Adaptable component frameworks: Using `vector` from the C++ standard library as an example

Bo Simonsen

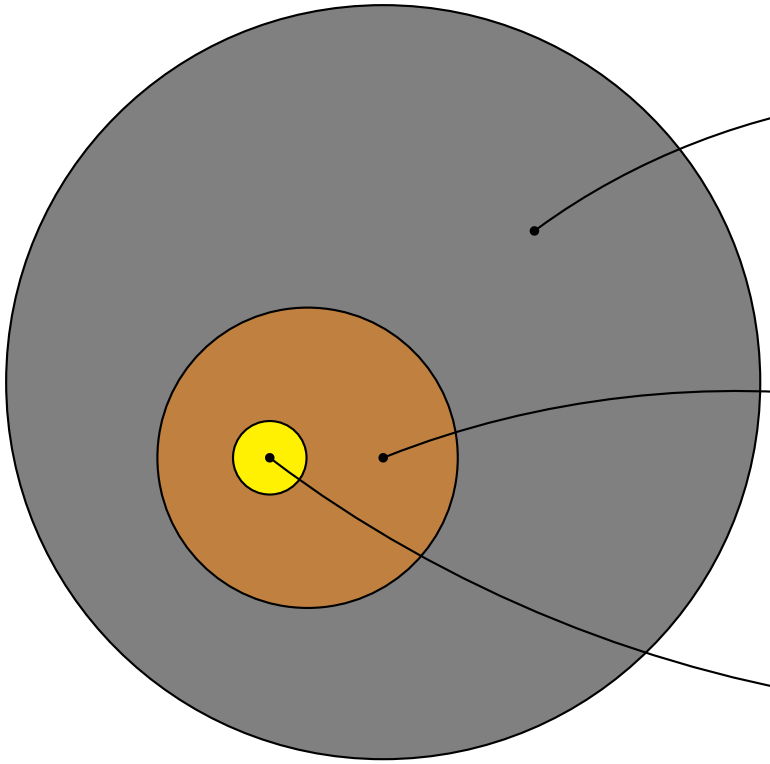
University of Copenhagen

Joint work with Jyrki Katajainen

These slides are available at

`http://cphstl.dk`

Component-based programming

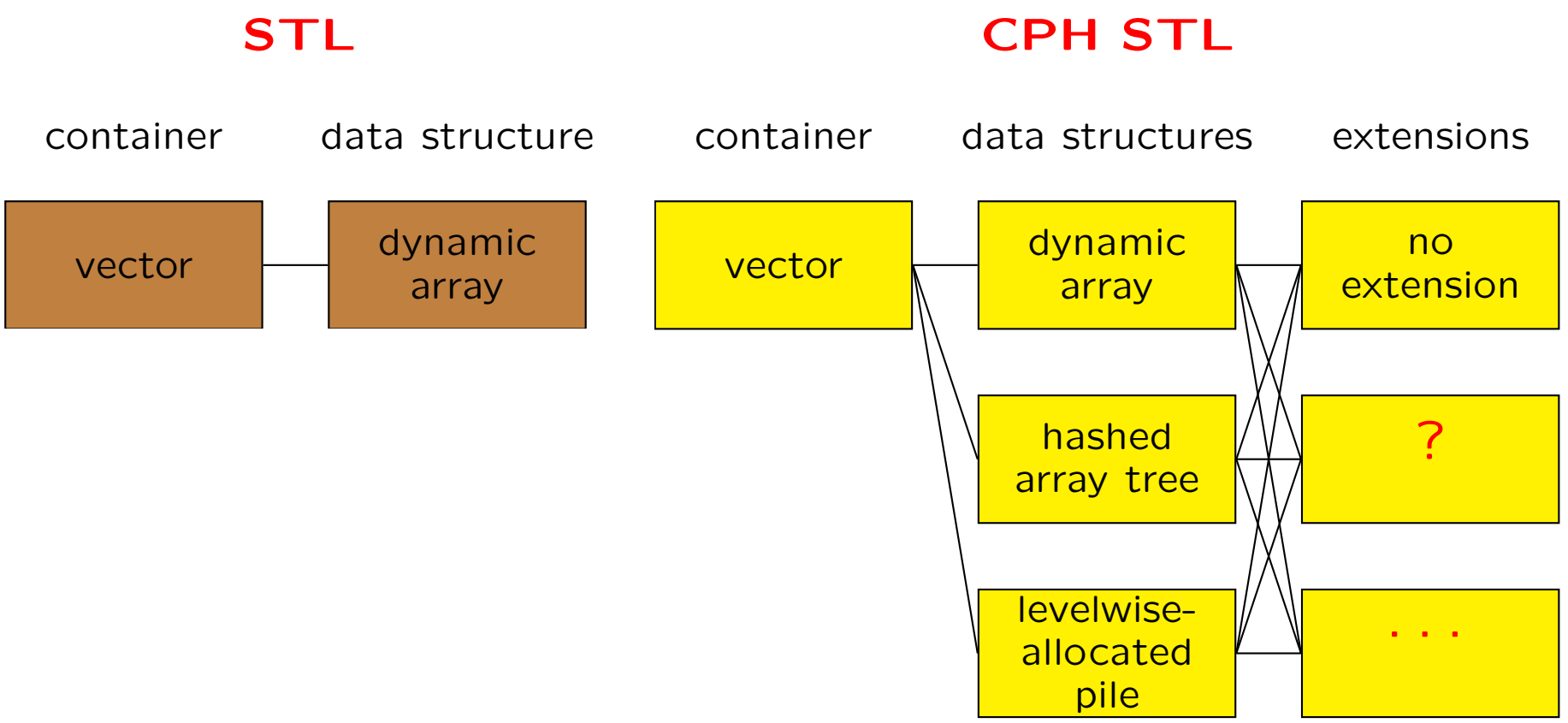


Component frameworks
A skeleton of a software component which is to be filled with implementation specific details.

Generic component frameworks
A component framework where the user provides the implementation specific details in form of policies.

Vector framework
A `vector` container stores a sequence of elements which can be accessed by indices or iterators at constant cost.

