Lecture Outline

- Background and Motivation
- Design
- Examples
- Semantics
- Implementation
- Future Works
- Availability
Interval containers simplified the implementation of date and time related tasks

- Decomposing "histories" of attributed events into segments with constant attributes.
- Working with time grids, e.g. a grid of months.
- Aggregations of values associated to date or time intervals.

… that occurred frequently in programs like

- Billing modules
- Therapy scheduling programs
- Hospital and controlling statistics
Background is the date time problem domain ...

... but the scope of the **Itl** as a generic library is more general:

- An **interval_set** is a set
  that is implemented as a set of intervals

- An **interval_map** is a map
  that is implemented as a map of interval value pairs
There are two aspects in the design of interval containers

**Fundamental aspect**

```
interval_set<int> mySet;
mySet.insert(42);
bool has_answer = mySet.contains(42);
```

- On the fundamental aspect an interval_set can be used just as a set of elements
- Set theoretic operations are supported
- Abstracts from sequential and segmental information

**Segmental aspect**

- Allows to access and iterate over the *segments* of interval containers
Addability and Subtractability

- All of itl's (interval) containers are *Addable* and *Subtractable*
- They implement *operators* `+=`, `+`, `-=` and `-`

<table>
<thead>
<tr>
<th></th>
<th><code>+=</code></th>
<th><code>-=</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>sets</td>
<td>set union</td>
<td>set difference</td>
</tr>
<tr>
<td>maps</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

A possible implementation for maps

- Propagate addition/subtraction to the associated values
- . . . or aggregate on overlap
- . . . or aggregate on collision
Aggregate on overlap

\[ I - J \rightarrow a \]
\[ I \cap J \rightarrow (a + b) \]
\[ J - I \rightarrow b \]

- Decompositional effect on Intervals
- Accumulative effect on associated values

\( I, J \): intervals, \( a, b \): associated values
Aggregate on overlap, a minimal example

typedef itl::set<string> guests;
interval_map<time, guests> party;

party += make_pair(
    interval<time>::rightopen(20:00, 22:00), guests("Mary"));

party += make_pair(
    interval<time>::rightopen(21:00, 23:00), guests("Harry"));

// party now contains
[20:00, 21:00)->{"Mary"}  
[21:00, 22:00)->{"Harry","Mary"}  //guest sets aggregated  
[22:00, 23:00)->{"Harry"}
### The Itl's class templates

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Style</th>
<th>Sets</th>
<th>Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td></td>
<td>interval</td>
<td></td>
</tr>
<tr>
<td>joining</td>
<td>interval_set</td>
<td>interval_map</td>
<td></td>
</tr>
<tr>
<td>separating</td>
<td>separate_interval_set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>splitting</td>
<td>split_interval_set</td>
<td>split_interval_map</td>
<td></td>
</tr>
<tr>
<td>element</td>
<td>set</td>
<td>map</td>
<td></td>
</tr>
</tbody>
</table>
Interval Combining Styles: **Joining**

- Intervals are joined on overlap or on touch
- . . . *for maps*, if associated values are equal
- Keeps interval_maps and sets in a minimal form

**interval_set**

\[
\begin{align*}
\{ [1 & 3) \} \\
+ & [2 4) \\
+ & [4 5) \\
= & \{ [1 & 4) \} \\
= & \{ [1 & 5) \}
\end{align*}
\]

**interval_map**

\[
\begin{align*}
\{ [1 & 3) \rightarrow 1 \} \\
+ & [2 4) \rightarrow 1 \\
+ & [4 5) \rightarrow 1 \\
= & \{ [1 2) & [2 3) & [3 4) \\
\rightarrow 1 & \rightarrow 2 & \rightarrow 1 \\
= & \{ [1 2) & [2 3) & [3 & 5) \\
\rightarrow 1 & \rightarrow 2 & \rightarrow 1
\end{align*}
\]
Interval Combining Styles: **Splitting**

