Ordering, Min, Max, and MinMax

Lecture 15

Hartmut Kaiser

https://teaching.hkaiser.org/fall2023/csc3380/

Software Development Notes

Agile Software Development

- An approach to software development that emphasizes:
  - Small, collaborative development teams
    - Usually 4-6 members of the team
  - Working closely with customers/stakeholders
    - Have a user representative, that interfaces regularly with customers / users / stakeholders, on the team
  - Developing software in an short iterative cycles
    - requirements / code / testing / documentation
  - Building working versions of code often
    - Iterative Development
    - Often employs Continuous Integration/Continuous Deployment (CI/CD)
    - Test-driven development (TDD)

Agile Manifesto

- "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

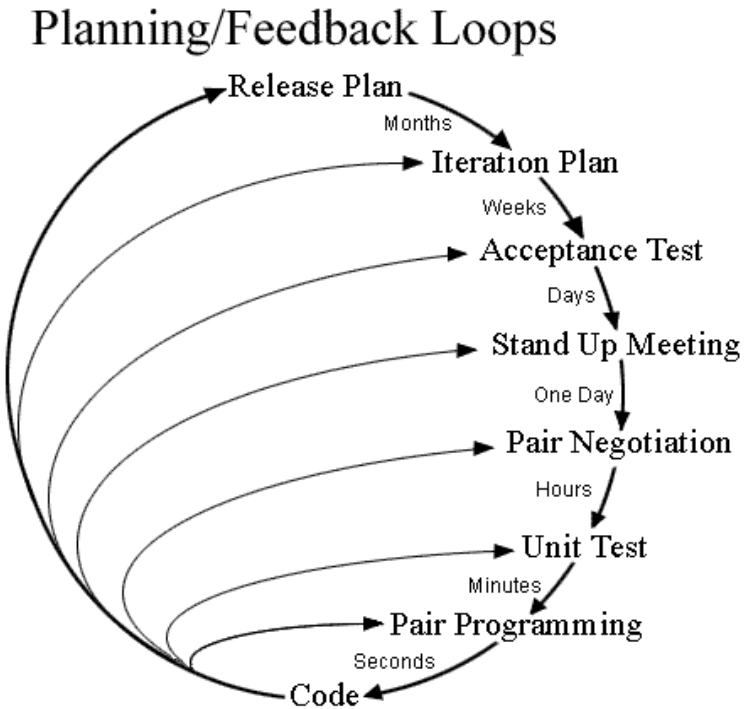
  - **Individuals and interactions** over processes and tools
  - **Working software** over comprehensive documentation
  - **Customer collaboration** over contract negotiation
  - **Responding to change** over following a plan

- That is, while there is value in the items on the right, we value the items on the left more."

https://www.agilealliance.org/agile101/the-agile-manifesto/

Agile Method: Extreme Programming

- Characterized by feedback loops
- On-site customer
- Planning game (with customer)
- Avoid features until needed
- Pair programming
  - Two programmers, one computer
  - One person writes, the other dictates
  - Typically writer is less skilled



Agile Method: SCRUM

- Scrum is a simple set of roles, responsibilities, and meetings that never change
- Time is divided into short work cadences, known as sprints, typically two to four weeks long
- The product is kept in a potentially shippable (properly integrated and tested) state at all times
- At the end of each sprint, stakeholders and team members meet to see a demonstrated potentially shippable product increment and plan its next steps

