Specialeforsvar: Fundamentet for et fleksibelt container bibliotek

Foundations of an adaptable container library

Bo Simonsen

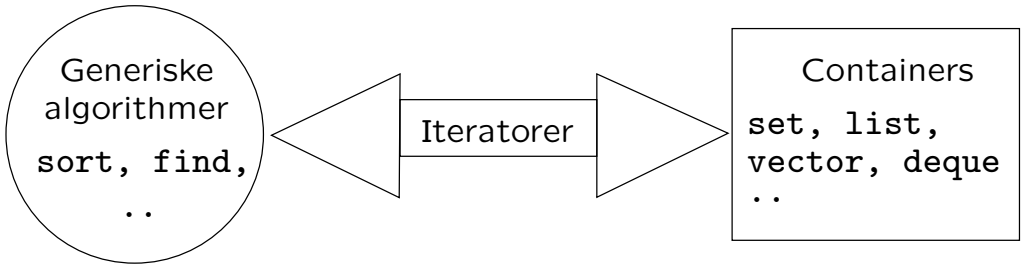
Datalogisk Institut, Københavns Universitet

Denne præsentation, afhandlingen, samt errata kan hentes på `http://bo.geekworld.dk/master.html`.

Overblik

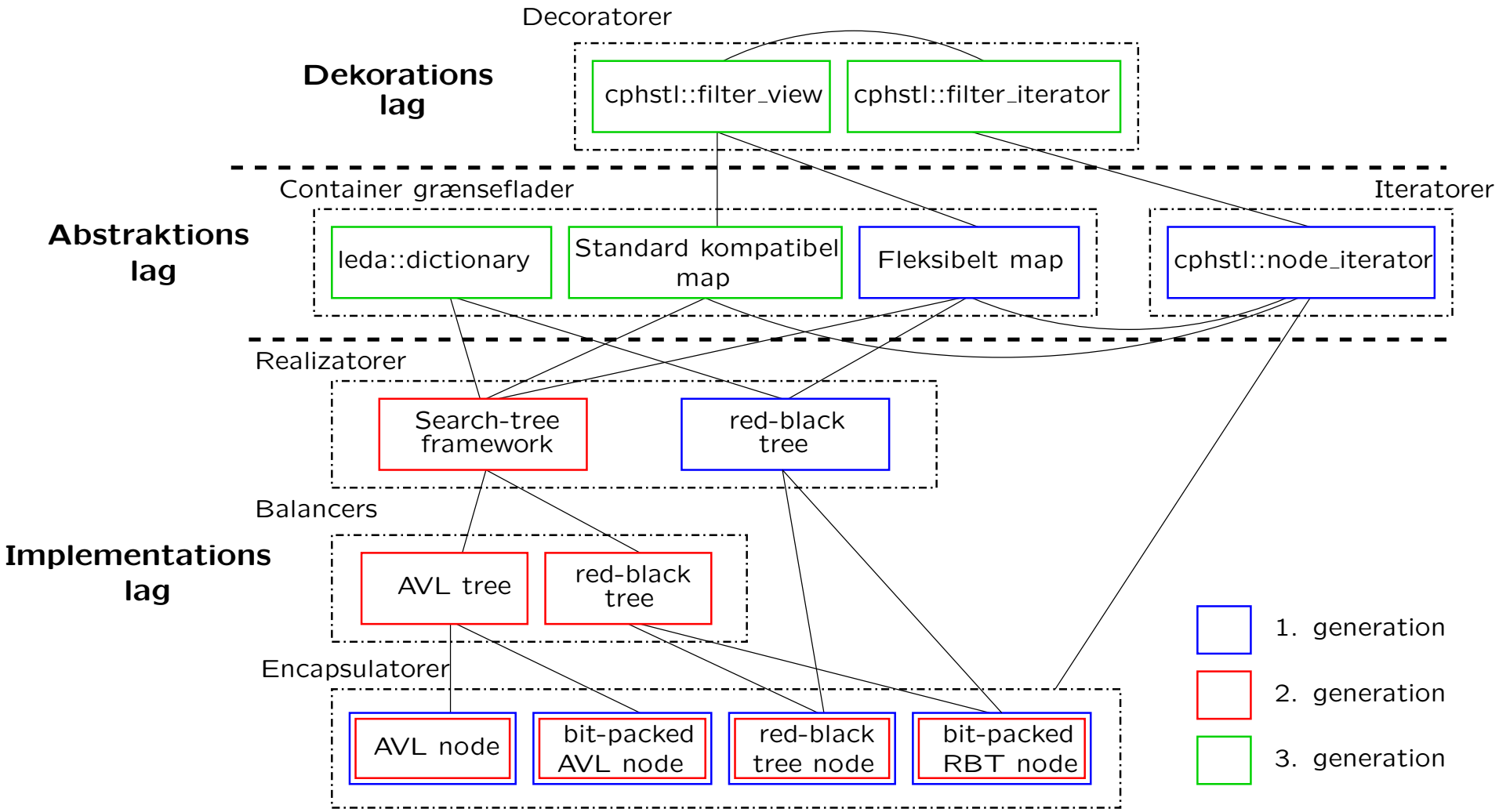
- Specialet omhandler arkitekturen og designet af CPH STL, samt løsninger af problemer i forbindelse med brug af CPH STL.
- STL er et generisk program bibliotek for C++ der tilbyder algoritmiske byggeklodser. STL tilbyder en implementation af en data struktur for hver container.
- CPH STL er en udvidet udgave af STL der tilbyder forskellige implementationer af data strukturer med forskellige karakteriska blandt andet: ydelse, plads effektivitet og sikkerhed.
- Specialet består af tre artikler, hvoraf to af dem er skrevet sammen med Jyrki Katajainen.

