



# Embedded Programmierung mit modernem C++

Rainer Grimm

Training, Mentoring und  
Technologieberatung

# Embedded Programmierung

## Ansprechpartner

- Rainer Grimm
- Telefon 07472 917441
- E-Mail: [schulung@ModernesCpp.de](mailto:schulung@ModernesCpp.de)

## Versionsgeschichte

Version	Datum	Beschreibung	Autoren
1.0	2016-04-22	Initiale Version	Grimm Rainer Grimm Marius
1.1	2017-09-15	Präsentation vereinheitlicht	Grimm Rainer
1.2	2023-05-17	Layout von 3/4 => 16/9	Grimm Rainer

# Mythen

- Templates blähen den Code auf.
- Objekte müssen im Heap erzeugt werden.
- Exceptions sind teuer.
- C++ ist langsam und benötigt zu viel Speicher.
- C++ ist zu gefährlich in sicherheitskritischen Systemen.
- In C++ muss man objektorientiert programmieren.
- C++ kann man nur für Applikationen verwenden.
- Die iostream-Bibliothek ist zu groß, die STL-Bibliothek zu langsam.
- ➔ C++ ist ein nettes Spielzeug. Wir beschäftigen uns hier aber mit den richtigen Problemen.

# Fakten

## MISRA C++

- Motor Industry Software Reliability Association
- Regeln für C++ in kritischen (embedded) Systemen
- Aussagen über C++
  - *C++ gives good support for high-speed, low-level, input/output operations, which are essential to many embedded systems.*
  - *The increased complexity of applications make the use of a high-level language more appropriate than assembly language.*
  - *C++ compilers generate code with similar size and RAM requirements to those of C.*

# Fakten

## [TR18015.pdf](#)

- Technical report über die Performance von C++
- Spezieller Fokus auf eingebettete Systeme
- Widerlegen die Mythen
  - Namespaces have no significant overhead in size and performance.
  - The C++ casts `const_cast`, `static_cast`, and `reinterpret_cast` differ neither in size nor in performance from their C pedant.
  - A class with virtual functions has the overhead of a pointer and a virtual function table.
  - The call of a virtual function is as expensive as the call of a free function with the help of a pointer that is stored in a table.
  - The inlining of a function causes significant performance benefits and is close to the performance of a C macro.
  - Modern C++ compilers can massively reduce the number of template instantiations.

# Fakten

## AUTOSAR Guidelines

- basieren auf C++14
- aktualisieren die MISRA C++ Guidelines
- erlaubt die Verwendung von dynamischem Speicher
- verbietet
  - `malloc`, `free` und C-casts
  - `const_cast`, `dynamic_cast` und `reinterpret_cast`
  - `typedef` **Bezeichner**
  - Unions
  - Mehrfachvererbung
  - `friend` Deklarationen
  - benutzerdefinierte Literale
  - dynamische Ausnahmespezifikationen (`throw`)
- stellt die Unterschiede zu den C++ Core Guidelines dar

# Fakten

## C++ Core Guidelines

- Regeln für das Schreiben von modernem C++ Code
- Herausgeber: Bjarne Stroustrup und Herb Sutter
- die Guidelines Support Library verifiziert die Regeln
- Artikel zu den [Guidelines](#)

# Besondere Anforderungen

Sicherheitskritische Systeme

Hohe Performanz

Eingeschränkte Ressourcen

Mehrere Aufgaben gleichzeitig



# Besondere Anforderungen

Sicherheitskritische Systeme

Hohe Performanz

Eingeschränkte Ressourcen

Mehrere Aufgaben gleichzeitig

# Vereinheitlichte Initialisierung mit { }

**Grundregel:** Eine { }-Initialisierung ist in allen Initialisierungen anwendbar.

- Zwei Formen:
  - Direkte Initialisierung  
`string str{"my String"};`
  - Kopierinitialisierung  
`string str = {"my String"};`

# Vereinheitlichte Initialisierung mit { }

Die Initialisierung mit { } erlaubt keine Verengung (*narrowing conversion*).

➡ Heimlicher Verlust der Datengenauigkeit.

```
int i1(3.14);           // OK
int i2{3.14};           // ERROR
int i3 = {3.14};        // ERROR

char c1(999);           // OK
char c2{999};           // ERROR

char c3{8};             // OK
```

# Vereinheitlichte Initialisierung



- Beispiele:
  - `uniformInitialization.cpp`
  - `initializerList.cpp`
- Aufgaben:
  - Initialisieren Sie die verschiedenen Container `std::array`, `std::vector`, `std::set` und `std::unordered_multiset` mit der `{-10, 5, 1, 4, 5}`-Initialisiererliste.
    - Lösung: `initializerList.cpp`
- Weitere Informationen:
  - [std::initializer\\_list](#)

# Automatische Typableitung: `auto`

Der Compiler bestimmt den Typ aus seinem Initialisierer:

```
auto myDoub = 3.14;
```

- Verwendet die Mechanismen für die Bestimmung der Argumente eines Funktions-Templates.
- Ist sehr hilfreich in komplexen Templateausdrücken.
- Erlaubt mit unbekannten Typen zu arbeiten:

```
auto func = []{return 5;}
```

- Ist mit Vorsicht bei Initialisiererlisten zu verwenden.

```
auto myInt{2011};
```

```
auto myInt2 = {2011};
```

- Verbindet die dynamische Typisierung einer Interpreter- mit der statischen Typisierung einer Compiler-Sprache.

# auto: Refaktorisierung

```
auto a= 5;
auto b= 10;
auto sum = a * b * 3;
auto res = sum + 10;
std::cout << typeid(res).name();           // i
```

```
auto a2 = 5;
auto b2 = 10.5;
auto sum2 = a2 * b2 * 3;
auto res2 = sum2 * 10;
std::cout << typeid(res2).name();          // d
```

```
auto a3 = 5;
auto b3 = 10;
auto sum3 = a3 * b3 * 3.1f;
auto res3 = sum3 * 10;
std::cout << typeid(res3).name();          // f
```

# auto-matisch initialisiert

```
struct T1{};  
class T2{  
    public:  
        T2() {}  
};
```

```
int n;                // OK
```

```
int main(){  
    int n;             // ERROR  
    std::string s;     // OK  
    T1 t1;             // OK  
    T2 t2;             // OK  
}
```

```
struct T1 {};  
class T2{  
    public:  
        T2() {}  
};
```

```
auto n = 0;
```

```
int main(){  
    auto n = 0;  
    auto s = ""s;  
    auto t1 = T1();  
    auto t2 = T2();  
}
```

# Automatische Typableitung: `auto`



- Beispiele:
  - `auto.cpp`
  - `autoExplicit.cpp`
- Aufgaben:
  - Machen Sie die Typableitung von `auto` explizit.
    - Versuchen Sie möglichst viele Verwendungen von `auto` in `autoExplicit.cpp` durch den expliziten Typ zu ersetzen.
    - Beachten Sie die notwendigen Header-Dateien.
    - Lösung: `autoExplicit.cpp`
- Weitere Informationen:
  - [auto](#)



# Aufzählungen mit Gültigkeitsbereich

Aufzählungen mit Gültigkeitsbereich werden auch streng typisierte Aufzählungstypen genannt.

```
enum class StrongColor{red, blue, green};
```

- Regeln

- Lassen sich nur im Gültigkeitsbereich der Aufzählung ansprechen.
- Konvertieren nicht implizit zu `int`.
- Verschmutzen nicht den globalen Namensbereich.
- Der zugrunde liegender Typ ist per Default `int`, kann aber explizit angegeben werden.

```
enum class StrongColor: char{red, blue, green};
```

Aufzählungen mit Gültigkeitsbereich können vorwärts deklariert werden.

# Aufzählungen mit Gültigkeitsbereich



- Beispiele:
  - `enum.cpp`
- Weitere Informationen:
  - [enum](#)

# `nullptr` statt 0 oder NULL

`nullptr` ist ein richtiger Zeiger

- verweist auf kein Datum und lässt sich nicht dereferenzieren
  - kann mit allen Zeigern verglichen und in alle Zeiger impliziert konvertiert werden
  - kann nur in einen Wahrheitswert konvertiert werden
- 
- Räumt mit der Mehrdeutigkeit der Zahl 0 und dem Makro NULL auf.
    - **0**: wird entweder als Nullzeiger `((void*) 0)` oder die natürliche Zahl 0 interpretiert.
    - **NULL**: lässt sich in der Regel in einen integralen Typ konvertieren.

# Nullzeiger-Literal: `nullptr`



- Beispiele:
  - `nullptr.cpp`
- Weitere Informationen:
  - [nullptr](#)

# Benutzerdefinierte Literale



Syntax: <built\_in-Literal> +    + <Suffix>

- Natürliche Zahlen: `10101010_b`
- Fließkommazahlen: `123.45_km`
- C-String-Literale: `"hello"_i18n`
- Zeichen-Literale: `'1'_character`

# Benutzerdefinierte Literale

Die C++-Laufzeit bildet die benutzerdefinierten Literale auf den entsprechenden Literal-Operatoren ab.

- `1_m` ➔ `operator "" _m(1){ ...`
- `"hello"_i18n` ➔ `operator "" _i18n("hello", 5)`
- Den Literal-Operator gibt es in der
  - cooked- und raw-Form für natürliche Zahlen und Fließkommazahlen.
  - raw-Form für C-String Literale und Zeichen-Literale.
- Die cooked-Form besitzt die höhere Priorität.

# Benutzerdefinierte Literale

- Cooked Form

- Nimmt ihr Argument als `long double` bzw. `unsigned long long int` an

`1.45_km` ➡ `operator "" _km(1.45)`

- Raw-Form

- Nimmt ihr Argument als `(const char*, size_t)`, `(const char*)` bzw. `const char` an

`1.45_km` ➡ `operator "" _km("1.45")`

- Regeln

- zwischen `""` und `_km` muss ein Leerzeichen sein
  - Literale sollen mit Unterstrich `_km` beginnen, um sie von den built-in Literalen zu unterscheiden

# Built-in Literale

C++14 bringt einen Satz an built-in Literalen mit.

Typ	Präfix/Suffix	Beispiel
Binäre Zahl	0b	0b10
<code>std::string</code>	s	"HELLO"s
<code>complex&lt;double&gt;</code>	i	5i
<code>complex&lt;long double&gt;</code>	il	5il
<code>complex&lt;float&gt;</code>	if	5if
<code>std::chrono::hours</code>	h	5h
<code>std::chrono::minutes</code>	min	5min
<code>std::chrono::seconds</code>	s	5s
<code>std::chrono::milliseconds</code>	ms	5ms
<code>std::chrono::microseconds</code>	us	5us
<code>std::chrono::nanoseconds</code>	ns	5ns



# Benutzerdefinierte und built-in Literale



- Beispiele:
  - `userDefinedLiteral.cpp`
  - `built_inLiteral.cpp`
- Aufgaben:
  - Erweitern Sie `MyDistance`.
    1. `MyDistance` soll die Längeneinheit Fuß (0.3048m) und Meile (1609.344m) unterstützen.  
Welche Suffixe wählen Sie?
    2. Ihre tägliche Fahrtstrecke mit dem Auto besteht aus mehreren Fahrten zur Arbeit, ihren Fahrten zum Einkaufen und zum Fitness-Studio.  
Wie müssen Sie `MyDistance` erweitern, damit Sie die Gesamtdistanz in einem Ausdruck direkt ausrechnen können?  

```
Distance::myDistance myDisPerWeek;  
myDistPerWeek= 10 * work + 2 * shopping + 4 * workout;
```
- Lösung: `userDefinedLiteralExtended.cpp`

# Zusicherungen zur Compilezeit

`static_assert(Ausdruck, Text)` gibt den Text als Fehlermeldung aus, falls der Ausdruck zu falsch evaluiert.

- `static_assert`
  - kann überall im Sourcecode verwendet werden.
  - besitzt keinen Einfluss auf die Laufzeit des Programms.
  - lässt sich ideal mit der neuen Type-Traits-Bibliothek kombinieren.
    - Die Type-Traits-Bibliothek erlaubt mächtige Typabfragen zur Übersetzungszeit.
    - `static_assert` validiert die Type-Traits Aufrufe.

# Zusicherungen zur Compilezeit

- Beispiele:
  - `staticAssert.cpp`
- Aufgaben:
  - Stellen Sie sicher, dass der Algorithmus `gcd` nur mit natürlichen Zahlen verwendet werden kann. Testen Sie ihren Algorithmus.

```
template<typename T>
T gcd(T a, T b){
    if( b == 0 ){
        return a;
    }
    else{
        return gcd(b, a % b);
    }
}
```

- Lösung: `staticAssertGcd.cpp`
- Weitere Informationen:
  - [static\\_assert](#)
  - [Weitere Variationen](#) des gcd Algorithmus

# Initialisieren der Klassenelemente

- Klassenelemente lassen sich direkt initialisieren.

```
class MyClass{  
    const static int oldX = 5;           // C++98  
    int newX = 5;                       // C++11  
    vector<int> myVec{1, 2, 3, 4, 5};    // C++11  
};
```

- Wird ein Klassenelement über die Initialisiererliste des Konstruktors initialisiert, besitzt dieser Aufruf höhere Präzedenz.

```
struct MyClass{  
    MyClass() = default;  
    explicit MyClass(int n):newX(n){}  
    int newX = 5;  
};
```

# Initialisieren der Klassenelemente



- Beispiele:
  - `classMemberInitializer.cpp`
  - `classMemberInitializerWidget.cpp`
- Aufgaben:
  - Vereinfachen Sie die Konstruktoren.
    - Die Konstruktoren der Klasse `Widget` in `classMemberInitializerWidget.cpp` lassen sich deutlich vereinfachen. Wenden Sie dazu das direkte Initialisieren der Klassenelemente an.
      - Lösung: `classMemberInitializerWidget.cpp`

# Besondere Anforderungen

Sicherheitskritische Systeme

**Hohe Performanz**

Eingeschränkte Ressourcen

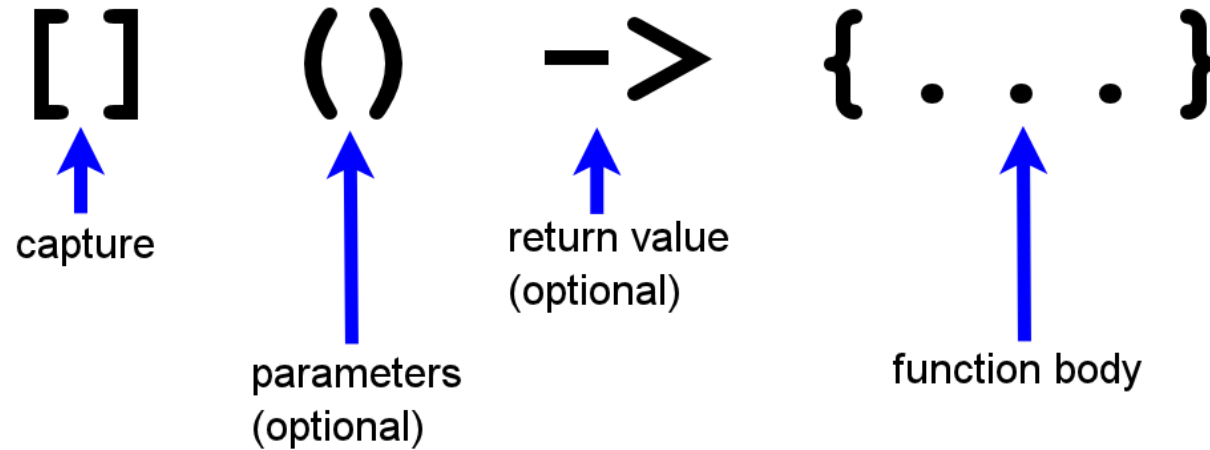
Mehrere Aufgaben gleichzeitig

# Lambda-Funktionen



- Lambda-Funktionen
  - Sind Funktionen ohne Namen.
  - Definieren ihre Funktionalität an Ort und Stelle.
  - Können wie Daten kopiert werden.
  - Können ihren Aufrufkontext speichern.
- Lambda-Funktionen sollen
  - Kurz und knackig sein.
  - Selbsterklärend sein.

