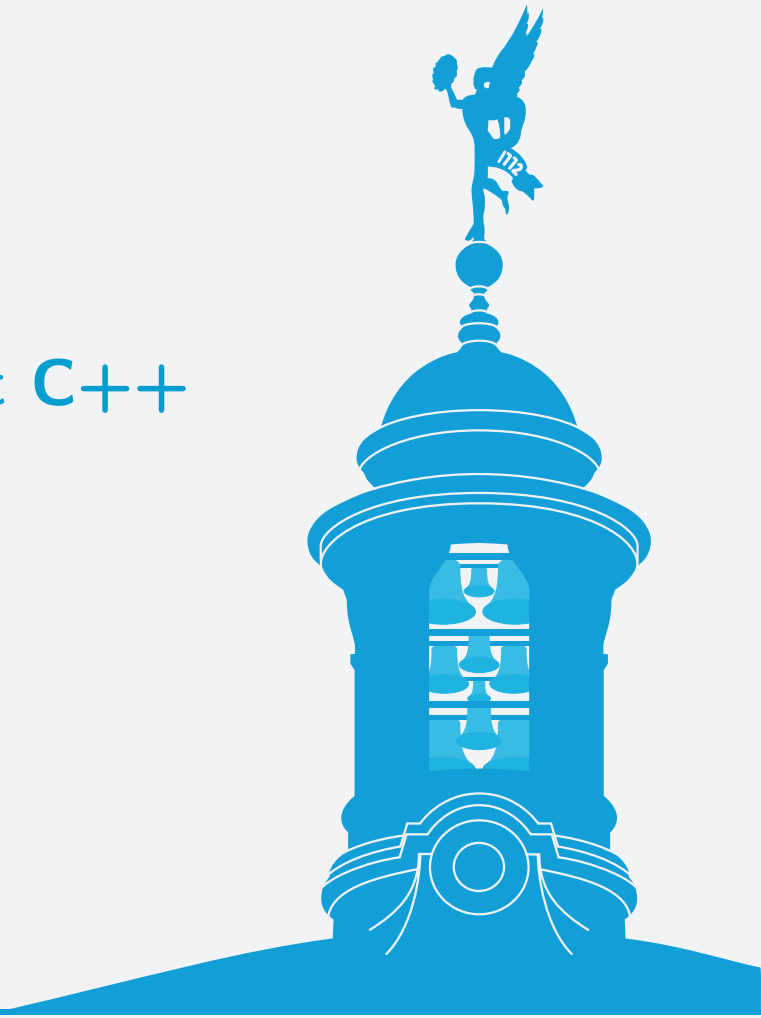
WWU
MÜNSTER

Einführung in die Programmierung mit C++

Tobias Leibner

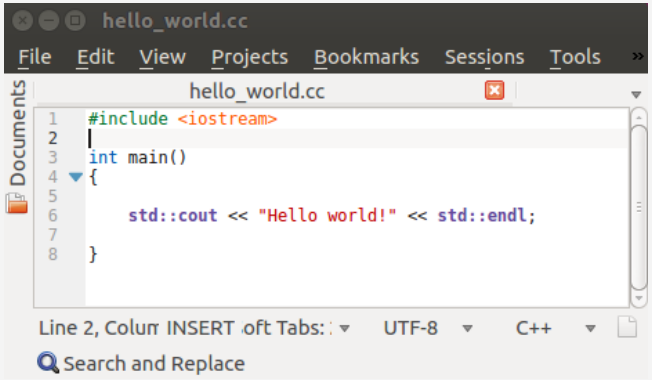


living.knowledge

outline

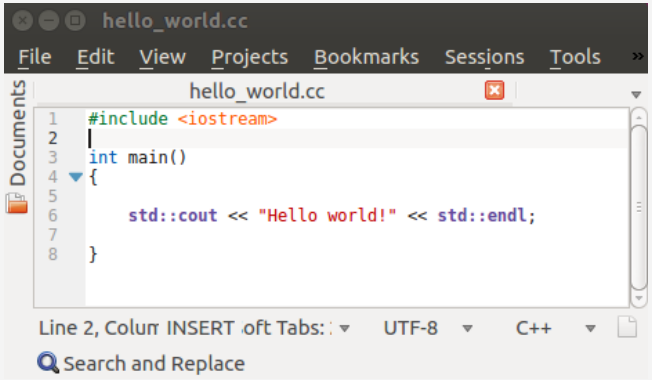
quick start: Arbeitsablauf

Textdatei (ASCII)



quick start: Arbeitsablauf

Textdatei (ASCII)

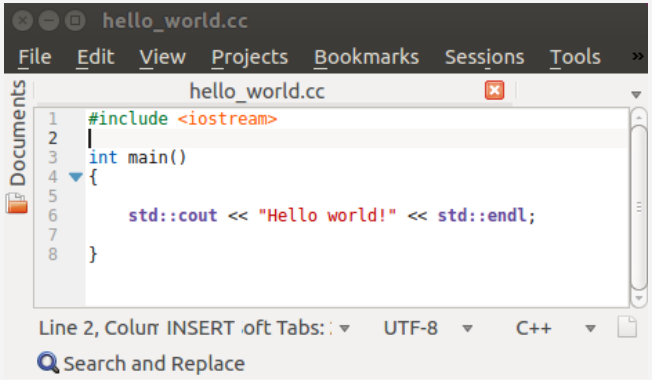


Binärdatei

```
(falbr_01@ADLER36) [14:13:33|Thu Sep 22] [~/Praktikum]
-----> ll
total 16K
-rwx------ 1 falbr_01 o0num 9.1K 22.09.2016 14:13 hello_world
-rw------- 1 falbr_01 o0num   83 22.09.2016 13:13 hello_world.cc
(falbr_01@ADLER36) [14:13:34|Thu Sep 22] [~/Praktikum]
-----> ./hello_world
Hello world!
```

quick start: Arbeitsablauf

Textdatei (ASCII)



Compiler (g++)

```
(falbr_01@ADLER36) [14:13:26|Thu Sep 22] [~/Praktikum]
—> g++ -Wall -o hello_world hello_world.cc
```

Binärdatei

```
(falbr_01@ADLER36) [14:13:33|Thu Sep 22] [~/Praktikum]
—> ll
total 16K
-rwx------ 1 falbr_01 o0num 9.1K 22.09.2016 14:13 hello_world
-rw------- 1 falbr_01 o0num   83 22.09.2016 13:13 hello_world.cc
(falbr_01@ADLER36) [14:13:34|Thu Sep 22] [~/Praktikum]
—> ./hello_world
Hello world!
```

quick start: Hello world!

- Ordner und Datei mit Hilfe des Terminals erstellen:

```
mkdir C++-Praktikum
cd C++-Praktikum
mkdir day_1
cd day_1
touch hello_world.cc
```

quick start: Hello world!

- Ordner und Datei mit Hilfe des Terminals erstellen:

```
mkdir C++-Praktikum
cd C++-Praktikum
mkdir day_1
cd day_1
touch hello_world.cc
```

- `hello_world.cc` mit einem Editor (zB `kate`) editieren:

```
#include <iostream>

int main()
{
  std::cout << "Hello world!" << std::endl;

  return 0;
}
```

quick start: Hello world!

- Ordner und Datei mit Hilfe des Terminals erstellen:

```
mkdir C++-Praktikum
cd C++-Praktikum
mkdir day_1
cd day_1
touch hello_world.cc
```

- `hello_world.cc` mit einem Editor (zB `kate`) editieren:

```
#include <iostream>

int main()
{
  std::cout << "Hello world!" << std::endl;

  return 0;
}
```

- Den Quellcode mit einem Compiler in Maschinencode übersetzen (kompilieren):

```
g++ -std=c++11 -Wall -o hello_world hello_world.cc
```

quick start: Hello world!

- Ordner und Datei mit Hilfe des Terminals erstellen:

```
mkdir C++-Praktikum
cd C++-Praktikum
mkdir day_1
cd day_1
touch hello_world.cc
```

- `hello_world.cc` mit einem Editor (zB `kate`) editieren:

```
#include <iostream>

int main()
{
  std::cout << "Hello world!" << std::endl;

  return 0;
}
```

- Den Quellcode mit einem Compiler in Maschinencode übersetzen (kompilieren):

```
g++ -std=c++11 -Wall -o hello_world hello_world.cc
```

- Das Programm ausführen:

```
./hello_world
```

Hello world!

hello_world.cc

```cpp
#include <iostream> // <- iostream.hh contains std::cout, std::endl

int main()
{
  std::cout << "Hello world!" << std::endl;                        // std::endl means new line

  std::cout << "\n"                                                // \n also
            << "The first letter of my name is " << 'T' << ",\n"
            << "I am " << 29 << " years old\n"                     // std::cout can print
            << "and pi=" << 3.1415926 << "!" << std::endl;         // many things

  return 0;
}
```

Hello world!

hello_world.cc

```cpp
#include <iostream>

int main()
{
  char letter = 'T';
  int age = 29;
  double pi = 3.1415926;

  std::cout << "Hello world!" << std::endl;

  std::cout << "\n"
            << "The first letter of my name is " << letter << ",\n"
            << "I am " << age << " years old\n"
            << "and pi=" << pi << "!" << std::endl;
  return 0;
}
```

Hello world!

hello_world.cc

```cpp
#include <iostream>

int main()
{
  char name[] = "Tobias";
  int age = 29;
  double pi = 3.1415926;

  std::cout << "Hello world!" << std::endl;

  std::cout << "\n"
            << "The first letter of my name is " << name[0] << ",\n"
            << "I am " << age << " years old\n"
            << "and pi=" << pi << "!" << std::endl;
  return 0;
}
```

intermezzo: arrays and strings

arrays_and_strings.cc

```cpp
#include <iostream>

int main()
{
  // declare and initialize (define) arrays
  char word[] = "foo";
  int array_of_ints[] = {-2, 1, 4, 5};

  // declare array
  int vec[3];
  // initialize later
  vec[0] = -3;
  vec[1] = 2;
  vec[2] = 1;

  // print some information
  std::cout << "vec[0]: " << vec[0] << std::endl;
  std::cout << "vec[1]: " << vec[1] << std::endl;
  std::cout << "vec[2]: " << vec[2] << std::endl;

  return 0;
}
```

intermezzo: arrays and strings

arrays_and_strings.cc

```cpp
#include <iostream>

int main()
{
  // declare and initialize (define) arrays
  char word[] = "foo";
  int array_of_ints[] = {-2, 1, 4, 5};

  // ...
  word = "bar"; // <- does not compile
}
```

intermezzo: arrays and strings

arrays_and_strings.cc

```cpp
#include <iostream>
#include <string>   // contains std::string

int main()
{
  // use std::string for words
  std::string word = "foo";

  // ...
  word = "bar";                    // <- works now
  word = " really long sentence"; // <- can handle strings of "arbitrary" length
}
```

intermezzo: arrays and strings

arrays_and_strings.cc

```cpp
#include <iostream>

int main()
{
  // ...
  int vec[3];
  vec[0] = -3;
  vec[1] = 2;
  vec[2] = 1;

  // print some information
  std::cout << "vec[-1]: " << vec[-1] << std::endl; // <- what happens?
}
```

