Advanced Software Development

Allgemeine Neuerungen in C++ 11 und C++ 14

Überblick

- Variablendeklaration mit auto
- Erweiterte for-Schleife
- Rvalue Referenzen
- STL-Erweiterungen
  - Erweiterungen bestehender Container
  - Neue Container und Algorithmen
- Verschiedene Neuerungen

Variablendeklaration mit auto

- Typinferenz: Variable bekommt den Typ des zugewiesenen Ausdruck
- Beispiele:

```
auto x = 5;       // int
auto y = x + 1;   // int

vector<int> v;
…
for (auto iter = v.begin(); iter != v.end(); ++iter) {
  …
}

auto &firstElem = x[0];  // int &
```

Return Type Deduction

- Typinferenz für Funktionsrückgabewert
- Beispiele:

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) {
  return x + y;
}

template <typename TCont>
auto maxElement(TCont const &cont) {
  return *max_element(cont.begin(), cont.end());
}
```

Erweiterte for-Schleife (1)

- Iteration über alle Elemente eines Containers

```
vector<int> v;
…
for (int x : v) {
  cout << x << endl;
}
for (int &x : v) {
  x++;
}
```

Erweiterte for-Schleife (2)

- Funktioniert mit allen Containern, die begin und end Methoden haben
- Alternativ mit globalen begin/end Funktionen

```
int array[10];
…
for (int x : array) {
  ...
}
// gleichbedeutend mit:
for (auto p = begin(x); p != end(x); ++p) {
  int x = *p;
  ...
}
```

Rvalue Referenzen

- Neue Art von Referenzen
- Ermöglichen „Move“-Semantik und helfen damit, unnötige Kopien von Objekten zu vermeiden
- Ermöglichen „Perfect Forwarding“

Lvalue vs. Rvalue Referenzen

- Lvalue-Referenz erlaubt nur die Zuweisung von „lvalues“:

```
string str;
string &s1 = str;         // OK (lvalue)
string &s2 = str + " ";   // nicht erlaubt (rvalue)
```

- Rvalue-Referenz erlaubt nur die Zuweisung von „rvalues“:

```
string str;
string &&s1 = str;        // nicht erlaubt (lvalue)
string &&s2 = str + " ";  // OK (rvalue)
```

Move-Semantik: Motivation

```
class SomeClass {
public:
  SomeClass(std::string const &str) : mStr(str) {}
  …
private:
  std::string mStr;
};

…

string s1 = "abc";
SomeClass obj1(s1);       // Kopie von s1
SomeClass obj2("xyz");    // Kopie eines temporären string
SomeClass obj3(s1 + " "); // Kopie eines temporären string
```

Move-Semantik: Implementierung

```
class SomeClass {
public:
  SomeClass(string const &str) : mStr(str) {}
  SomeClass(string &&str) : mStr(std::move(str)) {}
  …
private:
  std::string mStr;
};

…

string s1 = "abc";
SomeClass obj1(s1);       // Kopie von s1
SomeClass obj2("xyz");    // Move-Semantik
SomeClass obj3(s1 + " "); // Move-Semantik
SomeClass obj4(move(s1)); // Move-Semantik, s1 ist danach leer
```

Move Constructor

```
class DynamicObject {
public:
  DynamicObject() : mObject(new Object) {}
  ~DynamicObject() { delete mObject; }

  // Copy Constructor
  DynamicObject(DynamicObject const &other)
    : mObject(new Object(*other.mObject)) {}

  // Move Constructor
  DynamicObject(DynamicObject &&other) : mObject(other.mObject) {
    other.mObject = 0;
  }

private:
  Object *mObject;
};
```

Perfect Forwarding (1)

- Ermöglicht die korrekte generische Weitergabe von Parametern
- Beispiel:

```
template <typename T>
void someFunction(T const &x) {
  …
}

template <typename T>
void someFunction(T &&x) {
  …
}

// ges.: Funktion otherFunction, die die richtige Variante von
// someFunction je nach übergebenem Parameter aufruft
```

Perfect Forwarding (2)

```
template <typename T>
void otherFunction(T &&x) {
  …
  someFunction(std::forward<T>(x));
  …
}

string str;
otherFunction(str);          // Copy-Variante
otherFunction(str + " ");    // Move-Variante
otherFunction(move(str));    // Move-Variante
```

Rückgabe von Rvalue-Referenzen

