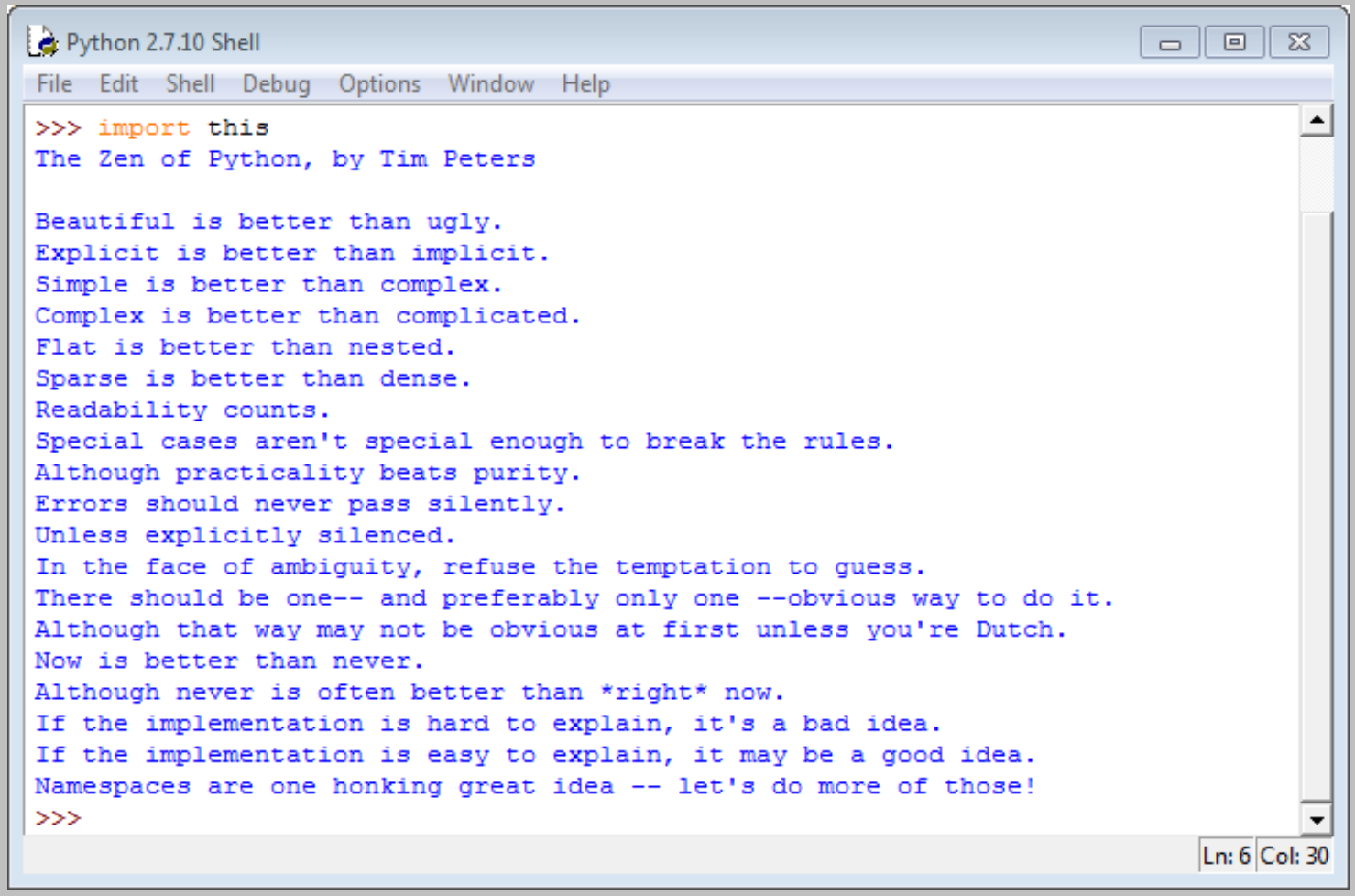
Advanced Developers Conference
Development for Professionals!

15 Tipps

oder warum es nur 10 wurden

Rainer Grimm
Schulungen, Coaching und Technologieberatung

The Zen of Python: Tim Peters



The Zen of ...

- Disclaimer:
  - Ich warte noch auf wertvolle Tipps.
  - Vielleicht sind 10 Tipps schon ausreichend in C++.
  - 10 ist eine sehr gute Zahl.
  - The Zen of C++ ist unvollendet.

→ The Zen of ...

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

The Zen of ...

- **Vermeide implizite Typkonvertierungen.**
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

Vereinheitlichte Initialisierung mit { }

{ } – Initialisierung verhindert Verengung (*narrowing*).

➡ Heimlicher Verlust der Datengenauigkeit.

```
int i1(3.14);                    // OK
int i2{3.14};                    // ERROR
int i3= {3.14};                  // ERROR

char c1(999};                    // OK
char c2{999};                    // ERROR

char c3{8};                      // OK
```

`nullptr` statt `0` oder `NULL`

`nullptr` ist ein richtiger Zeiger

- verweist auf kein Datum und lässt sich nicht dereferenzieren
- kann mit allen Zeigern verglichen und in alle Zeiger impliziert konvertiert werden
- kann nur in einen Wahrheitswert konvertiert werden

- Räumt mit der Mehrdeutigkeit der Zahl 0 und dem Makro NULL auf.
  - **0**: wird entweder als Nullzeiger (`(void*)0`) oder die natürliche Zahl 0 interpretiert.
  - **NULL**: lässt sich in der Regel in einen integralen Typ konvertieren.