Our world view



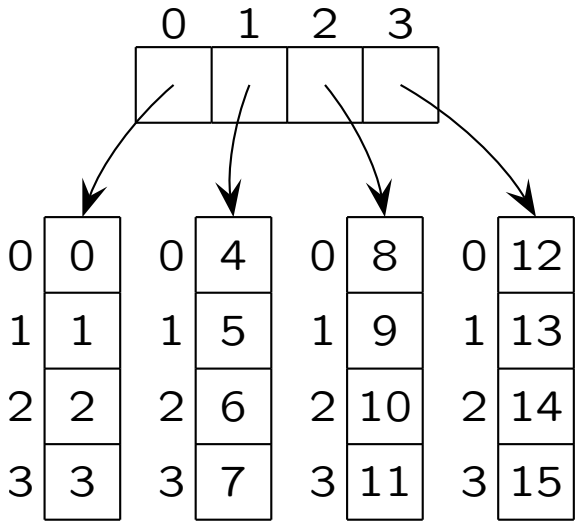
We provide several data structures because of **space efficiency** and **worst-case time complexity**.

Why alternative data structures?

Observation: For `libstdc++ vector` the worst-case space consumption is unbounded because the array is never contracted.

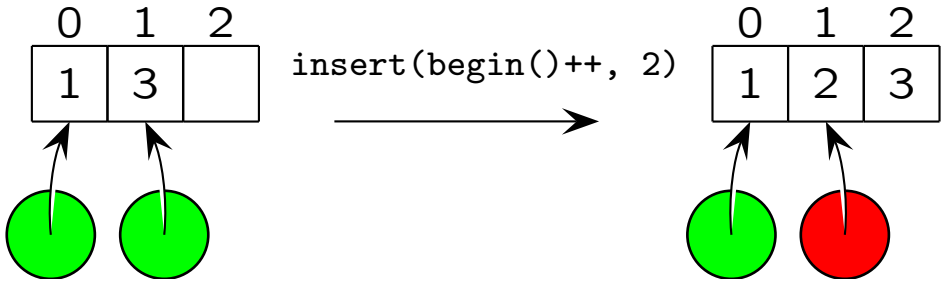
Problem: Consider a `vector` consisting of large objects used in a long-running application. This behaviour is unacceptable.

Solution: We provide an implementation (hashed array tree) requiring $n+O(\sqrt{n})$ worst-case space (n denoting the # of elements stored).

