Metaprogrammierung in C/C++

Vortrag im WP Deeply Embedded

Philipp Bandow
Jakob Kasicz
Orhan Aykac

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Inhalt

1. Metaprogrammierung mit dem Präprozessor
2. Templates
   - Einleitung
   - Parameter und Typen
   - Abhängige Namen
   - Benutzung
   - Explizite Spezialisierung
   - Partielle Spezialisierung
   - Beispiele
3. Code Bloat

HAW HAMBURG

Metaprogrammierung mit dem Präprozessor

Metaprogrammierung mit dem Präprozessor

- Sprachelemente stammen aus C
- Verwendbar auch in C++
- Hauptsächlicher Einsatzzweck: Wiederholung oder Manipulation von bereits vorhandenem Code

HAW HAMBURG

Präprozessor Direktiven (C-Makros)

- Werden mit der **#define** Direktive erstellt
- Quellcode wird vom Präprozessor bearbeitet und alle mit **#define** erstellten Makros werden ersetzt
- Meistens werden Makros für Konstanten verwendet (eher in C als in C++), sie können aber auch verschachtelt werden

HAW HAMBURG

Einfache Makros (der Vollständigkeit halber)

```
1 #define NUM 42
2 #define STRING "ein␣String\n"
3 ...
4 cout << STRING << endl;
```

Mehrzeilige Makros

- Das \-Zeichen erlaubt es Makros, welche normalerweise "Einzeiler" sind, trotzdem in mehreren Zeilen zu schreiben
- Man sollte allerdings beachten, dass der Präprozessor die Makros trotzdem als "Einzeiler" verarbeitet

```
1 #define STRINGOUT cout << "Stringdingdong" \
2 endl;
```

Parametersubstitution

- Makros dürfen ähnlich wie Funktionen parametriert werden
- Man kann an dem Beispiel erkennen das es **KEINE** Typüberprüfung gibt
- Es darf fast jedes "Wort" als Parameter übergeben werden, es dürfen allerdings keine "," vorkommen, da diese die Parameter trennen!

HAW HAMBURG

Parametersubstitution

- Makros dürfen ähnlich wie Funktionen parametriert werden
- Man kann an dem Beispiel erkennen das es **KEINE** Typüberprüfung gibt
- Es darf fast jedes "Wort" als Parameter übergeben werden, es dürfen allerdings keine "," vorkommen, da diese die Parameter trennen!

HAW HAMBURG

Parametersubstitution

- Makros dürfen ähnlich wie Funktionen parametriert werden
- Man kann an dem Beispiel erkennen das es **KEINE** Typüberprüfung gibt
- Es darf fast jedes "Wort" als Parameter übergeben werden, es dürfen allerdings keine "," vorkommen, da diese die Parameter trennen!

HAW HAMBURG

Klammern von Parametern

- Wie erwähnt ist eine Markodefinition eine reine Substitution, von daher ist gerade bei mathematischen Ausdrücken Vorsicht geboten und es ist gängige Praxis "viel zu Klammern"

```
1 #define SQRE1(x) ((x) * (x))
2 #define SQRE2(x) (x * x)
```

```
1 int a= SQRE1(5+1); // a= ((5+1)*(5+1))= 36
2 int b= SQRE2(5+1); // b= (5+1*5+1)= 11
```

Klammern von Parametern

- Wie erwähnt ist eine Markodefinition eine reine Substitution, von daher ist gerade bei mathematischen Ausdrücken Vorsicht geboten und es ist gängige Praxis "viel zu Klammern"

```
1 #define SQRE1(x) ((x) * (x))
2 #define SQRE2(x) (x * x)
```

```
1 int a= SQRE1(5+1); // a= ((5+1)*(5+1))= 36
2 int b= SQRE2(5+1); // b= (5+1*5+1)= 11
```

Klammern von Parametern

- Wie erwähnt ist eine Markodefinition eine reine Substitution, von daher ist gerade bei mathematischen Ausdrücken Vorsicht geboten und es ist gängige Praxis "viel zu Klammern"

```
1 #define SQRE1(x) ((x) * (x))
2 #define SQRE2(x) (x * x)
```

```
1 int a= SQRE1(5+1); // a= ((5+1)*(5+1))= 36
2 int b= SQRE2(5+1); // b= (5+1*5+1)= 11
```