Hello world!

```cpp
#include <iostream> // <- for std::cout, std::endl
#include <string>   // <- for std::string

int main()
{
  std::string name = "Tobias";
  int age = 29;
  double pi = 3.1415926;

  std::cout << "Hello world!" << std::endl;

  std::cout << "\n"
            << "The first letter of my name is " << name[0] << ",\n" // <- std::string can be
            << "I am " << age << " years old\n"                      //    indexed like an
            << "and pi=" << pi << "!" << std::endl;                  //    array

  return 0;
}
```

Zusammenfassung

- Quellcode $\Rightarrow$ Compiler $\Rightarrow$ Maschinencode
- Variablen:
  `type name = value;`
- Arrays fester Größe:
  `type name[size] = {a, b, c, ...};`
- Indizierung eines arrays der Länge `size` von 0 bis `size`-1:
  `name[0] = ...;`
- Ausgabe:
  `std::cout << "word" << 3 << ... ;` (`#include <iostream>`)
- Strings (beliebiger Größe):
  `std::string phrase = "This is a sentence";` (`#include <string>`)
- Statements **müssen** mit `;` abgeschlossen werden!

intermezzo: Formatierung

```
#include <iostream>
int main() { char letter =
'T'; std::cout << letter <<

  std::endl; int
             age = 29; std::cout <<
             age << std::endl;
double pi = 3.1415926; std::cout
<< pi << std::endl;return 0;}
```

intermezzo: Formatierung

```
#include <iostream>
int main() { char letter =
'T'; std::cout << letter <<

  std::endl; int
             age = 29; std::cout <<
             age << std::endl;
double pi = 3.1415926; std::cout
<< pi << std::endl;return 0;}
```

```
#include <iostream>
int main()
{
  char letter = 'T';
  int age = 29;
  double pi = 3.1415926;

  std::cout << letter << std::endl;
  std::cout << age << std::endl;
  std::cout << pi << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()
{
  int a, b;

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()                          // try the following: a = 2 000 000 000
{                                   //                    b =   200 000 000
  int a, b;

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()                          // try the following: a = 2 000 000 000
{                                   //                    b =   200 000 000
  unsigned int a, b; // <-

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()                          // try the following: a = 2 000 000 000
{                                   //                    b =   200 000 000
  long int a, b; // <-

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()                          // try the following: a = 0
{                                   //                    b = 0
  int a, b; // <-

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()                          // try the following: a = 0
{                                   //                    b = 0
  double a, b; // <-

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Datentypen und interaktiver input

interactive_input.cc

```
#include <iostream>

int main()                          // try the following: a = 1.2
{                                   //                    b = 0
  double a, b; // <-

  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "\n"
            << "a + b = " << a + b << std::endl;
  std::cout << "a - b = " << a - b << std::endl;
  std::cout << "  a*b = " <<   a*b << std::endl;
  std::cout << "  a/b = " <<   a/b << std::endl;

  return 0;
}
```

Zusammenfassung

In C++ gibt es grundverschiedene Typen,

- für ganze Zahlen $\in \mathbb{Z}$: `int` ($\pm 2.14 \cdot 10^9$), `long int`,
- für ganze nichtnegative Zahlen $\in \mathbb{N} \cup 0$: `unsigned int` ($0 \ldots 4.29 \cdot 10^9$),
- für rationale Zahlen $\in \mathbb{R}$: `double` ($\pm 1.7 \cdot 10^{308}$, genau bis auf 15 Stellen),

```
double one      = 1.0;
double also_one = 1.000000000000000001;
```

und Operatoren (z.B. +, −, ∗, /), um diese zu kombinieren.

$\Rightarrow$ Deren Ergebnis hängt vom Typ der Argumente ab!

- `1/2`: 0 `int/int` = `int`
- `1.0/2.0`: 0.5 `double/double` = `double`
- `1.0/2`: 0.5 `double/int` = `double`
- Zuweisungen können den Typ ändern:

```
int number = 3.0/2.0; // number = 1
```

Kommandozeilenargumente

gesucht: ein Programm “`./divide 5 2.0`”
mit Resultat “`5 divided by 2 is 2.5`”

Kommandozeilenargumente

gesucht: ein Programm "`./divide 5 2.0`"
mit Resultat "`5 divided by 2 is 2.5`"

`divide.cc`

```
#include <iostream>

int main(int argc, char* argv[])
{
  std::cout << "The name of the program is " << argv[0]
            << " and it was called with " << argc << " arguments." << std::endl;

  return 0;
}
```

- `argc`: Anzahl der Kommandozeilenargumente (inkl. Programmaufruf)
- `argv`: array der Kommandozeilenargumente (mit Länge `argc`)

Kommandozeilenargumente

divide.cc

```cpp
#include <iostream>

int main(int argc, char* argv[])
{
  if (argc == 2) {                      // ./divide 5
    std::cout << "Please provide both numbers!" << std::endl;
    return 1;
  } else if (argc != 3) {               // ./divide (or ./divide 5 2.5 too many)
    std::cout << "Call " << argv[0] << " with two numbers!" << std::endl;
    return 1; // <- ends the program
  }

  std::cout << "The name of the program is " << argv[0] << "\n"
            << "and it was called with " << argc << " arguments." << std::endl;

  return 0;
}
```

Kommandozeilenargumente

divide.cc

```cpp
#include <iostream>
#include <cstdlib> // <- for atoi atof conversion

int main(int argc, char* argv[])
{
  if (argc == 2) {                    // ./divide 5
    std::cout << "Please provide both numbers!" << std::endl;
    return 1;
  } else if (argc != 3) {             // ./divide (or ./divide 5 2.5 too many)
    std::cout << "Call " << argv[0] << " with two numbers!" << std::endl;
    return 1; // <- ends the program
  }
  // from here on we can be sure that argc == 3!

  int nominator = std::atoi(argv[1]);
  double denominator = std::atof(argv[2]);

  // print out result ...

  return 0;
}
```

logische Operationen

```
bool something = true;
bool another = false;
// most things convert to bool
bool this_is_false = 0;
bool this_is_true = 13; // anything not 0 is true
// bool converts to int
int something_int = something; // 1
int another_int   = another;   // 0
```

boolsche operatoren:

```
int a = 10;
int b = -2;

bool result1 =       (a  < b); // false
bool result2 =       (a >= b); // true
bool result3 =       (a == b); // false
bool result4 =       (a != b); // true
bool result5 =      !(a != b); // false
bool result6 = (!result5 > a); // ?
```

Verknüpfungen mit && und ||:

```
double a = 1;
double b = 1.0 + 1e-14;

bool different =  (a < b) ||  (a > b);
bool similar   = !(a < b) && !(a > b);
```

- &&: nur `true`, falls beide Argumente `true` sind
- ||: `true`, falls mindestens ein Argument `true` ist

if/else

```
if (/*condition is true*/) {
  // this happens
}

if (/*condition is true*/) {
  // this happens
} else {
  // that happens
}


if (/*condition*/) {
  // this
} else if (/*condition*/) {
  // that
} else if (/*condition*/) {
  // or that
} else {
  // finally
}
```

```
if (argc != 3) {
  std::cout << "Please provide...";
}

if (a < 0) {
  std::cout << "a is negative" << std::endl;
} else {
  int b = 5;
  double result = a/b;
}

if (a == 0) {
  std::string = "a word or two";
} else if (argc == 1) {
  std::cout << "Error!" << std::endl;
  return 1;
} else if (1 > 2) {
  a = 5;
} else if (1 < 2) {
  a = 6
} else {
  std::cout << "We should never get here!";
}
```

intermezzo: Lebensdauer von Variablen

Aber Vorsicht:

scope.cc

```
#include <iostream>

int main()
{
  if (1 != 2) {
    double number = 3.0;
  } else {
    double number = 5.0;
  }

  std::cout << "The number is " << number << std::endl;

  return 0;
}
```

intermezzo: Lebensdauer von Variablen

scope.cc

```cpp
#include <iostream>

int main()
{
  double number;          // declare number beforehand
  if (1 != 2) {
    number = 3.0;         // <- this changes number from above
  } else {
    double number = 5.0; // <- this defines a new variable with the same name
  }

  std::cout << "The number is " << number << std::endl;

  return 0;
}
```

intermezzo: Lebensdauer von Variablen

- Variablen gelten nur in dem Block, in dem sie deklariert werden!

```
int main()
{
  int a = 0;    // a lives from here

  if (a > 1) {
    int b = 0; //   b lives from here
    // ...
  }            //   b is destroyed here

}              // a is destroyed here
```

intermezzo: Lebensdauer von Variablen

- Variablen gelten nur in dem Block, in dem sie deklariert werden!

```
int main()
{
  int a = 0;    // a lives from here

  if (a > 1) {
    int b = 0; //   b lives from here
    // ...
  }            //   b is destroyed here

}              // a is destroyed here
```

- Variablen im gleichen Block haben Vorrang!

