# Wie geht es weiter mit C++?

Beschlossene und geplante Änderungen des C++-Standards

Frank Listing
f.listing@microconsult.com

Seit der Version 11 bewegt sich wieder etwas im Hause C++.

Hat C++ 11 große Änderungen gebracht, kam mit der Version 14 eher ein "Maintenance Update" mit wenig neuen Features und vor allem Korrekturen und Feintuning.

Die Version 17 - auch 1z genannt - wird wieder als großes Release bezeichnet und enthält dementsprechend wieder mehr neue Features.

Der Vortrag zeigt eine Auswahl neuer Sprachfeatures der Versionen 14 und 17.

# C++ 14

Der Standard C++ 14 ist schon seit 2 Jahren verabschiedet.

Moderne Compiler haben die beschlossenen Änderungen praktisch vollständig umgesetzt.

Eine Übersicht, welcher Compiler welches Feature implementiert hat, ist auf folgender Webseite zu finden:

http://en.cppreference.com/w/cpp/compiler_support

# decltype(auto)

Das Schlüsselwort `decltype` kann nun in Kombination mit `auto` verwendet werden, um z.B. Referenztypen als Referenz zu erhalten.

```
auto a = 42;              // int
auto &b = a;              // int&
auto c = b;               // int
decltype(auto) d = b;     // int&
```

# auto als Rückgabewert von Funktionen

Ab C++ 14 kann mit `auto` der Rückgabewert von Funktionen vom Compiler ermittelt werden.

```
auto add(int x, int y)
{
    return x + y;
}

template <typename T, typename U>
auto add(T x, U y)
{
    return x + y;
}
```

Es gibt `auto` zwar schon in C++ 11, aber ohne automatische Typ-Erkennung.

```
auto add(int x, int y) -> int ...
```

# Template-Konstanten

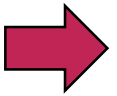
Die Template-Konstanten erleichtern die Bereitstellung von konstanten Werten für unterschiedliche Datentypen.

```
template<typename T>
constexpr T pi = 3.14159265358979323846264338328L;

auto constexpr pf = pi<float>;         // pf --> float

auto constexpr pd = pi<double>;        // pd --> double

auto constexpr pld = pi<long double>; // pld --> long double
```

# Binäre Literale

Mit Hilfe des Präfixes `0b` können Zahlen nun auch binär dargestellt werden.

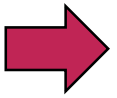
```
auto n1 = 0b0101;
auto n2 = 0b111011011011111101110111;
```



```
0x5
0x76df77
```

Weiterhin können ab C++ 14 numerische Literale mit Hilfe des Apostrophen (`'`) strukturiert werden.

```
auto n3 = 3'020'011;
auto n4 = 0b111'0110'1101'1111'0111'0111;
```



```
3020011
7790455
```

# Generische Lambdas

Lambda-Funktionen können generisch geschrieben werden. Für die Parametertypen wird `auto` angegeben.

Damit ist die Verwendung von Lambda-Funktionen im Zusammenhang mit Templates einfacher.

```
auto square = [](auto i) {return i * i; };

auto result1 = square(5); // result1 -> int
cout << result1 << endl;

auto result2 = square(5.35); // result2 -> double
cout << result2 << endl;
```

# Funktionslokale Konstanten bzw. Variablen

Für Lambda-Funktionen können eigene Capture-Variablen eingeführt werden. Damit bekommt eine Lambda-Funktion eigenen Speicher.

```
auto calc = [i = 39]() {return i + 5; };
```

```
auto x = 23;
auto calc = [i = x]() {return i + 5; };
```

Diese Variable muss zwingend initialisiert werden.

Sie wird ein Member des Funktors, in den die Lambda-Funktion konvertiert wird.

# Funktionslokale Konstanten bzw. Variablen

Wenn diese Variable geändert werden soll, muss die Lambda-Funktion `mutable` sein.

```
int main()
{
    auto x = 23;
    auto calc = [i = x]() mutable {return i += 5; };

    std::cout << calc() << "\n"; // i -> 28
    std::cout << calc() << "\n"; // i -> 33
    std::cout << calc() << "\n"; // i -> 38
}
```

# C++ 17

Die Version 17 des C++-Standards ist bereits „Feature Complete“. Jetzt müssen die Features noch offiziell verabschiedet werden.

Einige Compiler haben schon Teile des kommenden Standards umgesetzt.

