



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Programmierpraktikum Technische Informatik (C++)



2.6.2022

Überblick

- Objektorientierte Programmierung II
 - Static
 - Templates
 - Namensauflösung
 - Exception Safety
 - Operatorüberladung
- Aufgaben zu Move
- Zufallszahlen
- Tutorial



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Objektorientierte Programmierung II



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Static

Statische Member

```
struct Foo {  
    static int count;  
    static Foo createFoo();  
};
```

```
//...
```

```
Foo f = Foo::createFoo();  
std::cout<<Foo::count;
```

- Aus Java vielleicht bekannt: Statische Member

Statische Member

```
struct Foo {  
    static int count;  
    static Foo createFoo();  
};
```

```
//...
```

```
Foo f = Foo::createFoo();  
std::cout<<Foo::count;
```

- Aus Java vielleicht bekannt: Statische Member
- Statische Member sind Daten und Funktionen, die zu einer Klasse gehören, aber nicht an eine Instanz gebunden sind

Statische Member

```
struct Foo {  
    static int count;  
    static Foo createFoo();  
};
```

```
//...
```

```
Foo f = Foo::createFoo();  
std::cout<<Foo::count;
```

- Aus Java vielleicht bekannt: Statische Member
- Statische Member sind Daten und Funktionen, die zu einer Klasse gehören, aber nicht an eine Instanz gebunden sind
- Deklaration mit dem Keyword `static`

Statische Member

```
struct Foo {  
    static int count;  
    static Foo createFoo();  
};
```

```
//...
```

```
Foo f = Foo::createFoo();  
std::cout<<Foo::count;
```

- Aus Java vielleicht bekannt: Statische Member
- Statische Member sind Daten und Funktionen, die zu einer Klasse gehören, aber nicht an eine Instanz gebunden sind
- Deklaration mit dem Keyword `static`
- Zugriff auf statische Member über `classname::member`
 - Klasse fungiert ähnlich einem Namespace für statische Member und enthaltene Typen

Statische Member

```
struct Foo {  
    static int count;  
    static Foo createFoo();  
};
```

```
//...
```

```
Foo f = Foo::createFoo();  
std::cout<<Foo::count;
```

- Wesentlicher Unterschied zwischen statischen und normalen Membervariablen: Statische Membervariablen müssen außerhalb der Klasse definiert werden. (Definition außerhalb nicht notwendig, wenn Variable als Inline definiert (gcc 7+).)
 - Unterliegen der One Definition Rule, Definition sollte daher in einer .cpp-Datei liegen

Statische Member

```
struct Foo {
    static int count;
    static Foo createFoo();
};
int Foo::count;
//...
Foo f = Foo::createFoo();
std::cout<<Foo::count;
```

- Wesentlicher Unterschied zwischen statischen und normalen Membervariablen:
Statische Membervariablen müssen außerhalb der Klasse definiert werden.
(Definition außerhalb nicht notwendig, wenn Variable als Inline definiert (gcc 7+).)
 - Unterliegen der One Definition Rule, Definition sollte daher in einer .cpp-Datei liegen

Lokale statische Variablen

- **static** kann neben Membern auch auf Funktionen und lokale Variablen angewendet werden
 - Mit jeweils unterschiedlichen Auswirkungen

Lokale statische Variablen

- **static** kann neben Membern auch auf Funktionen und lokale Variablen angewendet werden
 - Mit jeweils unterschiedlichen Auswirkungen
- Für lokale Variablen bedeutet **static**, dass die Variable nur einmal erstellt wird

Lokale statische Variablen

- `static` kann neben Membern auch auf Funktionen und lokale Variablen angewendet werden
 - Mit jeweils unterschiedlichen Auswirkungen
- Für lokale Variablen bedeutet `static`, dass die Variable nur einmal erstellt wird
- Die Variable wird beim ersten Durchlauf der Funktion initialisiert

Lokale statische Variablen

- **static** kann neben Membern auch auf Funktionen und lokale Variablen angewendet werden
 - Mit jeweils unterschiedlichen Auswirkungen
- Für lokale Variablen bedeutet **static**, dass die Variable nur einmal erstellt wird
- Die Variable wird beim ersten Durchlauf der Funktion initialisiert
- Im Gegensatz zu normalen Variablen endet die Lebenszeit der Variable nicht mit Verlassen des Scopes
 - Spätere Durchläufe verwenden die im ersten Durchlauf erstellte Variable weiter, die Initialisierung wird also nicht noch einmal durchgeführt
 - Zerstörung der Variable erst mit dem Ende des Programms

Lokale statische Variablen

- **static** kann neben Membern auch auf Funktionen und lokale Variablen angewendet werden
 - Mit jeweils unterschiedlichen Auswirkungen
- Für lokale Variablen bedeutet **static**, dass die Variable nur einmal erstellt wird
- Die Variable wird beim ersten Durchlauf der Funktion initialisiert
- Im Gegensatz zu normalen Variablen endet die Lebenszeit der Variable nicht mit Verlassen des Scopes
 - Spätere Durchläufe verwenden die im ersten Durchlauf erstellte Variable weiter, die Initialisierung wird also nicht noch einmal durchgeführt
 - Zerstörung der Variable erst mit dem Ende des Programms
- Beispiel: **static** std::string = "foo"

Statische Funktionen und anonyme Namespaces

- Die Deklaration einer (freien) Funktion als `static` besagt, dass diese Funktion nur in der jeweiligen Compileunit existiert
 - Ermöglicht dem Compiler zusätzliche Optimierungen
 - Verschiedene Compileunits: Jeweils eigene Versionen der Funktion
 - Beispiel: `static int foo(){...}`

Statische Funktionen und anonyme Namespaces

- Die Deklaration einer (freien) Funktion als `static` besagt, dass diese Funktion nur in der jeweiligen Compileunit existiert
 - Ermöglicht dem Compiler zusätzliche Optimierungen
 - Verschiedene Compileunits: Jeweils eigene Versionen der Funktion
 - Beispiel: `static int foo(){...}`
- Statische freie Funktionen sind in C++ deprecated
- Bevorzugte Alternative: Anonyme Namespaces

Statische Funktionen und anonyme Namespaces

- Die Deklaration einer (freien) Funktion als `static` besagt, dass diese Funktion nur in der jeweiligen Compileunit existiert
 - Ermöglicht dem Compiler zusätzliche Optimierungen
 - Verschiedene Compileunits: Jeweils eigene Versionen der Funktion
 - Beispiel: `static int foo(){...}`
- Statische freie Funktionen sind in C++ deprecated
- Bevorzugte Alternative: Anonyme Namespaces
- Mit `namespace { ... }` wird ein anonymer Namespace erzeugt
- Anonyme Namespaces erhalten in jeder Compileunit einen (versteckten) eindeutigen Namen
 - Der “gleiche” anonyme Namespace hat in unterschiedlichen Compileunits unterschiedliche (interne) Namen
 - Vergebener Name für den Programmierer nicht relevant



Templates

Templatefunktionen

```
template<typename Container>
typename Container::value_type Sum(const Container& container,
    typename Container::value_type accum)
{
    for(const auto& entry: container)
        accum += entry;
    return accum;
}
```

Templatefunktionen

```
template<typename Container>
typename Container::value_type Sum(const Container& container,
    typename Container::value_type accum)
{
    for(const auto& entry: container)
        accum += entry;
    return accum;
}
```

- `template<TemplateArguments>` vor der eigentlichen Deklaration/Definition identifiziert ein Template
 - *TemplateArguments* in der Deklaration bereits verwendbar
- *TemplateArguments* ist kommaseparierte Liste von Deklarationen

Templateargumente

- Typargumente werden mit `typename` *ArgumentName* oder `class` *ArgumentName* deklariert
 - Fast immer identisch
 - In einigen Situationen ist nur eine Variante erlaubt, die andere führt dann zu einem Compilerfehler

Templateargumente

- Typargumente werden mit `typename ArgumentName` oder `class ArgumentName` deklariert
 - Fast immer identisch
 - In einigen Situationen ist nur eine Variante erlaubt, die andere führt dann zu einem Compilerfehler
- Templateargumente können auch Werte sein, Deklaration entsprechend mit `Type ArgumentName`
 - Nur Ganzzahltypen, Bool und Pointer als Nichttyp-Argumente zugelassen
 - Argument muss zur Compilezeit bekannt sein, daher Pointerargumente nur in Ausnahmefällen nützlich

Templateargumente

- Typargumente werden mit `typename ArgumentName` oder `class ArgumentName` deklariert
 - Fast immer identisch
 - In einigen Situationen ist nur eine Variante erlaubt, die andere führt dann zu einem Compilerfehler
- Templateargumente können auch Werte sein, Deklaration entsprechend mit `Type ArgumentName`
 - Nur Ganzzahltypen, Bool und Pointer als Nichttyp-Argumente zugelassen
 - Argument muss zur Compilezeit bekannt sein, daher Pointerargumente nur in Ausnahmefällen nützlich

```
template<typename T, unsigned I, bool B>
void foo(T a){...}
```


Templatefunktionen

```
template<typename Container>
typename Container::value_type Sum(const Container& container,
    typename Container::value_type accum)
{
    for(const auto& entry: container)
        accum += entry;
    return accum;
}
```

- Die Definition einer Templatefunktion muss für den Compiler sichtbar sein
 - Sonst keine Instanziierung möglich
 - Lässt sich in gewissem Maße umgehen, ist aber meistens nicht die Mühe wert
- Daher Templatefunktionen (und Klassen) im Header definieren!
 - Templates sind von der One Definition Rule ausgenommen

Templates immer im Header definieren!