http://scrummethodology.com/

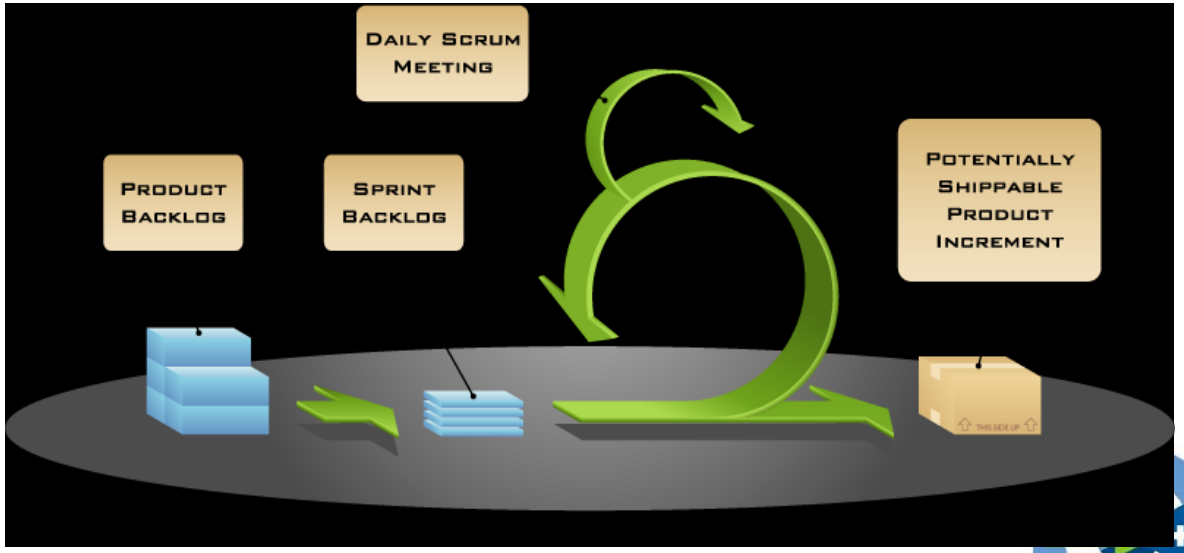
Scrum Roles

- Product Owner
  - The Product Owner should be a person with vision, authority, and availability
  - The Product Owner is responsible for continuously communicating the vision and priorities to the development team
- Scrum Master
  - The Scrum Master acts as a facilitator for the Product Owner and the team
  - The Scrum Master does not manage the team
  - The Scrum Master works to remove any impediments that are obstructing the team from achieving its sprint goals
- Team
  - A Scrum development team contains about 3-9 fully dedicated members, ideally in one team room protected from outside distractions
  - For software projects, a typical team includes a mix of software engineers, architects, programmers, analysts, QA experts, testers, and UI designers

http://scrummethodology.com/

Scrum Phases/Steps

- Product Backlog Creation
- Sprints
  - Sprint Planning and Sprint Backlog Creation
    - Team commits to work to be accomplished in the Sprint
  - Work on Sprint/Daily Scrum Meetings
    - Development
    - Testing
    - Documentation
  - Product Demonstration
  - Retrospective



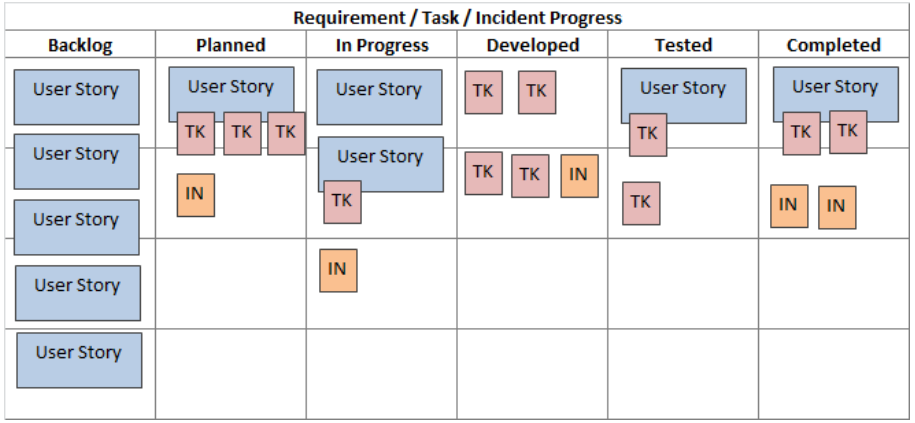
Agile Method: Kanban

- Kanban is a method for managing the creation of products with an emphasis on continual delivery while not overburdening the development team
- Kanban is based on 3 basic principles:
  - Visualize what you do today (workflow): seeing all the items in context of each other can be very informative
  - Limit the amount of work in progress (WIP): this helps balance the flow-based approach so teams don't start and commit to too much work at once
  - Enhance flow: when something is finished, the next highest priority thing from the backlog is pulled into play

