Template Deklarationen

⟨Templ'Dekl'⟩ ⟶ "`template`" "`<`" [⟨Templ'Param'⟩ {"`,`" ⟨Templ'Param'⟩}] "`>`" ⟨Dekl'⟩
⟨Templ'Param'⟩ ⟶ ⟨Templ'Typ'Param⟩ | ⟨Param'Dekl'⟩
⟨Templ'Typ'Param'⟩ ⟶ ("`class`" | "`typename`") [⟨Ident⟩] ["`=`" ⟨Typ⟩]
| "`template`" "`<`" [⟨Templ'Param'⟩ {"`,`" ⟨Templ'Param'⟩}] "`>`" "`class`" [⟨Ident⟩] ["`=`" ⟨Id'Expr'⟩]

- `template` deklariert parametrisierte, nicht-abgeschlossene Familie von Objekten (Funktionen, Typaliase, Variablen, Klassen) ⟨Dekl'⟩
- Parameter der Familie entweder wie Funktionsparameter (Werte) ⟨Param'Dekl'⟩ oder Typparameter ⟨Templ'Typ'Param'⟩
- Typparameter dürfen wiederum als templates mit Parametern gefordert werden
- Ansatz: wann immer Objekt aus Familie verwendet wird, setze passende Parameter ein in neue Kopie der Vorlage (engl. template)

Einschub: Arrays in STL (`array`)

- Datentyp `array<`T`, `n`>` für beliebigen Typ T und Ganzzahl n
- Dünner wrapper um C-Array der Form T`[`n`]`
- Vorteil: Methoden, Operatoren, Iteratoren analog zu `vector`

Verwendung array

```
array<double, 3> a1{3, 2, 1};

sort(a1.begin(), a1.end());

cout << "a1[0.."
     << a1.size() - 1
     << "]:";
for (const double& x: a1)
  cout << " " << x;
cout << endl;
```

```
a1[0..2]: 1 2 3
```

Beispiel: Funktionstemplate p-Norm

array_pnorm.cpp

```cpp
#include <iostream>
#include <array>
#include <cstddef>
#include <cmath>

using namespace std;

template<class T = double, unsigned int p = 2, size_t dim>
  T pNorm(const array<T, dim>& v) {
    T x = 0;
    for (const T& y: v)
      x += pow(y, p);
    return pow(x, static_cast<T>(1) / p);
}

int main() {
  array<double, 3> a1{1, 2, 3};
  cout << pNorm<double, 2, 3>(a1) << endl;

  array<double, 5> a2{5, 4, 3, 2, 1};
  cout << pNorm<double, 3>(a2) << endl;
}
```

```
3.74166
6.0822
```

(Explizite) Instanziierung von Templates

- Templates spezifizieren Familie von Objekten
- Bevor konkretes Objekt aus der Familie verwendet werden kann, muss es instanziiert werden — Argumente werden eingesetzt in Deklaration
- Explizite Instanziierung möglich (siehe Beispiel)
- Gewöhnlicher: implizite Instanziierung bei erster Verwendung in jeder Übersetzungseinheit

instantiate_array_demo.cpp

```cpp
#include <array>

using namespace std;

template class std::array<double, 3>;

int main() {
  return 0;
}
```

Template Spezialisierung

⟨Templ'Spec'⟩ ⟶ "`template`" "<" [⟨Templ'Param'⟩ {"`,`" ⟨Templ'Param'⟩}] ">" ⟨Dekl'⟩

- Mit ⟨Dekl'⟩ Definition für Objekt aus bereits bekanntem Template
- Zweck: effizientere Implementierung
- Wenn Parameter vorhanden ⟶ *partielle* Spezialisierung
- Partielle Spezialisierung nur für Klassen und Variablen

Beispiel: spezialisierte 2-Norm

array_2norm.cpp

```cpp
#include <iostream>
#include <array>
#include <cstddef>
#include <cmath>

using namespace std;

template<class T = double, unsigned int p = 2, size_t dim>
  T pNorm(const array<T, dim>& v) {
    T x = 0;
    for (const T& y: v)
      x += pow(y, p);
    return pow(x, static_cast<T>(1) / p);
}

template<>
  double pNorm<double, 2, 3>(const array<double, 3>& v) {
    return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
}

int main() {
  array<double, 3> a1{1, 2, 3};
  cout << pNorm<double, 2, 3>(a1) << endl;

  array<double, 5> a2{5, 4, 3, 2, 1};
  cout << pNorm<double, 3>(a2) << endl;
}
```

```
3.74166
6.0822
```

Substitution Failure is Not An Error (SFINAE)

- Situation: template mit partieller Spezialisierung, Einsetzen von Argumenten in Spezialisierung scheitert (Klassenkomponenten fehlen, eine Typkonvertierung ist nicht möglich, …)
- *Kein* Kompilierfehler
- Stattdessen: Streichen dieser Spezialisierung als Kandidat, i.A. Wahl einer Anderen
- Gelegentlich: explizites Ausnutzen hiervon um Spezialisierungen einzuschränken auf nur bestimmte Argumente

