Überladen von Methoden/Operatoren

☑ **Lösung zu Aufgabe 5.1** Die zwei fehlerhaften Zeilen sowie weiterführende Erklärungen finden Sie in den Kommentaren von folgendem Code.

```
char f() { return '\n'; } // OK
bool f() { return true; } // ERR: Rueckgabetyp nicht Teil der Signatur
void f(char a) {}         // OK
void f(char b) {}         // ERR: Parametername nicht Teil der Signatur
void f(bool a) {}         // OK wegen anderem Parametertyp
void f(char a, bool b) {} // OK wegen anderer Parameteranzahl
void f(bool a, char b) {} // OK wegen anderer Parameterreihenfolge
void f(int& i) {}         // OK
void f(int&& i) {}        // OK wegen anderer Kategorie von Referenz
void f(const int& i) {}   // OK wegen zusaetzlichem const-qualifier
```

☑ **Lösung zu Aufgabe 5.2** Der einzige fehlerhafte Vergleich ist der dritte, also `6 < r`. Hier wird ein `int` links mit einem `Rational` rechts verglichen. Wenn sie als Member-Funktionen implementiert sind, dann haben binäre Operatoren wie beispielsweise `operator<` den entscheidenden Nachteil, dass sie die implizite Konvertierung des linken Operanden (hier von `int` nach `Rational`) nicht unterstützen. Da diese Asymmetrie zwischen dem linken und rechten Operanden doch sehr ärgerlich sein kann, werden binäre Operatoren typischerweise als freie Funktion implementiert, wodurch die Asymmetrie verschwindet. Falls eine solche freie Funktion Zugriff auf private Daten benötigt, dann wird sie zusätzlich als `friend` deklariert.

```
bool operator<(const Rational& first, const Rational& second) {
  return first.m_num * second.m_denom < second.m_num * first.m_denom;
}
```

☑ **Lösung zu Aufgabe 5.3** Wir implementieren gleich zwei Overloads des Indexoperators, und zwar einmal für veränderbare und einmal für unveränderbare Objekte. Der erste Overload gibt `int&` zurück und ermöglicht dadurch die Veränderung der Einträge eines `FancyVector`. Der zweite Overload gibt `const int&` zurück und unterbindet dadurch die Veränderung der Einträge eines `const FancyVector`.

Für die Umwandlung des Index selbst ist es nicht unwichtig, zu bemerken, dass der Parameter `index` vom Typ `int` ist (d.h. eine Zahl, die positiv oder negativ sein kann), während `m_vec.size()` vom Typ `size_t` ist (d.h. eine Zahl, die immer positiv ist). Um uns vor unangenehmen Überraschungen durch implizite Konvertierungen zu schützen, speichern wir deshalb die Grösse von `m_vec` zuerst in einer separaten Variable `size` vom Typ `int`, bevor wir damit zu rechnen beginnen.

```
class FancyVector {
private:
  std::vector<int> m_vec;
public:
  explicit FancyVector(size_t size, int value = 0)
  : m_vec(size, value) {}
  // Indexoperator fuer veraenderbare Objekte:
  int& operator[](int index) {
    int size = static_cast<int>(m_vec.size());
    return m_vec[index >= 0 ? index : size + index];
  }
  // Indexoperator fuer unveraenderbare Objekte:
  const int& operator[](int index) const {
    int size = static_cast<int>(m_vec.size());
    return m_vec[index >= 0 ? index : size + index];
  }
};
```