40 Jahre Evolution: Von Funktionen zu Coroutinen

Rainer Grimm

Training, Mentoring und Technologieberatung

Callable

- Funktion
- Überladen von Funktionen
- Funktionsobjekt
- Lambda-Ausdruck
- Coroutine

Callable

- Funktion
- Überladen von Funktionen
- Funktionsobjekt
- Lambda-Ausdruck
- Coroutine

Funktion

Eine Funktion ist eine Abfolge von Anweisungen, die eine bestimmte Aufgabe erfüllen und als Einheit verpackt sind.

- Implementierung
  - Jeder Funktionsaufruf erzeugt einen Stackframe auf dem Stack
  - Der Stackframe enthält die privaten Daten des Funktionsaufrufs (Parameter, Locals und die Rücksprungadresse)
  - Am Ende der Funktion wird der Stackframe gelöscht

Funktion in der Mathematik != Funktion in der Programmierung

Reine Funktionen

Reine Funktionen (Mathematische Funktionen)

- erzeugen dasselbe Ergebnis, wenn sie dieselben Argument erhalten (referentielle Transparenz).
- besitzen keine Seiteneffekte.
- ändern nicht den Status des Programms.

- Vorteile
  - Einfach zu testen und zu refaktorieren
  - Die Aufrufreihenfolge von Funktionen kann geändert werden
  - Automatisch parallelisierbar
  - Ergebnisse können zwischengespeichert werden

Reine Funktionen

Die Arbeit mit reinen Funktionen basiert auf Disziplin

➡ Klassische Funktionen, Meta-Funktionen, `constexpr-` oder `consteval`-Funktionen verwenden

- Funktion

```
int powFunc(int m, int n){
    if (n == 0) return 1;
    return m * powFunc(m, n-1);
}
```

- Meta-Funktion

```
template<int m, int n>
struct PowMeta {
    static int const value = m * PowMeta<m, n-1>::value;
};
template<int m>
struct PowMeta<m, 0>{
    static int const value = 1;
};
```

Reine Funktionen

- `constexpr` Funktion (C++14)

```
constexpr int powConstexpr(int m, int n) {
    int r = 1;
    for(int k=1; k<=n; ++k) r*= m;
    return r;
}
```

- `consteval` Funktion (C++20)

```
consteval int powConsteval(int m, int n) {
    int r = 1;
    for(int k=1; k<=n; ++k) r*= m;
    return r;
}
```

1024 == powFunc(2, 10) == PowMeta<2, 10>()::value
== powConstexpr(2, 10) == powConsteval(2, 10)

Callable

- Funktion
- **Überladen von Funktionen**
- Funktionsobjekt
- Lambda-Ausdruck
- Coroutine

Überladen von Funktionen

Das Überladen von Funktionen ermöglicht es, mehrere Funktionen mit demselben Namen, aber unterschiedlichen Parametern zu erstellen.

- Der Compiler versucht, die einzige, am besten geeignete Funktion auf der Grundlage des Überladens von Funktionen zu finden.

- Implementierung:
  - Der Compiler dekoriert die Funktionsnamen mit den Funktionsparametern

➡ Name Mangling

Funktion	GCC 8.2	MSVC 19.16
`print(int)`	`_Z5printi`	`?print@@YAXH@Z`
`print(double)`	`_Z5printd`	`?print@@YAXN@Z`
`print(const char*)`	`_Z5printPKc`	`?print@@YAXPEBD@Z`
`print(int, double, const char*)`	`_Z5printidPKc`	`?print@@YAXHNPEBD@Z`

Überladen von Funktionen

- Die einzige am besten passende Funktion
  - Funktionen: Weniger und weniger kostspielige Konvertierungen sind besser
  - Funktionen und Funktions-Templates: Funktionen sind besser
  - Spezialisierung von Funktion-Templates: Stärker spezialisierte Funktions-Templates sind besser
  - Concepts: Stärker eingeschränkte Funktions-Templates sind besser

Die ganze Geschichte: Overload resolution on cppreference.com

Überladen von Funktionen

- Templates

```
void onlyDouble(double){}

template <typename T>
void onlyDouble(T) = delete;

int main(){
    onlyDouble(3.14);  // OK
    onlyDouble(3.14f); // ERROR
}
```

Überladen von Funktionen

- Concepts

```
template<std::forward_iterator I> void advance(I& iter, int n) {}
template<std::bidirectional_iterator I> void advance(I& iter, int n){}
template<std::random_access_iterator I> void advance(I& iter, int n){}

std::forward_list myFL {1, 2, 3};
std::list  myL{1, 2, 3};
std::vector myV{1, 2, 3};

std::list<int>::iterator lIt = myL.begin();
advance(lIt, 1);       // std::bidirectional_iterator

std::vector<int>::iterator vIt = myV.begin();
advance(vIt, 1);       // std::random_access_iterator

std::forward_list<int>::iterator fwIt = myFL.begin();
advance(fwIt, 1);      // std::forward_iterator
```

Callable

- Funktion
- Überladen von Funktionen
- **Funktionsobjekt**
- Lambda-Ausdruck
- Coroutine

Funktionsobjekt

Ein Funktionsobjekt (auch Funktor genannt) ist ein Objekt, das aufgerufen kann wie eine Funktion.

