Programmieren in C++

Funktionale Programmierung

Inhalt

- Funktionale Programmierung in C++
- Funktor
- Lambda
- Closure
- Funktionsobjekt

Funktionale Programmierung

- C++ ist ein Multi-Paradigmen-Sprache
  - imperatives Programmieren
  - objektorientiertes Programmieren
  - funktionales Programmieren
- Funktionale Programmierung
  - Programme sind Funktionen
  - Objekte sind unveränderbar (immutable)
  - Funktionen produzieren neue Objekte
  - Funktionen sind selber auch Objekte
- Funktionale Programmiersprachen
  - LISP
  - Haskell
  - F#
  - Scala

Funktionale Elemente von C++

- Funktion
  - typisierte Parameterlisten
  - variable Anzahl Parameter
  - global oder als Methode einer (unveränderbaren) Klasse
- Funktor
  - Klasseninstanz, welche den Funktionsoperator operator()(…) überlädt
- Funktionszeiger
  - Adresse auf eine Funktion
- Methodenzeiger
  - Adresse auf eine an eine Instanz gebundene Methode
- Lambda
  - anonymer Funktor (kann auch innerhalb einer Funktion definiert sein)
- Funktionsobjekt
  - Verallgemeinerung all dieser Konzepte
  - Instanz der Klasse functional aus dem Header <functional>

Funktor

- Objekte als Funktionen
  - die Aufgabe einer Funktion wird von einem Objekt übernommen
  - Überladen des Funktionsoperators operator()(…)
  - Anwenden des Funktionsoperators entspricht dem Aufrufen des Funktionsoperators für das entsprechende Objekt
- Beispiel

```
enum Modus { Rad, Deg, Gon };
class Sinus {
   Modus m_mode;
public:
   Sinus(Modus mode = Rad)
      : m_mode(mode) {}
   double operator()(double arg) {
      switch(m_mode) {
         case Rad: return sin(arg);
         case Deg: return sin(arg/180.0*M_PI);
         case Gon: return sin(arg/200.0*M_PI);
} } };
```

Einsatz

```
int main() {
   Sinus sinrad;
   Sinus sindeg(Deg);

   cout << sinrad(M_PI/4);
   cout << sindeg(45.0);
}
```

Lambda

- Syntax
  - Zugriffsdeklaration Parameterliste [-> Rückgabetyp] Funktionskörper
- Beispiel
  - `[bias] (int x, int y) -> int { return bias + x + y; }`
- Zugriffsdeklaration
  - gibt in eckigen Klammern an, auf welche Variablen der Umgebung zugegriffen werden kann
- Parameterliste
  - Deklaration der Funktionsargumente analog zu normalen Funktionen
- Rückgabetyp
  - die Angabe des Rückgabetyps ist optional (kann vom Compiler selber ermittelt werden), darf auch void sein (Prozedur)
- Funktionskörper
  - ein gewöhnlicher Funktionskörper mit oder ohne return-Anweisung

Lambda Zugriffsdeklaration

- Hintergrund
  - dort wo der Lambda-Ausdruck definiert wird, existiert eine lokale Umgebung bestehend aus lokalen Variablen und Instanzvariablen
  - in der Zugriffsdeklaration wird angegeben, auf welche Variablen der Umgebung zugegriffen wird und ob der Zugriff by-value oder by-reference stattfinden soll
  - auf statische und globale Variablen kann immer zugegriffen werden, auch ohne Angabe in der Zugriffsdeklaration
- Beispiele
  - [bias] auf die Variable bias wird by-value zugegriffen
  - [&bias] auf die Variable bias wird by-reference zugegriffen
  - [=] auf alle Variablen der Umgebung wird by-value zugegriffen
  - [&] auf alle Variablen der Umgebung wird by-ref. zugegriffen
  - [this] auf alle Member der übergebenen Instanz wird by-pointer zug.
  - [=, &bias] nur auf bias wird by-ref. zugegriffen, sonst by-value
  - [factor, &bias] auf factor wird by-value und auf bias by-ref. zugegriffen

Einsatz von Lambda

- Beispiel: Iterieren durch die Elemente eines Containers

```
vector<int> v;
```

  - ohne Lambda

```
for ( auto it = v.begin(), end = v.end(); it != end; it++ ) {
  cout << *it;
}
```

  - mit Lambda

```
for_each( v.begin(), v.end(), [] (int val) {
  cout << val;
});
```

- Absteigend Sortieren mittels eines Komparators

```
sort(v.begin(), v.end(), [] (int v1, int v2) {
  return v1 > v2;
});
```

Lambda hinter der Kulisse

```
void fLambda() {
   int notUsed = 3, byval = 4, byref = 5;
   auto op = [byval, &byref](int i) {
      ++byref; return i + byval + byref;
   };
   cout << op(10) << endl;
}

void fFunctor()  {
   int notUsed = 3, byval = 4, byref = 5;
   class Op {
      const int m_val;
      int& m_ref;
      Op(int val, int& ref) : m_val{val}, m_ref{ref} {}
      int operator()(int i) const {
         ++m_ref; return i + m_val + m_ref;
      }
   } op(byval, byref);
   cout << op(10) << endl;
}
```

Closure

- Closure
  - anonyme Funktion, welche Zugriff auf ihren Erstellungskontext hat und diesen auch verändern kann
  - Umsetzung in C++: lambda mittels mutable veränderbar machen
- Beispiel

```
int f = 2; // f ist eine lokale Variable

auto l0 = [f](int x) {
     return x*f++;
};    // nicht möglich, weil f im Lambda unveränderbar ist
auto l1 = [&f](int x) { return x*f++; };  // f wird bei by ref übergeben
auto l2 = [f](int x) mutable {
     return x*f++;
}; // f ist im Lambda veränderbar

cout << "value = " << l1(3) << ", f = " << f << endl; // value = 6, f = 3
cout << "value = " << l1(3) << ", f = " << f << endl; // value = 9, f = 4
cout << "value = " << l2(3) << ", f = " << f << endl; // value = 6, f = 4
cout << "value = " << l2(3) << ", f = " << f << endl; // value = 9, f = 4
```

Funktionsobjekte im Einsatz

```
#include <functional> // … <vector>, <numeric>
void main() {
   // Deklaration des Funktionsobjekts
   function<float (float a, int x)> func;
   vector<int> v{1, 2, 3, 4, 5};

   func = … // Definition des Funktionsobjekts
            // (Funktor, Funktionszeiger, Methodenzeiger, Lambda)
            // siehe nächste Folie

   // Einsatz des Funktionsobjekts in einem Algorithmus
   float r = accumulate(v.cbegin(), v.cend(), 1.0f, func);
}
```

Verschiedene Funktionsobjekte

Funktor, Funktionszeiger

```
struct Funktor {
   float m_div;
   Funktor(float f) : m_div(f) {}
   float operator()(float a, int x) const
   {
      return a + x/m_div;
   }
};
----
func = Funktor(2.0f);
----
float foo(float a, int x) { return a +
x/2.0f;}
func = &foo;
```

Methodenzeiger, Lambda

```
struct C {
   float m_div;
   C(float f) : m_div(f) {}
   float meth(float a, int x) const {
         return a + x/m_div;
   }
};
----
C c(2.0f);
func = bind(&C::meth, &c, _1, _2);
----
func = [](float a, int x) { return a +
x/2.0f; };
```