The View Template Library

Martin Weiser / Gary Powell

Outline:

Introduction
Features
Usage example
Implementation techniques
Performance test

Views and VTL

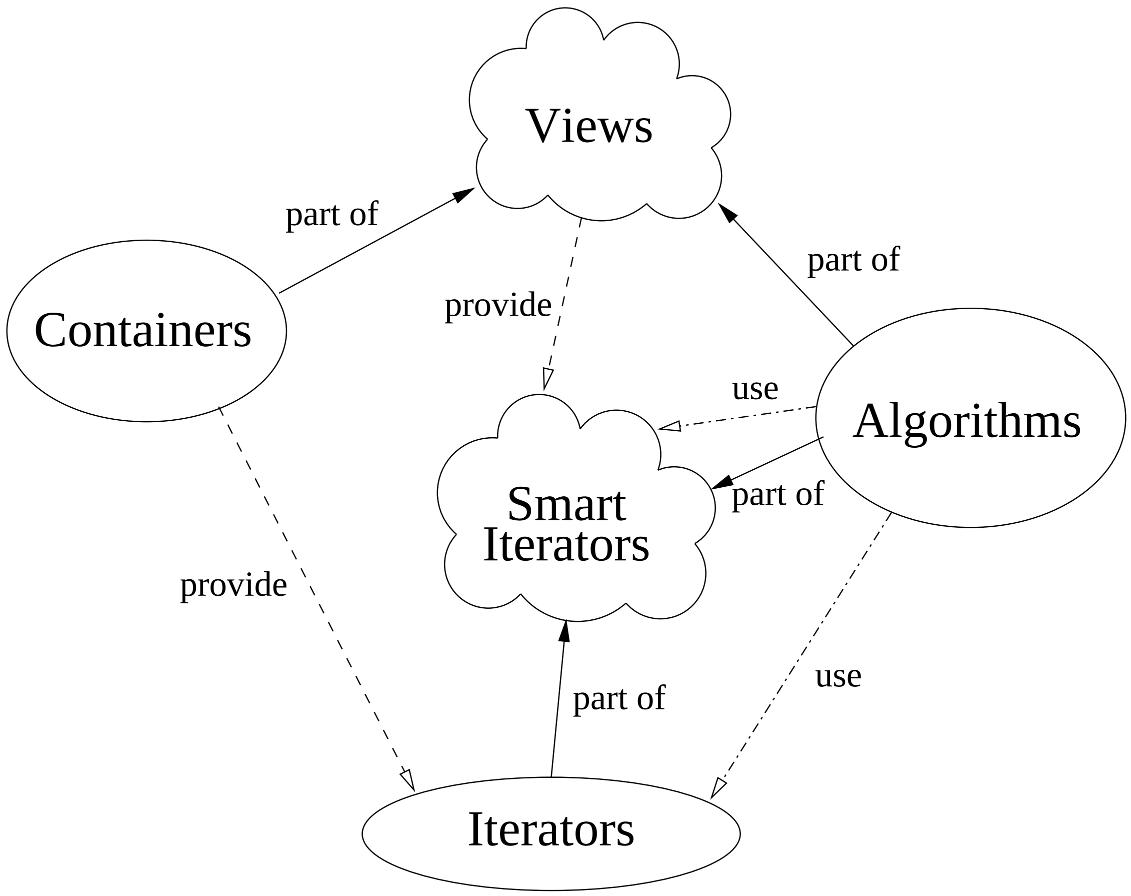
Views

- are *container adaptors* like `queue` and `stack`
- provide access to transformed data elements or a subset of data elements from an underlying container
- have the usual STL container interface
- are smart iterator factories

The View Template Library (VTL)

- implements a number of views with different functionality
- is highly flexible and adaptive
- uses traits and template metaprogramming to achieve this

Views Relatives



Containers: Views present transformed or filtered data elements from the underlying container.

Smart Iterators: Views create smart iterators, which do all the hard work.

Algorithms: Views (and their smart iterators) encapsulate some algorithmic functionality.

