FH Jülich

Seminararbeit

C++11: Einführung in den neuen Standard

Author:
Samir Benmendil

Betreuer:
Prof. Dr. Christof Schelthoff
Dipl. Ing. Jan-Simon Schmidt

16. Dezember 2011

Inhaltsverzeichnis

1 Einleitung

C++ ist eine statisch getypte, objektorientierte Sprache, die in 1979 von Bjarne Stroustup entwickelt wurde. Sie wird oft als veraltet bezeichnet, denn sie beruht auf C, welches zwischen 1969 und 1973 von Dennis Ritchie entwickelt wurde, und garantiert volle Kompatibilität mit C.

In dem neuen Standard, der hier vorgestellt wird, wurden verschiedene Änderungen und Erweiterungen implementiert, die C++ wieder zu einer modernen Sprache machen.

2 Verbesserungen am Sprachkern

2.1 Rvalue-Referenzen und Move-Konstruktoren

In vorherigen Versionen von C++ wurden Rvalues als unveränderbar angenommen, es war jedoch in manchen Fällen möglich und auch nützlich, diese Werte zu verändern.

C++11 definiert einen neuen non-const Referenztypen, welcher auf temporäre Rvalues verweist, die nach der Initialisierung noch verändert werden dürfen, um das Verschieben von Objekten zu erlauben.

Ein großes Problem bei C++03 waren die teuren und unnötigen deepcopies, die implizit erzeugt werden, wenn Objekte by value übergeben werden.

Listing 1: Copy- und Move-Konstruktor

```
class Foo
{
   Bar *b;
public:
   // deep-copy
   Foo( const Foo & other )
      : b( other.b )
   {
      if ( b ) b->ref();
   }
   Foo & operator=( const Foo & other ) {
      if ( other.b ) other.b->ref();
      if ( b ) b->unref();
      b = other.b;
      return *this;
   }
   // move
   Foo( Foo && other )
      : b( other.b )
   {
      other.b = 0;
   }
   Foo & operator=( Foo && other ) {
      std::swap( b, other.b );
      return *this;
   }
};
```

In `C++11` gibt es einen Move-Konstruktor, der eine Rvalue-Referenz als Parameter annimmt. Dieser gibt eine Kopie des Pointers zu dem Objekt zurück und setzt den alten auf `null`.

2.2 `constexpr`

Konstante Ausdrücke konnten schon seit langem in `C++` optimiert werden. Ausdrücke wie z.B `3+4` konnten schon während des Kompilierens berechnet werden. Diese Optimierung endete aber, sobald eine Funktion im Spiel war.

Listing 2: Array-Initialisierung

```
int sum(int a, int b) { return a + b; }
int = array[sum(3,4)];  // Nicht möglich
```

Listing 2 ist nicht möglich, da eine array-Initialisierung durch einen Konstanten Wert erfolgen muss; jedoch kann der Compiler die Funktion `sum()`

nicht als konstant betrachten, da er nicht vorhersagen kann, dass diese sich während der Laufzeit nicht ändert.

Im neuen Standard gibt es die Möglichkeit, Funktionen mit konstanten Rückgabewerten zu definieren. Dem Compiler wird durch das Schlüsselwort `constexpr` angedeutet, dass der Rückgabewert sich während der Laufzeit nicht ändert. Auflistung 2 kann wie folgt umgeschrieben werden:

Listing 3: `constexpr` Array-Initialisierung

```
constexpr int sum(int a, int b) { return a + b; }
int = array[sum(3,4)]; // Erstellt ein array der Größe 7
```

Konstruktoren können auch als `constexpr` deklariert werden, wenn sie ausschließlich aus einer konstanten Initialisierungsliste bestehen, deren Ausdrücke ebenfalls `constexpr` sind.