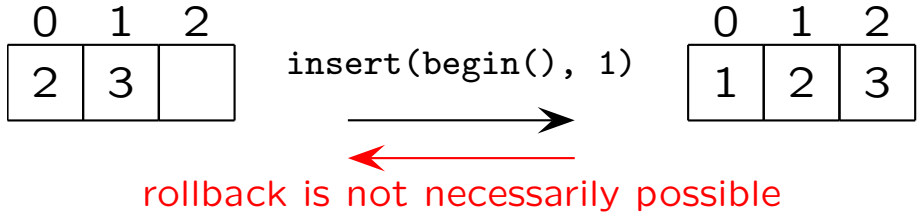


Why extensions?

Referential Integrity: Undesirable behaviour in some cases:



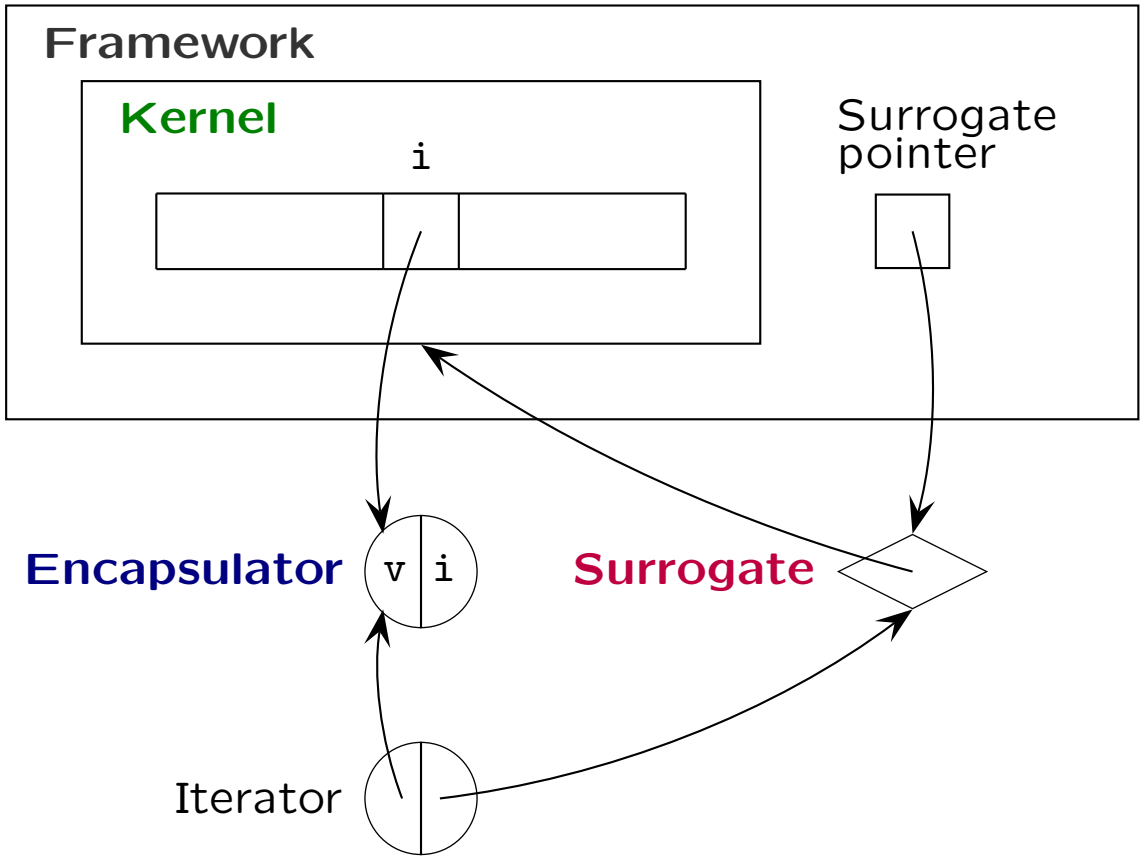
Strong Exception Safety: Undesirable behaviour in some cases:



Contributions

- We show how to build a component framework for STL containers while maintaining standard compliance.
- We show that component frameworks have an acceptable performance overhead.
- We show how to provide strong exception safety and referential integrity for `vector`, and analyse the cost of safety in terms of running time.

Vector framework concepts



Factorizes container operations.

