Kurseinheit 10: Erweitertes C++

1. **Intelligente Zeiger**
2. Lambdas

Garbage Collection in C++

In einigen Programmiersprachen werden dynamisch allokierte Ressourcen vom System selbst verwaltet und freigegeben, wenn diese nicht mehr benötigt werden. In C++ gibt es diesen Mechanismus nicht, um maximale Kontrolle über das System zu ermöglichen. Gewisse Konstellationen sind daher sehr fehleranfällig und dynamische Allokation wird gerne gemieden.

```
void DynamicMemory()
{
    // allocate
    auto* pArray = new int[100];

// … // do some work

    // clean up
    delete[] pArray;
}
```

Delete kann vergessen werden.
Sehr fehleranfällig.

Lösung? →

```
void StaticMemory()
{
    int ui32Array[100];
    // do some work
    // …
}
```

Sauber, kein Memory Leak möglich.

```
void StaticDynamicMemory()
{
    int ui32Array[size];
    // do some work
    // …
}
```

Lösung?

Garbage Collection in C++

Nicht immer lässt sich eine dynamische Allokation von Speicher vermeiden. Werden Ressourcen allokiert, müssen diese auch manuell wieder freigegeben werden, sonst drohen Memory-Leaks. Idealerweise wird allokierter Speicher automatisch freigegeben sobald dieser nicht mehr benötigt wird.

Dynamische Komponente

Fallbeispiel 1

```
void DynamicMemory()
{
    int ui32Array[size];
    // do some work
    // …
}
```

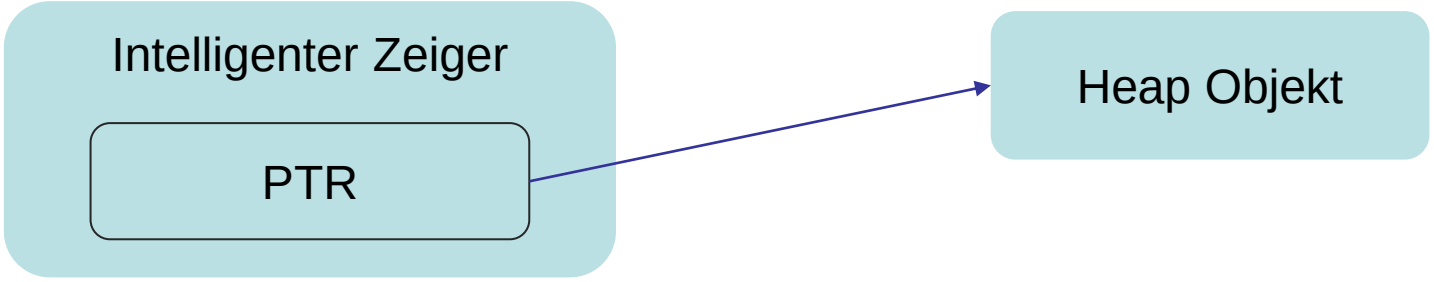
Fallbeispiel 2

```
class A { };
class B : public A { };

void DynamicMemory()
{
  A* pA = nullptr;
  switch(input) {
    case 'b':
          pA = new B();
          break;
  }
  delete pA;
}
```

Ein Intelligenter Zeiger ist ein Objekt welches einen Zeiger auf ein Heap allokiertes Objekt als Attribut besitzt

- Verhält sich wie ein regulärer C++ Zeiger durch überladen von
  - *,->,[], …
- Hilft dabei Speicher zu verwalten
  - Intelligente Zeiger löschen das Heap-Objekt zum rechten Zeitpunkt
  - Rufen den Destruktor des Heap-Objekts auf



Bei geschickter Anwendung und Auswahl der intelligenten Zeiger müssen dynamische Ressourcen **nicht manuell gelöscht** werden.

