# Is the CPH STL an active library?

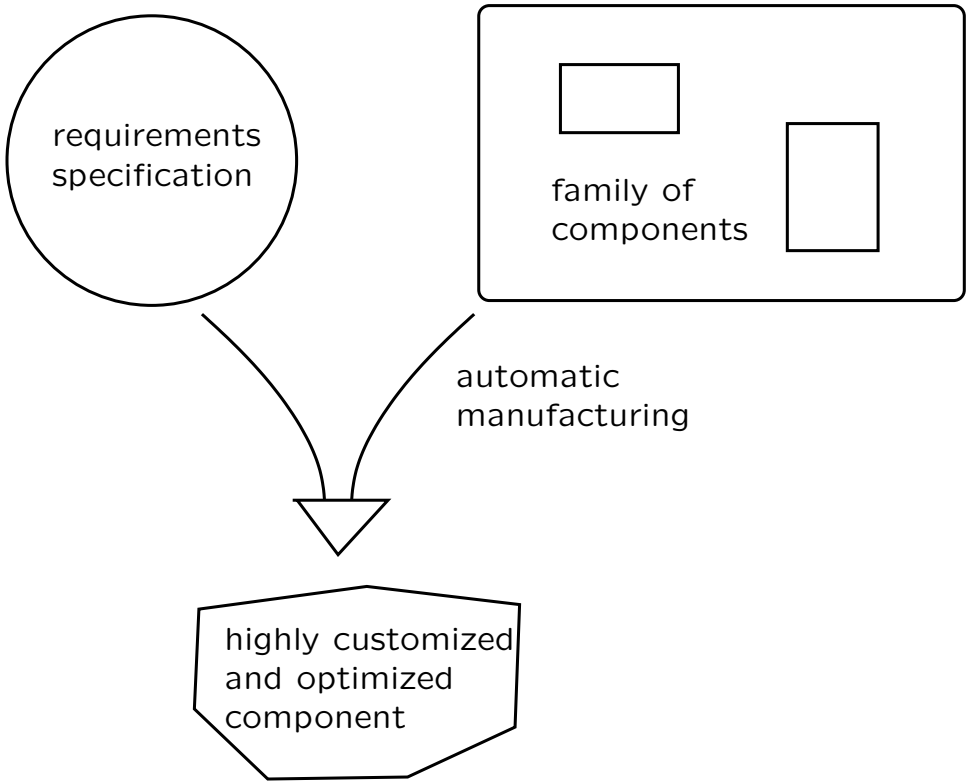
Jyrki Katajainen (University of Copenhagen)

Joint work with Bo Simonsen (University of Copenhagen)


11 May 2009

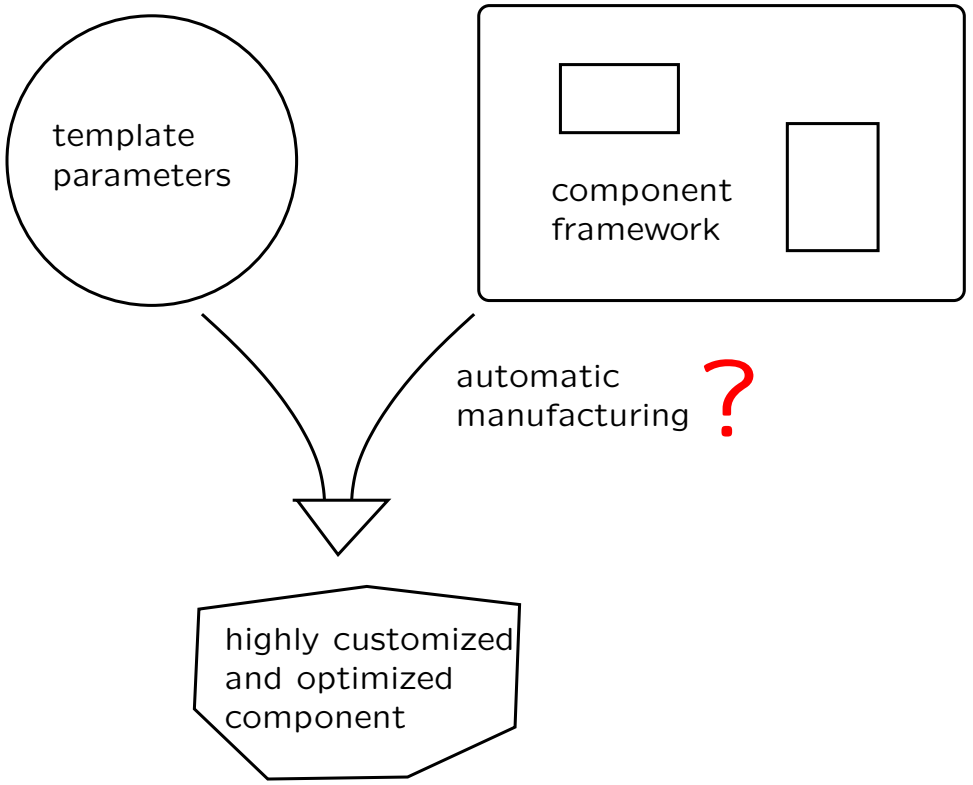
These slides are available at
http://cphstl.dk

