Funktionstypen

⟨Fun'typ⟩ → ⟨Rückgabetyp⟩ “(” ⟨Parametertypliste⟩ “)”

Funktionen haben Typen. Syntaktisch besteht der Typ einer Funktion aus dem Rückgabetyp gefolgt von den Typen der Parameter in runden Klammern.

Funktionszeiger

Analog wie sich für eingebaute Datentypen (`int`, `double`, etc.) bzw. den Namen von Klassen jeweils ein assoziierter Zeigertyp ableiten lässt, ist das auch für Funktionstypen möglich.

Beispiel (Vereinbarung und Aufruf von Funktionszeigern). Wir vereinbaren ein Typsynonym für einen Funktionstyp, den Typ von Funktionen mit einem Parameter vom Typ `double` und Rückgabetyp `double`.

Im Hauptprogramm vereinbaren wir eine Variable `funptr` vom Typ Zeiger auf Funktion mit einem Parameter vom Typ `double` und Rückgabetyp `double` und weisen ihr als Wert eine Adresse zu, die assoziiert ist mit der Funktion `sqrt` aus `cmath`. Hierfür wird der Name `cmath` automatisch konvertiert in eine geeignete Speicheradresse. Danach wird der Funktionsaufrufsoperator auf die Variable `funptr` angewandt was die mit der in `funptr` enthaltenen Speicheradresse assoziierte Funktion mit den gegebenen Parametern aufruft.

funtyp_demo.cpp

```cpp
#include <cmath>
#include <iostream>

using namespace std;

using ArithFun = double(double);

int main() {
  ArithFun* funptr = sqrt;

  cout << funptr(2) << endl;
}
```

```
1.41421
```

Funktionsnamen werden automatisch umgewandelt in Funktionszeiger, wo benötigt.

Es kann der Funktionsaufrufsoperator auf Funktionszeiger-Werte angewandt werden. Es wird hierbei die mit der Speicheradresse, die dargestellt wird vom Funktionszeiger-Wert, assoziierte Funktion mit den gegebenen Parametern aufgerufen.

Vereinbarungssyntax

Die Vereinbarung von Namen erfolgt stets in der Form $T\ \ D$ mit T einem Datentyp und D dem *Deklarator*. Der Deklarator enthält stets den zu vereinbarenden Namen.

Beispiel. Wir vereinbaren einen Namen `x` als Variable vom Typ `double`:

```
double x;
```

Der Deklarator kann auch eine Parameterliste bzw. eine Parametertypliste enthalten.

Beispiel. Wir vereinbaren einen Namen `f` als Funktion von einem Parameter vom Typ `double` und mit Rückgabetyp `double`:

```
double f(double);
```

Beispiel. Wir vereinbaren einen Namen `f` als Funktion von einem Parameter vom Typ `double` und mit Rückgabetyp Zeiger auf `double`:

```
double *f(double);
```

Wird der „Zeiger-auf“-Operator `*` bei einem Ausdruck wie zur Vereinbarung eines Namens als Funktion angewandt auf den zu vereinbarenden Namen (statt z.B. auf den Rückgabetyp wie im Beispiel), so wird der Name vereinbart als Variable von einem Typ der Form Zeiger auf Funktion.

Beispiel. Wir vereinbaren einen Namen `fptr` als Variable vom Typ Zeiger auf Funktion von einem Parameter vom Typ `double` und mit Rückgabetyp `double`:

```
double (*f)(double);
```

Als Faustregel lässt sich sagen, dass der Typ mit dem ein Name innerhalbe einer Deklaration vereinbart wird, von innen nach außen zu lesen ist.

Beispiel. Wir vereinbaren eine Funktion `signal` mit zwei Parametern vom Typ `int` und vom Typ Zeiger auf Funktion von einem Parameter `int` ohne Rückgabewert. Die Funktion `signal` hat Rückgabetyp Zeiger auf Funktion mit einem Parameter vom Typ `int` und ohne Rückgabewert.

```
void (*signal(int, void (*func)(int)))(int);
```

I.A. ist der „Umweg“ über ein Typsynonym der direkten Vereinbarungssyntax vorzuziehen.