```
string &&makestr1() {
  string s;
  …
  return move(s);      // Referenz auf lokales Objekt!
}

string makestr2() {
  string s;
  …
  return move(s);      // Aufruf Move-Konstruktor
}

…

string s = makestr2(); // Aufruf Move-Konstruktor
```

Return Value Optimization

- Erlaubt dem C++-Compiler, den Aufruf des Copy Constructors bei return zu unterbinden
- Bereits im vorigen C++-Standard vorhanden
- Beispiel:

```
string makestr3() {
  // versteckter Parameter: Adresse des Rückgabe-Objekts
  string s;   // Konstruktion an der richtigen Adresse
  …
  return s;   // nichts zu tun
}

string s = makestr3();  // Übergabe der Adresse von s
```

Rvalue Referenzen in der STL

- Alle STL Container sind „movable“
- Methoden zum Hinzufügen von Elementen (z.B. push_back) haben Move-Semantik für rvalue-Parameter
- Varianten des String-Operator + mit Move-Semantik für einen der beiden Parameter
- Algorithmen machen von der Move-Semantik Gebrauch (z.B. swap, sort, …)

STL-Container Erweiterungen (1)

- Verwendung von Rvalue Referenzen
- Methoden cbegin(), cend(), crbegin(), crend(): liefern immer einen const_iterator bzw. const_reverse_iterator

```
vector<int> v;
…
for (auto iter = v.cbegin(); iter != v.cend(); ++iter) {
  …
}
```

STL-Container Erweiterungen (2)

- emplace-Methoden: In-Place Konstruktion von Objekten direkt im Container

```
class Person {
public:
  Person(string const &name, int age);
};

vector<Person> persons;
persons.emplace_back("Anton", 50);
```

- Varianten je nach Container: emplace_back, emplace_front, emplace

Neue Container

- std::array (Array fixer Größe)
- std::tuple (Tupel)
- std::forward_list (einfach verkettete Liste)
- std::unordered_set / unordered_map / unordered_multiset / unordered_multimap (Hash-Container)

std::array (<array>)

- Wrapper um Arrays fixer Größe
- Verwendung ähnlich wie STL-Container
- Beispiel:

```
array<int, 5> arr1;  // nicht initialisiert!
array<int, 4> arr2 = {1, 2, 3, 4};
fill(arr1.begin(), arr1.end(), 0);
// oder: fill_n(arr1.begin(), arr1.size(), 0);
arr1[3] = 1;
for (auto x : arr1) {
  …
}
```

std::tuple (<tuple>)

- Verallgemeinerung eines std::pair, erlaubt beliebige Anzahl von Werten
- Beispiel:

```
tuple<int, int, double> t(3, 4, 0.5);
int x = get<0>(t) + get<1>(t);
double d = get<2>(t);
…
t = make_tuple(2, 3, 0.1);
```

Hash-Container (1)

- unordered_map, unordered_multimap, unordered_set, unordered_multiset
- Header <unordered_map> bzw. <unordered_set>
- Generische Hashtabelle
- Verwendung ähnlich zu std::map etc.

Hash-Container (2)

- Wesentliche Unterschiede zu std::map
  - Keine definierte Reihenfolge
  - Für eigene Typen Hash-Funktion nötig
  - Verwendet Operator ==, nicht < zum Vergleich

Neue Algorithmen (Auszug)

- all_of, any_of, none_of
- copy_if
- copy_n
- is_sorted, is_sorted_until
- minmax, minmax_element
- move

Kleinere Neuerungen

- >> bei Template-Deklarationen erlaubt

```
vector<vector<int>> v;  // statt vector<vector<int> >
```

- nullptr für Zeiger

```
Object *pObj = nullptr;  // statt Object *pObj = 0;
```

- Enum-Klassen

```
enum class Color { Red, Green, Blue };
Color c = Color::Red;

// mit expliziter Typangabe
enum class Color : char { Red, Green, Blue };
```

Kleinere Neuerungen

- Gruppierung von Ziffern

```
int million = 1'000'000;
```

- Binäre Literale

```
int binary = 0b1010;
binary = 0b1010'0010'0100'1100;
```

- Raw Strings

```
string r1 = R"(He said "Hello")";
string r2 = R"(C:\Windows)";
string r3 = R".(a raw string containing ")").";
```

Member Initialisierung

```
class MyClass {
public:
  MyClass() {}
  MyClass(int x, int y) : x(x), y(y) {}

private:
  int x = 0;
  int y = 0;
  bool b = false;
};
```

Konstruktor Delegation/Vererbung

- Konstruktor Delegation