nullptr statt 0 oder NULL

```
std::string overloadTest(char*){ return "char*";}
std::string overloadTest(int){ return "int"; }

...

int* pi = nullptr;                    // OK
// int i= nullptr;                    // cannot convert std::nullptr_t to int
bool b = nullptr;                     // OK. b is false.

std::cout << std::boolalpha << "b: "  << b;                          // false
std::cout << "overloadTest(0)= " <<  overloadTest(0);                // int

std::cout << "overloadTest(nullptr)= " <<  overloadTest(nullptr);    // char*

std::cout << "overloadTest(NULL)= " << overloadTest(NULL);           // ERROR
```

Aufzählungen mit Gültigkeitsbereich

- Aufzählungen mit Gültigkeitsbereich werden auch streng typisierte Aufzählungstypen genannt.

```
enum class StrongColor{ red, blue, green };
```

- Regeln
  - Lassen sich nur im Gültigkeitsbereich der Aufzählung ansprechen.
  - Konvertieren nicht implizit zu `int`.
  - Verschmutzen nicht den globalen Namensbereich.
  - Der zugrunde liegende Typ ist per Default `int`, kann aber explizt angegeben werden.

```
enum class StrongColor: char{ red, blue, green };
```

➡ Können vorwärts deklariert werden

Aufzählungen mit Gültigkeitsbereich

```
enum class Color1{
  red,
  blue,
  green
};
enum struct Color2: char{
  red= 100,
  blue, // 101
  green // 102
};
...
std::cout << static_cast<int>(Color1::red) << std::endl;    // 0
std::cout << static_cast<int>(Color2::red) << std::endl;    // 100

std::cout << "sizeof(Color1)= "  << sizeof(Color1);         // 4
std::cout << "sizeof(Color2)= "  << sizeof(Color2);         // 1
```

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- **Programmiere deklarativ.**
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

Methoden anfordern

- Fordere spezielle Methoden und Operatoren vom Compiler an.
  - Beispiele: Standard-, Kopierkonstruktor und Destruktor; Zuweisungsoperator, `operator new`

```
class MyType{
  public:
    MyType(int val) {}
    MyType()= default;
    virtual ~MyType()= default;
    explicit MyType(const MyType&)= default;
};
```

Funktionsaufrufe unterdrücken

- Eine nicht kopierbare Klasse

```
class NonCopyClass{
  public:
    NonCopyClass()= default;
    NonCopyClass& operator =(const NonCopyClass&)= delete;
    NonCopyClass (const NonCopyClass&)= delete;
};
```

- Eine Funktion, die nur `double` annimmt

```
void onlyDouble(double){}
template <typename T> void onlyDouble(T)= delete;
int main(){
  onlyDouble(3);
};
```

➡ `Error: use of deleted function »void onlyDouble(T) [mit T = int]«`

Explizites überschreiben

Optionale Kontrolle durch den Compiler

```
class Base {
  void func1();
  virtual void func2(float);
  virtual void func3() const;
  virtual long func4(int);
};

class Derived: public Base {
  virtual void func1() override;                  // ERROR
  virtual void func2(double) override;            // ERROR
  virtual void func3() override;                  // ERROR
  virtual int func4(int) override;                // ERROR
  virtual long func4(int) override;               // OK
};
```

Überschreiben unterbinden

- Methoden

```
class Base {
  virtual void h(int) final;
};
class Derived: public Base {
  virtual void h(int);                    // ERROR
  virtual void h(double);                 // OK
};
```

- Klassen

```
struct Base final{};
struct Derived: Base{};                   // ERROR
```

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- **Unterstütze automatische Optimierungen.**
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

Copy versus Move: std::swap

```
std::vector<int> a, b;
swap(a,b);

template <typename T>
void swap(T& a, T& b){
  T tmp(a);
  a= b;
  b= tmp;
}
template <typename T>
void swap(T& a, T& b){
  T tmp(std::move(a));
  a= std::move(b);
  b= std::move(tmp);
}
```

`T tmp(a);`

- Allokiert `tmp` und jedes Element von `tmp`.
- Kopiert jedes Element von `a` nach `tmp`.
- Deallokiert `tmp` und jedes Element von `tmp`.

`T tmp(std::move(a));`

- Verbiegt den Zeiger von `tmp` auf `a`.

Copy als Fallback für Move

```
template <typename T>
void swap(T& a, T& b){
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
struct MyData{
  std::vector<int> myData;
  MyData():myData({1,2,3,4,5}){}
  MyData(const MyData& m):myData(m.myData){ ... }
  MyData& operator=(const MyData& m){ ... }
};
...
MyData a,b;
swap(a,b);
```

➡ Programmieren Sie für die zukünftige Optimierung.

constexpr

`constexpr` Funktionen können

- zur Compilezeit ausgewertet werden, wenn sie mit konstanten Ausdrücken aufgerufen werden.
  - sind implizit threadsicher
  - geben dem Compiler tiefen Einblick in den Code
- zur Laufzeit mit nicht konstanten Ausdrücken aufgerufen werden.
- in C++14 keine statische oder thread lokale Variablen enthalten.

`constexpr` Funktionen können zur Compilezeit und Laufzeit ausgewertet werden.

Deklarieren Sie ihre Funktionen als `constexpr`.