```
int main()
{
  double number = 1.0                  // number_outer

  if (1 != 2) {
    double number = 5.0;               // number_inner
    std::cout << number << std::endl   // 5.0
  }
}
```

die `while` Schleife

while.cc

```
#include <iostream>

int main()
{
  double number;
  std::cout << "Please enter a positive number: ";
  std::cin >> number;

  while (number < 0.0) {
    std::cout << "The number " << number << " is not positive, try again: ";
    std::cin >> number;
  }

  std::cout << "The number " << number << " is positive, yeah!" << std::endl;

  return 0;
}
```

die `while` Schleife

Prinzip:

```
while (/*this condition is true*/) {
  // execute this code
  // ...
}
```

Aber Vorsicht:

```
int i = 1;

while (i > 0) {
  std::cout << i << std::endl;
  i += 2;                       // <- the same as i = i + 2;
}
```

$\Rightarrow$ Programmabbruch mit `Strg + C`

die `for` Schleife

gesucht: Programm, welches eine Summe

$$\sum_{i=0}^{I-1} \frac{i}{2}$$

für beliebige $I \in \mathbb{N}$ berechnet.

sum.cc

```
#include <iostream>

int main()
{
  int I = 50;
  double result = 0.0;

  for (int i = 0; i < I; ++i) {
    result += i/2.0;
  }

  std::cout << result << std::endl;

  return 0;
}
```

die `for` Schleife

Prinzip:

```
for (1_init; 2_check_condition; 4_count) {
  3_execute;
}
```

Aber Vorsicht:

```
for (double a = 100.0; true; a = 50) {
  std::cout << "loops are dangerous" << std::endl;
}
```

(Fast) beliebige Anweisungen möglich:

```
for (int counter = 0;
     !(counter > 5) < 1;
     counter *= 5/3.0 - 23*67) {
  result += 1.0;
}
```

input/output

file_io.cc

```
#include <iostream>
#include <fstream>

int main()
{
  std::ifstream input("hello_world.cc");
  std::string line;

  while (std::getline(input, line)) {
    std::cout << line;
  }

  return 0;
}
```

input/output

file_io.cc

```
#include <iostream>
#include <fstream>

int main()
{
  std::ifstream input("hello_world.cc");
  std::ofstream output("data.txt");
  std::string line;

  while (std::getline(input, line)) {
    output << line << "\n";
  }

  return 0;
}
```

Check with: `diff hello_world.cc data.txt`

input/output

Streams verhalten sich gleich, egal ob dahinter ein Terminal oder eine Datei steckt.

- Output mit Operator <<:

```
std::cout << "word " << 3 << std::endl;   // #include <iostream>

std::ofstream output_file("tmp.txt");     // #include <fstream>
output_file << "word " << 3 << std::endl;
```

- Input mit Operator >>:

```
std::string str;                    // #include <string>
int n;

std::cin >> str >> n;               // #include <iostream>

std::ifstream input_file("tmp.txt"); // #include <fstream>
input_file >> str >> n;
```

fundamentale Typen und die Standard Bibliothek

Fundamentale Typen sind immer bekannt:

- Buchstaben: `char letter = 'A';`
- ganze Zahlen: `int n = -1; long int m = 20000000000;`
- positive Zahlen: `unsigned int = 0; long unsigned int = 20000000000;`
- reelle Zahlen: `double x = -1.3; long double xtra_large = 1e310;`

Zusätzliche Typen und Funktionalität aus der Standardbibliothek:

- `#include <iostream>`
  Terminal input (`std::cin`) und output (`std::cout`)
- `#include <fstream>`
  Dateien lesen (`std::ifstream`) und schreiben (`std::ofstream`)
- `#include <string>`
  Zeichenketten: `std::string phrase = "This here.";`
  Zeile einlesen: `std::getline(...)`

Visualisierung mit gnuplot

visualize.cc

```cpp
#include <fstream>

int main()
{
  std::ofstream output("data.txt");

  for (double x = 0; x < 5; x += 0.75) {
    output << x << "\t" << x*x << std::endl;
  }
}
```

- gnuplot im Terminal starten
- Daten visualisieren: `plot "data.txt"`
- Daten als glatte Funktion visualisieren: `plot "data.txt" smooth cspline`
- gnuplot beenden: `quit`

Debugging mit gdb

Was macht das folgenden Programm?

error.cc

```
int main()
{
  int a = 1;
  int b = 0;
  double plus = a + b;
  double div = a/b;
}
```

Debugging mit gdb

- Programm mit debugging Optionen **-g** kompilieren:
  `g++ -g -std=c++11 -Wall -o error error.cc`
- Programm mit gdb starten:
  `gdb ./error`
- `run` (r)
- `quit`

Zusammenfassung Tag 1

- fundamentale Typen: `char`, `int`, `long int`, `unsigned int`, `double`, `bool`
- arrays: `int vec[2];`
- `std::string phrase = "arbitrarily long"` `#include <string>`
- Input/Output (Terminal): `std::cin`, `std::cout` `#include <iostream>`
- Input/Output (Datei): `std::ifstream`, `std::ofstream` `#include <fstream>`
- `if`/`else`, `while`, `for`
- lokale Variablen gelten nur innerhalb ihres Blocks (`{ ...}`)
- Vorsicht: Informationsverlust durch Konvertierung: `int pi = 3.1415; // 3`
- Vorsicht: Informationsverlust durch Ganzzahldivision: `double o_five = 1/2; // 0.0`

noch einmal: Debugging mit gdb

Was macht das folgenden Programm?

error.cc

```cpp
#include <iostream>

int main()
{
  int result = 10;
  for (int i = 0; i < 9; ++i) {
    result *= 10;
  }
  std::cout << result << std::endl;
}
```

noch einmal: Debugging mit gdb

- Programm mit debugging Optionen -**g** kompilieren:
  `g++ -g -Wall -o error error.cc`
- Programm mit gdb starten:
  `gdb ./error`
- `break 7  (b 7)`
- `run  (r)`
- `display i`
- `delete 1 // removes breakpoint number 1`
- `watch result`
- `continue  (c)`
- `quit`

outline

Tag 1: Grundlagen

Tag 2: funktionale Programmierung

Tag 3: objektorientierte Programmierung (Teil 1)

Tag 4: objektorientierte Programmierung (Teil 2)

Tag 5: Was C++ noch alles kann

Tag 2: funktionale Programmierung

```
cd C++-Praktikum
mkdir day_2
cd day_2
```

Funktionen

Wo wir schon Funktionen benutzt haben:

- ```
  int main()
  {
    // ...
  }
  ```
- `std::getline(input, line)` (`#include <string>`)
- `std::atoi(argv[1])` (`#include <cstdlib>`)
- `std::abs(x)`, `std::log(x)`, `std::sin(x)` (`#include <cmath>`)

Warum Funktionen hilfreich sind:

- Um wiederverwendbaren Code nur einmal zu schreiben.

```
double x_n       = power(x, n);
double minus_1_n = power(-1, n);
```

- Um ein Programm zu strukturieren (`main` enthält nur grobe Struktur).

```
double pi_n  = approximate_pi(n);
double pi_2n = approximate_pi(2*n);

double eoc = calculate_eoc(pi_n, pi_2n, n, 2*n);
```
```

Funktionen

functions.cc

```cpp
#include <iostream>

int main()
{
  int a = 1;
  int b = 5;

  std::cout << "The sum of " << a << " and " << b << " is " << a + b << std::endl;

  return 0;
}
```

Funktionen

functions.cc

```cpp
#include <iostream>

int add(int x, int y)
{
  return x + y;
}


int main()
{
  int a = 1;
  int b = 5;

  std::cout << "The sum of " << a << " and " << b << " is " << add(a, b) << std::endl;

  return 0;
}
```

Funktionen

Funktionen mit Argumenten und Rückgabe:

```
ReturnType function_name(ArgumentType1 arg1,
                         ArgumentType2 arg2,
                        /* arbitrary number of arguments */)
{
  // this is the function body
  // ...

  return something_of_type_ReturnType;
}
```

Aber Vorsicht:

```
int half(double a)
{
  return a/2.0:
}

int main()
{
  double o_five = half(1.0);

  return 0;
}
```

Funktionen

functions.cc

```cpp
#include <iostream>

int add(int x, int y)
{
  return x + y;
}

void print_sum(int x, int y)
{
  std::cout << "The sum of " << x << " and " << y << " is " << add(x, y) << std::endl;
}


int main()
{
  int a = 1;
  int b = 5;

  print_sum(a, b);

  return 0;
}
```

Funktionen

Funktionen mit Argumenten *ohne* Rückgabe:

```
void function_name(ArgumentType1 arg1,
                   ArgumentType2 arg2,
                   /* arbitrary number of arguments */)
{
  // this is the function body
  // ...

  //return; // (not required)
}
```

Funktionen

functions.cc

```cpp
#include <iostream>

int add(int x, int y); // <- declaration of add

void print_sum(int x, int y)
{
  std::cout << "The sum of " << x << " and " << y << " is " << add(x, y) << std::endl;
}

int add(int x, int y) // <- definition of add
{
  return x + y;
}

int main()
{
  int a = 1;
  int b = 5;

  print_sum(a, b);

  return 0;
}
```

Funktionen

functions.cc

```
#include <iostream>

int add(int x, int y);

void print_sum(int x, int y);

int guess()
{
  static int counter = 0;
  if (counter == 0) {
    counter += 3;
    return 7;
  } else {
    counter -= 1;
    return counter*2;
  }
}

int main()
{
  guess();
  return 0;
}
```

Funktionen

Funktionen *ohne* Argumente und Rückgabe:

```
ReturnType function_name()
{
  // this is the function body
  // ...

  return something_of_type_ReturnType;
}
```

static

Variable die als **static** markiert werden werden nur beim ersten Aufruf initialisiert und behalten ihren Wert:

```
void stupid_function()
{
  static int counter = 1;
  std::cout << "I have been called " << counter << " times!" << std::endl;
  counter += 1;
}
```

Funktionen

functions.cc

```
#include <iostream>

int add(int x, int y);

void print_sum(int x, int y);

int guess();

void do_work()
{
  print_sum(guess(), guess());
}

int main()
{
  do_work();

  return 0;
}
```

Funktionen

Funktionen *ohne* Argumente und *ohne* Rückgabe:

```
void function_name()
{
  // this is the function body
  // ...
}
```

Funktionen mit default Argumenten

```cpp
#include <iostream>
#include <cmath> // <- for std::abs

bool similar(double a, double b, double tolerance = 1e-15)
{
  return std::abs(a - b) < tolerance;
}

int main()
{
  double a, b;
  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "These numbers are ";
  if (!similar(a, b)) {
    std::cout << "not ";
  }
  std::cout << "similar!" << std::endl;

  return 0;
}
```

Funktionen mit default Argumenten

```cpp
#include <iostream>
#include <cmath> // <- for std::abs

bool similar(double a, double b, double tolerance = 1e-15)
{
  return std::abs(a - b) < tolerance;
}

int main()
{
  double a, b;
  std::cout << "enter a: ";
  std::cin >> a;
  std::cout << "enter b: ";
  std::cin >> b;

  std::cout << "These numbers are ";
  if (!similar(a, b, 1e-3)) {          // <- tolerance = 1e-3
    std::cout << "not ";
  }
  std::cout << "similar!" << std::endl;

  return 0;
}
```

Funktionen

Funktionen mit default-Argumenten:

```
ReturnType function_name(ArgumentType1 arg1,
                         ArgumentType2 arg2 = default_arg2_of_type_ArgumentType2,
                        /* arbitrary number of arguments */)
{
  // this is the function body
  // ...

  return something_of_type_ReturnType;
}
```

Rekursion

gesucht: Ein Programm welches die Fakultät

$$n! = \prod_{k=1}^{n} k$$

einer positiven Zahl $k \in \mathbb{N}$ berechnet.

