Funktionstypen und Funktionszeiger

⟨Fun'typ⟩ ⟶ ⟨Rückgabetyp⟩ "(" ⟨Param'liste⟩ ")"

- Zeiger von Funktionstypen möglich
- Auto. Umwandlung Funktionsname ⟶ Zeiger auf Funktion
- Funktionsauswertungsoperator für Zeiger auf Funktion definiert

Funktionstypen

```
using ArithFun = double(double);

int main() {
  ArithFun* funptr = sqrt;

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

```
1.41421
```

Vereinbarungssyntax für Zeiger auf Funktion

- Statt „Umweg“ mit Typalias für Funktionstyp
- Historisch üblicher (auch `typedef`)
- Nicht zu empfehlen

Funktionszeigervereinbarung

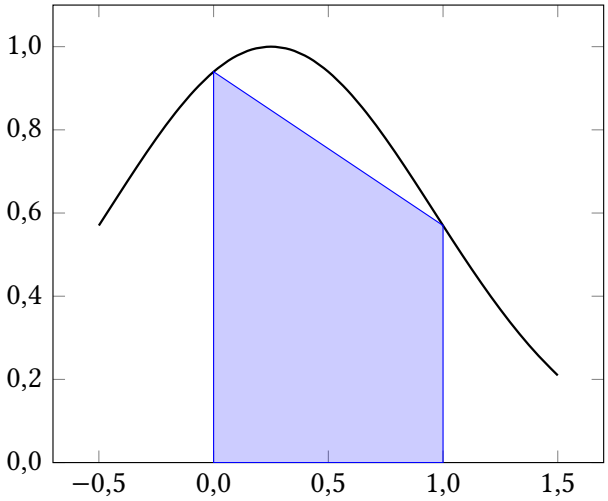
```
int main() {
  double (*funptr)(double) = sqrt;

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

```
1.41421
```

Beispiel: Sehnentrapezregel

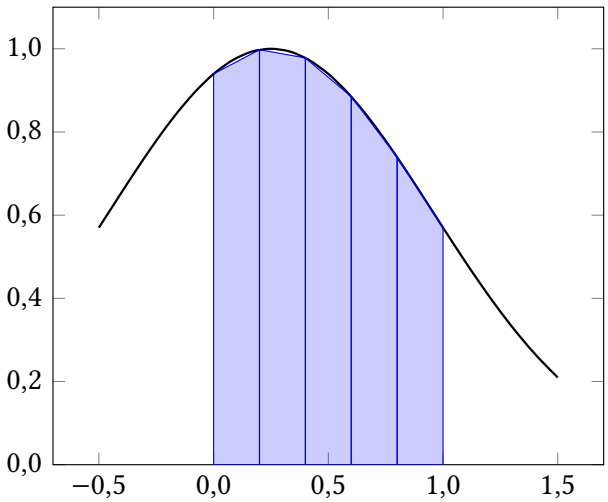
$$\int_a^b f(x)\ \mathrm{d}x \approx (b-a)\frac{f(a)+f(b)}{2}$$



Beispiel: Sehnentrapezregel

$$\begin{aligned}\int_a^b f(x)\ \mathrm{d}x &\approx (b-a)\frac{f(a)+f(b)}{2}\\ &\approx \sum_{i=0}^{n-1}(x_{i+1}-x_i)\frac{f(x_i)+f(x_{i+1})}{2}\\ &= h\left(\frac{f(a)+f(b)}{2}+\sum_{i=1}^{n-1}f(x_i)\right)\end{aligned}$$

mit $h=\frac{b-a}{n}$, $x_i = a+ih$



Beispiel: Sehnentrapezregel (Funktionszeiger)

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
```

Beispiel: Sehnentrapezregel (Typsynonym)

trapezoidal_ftyp.cpp

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

using namespace std;

using ArithFun = double(double);

double trapezoidal(ArithFun 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 *= 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 STL (`function`)

- Motivation: Funktionsparameter nicht nur Zeiger auf Funktion sondern *beliebige Funktionsobjekte*
- STL stellt Klasse `function<`T`>` für T Funktionstyp (z.B. `double(double)`)
- Header `<functional>`
- `operator()` passend überladen; `function<`T`>` ist selbst Funktionsobjekt
- Stellt Konstruktor mit einem Parameter für beliebiges Funktionsobjekt mit passendem `operator()` (auch Zeiger auf Funktion) ⟶ implizite Typkonvertierung
- Deswegen als Funktionsparameter immer lieber `function<`T`>` statt Zeiger auf Funktion

Beispiel: Sehnentrapezregel (`function`)

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;
}
```

```cpp
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`)

- Oft nützlich: manche Parameter einer Funktion vorgeben/vertauschen/…
- STL stellt hierfür Funktion `bind`
- Funktion `bind` nimmt Funktionsobjekt und beliebig viele Argumente, kopiert gegebene Parameter in zurückgegebenen Wert
- Spezieller Namensraum von Platzhaltern für Argumente des Funktionsaufrufs `std::placeholders`: `_1`, `_2`, …
- `operator()` auf Wert erzeugt von `bind` übergibt Parameter aus Wert und aus Funktionsaufruf

Vertausche

```
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 (`bind`)

trapezoidal_bind.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
```

Partielle Funktionsanwendung (`bind`) – Referenzen

Funktion `ref` (oder `cref` für `const`) zur Erzeugung von wrapper-Objekten für Referenzen als gebundene Argumente von `bind`

Inkrementiere

```
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`)

Für T eine Klasse und m eine Methode von T:
`mem_fn(&`T`::`m`)` liefert Funktionsobjekt mit erstem Parameter Referenz (oder Zeiger) auf Objekt von T und restlichen Parametern wie m.

Beispiel: Sehnentrapezregel (`mem_fn`)

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;
}
```

```cpp
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

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

- Erstellt eine *closure*, captures (*gebundene Variablen*) werden kopiert
- Typ von Lambda-Ausdrücken *nicht ausdrückbar*
  - entweder konvertieren, oder
  - `auto`

```cpp
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); };
```

Beispiel: Sehnentrapezregel (Lambda-Ausdrücke)

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
```

Beispiele: Datenstrukturen von Funktionsobjekten

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
>
```