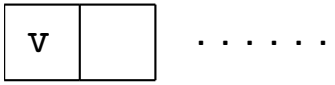
Realizes a minimal implementation of a data structure: `grow`, `shrink`, `access`.

Encapsulates a value in a small object. This object can either be allocated or stored directly in the array.

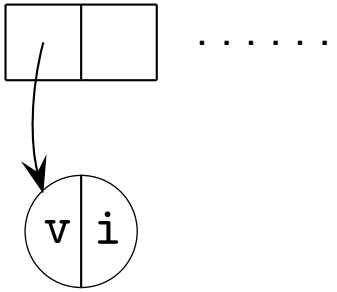
Provides a proxy for the kernel to ensure `swap` does not invalidate iterators.

Encapsulators

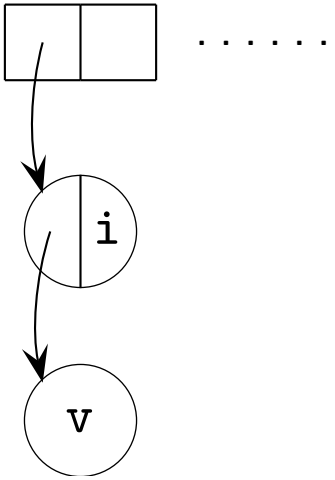
Elements stored directly



Elements stored indirectly

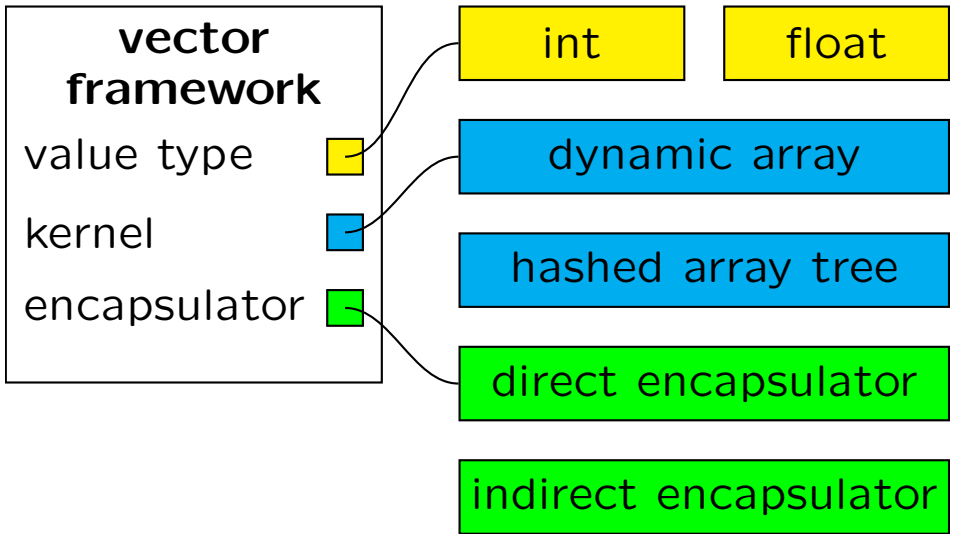


Elements stored doubly indirectly



Why component frameworks?

