Embedded Software Engineering Kongress

Wie geht es weiter mit C++?

Beschlossene und geplante Änderungen des C++-Standards

Frank Listing, MicroConsult GmbH

Seit der Version 11 bewegt sich wieder etwas im Hause C++. Wurde lange Zeit die Weiter-entwicklung des C++ Standards von der (Programmier-) Öffentlichkeit kaum wahrgenommen, hat die Version 11 von C++ der Programmiersprache neuen Schwung verliehen.

Nachdem C++ 11 große Änderungen gebracht hat, so 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.

Im Folgenden wird eine Auswahl neuer Sprachfeatures der C++-Versionen 14 und 17 gezeigt.

C++ 14

Der Standard C++ 14 ist schon seit 2 Jahren verabschiedet und damit fast ein „alter Hut". Moderne Compiler haben die beschlossenen Änderungen praktisch vollständig umgesetzt.

Eine Übersicht, welcher Compiler welches Feature implementiert hat, findet sich auf der Webseite http://en.cppreference.com/w/cpp/compiler_support.

Eine der kleineren Neuerungen ist die Kombination von decltype mit auto, um den ursprünglichen Datentypen einer Variablen weiterzureichen. Damit bleiben z.B. Referenzen erhalten (Listing 1).

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

Listing 1: decltype(auto)

Weiterhin kann auto nun auch verwendet werden, um den Rückgabetypen einer Funktion vom Compiler automatisch ermitteln zu lassen (Listing 2).

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

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

Listing 2: auto als Rückgabewert

Eine weitere kleine Erleichterung ist die Bereitstellung von Konstanten in Template-Form. Damit kann auf einfache Art dieselbe Konstante für unterschiedliche Datentypen bereitgestellt werden (Listing 3):

```
template<typename T> constexpr T pi = 3.1415926535897932384626433832 8L;

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

Listing 3: Template-Konstanten

Auch bei der Darstellung von Zahlen gab es Erweiterungen. Zu den schon bekannten Darstellungsformaten oktal, dezimal und hexadezimal kommt nun die Möglichkeit hinzu, Zahlen in Binärform im Code abzulegen (Listing 4). Außerdem können numerische Literale mit Hilfe von Apostrophen strukturiert werden (Listing 5).

```
auto n1 = 0b0101;
auto n2 = 0b1110110110111110111 0111;
```

Listing 4: Binärdarstellung

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

Listing 5: Strukturierung

Auch bei den Lambda-Funktionen gibt es Neuerungen. Die Lambdas können nun auch generisch geschrieben werden. Für die Parametertypen wird auto angegeben. Der Compiler ermittelt die Datentypen für die Parameter aus dem Kontext der Umgebung (Listing 6). 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;
```

Listing 6: Generische Lambdas

Eine weitere Neuerung für Lambda-Funktionen sind funktionslokale Capture-Variablen. Damit bekommt eine Lambda-Funktion eigenen Speicher (Listing 7). So eine Variable muss zwingend initialisiert werden. Sie wird ein Member des Funktors, in den die Lambda-Funktion konvertiert wird.

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

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

Listing 7: Eigene Capture-Variablen

Das soll als kleine Auswahl der neuen Sprachfeatures von C++ 14 reichen. Daneben gibt es natürlich noch etliche Erweiterungen der Standardbibliothek.

C++ 17

Die Version 17 des C++-Standards ist bereits „Feature Complete“. Jetzt müssen die Features noch offiziell verabschiedet werden. Einige Compiler haben auch schon Teile des kommenden Standards umgesetzt.

Eine Übersicht, welcher Compiler welches Feature implementiert hat, findet sich auf der Webseite http://en.cppreference.com/w/cpp/compiler_support.

Der Komfort im Umgang mit Templates wird immer weiter verbessert. So war es in der Vergangenheit bereits möglich, die Template-Parameter von Funktionen automatisch aus den übergebenen Argumenten abzuleiten. 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 (Listing 8).

