Modernes C++

Alexander Heinrich

20. September 2019

Features

- Abwärtskompatibel
  - 20 Jahre alte Klassen können noch benutzt werden
  - C Bibliotheken können direkt benutzt werden
  - Wird in 20 Jahren noch hier sein
  - Alles in diesem Vortrag gezeigte *kann* in alten Code-Bases angewandt werden
- Reiche Auswahl an Bibliotheken, Tooling und Erfahrung
- Moderne Sprache mit modernen Features
- Abstraktionen die nichts kosten (*Laufzeit*)

Toolchain Support

OpenWrt, Mitte 2015	GCC 5.2	C++14
Yocto/OpenEmbedded, Mitte 2015	GCC 5.2	C++14
ESP32 Toolchain, Anfang 2017	GCC 5.2	C++14
Ubuntu 16.04	GCC 5.3	C++14
	Clang 6[1]	C++17
Ubuntu 18.04	GCC 8.3.0	C++17

- Nur in Ausnahmefällen werden bestimmte Features der Standard-Bibliothek nicht unterstützt[2]
- Aktuellere Compiler haben deutlich bessere Fehlermeldungen

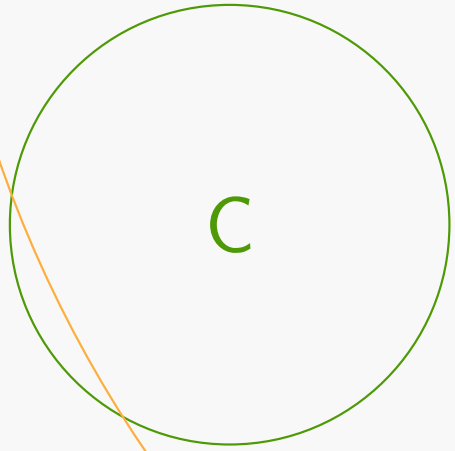
[1] Release, März 2018

[2] `https://en.cppreference.com/w/cpp/compiler_support`

Mengendiagramm C/C++

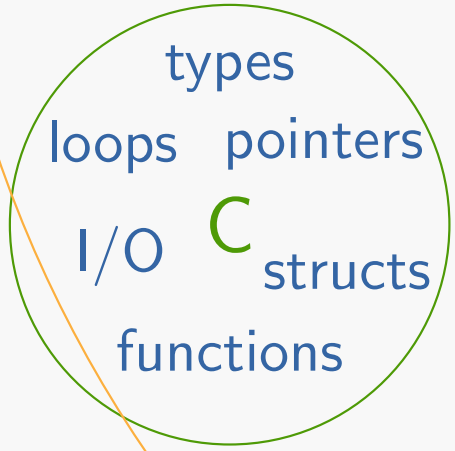
Mengendiagramm C/C++

Mengendiagramm C/C++