- Intervals are split on overlap and kept separate on touch
- All interval borders are preserved (insertion memory)

\[
\text{split\_interval\_set} = \{ [1, 3), [2, 4), [4, 5) \} + \{ [1, 2), [2, 3), [3, 4) \} = \{ [1, 2), [2, 3), [3, 4), [4, 5) \}
\]

\[
\text{split\_interval\_map} = \{ [1, 3) \rightarrow 1 \} + \{ [2, 4) \rightarrow 1 \} + \{ [4, 5) \rightarrow 1 \} = \{ [1, 2), [2, 3), [3, 4), [4, 5) \} \rightarrow 1 \rightarrow 2 \rightarrow 1
\]
Interval Combining Styles: **Separating**

- Intervals are joined on overlap but kept separate on touch
- Preserves borders that are never crossed (preserves a hidden grid).

\[
\text{separate\_interval\_set}
\begin{align*}
&= \{ [1, 3) \} \\
+& [2, 4) \\
+& [4, 5) \\
= & \{ [1, 4) \} \\
= & \{ [1, 4) [4, 5) \}
\end{align*}
\]
Examples

A few instances of intervals (interval.cpp)

```cpp
interval<int> int_interval = interval<int>::\texttt{closed}(3, 7);

interval<double> sqrt_interval
    = interval<double>::\texttt{rightopen}(1/sqrt(2.0), sqrt(2.0));

interval<std::string> city_interval
    = interval<std::string>::\texttt{leftopen}("Barcelona", "Boston");

interval<boost::ptime> time_interval
    = interval<boost::ptime>::\texttt{open}(
        time_from_string("2008-05-20 19:30"),
        time_from_string("2008-05-20 23:00")
    );
```
A way to iterate over months and weeks
(month_and_week_grid.cpp)

```cpp
#include <boost/itl/boost/itl/boost:gregorian.hpp>  //boost::gregorian plus adapter code
#include <boost/itl/boost/itl/split_interval_set.hpp>

// A split_interval_set of gregorian dates as date_grid.
typedef split_interval_set< boost::gregorian::date > date_grid;

// Compute a date_grid of months using boost::gregorian.
date_grid month_grid( const interval< date >& scope )
{
    date_grid month_grid;
    // Compute a date_grid of months using boost::gregorian.
    ... return month_grid;
}

// Compute a date_grid of weeks using boost::gregorian.
date_grid week_grid( const interval< date >& scope )
{
    date_grid week_grid;
    // Compute a date_grid of weeks using boost::gregorian.
    ... return week_grid;
}
```
A way to iterate over months and weeks

```cpp
void month_and_time_grid()
{
    date someday = day_clock::local_day();
    date thenday = someday + months(2);
    interval<date> scope = interval<date>::rightopen(someday, thenday);

    // An intersection of the month and week grids ...
    date_grid month_and_week_grid
        = month_grid(scope) & week_grid(scope);

    // ... allows to iterate months and weeks. Whenever a month
    // or a week changes there is a new interval.
    for (date_grid::iterator it = month_and_week_grid.begin();
        it != month_and_week_grid.end(); it++)
    {
        . . .   }

    // We can also intersect the grid into an interval_map to make
    // shure that all intervals are within months and week bounds.
    interval_map<boost::gregorian::date, some_type> accrual;
    compute_some_result(accrual, scope);
    accrual &= month_and_week_grid;
}
```
Aggregating with interval_maps

Computing averages via implementing `operator +=`
(partys_guest_average.cpp)

```cpp
class counted_sum {
public:
    counted_sum(): _sum(0), _count(0) {}  
    counted_sum(int sum): _sum(sum), _count(1) {}  

    int sum() const { return _sum; }
    int count() const { return _count; }
    double average() const
    { return _count == 0 ? 0.0 : _sum/static_cast<double>(_count); }

    counted_sum& operator += (const counted_sum& right)
    { _sum += right.sum(); _count += right.count(); return *this; }

private:
    int _sum;  
    int _count;
};

bool operator == (const counted_sum& left, const counted_sum& right)
{ return left.sum() == right.sum() && left.count() == right.count(); }
```
Aggregating with interval_maps

