Processing Containers

This section discusses how to process several containers and the same time using the example of separate containers of indices.

WE'LL COVER THE FOLLOWING

Processing Several Containers at a Time

When using parallel algorithms, you might sometimes want to access other containers.

For example, you might want to execute `for_each` on two containers.

The main technique is to get the index of the element currently being processed. Then you can use that index to access other containers (assuming the containers are of the same size).

We can do it in a few ways:

- by using a separate container of indices
- by using zip iterators/wrappers

Let's have a look at these techniques.

Separate container of indices

```
void Process(int a, int b) { }

int main()
{
  std::vector<int> v(100);
```

```
  std::vector<int> w(100);
  std::iota(v.begin(), v.end(), 0);
  std::iota(w.begin(), w.end(), 0);

  std::vector<size_t> indexes(v.size());
  std::iota(indexes.begin(), indexes.end(), 0);

  std::for_each(std::execution::par, indexes.begin(), indexes.end(),
                [&v, &w](size_t& id) {
                  Process(v[id], w[id]);
                });
}
```

As you can see in the above code, a separate vector of indices had to be generated. Since the order of execution is not specified, we cannot iterate using a simple `i` variable.

Zip Iterators

A more elegant technique is to use zip iterators:

```
#include <algorithm>
#include <execution>
#include <iostream>
#include <numeric>
#include <vector>

template <typename T, typename U>
class vec_zip_iterator {
  using it_type_first = typename std::vector<T>::iterator;
  using it_type_second = typename std::vector<U>::iterator;
  it_type_first it1;
  it_type_second it2;

  public:
  vec_zip_iterator() = default;

  vec_zip_iterator(it_type_first iterator1, it_type_second iterator2)
    : it1{ iterator1 }, it2{ iterator2 }
  {}

  vec_zip_iterator& operator++() {
    ++it1;
    ++it2;
    return *this;
  }

  friend vec_zip_iterator operator+(const vec_zip_iterator& a, long int n) {
    return vec_zip_iterator(a.it1 + n, a.it2 + n);
  }

  friend long int operator-(const vec_zip_iterator& a, const vec_zip_iterator& b) {
    return a.it1 - b.it1;
  }

  bool operator!=(const vec_zip_iterator& o) const {
```

```
    return it1 != o.it1 && it2 != o.it2;
  }


  bool operator==(const vec_zip_iterator& o) const {
    return !operator!=(o);
  }

  std::pair<T&, U&> operator*() const {
    return { *it1, *it2 };
  }
};

namespace std {
  template <typename T, typename U>
  struct iterator_traits<vec_zip_iterator<T, U>> {
    using iterator_category = std::random_access_iterator_tag;
    using value_type = typename std::pair<T, U>;
    using difference_type = long int;
  };
}

template <typename T, typename U>
class vec_zipper {
  using vec_type_first = typename std::vector<T>;
  using vec_type_second = typename std::vector<U>;
  vec_type_first &vec1;
  vec_type_second &vec2;

  public:
  vec_zipper(vec_type_first &va, vec_type_second &vb)
    : vec1{ va }, vec2{ vb }
  {}

  vec_zip_iterator<T, U> begin() const {
    return { std::begin(vec1), std::begin(vec2) };
  }
  vec_zip_iterator<T, U> end() const {
    return { std::end(vec1), std::end(vec2) };
  }
};

void Process(int, int )
{

}

int main()
{
  std::vector<int> v(100);
  std::vector<int> w(100);
  std::iota(v.begin(), v.end(), 0);
  std::iota(w.begin(), w.end(), 0);

  // elegant solution with zips:
  vec_zipper<int, int> zipped{ v, w };
  std::for_each(std::execution::seq, zipped.begin(), zipped.end(),
                [](const std::pair<int&, int&>& twoElements) {
                  Process(twoElements.first, twoElements.second);
                }
               );

  return 0;
```

```
}
```