Wird `typedef` statt `using` zur Vereinbarung eines Typsynonyms der Form Zeiger auf Funktion verwendet, so muss hierfür die direkte Vereinbarungssyntax verwendet werden.

Funktionstypen als Parameter

Wird der Parameter einer Funktion mit einem Funktionstyp vereinbart (entweder unter Verwendung eines Typsynonyms oder mit direkter Vereinbarungssyntax), so akzeptiert die Funktion als Parameter an dieser Stelle stattdessen einen Parameter vom entsprechenden Typ der Form Zeiger auf Funktion.

Beispiel (Sehnentrapezregel mit Zeiger auf Funktion). Der erste Parameter der Funktion `trapezoidal` ist mit Typ Funktion mit einem Parameter vom Typ `double` und Rückgabetyp `double` vereinbart. Dennoch akzeptiert `trapezoidal` als ersten Parameter einen Wert vom Typ Zeiger auf Funktion mit einem Parameter vom Typ `double` und Rückgabetyp `double` vereinbart.

Beim Aufruf von `trapezoidal` im Hautprogramm wird der Name der Funktion `f` automatisch umgewandelt in einen Wert von einem Typ der Form Zeiger auf Funktion. Der Wert ist eine Speicheradresse geeignet assoziiert ist mit der Funktion `f`.

Den Parameter der Funktion `trapezoidal` stattdessen als `double (*g)(double)` zu vereinbaren würde sich genau identisch verhalten.

trapezoidal_fptr.cpp

```cpp
#include <iostream>
#include <cmath>

using namespace std;

double trapezoidal(double g(double),
    int n, double a = 0, double b = 1) {
  const double h = (b - a) / n;
  double s = (g(a) + g(b))/2;
  for (int i = 1; i <= n - 1; i++)
    s += g(a + i*h);
  return h * s;
}

double f(double x) {
  x -= 0.25;
  return exp(-x * x);
}

int main() {
  for (int n = 1; n <= 512; n *= 2)
    cout << trapezoidal(f, n) << endl;

  return 0;
}
```

```
0.754598
0.847006
0.868203
0.873407
0.874702
0.875025
0.875106
```

```
0.875126
0.875131
0.875133
```

Funktionsobjekte in der Standardlibrary (aus <functional>)

Die Standardlibrary stellt diverse Werkzeuge zur Verfügung um mit (Repräsentationen von) Funktionen als Werten arbeiten zu können.

Generische Funktionsobjekte (function)

`function<T>` ist eine generische Klasse für Funktionsobjekte deren Funktionsaufrufsoperatoren sich verhalten wie eine Funktion vom Typ T.

Die Klasse `function<T>` hat einen Konstruktor der als einzigen Parameter einen beliebigen Wert akzeptiert für den der Funktionsaufrufsoperator kompatibel überladen ist. Insb. also auch Werte vom Typ Zeiger auf Funktion vom Typ T. Der Konstruktor nimmt (da er mit nur einem Parameter aufgerufen werden kann) Teil an der impliziten Typkonvertierung.

Für `function<T>` ist der Funktionsaufrufsoperator natürlich ebenfalls kompatibel zum Typ T überladen.

Beispiel (Sehnentrapezregel mit generischen Funktionsobjekten). Wir vereinbaren die Funktion `trapezoidal` mit einem Parameter vom Typ `function<T>` für T der Typ von Funktionen mit einem Parameter vom Typ `double` und Rückgabewert `double`, also `double(double)`.

Beim ersten Aufruf von `trapezoidal` im Hauptprogramm wird der Name der Funktion `f` zunächst automatisch umgewandelt in einen passenden Wert von einem Typ der Form Zeiger auf Funktion und dann von dort weiter (durch implizite Verwendung des Konstruktors der Klasse `function<T>`) in ein Objekt der Klasse `function<T>`.