When to use Views

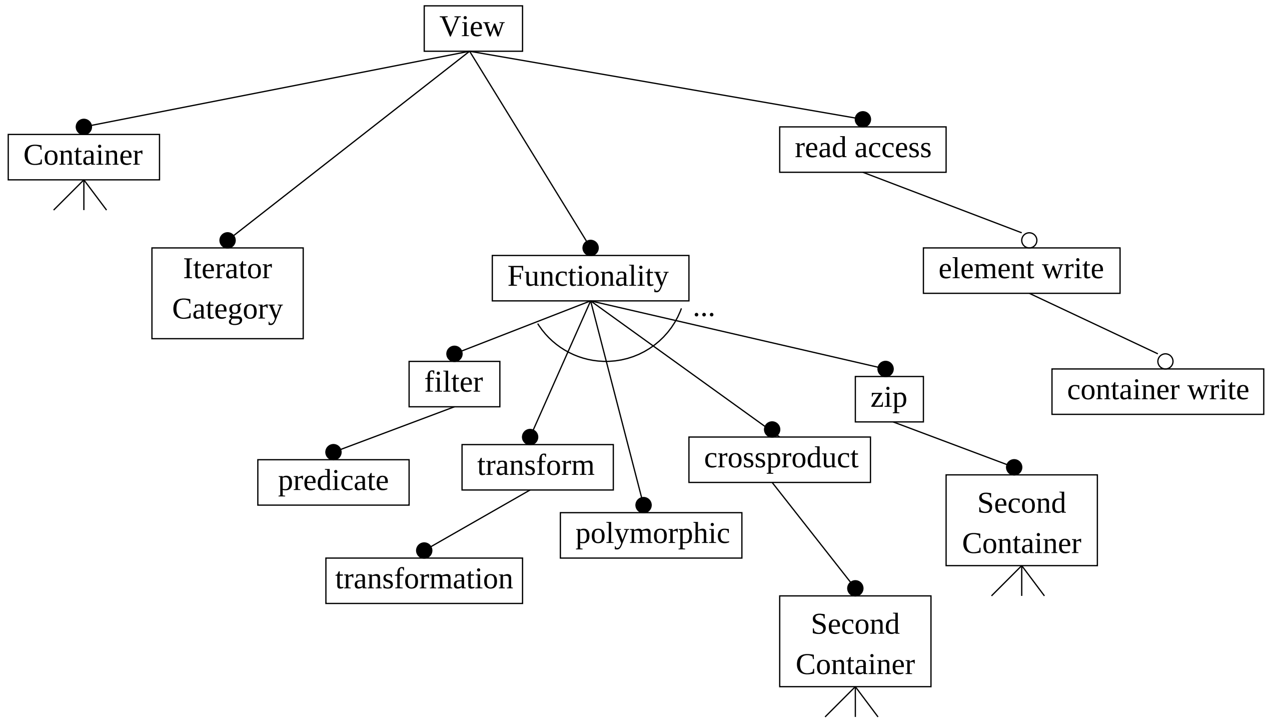
When you need to access on the fly generated/selected data trough a container interface.

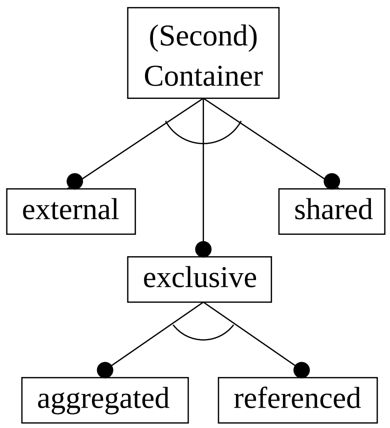
Items to consider choosing between copying, smart iterators, and views:

- time / space requirements
- data consistency
- partial / multiple data access

- iterator invalidation
- interface consistency
- access to metadata
- passing data by value

Views Features





thin abstraction layer → features bound statically for typesafety and performance reasons

VTL Functionality Overview

transform view presenting the results of a transformation applied to the underlying data elements: $[a, b, c] \mapsto [f(a), f(b), f(c)]$

filter view presenting only elements satisfying a predicate: $[a, b, c] \mapsto [a, c]$ assuming $p(a) = p(c) =$ true and $p(b) =$ false

