C++: Typsicher

Verwenden Sie Templates, um die Lesbarkeit zu verbessern, Tippfehler zu reduzieren und Fehler während der Kompilierung zu erkennen

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

Typsicherheit

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]

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]

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]

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]

https://www.AndreasFertig.Info
post@AndreasFertig.Info

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

- Typsicherheit

```
1
2
3
4
5
6
7
8
9
10
11 void Foo(bool first, bool addNewLine);
12
13 void UseFoo(bool first, bool addNewLine)
14 {
15   Foo(addNewLine, first);
16 }
17
18 void Main()
19 {
20   UseFoo(true, false);
21 }
```

Motivation

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

```
#define STRONG_BOOL(typeName)                   \
  enum class typeName : bool                    \
  {                                             \
    No  = false,                                \
    Yes = true                                  \
  }

STRONG_BOOL(First);
STRONG_BOOL(AddNewLine);

void Foo(First first, AddNewLine addNewLine);

void UseFoo(First first, AddNewLine addNewLine)
{
  // Foo(addNewLine, first);
  Foo(first, addNewLine);
}

void Main()
{
  UseFoo(First::Yes, AddNewLine::No);
}
```


Motivation

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

```
void Write(const char* data, const size_t len)
{
  printf("f: %s\n", __PRETTY_FUNCTION__);

  const int fd = Open();

  for(size_t i = 0; i < len; ++i) {
    write(fd, data[i]);
  }

  Close(fd);
}

void Main()
{
  char buffer[10]{};
  Write(buffer, sizeof(buffer));

  char buffer2[20]{};
  Write(buffer2, sizeof(buffer2));
}
```

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 }
```


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 }
```


Sicherer Elementzugriff

- Array-Zugriff mit Bereichsprüfung
  - Die Prüfung ob das gewünschte Element $>= 0$ geht auch ohne Templates.
  - Die Stärke liegt hier in der Art wie `arr` definiert ist.
  - Der Compiler kennt die Größe des Arrays und gibt mit `N` immer den richtigen Wert an.
  - Kein Mitschleifen des Längenparameters und evtl. vertun.

```
1  //
2  // at() — Bounds-checked way of accessing builtin
3  // arrays, std::array, std::vector
4  //
5  template<class T, std::size_t N>
6  constexpr T& at(T (&arr)[N], const int i)
7  {
8    Expects(i >= 0 && i < narrow_cast<int>(N));
9    return arr[static_cast<std::size_t>(i)];
10 }
```

Aus der Guideline Support Library [3]

Sicherer Elementzugriff

- Array-Zugriff mit Bereichsprüfung
  - Die Prüfung ob das gewünschte Element $>= 0$ geht auch ohne Templates.
  - Die Stärke liegt hier in der Art wie `arr` definiert ist.
  - Der Compiler kennt die Größe des Arrays und gibt mit `N` immer den richtigen Wert an.
  - Kein Mitschleifen des Längenparameters und evtl. vertun.
- Leider geht hier kein `static_assert`.
  - Prüfung somit zur Laufzeit.

```
1 //
2 // at() — Bounds-checked way of accessing builtin
3 // arrays, std::array, std::vector
4 //
5 template<class T, std::size_t N>
6 constexpr T& at(T (&arr)[N], const int i)
7 {
8   Expects(i >= 0 && i < narrow_cast<int>(N));
9   return arr[static_cast<std::size_t>(i)];
10 }
```

Aus der Guideline Support Library [3]

`static_assert`

- `static_assert`:
  - Prüfung wird zur Compile-Zeit ausgeführt.
  - Seit C++17 auch ohne Nachricht möglich.
  - Code wandert garantiert nicht in das Binary.
  - Passed / Failed wird zur Compile-Zeit ermittelt.

```
1 static_assert( sizeof(size_t) != 4,
2                "Ups size_t wrong" );
```

Compile-Zeit Prüfungen

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

```
1  void makeMACAddrStr(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 makeMACAddrStr(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 }
```

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 makeMACAddrStr(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 }
```


Trailing return-type

- Seit C++11 verfügbar.
- Syntax sieht *gewöhnungsbedürftig* aus.
- Gründe für trailing return-types:
  - Return-Wert hängt vom den Argument-Typen ab.
- Zwingend erforderlich für `auto`-Return-Type in C++11.