Beim zweiten Aufruf wird das temporäre Objekt der Klasse `NormV` ebenfalls implizit konvertiert in ein Objekt der Klasse `function<T>`. Da der Funktionsaufrufsoperator für die Klasse `NormV` kompatibel zu T überladen ist und somit ein Objekt der Klasse `NormV` ein akzeptabler Parameterwert für den einen Parameter des Konstruktors von `function<T>` wäre, können Objekte der Klasse `NormV` implizit konvertiert werden zu Objekten der Klasse `function<T>`.

trapezoidal_stl.cpp

```cpp
#include <iostream>
#include <cmath>
#include <functional>

using namespace std;

class NormV {
  private:
    double mw, stdabw;
```

```
  public:
    NormV (double mw_ = 0, double stdabw_ = 1)
      : mw(mw_), stdabw(stdabw_) {}

    double operator()(double x) const {
      return M_2_SQRTPI / (2 * M_SQRT2 * stdabw)
        * exp(-((x - mw) * (x - mw)) / (2*stdabw*stdabw));
    }
};

double trapezoidal(function<double(double)> g,
    int n, double a = 0, double b = 1) {
  const double h = (b - a) / n;
  double s = (g(a) + g(b))/2;
  for (int i = 1; i <= n - 1; i++)
    s += g(a + i*h);
  return h * s;
}

double f(double x) {
  x -= 0.25;
  return exp(-x * x);
}

int main() {
  for (int n = 1; n <= 512; n *= 4)
    cout << trapezoidal(f, n) << endl;

  cout << endl;

  for (int n = 1; n <= 512; n *= 4)
    cout << trapezoidal(NormV{}, n) << endl;

  return 0;
}
```

```
0.754598
0.868203
0.874702
0.875106
0.875131

0.320457
0.340082
0.341266
0.34134
0.341344
```

Partielle Funktionsanwendung (`bind`)

Die Standardlibrary stellt eine Funktion `bind` zur Verfügung. Die Rückgabewerte der Funktion sind nicht näher spezifizierte Funktionsobjekte. Als ersten Parameter akzeptiert die Funktion ein beliebiges Funktions-

objekt. Wird der Funktionsaufrufparameter auf einen Rückgabewert der Funktion `bind` angewandt, so wird der Funktionsaufrufsoperator des als ersten Parameter übergebenen Funktionsobjektes ausgewertet.

Die weiteren Parameter der Funktion `bind` nach dem Ersten beschreiben wie beim eventuellen Aufruf des Funktionsaufrufsoperators auf dem Rückgabewert der Funktion `bind` der Funktionsaufruf des unterliegenden Funktionsobjektes (dem ersten Parameter) erfolgen soll. Zulässig als Parameter sind zunächst beliebige Werte (diese werden kopiert und die Kopie beim eventuellen Funktionsaufruf unverändert übergeben).

Zusätzlich enthält der Namensraum `placeholders` unterhalb von `std` spezielle Werte `_1`, `_2`, ... die ebenfalls zulässig sind als Parameter der Funktion `bind`. Beim eventuellen Funktionsaufruf eines Rückgabewerts der Funktion `bind` werden die dabei angegebenen Parameter der Reihenfolge nach abgebildet auf die Platzhalterwerte. Wo beim Aufruf der Funktion `bind` ein Platzhalter als Parameter stand, wird dann an das unterliegende Funktionsobjekt der entsprechende Parameter des eventuellen Funktionsaufrufs des Rückgabewerts der Funktion `bind` übergeben.

Beispiel (Ändern der Parameterreihenfolge einer Funktion). Wir rufen die Funktion `bind` auf mit dem Namen einer Funktion als ersten Parameter. Dieser wird automatisch umgewandelt in einen entsprechenden Funktionszeiger.

Die weiteren Parameter sind hier zwei Platzhalter, jedoch in umgekehrter Reihenfolge. Beim Aufruf des resultierenden Rückgabewertes (den wir der Variable `rev_sub` zuweisen) werden die dabei angegebenen Parameter daher *in umgekehrter Reihenfolge* an die unterliegende Funktion `do_sub` übergeben.

flip_demo.cpp

```cpp
#include <iostream>
#include <functional>

using namespace std;

double do_sub(double a, double b) {
  return a - b;
}

int main() {
  using namespace placeholders;
  function<double(double, double)>
    rev_sub = bind(do_sub, _2, _1);
  cout << rev_sub(3, 1) << endl;
}
```

```
-2
```