STL



- En **container** er en class template der kan opdateres dynamisk ved hjælp af de funktioner der stilles tilrådighed af containeren.
- En **generisk algoritme** er en function template der udfører operationer på en container eller en sekvens.
- En **iterator** er en type der gør det muligt for de generiske algoritmer at tilgå de data der er gemt i en container.
- Ordning og hukommelses allokering er afkoblet og placeret i henholdsvis en **comparator** og **allocator**.

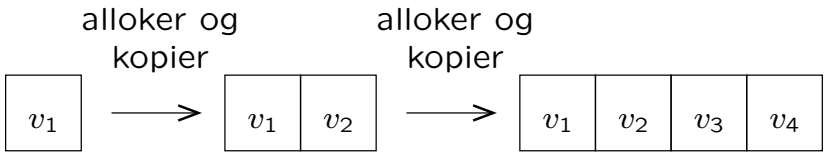
Arkitekturen i CPH STL



Case study: vector

En **vector** kan indeholde en sekvens af elementer der kan tilgås via iteratorer eller indices i konstant tid.

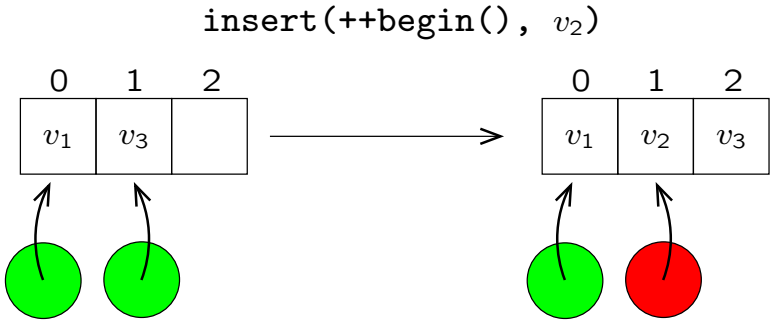
Jvf. C++ standarden er den eneste mulige implementation et dynamisk array da elemeterne skal opbevares i et sammenhængende hukommelses segment.



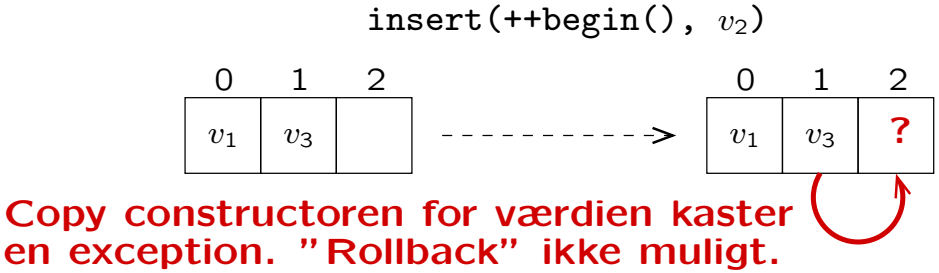
Vores implementationer er ikke altid standard kompatible.

