Die C++ Standardbibliothek

Sequentielle Container: `std::deque`

Robert Sauer

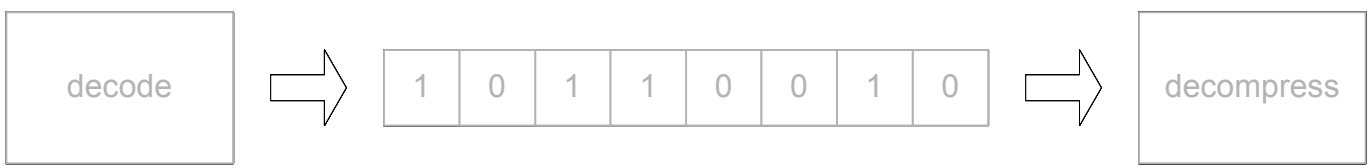
sauer@informatik.hu-berlin.de

Inhalt

- Motivation
- Datenstruktur Deque - Charakteristika
- Container Klasse **`std::deque`**
  - Unterstützte Konzepte
  - Interface
- Verwendung
- Literatur

Motivation - Problemstellung

- Datenempfang:
  - Dekodieren, danach Dekomprimieren
  - Kleine Elemente (**`char`** oder **`w_char`**)
  - Dekodierer und Dekomprimierer unabhängig (z.B. zwei threads)

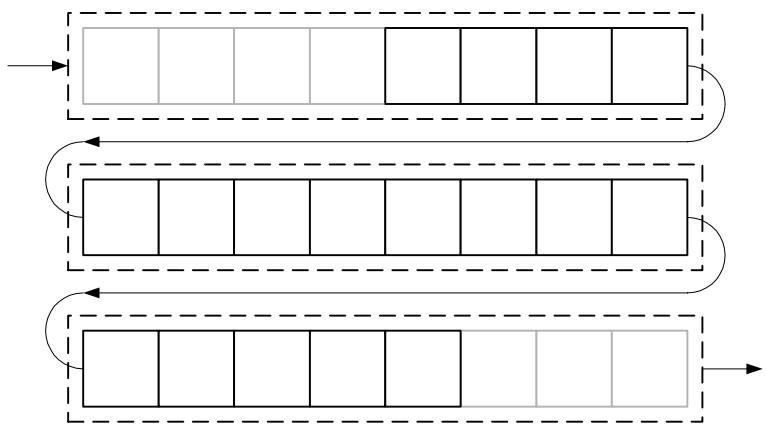


Motivation - Lösungssuche

- Warteschlange:
  - Einfügen vorne, Entnehmen hinten
    - → Operationen effizient und unabhängig voneinander
  - Dynamische Größe
- Naiver Ansatz: linked list
  - Scheidet aus, da inakzeptables Verhältnis Elementgröße vs. Knotengröße

Motivation – Gängige Lösung

- Deque
  - Double Ended Queue
  - Zweistufig organisiert
    - Block, Chunk
    - Element, Slot
  - Mittelding zwischen linked list und vector



Deque - Charakteristika

- Aussprache reimt auf „Scheck”
- Double Ended Queue, nach E. J. Schweppe (siehe Knuth: TAoCP Vol. 1, 2nd Ed., 1973)
- Auch: Deck of Cards (analog zu interner Repräsentation)
- Manipulation an Begin oder Ende: **O(1)**
- Manipulation in Mitte: **O(n)**
- Wahlfreier Elementzugriff: **O(1)**

Deque – Charakteristika (cont.)

[+/-] Größenverhältnis Nutz- zu Verwaltungsdaten liegt zwischen Vektor und linked list.

[+] Erlaubt effizientes Vergrößern **und** Verkleinern.

[-] Intern kein fortlaufendes Feld im C-Stil (nützlich für älteren Code).

