Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Iteratorentwurf

Phillip Raffeck, Florian Schmaus, Simon Schuster

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://www4.cs.fau.de

Sommersemester 2020

Aufgabe 5 – Testen

- Verwendung von GNU/Linux (kein eCos mehr)
- Umsetzung einer Spezifikation
  - Produkt
  - Testfälle
- Implementiert werden soll eine *Prioritätswarteschlange*
- Einfügen, Entfernen, *Iterieren*

  ⇝ `for (... x = ...; x = ...; ++x) ...!`
  - Wie implementiert man sowas?

Iterator – 1. Versuch

- *Datenstruktur als Array* im Header vereinbaren
- Zugriff durch Zeigerarithmetik

```
1 typedef struct Element { ... } Element;
2 Element elements[ELEMENTS_SIZE];
3 ...
4   for (size_t i = 0; i < ELEMENTS_SIZE; ++i)
5     { use(elements[i]); }
```

Iterator – 1. Versuch

- *Datenstruktur als Array* im Header vereinbaren
- Zugriff durch Zeigerarithmetik

```
1 typedef struct Element { ... } Element;
2 Element elements[ELEMENTS_SIZE];
3 ...
4   for (size_t i = 0; i < ELEMENTS_SIZE; ++i)
5     { use(elements[i]); }
```

- *Vorteile:*
  - Einfache Implementierung
  - Für den Compiler leicht zu optimieren
- *Nachteil:* Implementierung offen gelegt

$\rightsquigarrow$ Verpflichtung gegenüber Benutzer

Iterator – 2. Versuch

- *Iterator als Teil des Objekts*
- Header:

```
1 typedef struct Elements Elements;
2 void El_reset_iterator(Elements *self);
3 void El_next(Elements *self);
4 bool El_isAtEnd(Elements *self);
5 int64_t El_iterator_value(Elements *self);
```

- Verwendung:

```
1 El_reset_iterator(dings);
2 while(!El_isAtEnd(dings)) {
3   use(El_iterator_value(dings));
4   El_next(dings);
5 }
```

Iterator – 2. Versuch

- Implementierung:

```
1  typedef struct Element { int64_t value; } Element;
2  struct Elements {
3    Element elements[ELEMENTS_SIZE];
4    Element *it;
5  };
6  void El_reset_iterator(Elements *self)
7    { self->it = &self->elements }
8  void El_next(Elements *self)
9    { self->it = self->it + 1; }
10 bool El_isAtEnd(Elements *self)
11   { return self->it
12       == &(self->elements[ELEMENTS_SIZE]); }
13 int64_t El_iterator_value(Elements *self)
14   { return self->it->value; }
```

Iterator – 2. Versuch

- Implementierung:

```
1  typedef struct Element { int64_t value; } Element;
2  struct Elements {
3    Element elements[ELEMENTS_SIZE];
4    Element *it;
5  };
6  void El_reset_iterator(Elements *self)
7    { self->it = &self->elements }
8  void El_next(Elements *self)
9    { self->it = self->it + 1; }
10 bool El_isAtEnd(Elements *self)
11   { return self->it
12       == &(self->elements[ELEMENTS_SIZE]); }
13 int64_t El_iterator_value(Elements *self)
14   { return self->it->value; }
```

- *Vorteil:* Kapselung sehr gut
- *Nachteile:*
  - Für den Compiler evtl. nicht mehr optimierbar (Schleife ausrollen)
  - So nur ein Iterator gleichzeitig möglich

Iterator – 3. Versuch

- *Iterator als eigenes Objekt*
- Header:

```
1  typedef struct Elements Elements;
2  typedef struct El_Iterator El_Iterator;
3
4  El_Iterator *El_begin(Elements *self);
5  void El_Iterator_destroy(El_Iterator *self);
6  void El_Iterator_next(El_Iterator *self);
7  bool El_Iterator_isAtEnd(El_Iterator *self);
8  int64_t El_Iterator_value(El_Iterator *self);
```

- Verwendung:

```
1  El_Iterator *it;
2  for (it = El_begin(dings);
3       not El_Iterator_isAtEnd(it);
4       El_Iterator_next(it)) {
5    use(El_Iterator_value(it))
6  }
7  El_Iterator_destroy(it);
```

Iterator – 3. Versuch

- Implementierung:

```
1  typedef struct Element { int64_t value; } Element;
2  struct Elements { Element elements[ELEMENTS_SIZE]; };
3  struct El_Iterator {
4    Element *position;
5    Element *end;
6  };
7
8  El_Iterator *El_begin(Elements *self) {
9    El_Iterator *ret = malloc(sizeof(El_Iterator));
10   if (ret == NULL) { return NULL; }
11   ret->position = self->elements;
12   ret->end = &self->elements[ELEMENTS_SIZE];
13   return ret;
14 }
15
16 void El_Iterator_next(El_Iterator *self)
17   { self->position += 1; }
18 bool El_Iterator_isAtEnd(El_Iterator *self) { ... }
19 int64_t El_Iterator_value(El_Iterator *self) { ... }
20 void El_Iterator_destroy(El_Iterator *self) { ... }
```

Iterator – 3. Versuch

- *Vorteile:*
  - Vollständige Kapselung
  - Beliebig viele Iteratoren möglich
- *Nachteil:*
  - Iterator muss nach Gebrauch beseitigt werden
  - Compiler hat evtl. Probleme zu optimieren

$\Rightarrow$ Verwendung in der Übung

Aufgabe 5 – Testen

- Alles gezeigte ist nur als Beispiel zu verstehen!
- Andere Datenstrukturen können in der Übung hilfreicher sein.
- Wir empfehlen die Verwendung von **Kapselungsvariante 3**.