Problemer og mulige udvidelser

Referentiel Integritet: Uønsket adfærd i nogle tilfælde:

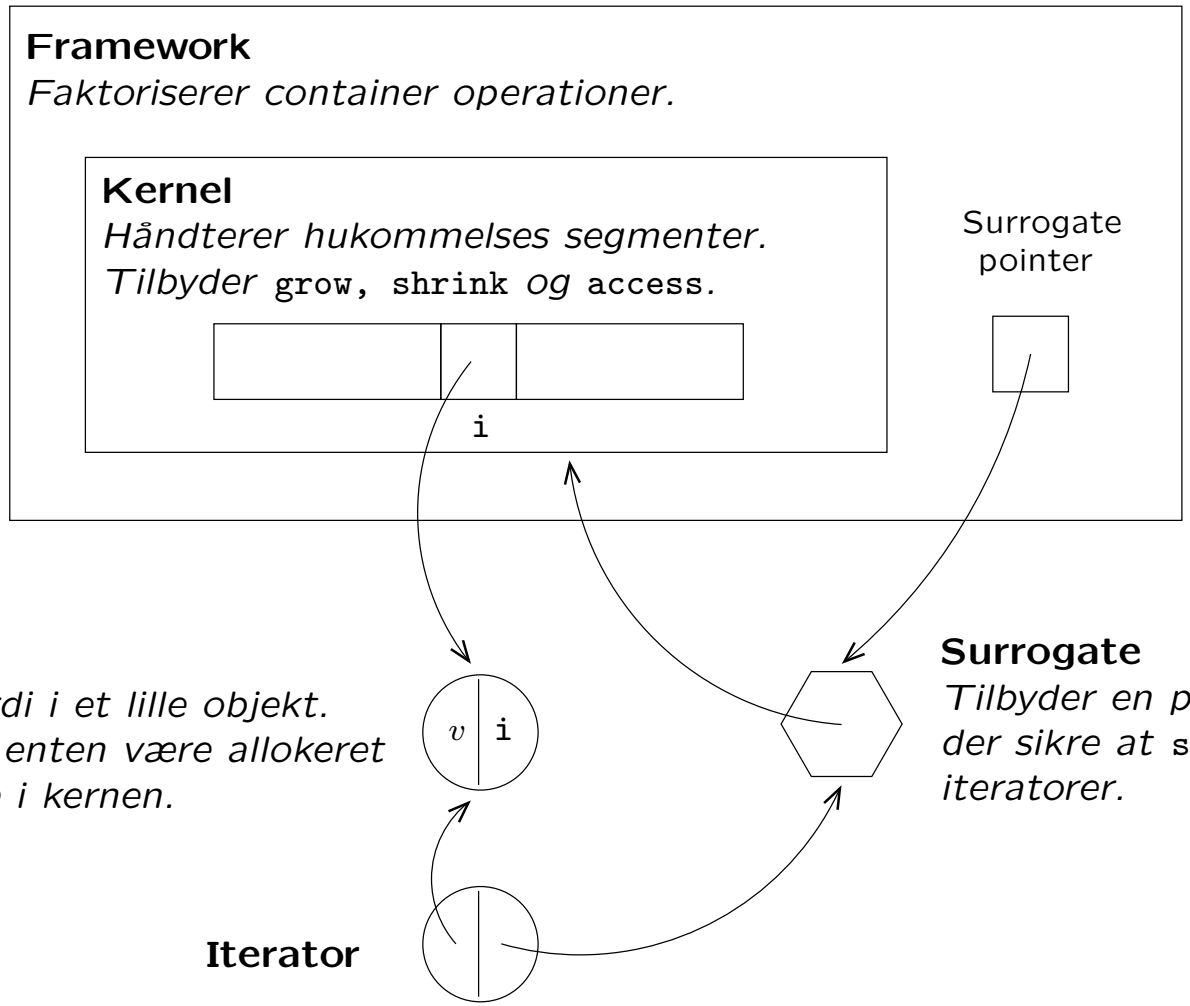


Stærk undtagelses sikkerhed: Uønsket adfærd i nogle tilfælde:

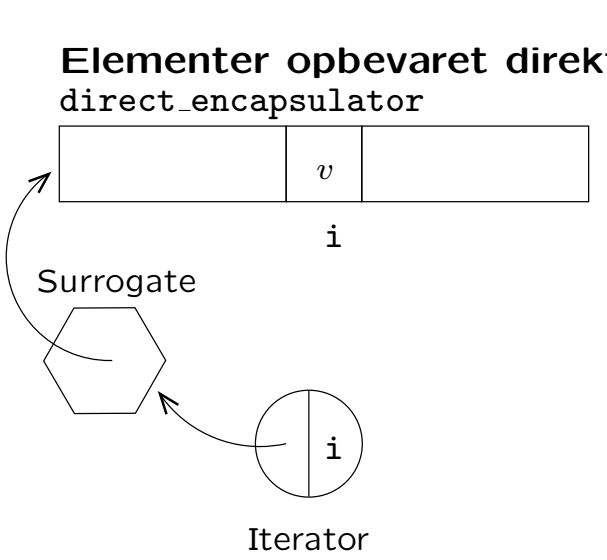


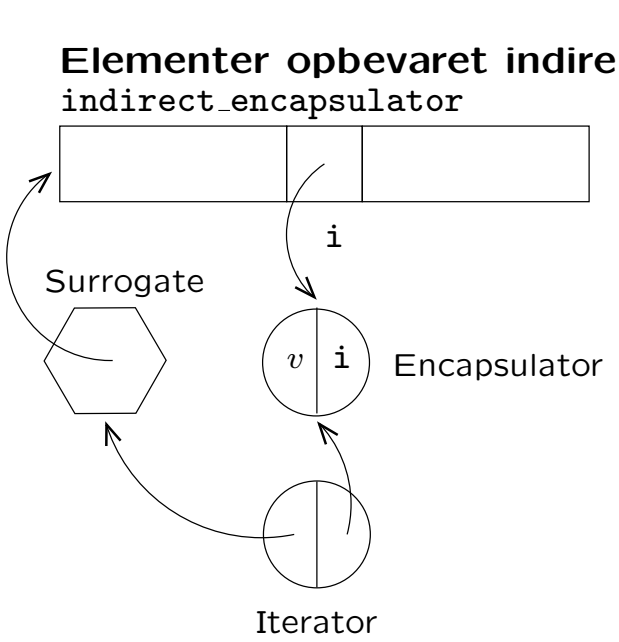
Pladseffektivitet og **værstefalds tidskompleksitet**.

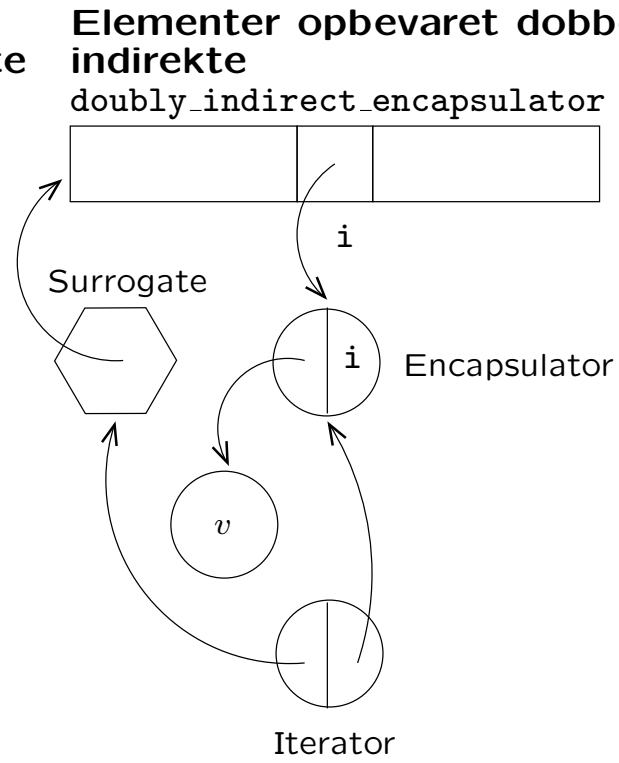
Vector framework



Encapsulatorer







- **Stærk undtagelses sikkerhed**
- **Referentiel Integritet**

(+) **Stærk undtagelses sikkerhed**
\+ **Referentiel Integritet**

\+ **Stærk undtagelses sikkerhed**
\+ **Referentiel Integritet**

Kerner

Alle kerner: indsættelse samt sletning i fronten og midten i $O(n)$ værstefalds tid ($n = \#$ elementer).

Dynamisk array:

`push_back` og `pop_back` i amortiseret $O(1)$ tid. Plads forbrug $6n + O(1)$.

Hashed array tree:

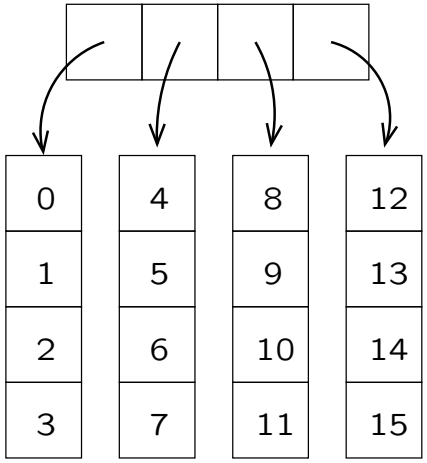
`push_back` og `pop_back` i amortiseret $O(1)$ tid.

