

# Template-Metaprogrammierung

# B1

## In diesem Kapitel

- ▶ Verfeinern Sie Ihre Kenntnisse über Template-Parameter
- ▶ Erfahren Sie, dass man auch nicht-generische Argumente in Templates einsetzen kann
- ▶ Hören Sie vom Begriff der expliziten Spezialisierung
- ▶ Bekommen Sie einen klitzekleinen Ausblick auf die so genannte Metaprogrammierung mit Hilfe von Templates
- ▶ Erschrecken Sie danach nie mehr, wenn Sie Templates sehen

---

**N**un kommt etwas wirklich Abgefahrenes. Wenn Sie zu den Leuten gehören, denen Logikrätsel keinen Spaß machen, so können Sie den Rest dieses Kapitels zunächst mal überspringen. Aber vielleicht sollten Sie doch reinschnuppern – Sie bekommen nämlich etwas gezeigt, was noch relativ unbekannt ist. Etwas, das noch nicht jeder kann. Mit einem Wort: Was zum Angeben. Nun überzeugt?

Vor dem Lohn steht wie immer – lästige Sache – die Arbeit. Zum besseren Verständnis der Metaprogrammierung mit Hilfe von Template-Klassen lernen Sie in zwei Abschnitten zunächst etwas über weitere Feinheiten der Template-Parameter.

## »Normale« Parameter als Template-Parameter

C++ ist so eine Sprache, deren Eigenschaften manchmal ein wenig – hm – bitchy sind. Als anständiger Mensch wissen Sie hoffentlich nicht, was das Wort auf Deutsch bedeutet, und ich werde es auch bestimmt nicht übersetzen.

Gleich stoßen Sie wieder auf so eine Eigenschaft, die wieder mal am bekannten Weltbild rüttelt. Wie Sie wissen und gelernt haben, werden Template-Deklarationen von

```
template <class T>
```

oder

```
template <typename T>
```

eingeleitet. Wenn Sie viel lesen – und vor allem dann, wenn Sie viel über C++ lesen – wird Ihnen irgendwann eine derartige Zeile über den Weg laufen:

```
template <int size>
```

Bitte seien Sie an dieser Stelle erschüttert, denn eigentlich passt das überhaupt nicht zum bekannten Wissen über Templates. Üblicherweise ist der generische Parameter ein Typ, hier aber steht auf einmal ein `int`. Rein intuitiv wird klar, dass mit der Angabe von `<int size>` keinesfalls verschiedene Typen parametrisiert werden. Was aber wird dann parametrisiert? »Herr Ober, bitte einmal ein Beispiel für `int` als Template-Parameter!« »Medium oder durch?«

```
#include <iostream>
using namespace std;
template <typename T, int size>
class Array
{
public:
    Array() {};
    void setVal(int index, T value)
    {
        if ( (index >= 0) && (index < size) )
            m_Array[index] = value;
    }
    T getVal(int index) const
    {
        if ( (index >= 0) && (index < size) )
            return m_Array[index];
        else
            return T();
    }
    int getSize() const
    {
        return size;
    }
private:
    T m_Array[size];
};
int main()
{
    Array<int, 5> array;
    for (int i = 0; i < array.getSize(); i++)
    {
        array.setVal(i, 25);
    }
    for (int j = 0; j < array.getSize(); j++)
    {
        cout << "j = " << array.getVal(j) << endl;
    }
    return 0;
}
```

---

*Listing Bonus.1.: kapBonus/stdpars.cpp*

Die Klasse selbst ist bezüglich des Template-Parameters `<typename T>` für Sie vollständig verständlich. Und wenn Sie sich das `<int size>` genauer ansehen, stellen Sie fest, dass der konkrete Parameter einfach nur eine Konstante ist. Mehr nicht.

Die Fakten im Überblick.

- ✓ Als Template-Parameter sind auch nicht-generische Parameter zugelassen. Hierbei handelt es sich um Konstanten, die bereits zur Laufzeit bekannt sind.



Erlaubt sind für nicht-generische Template-Parameter nur die Typen `int` (mit seinen Abwandlungen `long`, `unsigned` usw.) sowie Zeiger und Referenzen. Typen wie `float` oder `double` sind nicht erlaubt.