Variadic Makros

- Variadic Makros sind Makros mit einer Parameterliste, diese ist Variabel und wird mit (...) angegeben. Die Übergabe findet mit dem vordefinierten **__VA_ARGS__** Makro statt

```
1 #define ERROR(...) \
2   printf("file: %s, line: %d, error: ", ) \
3   __FILE__, __LINE__, __VA_ARGS__)
```

```
1 int main(void) {
2   ERROR("Dies ist Fehler Nr %d", 100);
3 }
```

Variadic Makros

- Variadic Makros sind Makros mit einer Parameterliste, diese ist Variabel und wird mit (...) angegeben. Die Übergabe findet mit dem vordefinierten __**VA_ARGS**__ Makro statt

```
1 #define ERROR(...) \
2   printf("file: %s, line: %d, error: ", ) \
3   __FILE__, __LINE__, __VA_ARGS__)
```

```
1 int main(void) {
2   ERROR("Dies ist Fehler Nr %d", 100);
3 }
```

Variadic Makros

- Variadic Makros sind Makros mit einer Parameterliste, diese ist Variabel und wird mit (...) angegeben. Die Übergabe findet mit dem vordefinierten __**VA_ARGS**__ Makro statt

```
1 #define ERROR(...) \
2   printf("file: %s, line: %d, error: ", ) \
3   __FILE__, __LINE__, __VA_ARGS__)
```

```
1 int main(void) {
2   ERROR("Dies ist Fehler Nr %d", 100);
3 }
```

Der #-Operatror

- Wandelt einen Makroparameter in einen String um
- Darf nur innerhalb eines Makros verwendet werden
- Es ist jedoch nicht möglich den #-Operator auf einen Token anzuwenden

```
1 #define TOSTRING(a) #a
2 #define SOGEHTESNICHT #fehler
3 ...
4 cout << TOSTRING(test string) << endl //
     korrekt
5 cout << SOGEHTESNICHT << endl // fehler
```

Der ##-Operator

- Operator zu Konkatenation von 2 Token
- Darf ebenso nur in Makros verwendet werden
- Macht die "Horizontale Iteration" möglich. (zur Horizontalen Iteration später mehr)

Der ##-Operator

- Operator zu Konkatenation von 2 Token
- Darf ebenso nur in Makros verwendet werden
- Macht die "Horizontale Iteration" möglich. (zur Horizontalen Iteration später mehr)

Der ##-Operator

- Operator zu Konkatenation von 2 Token
- Darf ebenso nur in Makros verwendet werden
- Macht die "Horizontale Iteration" möglich. (zur Horizontalen Iteration später mehr)

HAW HAMBURG

Verschachtelte Makros

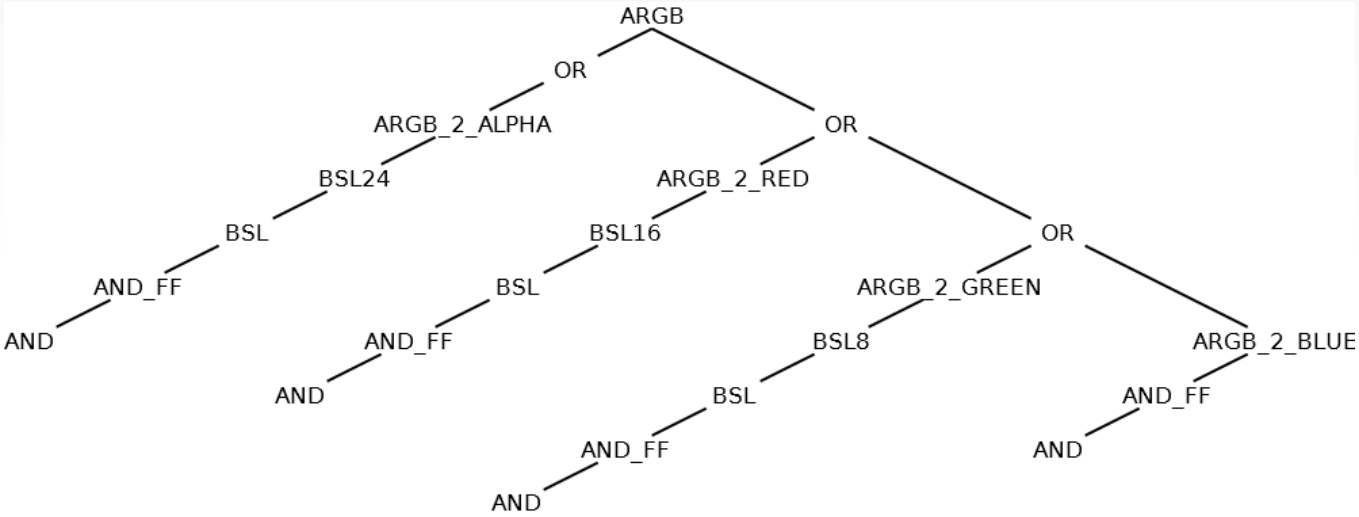
- Makros lassen sich beliebig verschachteln
- Es muss darauf geachtet werden, dass nur von "oben nach unten" verschachtelt werden kann

