

Introducing C++ View Objects



Gabhan Berry

XLCubed

gabhan.berry@xlcubed.com



Introduction

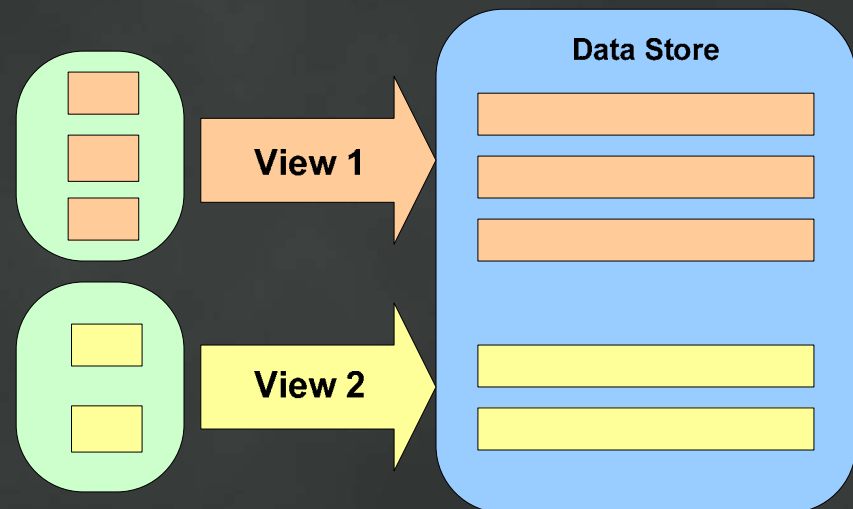
- The concept of a view of data is well known in database development.
- View concept can be applied to C++ and STL programming
 - This session shows you how

Session Contents

- The View Concept
- Designing a View class
 - Two approaches
- Building View objects
- Worked Examples
- Memory efficiency of Views
- Summary
- Resources and Links

The Concept of a View

- Views separate the data storage from the data presentation
- Views are layers of abstraction above the real data; they are perspectives of the data



The Concept of a View: Data Usage Models

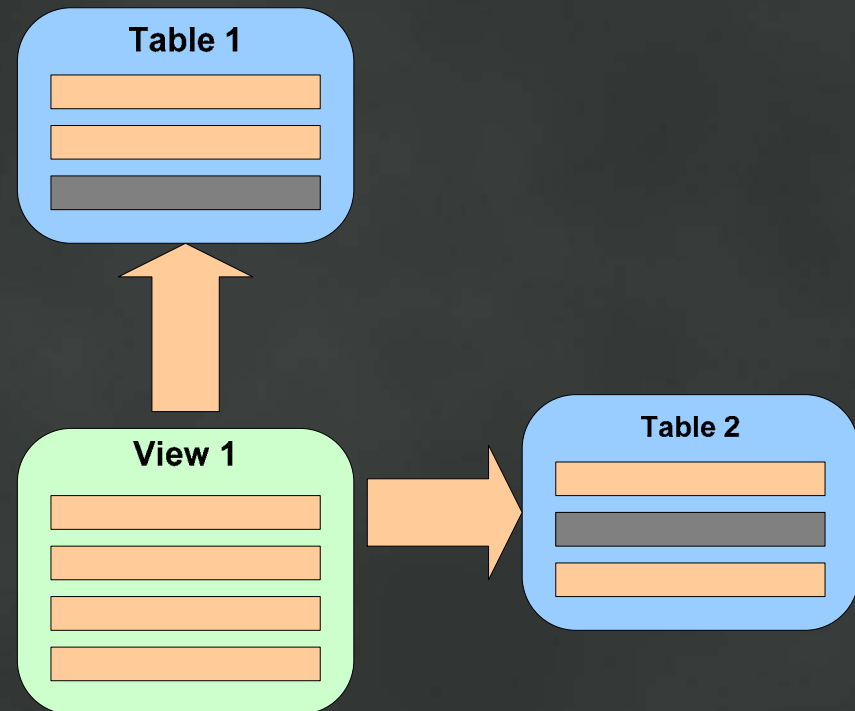
- **Views are Consumer Centric**
 - Designed with the data consumer in mind
- **Data Storage is Operational Centric**
 - Designed for run time efficiency/convenience
- **These two models don't always mix well**