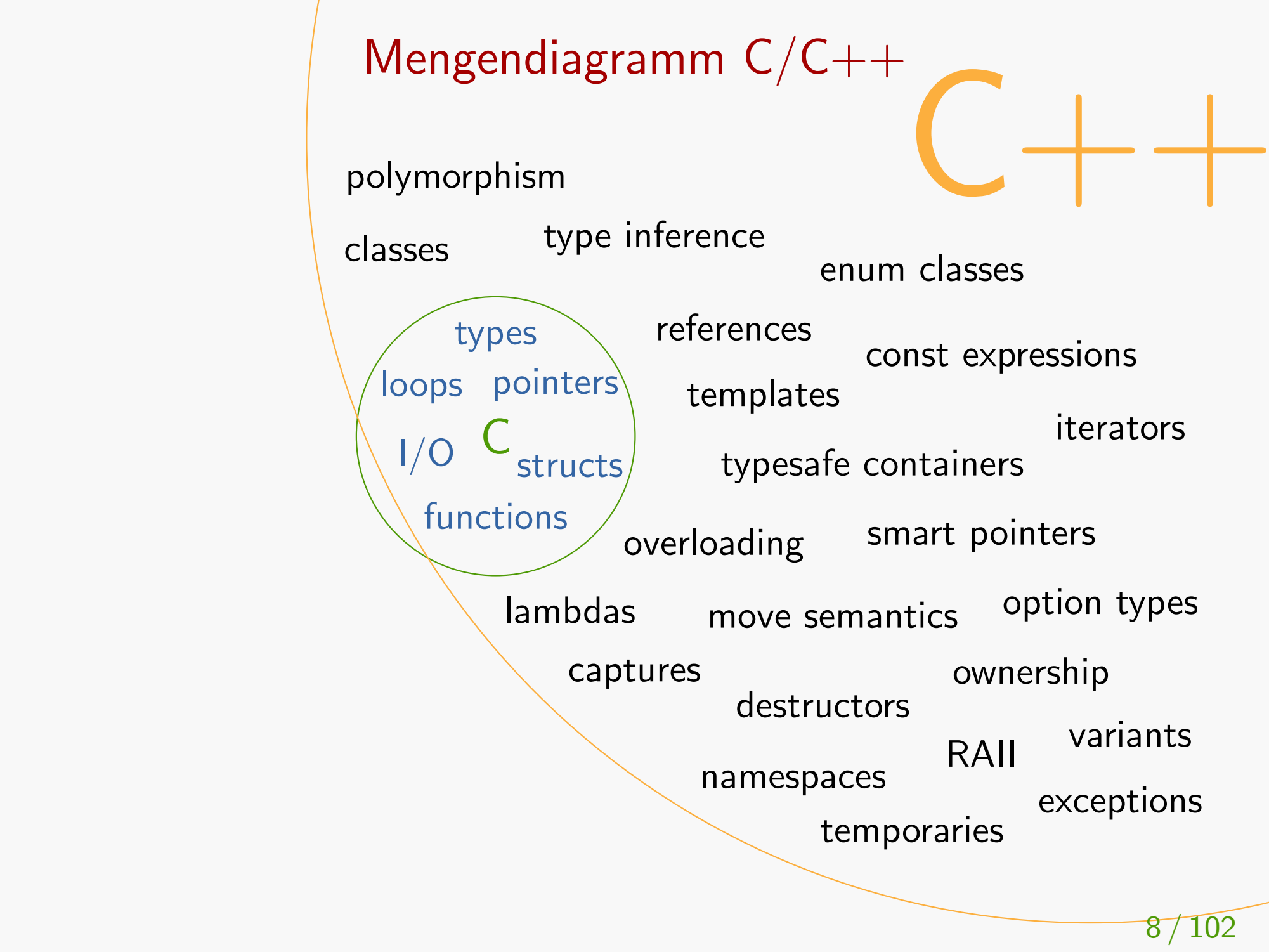


Mengendiagramm C/C++

C++



Mengendiagramm C/C++



Initialisierung

- Zu viele Arten ein Objekt zu initialisieren[1] [2]
- In manchen Fällen uninitialisierter/undefinierter Wert
- Verhalten unterscheidet sich zwischen Objekten, Primitiven Typen, Structs mit und ohne Konstruktor etc.
- Unterschiede zwischen statischen und lokalen Variablen
- Value-, Direct-, Copy-, List-, Aggregate- und Reference-Initialisation

[1] https://blog.tartanllama.xyz/initialization-is-bonkers

[2] http://mikelui.io/2019/01/03/seriously-bonkers.html

Uniform Initialization

- Seit C++11
- Vereinfachte Regeln
- Syntax: geschweifte Klammern {}

Uniform Initialization

```cpp
int x{};       /* Value: 0 */
int x{5};      /* Value: 5 */
int x{5.3};    /* Compilation error, narrowing */
float x{5.3}; /* Value: 5.3 */
bool result{true}; /* Value: true */

std::string name{"Bob"}; /* Pass "Bob" to constructor */

std::vector<int> v{}; /* Empty vector */
std::vector<int> v{1, 2, 3}; /* Contains 1, 2 and 3 */
```

