FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 7 – Werte, Zeiger und Referenzen

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Werte, Zeiger und Referenzen

Inhalt

FH MÜNSTER
University of Applied Sciences

- Wert- und Referenzsemantik
  - L-Werte und R-Werte
  - Kopierkonstruktor und Bewegender Konstruktor
  - Rule of Three und Rule of Five
- Dynamische Speicherbeschaffung
  - Wann ist dynamische Speicherbeschaffung sinnvoll?
  - Nachteile der dynamischen Speicherbeschaffung
  - Smart Pointer

Werte, Zeiger und Referenzen

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	consteval	else	mutable	signed	typeid
asm	constexpr	enum	namespace	sizeof	typename
auto	constinit	explicit	**new**	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	operator	struct	virtual
catch	co_return	float	private	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	**delete**	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Wertsemantik

Wertsemantik
Begriffserklärung

```
class Klasse
{
  // irgendein Quelltext
  // ...
private:
  int a;
  int b;
}
```

- **Wertsemantik:** Auf Objekte wird direkt zugegriffen.

Beispiel 1:

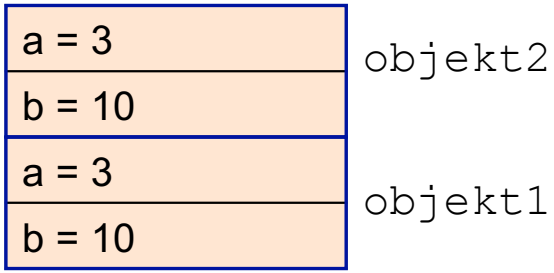
```
Klasse objekt1;
Klasse objekt2 {objekt1};
```

Beispiel 2:

```
void foo(Klasse objekt2)
{
  // irgendein Quelltext
}
```

```
Klasse objekt1;
foo(objekt1);
```

Datenspeicher:



`objekt1` und `objekt2` belegen unterschiedliche Bereiche im Speicher.

Wertsemantik

Der Kopierkonstruktor

- Der **Kopierkonstruktor** (Copy Constructor) dient dazu, ein Objekt mit einem anderen Objekt derselben Klasse zu initialisieren.
- Der **Kopierkonstruktor** wird (wie der Standardkonstruktor) automatisch vom System erzeugt und kann überschrieben werden.

➢ Es gibt Fälle in denen der Default-Kopierkonstruktor überschrieben werden muss. (Stichwort: dynamische Speicherbeschaffung)

Aufruf Standardkonstruktor

```
Klasse objekt1;
Klasse objekt2 {objekt1};
```

Aufruf Kopierkonstruktor

Signatur des Kopierkonstruktors:

```
Klasse(const Klasse& objekt);
```

Default Kopierkonstruktor erzwingen:

```
Klasse(const Klasse& objekt) = default;
```

Kopierkonstruktor unterdrücken:

```
Klasse(const Klasse& objekt) = delete;
```

Wertsemantik

Zuweisungsoperator =

- Compiler stellt automatisch den Zuweisungsoperator `operator=` (implizit generiert) bereit, der ein Objekt elementweise kopiert (siehe Kopierkonstruktor)
- Zuweisungsoperator kann überladen werden.
- Führt zu einer Kette von Zuweisungsoperator-Aufrufen, falls ein Objekt von einer Klasse abgeleitet ist oder ein anderes Objekte als Membervariable enthält.

```
Klasse obj1, obj2;
obj1 = obj2; // d.h. obj1.operator=(obj2)
```

Aufruf Zuweisungsoperator

Signatur des Zuweisungsoperators:

```
Klasse& operator=(const Klasse& other);
```

Default Zuweisung erzwingen:

```
Klasse& operator=(const Klasse& other) = default;
```

Zuweisung unterdrücken:

```
Klasse& operator=(const Klasse& other) = delete;
```

Wertsemantik

Bei Funktionsaufruf

Call-By-Value

```
void foo(const std::vector<int> vec) {
  // mache etwas mit Vektor
}
```

Auch beim Aufruf von Funktionen werden Objekte kopiert um Funktionsparameter als lokale Variablen anzulegen. *Je nach Speicherbelegung durch das Objekt und Häufigkeit des Funktionsaufrufs führt dies zu stark erhöhtem Speicherbedarf!*

Wertsemantik

Begriffserklärung: L-Wert und R-Wert

- Ein L-Wert (engl. lvalue) verfügt über einen Namen über den das Programm auf die Adresse des L-Wertes zugreifen kann. L-Werte können auf der linken oder der rechten Seite einer Zuweisung stehen.

