Container-Elemente

Erik Pischel

17. Mai 2002

Gliederung

1. Was muß ein Container-Element können?
2. Container-Elemente und Ausnahmen
3. Zeiger als Container-Elemente
4. Ein Container, der keiner ist

Was muß ein Container-Element können?

a) Basiskonzepte

b) Code-Äquivalente

Basiskonzepte

1. Default konstruierbar
2. Zuweisbarkeit
3. Gleichheit-Relation
4. Klein-Als-Relation
   a) Irreflexive/Strenge Halbordnung
   b) „Strict Weak Ordering“
   c) Totale Ordnung

Code-Äquivalente

1. Default konstruierbar: default Konstruktor X()
2. Zuweisbarkeit: Copy-constructor X(**const** X&), Zuweisungsoperator X& **operator**=(**const** X&)
3. Gleichheitsrelation: Gleich-Operator **bool operator**==(**const** X&)
4. Klein-Als-Relation: Kleiner-Als-Operator **bool operator**<(**const** X&)
5. Zusätzlich: Destruktor

Gliederung

1. Was muß ein Container-Element können?
2. *Container-Elemente und Ausnahmen*
3. Zeiger als Container-Elemente
4. Ein Container, der keiner ist

Container-Elemente und Ausnahmen

- Anforderungen der STL an Container-Elemente
- Garantien der STL
- Bsp: was kann hier schief gehen?

```
void f(vector<X>& v, const X& g) {
  v[2] = g;
  v.push_back(g);
  sort(v.begin(), v.end());
  vector<X> u = v;
   // ...
}
```

Anforderungen bzgl. Ausnahmen

- Wenn eine Ausnahme ausgelöst wird
  - muß Container-Element in einem gültigen Zustand sein
  - kein „resource leak“
- keine Ausnahmen in Destruktoren
- Bsp:

```
template<class T> class [Un]Safe {
  T* p;
 ...
}
```

Beispiel

Konstruktor

```
Safe():p(new T) {}
Unsafe(T* pp):p(pp) {}
```

Destruktor:

```
~Safe() { delete p; }
~Unsafe() { if (!p->destructible()) throw E(); delete p; }
```

Zuweisungsoperator:

```
Safe& operator=(const Safe& a) { *p = *a.p; return *this; }
Unsafe& operator=(const Unsafe& a) {
  p->~T()
  new(p) T(a.p); return *this;
}
```

Regeln

- Sicheres Update eines Objektes
  - Alten Zustand vorhalten, bis neuer Zustand etabliert
- Resourcen freigeben
  - „resouce acquisition is initialization“-Technik
- Objekte in gültigem Zustand belassen

Garantien der STL

Solange sich die Container-Elemente wohl verhalten:

- „Basic guarantee“ für alle Operationen
- „Strong guarantee“ für Schlüsseloperationen
- „Nothrow guarantee“ für einige Operationen

Zusamenfassung

- Ausnahmensichere Container-Elemente
- Garantien der STL
- Literaturtip: Kapitel von Stroustrup unter
  `http://www.research.att.com/~bs/3rd_safe0.html`
  als PDF

Gliederung

1. Was muß ein Container-Element können?
2. Container-Elemente und Ausnahmen
3. *Zeiger als Container-Elemente*
4. Ein Container, der keiner ist

Zeiger als Container-Elemente

a) Problem

b) Container und Eigentümerschaft

c) Lösungen

d) Zusammenfassung

Problem

Was passiert?

```
{
 std :: vector<my_type*> v;
 for (int i = 0; i < N; ++i)
     v. insert (new my_type(i));
  ...
} // v is destroyed here
```

Gibt es ein Memory leak?

Problem

- Warum ruft ~vector nicht **delete**?
- $\rightarrow$ vector$<$T$>$ hat keine speziellen Regeln für T = X$*$!
- Fragen:
  - Warum nicht?
  - Wie kann ich mir helfen?

Zeiger als Container-Elemente

a) Problem

b) *Container und Eigentümerschaft*

c) Lösungen

d) Zusammenfassung

Container und Eigentümerschaft

- Container haben Wertesemantik
- Referenzsemantik mittels Zeiger
- Warum keine Extra-Behandlung?
  - Uniformität
  - Was bringt es, wenn vector<T*> die referenzierten Objekte besitzt?

Zeiger als Container-Elemente

a) Problem

b) Container und Eigentümerschaft

c) *Lösungen*

d) Zusammenfassung

Besitz von referenzierten Objekten

- Frage: Wozu?
- Antwort: Polymorphie $\rightarrow$ ist sonst nicht möglich!
- Wie kann ich das erreichen?
- Lösung 1: my_vector
  - Wrapper für vector
  - Destruktor löscht alle Elemente

Lösung 1: my_vector

```
template <class T>
class my_vector : private std::vector<T*>
{
  //    ...
public:
  //    ...
   ~my_vector() {
      for ( iterator  i = begin(); i != end(); ++i)
         delete *i;
  }
};
```

Problematisch

Lösung 2 + 3 + 4

auto_ptr std :: auto_ptr<T> hält ein T*

- löscht Objekt im Destruktor
- Problem: auto_ptr ist nicht zuweisbar

Garbage Collector Objekt wird gelöscht, wenn es nicht mehr referenziert wird

- Lösungen verfügbar
- „Eigentümerschaft“ wird irrelevant:
  das Problem wird wegdefiniert!

Arena Verwaltet Liste aller Zeiger

- Günstig, wenn: „Ich weiß, wann ich meine Objekte löschen muß!“

Zusammenfassung

- Wann sollen die referenzierten Objekte gelöscht werden?
- referenzierte Objekte sollten nicht vom Container „besessen“ werden
- Für nicht-polymorphe Objekte: container<T>
- Niemals container<auto_ptr> !
- Polymorphe Objekte: container<T>
  - Lebenszeit unbekannt und unproblematisch: GC
  - Lebenszeit bekannt: Arena am einfachsten

Gliederung

1. Was muß ein Container-Element können?
2. Container-Elemente und Ausnahmen
3. Zeiger als Container-Elemente
4. *Ein Container, der keiner ist*

vector<bool>

- Ist kein Container
  - Container-Anforderung: **operator**[] gibt T* zurück
  - vector<**bool**> gibt Proxy-Objekt zurück
- Enhält keine **bool**s, sondern eine Art „bitfields“
- Besser nutze
  - deque<**bool**>
  - bitset

Gesamtzusammenfassung

- Anforderungen an eine Container-Element
- *Exception-safe* Container-Elemente
- Zeiger als Container-Elemente: Wann, Warum und Wie?
- vector<**bool**> ist kein Container