CMPT 373
Software Development Methods

Thinking in Sequences: Find, Filter, Map, & Reduce

Nick Sumner
wsumner@sfu.ca

Operations on Collections

- Programs usually work with *collections* of data

Operations on Collections

- Programs usually work with *collections* of data
  - Choose some subset of data to work with

Operations on Collections

- Programs usually work with *collections* of data
  - Choose some subset of data to work with
  - Use the values of selected data

Operations on Collections

- Programs usually work with *collections* of data
  - Choose some subset of data to work with
  - Use the values of selected data
  - Create new values based based those seen

Operations on Collections

- Programs usually work with *collections* of data
  - Choose some subset of data to work with
  - Use the values of selected data
  - Create new values based based those seen
  - Maybe even update the existing collection

Operations on Collections

- Programs usually work with *collections* of data
  - Choose some subset of data to work with
  - Use the values of selected data
  - Create new values based based those seen
  - Maybe even update the existing collection
- This is pervasive at all levels
  - Data structures
  - Databases
  - Distributed stores
  - …

Operations on Collections

- Programs usually work with *collections* of data
  - Choose some subset of data to work with
  - Use the values of selected data
  - Create new values based based those seen
  - Maybe even update the existing collection
- This is pervasive at all levels
  - Data structures
  - Databases
  - Distributed stores
  - …
- And it is *error prone* and *easy to overcomplicate*

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollment date offset
grouped by GPA in 0.5 increments.

Guidance on Collections

- From the very first year (ideally, semester) you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollmen
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollment
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

What can go wrong?
What is challenging?

There is at least 1
rare bug in this code!

Guidance on Collections

- From the very first year (ideally, semester) you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollmen
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

What can go wrong?
What is challenging?

There is at least 1
rare bug in this code!

Guidance on Collections

- From the very first year (ideally, semester) you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollmen
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

What can go wrong?
What is challenging?

There is at least 1
rare bug in this code!

Guidance on Collections

- From the very first year (ideally, semester) you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollmen
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

What can go wrong?
What is challenging?

There is at least 1
rare bug in this code!

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollmen
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

What can go wrong?
What is challenging?

There is at least 1
rare bug in this code!

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?

Given a collection of students,
for all students in year 3+,
determine their average enrollmen[t]
grouped by GPA in 0.5 increments.

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

What can go wrong?
What is challenging?

There is at least 1
rare bug in this code!

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?
- The smaller implementation details get in the way of
  what exactly is going on
  why you believe it is correct

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?
- The smaller implementation details get in the way of
  what exactly is going on
  why you believe it is correct
- Significant effort is spent on handling
  common corner cases of collections instead of *goal oriented logic*

Guidance on Collections

- From the very first year (ideally, semester)
  you are told to break a problem into smaller parts.
  - But how do you go about doing that?
  - What are the primary parts to consider?
- The smaller implementation details get in the way of
  what exactly is going on
  why you believe it is correct
- Significant effort is spent on handling
  common corner cases of collections instead of *goal oriented logic*
- Breaking the problem apart into pieces helps clarify these steps

Slight Improvements

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```


```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Slight Improvements

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Some library & language features raise the level of abstraction.

Slight Improvements

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Some library & language features raise the level of abstraction.

What is the simplest (and maybe not most effective) way we can improve this?

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Slight Improvements

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Some library & language features raise the level of abstraction.

What is the simplest (and maybe not most effective) way we can improve this?

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (const Student& student : students) {
  if (student.year >= 3) {
    int bucket = int(student.gpa / 0.5);
    buckets[bucket].offset += student.enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Slight Improvements

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}
```

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (const Student& student : students) {
  if (student.year >= 3) {
    int bucket = int(student.gpa / 0.5);
    buckets[bucket].offset += student.enrollment;
    buckets[bucket].count  += 1;
  }
}
```

Some library & language features raise the level of abstraction.

If we want to do better, we need to separate the *higher level operations*

Slight Improvements

Slight Improvements

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });
```

Slight Improvements

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });
```

Slight Improvements

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {}, &BucketData::bucket);
```

Slight Improvements

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {}, &BucketData::bucket);

std::array<std::span<BucketData>,9> buckets;
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
    [remainder](const auto& s) { return s.bucket != remainder.front().bucket; });
  buckets[remainder.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}
```

Slight Improvements

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {}, &BucketData::bucket);

std::array<std::span<BucketData>,9> buckets;
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
    [remainder](const auto& s) { return s.bucket != remainder.front().bucket; });
  buckets[remainder.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}
```

This code is even longer!
Was all that a waste?!

Slight Improvements

Filter

```
std::vector<Student> selected;
selected                (students.size());
std::ran               y_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });
```

Map

```
struct BucketData { int offset; int bucket; };
std::vec               etData> projected(selected.size());
std::ran               nsform(selected, projected.begin(),
  []  (co              tudent& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {}, &BucketData::bucket);
```

Group

```
std::array<std::span<BucketData>,9> buckets;
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
    [rem                 nst auto& s) { return s.bucket != remainder.front().bucket; });
  bucket               r.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}
```

This code is even longer!
Was all that a waste?!

Slight Improvements

```
std::vector<Student> selected;
selecte… (students.size());
std::ran… y_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vec… etData> projected(selected.size());
std::ran… nsform(selected, projected.begin(),
  [] (co… tudent& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {}, …

std::array<std::span<BucketData> …
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
    [rem… nst auto& s) { return s.bucket != remainder.front().bucket; });
  bucket… r.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}
```

Filter

Map

Group

This code is even longer!
Was all that a waste?!