- ✓ Im obigen Beispiel kann auch rasch ein Array für `double` (`Array<double, 20>`) oder für eine andere komplexe Klasse (`Array<Dummy, 10>`) erstellt werden.
- ✓ Generische und nicht-generische Template-Parameter können gemischt werden, aber das haben Sie schon im Beispiel gesehen.
- ✓ Gerade für nicht-generische Template-Parameter gewinnen Defaultwerte für die Template-Parameter neue Bedeutung. Eine kleine Änderung

```
template <typename T, int size = 100>
class Array
```

legt bei der Angabe von `Array<int>` genau 100 Elemente an.

## Keine Regel ohne Ausnahme - explizite Spezialisierung

Die Metaprogrammierung mit Hilfe von Templates beruht vor allem auf einer Eigenschaft von Templates, die sich *explizite Spezialisierung* nennt. Diese Möglichkeit ist in die Welt der Templates aufgenommen worden, weil man manchmal Ausnahmen von der Regel benötigt.

Betrachten Sie das folgende kurze Programm, das eine Template-Klasse realisiert.

```
#include <iostream>
using namespace std;
template <class T>
class A
{
public:
    A(T val)
    {
        cout << "in allgemeinem A::A" << endl;
        m_Value = val;
    }
    T getLeftShiftedValue() const
    {
        return m_Value << 1;
    }
private:
    T m_Value;
};

int main()
{
    A<int>    a1(4);
```

```
    cout << a1.getLeftShiftedValue() << endl;
    return 0;
}
```

---

*Listing Bonus.2: kapBonus.2/explicspec.cpp*

Nicht gerade weltbewegend, die einzige Methode der Klasse A liefert den nach links geschobenen Wert zurück. Der Knackpunkt dieser Klasse zeigt sich, sobald Sie die Zeilen

```
A<double> a2(3.14);
cout << a2.getLeftShiftedValue() << endl;
```

einfügen. Dies ist nicht möglich, weil für den Typ `double` kein Shift-Operator `<<` erklärt ist. Dabei kann man die Operation grundsätzlich ausführen: Wie Sie als alter Händer wissen, entspricht ein Links-Shift einer Multiplikation mit 2. Man müsste also für diese generelle Template-Regel eine Ausnahme für den Typ `double` machen ...

Fügen Sie die folgende explizite Spezialisierung der Klasse A für den Typ `double` vor der Funktion `main` in das Beispielprogramm ein.

```
template <>
class A<double>
{
public:
    A(double val)
    {
        m_Value = val;
    }
    double getLeftShiftedValue() const
    {
        return m_Value * 2.0;
    }
private:
    double m_Value;
};
```

---

*Listing Bonus.3: vollständiger Quellcode in kapBonus.3/explicspec2.cpp*

Und siehe, auf einmal führen die beiden Zeilen

```
A<double> a2(3.14);
cout << a2.getLeftShiftedValue() << endl;
```

nicht mehr zu einem Fehler, sondern es wird schön brav eine 6.28 ausgegeben. Man kann also auch generische Klassen für spezielle Typen einschränken oder für spezielle Typen bestimmte Methoden anders implementieren.

Nehmen Sie es übrigens mit dem Sinn dieses Beispiels nicht so arg genau –für so eine Sache würde ich keine Template-Klasse nehmen, genauso wenig wie Sie. Daher entspricht dieses Beispiel eher dem Motto »A little inaccuracy sometimes saves a ton of explanation«, zu Deutsch: Lassen wir fünf gerade sein.



Unter expliziter Spezialisierung versteht man, dass für eine Template-Klasse für einen bestimmten Template-Parameter eine eigene Spezialisierung verwendet wird, die sich von der allgemeinen Definition der Klasse unterscheidet. Eine solche explizite Spezialisierung wird von einem `template < >` eingeleitet, gefolgt vom Klassennamen und dem Typ, der hier spezialisiert wird:

```
template < >  
class A<double>
```

- ✓ Es ist möglich, eine Klasse für mehr als einen Typ explizit zu spezialisieren.



Wenn Sie für eine Template-Klasse auf einmal lauter explizite Spezialisierungen für diverse Typen schreiben, sollten Sie noch einmal nachdenken, ob die Template-Klasse die richtige Wahl war. Letztlich widerspricht nämlich die explizite Spezialisierung dem Prinzip der generischen Programmierung und sollte auf Ausnahmen beschränkt bleiben.

