C++: Typsicher

Fehler zur Compile-Zeit finden und Tippfehler reduzieren

Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info
@Andreas__Fertig

Typsicherheit

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Typsicherheit - Definition

... implements type safety, which basically means that you can't use a variable inappropriately for the terms of its type. In other words, according to the rules of type safety, you can divide by zero — that's not a type error — but you can't perform an arithmetic operation with a string unless you do some conversions first.
Type safety can be described in a language specification and is enforced in a development environment like Xcode; it can also be enforced with careful programming. That's part of the problem: Without enforced type safety, it's very easy to slip up and create unsafe code. "

— Feiler [1]

Andreas Fertig v1.0 C++: Typsicher 3

Typsicherheit - Definition

... implements type safety, which basically means that you can't use a variable inappropriately for the terms of its type. In other words, according to the rules of type safety, you can divide by zero — that's not a type error — but you can't perform an arithmetic operation with a string unless you do some conversions first.
Type safety can be described in a language specification and is enforced in a development environment like Xcode; it can also be enforced with careful programming. That's part of the problem: Without enforced type safety, it's very easy to slip up and create unsafe code. "

— Feiler [1]

Andreas Fertig v1.0 C++: Typsicher 4

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Typsicherheit - Definition

> ... implements type safety, which basically means that you can't use a variable inappropriately for the terms of its type. In other words, according to the rules of type safety, you can divide by zero — that's not a type error — but you can't perform an arithmetic operation with a string unless you do some conversions first.
> Type safety can be described in a language specification and is enforced in a development environment like Xcode; it can also be enforced with careful programming. That's part of the problem: Without enforced type safety, it's very easy to slip up and create unsafe code. "
>
> — Feiler [1]

Andreas Fertig v1.0 | C++: Typsicher | 5

Typsicherheit - Definition

> ... implements type safety, which basically means that you can't use a variable inappropriately for the terms of its type. In other words, according to the rules of type safety, you can divide by zero — that's not a type error — but you can't perform an arithmetic operation with a string unless you do some conversions first.
> Type safety can be described in a language specification and is enforced in a development environment like Xcode; it can also be enforced with careful programming. That's part of the problem: Without enforced type safety, it's very easy to slip up and create unsafe code. "
>
> — Feiler [1]

Andreas Fertig v1.0 | C++: Typsicher | 6

Typsicherheit - Definition

> Type safety means that the compiler will validate types while compiling, and throw an error if you try to assign the wrong type to a variable. "
>
> — Razin [2]

Motivation

```
1 short int max(short int a, short int b)
2 {
3   return (a > b) ? a : b;
4 }
```

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Motivation

- Simple Funktion; sicher?
  - Code compiliert.
  - Und funktioniert...

```
1
2 short int max(short int a, short int b)
3 {
4   return (a > b) ? a : b;
5 }
6
7 void Main()
8 {
9   short int          a = 1;
10  unsigned short int b = 65530;
11
12  printf("max: %d\n", max(a, b));
13 }
```

Motivation

- Simple Funktion; sicher?
  - Code compiliert.
  - Und funktioniert... meistens

```
1
2 short int max(short int a, short int b)
3 {
4   return (a > b) ? a : b;
5 }
6
7 void Main()
8 {
9   short int          a = 1;
10  unsigned short int b = 65530;
11
12  printf("max: %d\n", max(a, b));
13 }
```

```
$ ./a.out
max: 1
```

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

C++ Insights

C++ Insights
https://cppinsights.io

Issues About

Made by Andreas Fertig
Powered by Flask and CodeMirror

Source:

```
1 #include <cstdio>
2
3 short int max(short int a, short int b)
4 {
5   return (a > b) ? a : b;
6 }
7
8 void Main()
9 {
10   short int          a = 1;
11   unsigned short int b = 65530;
12
13   printf("max: %d\n", max(a, b));
14 }
```

Insight:

```
1 #include <cstdio>
2
3 short max(short a, short b)
4 {
5   return (static_cast<int>(a) > static_cast<int>(b)) ? a : b;
6 }
7
8
9 void Main()
10 {
11   short a = 1;
12   unsigned short b = 65530;
13   printf("max: %d\n", static_cast<int>(max(a, static_cast<short>(b))));
14 }
15
```