Computing averages via implementing \texttt{operator +=}

```cpp
void partys_height_average()
{
    interval_map<ptime, counted_sum> height_sums;

    height_sums += (make_pair(
                      interval<ptime>::rightopen(
                          time_from_string("2008-05-20 19:30"),
                          time_from_string("2008-05-20 23:00")),
                      counted_sum(165)) // Mary is 1,65 m tall.
                      );

    // Add height of more pary guests . . .

    interval_map<ptime, counted_sum>::iterator height_sum_ = height_sums.begin();
    while (height_sum_ != height_sums.end())
    {
        interval<ptime> when = height_sum_ -> first;
        double height_average = (*height_sum_++) . second . average();

        cout << "[" << when . first() << " - " << when . upper() << ""
        << ": " << height_average << " cm" << endl;
    }
}
```
Interval containers allow to express a variety of date and time operations in an easy way.

Example `man_power.cpp` ...

Subtract weekends and holidays from an interval_set

```
worktime -= weekends(scope)
worktime -= german_reunification_day
```

Intersect an interval_map with an interval_set

```
claudias_working_hours &= worktime
```

Subtract and interval_set from an interval_map

```
claudias_working_hours -= claudias_absense_times
```

Adding interval_maps

```
interval_map<date,int> manpower;
manpower += claudias_working_hours;
manpower += bodos_working_hours;
```
Interval_maps can also be intersected

Example user_groups.cpp

typedef boost::itl::set<string> MemberSetT;
typedef interval_map<date, MemberSetT> MembershipT;

void user_groups()
{
  . . .

  MembershipT med_users;
  // Compute membership of medical staff
  med_users += make_pair(member_interval_1, MemberSetT("Dr.Jekyll"));
  med_users += . . .

  MembershipT admin_users;
  // Compute membership of administation staff
  med_users += make_pair(member_interval_2, MemberSetT("Mr.Hyde"));
  . . .

  MembershipT all_users = med_users + admin_users;
  MembershipT super_users = med_users & admin_users;
  . . .
}
The semantics of *itl sets* is based on a concept *itl::Set*

*itl::set*, *interval_set*, *split_interval_set* and *separate_interval_set* are models of concept *itl::Set*

```
// Abstract part
empty set:       Set::Set()
subset relation: bool Set::contained_in(const Set& s2)const
equality:        bool is_element_equal(const Set& s1, const Set& s2)
set union:       Set& operator += (Set& s1, const Set& s2)
                Set operator +  (const Set& s1, const Set& s2)
set difference: Set& operator -= (Set& s1, const Set& s2)
                Set operator -  (const Set& s1, const Set& s2)
set intersection: Set& operator &= (Set& s1, const Set& s2)
                Set operator &  (const Set& s1, const Set& s2)

// Part related to sequential ordering
sorting order:  bool operator <  (const Set& s1, const Set& s2)
lexicographical equality:
                bool operator == (const Set& s1, const Set& s2)
```
The semantics of `itl maps` is based on a concept `itl::Map`.

`itl::map`, `interval_map` and `split_interval_map` are models of concept `itl::Map`.

```cpp
// Abstract part
empty map:   Map::Map()
submap relation: bool Map::contained_in(const Map& m2)const
equality:    bool is_element_equal(const Map& m1, const Map& m2)
map union:   Map& operator += (Map& m1, const Map& m2)
             Map operator +  (const Map& m1, const Map& m2)
map difference: Map& operator -= (Map& m1, const Map& m2)
                Map operator -  (const Map& m1, const Map& m2)
map intersection: Map& operator &== (Map& m1, const Map& m2)
                  Map operator &  (const Map& m1, const Map& m2)

// Part related to sequential ordering
sorting order: bool operator <  (const Map& m1, const Map& m2)
lexicographical equality:
                  bool operator == (const Map& m1, const Map& m2)
```
Defining semantics of ITL concepts via sets of laws

aka C++0x axioms

Checking law sets via automatic testing:

A Law Based Test Automaton LaBatea

Commutativity\(<T a, U b, +>: a + b = b + a;\)
Lexicographical Ordering and Equality

For all itl containers `operator <` implements a *strict weak ordering*.

The *induced equivalence* of this ordering is *lexicographical equality* which is implemented as `operator ==`

This is in line with the semantics of SortedAssociativeContainers
Subset Ordering and Element Equality

For all itl containers function `contained_in` implements a *partial ordering*.