factorial.cc

```cpp
#include <iostream>

int factorial(int i)
{
  // ...
}

int main()
{
  // read number n ...
  // check for positiveness ...

  std::cout << "factorial(" << n << ") = " << factorial(n) << std::endl;

  return 0;
}
```

Funktionen in eigene Header auslagern

functions.hh

```
int factorial(int i)
{
  // ...
}
```

factorial.cc

```
#include <iostream>
#include "functions.hh" // <-

int main()
{
  // read number n ...
  // check for positiveness ...

  std::cout << "factorial(" << n << ") = " << factorial(n) << std::endl;

  return 0;
}
```

Funktionen in eigene Header auslagern

functions.hh

```
int factorial(int i)
{
  // ...
}
```

factorial.cc

```
#include <iostream>
#include "functions.hh"
#include "functions.hh" // <-

int main()
{
  // read number n ...
  // check for positiveness ...

  std::cout << "factorial(" << n << ") = " << factorial(n) << std::endl;

  return 0;
}
```

Funktionen in eigene Header auslagern

functions.hh

```
#ifndef FUNCTIONS_HH
#define FUNCTIONS_HH

int factorial(int i)
{
  // ...
}

#endif // FUNCTIONS_HH
```

factorial.cc

```
#include <iostream>
#include "functions.hh"

int main()
{
  // read number n ...
  // check for positiveness ...

  std::cout << "factorial(" << n << ") = " << factorial(n) << std::endl;

  return 0;
}
```

swap.cc

```
#include <iostream>

void swap(double a, double b)
{
  double tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  double a = 1.;
  double b = 10.;

  std::cout << "before:" << std::endl;
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;

  swap(a, b);

  std::cout << "\nafter:" << std::endl;
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;

  return 0;
}
```

lokale und globale Variablen

Variablen die innerhalb einer Funktion definiert werden

```
void swap(double a, double b) // a and b live from here
{
  double tmp = a;             // tmp lives from here
  // ...
}                             // all are destroyed here
```

sind nur innerhalb dieser Funktion bekannt (**lokale Variablen**).

Lokale Variablen haben Vorrang:

```
void swap(double a, double b) // swap_a, swap_b
{
  double tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  double a = 1.;  // main_a
  double b = 10.; // main_b

  swap(a, b);
```

Referenzen

reference.cc

```cpp
#include <iostream>

int main()
{
  int number = 1;
  int& view_on_number = number; // can be used like an int

  std::cout << "number = " << number << "\n"
            << "view_on_number = " << view_on_number << std::endl;

  // change number
  // change view_on_number

  return 0;
}
```

Referenzen

```cpp
#include <iostream>
#include <string>

int main()
{
  std::string name = "Tobias";
  char& first_letter = name[0];

  std::cout << "First letter: " << first_letter << std::endl;

  name = "Leibner";

  std::cout << "First letter: " << first_letter << std::endl;

  return 0;
}
```

Output:

```
First letter: F
First letter: S
```

swap.cc

```
#include <iostream>

void swap(double& a, double& b)
{
  double tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  double a = 1.;
  double b = 10.;

  std::cout << "before:" << std::endl;
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;

  swap(a, b);

  std::cout << "\nafter:" << std::endl;
  std::cout << "a = " << a << std::endl;
  std::cout << "b = " << b << std::endl;

  return 0;
}
```

sort

gesucht: Eine Funktion, die die Einträge eines Vektors nach Größe sortiert.

sort.cc

```cpp
#include <vector> // <- contains std::vector of "arbitrary" types
#include "sort.hh"

int main()
{
  std::vector<double> vec = {3.0, -2.0, -3.0, 1.0};

  print(vec); // 3 -2 -3 1
  sort(vec);
  print(vec); // -3 -2 1 3

  return 0;
}
```

sort.hh

```
#ifndef SORT_HH
#define SORT_HH

#include <iostream>
#include <vector>

void swap(double& a, double& b)
{
  double tmp = b;
  b = a;
  a = tmp;
}

void print(std::vector<double>& vec)
{
  for (unsigned int i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << " ";
  }
  std::cout << std::endl;
}

void sort(std::vector<double>& vec)
{
  // ?
}

#endif // SORT_HH
```

Pass-by-value vs. pass-by-reference

```cpp
#include <iostream>
#include <vector>


void print_length(std::vector<double>& vec)
{
  std::cout << "The vector has length " << vec.size() << std::endl;
}


int main()
{
  std::vector<double> large_vec(900000000, 0.0);

  print_length(large_vec); // <- have a look at your memory with htop

  return 0;
}
```

Pass-by-value vs. pass-by-reference

- pass-by-value

```
ReturnType function_name(ArgumentType arg)
{
  // arg is a local copy
}
```

- pass-by-reference

```
ReturnType function_name(ArgumentType& arg)
{
  // arg is a reference to an outside variable
}
```

const

Variablen (und Referenzen), welche als **const** definiert werden, können nicht mehr geändert werden.

const.cc

```
int main()
{
  const double pi = 3.141592654;
  pi = 4;

  return 0;
}
```

const

const.cc

```
#include <cmath>

double compute_error(double& a, double& b)
{
  return std::abs(a - b);
}


int main()
{
  const double pi = 3.141592654;

  compute_error(pi, 3.1);

  return 0;
}
```

outline

Typen und Objekte in C++

```
double pi = 3.1416;

pi + 1;
pi -= 1;
pi = 4;
```

Der Typ (`double`) eines Objekts (`pi`) legt fest,

- welchen Wert ein Objekt haben kann ($\pm 1.7 \cdot 10^{308}$), und
- welche Funktionalität ein Objekt hat (a.k.a. “was man damit machen kann”):
  - `+`
  - `-=`
  - `=`

Vektoren

gesucht: Eine Möglichkeit, Vektoren $v \in \mathbb{R}^d$ für $d \in \mathbb{N}^{>0}$ in C++ abzubilden, welche die folgenden Eigenschaften besitzen:

- Vektoren können addiert und subtrahiert werden: $v = u + w$, $v = u - w$
- Vektoren können miteinander multipliziert werden (Skalarprodukt): $u \cdot w$

$$u \cdot w := \sum_{i=0}^{d-1} u_i w_i$$

- Man kann die Norm von Vektoren berechnen:

$$|v| := \sqrt{\sum_{i=0}^{d-1} v_i^2}$$

1. Versuch: `std::vector`

vectors.cc

```cpp
#include <vector>

int main()
{
  std::vector<double> u(3, 1.0);
  std::vector<double> v(3, 2.0);

  u + v;

  return 0;
}
```

2. Versuch: einen eigenen Typen erfinden

vectors.cc

```
#include <vector>

int main()
{
  Vector u(3, 1.0); // <- we would like to have our own Vector
  Vector v(3, 2.0);

  return 0;
}
```

2. Versuch: einen eigenen Typen erfinden

vector.hh

```
#ifndef VECTOR_HH
#define VECTOR_HH

#include <vector>

class Vector
{
public:
  Vector(unsigned int sz, double value) // <- This special function is called
    : data_(sz, value)                  //    when a new Vector is created.
    , size_(sz)
  {
  }

  std::vector<double> data_;
  unsigned int size_;
};


#endif // VECTOR_HH
```

2. Versuch: einen eigenen Typen erfinden

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  std::cout << "The size of u is " << u.size_ << std::endl;
}
```

Eigene Typen: Klassen in C++

```
Vector v(3, 0.0);
```

- Der Zustand eines Objekts (v) wird durch seine Member (Daten) bestimmt:

```
class Vector
{
  // ...
  std::vector<double> data_; // 0.0, 0.0, 0.0
  unsigned int size_;        // 3
};
```

Zugriff auf Member ist mit . möglich:

```
v.size_; // 3
```

- Funktionalität?

Vector

vectors.cc

```
#include <iostream>
#include <cmath>      // <- for std::sqrt
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  // compute l2 norm of u
  double norm = 0.0;
  // ...
  norm = std::sqrt(norm);
  std::cout << "The l2 norm of u is " << norm << std::endl;
}
```

Vector

vectors.cc

```
#include <iostream>
#include <cmath>      // <- for std::sqrt
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  // compute l2 norm of u
  double norm = 0.0;
  // ...
  norm = std::sqrt(norm);
  std::cout << "The l2 norm of u is " << norm << std::endl;

  // compute l2 norm of v
  double norm_v = 0.0;
  // ...
  norm_v = std::sqrt(norm_v);
  std::cout << "The l2 norm of v is " << norm_v<< std::endl;
}
```

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  std::cout << "The l2 norm of u is " << u.l2_norm() << std::endl;
  std::cout << "The l2 norm of v is " << v.l2_norm() << std::endl;
}
```

Vector

vector.hh

```
#include <cmath>

class Vector
{
public:
  Vector(unsigned int sz, double value)
    : data_(sz, value)
    , size_(sz)
  {
  }

  double l2_norm()
  {
    double result = 0.0;
    // ...
    return std::sqrt(result);
  }

  std::vector<double> data_;
  unsigned int size_;
};
```

Vector

gesucht Eine Möglichkeit, zwei Vektoren zu addieren

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  Vector w = u.add(v);
}
```

Vector

vector.hh

```
#include <cassert>

class Vector
{
public:
  // ...

  Vector add(const Vector& other)
  {
    assert(other.size_ == size_);
    Vector result(size_, 0.0);
    // ...
    return result;
  }

  std::vector<double> data_;
  unsigned int size_;
};
```

assert

Die Funktion

```
assert(bool condition);
```

aus dem Header `#include <cassert>` beendet das Programm, falls `condition == false`.

Dieser check kann aus Effizienzgründen deaktiviert werden, kann also häufig verwendet werden!

Eigene Typen: Klassen in C++

```
Vector v(3, 0.0);
```

- Der Zustand eines Objekts wird durch seine Member (Daten) bestimmt:

```
v.data_; // 0.0, 0.0, 0.0
v.size_; // 3
```

- Die Funktionalität eines Objekts wird durch seine Methoden (Funktionen) bestimmt:

```
v.l2_norm();
w = u.add(v);
```

operator+

gesucht Eine Möglichkeit, zwei Vektoren zu addieren: `Vector w = u + v;`

"`Vector = Vector + Vector`"

operator+

gesucht Eine Möglichkeit, zwei Vektoren zu addieren: `Vector w = u + v;`

"`Vector = Vector + Vector`"

aber `Vector w = u + v` ist äquivalent zu

```
Vector w = u.operator+(v);
```

Lösung Die Klasse `Vector` benötigt also eine Methode (Funktion)

```
Vector operator+(const Vector& other);
```

Vector

vector.hh

```
#include <cassert>

class Vector
{
public:
  // ...
  Vector add(const Vector& other) // or just rename add to operator+
  {
    assert(other.size_ == size_);
    Vector result(size_, 0.0);
    for (unsigned int i = 0; i < size_; ++i)
      result.data_[i] = data_[i] + other.data_[i];
    return result;
  }

  Vector operator+(const Vector& other)
  {
    return add(other);
  }

  std::vector<double> data_;
  unsigned int size_;
};
```

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  Vector w = u + v;

  // std::cout << w << std::endl; // <- what about this?
}
```

Ausgabe von eigenen Typen: `operator<<`

Problem Ausgabe von eigenen Typen (z.B. `Vector`) in eine Datei oder auf dem Terminal.