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- **Sei nicht schlauer als der Compiler.**
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

auto: Refaktorierung

```
auto a= 5;
auto b= 10;
auto sum=  a * b * 3;
auto res= sum + 10;
std::cout << typeid(res).name();      // i

auto a2= 5;
auto b2= 10.5;
auto sum2= a2 * b2 * 3;
auto res2= sum2 * 10;
std::cout << typeid(res2).name();    // d

auto a3= 5;
auto b3= 10;
auto sum3= a3 * b3 * 3.1f;
auto res3= sum3 * 10;
std::cout << typeid(res3).name();   // f
```

Smart Pointer: unique_ptr

```
struct MyInt{
  MyInt(int i):i_(i){}
  ~MyInt(){
    std::cout << "Bye from " << i_ << std::endl;
  }
  int i_;
};

{
  std::unique_ptr<MyInt> uniquePtr1{ new MyInt(1998) };
  std::unique_ptr<MyInt> uniquePtr2{ std::move(uniquePtr1) };
  {
    std::unique_ptr<MyInt> localPtr{ new MyInt(2003) };
  }                                                    // Bye from 2003
  uniquePtr2.reset(new MyInt(2011));                   // Bye from 1998
  MyInt* myInt= uniquePtr2.release();
  delete myInt;                                        // Bye from 2011
}
```

Smart Pointer: shared_ptr

```
struct MyInt{
  MyInt(int v):val(v){ std::cout << "  Hello: " << val << std::endl; }
  ~MyInt(){ std::cout << "  Good Bye: " << val << std::endl; }
private:
  int val;
};

{
  std::shared_ptr<MyInt> sharPtr(new MyInt(1998));      // Hello: 1998
  std::cout << sharPtr.use_count();                     // 1
  {
    std::shared_ptr<MyInt> locSharPtr(sharPtr);
    std::cout << locSharPtr.use_count();                // 2
  }
  std::cout << sharPtr.use_count();                     // 1
  std::shared_ptr<MyInt> globSharPtr= sharPtr;
  std::cout << sharPtr.use_count();                     // 2
  globSharPtr.reset();
  std::cout << sharPtr.use_count();                     // 1
  sharPtr= std::shared_ptr<MyInt>(new MyInt(2011));     // Hello: 2011 und Good Bye: 1998
}                                                       // Good Bye 2011
```

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- **Behalte das große Bild im Auge.**
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

Sequenzielle Konsistenz

```
struct MySingleton{
  static MySingleton* getInstance(){
    MySingleton* sin= instance.load();
    if ( !sin ){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin= instance.load();
      if( !sin ){
        sin= new MySingleton();
        instance.store(sin);
      }
    }
    return sin;
  }
private:
  MySingleton()= default;
  ~MySingleton()= default;
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;
  static std::atomic<MySingleton*> instance;
  static std::mutex myMutex;
};
```

Bruch der Sequenzielle Konsistenz

```
struct MySingleton{
  static MySingleton* getInstance(){
    MySingleton* sin= instance.load(std::memory_order_acquire);
    if ( !sin ){
      std::lock_guard<std::mutex> myLock(myMutex);
      sin= instance.load(std::memory_order_relaxed);
      if( !sin ){
        sin= new MySingleton();
        instance.store(sin,std::memory_order_release);
      }
    }
    return sin;
  }
private:
  MySingleton()= default;
  ~MySingleton()= default;
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;
  static std::atomic<MySingleton*> instance;
  static std::mutex myMutex;
};
```

Meyers Singleton

```
struct MySingleton{
  static MySingleton& getInstance(){
    static MySingleton instance;
    return instance;
  }
private:
  MySingleton()= default;
  ~MySingleton()= default;
  MySingleton(const MySingleton&)= delete;
  MySingleton& operator=(const MySingleton&)= delete;
};
```

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- **Vermeide undefiniertes Verhalten.**
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

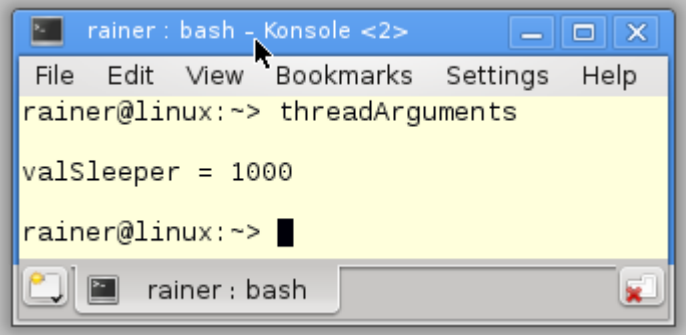
auto-matisch initialisiert

```
struct T1 {};
struct T2{
    int mem;            // ERROR
 public:
    T2() {}
};

int n;                  // OK

int main(){
  int n;                // ERROR
  std::string s;        // OK
  T1 t1;                // OK
  T2 t2;                // OK
}
```

```
struct T1 {};
struct T2{
    int mem= 0;    // auto mem=0; ERROR
 public:
    T2() {}
};

auto n= 0;

int main(){
  auto n= 0;
  auto s= ""s;
  auto t1= T1();
  auto t2= T2();
}
```

Threads *detachen*

```
struct Sleeper{
    Sleeper(int& i_):i{i_}{};
    void operator() (int k){
      for (unsigned int j= 0; j <= 5; ++j){
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        i += k;
      }
      std::cout << std::this_thread::get_id();
    }
  private:
    int& i;
};
int main(){
  int valSleeper= 1000;
  std::thread t(Sleeper(valSleeper),5);
  t.detach();
  std::cout << "valSleeper = " << valSleeper << std::endl;
}
```



The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- **Achte auf die Lesbarkeit des Codes.**
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- Strebe nach Einfachheit.

