Übungen zur Vorlesung

# "Objektorientiertes Programmieren im Wissenschaftlichen Rechnen"

Dr. Olaf Ippisch, Ole Klein

Abgabe am 07. 05. 2013 in der Vorlesung

---

## ÜBUNG 1 VERKETTETE LISTE (MIN/MAX)

Die verkettete Liste soll Methoden erhalten, die das kleinste oder das größte Element finden:

```
const Node * findMin() const;
const Node * findMax() const;
```

1. Schreiben sie sich eine freie Funktion `void testListMinMax()`, mit der sie anschließend ihre Implementierung testen:
   - was sind gute Testfälle?
   - an welche Seiteneffekte muss man denken?
   - ändern sie die Liste auch zwischendrin.
2. Implementieren sie die Methode `findMin` und `findMax`. Modifizieren sie keine anderen Methoden.
3. Ergänzen sie die Implementierung um *Caching*.
   - Die Suche nach dem größten oder kleinsten Eintrag ist $O(N)$, wenn $N$ die Länge der Liste ist.
   - Wenn min/max mehrfach aufgerufen wird, soll nur einmal das entsprechende Element gesucht werden, der Wert soll bei der ersten Auswertung gecached werden.

*6 Punkte*

## ÜBUNG 2 SHARED_PTR UND WEAK_PTR

Nachdem Sie sich in der Vorlesung mit Smartpointern beschäftigt haben, schauen Sie sich nochmals Ihre bisherigen Programme an. Dabei finden Sie mehrere Aufgaben, die sich mit verketteten Listen beschäftigen. Da es sehr praktisch wäre, wenn die Liste ihre Speicherverwaltung selbst durchführen würde, entscheiden Sie sich Ihre Implementierung zu modifizieren.

1. Stellen Sie die Klasse vom vorherigen Zettel auf Smartpointer um, d.h. tauschen Sie `Node *` gegen `shared_ptr<Node>` aus.
2. Was genau passiert, wenn die Liste gelöscht wird? In welcher Reihenfolge werden die Destruktoren aufgerufen, und wie wird der Speicher freigegeben?
3. Modifizieren Sie Ihre Implementierung erneut, um eine Ringstruktur zu erhalten: das letzte Element soll wieder auf das erste zeigen. Was ist hieran problematisch? Wie können Sie das Problem durch die Verwendung von `weak_ptr<Node>` vermeiden?

*6 Punkte*

## ÜBUNG 3 KNOBELAUFGABE

*Probieren Sie es ruhig praktisch aus.*

Sie haben folgendes Programm:

```
void foo ( const int ** );

int main()
{
    int ** v = new int * [10];
    foo(v);

    return 0;
}
```

der Compiler wird ihnen nun einen Fehler melden, weil sie `int **` nach `const int **` umwandeln:

```
g++ test.cc -o test
test.cc: In function 'int main()':
test.cc:6: error: invalid conversion from
  'int**' to 'const int**'
test.cc:6: error:   initializing argument 1
  of 'void foo(const int**)'
```

- Eigentlich kann man doch immer von nicht-`const` nach `const` umwandeln…
- warum geht das hier nicht?

Tipp:

Warum folgendes Programm nicht kompiliert ist klar:

```
const int * bar ();

int main()
{
    int ** v = new int * [10];
    v[0] = bar();

    return 0;
}
```

Was hat dieses Programm mit dem obigen gemein?

*6 Punkte*

## ÜBUNG 4 CONSTNESS

Sie habe eine Liste von Funktionenprototypen, einige Variablen und Zuweisungen. Welche Konstruktionen sind nicht zulässig und warum?

```
int foo ( const int & );
int bar ( int & );

int main()
{
  int i = 0;
  int & j = i;
  static const int f = i;
  int * const p = 0;
  p = &i;
  * p = f;
  const int & l = j;
  const int & k = f;
  foo ( j );
  bar ( l );
  foo ( k );
}
```

*2 Punkte*