Advantages of the View Concept

- **Combine Operational and Consumer Centric Models**
 - Data is stored in an optimised format but presented in ways most useful for consumers
- **Data integrity**
 - The same data can be viewed in multiple ways without copies of the data being made
- **Efficiency**
 - Fewer copies of data means less memory being used

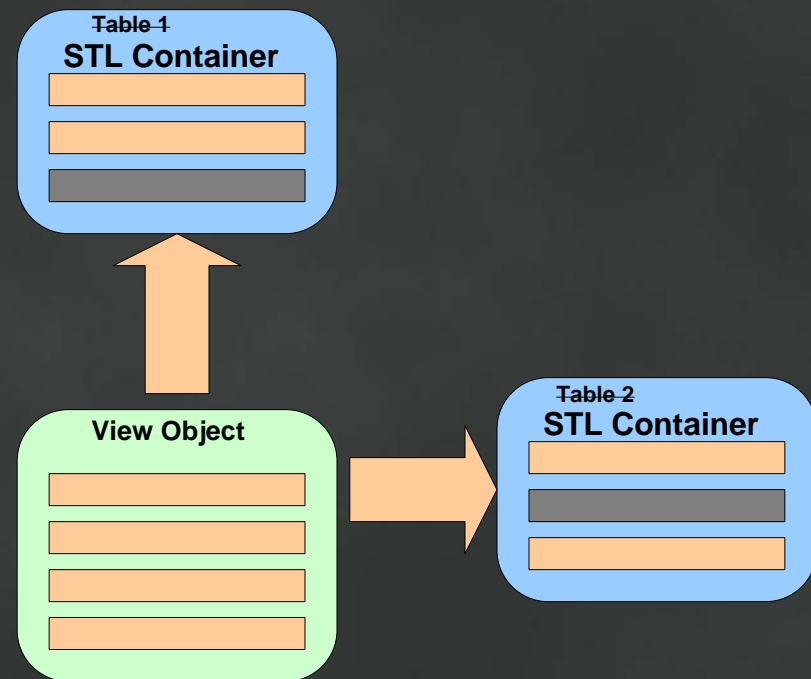
Real Life Views: Databases

- Data from multiple underlying tables is gathered into one coherent table-like structure



Views in C++ and the STL

- Where database views use tables, C++ views use STL containers



Views in C++ and the STL

- **Some places where views are useful:**
 - Consumers need only a subset of the data
 - Consumers need to see the data in multiple different ways
 - You need to order a container in multiple ways
 - Combining some data from one container with some data from others
 - You need to minimise memory usage

Designing a View Class: Two Approaches

- **Iterator Adaptor Approach**
 - Uses iterator adaptors, predicates and STL algorithms to build views
 - View contents are built by filtering the data as it is iterated over
- **Distinct View Object (subject of this session)**
 - Stores links to the underlying data in a specific View object
 - Only one class definition for all view types
 - Any algorithm or custom logic used to build view

Designing a View Class: Comparison of the Approaches

- **Iterator Adaptor Approach**
 - The View instance is an abstract/virtual object
 - Need to write iterator adaptors using predicates
 - Harder to “pick up and use” – especially for non STL gurus or junior programmers
 - Many View class definitions i.e. `transform_view<>`, `sorted_view<>` etc
- **Distinct View Object Approach**
 - The View instance is a distinct/actual object (like any other)
 - Only one View class definition
 - Easy to understand, use and extend (view hierarchies etc)
- Both approaches are based on the database view concept.
- *This session discusses the Distinct View Object Technique*