Lambda-Funktionen

```
std::vector<int> myVec(20);
std::iota(myVec.begin(),myVec.end(),0);

std::function< bool(int)> myBindPred=
   std::bind( std::logical_and<bool>(),
        std::bind( std::greater <int>(),std::placeholders::_1,9 ),
        std::bind( std::less <int>(),std::placeholders::_1,16 ));
myVec.erase(std::remove_if(myVec.begin(),myVec.end(),myBindPred),myVec.end()
);


std::vector<int> myV(20);
std::iota(myV.begin(),myV.end(),0);

auto myLambdaPred= [](int a){return (a>9) && (a<16);};
myV2.erase(std::remove_if(myV2.begin(),myV2.end(),myLambdaPred),myV2.end());
```

Range-basierte for-Schleife

```
int myArray[5] = {1, 2, 3, 4, 5};
for (int& x : myArray) x *= 2;
for (int x: myArray) std::cout << x << " ";       // 2 4 6 8 10

std::vector<int> vecInt({1, 2, 3, 4, 5});
for (int& x: vecInt) x *= 2;
for (int x: vecInt) std::cout << x << " ";       // 2 4 6 8 10

std::string str= {"Only for Testing Purpose."};
for (char& c: str) c=std::toupper(c);
for (char c: str) std::cout << c;                // ONLY FOR TESTING PURPOSE.

str= {"Only for Testing Purpose."};
for (char& c: str) c=std::isupper(c)? std::tolower(c): std::toupper(c);
for (char c: str) std::cout << c;                // oNLY FOR tESTING pURPOSE
```

Range-basierte for-Schleife

```
std::map<std::string,std::string> phonebook{
   {"Bjarne Stroustrup","+1 (212) 555-1212"},
   {"Gabriel Dos Reis", "+1 (858) 555-9734"},
   {"Daveed Vandevoorde","+44 99 74855424"}
}

std::map<std::string,std::string>::iterator mapIt;
for (mapIt= phonebook.begin(); mapIt != phonebook.end(); ++mapIt){
  std::cout << mapIt->first <<  ": " << mapIt->second << std::endl;
}


for ( auto mapIt: phonebook ){
  std::cout << mapIt.first << ": " << mapIt.second << std::endl;
}

   // Bjarne Stroustrup: +1 (212) 555-1212
      Daveed Vandevoorde: +44 99 74855424
      Gabriel Dos Reis:+ 1 (858) 555-9734
```

Automatische Typableitung

```
auto myInts={1,2,3};
std::initializer_list<int> myInts={1,2,3};

auto myIntBegin= myInts.begin();
std::initializer_list<int>::iterator myIntBegin= myInts.begin();

auto func= [](const std::string& a){ return a;};
std::function< std::string(const std::string&) > func=
           [](const std::string& a){ return a;};

auto begin = std::chrono::system_clock::now();
std::chrono::time_point<std::chrono::system_clock> begin =
std::chrono::system_clock::now();
```

The Zen of ...

Vermeide implizite Typkonvertierungen.

Programmiere deklarativ.

Unterstütze automatische Optimierungen.

Sei nicht schlauer als der Compiler.

Behalte das große Bild im Auge.

Vermeide undefiniertes Verhalten.

Achte auf die Lesbarkeit des Codes.

Lasse dir helfen.

Kenne deine Bibliotheken.

Strebe nach Einfachheit.

`static_assert` und Type-Traits

- `static_assert`
  - besitzt keinen Einfluss auf die Laufzeit des Programmes.
  - lässt sich ideal mit der neuen Type-Traits-Bibliothek kombinieren.

    ➡ `static_assert` validiert die Type-Traits Aufrufe.

- Stelle sicher,
  - dass eine 64-bit Architektur vorliegt.

```
static_assert(sizeof(long) >= 8,"no 64-bit code generation");
```