```
class MyClass {
public:
  MyClass(string const &name, int value = 0);
  MyClass(int value) : MyClass("", value) {}
};
```

- Konstruktor Vererbung

```
class DerivedClass : MyClass {
public:
  using MyClass::MyClass;
};

DerivedClass x("abc", 1);
```

Initialisierungslisten

- Initialisierung von Containern

```
vector<int> v = { 1, 2, 3 };
list<int> l = { 3, 4, 5 };
map<string, int> m = { { "abc", 1 }, { "def", 2 } };
```

- Typ std::initializer_list

```
void f(initializer_list<int> params) {
  for (int x : params) { ... }
}

f({1, 3, 5, 7});

for (int x : {3, 5, 7, 9, 11}) { ... }
```

Uniform Initialization

- Alternative Syntax zur Initialisierung

```
int x { 0 };
pair<int, int> p { 1, 2 };
vector<int> v { 1, 2, 3 };
int *ptr {};  // int *ptr = nullptr;

vector<string> vs { 3, "abc" };
// gleichbedeutend mit vector<string> vs(3, "abc")

vector<int> vi1 { 3, 0 };  // {3, 0}
vector<int> vi2(3, 0);     // {0, 0, 0}
```

default Methoden

- erzeugt Standard-Implementierung

```
class Object {
public:
  Object *create() const { return new Object; }
  Object *copy() const { return new Object(*this); }
private:
  // ...
  Object() = default;
  Object(Object const &obj) = default;
};
```

- möglich für Default/Copy/Move Konstruktor und Zuweisungsoperatoren

Gelöschte Methoden/Funktionen

- Verhindert Erzeugung der Standard-Implementierung

```
class NonCopyable {
public:
  NonCopyable() = default;
  NonCopyable(NonCopyable const &) = delete;
  NonCopyable &operator=(NonCopyable const &) = delete;
};
```

- Für beliebige Methoden/Funktionen

```
void DoubleFunction(double x);
void DoubleFunction(float) = delete;
void DoubleFunction(int) = delete;
```

override und final

```
class A {
public:
  virtual void m1();
  virtual void m2() const;
  virtual void m3() final;
};

class B final : public A {
public:
  virtual void m1() override;
  virtual void m2() override;  // Fehler: keine passende Methode
  virtual void m3();  // Fehler: m3 wurde als final deklariert
};

class C : public B {  // Fehler: Klasse B ist final
};
```

Typdefinition mittels using

- Einfache Typdefinition (wie typedef)

```
using IntVector = vector<int>;
using IntFunctionPtr = int(*)(int);
```

- Template-Parameter möglich

```
template <typename T>
using Container = vector<T>;

Container<int> cont;
```

Statische Assertionen

- Assertionen, die zur Compilezeit geprüft werden
- Nicht erfüllte statische Assertion führt zu Compilezeitfehler
- Beispiel:

```
static_assert(sizeof(int) >= 4,
  "This code only works when int is at least 4 bytes");
```

Benutzerdefinierte Literale

```
auto operator ""_k(unsigned long long x) {
  return x * 1000;
}

auto k = 700_k;
```

- Parametertypen: unsigned long long, long double, char-Typen (char, wchar_t etc.)
- Namen ohne führenden _ sind reserviert für Standardbibliothek

Benutzerdefinierte String-Literale

- Zwei Parameter (Zeichen und Länge)

```
class MyString {
public:
  MyString(const char *str, size_t length)
    : mString(str), mLength(length) {}
private:
  size_t mLength;
  const char *mString;
};

MyString operator ""_myStr(const char *chars, size_t nr) {
  return MyString(chars, nr);
}

auto myStr = "a string"_myStr;
```

Vordefinierte Literale

- std::string

```
string s = "abc"s;
```

- Komplexe Zahlen (std::complex)

```
auto c = 1.0i;  // complex<double>
```

- Zeiten (std::chrono::duration)

```
auto duration = 1h + 30min;
```

Weitere Neuerungen

- Smart Pointer (=> Kapitel „Smart Pointer“)
- Lambda-Ausdrücke und Funktionsobjekte (=> Kapitel „Funktionsobjekte“)
- Variadic Templates und Type Traits (=> Kapitel „Templates und Template-Metaprogrammierung“)
- Multithreading (=> eigenes Kapitel)
- …