Listing 4: `constexpr` Konstruktor

```
class Complex
{
 double m_a, m_b;
public:
 constexpr Complex() : m_a{0}, m_b{0} {}
 constexpr Complex( double a, double b ) : m_a{a}, m_b{b} {}
 constexpr double a() const { return m_a; }
 constexpr double b() const { return m_b; }
};
class Polar
{
 double m_r, m_phi;
public:
 constexpr Polar() : m_r{0}, m_phi{0} {}
 constexpr Polar( double r, double phi)
   : m_r{r}, m_phu{phi} {}
 constexpt Polar( const Complex & c )
   // wenn sqrt() und atan2() ebenfalls constexpr sind
   : m_r{ sqrt(c.a() * c.a() + c.b() * c.b() ) }
   , m_phi{ atan2( c.a(), c.b() ) } {}
};
constexpr Polar pols[] =
{
 Polar{ 0, 0, 100, 100 },
 Polar{ Complex{ 0, -1 } }
};
```

In Beispiel 4 wird `pols` beim Kompilieren berechnet und der Linker legt die Variable dann im Read-only-Speicher ab.

`constexpr` können die Laufzeit eines Programms deutlich verringern, erhöhen aber die Übersetzungszeit. Genau so wie `const` in der Vorgängerversion, sollten `constexpr` überall da verwendet werden, wo es möglich ist.

2.3 Änderung der Definition von PODs

Ein POD `struct`[1] ist eine Klasse, die sowohl trivial ist als auch ein Standardlayout hat.

Triviale Klassen:

1. besitzen keine nicht-trivialen Konstruktoren.
2. besitzen keine nicht-trivialen Copy- und Move-Konstruktoren.
3. besitzen keine nicht-trivialen Copy- und Move-Zuweisungsoperatoren.
4. besitzen einen trivialen, nicht-virtuellen Destruktor.

Eine Klasse hat ein Standard-Layout, wenn:

1. sie weder virtuelle Funktionen noch virtuelle Basisklassen hat.
2. keine nicht-statischen Attribute eines nicht-Standardlayout-Klassentypen hat.
3. alle Attribute die gleiche Zugriffskontrolle haben (`public`, `private`, `protected`).

2.4 Externe Templates

Jedes Mal, wenn dem Compiler in `C++03` ein Template begegnet, muss dieses instanziiert werden. Wenn das Template in mehreren Dateien mit den gleichen Typen instanziiert wird, kann das die Übersetzungszeit drastisch erhöhen. Mit `extern` kann dem Compiler mitgeteilt werden, dass das Template in dieser Übersetzungseinheit nicht instanziiert werden soll.

```
// Template muss instanziiert werden.
template class std::vector<MyClass>;

// Template wird hier nicht instanziiert.
extern template class std::vector<MyClass>;
```

[1] POD steht für „plain old data“

2.5 Initialisierungslisten

C++03 hat die Initialisierungslisten von C geerbt. Man kann ein einfaches Objekt durch eine Liste von Argumenten innerhalb geschweifter Klammern initialisieren. Dies war jedoch nur für Klassen möglich, die die Charakteristik eines POD-`struct` haben.

Es ist in C++11 möglich, Initialisierungslisten für beliebige Klassen zu benutzen, indem man einen Konstruktor definiert, dessen Parametertyp eine `std::initializer_list` ist. Es ist ebenfalls möglich, diese Erweiterung bei Funktionen zu verwenden.

Listing 5: `std::initializer_list`

```
class Foo
{
public:
    Foo(std::initializer_list<int>);
    void foo_function(std:initializer_list<string>);
}

Foo foo = { 1, 1, 2, 3, 5, 8};
Foo foo2 {13, 21, 34, 55};
foo_function({"Hallo", "Welt", "!"};
```

Eine `std::initializer_list` kann nur statisch während der Übersetzung erstellt werden, deren Inhalt ist daher konstant.

2.6 Uniforme Initialisierung

C++03 hatte verschiedene Probleme mit Objektinitialisierungen. Ein Konstruktoraufruf sieht genau so aus, wie ein Funktionsaufruf und es konnten nur Aggregate und POD-Typen mit Initialisierungslisten initialisiert werden.

C++11 erweitert die Initialisierungssyntax so, dass Variabelinitialisierungen uniform werden.