Lösung Eine Funktion `operator<<`

- Jeder output Stream (`std::cout`, `std::ofstream`) ist vom Typ `std::ostream`. Wir benötigen also eine freie Funktion

```
std::ostream& operator<<(std::ostream& out, const Vector& vec);
```

- Bemerkung: es gibt viele verschiedene `operator<<` Funktionen (Überladung), z.B.

```
std::ostream& operator<<(std::ostream& out, const double& number);
std::ostream& operator<<(std::ostream& out, const std::string& word);
```

Vector

vector.hh

```
#include <iostream>

class Vector
{
public:
  // ...

  std::vector<double> data_;
  unsigned int size_;
};


std::ostream& operator<<(std::ostream& out, const Vector& vec)
{
  for (unsigned int i = 0; i < vec.size_; ++i) {
    out << vec.data_[i] << " ";
  }
  return out;
}
```

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  Vector w = u + v;

  std::cout << w << std::endl;
}
```

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  Vector w = u + v;

  std::cout << w << std::endl;
}
```

⇒ Ändern Sie `operator<<` so, dass die Ausgabe `[3.0 3.0 3.0]` ergibt (und `[]` für Vektoren der Länge 0).

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);

  u.size_ = 4; // <- What about this?

  std::cout << u << std::endl;
}
```

Vector

Problem Änderung der Member kann Zustand eines Objekts zerstören.

Lösung Direkten Zugriff auf Member verbieten, wenn dadurch der Zustand eines Object inkonsistent werden kann.

vector.hh

```
class Vector
{
public:
  // ...

private:
  std::vector<double> data_; // <- No one from outside is allowed to
  unsigned int size_;        //    access these private members!
};
```

Vector

Problem Änderung der Member kann Zustand eines Objekts zerstören.

Lösung Direkten Zugriff auf Member verbieten, wenn dadurch der Zustand eines Object inkonsistent werden kann.

- Änderung des Zustands nur durch Methoden erlauben.

vector.hh

```
class Vector
{
public:
  // ...

  unsigned int size()
  {
    return size_; // <- But all methods of Vector are allowed to!
  }

  double get_entry(unsigned int i)
  {
    assert(i < size_);
    // ...
  }

  void set_entry(unsigned int i, double x)
  {
    // ...
  }

private:
  std::vector<double> data_; // <- No one from outside is allowed to
  unsigned int size_;        //    access these private members!
};
```

private

Zugriff auf `private` Member ist von außen nicht mehr möglich.

⇒ Anpassung von `operator<<` nötig.

Problem Wenn man ein `const` Objekt hat, z.B.

```
const Vector u(3, 0.0);
```

oder eine `const` Referenz, z.B.

```
void function(const Vector& vec) { ... }
```

darf man keine Methoden benutzen, die das Objekt ändern könnten.

Lösung "Ungefährliche" Methoden (welche das Objekt nicht ändern) als `const` deklarieren!

vector.hh

```
class Vector
{
public:
  // ...

  Vector operator+(const Vector& other) const { ... }
  Vector operator=(const Vector& other) const { ... }

  unsigned int size() const
  {
    return size_;
  }

  double get_entry(unsigned int i) const
  {
    // ...
  }

  void set_entry(unsigned int i, double x)
  {
    // ...
  }

private:
  std::vector<double> data_;
  unsigned int size_;
};
```

Zwischenbilanz: Klassen

- Klassen sind eigene Typen und werden mit dem Keyword `class` erstellt:

```
class Dummy
{
  // ...
}; // <- mind the ;!
```

- Klassen können eigene Daten als Member haben:

```
class Dummy
{
  // ...
  int number_;
  std::string name_; // <- mind the _
};
```

- Klassen können eigene Funktionalität (Methoden) haben:

```
class Dummy
{
  // ...
  void print_something()
  {
    std::cout << "something" << std::endl;
  }
};
```

Zwischenbilanz: Klassen

- Member Variablen sollte immer mit _ enden, um Namenskonflikte zu vermeiden.

```
class Dummy
{
  int size()                    // <- or that one?
  {
    return size; // <- which one?
  }
  int size;             // <- this one
};
```

- Zugriff auf Member und Methoden von außen kann mit **private**: untersagt werden:

```
class Dummy
{
public:
  int dim;

  std::string name()
  {
    return name_;
  }
private:
  std::string name_;
};
```

```
std::cout << dummy.dim << std::endl;
dummy.dim = 3;

std::cout << dummy.name_; // compile error
std::cout << dummy.name();
```

Zwischenbilanz: Klassen

- Methoden, die den Zustand eines Objektes nicht verändern, können mit `const` markiert werden:

```
class Dummy
{
  // ...
  int size() const
  {
    return size_;
  }
private:
  int size_;
};
```

```
void print(const Dummy& dummy)
{
  std::cout << "Dummy has size "
            << dummy.size() << std::endl;
}
```

Zwischenbilanz: Klassen

- Der Konstruktor hat denselben Namen wie die Klasse und ist eine spezielle Funktion die aufgerufen wird, wenn ein Objekt erzeugt wird:

```
class Dog
{
public:
  Dog(std::string name)
  {
  }

  Dog(int age)
  {
  }
};
```

```
Dog struppi("Struppi");
Dog tim(4);
```

- Es kann eine beliebige Menge von Konstruktoren geben, solange diese unterschiedliche Signatur haben.

Zwischenbilanz: Klassen

- Um die Member (Daten) eines Objektes bei der Erzeugung zu initialisieren, kann man die "constructor initializer list" benutzen:

```
class Dog
{
public:
  Dog(std::string name)
    : name_(name) // same order: 1. name_
    , age_(0)     //              2. age_
  {
  }

  Dog(int age)
    : name_("good dog")
    , age_(age)
  {
  }

private:
  std::string name_; // as here
  int age_;
};
```

```
Dog struppi("Struppi"); // age_ = 0
                        // name_ = "Struppi"

Dog tim(4); // name_ = "good dog"
            // age_  = 4
```

Zwischenbilanz: Klassen

- Nur so ist es möglich, `const` member zu haben, die nie verändert werden können:

```
class StaticVector
{
public:
  StaticVector(int sz)
    : size(sz)
  {
  }

  //void change_size(int sz)
  //{
  //  size = sz; // <- would not compile
  //}

  const int size;
};
```

```
StaticVector vec(3);

// vec.size = 5;       // <- not allowed

// vec.change_size(5); // <- not allowed
```

operator+=

gesucht: Eine Möglichkeit `u += v` zu rechnen.

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  std::cout << u << std::endl;
  u += v;
  std::cout << u << std::endl;
}
```

operator+=

gesucht: Eine Möglichkeit `u += v` zu rechnen.

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  std::cout << u << std::endl;
  u += v;
  std::cout << u << std::endl;
}
```

⇒ **operator+=**

vector.hh

```
class Vector
{
public:
  // ...

  void operator+=(const Vector& other)
  {
    // ...
  }

  // ...
};
```

Vector

vectors.cc

```
#include <iostream>
#include "vector.hh"

int main()
{
  Vector u(3, 1.0);
  Vector v(3, 2.0);

  std::cout << (u += v) << std::endl;
}
```

gesucht: Eine Möglichkeit, sich selbst zurück zu geben.

Lösung: `*this`

vector.hh

```
class Vector
{
public:
  // ...

  Vector& operator+=(const Vector& other)
  {
    // ...
    return *this;
  }

  // ...
};
```

operator[]

gesucht: Eine Möglichkeit, auf einzelne Einträge direkt zuzugreifen:

```
std::cout << v[0] << std::endl;
```

operator[]

gesucht: Eine Möglichkeit, auf einzelne Einträge direkt zuzugreifen:
`std::cout << v[0] << std::endl;`

aber: `v[0]` ist äquivalent zu

```
v.operator[](0)
```

Lösung: Die Klasse `Vector` benötigt also eine Methode (Funktion)

```
double operator[](int i) const { ... }
```

operator[]

gesucht: Eine Möglichkeit, auf einzelne Einträge direkt zuzugreifen, und zwar
lesend: `std::cout << v[0] << std::endl;`
schreibend: `u[0] = 4.0;`

aber: `v[0] = 4.0;` ist äquivalent zu

```
v.operator[](0) = 4.0;
```

operator[]

gesucht: Eine Möglichkeit, auf einzelne Einträge direkt zuzugreifen, und zwar
lesend: `std::cout << v[0] << std::endl;`
schreibend: `u[0] = 4.0;`

aber: `v[0] = 4.0;` ist äquivalent zu

```
v.operator[](0) = 4.0;
```

Lösung: Die Klasse `Vector` benötigt also eine Methode (Funktion)

```
double& operator[](int i) { ... }
```

outline

Was passiert beim Anlegen eines Objektes?

classes.cc

```
#include <iostream>
#include <string>

class Dummy
{
public:

  std::string name;
};

int main()
{
  Dummy a;

  std::cout << "a.name = '" << a.name << "'" << std::endl;
}
```

Der Compiler erstellt immer automatisch einen Default-Konstruktor

```
class Dummy
{
public:
  Dummy()
  {}

  std::string name;
};
```

(falls alle Member Default-konstruierbar sind).

Aber Vorsicht:

classes.cc

```cpp
#include <iostream>
#include <string>

class Dummy
{
public:

  std::string name;
  double age;
};

int main()
{
  Dummy a;

  std::cout << "a.name = '" << a.name << "'" << std::endl;
  std::cout << "a.age  = '" << a.age << "'" << std::endl;   // <- Does this make sense?
}
```

Aber Vorsicht:

"Default-konstruierbar" bedeutet: eine Variable von diesem Typ kann angelegt werden, z.B.