Beispiel (Sehnentrapezregel mit partieller Funktionsanwendung). Wir übergeben an die Funktion `trapezoidal` den Rückgabewert eines Aufrufs der Funktion `bind`.

trapezoidal_bind.cpp

```cpp
#include <iostream>
#include <cmath>
#include <functional>
```

```
using namespace std;

double trapezoidal(function<double(double)> g,
    int n, double a = 0, double b = 1) {
  const double h = (b - a) / n;
  double s = (g(a) + g(b))/2;
  for (int i = 1; i <= n - 1; i++)
    s += g(a + i*h);
  return h * s;
}

double f(double x, double offset=0) {
  x += offset;
  return exp(-x * x);
}

int main() {
  using namespace placeholders;
  function<double(double)> f_shifted
    = bind(f, _1, -0.25);

  for (int n = 1; n <= 512; n *= 2)
    cout << trapezoidal(f_shifted, n) << endl;

  return 0;
}
```

```
0.754598
0.847006
0.868203
0.873407
0.874702
0.875025
0.875106
0.875126
0.875131
0.875133
```

Referenzen (`ref`, `cref`)

Die Standardlibrary stellt zwei Funktion `ref` und `cref` zur Verfügung. Diese akzeptieren als Parameter jeweils eine Referenz bzw. eine konstante Referenz. Ihre Rückgabewerte sind ebenfalls als Parameter der Funktion `bind` zulässig und werden von dieser speziell behandelt. Bei Auswerten des Aufrufs der Funktion `bind` selbst wird eine Kopie der an `ref` bzw. `cref` übergebenen Referenz im Rückgabewert der Funktion `bind` gespeichert. Bei eventuellem Aufruf des Funktionsaufrufsoperators auf den Rückgabewert der Funktion `bind` werden dann die kopierten Referenzen übergeben.

Beispiel (Partielle Funktionsanwendung mit Referenzparameter). Wir vereinbaren eine Funktion `inc` mit einem Referenzparameter. Im Hauptprogramm rufen wir die Funktion `bind` mit insb. einem Parameter dem Rückgabewert eines Aufrufs der Funktion `ref` auf. Beim eventuellen Funktionsaufruf des Rückgabewertes der Funktion `bind` wird dann eine Referenz auf die Variable `i` an die unterliegende Funktion `inc` übergeben.

inc_demo.cpp

```cpp
#include <iostream>
#include <functional>

using namespace std;

void inc(int& n) { n++; }
int main() {
  int i = 0;
  function<void()>
    do_inc = bind(inc, ref(i));

  cout << i << endl;
  do_inc();
  cout << i << endl;
}
```

```
0
1
```

Funktionsobjekte für Methoden (`mem_fn`)

Die Standardlibrary stellt eine Funktion `mem_fn` zur Verfügung. Als Parameter erhält die Funktion einen Zeiger auf die Methode einer Klasse. Für eine Klasse T und eine Methode m von T ist die Syntax einen Zeiger auf m zu konstruieren: `&`T`::`m.

Als Rückgabewert liefert `mem_fn` ein Funktionsobjekt von nicht näher spezifiertem Typ mit erstem Parameter eine Referenz (oder ein Zeiger) auf ein Objekt der Klasse T. Die restlichen Parametern des zurückgegebenen Funktionsobjektes sind identisch zu denen der Method m.

Wird der Funktionsaufrufsoperator auf einen Rückgabewert von `mem_fn` angewandt so wird die unterliegende Methode auf dem als ersten Parameter gegebene Objekt aufgerufen.

Beispiel (Sehnentrapezregel mit Methode). Wir rufen die Funktion `trapezoidal` auf auf den Rückgabewert eines Aufrufs der Funktion `mem_fn`. Die unterliegende Methode ist hierbei die Methode `eval` der Klasse `NormV`.

