10 kleine Dinge, die C++ einfacher machen

Wartbaren Code durch den Einsatz von modernen C++ features

Dominik Berner, bbv Software Services

Die neuen Standards haben die Programmiersprache C++ merklich modernisiert und teilweise ganz neue Programmierparadigmen in die Welt von C++ eingebracht.

Die "großen" Änderungen wie Variadic Templates, `auto`, Move-Semantik, Lambda-Ausdrücke und weitere haben für viel Diskussionsstoff gesorgt und sind dementsprechend weit herum bekannt. Nebst den Sprachfeatures hat auch die Standard-Bibliothek eine merkliche Erweiterung erfahren und viele Konzepte aus Bibliotheken wie `boost` wurden so standardisiert. Nebst diesen sehr spürbaren (und teilweise auch umstrittenen) Features gibt es eine ganze Menge an kleinen -aber-feinen Spracherweiterungen, die oft weniger bekannt sind oder übersehen werden.

Gerade weil diese Features oft sehr klein und teilweise fast unsichtbar sind, haben sie großes Potential um im Programmiereralltag das Leben einfacher zu machen und Code ohne schwerwiegende Eingriffe sanft zu modernisieren. Es ist oft so, dass man bei der Arbeit mit bestehendem Code nicht die Möglichkeit hat, große strukturelle oder von außen sichtbare Änderungen vorzunehmen, aber genau hier können die "kleinen Features" helfen, Code aktuell und wartbar zu halten.

Moderner, wartbarer Code

Wartbarkeit, Lesbarkeit und Code-Qualität sind Themen die aus der heutigen Software-Entwicklung nicht mehr wegzudenken sind. Der Vorteil von Software gegenüber Hardware ist, das sie sich relativ leicht anpassen und überarbeiten lässt und insbesondere dort, wo agil gearbeitet wird, geschieht das oft sehr bewusst und immer wieder. Durch diese Volatilität werden gewinnen diese Qualitätsmerkmale noch mehr an Gewicht, denn schlechter Code macht den Vorteil der einfachen Bearbeitung mehr als zunichte. Dinge wie Clean Code, das SOLID-Prinzip oder Paradigmen wie Low Coupling, Strong Cohesion sind wichtige Aspekte von Codequalität, aber Qualität beginnt bereits bei der Verwendung der Sprache selbst. Das Verwenden der zur Verfügung gestellten Sprachfeatures und -Funktionen hilft die **Absicht hinter dem geschriebenen Code** zu verdeutlichen und erleichtert oft auch das automatische Verifizieren dieser Absichten. Zudem kann oft dadurch die Menge von geschriebenem Code reduziert werden, was dem Prinzip von "Less code means less bugs" in die Hände spielt.

Ein Beispiel zur Illustration

Ein einfacher Algorithmus kann sehr kompliziert zu verstehen sein, wenn die Schreibweise nicht den Erwartungen entsprechen oder der Autor sich einen besonders schlauen Hack zur Optimierung einfallen ließ. So kann zum Beispiel das tauschen von zwei variablen `x` und `y` wie folgt geschrieben werden:

```
x = x ^ y;
y = y ^ x;
x = x ^ y;
```

Dieser XOR-Swap ist zwar speicher-effizient und hat in ganz spezifischen Fällen seine Daseinsberechtigung, aber intuitiv lesbar ist die Operation nicht. Selbst mit einem Code-Kommentar versehen zwingt dieses einfache Beispiel dem Leser unnötige Denkarbeit auf. Dem gegenübergestellt liest sich das folgende Beispiel viel einfacher:

```
std::swap(x,y);
```

Die folgenden 10 kleinen Features und Erweiterungen hauptsächlich aus den modernen C++-Standards helfen Code kompakt und lesbar zu halten und somit die Code-Qualität zu verbessern.