Listing 6: Uniforme Initialisierung

```
struct Foo
{
    int a;
    double b;
};

struct Bar
{
    Bar(int a, double b) : m_a{a}, m_b{b} { };

    Foo get_foo() { return { a, b }; }

private:
    int m_a;
    double m_b;
};

// eine initializer_list wird hier implizit benutzt.
Foo foo{42, 3.14};
// Konstruktor wird aufgerufen.
Bar bar{21, 6.28};
// Array-Initialisierung mit new
int * a = new int[3] {2, 3, 6};
```

`Foo` ist ein POD-struct, deshalb wird `foo` mit einer Initialisierungsliste erstellt. Bei `bar` hingegen wird der Konstruktor aufgerufen. Die Funktion `get_foo` gibt ein Objekt `Foo` zurück, ohne dass der Typ des Objektes explizit angegeben wird.

Traditionelle Konstruktorsyntax ist jedoch in manchen Fällen noch erforderlich. Wenn eine Klasse ebenfalls einen `initializer_list`-Konstruktor hat, wird dieser bevorzugt, wenn die Variable mit geschweiften Klammern initialisiert wird.

2.7 Typinferenz

In vorherigen Versionen von `C++` musste der Typ einer Variable explizit festgelegt werden. Jedoch wurde das zu einem großen Problem, wenn man mit Templates gearbeitet hatte. Der Rückgabewert von Templates ist nicht immer leicht Ausdrückbar wie im Listing 7 zu sehen ist. `unknown_type` muss vom Typ `T+S` sein, jedoch ist dieser nicht bekannt.

Listing 7: Typinferenz

```
template <class T, class S>
unknown_type sum(const T & t, const S & s) { return s+t; }
```

Der neue Standard ermöglicht es, mit `auto` den Typ einer Variablen automatisch zu erkennen; mit `decltype` kann der Typ eines Ausdrucks während der Übersetzung bestimmt werden.

Listing 8: `auto`, `decltype`

```
auto foo = 5; // foo wird zum Integer
decltype(foo) bar = 6; // bar wird zum selben Typ wie foo
```

Um diese Erweiterungen bei Templates anzuwenden, muss die Syntax etwas verändert werden:

Listing 9: `auto`, `decltype`

```
template <class T, class S>
auto sum(const T & t, const S & s)
  -> decltype (s+t) { return s+t; }
```

2.8 Bereichsbasierte For-Schleifen

`C++11` führt die `foreach`-Schleife in die Sprache ein. Es ist jetzt möglich, eine einfache Schleife zu schreiben, die über jedes Element einer Liste iteriert. Diese Schleife funktioniert bei `C`-arrays, Initialisierungslisten und jedem Typ, der `begin()` und `end()` Funktionen hat, die einen Iterator zurückgeben.

Listing 10: Range-based for

```
int foo[5] = { 1, 2, 3, 4, 5 };
for (int &f : foo) {
    f += 10;
}
```

2.9 Lambda-Funktionen

Lambda-Funktionen, auch anonyme Funktionen genannt, sind eine neue Sprachfunktion in `C++`. Diese Funktionen werden wie folgt geschrieben:

Listing 11: Lambda-Funktionen

```
[] (int a, int b) { return a + b; }
[] (int a, int b) -> int { return a + b; }
```

Wenn der Rückgabetyp für jeden Aufruf der Funktion der gleiche bleibt, kann er weggelassen werden und wird mit `decltype(a+b)` automatisch erkannt.

Lambda-Funktionen können Variablen verändern, die außerhalb des Funktionsblocks initialisiert wurden. Diese Variablen werden in den eckigen Klammern vor der Funktion definiert. Dies ermöglicht es, die Variablen entweder by value oder by reference zu übergeben. Folgende Tabelle zeigt die verschiedenen Methoden, um Variablen an die Lambda-Funktion zu übergeben.

Tabelle 1: Lambda-Funktionen

`[]`	keine Variablen werden übergeben
`[a]`	`a` wird by value übergeben
`[&a]`	`a` wird by reference übergeben
`[=]`	alle Variablen werden by value übergeben
`[&]`	alle Variablen werden by reference übergeben
`[a, &b]`	`a` wird by value übergeben, `b` wird by reference übergeben

Listing 12 zeigt ein Beispiel von Lambda-Funktionen im Einsatz. Der Maximalwert von `list` wird berechnet.

