Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

15. November 2019

Funktionale Programmierung

Inhalt

- Definition
- Funktionale Elemente in C++
- Funktor
- Lambda
- Funktionsobjekte
- Manuelles statisches Binding

Definition

- C++ ist eine Multiparadigmen-Sprache
  - imperativ
  - objektorientiert
  - funktional
- Funktionale Programmiersprachen
  - LISP
  - Haskell
  - F#
  - Scala
- Eigenschaften funktionaler Programmiersprachen
  - Programme sind Funktionen
  - Objekte sind unveränderlich (immutable)
  - Funktionen produzieren neue Objekte
  - Funktionen sind auch Objekte

Funktionale Elemente in C++

- Funktion
  - typisierte Parameterlisten
  - variable Anzahl Parameter (variadic)
  - global oder als Methode eine (unveränderbaren) Klasse
- Funktor
  - Klasseninstanz mit Funktionsoperator
- Funktions-Pointer
  - Adresse auf eine Funktion
- Methoden-Pointer
  - Adresse auf eine Methode einer Klasseninstanz
- Lambda
  - Lambda-Expression erzeugt einen anonymen Funktor
  - kann innerhalb von Funktionen definiert werden
- Funktionsobjekt
  - Verallgemeinerung all dieser Konzepte

Funktor

- Die Aufgabe einer Funktion wird von einem Objekt übernommen
- Überladen des Funktionsoperators: `operator()(...)`
- virtuelle Funktionen und Vererbung möglich

```
1  struct Duplicator {
2    double operator()(double arg) {
3      return arg * 2;
4    }
5  };
6
7  Duplicator d;
8  auto result = d(21);     // 42
```

Beispiel: Funktor

```
1  struct Multiplier {
2    explicit Multiplier(double value = 2)
3      : value_(value) {}
4
5    double operator()(double arg) const {
6      return value_ * arg;
7    }
8
9   private:
10   double value_;
11 };
12
13 Multiplier duplicator;
14 auto r1 = duplicator(21);   // 42
15
16 Multiplier triplicator(3);
17 auto r2 = triplicator(21);  // 63
```

Lambda

```
1  // Syntax:
2  [Zugriffsdeklaration](Parameterliste) -> Rückgabetyp { body }
3
4  // Beispiel:
5  [bias](int x, int y) -> int { return bias + x + y; }
```

- Zugriffsdeklaration definiert Zugriff nach aussen
  - auch Capture-Clause genannt
- Rückgabetyp ist optional
- Lambda-Expression erzeugt einen neuen anonymen Typ

Zugriffsdeklaration (Capture Clause)

- `[]` kein Zugriff auf äussere Variablen
- `[=]` alle sichtbaren Variablen sind verfügbar via Kopie
- `[&]` alle sichtbaren Variablen sind verfügbar via Referenz (veränderbar)
- `[my_var]` nur my_var wird reinkopiert
- `[&my_var]` nur my_var ist via Referenz verfügbar
- `[this]` this-Pointer wird reinkopiert und macht somit alle Member der Instanz verfügbar
- Mischen erlaubt, z.B: `[=, &my_var]`
- Lambda capture expressions erlaubt: `[value = std::move(uptr)]`

Best-Practice

- so wenig wie möglich verfügbar machen
- Lebensdauer des Lambdas kann länger sein als z.B. Klasseninstanz!