https://www.versionone.com/what-is-kanban/

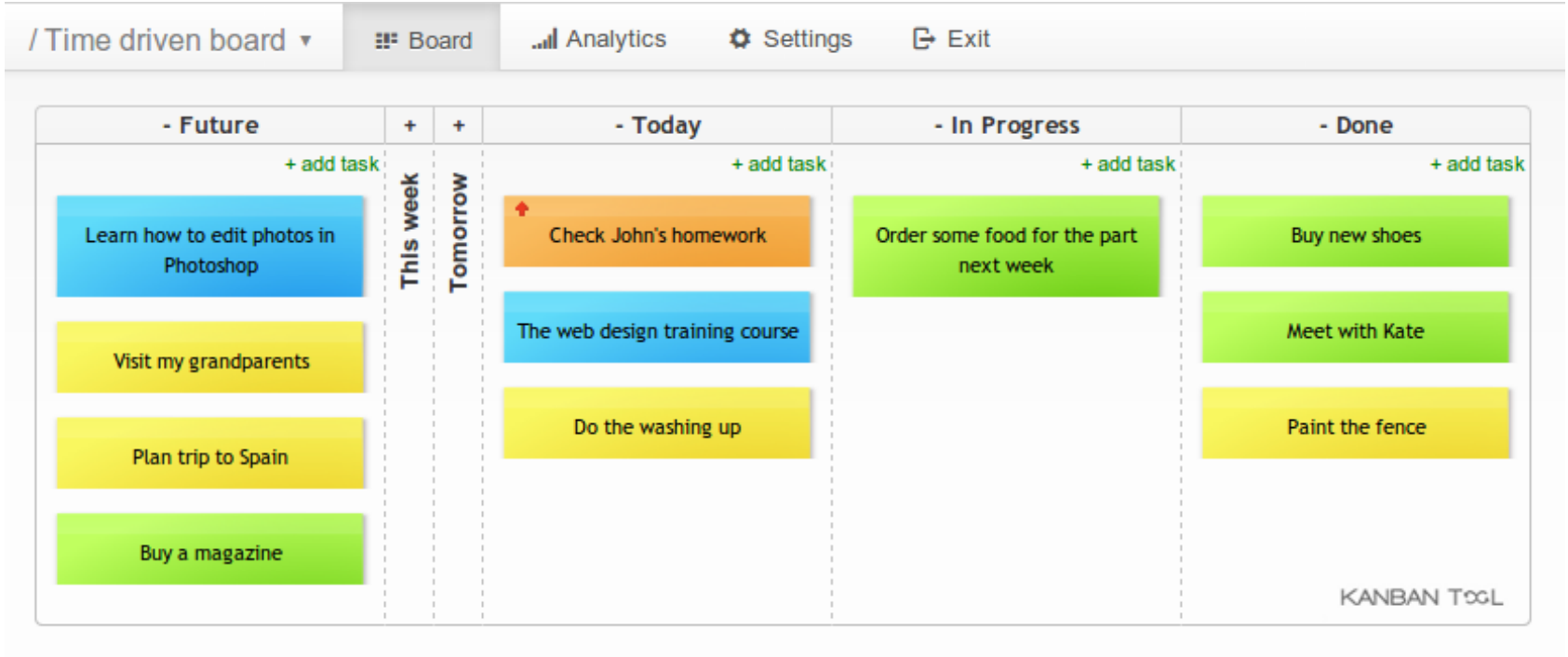
The Kanban Board

- The Kanban process begins with a prioritized list of features, called a backlog
- Agile team members work on the highest priority feature, based on their team role
- Getting features to completion in priority order is the focus
- There is no set timeframe to complete features



https://www.inflectra.com/methodologies/kanban.aspx

Online Kanban tool:



https://kanbantool.com/online-kanban-board

Ordering, Min, Max, and MinMax

Component Efficiency

- What's the criteria for a 'good' programming language?
  - You can implement general-purpose components (algorithms, data structures) that are universally usable
  - Without losing efficiency
- Efficiency
  - Relative efficiency
    - A component is **relatively efficient** if when instantiated it's as efficient as a non-generic written in the same language
  - Absolute efficiency
    - A component is **absolutely efficient** if when instantiated it is as efficient as anything which could be done on a given machine. Basically you know it's as fast as assembly language.