Separating these pieces makes code easier to maintain.
Most languages today make them *simple* & *efficient*.

Slight Improvements

```
std::vector<Student> selected;
selected            e(students.size());
std::ran            y_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vec            etData> projected(selected.size());
std::ran            nsform(selected, projected.begin(),
  [] (const Student& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {},

std::array<std::span<BucketData>
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(
    [rem            onst auto& s) { return s.bucket != remainder.front().bucket; });
  bucket            er.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}
```

Filter

Map

Group

This code is even longer!
Was all that a waste?!

Separating these pieces makes code easier to maintain.
Most languages today make them *simple* & *efficient*.

For simpler problems than this,
removing the bookkeeping alone is worth it.

Slight Improvements

```
std::vector<Student> selected;
selecte              (students.size());
std::ran             y_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vec             etData> projected(selected.size());
std::ran             nsform(selected, projected.begin(),
  [] (co      tudent& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {},

std::array<std::span<BucketData>
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if
    [rem          nst auto& s) { return s.bucket != remainder.front().bucket; });
  bucket          r.front().bucket] = std::sp                 dEnd};
  remainder = std::span{foundEnd, remainder.
}
```

Filter

Map

Group

This code is even longer!
Was all that a waste?!

Separating these pieces makes code easier to maintain.
Most languages today make them *simple* & *efficient*.

For simpler problems than this,
removing the bookkeeping alone is worth it.

We can actually break it down
in simpler ways, too.

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on
  - Map – ***transform*** each value into a new value

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on
  - Map – ***transform*** each value into a new value
  - Reduce – ***combine*** values into a result

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on
  - Map – ***transform*** each value into a new value
  - Reduce – ***combine*** values into a result
  - Find – ***identify*** a useful location / boundary in data

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on
  - Map – ***transform*** each value into a new value
  - Reduce – ***combine*** values into a result
  - Find – ***identify*** a useful location / boundary in data

Most things you do are a combination of these steps.
This shrinks the solution space!

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on
  - Map – ***transform*** each value into a new value
  - Reduce – ***combine*** values into a result
  - Find – ***identify*** a useful location / boundary in data
- How these primitives are spelled varies (e.g. in classic C++)
  - Filter – partition, stable_partition, copy_if
  - Map – transform
  - Reduce – accumulate, reduce
  - Find – find, find_if

Breaking Problems into Pieces

- Most problems on collections break down into *operations on collections*
  - Filter – ***select*** a subset of data to work on
  - Map – ***transform*** each value into a new value
  - Reduce – ***combine*** values into a result
  - Find – ***identify*** a useful location / boundary in data
- How these primitives are spelled varies (e.g. in classic C++)
  - Filter – partition, stable_partition, copy_if
  - Map – transform
  - Reduce – accumulate, reduce
  - Find – find, find_if
- One of the first things you should do in a new language is figure out how these are spelled
  - Java (streams), C# (LINQ), Python (builtins+comprehensions), C++ (STL & ranges)

Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });
```

Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });
```

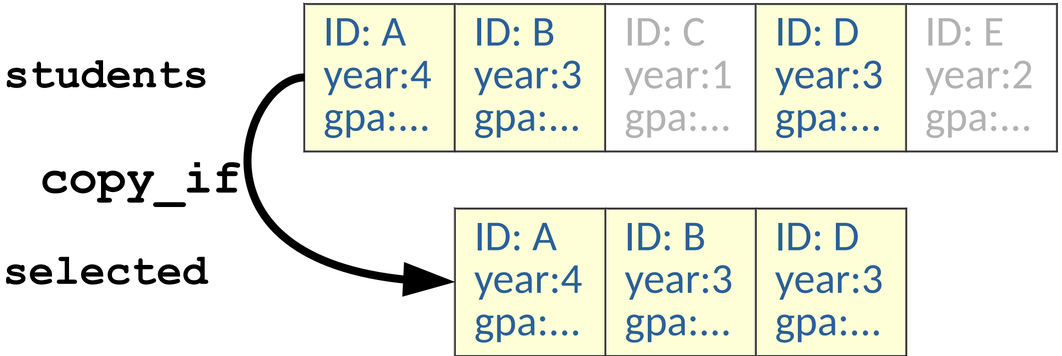
students				
ID: A year:4 gpa:...	ID: B year:3 gpa:...	ID: C year:1 gpa:...	ID: D year:3 gpa:...	ID: E year:2 gpa:...

selected

Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });
```



Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

auto subrange = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });
```

Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

auto subrange = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });
```

students

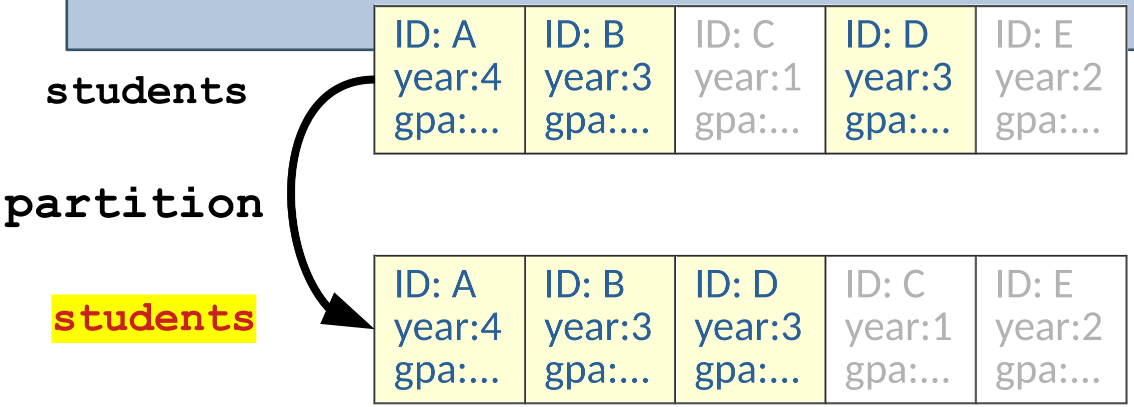
ID: A year:4 gpa:…	ID: B year:3 gpa:…	ID: C year:1 gpa:…	ID: D year:3 gpa:…	ID: E year:2 gpa:…

Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

auto subrange = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });
```

