Algorithms in C++

Lectures on Modern Scientific Programming 2016

Dániel Berényi
Wigner GPU Lab

Foreword – Functions and Lambdas

First some introduction to the new C++11, C++14 syntax around functions that we'll use on *all* sessions today

Foreword – Functions and Lambdas

A classic function (compound statement):

```
double sq(double x){ return x*x; }
```

Foreword – Functions and Lambdas

A classic function (compound statement):

```
double sq(double x){ return x*x; }
```

An equivalent lambda expression (since C++11):

```
[](double x)->double{ return x*x; }
```

Foreword – Functions and Lambdas

A classic function (compound statement):

```
double sq(double x){ return x*x; }
```

An equivalent lambda expression:

```
// the value of this expression is 25.0:
([](double x)->double{ return x*x; })(5.0)
```

Foreword – Functions and Lambdas

A classic function (compound statement):

```
double sq(double x){ return x*x; }
```

An equivalent lambda expression, now bound to a name:

```
auto    sq = [](double x)->double{ return x*x; };
```

Foreword – Functions and Lambdas

A lambda function is just a syntactic sugar to an instance of a struct with an overloaded operator():

```
[](double x)->double{ return x*x; }
```

```
struct ...
{
   double operator()(double x) const { return x*x; }
};
```

Foreword – Functions and Lambdas

C++11 introduced return type deduction, so we can write:

```
[](double x)->double{ return x*x; }


// the same lambda, return type is automatically calculated!
[](double x)        { return x*x; }
```

Foreword – Functions and Lambdas

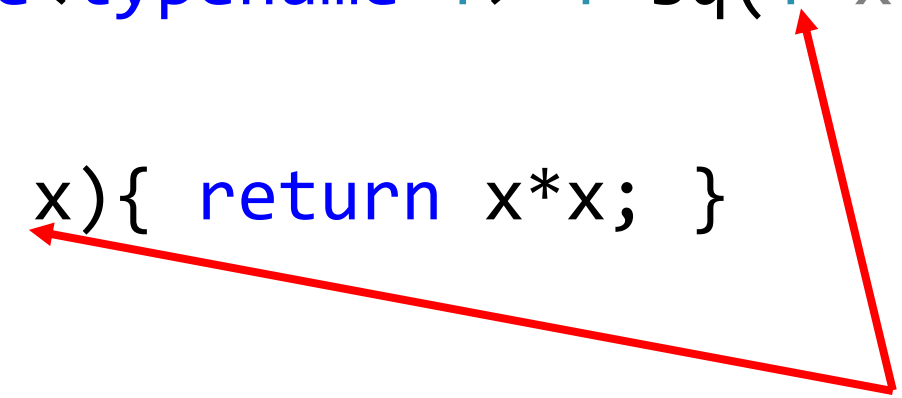
C++14 introduced generic lambdas that act as templated function objects:

```
template<typename T> T sq(T x){ return x*x; }

[](auto x){ return x*x; }
```

Foreword – Functions and Lambdas

C++14 introduced generic lambdas that act as templated function objects:

```
template<typename T> T sq(T x){ return x*x; }

[](auto x){ return x*x; }
```



These work with any type,
that can be multiplied by the operator*

Foreword – Functions and Lambdas

Again, generic lambdas are just syntactic sugar for structs with templated operator()s

```
[](auto x){ return x*x; }
```

```
struct ...
{
   template<typename T>
   T operator()(T x) const { return x*x; }
};
```

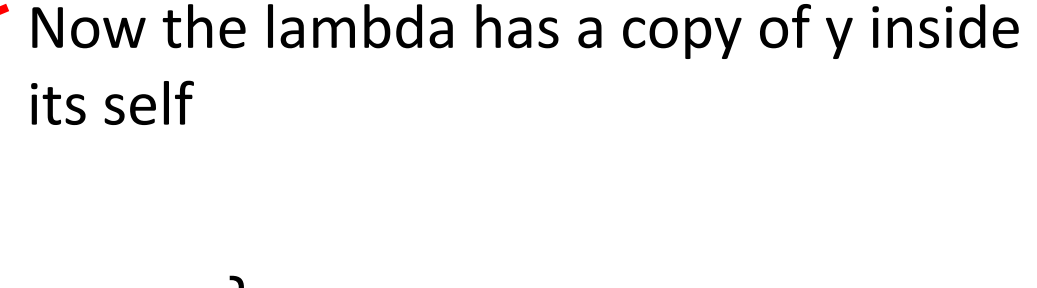
Foreword – Functions and Lambdas

Lambdas can capture variables from the enclosing scopes:

```
{
    double y = 2.0;
    auto f = [=](double x){ return x+y; };
    double z = f(5.0); // z == 7.0
}
```

Foreword – Functions and Lambdas

Lambdas can capture variables from the enclosing scopes:




```
{
    double y = 2.0;
    auto f = [=](double x){ return x+y; };
    double z = f(5.0); // z == 7.0
}
```

Foreword – Functions and Lambdas

Lambdas can capture variables from the enclosing scopes:

```
{
   double y = 2.0;
   auto f = [=](double x)
            {
              return x+y;
            };
   double z = f(5.0);
}
```

The equivalent struct is:

```
struct ...
{
   double y;
   ...(double y_in):y(y_in){}

   double operator()(double x)
          const { return x+y; }
};
```

Foreword – Functions and Lambdas

Lambdas can capture variables from the enclosing scopes:

You can also capture by reference too:

```
{
    double y = 2.0;
    auto f = [&](double x)
             {
               return x+y;
             };
    double z = f(5.0);
}
```

```
struct ...
{
    double& y;
    ...(double& y_in):y(y_in){}

    double operator()(double x)
           const { return x+y; }
};
```

Algorithms

Outline

- Containers and algorithms are the basic building blocks of structured programs

- Using these constructs reduces the amount of potential mistakes

The problem with C

- Consider a the following task:

  Given an array, calculate the sum of squares of its elements.

The problem with C

- Consider a traditional for loop solving the task:

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```

The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```

The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```

Does the type of the indexing variable is appropriate for the desired range?