Abstraction Penalty

- The abstraction penalty is the ratio of runtime between a templated operation (say, `find` on a `vector<int>`) and the trivial nontemplated equivalent (say a loop over an array of `int`)
- Modern compilers have an abstraction penalty of 1-2%
- I still program in C++ because as far as I could ascertain it's the only language which allows me
  - Generality and absolute efficiency
  - I can program as general as I like
  - I can talk about things like monoids and semi-groups
  - When it compiles I could look at assembly code and see it is good. It is absolutely efficient

Abstraction Penalty

- Components that are sufficient to see if a language is efficient:
  - `swap`: takes two things and swaps them
  - `min`: takes two things and figures out which one is smaller
  - `linear search`: goes through a bunch of stuff and finds the one you want
- Are those too simple?
  - How can you solve complicated things **efficiently** if you can't solve those simple things?
  - Even simple things can be exciting

Swap

Swap

- Swap is a facility that – well swaps two values:

```
int a = 42, b = 43;
std::swap(a, b);
std::cout << "a: " << a << ", b: " << b << "\n";      // a: 43, b: 42
```

- Why is it important?
  - Sorting sequences
  - Reversing sequences
  - Rotating sequences
  - …
- Group theory has shown, that any permutation of a sequence can be produced by a sequence of swaps

General swap

```
template <std::semiregular T>
void swap_copy(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

- T must be semi-regular because of the copy construction
- This will work for any (semi-regular) type, however unfortunately, this generic solution will be slow for some types. Why?

Slow for some Types

- For instance, for std::vector<> this would be very slow
- What we need is something like:

```
template <std::semiregular T>
void swap(std::vector<T>& a, std::vector<T>& b) {
  // swap headers of a and b
  // fix back pointers, if appropriate (for things like linked lists)
}
```

Special case for (unsigned) integers

- General swap requires additional memory location, can we do better?

```
template <std::unsigned_integral T>
void swap_xor(T& a, T& b)
{
    // if a is identical to b the result is always 0
    if (&a != &b) {
        a = a ^ b;
        b = a ^ b;
        a = a ^ b;
    }
}
```

- Why only unsigned integers?

Special case for (unsigned) integers

- The ^ symbol is bitwise exclusive or. The expression a ^ b means a is true, or b is true, but not both. It is defined by the following truth table:

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

swap_xor: Proof

- We need to use the fact that XOR is associative and commutative.

1. $a = (a \wedge b)$	$b = b$	`a = a ^ b;`
2. $a = (a \wedge b)$	$b = (a \wedge b) \wedge b$ $= a \wedge (b \wedge b)$ $= a \wedge 0$ $= a$	`b = a ^ b;`
3. $a = (a \wedge b) \wedge a$ $= b \wedge (a \wedge a)$ $= b \wedge 0$ $a = b$	$b = a$	`a = a ^ b;`

Really general swap

```
template <std::semiregular T>
void swap(T& a, T& b)
{
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

- C++11 introduced move semantics that are meant to do exactly what we need

Min & Max

Min & Max

- `min` and `max` are two facilities that return the smaller (larger) of two values
- C++ standard has both: `std::min` and `std::max`
- Based on total ordering or strict weak ordering criteria

StrictWeak and Total Ordering

- A `StrictWeaklyOrdered` is a Binary Predicate that compares two objects, returning true if the first precedes the second
  - Invoke function on an element and totally order what it returns
  - Applying `TotallyOrdered` to equivalence classes
- `StrictWeaklyOrdered`
  - Partial ordering:
    - Irreflexivity: `!f(x, x)`
    - Antisymmetry: `f(x, y)` ⇔ `!f(y, x)`
    - Transitivity: `f(x, y) && f(y, z)` ⇔ `f(x, z)`
  - Connectedness: `!f(a, b) && !f(b, a)` ⇔ `a ≅ b (equivalence)`
  - Transitivity of equivalence
    - if x ≅ *y* and y ≅ *z*, then x ≅ *z*
- `TotallyOrdered` is a Binary Predicate similar to `StrictWeaklyOrdered`
  - Connectedness: `!f(a, b) && !f(b, a)` ⇔ `a == b (equality)`
  - Transitivity of equality
    - if x == *y* and y == *z*, then x == *z*