Designing a View Class: Requirements

- Contains links to the underlying data, not copies of it
- Grants access to the underlying data
- Can reference multiple underlying containers
- Is not restricted by the ordering of the underlying container

Designing a View Class

- A view contains a vector of links to the underlying container
- Call these links **view elements**.

```
template<
class _CTN,
class _ELEMENT = view_element< _CTN >
>
struct view{
    typedef typename _ELEMENT element_type;
    vector< element_type > elements;
};
```

- Vector gives us maximum flexibility in storing and using the view elements
- View is specialised by the underlying container type. See `view_element` for the reason (next slide)

Designing a View Class: view_element

- Each view_element stores an iterator and returns the real, underlying data using a value() method
- Container type (_CTN) is used to determine the iterator and the underlying value type.

```
template<class _CTN>
struct view_element{
    typename _CTN::iterator c_itr;
    typedef typename _CTN::value_type value_type;
    virtual const value_type& value() const{
        return *c_itr;
    }
};
```

Designing a View Class: `view::operator[]`

- Define `view::operator[]` to return the underlying data using the `view_element::value()` method
- Consumers access the underlying data using `view::elements` or `view::operator[]`

```
template<
class _CTN,
class _ELEMENT = view_element< _CTN >
>
struct view{
    typedef typename _ELEMENT element_type;
    vector< element_type > elements;
    typedef typename _CTN::value_type value_type;
    const value_type& operator[]( size_t index ) const {
        return elements.at( index ).value();
    }
};
```

Designing a View Class

- So to create a view object:

```
view<underlying container type> myView;  
i.e. view< vector<wstring> > myView;
```

- Container type is used to determine the iterator and the underlying value type.
- To access the data, use `view::operator[]` or `view::elements`.

```
typedef view< vector<wstring> > wstring_vec_view  
wstring_vec_view myView;
```

// Both of these return the same wstring from the underlying container.

```
wstring_vec_view::value_type val; //val is of type wstring  
val = myView[0];  
val = myView.elements[0].value();
```

Designing a View Class: Building and using Views

- For building views, we need ways of creating `view_element`s from iterators and a way to add elements into a view
 - Define `view_element` constructors and a `view::add()` method
- For using views, we need ways of converting `view_element`s to the underlying data type
 - Define operators to do so

Designing a View Class: Building and using Views

- `view_element` constructors and operators

```
template<class _CTN>
struct view_element{
    typename _CTN::iterator c_itr;
    typedef typename _CTN::value_type value_type;
    // .. other code omitted for clarity

    // Construct a view_element from an iterator
    view_element<_CTN>( typename _CTN::iterator& itr){
        c_itr = itr;
    }

    // Cast a view_element to the underlying data type
    operator value_type(){
        return value();
    }
}
```

Designing a View Class: Building and using Views

- `view::add()` method

```
template<
class _CTN,
class _ELEMENT = view_element< _CTN >
>
struct view{
    typedef typename _ELEMENT element_type;
    vector<_ELEMENT> elements;
    // ... other code omitted for clarity ...

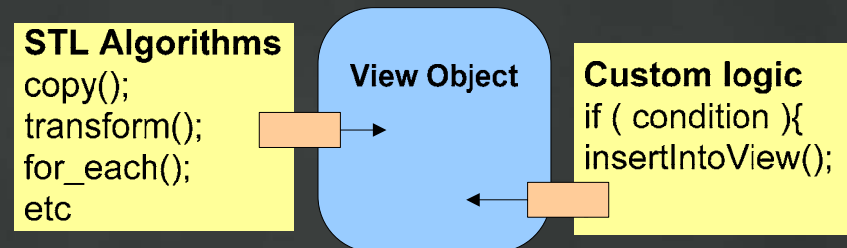
    // Add new view_elements to the view
    void add( typename _CTN::iterator& itr ){
        elements.push_back( element_type(itr) );
    }
};
```