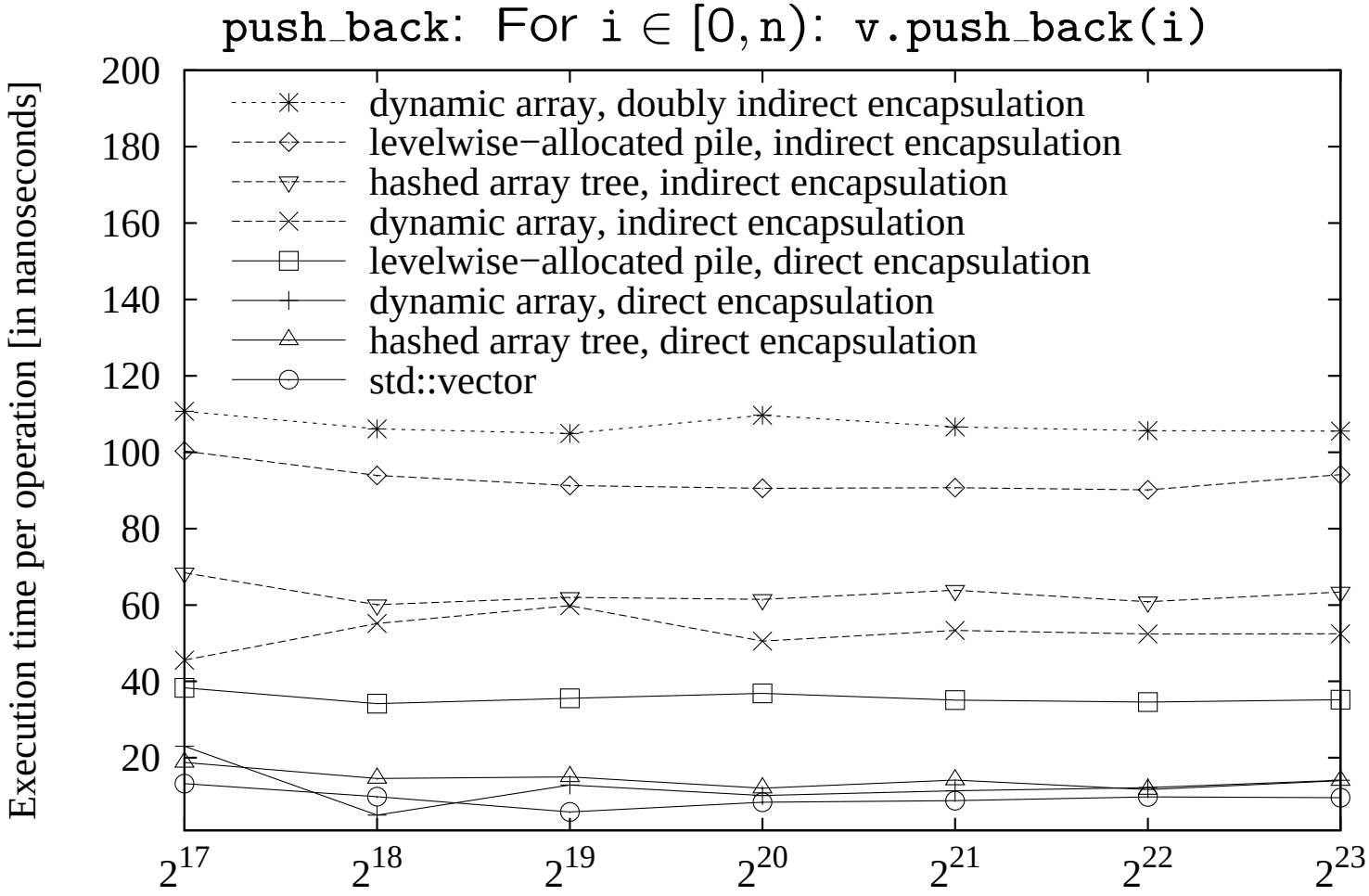
Adaptability:



Maintainability: The container is composed by reusable components; this allows us to obtain a high degree of code reuse.

Fair benchmarking: Ideally, the benchmark results reflect the efficiency of the data structures, not the skills of the programmers.

Benchmarks


push_back: For i ∈ [0, n): v.push_back(i)
Execution time per operation [in nanoseconds]
200
180
160
140
120
100
80
60
40
20
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
2^{22}
2^{23}
dynamic array, doubly indirect encapsulation
levelwise−allocated pile, indirect encapsulation
hashed array tree, indirect encapsulation
dynamic array, indirect encapsulation
levelwise−allocated pile, direct encapsulation
dynamic array, direct encapsulation
hashed array tree, direct encapsulation
std::vector