The problem with C

- How many potential mistakes can you make?

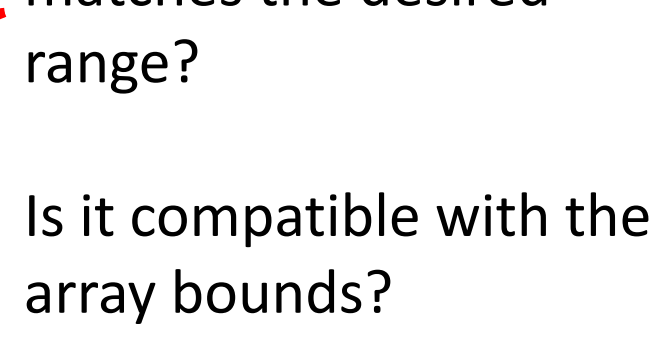
```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
    sqsum += arr[i]*arr[i];
}
```

Does the first index really marks the beginning of the desired range?

The problem with C

- How many potential mistakes can you make?

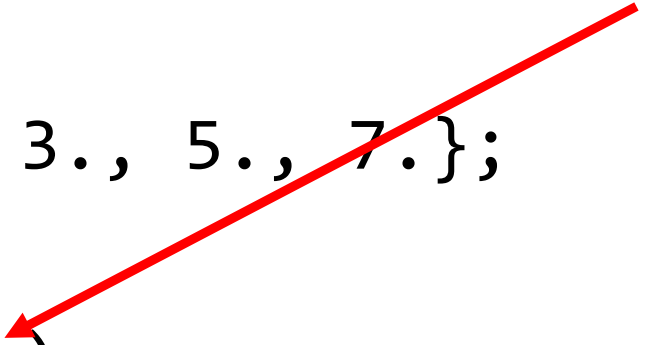
```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```



The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```



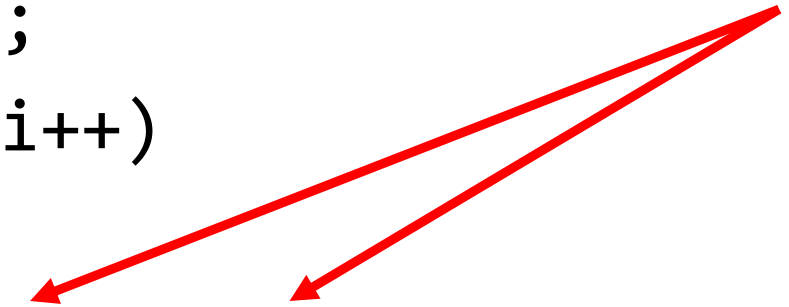
Does the index stepping is what we would like to achieve?

Lets not miss elements accidentally…

The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```

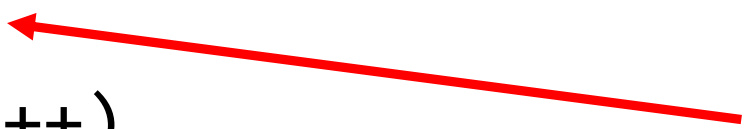


Are we using the right index *everywhere* where we are indexing?!

The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```



Are the loop preconditions match what we would like to do?

Does the type of the variable match our intent?

The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
    sqsum += arr[i]*arr[i];
}
```

Are the loop postconditions fine?

ie.: the “result” of the loop is in `sqsum`.

The problem with C

- How many potential mistakes can you make?

```
double arr[4] = {1., 3., 5., 7.};
double sqsum = 0.0;
for(int i=0; i<4; i++)
{
     sqsum += arr[i]*arr[i];
}
```

This list of mistakes was not exhaustive!!!

Why is it safer with algorithms?

- How does this look like in C++?

```
std::array<double, 4> arr = {1., 3., 5., 7.};

auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [](double last, double current)
                { return last + current*current; } );
```

Why is it safer with algorithms?

- Let's analyze!

Range selection is explicit, but the array size is not!

No indices appear.

Here the whole array will be iterated over.

```
auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [](double last, double current)
                { return last + current*current; } );
```

Why is it safer with algorithms?

- Let's analyze!

```
auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [](double last, double current)
                { return last + current*current; } );
```

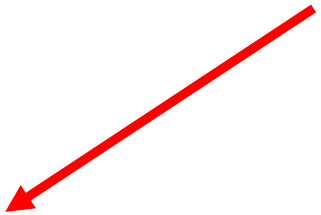
Starting value of summation is given, but it lives inside the function, not in the enclosing scope!

Why is it safer with algorithms?

- Let's analyze!

```
auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [](double last, double current)
                { return last + current*current; } );
```

The operation to be done at each step is explicitly separated from the array, only deals with actual values!



Comparison of algorithms and loops

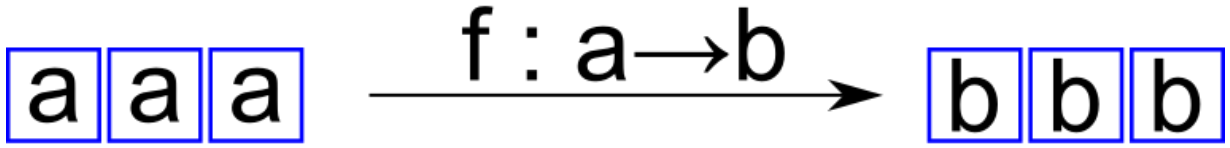
- There is no single silver bullet, but different tools with different compromises:

Loops	Algorithms
Highest flexibility of iteration	More limited forms of iteration
Many possible mistakes to make	Much less room for errors
You can do ”anything” with them	Solve the most common 70-80% of your tasks
You’re responsible for multiple things including safety, performance	The algorithm takes a large part of the responsibility from you
Heavily tied to arrays and indexibles	Works with more general structures
Does not reflect intent clearly	More clearly reflects intent