Console:

```
Insights exited with result code: 0
```


Motivation

- Simple Funktion; sicher?
  - Code compiliert.
  - Und funktioniert... meistens
- Ein (einfaches) Template bring viel Schutz.
  - Das Template passt nur, wenn beide Parameter stimmen.
  - Jetzt gibt es keine implizite Konversion.
  - Harter Fehler beim Compilieren.

```
1 template<typename T>
2 T max(T a, T b)
3 {
4   return (a > b) ? a : b;
5 }
6
7 void Main()
8 {
9   short int          a = 1;
10  unsigned short int b = 65530;
11
12  // printf("max: %d\n", max(a, b));
13 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Mehr Motivation?

Mehr Motivation?

Jaaaaaaaaa

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Motivation

- Typsicherheit
- Fehler vermeiden bzw. früh erkennen
  - Fehler bereits zur Compile-Zeit erkennen.

```
1 void Write(const char* data, const size_t len)
2 {
3   printf("f: %s\n", __PRETTY_FUNCTION__);
4
5   const int fd = Open();
6
7   for(size_t i = 0; i < len; ++i) {
8     write(fd, data[i]);
9   }
10
11  Close(fd);
12 }
13
14 void Main()
15 {
16   char buffer[10]{};
17   Write(buffer, sizeof(buffer));
18
19   char buffer2[20]{};
20   Write(buffer2, sizeof(buffer2));
21 }
```


Motivation

- Typsicherheit
- Fehler vermeiden bzw. früh erkennen
  - Fehler bereits zur Compile-Zeit erkennen.

```
1 template<typename T>
2 void Write(const T& data)
3 {
4   printf("f: %s\n", __PRETTY_FUNCTION__);
5
6   const int fd = Open();
7
8   for(const auto& c : data) {
9     write(fd, c);
10  }
11
12  Close(fd);
13 }
14
15 void Main()
16 {
17   char buffer[10]{};
18   Write(buffer);
19
20   char buffer2[20]{};
21   Write(buffer2);
22 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Motivation

- Typsicherheit
- Fehler vermeiden bzw. früh erkennen
  - Fehler bereits zur Compile-Zeit erkennen.
- Code-bloat
  - Immer ein Risiko mit Templates.

```
1 template<typename T>
2 void Write(const T& data)
3 {
4   printf("f: %s\n", __PRETTY_FUNCTION__);
5
6   const int fd = Open();
7
8   for(const auto& c : data) {
9     write(fd, c);
10  }
11
12  Close(fd);
13 }
14
15 void Main()
16 {
17   char buffer[10]{};
18   Write(buffer);
19
20   char buffer2[20]{};
21   Write(buffer2);
22 }
```

```
$ ./a.out
f: void Write(const T&) [with T = char [10]]
f: void Write(const T&) [with T = char [20]]
```


Motivation

- Typsicherheit
- Fehler vermeiden bzw. früh erkennen
  - Fehler bereits zur Compile-Zeit erkennen.
- Code-bloat
  - Immer ein Risiko mit Templates.
  - Cleveres Refactoring hilft.

```
1 namespace details {
2   void Write(const char* data, const size_t size)
3   {
4     const int fd = Open();
5
6     for(int i = 0; i < size; ++i) {
7       write(fd, data[i]);
8     }
9
10    Close(fd);
11  }
12 }  // namespace details
13
14 template<typename T>
15 void Write(const T& data)
16 {
17   printf("f: %s\n", __PRETTY_FUNCTION__);
18
19   details::Write(data, sizeof(data));
20 }
21
22 void Main()
23 {
24   char buffer[10]{};
25   Write(buffer);
26
27   char buffer2[20]{};
28   Write(buffer2);
29 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Compile-Zeit Prüfungen

- Typische Funktion?
  - Zwei Zeiger und eine Längenangabe.
  - Immerhin: Prüfung ob die Größe passt.

```
 1 void macAddr2Str(const MACAddress* pMacAddr,
 2                  char*             pStr,
 3                  size_t            maxStrLen)
 4 {
 5   if(maxStrLen < (ETH_ADDR_LEN * 2)) {
 6     /* string to small */
 7     return;
 8   }
 9
10   // ...
11 }
```

Compile-Zeit Prüfungen

- Typische Funktion?
  - Zwei Zeiger und eine Längenangabe.
  - Immerhin: Prüfung ob die Größe passt.
- Reduzieren der Parameter mit Hilfe des Compilers.
  - Und Zeiger durch Referenz ersetzt.