```
1  template<class T, class U>
2  auto multiply(T const& a, U const& b) -> decltype(a * b)
3  {
4    return a * b;
5  }
6
7  void Main()
8  {
9    multiply(1, 2.0);
10 }
```

Trailing return-type

- Seit C++11 verfügbar.
- Syntax sieht *gewöhnungsbedürftig* aus.
- Gründe für trailing return-types:
  - Return-Wert hängt vom den Argument-Typen ab.
- Zwingend erforderlich für `auto`-Return-Type in C++11.
  - Mit C++14 nicht mehr erforderlich.

```
1  template<class T, class U>
2  auto multiply(T const& a, U const& b)
3  {
4    return a * b;
5  }
6
7  void Main()
8  {
9    multiply(1, 2.0);
10 }
```


Kompatibilität

- Trailing return-type alternativ interpretiert.
  - Trailing return-type kann für Kompatibilitätsbehandlung verwendet werden.

HEAD - 1

```
1  class Foo
2  {
3  public:
4    int Fest() const { return 23; }
5  };
```

HEAD

```
1  class Foo
2  {
3  public:
4    int Test() const { return 22; }
5  };
```

```
1
2  int getWrapper(const Foo& foo)
3  {
4    return foo.Test();
5  }
6
7
8
9
10
11
12
13 void Main()
14 {
15   Foo foo;
16
17   printf("%d\n", getWrapper(foo));
18 }
```

Kompatibilität

- Trailing return-type alternativ interpretiert.
  - Trailing return-type kann für Kompatibilitätsbehandlung verwendet werden.

HEAD - 1

```
1 class Foo
2 {
3 public:
4   int Fest() const { return 23; }
5 };
```

HEAD

```
1 class Foo
2 {
3 public:
4   int Test() const { return 22; }
5 };
```

```
1  template<typename T>
2  auto getWrapper(const T& foo) -> decltype(foo.Test())
3  {
4    return foo.Test();
5  }
6
7  template<typename T>
8  auto getWrapper(const T& foo) -> decltype(foo.Fest())
9  {
10   return foo.Fest();
11 }
12
13 void Main()
14 {
15   Foo foo;
16
17   printf("%d\n", getWrapper(foo));
18 }
```


`constexpr if`

- Ein Erweiterung von `constexpr`.
  - Dieses `if` sowie alle Verzweigungen werden zur Compile-Zeit evaluiert.
  - Nur der Zweig welcher `true` ergibt bleibt erhalten.
- Wichtig: Alle Statements welche *nicht* von einem Template-Parameter abhängen, auch wenn sie verworfen werden, müssen valide sein.

```
1  template<typename T>
2  auto Foo()
3  {
4    if constexpr(is_same_v<T, int>)
5    {
6      return 1;
7    }
8    else if  constexpr (is_same_v<T, float>)
9    {
10     return 1.1f;  // float
11   }
12   else
13   {
14     return 1.1;  // double
15   }
16 }
17
18 void Main()
19 {
20     auto x = Foo<int>();
21     auto y = Foo<float>();
22     auto z = Foo<double>();
23 }
```

constexpr if

- Ein Erweiterung von `constexpr`.
  - Dieses `if` sowie alle Verzweigungen werden zur Compile-Zeit evaluiert.
  - Nur der Zweig welcher `true` ergibt bleibt erhalten.
- Wichtig: Alle Statements welche *nicht* von einem Template-Parameter abhängen, auch wenn sie verworfen werden, müssen valide sein.
- Vorsicht bei return-type-deduction!
  - `constexpr` ist hier auch im `else if`-Zweig erforderlich.

```
1  template<typename T>
2  auto Foo()
3  {
4    if constexpr(is_same_v<T, int>)
5    {
6      return 1;
7    }
8    else if  constexpr (is_same_v<T, float>)
9    {
10     return 1.1f;  // float
11   }
12   else
13   {
14     return 1.1;  // double
15   }
16 }
17
18 void Main()
19 {
20     auto x = Foo<int>();
21     auto y = Foo<float>();
22     auto z = Foo<double>();
23 }
```


Variadische Templates

- Mit C++11 gibt es *variadische Templates*:
  - Templates die beliebig viele Parameter aufnehmen.
  - Bereits von Makros oder Funktionen bekannt.