Abhängige Namen

```
template<typename Container>
typename Container::value_type Sum(const Container& container,
    typename Container::value_type accum)
{
    for(const auto& entry: container)
        accum += entry;
    return accum;
}
```

- `Container::value_type` ist sogenannter abhängiger Typ
 - Hängt vom Templateargument `Container` ab
- Kann erst bei Instanziierung des Templates ausgewertet werden
 - Syntax wird bereits beim Parsen der Deklaration überprüft
- Woher weiß der Compiler, ob `Container::value_type` ein Typ ist?
- Antwort: Gar nicht

Abhängige Namen

```
template<typename Container>
typename Container::value_type Sum(const Container& container,
    typename Container::value_type accum)
{
    for(const auto& entry: container)
        accum += entry;
    return accum;
}
```

- Bei abhängigen Namen geht der Compiler davon aus, dass es sich um Variablen handelt
- Abhängige Typnamen müssen explizit mit `typename` als solche identifiziert werden
 - Allerdings nicht immer, an einigen Stellen, wo nur Typen stehen dürfen, kann dies weggelassen werden

Argumentdeduzierung

```
template<typename Container>
typename Container::value_type Sum(const Container& container,
    typename Container::value_type accum)
{
    for(const auto& entry: container)
        accum += entry;
    return accum;
}
```

- Bekannt: Templateargumente können deduziert werden
 - Aufruf mit `Sum(std::vector<int>{0,1,2,3},0)` ist möglich
- **Achtung:** Abhängige Namen werden für Deduzierung ignoriert
 - `typename<T> void foo(typename T::value_type arg)` kann nicht deduziert werden
- Bei mehrfacher Deduzierung müssen die Typen übereinstimmen
 - Für `template<class T> void bar(T a, T b);` ist `bar(0, 0u)` ungültig

Templateklassen

```
template<typename T>
class Interval {
    T Start;
    T End;
public:
    Interval(const T& start, const T& end) : Start(start), End(end) {}
    Interval Intersect(const Interval& other) const {...}
};
```

- Definition von Templateklassen wie für Templatefunktionen

Templateklassen

```
template<typename T>
class Interval {
    T Start;
    T End;
public:
    Interval(const T& start, const T& end) : Start(start), End(end) {}
    Interval Intersect(const Interval& other) const {...}
};
```

- Definition von Templateklassen wie für Templatefunktionen
- Klassenname in ihrem Rumpf ohne Templateargumente verwendbar
 - Entspricht genau der aktuellen Instanziierung, andere Instanziierungen müssen mit Templateparameter angegeben werden

Templateklassen

```
template<typename T>
class Interval {
    T Start;
    T End;
public:
    Interval(const T& start, const T& end) : Start(start), End(end) {}
    Interval Intersect(const Interval& other) const {...}
};
```

- Definition von Templateklassen wie für Templatefunktionen
- Klassenname in ihrem Rumpf ohne Templateargumente verwendbar
 - Entspricht genau der aktuellen Instanziierung, andere Instanziierungen müssen mit Templateparameter angegeben werden
- Alle Member müssen im Header definiert sein, insbesondere auch statische Membervariablen
 - Für `template<class T> struct Foo { static int bar; }`
Definition etwa über `template<class T> Foo<T>::bar = 5;`

Templatespezialisierungen

- Manchmal wird für bestimmte Typen eine dedizierte Implementation benötigt
 - Weil die generische Implementation für diesen Typ nicht möglich ist
 - Weil die generische Implementation für diesen Typ nicht optimal ist
- Nach Möglichkeit sollten diese Unterschiede nicht nach außen sichtbar sein
 - Überraschend, wenn bestimmte Typen nicht mit der Templateklasse, sondern mit einer anderen, fast identischen Klasse verwendet werden müssen

Templatespezialisierungen

- Manchmal wird für bestimmte Typen eine dedizierte Implementation benötigt
 - Weil die generische Implementation für diesen Typ nicht möglich ist
 - Weil die generische Implementation für diesen Typ nicht optimal ist
- Nach Möglichkeit sollten diese Unterschiede nicht nach außen sichtbar sein
 - Überraschend, wenn bestimmte Typen nicht mit der Templateklasse, sondern mit einer anderen, fast identischen Klasse verwendet werden müssen
- Lösung: Templatespezialisierung
- Gibt vor, dass ein bestimmter Implementationspfad verwendet werden soll, wenn bestimmte Typen/Werte als Templateargumente eingesetzt werden

Beispiel Templatespezialisierungen

```
template<typename T>
class Constructor
{
public:
    T construct(int a) { return T{a}; }
};
```

```
template<>
class Constructor<void>
{
public:
    void construct(int a) { }
};
```

Variadic Templates

- In einigen Fällen ist eine variable Anzahl von Templateargumenten gewünscht
 - `std::tuple`
 - `vector.emplace_back`

Variadic Templates

- In einigen Fällen ist eine variable Anzahl von Templateargumenten gewünscht
 - `std::tuple`
 - `vector.emplace_back`
- Möglich über eine festgelegte Anzahl an Argumenten mit Defaultwerten, die im unbenutzten Fall verwendet werden
 - Sehr unschöne Lösung
 - Willkürlich gewählte Obergrenze für Argumente

Variadic Templates

- In einigen Fällen ist eine variable Anzahl von Templateargumenten gewünscht
 - `std::tuple`
 - `vector.emplace_back`
- Möglich über eine festgelegte Anzahl an Argumenten mit Defaultwerten, die im unbenutzten Fall verwendet werden
 - Sehr unschöne Lösung
 - Willkürlich gewählte Obergrenze für Argumente
- Seit C++11: Variadic Templates
- Definition eines Templatearguments (Argument Pack) mit `typename... name` definiert eine beliebige Anzahl an Argumenten
- Argument Pack kann mit `name...` in eine kommaseparierte Liste entpackt werden

Variadic Templates

- In einigen Fällen ist eine variable Anzahl von Templateargumenten gewünscht
 - `std::tuple`
 - `vector.emplace_back`

- Möglich über eine festgelegte Anzahl an Argumenten mit Defaultwerten, die im unbenutzten Fall verwendet werden

- Seh
- Will

```
template<typename T, typename... Args>
void emplaceBack(std::vector<T>& container, Args...
arguments)
{
    container.push_back(T(arguments...));
}
```

- Seit C++
- Definition eines Templatearguments (Argument Pack, mit `typename... Args` definiert eine beliebige Anzahl an Argumenten
- Argument Pack kann mit `name...` in eine kommaseparierte Liste entpackt werden



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Name Lookup

Name Lookup

- Bereits angesprochen: Namensauflösung in C++ ist recht komplex
 - Im Folgenden erläuterte Regeln sind starke Vereinfachung

Name Lookup

- Bereits angesprochen: Namensauflösung in C++ ist recht komplex
 - Im Folgenden erläuterte Regeln sind starke Vereinfachung
- Scopes werden i.A. rückwärts durchsucht
 - Zunächst wird der aktuelle Scope durchsucht, danach der übergeordnete und so weiter
 - Bei Memberfunktionen werden auch die Basisklasse und ihre übergeordneten Namespaces durchsucht
 - Die Suche wird abgebrochen, sobald der Name gefunden ist
 - Namen aus übergeordneten Scopes werden verdeckt und z.B. für Overload Resolution nicht betrachtet

Name Lookup

- Bereits angesprochen: Namensauflösung in C++ ist recht komplex
 - Im Folgenden erläuterte Regeln sind starke Vereinfachung

- Scopes werden i.A. rückwärts durchsucht

- Zunächst wird der Scope der Funktion durchsucht

- Bei Memberfunktion: Der Scope der Klasse wird durchsucht

- Die Suche wird abgebrochen, wenn ein Symbol gefunden wird

- Namen aus übergeordneten Namespaces werden nicht betrachtet

```
void bar();
namespace Bar
{
    void bar(int);
    namespace Qux
    {
        void baz() { bar(); }
    }
}
```

und so weiter
geordneten Namespaces

erload Resolution nicht

Compilerfehler:

```
error: too few arguments to function 'void
Bar::bar(int)'
```

Name Lookup II

- Lookup für qualifizierte Namen funktioniert ähnlich
 - Für `Foo::bar()` wird im aktuellen Scope nach `Foo` gesucht, danach in übergeordnetem
 - Sobald ein `Foo` gefunden ist, wird die Suche nach `Foo` abgebrochen und in dem gefundenen `Foo` nach `bar()` gesucht
 - Compilerfehler, falls `bar` in **diesem** `Foo` nicht gefunden wird

Name Lookup II

- Lookup für qualifizierte Namen funktioniert ähnlich
 - Für `Foo::bar()` wird im aktuellen Scope nach `Foo` gesucht, danach in übergeordnetem
 - Sobald ein `Foo` gefunden wird, wird in dem gefundenen Scope nach `bar` gesucht
 - Compilerfehler

```
namespace Foo
{
    void bar();
}
namespace Bar
{
    namespace Qux
    {
        void baz() { Foo::bar(); }
    }
}
```

Findet `::Foo::bar`

(Führendes `::` identifiziert globalen Namespace)

Name Lookup II

■ Lookup für qualifizierte Namen funktioniert ähnlich

- Für `Foo::bar()` v
- Sobald ein `Foo` ge
- Foo nach `bar()` g
- Compilerfehler, fal

```
namespace Foo
{
    void bar();
}

namespace Bar
{
    namespace Foo
    {}
    namespace Qux
    {
        void baz() { Foo::bar(); }
    }
}
```

Compilerfehler:

error: 'bar' is not a member of 'Bar::Foo'

n in übergeordnetem
und in dem gefundenen

Name Lookup II

- Lookup für qualifizierte Namen funktioniert ähnlich
 - Für `Foo::bar()` wird im aktuellen Scope nach `Foo` gesucht, danach in übergeordnetem
 - Sobald ein `Foo` gefunden ist, wird die Suche nach `Foo` abgebrochen und in dem gefundenen `Foo` nach `bar()` gesucht
 - Compilerfehler, falls `bar` in **diesem** `Foo` nicht gefunden wird
- Es werden nur bereits deklarierte Namen gefunden
 - Ausnahme: Inlinedefinitionen von Memberfunktionen dürfen auch später deklarierte Member verwenden
 - In der Klasse definierte Friend-Funktionen verwenden dieselben Lookupregeln wie Memberfunktionen