Einfache Implementierung

- SimplePtr
  - Konstruktor mit Zeiger als Übergabeparameter
  - Destruktor welcher das referenzierte Objekt löscht
  - * und -> Operatoren überladen für direkten Zugriff auf referenziertes Objekt

```
#pragma once

template<typename T>
class SimplePtr
{
public:
  SimplePtr(T* ptr) : ptr_(ptr) {}
  ~SimplePtr()
  {
    delete ptr_;
    ptr_ = nullptr;
  }

  T& operator*() { return *ptr_; }

private:
  T* ptr_; }
```

Einfache Implementierung

```
void foo()
{
    SimplePtr<A> ptr(new B());

    // no memory leak
}
```

Return der Funktion foo löst Destruktor von Objekt ptr der Klasse SimplePtr aus. Destruktor führt delete auf internen Zeiger ptr_ aus.

- SimplePtr vermeidet einfache Memory-Leaks
- Stößt schnell an Grenzen, kann z.B. nicht umgehen mit
  - Arrays
  - Objektkopien
  - Neuzuweisung des Heap-Objekts
  - Vergleichsoperatoren
  - …

Intelligente Zeiger der STL - std::unique_ptr<T>

- std::unique_ptr übernimmt den Besitz eines Zeigers
- Templateklasse für beliebige Datentypen
- Destruktor löst delete auf den beinhaltenden Zeiger aus

```
void MemoryLeak()
{
  B* pB = new B();
  // memory leak
}

void NonLeaking()
{
  std::unique_ptr<B> b(new B());
  // no memory leak
}
```

Löschen von Objekt B kann nicht vergessen werden. Objekt wird automatisch beim verlassen der Funktion gelöscht. Reduziert die Fehleranfälligkeit

Intelligente Zeiger der STL - std::unique_ptr<T>

```
void UniquePtrOperations()
{
   unique_ptr<B> b(new B());

   // calling a method of class B
   b->MethodCall();

   // deallocate current object of B and create a new one
   b.reset(new B());

   // release responsibility for deletion
   auto* pB = b.release();
   delete pB;
}
```

Die Verantwortung für das dynamisch allokierte Objekt kann durch Aufruf der Methode release() abgegeben werden.

Intelligente Zeiger der STL - std::unique_ptr<T>

Objekte bewegen

```
void foo()
{
  unique_ptr<int> x(new int(20)); // ok
  unique_ptr<int> y(x);           // fail - no copy allowed
  unique_ptr<int> z;              // ok - z is nullptr
  z = x;                          // fail - no assignment allowed
}
```

```
void foo()
{
  unique_ptr<int> x(new int(20));
  // move object ownership from x to y
  std::unique_ptr<int> y(x.release());
  // move object ownership from y to z
  unique_ptr<int> z;
  z.reset(y.release());
}
```

Direktes Bewegen des Objektbesitzes zwischen unique_ptr Objekten ist nicht möglich. Kein Kopieren, nur verschieben.

Intelligente Zeiger der STL - std::unique_ptr<T>

```
void foo()
{
  std::unique_ptr<int[]> x(new int[20]);

  x[0] = 100;
  x[1] = 200;
}
```

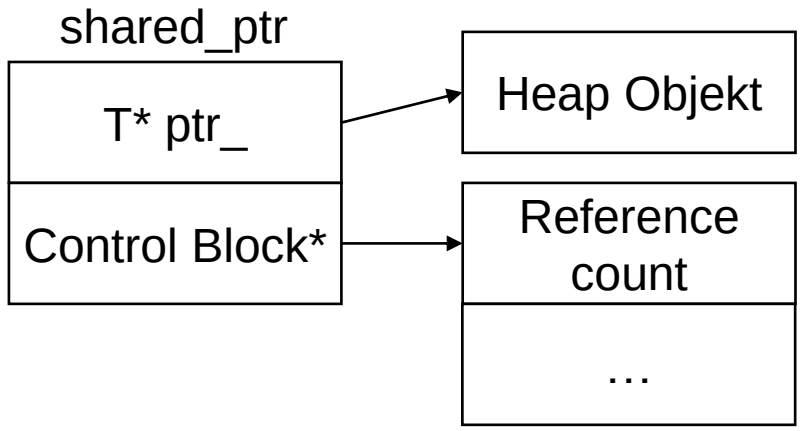
Arrays: delete[] wird durch den Destruktor von unique_ptr aufgerufen

Intelligente Zeiger der STL
Erweitertes Ressourcen-Management

- std::unique_ptr ermöglicht keine multiplen Zeiger auf ein Objekt
- Besitz des Objektes wird durch immer einen unique_ptr gesteuert.