# Lambda-Funktionen: Syntax



- `[]` : Bindung der verwendeten Variablen per Copy oder Referenz möglich
- `()` : Bei Parametern notwendig
- `->` : Bei komplexeren Lambda-Funktionen notwendig
- `{ }` : Funktionskörper , per Default `const`  
`[] () mutable -> { . . . }` besitzt einen nicht-konstanten Funktionskörper



# Lambda-Funktionen: Closure

Lambda-Funktionen können ihren Aufrufkontext binden.

➔ Closure

Bindung	Beschreibung
<code>[]</code>	Kein Bezug
<code>[a]</code>	a per Kopie
<code>[&amp;a]</code>	a per Referenz
<code>[=]</code>	Alle verwendeten Variablen per Kopie
<code>[&amp;]</code>	Alle verwendeten Variablen per Referenz
<code>[=, &amp;a]</code>	Standardmäßig per Kopie; a per Referenz
<code>[&amp;, a]</code>	Standardmäßig per Referenz; a per Kopie
<code>[this]</code>	Daten und Mitglieder der umgebenden Klasse per Kopie
<code>[l = std::move(lock)]</code>	Verschiebt lock (C++14)

# Lambda-Funktionen: First-Class

Lambda-Funktionen können sowohl in Variablen gespeichert, als auch als Argument oder Rückgabewert einer Funktion verwendet werden.

## ➡ First-Class-Function

```
auto addTwoNumber= [](int a, int b){return a + b};
```

```
std::thread th([]{std::cout << "Hello from thread" << std::endl;});
```

```
std::function<int(int, int)> makeAdd(){  
    return [](int a, int b){return a + b};  
}
```

```
std::function<int(int, int)> myAdd= makeAdd();  
myAdd(2000, 11);    // 2011
```

# Generische Lambdas: C++14

In C++14 können Lambda-Funktionen den Typ ihrer Argumente automatisch bestimmen.

```
auto add11 = [](int i, int i2){return i + i2;};  
auto add14 = [](auto i, auto i2){return i + i2;};  
  
std::vector<int> myVec{1, 2, 3, 4, 5};  
auto res11 = std::accumulate(myVec.begin(), myVec.end(), 0, add11);  
auto res14 = std::accumulate(myVec.begin(), myVec.end(), 0, add14);  
    // res11 == res14 == 15;  
  
std::vector<std::string> myVecStr{"Hello"s, " World"s};  
auto st = std::accumulate(myVecStr.begin(), myVecStr.end(), ""s, add14);  
std::cout << st << std::endl;    // Hello World
```

# Lambda-Funktionen



- Beispiele:
  - `lambdaFunction.cpp`
  - `lambdaFunctionClosure.cpp`
  - `lambdaFunctionCapture.cpp`
  - `lambdaFunctionThis.cpp`
  - `lambdaFunctionGeneric.cpp`
- Aufgaben:
  - Das Programm `lambdaFunctionCapture.cpp` besitzt undefiniertes Verhalten. Korrigieren Sie das Programm.
    - Lösung: `lambdaFunctionCapture.cpp`
  - Die Regeln rund um Lambda-Funktionen werden schnell anspruchsvoll. Überzeugen Sie sich.
- Weitere Informationen:
  - [Lambda-Funktionen](#)

# Type-Traits

Ermöglichen zur Übersetzungszeit Typabfragen, Typvergleiche und Typtransformationen.

➡ Type-Traits besitzen keinen Einfluss auf die Laufzeit des Programms

- Benötigen die Headerdatei `<type_traits>`.
- Anwendung von Template Metaprogramming
  - Programmierung zur Compilezeit
  - Programmierung auf Typen und nicht auf Werten
  - Compiler interpretiert die Templates und transformiert diese in C++-Quelltext

# Type-Traits: Ziele

- Optimierung
  - Code, der sich beim Übersetzen selbst optimiert
    - ➡ Abhängig vom Typ einer Variable wird ein bestimmter Algorithmus ausgewählt
  - Optimierte Versionen von `std::copy`, `std::fill` oder `std::equal`
    - ➡ Algorithmen können direkt auf Speicherbereichen angewandt werden
- Korrektheit
  - Typinformationen werden zur Übersetzungszeit evaluiert
  - die evaluierten Typinformationen definieren mit `static_assert` verbindliche Zusicherungen an den Code

# Type-Traits

- Typabfragen

- Primäre Typkategorien (::value)

- `std::is_pointer<T>`, `std::is_integral<T>`,  
`std::is_floating_point<T>`

- Zusammengesetzte Typkategorien (::value)

- `std::is_arithmetic<T>`, `std::is_object<T>`

- Typvergleiche (::value)

- `std::is_same<T, U>`, `std::is_base_of<Base, Derived>`,  
`std::is_convertible<From, To>`

- Typtransformationen (::type)

- `std::add_const<T>`, `std::remove_reference<T>`,  
`std::make_signed<T>`, `std::add_pointer<T>`

# Type-Traits



- Beispiele:
  - `typeTraitsTypeCategories.cpp`
  - `typeTraitsCopy.cpp`
  - `typeTraitsGcd.cpp`
- Aufgaben:
  - Modifizieren Sie ein `int`-Typ zu Compilezeit?
    - Fügen Sie `const` zu ihrem Typ hinzu.
    - Entfernen Sie `const` von ihrem Typ.
    - Vergleichen Sie Ihren Typ mit einem `const int`.
    - Lösung: `typeModifications.cpp`
- Weitere Informationen:
  - [Type-Traits](#)
  - [Weitere Variationen](#) des gcd Algorithmus (siehe Type-Traits Correctness)



# Konstante Ausdrücke: constexpr

## Konstante Ausdrücke

- können zur Compilezeit evaluiert werden.
- geben dem Compiler tiefen Einblick in den Code.
- sind implizit thread-sicher.
- Variablen `constexpr double myDouble= 5.2;`
  - sind implizit `const`.
- Funktionen 

```
constexpr int fac(int n){ return n > 0 ?  
                                n * fac(n-1):  
                                1;  
                                }
```

  - müssen einen Wert zurückliefern.
  - werden zur Compilezeit evaluiert, wenn sie mit konstanten Ausdrücken aufgerufen werden.
  - können nur einen Funktionskörper besitzen der aus einer Rückgabeanweisung besteht.
  - der Rückgabewert muss selbst ein konstanter Ausdruck sein.
  - sind implizit inline.

# Konstante Ausdrücke: constexpr

- Benutzerdefinierte Typen

```
struct MyDouble{  
    double myVal;  
    constexpr MyDouble(double v) : myVal(v) {}  
    constexpr double getVal() {return myVal;}  
};
```

- Der Konstruktor muss selbst ein konstanter Ausdruck sein.
- Der benutzerdefinierte Typ kann Memberfunktionen besitzen, die selbst konstante Ausdrücke sind.
- Instanzen von `MyDouble` können zur Compilezeit instanziiert werden.

- Mit **C++14** können Funktionen

- Variablen enthalten, die mit einer Konstanten initialisiert wurde.
- bedingte Sprung- und Iterationsanweisungen enthalten.
- keine statische oder `thread_local` Variablen enthalten.

# Konstante Ausdrücke: constexpr



- Beispiele:
  - `constExpression.cpp`
  - `constExpressionCpp14.cpp`
- Aufgaben:
  - Verwenden Sie `MyDouble` in einem Programm.
    - Wie können Sie sicherstellen, dass Instanzen von `MyDouble` tatsächlich zur Compilezeit erzeugt werden?
    - Was passiert, wenn `MyDouble` mit einem nicht konstanten Ausdruck instanziiert wird?

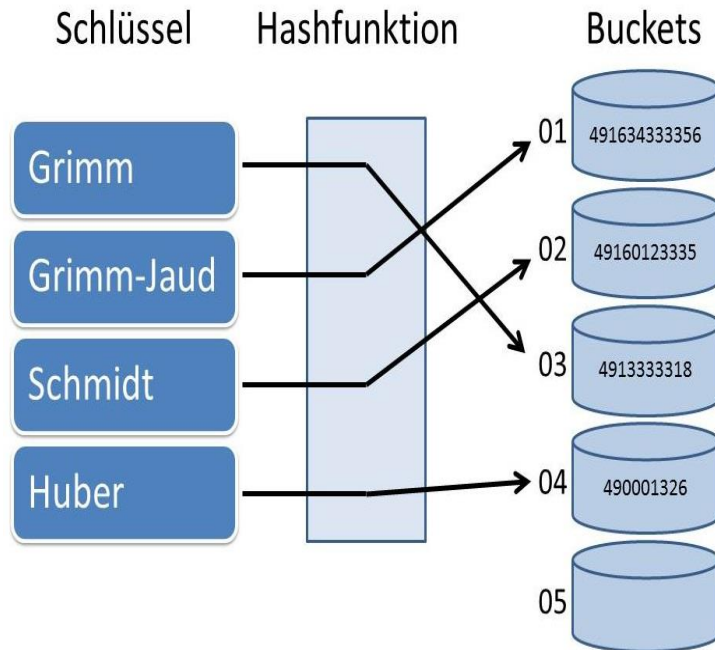
```
struct MyDouble{  
    double myVal;  
    constexpr MyDouble(double v): myVal(v){}  
    constexpr double getVal(){ return myVal; }  
};
```

- Weitere Informationen:
  - [constexpr](#)
  - [Distanzberechnungen zur Compilezeit](#)

# Assoziative Container

Assoziativer Container	Schlüssel sortiert	Wert zugeordnet	Mehrere gleiche Schlüssel	Zugriffszeit	Standard
<code>std::set</code>	ja	nein	nein	logarithmisch	C++98
<code>std::unordered_set</code>	nein	nein	nein	konstant	C++11
<code>std::map</code>	ja	ja	nein	logarithmisch	C++98
<code>std::unordered_map</code>	nein	ja	nein	konstant	C++11
<code>std::multiset</code>	ja	nein	ja	logarithmisch	C++98
<code>std::unordered_multiset</code>	nein	nein	ja	konstant	C++11
<code>std::multimap</code>	ja	ja	ja	logarithmisch	C++98
<code>std::unordered_multimap</code>	nein	ja	ja	konstant	C++11

# Ungeordnete assoziative Container



Klassische Anwendung:

- Schlüssel: Familienname
- Wert: Telefonnummer

```
std::unordered_map<std::string, int> {{"Grimm", 4916343333},  
    {"Grimm-Jaud", 491601233}, {"Schmidt", 49133318}, {"Huber", 4900013}};
```



Die Hashfunktion bildet den Schlüssel in konstanter Zeit auf den Index eines Buckets ab.

# Ungeordnete assoziative Container

## Die lange Geschichte der ungeordneten Container in C++

- Die ungeordnete Container haben es nicht mehr in den C++98 Standard geschafft.
- Dieser Mangel wurde durch die geordneten Container kompensiert.
- Seit C++11 sind sie im C++-Standard.

- Performance Matters:

- Die Zugriffszeit der
  - geordnete assoziative Container ist **logarithmisch**.
  - ungeordnete assoziative Container ist **konstant**.

# Die Hashfunktion

## Die Hashfunktion

- gibt es für die built-in Datentypen.
- gibt es für `std::string` und `std::wstring`.
- ist gut, falls sie die Schlüssel mit wenig Kollisionen gleichmässig auf die Buckets verteilt.
- lässt sich für eigene Datentypen definieren.

```
struct MyHash{  
    std::size_t operator() (MyInt m) const{  
        std::hash<int> hasVal;  
        return hasVal(m.val);  
    }  
};
```

# Ungeordnete assoziative Container

- **Kollisionen:**

- Ungeordnete assoziative Container speichern ihre Schlüssel in den Buckets.
- Unterschiedliche Schlüssel mit gleichem Hashwert können im gleichen Bucket landen.
- Der Zugriff auf den Bucket ist konstant, die Suche im Bucket in der Regel linear.

- **Kapazität:**

- Anzahl der Buckets

- **Ladefaktor:**

- durchschnittliche Anzahl der Elemente je Bucket

- **Rehashing:**

- Neue Buckets werden in der Regel erzeugt, wenn der Ladefaktor größer als 1 ist.




# Ungeordnete assoziative Container



- Beispiele:
  - `unorderedMap.cpp`
  - `unorderedMapMultimap.cpp`
  - `unorderedMapHash.cpp`
  - `unorderedOrderedContainerPerformance.cpp`
  - `unorderedSetHashInfo.cpp`
- Aufgaben:
  - Verwenden Sie in dem Programm `unorderedMapMultimap.cpp` `std::unordered_set` statt `std::unordered_map` und `std::unordered_multiset` statt `std::unordered_multimap`.
    - Lösung: `unorderedSetMultiset.cpp`

# Templates

Templates sind Schablonen für Klassen (Klassen-Templates) oder Funktionen (Funktions-Templates), aus denen der Compiler konkrete Klassen oder Funktionen erzeugt.

- Klassen-Templates oder auch Funktions-Templates beschreiben Familien von Klassen oder Funktionen.
- Templates spielen eine wichtige Rolle in der Entwicklung generischer Bibliotheken.  Standard Template Library

# Funktions-Templates

Ein Funktions-Template wird definiert, indem der Funktionsdefinition das Schlüsselwort `template`, gefolgt von den Typ- oder Nichttyp-Parametern, vorangestellt wird.

- Durch die Schlüsselwörter `class` oder `typename` werden die Parameter deklariert.
- Für den ersten Typ-Parameter hat sich der Name `T` etabliert.
- Die Parameter können in gewohnter Weise im Funktionskörper verwendet werden.

```
template <typename T>  
void xchg(T&x , T&y){  
...
```

```
template <int N>  
int nTimes(int n){  
...
```

# Funktions-Templates: Instanziierung

Der Prozess, die Template-Parameter durch konkrete Argumente zu ersetzen, wird als Instanziierung des Templates bezeichnet.

- Der Compiler
  - erzeugt automatisch eine Instanz eines Funktions-Templates aufgrund der Argumente.
  - kann nur automatisch ein Funktions-Template erzeugen, falls er die Template-Argumente ableiten kann.

```
template <typename T>  
void xchg(T& x, T& y){ ...
```

```
int a, b;  
xchg(a, b);
```

```
template <int N>  
int nTimes(int n){ ...
```

```
int n = 5;  
nTimes<10>(n);
```



Falls der Compiler die Template-Argumente aus den Funktionsargumenten nicht ableiten kann, müssen diese explizit angegeben werden.

# Funktions-Templates: Überladung

Funktions-Templates können überladen werden.

- Es gelten dabei die folgenden Regeln:
  1. Templates unterstützen keine automatische Typkonvertierung.
  2. Ist eine freie Funktion eine genauso gute oder bessere Wahl wie ein Funktions-Template, wird die freie Funktion vorgezogen.
  3. Durch einen Aufruf der Form `func<type>(...)` mit einem Template-Argument `type` wird explizit ein Funktions-Template aufgerufen.
  4. Durch einen Aufruf mit leerer Template-Argumentliste `func<>(...)` zieht der Compiler nur Funktions-Templates in Betracht.

# Funktions-Templates



- Beispiele:
  - `templateFunctionsTemplates.cpp`
  - `templateFunctionsTemplatesOverloading.cpp`

# Klassen-Templates

Ein Klassen-Template wird definiert, indem der Klassendefinition das Schlüsselwort `template`, gefolgt von den Typ- oder Nichttyp-Parameter, vorangestellt wird.

- Durch die Schlüsselwörter `class` oder `typename` werden die Parameter deklariert.
- Die Parameter können in gewohnter Weise im Klassenkörper verwendet werden.
- Die Memberfunktionen der Klassen-Templates können innerhalb oder außerhalb der Klasse definiert werden.

```
template <typename T, int N>  
class Array{  
    T elem[N];  
    ...  
}
```

# Klassen-Templates: Instanziierung

Der Prozess, die Template-Parameter durch konkrete Argumente zu ersetzen, wird als Instanziierung des Templates bezeichnet.