Equivalence vs. Equality

- Equational reasoning must be applied:
  - Equivalence is reflexive, symmetric, and transitive:

$$a \cong a$$

$$a \cong b \Leftrightarrow b \cong a$$

$$(a \cong b) \wedge (b \cong c) \Leftrightarrow (a \cong c)$$

  - Equality implies substitutability:

$$\text{for any function f on T, a} == b \Rightarrow f(a) == f(b)$$

  - Inequality must be the negation of equality:

$$(a \neq b) \Leftrightarrow \neg(a == b)$$

  - There is a fundamental connection between < and ==.
    - If !(b < a) then it must be the case that b >= a.

Min

- We write an incorrect version and then fix it. We will start with `TotallyOrdered` because we are more familiar with it

```
template <std::totally_ordered T>
T const& min_flawed(T const& a, T const& b)
{
    if (a < b) {
        return a;
    }
    else {
        return b;
    }
}
```

- We need to pass things by `const` reference
  - Allow to invoke using constant objects while avoiding to copy things

Min, corrected version

- What should `min` return if the arguments are equal?
  - Does it matter?
- When we sort a range which is already sorted, we should not do any operations. Nothing should be swapped
- When we sort two things, the first guy should be the min afterwards and the second guy should be the max
- Default ordering should be ascending (one, two, three, …)
- We can conclude that `min` should return the first argument if both are equal

Sort2

- We previously said that `min`, `max`, and `sort` should work together naturally.
  - To see how they should all work, let's write `sort2`, which sorts two things

```
template <typename T, typename Compare>
  requires(std::strict_weak_order<Compare, T, T>)
void sort2(T& a, T& b, Compare cmp)
{
    if (cmp(b, a))
    {
        swap(a, b);
    }
}
```

- Sorting in-place is preferable
- Order of comparison same as for `min`
  - After calling `sort2`, `a` should be the minimum and `b` should be the maximum
  - If `a` and `b` are equivalent, we don't want to swap them

Min, corrected version

- Let's avoid swapping values unnecessarily

```
template <std::totally_ordered T>
T const& min(T const& a, T const& b)
{
    if (b < a) {
        return b;
    }
    else {
        return a;
    }
}
```

Min, corrected, final version

- Let's avoid swapping values unnecessarily

```
template <typename T, typename Compare>
  requires(std::strict_weak_order<Compare, T, T>)
T const& min(T const& a, T const& b, Compare cmp)
{
    if (cmp(b, a)) {
        return b;
    }
    return a;
}
```

- Why do we pass `cmp` as a generic type argument instead of a function pointer?
  - Functionality: allows for function objects
  - Performance: compiler can optimize function call overhead

Min, back to TotallyOrdered

- `TotallyOrdered` version, using `std::less` function object:

```
template <std::totally_ordered T>
T const& min(T const& a, T const& b)
{
     return min(a, b, std::less<T>()); // std::less<T>() is an object
}
```

- Possible implementation of `std::less`:

```
template <std::totally_ordered T>
struct less
{
    bool operator()(T const& a, T const& b) const
    {
        return a < b;
    }
};
```

Max

- Implementing max correctly is hard
  - It seems to be just calling `min` using `operator>()`

```
template <std::totally_ordered T>
T const& max(T const& a, T const& b)
{
     return min(a, b, std::greater<T>());
}
```

  - One couldn't be more wrong about this! Why?