Paper outline

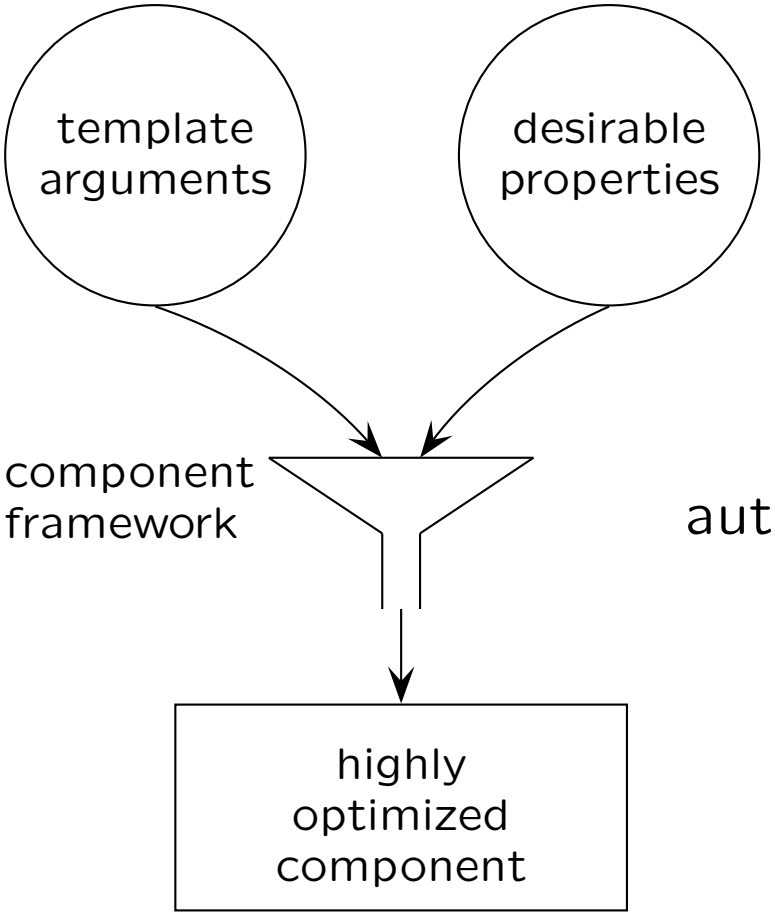
Our paper can serve as a starting point for building new versions of the STL, or in general, generic algorithmic libraries.

We provide

- discussion of possible data structures for realizing `vector`,
- discussion of possible optimizations,
- details regarding the design,
- more benchmarks,
- experiences and lessons learned.

All code for the framework is available in an electronic appendix (http://cphstl.dk).

Adaptivity



automatically at compile time ?

Wanted: Compile-time reflection

Element copying is a recurring operation in the vector framework. A wide-known optimization technique can be used to speed up this operation.

Optimization: For plain-old-data (POD) types, use `memcpy()` for copying.

Solution: Use `std::tr1::is_pod<V>::value` to perform the check whether the data type stored in the `vector` is POD.

Problem: According to TR1 it is unspecified when this expression evaluate to true.

Object copying can be expensive because copying is done object destruction and construction. Can we do any optimization to speed up object copying?

Wanted: Compile-time profiling

Optimization: Use indirect encapsulation if it is more profitable.

Approximation: Use indirect encapsulation for class types (only pointers are copied).

```
template <typename V, typename A>
class encapsulator_selector {
public:
  typedef cphstl::direct_encapsulator<V, A> E;
  typedef cphstl::indirect_encapsulator<V, A> F;
  typedef typename cphstl::if_then_else<std::tr1::is_class<V>::value,
    F, E>::type type;
};
```

Problem: Consider a class containing a built-in type.

Ideal solution: A compiler with profiling support.