- Ein Klassen-Template kann im Gegensatz zu einem Funktions-Template seine Argumente nicht automatisch ableiten. ➡ Jedes Template-Argument muss explizit in spitzen Klammern angegeben werden.

```
template <typename T>  
void xchg(T& x, T&y){ ...
```

```
int a, b;  
xchg(a, b);
```

```
template <typename T, int N>  
class Array{ ...
```

```
Array<double, 10> doubleArray;  
Array<Account, 1000> accountArray;
```



# Klassen-Templates: Generische Memberfunktionen

Generische Memberfunktionen sind Funktions-Templates, die in Klassen oder Klassen-Templates verwendet werden.

- Generische Memberfunktionen können innerhalb oder außerhalb der Klasse definiert werden.

```
template <class T, int N>
class Array{
public:
    template <class T2>
    Array<T,N>& operator = (
    ...
```

```
template <class T, int N>
class Array{
public:
    template <class T2>
    Array<T, N>& operator = (const Array<T2, N>& a);
    ...
};
template<class T, int N>
template<class T2>
    Array<T, N>& Array<T, N>::operator = (const Array<T2, N>& a{
    ...
```



Der Destruktor und der Copy-Konstruktor können keine Templates sein.

# Klassen-Templates: Vererbung

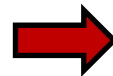
Klassen und Klassen-Templates können in allen Kombinationen voneinander abgeleitet werden.



- Falls eine Klasse oder ein Klassen-Template von einem Klassen-Template abgeleitet wird, stehen in der abgeleiteten Klasse bzw. dem Klassen-Template nicht automatisch die Attribute und Memberfunktionen der Basisklasse zur Verfügung.

```
template <typename T>
struct Base{
    void func() { ...
};

template <typename T>
struct Derived: Base<T>{
    void func2(){
        func();          // ERROR
    ...
}
```



## 3 Lösungen:

- Aufruf durch den this-Zeiger qualifizieren: `this->func()`
- Name der Memberfunktion durch `using` einführen: `using Base<T>::func`
- Memberfunktion der Basisklasse direkt aufrufen: `Base<T>::func()`


# Klassen-Templates



- Beispiele:
  - `templateClassTemplate.cpp`
  - `templateClassTemplateMethods.cpp`
  - `templateClassTemplateInheritance.cpp`

# Klassen-Templates: Freunde

Freunde eines Klasse-Templates haben Zugriff auf alle Mitglieder der Klasse.

- Eine Klasse oder eine Klassen-Template kann Freundschaften zu Klassen oder Klassen-Templates, Funktionen- oder Funktions-Templates, aber auch Typen aussprechen.
  - Regeln:
    1. Die Deklaration von Freuden können an beliebiger Stelle in der Klassendeklaration stehen.
    2. Die Zugriffsrechte, unter der Freund-Deklarationen stehen, besitzen keine Bedeutung.
    3. Freundschaft wird nicht vererbt.
    4. Freundschaften sind nicht transitiv.
-  Ein Freund besitzt volle Zugriffsrechte auf die Klasse.

# Klassen-Templates: Allgemeine Freunde

Ein Klassen-Template oder eine Klasse kann seine bzw. ihre Freundschaft gegenüber jeder Instanz eines Klassen- oder Funktions-Templates aussprechen.

```
template <typename T> int myFriendFunction(T);  
template <typename T> class MyFriend;  
  
template <typename T>  
class GrantingFriendshipAsClassTemplate{  
    template <typename U> friend int myFriendFunction(U);  
    template <typename U> friend class MyFriend; ...  
}
```



Wenn ein Klassen-Template Freundschaft zu einem weiteren Template ausspricht, müssen sich die Namen der Template-Parameter unterscheiden.

# Klassen-Templates: Spezielle Freunde

Eine spezielle Freundschaft entsteht dann, wenn die Freundschaft vom Typ des Template-Parameters abhängig ist.

```
template <typename T> int myFriendFunction(T);  
template <typename T> class MyFriend;
```

```
template <typename T>  
class GrantingFriendshipAsClassTemplate{  
    friend int myFriendFunction<>(double);  
    friend class MyFriend<int>  
    friend class MyFriend<T>;  
};
```



Wenn der Name des Template-Parameters identisch mit dem des befreundeten Klassen-Templates ist, besteht die Freundschaft zwischen Instanzen des gleichen Typs.

# Klassen-Templates: Freund zu Typen

Ein Klassen-Template kann seine Freundschaft zu einem Typ-Parameter aussprechen.

```
template <typename T>
class Array{
    friend T;
    ...
};

Array<Account> myAccount;
```


# Klassen-Templates: Freunde



- Beispiele:
  - `templateClassTemplateGeneralFriendship.cpp`
  - `templateClassTemplateSpecialFriendship.cpp`
  - `templateClassTemplateTypeFriendship.cpp`



# Templates: Alias-Templates

Alias-Templates oder auch Template-Typedefs erlauben es, Synonyme auf teilweise gebundene Templates zu vergeben.  Partielle Spezialisierung von Templates

```
template <typename T, int Line, int Col>
class Matrix{
    ...
};

template <typename T, int Line>
using Square = Matrix<T, Line, Line>;

template <typename T, int Line>
using Vector = Matrix<T, Line, 1>;
```

Matrix<int, 5, 3> ma;  
Square<double, 4> sq;  
Vector<char, 5> vec;



Alias-Templates können nicht weiter spezialisiert werden.

# Template-Parameter

C++ unterstützt drei verschiedene Arten von Template-Parameter

## 1. Typ-Parameter

```
std::vector<int> vec = {1, 2, 3, 4, 5};
```

## 2. Nichttyp-Parameter

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

## 3. Template-Template-Parameter

```
template <typename T, template <typename, typename> class Cont>  
class Matrix{ ...  
Matrix<int, std::vector> myIntVec;
```

# Template-Parameter: Typen

Typ-Parameter stellen den Standardfall für Template-Argumente dar.

- Diese sind in der Regel Klassentypen und fundamentale Typen.

```
class Account;
```

```
template <typename T>  
class ClassTemplate{};
```

```
ClassTemplate<int> clTempInt;  
ClassTemplate<double> clTempDouble;  
ClassTemplate<Account> clTempAccount;
```

```
ClassTemplate<std::string> clTempString;
```

# Template-Parameter: Nichttypen

Nichttyp-Parameter sind Template-Parameter, die zur Kompilierzeit evaluiert werden können.

- Die folgenden Typen sind erlaubt:
  - Ganzzahlige Konstanten und Aufzählungen
  - Zeiger auf Objekte, Funktionen und auf Attribute und Memberfunktionen von Klassentypen
  - Referenzen auf Objekte und Funktionen
  - `std::nullptr_t`-Konstante



Fließkommazahlen und Strings sind als Nichttyp-Parameter nicht erlaubt.

# Dependent Names

Namen können vom Template Parameter abhängen.

➡ Dependent Names

```
template <typename T>
struct X: B<T>{
    typename T::A* pa;

    void f(B<T>* pb) {
        static int i = B<T>::i;
        pb->j++;
    }
};
```

*Dependent Names* werden bei der Template Instanziierung aufgelöst.  
Nicht *Dependent Names* werden bei der Template Definition aufgelöst.

# Dependent Names

Ein von einem Template Parameter  $T$  abhängiger qualifizierter Name  $T :: x$  kann ein

- Typ sein.
- Nichttyp sein.
- Template sein.

➔ Der Compiler nimmt per Default an, dass  $T :: x$  ein Nichttyp ist.

Der Compiler muss davon überzeugt werden, dass  $T :: x$  ein Typ oder Template ist.

# Dependent Names

Der abhängig Name ist ein Typ ➡ `typename`.

```
template <typename T>
void test() {
    std::vector<T>::const_iterator* p1;           // 1
    typename std::vector<T>::const_iterator* p2; // OK
}
```

Ohne `typename` wird der Ausdruck 1 als Multiplikation interpretiert.

# Dependent Names

Der abhängig Name ist ein Template ➡ `template`.

```
template<typename T>
struct S{
    template <typename U> void func(){}
}
template<typename T>
void func2(){
    S<T> s;
    s.func<T>();           // 1
    s.template func<T>();  // OK
}
```

Ohne `template` wird der Ausdruck 1 als kleiner (<) interpretiert.



# Template-Parameter



- Beispiele:
  - `templateTypParameter.cpp`
  - `templateNotTypeParameter.cpp`
  - `templateTemplateTemplatesParameter.cpp`

# Template-Parameter: Variadic-Templates

Ein Variadic-Template ist ein Template, das beliebig viele Parameter annehmen kann.

```
template <typename ... Args>  
void variadicTemplate(Args ... args) { . . . }
```

## ■ Parameter-Pack:

- Durch die Ellipse (...) wird Args- bzw. args zum Parameter-Pack.
- Args ist ein Template-Parameter-Pack, args ein Funktions-Parameter-Pack.
- Parameter-Packs können nur gepackt und entpackt werden.
- Steht die Ellipse links von Args, wird das Parameter-Pack gepackt, rechts von Args, wird es entpackt.



Aufgrund der Funktionsargumente kann der Compiler die Template-Argumente ableiten.

# Template-Parameter: Variadic-Templates

Variadic-Templates werden häufig in der Standard Template Library verwendet:

- `sizeof-Operator`, `std::tuple`, `std::thread`
- Die Verwendung von Parameter-Packs folgt einem typischen Muster:
  - Führe eine Operation rekursiv auf dem ersten Element des Parameter-Packs aus und reduziert nach jeder Iteration das Parameter-Pack um sein erstes Element.
  - Damit endet die Rekursion nach endlich vielen Schritten.

```
template<>
struct Mult<>{ ... }

template<int i, int ... tail >
struct Mult<i, tail ...>{ ...
```

# Template-Parameter: Variadic-Templates



- Beispiele:
  - `templateVariadicTemplates.cpp`
  - `templatePerfectForwarding.cpp`

# Template-Argumente: Argument-Ableitung

Template-Argumente lassen sich nur automatisch für Funktions-Templates bestimmen.

- Der Compiler leitet die Template-Argumente aus den Funktionsargumenten ab. ➡ Funktions-Templates fühlen sich wie gewöhnliche Funktionen an.
- Konvertierungen:
  - Bei der Bestimmung der Template-Parameter aus den Funktionsargumenten wendet der Compiler nur einfache Konvertierungen an.
  - Der Compiler entfernt gegebenenfalls das äußere `const/volatile` von den Funktionsargumenten und konvertiert C-Arrays oder Funktionen in Zeiger.

# Template-Argumente: Argument-Ableitung

Da keine Konvertierungen stattfinden, müssen die Typen der Funktionsargumente, die die Template-Parameter festlegen, identisch sein.

```
template <typename T>
bool isSmaller(T fir, T sec){
    return fir < sec;
}
isSmaller(1, 5LL);    // ERROR int != long long int
```

- Wird ein zweiter Template-Parameter für das zweite Funktionsargument verwendet, nimmt der Compiler den Funktionsaufruf an.

```
template <typename T, typename U>
bool isSmaller(T fir, U sec){
    return fir < sec;
}
isSmaller(1, 5LL);    // OK
```

# Template-Argumente: Argument-Ableitung

## Explizite Template-Argumente

- sind notwendig, wenn sich die Template-Parameter nicht aus den Funktionsargumenten ableiten lassen.
- werden benötigt, wenn eine spezifische Instanz eines Funktions-Templates verwendet werden soll.

```
template <typename R, typename T, typename U>  
R add(T fir, U sec){  
    return fir * sec;  
}  
add<long long int>(1000000, 1000000LL);
```



Fehlende Template-Parameter werden automatisch aus den Funktionsargumenten abgeleitet.

# Template-Argumente: Argument-Ableitung

## Automatischer Rückgabotyp

- Durch `auto` und `decltype` lassen sich in der alternativen  
Funktionssyntax Funktions-Templates schreiben, die automatisch ihren  
Rückgabetype bestimmen.

```
template< typename T1, typename T2>  
auto add(T1 fir, T2 sec) -> decltype(fir + sec) {  
    return fir + sec;  
}  
auto res = add(1.2, 5);
```

- `auto`: leitet die Syntax für den verzögerten Rückgabotyp ein
- `decltype`: erklärt den Rückgabotyp



In C++14 ist `decltype(fir + sec)` nicht mehr notwendig.



# Template-Argumente: Default Argumente

## Default-Template-Argumente

- Können für Template-Parameter von Klassen- und Funktions-Templates vorgegeben werden
- Sobald in Template-Parameter ein Default-Argument erhält, müssen alle weiteren Template-Parameter auch Default-Argumente besitzen

```
template <typename T, typename Pred = std::less<T>>  
bool isSmaller(T fir, T sec, Pred pred = Pred()) {  
    return pred(fir, sec);  
}
```

# Template-Argumente



- Beispiele:
  - `templateArgumentDeduction.cpp`
  - `templateAutomaticReturnType.cpp`
  - `templateDefaultArgument.cpp`

# Templates: Spezialisierung

Templates beschreiben das Verhalten von Familien von Klassen und Funktionen.

- Oft ist es notwendig, dass besondere Typen oder Nichttypen besonders behandelt werden.
- Dazu können Templates vollständig, im Falle von Klassen auch teilweise spezialisiert werden.
- Die Memberfunktionen und Attribute von Spezialisierungen müssen nicht identisch sein.
- Neben den allgemeinen oder primären Template können auch partielle und vollständige spezialisierte Templates koexistieren.



Der Compiler zieht vollständige den partiellen, partiellen den primären Templates vor.

# Templates: Primäre Template

Das primäre Template muss zuerst deklariert werden, bevor die Deklaration der partiellen oder vollständigen Spezialisierungen des Templates folgt.

- Falls das primäre Template nicht benötigt wird, ist eine Deklaration ausreichend.

```
template <typename T, int Line, int Column>  
class Matrix;
```

```
template <typename T>  
class Matrix<T, 3, 3>{};
```

```
template <>  
class Matrix<int, 3, 3>{};
```

# Templates: Partielle Spezialisierung

## Die partielle Spezialisierung eines Templates

- wird nur für Klassen-Templates unterstützt.
- besitzt sowohl Template-Argumente als auch Template-Parameter.

```
template <typename T, int Line, int Column>
class Matrix{};
template <typename T>
class Matrix<T, 3, 3>{};
template <int Line, int Column>
class Matrix<double, Line, Column>{};

Matrix<int, 3, 3> m1;           // class Matrix<T, 3, 3>
Matrix<double, 10, 10> m2;     // class Matrix<double, Line, Column>
Matrix<std::string, 2, 2> m3;  // class Matrix
```

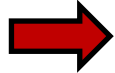
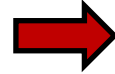
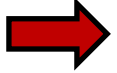
# Templates: Partielle Spezialisierung

Regeln für die partielle Spezialisierung:


1. Der Compiler wendet die partielle Spezialisierung an, wenn die Argumente der instanziierten Klasse eine Teilmenge der Liste der Template-Argumente sind.
2. Die unspezifizierten Template-Argumente müssen als Template-Parameter angegeben werden.
3. Die Länge und Reihenfolge der Liste der Template-Argumente muss der der Template-Parameter des primären Templates entsprechen.
4. Werden für die Template-Parameter Default-Argumente verwendet, müssen diese nicht in der Liste der Template-Argumente angegeben werden. Default-Argumente sind nur für primäre Templates zulässig.

# Templates: Partielle Spezialisierung

Drei Regeln für die Auswahl der richtigen Spezialisierung:

1. Der Compiler findet nur eine Spezialisierung.  Er erzeugt eine Instanz aus dieser.
2. Der Compiler findet mehrere Spezialisierungen.  Er verwendet die am meisten spezialisierte.  
Findet der Compiler keine am meisten spezialisierte Klassen-Templates, kommt es zu einem Compilerfehler.
3. Der Compiler findet keine Spezialisierung.  Er erzeugt eine Instanz aus dem primären Template.

■ Ein Template A ist mehr spezialisiert, als ein Template B:

- Alle Argumente, die A annehmen kann, kann auch B annehmen.
-  ann Argumente annehmen, die A nicht annehmen kann.

# Templates: Vollständige Spezialisierung

Bei einer vollständigen Spezialisierung eines Templates sind alle Template-Parameter durch Template-Argumente vorgegeben.