Vererbung kontrollieren mit `override` und `final`

Vererbung ist für viele Programmierer Fluch und Segen gleichermaßen. Einerseits hilft sie of Code-Duplizierung zu vermeiden, andererseits gibt es dabei - insbesondere in C++ - viele Stolpersteine, die beachtet werden müssen. Gerade bei Refactorings an Basisklassen kommt es immer wieder vor, dass die abhängigen Klassen vergessen werden und man dies erst zur Laufzeit merkt. Das Schlüsselwort `override` liefert hier seit C++11 Abhilfe. Wann immer eine Funktion in einem Vererbungsbaum überschrieben wird, sollte `override` verwendet werden. Damit wird eine überschriebene Funktion automatisch virtuell und der Compiler erhält die Möglichkeit um zu überprüfen, ob auch tatsächlich eine Methode überschrieben wird und ob die überschriebene Methode auch tatsächlich virtuell ist.

```
struct Base
{
  virtual int func() { return 1; }
};

struct Derived : public Base
{
  // Compiler-error if Base::func does not exist or is not virtual
  int func() override { return 2; };
};
```

Noch mehr Kontrolle über den Vererbungsbaum erhält man, wenn man die Vererbung ab einem gewissen Punkt komplett unterbinden kann. Der Spezifikator `final` zeigt an, dass eine Klasse oder virtuelle Funktion nicht weiter überschrieben werden kann. Dies verringert zwar den Schreibaufwand nicht, aber kommuniziert ganz klar eine Absicht hinter einen Stück Code, nämlich dass keine weitere

Vererbung erwünscht ist. Hier hilft sogar der Compiler mit indem die Kompilierung fehlschlägt, sollte man dies doch versuchen.

```
class Base final
{ };

class Derived : public Base {} // Compiler error
class Base
{
  virtual void f();
};

class Derived : public Base
{
  // f cannot be overriden by further base classes
  void f() override final;
};
```

`using`-Deklarationen und Konstruktorenvererbung

Code zu duplizieren ist dem Programmierer ein Graus, selbst wenn es sich hier um generierten Code handelt. `using`-Deklarationen erlauben es dem Programmierer ein Symbol von einer deklarativen Region, wie Namensräume, Klassen und Strukturen in einen anderen zu "importieren" ohne dass zusätzlicher Code generiert wird. Bei Klassen ist dies vor allem nützlich um Konstruktoren von Basisklassen direkt zu übernehmen, ohne dass alle Varianten neu geschrieben werden müssen. Ein weiteres Beispiel ist um kovariante Implementierungen in abgeleiteten Klassen explizit zu gestalten. Damit wird dem dem Leser klar signalisiert, dass hier eine "fremde" Implementation verwendet wird, die keine funktionale Modifikation erfahren hat.

```
struct A
{
  A() {}
  explicit A(char c {}

  int get_x();
  int func();
}

struct B : public A
{
  using A::A; // get all constructors from A

  using A::func;
  int func(int); // could possibly mask A::func()

  private:
    using A::get_x;  // <-- get_x is now private
}
```

Für Klassen und Strukturen funktioniert das schon länger; seit C++17 funktioniert das übernehmen von Symbolen auch für (verschachtelte) Namensräume:

```
void f(){ }

namespace X
{
  void x() {};
  void y() {};
  void z() {};
}

namespace I::K::L
{
  using ::f; // f() is available in I::K::L now
  using X::x; // x() is available in I::K::L (dropped namespace X)
}
```

Weiterleiten von Konstruktoren
Andere High-Level Programmiersprachen kennen das "Verketten" von Konstruktoren schon länger und seit C++11 ist dies auch in endlich C++ möglich. Die Vorteile von weniger dupliziertem Code und damit einfacherer Lesbarkeit und somit bessere Wartbarkeit liegen dabei auf der Hand. Gerade bei Konstruktoren die intern komplizierte Initialisierungen und/oder Checks durchführen hilft dies sehr und fördert die Umsetzung des RAII (Resource Allocation is Initialisation) Paradigmas.

```
class DelegatingCtor
{
    int number_;
  public:
    DelegatingCtor(int n) : number_(n) {}
    DelegatingCtor() : DelegatingCtor(42) {};

}
```