The *induced equivalence* of this ordering is *equality of elements* which is implemented as function `is_element_equal`.
itl::Sets

All itl sets implement a Set Algebra, which is to say satisfy a “classical” set of laws . . .

- . . . using is_element_equal as equality
- Associativity, Neutrality, Commutativity (for + and &)
- Distributivity, DeMorgan, Symmetric Difference

Most of the itl sets satisfy the classical set of laws even if . . .

- . . . lexicographical equality: operator == is used
- The differences reflect proper inequalities in sequence that occur for separate_interval_set and split_interval_set.
Semantics

Concept Induction / Concept Transition

- The semantics of itl::Maps appears to be *determined* by the *codomain type* of the map

<table>
<thead>
<tr>
<th>is model of</th>
<th>if</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map&lt;D, Monoid&gt;</td>
<td>Monoid</td>
<td>interval_map&lt;int,string&gt;</td>
</tr>
<tr>
<td>Map&lt;D, Set&gt;</td>
<td>Set</td>
<td>C1 interval_map&lt;int,set&lt;int&gt;&gt;</td>
</tr>
<tr>
<td>Map&lt;D, CommutMonoid&gt;</td>
<td>CommutMonoid</td>
<td>interval_map&lt;int,unsigned&gt;</td>
</tr>
<tr>
<td>Map&lt;D, AbelianGroup&gt;</td>
<td>AbelianGroup</td>
<td>C2 interval_map&lt;int,int,total&gt;</td>
</tr>
</tbody>
</table>

- Conditions *C1* and *C2* restrict the *Concept Induction* to specific *map traits*
  - **C1**: Value pairs that carry a *neutral element* as associated value are always deleted (Trait: *absorbs_neutrons*).
  - **C2**: The map is *total*: Non existing keys are implicitly mapped to neutral elements (Trait: *is_total*).
Itl containers are implemented based on std::set and std::map

Basic operations like adding and subtracting intervals or interval value pairs perform with a time complexity between* amortized $O(\log n)$ and $O(n)$, where $n$ is the number of intervals of a container.

Operations like addition and subtraction of whole containers are having a worst case complexity of $O(m \log(n+m))$, where $n$ and $m$ are the numbers of intervals of the containers to combine.

* : Consult the library documentation for more detailed information.
Future Works

Implementing interval_maps of sets more efficiently

Revision of features of the extended itl (itl_plus.zip)

- Decomposition of histories: k histories $h_k$ with attribute types $A_1, ..., A_k$ are “decomposed” to a product history of tuples of attribute sets:
$$ (h_1<T,A_1>,..., h<T,A_k>) \rightarrow h<T, (\text{set}<A_1>,..., \text{set}<A_k>)> $$

- Cubes (generalized crosstables): Applying aggregate on collision to maps of tuple value pairs in order to organize hierachical data and their aggregates.
Availability

- Itl project on sourceforge (version 2.0.1)
  http://sourceforge.net/projects/itl
- Latest version on boost vault/Containers (3.2.0)
  http://www.boostpro.com/vault/ → containers
  - itl_3_2_0.zip : Core itl in preparation for boost
  - itl_plus_3_2_0.zip : Extended itl including histories, cubes and automatic validation (LaBatea).
- Online documentation at
  http://www.herold-faulhaber.de/
  - Doxygen generated docs for (version 2.0.1)
    http://www.herold-faulhaber.de/itl/
  - Latest boost style documentation (version 3.2.0)
Availability

- Boost sandbox
  https://svn.boost.org/svn/boost/sandbox/itl/

- Core itl: Interval containers proposed for boost
  https://svn.boost.org/svn/boost/sandbox/itl/boost/itl/
  https://svn.boost.org/svn/boost/sandbox/itl/libs/itl/

- Extended itl_xt: interval_bitset, “histories”, cubes
  https://svn.boost.org/svn/boost/sandbox/itl/boost/itl_xt/
  https://svn.boost.org/svn/boost/sandbox/itl/libs/itl_xt/