- Die Liste der Template-Parameter ist auf eine leere Liste reduziert.

```
template <typename T>
struct Type{
    std::string getName() const {return "unknown";};
};
template <>
struct Type<Account>{
    std::string getName() const {return "Account";};
};
```



# Templates: Vollständige Spezialisierung

Werden die Memberfunktionen eines vollständig spezialisierten Klassen-Templates außerhalb der Klasse definiert, folgen die Template-Argumente unmittelbar auf den Namen der Klasse in spitzen Klammern.

```
template <typename T, int Line, int Column>{  
    struct Matrix;
```

```
template <>  
struct Matrix<int, 3, 3>{  
    int numberOfElements() const;  
};
```



```
// template <>  
int Matrix<int, 3, 3>::numberOfElements() const {  
    return 3 * 3;  
};
```

# Templates: Spezialisierung



- Beispiele:
  - `templateSpecialization.cpp`
  - `templateSpecializationPrimary.cpp`
  - `templateSpecializationFull.cpp`
  - `templateSpecializationExternal.cpp`

# Templates: CRTP

## CRTP

- steht für das **C**uriously **R**ecurring **T**emplate **P**attern
- eine Klasse wird von einem Klassen-Template abgeleitet, das sich selbst als Klassen-Template besitzt

```
template<class T>
class Base{
    ...
};

class Derived : public Base<Derived>{
    ...
};
```



CRTP erlaubt statischen Polymorphismus.

# Templates: CRTP

## Mixins mit CRTP

- beim Mixin wird neuer Code zu einer Klasse hinzugemischt
- die Klasse `std::enable_shared_from_this` wendet CRTP an

## Typischer Anwendungsfall

- eine Klasse wird um die Fähigkeit erweitert, ihre Instanzen auf Gleichheit und Ungleichheit zu vergleichen

# Templates: CRTP



- Beispiele:
  - `templateCRTP.cpp`
  - `templateCRTPEquality.cpp`

# Besondere Anforderungen

Sicherheitskritische Systeme

Hohe Performanz

**Eingeschränkte Ressourcen**

Mehrere Aufgaben gleichzeitig

# Memberfunktionen anfordern und unterdrücken

- Der Compiler erzeugt bei Bedarf sehr viele spezielle Memberfunktionen:
  - Default-Konstruktor und Destruktor
  - (Copy/Move)-Konstruktor, (Copy/Move)-Zuweisungsoperator
  - Operatoren `new` und `delete` in der einfachen Form und für C-Arrays
- Mit den Schlüsselwörtern `default` und `delete` lässt sich das Erzeugen und Unterdrücken von Memberfunktionen explizit steuern.
- Während eine als `default` deklarierte Memberfunktion diese vom Compiler anfordert, unterdrückt `delete` eine Memberfunktion, die zur Verfügung stünde.



Der Programmierer definiert das Interface, der Compiler sorgt für die Implementierung.

# Memberfunktionen anfordern: `default`

Der Compiler erzeugt Memberfunktionen nach folgenden Charakteristiken:

- Sie besitzen `public`-Zugriffsrechte und sind nicht virtuell.
  - Der Copy-Konstruktor und Copy-Zuweisungsoperator erwarten konstante Lvalue-Referenzen.
  - Der Move-Konstruktor und Move-Zuweisungsoperator erwarten nicht-konstante Rvalue-Referenzen.
- 
- Die angeforderten Memberfunktionen dürfen nicht als `explicit` deklariert werden und keine Ausnahmespezifikationen besitzen.



# Memberfunktionen unterdrücken: `delete`

- Durch `delete` lässt sich rein deklarativ erklären, dass eine automatisch vom Compiler erzeugte Memberfunktion nicht zur Verfügung steht.
- In Kombination mit `default` lassen sich Klassen erzeugen, deren Objekte
  - Nicht kopiert werden können.
  - Nur auf dem Stack angelegt werden können.
  - Nur auf dem Heap angelegt werden können.



`delete` kann auch auf Funktionen angewandt werden.

# default und delete



- Beispiele:
  - `default.cpp`
  - `delete.cpp`
- Aufgaben:
  - Schreiben Sie ein Klassen-Template, dass sich nur mit einem `int`-Wert erzeugen lässt.

```
OnlyInt(5); // ok
OnlyInt(5L); // ERROR
```
  - Lösung: `delete.cpp`

# Rvalues

Rvalues sind

- temporäre Objekte.
- Objekte ohne Namen.
- Objekte, von denen die Adresse nicht bestimmt werden kann.

➡ Der Rest sind Lvalues

- Lvalues können auf der linken Seite einer Zuweisung stehen. Rvalues stehen auf der rechten Seite einer Zuweisung.

```
int lvalue = 1998;  
lvalue = 2011;  
const int lvalue2 = 2011;  
lvalue2 = 2011;           // ERROR
```

```
int defInt = int{};  
int res= 2000 + 11;  
auto func = []{std::cout << "2011" << std::endl;};
```

# Lvalue- und Rvalue-Referenzen

Lvalue-Referenzen werden durch ein **&**-Symbol deklariert.

Rvalue-Referenzen werden durch zwei **&&**-Symbole deklariert.

- Lvalues können nur an Lvalue-Referenzen, Rvalues können an Rvalue-Referenzen oder an **konstante** Lvalue-Referenzen gebunden werden.

```
MyData myData;
```

```
MyData& lvalueRef(myData);
```

```
MyData&& rvalueRef(MyData());
```

```
const MyData& constLValueRef(MyData());
```



Das Binden eines Rvalues an eine Rvalue-Referenz besitzt höhere Priorität.

# Rvalue-Referenzen: Anwendungen

- Move-Semantik
  - Billiges Verschieben eines Objekts statt teurem Kopieren.
  - Keine Speicherallokation und Deallokation.
  - Nicht kopierbare aber verschiebbare Objekte können *by Value* übergeben werden.
- Perfect Forwarding
  - Reiche ein Objekt mit seinen identischen Objekteigenschaften weiter.

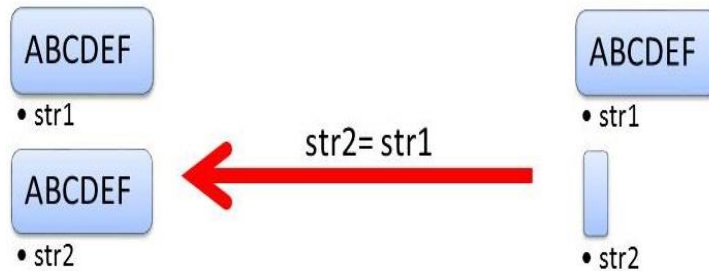
# Copy- versus Move-Semantik

- Eine Klasse unterstützt die **Copy-Semantik**, wenn sie einen Copy-Konstruktor und einen Copy-Zuweisungsoperator anbietet.
- Eine Klasse unterstützt die **Move-Semantik**, wenn sie einen Move-Konstruktor und einen Move-Zuweisungsoperator anbietet.
- Besitzt eine Klasse einen Copy-Konstruktor, sollte sie auch einen Copy-Zuweisungsoperator anbieten. Entsprechendes gilt für den Move-Konstruktor und Move-Zuweisungsoperator.

# Copy- versus Move-Semantik

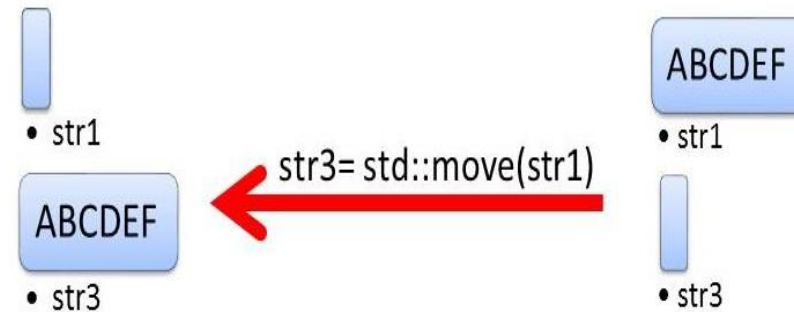
## Copy

```
string str1("ABCDEF");  
string str2;  
str2 = str1;
```



## Move

```
string str1("ABCDEF");  
string str3;  
str3 = std::move(str1);
```



# Copy versus Move: `std::swap`

```
std::vector<int> a, b;  
swap(a, b);
```

```
template <typename T>  
void swap(T& a, T& b){  
    T tmp(a);  
    a = b;  
    b = tmp;  
}
```

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

`T tmp(a);`

- Allokiert `tmp` und jedes Element von `tmp`.
- Kopiert jedes Element von `a` nach `tmp`.
- Deallokiert `tmp` und jedes Element von `tmp`.

`T tmp(std::move(a));`

- Verbiegt den Zeiger von `tmp` auf `a`.



# Move-Semantik: `std::move`

Die Funktion `std::move` verschiebt explizit ihre Ressource.

- `std::move`
  - benötigt den Header `<utility>`
  - konvertiert sein Argument in eine Rvalue Referenz
  - der Compiler wendet auf die Rvalue Referenz Move-Semantik an
  - ist unter der Decke ein `static_cast` auf eine Rvalue Referenz  
`static_cast<std::remove_reference<decltype(arg)>::type&&>(arg);`



Copy-Semantik ist ein Fallback für die Move-Semantik.

# Move-Semantik: STL

Jeder Container der STL und `std::string` erhält zwei neue Memberfunktionen um Move-Semantik anzubieten.

- Move-Konstruktor
- Move-Zuweisungsoperator

- Diese Memberfunktionen nehmen ihre Argumente als **nicht-konstante** Rvalue Referenzen an.

- Beispiel

```
vector{  
    vector(vector&& vec);           // Move-Konstruktor  
    vector& operator = (vector&& vec); // Move-Zuweisungso.  
    vector(const vector& vec);      // Copy-Konstruktor  
    vector& operator = (const vector& vec); // Copy-Zuweisungso.  
    . . .
```



Der klassische Copy-Konstruktor und Copy-Zuweisungsoperator nimmt seine Argumente als **konstante** Lvalue Referenz an.

# Move-Semantik: Eigene Datentypen

Eigene Datentypen können Move- und Copy-Semantik anbieten.

- **Beispiel:**

```
class MyData{  
    MyData(MyData&& m) = default;  
    MyData& operator = (MyData&& m) = default;  
    MyData(const MyData& m) = default;  
    MyData& operator = (const myData& m) = default;  
};
```

- Die Move-Semantik besitzt höhere Priorität als die Copy-Semantik

# Automatisch erzeugte Memberfunktionen

Die sechs speziellen Memberfunktionen

- Default-Konstruktor und Destruktor
- Copy-Konstruktor und Copy-Zuweisungsoperator
- Move-Konstruktor und Move-Zuweisungsoperator

Jede der sechs speziellen Memberfunktionen wird vom Compiler automatisch erzeugt, wenn alle Attribute der Klasse und ihrer Basisklassen diese spezielle Memberfunktionen anbietet.

# Automatisch erzeugte Memberfunktion

compiler implicitly declares

	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

user declares

von [Howard Hinnant](#)

- user-declared: eine Memberfunktion, die deklariert wird (definiert, defaulted oder deleted)
- defaulted: eine Memberfunktion, die der Compiler erzeugt oder mittels `default` angefordert wird

# Move-Semantik



- Beispiele:
  - `rvalueReference.cpp`
  - `copyMoveSemantic.cpp`
  - `swap.cpp`
  - `bigArray.cpp`
- Aufgaben:
  - In dem Programm `bigArray.cpp` wird ein `BigArray` mit 10 Milliarden Elementen auf einen `std::vector` geschoben.
    - Übersetzen Sie das Programm und messen Sie die Performanz.
    - Erweitern Sie `BigArray` um Move-Semantik und führen Sie das Programm nochmals aus. Wie groß ist der Performanzgewinn?
      - Lösung: `bigArray.cpp`
- Weitere Informationen:
  - [Rvalue References Explained](#) von Thomas Becker

# Perfect Forwarding

Grundproblem: Eine Funktion möchte ihre Daten als Referenz erhalten.

```
struct BigData{
    BigData(vector<int>& d): data(d){}           // Lvalue-Ref
    BigData(const vector<int>& d): data(d){}      // const Lvalue-Ref
    ...
};

struct BigData2{
    template<typename T>
    BigData2(T&& d): data(std::forward<T>(d)){} // Lvalue- und Rvalue-Ref
};
```

➡ Bei n Parameter sind  $2^n$  Funktionen notwendig.

# Perfect Forwarding

Perfekt Forwarding ermöglicht es, Funktions-Templates zu schreiben, die ihre Argumente identisch weiterreichen.

➡ Die Lvalue und Rvalue-Eigenschaften eines Objektes werden respektiert.

- `std::forward`
  - Stroustrup: "*... a heretofore unsolved problem in C++.*"
  - mächtiges Werkzeug des Bibliotheksautor für generische Funktions-Templates
  - typischer Anwendungsfall: Fabrikfunktionen oder Konstruktoren



# Perfect Forwarding



- Beispiele:
  - `perfectForwarding.cpp`
- Aufgaben:
  - Verwenden Sie die generische Fabrikfunktion aus `perfectForwarding.cpp` und erweitern sie diese zu einem Variadic Template. Wenden Sie sie auf möglichst viele verschiedene Typen. Die Initialisierungswerte sollen Lvalue- und Rvalue-Werte sein.
    - Lösung: `perfectForwarding.cpp`
- Weitere Informationen:
  - [perfect forwarding](#) von Thomas Becker

# Speicherverwaltung

- C++ erlaubt das dynamische Allokieren und Freigeben von Speicher.
- Dynamischer Speicher (Heap) wird explizit durch den Programmierer angefordert und wieder freigegeben.
- Zum Anfordern des Speichers stehen die Operatoren `new` und `new[]`, zur Freigabe die Operatoren `delete` und `delete[]` zur Verfügung.
- Der Compiler verwaltet seinen Speicher automatisch auf dem Heap.



Smart Pointer verwalten ihren Speicher automatisch.

# Speicherallokation: `new`

- Ermöglicht es, Speicher für die Instanzen eines Typs dynamisch zu allozieren

```
int* i = new int;  
double* d = new double(10.0);  
Point* p = new Point(1.0, 2.0);
```

- `new` bewirkt, dass zuerst der Speicher alloziert und dann das Objekt initialisiert wird
- Die Argumente in runden Klammern sind die Argumente für den Konstruktor
- `new` liefert als Ergebnis einen Zeiger auf den passenden Typ zurück
- Ist das allozierte Objekt Instanz einer abgeleiteten Klasse, werden mehrere Konstruktoren aufgerufen

# Speicherallokation: `new [ ]`

- Ermöglicht es, Speicher für ein C-Array zu allozieren

```
double* d = new double[5];  
Point* p = new Point[10];
```

- Die Klasse der zu allozierenden Objekte muss einen Default-Konstruktor besitzen
- Der Default-Konstruktor wird für jedes zu instanziiierende Objekt aufgerufen



Die STL-Container und der C++-String verwalten ihren Speicher automatisch.

# Speicherallokation: Placement-new

- Ermöglicht es, Objekte oder C-Arrays in einem vorgegebenen Speicherbereich zu instanziiieren.

```
char* memory = new char[sizeof(Account)]; // allocate std::size_t
Account* acc = new(memory) Account;       // instantiate acc in memory
```

- Der Header `<new>` ist notwendig.
- Kann global und für eigene Typen überladen werden
- Typische Anwendungsfälle:
  - Explizite Speicherallokation
  - Vermeidung von Ausnahmen
  - Debugging

# Fehlgeschlagene Allokation

- Schlägt eine Allokation fehl, löst `new` oder `new []` eine `std::bad_alloc`-Ausnahme aus
- Wird `new` oder `new []` mit der Konstante `std::nothrow` aufgerufen, gibt eine fehlgeschlagene Allokation einen Null-Zeiger zurück.

```
char* c = new(std::nothrow) char[10];  
if (c){  
    delete c;  
}  
else{  
    // an error occurred  
}
```

# Speicherfreigabe: delete

- Mit dem Operator `delete` wird der mit dem Operator `new` allozierte Speicher wieder freigegeben

```
Point* p = new Point(1.0,2.0);  
delete p;
```

- Falls das zerstörte Objekt zu einer abgeleiteten Klasse gehört, werden gegebenenfalls mehrere Destruktoren aufgerufen
- Nachdem der Speicher freigegeben ist, ist der Zugriff auf das Objekt undefiniert

Wird ein mit `new` alloziertes Objekt mit `delete []` wieder freigegeben, stellt dies ein undefiniertes Verhalten dar.