Argument Dependent Lookup

- Für unqualifizierte Funktionsaufrufe (z.B. `swap(x, y)`) kommt eine weitere Dimension hinzu:
Argument Dependent Lookup (ADL)

Argument Dependent Lookup

- Für unqualifizierte Funktionsaufrufe (z.B. `swap(x, y)`) kommt eine weitere Dimension hinzu:
Argument Dependent Lookup (ADL)
- Abhängig von den Typen der Argumente werden zusätzliche Namespaces und Klassen durchsucht

Argument Dependent Lookup

- Für unqualifizierte Funktionsaufrufe (z.B. `swap(x, y)`) kommt eine weitere Dimension hinzu:
Argument Dependent Lookup (ADL)
- Abhängig von den Typen der Argumente werden zusätzliche Namespaces und Klassen durchsucht
- Für jedes Argument wird am Typ des Arguments eine zusätzliche Suche gestartet
 - Durchsucht den Typ selber, dessen Basisklassen und jeweils übergeordnete Namespaces

Argument Dependent Lookup

- Für unqualifizierte Funktionsaufrufe (z.B. `swap(x, y)`) kommt eine weitere Dimension hinzu:
Argument Dependent Lookup (ADL)
- Abhängig von den Typen der Argumente werden zusätzliche Namespaces und Klassen durchsucht
- Für jedes Argument wird am Typ des Arguments eine zusätzliche Suche gestartet
 - Durchsucht den Typ selber, dessen Basisklassen und jeweils übergeordnete Namespaces
- Durch diese Regel gehören freie Funktionen im selben Namespace zur Schnittstelle einer Klasse
 - Direkte Assoziation freier Funktionen mit den jeweiligen Klassen
 - Klarer Gegensatz zu Sprachen wie Java

Argument Dependent Lookup

- Für unqualifizierte Funktionsaufrufe (z.B. `swap(x, y)`) kommt eine weitere Dimension hinzu:
Argument Dependent Lookup (ADL)
- Abhängig von den Typen der Argumente werden zusätzliche Namespaces und Klassen durchsucht
- Für jedes Argument wird am Typ des Arguments eine zusätzliche Suche gestartet
 - Durchsucht den Typ selber, dessen Basisklassen und jeweils übergeordnete Namespaces
- Durch diese Regel gehören freie Funktionen im selben Namespace zur Schnittstelle einer Klasse
 - Direkte Assoziation freier Funktionen mit den jeweiligen Klassen
 - Klarer Gegensatz zu Sprachen wie Java
- Für ADL müssen Funktionen unqualifiziert aufgerufen werden

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`
- `std::swap(a, b)` erhält die Argumente als Lvalue-Referenzen und vertauscht ihren Inhalt

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`
- `std::swap(a, b)` erhält die Argumente als Lvalue-Referenzen und vertauscht ihren Inhalt

Beispielimplementation von `std::swap`

```
void swap(T& a, T& b)
{
    T tmp{std::move(a)};
    a = std::move(b);
    b = std::move(tmp);
}
```

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`
- `std::swap(a, b)` erhält die Argumente als Lvalue-Referenzen und vertauscht ihren Inhalt
- Erstellung eines Temporary oft aufwendig
 - Effizientere Implementation möglich

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`
- `std::swap(a, b)` erhält die Argumente als Lvalue-Referenzen und vertauscht ihren Inhalt
- Erstellung eines Temporary oft aufwendig
 - Effizientere Implementation möglich
- Bei Entwicklung einer Klasse wird daher häufig eine spezialisierte Swaproutine geschrieben

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`
- `std::swap(a, b)` erhält die Argumente als Lvalue-Referenzen und vertauscht ihren Inhalt
- Erstellung eines Temporary oft aufwendig
 - Effizientere Implementation möglich
- Bei Entwicklung einer Klasse wird daher häufig eine spezialisierte Swaproutine geschrieben
- **Problem:** Funktionen in `std` dürfen nicht überladen werden

std::swap

- Ein wichtiges Beispiel für den Nutzen von ADL ist `std::swap`
 - Definiert im Header `utility`
- `std::swap(a, b)` erhält die Argumente als Lvalue-Referenzen und vertauscht ihren Inhalt
- Erstellung eines Temporary oft aufwendig
 - Effizientere Implementation möglich
- Bei Entwicklung einer Klasse wird daher häufig eine spezialisierte Swaproutine geschrieben
- **Problem:** Funktionen in `std` dürfen nicht überladen werden
- **Lösung:** `swap` wird im Namespace der Klasse deklariert und über ADL aufgerufen
 - Aufrufe von `swap` sollten daher immer unqualifiziert sein

Using

- `swap(a, b)` funktioniert nur, wenn `swap` über ADL gefunden wird
 - Ansonsten muss `std::swap(a, b)` verwendet werden

Using

- `swap(a, b)` funktioniert nur, wenn `swap` über ADL gefunden wird
 - Ansonsten muss `std::swap(a, b)` verwendet werden
- Besser wäre es, wenn `std::swap` automatisch verwendet wird, falls keine Spezialisierung über ADL gefunden wird

Using

- `swap(a, b)` funktioniert nur, wenn `swap` über ADL gefunden wird
 - Ansonsten muss `std::swap(a, b)` verwendet werden
- Besser wäre es, wenn `std::swap` automatisch verwendet wird, falls keine Spezialisierung über ADL gefunden wird
- Lösung über `using namespace std`; möglich
 - Importiert den gesamten Namespace
 - Hebelt das Konzept von Namespaces aus

Using

- `swap(a, b)` funktioniert nur, wenn `swap` über ADL gefunden wird
 - Ansonsten muss `std::swap(a, b)` verwendet werden
- Besser wäre es, wenn `std::swap` automatisch verwendet wird, falls keine Spezialisierung über ADL gefunden wird
- Lösung über `using namespace std`; möglich
 - Importiert den gesamten Namespace
 - Hebelt das Konzept von Namespaces aus
- `using` kann verwendet werden, um einzelne Deklarationen aus einem Namespace zu importieren
 - `using std::swap`;

Using und Swap

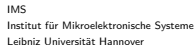
```
void bubblesort(std::vector<T>& data) {
    bool done = false;
    while(!done) {
        done = true;
        for(std::size_t i = 0; i < data.size() - 1; ++i)
            if(data[i] > data[i + 1]) {
                using std::swap;
                swap(data[i], data[i + 1]);
                done = false;
            }
    }
}
```

- Aufruf von swap immer mit `using std::swap; swap(a, b);`!
- Ähnliches gilt für einige andere Funktionen der Standardbibliothek, insbesondere `std::begin` und `std::end`

Using und Swap

```
void bubblesort(std::vector<T>& data) {
    bool done = false;
    while(!done) {
        done = true;
        for(std::size_t i = 0; i < data.size() - 1; ++i)
            if(data[i] > data[i + 1]) {
                using std::swap;
                swap(data[i], data[i + 1]);
                done = false;
            }
    }
}
```

- Aufruf von swap immer mit `using std::swap; swap(a, b);`!
 - Ähnliches gilt für einige andere Funktionen der Standardbibliothek, insbesondere `std::begin` und `std::end`
- `using`-Deklarationen immer so eng wie möglich halten
 - Nur benötigte Funktionalität importieren, nur im jeweilig kleinsten umschließenden Scope



Exception-Safety

Exception-Safety

- Exceptionsicherheit ist wichtiges Konzept in der Programmierung
 - Beachtung dieses Konzepts besonders für die korrekte Implementation von Copy/Move-Operationen wichtig (später mehr)

Exception-Safety

- Exceptionsicherheit ist wichtiges Konzept in der Programmierung
 - Beachtung dieses Konzepts besonders für die korrekte Implementation von Copy/Move-Operationen wichtig (später mehr)
- Elementare Frage: In welchem Zustand befinden sich die Datenstrukturen, falls eine Operation eine Exception wirft?
 - Die Struktur könnte korrupt sein
 - Korrekte Programmausführung nach Fangen der Exception vielleicht nicht möglich

Exception-Safety

- Exceptionsicherheit ist wichtiges Konzept in der Programmierung
 - Beachtung dieses Konzepts besonders für die korrekte Implementation von Copy/Move-Operationen wichtig (später mehr)
- Elementare Frage: In welchem Zustand befinden sich die Datenstrukturen, falls eine Operation eine Exception wirft?
 - Die Struktur könnte korrupt sein
 - Korrekte Programmausführung nach Fangen der Exception vielleicht nicht möglich
- Bei Betrachtung der Exception Safety werden Operationen in Kategorien eingeteilt

Kategorien der Exception-Safety

- No-Throw Guarantee: Die Operation kann nicht fehlschlagen

Kategorien der Exception-Safety

- No-Throw Garantie: Die Operation kann nicht fehlschlagen
- Strong Garantie: Die Operation ist entweder erfolgreich oder ohne sichtbaren Effekte
 - Bei Fehlschlag wird die Datenstruktur im ursprünglichen Zustand (vor Beginn der Operation) belassen
 - Transaktionssemantik

Kategorien der Exception-Safety

- No-Throw Guarantee: Die Operation kann nicht fehlschlagen
- Strong Guarantee: Die Operation ist entweder erfolgreich oder ohne sichtbaren Effekte
 - Bei Fehlschlag wird die Datenstruktur im ursprünglichen Zustand (vor Beginn der Operation) belassen
 - Transaktionssemantik
- Basic Guarantee: Zustand bei Fehlschlag ist undefiniert, aber gültig
 - Es dürfen keine Ressourcen (v.a. Speicher) geleakt werden

Kategorien der Exception-Safety

- No-Throw Guarantee: Die Operation kann nicht fehlschlagen
- Strong Guarantee: Die Operation ist entweder erfolgreich oder ohne sichtbaren Effekte
 - Bei Fehlschlag wird die Datenstruktur im ursprünglichen Zustand (vor Beginn der Operation) belassen
 - Transaktionssemantik
- Basic Guarantee: Zustand bei Fehlschlag ist undefiniert, aber gültig
 - Es dürfen keine Ressourcen (v.a. Speicher) geleakt werden
- No Guarantee: Keinerlei Garantien über den Zustand der Anwendung

Kategorien der Exception-Safety

- No-Throw Guarantee: Die Operation kann nicht fehlschlagen
- Strong Guarantee: Die Operation ist entweder erfolgreich oder ohne sichtbaren Effekte
 - Bei Fehlschlag wird die Datenstruktur im ursprünglichen Zustand (vor Beginn der Operation) belassen
 - Transaktionssemantik
- Basic Guarantee: Zustand bei Fehlschlag ist undefiniert, aber gültig
 - Es dürfen keine Ressourcen (v.a. Speicher) geleakt werden
- No Guarantee: Keinerlei Garantien über den Zustand der Anwendung
- Immer eine möglichst hohe Garantie anstreben!

