SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++-Programmierung

Verhalten, static, namespace, Operatorüberladung, ADL, Referenzen

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

Verhalten

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Implementierungsabhängiges Verhalten
- Unspezifiziertes Verhalten
- Undefiniertes Verhalten

Implementierungsabhängiges Verhalten

§3.4.1

SAARLAND UNIVERSITY
COMPUTER SCIENCE

implementation-defined behavior
unspecified behavior where each implementation documents how the choice is made

EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

Unspezifiziertes Verhalten

§3.4.4

SAARLAND UNIVERSITY
COMPUTER SCIENCE

unspecified behavior

behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated.

Undefiniertes Verhalten

§3.4.3

undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

- Bekannterer Fall: Merhfaches verändern eines Objekt in einem Ausdruck

Sequenzpunkte

Zwischen zwei aufeinanderfolgenden Sequenzpunkten ...

- darf ein Objekt nur einmal beschrieben werden

```
i = i + (i = 3); // undefiniert!
```

- Wenn ein Objekt beschrieben wird, so darf das lesen nur zum Berechnen des neuen Wertes geschehen

```
k = i + ++i; /* undefiniert: erstes lesen wird
                * nicht vom ++ verwendet */
```

Sequenzpunkte sind

- Anfang/Ende eines gesamten Ausdrucks

```
if (++i != 10) ++i; // OK, getrennte Asudruecke
```

- `&&`, `||`, `,` (wenn nicht überladen)

```
++i, ++i; // OK, wie i += 2;
```

- `?` des trinären Operators

```
k = ++i != 10 ? i : 0; // OK
```

- Direkt vor einem Funktionsaufruf nachdem seine Argumente berechnet wurden

static

Unterschiedliche Bedeutungen:

- Methoden/Attribute: Methode/Attribut der Klasse, nicht von Objekten
- Funktionen/Globale Variablen: Nur direkt zugreifbar in selber Übersetzungseinheit

Statische Methoden/Attribute

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
struct A
{
  static void m(); // kein this
  static int  x;
};

void f(A* a)
{
  // Zugriff mittels Klassenname
  A::m();
  A::x = 42;
  // Auch erlaubt, selbe Semantik
  a->m();
  a->x = 42;
}
```

- Statische Methoden können direkt nur statische Attribute/Methoden der Klasse verwenden

Statische Funktionen/globale Variablen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

a.c:

```
static void f(void) { /* ... */ }

void g(void) { f(); /* OK */ }
```

b.c:

```
void h() { f(); /* geht nicht */ }
```

- Funktionen/Variablen, die nur in einer Übersetzungseinheit verwendet werden sollten immer `static` sein

Namensräume

Namespaces

- `namespace` fasst Symbole unter einem gemeinsamen Präfix zusammen
- Verwendung von außerhalb des Namensraums mittels `::`
- Symbole `A::x` können mittels `using A::x;` in andere Namensräume eingeblendet werden
  → Bei Funktionen gilt dies für alle Überladungen
- Namensräume sind schachtelbar
- Gesamter Namensraum kann mittels `using namespace N` eingeblendet werden → unschön
- `using` nie global in Headern verwenden!
  → namespace pollution (Verschmutzung des Namensraums)
- Globales Symbol `x` immer mittels `::x` zugreifbar
- Alias: `namespace kurz = sehr_langer_name;`

```
namespace N {
  void f();
  int x;
}
```

Beispiel

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
namespace N
{
  void f(double) {}
  void f(char const*) {}
  int x;
  namespace O { int y; }
}

int main()
{
  N::f(3.14159);
  ::N::f(3.14159); // selbe Bedeutung
  N::O::y = 0;

  using N::f;
  f("hallo");

  using namespace N;
  x = 23;
  O::y = 42;
}
```

using namespace in Namensräumen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
#include <iostream>

namespace N
{
  void f(double) {
    std::cout << "N::f(double)" << std::endl;
  }
}

namespace O
{
  using namespace N;
  void f(int) {
    std::cout << "O::f(int)" << std::endl;
  }
}

int main() { O::f(3.14159); }
```

using namespace in Namensräumen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
#include <iostream>

namespace N
{
  void f(double) {
    std::cout << "N::f(double)" << std::endl;
  }
}

namespace O
{
  using namespace N;
  void f(int) {
    std::cout << "O::f(int)" << std::endl;
  }
}

int main() { O::f(3.14159); }
```

- Ausgabe: `"O::f(int)"`

using namespace in Namensräumen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
#include <iostream>

namespace N
{
  void f(double) {
    std::cout << "N::f(double)" << std::endl;
  }
}

namespace O
{
  using namespace N;
  void f(int) {
    std::cout << "O::f(int)" << std::endl;
  }
}

int main() { O::f(3.14159); }
```

- Ausgabe: `"O::f(int)"`
- Lösung: `using N::f` in `namespace O`

Operatorüberladung

- Fast jeder Operator kann überladen werden
- Überladene Operatoren sind nur andere Schreibweise für Funktionen/Methoden
- `&&`, `||`, `,` niemals überladen
  $\longrightarrow$ Definierte Auswertungsreihenfolge geht verloren
- Maßvoll einsetzen, z.B. für arithmetische Typen wie komplexe Zahlen

Beispiel

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
#include <iostream>

struct X { int i; };

std::ostream& operator <<(std::ostream& o, X const& x)
{
  o << "X: " << x.i << std::endl;
}

int main()
{
  X x;
  x.i = 23;
  std::cout << x;
  /* Aequivalent: Funktion namens "operator <<" mit
   * 2 Parametern */
  operator <<(std::cout, x);
}
```

Argument Dependent Lookup (ADL)

Auch: Koenig-Lookup

```
#include <iostream>

int main()
{
  std::cout << "hello, world" << std::endl;
}
```

- Wieso funktioniert das?
- Es gibt kein `::operator <<()`
- Es gibt nur `std::operator <<()`

Argument Dependent Lookup (ADL)

Auch: Koenig-Lookup

```
#include <iostream>

int main()
{
  std::cout << "hello, world" << std::endl;
}
```

- Wieso funktioniert das?
- Es gibt kein `::operator <<()`
- Es gibt nur `std::operator <<()`
- Lösung: Typ von `cout` (`std::ostream`) ist in `std`, daher wird auch dort nach passender Funktion gesucht

Referenzen


```
int  i;
int& r = i; // r ist Alias fuer i
r = 23;
cout << i << endl; // 23
```

- Ständiges dereferenzieren von Zeigern stört Lesefluss
  $\longrightarrow$ Besonders in Verbindung mit Operatorüberladung

  ```
  *c = *a + *b;
  ```

- Referenz ist ein Alias für ein anderes Objekt
- Kann nur initialisiert werden, danach verhält es sich wie das Objekt
- Häufigste Verwendung: Funktionsparameter