```
1  static inline const char*
2  Normalize(const std::string& arg)
3  {
4    return arg.c_str();
5  }
6
7  template<class T>
8  static inline const T& Normalize(const T& arg)
9  {
10   return arg;
11 }
12
13 template<typename... Ts>
14 void Log(const char* fmt, const Ts&... ts)
15 {
16   printf(fmt, Normalize(ts)...);
17 }
```

Beispiel aus einem älteren Vortrag von Andrei Alexandrescu [4]

Variadische Templates

- Mit C++11 gibt es *variadische Templates*:
  - Templates die beliebig viele Parameter aufnehmen.
  - Bereits von Makros oder Funktionen bekannt.
- Lieber nur eine Funktion `Normalize()`?
  - Das ist mit C++17 möglich.

```cpp
template<typename T>
static inline decltype(auto) Normalize(const T& arg)
{
  if constexpr(std::is_same_v<T, std::string>) {
    return arg.c_str();
  } else {
    return arg;
  }
}

template<typename... Ts>
void Log(const char* fmt, const Ts&... ts)
{
  printf(fmt, Normalize(ts)...);
}
```


Variadische Templates

- Mit C++11 gibt es *variadische Templates*:
  - Templates die beliebig viele Parameter aufnehmen.
  - Bereits von Makros oder Funktionen bekannt.
- Lieber nur eine Funktion `Normalize()`?
  - Das ist mit C++17 möglich.

```cpp
enum class LogLevel
{
  Info,
  Warning,
  Error,
  Fatal
};

constexpr LogLevel GetLogLevel() {
  return LogLevel::Warning;
}

template<typename T>
static inline decltype(auto) Normalize(const T& arg)
{
  if constexpr(std::is_same_v<T, std::string>) {
    return arg.c_str();
  } else {
    return arg;
  }
}

template<LogLevel level, typename... Ts>
void Log(const char* fmt, const Ts&... ts) {
  if constexpr(GetLogLevel() <= level) {
    printf(fmt, Normalize(ts)...);
  }
}
```

Variadische Templates

- 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("Equal: %d\n", COMPARE_MAC_V(macA.value, 1));
23   printf("Not equal: %d\n",
24          !COMPARE_MAC_V(macB.value, 1));
25   printf("-------------------\n");
26   printf("Not equal: %d\n",
27          !COMPARE_MAC(macA.value, macB.value));
28   printf("Equal: %d\n",
29          COMPARE_MAC(macA.value, macC.value));
30 }
```


Variadische Templates

- 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("Equal: %d\n", Compare(macA.value, 1));
13   printf("Not equal: %d\n", !Compare(macB.value, 1));
14   printf("-------------------\n");
15   printf("Not equal: %d\n",
16          !Compare(macA.value, macB.value));
17   printf("Equal: %d\n",
18          Compare(macA.value, macC.value));
19 }
```

Variadische Templates

```cpp
namespace details::array_single_compare {
  template<typename T, size_t N, typename TTo, size_t... I>
  constexpr bool Compare(const T (&ar)[N], const TTo& to, std::index_sequence<I...>)
  {
    return ((to == ar[I]) && ...);
  }
} /* namespace details::array_single_compare */

template<typename T, size_t N, typename TTo>
constexpr bool Compare(const T (&ar)[N], const TTo& to)
{
  return details::array_single_compare::Compare(ar, to, std::make_index_sequence<N>{});
}

namespace details::array_compare {
  template<typename T, size_t N, size_t... I>
  constexpr bool Compare(const T (&ar)[N], const T (&to)[N], std::index_sequence<I...>)
  {
    return ((to[I] == ar[I]) && ...);
  }
} /* namespace details::array_compare */

template<typename T, size_t N>
constexpr bool Compare(const T (&ar)[N], const T (&to)[N])
{
  return details::array_compare::Compare(ar, to, std::make_index_sequence<N>{});
}
```


Length + Value

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

```cpp
bool Send(const char* data, size_t size)
{
  if(!data) {
    return false;
  }

  return write(data, size);
}

void Read(const char* data, size_t size)
{
  if(!data) {
    return;
  }

  // fill buffer with data
}

void Main()
{
  char buffer[1024]{};

  Read(buffer, sizeof(buffer));
  Send(buffer, sizeof(buffer));

  char buffer2[2048]{};

  Read(buffer, sizeof(buffer2));
  Send(buffer, sizeof(buffer2));
}
```

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!