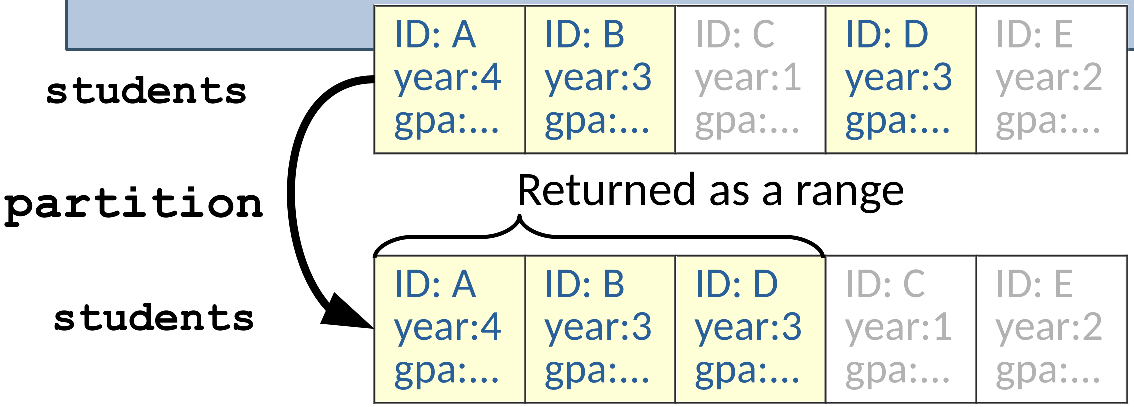


Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

auto subrange = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });
```

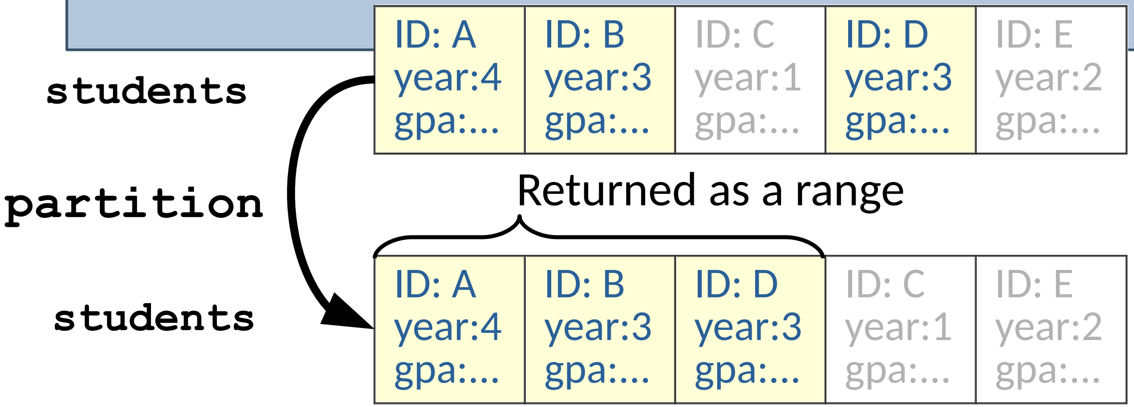


Filter

- Given a predicate p, identify & group the elements for which p is true
  - std::partition, stable_partition, std::copy_if

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

auto subrange = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });
```



Mutability makes it one line

Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```

Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

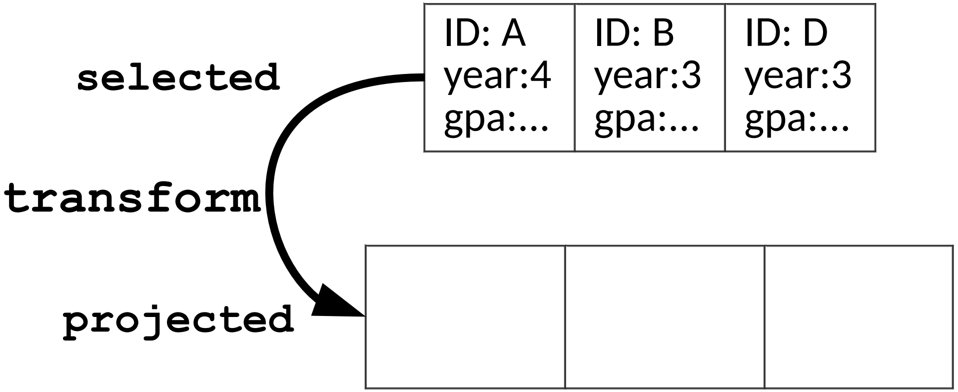
```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```

selected	ID: A year:4 gpa:…	ID: B year:3 gpa:…	ID: D year:3 gpa:…
projected			

Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

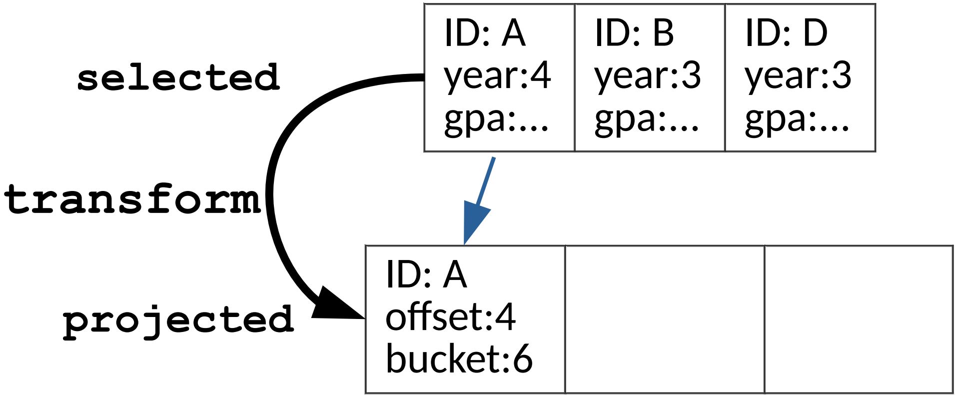
```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```



Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

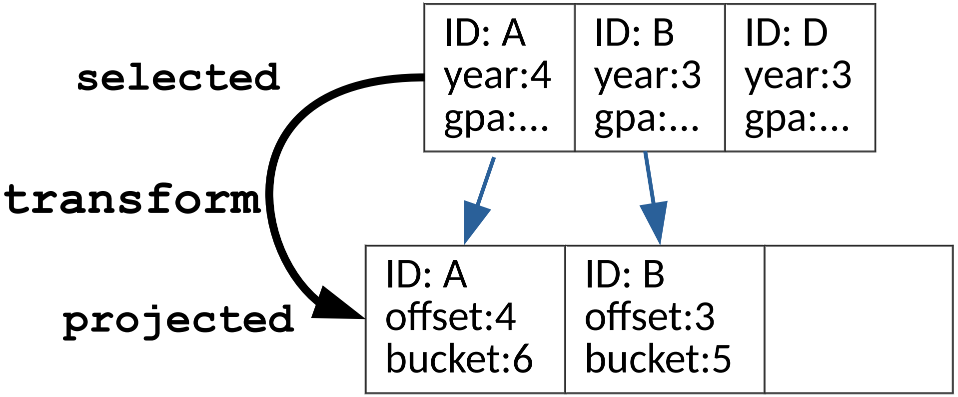
```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```



Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

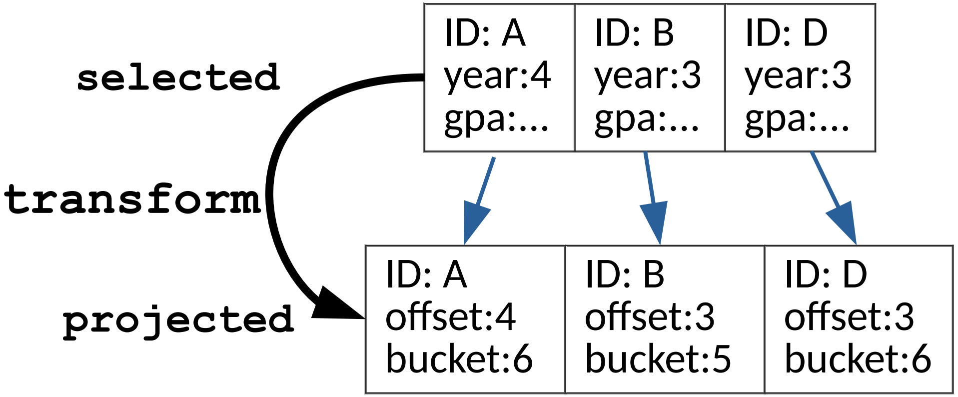
```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```



Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

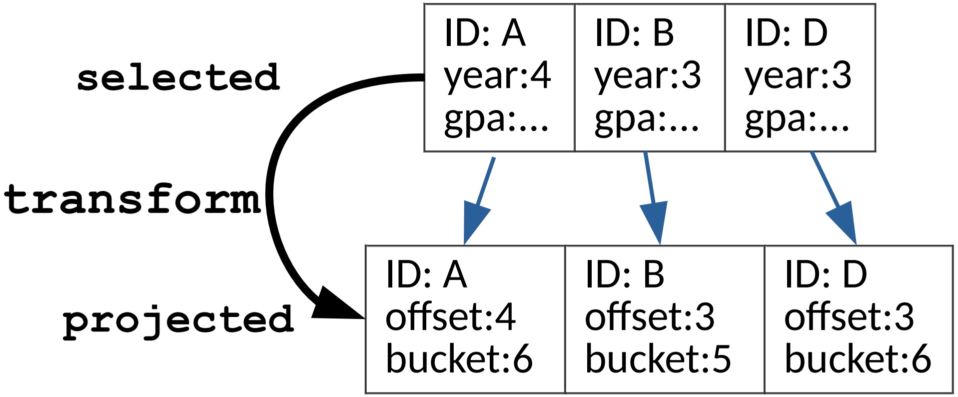
```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```



Map

- Apply a function to each element of a collection and store the result as desired
  - std::transform

```
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) {
    return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
});
```



The resulting type can be different, but this is not required

Reduce

- Combine results of processing different elements
  - std::accumulate, std::reduce

```
std::vector numbers = { 0, 1, 2, 3, 4, 5, 6, 7 };
auto sum = ...
```

Reduce

- Combine results of processing different elements
  - std::accumulate, std::reduce