Theoretical foundations

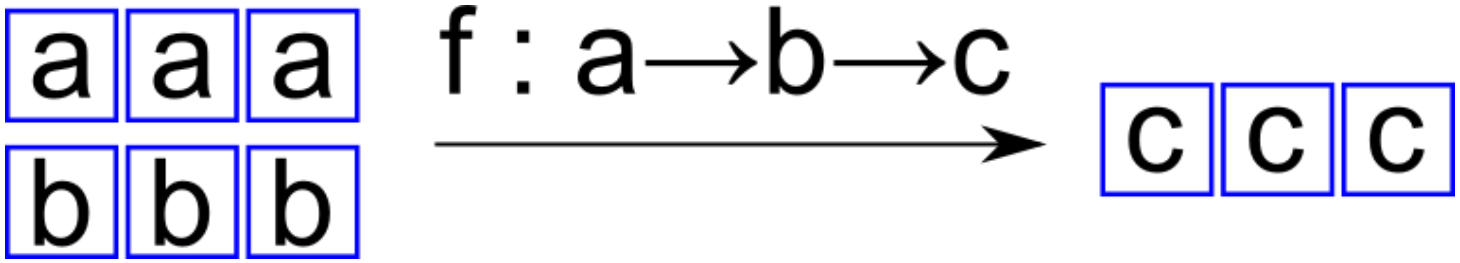
- Algorithms does not come from thin air
- They have well defined roots in functional programming and category theory (even if historically they were discovered in alternative ways).
- There are three extremely common algorithms that worth reviewing and recognizing in every context possible.

Theoretical foundations: Map



- Map is the most fundamental algorithm
- Take a structure of elements and apply a single argument function to each and every element independently!
- Note that the types of the elements might be different.

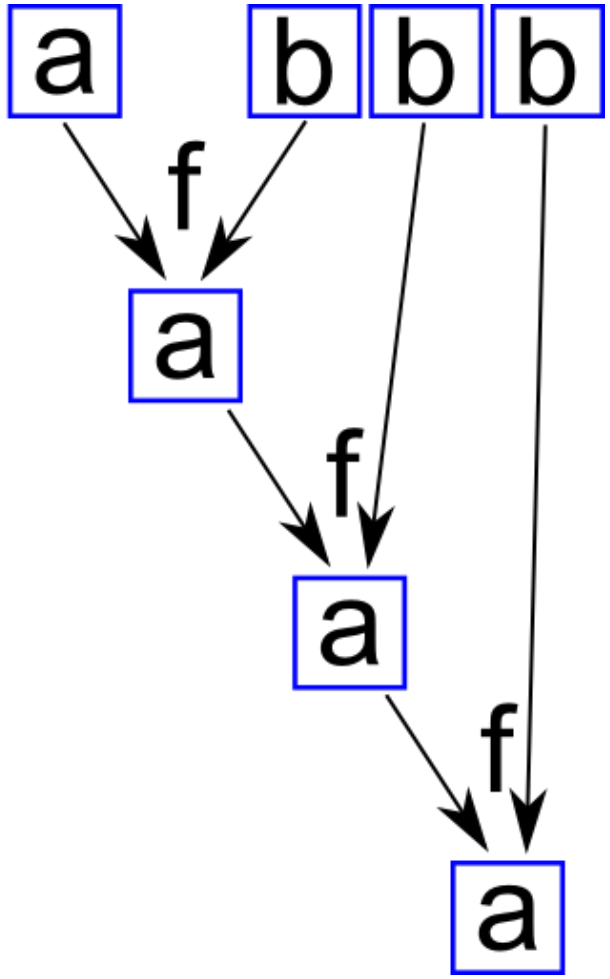
Theoretical foundations: Zip



- Zip is a slight generalization of Map
- Takes two structures of the same kind, but potentially different type of elements and apply a two argument function to pairs of elements at the same location independently!

Theoretical foundations: Fold

$$f : a \rightarrow b \rightarrow a$$



- Fold is the basic construct for accumulations and reductions

- Take a structure
  a starting element
  and a two argument function
  and apply the function repeatedly for each element and the previous result.

C++ dictionary

- The analogue of Map is called `std::transform`
  However, if the result is `void`, the proper analogue of Map is `std::for_each`.

- The analogue of Zip is also `std::transform` (but has different arguments)

- The analogue of Fold is `std::accumulate`.

Simple use cases

- Lets try to recognize the algorithms in simple cases!

Simple use cases

- Task: multiply some data by a scalar

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [](double x){ return 2. * x; });
```

Simple use cases

- Task: multiply some data by a scalar:

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [](double x){ return 2. * x; });
```

Simple use cases

- Task: multiply some data by a scalar:

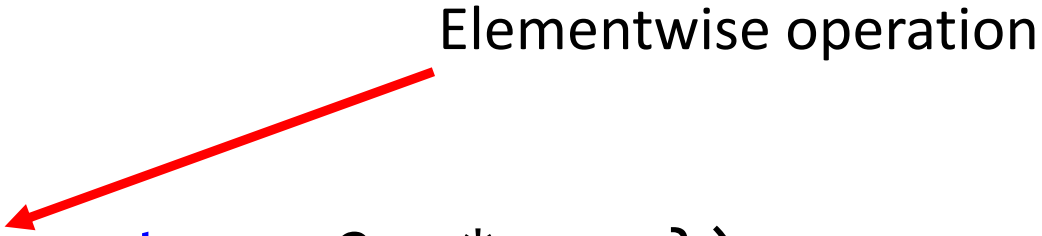
```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [](double x){ return 2. * x; });
```

Start of output range, length is deduced from input range.

Simple use cases

- Task: multiply some data by a scalar:

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [](double x){ return 2. * x; });
```



Simple use cases

- Task: divide some data by a scalar:

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [](double x){ return x / 10.; });
```

Simple use cases

- Task: multiply by a scalar from the enclosing scope:

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
double factor = 137.;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [=](double x){ return factor * x; });
```

Simple use cases

- Task: multiply by a scalar from the enclosing scope:

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::array<double, 4> res;
double factor = 137.;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [=](double x){ return factor * x; });
```

The value should be captured by the lambda to be used!