Listing 12: Lambda-Funktionen

```
std::vector<int> list;
int max = 0;
std::for_each(list.begin(), list.end(), [&max](int x) {
    max = x > max ? x : max;
});
```

2.10 Objekterstellung

`C++03` erlaubte es nicht, dass Konstruktoren andere Konstruktoren der selben Klasse aufrufen. Jeder Konstruktor musste daher alle Klassenattribute selber initialisieren oder eine gemeinsame Funktion aufrufen. Ebenfalls konnten abgeleitete Klassen nicht mit Konstruktoren der Vaterklasse initialisiert werden, auch wenn diese gereicht hätten.

Im neuen Standard wurden diese Probleme gelöst. Auflistung 13 zeigt wie Delegation implementiert wurde.

Listing 13: Delegation

```
class Base
{
    int m_a;
public:
    Base(int a) : m_a{a} {}
    Base() : Base{42} {}
}

class Derived : public Base
{
    int m_b = 1;
public:
    using Base::Base;
}

Base base; // base hat m_a = 42
// derived benutzt den Konstruktor den er von Base geerbt hat
Derived derived{12};
```

Der Konstruktor `Base()` ruft erst den Konstruktor `Base(int a)` mit `a = 42` auf, bevor der Rumpf des Konstruktors ausgeführt wird. Die Klasse `Derived` definiert keine eigene Konstruktoren, sondern benutzt die, die sie von `Base` geerbt hat. Es können entweder alle Konstruktoren des Vaters benutzt werden oder keine.

Es ist mittlerweile auch möglich, Attribute dort zu initialisieren, wo sie deklariert wurden: `m_b` wird also mit `1` initialisiert, solange es nicht von einem Konstruktor überschrieben wird.

2.11 Explizite Überladung

In `C++03` war es möglich, versehentlich eine neue virtuelle Funktion zu erstellen, obwohl man eigentlich eine Funktion der geerbten Klasse überladen wollte. Zum Beispiel:

Listing 14: Virtuelle Funktion

```
class Base
{
    virtual void foo(int);
};
class Derived : Base
{
    virtual void foo(double);
}
```

Die `Derived::foo` Funktion sollte eigentlich die Version der Basisklasse ersetzen, da jedoch die Funktionsdeklarationen nicht genau übereinstimmen, wird eine weitere virtuelle Funktion erstellt. Dieses Problem kommt häufig dann vor, wenn ein Programmierer die Deklaration der Basisklassenfunktion verändert.

Die Syntax von `C++11` ermöglicht es, dieses Problem bei der Kompilation zu erkennen, indem `override` hinter der Funktionsdeklaration eingesetzt wird. Dies teilt dem Compiler mit, dass die Funktion eine andere überladen muss. Wenn der Compiler keine Funktion mit dieser Deklaration finden kann, bricht er ab und informiert den Benutzer dass ein Fehler aufgetreten ist.

Man will auch manchmal, dass eine Klasse oder Funktion nicht vererbt bzw. überladen wird. Mit `final` ist das im neuen Standard möglich.

Listing 15: `override`

```
class Final final {};
// Fehler, Final darf nicht vererbt werden.
class Derived : Final {};

class Base
{
    virtual void foo(int);
    virtual void bar() final;
}
class Derived : Base
{
    // Fehler, foo(double) gibt es nicht in Base
    virtual void foo(double) override;
    // Fehler, bar() wurde in Base als final deklariert
    virtual void bar();
}
```

3 Funktionalitätserweiterungen

3.1 Variadische Templates

Ab `C++11` können Klassen- und Funktions-Templates eine beliebige Anzahl an Parametern annehmen.

3.2 `default` und `delete` Memberfunktionen

Der `C++` Compiler erstellt für jede Klasse ein Default-Konstruktor, ein Copy-Konstruktor, ein Zuweisungsoperator (`operator=`) und einen Destruktor, wenn diese nicht deklariert wurden. `C++11` erlaubt es explizit festzulegen, dass die Default-Implementationen benutzt werden. Da der Compiler keinen Default-Konstruktor erstellt, wenn irgend ein Konstruktor deklariert wurde ist es zum Vorteil dass `C++11` es erlaubt die Default-Implementationen des Konstruktors zu deklarieren.