# Speicherfreigabe: `delete [ ]`

- Für die Freigabe eines C-Arrays, das mit `new [ ]` alloziert wurde, ist der Operator `delete [ ]` zuständig

```
Point* p = new Point[15];  
delete [ ] p;
```

- Bei der Verwendung von `delete [ ]` werden im Gegensatz zu `delete` alle Destruktoren aufgerufen

Wird ein mit `new [ ]` alloziertes C-Array mit `delete` wieder freigegeben, stellt dies undefiniertes Verhalten dar.



# Speicherverwaltung



- Beispiele:
  - `raii.cpp`
  - `overloadOperatorNewAndDelete.cpp`
  - `myNew.hpp`
  - `myNew2.hpp`
  - `myNew3.hpp`
  - `overloadOperatorNewAndDelete2.cpp`
  - `myNew4.hpp`
  - `myNew5.hpp`

# Speicherverwaltung



- Aufgaben:
  - `operator new` und `delete` lassen sich an die eigenen Bedürfnisse anpassen.
    - Studieren Sie die Programme `overloadOperatorNewAndDelete.cpp` und `overloadOperatorNewAndDelete2.cpp`.
    - Eine detaillierte Erläuterung finden Sie auf [www.grimm-jaud.de](http://www.grimm-jaud.de).
- Weitere Informationen:
  - [operator new and delete](#)

# STL: Sequentielle Container

Kriterium	<code>std::array</code>	<code>std::vector</code>	<code>std::deque</code>	<code>std::list</code>	<code>std::forward_list</code>
<b>Größe</b>	statisch	dynamisch	dynamisch	dynamisch	dynamisch
<b>Implementierung</b>	statisches Array	dynamisches Array	Sequenz von Arrays	doppelt verkettete Liste	einfach verkettete Liste
<b>Zugriff</b>	wahlfrei	wahlfrei	wahlfrei	vor- und rückwärts	vorwärts
<b>Optimiert für Operationen am</b>		Ende $O(1)$	Anfang und Ende $O(1)$	<ul style="list-style-type: none"> <li>Anfang und Ende <math>O(1)</math></li> <li>Überall <math>O(1)</math></li> </ul>	<ul style="list-style-type: none"> <li>Anfang <math>O(1)</math></li> <li>Überall <math>O(1)</math></li> </ul>
<b>Speicherreservierung</b>		ja	nein	nein	nein
<b>Speicherfreigabe</b>		<code>shrink_to_fit()</code>	<ul style="list-style-type: none"> <li>manchmal</li> <li><code>shrink_to_fit()</code></li> </ul>	immer	immer
<b>Stärken</b>	<ul style="list-style-type: none"> <li>keine Speicherallokation</li> <li>minimale Speicheranforderungen</li> </ul>	95% Prozent Lösung	Einfügen und Löschen am Anfang und Ende	Einfügen und Löschen an beliebiger Position	<ul style="list-style-type: none"> <li>Schnelles Einfügen und Löschen</li> <li>minimale Speicheranforderungen</li> </ul>
<b>Schwäche</b>	keine dynamische Speicherallokation	Einfügen oder Löschen an beliebiger Position $O(n)$	Einfügen oder Löschen an beliebiger Position $O(n)$	kein wahlfreier Zugriff	kein wahlfreier Zugriff

# std::array

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

```
std::array<int, 10> myArr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- `std::array`
  - benötigt die Headerdatei `<array>`
  - ist ein homogener Container fester Länge
  - verbindet die Speicher- und Laufzeitcharakteristik des C-Arrays mit dem Interface eines C++-Vektors

# std::array

## Initialisierung der Elemente

`std::array<int, 10> arr` ➡ Elemente werden nicht initialisiert

`std::array<int, 10> arr{}` ➡ Elemente werden Default-initialisiert

`std::array<int, 10> arr{1, 2, 3, 4, 5}` ➡ Restlichen Elemente werden Default-initialisiert

## ▪ Indexzugriff

`arr[n]` ➡ Array-Grenzen werden nicht geprüft

`arr.at(n)` ➡ Array-Grenzen werden geprüft (`std::range_error` Ausnahme)

## ▪ Verwandtschaft mit `std::tuple`

`arr[4]`      `std::get<4>(arr)`



# `std::array`

- Beispiele:
  - `array.cpp`
- Aufgaben:
  - Greifen Sie über die Indexgrenzen eines `std::array` hinaus.
    - `std::array` unterstützt den Zugriff auf seine Elemente mit dem Indexoperator `[]` und der `at`-Funktion. Die `at`-Funktion überprüft dabei ihre Grenzen. Schreiben Sie ein Programm, dass mit dem Indexoperator `[]` und der `at`-Funktion über die Grenzen des `std::array` hinausgreift und führen Sie dies aus.
    - Lösung: `array.cpp`
- Weitere Informationen:
  - [`std::array`](#)

# Smart Pointer: Übersicht

Smart Pointer sind intelligente Zeiger in C++, die den Lebenszyklus ihrer anvertrauten Ressource automatisch verwalten.

- Smart Pointer
  - allokieren und deallokieren ihre Ressource automatisch im Konstruktor und Destruktor entsprechend dem RAI-Idiom (**R**esource **A**cquisition **I**s **I**nitialization).
  - bieten explizites Speichermanagement mit Reference Counting an.
  - sind C++ Antwort auf Garbage Collection.
  - geben ihre Ressource genau dann frei, wenn der Smart Pointer seine Gültigkeit verliert.
  - gibt es in vier verschiedenen Ausprägungen.
  - benötigen die Headerdatei `<memory>`.

# Smart Pointer: Vergleich

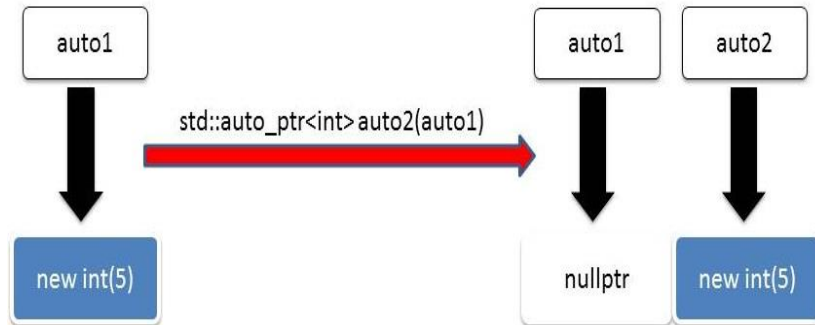
Name	Im C++ Standard	Beschreibung
<code>std::auto_ptr</code>	C++98	<ul style="list-style-type: none"><li>▪ Besitzt die Ressource exklusiv.</li><li>▪ Verschiebt beim Kopieren heimlich die Ressource.</li></ul>
<code>std::unique_ptr</code>	C++11	<ul style="list-style-type: none"><li>▪ Besitzt die Ressource exklusiv.</li><li>▪ Kann nicht kopiert werden.</li><li>▪ Verwaltet nicht kopierbare Objekte.</li></ul>
<code>std::shared_ptr</code>	C++11	<ul style="list-style-type: none"><li>▪ Teilt sich die Ressource.</li><li>▪ Bieten einen Referenzzähler auf die gemeinsame Ressource an und verwaltet diese automatisch</li><li>▪ Löscht die Ressource, sobald der Referenzzähler 0 ist.</li></ul>
<code>std::weak_ptr</code>	C++11	<ul style="list-style-type: none"><li>▪ Leiht sich die Ressource aus.</li><li>▪ Hilft zyklische Referenzen aufzubrechen.</li><li>▪ Verändert nicht den Referenzzähler.</li></ul>



# std::auto\_ptr versus std::unique\_ptr

## std::auto\_ptr

```
std::auto_ptr<int> auto1(new int(0));  
std::auto_ptr<int> auto2(auto1);
```



## std::unique\_ptr

```
std::unique_ptr<int> u1(new int(0));  
std::unique_ptr<int> u2(std::move(u1));
```



# `std::unique_ptr`

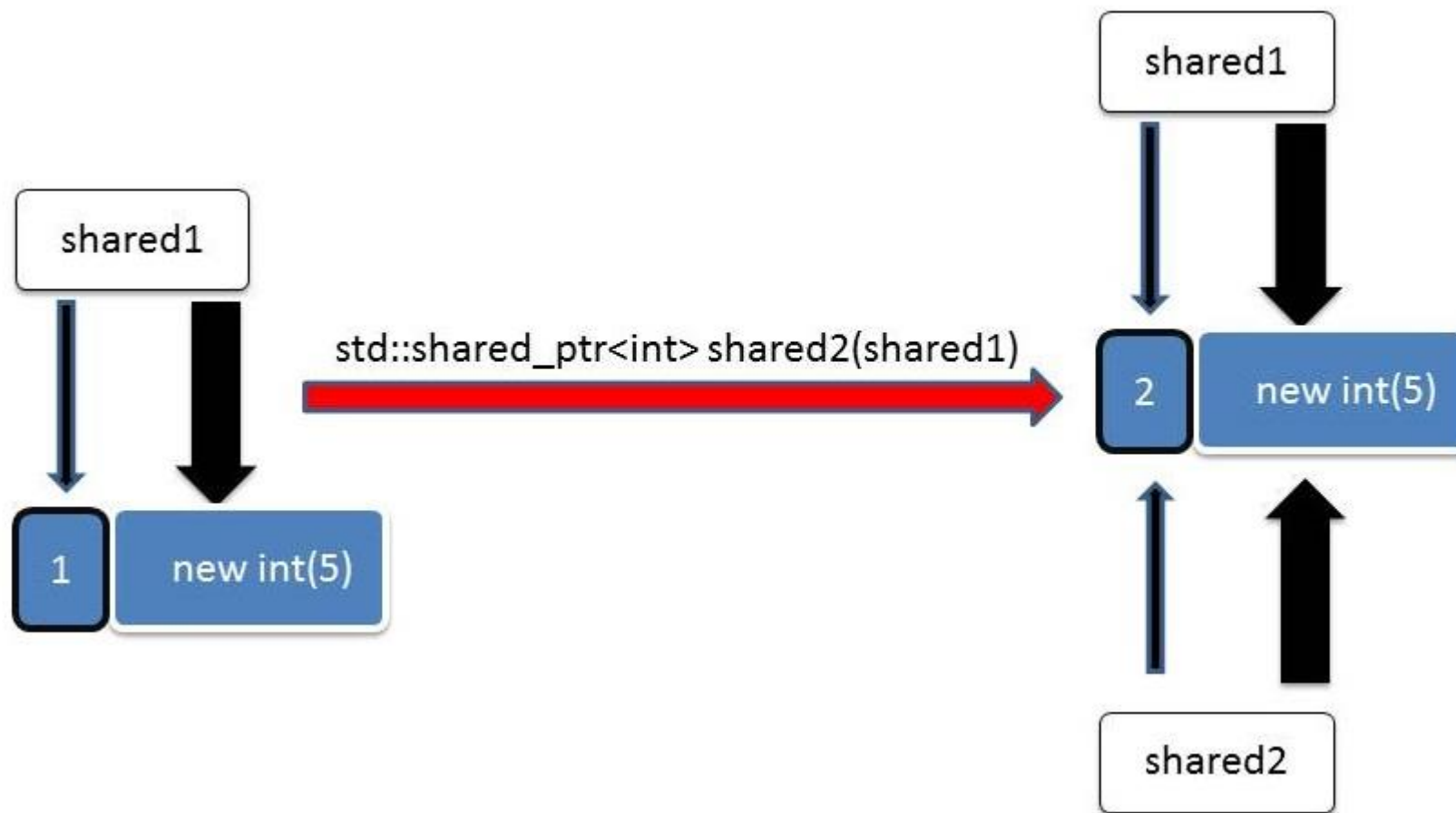
Der `std::unique_ptr` verwaltet exklusiv den Lebenszyklus seiner Ressource nach dem RAI-Idiom.

- `std::unique_ptr`
  - ist der Ersatz für den *deprecated* Smart Pointer `std::auto_ptr`
    - ➡ `std::unique_ptr` kann nicht kopiert werden.
  - lässt sich in den Container und Algorithmen der STL verwenden.
    - ➡ Container und Algorithmen dürfen keine Copy-Semantik verwenden.
  - besitzt *minimalen* Verwaltungsaufwand.
  - lässt sich über eine Löschfunktion parametrisieren:  
`std::unique_ptr<T, Deleter>`.
  - besitzt eine Spezialisierung für Arrays: `std::unique_ptr<T[]>`.

# `std::unique_ptr`

Memberfunction	Beschreibung
<code>uniq.release()</code>	Gibt einen Zeiger auf die Ressource zurück und gibt diesen frei.
<code>uniq.get()</code>	Gibt einen Zeiger auf die Ressource zurück.
<code>uniq.reset(ptr)</code>	<ul style="list-style-type: none"><li>▪ Ersetzt die Ressource.</li><li>▪ Destruiert die alte Ressource.</li></ul>
<code>uniq.get_deleter()</code>	Gibt die Löschfunktion zurück.
<code>std::make_unique(...)</code>	<ul style="list-style-type: none"><li>▪ Erzeugt die Ressource und gibt sie in einem <code>std::unique_ptr</code> zurück.</li><li>▪ Steht mit C++14 zur Verfügung.</li></ul>

# `std::shared_ptr`



# `std::shared_ptr`

`std::shared_ptr` teilen sich gemeinsam eine Ressource und geben diese gegebenenfalls wieder frei.

- `std::shared_ptr`
  - besitzt einen Verweis auf die Ressource und den Referenzzähler.
  - stellt C++'s Antwort auf Garbage Collection dar.
  - besitzt *mehr/weniger* Verwaltungsaufwand in Zeit und Speicher als ein `std::unique_ptr`.
  - löscht die Ressource deterministisch.
  - kann eine eigene Löschfunktion verwenden:  

```
shared_ptr<int> shPtr(new int, Del());
```
  - Die Verwendung des Kontrollblocks ist thread-sicher.

# `std::shared_ptr`

Memberfunktion	Beschreibung
<code>sha.unique()</code>	Prüft, ob der <code>std::shared_ptr</code> der alleinige Besitzer der Ressource ist.
<code>sha.use_count()</code>	Gibt den Wert des Referenzzählers zurück.
<code>sha.get()</code>	Gibt einen Zeiger auf die Ressource zurück.
<code>sha.reset(ptr)</code>	<ul style="list-style-type: none"><li>▪ Ersetzt die Ressource.</li><li>▪ Destruiert gegebenenfalls die Ressource.</li></ul>
<code>sha.get_deleter()</code>	Gibt die Löschfunktion zurück.
<code>std::make_shared(...)</code>	Erzeugt eine Ressource und verwaltet diese.

# `std::shared_ptr` von `this`

`std::shared_ptr` von `this` erlaubt das einfache Erzeugen von geteilten Objekten.

- `std::enable_shared_from_this`: Basisklasse der geteilten Objekte
- `shared_from_this`: gibt das geteilte Objekt zurück