- Validater LaBatea:
  Compiles with msvc-8.0 or newer, gcc-4.3.2 or newer
  https://svn.boost.org/svn/boost/sandbox/itl/boost/validate/
  https://svn.boost.org/svn/boost/sandbox/itl/libs/validate/
An Introduction to the Interval Template Library

Lecture
held at the Boost Library Conference 2009

2009-05-08
Updated version 3.2.0  2009-12-02
Lecture Outline

- Background and Motivation
- Design
- Examples
- Semantics
- Implementation
- Future Works
- Availability
Background and Motivation

Interval containers simplified the implementation of date and time related tasks

- Decomposing “histories” of attributed events into segments with constant attributes.
- Working with time grids, e.g. a grid of months.
- Aggregations of values associated to date or time intervals.

... that occurred frequently in programs like

- Billing modules
- Therapy scheduling programs
- Hospital and controlling statistics
Background is the date time problem domain …

… but the scope of the Itl as a generic library is more general:

- **an interval_set is a set**
  that is implemented as a set of intervals

- **an interval_map is a map**
  that is implemented as a map of interval value pairs
There are two aspects in the design of interval containers

**Fundamental aspect**

```cpp
interval_set<int> mySet;
mySet.insert(42);
bool has_answer = mySet.contains(42);
```

- On the fundamental aspect an interval_set can be used just as a set of elements
- Set theoretic operations are supported
- Abstracts from sequential and segmental information

**Segmental aspect**

- Allows to access and iterate over the *segments* of interval containers
Addability and Subtractability

- All of itl's (interval) containers are Addable and Subtractable
- They implement operators +=, +, -= and -

<table>
<thead>
<tr>
<th></th>
<th>+=</th>
<th>-=</th>
</tr>
</thead>
<tbody>
<tr>
<td>sets</td>
<td>set union</td>
<td>set difference</td>
</tr>
<tr>
<td>maps</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

A possible implementation for maps

- Propagate addition/subtraction to the associated values
- . . . or aggregate on overlap
- . . . or aggregate on collision
Aggregate on overlap

\[ I - J \rightarrow a \]
\[ I \cap J \rightarrow (a + b) \]
\[ J - I \rightarrow b \]

Decompositional effect on Intervals

Accumulative effect on associated values

\( I, J \): intervals, \( a, b \): associated values
Aggregate on overlap, a minimal example

typedef itl::set<string> guests;
interval_map<time, guests> party;

party += make_pair(
    interval<time>::rightopen(20:00, 22:00), guests("Mary"));

party += make_pair(
    interval<time>::rightopen(21:00, 23:00), guests("Harry"));

// party now contains
[20:00, 21:00)->{"Mary"}
[21:00, 22:00)->{"Harry","Mary"}  //guest sets aggregated
[22:00, 23:00)->{"Harry"}
<table>
<thead>
<tr>
<th>Granularity</th>
<th>Style</th>
<th>Sets</th>
<th>Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>interval</td>
<td></td>
<td></td>
</tr>
<tr>
<td>joining</td>
<td>interval_set</td>
<td>interval_map</td>
<td></td>
</tr>
<tr>
<td>separating</td>
<td>separate_interval_set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>splitting</td>
<td>split_interval_set</td>
<td>split_interval_map</td>
<td></td>
</tr>
<tr>
<td>element</td>
<td>set</td>
<td>map</td>
<td></td>
</tr>
</tbody>
</table>
Interval Combining Styles: \textit{Joining}

- Intervals are joined on overlap or on touch
- \textit{for maps}, if associated values are equal
- Keeps interval\_maps and sets in a minimal form

\begin{align*}
\text{interval\_set} & \quad \{[1, 3)\} + [2, 4) + [4, 5) = \{[1, 4)\} = \{[1, 5)\} \\
\text{interval\_map} & \quad \{1, 3) \rightarrow 1\} + [2, 4) \rightarrow 1 + [4, 5) \rightarrow 1 \\
& \quad = \{(1, 2) [2, 3) [3, 4) \rightarrow 1 \rightarrow 2 \rightarrow 1\} \\
& \quad = \{(1, 2) [2, 3) [3, 5) \rightarrow 1 \rightarrow 2 \rightarrow 1\}
\end{align*}
Interval Combining Styles: Splitting
- Intervals are split on overlap and kept separate on touch
- All interval borders are preserved (insertion memory)