Durch einen geeigneten Aufruf der Funktion `bind` erzeugen wir ein Funktionsobjekt dass bei Auswertung des Funktionsaufrufsoperators die Methode `eval` auf das Objekt `stdNorm` anwendet.

trapezoidal_mem_fn.cpp

```cpp
#include <iostream>
#include <cmath>
#include <functional>

using namespace std;

class NormV {
  private:
```

```
    double mw, stdabw;

  public:
    NormV (double mw_ = 0, double stdabw_ = 1)
      : mw(mw_), stdabw(stdabw_) {}

    double eval(double x) const {
      return M_2_SQRTPI / (2 * M_SQRT2 * stdabw)
        * exp(-((x - mw) * (x - mw)) / (2*stdabw*stdabw));
    }
};

double trapezoidal(function<double(double)> g,
    int n, double a = 0, double b = 1) {
  const double h = (b - a) / n;
  double s = (g(a) + g(b))/2;
  for (int i = 1; i <= n - 1; i++)
    s += g(a + i*h);
  return h * s;
}

int main() {
  using namespace placeholders;
  NormV stdNorm{};
  function<double(double)> stdNormEval =
    bind(mem_fn(&NormV::eval), ref(stdNorm), _1);

  for (int n = 1; n <= 512; n *= 4)
    cout << trapezoidal(stdNormEval, n) << endl;

  return 0;
}
```

```
0.320457
0.340082
0.341266
0.34134
0.341344
```

Lambda-Ausdrücke

In Form von Lambda-Ausdrücken stellt C++ spezielle Syntax zur Verfügung um Funktionsobjekte in strikt mächtigerer Weise zu erzeugen als die beschriebenen Werkzeuge.

⟨Lambda⟩ → “[” [⟨Capture⟩ {“,” ⟨Capture⟩}] “]” “(” ⟨Params⟩ “)” “->” ⟨Type⟩ “{” ⟨Body⟩ “}”
⟨Capture⟩ → [“&”] ⟨Variable⟩
| “this”

Syntaktisch besteht ein Lambda-Ausdruck aus einer Spezifikation der sog. *captures*, gefolgt von einer Parameterliste, dem Rückgabetyp und dann einem Körper, analog zu einer gewöhnlichen Funktion.

Der Wert des Lambda-Ausdrucks ist eine sog. *closure*. Ein Funktionsobjekt analog zu denen die z.B. die Funktion `bind` zurückliefert.

Wird der Name einer Variable als capture erwähnt, so wird (ähnlich wie bei einem Aufruf von `bind`) standardmäßig eine Kopie des Wertes dieser Variable angelegt. Wir sagen der Lambda-Ausdruck *überträgt* die erwähnte Variable in seinen Körper. Die Variable steht dann (zusätzlich qualifiziert als `const`) innerhalb des Körpers des Lambdas zur Verfügung. Wird dem Namen der Variable im zugehörigen capture ein `&` vorne angestellt, so wird statt einer Kopie eine Referenz angelegt. Wird statt eines Variablennamens das Schlüsselwort `this` angegeben (was nur zulässig ist wenn der Lambda-Ausdruck innerhalb einer Methode steht) so steht innerhalb des Körpers der Methode der implizite Parameter `this` zur Verfügung und zeigt auf das selbe Objekt wie `this` an der Stelle an der der Ausdruck selbst steht.

Die Parameterliste spezifiziert welche Parameter und von welchem Typ die closure bei ihrem eventuellen Aufruf akzeptiert.

Wird der Funktionsaufrufsoperator auf eine closure angewandt so wird der Körper des Lambda-Ausdrucks, mit dem die closure erstellt wurde, ausgeführt. Es stehen dabei die Parameter aus der Parameterliste zur Verfügung. Der Körper des Lambda-Ausdrucks kann, genauso wie eine Funktion, `return`-Anweisungen enthalten. Die an die `return`-Anweisungen übergebenen Werte müssen implizit konvertierbar sein in den Typ der als Rückgabetyp des Lambda-Ausdrucks spezifiziert wurde. Wurde statt einem Rückgabetyp `void` angegeben, so akzeptieren die `return`-Anweisungen innerhalb des Körpers des Lambda-Ausdrucks keinen Rückgabewert.

Beispiel (Sehnentrapezregel mit Lambda-Ausdrücken). Wir rufen die Funktion `trapezoidal` auf eine mit einem Lambda-Ausdruck erzeugte closure.