Exception-Safety und Komposition

- Viele Funktionen rufen intern andere Funktionen auf
 - Welche Garantien gelten für derart zusammengesetzte Funktionen?

Exception-Safety und Komposition

- Viele Funktionen rufen intern andere Funktionen auf
 - Welche Garantien gelten für derart zusammengesetzte Funktionen?
- Werden ausschließlich No-Throw-Operationen ausgeführt, so ist die Funktion selbst auch No-Throw

Exception-Safety und Komposition

- Viele Funktionen rufen intern andere Funktionen auf
 - Welche Garantien gelten für derart zusammengesetzte Funktionen?
- Werden ausschließlich No-Throw-Operationen ausgeführt, so ist die Funktion selbst auch No-Throw
- Erfüllen alle Operationen die Basic Guarantee, so gilt dies auch für die Funktion
 - No-Throw und Strong Guarantee beinhalten jeweils die Basic Guarantee

Exception-Safety und Komposition

- Viele Funktionen rufen intern andere Funktionen auf
 - Welche Garantien gelten für derart zusammengesetzte Funktionen?
- Werden ausschließlich No-Throw-Operationen ausgeführt, so ist die Funktion selbst auch No-Throw
- Erfüllen alle Operationen die Basic Guarantee, so gilt dies auch für die Funktion
 - No-Throw und Strong Guarantee beinhalten jeweils die Basic Guarantee
- Werden mehrere Operationen mit Strong Guarantee aufgerufen, erfüllt die Funktion nicht zwingend die Strong Guarantee
 - Transaktionssemantik nicht garantiert
 - Basic Guarantee ist allerdings erfüllt

Exception-Safety und Komposition

- Viele Funktionen rufen intern andere Funktionen auf
 - Welche Garantien gelten für derart zusammengesetzte Funktionen?
- Werden ausschließlich No-Throw-Operationen ausgeführt, so ist die Funktion selbst auch No-Throw
- Erfüllen alle Operationen die Basic Guarantee, so gilt dies auch für die Funktion
 - No-Throw und Strong Guarantee beinhalten jeweils die Basic Guarantee
- Werden mehrere Operationen mit Strong Guarantee aufgerufen, erfüllt die Funktion nicht zwingend die Strong Guarantee
 - Transaktionssemantik nicht garantiert
 - Basic Guarantee ist allerdings erfüllt
- Für eine Strong Guarantee dürfen nach einer Änderung persistenter Daten nur noch No-Throw-Operationen ausgeführt werden!

Noexcept

- Für exceptionsicheren Code sind No-Throw-Routinen wichtig

Noexcept

- Für exceptionsicheren Code sind No-Throw-Routinen wichtig
- Kennzeichnung von No-Throw-Operationen im Quelltext ist sinnvoll
 - Erleichtert das Schreiben von Code mit starker Exceptionsicherheit
 - Ermöglicht Optimierungen abhängig von der Exceptionsicherheit gewisser Operationen

Noexcept

- Für exceptionsicheren Code sind No-Throw-Routinen wichtig
- Kennzeichnung von No-Throw-Operationen im Quelltext ist sinnvoll
 - Erleichtert das Schreiben von Code mit starker Exceptionsicherheit
 - Ermöglicht Optimierungen abhängig von der Exceptionsicherheit gewisser Operationen
- Seit C++11: noexcept-Spezifikation
 - Beispiel: `int foo(int a, int b) noexcept {...}`

Noexcept

- Für exceptionsicheren Code sind No-Throw-Routinen wichtig
- Kennzeichnung von No-Throw-Operationen im Quelltext ist sinnvoll
 - Erleichtert das Schreiben von Code mit starker Exceptionsicherheit
 - Ermöglicht Optimierungen abhängig von der Exceptionsicherheit gewisser Operationen
- Seit C++11: noexcept-Spezifikation
 - Beispiel: `int foo(int a, int b) noexcept {...}`
- noexcept-Funktionen dürfen keine Exceptions werfen
 - Selbe Regeln wie für Destruktoren: Exceptions dürfen die Funktion nicht verlassen, `try...catch` in der Funktion ist Ok
 - Bei Verstößen wird `std::terminate` aufgerufen und das Programm beendet (ohne Destruktoraufrufe!)



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Operatorüberladung

Überladung von Operatoren

- Für Klassen der Standardbibliothek werden bereits häufig Operatoren verwendet
 - Beispiele: `std::string{"foo"} == "foo"` oder
`std::unique_ptr<int> u{...}; *u = 5;`

Überladung von Operatoren

- Für Klassen der Standardbibliothek werden bereits häufig Operatoren verwendet
 - Beispiele: `std::string{"foo"} == "foo"` oder
`std::unique_ptr<int> u{...}; *u = 5;`
- Man sagt, die Operatoren sind für die jeweilige Klasse **überladen**

Überladung von Operatoren

- Für Klassen der Standardbibliothek werden bereits häufig Operatoren verwendet
 - Beispiele: `std::string{"foo"} == "foo"` oder
`std::unique_ptr<int> u{...}; *u = 5;`
- Man sagt, die Operatoren sind für die jeweilige Klasse **überladen**
- Jetzt auch Überladung von Operatoren für eigene Klassen

Überladung von Operatoren

- Für Klassen der Standardbibliothek werden bereits häufig Operatoren verwendet
 - Beispiele: `std::string{"foo"} == "foo"` oder
`std::unique_ptr<int> u{...}; *u = 5;`
- Man sagt, die Operatoren sind für die jeweilige Klasse **überladen**
- Jetzt auch Überladung von Operatoren für eigene Klassen
- Die meisten Operatoren können in C++ überladen werden
 - Ausnahmen: Scope Zugriff (`::`), Memberzugriff (`.` und `.*`), Ternary Operator (`?:`), `sizeof` und `typeid`

Operatorüberladung

```
struct Vec {
    double x;
    double y;
    Vec& operator+=(const Vec& b) {
        this->x += b.x;
        this->y += b.y;
        return *this;
    }
};

Vec operator+(const Vec& a, const Vec& b){
    Vec tmp(a);
    tmp += b;
    return tmp;
}
```

- Viele Operatoren können als Memberfunktion oder als freie Funktion überladen werden
 - `friend`-Funktionen sind freie Funktionen!
- Für Memberfunktion wird erstes Argument des Operators als `this` übergeben
- Freie Funktionen für ADL im Namespace der Klasse definieren

Operatorüberladung

```
struct Vec {  
    double x;  
    double y;  
    Vec& operator+=(const Vec& b) {  
        this->x += b.x;  
        this->y += b.y;  
        return *this;  
    }  
};  
Vec operator+(const Vec& a, const Vec& b){  
    Vec tmp(a);  
    tmp += b;  
    return tmp;  
}
```

- Überladene Operatoren sind Funktionen mit speziellem Namen
 - Analog zu Konstruktoren und Destruktoren
- Name für einen überladenen Operator: `operator`*op*
 - Beispiel: `operator==`

Freie Funktionen oder Memberfunktionen

- Viele Operatoren können als freie Funktionen oder Memberfunktionen überladen werden
 - Ausnahmen: =, [], (), -> nur als Memberfunktionen

Freie Funktionen oder Memberfunktionen

- Viele Operatoren können als freie Funktionen oder Memberfunktionen überladen werden
 - Ausnahmen: `=`, `[]`, `()`, `->` nur als Memberfunktionen
- Welche Variante ist zu bevorzugen?