- Ein R-Wert (engl. rvalue) ist ein temporärer Wert, der z.B. aus einer Addition resultiert oder von einer Funktion zurückgegeben wird. Auch Literale und Konstanten sind R-Werte. R-Werte können daher nur auf der rechten Seite einer Zuweisung stehen.

```
int a = 0; // a ist L-Wert, 0 ist R-Wert
int b = 3 + 4; // b ist L-Wert, 3 + 4 ist R-Wert
int c = b + 4; // c ist L-Wert, b + 4 ist R-Wert
7 = a; //FEHLER!! 7 ist ein rvalue
1 + 4 = b; //FEHLER!! 1+4 ist ein rvalue
```

Referenzsemantik

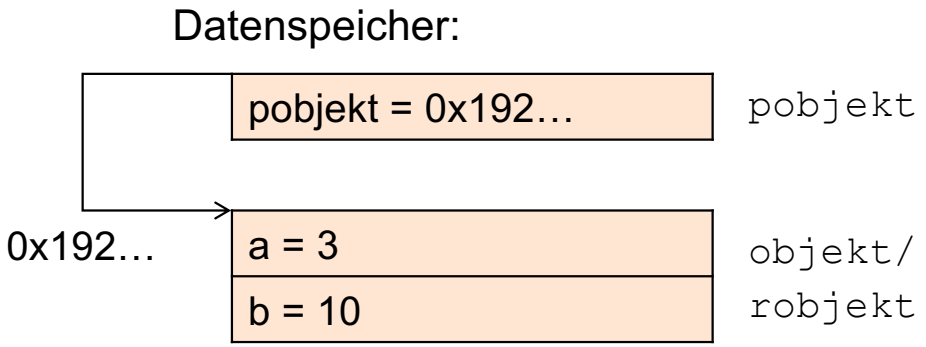
Referenzsemantik
Begriffserklärung

```
class Klasse
{
  // irgendein Quelltext
  // ...
private:
  int a;
  int b;
}
```

- **Referenzsemantik:** Zugriff auf Objekte über Referenzen (Aliasnamen).

```
Klasse objekt;
Klasse* pobjekt {&objekt}; // Zeiger
Klasse& robjekt {objekt}; // Referenz
```

Java verwendet immer eine Referenzsemantik.
In C++ kann eine Referenzsemantik mit Zeigern oder Referenzen realisiert werden.



Die Referenz `robjekt` erhält keinen eigenen Bereich im Speicher. Der Zeiger `pobjekt` schon.

Referenzsemantik

Bei Funktionsaufruf (Beispiel)

Call-By-Value

```
void foo(const std::vector<int> vec) {
  // mache etwas mit Vektor
}
```

Mit Hilfe von Zeigern und Referenzen können Kopien von Objekten und Variablen an Stellen vermieden werden, an denen das Kopieren nicht notwendig ist.

Call-By-Reference

```
void foo(const std::vector<int>* pVec) {
  // mache etwas mit Vektor
}
```

Optimiert, aber Dereferenzierung von `pVec` notwendig und somit muss Quelltext verändert werden.
Zeiger `pVec` wird im Speicher angelegt.

```
void foo(const std::vector<int>& rVec) {
  // mache etwas mit Vektor
}
```

Optimiert!

Referenzsemantik

Achtung: Bei Rückgabewerten

```
int foo(int i) {
  return i;
}
```

Rückgabe per Wert.

```
int& foo(int i) {
  return i;
}
```

Rückgabe per Referenz.
Bei einer Rückgabe per Referenz ist darauf zu achten, dass die referenzierte Variable noch existiert. *Hier: Rückgabe per Referenz basiert auf Rückgabe einer lokalen Funktions-Variable. Die Nach Funktionsaufruf nicht mehr im Speicher existiert!*

Referenzsemantik
Referenzen auf L-Werte

- **Referenz** auf L-Wert wird durch **&** ausgedrückt. **(Alles schon bekannt!)**
- Eine Referenz auf einen L-Wert ist ebenfalls ein L-Wert.
- Eine Referenz auf einen L-Wert darf nie auf einen R-Wert verweisen.

