embeff
BETTER EMBEDDED.

Generic Programming for the masses

with C++20 Concepts

Daniel Penning
embeff GmbH

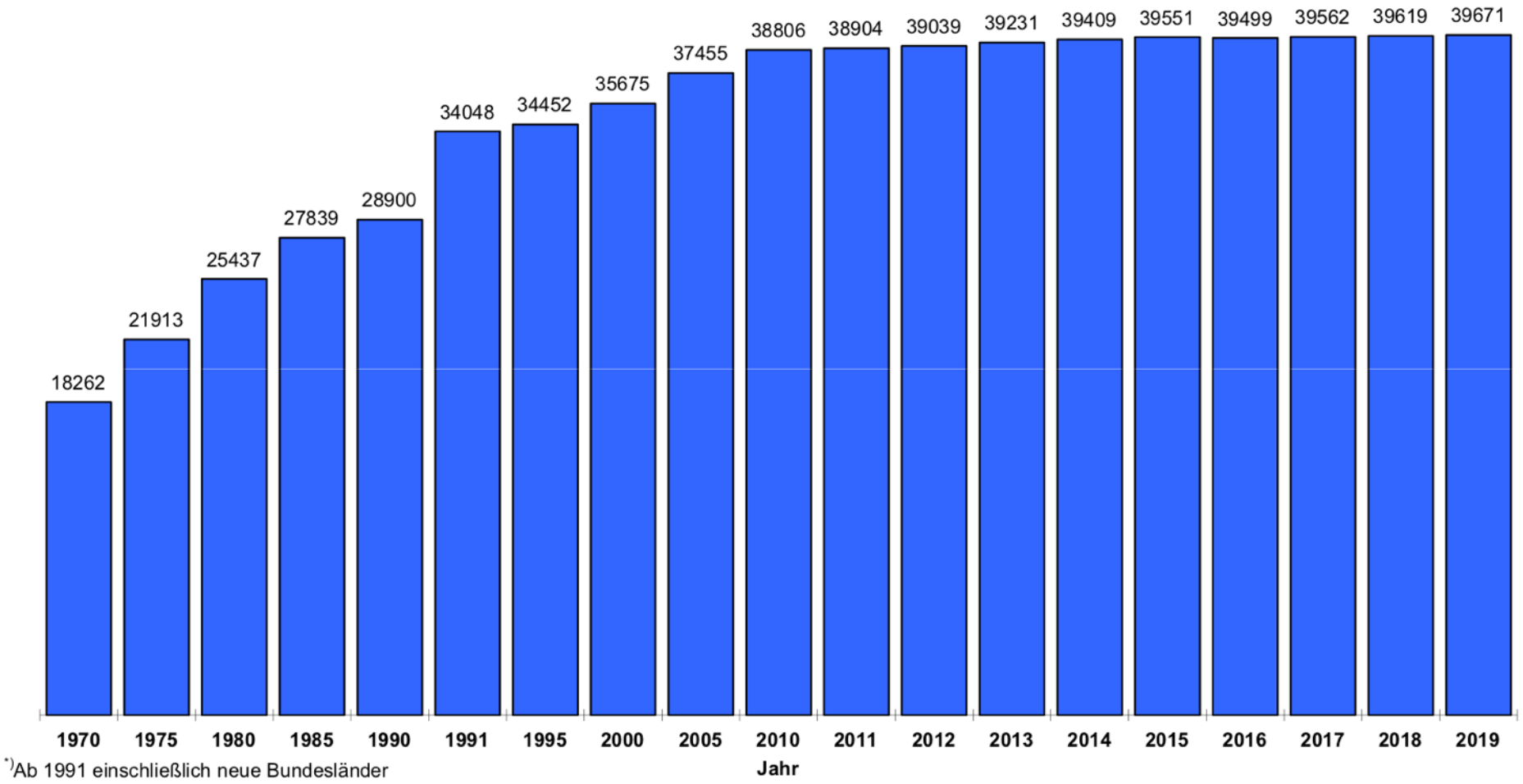
Genoa, Italy. 14.08.2018

~40.000 bridges

Brücken an Bundesfernstraßen

Anzahl*) der Brücken (Gesamtbauwerke)