\[
\text{split\_interval\_set} \\
\{ [1 3) \} + [2 4) + [4 5) = \{ [1 2) [2 3) [3 4) [4 5) \}
\]

\[
\text{split\_interval\_map} \\
\{ [1 3) \rightarrow 1 \} + [2 4) \rightarrow 1 + [4 5) \rightarrow 1 = \{ [1 2) [2 3) [3 4) \rightarrow 1 \rightarrow 2 \rightarrow 1 \}
\]
Interval Combining Styles: *Separating*

- Intervals are joined on overlap but kept separate on touch
- Preserves borders that are never crossed (preserves a hidden grid).

```
separate_interval_set
{{[1 3)}, {[2 4), [4 5)}}
+ [+ ]
= {{[1 4) }
= {{[1 4) [4 5) }}
```


A few instances of intervals (interval.cpp)

```cpp
interval<int> int_interval = interval<int>::closed(3,7);
interval<double> sqrt_interval
    = interval<double>::rightopen(1/sqrt(2.0), sqrt(2.0));
interval<std::string> city_interval
    = interval<std::string>::leftopen("Barcelona", "Boston");
interval<boost::ptime> time_interval
    = interval<boost::ptime>::open(
        time_from_string("2008-05-20 19:30"),
        time_from_string("2008-05-20 23:00")
    );
```
A way to iterate over months and weeks

(month_and_week_grid.cpp)

```cpp
#include <boost/itl/gregorian.hpp> // boost::gregorian plus adapter code
#include <boost/itl/split_interval_set.hpp>

// A split_interval_set of gregorian dates as date_grid.
typedef split_interval_set< boost::gregorian::date > date_grid;

// Compute a date_grid of months using boost::gregorian.
date_grid month_grid( const interval<date>& scope )
{
    date_grid month_grid;
    // Compute a date_grid of months using boost::gregorian.
    . . .
    return month_grid;
}

// Compute a date_grid of weeks using boost::gregorian.
date_grid week_grid( const interval<date>& scope )
{
    date_grid week_grid;
    // Compute a date_grid of weeks using boost::gregorian.
    . . .
    return week_grid;
}
```
A way to iterate over months and weeks

```cpp
void month_and_time_grid()
{
    date someday = day_clock::local_day();
    date thenday = someday + months(2);
    interval<date> scope = interval<date>::rightopen(someday, thenday);

    // An intersection of the month and week grids ...
    date_grid month_and_week_grid
        = month_grid(scope) & week_grid(scope);

    // ... allows to iterate months and weeks. Whenever a month
    // or a week changes there is a new interval.
    for(date_grid::iterator it = month_and_week_grid.begin();
        it != month_and_week_grid.end(); it++)
    {
        // We can also intersect the grid into an interval_map to make
        // shure that all intervals are within months and week bounds.
        interval_map<boost::gregorian::date, some_type> accrual;
        compute_some_result(accrual, scope);
        accrual &= month_and_week_grid;
    }
}
```
Examples

Aggregating with interval_maps
Computing averages via implementing operator +=
(partys_guest_average.cpp)

class counted_sum
{
public:
    counted_sum(): _sum(0), _count(0) {}
    counted_sum(int sum): _sum(sum), _count(1) {}
    int sum() const { return _sum; }
    int count() const { return _count; }
    double average() const
    { return _count==0 ? 0.0 : _sum/static_cast<double>(_count); }
    counted_sum& operator += (const counted_sum& right)
    { _sum += right.sum(); _count += right.count(); return *this; }
private:
    int _sum;
    int _count;
};

bool operator == (const counted_sum& left, const counted_sum& right)
{ return left.sum()==right.sum() && left.count()==right.count(); }
Aggregating with interval\_maps

