vector` to `std::list` or other container freely without changing much if you use iterator such rule doesn't apply to `operator[]`.

edited Jan 17 '13 at 7:38

Sebastian Mach
28.8k 2 74 109

answered Jan 17 '13 at 7:31

sense to me. Since maps don't have to have a numerical key then if I was to change container classes then I would have to modify the loop to accommodate for the map container. With an iterator no matter which container I change it to it will be suitable for the loop. Thanks for the answer :) – CodingMadeEasy Jan 17 '13 at 7:46

It always depends on what you need.

You should use `operator[]` when you **need** direct access to elements in the vector (when you need to index a specific element in the vector). There is nothing wrong in using it over iterators. However, you must decide for yourself which (`operator[]` or iterators) suits best your needs.

Using iterators would enable you to switch to other container types without much change in your code. In other words, using iterators would make your code more generic, and does not depend on a particular type of container.

edited Jan 17 '13 at 7:38

answered Jan 17 '13 at 7:17

Mark Garcia
12.5k 3 36 87

So you're saying that I should use the [] operator instead of an iterator? – CodingMadeEasy Jan 17 '13 at 7:24

1 @CodingMadeEasy It always depends on what you want and what you need. – Mark Garcia Jan 17 '13 at 7:25

Yea that makes sense. I'll just keep working at it and just see which one is the most suitable for each situation – CodingMadeEasy Jan 17 '13 at 7:32

But `operator[]` is just as direct as iterators. Both just give references to elements. Did you mean `when you need to be able to manually index into a container`, e.g. `cont[x] < cont[x-1]`? – Sebastian Mach Jan 17 '13 at 7:41

With a vector iterators do no offer any real advantage. The syntax is uglier, longer to type and harder to read.

Iterating over a vector using iterators is not faster and is not safer (actually if the vector is possibly resized during the iteration using iterators will put you in big troubles).

The idea of having a generic loop that works when you will change later the container type is also mostly nonsense in real cases. Unfortunately the dark side of a strictly typed language without serious typing inference (a bit better now with C++11, however) is that you need to say what is the type of everything at each step. If you change your mind later you will still need to go around and change everything. Moreover different containers have very different trade-offs and changing container type is not something that happens that often.

The only case in which iteration should be kept if possible generic is when writing template code, but that (I hope for you) is not the most frequent case.

The only problem present in your explicit index loop is that `size` returns an unsigned value (a design bug of C++) and comparison between signed and unsigned is dangerous and surprising, so better avoided. If you use a decent compiler with warnings enabled there should be a diagnostic on that.

Note that the solution is not to use an unsiged as the index, because arithmetic between unsigned values is also apparently illogical (it's modulo arithmetic, and `x-1` may be bigger than `x`). You instead should cast the size to an integer before using it. It **may** make some sense to use unsigned sizes and indexes (paying a LOT of attention to every expression you write) only if you're working on a 16 bit C++ implementation (16 bit was the reason for having unsigned values in sizes).

As a typical mistake that unsigned size may introduce consider:

```
void drawPolyline(const std::vecto
{
    for (int i=0; i<points.size()-
```

the value `points.size()-1` will be a huge positive number, making you looping into a segfault. A working solution could be

```
for (int i=1; i<points.size(); i++
    drawLine(points[i - 1], points
```

but I personally prefer to always remove `unsinged` -ness with `int(v.size())` .

The ugliness of using iterators in this case is left as an exercise for the reader.

edited May 23 '17 at 12:17

Community♦
1 1

answered Jan 17 '13 at 7:35

6502
82.5k 12 111 197

2 Would you elaborate why `size()` being unsigned is a design bug? I can't see a single reason how `for(int i = 0; ...)` could be preferable to `for(size_t i; ...)` . I've encountered problems with 32-bit indexing on 64-bit systems. – Angew Jan 17 '13 at 7:41

1 -1: C++ has "serious" type inference. What do you mean? – Sebastian Mach Jan 17 '13 at 7:43

3 virtual -1: `ugly, longer to type, harder to read` -> a) this is POV, b) `for(auto x : container)` ?? – Sebastian Mach Jan 17 '13 at 7:44

2 @6502: Regarding size_t's unsignedness: No, it simply means I haven't heard of it yet. And google is relatively silent on the topic for different searches, pointing me (like you) to one of Alf's answers, which makes sense and sounds plausible, but isn't backed up by citations itself. I am not sure why "never heard of it" is the same as "I disagree" to you; that's a ton of speculation. And no, pure reasoning and deep C++ knowledge is not enough; the C++ standard does not contain such anecdote, neither does logic. – Sebastian Mach Jan 17 '13 at 9:35

2 I mostly agree that unsigned types are unfortunate, but since they're baked into the standard libraries I also don't see good means of avoiding them. An "unsigned type

size is larger than `INT_MAX` then obviously you can't convert it to `int` and the code fails. `long long` would be safer (especially as it's finally standard). I will never create a vector with 2^63 elements but I might with 2^31. – Steve Jessop Jan 17 '13 at 9:53

By writing your client code in terms of iterators you abstract away the container completely.

Consider this code:

```
class ExpressionParser // some gen
{
public:
    template<typename It>
    void parse(It begin, const It 
    {
        using namespace std;
        using namespace std::placel
        for_each(begin, end,
            bind(&ExpressionParser
    }
    // process next char in a stre
    void process_next(char c);
};
```

client code:

```
ExpressionParser p;

std::string expression("SUM(A) FOR
p.parse(expression.begin(), expres

std::istringstream file("expressio
p.parse(std::istringstream<char>(f

char expr[] = "[12a^2 + 13a - 5] w
p.parse(std::begin(expr), std::end
```

Edit: Consider your original code example, implemented with :

```
using namespace std;

vector<int> myIntVector;
// Add some elements to myIntVecto
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

copy(myIntVector.begin(), myIntVec
    std::ostream_iterator<int>(cou
```

edited Jan 17 '13 at 9:43

answered Jan 17 '13 at 9:36

utnapistim
21.4k 2 31 72

Nice example, but the

cplusplus.com suggests that it can't be turned off *in this case* because a special `sentry` object is created to leave it on... Ugh.) So e.g. if your `expr` was in the file `expression.txt`, the second call to `p.parse()` would (perhaps unavoidably) read `witha` from it as a single token. – j_random_hacker Apr 18 '16 at 16:02

The nice thing about iterator is that later on if you wanted to switch your vector to a another STD container. Then the forloop will still work.

answered Jan 17 '13 at 7:18
Caesar
6,728 4 28 59

its a matter of speed. using the iterator accesses the elements faster. a similar question was answered here:

What's faster, iterating an STL vector with vector::iterator or with at()?

Edit: speed of access varies with each cpu and compiler

edited May 23 '17 at 10:31
Community♦
1 1

answered Jan 17 '13 at 7:18
Nicolas Brown
1,039 1 7 15

But in that post you just showed me it said that indexing is much faster :/ – CodingMadeEasy Jan 17 '13 at 7:21

my bad, i read the results from the benchmark underneath that one. I've read elsewhere where it states using teh iterator is faster than indexing. I'm going to try it myself. – Nicolas Brown Jan 17 '13 at 7:23

Alright well thanks and let me know the results that you get – CodingMadeEasy Jan 17 '13 at 7:30

2 `at()` is different because it range checks and conditionally throws. There's no consistent performance benefit for iterators over indexing or vice versa - anything you

architectures etc. – Tony Delroy Jan 17 '13 at 7:31

i agree with @TonyD. In the link i posted, one person is saying indexing is faster while another is saying using the iterator is faster. I tried the code posted; the loop with the iterator took 40 seconds while the one using indexing only took 4. It's only a slight speed difference tho – Nicolas Brown Jan 17 '13 at 7:41