Stand: 01.03.2019



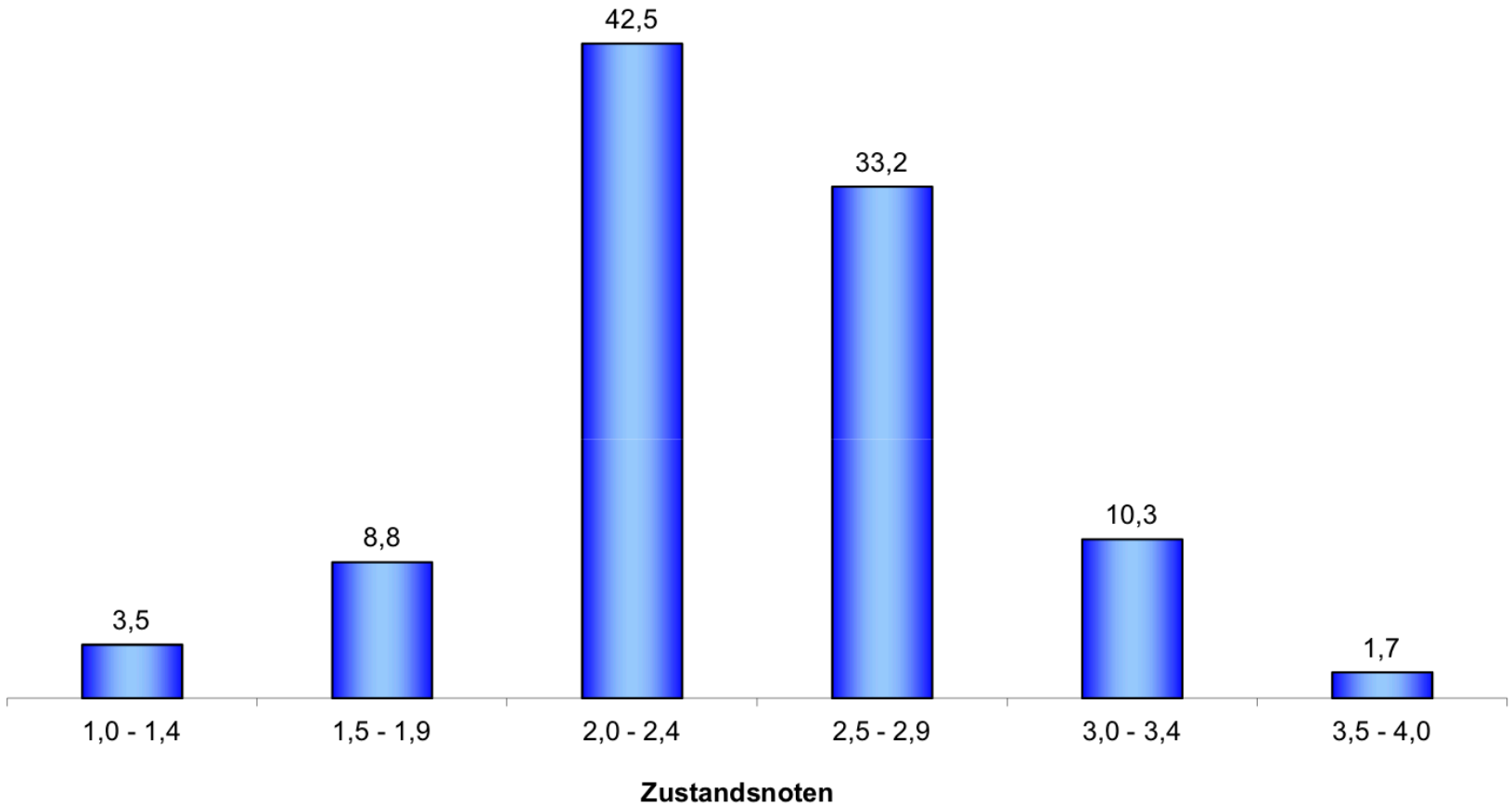
*)Ab 1991 einschließlich neue Bundesländer

~670 bridges
„unsatisfactory“

Brücken an Bundesfernstraßen

Zustandsnoten nach Brückenflächen der Teilbauwerke in Prozent

Stand: 01.03.2019



| Collapses?

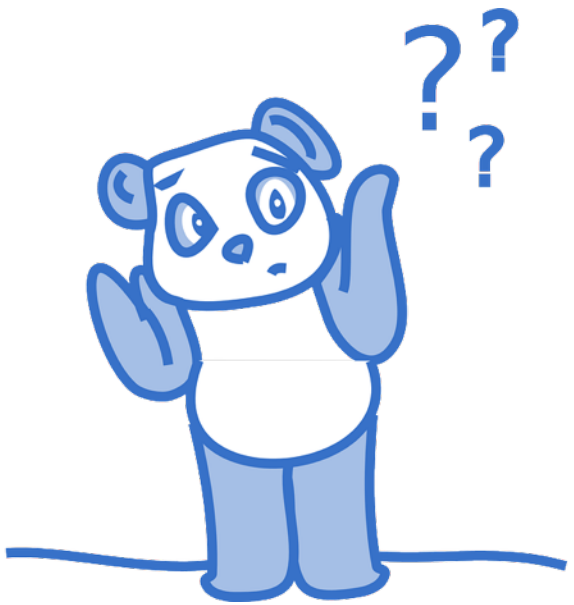
07.11.2012

Fußgängerbrücke

Deutschland, Hof

Eine baufällige Brücke, die aufgrund von Geldmangel nicht saniert werden konnte, stürzte in der Nacht ein. Der schlechte Zustand der Brücke war bekannt und die Brücke war gesperrt.

https://www.brueckenweb.de



- Known materials.
- Known construction methods.
- Experience.

| What is not done?

| Embedded Software

- What do we reuse?
  - C und C++
  - Know-How? Proven solutions? …

> *"We don't solve problems, we approximate solutions."*
>
> Sean Parent, A possible future of software development, 2007

- Is bridge building really that much easier… ?