Beispiele:

```
int lw {1}; // lw ist L-Wert
int& rlw1 {lw}; // Referenz rlw1 verweist auf L-Wert
int& rlw2 {lw + 13}; // FEHLER! Eine L-Wert Referenz verweist nie auf R-Wert
```

```
// const Referenzen können auf L- und R-Werte verweisen:
const int& crlw {lw + 13}; // OK!
```

Referenzsemantik

L-Werte Referenz als Funktionsargument

- Funktion erwartet L-Wert als Funktionsargument:

```
void foo(Typ& t)
```

➢ **Vorteil:** Eine außerhalb initialisiertes Objekt/Variable (L-Wert) kann in der Funktion verändert werden. **Aber:** Funktion erlaubt keinen temporären R-Wert als Funktionsargument.

Beispiel:

```
// Implementierung
void verändere(Quadrat& quadrat)
{
  quadrat.set_grösse(3);
}
```

außerhalb der Funktion erzeugtes Objekt mit dem Namen `quadrat` (L-Wert).

```
// Funktionsaufruf
Quadrat quadrat{4}
verändere(quadrat);
quadrat.get_grösse(); // gibt 3 zurück
```

Referenzsemantik

const L-Werte Referenz als Funktionsargument

- Funktion erwartet R-Wert und L-Wert als Funktionsargument:

```
void foo(const Typ& t)
```

➢ **Vorteil:** ein temporäres Funktionsargument (R-Wert) kann an die Funktion übergeben werden.
Aber: Da das Funktionsargument konstant ist, ist es innerhalb der Funktion nicht veränderbar!

Beispiel:

```
// Implementierung
void zeichne(const Quadrat& quadrat)
{
  quadrat.zeichnen();
}
```

```
// Funktionsaufruf
zeichne( Quadrat{4} );
```

`Quadrat{4}` ist ein temporäres Objekt (R-Wert) da es keinen Bezeichner (Namen) hat.

Referenzsemantik

Referenzen auf R-Werte

- **Referenz** auf R-Wert wird durch **&&** ausgedrückt (seit C++ 11).
- Eine Referenz auf einen R-Wert ist ein L-Wert.
- Eine Referenz auf einen R-Wert darf nie auf einen L-Wert verweisen.

Beispiele:

```
int&& rrlw1 {lw + 13}; // OK! Eine R-Wert Referenz verweist auf R-Wert
int&& rrlw2 {150}; // OK! 150 ist R-Wert
int&& rrlw3 {lw}; // FEHLER! R-Wert Referenz verweist nie auf L-Wert
```

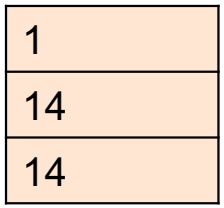
Referenzsemantik

Referenzen auf R-Werte

Beispiele:

```
int lw {1};
int lw1 {lw + 13}; // bisherige Initialisierung
int&& rrlw1 {lw + 13}; // Initialisierung mit R-Wert Referenz
```

- *Bisherige Initialisierung*



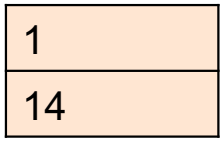
`lw`

`lw + 13` (temporärer Speicher)

`lw1`

- *Effizienzvorteil einer R-Wert Referenz bei Initialisierung*

Allerdings: Wird häufig schon im Compiler erkannt und optimiert



`lw`

`rrlw1` (`lw + 13` temporärer Speicher erhält `rrlw1` als Alias)

Referenzsemantik

R-Werte Referenz als Funktionsargument

- Funktion erwartet R-Wert als Funktionsargument:

```
void foo(Typ&& t)
```

➢ **Vorteil:** ein temporäres Funktionsargument (R-Wert) kann in der Funktion **benutzt und verändert** werden, **ohne dass eine lokale Kopie** gemacht werden muss.