- Lösung: Reference Counting
- Inkrementieren und Dekrementieren einer Zählvariablen
- Letztes Dekrement auf 0 löscht das Objekt
  - Mehrere Zeiger teilen sich Besitz eines Objekts
  - std::shared_ptr

- Nachteile:
  - Overhead (Performanz)
  - Zyklen können nicht aufgelöst werden



Intelligente Zeiger der STL - std::shared_ptr<T>

```
void foo()
{
  std::shared_ptr<int> x(new int(20));
  // count: 1
  {
    std::shared_ptr<int> y = x;
    // count: 2
  }
  // count: 1
}
// count: 0 -> delete
```

Mit jedem weiteren std::shared_ptr auf ein Objekt wird der Referenzzähler um 1 inkrementiert

Intelligente Zeiger der STL - std::shared_ptr<T>

```
void foo()
{
  std::shared_ptr<int> x(new int(20));
  // count: 1
  foo2(x); // count: 2 (call by value -> copy)
  // count: 1
}
// count: 0
```

```
void foo2(std::shared_ptr<int> ptr)
{
  // ptr::count 2
}
```

Funktionsaufruf Call-By-Value: Kopie von std::shared_ptr wird an foo2 übergeben. Referenzzähler wird um 1 erhöht. Das Objekt wird während der Laufzeit von foo2 nicht gelöscht.

Intelligente Zeiger der STL - std::shared_ptr<T>

```
void foo()
{
  std::shared_ptr<int> x(new int(20));
  // count: 1
  foo2(x); // count: 1 (call by reference -> no copy)
  // count: 1
}
// count: 0
```

```
void foo2(std::shared_ptr<int>& ptr)
{
  // ptr::count 1
}
```

Schneller jedoch Fehleranfälliger

Funktionsaufruf Call-By-Reference: Keine Kopie von std::shared_ptr, daher wird der Referenzzähler nicht inkrementiert. Wird x in foo gelöscht, wird auch das Objekt auf welches ptr verweist gelöscht → Laufzeitfehler

Intelligente Zeiger der STL - std::shared_ptr<T>

```
class A { };
class B : public A { };

Std::shared_ptr<B> pB(new B());
Foo2(pa);
```

Polymorphismus?

```
void foo2(std::shared_ptr<A>& ptr)
{
}
```

Compiler Error

```
void foo2(std::shared_ptr<A> ptr)
{
}
```

Implizite Typumwandlung

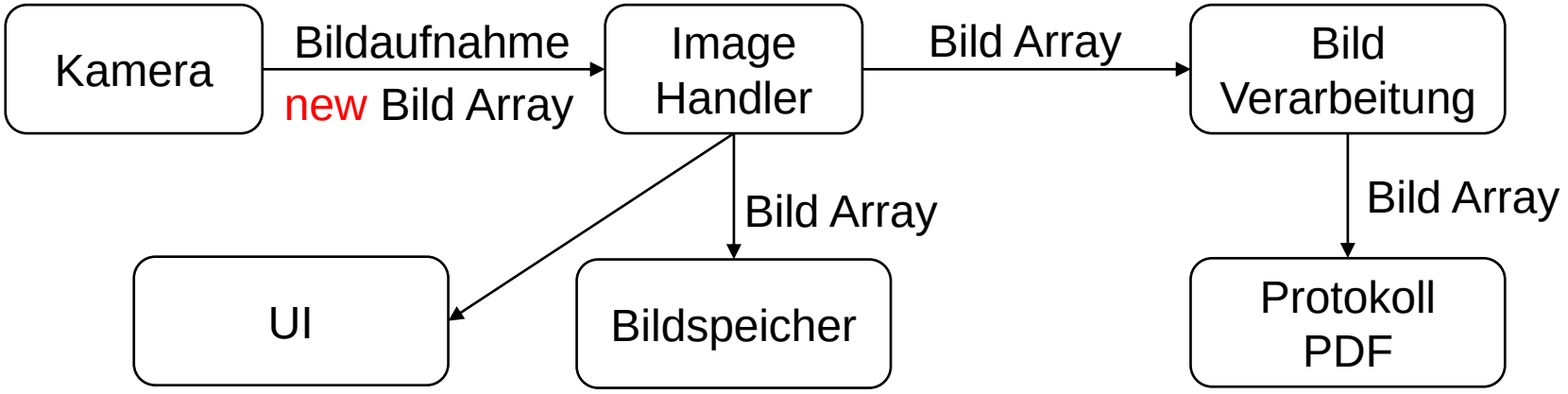
Funktionsaufruf Call-By-Reference: Keine implizite Typumwandlung möglich. Beim Einsatz von Polymorphismus einen std::shared_ptr immer per Value übergeben.