Computing averages via implementing operator +=

```cpp
void party\_height\_average()
{
    interval\_map<ptime, counted\_sum> height\_sums;

    height\_sums += {
        make\_pair(
            interval<ptime>::rightopen(
                time\_from\_string("2008-05-20 19:30"),
                time\_from\_string("2008-05-20 23:00")),
            counted\_sum(165))  // Mary is 1,65 m tall.
    };

    // Add height of more pary guests . . .
    interval\_map<ptime, counted\_sum>::iterator height\_sum\_ =
        height\_sums.begin();
    while(height\_sum\_ != height\_sums.end())
    {
        interval<ptime> when = height\_sum\_->first;
        double height\_average = (*height\_sum\_++).second\_average();
        cout << "[" << when\_first() << " - " << when\_upper() << "]" << " cm" << endl;
    }
}
```
Examples

- Interval containers allow to express a variety of date and time operations in an easy way.
  - Example `man_power.cpp` ...
  - Subtract weekends and holidays from an interval_set
    ```cpp
    worktime -= weekends(scope)
    worktime -= german_reunification_day
    ```
  - Intersect an interval_map with an interval_set
    ```cpp
    claudias_working_hours &== worktime
    ```
  - Subtract an interval_set from an interval_map
    ```cpp
    claudias_working_hours -= claudias_absense_times
    ```
  - Adding interval_maps
    ```cpp
    interval_map<date,int> manpower;
    manpower += claudias_working_hours;
    manpower += bodos_working_hours;
    ```


**Examples**

Interval maps can also be intersected

**Example user_groups.cpp**

```cpp
typedef boost::itl::set<string> MemberSetT;
typedef interval_map<date, MemberSetT> MembershipT;

void user_groups()
{
    MembershipT med_users;
    // Compute membership of medical staff
    med_users += make_pair(member_interval_1, MemberSetT("Dr.Jekyll"));
    med_users += ...
    MembershipT admin_users;
    // Compute membership of administration staff
    med_users += make_pair(member_interval_2, MemberSetT("Mr.Hyde"));
    ...
    MembershipT all_users = med_users + admin_users;
    MembershipT super_users = med_users & admin_users;
    ...
}
```
The semantics of **itl sets** is based on a concept **itl::Set**

- **itl::set**, **interval_set**, **split_interval_set**
- and **separate_interval_set** are models of concept **itl::Set**

```cpp
// Abstract part
empty set:    Set::Set();
subset relation: bool Set::contains(const Set& s2)const
equality:     bool is_element_equal(const Set& s1, const Set& s2)
set union:    Set& operator += (Set& s1, const Set& s2)
set difference: Set& operator -= (const Set& s1, const Set& s2)
set intersection: Set& operator &=(const Set& s1, const Set& s2)

// Part related to sequential ordering
sorting order: bool operator < (const Set& s1, const Set& s2)
lexicographical equality: bool operator == (const Set& s1, const Set& s2)
```
The semantics of *itl maps* is based on a concept *itl::Map*

- *itl::map*, *interval_map* and *split_interval_map* are models of concept *itl::Map*

```cpp
// Abstract part
empty map: Map::Map()
submap relation: bool Map::contained_in(const Map& m2)const
equality: bool is_element_equal(const Map& m1, const Map& m2)
map union: Map& operator += (Map& m1, const Map& m2)
map difference: Map& operator -= (const Map& m1, const Map& m2)
map intersection: Map& operator &= (const Map& m1, const Map& m2)

// Part related to sequential ordering
sorting order: bool operator < (const Map& m1, const Map& m2)
lexicographical equality: bool operator == (const Map& m1, const Map& m2)
```
Semantics

- Defining semantics of ITL concepts via sets of laws
  - aka C++0x axioms
- Checking law sets via automatic testing:
  - A Law Based Test Automaton Labatea

Commutativity<T a, U b, +>:
\[ a + b = b + a; \]
Lexicographical Ordering and Equality

- For all `std::` containers `operator <` implements a **strict weak ordering**.

- The **induced equivalence** of this ordering is **lexicographical equality** which is implemented as `operator ==`

- This is in line with the semantics of `SortedAssociativeContainers`
Semantics

Subset Ordering and Element Equality

- For all itl containers function `contained_in` implements a *partial ordering*.