Beispiel:

```
// Implementierung
void zeichne(Quadrat&& quadrat)
{
  quadrat.veraendern();
  quadrat.zeichnen();
}
```

```
// Funktionsaufruf
zeichne( Quadrat{4} );
```

`Quadrat{4}` ist ein temporäres Objekt (R-Wert) da es keinen Bezeichner (Namen) hat.

Referenzsemantik
Bewegender Konstruktor

Ein Konstruktor der mit R-Wert initialisiert wird, nennt sich bewegender Konstruktor und bewegt Daten des übergebenen temporären Objektes in das aktuell zu erzeugende Objekt **ohne zu kopieren**.

- Signatur eines bewegenden Konstruktors:

```
Klasse(Klasse&& rw);
```

- Der bewegende Konstruktor reserviert keinen neuen Speicher und ist somit sehr effizient!
- Auch der bewegende Konstruktor wird wie der Standardkonstruktor `Klasse()` und der Kopierkonstruktor `Klasse(const Klasse& objekt)` implizit erzeugt, wenn er nicht überschrieben wird. Auch hier wird mit **`default`** und **`delete`** der Konstruktor erzwungen bzw. unterdrückt.
- **Analog:** Bewegender Zuweisungsoperator `Klasse& operator=(Klasse&& rw);`

Referenzsemantik
Move-Funktion

- Die Funktion

  `std::move(arg)`

  in Headerdatei `<utility>` interpretiert ein Funktionsargument **`arg`** als R-Wert.

- Dadurch können Daten von einem Objekt in ein anderes bewegt werden ohne aufwändig Daten zu kopieren.

- ***Achtung:*** **`std::move(arg)`** *sollte nur benutzt werden, wenn die Daten in dem ausgehenden Objekt nicht mehr benötigt werden.*

Referenzsemantik

Move-Funktion

Beispiel:

```
std::string a("Hallo");
std::string b("Welt");
```

```
// Tausche Inhalte von a und b
std::string tmp1 {a};
a = b;
b = tmp1;
```

```
// Spart drei Kopieraktionen
std::string tmp2 {std::move(a)};
a = std::move(b);
b = std::move(tmp2);
```

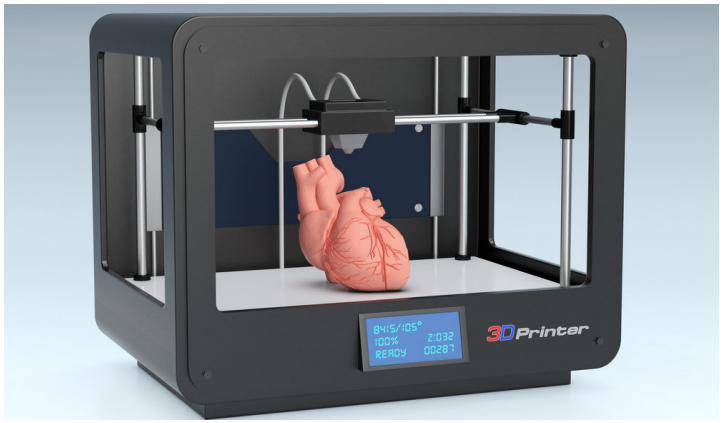
Zusammenfassung

- C++ unterscheidet grundsätzliche zwischen **L-Werten** und **R-Werten.**
- **L-Werte** haben einen Namen und können auf der linken und rechten Seite einer Zuseisung stehen (auch `const` Variablen sind daher L-Werte).
- **R-Werte** sind temporäre Objekte/Variablen ohne Namen, auf die nicht zugegriffen werden kann und die somit nicht überschrieben werden können.
- Entsprechend gibt es **L-Wert Referenzen (&)** und **R-Wert Referenzen (&&)**.
- R-Wert Referenzen geben R-Werten einen Namen und erlauben dadurch den Zugriff und somit auch die Veränderung von R-Werten
- Mit der Referenzsemantik lässt sich die Laufzeit eines Programms verkürzen

Dynamische Speicherbeschaffung

Dynamische Speicherbeschaffung

Wann ist das sinnvoll?