Intelligente Zeiger der STL - std::shared_ptr<T>

```
void foo()
{
  std::shared_ptr<int> x(new int(20));
  auto y = std::move(x); // efficient move
}
```

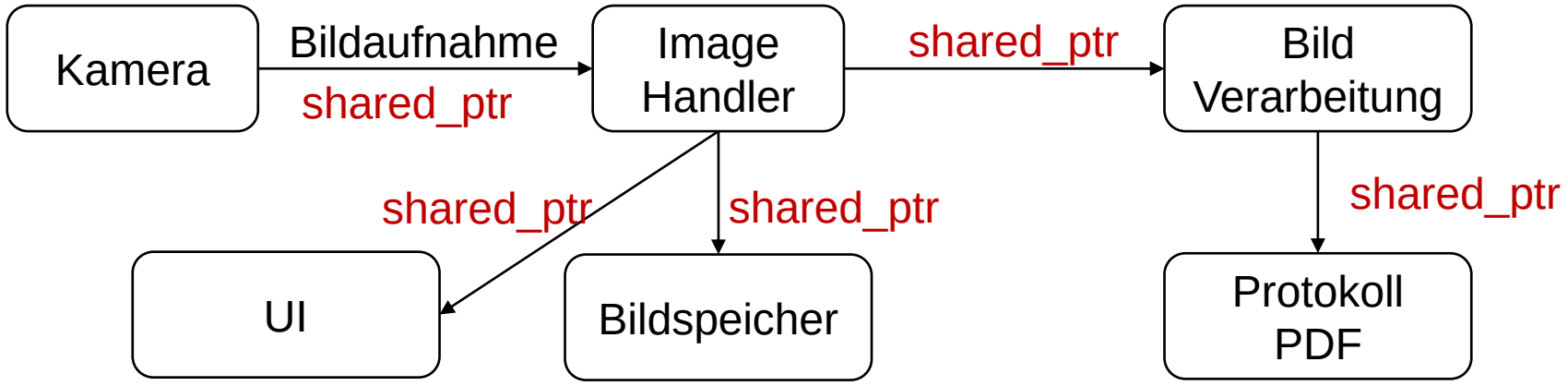
Move-Semantik für schnelle Weitergabe von Objekten

Intelligente Zeiger der STL - std::shared_ptr<T>
Systembeispiel



- Welches Modul hat letzten Zugriff auf das Bild-Objekt?
- Wo werden allokierte Daten gelöscht?
- Wann?
- Wer?