- The *induced equivalence* of this ordering is *equality of elements* which is implemented as function `is_element_equal`. 
Semantics

- Itl::Sets
- All Itl sets implement a Set Algebra, which is to say satisfy a "classical" set of laws . . .
  - . . . using is_element_equal as equality
  - Associativity, Neutrality, Commutativity (for + and \&)
  - Distributivity, DeMorgan, Symmetric Difference

- Most of the Itl sets satisfy the classical set of laws even if . . .
  - . . . lexicographical equality: operator == is used
  - The differences reflect proper inequalities in sequence that occur for separate_interval_set and split_interval_set.
Semantics

Concept Induction / Concept Transition

The semantics of itl::Maps appears to be determined by the codomain type of the map

<table>
<thead>
<tr>
<th>is model of</th>
<th>if</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map&lt;D, Monoid&gt;</td>
<td>Monoid</td>
<td>interval_map&lt;int, string&gt;</td>
</tr>
<tr>
<td>Map&lt;D, Set&gt;</td>
<td>Set</td>
<td>C1</td>
</tr>
<tr>
<td>Map&lt;D, CommutMonoid&gt;</td>
<td>CommutMonoid</td>
<td>interval_map&lt;int, unsigned&gt;</td>
</tr>
<tr>
<td>Map&lt;D, AbelianGroup&gt;</td>
<td>AbelianGroup</td>
<td>C2</td>
</tr>
</tbody>
</table>

Conditions C1 and C2 restrict the Concept Induction to specific map traits

- C1: Value pairs that carry a neutral element as associated value are always deleted (Trait: absorbs_neutrons).
- C2: The map is total: Non existing keys are implicitly mapped to neutral elements (Trait: is_total).
Itl containers are implemented based on `std::set` and `std::map`.

- Basic operations like adding and subtracting intervals or interval value pairs perform with a time complexity between* amortized $O(\log n)$ and $O(n)$, where $n$ is the number of intervals of a container.

- Operations like addition and subtraction of whole containers are having a worst case complexity of $O(m \log(n+m))$, where $n$ and $m$ are the numbers of intervals of the containers to combine.

*: Consult the library documentation for more detailed information.
Future Works

- Implementing interval_maps of sets more efficiently
- Revision of features of the extended itl (itl_plus.zip)
  - Decomposition of histories: \( k \) histories \( h_k \) with attribute types \( A_1, ..., A_k \) are "decomposed" to a product history of tuples of attribute sets:
    \[
    (h_{1\langle T,A_1\rangle}, ..., h_{T,A_k\rangle}) \rightarrow h_{T, (set\langle A_1\rangle, ..., set\langle A_k\rangle)}
    \]

- Cubes (generalized crosstables): Applying aggregate on collision to maps of tuple value pairs in order to organize hierachical data and their aggregates.
Availability

- Itl project on sourceforge (version 2.0.1)
  http://sourceforge.net/projects/itl
- Latest version on boost vault/Containers (3.2.0)
  http://www.boostpro.com/vault/ → containers
  - itl_3_2_0.zip : Core itl in preparation for boost
  - itl_plus_3_2_0.zip : Extended itl including histories, cubes and automatic validation (LaBatea).
- Online documentation at
  http://www.herold-faulhaber.de/
  - Doxygen generated docs for (version 2.0.1)
    http://www.herold-faulhaber.de/itl/
  - Latest boost style documentation (version 3.2.0)
Availability

- Boost sandbox
  https://svn.boost.org/svn/boost/sandbox/itl/

- Core itl: Interval containers proposed for boost
  https://svn.boost.org/svn/boost/sandbox/itl/boost/itl/
  https://svn.boost.org/svn/boost/sandbox/itl/libs/itl/

- Extended itl_xt: interval_bitset, “histories”, cubes
  https://svn.boost.org/svn/boost/sandbox/itl/boost/itl_xt/
  https://svn.boost.org/svn/boost/sandbox/itl/libs/itl_xt/

- Validater LaBatea:
  Compiles with msvc-8.0 or newer, gcc-4.3.2 or newer
  https://svn.boost.org/svn/boost/sandbox/itl/boost/validate/
  https://svn.boost.org/svn/boost/sandbox/itl/libs/validate/