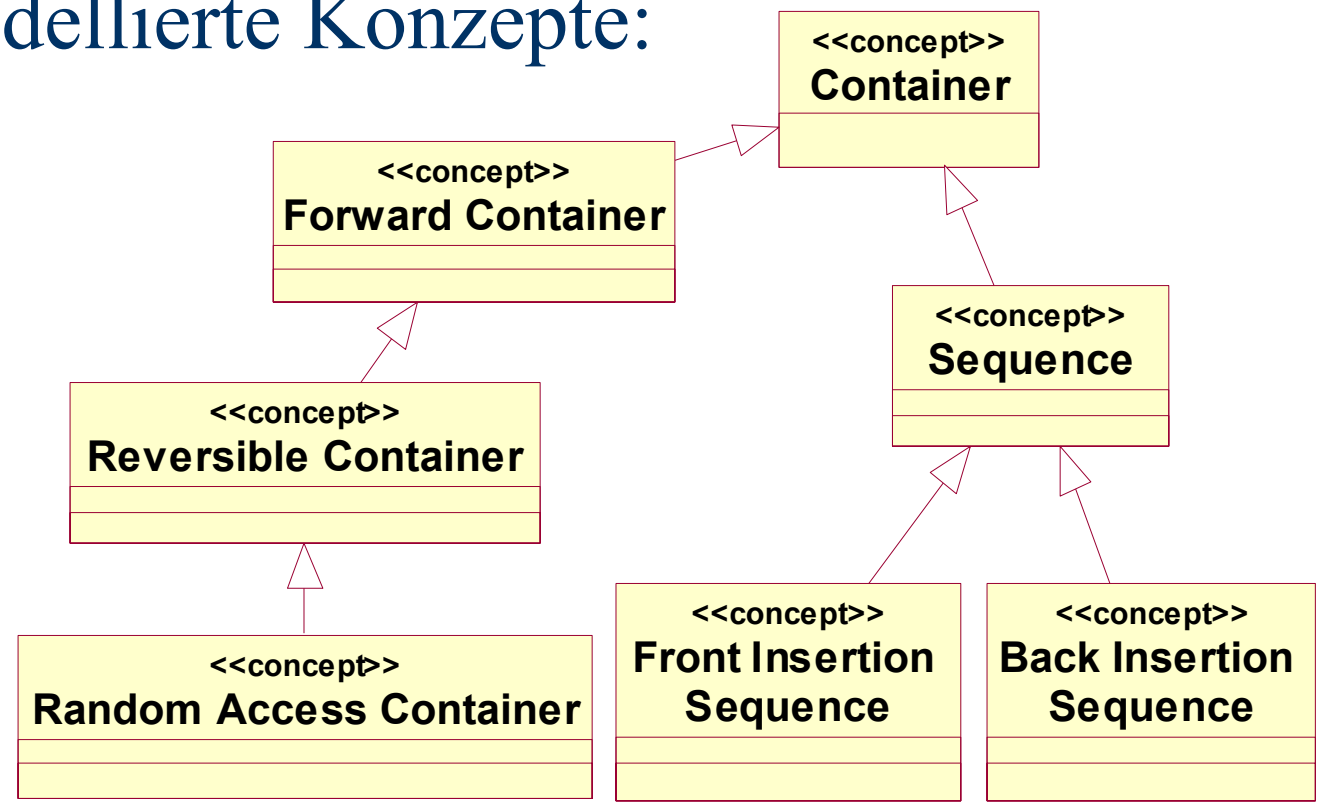
[-] Durch zweistufige Organisation Elementzugriff und Iteration etwas langsamer als bei fortlaufenden Feldern.

Container Klasse `std::deque`

- Header: `<deque>`
- Deklariert im Namensraum `std`
- Container-typische Template-Signatur
  `template<class T, class A = allocator<T> >`
- Interface weitestgehend identisch mit
  `std::vector`

`std::deque` – Konzepte

- Modellierte Konzepte:



`std::deque` – Typen

- Elementare Typen:
  - **`value_type`** → **`T`**
  - **`pointer`** → **`T*`**
  - **`const_pointer`** → **`const T*`**
  - **`reference`** → **`T&`**
  - **`const_reference`** → **`const T&`**
  - **`size_type`** → vorzeichenlose Ganzzahl
  - **`difference_type`** → Ganzzahl

`std::deque` – Typen (cont.)

- Iteration:
  - **`iterator`**
  - **`const_iterator`**
  - **`reverse_iterator`**
  - **`const_reverse_iterator`**
  - → Prinzipbedingt niemals **`T*`**
    - → Erzeugung und Kopieren von Instanzen teurer als bei **`std::vector`**

`std::deque` – Lebenszyklus

- Konstruktoren:
  - `deque()`
  - `explicit deque(size_type n)`
  - `deque(size_type n, const T& t)`
  - `deque(const deque&)`
  - `template<class InpIt>`
    `deque(InpIt first, InpIt last)`
- Destruktor:
  - `~deque()`