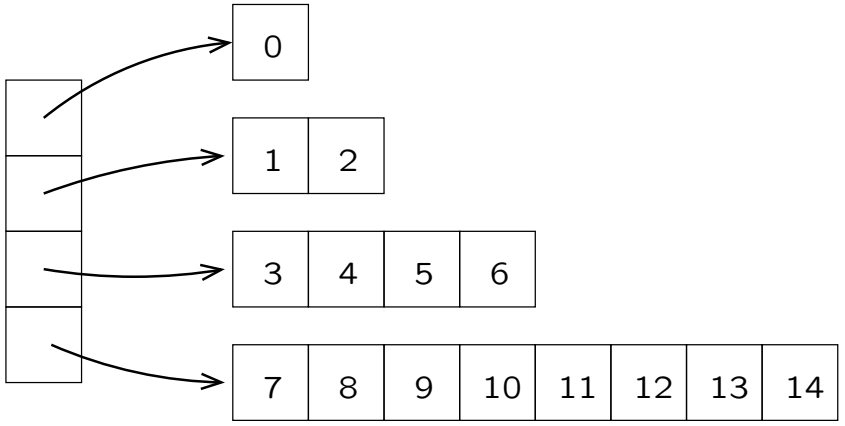
Plads forbrug $n + O(\sqrt{n})$.

Levelwise-allocated pile:

`push_back` og `pop_back` i $O(1)$ værstefalds tid.

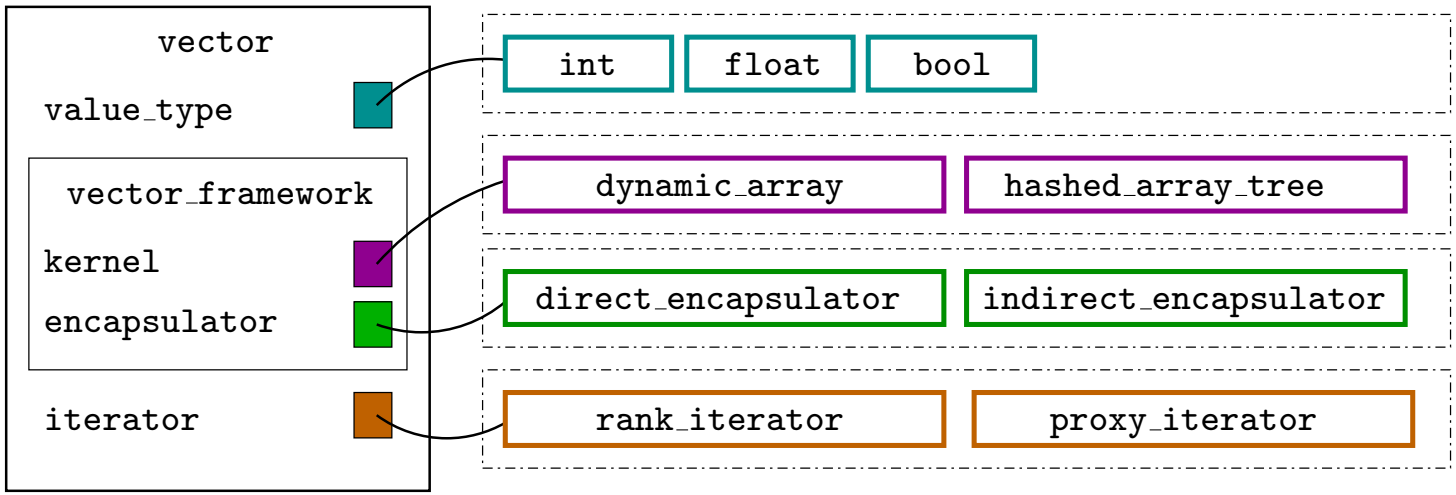
Plads forbrug $2n + O(\lg n)$.





Hvorfor component framework?