```
int i;
double pi;
Dummy a;
```

aber *nicht*, dass diese auch sinnvolle Werte enthält!

Also: es sollte einen Konstruktor geben

classes.cc

```cpp
#include <iostream>
#include <string>

class Dummy
{
public:
  Dummy()
    : name("no name")
    , age(0)
  {
    std::cout << "Dummy() constructor called" << std::endl;
  }

  std::string name;
  double age;
};

int main()
{
  Dummy a;
  std::cout << "a.name = '" << a.name << "'" << std::endl;
  std::cout << "a.age  = '" << a.age << "'" << std::endl;
}
```

Also: es sollte einen Konstruktor geben

classes.cc

```
#include <iostream>
#include <string>

class Dummy
{
public:
  Dummy(std::string nm = "no name")
    : name(nm)
  {
    std::cout << "Dummy(" << name << ") constructor called" << std::endl;
  }

  std::string name;
};

int main()
{
  Dummy a("a");

  std::cout << "a.name = '" << a.name << "'" << std::endl;
}
```

Warum kann ein Objekt kopiert werden?

classes.cc

```
#include <iostream>
#include <string>

class Dummy
{
public:
  Dummy(std::string nm = "no name")
    : name(nm)
  {
    std::cout << "Dummy(" << name << ") constructor called" << std::endl;
  }

  std::string name;
};

int main()
{
  Dummy a("a");
  Dummy b = a;  // <- ?

  std::cout << "b.name = '" << b.name << "'" << std::endl;
}
```

Warum kann ein Objekt kopiert werden?

Der Compiler erstellt immer automatisch einen Copy-Konstruktor, der alle Member kopiert

```
class Dummy
{
public:
  // ...

  Dummy(const Dummy& other)
    : name(other.name)
  {}

  std::string name;
};
```

(falls alle Member kopierbar sind).

Also: eigener Copy-Konstruktor in Spezialfällen

classes.cc

```cpp
#include <iostream>
#include <string>

class Dummy
{
public:
  // ...
  Dummy(const Dummy& other)
    : name("copy of " + other.name)
  {
    std::cout << "Dummy(" << name << ") copy-constructor called" << std::endl;
  }

  std::string name;
};

int main()
{
  Dummy a;
  Dummy b = a;

  std::cout << "b.name = '" << b.name << "'" << std::endl;
}
```

Warum kann einem Objekt ein neuer Wert zugewiesen werden?

classes.cc

```
#include <iostream>
#include <string>

class Dummy
{
public:
  // ...
  Dummy(const Dummy& other)
    : name("copy of " + other.name)
  {
    std::cout << "Dummy(" << name << ") copy-constructor called" << std::endl;
  }

  std::string name;
};

int main()
{
  Dummy a("a");
  Dummy b = a;
  Dummy c("c");
  c = b;        // <- ?

  std::cout << "c.name = '" << c.name << "'" << std::endl;
}
```

Warum kann einem Objekt ein neuer Wert zugewiesen werden?

Der Compiler erstellt immer automatisch einen Zuweisungsoperator, der alle Member kopiert

```
class Dummy
{
public:
  // ...
  Dummy& operator=(const Dummy& other)
  {
    name = other.name;
    return *this;
  }

  std::string name;
};
```

(falls alle Member zuweisbar sind).

Also: eigener `operator=` in Spezialfällen

classes.cc

```
class Dummy
{
public:
  // ...
  Dummy& operator=(const Dummy& other)
  {
    std::cout << "Dummy(" << name << ") assignment operator called" << std::endl;
    name = "assigned from " + other.name;
    return *this;
  }

  std::string name;
};

int main()
{
  Dummy a("a");
  Dummy b = a;
  Dummy c("c");
  c = b;

  std::cout << "c.name = '" << c.name << "'" << std::endl;
}
```

Regeln beim Anlegung neuer Objekte

Bei einer Klasse `class ThisType {};`

- Default-Konstruktor: `ThisType()`
  ⇒ wird automatisch vom Compiler angelegt, falls benötigt (und falls alle Member Default konstruierbar sind)
- Copy-Konstruktor: `ThisType(const ThisType& other)`
  ⇒ wird automatisch vom Compiler angelegt, falls benötigt (und falls alle Member kopierbar sind)
- Konstruktor-Initializer-List: kann bei jedem Konstruktor verwendet werden

  `ThisType(...) : member_1_(/*args_1*/), member_2_(/*args_2*/), ... {}`

  Dabei bezeichnet `/*args_1*/` alles, was der Konstruktor von `member_1_` benötigt.
- Assignment-Operator: `ThisType& operator=(const ThisType& other)`
  ⇒ wird automatisch vom Compiler angelegt, falls benötigt (und falls alle Member zuweisbar sind)

`default` und `delete`

Regeln beim Anlegung neuer Objekte

⇒ Wenn man etwas anderes als eine Kopie der Member benötigt, muss man

- `ThisType(const ThisType& other)`
- `ThisType& operator=(const ThisType& other)`

selber implementieren.

⇒ Wenn man nur eine Kopie der Member benötigt, sollte man

- `ThisType(const ThisType& other) = default;`
- `ThisType& operator=(const ThisType& other) = default;`

schreiben.

⇒ Wenn man eine Kopie oder Zuweisung verbieten möchte:

- `ThisType(const ThisType& other) = delete;`
- `ThisType& operator=(const ThisType& other) = delete;`

Kopie oder Zuweisung verbieten

non_copyable.cc

```
#include <iostream>

class NonCopyable
{
public:
  NonCopyable(int n)
    : number(n)
  {}

  NonCopyable(const NonCopyable& other) = default;

  int number;
};


int main()
{
  NonCopyable a(1);
  NonCopyable b = a;

  std::cout << "b.number = " << b.number << std::endl;
}
```

Kopie oder Zuweisung verbieten

non_copyable.cc

```
#include <iostream>

class NonCopyable
{
public:
  NonCopyable(int n)
    : number(n)
  {}

  NonCopyable(const NonCopyable& other) = delete; // <-

  int number;
};


int main()
{
  NonCopyable a(1);
  NonCopyable b = a;

  std::cout << "b.number = " << b.number << std::endl;
}
```

Vererbung

Vererbung

Oder: wie man einen neuen Typ auf einem bestehenden aufbaut.

classes.cc

```
class Dummy
{
  // ...
  std::string name;
}

class OldDummy
  : public Dummy     // <- is like a Dummy
{
public:
  int age;           //    and a bit more
};

int main()
{
  OldDummy a;

  std::cout << "a.name = '" << a.name << "'" << std::endl;
  std::cout << "a.age  = '" << a.age << "'" << std::endl;
}
```

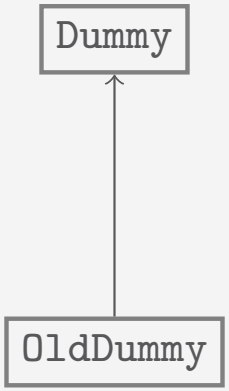
Vererbung

Einen neuen Typen

```
class Derived : public Base {/*...*/};
```

zu definieren, bedeutet dass `Derived` Zustand und Funktionalität von `Base` erbt.

"`Derived` ist von `Base` abgeleitet"



Was passiert hier?

classes.cc

```
class OldDummy
  : public Dummy
{
public:
  int age;
};


int main()
{
  OldDummy a;

  std::cout << "a.name = '" << a.name << "'" << std::endl; // Why 'no name'?
  std::cout << "a.age  = '" << a.age  << "'" << std::endl;
}
```

Es wird immer sicher gestellt, dass der Speicher für alle Objekte reserviert wird, bevor sie benutzt werden können.

⇒ Bei `class OldDummy : public Dummy {};` wird zuerst aller Speicher im Zusammenhang mit `Dummy` reserviert, dann der Speicher für `OldDummy`.

⇒ Wenn ein Objekt vom Typ `OldDummy` erstellt wird, wird immer zuerst der Konstruktor von `Dummy` aufgerufen.

classes.cc

```
class OldDummy
  : public Dummy
{
public:
  OldDummy(std::string nm, int i = 0)
    : age(i)
  {
    std::cout << "OldDummy(" << nm << ", " << age << ") constructor called" << std::endl;
  }

  int age;
};

int main()
{
  OldDummy a("a");

  std::cout << "a.name = '" << a.name << "'" << std::endl;
  std::cout << "a.age  = '" << a.age << "'" << std::endl;
}
```

Wie kann der Konstruktor der Basisklasse aufgerufen werden?

classes.cc

```
class OldDummy
  : public Dummy
{
public:
  OldDummy(std::string nm, int i = 0)
    : Dummy(nm) // <- Treat base class as first member!
    , age(i)
  {
    std::cout << "OldDummy(" << name << ", " << age << ") constructor called" << std::endl;
  }

  int age;
};


int main()
{
  OldDummy a("a");

  std::cout << "a.name = '" << a.name << "'" << std::endl;
  std::cout << "a.age  = '" << a.age << "'" << std::endl;

  return 0;
}
```

Wie verhält es sich mit dem Copy-Konstruktor?

classes.cc

```
class OldDummy
  : public Dummy
{
public:
  // ...

  int age;
};


int main()
{
  OldDummy a("a");
  OldDummy b = a;

  std::cout << "b.name = '" << b.name << "'" << std::endl;
  std::cout << "b.age  = '" << b.age  << "'" << std::endl;

  return 0;
}
```

⇒ Der Copy-Konstruktor von `Dummy` wird zuerst aufgerufen!

```
class OldDummy
  : public Dummy
{
public:
  // ...