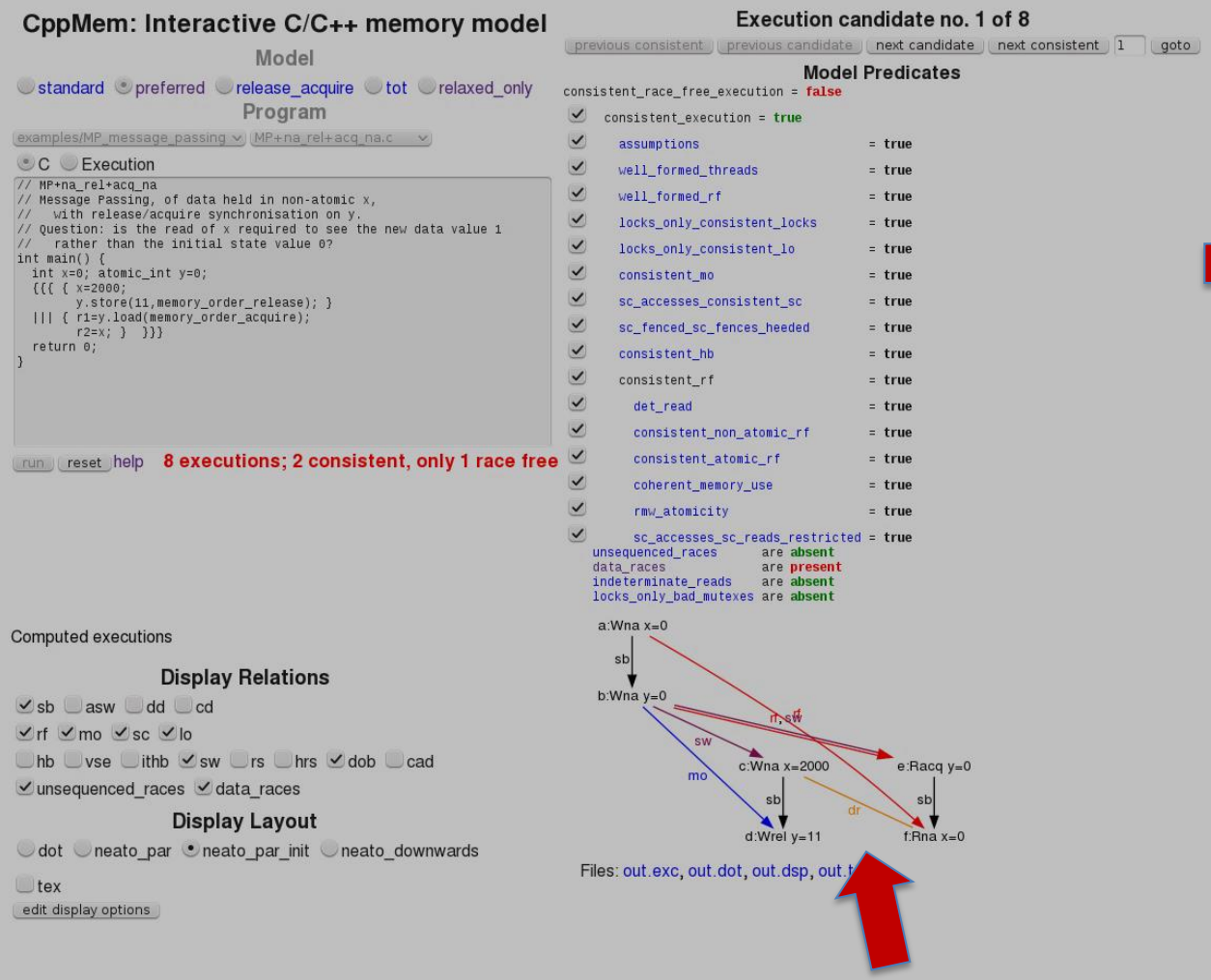
  - dass ein arithmetischer Typ vorliegt.

```
static_assert(is_arithmetic<T>::value, "arg must be arithmetic");
```

static_assert und Type-Traits

```
template<typename T1, typename T2>
typename std::conditional <(sizeof(T1) < sizeof(T2)), T1, T2>::type
gcd(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be an integral!");
  static_assert(std::is_integral<T2>::value, "T2 should be an integral!");
  if( b == 0 ){ return a; }
  else{ return gcd(b, a % b); }
}

...

std::cout << gcd(100,10)  << std::endl;         // 10
std::cout << gcd(100,10LL) << std::endl;        // 10

std::cout << gcd ("100","10") << std::endl;     // ERROR
```

Statische Codeanalyse: CppMem



[CppMem]

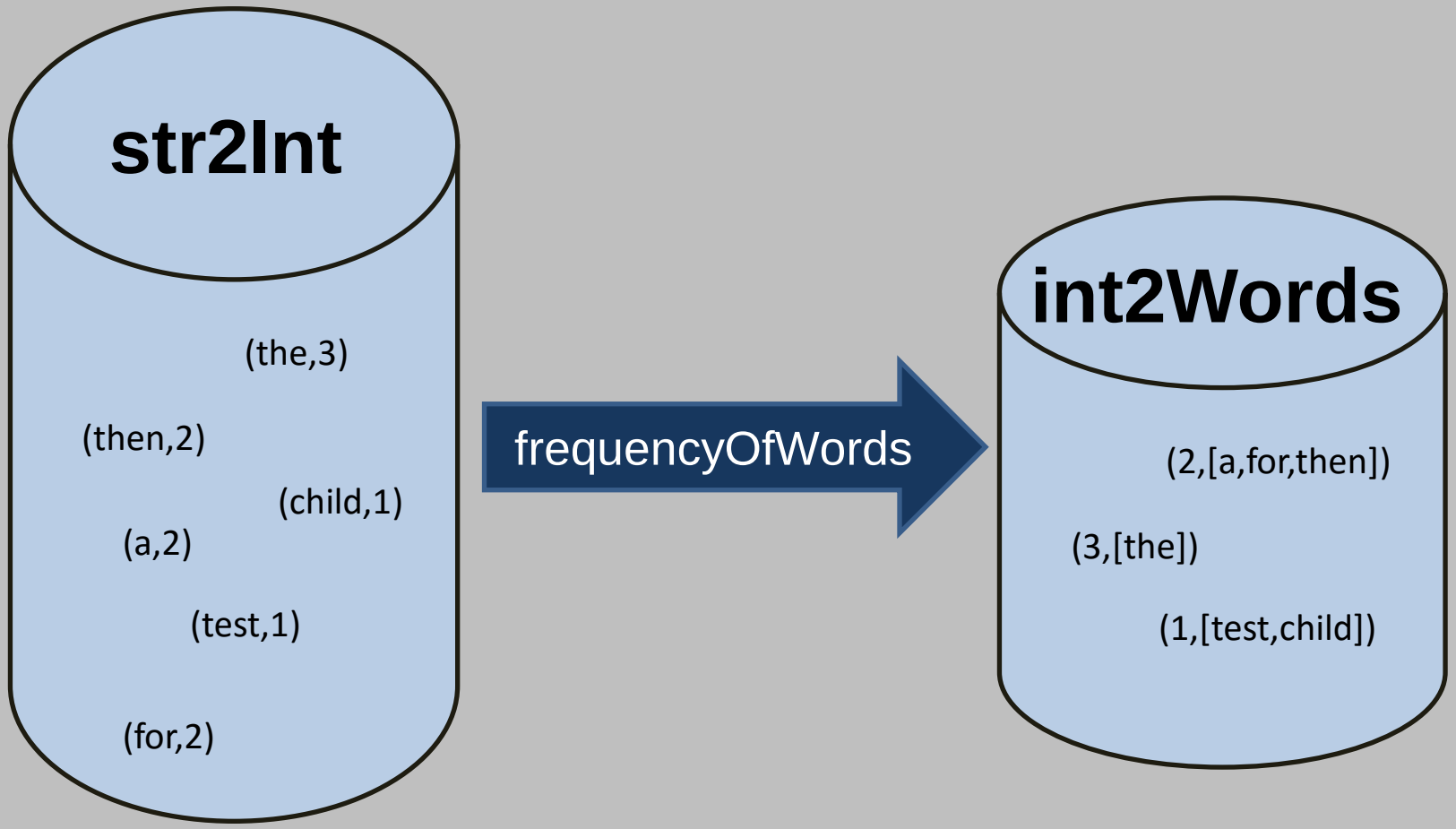
The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- **Kenne deine Bibliotheken.**
- Strebe nach Einfachheit.

