Metaprogramming und Concepts

☑ **Lösung zu Aufgabe 8.1** Die erhaltenen Ausgaben sind der Reihe nach `A C B`. Es ist zwar so, dass alle drei Typen `X1`, `X2` und `X3` jeweils alle drei Symbole `a`, `b` und `c` definieren. Allerdings hat immer nur genau eines dieser definierten Symbole die von den Konzepten `A`, `B` und `C` erwartete Form. Konzept `A` erwartet eine *Member-Variable* `a`, Konzept `B` erwartet einen *Typ-Namen* `b` und Konzept `C` erwartet eine *Member-Funktion* `c` ohne Argumente. Deshalb existiert für jeden der drei Aufrufe der Funktion `f` ein eindeutiger Overload, der vom Compiler ausgewählt wird.

☑ **Lösung zu Aufgabe 8.2** Der gezeigte Code in der Aufgabenstellung enthält die folgenden vier grundlegenden Fehler.

- Die beiden Ausdrücke `Binomial<N-1, K-1>` und `Binomial<N-1, K>` repräsentieren jeweils einen Typ und nicht den berechneten Wert des Binomialkoeffizienten. Um auf den Wert zuzugreifen, mit dem dann weitergerechnet werden kann, müssen die Ausdrücke `Binomial<N-1, K-1>::value` sowie `Binomial<N-1, K>::value` verwendet werden.
- Damit aber mit Hilfe der Syntax `::` auf den berechneten Wert zugegriffen werden kann, muss es sich bei `value` um eine statische Member-Variable handeln. Es muss deshalb vor jeder Definition das Schlüsselwort `static` noch hinzugefügt werden.
- Statische Member-Variablen, die innerhalb der Definition einer Klasse gleich initialisiert werden, müssen jedoch als `const` oder noch besser als `constexpr` markiert sein. Da es sich bei den berechneten Member-Variablen um Kompilierzeit-Konstanten handelt, ist es aber sowieso eine gute Idee, diese entsprechend zu markieren.
- Der letzte Fehler ist subtil. Er macht sich nur bei der Instanziierung von `Binomial<0, 0>` bemerkbar; das heisst, wenn die konkreten Werte `N=0` und `K=0` eingesetzt werden. Dieser Spezialfall wird von beiden Spezialisierungen des Templates abgedeckt. Weil jedeoch keine dieser beiden Spezialisierungen spezifischer als die andere ist, hat der Compiler keinen Grund, die eine der anderen vorzuziehen. Hierbei spielt es auch überhaupt keine Rolle, dass die Member-Variable `value` für beide Spezialisierungen den gleichen Wert `1` zugewiesen bekäme.

  Dieser Fehler lässt sich beheben, indem man für den Spezialfall `N=0` und `K=0` noch eine weitere Spezialisierung hinzufügt, die noch spezifischer ist als die beiden bestehenden Spezialisierungen:

```
template <>
struct Binomial<0, 0> {
  static constexpr int value = 1;
};
```

☑ **Lösung zu Aufgabe 8.3** Die folgende Definition des Konzepts hat die gewünschten Eigenschaften und produziert genau die verlangten Ausgaben auf den letzten vier Zeilen des in der Aufgabenstellung gegebenen Codes. In der Implementation dieses Konzepts wird ausgenutzt, dass die Zuweisung eines Zeigers vom Typ `T1*` an einen Zeiger vom Typ `T2*` möglich ist, wann immer `T1` in der Vererbungshierarchie unter `T2` steht.

```
template <typename T1, typename T2>
concept IsSubtypeOf = requires(T1* x1, T2* x2) {
  x2 = x1; // Forderung: Zuweisung von T1* an T2* ist moeglich
};
```

Wenn man das Konzept auf gleiche Weise aber mit *Werten* vom Typ `T1` und `T2` anstelle von *Zeigern* mit Typ `T1*` und `T2*` implementiert hätte, dann hätte es auf der allerletzten Zeile nicht die gewünschte Ausgabe produziert. Weil im gegebenen Code bereits eine implizite Umwandlung von `Child` nach `Other` definiert ist, wäre dann `Child` fälschlicherweise als Untertyp von `Other` erkannt worden.