Freie Funktionen oder Memberfunktionen

- Viele Operatoren können als freie Funktionen oder Memberfunktionen überladen werden
 - Ausnahmen: `=`, `[]`, `()`, `->` nur als Memberfunktionen
- Welche Variante ist zu bevorzugen?
- Wesentlicher Unterschied: `this`-Argument von Memberfunktionen muss den entsprechenden Typ haben
 - Keine impliziten Konvertierungen (außer abgeleitete zu Basisklasse)
 - Asymmetrische Operatoren (z.B. `2*Vec(1,2)`) nicht implementierbar

Freie Funktionen oder Memberfunktionen

- Viele Operatoren können als freie Funktionen oder Memberfunktionen überladen werden
 - Ausnahmen: `=`, `[]`, `()`, `->` nur als Memberfunktionen
- Welche Variante ist zu bevorzugen?
- Wesentlicher Unterschied: **this**-Argument von Memberfunktionen muss den entsprechenden Typ haben
 - Keine impliziten Konvertierungen (außer abgeleitete zu Basisklasse)
 - Asymmetrische Operatoren (z.B. `2*Vec(1,2)`) nicht implementierbar
- Für einige Operatoren machen implizite Konvertierungen (im ersten Argument) wenig Sinn, daher als Memberfunktionen implementieren
 - Unäre Operatoren (z.B. `++X`)
 - Kombinierte Zuweisungsoperationen (z.B. `X+=Y`)

Freie Funktionen oder Memberfunktionen

- Viele Operatoren können als freie Funktionen oder Memberfunktionen überladen werden
 - Ausnahmen: `=`, `[]`, `()`, `->` nur als Memberfunktionen
- Welche Variante ist zu bevorzugen?
- Wesentlicher Unterschied: **this**-Argument von Memberfunktionen muss den entsprechenden Typ haben
 - Keine impliziten Konvertierungen (außer abgeleitete zu Basisklasse)
 - Asymmetrische Operatoren (z.B. `2*Vec(1,2)`) nicht implementierbar
- Für einige Operatoren machen implizite Konvertierungen (im ersten Argument) wenig Sinn, daher als Memberfunktionen implementieren
 - Unäre Operatoren (z.B. `++X`)
 - Kombinierte Zuweisungsoperationen (z.B. `X+=Y`)
- Symmetrisch definierte Operatoren als freie Funktionen definieren
 - Arithmetische Operatoren (z.B. `X+Y`), Vergleichsoperatoren (z.B. `X<Y`)

Ausgabe über Streamout

- Streamout-Operator nicht als Memberfunktion implementierbar

Ausgabe über Streamout

- Streamout-Operator nicht als Memberfunktion implementierbar
- Grund: Erstes Argument von `cout<<MyClass()` ist Outstream
 - `std::ostream` kann als Teil der Standardbibliothek nicht modifiziert werden
 - Wäre keine gut skalierende Lösung

Ausgabe über Streamout

- Streamout-Operator nicht als Memberfunktion implementierbar
- Grund: Erstes Argument von `cout<<MyClass()` ist Outstream
 - `std::ostream` kann als Teil der Standardbibliothek nicht modifiziert werden
 - Wäre keine gut skalierende Lösung
- Streamausgabe daher immer als freie Funktion implementieren!

Ausgabe über Streamout

- Streamout-Operator nicht als Memberfunktion implementierbar
- Grund: Erstes Argument von `cout<<MyClass()` ist Outstream
 - `std::ostream` kann als Teil der Standardbibliothek nicht modifiziert werden
 - Wäre keine gut skalierende Lösung
- Streamausgabe daher immer als freie Funktion implementieren!
- Streamparameter immer als Referenz übergeben und zurückgeben!

Ausgabe über Streamout

- Streamout-Operator nicht als Memberfunktion implementierbar
- Grund: Erstes Argument von `cout<<MyClass()` ist Outstream
 - `std::ostream` kann als Teil der Standardbibliothek nicht modifiziert werden
 - Wäre keine gut skalierende Lösung
- Streamausgabe daher immer als freie Funktion implementieren!
- Streamparameter immer als Referenz übergeben und zurückgeben!

```
struct Vec
{
    double x;
    double y;
};
std::ostream& operator<<(std::ostream& os, const Vec& v)
{ return os<<"("<<v.x<<" , "<<v.y<<" )"; }
```

Allgemeine Regeln zur Operatorüberladung

- Operatorüberladungen sollten intuitiv verständlich sein
 - Nur Operatorüberladung verwenden, wenn das Verhalten gängiger Konvention entspricht

Allgemeine Regeln zur Operatorüberladung

- Operatorüberladungen sollten intuitiv verständlich sein
 - Nur Operatorüberladung verwenden, wenn das Verhalten gängiger Konvention entspricht
- Zusammengehörige Operatoren immer gemeinsam überladen
 - Beispiele: + und +=, == und !=, ...

Allgemeine Regeln zur Operatorüberladung

- Operatorüberladungen sollten intuitiv verständlich sein
 - Nur Operatorüberladung verwenden, wenn das Verhalten gängiger Konvention entspricht
- Zusammengehörige Operatoren immer gemeinsam überladen
 - Beispiele: + und +=, == und !=, ...
- Wichtige Grundregel: Normale Transformationsregeln von Operationen sollten auch für überladene Operatoren eingehalten werden
 - $a+=b$ sollte dasselbe Ergebnis wie $a=a+b$ liefern
 - $a<b$, $b>a$, $!(a>=b)$ und $!(b<=a)$ sollten alle das gleiche Ergebnis liefern

Allgemeine Regeln zur Operatorüberladung

- Operatorüberladungen sollten intuitiv verständlich sein
 - Nur Operatorüberladung verwenden, wenn das Verhalten gängiger Konvention entspricht
- Zusammengehörige Operatoren immer gemeinsam überladen
 - Beispiele: + und +=, == und !=, ...
- Wichtige Grundregel: Normale Transformationsregeln von Operationen sollten auch für überladene Operatoren eingehalten werden
 - $a+=b$ sollte dasselbe Ergebnis wie $a=a+b$ liefern
 - $a<b$, $b>a$, $!(a \geq b)$ und $!(b \leq a)$ sollten alle das gleiche Ergebnis liefern
- Von zusammengehörigen Operatoren wird nur einer vollständig implementiert, die anderen verwenden diese Implementation!

Copy Constructor und Assignment-Operator

- C++-Objekte haben Value-Semantik, werden also häufig kopiert

Copy Constructor und Assignment-Operator

- C++-Objekte haben Value-Semantik, werden also häufig kopiert
- Kopieroperation kann wie andere Operationen in C++ umdefiniert werden

Copy Constructor und Assignment-Operator

- C++-Objekte haben Value-Semantik, werden also häufig kopiert
- Kopieroperation kann wie andere Operationen in C++ umdefiniert werden
- C++ kennt dafür zwei Konstrukte
 - Copy-Constructor für Initialisierung durch Kopie (`std::string str{otherStr};`)
 - Assignment-Operator für Zuweisung auf ein bestehendes Objekt (`str = otherStr;`)

Copy Constructor und Assignment-Operator

- C++-Objekte haben Value-Semantik, werden also häufig kopiert
- Kopieroperation kann wie andere Operationen in C++ umdefiniert werden
- C++ kennt dafür zwei Konstrukte
 - Copy-Constructor für Initialisierung durch Kopie (`std::string str{otherStr};`)
 - Assignment-Operator für Zuweisung auf ein bestehendes Objekt (`str = otherStr;`)
- Werden, wenn nicht manuell definiert, vom Compiler generiert

Überladung von Kopieroperationen

```
class Foo {  
private:  
    int* ptr;  
public:  
    //other constructors and functions  
    Foo(const Foo& other): ptr(new int(*other.ptr)) {}  
    Foo& operator=(const Foo& other) {  
        if(this != &other)  
            *this->ptr = *other.ptr;  
        return *this;  
    }  
    ~Foo() { delete this->ptr; }  
};
```

Überladung von Kopieroperationen

```
class Foo {
private:
    int* ptr;
public:
    //other constructors and functions
    Foo(const Foo& other): ptr(new int(*other.ptr)) {}
    Foo& operator=(const Foo& other) {
        if(this != &other)
            *this->ptr = *other.ptr;
        return *this;
    }
    ~Foo() { delete this->ptr; }
};
```

- Copy-Konstruktor ist normaler Konstruktor, erhält einen Parameter vom Typ `const T&`

Überladung von Kopieroperationen

```
class Foo {  
private:  
    int* ptr;  
public:  
    //other constructors and functions  
    Foo(const Foo& other): ptr(new int(*other.ptr)) {}  
    Foo& operator=(const Foo& other) {  
        if(this != &other)  
            *this->ptr = *other.ptr;  
        return *this;  
    }  
    ~Foo() { delete this->ptr; }  
};
```

- Assignment-Operator hat Parameter vom Typ `const T&` und gibt `T&` zurück
 - Rückgabewert sollte immer `*this` sein
 - Parametertyp darf auch `T` sein
 - Auf Behandlung von Selbstzuweisung achten

Rvalue-Referenzen

- `T&&` definiert eine Rvalue-Referenz
 - Wie “normale” Referenzen, zeigen aber explizit auf Rvalues
 - Achtung: Eine Rvalue-Referenz ist selber ein Lvalue und benötigt `std::move`, um als Rvalue verwendet zu werden

Rvalue-Referenzen

- `T&&` definiert eine Rvalue-Referenz
 - Wie “normale” Referenzen, zeigen aber explizit auf Rvalues
 - Achtung: Eine Rvalue-Referenz ist selber ein Lvalue und benötigt `std::move`, um als Rvalue verwendet zu werden
- Funktionen dürfen weder Lvalue- noch Rvalue-Referenzen auf temporäre Objekte oder lokale Variablen zurückgeben!