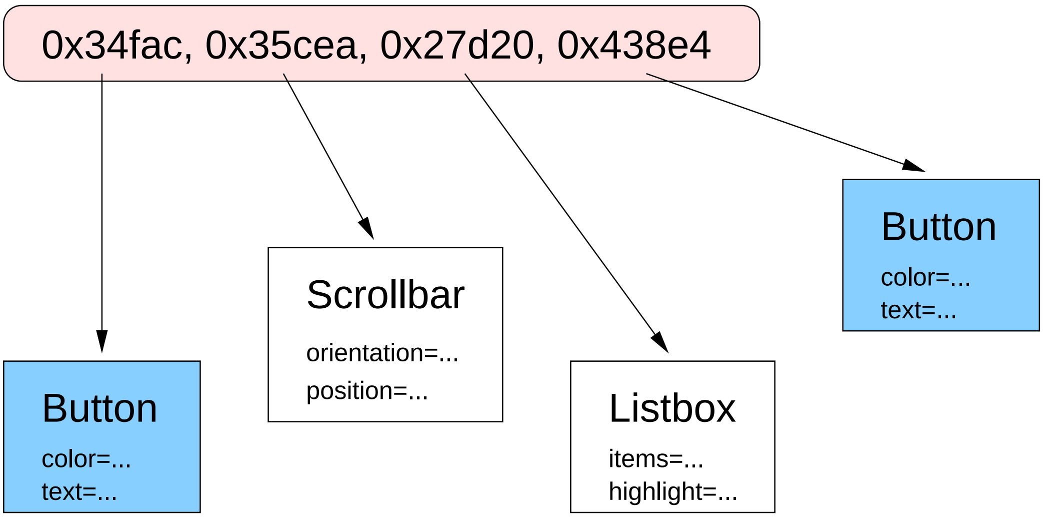
zip view presenting pairs of corresponding elements:
$([a, b, c], [\alpha, \beta, \gamma]) \mapsto [(a, \alpha), (b, \beta), (c, \gamma)]$

union view concatenating containers tail to head: $([a, b, c], [\alpha, \beta, \gamma]) \mapsto [a, b, c, \alpha, \beta, \gamma]$

... and a lot more

From Widgets to Buttons I

- polymorphic container of pointers to several widgets
- mimic a container of buttons

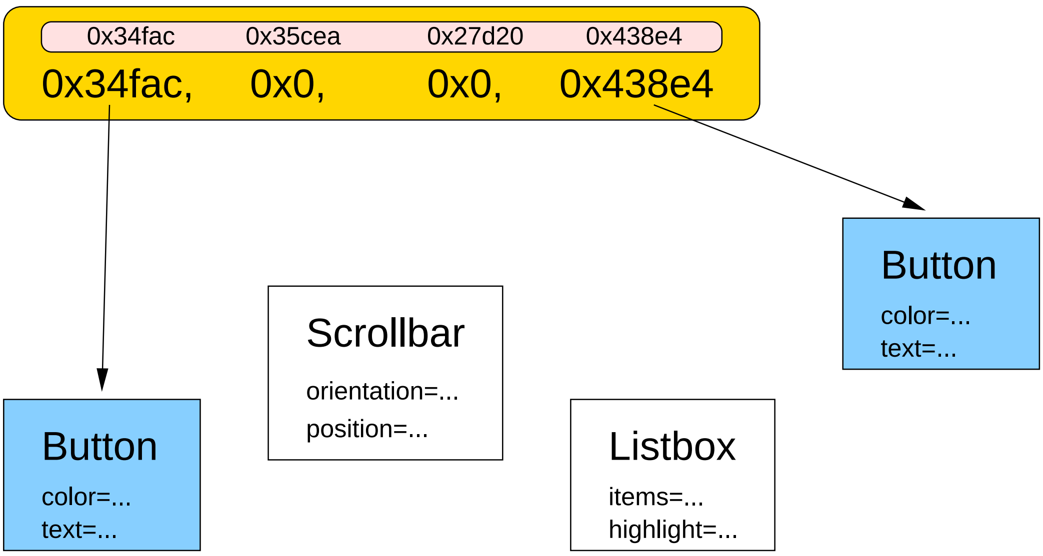


Three steps to solution:

1. `dynamic_cast<Button*>` to get pointers to buttons
2. take only non-null pointers
3. dereference pointers

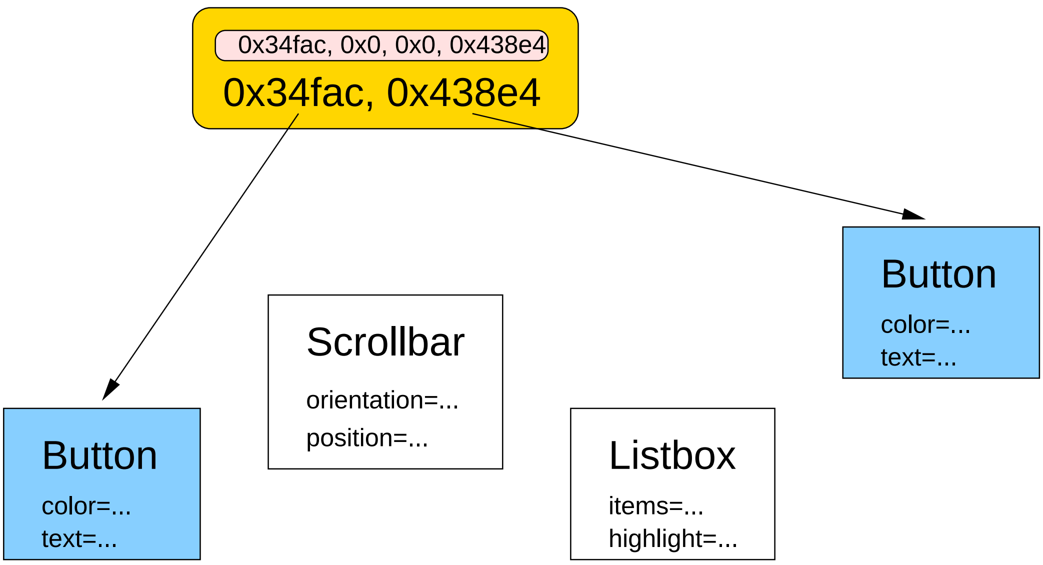
Step 1 Use a transform view with a `dynamic_cast` transformation to get pointers to buttons:

```
typedef transform_view<container_type,
                       downcast<Widget,Button> >
downcasted_type;

downcasted_type downcasted(container);
```



Step 2 Use a filter view with a non-null predicate to select the relevant pointers:

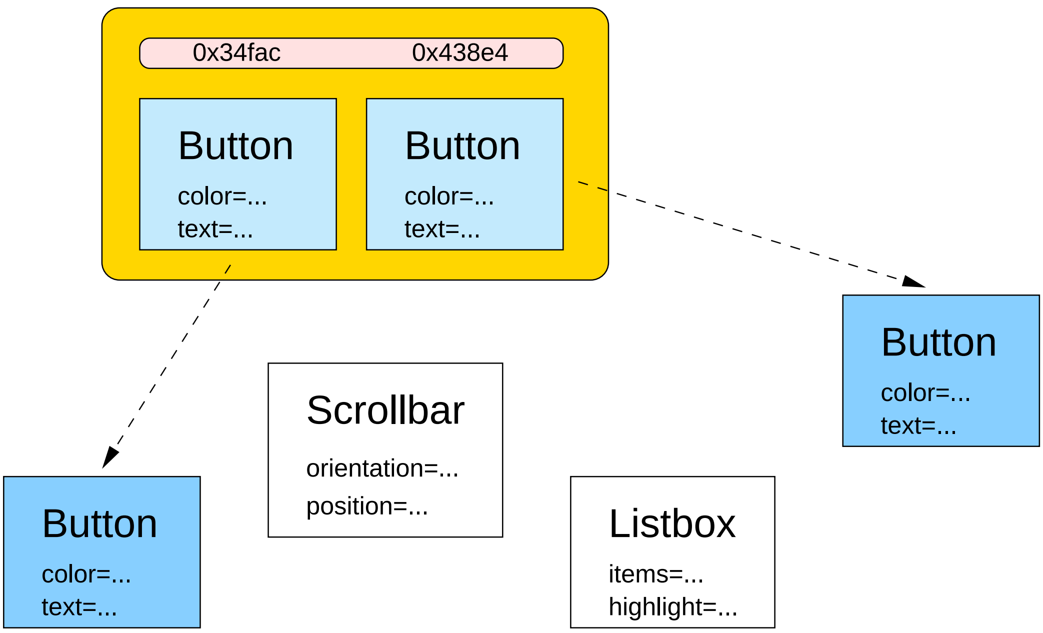
```
typedef filter_view<downcasted_type,
                    is_nonzero<Button*>,
                    ...,
                    aggregated_ownership>
filtered_type;

filtered_type filtered(downcasted);
```