HAW HAMBURG

Verschachtelte Makros

- Makros lassen sich beliebig verschachteln
- Es muss darauf geachtet werden, dass nur von "oben nach unten" verschachtelt werden kann

Baumstruktur des Makros



Baumstruktur

Primitive horizontale Iteration

- Dient dazu bestimmten Code zu expandieren
- Ist ein recht unflexibles, dafür simples Konzept
- Die gesamte Expansion geschieht einzeilig, daher der der Begriff "Horizontale Expansion"

HAW HAMBURG

Primitive horizontale Iteration

- Dient dazu bestimmten Code zu expandieren
- Ist ein recht unflexibles, dafür simples Konzept
- Die gesamte Expansion geschieht einzeilig, daher der der Begriff "Horizontale Expansion"

HAW HAMBURG

Primitive horizontale Iteration

- Dient dazu bestimmten Code zu expandieren
- Ist ein recht unflexibles, dafür simples Konzept
- Die gesamte Expansion geschieht einzeilig, daher der der Begriff "Horizontale Expansion"

HAW HAMBURG

Horizontale Iteration

- Bringt die einfache horizontale Iteration auf ein höheres Level der Allgemeingültigkeit
- Die Idee ist ein Makro an einen allgemeinen Iterationsprozess zu bergeben
- Strebt nach dem Konzept der Wiederverwendbarkeit von Code

HAW HAMBURG

Horizontale Iteration

- Bringt die einfache horizontale Iteration auf ein höheres Level der Allgemeingültigkeit
- Die Idee ist ein Makro an einen allgemeinen Iterationsprozess zu bergeben
- Strebt nach dem Konzept der Wiederverwendbarkeit von Code

HAW HAMBURG

Horizontale Iteration

- Bringt die einfache horizontale Iteration auf ein höheres Level der Allgemeingültigkeit
- Die Idee ist ein Makro an einen allgemeinen Iterationsprozess zu bergeben
- Strebt nach dem Konzept der Wiederverwendbarkeit von Code

HAW HAMBURG

Lokale Iteration

- Ist ein Konzept um die "enzeilige" Expansion der vorherigen Kozepte zu umgehen
- Das Prinzip beruht auf der Auslagerung der Iterationsschritte, welche mit "**#if #endif**" definiert sind und auf unterschiedlichen Zeilen liegen

HAW HAMBURG

Lokale Iteration

- Ist ein Konzept um die "enzeilige" Expansion der vorherigen Kozepte zu umgehen
- Das Prinzip beruht auf der Auslagerung der Iterationsschritte, welche mit "**#if #endif**" definiert sind und auf unterschiedlichen Zeilen liegen

Datei Iteration & Selbst Iteration

- Die Datei Iteration unterscheidet sich zur lokalen Iteration ansich nur dadurch, dass kein Makro expandiert wird, sondern in jedem Iterationsschritt eine Datei inkludiert wird
- Die Selbst Iteration lässt sich als Variante der Datei Iteration verstehen. Dabei befindet sich der Code nicht in einer zusätzlichen Datei, sondern innerhalb der Datei, in der die Iteration initialisiert wird.