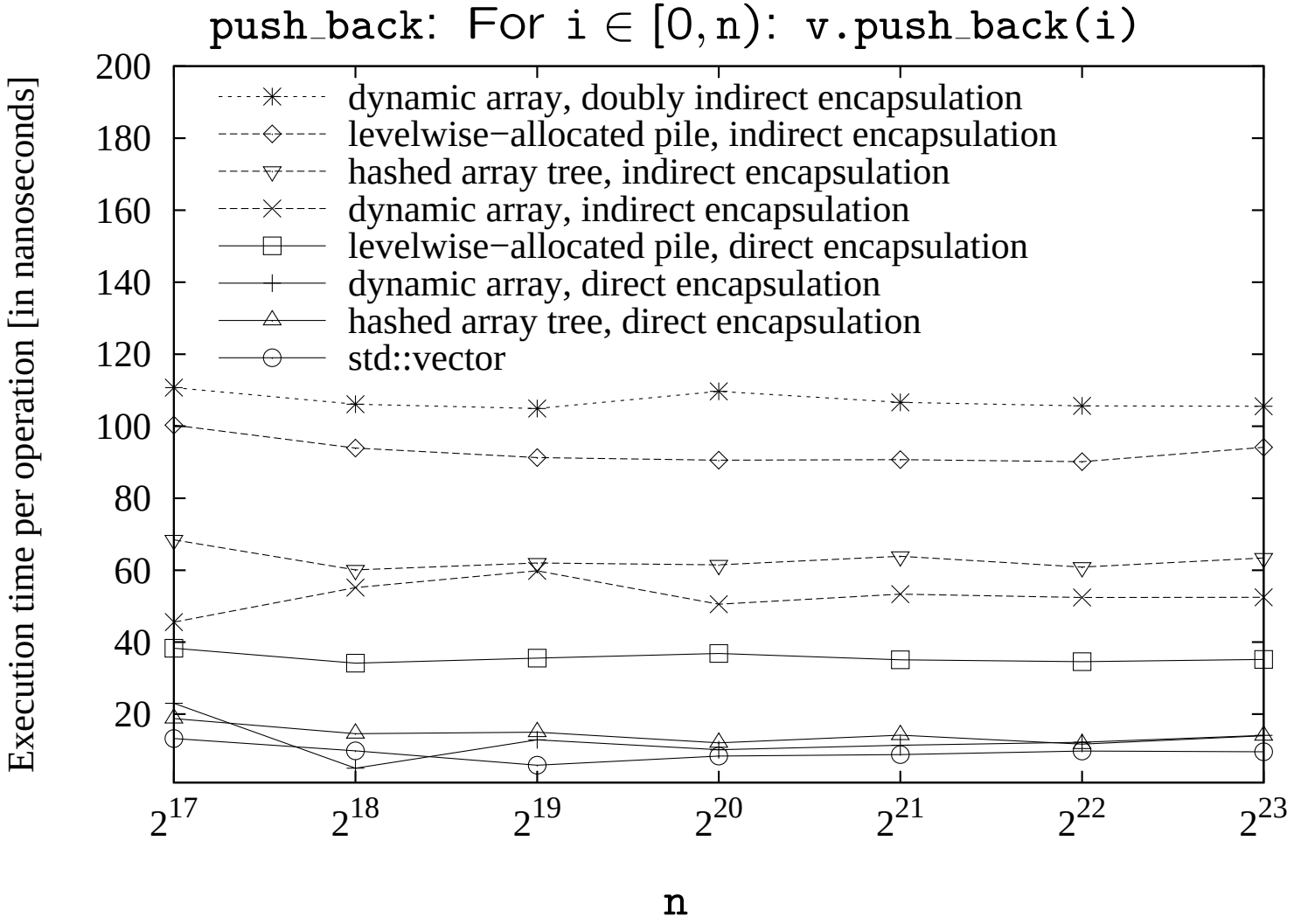
Fleksibilitet:



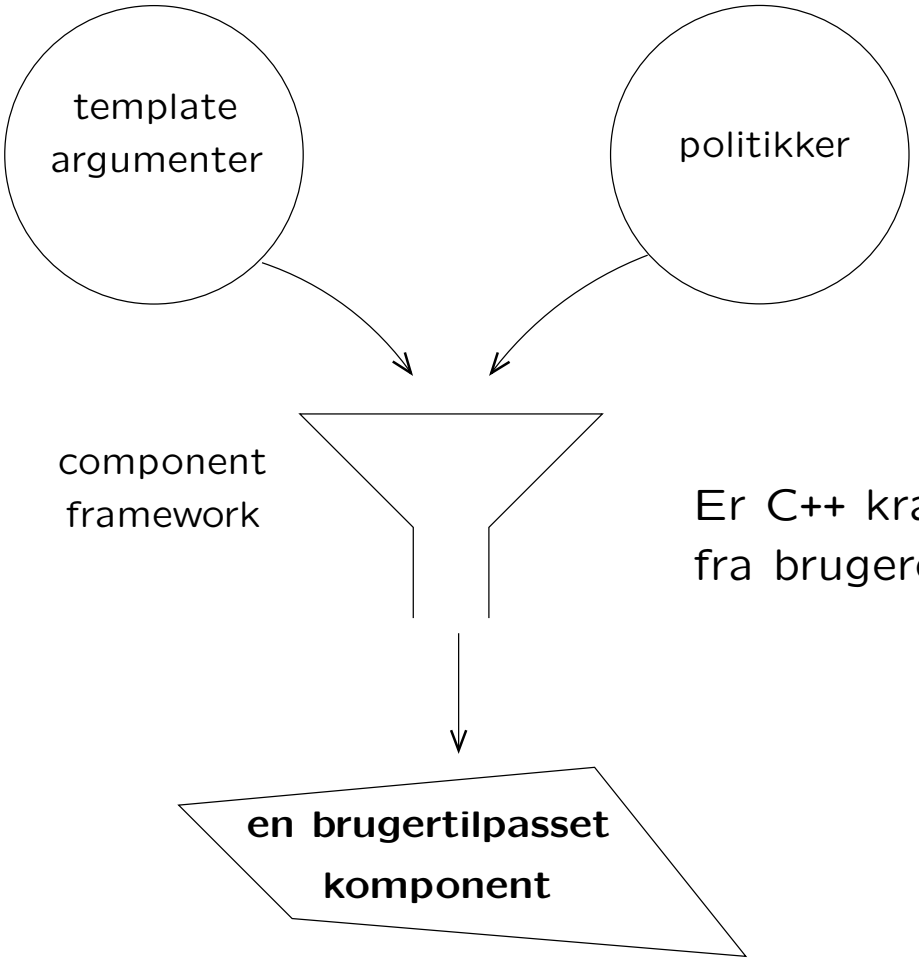
Vedligeholdelse: En container er sammensat af genbrugelige komponenter, dette giver os en høj grad af kode genbrug.

Fair benchmarking: Idéelt vil målingsresultaterne afspejle effektiviteten af data strukturene og ikke programmørenes dygtighed.

Effektivitet



Fleksibilitet (Adaptability)



Er C++ kraftigt nok
fra brugerens synspunkt ?

Problemer med container deklarationer

```
typedef cphstl::set<int,
  std::less<int>,
  std::allocator<int>,
  cphstl::tree<int, int,
    cphstl::unnamed::identity<int>,
    std::less<int>,
    std::allocator<int>,
    cphstl::avl_tree_node<int, true>,
    cphstl::avl_tree_balancer<
      cphstl::avl_tree_node<int, true>
    >
  >,
  cphstl::node_iterator<
    cphstl::avl_tree_node<int, true>,
    false>,
  cphstl::node_iterator<
    cphstl::avl_tree_node<int, true>,
    true>
> C;
```

Problemer med typiske deklarationer af containers:

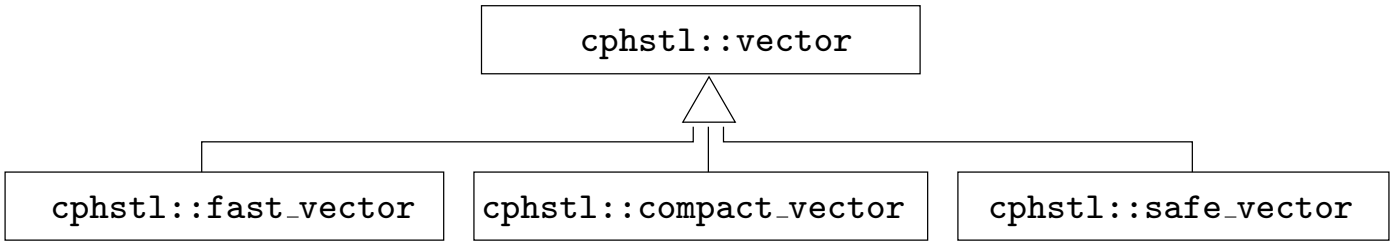
- Størrelse, læsbarhed,
- og forståelse.

Opstår ved:

- Det samme template argument gives flere gange,
- det enkelte template arguments betydning er ikke klart,
- og default argumenters betydning forsvinder ved overstyring.

Løsningerne

Prædefinerede container klasser, vh.a. nedarvning (eller typedef templates):