Rvalue-Referenzen

- `T&&` definiert eine Rvalue-Referenz
 - Wie “normale” Referenzen, zeigen aber explizit auf Rvalues
 - Achtung: Eine Rvalue-Referenz ist selber ein Lvalue und benötigt `std::move`, um als Rvalue verwendet zu werden
- Funktionen dürfen weder Lvalue- noch Rvalue-Referenzen auf temporäre Objekte oder lokale Variablen zurückgeben!
- Bereits bekannt: Lvalues können mit `std::move` in Rvalues umgewandelt werden
 - Bedeutung: Wert wird danach nicht mehr verwendet, nur noch neue Zuweisung oder Zerstörung möglich

Rvalue-Referenzen

- `T&&` definiert eine Rvalue-Referenz
 - Wie “normale” Referenzen, zeigen aber explizit auf Rvalues
 - Achtung: Eine Rvalue-Referenz ist selber ein Lvalue und benötigt `std::move`, um als Rvalue verwendet zu werden
- Funktionen dürfen weder Lvalue- noch Rvalue-Referenzen auf temporäre Objekte oder lokale Variablen zurückgeben!
- Bereits bekannt: Lvalues können mit `std::move` in Rvalues umgewandelt werden
 - Bedeutung: Wert wird danach nicht mehr verwendet, nur noch neue Zuweisung oder Zerstörung möglich
- Funktionen können nach Lvalue- bzw. Rvalueness ihrer Argumente überladen werden
 - Beispiel: `foo(std::string&&)` und `foo(const std::string&)`
 - Implementation für Rvalues kann performanter sein, da Inhalt des Objekts nicht mehr gebraucht wird

Move-Operationen

- Zur Erinnerung: `std::move` ist lediglich ein Cast von einem Referenztyp in einen anderen
 - Nur für den Compiler relevant, erzeugt keine Laufzeitoperationen
 - Bewirkt insbesondere noch keine Verschiebung von Daten

Move-Operationen

- Zur Erinnerung: `std::move` ist lediglich ein Cast von einem Referenztyp in einen anderen
 - Nur für den Compiler relevant, erzeugt keine Laufzeitoperationen
 - Bewirkt insbesondere noch keine Verschiebung von Daten
- Auswirkungen erst bei Aufruf von für Rvalues überladenen Funktionen

Move-Operationen

- Zur Erinnerung: `std::move` ist lediglich ein Cast von einem Referenztyp in einen anderen
 - Nur für den Compiler relevant, erzeugt keine Laufzeitoperationen
 - Bewirkt insbesondere noch keine Verschiebung von Daten
- Auswirkungen erst bei Aufruf von für Rvalues überladenen Funktionen
- Insbesondere zwei Operationen werden häufig nach Rvalueness überladen:
 - Konstruktoren: **Copy-Constructor** (erhält `const T&`) und **Move-Constructor** (erhält `T&&`)
 - Zuweisungsoperator: **Copy-Assignment** (erhält `const T&`) bzw. **Move-Assignment** (erhält `T&&`)

Move-Operationen

- Zur Erinnerung: `std::move` ist lediglich ein Cast von einem Referenztyp in einen anderen
 - Nur für den Compiler relevant, erzeugt keine Laufzeitoperationen
 - Bewirkt insbesondere noch keine Verschiebung von Daten
- Auswirkungen erst bei Aufruf von für Rvalues überladenen Funktionen
- Insbesondere zwei Operationen werden häufig nach Rvalueness überladen:
 - Konstruktoren: **Copy-Constructor** (erhält `const T&`) und **Move-Constructor** (erhält `T&&`)
 - Zuweisungsoperator: **Copy-Assignment** (erhält `const T&`) bzw. **Move-Assignment** (erhält `T&&`)
- Andere Funktionen werden selten nach Rvalueness überladen
 - `foo(const A&, const B&, const C&, const D&)`
würde bereits $2^4 = 16$ Overloads benötigen

Move-Constructor und -Assignment

- Compiler generiert Move-Operationen ebenso wie Copy-Operationen automatisch
 - Außer wenn eine der folgenden Operationen manuell definiert ist:
Copy-Constructor, Copy-Assignment, Destructor
 - Manuelle Definition einer Move-Operation unterdrückt auch die automatische Erzeugung der anderen

Move-Constructor und -Assignment

- Compiler generiert Move-Operationen ebenso wie Copy-Operationen automatisch
 - Außer wenn eine der folgenden Operationen manuell definiert ist:
Copy-Constructor, Copy-Assignment, Destructor
 - Manuelle Definition einer Move-Operation unterdrückt auch die automatische Erzeugung der anderen
- Move-Operationen können gegenüber Kopien deutlich effizienter sein
 - Die Kopieroperation eines Vektors alloziert Speicher und kopiert den gesamte Inhalt des Containers
 - Die Moveoperation ändert lediglich die internen Pointer

Automatische Verwendung von Move-Operationen

- Auch ohne Verwendung von `std::move` werden an verschiedenen Stellen Move-Operationen verwendet

Automatische Verwendung von Move-Operationen

- Auch ohne Verwendung von `std::move` werden an verschiedenen Stellen Move-Operationen verwendet
- Temporaries sind Rvalues und werden automatisch verschoben statt kopiert

Automatische Verwendung von Move-Operationen

- Auch ohne Verwendung von `std::move` werden an verschiedenen Stellen Move-Operationen verwendet
- Temporaries sind Rvalues und werden automatisch verschoben statt kopiert
- In `return`-Statements werden lokale Variablen als Rvalues aufgefasst
 - Rückgabe per-Value erzeugt in der Regel nur eine Move-Operation
 - Auch die Move-Operation kann häufig rausoptimiert werden

Automatische Verwendung von Move-Operationen

- Auch ohne Verwendung von `std::move` werden an verschiedenen Stellen Move-Operationen verwendet
- Temporaries sind Rvalues und werden automatisch verschoben statt kopiert
- In `return`-Statements werden lokale Variablen als Rvalues aufgefasst
 - Rückgabe per-Value erzeugt in der Regel nur eine Move-Operation
 - Auch die Move-Operation kann häufig rausoptimiert werden
- In der Standardbibliothek werden, wenn möglich, Move-Operationen statt Kopien durchgeführt
 - Aber wann ist es möglich?

Exception Safety und Move

- Damit die Verwendung von Move möglich ist, müssen zwei Bedingungen erfüllt sein:

Exception Safety und Move

- Damit die Verwendung von Move möglich ist, müssen zwei Bedingungen erfüllt sein:
 - 1 Das alte Objekt darf nicht mehr benötigt werden
 - Offensichtlich

Exception Safety und Move

- Damit die Verwendung von Move möglich ist, müssen zwei Bedingungen erfüllt sein:
 - 1 Das alte Objekt darf nicht mehr benötigt werden
 - Offensichtlich
 - 2 Es muss eine Strong Guarantee gewährleistet werden

Exception Safety und Move

- Damit die Verwendung von Move möglich ist, müssen zwei Bedingungen erfüllt sein:
 - 1 Das alte Objekt darf nicht mehr benötigt werden
 - Offensichtlich
 - 2 Es muss eine Strong Garantie gewährleistet werden
- Beispiel: Reallokation eines `std::vector`
 - Drei Schritte: Neuen Speicher allozieren, Elemente kopieren oder verschieben, alten Speicher freigeben

Exception Safety und Move

- Damit die Verwendung von Move möglich ist, müssen zwei Bedingungen erfüllt sein:
 - 1 Das alte Objekt darf nicht mehr benötigt werden
 - Offensichtlich
 - 2 Es muss eine Strong Garantie gewährleistet werden
- Beispiel: Reallokation eines `std::vector`
 - Drei Schritte: Neuen Speicher allozieren, Elemente kopieren oder verschieben, alten Speicher freigeben
- Alte Objekte im Vector werden nicht mehr benötigt, da sie nach der Operation zerstört werden würden

Exception Safety und Move

- Damit die Verwendung von Move möglich ist, müssen zwei Bedingungen erfüllt sein:
 - 1 Das alte Objekt darf nicht mehr benötigt werden
 - Offensichtlich
 - 2 Es muss eine Strong Garantie gewährleistet werden
- Beispiel: Reallokation eines `std::vector`
 - Drei Schritte: Neuen Speicher allozieren, Elemente kopieren oder verschieben, alten Speicher freigeben
- Alte Objekte im Vector werden nicht mehr benötigt, da sie nach der Operation zerstört werden würden
- Wie steht es mit der Strong Garantie?

Exception Safety und Move II

- Strong Guarantee wird bei der Vectorreallokation dadurch erreicht, dass erst nach dem Befüllen der neue Speicher eingehängt wird

Exception Safety und Move II

- Strong Guarantee wird bei der Vectorreallokation dadurch erreicht, dass erst nach dem Befüllen der neue Speicher eingehängt wird
- Werden Items verschoben und ein Move-Constructor wirft eine Exception, verletzt dies die Strong Guarantee

Exception Safety und Move II

- Strong Guarantee wird bei der Vectorreallokation dadurch erreicht, dass erst nach dem Befüllen der neue Speicher eingehängt wird
- Werden Items verschoben und ein Move-Constructor wirft eine Exception, verletzt dies die Strong Guarantee
- Move-Operationen werden in Standardcontainern nur verwendet, wenn sie No-Throw sind
 - Kennzeichnung als `noexcept`
 - Automatisch generierte Konstruktoren/Operatoren sind `noexcept`, wenn die entsprechenden Operationen der Member dies sind

Exception Safety und Move II

- Strong Guarantee wird bei der Vectorreallokation dadurch erreicht, dass erst nach dem Befüllen der neue Speicher eingehängt wird
- Werden Items verschoben und ein Move-Constructor wirft eine Exception, verletzt dies die Strong Guarantee
- Move-Operationen werden in Standardcontainern nur verwendet, wenn sie No-Throw sind
 - Kennzeichnung als `noexcept`
 - Automatisch generierte Konstruktoren/Operatoren sind `noexcept`, wenn die entsprechenden Operationen der Member dies sind

Für Move-Constructor und Move-Assignment immer `noexcept` anstreben!