## Template-Metaprogrammierung

Das Thema steht in der Überschrift – *Metaprogrammierung*. Was soll das sein? Unter Metaprogrammierung versteht man eine Art übergeordnete Programmierung oberhalb der normalen Programmierenebene. Ganz konkret: Man kann mit Metaprogrammierung und Templates Programme schreiben, die sich vor Beginn der Übersetzung erst noch selbst erstellen.

Achten Sie auf den wichtigen Punkt in diesem Satz: *vor* Beginn der Übersetzung. Da liegt der Reiz an der ganzen Geschichte, denn Rechenzeit während des Programmlaufs ist immer knapp. Es macht also Sinn, einige rechenintensive Schritte bereits vorher auszuführen, wo man viel Zeit hat, und dann nur noch das Ergebnis ins Programm zu schreiben.

So ganz fremd ist Ihnen das nicht, unter dem Stichwort *constant folding* ist das ein alter Hut. Angenommen, irgendwo in einem Programm steht die Zeile

```
int a = 5 + 8;
```

dann schreibt der Compiler ja in Wirklichkeit nicht die Rechenoperation für 5 und 8 in den Speicher. Sondern er berechnet die Konstanten und schreibt ins Programm sofort die 13. Es wäre ziemlich sinnlos, hierfür Rechenzeit während des Programms zu opfern.

Template-Metaprogrammierung ist eine dramatische Erweiterung dieses Prinzips auf ganze Funktionen.

Als Lehrbeispiel wollen wir für eine Zeichenkonstante die Anzahl der gesetzten Bits ermitteln. Nehmen Sie den Buchstaben 'm', dieser hat den ASCII-Wert 109, hexadezimal 0x6d. Wie viele Bits sind gesetzt? 0x6d = 0110.1101, also 5 Stück.

Eine entsprechende Funktion dazu sieht dann so aus

```
#include <iostream>  
using namespace std;  
int CountBits(char N)  
{  
    int bit7 = (N & 0x80) ? 1 : 0;  
    int bit6 = (N & 0x40) ? 1 : 0;
```

```
int bit5 = (N & 0x20) ? 1 : 0;
int bit4 = (N & 0x10) ? 1 : 0;
int bit3 = (N & 0x08) ? 1 : 0;
int bit2 = (N & 0x04) ? 1 : 0;
int bit1 = (N & 0x02) ? 1 : 0;
int bit0 = (N & 0x01) ? 1 : 0;
return bit0 + bit1 + bit2 + bit3 +
       bit4 + bit5 + bit6 + bit7;
}
int main()
{
    int c = CountBits('m');
    cout << "'m' hat " << c << " gesetzte Bits"
         << endl;
    return 0;
}
```

*Listing Bonus.4: kapBonus.4/no\_meta.cpp*

Die Funktion wird Ihnen vielleicht ein bisschen umständlich vorkommen mit den vielen Zwischenvariablen, aber damit werden Sie nachher im Metabeispiel leichter die Parallelen entdecken. Startet man das Programm, würfelt es tatsächlich eine 5 heraus. Toll, sieht aber sehr bekannt aus.

Kurze Atempause.



Unter Metaprogrammierung versteht man die Programmierung auf einer Ebene oberhalb des normalen Programms. Insbesondere werden durch Metaprogramme Programme oberhalb der normalen Codeebene erstellt und bereits während der Kompilierung ausgeführt.

- ✓ Schon immer war es der Versuch der Compilerbauer, möglichst viele Berechnungen während der Kompilierung auszuführen, weil man dadurch während des Programmlaufs schon das fertige Ergebnis in den Händen hält.



Solche Optimierungen funktionieren aber nur mit Konstanten, nicht mit Variablen.

### **Nicht-rekursive Template-Metaprogrammierung**

Jawohl, das ist eine Überschrift. So muss das aussehen. Normalerweise lässt man als Autor nach so einer Überschrift alle restlichen Seiten einfach leer, weil ohnehin nie jemand weiter liest. Haben Sie das schon mal gemerkt? In allen komplizierten Mathematik- und Physikbüchern sind ab etwa Seite 70 alle 300 weiteren Blätter leer. Ist aber noch nie jemandem aufgefallen.