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

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

    T getValue() { return value; }
};

int main()
{
    Store<double> s(4.2);  // before C++ 17
    Store s(4.2);          // since C++ 17
    auto storedValue = s.getValue();
}
```

Listing 8: Automatische Typerkennung für Template-Parameter

Weiterhin kann auto auch im Zusammenhang mit Template-Parametern, die keine Typparameter sind, verwendet werden. Damit wird auch wieder wertvoller Quellcode eingespart (Listing 9).

```
// before C++ 17
template <typename Type, Type value>
constexpr Type constant = value;
constexpr auto IntConstant42 = constant<int, 42>;

// since C++ 17
template <auto value>
constexpr auto constant = value;
constexpr auto IntConstant42 = constant<42>;
```

Listing 9: template<auto>

Im Zusammenhang mit variadischen Templates schafft template<auto> noch mehr Erleichterung. Das Beispiel in Listing 10 war vor C++ 17 ohne zusätzliche Containerklasse nicht umzusetzen.

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

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

Listing 10: template<auto…>

Die Einführung der variadischen Templates in C++ 11 vereinfacht die Implementierung von Funktionen mit variabler Parameteranzahl. Im neuen Standard wird die Anwendung von Operationen auf die einzelnen Parameter durch die sogenannten Fold Expressions weiter erleichtert (Listing 11).

```
// before C++ 17
auto sum()
{
    return 0;
}

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

// since C++ 17
template<class... T>
auto sum(T... param)
{
    return (... + param);
}
```

Listing 11: Fold Expressions

Type Traits dienen dazu, Varianten von Templates für spezielle Anwendungsfälle bereitzustellen. Im Zusammenhang mit variadischen Templates werden sie benötigt, um bestimmte Spezialfälle abzudecken. Mit C++ 11 wurde std::enable_if bereitgestellt, um noch flexibler zu werden. Allerdings kann der Code dann auch etwas unübersichtlich werden (Listing 12).

```
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" << "\n";
    }

    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" << "\n";
    }
};
```

Listing 12: std::enable_if

Mit dem neuen Feature constexpr if wird die Erstellung dieser Templates wesentlich vereinfacht. Der Code ist weniger verwirrend und einfacher zu lesen (Listing 13).

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

Listing 13: constexpr if

Auch bei den Lambda-Funktionen gibt es kleinere Verbesserungen. Sie sind nun implizit constexpr und lassen sich in entsprechenden Funktionen verwenden. Lambdas in Memberfunktionen können *this in der Capture-Liste angeben, um ohne Workaround nur auf einer Kopie der Membervariablen zu arbeiten.

Das Schlüsselwort inline kann mit dem neuen Standard auch für statische Membervariablen verwendet werden. Damit wird die separate Definition in der Implementierungsdatei überflüssig. Die Membervariable steht nur noch im Header (Listing 14).

```
// header
struct S
{
    const static inline int i = 5;
    void test()
    {
        std::cout << i << "\n";
    }
};
```

Listing 14: inline für Membervariablen

Eine weitere Änderung betrifft die Statements if, while und switch. Ähnlich, wie bei der for-Schleife, können lokale Variablen angelegt und initialisiert werden (Listing 15). Die angelegte Variable (hier p) ist im Beispiel nur im if- und im else-Zweig verfügbar.

```
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
    }
}
```

Listing 15: if-Initialisierung

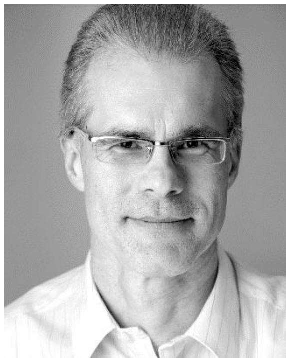
Neben den neuen Sprachfeatures wurde auch die Standardbibliothek kräftig erweitert und verbessert, z.B.

- 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, postfix-Inkrement für bool.
- …

Weiterführende Informationen – 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

Autor



Dipl.-Ing. Frank Listing ist seit 2002 Trainer und Projektcoach bei der MicroConsult GmbH mit dem Schwerpunkt Microsoft-Plattformen, objekt-orientierte Programmierung und Testen von Embedded Systemen und u.a. fachlich für das Thema .NET verantwortlich. Sein Wissen gibt er immer wieder auch in Publikationen und Fachvorträgen weiter.

Kontakt
f.listing@microconsult.de
www.microconsult.de