Uniform Initialization

```
struct Color { float r, g, b; };

Color black{}; /* All zero */
Color green{0.0, 1.0, 0.0};

Color getBlue() {
  return {0.0, 0.0, 1.0};
}
```

Uniform Initialization

```
/* Struct with default values */
struct Entry {
  int id{-1};
  std::string location{"Siegen"};
};

Entry siegen{};
Entry berlin{12, "Berlin"};
```

Uniform Initialization

```
class Foo {
  public:
  /* Set "id" to random value, copy given name
     to member variable, "age" will be set to 21. */
  Foo(const std::string &name): id{random()}, name{name}
  {
    /* Empty */
  }

  private:
  int id{-1};
  int age{21};
  std::string name{"Bob"};
};

Foo object{"My Name"};
```

Typinferenz

```
std::vector<std::string> getAllNames() {
   ...
}
const std::vector<std::string> names = getAllNames();
std::vector<std::string>::iterator start = names.begin();
std::vector<std::string>::iterator end = names.end();
```

Typinferenz

```
std::vector<std::string> getAllNames() {
   ...
}
const auto names = getAllNames();
auto start = names.begin();
auto end = names.end();
```

Foreach

```
std::vector<std::string> getAllNames() {
  ...
}
const std::vector<std::string> names = getAllNames();

for(std::vector<std::string>::iterator it = names.begin();
    it != names.end(); it++)
{
  std::cout << *it << std::endl;
}
```

Foreach

```
std::vector<std::string> getAllNames() {
  ...
}
for(const auto &name: getAllNames()) {
  std::cout << name << std::endl;
}
```

Foreach

```
std::unordered_map<std::string, std::string>
mapNamesToLocations(const std::vector<string> &names) {
  ...
}

const auto map = mapNamesToLocations(getAllNames());
/* C++11 */
for(const auto &mapping: map) {
  const auto &name = mapping.first;
  const auto &location = mapping.second;

  std::cout << name << " is in " << location << std::endl;
}
```

Foreach

```
std::unordered_map<std::string, std::string>
mapNamesToLocations(const std::vector<string> &names) {
  ...
}

const auto map = mapNamesToLocations(getAllNames());
/* C++17 */
for(const auto &[name, location]: map) {
  std::cout << name << " is in " << location << std::endl;
}
```

Features in Standard-Bibliothek

Zeit in Standard C

```c
time_t yesterday = time(NULL);

/* Wait one day */

time_t today = time(NULL);

time_t time_passed = today - yesterday;
```

Zeit in Standard C

```c
time_t yesterday = time(NULL);

/* Wait one day */

time_t today = time(NULL);

time_t time_passed = today - yesterday;

time_t surprise = today + yesterday;
```

std::chrono

```
auto yesterday = std::chrono::system_clock::now();

/* Wait one day */

auto today = std::chrono::system_clock::now();

auto time_passed = today - yesterday;
```

std::chrono

```
auto yesterday = std::chrono::system_clock::now();

/* Wait one day */

auto today = std::chrono::system_clock::now();

auto time_passed = today - yesterday;

auto surprise = today + yesterday; /* Compilation error */
```

std::chrono

```
using namespace std::chrono_literals;

std::this_thread::sleep_for(2h + 34min + 9s);
```

std::chrono

```
using namespace std::chrono_literals;

std::this_thread::sleep_for(2h + 34min + 9s);

void useTime(std::chrono::milliseconds time) { ... }

useTime(1min + 5s); /* Compiles down to 65000 */
```

std::chrono

```
using namespace std::chrono_literals;

std::this_thread::sleep_for(2h + 34min + 9s);

void useTime(std::chrono::milliseconds time) { ... }

useTime(1min + 5s); /* Compiles down to 65000 */

auto today = std::chrono::system_clock::now();
auto yesterday = today - 24h;
```

std::chrono

```
using namespace std::chrono_literals;

std::this_thread::sleep_for(2h + 34min + 9s);

void useTime(std::chrono::milliseconds time) { ... }

useTime(1min + 5s); /* Compiles down to 65000 */

auto today = std::chrono::system_clock::now();
auto yesterday = today - 24h;

auto surprise = 24h - today; /* Compilation error */
```