Simple use cases

- Task: change the type of the data (convert to string):

```
std::array<double,      4> arr = {1., 3., 5., 7.};
std::array<std::string, 4> res;
std::transform( arr.cbegin(),
                arr.cend(),
                res.begin(),
                [](double x)
                {
                    return std::to_string(x);
                });
```

Simple use cases

- Task: print all elements to the standard output:

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::for_each(arr.cbegin(),
              arr.cend(),
              [](double x)
              {
                  std::cout << x << " ";
              });
```

Simple use cases

- Now turn to Zips...

Simple use cases

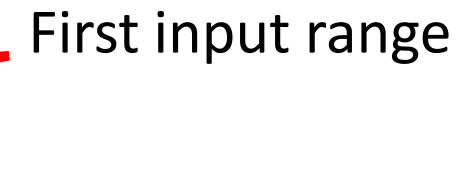
- Task: add two sets of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;
std::transform(arr1.cbegin(),
               arr1.cend(),
               arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x+y; });
```

Simple use cases

- Task: add two sets of numbers:

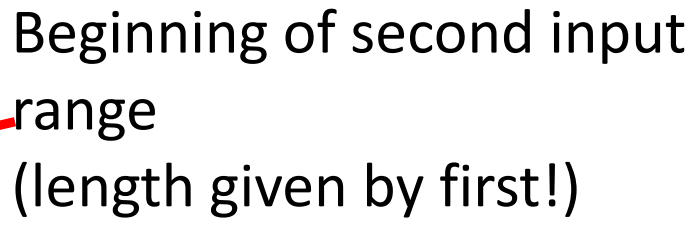
```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;
std::transform(arr1.cbegin(),
               arr1.cend(),
               arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x+y; });
```



Simple use cases

- Task: add two sets of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;
std::transform(arr1.cbegin(),
               arr1.cend(),
               arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x+y; });
```



Simple use cases

- Task: add two sets of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;
std::transform(arr1.cbegin(),
               arr1.cend(),
               arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x+y; });
```

Beginning of output
(length given by first range!)

Simple use cases

- Task: add two sets of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;
std::transform(arr1.cbegin(),
               arr1.cend(),
               arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x+y; });
```

Elementwise operation

Simple use cases

- Task: squared difference of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;
std::transform(arr1.cbegin(),
               arr1.cend(),
               arr2.cbegin(),
               res.begin(),
               [](double x, double y)
               { return (x-y)*(x-y); });
```

Simple use cases

- Task: create complex number from angle and radius:

```
std::array<double, 4> arr_r   = {1.,  3.,   5.,  7. };
std::array<double, 4> arr_phi = {0.5, 0.25, 0.1, 0.9};
std::array<std::complex<double>, 4> res;
std::transform(arr_r.cbegin(),
               arr_r.cend(),
               arr_phi.cbegin(),
               res.begin(),
               [](double r, double phi)
{
    return std::complex<double>(r*cos(phi), r*sin(phi));
});
```

Simple use cases

- Finally Folds...

Simple use cases

- Task: sum numbers (like in the very first example):

```
std::array<double, 4> arr = {1., 3., 5., 7.};

auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [](double last, double current)
                { return last + current; } );
```

Simple use cases

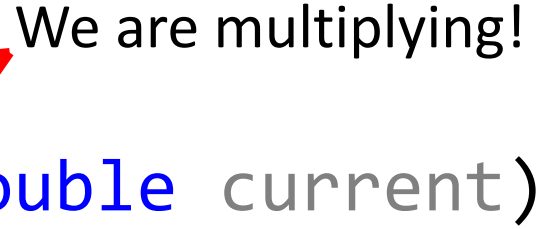
- Task: multiply numbers:

```
std::array<double, 4> arr = {1., 3., 5., 7.};

auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                1.0,
                [](double last, double current)
                { return last * current; } );
```

Simple use cases

- Task: multiply numbers:

```
std::array<double, 4> arr = {1., 3., 5., 7.};

auto res = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                1.0,
                [](double last, double current)
                { return last * current; } );
```



Simple use cases

- Task: calculate average, then standard deviation:

```
std::array<double, 4> arr = {1., 3., 5., 7.};

auto sum = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [](double last, double current)
                { return last + current; } );
auto mean = sum / ( (double)arr.size() );
```

Simple use cases

- Task: calculate average, then standard deviation:

```
auto mean = sum / ( (double)arr.size() );
auto sum2 = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                0.0,
                [=](double last, double current)
                {
                return last + (current - mean)*
                              (current - mean); } );
auto sd = std::sqrt(sum2 / ( arr.size() - 1.0 ));
```

Simple use cases

- Task: concatenate a collection of strings:

```
std::array<std::string, 4> arr_s = {"a1", "b2", "c3", "d4" };
auto s = std::accumulate(
                arr.cbegin(),
                arr.cend(),
                std::string(""),
                [](std::string last, std::string current)
                {
                    return last + current;
                } );
```

Simple use cases

- Task: dot product of two set of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;

std::transform(arr1.cbegin(), arr1.cend(), arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x*y; });
auto dot = std::accumulate(res.cbegin(), res.cend(), 0.0,
                           [](double last, double current)
                           { return last + current; });
```

Simple use cases

- Task: dot product of two set of numbers:

```
std::array<double, 4> arr1 = {1.,  3.,  5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<double, 4> res;

std::transform(arr1.cbegin(), arr1.cend(), arr2.cbegin(),
               res.begin(),
               [](double x, double y){ return x*y; });
auto dot = std::accumulate(res.cbegin(), res.cend(), 0.0,
                           [](double last, double current)
                           { return last + current; });
```

This is not so optimal,
we have a temporary, and two passes!

Simple use cases

- Task: dot product of two set of numbers, better solution:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};

auto dot =
std::inner_product(arr1.cbegin(), arr1.cend(), arr2.cbegin(),
                   0.0,
                   [](double x, double y){ return x+y; },
                   [](double x, double y){ return x*y; } );
```