HAW HAMBURG

Datei Iteration & Selbst Iteration

- Die Datei Iteration unterscheidet sich zur lokalen Iteration ansich nur dadurch, dass kein Makro expandiert wird, sondern in jedem Iterationsschritt eine Datei inkludiert wird
- Die Selbst Iteration lässt sich als Variante der Datei Iteration verstehen. Dabei befindet sich der Code nicht in einer zusätzlichen Datei, sondern innerhalb der Datei, in der die Iteration initialisiert wird.

HAW HAMBURG

Inhalt

1. Metaprogrammierung mit dem Präprozessor
2. Templates
   - Einleitung
   - Parameter und Typen
   - Abhängige Namen
   - Benutzung
   - Explizite Spezialisierung
   - Partielle Spezialisierung
   - Beispiele
3. Code Bloat

HAW HAMBURG

Template Einleitung

- Ein Template ist eine Mustervorlage
- Semantisch gleichartige Funktionen/Klassen mit Verschiedenen Parametertypen

- Beispiel: Funktion, die einen int Parameter ausgibt

```
1 void printData(int value) {
2   std::cout << "The value is: " << value <<
        std::endl;
3 }
```

- Beispiel: Funktion, die einen double Parameter ausgibt

```
1 void printData(double value) {
2   std::cout << "The value is: " << value <<
        std::endl;
3 }
```

- Lösung: Template

```
1 template<typename T>
2 void printData(T value) {
3     std::cout << "The value is: " << value <<
          std::endl;
4 }
```

Beispiel

```
1 int i = 3;
2 double d = 4.75;
3 std::string s("Hallo");
4 bool b = false;
5 printData(i); //The value is: 3
6 printData(d); //The value is: 4.75
7 printData(s); //The value is: Hallo
8 printData(b); //The value is: false
```

Parameter und Typen

Definition

- Beginnt mit dem Schlüsselwort **template**, hiernach folgt eine Liste von Template Parametern
- Darauf folgt eine Funktions- oder Klassen-Definition

Parameter

- Typen
- Werte
- Template

```
1 template<typename T>
2 function(...){...}
```

Template Type Parameter

- Templates, die sich auf einen Type beziehen

```
1 template<typename T1>
2 class MyClass{...};
```

```
1 template<class T1>
2 class MyClass{...};
```

HAW HAMBURG

Template Type Parameter

```
1 template<typename T>
2 void func(T value) {
3    const T& ref = value;
4    T* p = new T;
5    T temp(23);
6 }
```

Generierter Code für func<int>:

```
1 void func(int value) {
2   const int& ref = value;
3   int* p = new int;
4   int temp(23);
5 }
```

Template Non-Type Parameter

- Parameter sind compile-time constants
- Syntax ähnlich einer Variablen-Deklaration

Template Non-Type Parameter

```
1 template<int i>
2 class A{...};
```

```
1 template<double* dp>
2 class B{...};
```

```
1 template<void(*func)(int)>
2 void c(){...}
```

Template Non-Type Parameter

```
1 A<3> a3;
```

```
1 A<sizeof(std::string)> as;
```

```
1 double d;
2 B<&d> bpd;
3 B<NULL> bn;
```

Template Non-Type Parameter

```
1  void myfunc(int i);
2  struct MyClass {
3    static void staticFunc(int i);
4  };
```

```
1  int main(void) {
2    c<&myfunc>();
3    c<&MyClass::staticFunc>();
4  }
```

Template Template Parameter

- Ermöglicht einem Template ein anderes Template als Parameter hinzuzufügen
- Eine Klasse, die mehrere collections enthält, bietet dem Nutzer die Möglichkeit, den Type der collection zu bestimmen

HAW HAMBURG

Template Template Parameter

```
1 template<template<typename T> class
      ContainerType>
2 class MyClass {
3   ContainerType<int> intContainer;
4   ContainerType<std::string> stringContainer
        ;
5     ...
6 };
```

```
1 MyClass<vector> v;
2 MyClass<list> l;
```

Default Template Parameter

- Genau wie Funktionen können auch Template default Parameter haben
- Wenn ein Template einen default Parameter hat, müssen alle nachfolgenden Parameter auch default Werte haben
- Bei der Referenzierung können die Parameter weggelassen werden