std::array

```
/* Old C array */
int primes[5] = {2, 3, 5, 7, 11};
```

std::array

```
/* Old C array */
int primes[5] = {2, 3, 5, 7, 11};

/* C++ array, compiles down to old C array */
std::array<int, 5> primes{2, 3, 5, 7, 11};
```

std::array

```
/* Old C array */
int primes[5] = {2, 3, 5, 7, 11};

/* C++ array, compiles down to old C array */
std::array<int, 5> primes{2, 3, 5, 7, 11};

/* Can be used like any other container */
std::sort(primes.begin(), primes.end());

for(int number: primes) {
  std::cout << number << std::endl;
}
```

std::array

```
/* Old C array */
int primes[5] = {2, 3, 5, 7, 11};

/* C++ array, compiles down to old C array */
std::array<int, 5> primes{2, 3, 5, 7, 11};

/* Can be used like any other container */
std::sort(primes.begin(), primes.end());

for(int number: primes) {
  std::cout << number << std::endl;
}

/* size() member function compiles down to constant */
std::cout << "Size: " << primes.size() << std::endl;
```

Ressourcenverwaltung

Ressourcenverwaltung

- Manuell via `malloc()/free()`, `fopen()/fclose()` oder `new/delete`
- Garbage Collection
- Ownership: Alles muss einen Besitzer haben

Manuelle Ressourcenverwaltung

```
{
  /* Pointer dies with scope, object lives forever */
  FILE *file = fopen("test.txt", "wb");

  if(someCondition()) {
    return; /* File will leak */
  }

  fooBar(); /* May throw an exception and leak file */

  fclose(file); /* Can be forgotten or called twice
                   and will still compile */
}
```

Garbage Collection

```
{
  File file = FileFactoryManagerFactory.Open("test.txt");

  if(someCondition()) {
    return;
  }

  fooBar(); /* May throw */
}
```

Garbage Collection

```
{
  File file = FileFactoryManagerFactory.Open("test.txt");

  if(someCondition()) {
    return;
  }

  fooBar(); /* May throw */
}
```

- Garbage Collector räumt irgendwann mal auf
- Resource bleibt bis dahin belegt
- Schwierig bei externen Ressourcen (OpenGL: Daten in GPU)
- Finalizer sind nicht immer deterministisch

Garbage Collection

```
{
  File file = FileFactoryManagerFactory.Open("test.txt");

  if(someCondition()) {
    file.close(); /* Deterministic, but manual */
    return;
  }

  try { /* Enterprise Resource Management */
    fooBar();
  } catch(Exception e) {
    ...
  } finally {
    file.close(); /* May throw */
  }
}
```

Garbage Collection

```
{
  using(file = FileFactoryManagerFactory.Open("test.txt"))
  {
    if(someCondition()) {
      return;
    }

    fooBar();
  } /* File gets released */
}
```

Garbage Collection

```
{
  using(file = FileFactoryManagerFactory.Open("test.txt"))
  {
    if(someCondition()) {
      return;
    }

    fooBar();
  } /* File gets released */
}
```

- Wird schnell komplex → Schachtelung
- Muss explizit verwendet werden
- Nur für kurzlebige Ressourcen

Garbage Collection

```
{
  File file = FileFactoryManagerFactory.Open("test.txt");
  defer: file.close(); /* Call this when leaving scope */

  if(someCondition()) {
    return;
  }

  fooBar();
} /* Calls file.close() */
```

Garbage Collection

```
{
  File file = FileFactoryManagerFactory.Open("test.txt");
  defer: file.close(); /* Call this when leaving scope */

  if(someCondition()) {
    return;
  }

  fooBar();
} /* Calls file.close() */
```

- Kann vergessen, oder doppelt registriert werden
- Muss überall explizit angegeben werden

Manuelle Ressourcenverwaltung

```
{
  /* Pointer dies with scope, object lives forever */
  FILE *file = fopen("test.txt", "wb");

  if(someCondition()) {
    return; /* File will leak */
  }

  fooBar(); /* May throw an exception and leak file */

  fclose(file); /* Can be forgotten or called twice
                   and will still compile */
}
```

