1 Generic Maximum

In den Folien finden Sie die Definition des Funktions-Templates `min()`. Implementieren Sie eine verallgemeinerte Version von `max()`, die überall (Return-Typ und alle Parameter) verschiedene Typen annehmen kann. Für das Testen der Funktionalität können Sie folgenden Code verwenden:

```
1   const int i = 3;
2   const double d = 3.14;
3
4   std::cout << max(2, i) << std::endl;
5   std::cout << max<double>(i, d) << std::endl;
6   std::cout << max(i, d) << std::endl;
```

1.1 Aufgabe

Implementieren Sie die Funktion auf unterschiedliche Arten (mittels Überladung) und analysieren Sie, wann welche Variante aufgerufen wird.

1.2 Lösung

```
1  template<typename T>
2  T max(T x, T y) {
3    std::cout << "T max(T x, T y) = ";
4    return (x > y) ? x : y;
5  }
6
7  template<typename R, typename T, typename S>
8  R max(T x, S y) {
9    std::cout << "R max(T x, S y) = ";
10   return (x > y) ? x : y;
11 }
12
13 template<typename T, typename S>
14 auto max(T x, S y) {
15   std::cout << "auto max(T x, S y) = ";
16   return (x > y) ? x : y;
17 }
```

Output:

```
1  T max(T x, T y) = 3
2  R max(T x, S y) = 3.14
3  auto max(T x, S y) = 3.14
```

2 Container-Casting

Implementieren Sie ein Klassen-Template `Container`, welches ein einziges Element seines Template-Parameters speichert. Folgender Code soll dann möglich sein:

```
Container<int> a(1);
std::cout << a << std::endl;                    // 1
Container<double> b(3.14);
std::cout << b << std::endl;                    // 3.14
a = b;
std::cout << a << std::endl;                    // 3
std::cout << Container<int>(8.91) << std::endl;   // 8
```

2.1 Aufgabe

Implementieren Sie die Klasse. Zudem sollen Sie einen Konvertierungs-Konstruktor und einen Copy-Assignment-Operator implementieren, welche beide ebenfalls Templates sind.

2.2 Lösung

```
template<typename T>
class Container {
  template<typename K>
  friend class Container;

 public:
  template<typename K>
  explicit Container(const K& init) : data_(static_cast<T>(init)) {
  }

  template<typename K>
  Container& operator=(const Container<K>& rhs) {
    data_ = static_cast<T>(rhs.data_);
    return *this;
  }

  friend std::ostream& operator<<(std::ostream& os, const Container& c) {
    os << c.data_;
    return os;
  }

 private:
  T data_;
};
```

3 Any-Creator

In dieser Aufgabe sollen Sie ein Funktions-Template schreiben, welches einen `std::vector` von `std::any` Objekten erstellt. Folgender Code soll damit möglich sein.

```
auto container = CreateAnyVector("Hello", 3.14, 'A', true, 42);
for (const auto& any : container) {
  std::cout << any.type().name() << std::endl;
}
```

3.1 Aufgabe

Implementieren Sie die Funktion `CreateAnyVector` mittels Variadic Templates. Damit Sie die Pack-Expansion für einzelne Funktionsaufrufe machen können, bietet sich Rekursion oder eine Fold-Expression an.

3.2 Lösung

```
#include <any>
#include <iostream>
#include <utility>
#include <vector>

template<typename... Ts>
auto CreateAnyVector(const Ts&... args) {
  std::vector<std::any> container;
  (..., container.emplace_back(std::forward<const Ts&>(args)));
  return container;
}
```