Das Copy-and-Swap-Idiom

- Für Copy-Assignment ist eine Strong Garantie wünschenswert, für Move-Assignment eine No-Throw Garantie

Das Copy-and-Swap-Idiom

- Für Copy-Assignment ist eine Strong Garantie wünschenswert, für Move-Assignment eine No-Throw Garantie
- Einfacher Ansatz: Das sog. Copy-and-Swap-Idiom

Das Copy-and-Swap-Idiom

- Für Copy-Assignment ist eine Strong Guarantee wünschenswert, für Move-Assignment eine No-Throw Guarantee
- Einfacher Ansatz: Das sog. Copy-and-Swap-Idiom

```
struct Foo{
    std::string      name;
    std::vector<int> data;
    //Copy-Ctor und Move-Ctor
    friend void swap(Foo& a, Foo& b) noexcept {
        using std::swap;
        swap(a.name, b.name);
        swap(a.data, b.data);
    }
    Foo& operator=(const Foo& other) {
        Foo tmp(other);
        swap(*this, tmp);
        return *this;
    }
    Foo& operator=(Foo&& other) noexcept {
        Foo tmp(std::move(other));
        swap(*this, tmp);
        return *this;
    }
};
```

Das Copy-and-Swap-Idiom erklärt

```
friend void swap(Foo& a, Foo& b) noexcept {  
    using std::swap;  
    swap(a.name, b.name);  
    swap(a.data, b.data);  
}  
Foo& operator=(const Foo& other) {  
    Foo tmp(other);  
    swap(*this, tmp);  
    return *this;  
}  
};
```

- Eigentliche Kopie wird in den Copy-Constructor ausgelagert
 - Wiederverwendung von Code
 - Die Kopie kann zwar fehlschlagen, modifiziert aber noch keine persistenten Daten

Das Copy-and-Swap-Idiom erklärt

```
friend void swap(Foo& a, Foo& b) noexcept {
    using std::swap;
    swap(a.name, b.name);
    swap(a.data, b.data);
}
Foo& operator=(const Foo& other) {
    Foo tmp(other);
    swap(*this, tmp);
    return *this;
}
};
```

- Eigentliche Kopie wird in den Copy-Constructor ausgelagert
 - Wiederverwendung von Code
 - Die Kopie kann zwar fehlschlagen, modifiziert aber noch keine persistenten Daten
- Swap-Implementationen sind per Konvention `noexcept`
 - **Achtung:** Gilt für `std::swap` nur, wenn Move-Operationen `noexcept`
 - Copy-and-Swap benötigt zwingend spezialisierte Swap-Implementation!

Das Copy-and-Swap-Idiom erklärt

```
friend void swap(Foo& a, Foo& b) noexcept {
    using std::swap;
    swap(a.name, b.name);
    swap(a.data, b.data);
}
Foo& operator=(const Foo& other) {
    Foo tmp(other);
    swap(*this, tmp);
    return *this;
}
};
```

- Eigentliche Kopie wird in den Copy-Constructor ausgelagert
 - Wiederverwendung von Code
 - Die Kopie kann zwar fehlschlagen, modifiziert aber noch keine persistenten Daten
- Swap-Implementationen sind per Konvention `noexcept`
 - **Achtung:** Gilt für `std::swap` nur, wenn Move-Operationen `noexcept`
 - Copy-and-Swap benötigt zwingend spezialisierte Swap-Implementation!
- Nach der ersten Änderung persistenter Daten nur noch No-Throw Operationen, somit ist eine Strong Garantie gewährleistet

Defaulted und Deleted Member

- Spezielle Member (Copy-, Move- und Defaultconstructor,...) werden normalerweise automatisch generiert

Defaulted und Deleted Member

- Spezielle Member (Copy-, Move- und Defaultconstructor,...) werden normalerweise automatisch generiert
- In vielen Fällen wäre automatisch generierte Implementation ausreichend, auch wenn die Generierung unterdrückt ist
 - z.B.: Virtuelle Destruktoren

Defaulted und Deleted Member

- Spezielle Member (Copy-, Move- und Defaultconstructor,...) werden normalerweise automatisch generiert
- In vielen Fällen wäre automatisch generierte Implementation ausreichend, auch wenn die Generierung unterdrückt ist
 - z.B.: Virtuelle Destruktoren
- C++11 erlaubt es, spezielle Member explizit auf Default zu setzen
 - Syntax: *Member-deklaration* = default;

Defaulted und Deleted Member

- Spezielle Member (Copy-, Move- und Defaultconstructor,...) werden normalerweise automatisch generiert
- In vielen Fällen wäre automatisch generierte Implementation ausreichend, auch wenn die Generierung unterdrückt ist
 - z.B.: Virtuelle Destruktoren
- C++11 erlaubt es, spezielle Member explizit auf Default zu setzen
 - Syntax: *Member-deklaration* = default;
- Manchmal ist auch Unterdrückung der automatischen Generierung sinnvoll
 - Insbesondere für Copy- und Move-Operationen, wenn die Klasse selber Ressourcen verwaltet

Defaulted und Deleted Member

- Spezielle Member (Copy-, Move- und Defaultconstructor,...) werden normalerweise automatisch generiert
- In vielen Fällen wäre automatisch generierte Implementation ausreichend, auch wenn die Generierung unterdrückt ist
 - z.B.: Virtuelle Destruktoren
- C++11 erlaubt es, spezielle Member explizit auf Default zu setzen
 - Syntax: *Member-deklaration* = **default**;
- Manchmal ist auch Unterdrückung der automatischen Generierung sinnvoll
 - Insbesondere für Copy- und Move-Operationen, wenn die Klasse selber Ressourcen verwaltet
- Mit **=delete**; kann automatische Generierung unterbunden werden
 - Gibt an, dass die Funktion nicht existiert, darf also auch nicht mehr manuell definiert werden

Beispiel Default und Deleted Member

```
class ResManager
{
private:
    Ressource Data;
public:
    ResManager(const std::string& name);
    ResManager() = default;

    ResManager(const ResManager&) = delete;
    ResManager(ResManager&&)      = default;
    ResManager& operator=(const ResManager&) = delete;
    ResManager& operator=(ResManager&&)      = default;

    void reset(const std::string& name);
    void reset(const char*) = delete;

    virtual ~ResManager() = default;
};
```

Beispiel Default und Deleted Member

```
class ResManager
{
private:
    Ressource Data;
public:
    ResManager(const std::string& name);
    ResManager() = default;

    ResManager(const ResManager&) = delete;
    ResManager(ResManager&&) = default;
    ResManager& operator=(const ResManager&) = delete;
    ResManager& operator=(ResManager&&) = default;

    void reset(const std::string& name);
    void reset(const char*) = delete;

    virtual ~ResManager() = default;
};
```

Beispiel Default und Deleted Member

```
class ResManager
{
private:
    Ressource Data;
public:
    ResManager(const std::string& name);
    ResManager() = default;

    ResManager(const ResManager&) = delete;
    ResManager(ResManager&&) = default;
    ResManager& operator=(const ResManager&) = delete;
    ResManager& operator=(ResManager&&) = default;

    void reset(const std::string& name);
    void reset(const char*) = delete;

    virtual ~ResManager() = default;
};
```

Beispiel Default und Deleted Member

```
class ResManager
{
private:
    Ressource Data;
public:
    ResManager(const std::string& name);
    ResManager() = default;

    ResManager(const ResManager&) = delete;
    ResManager(ResManager&&) = default;
    ResManager& operator=(const ResManager&) = delete;
    ResManager& operator=(ResManager&&) = default;

    void reset(const std::string& name);
    void reset(const char*) = delete;

    virtual ~ResManager() = default;
};
```

Beispiel Default und Deleted Member

```
class ResManager
{
private:
    Ressource Data;
public:
    ResManager(const std::string& name);
    ResManager() = default;

    ResManager(const ResManager&) = delete;
    ResManager(ResManager&&) = default;
    ResManager& operator=(const ResManager&) = delete;
    ResManager& operator=(ResManager&&) = default;

    void reset(const std::string& name);
    void reset(const char*) = delete;

    virtual ~ResManager() = default;
};
```

Beispiel Default und Deleted Member

```
class ResManager
{
private:
    Ressource Data;
public:
    ResManager(const std::string& name);
    ResManager() = default;

    ResManager(const ResManager&) = delete;
    ResManager(ResManager&&) = default;
    ResManager& operator=(const ResManager&) = delete;
    ResManager& operator=(ResManager&&) = default;

    void reset(const std::string& name);
    void reset(const char*) = delete;

    virtual ~ResManager() = default;
};
```



Regel der Null

- Wann sollten Copy- und Assignment-Operationen überladen werden?

Regel der Null

- Wann sollten Copy- und Assignment-Operationen überladen werden?
- Besser vermeiden: Regel der Null

Regel der Null

- Wann sollten Copy- und Assignment-Operationen überladen werden?
- Besser vermeiden: Regel der Null
- Empfehlung: Klassen werden als Komposition von Komponenten entwickelt, die die Ressourcen verwalten
 - Compilergenerierte Operationen sollten ausreichen

Regel der Null

- Wann sollten Copy- und Assignment-Operationen überladen werden?
- Besser vermeiden: Regel der Null
- Empfehlung: Klassen werden als Komposition von Komponenten entwickelt, die die Ressourcen verwalten
 - Compilergenerierte Operationen sollten ausreichen
- Alle Klassen, die dennoch Ressourcen selbst verwalten, sollten dem Single Responsibility Prinzip folgen
 - Bedeutet, dass sie **nur** zur Verwaltung der Ressource dienen
 - Sofern möglich sollten sie auch nur eine Ressource verwalten

Regel der Null

- Wann sollten Copy- und Assignment-Operationen überladen werden?
- Besser vermeiden: Regel der Null
- Empfehlung: Klassen werden als Komposition von Komponenten entwickelt, die die Ressourcen verwalten
 - Compilergenerierte Operationen sollten ausreichen
- Alle Klassen, die dennoch Ressourcen selbst verwalten, sollten dem Single Responsibility Prinzip folgen
 - Bedeutet, dass sie **nur** zur Verwaltung der Ressource dienen
 - Sofern möglich sollten sie auch nur eine Ressource verwalten
- Klassen zur Ressourcenverwaltung existieren zum Großteil bereits in Bibliotheken
 - Bei ihrer Entwicklung sind die Regeln bzgl. Vorhandensein der Operationen aber zu beachten