Ownership

```
{
  std::vector<std::string> names{};

  /* Allocate strings and grow vector dynamically */
  names.push_back("Bob");
  names.push_back("Foo");
  names.push_back("super long string");
  names.push_back("Linux");
  ...

} /* Vector and strings get released here */
```

Destruktor

```
class Handle {
  public:
  /* Constructor */
  Handle(): file_descriptor{open_fd()} {}

  /* Destructor */
  ~Handle() {
    close(file_descriptor);
  }

  private:
  int file_descriptor{-1};
};
```

Destruktor

```
{
  Handle my_handle{}; /* Opens a file descriptor */
  ...


} /* Destructor is guaranteed to be called here */
```

Destruktor

```
{
  Handle my_handle{}; /* Opens a file descriptor */
  ...

  throwException();

} /* Destructor will still be called */
```

Destruktor

```
std::mutex my_mutex{};
...

{
  std::lock_guard<std::mutex> lock{my_mutex};
  ...

  throwException();

} /* Guard gets destroyed and releases mutex */
```

RAII

- Resource Acquisition Is Initialization
- Idee:
  - Externe Ressourcen haben eine undefinierte Lebenszeit
  - Objekte auf dem Stack haben eine definierte Lebenszeit
  - Ressourcen werden an Objekte gebunden
- Objekte „besitzen“ eine Ressource $\rightarrow$ Ownership

Destruktor

```
class Foo {
  public:
  Foo(const std::string &name): name{name}
  {
    numbers.push_back(12);
  }

  ~Foo() {
    /* No need to release "name" and "numbers"
       manually, they already have destructors. */
  }

  private:
  std::string name{};
  std::vector<int> numbers{};
};
```

Destruktor

```
class Foo {
  public:
  Foo(const std::string &name): name{name}
  {
    numbers.push_back(12);
  }


  /* No explicit destructor needed */



  private:
  std::string name{};
  std::vector<int> numbers{};
};
```

Ressourcenverwaltung

```
{
  std::string name{"Bob"};

  std::string foo{name};  /* Copy "Bob" to own memory */
  std::string bar = name; /* Copy "Bob" to own memory */

  name = "Lorem Ipsum"; /* Doesn't affect copies */

} /* All strings release their memory */
```

Ressourcenverwaltung

```
class Handle {
  public:
  Handle(): file_descriptor{open_fd()} {}

  ~Handle() {
    close(file_descriptor);
  }

  private:
  int file_descriptor{-1}; /* Copied by value */
};
```

Ressourcenverwaltung

```
{
  Handle my_handle{}; /* Opens a file descriptor */

  Handle h{my_handle}; /* Copies file descriptor (int) */


} /* File descriptor gets closed twice */
```

Ressourcenverwaltung

```
class Handle {
  public:
  Handle(): file_descriptor{open_fd()} {}
  ~Handle() { close(file_descriptor); }

  private:
  int file_descriptor{-1}; /* Copied by value */
};
```

Ressourcenverwaltung

```
class Handle {
  public:
  Handle(): file_descriptor{open_fd()} {}
  ~Handle() { close(file_descriptor); }

  /* Copy constructor */
  Handle(const Handle &other):
    file_descriptor{dup(other.file_descriptor)} {}

  /* Copy assignment */
  Handle& operator=(const Handle &other) { /* ... */ }

  private:
  int file_descriptor{-1}; /* Duplicated */
};
```

Ressourcenverwaltung

```
class Foo {
  public:
  Foo(const std::string &name): name{name} {}

  /* Copy members using their copy constructor */
  Foo(const Foo &other):
    name{other.name}, numbers{other.numbers} {}
  Foo& operator=(const Foo &other) { /* ... */ }

  private:
  std::string name{};
  std::vector<int> numbers{10, 11, 12};
};
```