```cpp
template<size_t N>
bool Send(const char (&data)[N])
{
  return write(data, N);
}

template<size_t N>
void Read(char (&data)[N])
{
  // fill buffer with data
}

void Main()
{
  char buffer[1024]{};

  Read(buffer);
  Send(buffer);

  char buffer2[2048]{};

  Read(buffer2);
  Send(buffer2);
}
```


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.

```cpp
bool Send(const std::array<char, 1024>& data)
{
  return write(data.data(), data.size());
}

void Read(std::array<char, 1024>& data)
{
  // fill buffer with data
}

void Main()
{
  std::array<char, 1024> buffer{};

  Read(buffer);
  Send(buffer);

  std::array<char, 2048> buffer2{};

  //  Read(buffer2);
  //  Send(buffer2);
}
```

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.

```cpp
template<size_t N>
bool Send(const std::array<char, N>& data)
{
  return write(data.data(), data.size());
}

template<size_t N>
void Read(std::array<char, N>& data)
{
  // fill buffer with data
}

void Main()
{
  std::array<char, 1024> buffer{};

  Read(buffer);
  Send(buffer);

  std::array<char, 2048> buffer2{};

  Read(buffer2);
  Send(buffer2);
}
```


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: [5]
  - Kann sowohl C-Array als auch `std::array` fassen.
  - Natürlich range-based for ready.

```cpp
template<typename T>
class span {
public:
  constexpr span() = default;
  constexpr span(T* start, const size_t len)
  : data_(start), length(len) { }

  template<size_t N>
  constexpr span(T (&arr)[N])
  : span(arr, N) { }

  template<size_t N>
  constexpr span(const T (&arr)[N])
  : span(arr, N) { }

  template<size_t N, class AT = std::remove_const_t<T>>
  constexpr span(std::array<AT, N>& arr)
  : span(arr.data(), arr.size()) { }

  constexpr size_t size() const { return length; }
  T*              data() const { return data_; }
  bool empty() const { return nullptr != data_; }

  T* begin() const { return data_; }
  T* end() const   { return data_ + length; }

private:
  T*     data_;
  size_t length;
};
```

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: [5]
  - 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 }
```


Length + Value

- Interessant hier ist das `&`.
  - Generell Regel: So oft es geht eine Referenz nutzen.

```
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 }
```

Length + Value

- Interessant hier ist das &.
  - Generell Regel: So oft es geht eine Referenz nutzen.
  - In diesem Fall wird der Code kleiner ohne & . [6]

```
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 }
```


abseil.io

abseil.io/tips/1

```
1 printf("%.*s\n", static_cast<int>(sv.size()), sv.data());
```


Vergleich

```
1  template<size_t DEST_STR_LEN>
2  void makeMACAddrStr(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 }
```

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 }
```

```
}
```

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] “GSL: Guideline Support Library”. https://github.com/Microsoft/GSL/

[4] Alexandrescu A., “GoingNative 2012 - Day 1 - Variadic templates are funadic”. https://www.youtube.com/watch?v=Hj3aoaEZckA

[5] Moene M., “string_view lite - A single-file header-only version of a C++17-like string_view for C++98, C++11 and later”. https://github.com/martinmoene/string-view-lite

[6] Benzaquen S., “Tip of the Week #93: using absl::Span”. https://abseil.io/tips/93

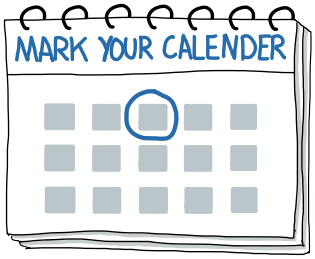
Bilder:

47: Franziska Panter

Nächste Events

- *C++: Be type-safe - The journey of determining the number of elements in an array*, NDC { Oslo }, June 14 2018
- *C++1x für eingebettete Systeme kompakt*, Seminar QA Systems, November 06 2018

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



Über Andreas Fertig

Foto: Lea Theweleit

Andreas arbeitet seit 2010 bei Philips Medizin Systeme als Softwareentwickler mit Schwerpunkt eingebettete Systeme.

Sein Fachgebiet ist der Entwurf und die Implementierung von C++ Softwaresystemen.

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