Simple use cases

- Task: dot product of two set of numbers, better solution:

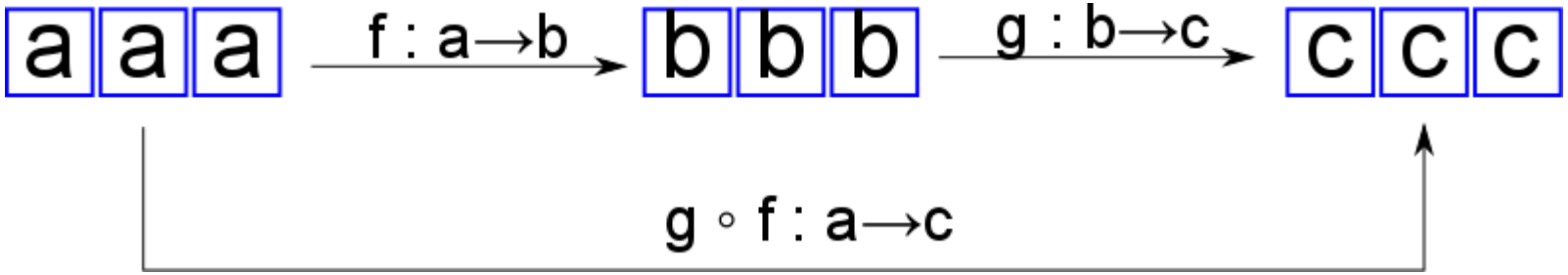
```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};

auto dot =
std::inner_product(arr1.cbegin(), arr1.cend(), arr2.cbegin(),
                   0.0,
                   [](double x, double y){ return x+y; },
                   [](double x, double y){ return x*y; } );
```

This is the operator used for folding

This is the operator applied pair wise!

More involved use cases

- Useful to know the following while planning:



- a composition of maps is the same as a map of compositions:

```
map(g, map(f, v)) = map(g ∘ f, v)
```

More involved use cases

- Task: take an array, multiply each element, convert to string and output it.

```
std::array<double, 4> arr = {1., 3., 5., 7.};
std::for_each(arr.cbegin(),
              arr.cend(),
              [](double x)
              {
                  std::cout << std::to_string(2.*x)
                  << " ";
              });
```

More involved use cases

- Do we always need to introduce a new function for the transformation?

More involved use cases

- Do we always need to introduce a new function for the transformation?

- Enter generic lambdas!

More involved use cases

- Lets create the generic composition function:

```
auto compose =
     [](auto f, auto g)
     {
          return [=](auto x)
                 {
                      return f(g(x));
                 };
     };
```

More involved use cases

- Lets create the generic composition function:

```
auto compose =
     [](auto f, auto g)
     {
          return [=](auto x)
                 {
                     return f(g(x));
                 };
     };
```

`compose` is a function, taking two functions, `f` and `g`.

More involved use cases

- Lets create the generic composition function:

```
auto compose =
     [](auto f, auto g)
     {
          return [=](auto x)
                 {
                     return f(g(x));
                 };
     };
```

It returns a new function, taking one argument, x while capturing f and g.

More involved use cases

- Lets create the generic composition function:

```
auto compose =
     [](auto f, auto g)
     {
          return [=](auto x)
                 {
                     return f(g(x));
                 };
     };
```

When this new function is called,
it will first apply g,
then f to the argument x.

More involved use cases

- Lets create the generic composition function:

```
auto compose =
     [](auto f, auto g)
     {
          return [=](auto x)
                 {
                     return f(g(x));
                 };
     };
```

This construction will work for
any two single-argument functions and
any type of argument,
as long as the types match!

More involved use cases

- Task: take an array, multiply each element, convert to string and output it.

```
std::array<double, 4> arr = {1., 3., 5., 7.};

auto mul_by_two = [](double x){ return 2.*x; };
auto convert    = [](double x){ return std::to_string(x); };
auto printer    = [](std::string x){ std::cout << x << " "; };

std::for_each(arr.cbegin(), arr.cend(),
              compose(printer, compose(convert, mul_by_two)));
```

Problems

- Unfortunately, nesting does not work well
- Consider the dyadic product!

Problems

- Unfortunately, nesting does not work well
- Consider the dyadic product:

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<std::array<double, 4>, 4> res;

std::transform(arr1.cbegin(), arr1.cend(), res.begin(),
[&](double x)
{
   std::transform(arr2.cbegin(), arr2.cend(), ???,
                  [=](double y){ return x*y; });
} );
```

Problems

- Unfortunately, nesting does not work well
- Consider the dyadic product:

We do not have access to the location to write to!

```
std::array<double, 4> arr1 = {1.,  3.,   5.,  7. };
std::array<double, 4> arr2 = {0.5, 0.25, 0.1, 0.9};
std::array<std::array<double, 4>, 4> res;

std::transform(arr1.cbegin(), arr1.cend(), res.begin(),
[&](double x)
{
   std::transform(arr2.cbegin(), arr2.cend(), ???,
                  [=](double y){ return x*y; });
} );
```

Problems

- Possible workaround:

```
auto transform2 = [](auto it_begin, auto it_end, auto f)
{
    using R = decltype(f(*it_begin));
    std::vector<R> res( std::distance(it_begin, it_end) );
    std::transform(it_begin, it_end, res.begin(), f);
    return res;
};
```

Problems

- Possible workaround:

Let the compiler find out the return element type, by "virtually" applying the function to the dereferenced iterator.

```
auto transform2 = [](auto it_begin, auto it_end, auto f)
{
    using R = decltype(f(*it_begin));
    std::vector<R> res( std::distance(it_begin, it_end) );
    std::transform(it_begin, it_end, res.begin(), f);
    return res;
};
```

Problems

- Possible workaround:

Allocate the return vector (we cannot find out the container type from the iterators alone!)
and reserve as many elements as the difference of the two iterators!

