Programmieren in C++

Metaprogramming und Concepts

Inhalt

- Template Meta Programming
- Rekursive Templates
- SFINAE
- Constraints und Concepts
- Konzepte erstellen

Template Meta Programming (TMP)

- TMP für Werte
  - Berechnungen werden bereits zur Kompilationszeit durchgeführt
  - Iteration mittels Rekursion (Turing complete)
- TMP für Typen
  - Ermittlung von passenden Typen

```
template<class LHS, class RHS>
auto plus(LHS l, RHS r) -> decltype(l+r) {
    static_assert(is_arithmetic_v<LHS> && is_arithmetic_v<RHS>,
    "LHS or RHS are not arithmetic");
    return l + r;
}
```

  - Abfrage oder Modifizierung von Eigenschaften generischer Typen (type traits), in C++20: concept
  - statischer Polymorphismus (Vererbung und Templates anstatt dynamischer Bindung)

Rekursive Templates

- Idee
  - aus Templates mit Wert-Parametern und aus spezialisierten Templates können rekursive Templates mit Verankerung (Spezialisierung) erzeugt werden
- Beispiel (Potenzieren zur Kompilationszeit)

```
template<size_t N> struct Dreihoch {
    static constexpr uint64_t Wert = 3*Dreihoch<N - 1>::Wert;
};
template<> struct Dreihoch<0> {
    static constexpr uint64_t Wert = 1;
};
int main() {
    cout << Dreihoch<11>::Wert << endl;
}
```

Verwendung von integral_constant

- Definition

```
template <typename T, T V>
struct integral_constant {
    static constexpr T value = V;
    using value_type = T;
};
```

- Beispiel

```
template<size_t N>
struct Dreihoch : integral_constant<uint64_t, 3*Dreihoch<N - 1>::value> {};
template<>
struct Dreihoch<0> : integral_constant<uint64_t, 1> {};
```

constexpr Funktionen

- Rekursive Templates ohne Type-Parameter können durch constexpr Funktionen ersetzt werden

```
constexpr uint64_t vierhoch(uint8_t n) {
    return (n == 0) ? 1 : 4*vierhoch(n - 1);
}
```

N hoch X

- volle Flexibilität ausgenutzt
  - C++20: Werte-Parameter darf auch floating-point sein

```
template<typename T, T N, size_t X>
struct NhochX {
    static constexpr T Wert = N*NhochX<T, N, X - 1>::Wert;
};
template<typename T, T N>
struct NhochX<T, N, 0> {
    static constexpr T Wert = 1;
};

int main() {
    constexpr auto x = NhochX<double, 0.5, 3>::Wert; // 0.125
}
```

TMP versus constexpr Funktionen

- constexpr Funktionen
  - können andere constexpr Funktionen aufrufen
  - können Variablen verwenden, welches mit constexpr Ausdrücken initialisiert werden müssen
  - dürfen std::vector und std::string verwenden
  - können Bedingungen und Schleifen enthalten
  - sind implizit inline
  - können zur Kompilationszeit oder zur Laufzeit ausgeführt werden

Characteristics	Template Metaprogramming	`constexpr` Functions
Execution time	Compile time	Compile time and runtime
Arguments	Types and values	Values
Programming paradigm	Functional	Imperativ
Modification	No	Yes
Control structure	Recursion	Conditions and loops
Conditional execution	Template specialisation	Conditional statements

SFINAE

- Substitution Failure Is Not An Error (SFINAE)
  - fehlgeschlagene Substitutionen von Template-Argumenten erzeugen keine Compiler-Fehler und führen zu keiner Instanziierung
  - ist das grundlegende Prinzip bei den type traits und concepts
- Beispiel

```
struct Y { using type = int; };

template<typename T>
void func(typename T::type) { cout << "func(T::type)" << endl; }

template<typename T>
void func(T) { cout << "func(T)" << endl; }


func<Y>(10);          // func(T::type), weil Y einen Typ type besitzt
func(10);             // func(T), weil int keinen Typ type besitzt
```

Ausnutzung der Spezialisierung

- Idee
  - in der Spezialisierung wird ein innerer Typ (member type) definiert oder eine statische Konstante anders initialisiert
- Beispiel
  - Überprüfung, ob zwei Template-Parameter gleich sind

```
template<typename T, typename U>
struct is_same : integral_constant<bool, false> {};

template<typename T>
struct is_same<T,T> : integral_constant<bool, true> {};

template<typename T, typename U>
constexpr bool is_same_v = is_same<T, U>::value;
```

Constraints und Concepts (1)

- requires *requires_expression*
  - verschiedenste Arten von Bedingungen werden unterstützt
- concept
  - eine logische Verknüpfung von Bedingungen (constraints)

```
template<class T, class U>
concept Same = is_same_v<T, U> && is_same_v<U, T>;
```

- Anwendung (verschiedene Schreibweisen möglich)

```
template<typename T>
void foo(T* p) {
        cout << "foo" << endl;
}
```

Constraints und Concepts (2)

- requires *requires_expression*
  - verschiedenste Arten von Bedingungen werden unterstützt
- concept
  - eine logische Verknüpfung von Bedingungen (constraints)

```
template<class T, class U>
concept Same = is_same_v<T, U> && is_same_v<U, T>;
```