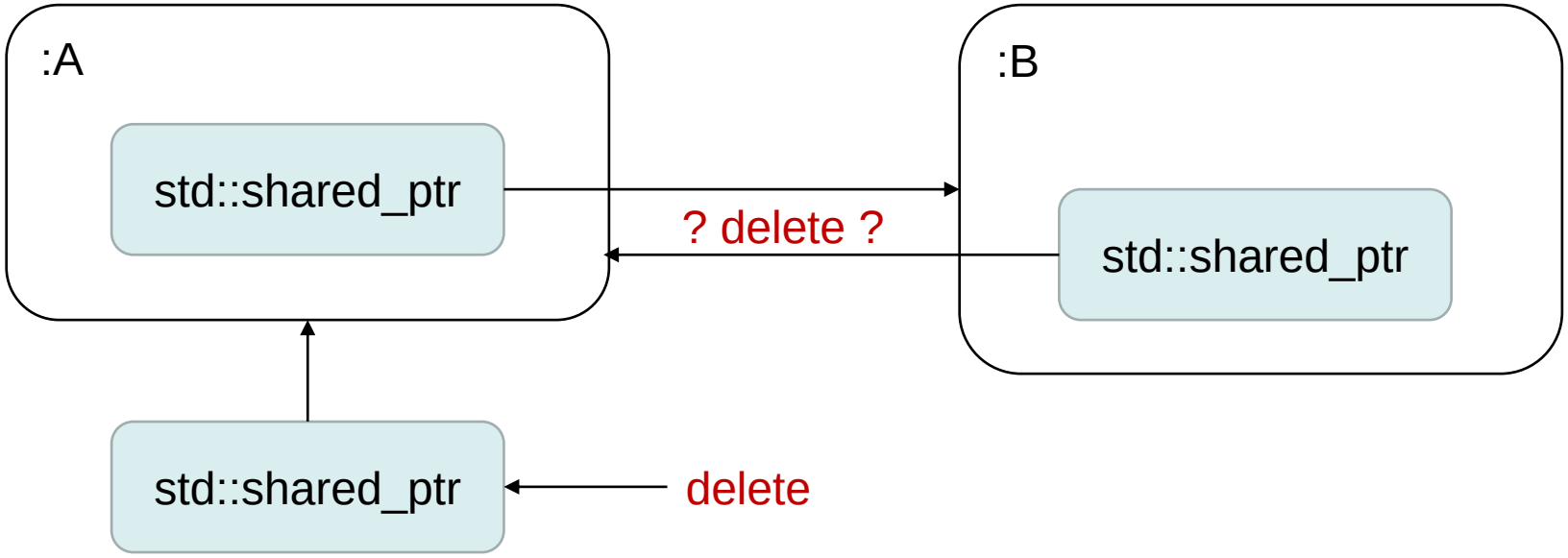
Intelligente Zeiger der STL - std::shared_ptr<T>
Systembeispiel



Freigabe durch letzten std::shared_ptr
Reihenfolge und zeitliches Timing nebensächlich.

Intelligente Zeiger der STL - std::shared_ptr<T>
Zyklen

- Ringzyklus kann nicht aufgelöst werden
- Lösung?



Intelligente Zeiger der STL
std::weak_ptr<T>

- Ähnlich std::shared_ptr
- Referenzzähler bleibt unangetastet
- Dereferenzierung nicht direkt möglich
- Zugriff über std::shared_ptr (lock)

```
void foo()
{
 std::shared_ptr<int> shared(new int(20));    // count: 1
 std::shared_ptr<int> shared2 = shared;       // count: 2
 std::weak_ptr<int> weak = shared;            // count: 2
 std::shared_ptr<int> access = weak.lock();   // count: 3
 *access = 25;
}
```

std::weak_ptr löst Probleme mit Zyklen. Referenzzähler nur bei Zugriff über lock() inkrementiert.

Intelligente Zeiger der STL
Zusammenfassung

- std::unique_ptr
  - Kann nicht kopiert, aber verschoben werden
  - Release() Löst Objekt von Zeiger-Objekt
- std::shared_ptr
  - Mehrere Besitzer des Heap-Objektes gleichzeitig
  - Kann kopiert und weitergereicht werden
  - Overhead
  - Probleme durch Zyklen
- std::weak_ptr
  - Referenzzähler nur bei Zugriff durch lock() inkrementiert
  - Keine direkte Dereferenzierung möglich
  - Löst Probleme mit Zyklen bei std::shared_ptr

Intelligente Zeiger der STL
Instanziierung

- Als Objekt mit Übergabeparameter an Konstruktor
- Durch std::make_shared bzw. std::make_unique

```
void CreatePointers()
{
  std::unique_ptr<int> unique(new int(20));
  std::shared_ptr<int> shared(new int(20));
}
```

```
void CreateMakeShared()
{
  auto unique = std::make_unique<int>(20);
  auto shared = std::make_shared<int>(20);
}
```

std::make_unique und std::make_shared geben die Übergabeparameter an den Konstruktor des Objekts vom Typ T weiter.