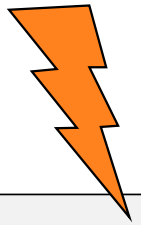


https://www.printer4you.com/magazin/bioprinting-3d-drucker-revolutionieren-die-medizin/, 25.6.21

- Dynamisch erzeugte Objekte werden im Heap gespeichert.
- **Vorteil:**
  - Deklaration und Initialisierung von Objekten kann an unterschiedlichen Orten im Quelltext erfolgen.
  - Objekte bleiben über Funktions-/Blockgrenzen erhalten und werden nicht automatisch am Ende einer Funktion / eines Blocks gelöscht.
  - Speicherbedarf muss erst zur Laufzeit bekannt sein.
  - *Beispiele*: Objekte die nicht abhängig sind von der Lebensdauer des erzeugenden Objektes und bei denen zur Kompilierzeit noch nicht klar ist ob oder wann und mit welcher Parametrisierung sie erzeugt werden sollen. Erst zur Laufzeit bekannter Speicherbedarf von Datenfeldern.
- **Nachteil:** Memory Leaks und unvorhersehbares Verhalten bei falscher Verwendung
  → später mehr

Dynamische Speicherbeschaffung

Die Schlüsselwörter **new** und **delete**



Speicherbeschaffung:

```
int* p1 = new int; // Dynamisch erzeugte Variable
int* p2 = new int{15}; // mit gleichzeitiger Initialisierung
int* pararray = new int[20]; // Dynamisch erzeugtes int-Array

Klasse* pobjekt1 = new Klasse; // Dynamisch erzeugtes Objekt
Klasse* pobjekt2 = new Klasse{1, 4}; // … mit Initialisierung
Klasse* pobjarray = new Klasse[10]; // Erzeuge dynamisches Klasse-Array
```

Bei Aufruf von **delete** wird automatisch der Destruktor der Klasse aufgerufen.

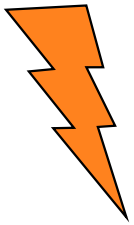
Speicherfreigabe:

```
delete p1;
delete p2;
delete [] parray;
```

```
delete pobjekt1;
delete pobjekt2;
delete [] pobjarray;
```

Dynamische Speicherbeschaffung

Die Schlüsselwörter `new` und `delete`



- Zu beachten bei der Verwendung von `new` und `delete`:
  - `delete` nur auf Objekte anwenden, die mit `new` erzeugt wurden
  - für jedes `new` darf es nur exakt ein `delete` geben
  - nach `delete` ist Zeiger undefiniert (nicht `nullptr`), muss also noch gegebenenfalls auf `nullptr` gesetzt werden
  - `delete` auf `nullptr` angewendet, bewirkt nichts und ist problemlos
  - Mit `new` erzeugte Objekte unterliegen nicht den Gültigkeitsbereichsregeln, die Zeigervariablen an sich aber schon. Ein Zeiger sollte somit bis zur Löschung des Speicherbereiches existieren!
  - Dynamische Arrays müssen mit `delete []` freigegeben werden sonst entstehen "verwitwete Objekte".

Fehler bei der Verwendung von `new` und `delete` sind schwer zu finden und werden in der Regel vom Compiler nicht erkannt. In modernem C++ sollte `new` und `delete` daher nicht verwendet werden! Sie sind aber notwendig, um "alten" Code lesen zu können bzw. für Übungszwecke.

Smart Pointer

Der "Intelligente" Zeiger

- Die Verwendung von Zeigern ist ein Schlüsselkonzept von C++
  - **Nachteil:** Leicht gemachte Fehler, die schwer zu finden sind:
    - Dereferenzierung von nicht initialisierten Zeigern
    - Mehrfachanwendung von `delete` auf einem Zeiger
    - sogenannte verwitwete Objekte
  - **Lösung**: Smart Pointer

Smart Pointer

Der "Intelligente" Zeiger

- Erwartungen an einen "intelligenten" Zeiger:
  - die Syntax soll möglichst der bekannten Schreibweise von Zeigern entsprechen
  - soll für verschiedene Klassen möglich sein
  - keine Einbußen bei Laufzeit
  - Ein "intelligenter" Zeiger hat niemals einen undefinierten Wert, d.h. er verweist auf ein Objekt oder auf ein "definiertes Nichts"
  - Dereferenzierung von nicht existierenden Objekten führt zu Exception und nicht zu einem Programmabsturz
  - Ein Objekt, auf das der Zeiger verweist, soll automatisch gelöscht werden, wenn es nicht mehr Gültig ist (vermeiden verwitweter Objekte)

- Details zum Aufbau eines Smart Pointer vgl.: Kapitel 8.5 in Breymann, U. (2020). *C++ programmieren: C++ lernen–professionell anwenden–Lösungen nutzen*. Carl Hanser Verlag GmbH Co KG.