- Ein Funktionsobjekt kann einen Zustand besitzen.

- Implementierung:

  - Der Compiler bildet einen Funktionsaufruf auf ein Objekt `obj obj(arguments)` auf den Funktionsaufrufoperator `obj.operator()(arguments)` ab.
  - Mit C++ Insights analysierte Funktionsobjekte.

Funktionsobjekt

- Operator
  - Arithmetik
    - `std::plus, std::minus, std::multiplies, std::divides, std::modulus, std::negate`
  - Vergleiche
    - `std::equal_to, std::not_equal_to, std::greater, std::less, std::greater_equal, std::less_equal`
  - Logisch
    - `std::logical_and, std::logical_or, std::logical_not`
  - Bitweise
    - `std::bit_and, std::bit_or, std::bit_xor, std::bit_not`

Funktionsobjekt

- Referenz Wrapper: `std::reference_wrapper<type>`
  - Speichert eine Referenz in einem kopierbaren Funktionsobjekt
  - Zwei Hilfsfunktionen:
    - `std::ref` erzeugt einen Referenz-Wrapper
    - `std::cref` erstellt einen konstanten Referenz-Wrapper

```
#include <functional>
#include <vector>

int main() {
    int a{2011};
    std::vector<std::reference_wrapper<int>> myIntVec{ std::ref(a) };
    a = 2014;
    myIntVec[0] << std::endl;   // 2014
}
```

Funktionsobjekt

Funktions-Wrapper und Partial Function Application

- `std::function`: Wrapper für ein Callable mit einer spezischen Funktionssignatur

- `std::bind`: bindet Argumente an ein Funktionsobjekt
- `std::bind_front`: bindet Argumente von links an ein Funktionsobjekt
- `std::bind_back`: bindet Argumente von rechts an ein Funktionsobjekt

Funktionsobjekt

```
using namespace std::placeholders;

std::function<int(int)> minus1 = std::bind(std::minus<int>(), 2020, _1);
std::cout << minus1(9);       // 2011

std::function<int()> minus2 = std::bind(minus1, 9);
std::cout << minus2();        // 2011

std::function<int(int, int)> minus3 = std::bind(std::minus<int>(), _2, _1);
std::cout << minus3(9, 2020); // 2011

std::function<int(int)> minus4 = std::bind_front(std::minus<int>(), 2020);
std::cout << minus4(9);       // 2011

std::function<int(int)> minus5 = std::bind_back(std::<int>(), 9);
std::cout << minus5(2020);    // 2011
```

Callable

- Funktion
- Überladen von Funktionen
- Funktionsobjekt
- **Lambda-Ausdruck**
- Coroutine

Lambda-Ausdruck

Ein Lambda-Ausdruck (anonyme Funktion) ist eine Funktionsdefinition, die keinen Namen besitzt.

- Schritte in der Entwicklung von Lambdas
  - C++11: Lambda-Ausdrücke
  - C++14: Generische Lambda-Ausdrücke
  - C++20: Template-Parameter für Lambda-Ausdrücke

- Implementierung:
  - Der Compiler erzeugt eine Klasse mit einem überladenen Funktionsaufrufoperator.
  - Lambda-Ausdrücke mit C++ Insights