```
class ShareMe: public
std::enable_shared_from_this<ShareMe>{
    std::shared_ptr<ShareMe> getShared(){
        return shared_from_this();
    }
};
```

# `std::weak_ptr`

`std::weak_ptr` ist kein klassischer Smart Pointer.

- `std::weak_ptr`
  - besitzt keine Ressource.
  - teilt sich die Ressource mit einem `std::shared_ptr`.
  - erlaubt keinen transparenten Zugriff auf die Ressource.

Der `std::weak_ptr` verändert nicht den Referenzzähler.

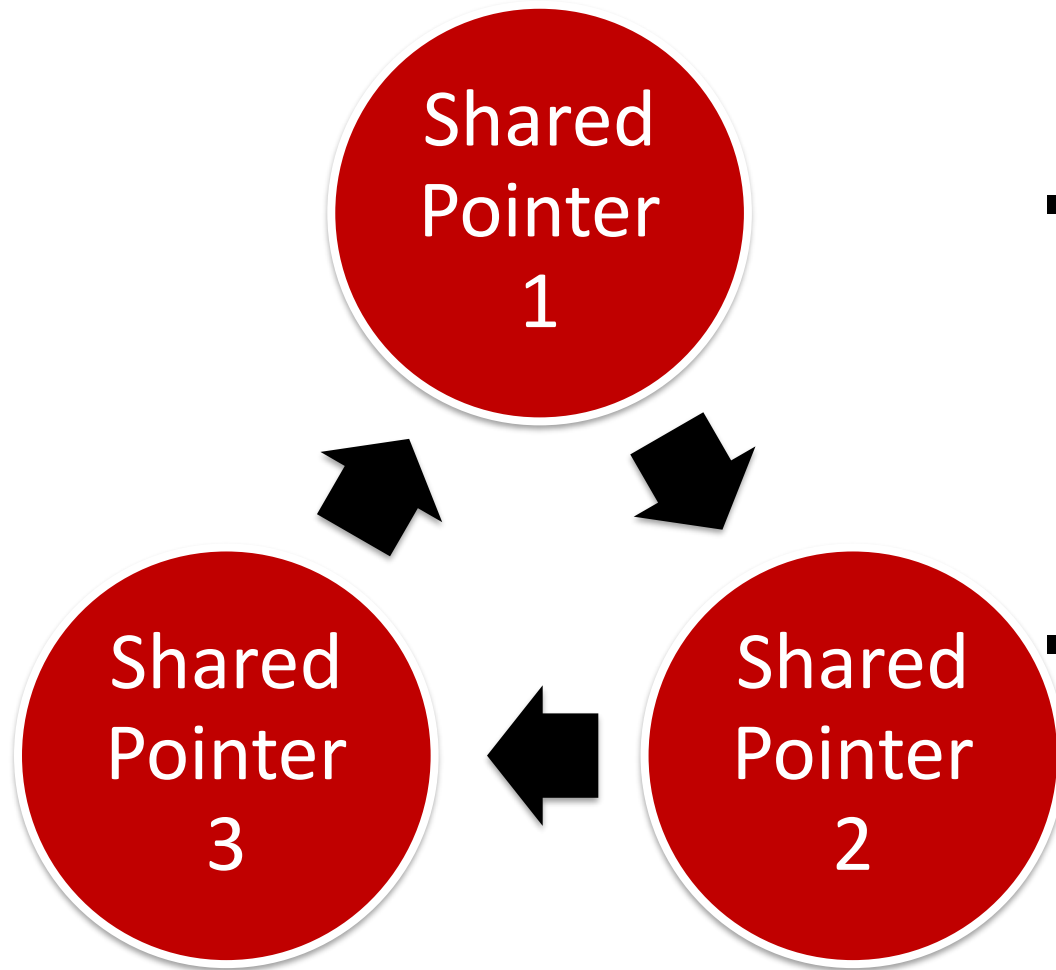
➡ Er hilft zyklische Referenzen von `std::shared_ptr` zu brechen.



# `std::weak_ptr`

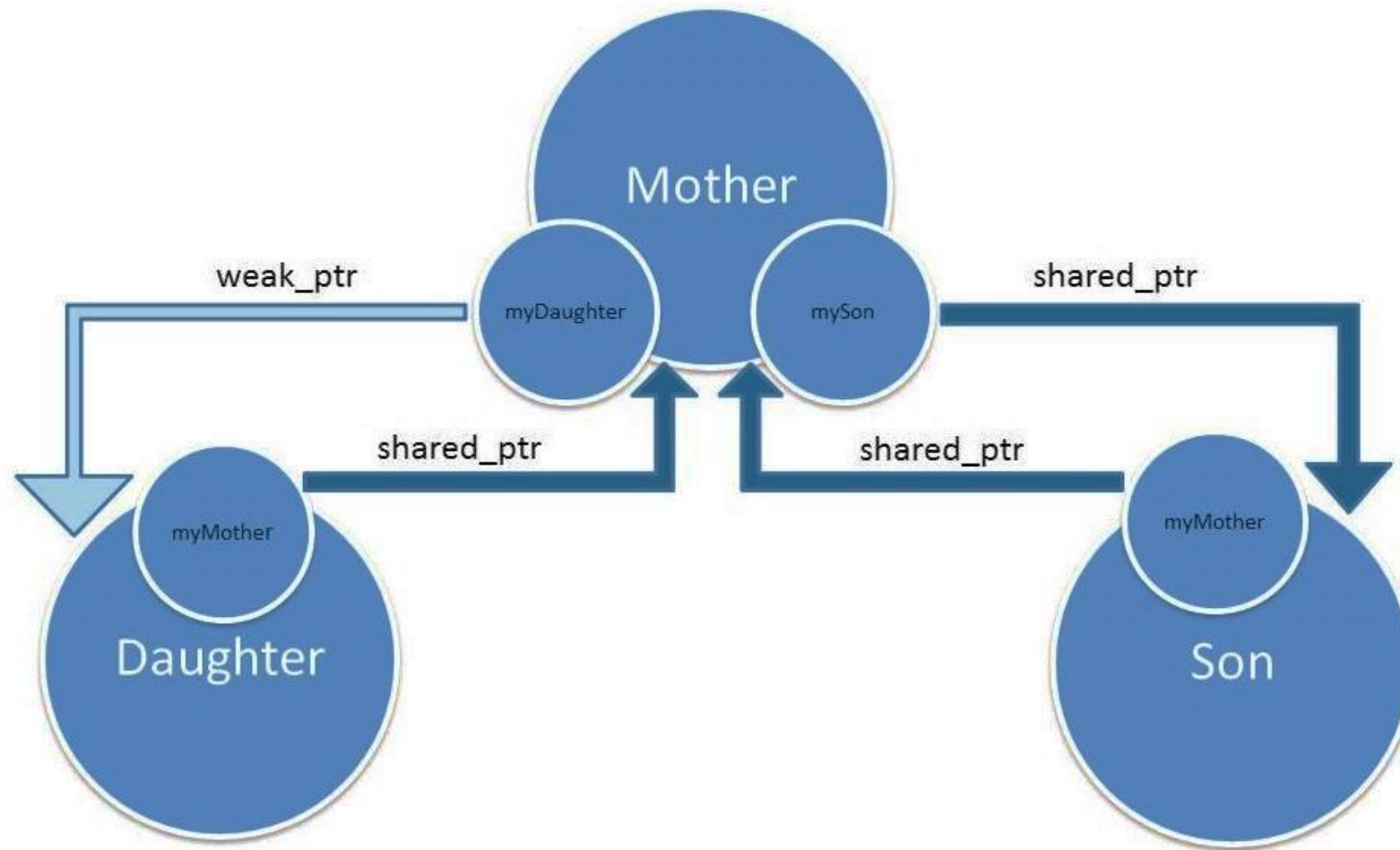
Memberfunktion	Beschreibung
<code>wea.expired()</code>	Prüft, ob die Ressource bereits gelöscht wurde.
<code>wea.use_count()</code>	Gibt den Wert des Referenzzählers zurück.
<code>wea.lock()</code>	Erzeugt einen <code>std::shared_ptr</code> auf die Ressource.
<code>wea.reset()</code>	Gibt die Ressource frei.

# Zyklische Referenzen



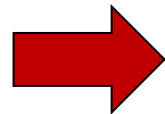
- Klassisches Problem:  
`std::shared_ptr` bilden einen Zyklus, so dass keine Ressource automatisch freigegeben werden kann.
- Rettung:  
`std::weak_ptr` bricht den Zyklus.

# Zyklische Referenzen



# Performanzvergleich

```
std::chrono::duration<double> st = std::chrono::system_clock::now();  
for (long long i = 0 ; i < 100000000; ++i){  
    int* tmp(new int(i));  
    delete tmp;  
    // std::unique_ptr<int> tmp(new int(i));  
    // std::unique_ptr<int> tmp = std::make_unique<int>(i);  
    // std::shared_ptr<int> tmp(new int(i));  
    // std::shared_ptr<int> tmp = std::make_shared<int>(i);  
}  
auto dur=std::chrono::system_clock::now() - st();  
std::cout << dur.count();
```



Zeigertyp	Zeit	Verfügbarkeit
new	2.93 s	C++98
std::unique_ptr	2.96 s	C++11
std::make_unique	2.84 s	C++14
std::shared_ptr	6.00 s	C++11
std::make_shared	3.40 s	C++11

# Smart Pointer



- Beispiele:
  - `uniquePtr.cpp`
  - `uniquePtrArray.cpp`
  - `sharedPtr.cpp`
  - `sharedPtrDeleter.cpp`
  - `weakPtr.cpp`
  - `cyclicReference.cpp`
  - `smartPointerPerformanceNative.cpp`
  - `smartPointerPerformanceUnique.cpp`
  - `smartPointerPerformanceShared.cpp`

# Smart Pointer



- Aufgaben:
  - Implementieren Sie die Klasse `ShareMe` und wenden sie diese an.
    - Objekte der Klasse `ShareMe` sollen auf Anfrage einen `std::shared_ptr` auf sich selbst zurückgeben.
    - Lösung: `shareMe.cpp`
  - Vergleichen Sie die Performanz von einfachen Zeigern mit der von Smart Pointern.
    - Nehmen Sie dazu die drei Programme als Ausgangsbasis:  
`smartPointerPerformanceNative.cpp`  
`smartPointerPerformanceShared.cpp`  
`smartPointerPerformanceUnique.cpp`
- Weitere Informationen:
  - [`std::unique\_ptr`](#)
  - [`std::shared\_ptr`](#)
  - [`std::weak\_ptr`](#)

# Besondere Anforderungen

Sicherheitskritische Systeme

Hohe Performanz

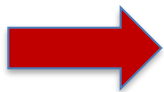
Eingeschränkte Ressourcen

Mehrere Aufgaben gleichzeitig

# Der Vertrag



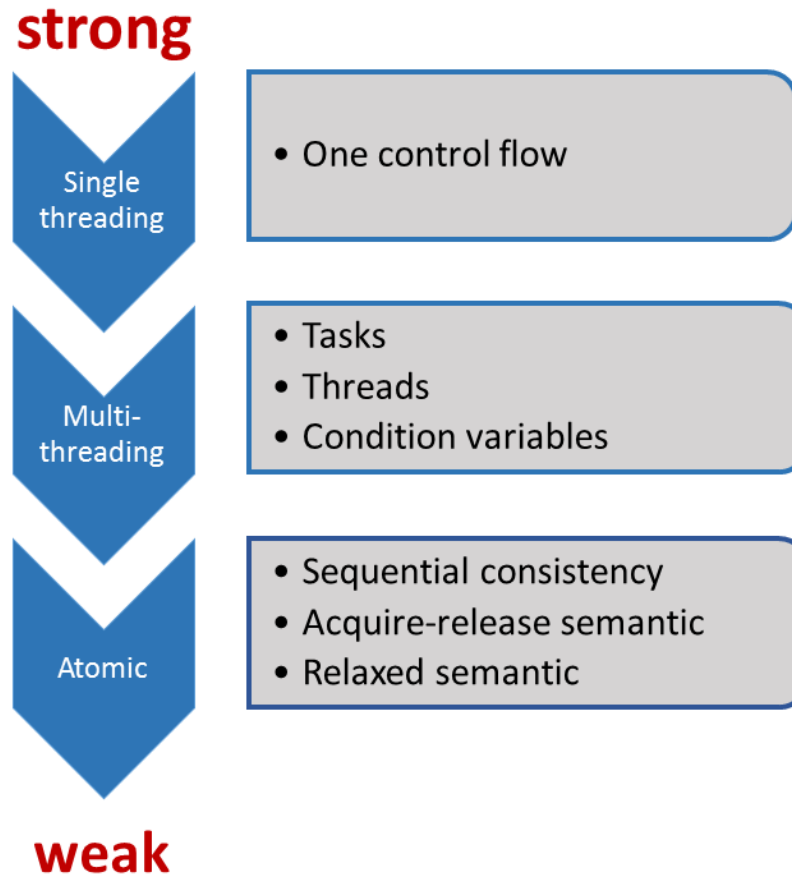
- Entwickler respektiert die Regeln
  - atomare Operationen
  - Partielle Ordnung von Operationen
  - Speichersichtbarkeit
- System besitzt die Freiheit zu optimieren
  - Compiler
  - Prozessor
  - Speicherebenen



Hoch optimiertes Programm, das auf die Plattform zugeschnitten ist.



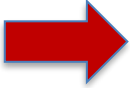
# Der Vertrag



- Mehr Optimierungspotential für das System
- Anzahl der möglichen Kontrollflüsse steigt exponentiell
- Zunehmend ein ausschließliches Gebiet für Domänenexperten
- Bruch der Intuition
- Feld für Mikrooptimierung

# Atomare Datentypen

Atomare Variablen sind die Grundlage für das C++-Speichermodell.

 Atomare Operationen auf atomare Variablen definieren die Synchronisations- und Ordnungsbedingungen.

- Synchronisations- und Ordnungsbedingungen gelten für atomare Variablen und nicht atomare Variablen.
- Synchronisations- und Ordnungsbedingungen werden von höheren Abstraktionen verwendet.
  - Threads und Tasks
  - Mutexe und Locks
  - Bedingungsvariablen
  - ...

# Atomare Datentypen

Operation	read	write	read-modify-write
test_and set			ja
clear		ja	
is_lock_free	ja		
load	ja		
store		ja	
exchange			ja
compare_exchange_weak compare_exchange_strong			ja
fetch_add, += fetch_sub, -=			ja
++, --			ja

Es gibt keine Multiplikation oder Division.

# Atomare Datentypen



- Beispiele:
  - `atomic.cpp`
- Aufgaben:
  - Verschaffen Sie sich einen Überblick:
    - Welche atomaren Datentypen gibt es?  
➔ [std::atomic](#)
    - Welche atomare Operationen gibt es?  
➔ [std::atomic](#)
    - Wie lässt sich die sequentielle Konsistenz aufbrechen?  
➔ [std::memory\\_order](#)

# Synchronisation und Ordnung

C++ kennt sechs verschiedene Speichermodelle.