`std::enable_if`

sfinae_narrow_demo.cpp

```cpp
#include <type_traits>
#include <iostream>
#include <cmath>

using namespace std;

class Approx {
  public:
    int x;

    template<class Integral,
      enable_if_t<is_integral<Integral>::value, bool> = true>
        Approx(Integral i): x(i) {}

    template<class Floating,
      enable_if_t<is_floating_point<Floating>::value, bool> =
     ↪  true>
        Approx(Floating d): x(round(d)) {}
};

int main() {
  Approx a1{3}, a2{5.7};

  cout << a1.x << " " << a2.x << endl;
}
```

Implementierung enable_if

```cpp
template<bool B, class T = void>
  class enable_if {};

template<class T>
  class enable_if<true, T> { public: using type = T; };

template<bool B, class T = void>
  using enable_if_t = typename enable_if<B, T>::type;
```

Implementierung is_integral

```cpp
template<class T>
  class is_integral {};

template<>
  class is_integral<int> {
    public: static constexpr bool value = true;
  };
```

```
3 6
```

Ausblick: Constraints & Concepts (C++20)

Constraint Angabe eines (hinreichend konstanten) `require`-Ausdrucks vom Typ `bool` bei Definition eines templates mit Schlüsselwort `requires` ⟶ Template nur instanziierbar wenn `require`-Ausdruck `true`

Concept Definition eines benannten `require`-Ausdrucks als template mit Schlüsselwort `concept` ⟶ So definiertes concept verwendbar als `require`-Ausdruck bei folgenden templates

```
template<class T>
  concept Integral = std::is_integral<T>::value;
template<class T>
  concept SignedIntegral = Integral<T> && std::is_signed<T>::value;

template<class T> requires SignedIntegral<T>
  void f(T);
```

Beispiel: Variablentemplate `pi_v`

template_pi_narrow.cpp

```
#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

template<class T>
  constexpr T pi_v
    = static_cast<T>(3.14159265358979323846264338327950288 4L);

int main() {
  cout
    << pi_v<int> << endl
    << setprecision(numeric_limits<float>::digits10 + 1)
    << pi_v<float> << endl
    << setprecision(numeric_limits<double>::digits10 + 1)
    << pi_v<double> << endl
    << setprecision(numeric_limits<long double>::digits10 + 1)
    << pi_v<long double> << endl;
}
```

```
3
3.141593
3.141592653589793
3.141592653589793239
```

Beispiel: Typalias-templates

template_downset.cpp

```cpp
#include <set>
#include <iostream>
#include <functional>

using namespace std;

template<class Key>
  using downset = set<Key, greater<Key>>;

int main() {
  downset<int> s;
  s.insert(3); s.insert(1); s.insert(2);

  cout << "s:";
  for (const int& x: s)
    cout << " " << x;
  cout << endl;
}
```

```
s: 3 2 1
```

template_ptr.cpp

```cpp
#include <iostream>

using namespace std;

template<class T>
  using ptr = T*;

ptr<int> x = nullptr;

int main() {
  int y = 7;
  x = &y;

  cout << *x << endl;
}
```

```
7
```

Beispiel: Polymorphe Klasse `Vektor`

poly_vector.cpp

```cpp
#include <iostream>

using namespace std;

template<class T>
  class Vektor {
    private:
      T* ap;
      int len;

    public:
      Vektor(int n = 0, const T& x = T{}): ap(nullptr), len(n) {
        if (!len) return;
        ap = new T[n];
        for (int i = 0; i < len; i++)
          ap[i] = T{x};
      }
      ~Vektor() { if (ap) delete[] ap; }
      Vektor& operator=(const Vektor& b) = delete;

      class iterator {
        friend class Vektor;

        private:
          Vektor* v;
          int pos;

          iterator(Vektor* v_, int pos_ = 0)
            : v(v_), pos(pos_) {}

        public:
          T& operator*() { return v->ap[pos]; }
          iterator& operator++() {
            if (pos < v->len) pos++;
            return *this;
          }
          bool operator!=(const iterator& other) const {
            return v != other.v || pos != other.pos;
          }
      };

      iterator begin() { return iterator{this}; }
      iterator end() { return iterator{this, len}; }
  };

int main() {
  Vektor<int> v{4};
  cout << "v: ";
  unsigned int count = 0;
  for (Vektor<int>::iterator it = v.begin(); it != v.end();
  ↪  ++it) {
    *it = count++; cout << *it << " ";
  }
  cout << endl;
}
```

```
v: 0 1 2 3
```

Beispiel: Partiell spezialisierte p-Norm

partial_pnorm.cpp

```cpp
#include <iostream>
#include <iomanip>
#include <limits>
#include <array>
#include <vector>
#include <cstddef>
#include <cmath>

using namespace std;

template<class Vec, unsigned int p = 2>
  class PNorm{
    private:
      using X = typename Vec::value_type;
    public:
      X operator()(const Vec& v) {
        X x = 0;
        for (const X& y: v)
          x += pow(y, p);
        return pow(x, static_cast<X>(1) / p);
      }
  };

template<class Vec>
  class PNorm<Vec, 2>{
    private:
      using X = typename Vec::value_type;
    public:
      X operator()(const Vec& v) {
        X x = 0;
        for (const X& y: v)
          x += y * y;
        return sqrt(x);
      }
  };

int main() {
  vector<double> c{5, 6};
  array<float, 2> a{3, 4};

  cout
    << setprecision(numeric_limits<double>::digits10 + 1)
    << PNorm<vector<double>>{}(c) << endl
    << PNorm<array<float, 2>, 3>{}(a) << endl;
}
```

```
7.810249675906654
4.497941493988037
```