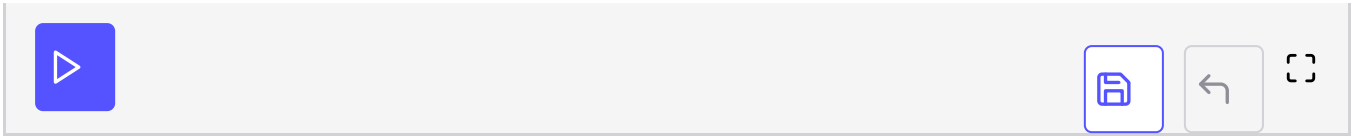
This is a more elegant approach as we combine two containers into a single sequence and then iterate at once. The example uses a custom implementation of a ZipIterator that works only with `std::vector`. You can improve the code and make it more general or use third-party zip iterators (like boost).

Try It Out

Here is the executable code for both the methods discussed above:

```
#include <algorithm>
#include <execution>
#include <iostream>
#include <numeric>
#include <vector>

template <typename T, typename U>
class vec_zip_iterator {
    using it_type_first = typename std::vector<T>::iterator;
    using it_type_second = typename std::vector<U>::iterator;
    it_type_first it1;
    it_type_second it2;

public:
    vec_zip_iterator() = default;

    vec_zip_iterator(it_type_first iterator1, it_type_second iterator2)
        : it1{ iterator1 }, it2{ iterator2 }
    {}

    vec_zip_iterator& operator++() {
        ++it1;
        ++it2;
        return *this;
    }

    friend vec_zip_iterator operator+(const vec_zip_iterator& a, long int n) {
        return vec_zip_iterator(a.it1 + n, a.it2 + n);
    }

    friend long int operator-(const vec_zip_iterator& a, const vec_zip_iterator& b) {
        return a.it1 - b.it1;
    }

    bool operator!=(const vec_zip_iterator& o) const {
        return it1 != o.it1 && it2 != o.it2;
    }

    bool operator==(const vec_zip_iterator& o) const {
        return !operator!=(o);
    }

    std::pair<T&, U&> operator*() const {
```

```
        return { *it1, *it2 };
    }
};


namespace std {
    template <typename T, typename U>
    struct iterator_traits<vec_zip_iterator<T, U>> {
        using iterator_category = std::random_access_iterator_tag;
        using value_type = typename std::pair<T, U>;
        using difference_type = long int;
    };
}

template <typename T, typename U>
class vec_zipper {
    using vec_type_first = typename std::vector<T>;
    using vec_type_second = typename std::vector<U>;
    vec_type_first &vec1;
    vec_type_second &vec2;

public:
    vec_zipper(vec_type_first &va, vec_type_second &vb)
        : vec1{ va }, vec2{ vb }
    {}

    vec_zip_iterator<T, U> begin() const {
        return { std::begin(vec1), std::begin(vec2) };
    }
    vec_zip_iterator<T, U> end() const {
        return { std::end(vec1), std::end(vec2) };
    }
};

void Process(int, int ) {

}

int main() {
    std::vector<int> v(100);
    std::vector<int> w(100);
    std::iota(v.begin(), v.end(), 0);
    std::iota(w.begin(), w.end(), 0);

    std::vector<size_t> indexes(v.size());
    std::iota(indexes.begin(), indexes.end(), 0);

    std::for_each(std::execution::seq, indexes.begin(), indexes.end(),
        [&v, &w](size_t& id) {
            Process(v[id], w[id]);
        }
    );

    // elegant solution with zips:
    vec_zipper<int, int> zipped{ v, w };
    std::for_each(std::execution::seq, zipped.begin(), zipped.end(),
        [](const std::pair<int&, int&>& twoElements) {
            Process(twoElements.first, twoElements.second);
        }
    );

    return 0;
}
```



The next lesson will discuss some common erroneous techniques which should be avoided.