Lambda-Ausdruck

- Template-Parameter für Lambda Ausdrücke

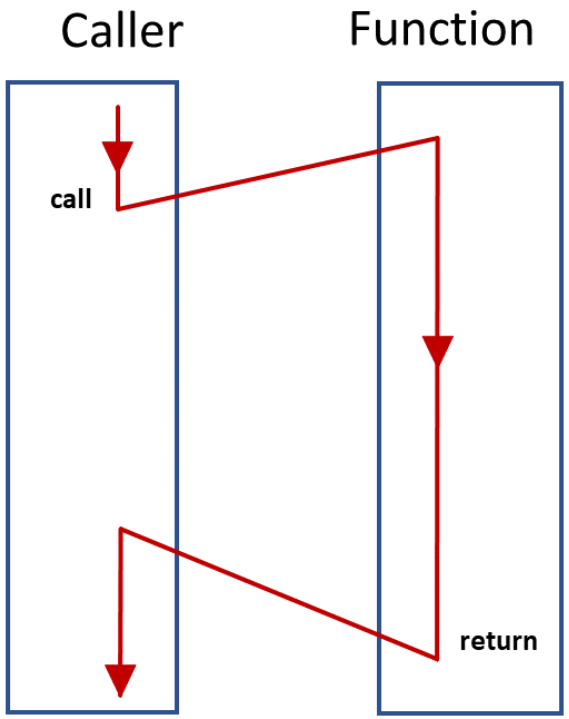
```
auto lambdaGeneric = [](const auto& container) { return container.size(); };

auto lambdaVector = []<typename T>(const std::vector<T>& vec) {
    return vec.size();
};

auto lambdaVectorIntegral = []<std::integral T>(const std::vector<T>& vec) {
    return vec.size();
};

std::deque<int> deq;              // OK for lambdaGeneric
std::vector<double> vecDouble;    // OK for lambdaGeneric, lambdaVector
std::vector<int> vecInt;          // OK for lambdaGeneric,
                                  // lambdaVector,lambdaVectorIntegral
```

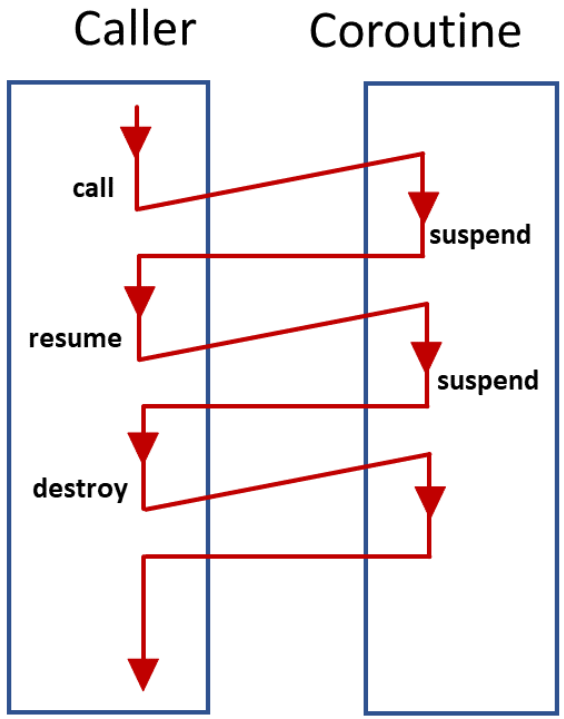
Callable

- Funktion
- Überladen von Funktionen
- Funktionsobjekt
- Lambda-Ausdruck
- **Coroutine**

Coroutinen

Coroutinen sind verallgemeinerte Funktionen, die ihre Ausführung unterbrechen und wieder aufnehmen können und dabei ihren Zustand speichern.





Charakteristiken

Zwei neue Konzepte

- `co_await expression`: Die Ausführung von `expression` pausieren und wieder aufnehmen
- `co_yield expression`: Unterstützung von Generatoren

- Typische Einsatzgebiete
  - Kooperative Tasks
  - Eventschleifen
  - Unendliche Datenströme
  - Pipelines

Charakteristiken

Design Principles (James McNellis)

- **Scalable**, to billions of concurrent coroutines
- **Efficient**: Suspend/resume operations comparable in cost to function call overhead
- **Open-Ended**: Library designers can develop coroutines libraries
- **Seamless Interaction** with existing facilities with no overhead.
- **Usable** in environments where exceptions are forbidden or not available.

Charakteristiken

	Funktion	Coroutine
Aufruf	`func(args)`	`func(args)`
Rückgabe	`return statement`	`co_return statement`
Unterbrechung		`co_await expression` `co_yield expression`
Wiederaufnahme		`coroutine_handle<>::resume()`

Eine Funktion ist eine Coroutine, falls sie einen Aufruf `co_return`, `co_await`, oder `co_yield` enthält.

Coroutinen: Generatoren

```
Generator getNext(int start = 0, int step = 1) {
    auto value = start;
    while(true) {
        co_yield value;
        value += step;
    }
}
...

auto gen = getNext(-10);
for (int i= 1; i <= 20; ++i) std::cout << gen.getNext() << " ";

auto gen2 = getNext(0, 5);
while (true) std::cout << gen2.getNext() << " ";
```

Implementierung

Callable

- Funktion
- Überladen von Funktionen
- Funktionsobjekt
- Lambda-Ausdruck
- Coroutine

Blog: www.ModernesCpp.com
Mentoring: www.ModernesCpp.org

Rainer Grimm

Training, Mentoring und
Technologieberatung