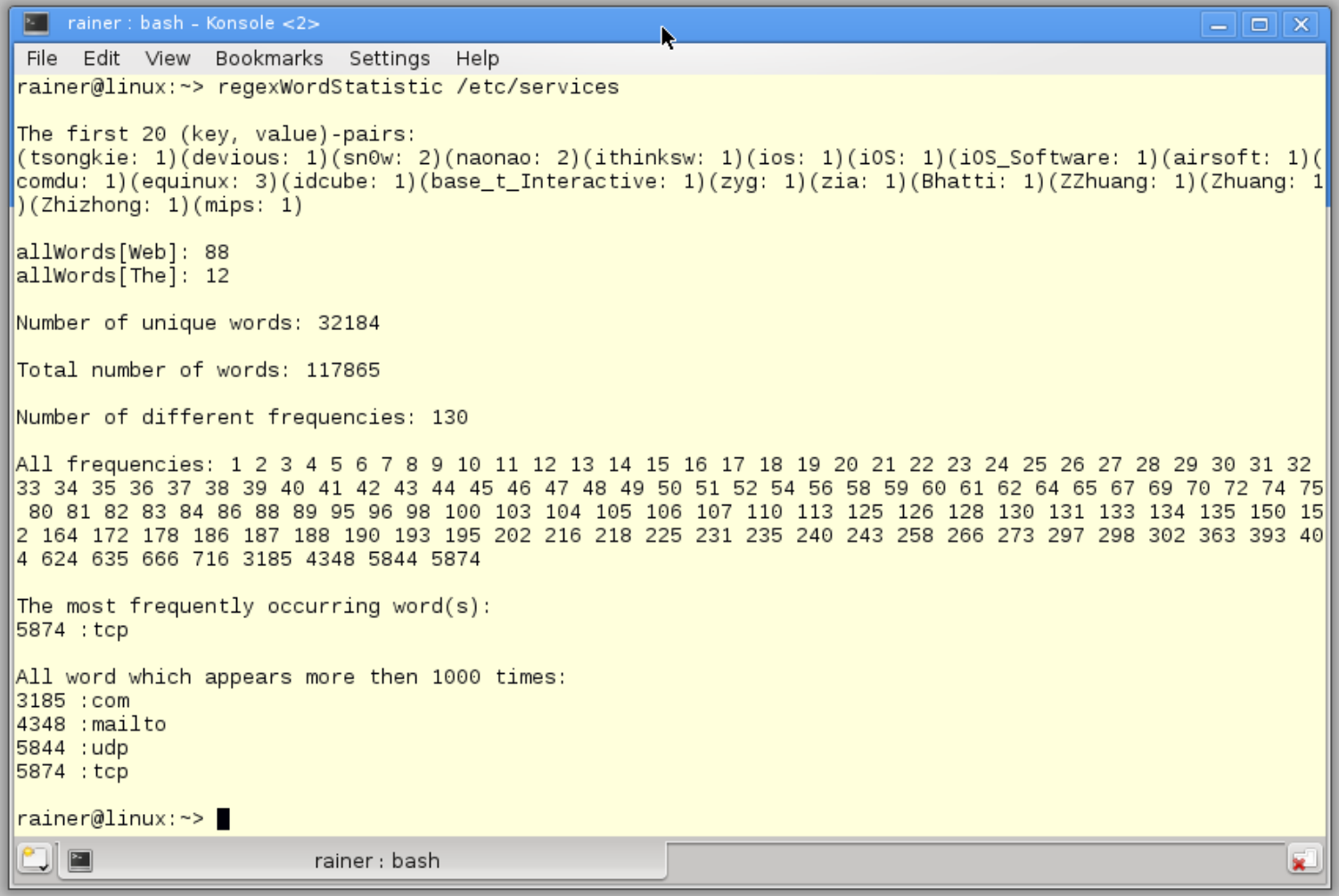
Reguläre Ausdrücke

```
using str2Int= std::unordered_map<std::string,std::size_t>;
using int2Words= std::map<std::size_t,std::vector<std::string>>;

str2Int wordCount(const std::string& text){
  std::regex wordReg(R"(\w+)");
  std::sregex_iterator wordItBegin(text.begin(),text.end(),wordReg);
  const std::sregex_iterator wordItEnd;
  str2Int allWords;
  for (; wordItBegin != wordItEnd;++wordItBegin){
    ++allWords[wordItBegin->str()];
  }
  return allWords;
}

int2Words frequencyOfWords(str2Int& wordCount){
 ...
}
```