Max

- So, if we define `max` with `min` and `>` it's not going to work. Suppose we have `a ≅ b` (they are both equivalent), then
  - `min(a, b)` → a
  - `max(a, b)` → a
- But we want `min` and `max` to do different things because they're both useful

```
template <typename T, typename Compare>
  requires(std::strict_weak_order<Compare, T, T>)
T const& max(T const& a, T const& b, Compare cmp)
{
    if (cmp(b, a)) {
        return a;
    }
    return b;
}
```

Max, TotallyOrdered

- For the sake of completeness:
  - `TotallyOrdered` version, using `std::less` function object:

```
template <std::totally_ordered T>
T const& max(T const& a, T const& b)
{
     return max(a, b, std::less<T>()); // std::less<T>() is an object
}
```

Min, Max, and MinMax

Min Element in a Range

- Let's find the min element in a range of values
  - (Suppose we want to find an item with smallest price)
- `std::min_element`: simple algorithm, but hidden details

Min Element in a Range

```
template <std::forward_iterator I, typename Compare>
    requires(std::strict_weak_order<Compare, std::iter_value_t<I>, std::iter_value_t<I>>)
I min_element(I first, I last, Compare cmp)
{
    if (first == last)
        return last;

    I min_el = first;
    ++first;

    while (first != last) {
        if (cmp(*first, *min_el)) {
            min_el = first;
        }
        ++first;
    }

    return min_el;
}
```

Min Element in a Range

- Note: this returns an iterator referring to the min element. Why?
- Because we probably want to update the value
  - E.g. you want to find the worst performing employee and fire him
  - You don't need the person itself, you need a handle on him/her
  - Often, you want to do things with what you found!
- Because the sequence might be empty
  - What would you return in that case?
  - The algorithm returns the end iterator
- Finds first of all minimal elements

Iterator Categories

Algorithms and Iterators

- Library defines five iterator categories
  - Each corresponds to a specific set of exposed container operations
  - Each library algorithm states what iterator category is compatible
    - Possible to understand what containers are usable with which algorithms
  - Each category corresponds to an access strategy for elements, also limits usable algorithms
    - For instance: single pass algorithms, or random access algorithms

Sequential Read-Only Access

- `std::find`:

```
template <std::input_iterator Iterator, typename T>
Iterator find(Iterator first, Iterator last, T const& val)
{
    for (/**/; first != last; ++first)
        if (*first == val)
            break;
    return first;
}
```

- Requires iterator operators: != (==), ++, * (for reading)
  - *Input iterators*
- Not possible to store a copy of the iterator 'to go back'
- All iterators we've seen so far are (at least) input iterators

Sequential Write-Only Access

- `std::copy`

```
template <std::input_iterator In, std::output_iterator Out>
Out copy(In begin, In end, Out dest)
{
    while (begin != end)
        *dest++ = *begin++;
    return dest;
}
```

- Required iterator operators: != (==), ++, * (for writing)
  - *Output Iterators*, example: std::back_inserter
- Implicit requirements:
  - Do not execute ++it more than once between assignments to *it
  - Do not assign a value to *it more than once without incrementing it
- Not possible to store a copy of the iterator to overwrite output

Sequential Read-Write Access

- `std::replace`

```
template <std::forward_iterator Fwd, typename X>
void replace(Fwd beg, Fwd end, X const& x, X const& y)
{
    while (beg != end) {
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}
```

- Required iterator operators: != (==), ++, * (for reading and writing), -> for member access
  - *Forward iterators*
- Storing copy of iterator possible! Very handy!

Reversible Access

- `std::reverse`

```
template <std::bidirectional_iterator BiDir>
void reverse(BiDir begin, BiDir end)
{
    while (begin != end) {
        --end;
        if (begin != end)
            swap(*begin++, *end);
    }
}
```

- Required iterator operators: !=, ==, ++, --, * (for reading and writing), -> for member access
  - *Bidirectional iterators*
- The standard-library container classes all support bidirectional iterators.
  - Except `std::forward_list` (singly linked list)

Random Access

- std::binary_search:

```
template <std::random_access_iterator Ran, class X>
bool binary_search(Ran begin, Ran end, X const& x)
{
    while (begin < end) {
        Ran mid = begin + (end - begin) / 2; // find the midpoint of the range

        // see which part of the range contains x; keep looking only in that part
        if (x < *mid)
            end = mid;
        else if (*mid < x)
            begin = mid + 1;
        else    // if we got here, then *mid == x so we're done
            return true;
    }
    return false;
}
```

Random Access

- Required iterator operators:
  - `!=` (`==`), `++`, `--`, `*` (for reading and writing), `->` for member access
  - Let's assume `p`, `q` are iterators, `n` is integer
    - `p + n`, `p - n`, and `n + p`
    - `p - q`
    - `p[n]` (equivalent to `*(p + n)`)
    - `p < q`, `p > q`, `p <= q`, and `p >= q`
- We've used `std::sort`
- Containers: `std::vector`, `std::string`
  - `std::list` is not random access (it's bidirectional), why?