```
auto transform2 = [](auto it_begin, auto it_end, auto f)
{
    using R = decltype(f(*it_begin));
    std::vector<R> res( std::distance(it_begin, it_end) );
    std::transform(it_begin, it_end, res.begin(), f);
    return res;
};
```

Problems

- Now we can write the dyadic product as:

```
auto res2 =
     transform2(arr1.cbegin(), arr1.cend(),
     [&](double x)
     {
          return transform2(arr2.cbegin(), arr2.cend(),
          [=](double y){ return x*y; });
     } );
```

Problems

- Now we can write the dyadic product as:

Outer transform iterates through array1, the lambda captures array2 by reference!

```
auto res2 =
    transform2(arr1.cbegin(), arr1.cend(),
    [&](double x)
    {
        return transform2(arr2.cbegin(), arr2.cend(),
        [=](double y){ return x*y; });
    } );
```

Problems

- Now we can write the dyadic product as:

The inner transform operates on the second array, returning one row of the result
the multiplication lambda captures x from the outer iteration.

```
auto res2 =
    transform2(arr1.cbegin(), arr1.cend(),
    [&](double x)
    {
        return transform2(arr2.cbegin(), arr2.cend(),
        [=](double y){ return x*y; });
    } );
```

Problems

- Now we can write the dyadic product as:

```
auto res2;
```

- The result type is:

```
std::vector<std::vector<double>>
```

Problems

- We can write out a nested vectors as nested `for_each` calls:

```
std::for_each(res2.cbegin(), res2.cend(),
              [](std::vector<double> const& row)
              {
                 std::for_each(row.cbegin(), row.cend(),
                               [](double x)
                               { std::cout << x << " "; });
                 std::cout << "\n";
              });
```

Problems

- Rule of thumb:

STL algorithms break down on multiple dimensions, nestings...

You might need a library for these tasks,
take a look at the Boost Multidimensional Array library.

Other algorithms in the STL

- There are many good algorithms available, see the complete list on cppreference!

- Large categories:
  - non-modifying operations: for_each, searching, counting, equality check...
  - modifying operations: transform, copy, fill, remove...
  - partitioning, sorting, searching sorted structures, set operations...
  - numerical and min/max...

- Now we are just picking some quite common ones.

Other algorithms in the STL

- Finding based on criteria
- Task: find the first negative value in a set of numbers:

```
std::array<double, 4> arr = {1., 3., -5., 7. };
auto it = std::find_if(  arr.cbegin(),
                         arr.cend(),
                         [](double x){ return x<0.0; });
std::cout << "location: " << std::distance(arr.cbegin(), it)
          << " value: " << *it << "\n";
```

Other algorithms in the STL

- Finding based on criteria
- Task: find the first negative value in a set of numbers:

```
std::array<double, 4> arr = {1., 3., -5., 7. };
auto it = std::find_if(  arr.cbegin(),
                         arr.cend(),
                         [](double x){ return x<0.0; });
std::cout << "location: " << std::distance(arr.cbegin(), it)
          << " value: " << *it << "\n";
```

You should check if it is not arr.end()!

Other algorithms in the STL

- Finding the maximum peak in a dataset:

```
std::array<double, 4> arr = {1., 3., 55., 7. };

auto it = std::max_element(arr.cbegin(), arr.cend());

std::cout << "location: " << std::distance(arr.cbegin(), it)
          << " value: " << *it << "\n";
```

Other algorithms in the STL

- Finding all values above threshold in a dataset...

- Turns out, you cant simply do that with algorithms efficiently, but it is easy to write a generic algorithm for this task!

Other algorithms in the STL

- Finding all values above threshold in a dataset...

```
auto find_all_if =

[](auto it_begin, auto it_end, auto& result, auto pred)
{
      auto it = it_begin;
      while( it != it_end )
      {
            it = std::find_if(it, it_end, pred);
            if( it != it_end ){ result.push_back(it); ++it; }
      }
};
```

Other algorithms in the STL

- Finding all values above threshold in a dataset...

```
auto find_all_if =

[](auto it_begin, auto it_end, auto& result, auto pred)
{
      auto it = it_begin;
      while( it != it_end )
      {
            it = std::find_if(it, it_end, pred);
            if( it != it_end ){ result.push_back(it); ++it; }
      }
};
```

When we found something, we add it to the list of results, and step forward.

Other algorithms in the STL

- Finding all values above threshold in a dataset:

```
std::vector<double> arr={1., 3., 55., 7., 17., 2., 8., 22.};

std::vector<std::vector<double>::const_iterator> result;
find_all_if(    arr.cbegin(),
                arr.cend(),
                result,
                [](double x){ return x > 4.0; });
```

Other algorithms in the STL

- Copying values.

- Trivial but extremely versatile function!

Other algorithms in the STL

- Copying values:

```
std::vector<double> arr ={1., 3., 5., 7., 9.};

std::vector<double> res;

std::copy(    arr.cbegin(),
              arr.cend(),
              std::back_inserter(res));
```

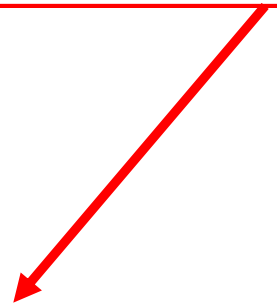
Note! The result vector does not need to be initialized!

Other algorithms in the STL

- Copying values:

```
std::vector<double> arr ={1., 3., 5., 7., 9.};

std::vector<double> res;

std::copy(    arr.cbegin(),
              arr.cend(),
              std::back_inserter(res));
```

The `back_inserter` will gradually fill `res` with values!



Other algorithms in the STL

- Copying values from file:

```
std::vector<double> data;
std::ifstream file("values.txt");

std::copy(     std::istream_iterator<double>(file),
               std::istream_iterator<double>(),
               std::back_inserter(data) );
```

Other algorithms in the STL

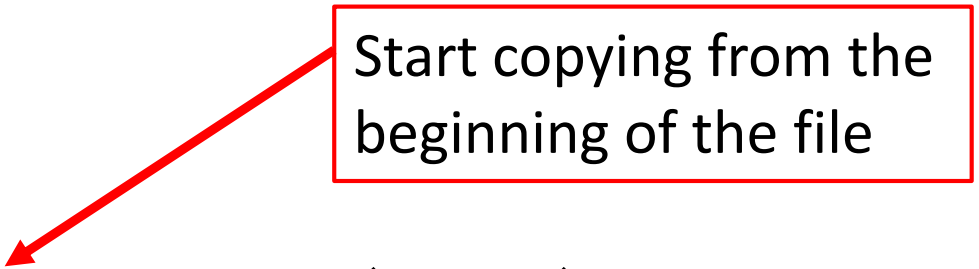
- Copying values from file:

```
std::vector<double> data;
std::ifstream file("values.txt");

std::copy(     std::istream_iterator<double>(file),
               std::istream_iterator<double>(),
               std::back_inserter(data) );
```

Other algorithms in the STL

- Copying values from file:

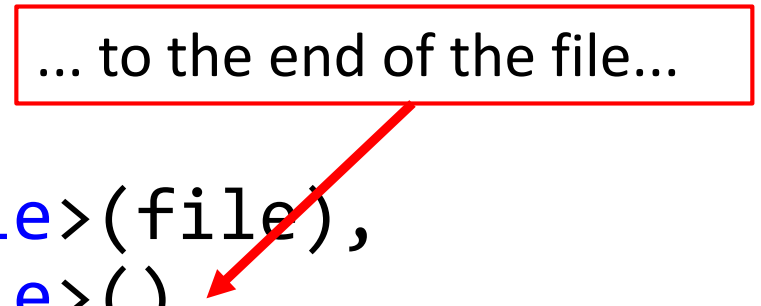
```
std::vector<double> data;
std::ifstream file("values.txt");

std::copy(     std::istream_iterator<double>(file),
               std::istream_iterator<double>(),
               std::back_inserter(data) );
```



Other algorithms in the STL

- Copying values from file:

```
std::vector<double> data;
std::ifstream file("values.txt");

std::copy(     std::istream_iterator<double>(file),
               std::istream_iterator<double>(),
               std::back_inserter(data) );
```



Other algorithms in the STL

- Copying values from file:

```
std::vector<double> data;
std::ifstream file("values.txt");

std::copy(     std::istream_iterator<double>(file),
               std::istream_iterator<double>(),
               std::back_inserter(data) );
```

... while interpreting the characters as doubles

and inserting them into data (effectively calls push_back)

Other algorithms in the STL

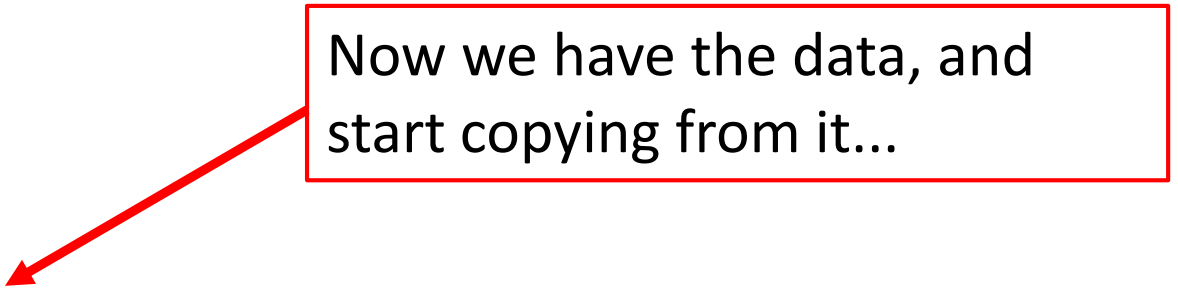
- Copying values to a stream:

```
std::vector<double> data;

std::copy(data.cbegin(),
          data.cend(),
          std::ostream_iterator<double>(std::cout,", "));
```

Other algorithms in the STL

- Copying values to a stream:

```
std::vector<double> data;

std::copy(data.cbegin(),
          data.cend(),
          std::ostream_iterator<double>(std::cout,", "));
```



Other algorithms in the STL

- Copying values to a stream:

```
std::vector<double> data;

std::copy(data.cbegin(),
          data.cend(),
          std::ostream_iterator<double>(std::cout,", "));
```

To an output stream iterator, bound to the standard output

Other algorithms in the STL

- Copying values to a stream:

```
std::vector<double> data;

std::copy(data.cbegin(),
          data.cend(),
          std::ostream_iterator<double>(std::cout,", "));
```

The elements will be separated by ", ".

Other algorithms in the STL

- Copying values to a stream:

```
std::vector<double> data;

std::copy(data.cbegin(),
          data.cend(),
          std::ostream_iterator<double>(std::cout,", "));
```

Problem: there will be a comma after the last element too...

Other algorithms in the STL

- Copying values to a stream, better solution:

```
std::vector<double> data;


std::copy( data.cbegin(),
           --data.cend(),
           std::ostream_iterator<double>(std::cout, ", ")
         );
std::cout << data.back() << "\n";
```

Other algorithms in the STL

- Copying values to a stream, better solution:

```
std::vector<double> data;