Default Template Parameter

```
1 template<typename T1, typename T2=int, int i
      = 23>
2 class MyClass {};
```

```
1 MyClass<double, std::string, 46> mc1; //
    Alle Parameter spezifiziert
2 MyClass<std::string, double> mc2; // i
    weggelasssen
```

Default Template Parameter

```
1 template<typename T1, typename T2=int, int i
      = 23>
2 class MyClass {};
```

```
1 MyClass<std::string, double, 23> mc3; //
    Normal
2 MyClass<int> mc4; // alle default
3 MyClass<int, int, 0> mc5; // T2 muss
    spezifiert werden, wenn wir i spezifieren
```

HAW HAMBURG

Abhängige Namen

- Abhängig Namen sind jegliche Namen in der Template Definition, die von einem Template Parameter abhängen
- Werden nur bei der Initialisierung aufgelöst

```
1 struct X {
2   int x;
3   typedef double Z;
4 };
5 struct Y {
6   typedef int x;
7   double Z;
8 };
```

```
1  template<typename T>
2  struct ZZ {
3    T::Z z1; // Falsch
4    typename T::Z z2; // OK
5
6    void func(T& t) {
7      t.x = 4; // OK, referenziert ein Objekt
8    }
9
10   typedef typename std::vector<T>::iterator
       VecIt; // OK
11 };
```

```
1  int main() {
2      X x;
3      Y y;
4
5      ZZ<X> zzx; // OK, X::Z ist ein Typ
6      ZZ<Y> zzy; // Falsch, Y::Z ist ein Objekt
7
8      zzx.func(x); // OK, X::x ist ein Objekt
9      zzy.func(y); // Falsch, Y::x ist ein Typ
10 }
```

HAW HAMBURG

Benutzung von Templates

```
1 template<typename T, unsigned i>
2 struct FixedArray {
3   T data[i];
4 };
```

```
1 FixedArray<int, 3> a; // Array vom Typ int
    mit 3 Elementen
2 FixedArray<int, 1+6/3> // Array vom Typ int
    mit 3 Elementen
```

```
1 template<typename T, unsigned i>
2 struct FixedArray {
3   T data[i];
4 };
```

```
1 FixedArray<int, 3> a; // Array vom Typ int
    mit 3 Elementen
2 FixedArray<int, 1+6/3> // Array vom Typ int
    mit 3 Elementen
```

```
1 template<template<typename T, typename
     Allocator> class Container>
2 struct ContainerPair {
3   Container<int, std::allocator<int> >
       intContainer;
4   Container<std::string, std::allocator<std
       ::string> > stringContainer;
5 };
```

```
1 ContainerPair<std::deque> deqCont; // zwei
     std::deques
2 ContainerPair<std::vector> vecCont; // zwei
     std::vectors
```

```
1 template<template<typename T, typename
    Allocator> class Container>
2 struct ContainerPair {
3   Container<int, std::allocator<int> >
      intContainer;
4   Container<std::string, std::allocator<std
      ::string> > stringContainer;
5 };
```

```
1 ContainerPair<std::deque> deqCont; // zwei
    std::deques
2 ContainerPair<std::vector> vecCont; // zwei
    std::vectors
```

Spezifikation der Template Parameter

```
1 template<typename T>
2 void func() {...}
```

```
1 int main(void) {
2   func<int>();
3   func<double>();
4 }
```

Spezifikation der Template Parameter

```
1 template<typename T>
2 void func() {...}
```

```
1 int main(void) {
2   func<int>();
3   func<double>();
4 }
```

Automatische Template Argument Erkennung

```
1  template<typename T>
2  void func(T val) {}
3
4  template<typename T, typename U>
5  void func2(U val) {
6    return T(val);
7  }
```

Automatische Template Argument Erkennung

```
1  int main(void) {
2    // T=int
3    func(3);
4    // T=double
5    func(3.5)
6
7    // T=int, U=double
8    func2<int>(3.5);
9    // T=std::vector<std::string>, U=int
10   func2<std::vector<std::string> >(5);
11   // T und U spezifiziert; T=std::vector<std
         ::string>, U=int
12   func2<std::vector<std::string>, int>(5.7)
         ;
13 }
```