```
std::vector numbers = { 0, 1, 2, 3, 4, 5, 6, 7 };
auto sum = std::accumulate(numbers.begin(), numbers.end());
```

Reduce

- Combine results of processing different elements
  - std::accumulate, std::reduce

```
std::vector numbers = { 0, 1, 2, 3, 4, 5, 6, 7 };
auto sum = std::accumulate(numbers.begin(), numbers.end());

auto product = std::accumulate(numbers.begin(), numbers.end(),
                                ...
```

Reduce

- Combine results of processing different elements
  - std::accumulate, std::reduce

```
std::vector numbers = { 0, 1, 2, 3, 4, 5, 6, 7 };
auto sum = std::accumulate(numbers.begin(), numbers.end());

auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

Reduce

- Combine results of processing different elements
  - std::accumulate, std::reduce

```
std::vector numbers = { 0, 1, 2, 3, 4, 5, 6, 7 };
auto sum = std::accumulate(numbers.begin(), numbers.end());

auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

- Reduce operations take
  - An *initial value*

Reduce

- Combine results of processing different elements
  - std::accumulate, std::reduce

```
std::vector numbers = { 0, 1, 2, 3, 4, 5, 6, 7 };
auto sum = std::accumulate(numbers.begin(), numbers.end());

auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

- Reduce operations take
  - An initial value
  - A *function* consuming the value computed so far & current element to compute a new value

Reduce

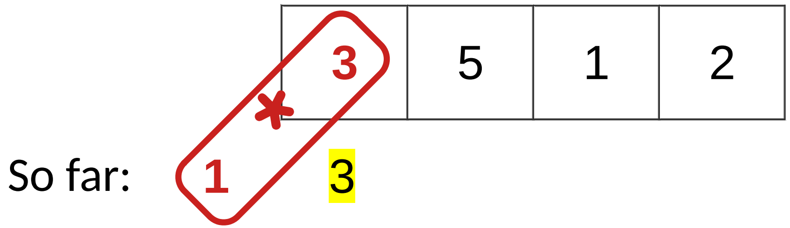
```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

3	5	1	2

So far: 1

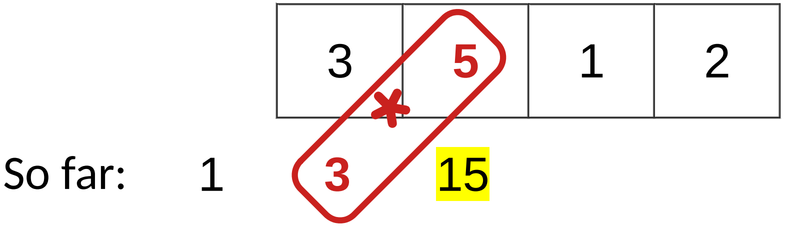
Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```



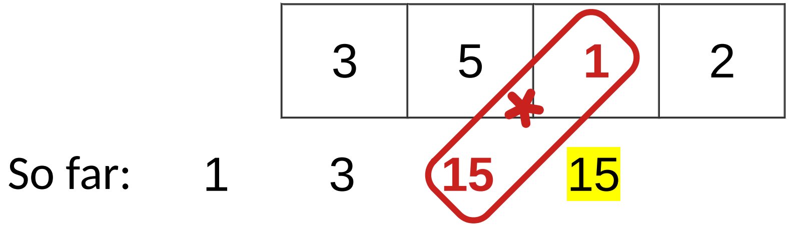
Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```



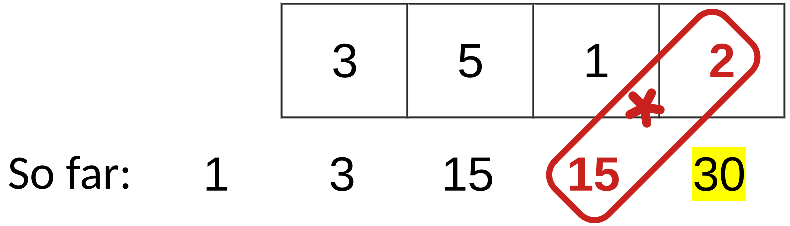
Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```



Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```



Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

3	5	1	2

So far: 1 3 15 15 **30**

Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

		3	5	1	2
So far:	1	3	15	15	30

- Reduce operations explicitly capture the inductive nature of loops

Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

		3	5	1	2
So far:	1	3	15	15	30

- Reduce operations explicitly capture the inductive nature of loops
  - Start with a base case

Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

	3	5	1	2
So far: 1	3	15	15	30

- Reduce operations explicitly capture the inductive nature of loops
  - Start with a base case
  - Each iteration computes the state so far

Reduce

```
std::vector numbers = { 3, 5, 1, 2 };
auto product = std::accumulate(numbers.begin(), numbers.end(),
                               1, std::multiplies{});
```

		3	5	1	2
So far:	1	3	15	15	**30**

- Reduce operations explicitly capture the inductive nature of loops
  - Start with a base case
  - Each iteration computes the state so far
  - When all iterations have completed,
    the final result should be the intended goal

Reduce

- Note: The computed value can be a different type than the elements!

Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    a collection of T
    an initial value U
    an operation (U,T) → U
    reduce computes a value U from a collection

Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    - a collection of T
    - an initial value U
    - an operation (U,T) → U

    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    a collection of T
    an initial value U
    an operation (U,T) → U
    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
   [](std::string sofar, int i) { return sofar + std::to_string(i); });
```

Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    a collection of T
    an initial value U
    an operation (U,T) → U
    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
  [](std::string sofar, int i) { return sofar + std::to_string(i); });
```

3	5	1	2

So far: ""

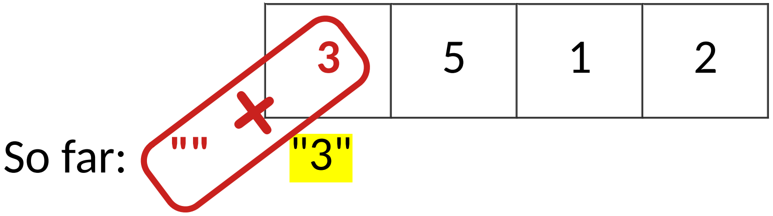
Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    - a collection of T
    - an initial value U
    - an operation (U,T) → U

    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
  [](std::string sofar, int i) { return sofar + std::to_string(i); });
```



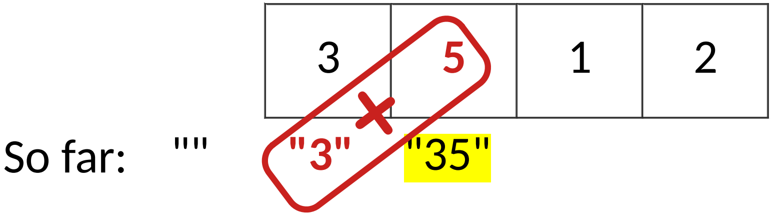
Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    - a collection of T
    - an initial value U
    - an operation (U,T) → U

    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
   [](std::string sofar, int i) { return sofar + std::to_string(i); });
```



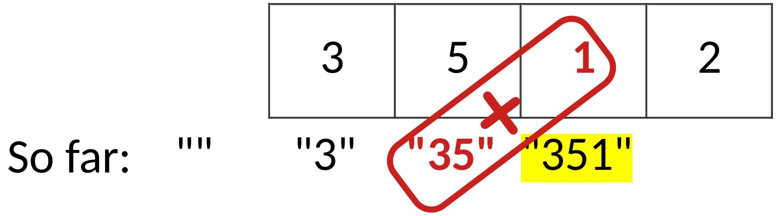
Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    - a collection of T
    - an initial value U
    - an operation (U,T) → U

    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
   [](std::string sofar, int i) { return sofar + std::to_string(i); });
```

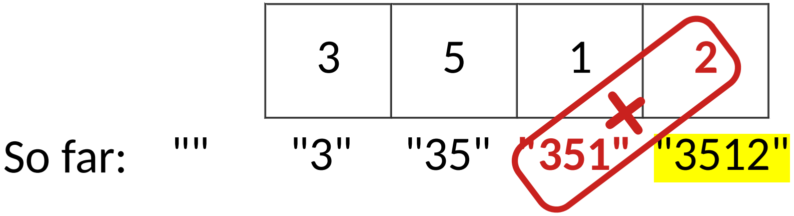


Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    a collection of T
    an initial value U
    an operation (U,T) → U
    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
   [](std::string sofar, int i) { return sofar + std::to_string(i); });
```



Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    - a collection of T
    - an initial value U
    - an operation (U,T) → U

    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
  [](std::string sofar, int i) { return sofar + std::to_string(i); });
```

	3	5	1	2
So far: ""	"3"	"35"	"351"	"3512"

- The computed state so far can be anything needed to capture the progress made toward the goal

Reduce

- Note: The computed value can be a different type than the elements!
  - Thus, given:
    a collection of T
    an initial value U
    an operation (U,T) → U
    reduce computes a value U from a collection

reduce: ([T], U, (U,T)→U) → U

```
std::vector numbers = { 3, 5, 1, 2 };
auto asString = std::accumulate(numbers.begin(), numbers.end(),std::string{},
  [](std::string sofar, int i) { return sofar + std::to_string(i); });
```

	3	5	1	2
So far: ""	"3"	"35"	"351"	"3512"

But do remember, concatenating strings like this is a poor *goal*.

- The computed state so far can be anything needed to capture the progress made toward the goal

Generality of Reduce

- In fact, this means most functions on loops can be written via a reduce!

Generality of Reduce

- In fact, this means most functions on loops can be written via a reduce!

```
bool
any_of(auto& collection, auto predicate) {
  return std::accumulate(collection.begin() collection.end(), false,
    [](bool sofar, auto& element) { return sofar || predicate(element); });
}
```

Generality of Reduce

- In fact, this means most functions on loops can be written via a reduce!

```
bool
any_of(auto& collection, auto predicate) {
  return std::accumulate(collection.begin() collection.end(), false,
    [](bool sofar, auto& element) { return sofar || predicate(element); });
}
```

```
auto
max(auto& collection, auto minimum) {
  return std::accumulate(collection.begin() collection.end(), minimum,
    [](auto sofar, auto& element) {
      return (element > sofar) ? element : sofar;
  });
}
```

Generality of Reduce

- In fact, this means most functions on loops can be written via a reduce!

```
bool
any_of(auto& collection, auto predicate) {
  return std::accumulate(collection.begin() collection.end(), false,
    [](bool sofar, auto& element) { return sofar || predicate(element); });
}
```

```
auto
max(auto& collection, auto minimum) {
  return std::accumulate(collection.begin() collection.end(), minimum,
    [](auto sofar, auto& element) {
      return (element > sofar) ? element : sofar;
  });
}
```

```
auto
count_if(auto& collection, auto predicate) {
  return std::accumulate(collection.begin() collection.end(), 0,
    [predicate](auto sofar, auto& element) {
      return sofar + (predicate(element) ? 1 : 0);
  });
}
```

Find

- Find clearly doesn't give us the ability to compute anything *new*

Find

- Find clearly doesn’t give us the ability to compute anything new
  - Some schools prefer to teach just filter, map, and reduce