Summary: Designing a View Class (complete source code in view.h)

- The view class contains a vector of `view_element` objects
- `view_element` objects encapsulate iterators and have a `value()` method that returns the real data via the iterator
- Views and `view_elements` are specialised by the underlying container type
- Consumers access the data container via the view object's `operator[]` or `elements` vector

Building View Objects

- Building the view is external to the view object itself.
- Use STL algorithms or custom logic to add `view_elements` into the view
- Following worked examples show how to build and use views



Building View Objects

- We can write a `build_view` function to build views from container ranges.

```
template<class _ITR, class _VIEW>
void build_view(_ITR first, _ITR last, _VIEW& v){
    for( ; first !=last; ++first ){
        v.add( first );
    }
}
```

Building View Objects

- Another `build_view` function using a function object to control insertion.

```
template<class _ITR, class _VIEW, class _FUNC>
void build_view(_ITR first, _ITR last, _VIEW& v, _FUNC func){
    for( ; first !=last; ++first ){
        if ( func( *first ) ){
            v.add( first );
        }
    }
}
```

Example: Simple View

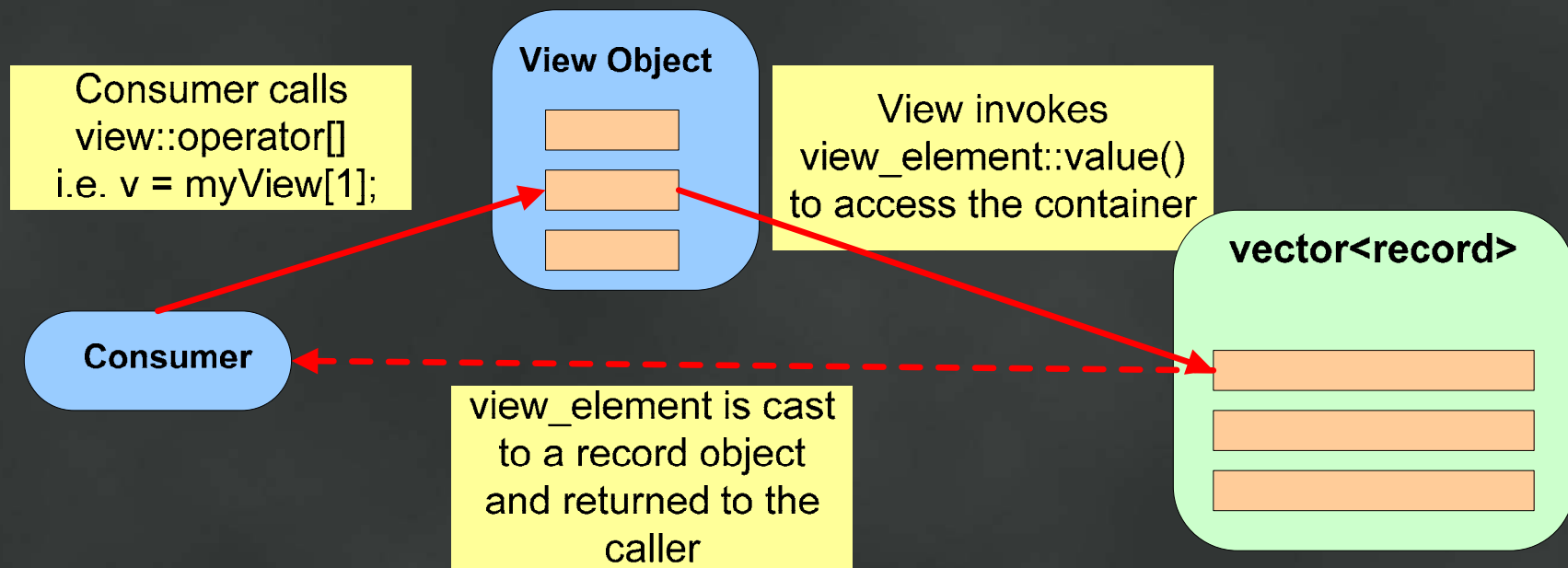
- Create a sorted view of the underlying container using custom sorting criteria
 - Underlying container will not be changed
 - Data will not be copied out of the container