Im Zusammenhang mit der Verwendung der oben genannten Konstruktorenvererbung mit `using` lässt sich Code so noch weiter komprimieren.

```
class Base
{
  public:
    Base(int x) : x_{x} {};
  private:
    int x_;
};

class Derived : public Base
{
  public:
    using Base::Base; // imports Base(int) as Derived(int)
    Derived(char) : Derived(123) {} // delegating ctor;
};
```

`= delete` - Löschen von Funktionen

Weniger Code heisst weniger Bugs, auch bei generiertem Code. Also erleichtern wir dem Compiler doch die Arbeit Code zu generieren, den wir gar nicht wollen und brauchen. Das Keyword `delete` für Funktionsdeklaration - nicht zu verwechseln mit dem entsprechenden Ausdruck um Objekte zu Löschen - ist eine weitere sehr starke Erweiterung in C++11, mit der ein Programmierer eine Absicht nicht nur Signalisieren sondern auch vom Compiler durchsetzen lassen kann. Mit der Verwendung von `= delete` kann explizit sichergestellt werden, das gewisse Operationen wie zum Beispiel Kopieren eines Objektes nicht vorgesehen und möglich sind. Natürlich sollte die "Rule of Five" auch beim Löschen von Funktionen beachtet werden.

```
struct NonCopyable {
  NonCopyable() = default;

  // disables copying the object through construction
  NonCopyable(const Dummy &) = delete;
  // disables copying the object through assignement
  NonCopyable &operator=(const Dummy &rhs) = delete;
};

struct NonDefaultConstructible {

// this struct can only be constructed through a move or copy
  NonDefaultConstructible() = delete;
};
```

Garantiertes verhindern von Kopien

Die garantiere Verhinderung von Kopien (engl. guaranteed copy elision) ist für den Programmierer meist unsichtbar, aber dahinter verbirgt sich grosses Potential für kleineren und saubereren Code. Diese Tilgung verhindert, dass unnötige Kopien von temporären Objekten erstellt werden, wenn sie unmittelbar nach dem erstellen einem Neuen Symbol zugewiesen werden. Einige Compiler, wie gcc unterstützen dies zwar schon länger, aber mit C++17 wurde das Auslassen von Kopien als garantiertes Verhalten in den Standard aufgenommen. Nebst dem Effekt, das so weniger Code generiert wird, lässt sie den Programmierer seine Absicht, dass ein Objekt nicht kopiert oder verschoben werden darf mit noch grösserer Konsequenz umzusetzen. Unter Verwendung mit dem oben genannten `= delete` lässt sich dies sehr deutlich ausdrücken.

```
class A {
public:
  A() = default;
  A(const A &) = delete;
  A(const A &&) = delete;
  A& operator=(const A&) = delete;
  A& operator=(A&&) = delete;
  ~A() = default;

};

// Without elision this is illegal, as it performs a copy/move of A which has
// deleted copy/move ctors
A f() { return A{}; }

int main() {

  // OK, because of copy elision. Copy/move constructing an anonymous A is not
  // neccessary
  A a = f();
}
```

Structured Bindings
Klassen und Strukturen sind nicht die einzige Möglichkeit, um das Handling von Daten zu strukturieren. Die Standardbibliothek stellt zudem eine ganze Menge Datencontainer für genau diese Zwecke zur Verfügung. Mit `std::tuple` und `std::array` wurden in C++11 zwei Datenstrukturen mit zur Compile-Time bekannter Grösse eingeführt. Während `std::array` eine relativ simple Modernisierung von C-Arrays darstellt wurde mit `std::tuple` wurde eine generische Möglichkeit geschaffen um heterogene Daten bequem im Programm herum zu reichen, ohne dass der Programmiere reine Datenklassen oder `structs` erstellen muss.
Seit C++17 ist der Zugriff auf die Inhalte dieser Datenstrukturen durch die Strukturierten Bindings sehr leichtgewichtig möglich:

```
auto tuple = std::make_tuple<1, 'a', 2.3>;

const auto [a, b, c] = tuple;
auto & [i,k,l] = tuple;
```