Listing 16: `default`

```
struct Foo
{
    // Default-Konstruktor wird explizit deklariert.
    Foo() = default;
    Foo(int);
};
```

Es ist jetzt auch erlaubt, Konstruktoren oder Funktionen zu löschen, wenn diese nicht benutzt werden sollen. Zum Vorteil wird diese Erweiterung bei Klassen, die nicht kopierbar sind, wie auch für das Singleton-Pattern. In Listing 17 werden der Zuweisungsoperator und der Copy-Konstruktor deaktiviert. Die Klasse Bar ist hiermit nicht kopierbar. Die Funktion `f(int)` wird ebenfalls deaktiviert, `f()` kann also nicht mit einem `int` aufgerufen werden. Implizites casting in ein `double` wird für diese Funktion nicht mehr unterstützt.

Listing 17: `delete`

```
struct Bar
{
    Bar & operator=(const Bar &) = delete;
    Bar(const Bar &) = delete;
    Bar() = default;

    int f(int) = delete;
    int f(double);
};
```

3.3 Statische Zusicherungen

Assertions wurden in `C++03` entweder zur Laufzeit mit `assert()` oder vom Präprozessor mit `#error` getestet. Diese Zusicherungen sind aber nicht für Templateparameter geeignet, da diese vor dem Kompilieren noch nicht zur Verfügung stehen. Die neuen `static_assert`-Makros testen während dem Übersetzen.

Listing 18: `static_assert`

```
static_assert ( expression, fehler_meldung );
```

3.4 Garbage Collection

Der neue Standard erlaubt Implementationen mit automatischer Speicherbereinigung. Es ist also den Entwicklern von Compilern freigestellt, ob sie einen Garbage-Collector unterstützen oder nicht.

4 C++ Standardbibliothek

4.1 Tupel

Dank variadischer Templates ist es möglich, Tupel in `C++` zu definieren.

Listing 19: `tuple`

```
typedef std::tuple<std::string, int> tuple_type;
tuple_type foo("Meaning of Life", 42);

int n = std::get<1>(foo); // n = 42
std::get<0>(foo) = "molybdenum";
```

4.2 Hashtabellen

Mit Hashtabellen wurde ein lang erwartetes Feature in die Standard Template Library implementiert. Kollisionen werden in dieser Version nur mit Verkettung gelöst. Um Kompatibilität mit nicht-Standardbibliotheken zu vermeiden, wurde `unordered` statt `hash` verwendet. Es gibt vier verschiedene Sorten von Hashtabellen:

1. `std::unordered_set` hat keine Duplikate.
2. `std::unordered_multiset` erlaubt Duplikate.
3. `std::unordered_map`
4. `std::unordered_multimap`

4.3 Reguläre Ausdrücke

Reguläre Ausdrücke werden ab sofort direkt von der Standard Bibliothek unterstützt. Dazu wird erstmals ein Objekt des Typs `std::regex` erstellt, mit dem man innerhalb von `strings` und `C-arrays` suchen und ersetzen kann. Das Ergebnis der Suche wird in einem `std::smatch` oder `std::cmatch` gespeichert, abhängend davon, ob man ein `string` oder ein `const char *` haben will.

Listing 20: `regex`

```
// Regex
const char *reg_esp = "[ ,.\\t\\n;:]";

std::regex rgx(reg_esp);
std::cmatch match;

const char *target = "Unseen University - Ankh-Morpork";

// Sucht alle Charakter, die durch 'resp_esp' getrennt wurden.
if( std::regex_search( target, match, rgx ) ) {
    // Wenn welche gefunden worden.

    const size_t n = match.size();
    for( size_t a = 0; a < n; a++ ) {
        std::string str( match[a].first, match[a].second );
        std::cout << str << "\n";
    }
}
```

4.4 Smart Pointers

C++11 führt drei verschiedene Smart Pointer ein.

1. `unique_ptr` können nicht einer anderen Variablen zugewiesen werden. Deren Copy-Konstruktor und Zuweisungsoperator wurden explizit deaktiviert. Es ist jedoch möglich, diese Zeiger mittels `std::move()` auf eine andere Variable zu verschieben.