  OldDummy(const OldDummy& other)
    : Dummy(other)           // <- Will copy name
    , age(other.age + 1)
  {
    std::cout << "OldDummy(" << name << ", " << age <<  ") copy-constructor called"
              << std::endl;
  }

  int age;
};


int main()
{
  OldDummy a("a");
  OldDummy b = a;

  std::cout << "b.name = '" << b.name << "'" << std::endl;
  std::cout << "b.age  = '" << b.age  << "'" << std::endl;
}
```

Wie verhält es sich mit dem assignment-Operator?

classes.cc

```
class OldDummy
  : public Dummy
{
public:
  // ...

  int age;
};


int main()
{
  OldDummy a("a");
  OldDummy b = a;
  OldDummy c("c", 4);
  c = b;

  std::cout << "c.name = '" << c.name << "'" << std::endl;
  std::cout << "c.age  = '" << c.age << "'" << std::endl;

  return 0;
}
```

classes.cc

```
class OldDummy
  : public Dummy
{
public:
  // ...
  OldDummy& operator=(const OldDummy& other)
  {
    Dummy::operator=(other); // Will assign name.
    age = other.age + 1;
    return *this;
  }

  int age;
};


int main()
{
  OldDummy a("a");
  OldDummy b = a;
  OldDummy c("c", 4);
  c = b;

  std::cout << "c.name = '" << c.name << "'" << std::endl;
  std::cout << "c.age  = '" << c.age << "'" << std::endl;
}
```

Zwischenbilanz: Vererbung

Eine abgeleiteten Klasse

```
class Derived : public Base {};
```

- erbt alle Member der Basisklasse;
- hat Zugriff auf den Konstruktor der Basisklasse "als ersten Member":
  `Derived() : Base(), ... {}`
- kann Member und Methoden der Basisklasse explizit durch den `Base::`-Präfix ansteuern:
  `Base::operator=`
- ist immer auch vom Typ der Basisklasse (*upcasting*).

upcasting

```
int main()
{
  //OldDummy a("a", 1);
  //OldDummy b = a;
  //OldDummy c("c", 4);
  //c = b;

  OldDummy old("old", 29);

  std::cout << "old.name = '" << old.name << "'" << std::endl;
  std::cout << "old.age  = '" << old.age << "'" << std::endl;

  OldDummy& view_on_old = old;

  std::cout << "view_on_old.name = '" << view_on_old.name << "'" << std::endl;
  std::cout << "view_on_old.age  = '" << view_on_old.age << "'" << std::endl;

  return 0;
}
```

upcasting

```
int main()
{
  //OldDummy a("a", 1);
  //OldDummy b = a;
  //OldDummy c("c", 4);
  //c = b;

  OldDummy old("old", 29);

  std::cout << "old.name = '" << old.name << "'" << std::endl;
  std::cout << "old.age  = '" << old.age << "'" << std::endl;

  Dummy& view_on_old = old; // <-

  std::cout << "view_on_old.name = '" << view_on_old.name << "'" << std::endl;
  std::cout << "view_on_old.age  = '" << view_on_old.age << "'" << std::endl;

  return 0;
}
```

⇒ Vorsicht: Informationsverlust durch upcasting möglich (*slicing*)!

virtuelle Funktionen

virtuelle Funktionen

functions.cc

```
#include <iostream>
#include <string>

struct Function
{
  double evaluate(double x) const
  {
    return 2.0*x;
  }
};

void eval_and_print(const Function& func, const std::string& name)
{
  double x = 1.0;
  std::cout << name << "(" << x << ") = " << func.evaluate(x) << std::endl;
}

int main()
{
  Function f;
  std::cout << f.evaluate(1.0) << std::endl;
  eval_and_print(f, "f");
}
```

virtuelle Funktionen

```
class Function
{
  // ...
};

class FunctionWithDerivative
  : public Function
{
public:
  double derivative(double x) const
  {
    return 1.0;
  }
};

int main()
{
  Function f;
  std::cout << f.evaluate(1.0) << std::endl;
  eval_and_print(f, "f");

  FunctionWithDerivative g;
  std::cout << "\n" << g.evaluate(1.0) << std::endl;
}
```

virtuelle Funktionen

```
class Function
{
  // ...
};

class FunctionWithDerivative
  : public Function
{
public:
  double evaluate(double x) const   { return x + 10; } // <-

  double derivative(double x) const { return 1.0; }    // <-
};

int main()
{
  Function f;

  std::cout << f.evaluate(1.0) << std::endl;
  eval_and_print(f, "f");

  FunctionWithDerivative g;
  std::cout << "\n" << g.evaluate(1.0) << std::endl;
}
```

virtuelle Funktionen

```
class Function
{
  // ...
};

class FunctionWithDerivative
  : public Function
{
public:
  double evaluate(double x) const   { return x + 10; }

  double derivative(double x) const { return 1.0; }
};

int main()
{
  Function f;
  std::cout << f.evaluate(1.0) << std::endl;
  eval_and_print(f, "f");

  FunctionWithDerivative g;
  std::cout << "\n" << g.evaluate(1.0) << std::endl;
  eval_and_print(g, "g");                                  // <-
}
```

virtuelle Funktionen

Problem: Upcasting! Da

```
void eval_and_print(const Function& func, const std::string& name)
```

ein Objekt vom Typ `const Function&` erwartet, wird beim Aufruf

```
FunctionWithDerivative g;
eval_and_print(g, "g");
```

das Objekt g vom Typ `FunctionWithDerivative` als Typ `Function` interpretiert.

virtuelle Funktionen

Problem: Upcasting! Da

```
void eval_and_print(const Function& func, const std::string& name)
```

ein Objekt vom Typ `const Function&` erwartet, wird beim Aufruf

```
FunctionWithDerivative g;
eval_and_print(g, "g");
```

das Objekt g vom Typ `FunctionWithDerivative` als Typ `Function` interpretiert.

Lösung Methoden als `virtual` markieren!

virtuelle Funktionen

```
class Function
{
public:
  virtual double evaluate(double x) const // <-
  {
    return 2.0*x;
  }
};

class FunctionWithDerivative
  : public Function
{
public:
  double evaluate(double x) const override // <-
  {
    return x + 10;
  }

  double derivative(double x) const
  {
    return 1.0;
  }
};
```

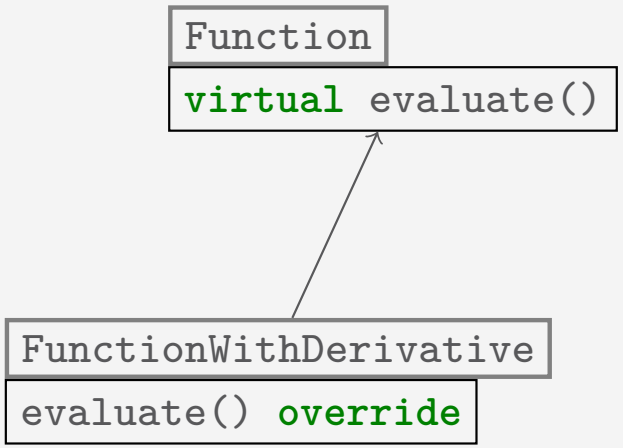
virtuelle Funktionen

Obwohl der Typ von g in `eval_and_print` zur *Kompilezeit* auf `Function` beschränkt wird (upcasting)

```
void eval_and_print(const Function& func,
                    /*...*/)
{
  func.evaluate(1.0);
}

FunctionWithDerivative g;
eval_and_print(g);
```

wird zur *Laufzeit* die richtige `evaluate` Methode ausgewählt.



Interfaces und pure `virtual`

Interfaces

gesucht: Eine Möglichkeit, eine bestimmte Funktionalität von Klassen zu erzwingen.

function_interface.hh

```
#ifndef FUNCTION_INTERFACE_HH
#define FUNCTION_INTERFACE_HH

#include <fstream>

class FunctionInterface
{
public:
  virtual std::string name() const = 0;
  virtual double evaluate(double x) const = 0;
};

void visualize(const FunctionInterface& func)
{
  std::ofstream output(func.name() + ".txt");
  for (double x = 0.0; x <= 1.0; x+= 0.01) {
    output << x << "\t" << func.evaluate(x) << std::endl;
  }
}

#endif // FUNCTION_INTERFACE_HH
```

Interfaces

test_functions.cc

```
#include "function_interface.hh"


class LinearFunction
  : public FunctionInterface
{
public:
  std::string name() const override
  {
    return "linear";
  }

};

int main()
{
  LinearFunction f;
  visualize(f);
}
```

outline

Error handling mit `exceptions`

exceptions

exceptions.cc

```
#include <iostream>
#include <cassert>

class Dummy
{
public:
  void do_something()
  {
    assert(age < 0);
    std::cout << "age = " << age << std::endl;
  }

  int age;
};

int main()
{
  Dummy a;
  a.age = 1;
  a.do_something();
}
```

exceptions

Problem bei `assert`:

- Das Programm wird sofort beendet (keine Reaktion möglich).
- Man erhält keine Informationen über den Zustand, der zum Abbruch führt.

exceptions

exceptions.cc

```
#include <iostream>
#include <string>
#include <stdexcept>

struct Dummy
{
  void do_something()
  {
    if (age >= 0)
      throw std::runtime_error("Dummy::do_something(), age = " + std::to_string(age));
    std::cout << "age = " << age << std::endl;
  }

  int age;
};

int main()
{
  Dummy a;
  a.age = 1;
  a.do_something();
}
```

exceptions

exceptions.cc

```
int main()
{
  try {
    Dummy a;
    a.age = 1;
    a.do_something();
  } catch (std::exception& e) {
    std::cout << "Error: " << e.what() << std::endl;
    return 1;
  }
  return 0;
}
```

pointer

pointer

Pointer enthalten die Speicheradresse eines Objekts:

pointer.cc

```
#include <iostream>

int main()
{
  double pi = 3.1416;
  double* pi_ptr = &pi; // &pi means: the memory address of pi

  std::cout << "pi = " << pi
            << "\npi_ptr = " << pi_ptr << std::endl;
}
```

pointer

Pointer können mit ∗ dereferenziert werden, um auf das ursprüngliche Objekt zuzugreifen.

pointer.cc

```
#include <iostream>

int main()
{
  double pi = 3.1416;
  double* pi_ptr = &pi;

  std::cout << "pi = " << pi
            << "\npi_ptr = " << pi_ptr << std::endl;

  std::cout << "\n*pi_ptr = " << *pi_ptr << std::endl;
}
```

pointer

Pointer können mit $*$ dereferenziert werden, um auf das ursprüngliche Objekt zuzugreifen.

pointer.cc

```
#include <iostream>

int main()
{
  double pi = 3.1416;
  double* pi_ptr = &pi;

  std::cout << "pi = " << pi
            << "\npi_ptr = " << pi_ptr << std::endl;

  std::cout << "\n*pi_ptr = " << *pi_ptr << std::endl;
}
```

⇒ Was ist der Vorteil gegenüber Referenzen?

dynamische Speicherverwaltung

Objekte werden auf dem Stack angelegt und können nur von begrenzter Größe (üblicherweise ca. 8MB) sein, sonst: `std::bad_alloc`!

- Historischer Nutzen von Pointern: für größere Objekte muss man mehr Speicher mit **`new`** vom Heap anfordern (**`new`** gibt einen Pointer zurück).

`memory_leak.cc`

```
#include <iostream>

void use_memory(int size)
{
  double** matrix = new double*[size];
  for (int i = 0; i < size; ++i)
    matrix[i] = new double[size];

  // entries of matrix can be accessed by matrix[i][j]
}

// ...
```

dynamische Speicherverwaltung

memory_leak.cc

```
// ...

int main()
{
  int size;
  std::cout << "enter size: ";
  std::cin >> size; // 2000000 for machines with 16GB RAM

  use_memory(size);

  std::cout << "This is the end!";
  std::cin >> size;

  return 0;
}
```

⇒ Mit `htop` beobachten!

⇒ Mit **`new`** erhaltener Speicher muss manuell mit **`delete`** freigegeben werden!

dynamische Speicherverwaltung