```
 1 template<size_t DEST_STR_LEN>
 2 void macAddr2Str(const MACAddress& macAddr,
 3                  char (&str)[DEST_STR_LEN])
 4 {
 5   if(DEST_STR_LEN < (ETH_ADDR_LEN * 2)) {
 6     /* string to small */
 7     return;
 8   }
 9
10   // ...
11 }
```

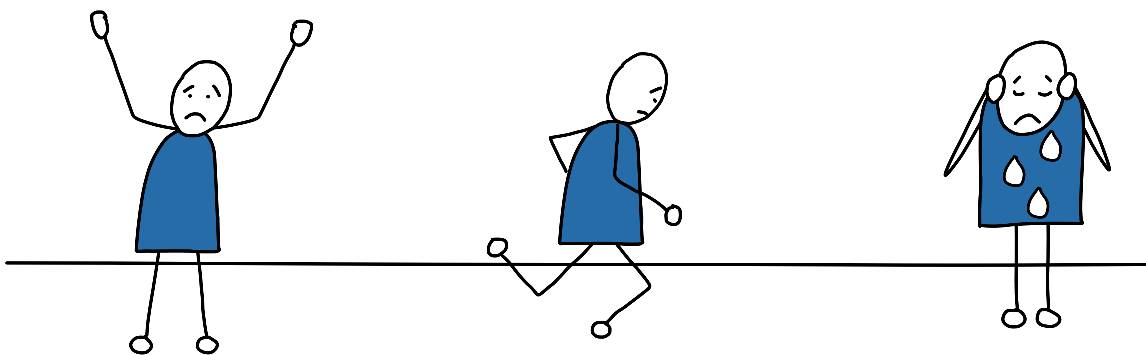
© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Compile-Zeit Prüfungen

- Typische Funktion?
  - Zwei Zeiger und eine Längenangabe.
  - Immerhin: Prüfung ob die Größe passt.
- Reduzieren der Parameter mit Hilfe des Compilers.
  - Und Zeiger durch Referenz ersetzt.
- Größenprüfung jetzt zur Compile-Zeit.

```
 1 template<size_t DEST_STR_LEN>
 2 void macAddr2Str(const MACAddress& macAddr,
 3                  char (&str)[DEST_STR_LEN])
 4 {
 5   /* string to small */
 6   static_assert(DEST_STR_LEN > (ETH_ADDR_LEN * 2),
 7                 "Destination string must be at least "
 8                 "the size of MAC-address plus "
 9                 "spaces.");
10
11   // ...
12 }
```


Don’t panic!



© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

MAC-Adressen vergleichen

- Vergleich von Datenstrukturen oft Schreibarbeit mit Tendenz zu Fehlern.
  - Beispielsweise die Prüfung ob eine MAC-Adresse gültig ist oder gleich einer anderen.

```
1 struct MACAddress
2 {
3   unsigned char value[6];
4 };
5
6 #define COMPARE_MAC_V(ma, v)                                    \
7   (((ma)[0] == (v)) && ((ma)[1] == (v)) &&                      \
8    ((ma)[2] == (v)) && ((ma)[3] == (v)) &&                      \
9    ((ma)[4] == (v)) && ((ma)[5] == (v)))
10
11 #define COMPARE_MAC(ma, mb)                                     \
12   (((ma)[0] == (mb)[0]) && ((ma)[1] == (mb)[1]) &&              \
13    ((ma)[2] == (mb)[2]) && ((ma)[3] == (mb)[3]) &&              \
14    ((ma)[4] == (mb)[4]) && ((ma)[5] == (mb)[5]))
15
16 void Main()
17 {
18   const MACAddress macA{1, 1, 1, 1, 1, 1};
19   const MACAddress macB{1, 1, 1, 1, 1, 2};
20   const MACAddress macC{1, 1, 1, 1, 1, 1};
21
22   printf("%d\n", COMPARE_MAC_V(macA.value, 1));
23   printf("%d\n", !COMPARE_MAC_V(macB.value, 1));
24
25   printf("%d\n", !COMPARE_MAC(macA.value, macB.value));
26   printf("%d\n", COMPARE_MAC(macA.value, macC.value));
27 }
```


MAC-Adressen vergleichen

- Vergleich von Datenstrukturen oft Schreibarbeit mit Tendenz zu Fehlern.
  - Beispielsweise die Prüfung ob eine MAC-Adresse gültig ist oder gleich einer anderen.
- Ziel:
  - Code wie rechts abgebildet.
  - Einfaches Vergleichen zweier gleich großer Arrays.
  - Vergleich aller Felder eines Arrays auf einen bestimmten Wert.