# Active library




Read [Czarnecki et al. 2000]

# Generic library like the CPH STL




Visit http://cphstl.dk

# Selection of the copying method

**Potential optimization:** If both in the source and the target the elements are stored in a contiguous memory segment, if the elements are POD types, and if the sizes of the elements in both arrays are the same, copy the elements using the fast `memcpy` function, which is available at the standard C library.

**Problem:** The technical report on C++ library extensions does not specify under what circumstances, if any, `std::tr1::is_pod<V>::value` is true.

```
1#include <cstdlib> // defines std::size_t
2#include <cstring> // defines std::memcpy
3#include <iterator> // defines standard iterator traits
4#include <tr1/type_traits> // defines standard type traits
5#include "type.h++" // defines cphstl::int2type
6
7namespace cphstl {
8  namespace {
9
10    enum copy_algorithms {fast, conservative};
11
12    template <typename I, typename O>
13    O copy(I p, I q, O r, cphstl::int2type<fast>) {
14      std::size_t n = q - p;
15      std::memcpy(r, p, n * sizeof(*p));
16      return r + n;
17    }
18
19    template <typename I, typename O>
20    O copy (I p, I q, O r, cphstl::int2type<conservative>) {
21      for (; p != q; ++p, ++r) {
```

```
22          *r = *p;
23        }
24        return r;
25      }
26    }
27
28    template <typename I, typename O>
29    O copy(I p, I q, O r) {
30      typedef typename std::iterator_traits<I>::value_type V;
31      typedef typename std::iterator_traits<O>::value_type U;
32      enum { algorithm =
33             std::tr1::is_pointer<I>::value &&
34             std::tr1::is_pointer<O>::value &&
35             std::tr1::is_pod<V>::value &&
36             std::tr1::is_pod<U>::value &&
37             sizeof(V) == sizeof(U) ? fast : conservative
38      };
39      return copy(p, q, r, cphstl::int2type<algorithm>());
40    }
41 }
```

# Selection of the storage policy

**Potential optimization:** If it is more expensive to copy an element than a pointer, store elements indirectly; otherwise store them directly.

**Problem:** No compile-time `costof` operator provided! We can only approximate this optimization.

```
 1 #include <memory> // defines std::allocator
 2 #include "vector-framework.h++" // defines cphstl:: ↩
     vector_framework
 3 #include "dynamic-array.h++" // defines cphstl::dynamic_array
 4 #include "plain-entry.h++" // defines cphstl::plain_entry
 5 #include "safe-entry.h++" // defines cphstl::safe_entry
 6 #include <tr1/type_traits> // defines standard type traits
 7 #include "type.h++" // defines cphstl::if_then_else
 8
 9 namespace cphstl {
10   namespace {
11
12     template <typename V, typename A>
13     class kernel_selector {
14     public:
15       typedef cphstl::plain_entry<V, A> E;
16       typedef cphstl::dynamic_array<V, A, E> K;
17       typedef cphstl::safe_entry<V, A> F;
18       typedef cphstl::dynamic_array<V, A, F> L;
19       typedef typename cphstl::if_then_else<std::tr1::is_class<V ↩
           >::value, L, K>::type type;
```

```
20      };
21  }
22
23    template <
24      typename V,
25      typename A = std::allocator<V>,
26      typename R = cphstl::vector_framework<V, A, typename
          kernel_selector<V, A>::type>,
27      ...
28    >
29    class vector {
30      ...
31    };
32  }
```

# Conclusions

- In C++ the facilities provided for compile-time reflection and template metaprogramming are still primitive.
- Yes, we have to take the step from generic programming to generative programming.

# Mini-project

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

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

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

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

$access(K, i)$ **const**. Return a reference to the $i$th element of $K$.

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

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

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

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