Example: Simple View

- Underlying container is a `vector<record>` where `record` is defined as:

```
struct record{
    size_id          id;
    vector<size_t>  values;
    wstring          wstrDesc;
};
```

Example: Simple View



Example: Simple View

```
typedef vector<record> record_vector;  
  
// Define the underlying container  
record_vector rvec;  
  
// Define the view object  
view<record_vector> rvec_view;  
  
// Build a view of the entire container  
build_view(rvec.begin(),rvec.end(),rvec_view);  
  
// Sort the view  
sort(  
    rvec_view.elements.begin(),  
    rvec_view.elements.end(),  
    record_vecview_id_less-than()  
);
```

Example: Simple View

- The underlying vector was not changed and can be accessed as normal
- Sorted access can be presented via the view object
- The sorted data can be accessed via the view using `view::operator[]` or `view::elements`

```
record r = rvec_view[0]; //record object returned, not view_element
```

```
// Iterate over the sorted view
for_each(
  rvec_view.elements.begin(),
  rvec_view.elements.end(),
  outputRecord( false )
);
```

Example: Simple View

- Worked example `simple_view.h` in session's source code.

Example: Using Custom Functors

- Build a view of the `vector<record>` object that contains only the records whose values have a sum \geq some value X .
 - Shows how to use the `build_view` function and a custom functor to build a interesting view object.

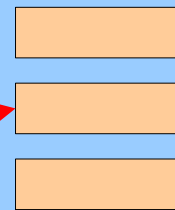
Example: Using Custom Functors

functor
bool operator()

Each element in the range is passed to the functor

If operator() returns true, element is added to the view

View Object



build_view

```
for ( ; first != last; ++first ){  
    if ( func( *first ) ){  
        v.add( first );  
    }  
}
```

Example: Using Custom Functors

- The call to `build_view` can have a functor passed in as the last parameter.
- Each record in the range is passed into the functor and only those that evaluate to true are added into the view.

```
build_view(  
    rvec.begin(),  
    rvec.end(),  
    rview,  
    sum_values_greaterthan( min_value )  
);
```

Example: Using Custom Functors

```
struct sum_values_greaterthan{
    size_t limit;

    sum_values_greaterthan( const size_t lim ){
        limit = lim;
    }

    bool operator()( record_vector::value_type& val ){
        return
            accumulate( val.values.begin(), val.values.end(), 0 ) >= limit;
    }
};
```

Example: Using Custom Functors

- Worked example
build_view_custom_functor.h in session's
source code.

Multi-Container Views

- Views can be built with any number of underlying containers
- Elements are inserted into the view in the same way as single container views
- Consumers of the view do not know about there being multiple underlying containers

Example: Simple Multi Container Views

```
// Build a view that contains the first 5 records from  
// two record_vector objects.
```

```
build_view(  
    rvec1.begin(),  
    rvec1.begin() + 5,  
    rview  
);
```

```
build_view(  
    rvec2.begin(),  
    rvec2.begin() + 5,  
    rview  
);
```

```
// rview now contains 10 elements. The first 5 come from rvec1 and the  
// second 5 come from rvec2.
```

Example: Simple Multi Container Views

- Worked example `simple_multi_container_view.h` in session's source code.