```
1 struct MACAddress
2 {
3   unsigned char value[6];
4 };
5
6 void Main()
7 {
8   const MACAddress macA{1, 1, 1, 1, 1, 1};
9   const MACAddress macB{1, 1, 1, 1, 1, 2};
10  const MACAddress macC{1, 1, 1, 1, 1, 1};
11
12  printf("%d\n", Compare(macA.value, 1));
13  printf("%d\n", !Compare(macB.value, 1));
14
15  printf("%d\n", !Compare(macA.value, macB.value));
16  printf("%d\n", Compare(macA.value, macC.value));
17 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

MAC-Adressen vergleichen - Variadische Templates

```
1 namespace details::array_single_compare {
2   template<typename T, size_t N, typename TTo, size_t... I>
3   constexpr bool Compare(const T (&ar)[N], const TTo& to, std::index_sequence<I...>)
4   {
5     return ((to == ar[I]) && ...);
6   }
7 } /* namespace details::array_single_compare */
8
9 template<typename T, size_t N, typename TTo>
10 constexpr bool Compare(const T (&ar)[N], const TTo& to)
11 {
12   return details::array_single_compare::Compare(ar, to, std::make_index_sequence<N>{});
13 }
14
15 namespace details::array_compare {
16   template<typename T, size_t N, size_t... I>
17   constexpr bool Compare(const T (&ar)[N], const T (&to)[N], std::index_sequence<I...>)
18   {
19     return ((to[I] == ar[I]) && ...);
20   }
21 } /* namespace details::array_compare */
22
23 template<typename T, size_t N>
24 constexpr bool Compare(const T (&ar)[N], const T (&to)[N])
25 {
26   return details::array_compare::Compare(ar, to, std::make_index_sequence<N>{});
27 }
```


Length + Value

- Wert und Länge zu transportieren ist:
  - Fehleranfällig
  - Mehr zu schreiben & lesen.

```
1 bool Send(const char* data, size_t size)
2 {
3   if(!data) {
4     return false;
5   }
6
7   return write(data, size);
8 }
9
10 void Read(const char* data, size_t size)
11 {
12   if(!data) {
13     return;
14   }
15
16   // fill buffer with data
17 }
18
19 void Main()
20 {
21   char buffer[1024]{};
22
23   Read(buffer, sizeof(buffer));
24   Send(buffer, sizeof(buffer));
25
26   char buffer2[2048]{};
27
28   Read(buffer, sizeof(buffer2));
29   Send(buffer, sizeof(buffer2));
30 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Length + Value

- Wert und Länge zu transportieren ist:
  - Fehleranfällig
  - Mehr zu schreiben & lesen.
- Mit einem simplen Template wird es (leicht) besser:
  - Länge muss nicht mehr angegeben werden.
  - **Aber:** Code-bloat Gefahr!

```
 1 template<size_t N>
 2 bool Send(const char (&data)[N])
 3 {
 4   return write(data, N);
 5 }
 6
 7 template<size_t N>
 8 void Read(char (&data)[N])
 9 {
10   // fill buffer with data
11 }
12
13 void Main()
14 {
15   char buffer[1024]{};
16
17   Read(buffer);
18   Send(buffer);
19
20   char buffer2[2048]{};
21
22   Read(buffer2);
23   Send(buffer2);
24 }
```


Length + Value

- Wert und Länge zu transportieren ist:
  - Fehleranfällig
  - Mehr zu schreiben & lesen.
- Mit einem simplen Template wird es (leicht) besser:
  - Länge muss nicht mehr angegeben werden.
  - **Aber:** Code-bloat Gefahr!
- Besser ist es mit `std::array`:
  - Problem hier: Die Größe muss immer gleich sein.