Ernst beiseite, schauen Sie sich das Problem der Bitzählung als Realisierung mit Hilfe von Templates an.

```
#include <iostream>
using namespace std;
```

```
template<unsigned char N>
class CountBits
{
    enum
    {
        bit7 = (N & 0x80) ? 1 : 0,
        bit6 = (N & 0x40) ? 1 : 0,
        bit5 = (N & 0x20) ? 1 : 0,
        bit4 = (N & 0x10) ? 1 : 0,
        bit3 = (N & 0x08) ? 1 : 0,
        bit2 = (N & 0x04) ? 1 : 0,
        bit1 = (N & 0x02) ? 1 : 0,
        bit0 = (N & 0x01) ? 1 : 0
    };
public:
    enum
    {
        nbits = bit0 + bit1 + bit2 + bit3 +
                bit4 + bit5 + bit6 + bit7
    };
};

int main()
{
    int c = CountBits<'m'>::nbits;
    cout << "'m' hat " << c << " gesetzte Bits"
          << endl;
    return 0;
}
```

---

*Listing Bonus.5: Auszüge aus kapBonus/meta.cpp*

Zunächst starten Sie das Programm und werden sehen, dass auch hier die 5 als Ergebnis erscheint. Was ist der Unterschied? Nun, das Ergebnis 5 wurde schon während der Kompilierung vom Compiler berechnet!

Eine Beschreibung der Funktionsweise finden Sie im grauen Kasten *Das Metaprogramm unter der Lupe*.

Sollten Sie noch grübeln, wo eigentlich der Trick liegt, lesen Sie sich die folgenden Punkte laut durch.



Die Ersetzung der Template-Parameter findet während der Kompilierung statt. Das Ergebnis in `nbits` liegt also beim Programmstart schon längst fest.

- ✓ Ruft man die Klasse `CountBits<>` mit einer anderen Zahl auf, wird eine neue Spezialisierung erzeugt und auch dafür der Wert richtig berechnet. Der Compiler kann die verschiedenen Spezialisierungen problemlos unterscheiden, da er ja verschiedene Klassen dafür erstellt.

- ✓ Im Zusammenhang mit Template-Metaprogrammierung trifft man sehr oft auf `enum` und den trinären Operator `?:`. Letzterer wird benutzt, um eine Alternative zum `if` zu finden, die man während der Kompilierung bereits benutzen kann.



Ein Aufruf der Form

```
int a = 64;
cout << CountBits<a>::nbits << endl;
```

ist falsch. Nur Konstanten sind erlaubt, `a` ist eine Variable.



### Das Metaprogramm unter der Lupe

Zur Berechnung der Bitanzahl wird eine Template-Klasse `CountBits` verwendet, die keinen Code enthält. Stattdessen definiert diese nur einige `enum`-Konstanten. Betrachten Sie exemplarisch eine Zeile vom ersten `enum`:

```
bit7 = (N & 0x80) ? 1 : 0,
```

Wie Sie wissen, können mit `enum` Konstanten mit Werten belegt werden, ein `enum {Hallo = 5};` belegt die Konstante `Hallo` mit der Zahl 5. Das Gleiche wird hier für die Konstante `bit7` getan.

Zugewiesen wird `bit7` das Ergebnis von `(N & 0x80) ? 1 : 0`, hier wird der trinäre Operator von C++ verwendet. Ist das Ergebnis von `N & 0x80` `true`, liefert der Ausdruck eine 1, ansonsten eine 0. Interessant ist das `N`, es handelt sich hierbei um den Template-Parameter der Klasse `CountBits`:

```
template<unsigned char N>
class CountBits
```

Je nach Wertigkeit des Bits werden auf gleiche Art und Weise auch die anderen 7 Konstanten berechnet, bis runter zu `bit0`.

Erzeugt man nun eine Spezialisierung der Form `CountBits<49>`, wird der Template-Parameter 49 während der Kompilierung eingesetzt und es werden jeweils die Ergebnisse für die Konstanten berechnet.

Um an das Ergebnis heranzukommen, verwendet man eine weitere `enum`-Konstante `nbits`.

```
public:
    enum
    {
        nbits = bit0 + bit1 + bit2 + bit3 +
            bit4 + bit5 + bit6 + bit7
    };
```

Zunächst muss dieser `enum`-Wert `public` sein, sonst könnte man nicht auf ihn außerhalb der Klasse zugreifen. Die Erzeugung des Werts ist einfach, man rechnet nur die bereits vorhandenen Konstanten zusammen. Dies ist eine Operation, die der Compiler im Rahmen des `constant folding` problemlos beherrscht.