```
void use_memory(int size)
{
  double** matrix = new double*[size];
  for (int i = 0; i < size; ++i)
    matrix[i] = new double[size];

  // do stuff with matrix

  // clean up
  for (int i = 0; i < size; ++i)
    delete matrix[i];
  delete matrix;
}
```

dynamische Speicherverwaltung

`double*` vs. `std::vector<double>`:

- Speicherlecks bei `double*`
- `std::vector<double>` kennt seine Größe:

```
void print_vector(double* vec, int size)
{
  for (int i = 0; i < size; ++i)
    // ...
}

void print_vector(std::vector<double>& vec)
{
  for (int i = 0; i < vec.size(); ++i)
    // ...
}
```

- `std::vector<double>` hält intern einen `double*`

managed memory

Wenn Sie doch einmal einen pointer benötigen: `#include <memory>`

- `std::shared_ptr`
- `std::unique_ptr`

Können wie Pointer verwendet werden, garantieren aber korrektes Speichermanagement (keine memory leaks, etc.).

Templates

templatisierte Funktionen

function_template.cc

```
#include <iostream>


template <class T>
void print_product(const T& left, const T& right)
{
  std::cout << left << " * " << right << " = " << left*right << std::endl;
}


int main()
{
  print_product(1, 2);      // will create print_product<int>
  print_product(2.0, -1.0); // will create print_product<double>
}
```

templatisierte Funktionen

function_template.cc

```
#include <iostream>


template <class T>
void print_product(const T& left, const T& right)
{
  std::cout << left << " * " << right << " = " << left*right << std::endl;
}


int main()
{
  print_product(1, 2);       // will create print_product<int>
  print_product(2.0, -1.0); // will create print_product<double>
  print_product(1, 'a');
}
```

templatisierte Funktionen

function_template.cc

```
#include <iostream>


template <class T>
void print_product(const T& left, const T& right)
{
  std::cout << left << " * " << right << " = " << left*right << std::endl;
}


int main()
{
  print_product(1, 2);       // will create print_product<int>
  print_product(2.0, -1.0); // will create print_product<double>
  print_product(1, 2.0);
}
```

templatisierte Funktionen

function_template.cc

```
#include <iostream>


template <class T>
void print_product(const T& left, const T& right)
{
  std::cout << left << " * " << right << " = " << left*right << std::endl;
}


int main()
{
  print_product(1, 2);       // will create print_product<int>
  print_product(2.0, -1.0); // will create print_product<double>
  print_product<double>(1, 2.0);
}
```

Rekursion mit Templates

recursive_templates.cc

```cpp
#include <iostream>


template <int n>
int factorial()
{
  static_assert(n > 0, "n has to be positive");
  return n*factorial<n - 1>();
}

template <>
int factorial<1>()
{
  return 1;
}


int main()
{
  std::cout << factorial<-1>() << std::endl;
  std::cout << factorial<10>() << std::endl; // All work has been done by the compiler!
}
```

templatisierte Klassen

```
#include "../day_3/vector.hh"

template <class T>
class Container
{
public:
  T data;
};


int main()
{
  Container<int> a;
  a.data = 1;

  Container<double> b;
  b.data = 2.0;

  Container<Vector> c;
  c.data = Vector(3, 0.0);
}
```

auto

auto: der Compiler weiß es eh ...

```
#include "../../day_3/vector.hh"

int func_i()
{
  return 1;
}

double func_d()
{
  return 1.0;
}

int main()
{
  auto number = 1;   // int
  auto letter = 'T'; // char
  auto pi = 3.1416;  // double
  auto i = func_i(); // int
  auto d = func_d(); // double
  auto x = Vector(3, 1.0) + Vector(3, 2.0); // Vector(3, 3.0)
  auto& view_on_number = number;       // int&
  const auto& view_on_letter = letter; // const char&
}
```

Die `std::` Bibliothek: Container und Algorithmen

std::vector

stl.cc

```
#include <iostream>
#include <vector>

template <class C>
void print(const C& container)
{
  std::cout << std::endl;
  for (unsigned int i = 0; i < container.size(); ++i)
    std::cout << container[i] << " ";
  std::cout << std::endl;
}

int main()
{
  std::vector<int> vec; // <- std::vector is a templated class
  vec.push_back(1);
  vec.push_back(-1);
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(-1);
  vec.push_back(3);
  print(vec); // <- print<std::vector<int>>() is created by the compiler
}
```

std::set

stl.cc

```cpp
#include <set>

// ...

int main()
{
  // ...

  print(vec);

  std::set<int> set; // <- std::set is a templated class
  set.insert(1);
  set.insert(-1);
  set.insert(1);
  set.insert(2);
  set.insert(-1);
  set.insert(3);
}
```

std::set

stl.cc

```
#include <set>

// ...

int main()
{
  // ...

  print(vec);

  std::set<int> set;
  set.insert(1);
  set.insert(-1);
  set.insert(1);
  set.insert(2);
  set.insert(-1);
  set.insert(3);

  print(set); // <-
}
```

⇒ Wie greift man auf die Einträge eines Containers zu (der keinen `operator[]` besitzt)?

Intermezzo: Iteratoren

Iteratoren

stl.cc

```
template <class C>
void print(const C& container)
{
  std::cout << std::endl;
  for (auto it = container.begin(); it != container.end(); ++it) // <-
    std::cout << *it << " ";                                     // <-
  std::cout << std::endl;
}
```

⇒ Iteratoren werden von allen `std::`-Containern bereit gestellt durch `begin()` und `end()`.

⇒ Zugriff auf den ursprünglichen Eintrag ist durch dereferenzieren möglich: `*it`.

Iteratoren

stl.cc

```
template <class C>
void print(const C& container)
{
  std::cout << std::endl;
  for (auto it = container.begin(); it != container.end(); ++it) // <-
    std::cout << *it << " ";                                     // <-
  std::cout << std::endl;
}
```

⇒ Iteratoren werden von allen `std::`-Containern bereit gestellt durch `begin()` und `end()`.

⇒ Zugriff auf den ursprünglichen Eintrag ist durch dereferenzieren möglich: `*it`.

⇒ Was macht `std::set`?

Intermezzo: range-based `for` loops

range-based `for` loops

stl.cc

```
// ...

int main()
{
  // ...
  print(set);

  for (auto& element : vec)
    element += 1;

  print(vec);

  for (auto element : {1, 2, 5})
    std::cout << element << std::endl;
}
```

Zurück zur `std::` Bibliothek: Container und Algorithmen

std::map (als eine Menge von std::pair)

map.cc

```cpp
#include <iostream>
#include <string>
#include <map>


int main()
{
  std::map<std::string, int> people;
  people["Tobias"]  = 29;
  people["Thomas"]  = 34;
  people["Sabrina"] = 33;

  // would have also worked:
  // for (auto individual : people) {
  for (std::pair<std::string, int> individual : people) {
    std::cout << individual.first << " is " << individual.second << std::endl;
  }

  return 0;
}
```

Intermezzo: lambdas

lambdas

lambdas.cc

```
#include <iostream>

int main()
{
  auto f = [](double x) { return x*x; };        // <- similar to 'double f(double x);'
  auto g = [](double x) { return 2.0*x + 3; };

  for (auto x : {-1.0, 0.0, 2.0}) {
    std::cout << "f(" << x << ") = " << f(x) << std::endl;
    std::cout << "g(" << x << ") = " << g(x) << std::endl;
  }
}
```

Zurück zur `std::` Bibliothek: Container und Algorithmen

std::sort

stl.cc

```
#include <algorithm> // <- std::set

// ...

int main()
{
  // ...
  vec = {-1, 3, 4, -2};
  print(vec);

  std::sort(vec.begin(), vec.end());
  print(vec);
}
```

std::sort

stl.cc

```
#include <algorithm> // <- std::set

// ...

int main()
{
  // ...
  vec = {-1, 3, 4, -2};
  print(vec);

  std::sort(vec.begin(), vec.end());
  print(vec);

                                    // comparator lambda: defines sorting
  std::sort(vec.begin(), vec.end(), [](int a, int b) {return a > b;});
  print(vec);
}
```

std::find

find.cc

```cpp
#include <iostream>
#include <set>
#include <vector>
#include <algorithm>

template <class C, class V>
void find_and_print_rest(const C& container, const V& value)
{
  auto result_it = std::find(container.begin(), container.end(), value);
  if (result_it != container.end())
    std::cout << "result: " << *result_it << std::endl;
  std::cout << "rest: ";
  for (; result_it != container.end(); ++result_it)
    std::cout << *result_it << " ";
  std::cout << "\n" << std::endl;
}

int main()
{
  std::set<int> set = {-1, 1, 2, 1, 3}; // {-1, 1, 2, 3}
  find_and_print_rest(set, 1);
}
```

std::find

find.cc

```
// ...

int main()
{
  std::set<int> set = {-1, 1, 2, 1, 3}; // {-1, 1, 2, 3}
  find_and_print_rest(set, 1);

  std::vector<int> vec = {-1, 1, 2, 1, 3};
  find_and_print_rest(vec, 1);
}
```

multithreading

threads.cc

```
#include <iostream>
#include <thread>

void do_lot_of_work(double size)
{
  std::cout << "starting " << size << "... " << std::endl;
  double tmp;
  for (double i = 0; i < size; ++i)
    tmp += i - i;
  std::cout << "... stopped " << size << std::endl;
}

int main()
{
  std::thread thread_1(do_lot_of_work, 1000000000.0);
  std::thread thread_2(do_lot_of_work, 200000000.0);
  std::thread thread_3(do_lot_of_work, 300000.0);
  std::thread thread_4(do_lot_of_work, 4000000000.0);

  thread_1.join();
  thread_2.join();
  thread_3.join();
  thread_4.join();
}
```

Compile with 'g++ -std=c++11 -Wall -g -pthread -o threads threads.cc'

Zusammenfassung dieser Woche

Zusammenfassung

- C++ ist eine stark typisierte Sprache, Programme sind i.d. Regel sehr effizient.
- Eigene Typen können über Klassen hinzugefügt werden, mathematische Konzepte direkt abgebildet werden.
- Es gibt eine große Menge an Hilfsmitteln und Funktionalität in der Standardbibliothek.

Zusammenfassung

- C++ ist eine stark typisierte Sprache, Programme sind i.d. Regel sehr effizient.
- Eigene Typen können über Klassen hinzugefügt werden, mathematische Konzepte direkt abgebildet werden.
- Es gibt eine große Menge an Hilfsmitteln und Funktionalität in der Standardbibliothek.

Wenn Sie Mathematik und Programmieren verbinden möchten:

$\Rightarrow$ herzlich Willkommen in der Numerischen Mathematik!