```
enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst  
};
```

- Per Default gilt die Sequenzielle Konsistenz.
  - Das Speichermodell für C# und Java.
  - `memory_order_seq_cst`
  - Implizites Argument bei atomaren Operationen

```
std::atomic<int> shared;  
shared.load()  $\approx$  shared.load(std::memory_order_seq_cst);
```

# Synchronisation und Ordnung

Ordnung in das Speichermodell bringt die Beantwortung zweier Fragen.

1. Für welche atomaren Operationen sind die Speichermodelle konzipiert?
2. Welche Synchronisations- und Ordnungsbedingungen definieren die Speichermodelle?

# Synchronisation und Ordnung

## 1. Für welche atomaren Operationen sind die Speichermodelle konzipiert?

- **read-Operationen:**

`memory_order_acquire` und `memory_order_consume`

- **write-Operationen:**

`memory_order_release`

- **read-modify-write-Operationen:**

`memory_order_acq_rel` und `memory_order_seq_cst`

`memory_order_relaxed` definiert keine Synchronisations- und Ordnungsbedingungen.

# Synchronisation und Ordnung

Funktion	read	write	read-modify-write
test_and set			ja
clear		ja	
is_lock_free	ja		
load	ja		
store		ja	
exchange			ja
compare_exchange_weak compare_exchange_strong			ja
fetch_add, += fetch_sub, -=			ja
++, --			ja

```
std::atomic<int> atom;
```

```
atom.load(std::memory_order_acq_rel)
```

```
atom.load(std::memory_order_release)
```



```
atom.load(std::memory_order_acquire)
```

```
atom.load(std::memory_order_relaxed)
```



# Synchronisation und Ordnung

## 2. Welche Synchronisations- und Ordnungsbedingungen definieren die Speichermodelle?

- **Sequenzielle Konsistenz**

- Globale Ordnung auf allen Threads

`memory_order_seq_cst`

- **Acquire-Release-Semantik**

- Ordnung zwischen Lese- und Schreibeoperationen der gleichen atomaren Variablen auf verschiedenen Threads

`memory_order_consume, memory_order_acquire,`  
`memory_order_release` und `memory_order_acq_rel`

- **Relaxed-Semantik**

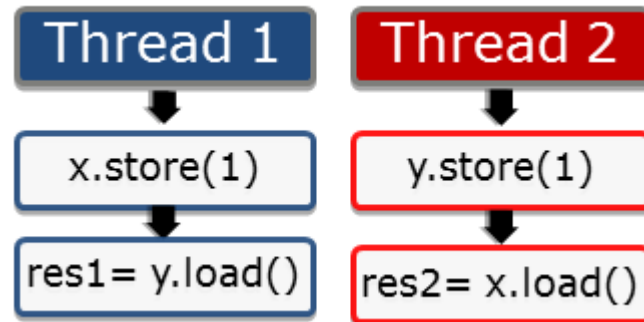
- Keine Synchronisations- oder Ordnungsbedingungen

`memory_order_relaxed`

# Synchronisations und Ordnung

## Sequenzielle Konsistenz (Leslie Lamport 1979)

1. Die Anweisungen eines Programms werden in der Sourcecodereihenfolge ausgeführt.
2. Es gibt eine globale Reihenfolge aller Operationen auf allen Threads.

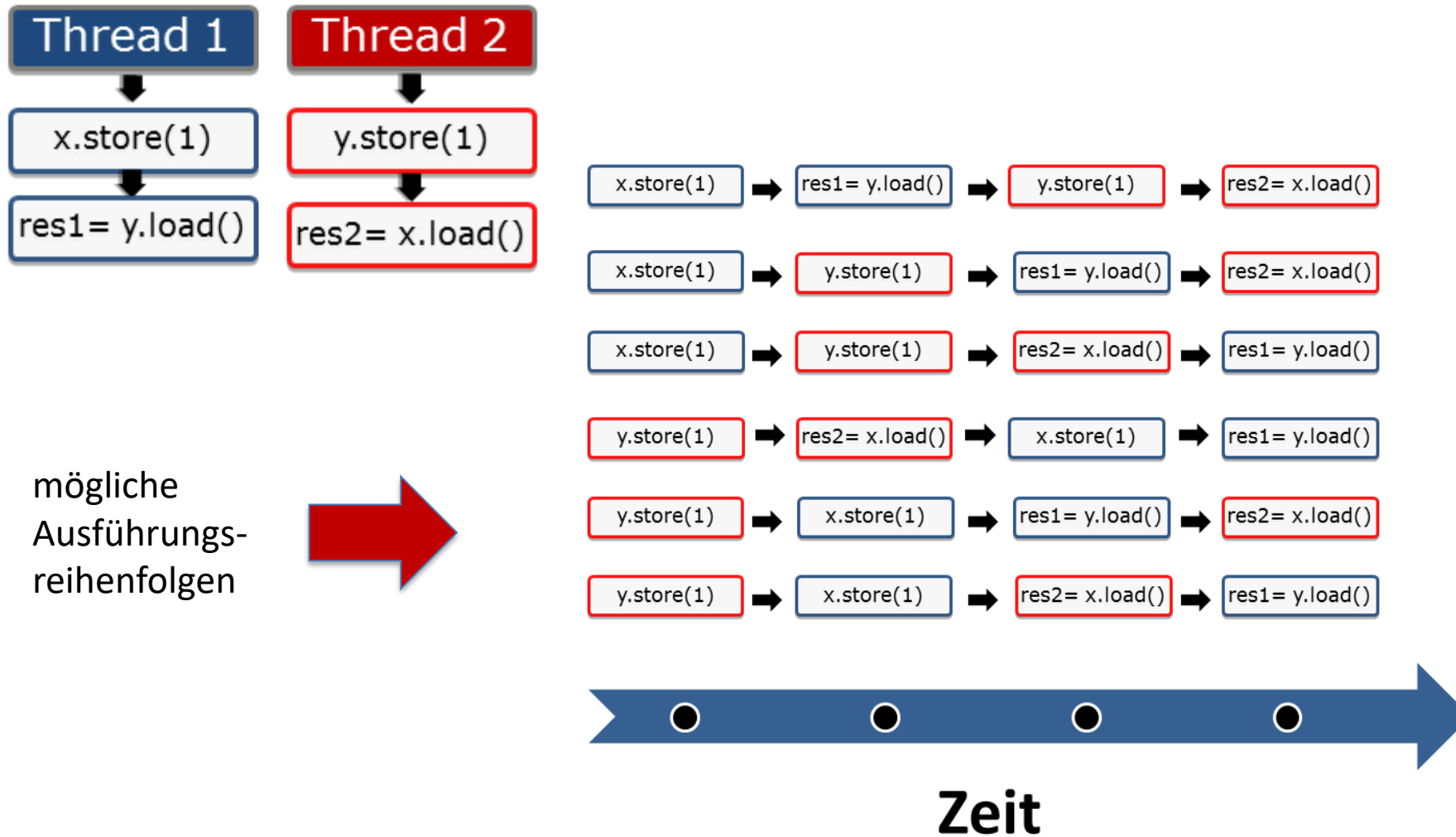


Sequenzielle Konsistenz ergibt



1. Die Befehle werden in der Reihenfolge ausgeführt, in der sie im Sourcecode stehen.
2. Jeder Thread sieht die Operationen jedes anderen Threads in der gleichen Reihenfolge (globaler Zeittakt).

# Synchronisation und Ordnung



# Threads erzeugen

Ein Thread als ausführbare Einheit (*thread of execution*) erhält seine aufrufbare Einheit und startet diese sofort. Er benötigt die Headerdatei `<thread>`.

- Eine aufrufbare Einheit kann

- eine Funktion sein.

```
std::thread t(function);
```

- ein Funktionsobjekt sein.

```
std::thread t(FunctionObject());
```

- eine Lambda-Funktion sein.

```
std::thread t([]{std::cout << "running" << std::endl;});
```

# Lebenszeit eines Threads

Der Vater (Erzeuger) muss sich explizit um die Lebenszeit seines Kindes kümmern. Die Lebenszeit des Threads endet mit dem Ende der aufrufbaren Einheit.

- Der Vater
  - wartet auf sein Kind `t`: `t.join()` ;
  - trennt sich von seinem Kind `t`: `t.detach()` ;
    - Daemon-Thread oder Service

# Datenübergabe an Threads

## Datenübergabe

```
std::string s{"C++11"};
```

- **per Kopie**

```
std::thread t([s]{std::cout << s << std::endl;});  
t.join();
```

- **per Referenz**

```
std::thread t([&s]{std::cout << s << std::endl;});  
t.detach();
```



In den Beispielen nimmt die Lambda-Funktion die Daten an.

# Operationen auf Threads

Memberfunktion	Beschreibung
<code>t.joinable()</code>	Prüft, ob der Thread <code>t</code> noch <code>join</code> oder <code>detach</code> unterstützt.
<code>t.get_id()</code> , <code>std::this_thread::get_id()</code>	Gibt die ID des Threads zurück.
<code>std::thread::hardware_concurrency()</code>	Hinweis auf die Anzahl der Threads, die gleichzeitig ausgeführt werden können.
<code>std::this_thread::sleep_until(abs_time)</code>	Legt den Thread bis zu einem Zeitpunkt schlafen.
<code>std::this_thread::sleep_for(rel_time)</code>	Legt den Thread für eine Zeitspanne schlafen.
<code>std::this_thread::yield()</code>	Bietet dem System an, einen anderen Thread auszuführen.



Die Argumente der `sleep`-Memberfunktionen sind Zeitobjekte.

# Threads



- Beispiele:
  - `threadCreate.cpp`
  - `threadArguments.cpp`
  - `threadLifetime.cpp`
  - `threadMethods.cpp`
- Aufgaben:
  - Variieren Sie die Schlafdauer von `Sleeper` in `threadArguments.cpp`.
    - ➔ Variation des Laufzeitverhaltens ersetzt keine explizite Synchronisation der Threads. (*undefined behaviour*)
    - Lösung: `threadArgumentsVariation.cpp`
  - Bestimmen Sie die `std::thread::hardware_concurrency()` auf Ihrer Plattform.
    - Lösung: `threadHardwareConcurrency.cpp`
- Weitere Informationen:
  - [thread](#)



# Gemeinsame Daten

## Gefahren

- **Data Race:** Mindestens zwei Threads verwenden ein Datum gleichzeitig, wobei mindestens ein Thread dieses modifiziert.
- **Kritischer Bereich** (*Critical Region*): Zusammenhängender Datenbereich, in dem nur ein Thread Zugriff auf die Daten haben darf.

➡ Das Programmverhalten ist undefiniert.

➡ Die Daten müssen vor gleichzeitigem Zugriff geschützt werden.



Ein Mutex (**mutal exclusion**) stellt den exklusiven Zugriff eines Threads auf die gemeinsamen Daten sicher.

# Mutexe

Ein Mutex stellt sicher, dass nur ein Thread exklusiv einen kritischen Bereich betreten kann.

- Die Mutex-Variationen benötigen die Headerdatei `<mutex>`.

Memberfunktion	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
<code>m.lock()</code>	ja	ja	ja	ja
<code>m.unlock()</code>	ja	ja	ja	ja
<code>m.try_lock()</code>	ja	ja	ja	ja
<code>m.try_lock_for(rel_t)</code>			ja	ja
<code>m.try_lock_until(abs_t)</code>			ja	ja

**C++14** enthält zusätzlich `std::shared_timed_mutex`.

# Deadlocks

## Unbekannter Code

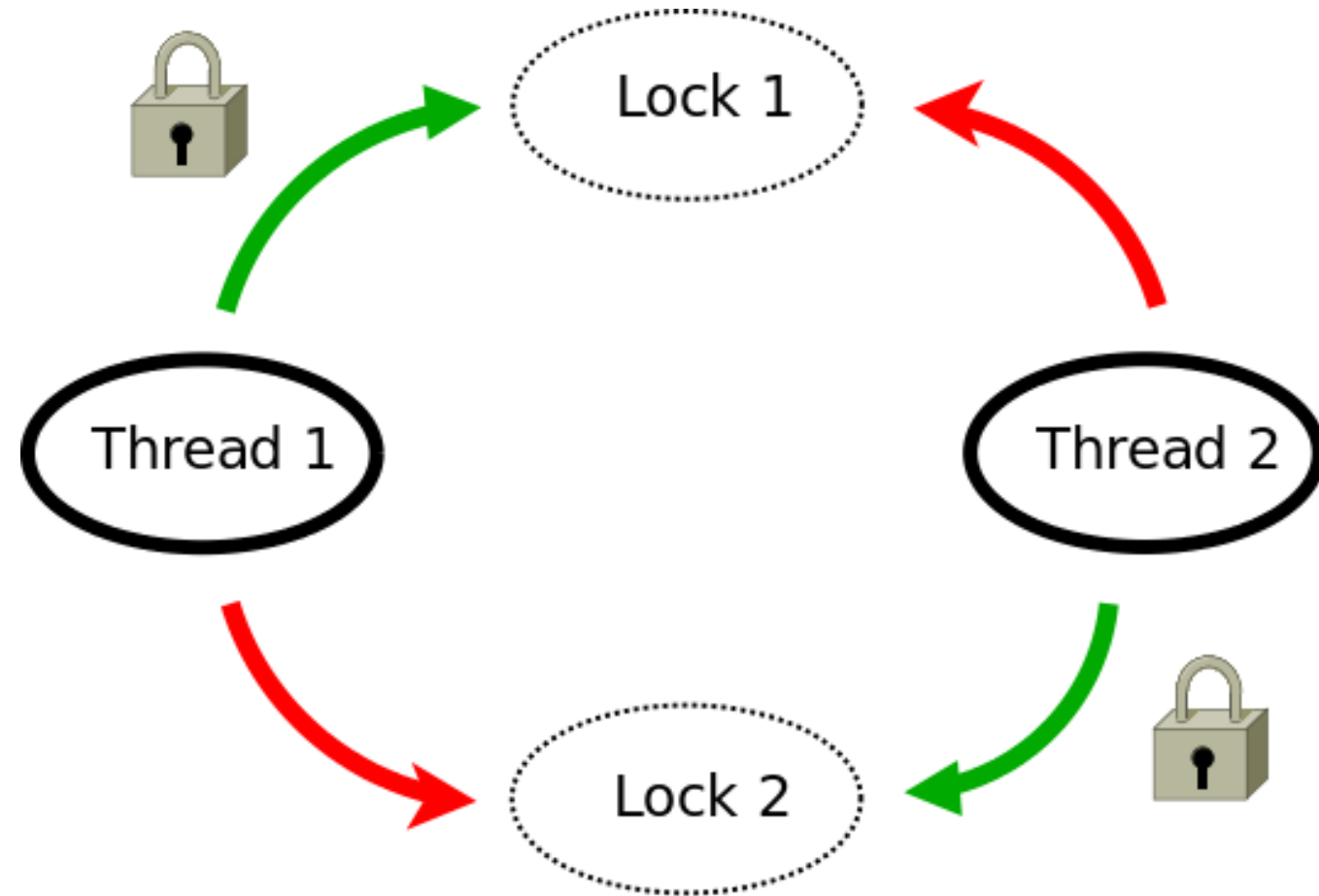
- Mutex im Einsatz:

```
mutex m;  
m.lock();  
sharedVariable = getVar();  
m.unlock();
```

- Probleme:
  - `getVar()` erzeugt eine Ausnahme.
  - verwendet einen Mutex.

# Deadlocks

Locken in verschiedener Reihenfolge



➡ Verwenden Sie `lock_guard` und `unique_lock`.

# Mutexe



- Beispiele:
  - `mutex.cpp`
- Aufgaben:
  - Führen Sie das Programm in `mutex.cpp` ohne Synchronisation des Ausgabekanals `std::cout` aus. Welche Ausgabe erhalten Sie?
  - Wie intelligent ist Ihre C++-Laufzeit?
    - Einen Mutex mehrmals zu locken stellt undefiniertes Verhalten dar. Dies kann zu einem Deadlock führen.  
Locken Sie rekursiv einen Mutex,
    - Lösung: `recursiveMutex.cpp`
- Weitere Informationen:
  - [`std::mutex`, `std::timed\_mutex`](#)
  - [`std::recursive\_mutex`, `std::recursive\_timed\_mutex`](#)
  - [`std::shared\_timed\_mutex`](#)

# Locks

`std::lock_guard` und `std::unique_lock`

- Verwalten die Lebenszeit ihres Mutex nach dem RAI-Idiom
  - Benötigen die Headerdatei `<mutex>`
- 
- RAI-Idiom (Resource Acquisition Is Initialization)
    - Die Lebenszeit der Ressource wird an die eines Objektes gebunden.
    - Im Konstruktor des Objektes wird die Ressource initialisiert, im Destruktor wieder freigegeben.
    - Das RAI-Idiom ist sehr beliebt in C++: Smart Pointer.



Sobald das Lock seine Gültigkeit verliert, wird seine Ressource Mutex vollkommen deterministisch freigegeben.

# `std::lock_guard`

`std::lock_guard` ist für den einfachen Anwendungsfall konzipiert.

- `std::lock_guard`
  - lockt automatisch den Mutex in seinem Konstruktor und gibt diesen wieder in seinem Destruktor frei.
  - ist billiger in der Anwendung als sein großer Bruder `std::unique_lock`.