Um das Ergebnis zu nutzen, greift man einfach auf `nbits` der Klasse `CountBits` zu, üblicherweise geht diese über `CountBits::nbits`. Doch halt, hier liegt eine Template-Klasse vor, man muss noch den Template-Parameter angeben:

```
int c = CountBits<'m'>::nbits;
```

Damit wird für die Konstante 'm' eine Spezialisierung der Template-Klasse `CountBits` erzeugt und man fragt die Konstante `nbits` dieser Klasse ab.

## Rekursive Template-Metaprogrammierung

Bereits die vorige Überschrift hat den Gedanken aufgedrängt, viele Template-Metaprogramme verwenden rekursive Algorithmen. Erinnern Sie sich an Ihre ersten Schritte in der Programmierung zurück, bestimmt haben Sie schon mal eine Fakultätsfunktion rekursiv programmieren müssen. Später haben Sie das wohl nie mehr gemacht, weil Sie wussten, dass rekursive Funktionen böse sind – durch den mehrfachen Einsprung in die gleiche Funktion wird viel Speicher benötigt. Und stattdessen ersetzten Sie die Rekursion durch Schleifen.

Nun gut, lassen wir die Vergangenheit einen Moment ruhen. Metaprogrammierung bedeutet in unserem Fall, dass das Programm noch nicht läuft. Also gibt es keine Schleifen. Um aber wiederholte Vorgänge auszuführen, verwendet man – Rekursion. Denn diese Art Aufruf kann der Compiler ausführen.

Betrachten Sie ein simples Beispiel für eine rekursive Funktion, die Fakultät. Sie wissen, dass  $n! = n * (n-1) * (n-2) * \dots$  ist.  $6! = 5 * 4 * 3 * 2 * 1$ , benötigt wird das vor allem in der Kombinatorik. Man kann eine Fakultät berechnen, indem man die aktuelle Zahl mit dem bisherigen Ergebnis von  $(n-1)!$  multipliziert. Eine rekursive Laufzeitvariante sieht so aus:

```
unsigned long Factorial(unsigned long n)
{
    if (n <= 1)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

Jede Rekursion benötigt eine Abbruchbedingung, weil irgendwo die kleinste Stelle erreicht ist, an der man die Berechnung direkt durchführen kann. Dies ist z.B. bei der Fakultät  $1!$  mit dem Ergebnis 1 der Fall.

Nachteil dieser Lösung: Es wird eben erst zur Laufzeit des Programms gerechnet. Schauen Sie sich nun die Template-Variante an.

```
template<int N>
class Factorial
{
public:
    enum
    {
        value = N * Factorial<N-1>::value
    }
};
```

```
};  
};  
template<>  
class Factorial<1>  
{  
public:  
    enum { value = 1 };  
};  
template<>  
class Factorial<0>  
{  
public:  
    enum { value = 1 };  
};
```

*Listing Bonus.6: Ausschnitt aus kapBonus/meta.cpp*

Ergänzen Sie gerade noch die `main`-Funktion mit der Zeile

```
int f = Factorial<6>::value;  
cout << "6! ist " << f << endl;
```

Zunächst die erstaunliche Tatsache: Diese Fakultätsberechnung funktioniert. Lernen Sie im grauen Kasten *Templates mit ein bisschen Fakultät* die genaue Funktionsweise.



### **Templates mit ein bisschen Fakultät**

Leichter zu verstehen ist die Template-Klasse `Factorial<N>`. Wie es vom Beispiel für die Bitzählung bekannt ist, wird in der Klasse `Factorial` eine öffentliche `enum`-Konstante definiert. Deren Wert lautet

```
value = N * Factorial<N-1>::value
```

Hier kommt die Rekursion zum Tragen, es wird die aktuelle Zahl mit dem Ergebnis der Fakultät von  $(n-1)$  multipliziert. Sie kennen bereits die Möglichkeit, das Ergebnis einer solchen Template-Berechnung über die `enum`-Konstante zu verwenden, hier wird dies über `Factorial<N-1>::value` erreicht. Die Multiplikation mit  $N$  ist einfach die Berechnung mit dem Wert der aktuellen Spezialisierung.

Der neue Trick hier ist die Spezialisierung für die Abbruchbedingung:

```
template<>  
class Factorial<1>  
{  
public:  
    enum { value = 1 };  
};
```

Die Template-Klasse `Factorial` wird für `<1>` noch einmal explizit definiert, wobei man hier dem `enum`-Wert `value` die 1, also die Abbruchbedingung, zuweist. Sobald also in den allgemein gehaltenen

nen Templates für  $N$  irgendwann  $N$  zu 1 wird, liefert der Aufruf von `Factorial<1>::value` die Abbruchbedingung 1 zurück.

Da  $0! = 1$  ist, muss dieser Trick auch noch einmal für `<0>` wiederholt werden. Entfernen Sie doch mal die Spezialisierung für `<1>` aus dem Code und versuchen Sie, das Programm zu kompilieren.

Wenn Sie nun noch nicht schielen, setze ich frei nach dem Motto »Einen hab' ich noch« ein weiteres Beispiel drauf. Nur damit Sie noch ein klein bisschen sicherer damit werden. Die folgende Template-Klasse berechnet den Zweierlogarithmus einer Zahl. Das ist oftmals ganz nützlich, weil man dadurch ermitteln kann, dass der Zweierlogarithmus von 8 die 3 ist und man daher 8 Werte in 3 Bits speichern kann.

```
template <int N>
class Log2
{
public:
    enum
    {
        value = 1 + Log2<N/2>::value
    };
};
template<
class Log2<1>
{
public:
    enum {value = 0};
};
```

---

*Listing Bonus.7: Auszüge aus kapBonus/meta.cpp*

Die Zeile zum Aufruf kann so aussehen

```
int a = Log2<8>::value;
cout << "Log2 von 8 ist " << a << endl;
```

Vom Funktionsablauf her sieht dies ähnlich aus wie bei der Fakultät. Es wird so lange die aktuelle Zahl durch 2 dividiert, bis man bei der 1 ankommt. Die Spezialisierung für `Log2<1>` liefert nun den Wert 0 zurück und erzeugt den Abbruch der Rekursion. In der Template-Klasse `Log2<N>` wird auf den Zweierlogarithmus von  $N/2$  immer eine 1 addiert.

Lustig, oder? Ich hoffe, es ist mir gelungen, Sie so neugierig zu machen, dass Ihr Forscherdrang für eigene Experimente geweckt wurde. Und geben Sie zu – nun wirken solche simplen Templates für Klassen richtig banal. Vergessen Sie über Ihrer Euphorie bezüglich der Metaprogrammierung nicht die folgenden wissenswerten Punkte.



Das vollständige Beispielprogramm finden Sie als `KAPBONUS/META.CPP` im Internet.



Schneller als zur Compilezeit geht nicht. Egal, wie schnell der Prozessor oder wie gut und effektiv die Programmiersprache sein mag – wenn das Ergebnis schon vor Programmstart feststeht, gewinnen Sie jeden Benchmark.

- ✓ Metaprogrammierung mit Hilfe von Templates wird vor allem für mathematische Anwendungen genutzt. Komplizierte Formeln, Rechenausdrücke oder langwierige rekursive Formeln für Folgen und Reihen werden dadurch dramatisch beschleunigt. Auch Tabellen mit vorberechneten Zahlenwerten lassen sich damit sehr leicht erzeugen.



Suchen Sie im Internet nach dem Begriff »Metaprogrammierung C++« und Sie werden noch weitere Beispiele finden.

- ✓ Diese Verfahren sind nur für konstante Ausdrücke geeignet.



Die Kompilierzeit steigt durch diese Anwendungen teilweise dramatisch an, bei komplizierten Beispielen steigt auch schon mal der Compiler aus.



Hier ist noch ein Gaumenkitzler für Freunde von Templates (KAPBONUS/TEASER.CPP).

```
template <class Derived, class Base>
class Check
{
    class Nope {};
    class Yep {char Dummy[3];};
    static Yep Test(Base*);
    static Nope Test(...);
public:
    enum {
        IsDerived = sizeof(Test(static_cast<Derived*>(0)))
                == sizeof(Yep)
    };
};
class X {};
class Y : public X {};
int main()
{
    cout << Check<Y, X>::IsDerived << endl;
    cout << Check<int, string>::IsDerived << endl;
    return 0;
}
```

Erraten Sie ohne Start des Programms, was hier passiert? Die Lösung steht hier – lesen Sie den Satz von hinten:

0 enie netsnosna ,tSi tetieleg B nov A nnew , 1 enie trefeil devireD ::<B ,A>kcehC