Intelligente Zeiger der STL
Instanziierung – Eine Laufzeitanalyse

- Unterschiedliche Performance durch unterschiedliches Handling
- 100.000.000 Objekte erstellen und löschen. Zeitmessung in Sekunden.

Compiler	Optimization	new	std::shared_ptr	std::make_shared	std::unique_ptr	std::make_unique
GCC	no	3.03	13.48	30.47	8.74	9.09
GCC	yes	3.03	6.42	3.24	3.07	3.04
cl.exe	no	8.79	25.17	18.75	11.94	13.00
cl.exe	yes	7.42	17.29	9.40	7.58	7.68

Quelle:
https://www.modernescpp.com/index.php/memory-and-performance-overhead-of-smart-pointer

std::make_unique und std::make_shared geben die Übergabeparameter an den Konstruktor des Objekts vom Typ T weiter. Optimization matters.

Kurseinheit 10: Erweitertes C++

1. Intelligente Zeiger
2. **Lambdas**

Einführung

Lambda Funktionen sind keine aufwändige Erweiterung der Programmiersprache, sondern bieten die Möglichkeit bereits zur Verfügung stehende Funktionalität einfacher und übersichtlicher einzusetzen, was zu einem enormen Gewinn der Programmiersprache beiträgt.

```
void main(void)
{
  auto lambda = [] ()
  {
     std::cout << "Hello Lambda" << std::endl;
  };

  lambda();
}
```

Definition

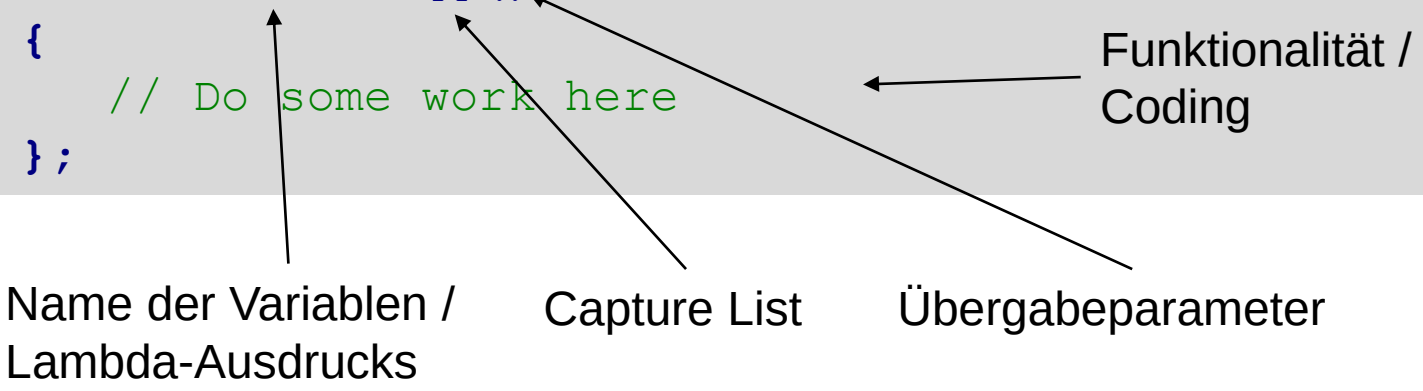
Ausführen / Aufruf der Lambda-Funktion

Lambda-Expressions sind Funktionen, welche im Code definiert und jederzeit aufgerufen werden können. **Vorherige Deklaration nicht notwendig**.

2. Lambda Expression

Definition

```
auto lambda = [] ()
{
    // Do some work here
};
```



Übergabeparameter

```
auto lambda = [] (const std::string& text)
{
  std::cout << text << std::endl;
};

lambda("Hello Lambda");
```

Übergabeparameter können an Lambda-Funktion beim Aufruf übergeben werden. Die Implementierung kann auf diese zugreifen. Es gelten die gleichen Regeln wie bei allen Funktionsaufrufen (Call-By-Value, …).