Find

- Find clearly doesn't give us the ability to compute anything new
  - Some schools prefer to teach just filter, map, and reduce
  - But it can add *efficiency*

Find

- Find clearly doesn't give us the ability to compute anything new
  - Some schools prefer to teach just filter, map, and reduce
  - But it can add *efficiency*
- Note: map, filter, & reduce will consider an *entire* collection

Find

- Find clearly doesn't give us the ability to compute anything new
  - Some schools prefer to teach just filter, map, and reduce
  - But it can add *efficiency*
- Note: map, filter, & reduce will consider an *entire* collection
- Find gives us the ability to short-circuit operations

Find

- Find clearly doesn't give us the ability to compute anything new
  - Some schools prefer to teach just filter, map, and reduce
  - But it can add *efficiency*
- Note: map, filter, & reduce will consider an *entire* collection
- Find gives us the ability to short-circuit operations

```
bool
any_of(auto& collection, auto predicate) {
  return std::ranges::find_if(collection, predicate) != collection.end();
}
```

Find

- Find clearly doesn't give us the ability to compute anything new
  - Some schools prefer to teach just filter, map, and reduce
  - But it can add *efficiency*
- Note: map, filter, & reduce will consider an *entire* collection
- Find gives us the ability to short-circuit operations

```
bool
any_of(auto& collection, auto predicate) {
  return std::ranges::find_if(collection, predicate) != collection.end();
}
```

While reduce processes the entire list,
this stops at the first match

Can we now make this clearer? (a bit)

Mutability & selection of how to connect the core ingredients affects the simplicity

Can we now make this clearer? (a bit)

```
auto selected = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });
```

Mutability & selection of how to connect the core ingredients affects the simplicity

Can we now make this clearer? (a bit)

```
auto selected = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });

auto getBucket = [] (const Student& s) {
  return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
};
```

Mutability & selection of how to connect the core ingredients affects the simplicity

Can we now make this clearer? (a bit)

```
auto selected = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });

auto getBucket = [] (const Student& s) {
  return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
};

std::ranges::sort(selected, {}, getBucket);
```

Mutability & selection of how to connect the core ingredients affects the simplicity

Can we now make this clearer? (a bit)

```
auto selected = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });

auto getBucket = [] (const Student& s) {
  return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
};

std::ranges::sort(selected, {}, getBucket);

std::array<std::span<BucketData>,9> buckets;
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
    [remainder](const auto& s) { return s.bucket != remainder.front().bucket; });
  buckets[remainder.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}
```

Mutability & selection of how to connect the core ingredients affects the simplicity

Can we now make this clearer? (a bit)

```
auto selected = std::ranges::partition(students,
  [](const Student& s) { return s.year >= 3; });

auto getBucket = [] (const Student& s) {
  return BucketData{min(int(s.gpa / 0.5), 8), s.offset};
};

std::ranges::sort(selected, {}, getBucket);

std::array<std::span<BucketData>,9> buckets;
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
     [remainder](const auto& s) { return s.bucket != remainder.front().bucket; });
  buckets[remainder.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}

std::array<float,9> averages;
std::ranges::transform(buckets, averages.begin(), [](auto& bucket) {
  return std::accumulate(bucket.begin(), bucket.end(), 0.0f,
                 [](float sofar, const auto& student) { return sofar + student.offset; })
      / (bucket.empty() ? 1 : bucket.size()); });
```

Mutability & selection of how to connect the core ingredients affects the simplicity

So why was the “improvement” complicated

- Operating eagerly requires (e.g.)
  - First selecting all data and storing it
  - Then mapping all data and storing it
  - Then grouping all data and storing it
  - Then analyzing all data and storing it

```
std::vector<Student> selected;
selected.reserve(students.size());
std::ranges::copy_if(students, std::back_inserter(selected),
  [](const Student& s) { return s.year >= 3; });

struct BucketData { int offset; int bucket; };
std::vector<BucketData> projected(selected.size());
std::ranges::transform(selected, projected.begin(),
  [] (const Student& s) { return BucketData{min(int(s.gpa / 0.5), 8), s.offset}; });

std::ranges::sort(projected, {}, &BucketData::bucket);

std::array<std::span<BucketData>,9> buckets;
auto remainder = std::span{projected};
while (!remainder.empty()) {
  auto foundEnd = std::ranges::find_if(remainder,
    [remainder](const auto& s) { return s.bucket != remainder.front().bucket; });
  buckets[remainder.front().bucket] = std::span{remainder.begin(), foundEnd};
  remainder = std::span{foundEnd, remainder.end()};
}

std::array<float,9> averages;
std::ranges::transform(buckets, averages.begin(), [](auto& bucket) {
  return std::accumulate(bucket.begin(), bucket.end(), 0.0f,
                   [](float sofar, const auto& student) { return sofar + student.offset; })
      / (bucket.empty() ? 1 : bucket.size()); });
```

So why was the “improvement” complicated

- Operating eagerly requires (e.g.)
  - First selecting all data and storing it
  - Then mapping all data and storing it
  - Then grouping all data and storing it
  - Then analyzing all data and storing it
- Instead, most languages compose operations *lazily*

So why was the “improvement” complicated

- Operating eagerly requires (e.g.)
  - First selecting all data and storing it
  - Then mapping all data and storing it
  - Then grouping all data and storing it
  - Then analyzing all data and storing it
- Instead, most languages compose operations *lazily*
  - Look at one element
    - Select it, map it, group it, & store it *as necessary*

So why was the "improvement" complicated

- Operating eagerly requires (e.g.)
  - First selecting all data and storing it
  - Then mapping all data and storing it
  - Then grouping all data and storing it
  - Then analyzing all data and storing it
- Instead, most languages compose operations *lazily*
  - Look at one element
    - Select it, map it, group it, & store it *as necessary*
  - Proceed to the next element

So why was the “improvement” complicated

- Operating eagerly requires (e.g.)
  - First selecting all data and storing it
  - Then mapping all data and storing it
  - Then grouping all data and storing it
  - Then analyzing all data and storing it
- Instead, most languages compose operations *lazily*
  - Look at one element
    - Select it, map it, group it, & store it *as necessary*
  - Proceed to the next element
- The APIs express operations to construct these lazy operations, removing this boilerplate!