From Widgets to Buttons IV

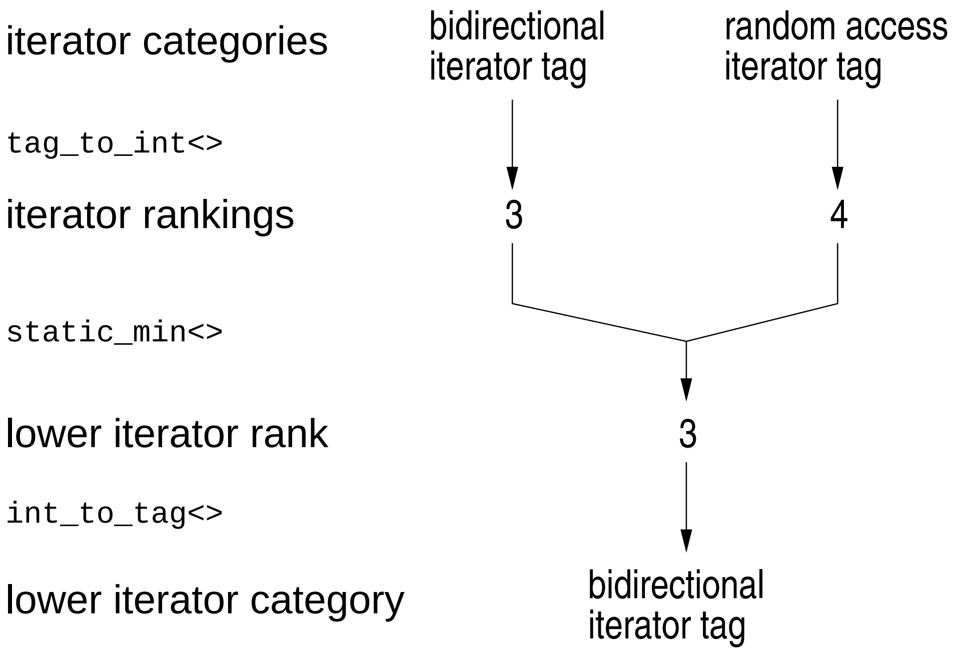
Step 3 Use a transform view with a dereferencing transformation to get references instead of pointers:

```
typedef transform_view<filtered_type,
                       dereference<Button*>,
                       ...,
                       aggregated_ownership>
dereferenced_type;

dereferenced_type button_view(filtered);

for_each(button_view.begin(),button_view.end(),
         do_something);
```



Type Deduction I

Need to select the lowest common denominator for iterator category (e.g. bidirectional from bidirectional and random access).




```
template <class T> struct tag_to_int;
template <>
struct tag_to_int<bidirectional_iterator_tag>
{ enum { rank = 3 }; };
template <>
struct tag_to_int<random_access_iterator_tag>
{ enum { rank = 4 }; };

template <int a, int b> struct static_min
{ enum { rank = a<b? a: b }; };
```

Type Deduction II


```
template <int rank> struct int_to_tag;
template <> stuct int_to_tag<3>
{ typedef bidirectional_iterator_tag type; }
template <> stuct int_to_tag<4>
{ typedef random_access_iterator_tag type; }
```

Putting it together:

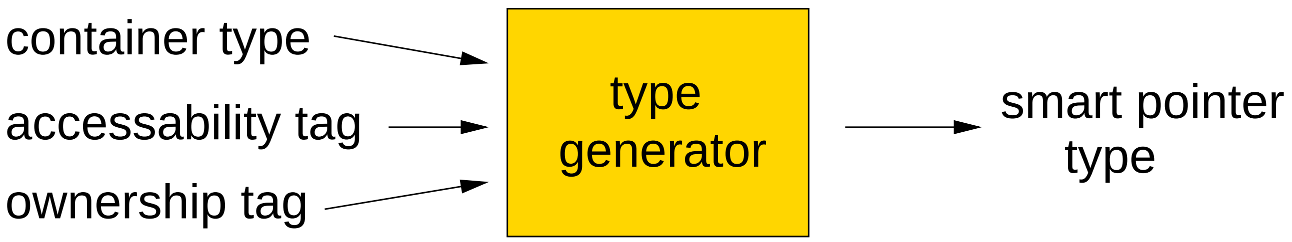
```
typedef typename
 int_to_tag<
  static_min<
   tag_to_int<bidirectional_iterator_tag>::rank,
   tag_to_int<random_access_iterator_tag>::rank
   >::rank
  >::type common_denominator;
```

results in `bidirectional_iterator_tag`.

Type Generation

- underlying container referenced by *smart pointer*
- smart pointer type depends nontrivially on container type, accessability and ownership features

→ don't specify smart pointer type as template parameter
→ compute smart pointer type internally with type generator



Advantages:

- clean, orthogonal, non-redundant interface
- implementation details hidden

Performance Comparison I

VTL

Simplistic performance test with model problem:

```
typedef pair<int,int> value;
vector<value> container;
// fill container ...
```

Compute minimal second element *using standard container interface*.

Compare solution *std*

```
vector<int> tmp(container.size());
transform(container.begin(),container.end(),
          select2nd<value>(),tmp.begin());
return *min_element(tmp.begin(),tmp.end());
```

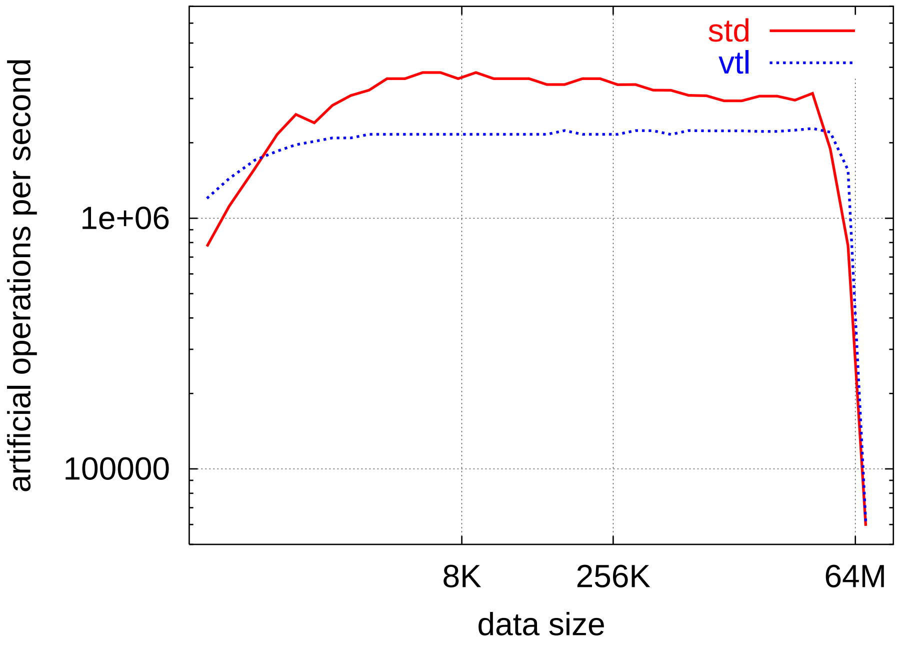
with solution *vtl*

```
transform_view<vector<value>,select2nd<value> >
  tmp(container,select2nd<value>());
return *min_element(tmp.begin(),tmp.end());
```

Performance Comparison II

Results on GCC 2.95.2, Sun Ultra 5/10.

Without optimization:



With optimization (-O3):