Eine Übersicht, welcher Compiler welches Feature implementiert hat, ist auf folgender Webseite zu finden:

http://en.cppreference.com/w/cpp/compiler_support

# Automatische Typerkennung für Template-Parameter

(Template argument deduction for class templates)

Bisher war es nur bei Template-Funktionen möglich die Template-Parameter anhand der Datentypen der übergebenen Argumente zu bestimmen. Bei Klassen mussten immer die Typen angegeben werden.

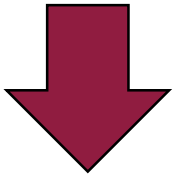
```
template<class T>
T max(T t1, T t2)
{
    return (t1 > t2) ? t1 : t2;
}

int main()
{
    auto maxValue = max(42, 23);
}
```

```
template<class T>
class Store
{
    T value;

public:
    Store(T t) : value(t) {}

    T getValue() { return value; }
};

int main()
{
    Store<double> s(4.2);
    auto storedValue = s.getValue();
}
```

Mit der automatischen Typerkennung für Template-Parameter von Klassen kann nun der Compiler aus den Argumenten, die dem Konstruktor übergeben werden, die Typen der Template-Parameter ermitteln.

```
template<class T>
class Store
{
    T value;

public:
    Store(T t) : value(t)
    {}

    T getValue()
    { return value; }
};
```

```
int main()
{
    Store<double> s(4.2);
    auto storedValue = s.getValue();
}
```



```
int main()
{
    Store s(4.2);
    auto storedValue = s.getValue();
}
```

# template <auto>

Template-Parameter, die keine Typ-Parameter sind, können nun mit `auto` deklariert werden.

```
template <typename Type, Type value>
constexpr Type constant = value;

constexpr auto IntConstant42 = constant<int, 42>;
```

```
template <auto value>
constexpr auto constant = value;

constexpr auto IntConstant42 = constant<42>;
```

Auch bei den variadischen Templates, erleichtert `auto` in einigen Fällen die Deklaration. Das folgende Beispiel war vor C++ 17 nicht so einfach umzusetzen.

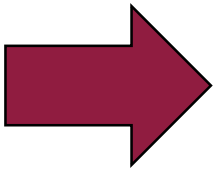
```
template <auto ... vs>
struct HeterogenousValueList
{
    ...
};

using ImportantList = HeterogenousValueList<42, 'X', 13u>;
```

# Fold Expressions

Fold Expressions vereinfachen die Programmierung variadischer Templates.

Vor allem die Programmierung von Funktionen mit variabler Parameteranzahl wird vereinfacht.

```
auto sum()
{
    return 0;
}

template<class T, class... Others>
auto sum(T arg0, Others... others)
{
    return arg0 + sum(others...);
}
```



```
template<class... T>
auto sum(T... param)
{
    return (... + param);
}
```

```
int main()
{
    std::cout << sum(1, 2, 3, 4) << "\n";           // 10
    std::cout << sum(7.1, 2.5, 8.3, 7.9) << "\n";   // 25.8
}
```

Ein einfache Print-Funktion mit Hilfe von Fold Expressions.

```
template<class... T>
void print(T&&... param)
{
    (std::cout << ... << param) << "\n";
}

int main()
{
    print(sum(1, 2, 3, 4));
    print(1, 2, 3, 4);
    print(7.1, "Otto", 8.3, 'x');
}
```

```
10
1234
7.1Otto8.3x
```

# constexpr if

Mit dem neuen Feature `constexpr if` wird die Erstellung von typabhängigen Templates wesentlich vereinfacht.

In der Version 11 wurde `std::enable_if` eingeführt, um typgebundene Varianten von Templates zu erstellen.

```
template<typename T>
struct Test
{
    template<typename D = T, typename std::enable_if<
                 std::is_same<D, int>::value, int>::type = 0>
    void doSomething()
    {
        std::cout << "Test uses int" << std::endl;
    }

    template<typename D = T, typename std::enable_if<
                 !std::is_same<D, int>::value, int>::type = 0>
    void doSomething()
    {
        std::cout << "Test does not use int" << std::endl;
    }
};
```

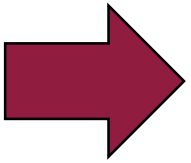
Ab C++ 17 ist der Code wesentlich besser zu lesen.

```
template<typename T>
struct Test
{
    void doSomething()
    {
        if constexpr(std::is_same<T, int>::value)
        {
            std::cout << "Test uses int" << std::endl;
        }
        else
        {
            std::cout << "Test does not use int" << std::endl;
        }
    }
};
```