Explizite Spezialisierung

- Templates können nicht einfach überladen werden
- Problematisch bei allgemein gehaltenen Templates

```
1 template<typename T>
2 void func(T value) {
3   value.add(3);
4 }
```

```
1 class T1 {
2 public:
3   void add(int i, double d = 0.0);
4 }
5 class T2 {
6 public:
7   std::string add(double d);
8 }
```

```
1 class T3 {
2 public:
3   void Add(int i);
4 }
```

Template func nicht nutzbar für T3

Lösung: Explizite Spezialisierung für T3

```
1 template <>
2 void func<T3>(T3 value) {
3   value.Add(3);
4 }
```

Unterschiede zur normalen Template Deklaration

- leere Template Parameter Liste
- spezialisierte Template Parameter Liste nach dem Funktionsnamen

Compiler

- Template ist spezialisiert
- Welche Parameter zu der Spezialisierung gehören
- Kann für Klasssen- und Funktionstemplates verwendet werden

HAW HAMBURG

```
1 template<typename T>
2 class X {
3   X(T x);
4 };
5 ...
6 template<>
7 classX<std::string> {
8   X(const std::string& s);
9 };
```

Rekursive Klassen Templates

```
1 template<unsigned i>
2 struct Fibonacci {
3   static const unsigned result = Fibonacci<i
        -1>::result + Fibonacci<i-2>::result;
4 };
```

Rekursive Klassen Templates

```
1 template<>
2 struct Fibonacci<0> {
3   static const unsigned result = 1;
4 };
5 template<>
6 struct Fibonacci<1> {
7   static const unsigned result = 1;
8 };
```

Rekursive Klassen Templates

```
1 int main(void) {
2     std::cout << Fibonacci<5>::result << std::
          endl;
3 }
```

Partielle Spezialisierung

- Erlaubt es, ein Template für einen Teil der Parameter zu spezialisieren

```
1 template<typename T, typename U>
2 struct SameType {
3   static const bool result = false;
4 };
```

```
1 template<typename T, typename U>
2 struct SameType {
3   static const bool result = false;
4 };
```

```
1 template<typename T>
2 struct SameType<T, T> {
3   static const bool result = true;
4 };
```

```
1 template<typename T, typename U>
2 struct SameType {
3   static const bool result = false;
4 };
```

```
1 template<typename T>
2 struct SameType<T, T> {
3   static const bool result = true;
4 };
```

```
1 int main(void) {
2     std::cout << "Is␣int␣of␣the␣same␣type␣as␣
          double?" << ()SameType<int, double>::
          result ? "Yes" : "No") << std::endl;
3     std::cout << "Is␣double␣of␣the␣same␣type␣
          as␣double?" << ()SameType<double,
          double>::result ? "Yes" : "No") << std
          ::endl;
4 }
```

```
1 template<typename T>
2 struct IsConst {
3   static const bool result = false;
4 };
```

```
1 template<typename T>
2 struct IsConst<const T> {
3   static const bool result = true;
4 };
```

```
1 template<typename T>
2 struct IsConst {
3   static const bool result = false;
4 };
```

```
1 template<typename T>
2 struct IsConst<const T> {
3   static const bool result = true;
4 };
```

```
1 template<typename T>
2 struct IsVector {
3   static const bool result = false;
4 };
```

```
1 template<typename T>
2 struct IsVector<std::vector<T> > {
3   static const bool result = true;
4 };
```

```
1 template<typename T>
2 struct IsVector {
3   static const bool result = false;
4 };
```

```
1 template<typename T>
2 struct IsVector<std::vector<T> > {
3   static const bool result = true;
4 };
```

Funktions Template Uberladung

- Es ist nicht möglich, Funktions Templates partiell zu spezialisieren
- Man kann aber Funktions Templates überladen

Funktions Template Uberladung

```
1 template<typename T>
2 void swap(T& lhs, T& rhs) {
3   T temp(lhs);
4   lhs = rhs;
5   rhs = temp;
6 }
```

- Performance Nachteil bei großen Objekten
- Es wird ein neues Objekt erstellt und 3x kopiert

Funktions Template Uberladung

```
1 template<typename T>
2 void swap(T& lhs, T& rhs) {
3   T temp(lhs);
4   lhs = rhs;
5   rhs = temp;
6 }
```