Iterator Ranges

- Algorithms take pair of iterators
  - `c.begin()` – refers to first element
  - `c.end()` – refers to first element after last ('one off')
- Allows simple handling of empty ranges, both iterators point to the one off element
- Allows for comparing iterators for equality (`!=/==`), no need to define notion of iterators being larger/smaller than others
- Allows to indicate 'out of range', see url_beg(), where we returned end (off by one) iterator if nothing was found
- Only caveat for end iterators is that they can't be dereferenced

Input/Output Iterators

- Why are they separate from forward iterators?
  - (no container exposes them)
- One reason is
  - Not all iterators come from containers
    - `std::back_inserter`, usable with any container supporting `push_back`
    - Iterators bound to streams: using iterator operations allowing to access `std::istreams` and `std::ostreams`

Input/Output Iterators

- The input stream iterator is an input-iterator type named `std::istream_iterator`:

```
// read ints from the standard input and append them to v
std::vector<int> v;
std::copy(
    std::istream_iterator<int>(cin), std::istream_iterator<int>(),
    std::back_inserter(v));
```

- Stream iterators are templates!
  - Need to specify type to read from input
  - Same as for normal input operations, which are always typed as well
  - Default constructed `std::istream_iterator` denotes end of input

Input/Output Iterators

- The output stream iterator is an output-iterator type named `std::ostream_iterator`:

```
// write the elements of v each separated from the other
// by a space
std::copy(
    v.begin(), v.end(),
    std::ostream_iterator<int>(std::cout, " "));
```

- ostream_iterator is a template as well
  - Need to specify type of required output
  - Second argument is separator (defaults to no separation)

Min and max Element in a Range

- How many comparisons do we need to find the min of five elements?
  - Four.
- In general why do we need `n - 1` comparisons.
  - Why no more?
    - We don't need to compare an element with itself.
  - Why no fewer? Maybe we could do it in `n - 2` with a clever algorithm?
    - We need to look at all elements to know which one is the `min`
- How many comparisons do we need to find `min` and `max` together?
  - Obviously we could do `2n - 2`, but what about fewer6?

Min and max Element in a Range

- Instead of comparing each element of the sequence against running min and max
  - Compare two (adjacent) elements amongst themselves
  - Compare the smaller one against the running min
  - Compare the larger one against the running max
- How many operations?
  - Three comparisons for every two elements: $3n/2$

Min and max Element in a Range

```
template <std::forward_iterator I, typename Compare>
    requires(std::strict_weak_order<Compare, std::iter_value_t<I>,
        std::iter_value_t<I>>)
std::pair<I, I> minmax_element(I first, I last, Compare cmp)
{
    // handle empty sequence
    // handle sequence with one element
    // initialize running min and max
    // loop over elements updating min and max
    while (first != last && next != last) {
       //...
    }

    // handle possible leftover element
}
```

Min and max Element in a Range

- Handle empty sequence

```
template <std::forward_iterator I, typename Compare> requires(...)
std::pair<I, I> minmax_element(I first, I last, Compare cmp)
{
    // handle empty sequence
    if (first == last) {
        return std::make_pair(last, last);
    }

    // handle sequence with one element
    // initialize running min and max
    // loop over elements updating min and max
    while (first != last && next != last) {
        // ...
    }
    // handle possible leftover element
}
```