Reguläre Ausdrücke



Reguläre Ausdrücke



Thread versus Task

Thread

```
int res;
thread t([&]{res= 3+4;});
t.join();
cout << res << endl;
```

Task

```
auto fut=async([]{return 3+4;});
cout << fut.get() << endl;
```

Kriterium	Thread	Task
Beteiligten	Erzeuger- und Kinderthread	Promise und Future
Kommunikation	gemeinsame Variable	Kommunikationskanal
Threaderzeugung	verbindlich	optional
Synchronisation	`join`-Aufruf wartet	`get`-Aufruf blockiert
Ausnahme im Kind-Thread	Kind- und Erzeuger-Thread terminieren	Rückgabewert des `get`-Aufrufes
Kritischer Bereich	ja	nein
Benachrichtigungen	nein	ja

Neue Container: Hashtabellen

```
map<string,int> m {{"Dijkstra",1972},{"Scott",1976}};
m["Ritchie"] = 1983;
for(auto p : m) cout << '{' << p.first << ',' << p.second << '}';
  // {Dijkstra,1972}{Ritchie,1983}{Scott,1976}


unordered_map<string,int> um {{"Dijkstra",1972},{"Scott",1976}};
um["Ritchie"] = 1983;
for(auto p : um) cout << '{' << p.first << ',' << p.second << '}';
  // {Ritchie,1983}{Dijkstra,1972}{Scott,1976}
```

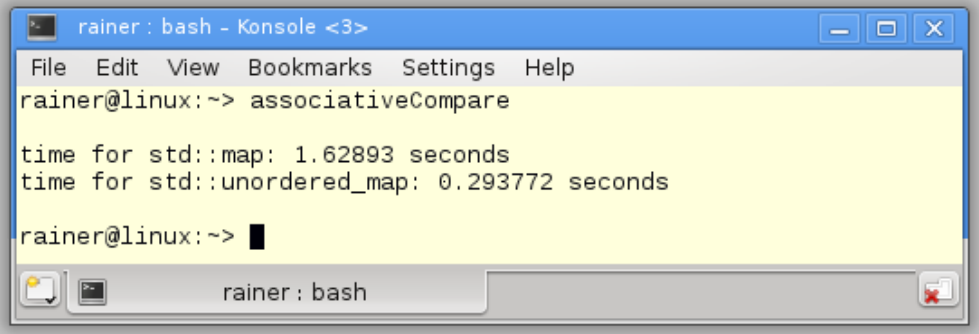
➡ Die Schlüssel der Hashtabelle sind nicht sortiert.

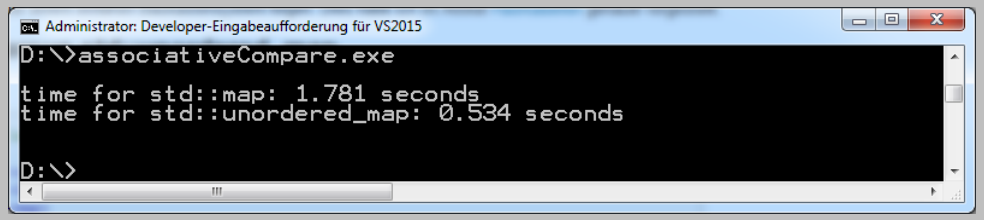
Hashtabellen

```
static const long long mapSize= 10'000'000;
static const long long accSize= 1'000'000;
...
std::map<int,int> myMap;
std::unordered_map<int,int> myHash;

for ( long long i=0; i < mapSize; ++i ){
  myMap[i]=i;
  myHash[i]= i;
}
std::vector<int> randValues;
randValues.reserve(accSize);
std::random_device seed;
std::mt19937 engine(seed());
std::uniform_int_distribution<> uniformDist(0,mapSize);
for ( long long i=0 ; i< accSize ; ++i) randValues.push_back(uniformDist(engine));

auto start = std::chrono::system_clock::now();
for ( long long i=0; i < accSize; ++i) myMap[randValues[i]];
std::chrono::duration<double> dur= std::chrono::system_clock::now() - start;
std::cout << "time for std::map: " << dur.count() << " seconds" << std::endl;
...
```