```
 1 bool Send(const std::array<char, 1024>& data)
 2 {
 3   return write(data.data(), data.size());
 4 }
 5
 6 void Read(std::array<char, 1024>& data)
 7 {
 8   // fill buffer with data
 9 }
10
11 void Main()
12 {
13   std::array<char, 1024> buffer{};
14
15   Read(buffer);
16   Send(buffer);
17
18   std::array<char, 2048> buffer2{};
19
20   //  Read(buffer2);
21   //  Send(buffer2);
22 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Length + Value

- Wert und Länge zu transportieren ist:
  - Fehleranfällig
  - Mehr zu schreiben & lesen.
- Mit einem simplen Template wird es (leicht) besser:
  - Länge muss nicht mehr angegeben werden.
  - **Aber:** Code-bloat Gefahr!
- Besser ist es mit `std::array`:
  - Problem hier: Die Größe muss immer gleich sein.
  - Alternative: Templates.
  - Problem dabei: Code-bloat.

```
 1 template<size_t N>
 2 bool Send(const std::array<char, N>& data)
 3 {
 4   return write(data.data(), data.size());
 5 }
 6
 7 template<size_t N>
 8 void Read(std::array<char, N>& data)
 9 {
10   // fill buffer with data
11 }
12
13 void Main()
14 {
15   std::array<char, 1024> buffer{};
16
17   Read(buffer);
18   Send(buffer);
19
20   std::array<char, 2048> buffer2{};
21
22   Read(buffer2);
23   Send(buffer2);
24 }
```


Length + Value

- Wert und Länge zu transportieren ist:
  - Fehleranfällig
  - Mehr zu schreiben & lesen.
- Mit einem simplen Template wird es (leicht) besser:
  - Länge muss nicht mehr angegeben werden.
  - **Aber:** Code-bloat Gefahr!
- Besser ist es mit `std::array`:
  - Problem hier: Die Größe muss immer gleich sein.
  - Alternative: Templates.
  - Problem dabei: Code-bloat.
- Noch eine Schippe besser: `span`
  - C++20 Single header Version von span: [3].
  - Kann sowohl C-Array als auch `std::array` fassen.
  - Natürlich range-based for ready.

```
 1 template<typename T>
 2 class span {
 3 public:
 4   constexpr span() = default;
 5   constexpr span(T* start, const size_t len)
 6   : data_(start), length(len) { }
 7
 8   template<size_t N>
 9   constexpr span(T (&arr)[N])
10   : span(arr, N) { }
11
12   template<size_t N>
13   constexpr span(const T (&arr)[N])
14   : span(arr, N) { }
15
16   template<size_t N, class AT = std::remove_const_t<T>>
17   constexpr span(std::array<AT, N>& arr)
18   : span(arr.data(), arr.size()) { }
19
20   constexpr size_t size() const { return length; }
21   T*               data() const { return data_; }
22   bool empty() const { return nullptr != data_; }
23
24   T* begin() const { return data_; }
25   T* end() const   { return data_ + length; }
26
27 private:
28   T*     data_;
29   size_t length;
30 };
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Length + Value

- Wert und Länge zu transportieren ist:
  - Fehleranfällig
  - Mehr zu schreiben & lesen.
- Mit einem simplen Template wird es (leicht) besser:
  - Länge muss nicht mehr angegeben werden.
  - **Aber:** Code-bloat Gefahr!
- Besser ist es mit `std::array`:
  - Problem hier: Die Größe muss immer gleich sein.
  - Alternative: Templates.
  - Problem dabei: Code-bloat.
- Noch eine Schippe besser: `span`
  - C++20 Single header Version von span: [3].
  - Kann sowohl C-Array als auch `std::array` fassen.
  - Natürlich range-based for ready.
  - Code aufgeräumt und sicher und mit wenig Overhead.

```
 1 bool Send(const span<char>& data)
 2 {
 3   return write(data.data(), data.size());
 4 }
 5
 6 void Read(span<char> data)
 7 {
 8   int i = 1;
 9   // fill buffer with data
10   for(auto& c : data) {
11     c = i;
12     ++i;
13   }
14 }
15
16 void Main()
17 {
18   std::array<char, 1024> buffer{};
19
20   Read(buffer);
21   Send(buffer);
22
23   char buffer2[2048]{};
24
25   Read(buffer2);
26   Send(buffer2);
27 }
```


Vergleich

```
 1 template<size_t DEST_STR_LEN>
 2 void macAddr2Str(const MACAddress& macAddr,
 3                  char (&str)[DEST_STR_LEN])
 4 {
 5   /* string to small */
 6   static_assert(DEST_STR_LEN > (ETH_ADDR_LEN * 2),
 7                 "Destination string must be at least "
 8                 "the size of MAC-address plus "
 9                 "spaces.");
10
11   // ...
12 }
```