Wanted: Support for generic overriding

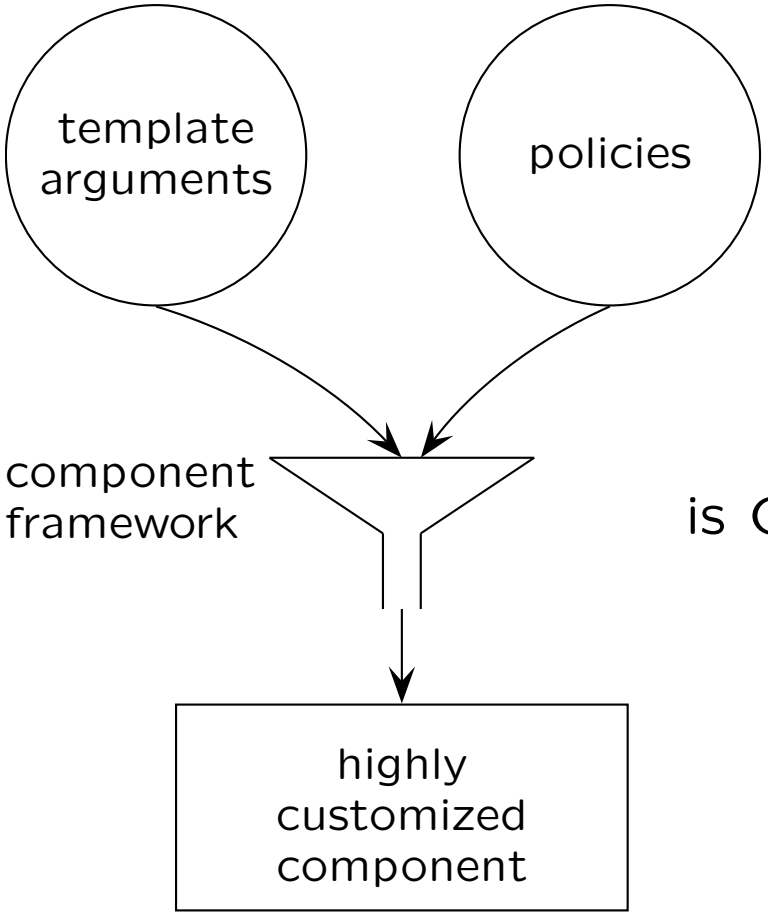
A kernel is defined by `grow(`δ`)`, `shrink()`, and `access(i)`; copying of elements is performed by

```
template <typename V, typename A, typename K>
void vector_framework<V, A, K>::block_copy_backward(size_type start,
size_type end, size_type s) {
  for (size_type i = end + s - 1; i ≥ start + s; --i) {
    slot_swap((*this).k.access(i), (*this).k.access(i - s));
  }
}
```

Optimization: If a kernel provides a `block_copy_backward` member function, this will be used.

Solution: This mechanism is implemented by SFINAE. Concept-based overloading would make the implementation more cleaner.

Adaptability



is C++ powerful enough ?

Wanted: Better encapsulators

```
#include <stdexcept> // defines std::domain_error
#include <stl-vector.h++> // defines cphstl::vector

class my_class {
public:
  my_class(int const& a) {
  }
  my_class const& operator=(my_class const&) {
    throw std::domain_error("...");
  }
};

int main() {
  cphstl::vector<my_class> v;
  v.insert(v.begin(), my_class(5));
  v[0] = my_class(6);
}
```

Wanted: Better usability

```
typedef int V;
typedef std::allocator<V> A;
typedef cphstl::direct_encapsulator<V, A> E;
typedef cphstl::dynamic_array<V, A, E> K;
typedef cphstl::vector_framework<V, A, K> R;
typedef cphstl::rank_iterator<R, false> I;
typedef cphstl::rank_iterator<R, true> J;
typedef cphstl::vector<V, A, R, I, J> C;
```

Observations:

- Template arguments are given several times.
- The meaning of each template parameter is not clear.
- Default values are not sufficient.

Our solution: Named template arguments

```
typedef cphstl::vector!<V=int,
                        R=cphstl::vector_framework!<
                          K=cphstl::dynamic_array!<
                            E=cphstl::direct_encapsulator
                          !>
                        !>,
                        I=cphstl::rank_iterator!<is_const=false!>,
                        J=cphstl::rank_iterator!<is_const=true!>
                      !> C;
```

- Template arguments are global.
- Missing template arguments are substituted by default values.

Currently implemented using a preprocessor. More details can be found in CPH STL report 2009-6.

Invitation for collaboration

CPH STL has been a great teaching tool for many years at our department. We have used in courses on

- generic programming and
- software construction.

We encourage others to use the CPH STL in their teaching.

An exercise

$create(K)$. Create an empty K.

$destroy(K)$. Destroy an empty K.

$size(K)$ **const**. Return the number of elements stored in K.

$size(K,\ s)$. Set the number of elements stored to s.

$max\_size(K)$ **const**. Return the maximum number elements that can be stored.

$capacity(K)$ **const**. Return the current capacity of K.

$access(K,\ i)$. Return a reference to the ith element of K.

$grow(K,\ \delta)$. Increase the size of K by $\delta \in \mathbb{N}$.

$shrink(K)$. Fit the capacity to the number of elements stored.

Assignment: Implement a new `vector` kernel for the CPH STL.

Proposal: A `vector` using $O(1)$ worst-case time for `push_back` and `pop_back` operations.