std::copy( data.cbegin(),
           --data.cend(),
           std::ostream_iterator<double>(std::cout, ", ")
         );
std::cout << data.back() << "\n";
```

Lets skip the last element
(step back 1 from the end)

Other algorithms in the STL

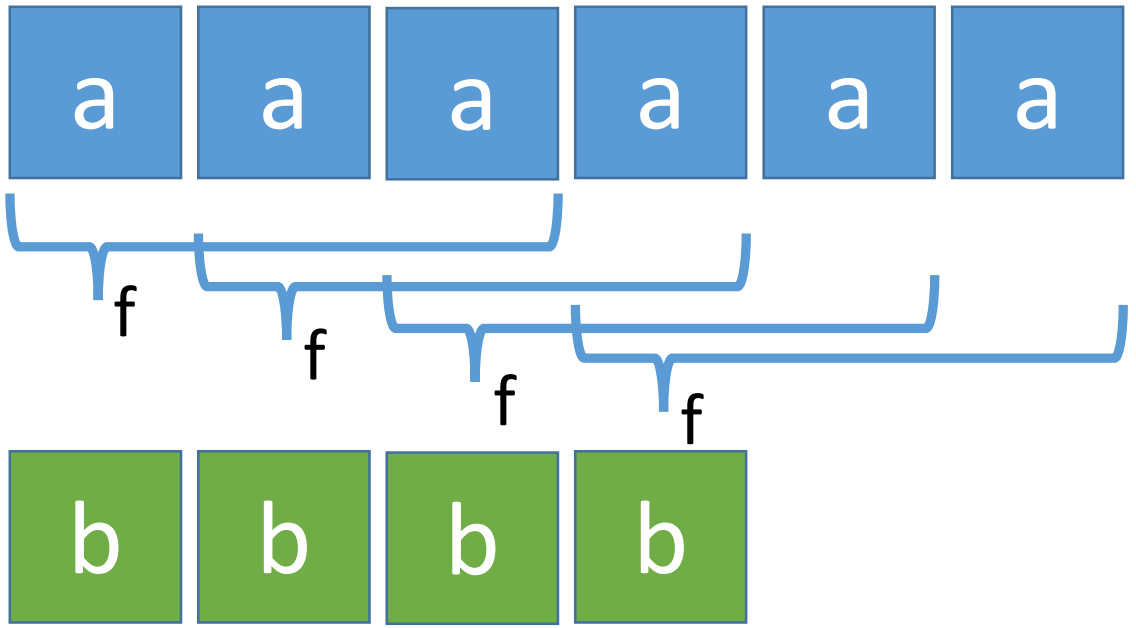
- Copying values to a stream, better solution:

```
std::vector<double> data;


std::copy( data.cbegin(),
           --data.cend(),
           std::ostream_iterator<double>(std::cout, ", ")
         );
std::cout << data.back() << "\n";
```

And write out the last one here.

Other algorithms in the STL

- Let's write an algorithm for the sliding window problem with window size 3:

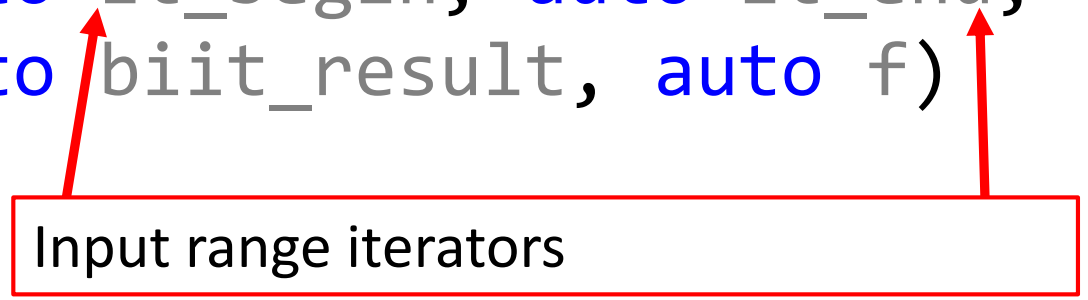


Sliding window algorithm

```
auto sliding_window3 = []( auto it_begin, auto it_end,
                           auto biit_result, auto f)
{
    auto it = it_begin;
    auto it_end2 = --(--it_end);
    while( it != it_end2 )
    {
        biit_result = f(*it, *(it+1), *(it+2));
        ++it;
    }
};
```

Sliding window algorithm

```
auto sliding_window3 = []( auto it_begin, auto it_end,
                           auto biit_result, auto f)
{
    auto it = it_begin;
    auto it_end2 = --(--it_end);
    while( it != it_end2 )
    {
        biit_result = f(*it, *(it+1), *(it+2));
        ++it;
    }
};
```

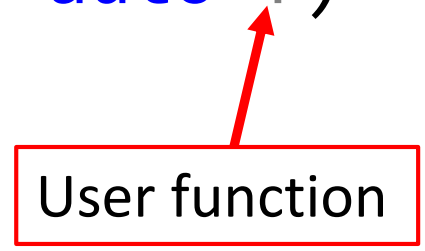


Sliding window algorithm

```
auto sliding_window3 = []( auto it_begin, auto it_end,
                           auto biit_result, auto f)
{
    auto it = it_begin;
    auto it_end2 = --(--it_end);
    while( it != it_end2 )
    {
        biit_result = f(*it, *(it+1), *(it+2));
        ++it;
    }
};
```

Sliding window algorithm

```
auto sliding_window3 = []( auto it_begin, auto it_end,
                           auto biit_result, auto f)
{
    auto it = it_begin;
    auto it_end2 = --(--it_end);
    while( it != it_end2 )
    {
        biit_result = f(*it, *(it+1), *(it+2));
        ++it;
    }
};
```



Sliding window algorithm

```
auto sliding_window3 = []( auto it_begin, auto it_end,
                           auto biit_result, auto f)
{
    auto it = it_begin;
    auto it_end2 = --(--it_end);
    while( it != it_end2 )
    {
        biit_result = f(*it, *(it+1), *(it+2));
        ++it;
    }
};
```

Step back 2 from the end

Otherwise we'd be over indexing here

Sliding window algorithm

- Application: moving average

```
std::vector<double> arr = {1., 3., 5., 7., 9., 11.};

std::vector<double> result;
sliding_window3(   arr.cbegin(), arr.cend(),
                   std::back_inserter(result),
                   [](double x0, double x1, double x2)
                   {
                        return (x0+x1+x2)/3.;
                   });
```

Summary

Why use algorithms?

- More structured and more expressive code
- Improved productivity
- Less possibility for errors

Summary

Why use algorithms?

- More structured and more expressive code
- Improved productivity
- Less possibility for errors

C++17: Parallel algorithms!

Summary

C++17: Parallel algorithms!

With the new C++ standard there will be parallel versions for all the algorithms in the standard library. All you'll need to do, is to specify 1 more parameter to the algorithm

STL: CPU Threads

GPU Vendors: GPU accelerated versions of algorithms!