2. Lambda Expression

Definition

```
auto lambda = [] ()
{
    // Do some work here
};
```

Funktionalität / Coding

Name der Variablen / Lambda-Ausdrucks

Capture List (Gedächtnis)

Übergabeparameter

Capture List

```
void foo()
{
  int x = 5;
  auto lambda = [x]()
  {
    std::cout << "X: " << x << std::endl;
  };

  x = 20;
  lambda();
}
```

Kein Übergabeparameter. Wert kommt von Capture-List

Output: „X: 5“

2. Lambda Expression

Capture List Copy

```
int x = 20;
Circle c;

[]  // nothing
[x] // copy x
[c] // copy of circle c
[=] // copy everything in scope (x and c)
```

Kopie der spezifizierten Objekte wird im Gedächtnis der Lambda-Funktion gespeichert. Ein nachträgliches Ändern der Variablen führt zu keiner Änderung im Lambda-Objekt.

Capture List Reference

```
int x = 20;
Circle c;

[&x] // Reference int& x
[&c] // Reference Circle& c
[&]  // Reference everything within scope (x and c)
[this] // -> access to attributes and methods of object
```

Variablen werden per Referenz im Gedächtnis der Lambda-Funktion gespeichert. Ein nachträgliches Verändern der Variablen führt zu einer Änderung im Lambda-Objekt.

Capture List Reference

```
int x = 20;
Circle c;
[&x] // Reference int& x
[&c] // Reference Circle& c
[&]  // Reference everything within scope (x and c)
```

Beispiel Reference

```
void foo()
{
  int x = 20;
  auto lambda = [&x] ()
  {
    std::cout << x;
  };

  x = 30;
  lambda();
}
```

Output: 30

Generelles [&=] sollte vermieden werden, da alle Variablen im Scope gespeichert werden.

Handling von Lambda-Funktionen

Lambda-Funktionen bieten eine sehr elegante und einfache Möglichkeit eine Funktion innerhalb des Codes zu definieren (In-Place).

Großer Nutzen: Weitergabe von Lambda-Funktionen

```
auto lambda = [] ()
   {
      std::cout << "Hello Lambda" << std::endl;
   };

   foo(. . . .);
}
```

```
void foo( . . . . . )
{
    lambda();
}
```

Lambda-Ausdrücke erstellen und zu einem späteren Zeitpunkt, an einem anderen Ort, Ausführen. Capture Variablen werden mit transportiert.

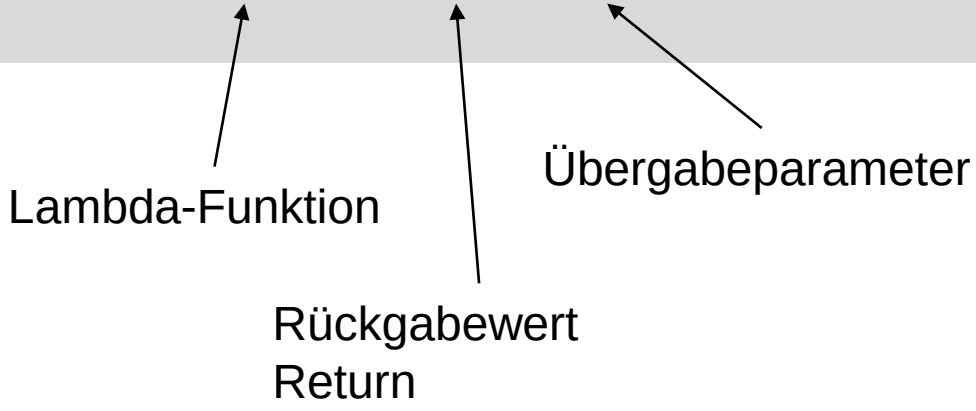
Handling von Lambda-Funktionen

Lambda-Funktionen bieten eine sehr elegante und einfache Möglichkeit eine Funktion innerhalb des Codes zu definieren (In-Place).

Großer Nutzen: Weitergabe von Lambda-Funktionen

```
auto lambda = [] () { ... };

std::function<void(void)> = lambda;
```



Handling von Lambda-Funktionen

Lambda-Funktionen bieten eine sehr elegante und einfache Möglichkeit eine Funktion innerhalb des Codes zu definieren (In-Place).