```
std::mutex myMutex;  
{  
    std::lock_guard<std::mutex> myLock(myMutex);  
    sharedVariable = getVar();  
}
```

# `std::unique_lock`

`std::unique_lock` ist für den anspruchsvollen Anwendungsfall konzipiert.

- `std::unique_lock` kann
  - ohne einen Mutex oder einen Mutex, der nicht gelockt ist, erzeugt werden.
  - explizit, auch wiederholt ein Lock setzen oder freigeben.
  - einen (gelockten) Mutex in einen anderen `std::unique_lock` verschieben.
  - testen, ob ein Lock einen Mutex besitzt.
  - zeitlich verzögert locken.
  - versuchsweise mit absoluter und relativer Zeitangabe locken.
  - zwei `std::unique_lock` Instanzen tauschen.



# std::unique\_lock

Memberfunktion	Beschreibung
<code>lk.lock()</code>	Lockt den assoziierten Mutex.
<code>lk.unlock()</code>	Gibt den assoziierten Mutex frei.
<code>lk.try_lock()</code> , <code>lk.try_lock_for(rel_time)</code> , <code>lk.try_lock_until(abs_time)</code>	lk versucht den Mutex zu locken.
<code>lk.release()</code>	Gibt den Mutex frei, ohne das Lock frei zu geben.
<code>lk.mutex()</code>	Gibt einen Zeiger auf den assoziierten Mutex zurück.
<code>lk.owns_lock()</code>	Testet, ob der Lock einen Mutex besitzt.
<code>std::lock(lk1, lk2, ...)</code>	Ermöglicht, mehrere Mutexe atomar zu locken.

In **C++14** gibt es zusätzlich `std::shared_timed_mutex`.

➡ Diese unterstützen mit `std::shared_lock` ein Reader-Writer-Locks.

# Locks



- Beispiele:
  - `lockGuard.cpp`
  - `uniqueLock.cpp`
  - `readerWriterLock.cpp`
- Aufgaben:
  - Mutexe sollen nicht verwendet werden?
    - Passen Sie das Programm `mutex.cpp` so an, dass es den Zugriff auf `std::cout` mit einem Lock schützt.  
Welcher Lock bietet sich an? (`std::unique_lock` oder `std::lock_guard`)
    - Lösung: `lock.cpp`
  - Implementieren Sie einen Countdown-Zähler von 10 – 0, der seine Zahlen im Sekundentakt herunter zählt.
    - Lösung: `countDown.cpp`

# Locks

- Weitere Informationen:
  - [std::lock\\_guard](#)
  - [std::unique\\_lock](#)
  - [std::shared\\_lock](#)

# Thread-sicheres Initialisieren der Daten

Werden Daten lesend während ihrer ganzen Lebenszeit verwendet, müssen diese nur sicher initialisiert werden.

➡ Das teure Locken der Variablen bei jedem Zugriff ist nicht notwendig.

- C++ bietet drei Möglichkeiten an
  - Konstante Ausdrücke
  - Die Funktion `std::call_once` und das Flag `std::once_flag`
  - Statische Variablen mit Blockgültigkeit

# Konstante Ausdrücke: constexpr

- Konstante Ausdrücke

- werden zur Übersetzungszeit initialisiert.
- können auch hinreichend einfache eigene Datentypen sein.

```
struct MyDouble{  
    constexpr MyDouble(double v): val(v){}  
    constexpr double getValue(){return val;}  
private:  
    double val  
};
```

```
constexpr MyDouble myDouble(10.5);  
std::cout << myDouble.getValue() << std::endl;
```

# `std::call_once` und `std::once_flag`

Die Funktion `std::call_once` und das `std::once_flag` Flag.

- `std::call_once` registriert eine aufrufbare Einheit.
- Die C++ Laufzeit stellt mit Hilfe des `std::once_flag` sicher, dass die registrierte Funktion genau einmal aufgerufen wird.

```
void initSharedDataFunction(){ ... }
```

```
std::once_flag initSharedDataFlag;
```

```
std::call_once(initSharedDataFlag, initSharedDataFunction);
```

# Statische Variable

Die C++11-Laufzeit sichert zu, dass eine statische Variablen mit Blockgültigkeit threadsicher initialisiert wird.

```
void blockScope() {  
    static int mySharedDataInt= 2011;  
}
```

# Sichere Initialisierung der Daten



- Beispiele:
  - `safeInitializationCallOnce.cpp`
  - `safeInitializationStatic.cpp`
  - `singleton.cpp`
  
- Aufgaben:
  - Die klassische Implementierung des Singleton-Patterns in der Datei `singleton.cpp` ist nicht threadsicher.  
Verwenden Sie die Funktion `std::call_once` und das Flag `std::once_flag` um `MySingleton` threadsicher zu implementieren.
    - Lösung: `singletonMultithreading.cpp`
  
- Weitere Informationen:
  - [`std::call\_once`](#) und [`std::once\_flag`](#)
  - [Double-Checked Locking Pattern](#)



# Thread-lokale Daten

Thread-lokale Daten werden durch das Schlüsselwort `thread_local` definiert.

- `thread_local` Daten
  - werden für jeden Thread erzeugt.
  - gehören exklusiv einem Thread.
  - verhalten sich wie statische Variablen.
    - Sie werden bei ihrer ersten Verwendung erzeugt.
    - Sie sind an die Lebenszeit ihres Threads gebunden.



Thread-lokale Daten werden auch gerne thread-lokaler Speicher genannt.

# Thread-lokale Daten



- Beispiele:
  - `threadLocal.cpp`
- Weitere Informationen:
  - [thread\\_local](#)

# Bedingungsvariablen

Bedingungsvariablen ermöglichen es, Threads über Benachrichtigungen zu synchronisieren.

- Typische Anwendungsfälle
  - Sender - Empfänger
  - Producer – Consumer
- `std::condition_variable`
  - benötigen der Headerdatei `<condition_var>`
  - kann sowohl die Rolle eines Sender als auch eines Empfängers einnehmen



Synchronisation von Threads ist mit Tasks meist einfacher.

# Bedingungsvariablen

- Sender schickt eine Benachrichtung

Memberfunktion	Beschreibung
<code>cv.notify_one()</code>	Benachrichtigt einen wartenden Thread.
<code>cv.notify_all()</code>	Benachrichtigt alle wartenden Threads.

- Empfänger wartet mit Hilfe eines Mutex auf die Benachrichtigung

Memberfunktion	Beschreibung
<code>cv.wait(lock, .... )</code>	Wartet auf die Benachrichtigung.
<code>cv.wait_for(lock, relTime, .... )</code>	Wartet eine relative Zeit auf die Benachrichtigung.
<code>cv.wait_until(lock, absTime, .... )</code>	Wartet eine absolute Zeit auf die Benachrichtigung.



Die `wait`-Memberfunktionen erhalten noch ein zusätzliches Prädikat, um sie gegen *spurious wakeup* und *lost wakeup* zu schützen.

# Bedingungsvariablen

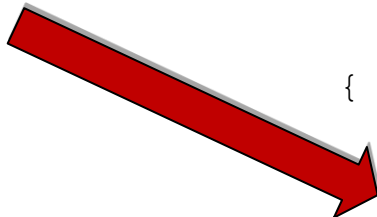
## Thread 1: Sender

- Macht seine Arbeit
- Benachrichtigt den Empfänger

```
// do the work
{
    lock_guard<mutex> lck(mut);
    ready= true;
}
condVar.notify_one();
```

## Thread 2: Empfänger

- Wartet mit dem Lock auf seine Benachrichtigung
- Erhält das Lock
  - prüft und schläft
- macht seine Arbeit
- gibt den Lock wieder frei



```
{
    unique_lock<mutex>lck(mut);
    condVar.wait(lck, []{return ready;});
    // do the work
}
```

# Bedingungsvariablen



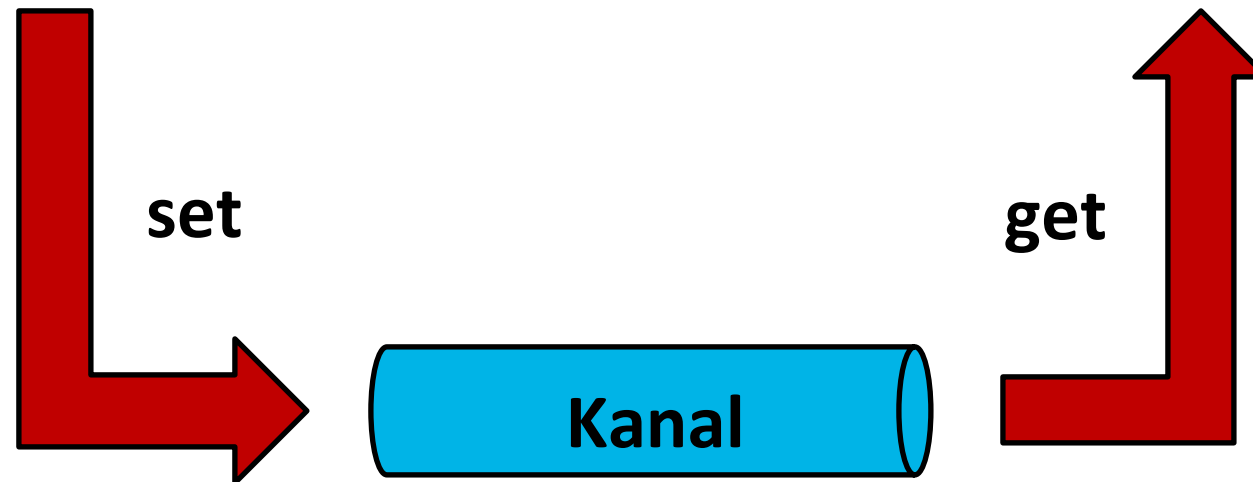
- Beispiele:
  - `conditionVariable.cpp`
- Aufgaben:
  - Schreiben Sie ein kleines Pingpongspiel
    - Zwei Threads sollen abwechselnd einen Wahrheitswert auf `true` bzw. `false` setzen. Dabei setze ein Thread der Wert auf `true` und signalisiert diese dem anderen Thread, der den Wert auf `false` setzt. Das Spiel soll nach einer endlichen Zahl von Ballwechseln beendet sein.
    - Lösung: `conditionVariablePingPong.cpp`
- Weitere Informationen:
  - [`std::condition\_variable`](#)

# Tasks als Datenkanäle

Ein Task verhält sich wie ein Datenkanal.

**Promise: Sender**

**Future: Empfänger**



- ist der Datensender.
- kann mehrere Futures bedienen.
- kann Werte, Ausnahmen und Benachrichtigungen schicken.

- ist der Datenempfänger.
- der `get`-Aufruf ist blockierend.

# Thread versus Task

## Thread

```
int res;  
thread t([&]{res = 3 + 4;});  
t.join();  
cout << res << endl;
```

## Task

```
auto fut = async([]{return 3 + 4;});  
cout << fut.get() << endl;
```

Kriterium	Thread	Task
Header-Datei	<thread>	<future>
Beteiligten	Erzeuger- und Kinderthread	Promise und Future
Kommunikation	gemeinsame Variable	Kommunikationskanal
Threaderzeugung	verbindlich	optional
Synchronisation	join-Aufruf wartet	get-Aufruf blockiert
Ausnahme im Kinderthread	Kinder- und Erzeugerthread enden	Rückgabewert des get -Aufrufs
Formen der Kommunikation	Werte	Werte, Benachrichtigungen und Ausnahmen



# `std::async`

## `std::async`

- erhält eine aufrufbare Einheit und deren Argumente.
- gibt ein `std::future` Objekt zurück um das Ergebnis der Funktion abzufragen.
- lässt sich mit `std::launch::async` explizit einem anderen Thread, oder mit `std::launch::deferred` (*lazy*) in dem gleichen Thread starten.



Die C++-Runtime entscheidet, ob `std::async` in einem neuen Thread ausgeführt wird. ➡ `std::async` ist die erste Wahl für einen Task.

# `std::async`



- Beispiele:

- `asyncLazyEager.cpp`
- `dotProduct.cpp`

- Aufgaben:

- Die Berechnung des Skalarprodukts in der Datei `dotProduct.cpp` lässt sich sehr gut parallelisieren.  
Starten sie vier asynchrone Funktionen zur Berechnung des Skalarprodukt und vergleichen Sie die Single- und Multithreaded Ausführungszeit.  
Was zeigt Ihnen die CPU-Auslastung an?
  - Lösung: `dotProductAsync.cpp`

- Weitere Informationen:

- [std::async](#)

# std::packaged\_task

`std::packaged_task` erlaubt es, einen einfachen Wrapper um eine aufrufbare Einheit zu erzeugen, so dass diese später ausgeführt werden kann.

- `std::packaged_task`

1. Verpacke die Aufgabe
2. Erzeuge den Future
3. Der Promise führt die Berechnung aus
4. Der Future holt das Ergebnis ab

```
int add(int a, int b){return a + b;}  
packaged_task<int(int, int)> sumT(add); // 1  
future<int> sumRes= sumT.get_future(); // 2  
sumT(2000, 11); // 3  
cout << sumResult.get() << endl; // 4
```

Der Future und der Promise lassen sich explizit in einen anderen Thread verschieben.

 Move-Semantik

# `std::packaged_task`



- Beispiele:
  - `packagedTask.cpp`
- Aufgaben:
  - Parametrisieren Sie die Summation von natürlichen Zahlen in dem Programm `packagedTask.cpp` so, dass die Anzahl der Threads von der Konstanten `std::thread::hardware_concurrency()` abhängt.  
Falls der Funktionsaufruf der Zahl 0 ergibt, gehen Sie von vier CPUs aus.
    - Lösung: `packagedTaskHardwareConcurrency.cpp`
- Weitere Informationen:
  - [`std::packaged\_task`](#)

# `std::promise`

`std::promise` und `std::future` erlauben die volle Kontrolle über die Task.

Memberfunktion	Beschreibung
<code>prom.get_future()</code>	Gibt den <code>std::future</code> zurück.
<code>prom.set_value(val)</code>	Setzt den Wert.
<code>prom.set_exception(ex)</code>	Setzt die Ausnahme.
<code>prom.set_value_at_thread_exit(val)</code>	Speichert den Wert und setzt ihn auf bereit, sobald der aktuelle Thread beendet wird.
<code>prom.set_exception_at_thread_exit(ex)</code>	Speichert die Ausnahme und setzt sie auf bereit, sobald der aktuelle Thread beendet wird.

# std::future

Memberfunktion	Beschreibung
<code>fut.share()</code>	Gibt einen <code>std::shared_future</code> zurück. <code>fut</code> besitzt danach nicht mehr den gemeinsamen Zustand.
<code>fut.get()</code>	Gibt den gemeinsamen Zustand zurück. Dies kann ein Wert, eine Benachrichtigung oder eine Ausnahme sein.
<code>fut.valid()</code>	Prüft, ob der gemeinsame Zustand vorliegt.
<code>fut.wait()</code>	Wartet, bis der gemeinsame Zustand vorliegt.
<code>fut.wait_for(rel_time)</code>	Wartet maximal eine relative Zeitspanne.
<code>fut.wait_until(abs_time)</code>	Wartet maximal bis zu einem Zeitpunkt.

`fut.wait()` erlaubt es, den Future mit dem Promise zu synchronisieren.

➡ Meist sind Promise und Future Bedingungsvariablen vorzuziehen.

# `std::shared_future`

Der Aufruf `fut.share()` erzeugt einen `std::shared_future`.

- `std::shared_future`
  - können unabhängig voneinander den assoziierten `std::promise` abfragen.
  - besitzen das gleiche Interface wie `std::future`.
  - kann direkt erzeugt werden.

```
std::shared_future<int> divResult= divPromise.get_future();
```

# Condition Variablen versus Tasks

Kriterium	Condition Variable	Task
Mehrfache Synchronisation möglich	Ja	Nein
Kritischer Bereich	Ja	Nein
Ausnahmebehandlung im Empfänger	Nein	Ja
Spurious wakeup	Ja	Nein
Lost wakeup	Ja	Nein



# `std::promise` und `std::future`



- Beispiele:
  - `promiseFuture.cpp`
  - `promiseFutureSynchronize.cpp`
- Aufgaben:
  - Ihr Promise soll eine Ausnahme auslösen.
    - Implementieren sie die Ausnahmebehandlung im assoziierten Future.
    - Lösung: `promiseFutureException.cpp`
- Weitere Informationen:
  - [`std::promise`](#)
  - [`std::future`](#)
  - [`std::shared\_future`](#)

# Online-Compiler

- Arne Mertz's Übersicht: Liste von Online C++ Compilern
  - [Coliru](#) (Bestandteil von cppreference.com)
  - [C++ Shell](#) (angenehm zu verwenden)
  - [Wandbox](#) (der Mächtigste)
  - [Visual C++ Online Compiler](#) (spricht Windows)
  - [Godbolt](#) (erzeugt Assembler Befehle)
  - [Cpp Insight](#) (zeigt Compiler Transformationen)
- [Interactive C/C++ memory model](#)
- [Interactive template meta shell](#)



Blog: [www.ModernesCpp.com](http://www.ModernesCpp.com)  
Mentoring: [www.ModernesCpp.org](http://www.ModernesCpp.org)

Rainer Grimm  
Training, Mentoring und  
Technologieberatung