Zu beachten ist, dass alle Variablen hier dieselbe `const`-ness haben und entweder alle als Referenz oder By-Value gelesen werden. Die Structured Bindings funktionieren auch im Zusammenhang mit Klassen, allerdings ist dies etwas Problematisch, da die Semantik von Klassenmembers keine starke Reihenfolge der Member vorsieht. Es gibt Möglichkeiten diese Semantik zu reimplementieren, allerdings ist dies vergleichsweise aufwändig.

Stark typisierte Enums
Einer der wohl am häufigsten verwendeten Möglichkeiten für eigene Datentypen mit klaren Wertebereichen zu erstellen waren schon in C die `enums` und auch heute werden sie noch oft und gerne verwendet. Ein oft zitiertes dabei Ärgernis ist, dass die Typensicherheit bei der Verwendung von Enums nur ungenügend sichergestellt ist. So war es in der Vergangenheit Möglich ein Wert eines Enum-Typs einer Variable eines anderen Enum-Typs zuzuweisen. Mit den den neuen Standards gehört dies bei korrekter Verwendung der Vergangenheit an. Wird einer `enum` Definition das Keyword `class` oder `struct` hinzugefügt wird daraus ein stark typisierter Datentyp und eine Verwendung mit einem anderen `enum`-Typ führt je nach Konfiguration zu einer Warnung oder einem Fehler beim Kompilieren. Sozusagen als zusätzlicher Bonus kann seit C++11 auch der unterliegende Datentyp für ein enum explizit angegeben werden, was der Portabilität des Codes zu Gute kommt.

```
enum Color : uint8_t { Red, Green, Blue };
enum class Sound { Boing, Gloop, Crack };

auto s = Sound::Boing;
```

Zeit-Literale mit `<chrono>`

Eine sehr häufige Verwendung von Daten mit klaren, aber nicht immer linearen Wertebereiten ist insbesondere bei Applikationen mit strikten Zeitanforderungen natürlich die Zeit selbst. Das handling von Zeiteinheiten ist für viele Programmierer ein Albtraum. Die Gründe sind vielfältig, von der nicht-linearen Aufteilung von Sekunden, Minuten und Stunden bis hin, dass schnell mal Verwirrung entsteht um welche Zeiteinheit sich bei einem Aufruf wie `sleep(100)`. Handelt es sich hier um Sekunden? Millisekunden? Mit der Einführung von `std::chrono` in C++11 und dem Hinzufügen von Zeitliteralen wird das Handling um einiges einfacher. Mit den Literalen können Zeitangaben mit einem einfachen Suffix im Code mit einer fixierten Einheit beziehungsweise mit einer fixen Auflösung deklariert werden. `<chrono>` liefert dabei alles zwischen Mikrosekunden und Stunden. Durch die Verwendung der von `std::chrono` mitgelieferten Zeiteinheiten lassen sich Zeitwerte bereits zur Compile-Time konvertieren und das lästige manuelle Umrechnen zur Laufzeit gehört der Vergangenheit an.

```
using namespace std::chrono_literals;

auto seconds = 10s;
auto very_small = 1us;

// automatic, compile-time conversion
if(very_small < seconds)
{
...
}
```

Verzweigungen mit Initialisierung

Auf den ersten Blick ist die Einführung direkten Initialisierung in `if`- und `switch`-Statements in C++17 eine Möglichkeit Code noch ein kleines bisschen kompakter schreiben. Ein weiter etwas versteckter Vorteil ist, dass der Programmierer seine Absicht, dass ein Symbol nur innerhalb einer Verzweigung verwendet wird deutlicher ausdrücken kann. Die Initialisierung direkt neben beziehungsweise. in der Bedingung zu haben verhindert auch die Gefahr, dass sie bei Refactorings (unabsichtlich) von der Verzweigung getrennt wird.

```
if(int i = std::rand(); i % 2 == 0)
{ }

switch(int i = std::rand(); i = %3)
{
  case 0:
    ...
  case 1:
    ...
  case 2:
    ...
}
```