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Aufgaben zu Move



Einleitung

- Es folgen einige Szenarien zu Move
- Aus Gründen der Einfachheit wird dafür `std::string` verwendet

Einleitung

- Es folgen einige Szenarien zu Move
- Aus Gründen der Einfachheit wird dafür `std::string` verwendet
- Bei strikter Befolgung des Standards wirft dies Probleme auf
 - Move-Semantik von `std::string` besagt, dass ein String nach dem Move in einem validen, aber unspezifizierten Zustand ist
 - Nicht sehr hilfreich, um das Verhalten zu erläutern
- Verwendung eines Typs mit spezifiziertem Verhalten würde Code komplizierter machen

Einleitung

- Es folgen einige Szenarien zu Move
- Aus Gründen der Einfachheit wird dafür `std::string` verwendet
- Bei strikter Befolgung des Standards wirft dies Probleme auf
 - Move-Semantik von `std::string` besagt, dass ein String nach dem Move in einem validen, aber unspezifizierten Zustand ist
 - Nicht sehr hilfreich, um das Verhalten zu erläutern
- Verwendung eines Typs mit spezifiziertem Verhalten würde Code komplizierter machen
- Daher Annahme über die Implementation:
Ein String, dessen Inhalt in einen anderen gemoved wurde, ist leer
 - Ist in der Realität abhängig vom Compiler und der Implementation nicht garantiert

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}
void bar(const std::string&) {}
void bar(std::string&&)    {}
int main()
{
    std::string str = "Hello world!";
    foo(str);
    std::cout<<str<<std::endl;
}
```

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}  
void bar(const std::string&) {}  
void bar(std::string&&)    {}  
int main()  
{  
    std::string str = "Hello world!";  
    foo(str);  
    std::cout<<str<<std::endl;  
}
```

- Antwort: "Hello world!"
- Begründung:
 - str ist Lvalue-Referenz
 - Daher wird der Copy-Constructor von `std::string` verwendet

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}
void bar(const std::string&) {}
void bar(std::string&&)    {}
int main()
{
    std::string str = "Hello world!";
    foo(std::move(str));
    std::cout<<str<<std::endl;
}
```

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}  
void bar(const std::string&) {}  
void bar(std::string&&)    {}  
int main()  
{  
    std::string str = "Hello world!";  
    foo(std::move(str));  
    std::cout<<str<<std::endl;  
}
```

- Leere Zeile
- Begründung:
 - Antwort: `std::move(str)` wandelt `str` in eine Rvalue-Referenz um
 - Daher wird der Move-Constructor von `std::string` verwendet
 - Inhalt von `str` wird in das Argument von `foo` verschoben, `str` ist danach leer

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}  
void bar(const std::string&) {}  
void bar(std::string&&)    {}  
int main()  
{  
    std::string str = "Hello world!";  
    bar(str);  
    std::cout<<str<<std::endl;  
}
```

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}  
void bar(const std::string&) {}  
void bar(std::string&&)    {}  
int main()  
{  
    std::string str = "Hello world!";  
    bar(str);  
    std::cout<<str<<std::endl;  
}
```

- Antwort: "Hello world!"
 - str ist Lvalue-Referenz
 - Daher wird der `const std::string&` Overload von bar verwendet
 - str wird somit nicht modifiziert

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}  
void bar(const std::string&) {}  
void bar(std::string&&)    {}  
int main()  
{  
    std::string str = "Hello world!";  
    bar(std::move(str));  
    std::cout<<str<<std::endl;  
}
```

Aufgabe A

Was ist die Ausgabe des folgenden Codes?

```
void foo(std::string)      {}
void bar(const std::string&) {}
void bar(std::string&&)    {}
int main()
{
    std::string str = "Hello world!";
    bar(std::move(str));
    std::cout<<str<<std::endl;
}
```

■ Antwort: "Hello world!"

- `std::move(str)` ist Rvalue-Referenz
- Also wird der `std::string&&` Overload von `bar` verwendet
- Aber: Rumpf von `bar` modifiziert das Argument nicht
- `str` ist nach dem Aufruf somit weiterhin unmodifiziert

Aufgabe B

Was ist die Ausgabe des folgenden Codes?

```
std::string foo(std::string&& arg) {  
    std::string baz{arg};  
    return baz;  
}  
  
int main() {  
    std::string bar = "foo";  
    std::string qux = foo(std::move(bar));  
    std::cout<<bar<<" "<<qux<<std::endl;  
}
```

Aufgabe B

Was ist die Ausgabe des folgenden Codes?

```
std::string foo(std::string&& arg) {
    std::string baz{arg};
    return baz;
}

int main() {
    std::string bar = "foo";
    std::string qux = foo(std::move(bar));
    std::cout<<bar<<" "<<qux<<std::endl;
}
```

■ Antwort: "foo foo"

■ Begründung:

- Innerhalb von `foo` ist `arg` ein Lvalue
- Sinnvolles Verhalten, da `arg` in dem Methodenrumpf beliebig häufig verwendet werden kann
- `std::string baz{arg};` erstellt somit eine Kopie von `arg`
- `arg` ist also wiederum nach dem Aufruf weiterhin unmodifiziert

Aufgabe B

Was ist die Ausgabe des folgenden Codes?

```
std::string foo(std::string&& arg) {  
    std::string baz{std::move(arg)};  
    return baz;  
}  
  
int main() {  
    std::string bar = "foo";  
    std::string qux = foo(std::move(bar));  
    std::cout<<bar<<" "<<qux<<std::endl;  
}
```

Aufgabe B

Was ist die Ausgabe des folgenden Codes?

```
std::string foo(std::string&& arg) {  
    std::string baz{std::move(arg)};  
    return baz;  
}  
  
int main() {  
    std::string bar = "foo";  
    std::string qux = foo(std::move(bar));  
    std::cout<<bar<<" "<<qux<<std::endl;  
}
```

- Antwort: " foo"
- Begründung:
 - `std::move` wandelt `arg` für den Konstruktoraufbau in einen Rvalue um
 - Also wird der Move-Constructor aufgerufen, der Inhalt von `arg` wird in `baz` verschoben
 - `arg` und damit auch `bar` sind hinterher leer

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

"default-ctor!"

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```


Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

"move-ctor!"

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```

"copy-ctor!"

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

"copy-ctor!": Keine Zuweisung, sondern Initialisierung bei Definition einer Variablen

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

"move-assign!"

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```


Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

"move-ctor!": Compiler hat einige Move-Constructoren herausoptimiert

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {
    Foo() { std::cout<<"default-ctor!\n"; }
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }
};

Foo bar(Foo arg) {
    Foo b = arg;
    b = std::move(arg);
    return b;
}

Foo baz(Foo arg) { return arg; }

int main() {
    Foo a;
    Foo b{std::move(a)};
    b = baz(bar(b));
}
```

Aufgabe C

Was ist die Ausgabe des folgenden Codes an der markierten Stelle?

```
struct Foo {  
    Foo() { std::cout<<"default-ctor!\n"; }  
    Foo(const Foo&) { std::cout<<"copy-ctor!\n"; }  
    Foo(Foo&&) { std::cout<<"move-ctor!\n"; }  
    Foo& operator=(const Foo&) { std::cout<<"copy-assign!\n"; return *this; }  
    Foo& operator=(Foo&&) { std::cout<<"move-assign!\n"; return *this; }  
};  
Foo bar(Foo arg) {  
    Foo b = arg;  
    b = std::move(arg);  
    return b;  
}  
Foo baz(Foo arg) { return arg; }  
int main() {  
    Foo a;  
    Foo b{std::move(a)};  
    b = baz(bar(b));  
}
```

"move-assign!"



IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover



Leibniz
Universität
Hannover

Zufallszahlen

Zufallszahlen in C++

■ Random Engine

- Erzeugt Folge von Pseudo-Zufallszahlen ausgehend von einem Startwert (Seed)
- `std::default_random_engine re;`

Zufallszahlen in C++

■ Random Engine

- Erzeugt Folge von Pseudo-Zufallszahlen ausgehend von einem Startwert (Seed)
- `std::default_random_engine re;`

■ Random Device

- Quelle “echter” Zufallszahlen (z.B. Uhrzeit), sinnvoll nur für Startwert einer Random Engine
- `std::random_device rd;`

Zufallszahlen in C++

■ Random Engine

- Erzeugt Folge von Pseudo-Zufallszahlen ausgehend von einem Startwert (Seed)
- `std::default_random_engine re;`

■ Random Device

- Quelle "echter" Zufallszahlen (z.B. Uhrzeit), sinnvoll nur für Startwert einer Random Engine
- `std::random_device rd;`

■ Random Distribution

- Erzeugt speziell verteilte Zufallszahlen
- Viele Verteilungen vorhanden: Gleichverteilung (int oder double), Normalverteilung, ...
- Verwendet eine Random Engine
- Bsp.: Gleichverteilte ganze Zahl zwischen 1 und 100:
`std::uniform_int_distribution<int> u(1,100);`
`int r = u(re);` erzeugt eine Zufallszahl

Zufallszahlen in C++

■ Random Engine

- Erzeugt Folge von Pseudo-Zufallszahlen ausgehend von einem Startwert (Seed)
- `std::default_random_engine re;`

■ Random Device

- Que... Random Engine
- `std::random_device rd{};`
- `std::default_random_engine engine{rd()};`

■ Random Distribution

- `std::uniform_int_distribution<int> dist{5,10};`
- `for(size_t i = 0; i < 10; ++i)`
- Erzeuge... `std::cout<<dist(engine)<<std::endl;`
- Viele Verteilungen vorhanden: Gleichverteilung (int oder double), Normalverteilung, ...
- Verwendet eine Random Engine
- Bsp.: Gleichverteilte ganze Zahl zwischen 1 und 100:
`std::uniform_int_distribution<int> u(1,100);`
`int r = u(re);` erzeugt eine Zufallszahl