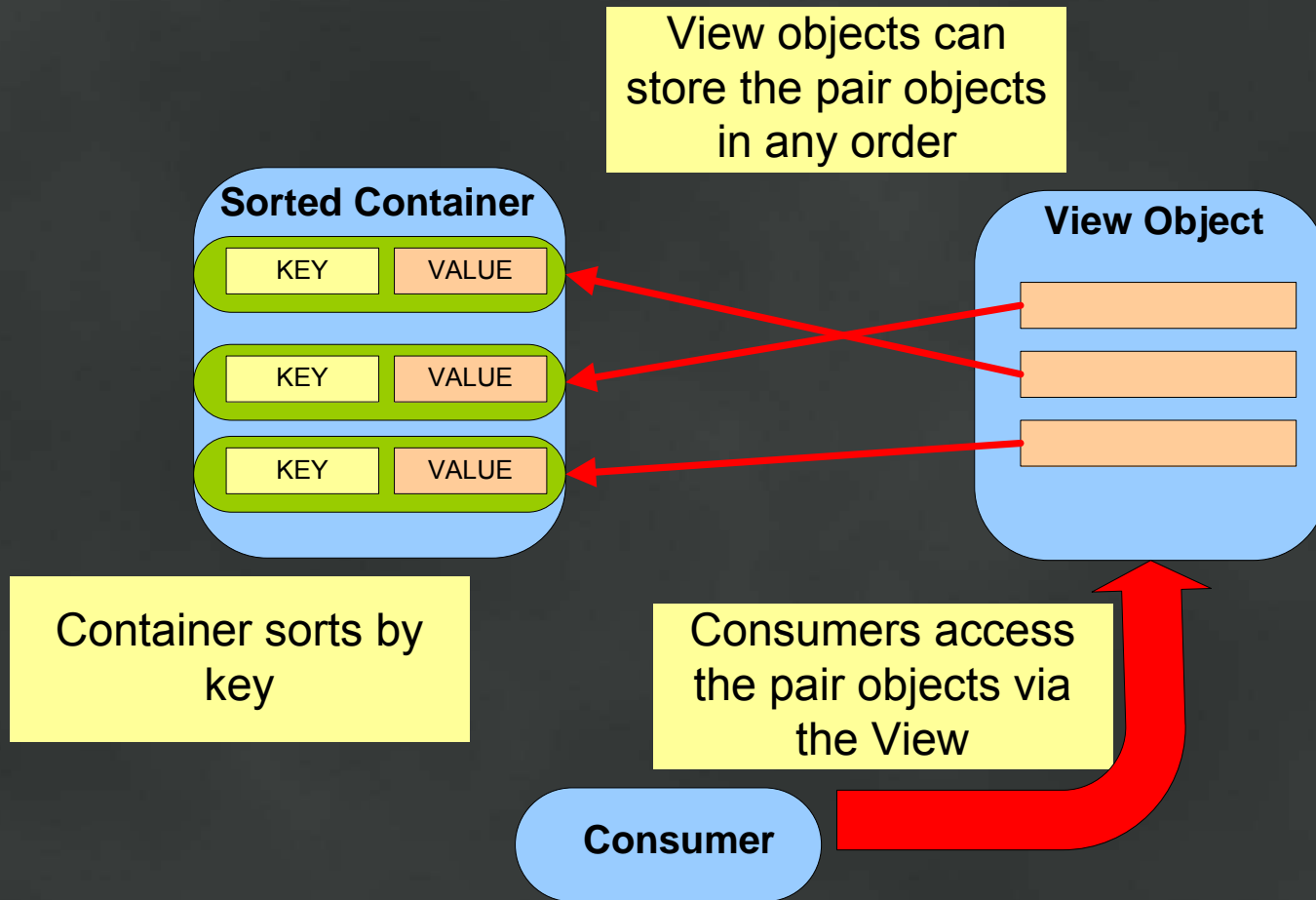
Views of Sorted Associative Containers

- Containers such as map and set associate a value with a key and store the values ordered by key
- Views provide a way to change the ordering of these containers without changing the container or copying the data

Example: Views of Sorted Associative Containers

- Re-sort a `map<size_t,record>` so that it is ordered by sum of `record::values` rather than `size_t`.
 - Use `build_view` with a custom functor
 - Functor compares the comparative sum of two `record::values` objects
 - Consumer sees a view of `std::pair` objects ordered by the functor, not a `std::map`.