Der zweite Lambda-Ausdruck übertragt das Funktionsobjekt `f` in Referenz in seinen Körper; würde nach Erstellen des Funktionsobjekts `g` der in `f` gespeicherte Wert verändert, so hätte dies Auswirkungen auf den eventuellen Aufruf von `g`. Der in der Variable `off` gespeicherte Wert wird in Kopie in den Körper des zweiten Lambda-Ausdrucks übertragen. Würde der Wert von `off` nach Erstellung von `g` verändert, so hätte dies keine Auswirkung beim eventuellen Aufruf von `g`.

trapezoidal_lambda.cpp

```cpp
#include <iostream>
#include <cmath>
#include <functional>

using namespace std;

double trapezoidal(function<double(double)> g,
    int n, double a = 0, double b = 1) {
  const double h = (b - a) / n;
  double s = (g(a) + g(b))/2;
  for (int i = 1; i <= n - 1; i++)
    s += g(a + i*h);
  return h * s;
}

int main() {
  function<double(double)> f
    = [](double x) -> double { return exp(-x * x); };

  double off = -0.25;
```

```
  function<double(double)> g
    = [&f, off](double x) -> double { return f(x + off); };

  for (int n = 1; n <= 512; n *= 2)
    cout << trapezoidal(g, n) << endl;

  return 0;
}
```

```
0.754598
0.847006
0.868203
0.873407
0.874702
0.875025
0.875106
0.875126
0.875131
0.875133
```

Beispiel. Wir übertragen eine Variable `i` in Referenz in den Körper eines Lambda-Ausdrucks. Beim eventuellen Aufrufs der closure, die der Lambda-Ausdruck erzeugt, wirken sich Änderungen an `i` im Körper des Lambda-Ausdrucks auch außerhalb des Lambda-Ausdrucks aus.

inc_lambda_demo.cpp

```cpp
#include <iostream>
#include <functional>

using namespace std;

int main() {
  int i = 0;
  function<void()>
    do_inc = [&i]() -> void { i++; };

  cout << i << endl;
  do_inc();
  cout << i << endl;
}
```

```
0
1
```

Beispiel: Auswertung arithmetischer Ausdrücke mit Funktionen

Wir erzeugen zu Programmstart eine endliche Abbildung `funs` die Namen von mathematischen Funktionen assoziiert mit Funktionsobjekten zum Typ `double(double)`. Die Verwendung von `static_cast` ist hier notwending da die arithmetischen Funktionen `exp`, `log`, etc. in `<cmath>` in hochgradig überladener Form imple-

mentiert sind sodass sonst die jeweilige implizite Konvertierung zu einem Objekt der Klasse `function<...>` nicht eindeutig wäre.

Im Hauptprogramm lesen wir in einer Schleife immer wieder den Namen einer arithmetischen Funktion und einen Wert vom Typ `double` ein. Wir wenden dann jeweils die entsprechende Funktion aus der endlichen Abbildung `funs` an und behandeln den Fall, dass keine Funktion von diesem Namen in `funs` hinterlegt wurde, als exception.

fun_eval.cpp

```cpp
#include <iostream>
#include <map>
#include <functional>
#include <cmath>

using namespace std;

using ArithFun = double(double);

const map<string, function<ArithFun>> funs{
  {"exp", static_cast<ArithFun*>(exp)},
  {"ln", static_cast<ArithFun*>(log)},
  {"log10", [](double x) -> double { return log(x)/log(10); }},
  {"cos", static_cast<ArithFun*>(cos)},
  {"sin", static_cast<ArithFun*>(sin)},
  {"arccos", static_cast<ArithFun*>(acos)},
  {"arcsin", static_cast<ArithFun*>(asin)}
};

int main() {
  while (true) {
    string fun;
    double val;
    cout << "> ";
    if (!(cin >> fun >> val))
      break;

    try {
      cout << (funs.at(fun))(val) << endl;
    } catch (const out_of_range&) {
      cout << "Unknown function: " << fun << endl;
    }
  }
}
```

```
> exp 1
2.71828
> log10 100
2
> arccos -1
3.14159
> sinh 0.25
Unknown function: sinh
>
```