Im Zusammenhang mit den oben genannten Structured Bindings kann die direkte Initialisierung sehr elegant verwendet werden. Im folgenden Beispiel wird versucht ein bereits existierender Wert in einer `std::map` zu überschreiben. Der Rückgabewert von `insert` wird direkt in einen Iterator und das Flag, ob die Operation erfolgreich war entpackt und kann somit direkt innerhalb der Abfrage verwendet werden.

```
std::map<char, int> map;
map['a'] = 123;

if(auto [it, inserted] = map.insert({'a', 1000}); !inserted)
{
  std::cout << "'a' already exists with value " << it->second << "\n";
}
```

Standardattribute

Wann immer ein Programmierer eine Annahme trifft, sollte dies im Code dokumentiert sein. Mit den Standardattributen können einige solcher Annahmen mit wenig Aufwand dokumentiert werden. Attribute sind seit längerem für verschiedene Compiler bekannt, allerdings war die Notation für die verschiedenen Compiler oft unterschiedlich. Seit C++17 wurde diese als `[[ attribute ]]` standardisiert was portablen den Code lesbarer macht. Zudem wurden verschiedene von allen Compilern unterstützte Standardattribute eingefügt, welche es dem Programmierer erlauben seine Absichten für gewisse Konstrukte explizit zu formulieren

`[[noreturn]]`	Zeigt an, dass eine Funktion nicht zurückkehrt, z.B. weil sie immer eine Exception wirft
`[[deprecated]]` `[[deprecated("reason")]]`	Zeigt an, dass die Verwendung dieser Klasse, Funktion oder Variable zwar erlaubt, aber nicht mehr empfohlen ist
`[[fallthrough]]`	Verwendet in switch-Statement um anzuzeigen, dass ein case:-block mit Absicht kein break beinhaltet
`[[nodiscard]]`	Produziert eine Compiler-Warnung, falls ein so markierter Rückgabewert nicht verwendet wird
`[[maybe_unused]]`	Unterdrückt Compiler-Warnungen bei nicht verwendeten Variablen. z.B. in Debug-Code <

Fazit
Diese 10 kleinen Features und Funktionen sind natürlich nur ein kleiner Teil davon, was modernes C++ ausmacht. Aber durch deren konsequente Anwendung kann Code mit relativ wenig Aufwand lesbarer und einfacher Verständlich gemacht werden, ohne dass die Komplette Struktur einer existierenden Codebase gleich umgeschrieben werden muss.

Zusammenfassung
Die Einführung der neuen Standards C++11/14/17 hat C++ merklich modernisiert. Nebst solchen großen Sprachfeatures wie smart-pointers, move semantics und varaidic templates gibt es auch noch eine ganze Menge an kleineren Erweiterungen, die oftmals unter dem Radar fliegen. Aber gerade diese Features können helfen, C++ Code merklich zu vereinfachen und wartbarer zu machen. Dies gekoppelt mit neuen Features in der STL kann helfen, viele kleine Fehlerchen schon beim Schreiben des Codes zu verhindern. Dass der Code sich dabei auch noch leichter liest und stabiler wird, sind weitere erfreuliche Nebeneffekte.
10 dieser kleinen aber feinen Features werden hier aufgezeigt und etwas genauer unter die Lupe genommen.

Autor
Dominik Berner ist ein Senior Software-Ingenieur bei der bbv Software Services AG mit einer Leidenschaft für modernes C++. Die Wartbarkeit von Code ist für ihn kein Nebeneffekt, sondern ein primäres Qualitätsmerkmal, das für die Entwicklung von langlebiger Software unabdingbar ist.
Als blogger und speaker auf Konferenzen und meetups weiß er, wie Inhalte zu verpacken sind, damit für das Publikum ein Mehrwert entsteht.