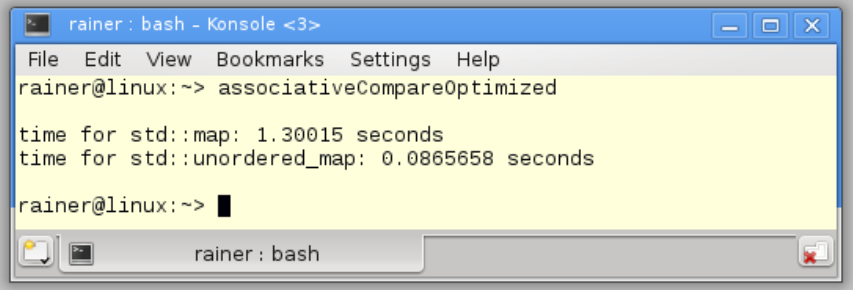
Hashtabellen

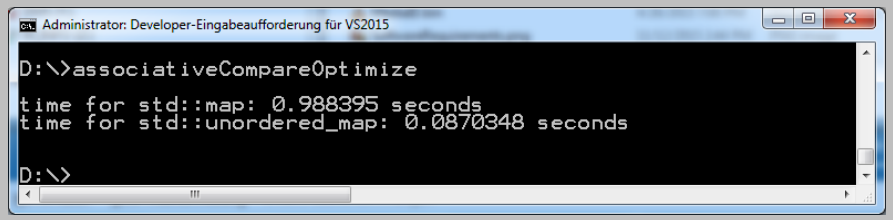
- Ohne Optimierung





- Mit Optimierung





➡ Erkenntnisse:

- `std::unordered_map` schlägt `std::map` deutlich
- Optimierung zahlt sich aus

The Zen of ...

- Vermeide implizite Typkonvertierungen.
- Programmiere deklarativ.
- Unterstütze automatische Optimierungen.
- Sei nicht schlauer als der Compiler.
- Behalte das große Bild im Auge.
- Vermeide undefiniertes Verhalten.
- Achte auf die Lesbarkeit des Codes.
- Lasse dir helfen.
- Kenne deine Bibliotheken.
- **Strebe nach Einfachheit.**

Generische Lambda-Funktionen

```
auto add=[](int i,int i2){ return i + i2; };
auto add14=[](auto i,auto i2){ return i + i2; };

std::cout << add(2000,11);                                    // 2011

std::cout << add14(2000,14);                                  // 2014
std::cout << add14(std::string("Hello "),std::string("World")); // Hello World

std::vector<int> myVec{1,2,3,4,5,6,7,8,9};
auto res= std::accumulate(myVec.begin(),myVec.end(),1,
                          [](int i, int i2){ return i * i2;});
std::cout << res;                                             // 362880

auto res14= std::accumulate(myVec.begin(),myVec.end(),1,
                            [](auto i, auto i2){ return i * i2;});
std::cout << res14;                                           // 362880
```

Tasks statt Bedingungsvariablen

```
void waitingForWork(){
  unique_lock<mutex> lck(mutex_);
  condVar.wait(lck,[]{return dataReady;});
  // do something
}


void setDataReady(){
  lock_guard<mutex> lck(mutex_);
  {
    dataReady=true;
  }
  condVar.notify_one();
}


...
thread t1(waitingForWork);
thread t2(setDataReady);
...
```

```
void waitingForWork(future<void>&& fut){
  fut.wait();
  // do something
}


void setDataReady(promise<void>&& prom){
  prom.set_value();
}


...
promise<void> sendReady;
auto fut= sendReady.get_future();
thread t1(waitingForWork,move(fut));
thread t2(setDataReady,move(sendReady));
...
```

Tasks versus Bedingungsvariablen

Kriterium	Bedingungsvariablen	Tasks
Mehrmalige Synchronisation möglich	Ja	Nein
Kritischer Bereich	Ja	Nein
Ausnahmebehandlung im Empfänger	Nein	Ja
Spurious wakeup	Ja	Nein
Lost wakeup	Ja	Nein

➡ Ziehen Sie Tasks Bedingungsvariablen vor.

Vereinheitlichte Initialisierung mit Initialisiererlisten

```
std::array<int,5> myArray= {-10,5,1,4,5};
for ( auto i: myArray) std::cout << i << " ";                    // - 10 5 1 4 5


std::vector<int> myVector= {-10,5,1,4,5};
for ( auto i: myVector) std::cout << i << " ";                   // - 10 5 1 4 5


std::set<int> mySet= {-10,5,1,4,5};
for ( auto i: mySet) std::cout << i << " ";                      // -10 1 4 5


std::unordered_multiset<int> myUnorderedMultiSet= {-10,5,1,4,5};
for ( auto i: myUnorderedMultiSet) std::cout << i << " ";    // -10 5 5 1 4
```

Vereinheitlichte Initialisierung mit { }

Neue Anwendungsfälle.

- Container der STL

```
std::vector<int> intVec{1,2,3,4,5};
```

- Konstante Heap-Array

```
const float* p= new const float[2]{1.2,2.1};
```

- Konstantes C-Array als Attribut einer Klasse

```
struct MyArray{
  MyArray(): data{1,2,3,4,5}{}
  const int data[5];
};
```

- Default-Initialisierung eines beliebigen Objekts

```
std::string s{};
```

- Initialisieren eines beliebigen Objekts

```
MyClass class{2011,3.14};
```

The Zen of ...

- Disclaimer:
  - Ich warte noch auf wertvolle Tipps.
  - Vielleicht sind 10 Tipps schon ausreichend in C++.
  - 10 ist eine sehr gute Zahl.
  - The Zen of C++ ist unvollendet.

→ The Zen of ...

Weitere Informationen

- **Modernes C++:** Schulungen, Coaching und Technologieberatung durch Rainer Grimm
  - www.ModernesCpp.de

- Blogs zu modernem C++
  - www.grimm-jaud.de (Deutsch)
  - www.ModernesCpp.com (Englisch)

- Kontakt
  - @rainer_grimm
  - schulungen@grimm-jaud.de


Rainer Grimm
Schulungen, Coaching und Technologieberatung