Listing 21: `unique_ptr`

```
std::unique_ptr<int> foo(new int(42));
std::unique_ptr<int> bar = foo; // Übersetzungsfehler
// foo2 zeigt auf den Speicher von foo, foo wird ungültig.
std::unique_ptr<int> foo2 = std::move(foo);

foo2.reset(); // Speicher wird befreit.
```

2. `shared_ptr` werden zur Referenzzählung benutzt. Jede Kopie eines `shared_ptr` besitzt den gleichen Pointer. Dieser Verweis wird nur dann gelöscht, wenn alle Instanzen des `shared_ptr` zerstört wurden.

Listing 22: `shared_ptr`

```
std::shared_ptr<int> foo(new int(42));
std::shared_ptr<int> foo2(new int(42));
int cnt = foo.use_count(); // cnt = 2

delete foo2;     // Speicher ist noch vorhanden
                 // foo2 ist ungültig
delete foo;      // Objekt wird zerstört
```

3. `weak_ptr` vermeiden zyklische Verweise, indem sie die gleichen Eigenschaften besitzen, wie `shared_ptr`, aber ohne den Referenzzähler zu erhöhen. Ein Objekt wird gelöscht, wenn es nur durch `weak_ptr` referenziert wird.

Listing 23: `weak_ptr`

```
std::shared_ptr<int> foo(new int(42));
std::weak_ptr<int> bar(new int(42));

delete foo;      // Speicher wird freigegeben
```

4.5 Zufallszahlengenerator

Die `C` Standardbibliothek bietet die Möglichkeit, Pseudozufallszahlen mit `rand` zu generieren, jedoch wurde dem Compiler überlassen, wie diese Zahlen zu berechnen sind. `C++` hat diese Funktionalität von `C` geerbt, `C++11` wird neue Methoden zur Generierung von Pseudozufallszahlen zur Verfügung stellen.

Diese Funktionalität wird in zwei geteilt: es gibt auf der einen Seite eine Generatorengine, die den Zustand des Zahlengenerators beinhaltet und Pseudozufallszahlen ausgibt, auf der anderen Seite wird eine mathematische Distribution festgelegt, die den Wertebereich der Zufallszahlen bestimmt.

Folgende Distributionen werden von `C++11` unterstützt:

1. `uniform_int_distribution`
2. `bernoulli_distribution`
3. `geometric_distribution`
4. `poisson_distribution`
5. `binomial_distribution`
6. `uniform_real_distribution`
7. `exponential_distribution`
8. `normal_distribution`
9. `gamma_distribution`

4.6 Type traits für Metaprogrammierung

Metaprogrammierung wird in `C++` häufig benutzt. Es wird dabei ein Template erstellt, das Werte während der Übersetzung und nicht während der Laufzeit berechnet. Dadurch wird das Programm um einiges beschleunigt, was aber mit einem Zeitverlust während des Kompilierens erkauft wird.

Type Traits können Eigenschaften von Objekten erkennen. Es ist dann möglich, mit dieser Information auf verschiedene Typen anders zu reagieren.

Listing 24 zeigt, wie man zwischen zwei Funktionen, basierend auf dem Parametertyp, wählt.

Listing 24: Type Traits

```
// Erste Funktion.
template< bool B > struct Algorithm {
  template<class T1, class T2> static int foo (T1 &, T2 &)
  { /*...*/ }
};

// Zweite Funktion.
template<> struct Algorithm<true> {
  template<class T1, class T2> static int foo (T1, T2)
  { /*...*/ }
};

// Wenn 'foobar' instanziiert wird, wird automatisch die
// richtige Funktion aufgerufen
template<class T1, class T2>
int foobar (T1 A, T2 B)
{
  // Benutzt die 2. Funktion nur wenn 'T1' und 'T2' beide
  // Integer sind, ansonsten wird die 1. Funktion ausgeführt.
  return Algorithm<std::is_integral<T1>::value
                && std::is_integral<T2>::value>::foo( A, B );
};
```

Listings

Literatur

[1] Bjarne Stroustrup, *The C++ Programming Language*. Addison Wesley, Massachusetts, Special Ed., 2010.

[2] [ISO N3242] *Working Draft, Standard for Programming Language C++*, 2011-02-28

[3] `http://en.wikipedia.org/wiki/C++11`

[4] `http://www.heise.de/developer/artikel/C-11-auch-ein-Stimmungsbild-1345406.html`