Die Standard Template Library

- Hashed Associative Container -

Jörg Lange

05.07.2002

Übersicht

- Einführung
- SGI - Implementierung
- Dinkumware - Implementierung
- Details
- Beispiele
- Zusammenfassung
- Quellen

Einführung

- nicht Element des C++ Standards
  (noch nicht)
- aber STL - kompatible hash - assoziative Container existieren
- verschiedene Hersteller → Unterschiede:
  - Interface
  - zugrundeliegende Datenstrukturen
  - Fähigkeiten

Einführung

- Welche hash - assoziativen Container gibt es ?
  - ***hash_set***
  - ***hash_multiset***
  - ***hash_map***
  - ***hash_multimap***

Einführung

- Wiederholung: ***Assoziative Container***

Container	**Elemente** (value_Type, sortiert nach key_Type)	
set	value_Type = key_Type	(unique key)
multiset	value_Type = key_Type	
map	pair(key_Type, value_Type)	(unique key)
multimap	pair(key_Type, value_Type)	

Einführung

- ... Container müssen
  - den Typ der Objekte kennen,
  - eine Vergleichsfunktion (für Schlüssel - Objekte) und
  - einen Allocator für diese Objekte besitzen
  - die Spezifikation einer Hash - Funktion beinhalten

Einführung

- Deklaration eines Hash - Containers

```
template<typname T,
         typname HashFunction,
         typname CompareFunction,
         typename Allocator = allocator<T> >
class   hash_container;
```

Einführung

- Verschiedene Anbieter
  - SGI → www.sgi.com kostenlos
  - STLport → www.stlport.com kostenlos
  - Dinkumware → www.dinkumware.com
  - Recursion Software (Object Space)
    → www.recursionsw.com 30 Tage trial

  ...

SGI - Implementation

- **hash_set und hash_multiset**

```
template<typename T,
         typename HashFunction = hash<T>,
         typename CompareFunction = equal_to<T>,
         typename Allocator = allocator<T> >
class  hash_[multi]set;
```

(Ausschnitt)

genaueres in hash_set.h, stl_hash_set.h, stl_hash_fun.h

SGI - Implementation

- **hash_map und hash_multimap**

```
template<typename Key;
         typename T,
         typename HashFunction = hash<T>,
         typename CompareFunction = equal_to<T>,
         typename Allocator = allocator<T> >
class  hash_[multi]map;
```

(Ausschnitt)

genaueres in hash_map.h, stl_hash_map.h, stl_hash_fun.h

SGI - Implementation

- Default - Vergleichsfunktion ist ´´equal_to<T>´´
- Elemente werden nicht mehr sortiert gespeichert
- Datenstruktur ist eine einfach verkettete Liste
- keine bidirektionalen Iteratoren (kein rbegin(), rend())
- Default - Hash - Funktionen

Dinkumware - Implementation

- **hash_set und hash_multiset**

```
template<typename T, typename CompareFunction>
class hash_compare;

template<typename T,
         typename HashingInfo = hash_compare<T, less<T> >,
         typename Allocator = allocator<T> >
class hash_[multi]set;
```

(Ausschnitt)

Dinkumware - Implementation

- **hash_map und hash_multimap**

```
template<typename T, typename CompareFunction>
class hash_compare;

template<typename Key,
         typename T,
         typename HashingInfo = hash_compare<T, less<T> >,
         typename Allocator = allocator<T> >
class hash_[multi]map;
```

Dinkumware - Implementation

- Elemente werden sortiert gespeichert
- Default - Vergleichsfunktion ist ´´less´´
- Datenstruktur ist doppelt verkettete Liste
- bidirektionale Iteratoren vorhanden
- ´´HashingInfo´´ beinhaltet Hash-Funktion, Vergleichs-funktion, sowie Werte für Bucketgröße und minimaler Bucketanzahl

Dinkumware - Implementation

- ´´ HashingInfo´´:

```
template<typename T, typename CompareFunction = less<T> >
class hash_compare{
      enum{
                  bucket_size = 4;          //max. Elemente pro Bucket
                  min_buckets = 8;          // min. Anzahl von Buckets
      };
size_t operator()(const T&) const           // Hash - Funktion
bool operator()(const T&,                   // Vergleichsfunktion
                  const T&) const;
...
};
```

(Ausschnitt)

Dinkumware vs.SGI

- Implementationen gehen unterschiedliche Wege
- SGI:
  - konventionelles hash - Schema, mit einem Array (Buckets) von Pointern auf einfach verkettete Listen mit Elementen
- Dinkumware:
  - hash-Schema, mit neuerer Datenstruktur
  - einem Array von Iteratoren auf eine doppelt verkettete Liste von Elementen, wobei benachbarte Iteratoren die Ausdehnung von Elementen in einem Bucket anzeigen

Details

- Member:
  - grundlegende Anforderungen wie bei Assoziativen Containern (fast)

    z.B. find(), insert() ...
  - Achtung bei SGI - Implementation, keine Reverse-Iteratoren
  - weitere Unterschiede in verschiedenen Implementationen möglich

Details

- zusätzliche Member:

  - hash_funct() //returns hashing function
  - key_eq() //returns key comparism function
  - count(k) //returns number of elements with key = k
  - bucket_count() //returns current number of hash buckets
  - resize(b) //restructs table to contain b buckets

Beispiele

- Beispiel für Vergleichsfunktion

```
struct Eqstr
{
        bool operator() (const char* s1, const char* s2) const
        {
                return strcmp(s1, s2) == 0;
        }
};
```

Beispiele

- Beispiel für Hash - Funktion

```
struct MyHash
{
        size_t operator() (int  x) const
        {
                return x%100;
        }
};
```

Beispiele

- Beispiele für Hash_Set

```
hash_set<int, hash<int> > set1;

hash_set<int, MyHash> set2;

hash_set<char*, hash<char*>, Eqstr> set3;
```

Zusammenfassung

- Komplexität:
  - find() konstant
  - insert() amortisiert konstant
- aber: Hash - Funktion und Bucket - Anzahl haben starken Einfluss auf Performanz

Zusammenfassung

- Hash - Associative - Container
  - für schnelles Finden von Elementen
  - falls Sortierung wichtig $\rightarrow$ Assoziative Container (SGI - Implementation)
  - Anwendung:

    z.B. hash_map als Wörterbuch

Hashed Associatve Container

Ende