Smart Pointer
in der C++ Standardbibliothek

seit C++11

- Zu finden in Header-Datei **<memory>**
  - **std::unique_ptr** ist eine Template Klasse:

```
std::unique_ptr<Typ> p( new Typ() )
```

  - p wird im Quelltext wie ein Zeiger benutzt, z.B. **->** zur Auswahl von Memberfunktionen oder –variablen, * zur Dereferenzierung.
  - **Aber**: kein Aufruf von **delete** mehr notwendig (erfolgt automatisch)

  - **std::make_unique** vereinfacht den obigen Ausdruck, so dass kein **new** notwendig ist:

```
auto p = std::make_unique<Typ>();
```

Smart Pointer
in der C++ Standardbibliothek

seit C++11

- Neben `std::unique_ptr` gibt es auch **`std::shared_ptr`**.
- Beim **`shared_ptr`** können mehrere Zeiger auf ein Objekt verweisen. Solange noch ein Zeiger existiert, wird das Objekt nicht aus dem Speicher gelöscht.
- Zu finden in Header-Datei **`<memory>`**
  - **`std::shared_ptr`** ist eine Template Klasse:

```
std::shared_ptr<Typ> p( new Typ() )
```

  - **`std::make_shared`** vereinfacht den obigen Ausdruck, so dass kein **`new`** notwendig ist:

```
auto p = std::make_shared<Typ>();
```

Dynamische Speicherbeschaffung

Mit Smart Pointer


```
// Unique-Pointer:
auto up1 = std::make_unique<int>(); // dynamisch erzeugte Pointer Variable
auto up2 = std::make_unique<int>(15); // mit gleichzeitiger Initialisierung
auto upararray = std::make_unique<int[]>(20); // dynamisch erzeugtes int-Array

auto upobj1 = std::make_unique<Klasse>(); // dynamisch erzeugtes Pointer Objekt
auto upobj2 = std::make_unique<Klasse>(1, 4); // … mit Initialisierung
auto upobjarray = std::make_unique<Klasse[]>(10); // erzeuge dynamisches Array

// Shared-Pointer:
auto sp1 = std::make_shared<int>(); // dynamisch erzeugte Shared-Pointer Variable
auto spobj1 = std::make_shared<Klasse>(); // Shared-Pointer Objekt
// wie oben
```

Rule of Three/Five

```
~X() = default; //Destruktor
X(X &&) = default; //bewegender Konstruktor
X& operator=(X&&) = default; //bewegender Zuweisungsop.
X(const X&) = default; //Kopierkonstruktor
X& operator =(const X&) = default;//kopierender Zuweisungsop.
```

- **Big Three:**
  - Kopierkonstruktor
  - Zuweisungsoperator
  - Destruktor

- **Rule of Three:** Wenn einer der Big Three selbst geschrieben wird, sollten die anderen auch selbst geschrieben werden
  - ➢ sonst undefiniertes Verhalten möglich

- **Hinweis**: Wenn mit dynamischer Speicherverwaltung gearbeitet wird, wird das Schreiben der Big Three meist notwendig

- **Big Five:**
  - Big Three
  - + Bewegender Konstruktor
  - + Bewegender Zuweisungsoperator

- **Verhalten:** Wenn einer der Big Three geschrieben wird, aber kein Bewegender Konstruktor, wird kein bewegender Konstruktor vom System erzeugt. Es wird der Kopierkonstruktor verwendet (analog: Zuweisungsoperator).

- **Rule of Five:** Die Big Three sollten durch Big Five ergänzt werden
  - ➢ sonst Performance einbußen

FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de