Min and max Element in a Range

- Handle sequence with one element

```
template <std::forward_iterator I, typename Compare> requires(...)
std::pair<I, I> minmax_element(I first, I last, Compare cmp)
{
    // handle empty sequence
    // handle sequence with one element
    I min_el = first;           // refers to current min
    ++first;
    if (first == last)
        return std::make_pair(min_el, min_el);

    // initialize running min and max
    // loop over elements updating min and max
    while (first != last && next != last) {
        // ...
    }
    // handle possible leftover element
}
```

Min and max Element in a Range

- Initialize running min and max

```
template <std::forward_iterator I, typename Compare> requires(...)
std::pair<I, I> minmax_element(I first, I last, Compare cmp)
{
    I min_el = first; ++first;       // refers to current min

    // initialize running min and max
    I max_el = first;       // refers to current max
    I next = first; ++next; // first and next refer to pair to examine

    if (cmp(*max_el, *min_el))
        std::swap(min_el, max_el);

    // loop over elements updating min and max
    while (first != last && next != last) {
        // ...
    }
    // handle possible leftover element
}
```

Min and max Element in a Range

- Loop over elements, updating min and max

```
template <std::forward_iterator I, typename Compare> requires(...)
std::pair<I, I> minmax_element(I first, I last, Compare cmp)
{
    I min_el = first; ++first;    // refers to current min
    I max_el = first;             // refers to current max
    I next = first; ++next;       // first and next are pair to examine

    // loop over elements updating min and max
    while (first != last && next != last) {
        I potential_min = first, potential_max = next;

        if (cmp(*potential_max, *potential_min))       // potential_min refers to smaller of the two
            std::swap(potential_max, potential_min);
        if (cmp(*potential_min, *min_el))              // update current min
            min_el = potential_min;
        if (!cmp(*potential_max, *max_el))             // update current max
            max_el = potential_max;

        first = next; ++first; ++next;                 // go to next pair of elements
    }
    // handle possible leftover element
}
```

Min and max Element in a Range

- Handle possible leftover element

```
template <std::forward_iterator I, typename Compare> requires(...)
std::pair<I, I> minmax_element(I first, I last, Compare cmp)
{
    I min_el = first;       // refers to current min
    I max_el = first;       // refers to current max
    I next = first; ++next; // first and next are pair to examine
    while (first != last && next != last) { ... }

    // handle possible leftover element
    if (first != last) {        // odd elements, one left over
        if (cmp(*first, *min_el))
            min_el = first;
        else if (!cmp(*first, *max_el))
            max_el = first;
    }
    return std::make_pair(min_el, max_el);
}
```

Min and max Element in a Range

- This algorithm was invented by Ira Pohl of UC Santa Cruz, in the mid-seventies
  - He also proved it was optimal
  - `std::minmax_element` is part of the C++ standard since C++11

Min and max Element in a Range

Comparisons			
n	minmax_simple	minmax	gain(%)
64	1.97	1.5	24
128	1.98	1.5	24
256	1.99	1.5	25
512	2	1.5	25
1024	2	1.5	25
2048	2	1.5	25
4096	2	1.5	25
8192	2	1.5	25
16384	2	1.5	25
32768	2	1.5	25
65536	2	1.5	25
131072	2	1.5	25
262144	2	1.5	25
524288	2	1.5	25
1048576	2	1.5	25
2097152	2	1.5	25
4194304	2	1.5	25
8388608	2	1.5	25
16777216	2	1.5	25
33554432	2	1.5	25
67108864	2	1.5	25

Comparisons per iteration

Times			
n	minmax_simple	minmax	gain(%)
64	3.67	1.02	72
128	4.24	1.13	73
256	4.31	1.08	75
512	4.41	1.08	76
1024	4.43	1.11	75
2048	4.43	1.1	75
4096	4.42	1.13	74
8192	4.42	1.12	75
16384	4.46	1.35	70
32768	4.63	1.14	75
65536	4.61	1.17	75
131072	4.79	1.17	76
262144	4.68	1.17	75
524288	4.65	1.14	75
1048576	4.92	1.21	75
2097152	4.82	1.24	74
4194304	4.88	1.18	76
8388608	4.79	1.16	76
16777216	4.98	1.35	73
33554432	4.7	1.36	71
67108864	4.7	1.18	75

ns/iteration

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP