IN2029: Programming in C++

Session 4 – Iterators and associative containers

Vahid Rafe

Department of Computer Science
City, University of London

Autumn term, 2025

Containers

- A container (called a “collection” in Java) is an object holding a collection of values.
  - A sequential container (*e.g.* **`vector`**, **`deque`**, **`list`**) holds a collection of values arranged in order.
  - An associative container (*e.g.* **`map`**) holds values that can be efficiently looked up by key.
- An iterator is an object representing a position within a container.
- Functions in the standard **`<algorithm>`** library tend to operate on iterators rather than containers (more next week).

Outline

We will introduce iterators by going through three versions of a program, all doing the same thing:

1. operating on a vector, using indices (numbers) to refer to positions in the vector
   - correct but inefficient, due to the limitations of vectors.
2. still operating on a vector, but using iterators to refer to positions in the vector
   - equivalent to version 1, but now easier to switch to a different container.
3. operating on a linked list, but iterators to refer to positions
   - same code as version 2, but now more efficient because it is operating on a linked list.

Printing out a vector

One of the exercises from last week was to write a function to print a vector. Here is a solution:

```
// write vector to an output stream
void write_vector(ostream &out, const vector<double> &v) {
    const auto n = v.size();
    out << "vector:";
    using vec_size = vector<double>::size_type;
    for (vec_size i = 0; i < n; ++i)
        out << ' ' << v[i];
    out << '\n';
}
```

Deleting all the zeroes from a vector

Task: write a function that deletes all the zeroes from a vector.

- The signature of our function will be

```
void delete_zeroes(vector<double> &v)
```

- To delete elements, we shall use the member function **`erase`**:

```
v.erase(v.cbegin() + i);
```

- The argument of **`erase`** is an iterator, an object representing a position in the vector container.
- Here **`v.cbegin()`** represents the position at the start of the vector, and **`v.cbegin()+ i`** represents the position **`i`** places further on.

The deletion function

```
// remove zero values from a vector
void delete_zeroes(vector<double> &v) {
    using vec_size = vector<double>::size_type;
    vec_size i = 0;
    while (i < v.size())
        if (v[i] == 0)
            v.erase(v.cbegin() + i);
        else
            ++i;
}
```

- **Caution**

We cannot optimize this by putting `v.size()` in a variable, because `erase` changes the size of the vector.

A performance problem

- **`erase`** can be expensive for vectors if the index is not at the end of the vector.
- Action of **`erase`** at position **`i`**:

positions before i	0	positions after i

$\Downarrow$

positions before i	positions after i

- All elements after the erased position must be moved down.
- As a result, **`delete_zeroes`** can take $O(n^2)$ time.
- We could fix this particular function with a different algorithm, but instead we'll switch to a different data structure (a linked list) for which **`erase`** is efficient.
- But first, we generalize from indices (which work with vectors) to iterators (which work with any container).

Random versus sequential access

To make our code work with a different data structure, we need to think about what primitives we really need. The fewer we require, the more options we will have.

- Vectors provide a powerful indexing operation `v[i]` (random access).
- For the two functions we want to write, it is sufficient to step through the elements one by one from the start of the container. (sequential access).
- The operations we need are:
  - access the current element
  - move to the next element
  - test whether we've reached the end of the container
- There are many structures (*e.g.* linked lists) that will support these operations but not indexing.

Iterators

- An iterator is an object representing a position in a container (*e.g.* a vector).
- Iterators for different container types are typically implemented differently.
- For any iterator `it`, the following operations are supported:
  - `*it` is a reference to the element currently pointed to by the iterator.
  - `++it` moves the iterator on to the next position.
  - `it == it2` or `it != it2`, where `it2` is another iterator for the same container, tests whether the two iterators are at the same position or not.
- Iterators on some container types support more operations (*e.g.* the `+` operation on vector iterators that we've already seen).

Iterator types for a container

Like every standard container, `vector<double>` defines two iterator types:

- `vector<double>::iterator`
- `vector<double>::const_iterator`

The difference is that for `const_iterator`, `*it` is a `const` reference, so that

```
*it = 1.2;
```

is only allowed for `iterator`.

Advice (*Effective Modern C++* Item 13)

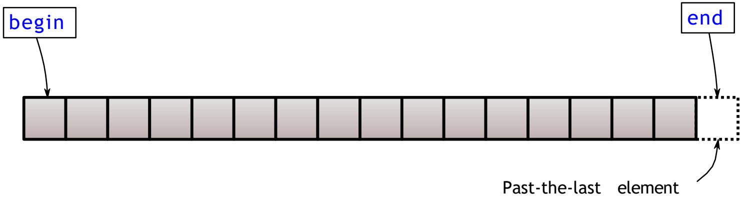
Prefer `const_iterator`s to `iterator`s (where possible).

(for the same reasons we use `const` wherever possible)

Standard iterators for a container

Every standard container provides the following member functions:

- begin() returns an **`iterator`** pointing at the first element of the container.
- end() returns an **`iterator`** pointing just after the last element of the container.
- cbegin() returns a **`const_iterator`** pointing at the first element of the container. (C++14)
- cend() returns a **`const_iterator`** pointing just after the last element of the container. (C++14)



Looping though a container

You can't use `*` on `end()` or `cend()`, but you can use `==`.

The standard idiom, when not changing the elements:

```
    for (auto it = v.cbegin(); it != v.cend(); ++it)
        // do something with *it
```

(Here the type of `it` is `vector<double>::const_iterator`.)

When changing the elements:

```
    for (auto it = v.begin(); it != v.end(); ++it)
        *it += 3;
```

(Here the type of `it` is `vector<double>::iterator`.)

Writing a vector using an iterator

We can rewrite our function to use sequential access via an iterator instead of random access:

```
// write vector to an output stream
void write_vector(ostream &out, const vector<double> &v) {
    out << "vector:";
    for (auto it = v.cbegin(); it != v.cend(); ++it)
        out << ' ' << *it;
    out << '\n';
}
```

Deleting zeroes using an iterator

```
// remove zero values from a vector
void delete_zeroes(vector<double> &v) {
    auto it = v.cbegin();
    while (it != v.cend())
        if (*it == 0)
            it = v.erase(it);
        else
            ++it;
}
```

- **Caution**

  We need to update `it` with the iterator returned by `erase`, because `erase` invalidates `it` and all later iterators.

This still has the same performance problem, but now it is easier to switch containers.

Standard sequential containers

vector implemented as an extensible array (like Java's `ArrayList`): efficiently supports **`size`**, **`push_back`**, **`back`**, **`pop_back`** and indexing.

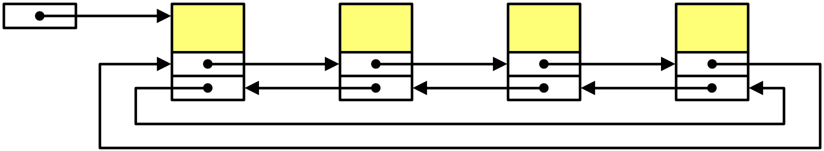
deque implemented as an extensible array with wraparound (like Java's `ArrayDeque`): efficiently supports **`size`**, **`push_back`**, **`back`**, **`pop_back`**, **`push_front`**, **`front`**, **`pop_front`** and indexing.

list implemented as a doubly-linked list (like Java's `LinkedList`): efficiently supports **`size`**, **`push_back`**, **`back`**, **`pop_back`**, **`push_front`**, **`front`**, **`pop_front`**, **`erase`** and **`insert`**, but not indexing.

Unlike in Java, there is no subtyping involved: the different containers just use the same names for their member functions, which makes it easy to switch containers.

Implementation of `list`

The standard **`list`** type is a doubly linked list (see session 5 of Data Structures and Algorithms), e.g.



This structure supports inserting and removing elements at any point in constant time. However, it cannot support indexing efficiently.

Writing a list using an iterator

To adapt **write_vector** to **list**, we just change **vector** to **list**:

```
// write list to an output stream
void write_list(ostream &out, const list<double> &v) {
    out << "list:";
    for (auto it = v.cbegin(); it != v.cend(); ++it)
        out << ' ' << *it;
    out << '\n';
}
```

This works because all the operations used are common to the two containers.

Deleting all the zeroes from a list

Similarly, we can adapt **delete_zeroes** to lists:

```
// remove zero values from a list
void delete_zeroes(list<double> &v) {
    auto it = v.cbegin();
    while (it != v.cend())
        if (*it == 0)
            it = v.erase(it);
        else
            ++it;
}
```

However, because we are now using lists, **erase** is fast, and **delete_zeroes** takes time $O(n)$.

Shorthand: range-based `for`

Looping through the whole container is common, C++11 introduced an abbreviation for it:

- not changing the elements (for small types):

```
for (auto x : v)
    // do something with x
```

- not changing the elements, but avoiding a copy:

```
for (const auto &x : v)
    // do something with x
```

- changing the elements:

```
for (auto &x : v)
    x += 3;
```

Copy of the value of an element

A range-based **for** loop with a value variable

```
for (auto x : v) {
    // body looking at x
}
```

is equivalent to

```
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    auto x = *it;
    // body looking at x
}
```

Use this when elements are small, and you don’t want to change them.

Constant reference to an element

A range-based **for** loop with a **const** reference

```
for (const auto &x : v) {
    // body looking at x
}
```

is equivalent to

```
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    const auto &x = *it;
    // body looking at x
}
```

Use this when elements are big, and you don't want to change them.

Reference to an element

A range-based **for** loop with a non-const reference

```
for (auto &x : v) {
    // body updating x
}
```

is equivalent to

```
for (it = v.begin(); it != v.end(); ++it) {
    auto &x = *it;
    // body updating x
}
```

Use this when you want to change elements of the container.

Writing a vector using range-based `for`

The `write_list` function using a range-based `for`:

```
// write list to an output stream
void write_list(ostream &out, const list<double> &v) {
    out << "list:";
    for (auto x : v)
        out << ' ' << x;
    out << '\n';
}
```

Updating a vector using range-based `for`

If we want to modify the container:

```
// add one to each element
void increment_list(list<double> &v) {
    for (auto &x : v)
        x += 1;
}
```

The `&`s here avoid copying, but they are both also crucial to ensuring that elements of the container get updated:

- `v` is an alias for the list passed as an argument.
- `x` is an alias for an element of the list.

The `map` container

`map<K, V>` is an associative container (like Java's `TreeMap`). Maps can be efficiently indexed like vectors, but by type `K` (instead of `int`):

```
map<string, int> days;
days["January"] = 31;
days["February"] = 28;
days["March"] = 31;
...
string n = "October";
cout << n << " has " << days[n] << " days\n";
```

- **Caution**

The expression *m*`[`*k*`]` creates an entry for *k* if none exists in *m* already.

Notional view of a `map`

A `map` stores an association of keys with values, in key order:

"April"	30
"August"	31
"December"	31
"February"	28
"January"	31
"July"	31
"June"	30
"March"	31
"May"	31
"November"	30
"October"	31
"September"	30

- Internally, `map` uses a form of balanced search tree (see sessions 7 and 8 of Data Structures and Algorithms).
- The `map` provides fast access by key.
- Iterators on the `map` provide sequential access to pairs of keys and values, in key order.

Counting occurrences of words

Task: read in words, and then print each unique word with the number of times it occurs in the input.

We shall use a map from words that we encounter to `int`:

```
map<string, int> count;
```

- This declares a map called `count`, and performs default initialization (as an empty map).
- In our program, the `int` will always be 1 or more – words that don't occur in the input won't have entries in the map.

Rest of the program:

1. read words, updating `count`
2. print contents of `count`

Computing the counts

The core of the the program is deceptively concise:

```
    // read input words, updating their counts
    string w;
    while (cin >> w)
        ++count[w];
```

The loop is familiar, but there's a lot going on in the last line:

- If the map has an entry for `w`, then `count[w]` is a reference to it.
- If not, an entry for `w` is created, given default initialization (for `int`, set to 0), and then `count[w]` is a reference to the new entry.
- In either case, the entry is then increased by `++`.

At the end of this loop, `count[w]` is the number of times that `w` occurred in the input, for each `w` that occurred.

Printing the counts

It remains to go though the map, printing words and associated counts:

```
    // write each word and its number of occurrences
    for (const auto &p : count)
        cout << p.first << '\t' << p.second << '\n';
```

- The loop goes through the map in key order.
- At each step `it` refers to a `pair<const string, int>`, an object with members
  - `first`, a `const` of type `string`
  - `second` of type `int`
- We use `&`, to avoid copying the pair.
- We use `const`, to declare that we will not change the pair.

Iterator form

An equivalent form of that loop using iterators would be

```
    for (auto it = count.cbegin(); it != count.cend(); ++it)
        cout << it->first << '\t' << it->second << '\n';
```

- We use `cbegin()` to get the `const_iterator`, because we are only looking at the values, not changing them.
- `it` has type `map<string, int>::const_iterator`.
- `*it` has type `pair<const string, int>`, with data members `first` and `second`.
- `it->first` is short for `(*it).first`