- Anwendung (verschiedene Schreibweisen möglich)

```
template<typename T> requires Same<T, int>
void foo(T* p) {
        cout << "foo" << endl;
}
```

Constraints und Concepts (3)

- requires *requires_expression*
  - verschiedenste Arten von Bedingungen werden unterstützt
- concept
  - eine logische Verknüpfung von Bedingungen (constraints)

```
template<class T, class U>
concept Same = is_same_v<T, U> && is_same_v<U, T>;
```

- Anwendung (verschiedene Schreibweisen möglich)

```
template<typename T>
void foo(T* p) requires Same<T, int> {
        cout << "foo" << endl;
}
```

Constraints und Concepts (4)

- requires *requires_expression*
  - verschiedenste Arten von Bedingungen werden unterstützt
- concept
  - eine logische Verknüpfung von Bedingungen (constraints)

```
template<class T, class U>
concept Same = is_same_v<T, U> && is_same_v<U, T>;
```

- Anwendung (verschiedene Schreibweisen möglich)

```
template<Same<int> T>
void foo(T* p) {
        cout << "foo" << endl;
}
```

Verwendung von Concepts

- Konzepte sind sinnvoll für Template-Parameter und für nicht definierte Parameter/Rückgabetypen (auto)

```
void funcWithAutoInline(const std::convertible_to<std::string> auto& x) {
    std::string v = x;
}
template <std::convertible_to<std::string> T>
void funcWithTemplateInline(const T& x) {
    std::string v = x;
}
template <typename T> requires std::convertible_to<T, std::string>
void funcWithTemplatePostfix(const T& x) {
    std::string v = x;
}
```

Verknüpfte Bedingungen

- Wenn ein Template-Parameter oder ein auto-Parameter mehrere Anforderungen erfüllen muss, so können diese Anforderungen logisch verknüpft werden.
  - solche Verknüpfungen können schnell schlecht lesbar werden

```
template <typename T>
requires std::integral<T> ||
    (std::invocable<T> && std::integral<typename std::invoke_result<T>::type>)
void function3(const T& x) {
    if constexpr (std::invocable<T>) {
        std::cout << "Result of call is " << x() << "\n";
    } else {
        std::cout << "Value is " << x << "\n";
    }
}
```

  - bei verknüpften Bedingungen empfiehlt sich die Definition von eigenen Konzepten

Eigene Konzepte definieren

- Syntax

  template <*typename T*>
  concept *Name* = *constraint_expression*;

  - *constraint_expression* kann folgende Teile beinhalten
    - logische constexpr Ausdrücke
    - andere Konzepte
    - requires Blöcke

- Beispiel

```
template <typename T>
concept MaybeInvokableIntegral =
    std::integral<T> ||
    (std::invocable<T> && std::integral<typename std::invoke_result<T>::type>);
```

Verwendung von requires (1)

- Ausführbarkeit eines Ausdrucks sicherstellen

```
template <typename T>
concept Addable = requires (T a, T b) {
    a + b;
};
```

- Sicherstellen, dass *T* einen inneren Typ ElementType besitzt

```
template <typename T>
concept TypeTest = requires {
    typename T::ElementType;
};
```

- Sicherstellen, dass *T* eine gültige Template Substitution ist

```
template <typename T, template<typename> class S>
concept TemplateTest = requires {
    typename S<T>;
};
```

Verwendung von requires (2)

- Rückgabetyp eines Ausdrucks überprüfen

```
template <typename T>
concept InvokeIntegral = requires (T a) {
    { a() } -> std::integral;
};
```

- Sicherstellen, dass ein Ausdruck keine Exception werfen darf

```
template <typename T>
concept CantThrow = requires (T a, T b) {
    { a = b } noexcept;
};
```

- Verschachtelung von requires Blöcken

```
template <typename T>
concept Nested = requires (T a) {
    requires sizeof(a) >= 4;
    requires std::integral<T>;
};
```

Welche Spezialisierung wird gewählt?

- Verfahren
  - unter allen passenden Spezialisierungen wird diejenige mit der meist spezifischen Bedingung gewählt
    - die Bedingungen werden in eine Normalform überführt, wobei jeder Constraint ein einzelnes Atom der Normalform darstellt
    - eine Bedingung mit mehr Atomen ist spezifischer als eine andere
  - wenn es mehre meistspezifische Bedingung gibt, dann gibt der Compiler einen Fehler aus (ambiguous)

```
template <typename T>
concept HasX = requires (T v) {
    v.x;
};
template <typename T>
concept HasXY = requires (T v) {
    v.x;
    v.y;
};
```

```
template <typename T>
concept Is2D = HasX<T> && requires (T v) {
    v.y;
};
```

Welche Spezialisierung wird gewählt?

```
struct X {
    int x;
};
struct Y {
   int x;
   int y;
};

void function(HasX auto x) {}
void function(HasXY auto x) {}

int main() {
   function(X{}); // OK, only one viable candidate
   function(Y{}); // fails, no winner
}
```

Welche Spezialisierung wird gewählt?

```
struct X {
    int x;
};
struct Y {
   int x;
   int y;
};

void function(HasX auto x) {}
void function(Is2D auto x) {}

int main() {
   function(X{}); // OK, only one viable candidate
   function(Y{}); // OK, Is2D is more specific
}
```