Ressourcenverwaltung

```
class Foo {
  public:
  Foo(const std::string &name): name{name} {}


  /* If all members can be copied, an explicit copy
     constructor is not needed */


  private:
  std::string name{};
  std::vector<int> numbers{10, 11, 12};
};
```

Ressourcenverwaltung

```
class Connection {
  public:
  Connection(): socket_fd{accept_client()} {}
  ~Connection() { close(socket_fd); }

  /* How to copy a Connection/Session? Open a
     second connection to the same client? */


  private:
  int socket_fd{-1};
};
```

Ressourcenverwaltung

```
class Connection {
  public:
  Connection(): socket_fd{accept_client()} {}
  ~Connection() { close(socket_fd); }

  /* Disable copying */
  Connection(const Connection &) = delete;
  Connection& operator=(const Connection &) = delete;

  private:
  int socket_fd{-1};
};
```

Ressourcenverwaltung

```
Connection client{};

Connection other{client}; /* Compilation error */
```

Ressourcenverwaltung

```
Connection client{};


void sendHello(Connection c) {
   ...
}

/* Pass by value -> Copy -> Compilation error */
sendHello(client);
```

Ressourcenverwaltung

```
Connection client{};


void sendHello(Connection &c) {
   ...
}

/* Pass by reference -> Ok */
sendHello(client);
```

Ressourcenverwaltung

```
class Foo {
  public:
  Foo(const std::string &name): name{name} {}

  /* Copying is implicitly disabled if a member can't
     be copied. When required, it must be implemented
     explicitly. */

  private:
  std::string name{};
  std::vector<int> numbers{10, 11, 12};
  Connection client{}; /* Can't be copied */
};
```

Ownership

```
std::vector<Handle*> v{};

{
  Handle handle{};

  v.push_back(&handle);


} /* Handle gets destroyed */

/* Vector contains dangling pointer */
```

Ownership

```
std::vector<Handle> v{};

{
  Handle handle{};

  v.push_back(handle); /* Pass by value -> Copy */


} /* Handle gets destroyed */

/* Vector owns copy */
```

Ownership

```
std::vector<std::thread> v{};

{
  std::thread my_thread{[]{ while(true); }};

  v.push_back(my_thread); /* Can't be copied,
                             Compilation error */

}
```

Ownership

```
std::vector<std::thread> v{};

{
  std::thread my_thread{[]{ while(true); }};

  v.push_back(std::move(my_thread)); /* No copy */

  /* Vector owns thread now */
}
```

Ownership

```
std::vector<std::string> v{};

{
  std::string huge{"Lorem ipsum dolor sit amet, consectetur

  v.push_back(huge); /* Copy is expensive */


} /* Huge string gets released */

/* Vector contains huge copy */
```

Ownership

```
std::vector<std::string> v{};

{
  std::string huge{"Lorem ipsum dolor sit amet, consectetur

  v.push_back(std::move(huge));
  /* huge is "" now */

} /* Empty string gets released */

/* Vector owns huge string */
```

Ownership

- Ressourcen wechseln den Besitzer $\rightarrow$ Ownership
- Nachteil: Klassen müssen Move-Konstruktor implementieren
  - Ausnahme: Alle Member haben einen Move-Konstruktor
- Nicht alle Klassen implementieren einen Move-Konstruktor
  - z.B. ältere Bibliotheken
- Manche Klassen markieren diesen explizit als „deleted“
- Ist eine Member-Variable nicht verschiebbar, wird die ganze Klasse implizit nicht verschiebbar

Ownership

```
std::vector<Connection> v{};

{
  Connection client{};

  v.push_back(std::move(client)); /* Can't be moved */
  v.push_back(client); /* Can't be copied */

}
```

Ownership

```
std::vector<std::unique_ptr<Connection>> v{};

{
  auto client = std::make_unique<Connection>();

  v.push_back(std::move(client)); /* Ok */
  /* client is nullptr here */

} /* Vector owns connection */
```

Ownership

```
void Queue::add(std::unique_ptr<Handle> handle) {
  ...
}

{
  auto handle = std::make_unique<Handle>();

  Queue::add(std::move(handle));
  /* handle is nullptr here, can't be used */

}
```