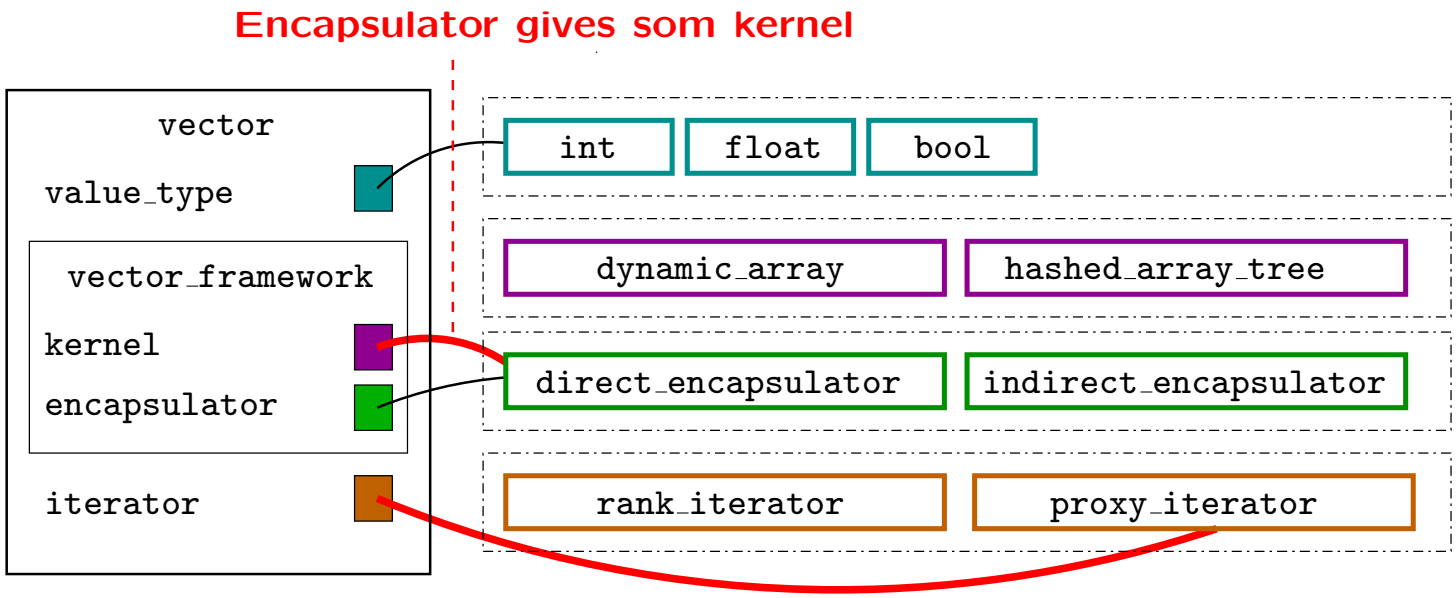
Named template arguments:

```
typedef cphstl::set!<V=int,
  R=cphstl::tree!<
    N=cphstl::avl_tree_node!<packed=true!>,
    B=cphstl::avl_tree_balancer
  !>,
  J=cphstl::node_iterator!<is_const=true!>,
  I=cphstl::node_iterator!<is_const=false!>
!> C;
```

- Template argumenter er globale.
- For manglende template argumenter benyttes default argumenter.

Et problem mere: Component mismatch

Et scenarie:



Ovenstående konfiguration giver mange ulæselige fejlmeddelser!

Løsningerne

Prædefinerede relationer: Component families, static assertions ved associative typer.

Autogenerede relationer: C++0x concepts (generelt *constraint polymorphism*), BOOST Concept checking library.

C++0x concepts

```
auto concept LessThanComparable < ↩
  typename T>
{
  bool operator<(T, T);
}

template <LessThanComparable T>
void min(T const& v1, T const& v2);
```

Component families

```
namespace cphstl {
  NEW_ENCAPSULATOR_FAMILY(proxy)
  JOIN_ENCAPSULATOR_FAMILY(proxy, ↩
    doubly_indirect_encapsulator)
  JOIN_ENCAPSULATOR_FAMILY(proxy, ↩
    indirect_encapsulator)
}
IS_ENCAPSULATOR_IN_FAMILY( ↩
  indirect_encapsulator, proxy)
```

C++0x (C++1x)

Language feature	C++	C++0x
Friend templates	○	●
Type inference	○	●
Constraint polymorphism	◐	◐
Partial specialization of member function	◐	◐
Compile-time reflection	◐	◐
Named template arguments	◐	◐
Typedef templates	○	●
Generic encapsulators	○	○
Inheriting constructors	○	●
Circular template arguments	◐	◐
Static assertions	◐	●

Konklusion

- Vores arkitektur tilbyder den nødvendige fleksibilitet for realisering af vores ønskede design samt den introducere blot et begrænset performance overhead (i de fleste tilfælde intet).
- Prisen for stærkere garantier for `vector` er meget dyr i det cache misses og hukommelses allokeringer er dyre.
- Elementerne i C++0x giver bedre værktøjer for biblioteks udvikling generelt.
- C++/C++0x er ikke stærkt nok til at konstruere et fleksibelt program bibliotek med ønsket brugervenlighed.