`std::deque` – Anfragen

- Füllstand:
  - `size_type size() const`
  - `size_type max_size() const`
  - `bool empty() const`
- Vergleichsoperatoren:
  - `!=, <, <=, ==, >=, >`

`std::deque` – Zuweisung

- Operator:
  - `deque& operator= (const deque&)`
- Methoden:
  - `assign(size_type n, const T& t)`
  - `template<class InpIt>`
    `assign(InpIt first, InpIt last)`
- Helfer:
  - `void swap(deque&)`
  - `friend void swap(deque& lhs, deque& rhs)`

`std::deque` – Elementzugriff

- Ungeprüft:
  - `reference operator[](size_type n)`
  - `const_reference operator[](size_type n) const`
- Geprüft
  - `reference at(size_type n)`
  - `const_reference at(size_type n) const`

  → Wirft evtl. `std::out_of_range` Exception

`std::deque` – Iteration

- Vorwärts:
  - `iterator {begin|end}()`
  - `const_iterator {begin|end}() const`
- Rückwärts:
  - `reverse_itereator {rbegin|rend}()`
  - `const_reverse_iterator {rbegin|rend}() const`

`std::deque` – Einfügen

- Kapazität:
  - **`void resize(size_type n, const T& t = T())`**
    - → Löscht überflüssige Elemente oder fügt bei Bedarf neue ein
    - → **Achtung:** Implementierung darf Speicher freigeben! Iteratoren und Referenzen werden ungültig!
  - **nicht: `reserve(..)`, `capacity()`**
    - → Deque-Prinzip macht diese überflüssig!

`std::deque` – Einfügen (cont.)

- Spezifische Elemente:
  - `insert( iterator pos, const T& t)`
  - `insert( iterator pos,`
    `size_type n, const T& t)`
  - `template<class InpIt>`
    `insert( iterator pos,`
    `InpIt first, InpIt last)`
  - → **Achtung:** Kopiert (dahinterliegende) Elemente! Iteratoren und Referenzen werden ungültig!

`std::deque` – Löschen

- Löschen:
  - `iterator erase(iterator pos)`
  - `iterator erase(iterator first, iterator last)`
  - `void clear()`

  → **Achtung:** Implementierung darf Speicher freigeben!
  Iteratoren und Referenzen werden ungültig!

`std::deque` – Back Insertion Sequence

- Elementzugriff:
  - **`reference back()`**
  - **`const_reference back() const`**
- Manipulation:
  - **`void push_back(const T&)`**
  - **`void pop_back()`**

    → Gelöschtes Element wird nicht geliefert

  - → **Achtung:** Kopiert keine Elemente!
    Iteratoren werden ungültig, Elementreferenzen nicht!

`std::deque` – Front Insertion Sequence

- Elementzugriff:
  - **`reference front()`**
  - **`const_reference front() const`**
- Manipulation:
  - **`void push_front(const T&)`**
  - **`void pop_front()`**

    → Gelöschtes Element wird nicht geliefert
  - → **Achtung:** Kopiert keine Elemente!
    Iteratoren werden ungültig, Elementreferenzen nicht!

`std::deque` – Verwendung

- Offensichtlich:
  - Bedarf für Container, der effizient sowohl an Begin als auch Ende manipuliert werden kann.
- Generell als Standard-Container?
  - Früher empfohlen, da Nachteile beim Zugriff durch effizientere Speicherverwaltung dominiert.
  - Inzwischen revidiert: Vermutlich aufgrund fehlender Unterstützung von C-Feld-Semantik.

`std::deque` – Verwendung (cont.)

- Weniger offensichtlich:
  - Bei sehr großen Datenmengen unbekannter Größe, die nur am Ende manipuliert werden.
  - → Bei Befüllung: Gefahr von Disk-thrashing.
    - **`std::vector`** mit internem Array größer als realer Speicher.
      - → Umkopieren erzeugt mit typischem LRU-Caching Serie von Seitenfehlern.
    - **`std::deque`** arbeitet nur auf Ende.
      - → Vordere Blöcke bleiben ausgelagert, keine Seitenfehler.

Literatur

- ANSI ISO C++ Standard
- N. Josuttis: „Die C++ Standardbibliothek“
- B. Stroustrup: „Die C++ Programmiersprache“
- Sgi STL (www.sgi.com/tech/stl/)
- H. Sutter: „More Exceptional C++“
- Guru of the week: (www.gotw.ca/gotw/)
- B. Milewski: DDJ, 05/2002, S. 34