Ownership

```
void Queue::add(std::shared_ptr<Handle> handle) {
  ...
}

{
  auto handle = std::make_shared<Handle>();

  Queue::add(handle); /* Copy ptr, increment refcount */
  sendToThread(handle); /* Copy ptr, increment refcount */

} /* Destructor decrements refcount */
```

Ownership

- Code und Architektur müssen Ownership berücksichtigen
- Die *meisten* Objekte wechseln ihren Besitzer nicht und sterben mit dem Scope → Stack oder Member-Variablen
- Viele Klassen implementieren bereits eigenes Copy und Move Verhalten
- Wenn nicht, oder wenn das nicht gewünscht ist, dann:
  `std::unique_ptr` → exakt ein Besitzer, `move-only`
  `std::shared_ptr` → mehrere Besitzer, `copyable`
  `std::weak_ptr` → bei zyklischen Referenzen
- Zeiger, `new` und `delete` für Sonderfälle oder C Anbindung

Pass by Value

Wenn ein kopierbares Objektes kopiert werden soll

```
void process(Foo foo) {
  foo.doSomeStuff(); /* Ok */
  someFunction(foo); /* Ok */


}

Foo foo{}
process(foo);
```

Move

Wenn ein verschiebbares Objekt verschoben werden soll

```
void process(Foo foo) {
  foo.doSomeStuff(); /* Ok */
  someFunction(foo); /* Ok */


}

Foo foo{}
process(std::move(foo));
```

Pass by Reference

Wenn ein Objekt nur benutzt, aber nicht behalten werden soll

```
void process(Foo &foo) {
  foo.doSomeStuff(); /* Ok */
  someFunction(foo); /* Ok */
  global_ptr = &foo; /* Bad, ptr may outlive foo */
  keep_ptr(&foo);    /* Bad, ptr may outlive foo */
}

Foo foo{};
process(foo);
```

Pass by Reference

Wenn ein Objekt nur benutzt, aber nicht behalten werden soll

```
void process(Foo &foo) {
  foo.doSomeStuff(); /* Ok */
  someFunction(foo); /* Ok */
  global_ptr = &foo; /* Bad, ptr may outlive foo */
  keep_ptr(&foo);    /* Bad, ptr may outlive foo */
}

auto foo = std::make_unique<Foo>();
process(*foo);
```

std::unique_ptr

Wenn *exklusiver* Besitz ausgedrückt werden soll

```
void process(std::unique_ptr<Foo> foo) {
  foo->doSomeStuff();              /* Ok */
  someFunction(*foo);              /* Ok */
  global_ptr = std::move(foo); /* Ok, moves ownership */
  keep_ptr(std::move(foo));    /* Bad, already moved */
}

auto foo = std::make_unique<Foo>();
process(std::move(foo));
```

std::shared_ptr

Wenn *gemeinsamer* Besitz ausgedrückt werden soll

```
void process(std::shared_ptr<Foo> foo) {
  foo->doSomeStuff(); /* Ok */
  someFunction(*foo); /* Ok */
  global_ptr = foo;   /* Ok, shared ownership */
  keep_ptr(foo);      /* Ok, shared ownership */
}

auto foo = std::make_shared<Foo>();
process(foo);
```

Zeiger

Zur Abwärtskompatibilität oder für Sonderfälle

```
{
  auto a = std::make_unique<Handle>();
  auto b = std::make_unique<Handle>();

  c_lib_use_handle(a.get());   /* Pass pointer */

  c_lib_keep_ptr(b.release()); /* Release pointer */

} /* Only a gets destroyed */
```

Rule of Three

- Wenn entweder ein Destruktor, Copy-Konstruktor oder Copy-Assignment implementiert werden, braucht man wahrscheinlich alle Drei
- Alternativ implementiert man den Destruktor und deleted die anderen Beiden[1]

[1] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#define-copy-move-and-destroy-consistently

Rule of Zero

- Wenn alle Member einer Klasse Destruktoren, Copy und Move implementieren, muss sich die Klasse nicht mehr selbst darum kümmern