```
 1 bool Send(const span<char>& data)
 2 {
 3   return write(data.data(), data.size());
 4 }
 5
 6 void Read(span<char> data)
 7 {
 8   int i = 1;
 9   // fill buffer with data
10   for(auto& c : data) {
11     c = i;
12     ++i;
13   }
14 }
15
16 void Main()
17 {
18   std::array<char, 1024> buffer{};
19
20   Read(buffer);
21   Send(buffer);
22
23   char buffer2[2048]{};
24
25   Read(buffer2);
26   Send(buffer2);
27 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Code Bloat Vermeiden

- Bereits bekannt.
  - Wurde bei der Code Bloat-Vermeidung weniger modern.
  - *Altmodisches `for`.*

```
1 namespace details {
2   void Write(const char* data, const size_t size)
3   {
4     const int fd = Open();
5
6     for(int i = 0; i < size; ++i) {
7       write(fd, data[i]);
8     }
9
10    Close(fd);
11  }
12 }  // namespace details
13
14 template<typename T>
15 void Write(const T& data)
16 {
17   printf("f: %s\n", __PRETTY_FUNCTION__);
18
19   details::Write(data, sizeof(data));
20 }
21
22 void Main()
23 {
24   char buffer[10]{};
25   Write(buffer);
26
27   char buffer2[20]{};
28   Write(buffer2);
29 }
```


Code Bloat Vermeiden

- Bereits bekannt.
  - Wurde bei der Code Bloat-Vermeidung weniger modern.
  - *Altmodisches `for`.*
- Mit `span` lässt sich der Code moderner gestalten.
  - Modernes `for`.
  - `sizeof` wird überflüssig.

```
1 void Write(const span<const char> data)
2 {
3     const int fd = Open();
4
5     for(const auto& c : data) {
6       write(fd, c);
7     }
8
9     Close(fd);
10 }
11
12 void Main()
13 {
14   char buffer[10]{};
15   Write(buffer);
16
17   char buffer2[20]{};
18   Write(buffer2);
19 }
```


© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

}

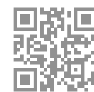
Ich bin Fertig.

https://cppinsights.io

Available online:

https://www.AndreasFertig.Info

Images by Franziska Panter:



https://panther-concepts.de

Quellen

[1] Feiler J., “Type safety in swift”. http://www.dummies.com/programming/macintosh/type-safety-in-swift/

[2] Razin , “What is type-safe?”. https://stackoverflow.com/questions/260626/what-is-type-safe

[3] Moene M., “span lite - A single-file header-only version of a C++20-like span for C++98, C++11 and later”. https://github.com/martinmoene/span-lite

Bilder:

22: Franziska Panter

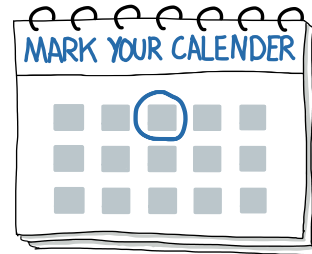
37: Franziska Panter

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info

Nächste Events

- *C++: Fast and Small - What are the Costs of Language Features*, C++ on Sea, February 05 2019
- *C++: Fast and Small - What are the Costs of Language Features*, Cpp Europe, February 26 2019
- *C++1x für eingebettete Systeme kompakt*, Seminar QA Systems, November 14 2018
- *C++ Templates - die richtige Dosis kompakt*, Seminar QA Systems, November 15 2018

Aktuelle Informationen unter:
`https://andreasfertig.info/talks.html`



Über Andreas Fertig

Foto: Lea Theweleit

Andreas studierte Informatik in Karlsruhe. Bereits seit seinem Studium befasst er sich mit eingebetteten Systemen und den damit einher gehenden Anforderungen und Besonderheiten. Seit 2010 ist er für die Philips Medizin Systeme als Softwareentwickler mit dem Schwerpunkt eingebettete Systeme tätig. Er verfügt über fundierte Kenntnisse von C++.

Freiberuflich arbeitet er als Dozent und Trainer. Zudem entwickelt er macOS Anwendungen und ist der Autor von cppinsights.io.

© 2018 Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info