Example: Views of Sorted Associative Containers



Example: Views of Sorted Associative Containers

```
// Build a simple view of the underlying map object
build_view( rmap.begin(), rmap.end(), rview );

// Sort the view using the cmp_record_values_size functor
sort(
rview.elements.begin(),
rview.elements.end(),
cmp_sum_values()
);
```

Example: Views of Sorted Associative Containers

```
struct cmp_sum_values{
    bool operator()( const record_view::element_type& lhs, const
        record_view::element_type& rhs ) const{
        size_t total1 = 0;
        size_t total2 = 0;

        total1 = accumulate(
            lhs.value().second.values.begin(),
            lhs.value().second.values.end(),
            0
        );

        total2 = accumulate(
            rhs.value().second.values.begin(),
            rhs.value().second.values.end(),
            0
        );

        return total1 < total2;
    }
};
```

Example: Views of Sorted Associative Containers

- Worked example `views_of_sorted_ctns.h` in session's source code.

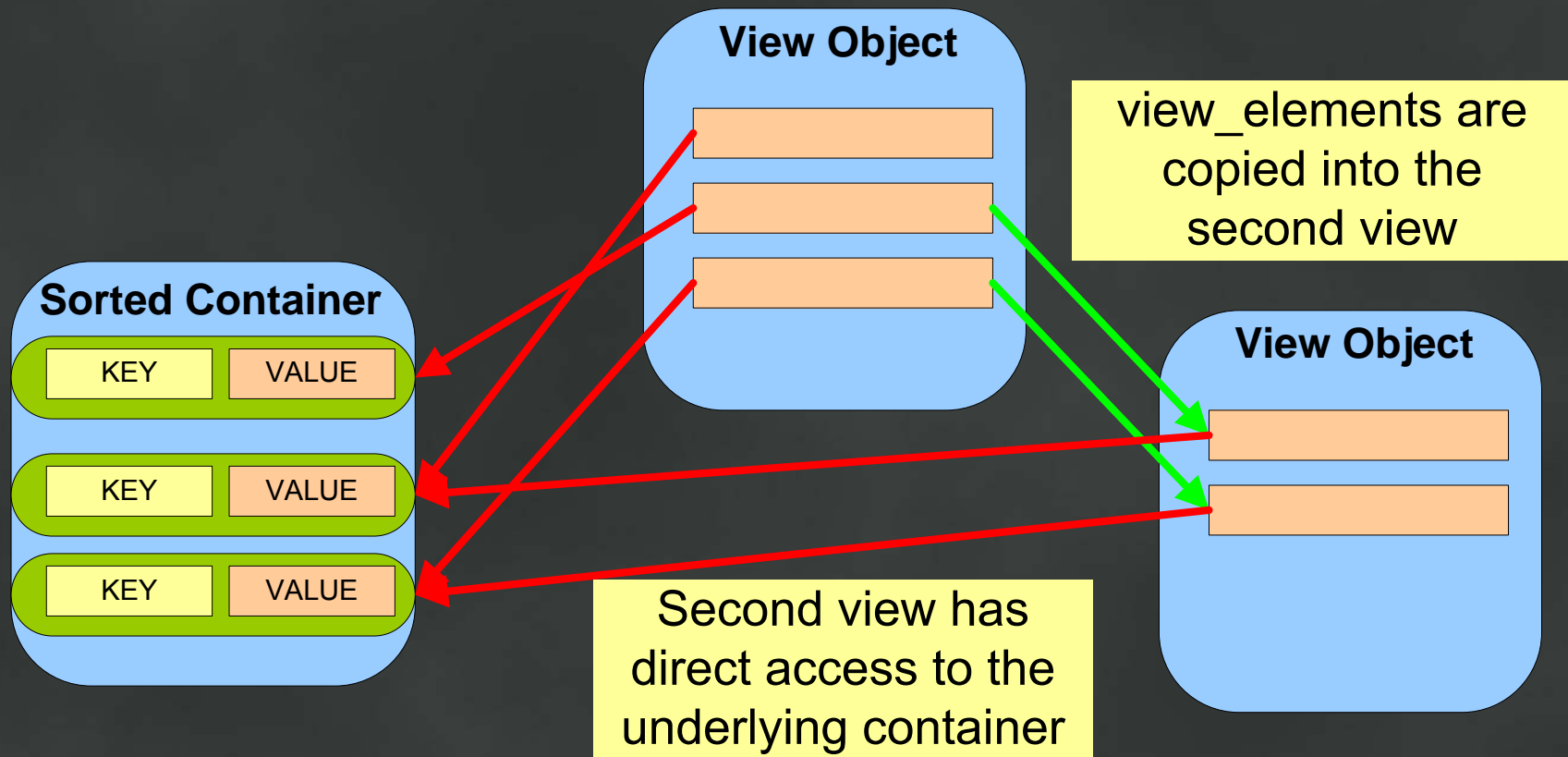
Cascading Views

- Views can be built on top of other views
- A useful technique for combining views
- One View provides the input for another
 - No limit on the number of levels of cascade

Example: Cascading Views

- Take the previous example of a view of a map.
- Build a second view containing the first two and last two elements from the sorted view
- The second view uses the first view as its input range but still references data in the same underlying container
- Result will be a second View object containing the top 2 and bottom 2 record vectors from the map based on `sum of record::values`

Example: Cascading Views



Example: Cascading Views

```
// Build view 1 (taken from the views of sorted containers example)

// Build a simple view of the underlying map object
build_view( rmap.begin(), rmap.end(), rview );

// Sort the view using the cmp_record_values_size functor
sort(
  rview.elements.begin(),