Rule of Zero

```
class Foo {
  private:
  std::string name{"Bob"};
  std::list<bool> values{};
  std::map<std::string, int> employees{};
};
{
  Foo foo{}; /* Can be copied and moved */

} /* foo releases all its resources here */
```

Rule of Zero

```
class Foo {
  private:
  std::string name{"Bob"};
  std::list<bool> values{};
  std::map<std::string, int> employees{};
};
class Bar { /* Can be copied, moved and destroyed */
  private:
  Foo foo{};
};
```

Lambdas

Lambdas

```
auto print_number = [](int number) {
  std::cout << number << std::endl;
};

print_number(9);
print_number(12);
```

Lambdas

```
auto print_number = [](int number) {
  std::cout << number << std::endl;
};

SQLite3 db{"/path/to/file.db"};

db.forEachId(print_number);
```

Lambdas

- Aufrufbare Funktionsobjekte
- Haben implizite Konstruktoren und Destruktoren
- Kopierbar und verschiebbar wie normale Objekte auch
- Können die Umgebung einschließen (*capture*)

Lambdas

```
{
  SQLite3 db{"/path/to/file.db"};



  auto name = db.findName(); /* Blocks thread */
  std::cout << name << std::endl;



}
```

Lambdas

```
{
  SQLite3 db{"/path/to/file.db"};
  EventLoop loop{};

  /* Runs query in background thread */
  db.findName(loop, [](const std::string &name) {
    std::cout << name << std::endl;
  }); /* Doesn't block */

  loop.run(); /* Runs our callback in this thread */
}
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{



  auto print_number = [](int number) {
    std::cout << number << std::endl;
  };

  runOnNumbers(print_number);


}
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  std::string prefix{"Number: "};

  /* Capture prefix by value -> Lambda owns a copy */
  auto print_number = [prefix](int number) {
    std::cout << prefix << number << std::endl;
  };

  runOnNumbers(print_number); /* Pass by value. Copies
                                lambda and its capture */

} /* print_number gets released here */
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  std::string prefix{"Number: "};

  /* Capture prefix by ref -> Lambda keeps reference */
  auto print_number = [&prefix](int number) {
    std::cout << prefix << number << std::endl;
  };

  runOnNumbers(print_number); /* Pass by value. Copies
                                lambda and its ref */

} /* print_number gets released here */
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  auto db = std::make_unique<SQLite3>("/path/to/file.db");

  /* Capture db ptr by value -> Compilation error */
  auto add = [db](int number) {
    db->addNumber(number);
  };

  runOnNumbers(add);


}
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  auto db = std::make_unique<SQLite3>("/path/to/file.db");

  /* Capture db ptr by ref -> Lambda keeps reference */
  auto add = [&db](int number) {
    db->addNumber(number);
  };

  runOnNumbers(add); /* Pass by value. Copies lambda
                        and its db ptr reference */

}
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  auto db = std::make_unique<SQLite3>("/path/to/file.db");

  /* Capture db by move -> Lambda owns db */
  auto add = [db{std::move(db)}](int number) {
    db->addNumber(number);
  };

  runOnNumbers(add); /* Can't be copied, but std::function
                        requires copyable lambda. Error */

}
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  auto db = std::make_shared<SQLite3>("/path/to/file.db");

  /* Capture db ptr by value -> Lambda owns shared ptr */
  auto add = [db](int number) {
    db->addNumber(number);
  };

  runOnNumbers(add); /* Copies lambda with shared ptr */


} /* db ptr, "add" and its shared ptr die here */
```

Lambdas

```
void runOnNumbers(std::function<void(int)> fun) { ... }

{
  auto db = std::make_shared<SQLite3>("/path/to/file.db");

  /* Capture db as weak ptr */
  auto add = [db_ptr{std::weak_ptr{db}}](int number) {
    if(auto db = db_ptr.lock()) /* Check if still alive */
      db->addNumber(number);
  };
  runOnNumbers(add); /* Copies lambda with weak ptr */


} /* db ptr, "add" and its weak ptr die here */
```