Großer Nutzen: Weitergabe von Lambda-Funktionen

```
std::function<int(int)> lambda = [] (int value)
{
  return value * 5;
};

int result = lambda(5);
```

result: 25

Handling von Lambda-Funktionen

Lambda-Funktionen bieten eine sehr elegante und einfache Möglichkeit eine Funktion innerhalb des Codes zu definieren (In-Place).

Großer Nutzen: Weitergabe von Lambda-Funktionen

```
auto lambda = []()
   {
      std::cout << "Hello Lambda" << std::endl;
   };

   foo(lambda);
}
```

```
void foo(std::function<void(void)> lambda)
{
    lambda();
}
```

Lambda-Ausdrücke erstellen und zu einem späteren Zeitpunkt, an einem anderen Ort, Ausführen. Capture Variablen werden mit transportiert.

Lambda: Ein tieferer Blick

Lambda-Funktionen werden im Code definiert und können im Scope befindliche Variablen speichern (siehe Capture List).

```
int x; Circle c;
auto lambda = [x, &c] (int value) { // code };
lambda(5);
```

```
// Simplified compiler generated lambda
class lambda_31_7 {
public:
   void operator()(int value) { code }

private:
  int x;
  Circle& c;
}
```

Compiler generierte Klasse

Lambda: Ein tieferer Blick
Vereinfachung durch Templates

```
int x; Circle c;
auto lambda = [x, &c] (auto value) { // code };
lambda(5);
```

Vereinfachung durch Template-Argumente

Compiler generierte Klasse

```
template<class T>
class lambda_31_7 {
public:
   void operator()(T value) { code }

private:
  int x;
  Circle& c;
}
```

Lambda: Anwendungsbeispiele
STL

Vergleichsoperator zur Sortierung von Circle Objekten

```
std::vector<Circle> circles;

std::sort(circles.begin(), circles.end(),
      [](const auto& lh, const auto& rh)
      {
         return lh.Radius() > rh.Radius();
      });
```

Vergleichsoperator als Lambda-Funktion

Sortieren nach Radius

```
std::sort(circles.begin(), circles.end(),
      [](const auto& lh, const auto& rh)
      {
         return lh.X() > rh.X();
      }
```

Sortieren nach Position

Vergleichsoperator muss nicht als Callback-Funktion oder als Methode der Klasse implementiert werden.

Situationsbedingte individuelle Anpassung möglich

Lambda: Anwendungsbeispiele
STL

Bedingte Suche in Container

```
std::vector<Circle> circles;

auto it = std::find_if(circles.begin(), circles.end(),
      [](const auto& c)
      {
         return c.Radius() > 100;
      });
```

Suchparameter als Lambda-Funktion

Kreis mit Radius > 100 finden

Suchparameter muss nicht als Methode oder Funktion implementiert werden

Situationsbedingte individuelle Anpassung möglich

Lambda: Anwendungsbeispiele
STL

std::for_each

```
std::vector<Circle> circles;                                    Custom Operation


std::for_each(circles.begin(), circles.end(),
      [](const auto& c)
      {
         std::cout << „Radius: „ << c.Radius() << std::endl;
      });
```

Benutzerdefiniertes Lambda wird für jedes Element in circles ausgeführt.
Capture-List erlaubt zusätzliche Daten zur Verarbeitung

Lambda: Anwendungsbeispiele
Threading

Thread-Funktion als Lambda

Thread Funktion als Lambda

```
WorkloadData data;

std::thread t(
[data]()
{
   std::cout << "Threading in C++ is great" << std::endl;
   data.Process();
});

// Share data via Capture List
// Run lambda in separate thread
t.run();
```

Zusammenfassung

- In-Place Definition
  - Im gleichen Scope, wo auch die Verwendung ist.
- Kein unnötiger Code
  - Klasse und Capture werden automatisch generiert.
  - Operator() wird automatisch generiert.
  - Häufig „Wegwerf-Code“
- Schnell in der Handhabung
  - Flexibel durch Capture und Übergabeparameter
  - Weitere Vereinfachung durch auto Syntax (C++17)
- Lambdas können auch in Containern gespeichert werden
  - Mapping
  - Funktions-Array
- Versionsabhängig
  - Fähigkeiten von Lambdas unterscheiden sich abhängig vom C++ Standard