- Performance Nachteil bei großen Objekten
- Es wird ein neues Objekt erstellt und 3x kopiert

Funktions Template Uberladung

```
1 class ExpensiveToCopy {
2 public:
3   void swap(ExpensiveToCopy &other);
4 };
```

```
1 template<>
2 void swap<ExpensiveToCopy>(ExpensiveToCopy&
    lhs, ExpensiveToCopy& rhs) {
3   lhs.swap(rhs);
4 }
```

Funktions Template Uberladung

```
1 class ExpensiveToCopy {
2 public:
3   void swap(ExpensiveToCopy &other);
4 };
```

```
1 template<>
2 void swap<ExpensiveToCopy>(ExpensiveToCopy&
    lhs, ExpensiveToCopy& rhs) {
3   lhs.swap(rhs);
4 }
```

Beispiele

```
1 int factorial(int n) {
2   return (n==0) ? 1 : n * factorial(n-1);
3 }
```

```
1 int main() {
2   std::cout << factorial(15) << std::endl;
3 }
```

```
1 int factorial(int n) {
2   return (n==0) ? 1 : n * factorial(n-1);
3 }
```

```
1 int main() {
2   std::cout << factorial(15) << std::endl;
3 }
```

```
1 template<int N>
2 struct Factorial {
3   enum { value = N * Factorial<N-1>::value
        };
4 };
5 template<>
6 Factorial<1> {
7   enum { value = 1 };
8 };
```

```
1 int main() {
2   std::cout << Factorial<15>::value << std::
        endl;
3 }
```

```
1 template<int N>
2 struct Factorial {
3   enum { value = N * Factorial<N-1>::value
        };
4 };
5 template<>
6 Factorial<1> {
7   enum { value = 1 };
8 };
```

```
1 int main() {
2   std::cout << Factorial<15>::value << std::
        endl;
3 }
```

Inhalt

1. Metaprogrammierung mit dem Präprozessor
2. Templates
   - Einleitung
   - Parameter und Typen
   - Abhängige Namen
   - Benutzung
   - Explizite Spezialisierung
   - Partielle Spezialisierung
   - Beispiele
3. Code Bloat

HAW HAMBURG

- In C++ wird das "dynamic dispatching" für Template Funktionen nicht direkt unterstützt
- Generische Algorithmen müssen zur Compile-Zeit für jeden Typ das Template instanziieren

- Das führt zu sogenanntem "code bloat" bzw. "template bloat
- "code bloat" ist bei dem Design von generischen Bibliotheken ein großes Thema

- Ein Verfahren, dass versucht, "code bloat" bei Templates in den Griff zu bekommen ist "template hoisting"
- Beim "template hoisting" wird ein Template in eine generische Basis Klasse und nicht generische Kind Klassen aufgeteilt

HAW HAMBURG

- Ein weiteres Verfahren ist, für verschiedene Algorithmen Typen zu vereinheitlichen → "type reduction"
- Die Vereinheitlichung gilt dann NUR für diesen Algorithmus
- Beispiel: Ein RGB-Bild soll in ein BGR-Bild kopiert werden
  →Das BGR Bild wird durch um-mapping der Farbkanäle in ein Pseudo-RGB Bild konvertiert
  →Somit muss nur noch die Kopierfunktion für RGB→RGB geschrieben werden

HAW HAMBURG

- Bei solchen Techniken ist Vorsicht geboten, da sie häufig aus unsicheren casts aufbauen
- Zu Beachten ist, dass unterschiedliche Algorithmen unterschiedliche Vereinheitlichungen erlauben

- Durch dieses Verfahren wird der Laufzeit-Nachteil in einen Compile-Zeit Nachteil umfunktioniert
  $\rightarrow$Der Code wird kleiner, die Compile-Zeit größer
- Generell entstehen bei exzessiver Template Programmierung oftmals viele Instanz mit ähnlichem bis fast vollständig identischem Code

HAW HAMBURG

Quellen

- **C++ Metaprogrammierung** - Sebastion Otte - Fachhochsule Wiesbaden
- **Introduction to C++ Templates** - Anthony Williams
- **Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat** - Lubomir Bourdev & Jaakko Järvi

HAW HAMBURG

Fragen?

Vielen Dank
für Ihre Aufmerksamkeit!