


Metaprogramming und Concepts

 **Aufgabe 8.1** Welche Ausgabe würden Sie auf den letzten drei Zeilen von folgendem Code erwarten?

```
template <typename T> concept A = requires(T x) { x.a; };
template <typename T> concept B = requires(T x) { typename T::b; };
template <typename T> concept C = requires(T x) { x.c(); };

void f(A auto x) { std::cout << "A" << std::endl; }
void f(B auto x) { std::cout << "B" << std::endl; }
void f(C auto x) { std::cout << "C" << std::endl; }

struct X1 { int a; using c = int; int b(); }; f(X1 {});
struct X2 { int b; using a = int; int c(); }; f(X2 {});
struct X3 { int c; using b = int; int a(); }; f(X3 {});
```


 **Aufgabe 8.2** Für zwei natürliche Zahlen $n, k \in \mathbb{N}$ drückt der sogenannte Binomialkoeffizient $\binom{n}{k}$ die Anzahl Möglichkeiten aus, wie man k von n Gegenständen auswählen kann. Über das Pascalsche Dreieck erhält man die rekursive Definition $\binom{n}{k} := \binom{n-1}{k-1} + \binom{n-1}{k}$ mit den Spezialfällen $\binom{n}{0} := 1$ und $\binom{n}{n} := 1$.

Ein Programmierer hat versucht, im folgenden Code ein Klassen-Template zu implementieren, das die Berechnung solcher Binomialkoeffizienten zur Kompilierzeit ausführen kann. Beispielsweise soll der Ausdruck `Binomial<5, 2>::value` den Wert 10 (weil $\binom{5}{2} = 10$) repräsentieren, der als konstanter Ausdruck im Programm-Code verwendet werden kann. Leider haben sich einige Fehler eingeschlichen. Versuchen Sie, alle Fehler zu finden, ohne dabei einen Compiler zu Rate zu ziehen.

```
template <int N, int K>
struct Binomial {
    int value = Binomial<N-1, K-1> + Binomial<N-1, K>;
};

template <int N>
struct Binomial<N, 0> {
    int value = 1;
};

template <int N>
struct Binomial<N, N> {
    int value = 1;
};
```

 **Aufgabe 8.3** Implementieren Sie von Grund auf ein Konzept `IsSubtypeOf<T1, T2>` (ohne Zuhilfenahme bereits bestehender Konzepte aus `std`), das überprüft, ob `T1` im Sinne der Vererbungshierarchie ein Untertyp von `T2` ist. Es ist für diese Aufgabe hilfreich, sich zu überlegen, welche Arten von Zeiger-Zuweisungen in C++ erlaubt sind. Folgender Code soll die angegebenen Ausgaben produzieren.

```
template <typename T1, typename T2>
void f(T1 t1, T2 t2) { std::cout << "Kein Untertyp" << std::endl; }
template <typename T1, typename T2> requires IsSubtypeOf<T1, T2>
void f(T1 t1, T2 t2) { std::cout << "Untertyp" << std::endl; }

struct Parent {};
struct Child : public Parent {};
struct Other { Other() {}; Other(Child c) {} };

f(Child {}, Parent{}); // Ausgabe: Untertyp
f(Parent{}, Child {}); // Ausgabe: Kein Untertyp
f(Other {}, Other {}); // Ausgabe: Untertyp
f(Child {}, Other {}); // Ausgabe: Kein Untertyp
```