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto bucketable =
```

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto bucketable = students
```

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto bucketable = students
  | std::ranges::views::filter(whichStudents)
```

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto bucketable = students
  | std::ranges::views::filter(whichStudents)
  | to<std::vector>();
std::ranges::sort(bucketable, {}, getBucket);
auto averages = bucketable
```

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto bucketable = students
  | std::ranges::views::filter(whichStudents)
  | to<std::vector>();
std::ranges::sort(bucketable, {}, getBucket);
auto averages = bucketable
  | views::group_by([] (const auto& s1, const auto& s2) {
      return s1.bucket == s2.bucket;
    })
```

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto bucketable = students
  | std::ranges::views::filter(whichStudents)
  | to<std::vector>();
std::ranges::sort(bucketable, {}, getBucket);
auto averages = bucketable
  | views::group_by([] (const auto& s1, const auto& s2) {
      return s1.bucket == s2.bucket;
    })
  | std::ranges::views::transform(average);
```

Streaming Collections APIs

- Streaming APIs work lazily on potentially infinite sequences of data

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) { return 0; } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; }) / range.size();
  }
};

auto averages = students
  | std::ranges::views::filter(whichStudents)
  | actions::group_by_key(getBucket);
  | std::ranges::views::transform(average);
```

Or eventually.
Do you see why this is not already the default?

Streaming Collections APIs

- Comparing again

Streaming Collections APIs

- Comparing again

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range) {
  if (range.empty()) {
    return 0;
  } else {
    return ranges::fold(range, 0.0f,
      [](auto sofar, auto& datum) { return sofar + datum.offset; })
      / range.size();
  }
};

auto averages = students
  | std::ranges::views::filter(whichStudents)
  | actions::group_by_key(getBucket);
  | std::ranges::views::transform(average);
```

Streaming Collections APIs

- Comparing again

```
auto whichStudents = [](const Student& s) { return s.year >= 3; };
auto getBucket = [] (const Student& s) { return min(int(s.gpa / 0.5), 8); };

auto average = [] (auto range
  if (range.empty()) {
    return 0;
  } else {
    return ranges::fold(range
      [](auto sofar, auto& da
      / range.size();
  }
};

auto averages = students
  | std::ranges::views::filte
  | actions::group_by_key(get
  | std::ranges::views::trans
```

```
struct EnrollmentData { int offset; int count; };
std::array<EnrollmentData,9> buckets;
buckets.fill(EnrollmentData{0, 0});

for (unsigned i = 0; i < students.size(); ++i) {
  if (students[i].year >= 3) {
    int bucket = int(students[i].gpa / 0.5);
    buckets[bucket].offset += students[i].enrollment;
    buckets[bucket].count  += 1;
  }
}

std::array<float> averages;
for (unsigned bucket = 0; bucket < buckets.size(); ++bucket) {
  averages[bucket] = buckets[bucket].offset /
    float(buckets[bucket].count ? buckets[bucket].count : 0);
}
```

Benefits of streaming APIs

- The most obvious benefit is clarity & maintainability

Benefits of streaming APIs

- The most obvious benefit is clarity & maintainability
- It turns out that they are also easily parallelizable!

Benefits of streaming APIs

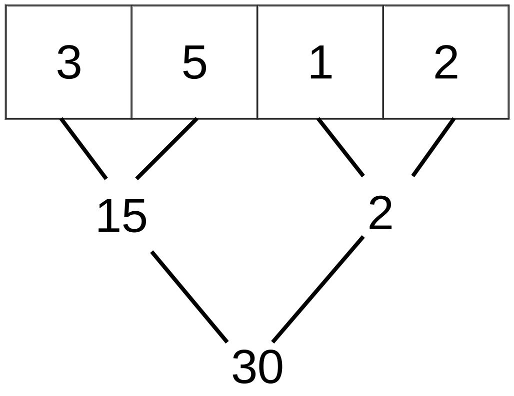
- The most obvious benefit is clarity & maintainability
- It turns out that they are also easily parallelizable!
  - map & filter are trivially parallelizable

Benefits of streaming APIs

- The most obvious benefit is clarity & maintainability
- It turns out that they are also easily parallelizable!
  - map & filter are trivially parallelizable
  - reduce is easily parallel when
    the operation is associative & commutative

Benefits of streaming APIs

- The most obvious benefit is clarity & maintainability
- It turns out that they are also easily parallelizable!
  - map & filter are trivially parallelizable
  - reduce is easily parallel when
    the operation is associative & commutative



Summary

- Avoid performing high level operations on loops yourself

Summary

- Avoid performing high level operations on loops yourself
- Break your problems down into sequences of find, filter, map, and reduce operations

Summary

- Avoid performing high level operations on loops yourself
- Break your problems down into sequences of find, filter, map, and reduce operations
- When possible, use streaming APIs for these operations for even better clarity & performance