  rview.elements.end(),
  cmp_sum_values()
);

// rview contains a sorted view of the map; sorted by sum of record::values
```

Example: Cascading Views

```
// Build view2. Use the std::copy algorithm to copy the first two and  
// last two view_elements from rview into rview2.
```

```
    copy(  
        rview.elements.begin(),  
        rview.elements.begin() + 2,  
        inserter( rview2.elements, rview2.elements.begin() )  
    );  
    copy(  
        rview.elements.end() - 2,  
        rview.elements.end(),  
        inserter( rview2.elements, rview2.elements.begin() )  
    );
```

Example: Cascading Views

- Worked example `views_of_views.h` in session's source code.

Memory Efficiency of Views

- Copying data uses more memory than building a view (in most cases)
 - view_element objects are smaller than the data they represent
 - Especially textual data
 - Therefore, view objects are smaller than the underlying containers

Example: Memory Efficiency of Views

- Underlying container is a `vector<record>` with 5000 records
- Want to have two perspectives, one sorted and one not sorted.
- Using a view reduces memory usage by approx 40% compared to using a sorted copy

Example: Memory Efficiency of Views

- Worked example `views_memory_usage.h` in session's source code.

Session Summary

- **Views combine the consumer and operational centric models**
 - Gives the programmer flexibility
 - Maintains Data Integrity
 - Memory efficient
- **Designing a View class:**
 - A view contains `view_elements`
 - A `view_element` contains an iterator to the real data
 - Use STL algorithms or custom logic to build and use views
- **Where to use a View class:**
 - Consumers need a subset of the data (filtered, ordered, custom criteria etc)
 - Combine multiple containers into one container
 - Alternative orderings for associative containers

Resources and Links

- All source code and examples in session's source code
- Contact me at:
 - gabhan.berry@xlcubed.com
- Questions?