# Capture *this in Lambda-Funktionen

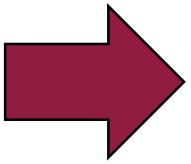
Lokale Variablen können in einer Capture-Liste kopiert oder referenziert werden.

```
int main()
{
    auto i = 5;
    auto l = [i]()mutable{i++;};
    l();

    std::cout << i << "\n";
}
```



5

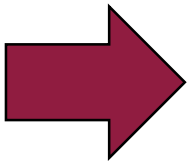
```
int main()
{
    auto i = 5;
    auto l = [&i]()mutable{i++;};
    l();

    std::cout << i << "\n";
}
```



6

In C++ 11 und 14 gibt es keine direkte Möglichkeit, eine Membervariable per Kopie einer Lambda-Funktion zu übergeben.

```
struct S
{
    int i = 5;

    void test()
    {
        auto l = [=]()mutable{i++;};
        l();

        std::cout << i << "\n";
    }
};
```



6

Die Membervariable `i` kann nicht in die Capture-Liste eingetragen werden und das `=` kopiert den `this`-Pointer. Damit ist `i` referenziert und nicht kopiert worden.

Workaround: Anlegen einer lokalen Kopie von `*this`.

C++ 11:

```
void test()
{
    auto temp = *this;
    auto l = [temp]()mutable{temp.i++;};
    l();

    std::cout << i << "\n";
}
```

C++ 14:

```
void test()
{
    auto l = [=, temp = *this]()mutable{temp.i++;};
    l();

    std::cout << i << "\n";
}
```

Mit C++ 17 wurde nun eine Möglichkeit vorgesehen, direkt mit einer Kopie zu arbeiten. Dafür wird in die Capture-Liste `*this` eingetragen.

```
struct S
{
    int i = 5;

    void test()
    {
        auto l = [*this]()mutable{i++;};
        l();

        std::cout << i << "\n";
    }
};
```

# Inline-Variablen

Eine weitere Verbesserung betrifft die Definition von statischen Membervariablen.

Vor C++ 17

```
// header

struct S
{
    const static int i;

    void test()
    {
        std::cout << i << "\n";
    }
};
```

```
// implementation

const int S::i = 5;
```

C++ 17

```
// header

struct S
{
    const static inline int i = 5;

    void test()
    {
        std::cout << i << "\n";
    }
};
```

# Initialisierung innerhalb von if, while, switch

Ähnlich wie bei der `for`-Schleife können nun, z.B. auch bei einer `if`-Abfrage, lokale Variablen erzeugt und initialisiert werden.

```
std::map<int, std::string> m;

void demo(int key, std::string value)
{
    if (auto p = m.try_emplace(key, value); !p.second)
    {
        std::cout << "Element already registered\n";
    }
    else
    {
        std::cout << value << " inserted\n";
    }
}

if(!p.second) // error: use of undeclared identifier 'p'
{ // do something more }
```

Die Variable `p` ist jetzt nur im `if` und im `else` Zweig verfügbar.

# Standardbibliothek

Neben den neuen Sprachfeatures wurde auch die Standardbibliothek kräftig erweitert und verbessert, z.B.

- Neue Datentypen (z.B. `std::variant` als typsicherer Union-Ersatz)
- Neue Hilfsklassen zum Aufruf aller Arten von Funktionspointern
- Klassen und Funktionen für den Zugriff auf das Dateisystem
- Neue Klassen für Multithreading
- Parallele Versionen von Standardalgorithmen
- Verbesserungen der Containerklassen
- Überflüssiges wurde entfernt!!! Z.B. `std::auto_ptr`, operator++ für `bool`.
- …

## Fazit

C++ 14 war vor allem ein Feintuning der Version 11.

Mit C++ 17 wurden nicht nur viele neue Features hinzugefügt, sondern C++ wurde auch (sehr vorsichtig) bereinigt.

- Überflüssige Konstrukte wurden entfernt (z.B. `auto_ptr`)
- Komplexe Ausdrücke vereinfacht (z.B. `enable_if` → `if constexpr`).

## MicroConsult Training & Coaching

- Objektorientierte Programmierung mit C++
- C++ für Fortgeschrittene: Erweiterte Nutzung gemäß ISO-Standard (C++11/C++98)
- Modernes C++: Neuerungen durch C++11 und C++14 bei Sprachsyntax, Bibliothek und Templates
- Embedded C++: Objektorientierte Programmierung für Mikrocontroller mit C++/EC++ und UML

## Wertvolle Fachinformationen rund um die Softwareentwicklung

stehen hier für Sie zum Download bereit.

**www.microconsult.de**

# Happy coding with C++