Templates und generische Programme

☑ **Lösung zu Aufgabe 7.1** Die Ausgaben der jeweiligen Zeilen sind in den Kommentaren von folgendem Code beschrieben.

```
f(1);            // Ausgabe: 1 Overloaded
f<double>(2);    // Ausgabe: 2 Generic
f<int>(3);       // Ausgabe: 3 Specialized
f<>(4);          // Ausgabe: 4 Specialized
f(5.0);          // Ausgabe: 5 Generic
f<double>(6.0); // Ausgabe: 6 Generic
f<int>(7.0);     // Ausgabe: 7 Specialized
f<>(8.0);        // Ausgabe: 8 Generic
```

Nicht überraschen sollte, dass die Anweisungen `f<double>(...)` und `f<int>(...)` unabhängig vom Typ des übergebenen Arguments im ersten Fall die generische Definition von `f` aufrufen (weil keine Spezialisierung für `double` existiert) und im zweiten Fall die für `int` spezialisierte Definition.

Die Anweisungen `f<>(...)` rufen ebenfalls immer eine Template-Definition von `f` auf (also niemals den Overload), überlassen dabei aber dem Compiler die Entscheidung, ob aufgrund des Typs des übergebenen Arguments die generische oder die für `int` spezialisierte Definition von `f` zu bevorzugen ist.

Die Anweisungen `f(...)` ganz ohne eckige Klammern `<` und `>` erlauben als einzige den Aufruf des Overloads von `f`. Falls das übergebene Argument den exakt gleichen Typ hat wie dasjenige des Overloads, dann wird tatsächlich auch dieser Overload aufgerufen (siehe die erste Ausgabe `1 Overloaded`).

☑ **Lösung zu Aufgabe 7.2** Damit die spezialisierte zweite Definition des Templates `S<T1, T2, T3>` verwendet wird, muss das erste Template-Argument `T1` den Typ `int` und das dritte Template-Argument `T3` den Typ `char` haben. Das zweite Template-Argument `T2` wurde nicht eingeschränkt und ist deshalb beliebig wählbar. Diese Bedingung trifft weder auf den Typ von der Variable `s1` noch von der Variable `s2` zu, weshalb die ersten zwei ausgegebenen Zeilen jeweils `Generic` sind. Die Variable `s3` mit Typ `S<int, int, char>` hingegen erfüllt die Bedingung und die ausgegebene Zeile ist `Specialized`.

Beim Typ von `s4` und `s5` ist das dritte Template-Argument `T3` jeweils unspezifiziert. Dies ist möglich, weil in der generischen Definition von `S` ein Default-Template-Argument `T3 = T2` gegeben ist. Die Argument-Liste wird deshalb zuerst bei `s4` zu `S<char, int, int>` und bei `s5` zu `S<int, char, char>` vervollständigt. Danach wird wieder ganz normal die oben beschriebene Bedingung überprüft, was zur Ausgabe `Generic` bei `s4` und zur Ausgabe `Specialized` bei `s4` führt.

☑ **Lösung zu Aufgabe 7.3** Der Programmierer hat höchst wahrscheinlich versucht, eine Funktion zu definieren, die eine beliebige Anzahl von numerischen Parametern entgegennimmt und dann die Summe all dieser gegebenen Zahlen berechnet und zurückgibt.

Leider aber führt das gezeigte Programm zu einem Kompilierfehler. Grund dafür ist, dass keine Rekursionsverankerung für das gezeigte variadische Funktionen-Template definiert wurde. Der Compiler reduziert zuerst den Aufruf `sum(a, b, c, d, e, f)` auf den Aufruf `sum(b, c, d, e, f)` mit einem Argument weniger, der wiederum auf `sum(c, d, e, f)` reduziert wird und so weiter, bis schliesslich der Aufruf `sum(f)` auf die leere Summe `sum()` ganz ohne Argumente verweist. Dies geht allerdings schief, weil die Definition einer solchen leeren Summe im Programm gänzlich fehlt.

Das Programm funktioniert, sobald man diese fehlende Definition der leeren Summe nachliefert:

```
double sum() {
  return 0.0;
}
```

Eine zweite Möglichkeit ist, die Summe mit einem Argument so neu zu definieren, dass kein rekursiver Aufruf mehr stattfindet und direkt der Wert des gegebenen Arguments zurückgegeben wird:

```
template <typename Number>
double sum(Number first) {
  return static_cast<double>(first);
}
```