Closure

- verallgemeinertes Konzept von Lambdas
- Closures sind veränderbar
- Lambda mittels `mutable` veränderbar machen

```
1  // Lambda als Closure
2  int bias = 2;
3  auto l = [bias](int x) mutable { return x * bias++; };
4  auto v = l(3);  // v = 6
5  // bias = 3 (Kopie innerhalb von l)
```

Wie funktionieren Lambdas?

```
1  int by_val = 4;
2  int by_ref = 5;
3
4  // Lambda
5  auto op1 = [by_val, &by_ref](int i) {
6    ++by_ref;
7    return i + by_val + by_ref;
8  };
9
10 auto res1 = op1(10);     // 20
11 // by_ref = 6
```

```
1  // Lambda als Funktor
2  struct Op {
3    Op(int by_val, int& by_ref)
4      : by_val_{by_val}
5      , by_ref_{by_ref} {}
6    int operator()(int i) const {
7      ++by_ref_;
8      return i + by_val_ + by_ref_;
9    }
10  private:
11   int by_val_;
12   int& by_ref_;
13 };
14 Op op2(by_val, by_ref);
15
16 auto res2 = op2(10);     // 21
17 // by_ref = 7
```

Funktionsobjekte: `std::function`

- Allgemeines Handle für Funktionsobjekte: `std::function`
- Kann unterschiedlich initialisiert werden:
  - durch Funktionspointer
  - durch Lambda
  - durch Funktor
  - durch Methodenpointer
- Definiert Signatur und Rückgabewert
- vereinheitlichte Verwendung
- Wichtig für Algorithmen

```
1 #include <functional>
2 using Converter = std::function<int(int)>;
3
4 int UseConverter(const Converter& conv, int arg) {
5   return conv(arg);
6 }
```

Initialisierung von Funktionsobjekten

```
1  // Funktionspointer
2  int ConvFunction(int a) {
3    return a * 2;
4  }
5  Converter conv1 = &ConvFunction;
6  // conv1 hat intern Pointer auf ConvFunction
7
8  // Lambda
9  Converter conv2 = [](int a){ return a * 3; };
10 // conv2 hat seine eigene Kopie der Lambda-Instanz
11
12 // Funktor
13 struct ConvFunktor {
14   int operator()(int a) const { return a * 4; }
15 };
16 Converter conv3 = ConvFunktor();
17 // conv3 hat seine eigene Kopie der Funktor-Instanz
```

Manuelles statisches Binding: `std::bind`

- Funktionsobjekte können nachträglich verändert werden
- einzelne Parameter können gebunden werden
- `std::placeholders` verwenden um freie Parameter zu definieren
- erlaubt einfachere Verwendung von Methodenpointer

```
1  int Sum(int a, int b, int c) {
2    return a + b + c;
3  }
4
5  Converter add30 = std::bind(&Sum, 10, std::placeholders::_1, 20);
6  // add30 hat intern Pointer auf Funktion Sum
7  // wir verwenden _1 um einen einzelnen freien Parameter zu definieren
8
9  std::cout << add30(1) << std::endl;   // 31
```

Beispiel: `std::bind` mit Methodenpointer

```
1  struct SumClass {
2    explicit SumClass(int a) : a_(a) {}
3    int Execute(int b, int c) {
4      return a_ + b + c;
5    }
6   private:
7    int a_;
8  };
9
10 SumClass add5(5);
11 Converter add7 = std::bind(&SumClass::Execute, &add5, 2, std::placeholders::_1);
12 // add7 hat intern nur Pointer auf Methode SumClass::Execute
13 // und Objekt add5
14
15 std::cout << add7(1) << std::endl;    // 8
```

Funktionsobjekte erlauben Funktionale Programmierung

- mittels Funktionsobjekten können Funktionen generisch werden
- Eine Funktion erhält eine Funktion als Parameter
  $\rightarrow \lambda - calculus$

```
1  std::vector values = { 1, 2, 3, 4 };
2
3  struct Adder {
4    int operator()(int a, int b) { return a + b; }
5  };
6  auto result_add = std::accumulate(values.cbegin(), values.cend(), 0, Adder());
7  std::cout << result_add << std::endl;   // 10
8
9  const auto lambda = [](auto a, auto b) {
10   return a * b;
11 };
12 auto result_mult = std::accumulate(values.cbegin(), values.cend(), 1, lambda);
13 std::cout << result_mult << std::endl;   // 24
```