Generic Programming*

David R. Musser†
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

Abstract

Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.

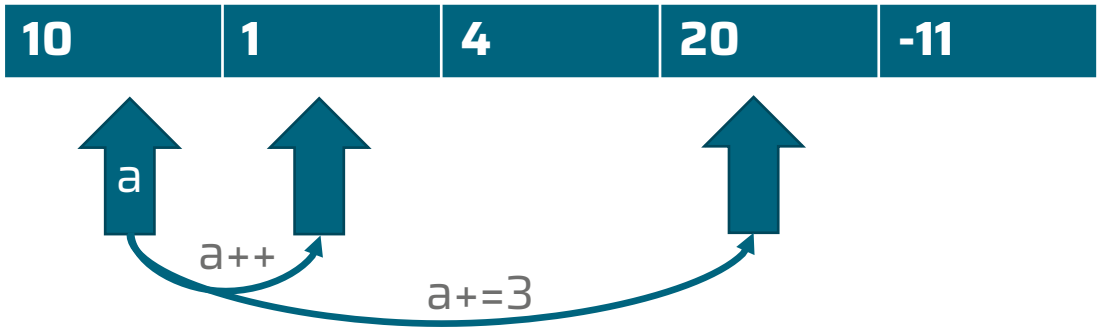
| Generic Programming (GP)

„Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.“

| GP in C++

```
template <typename InputIterator, typename Predicate> bool any_of(
    InputIterator first, InputIterator last, Predicate pred) {
    /* ... */
}
```

Iterators



int arr[5];
std::array<int,5>

std::list<int>

Iterators are the glue between data & algorithm.

| GP in C++

```
template <typename InputIterator, typename Predicate> bool any_of(
    InputIterator first, InputIterator last, Predicate pred) {
    for (; first != last; ++first) {
        if (pred(*first))
            return true;
    }
    return false;
}


int main() {
    std::array<int, 5> arr = {10, 1, 4, 20, -11};
    auto isNegative = [](int val) { return val < 0; };
    return any_of(arr.begin(),
}
```

Interface ambiguous for the user.

| GP in C++

```
struct Measurement {
    long id;
    int current_mA;
};

int main() {
    std::vector<Measurement> vec;
    std::sort(vec.begin(), vec.end());
}
```

Interface ambiguous for the compiler.

| Recap

✓GP makes code reuse possible.

– Template interface ambiguous for the user.
– Template interface ambiguous for the compiler.

✓**Concepts solve both problems.**

Concepts

```
template <typename T>
T const& min(T const& a, T const& b) {
    return (b < a) ? b : a;
}
```

```
struct Measurement {
    long id;
    int current_mA;
};
```

```
template <class T>
concept Comparable = requires(T const& lhs, T const& rhs) {
    { lhs < rhs } -> bool;
};
```

```
Measurement m1, m2;
/* Initialize m1 and m2 */
Measurement m3 = min(m1, m2);
```

```
error: cannot call function
'const T& min(const T&, const T&)
        [with T = Measurement]'
note: constraints not satisfied
```

The power of concepts

```
template <typename T>
concept StrongType = requires(T x) {
    { x + x } -> T;
    typename T::Base;
    { x.Raw() } -> typename T::Base;
    {sizeof(T) ==
       sizeof(typename T::Base)};
};
```

```
struct Ampere {
    using Base = int;

    explicit Ampere(int mA)
       : m_mA(mA) {}

    int Raw() const { return m_mA; }

    Ampere operator+(Ampere const& rhs) {
       return Ampere(m_mA + rhs.m_mA);
    }

    int m_mA;
};
static_assert(StrongType<Ampere>);
```

The power of concepts

```cpp
template <typename T, typename U>
concept SameSize = requires() {
    {sizeof(T) == sizeof(U)};
};


template <typename T>
concept StrongType = requires(T x) {
    { x + x } -> T;
    typename T::Base;
    { x.Raw() } -> typename T::Base;
} && SameSize<T, typename T::Base>;
```

```cpp
struct Ampere {
    using Base = int;

    explicit Ampere(int mA)
       : m_mA(mA) {}

    int Raw() const { return m_mA; }

    Ampere operator+(Ampere const& rhs) {
       return Ampere(m_mA + rhs.m_mA);
    }

    int m_mA;
};
static_assert(StrongType<Ampere>);
```

| Interface & Concepts

- Constrained Template Parameter

```
template <StrongType T>
void SerializeStrongType(T const& value)
```

- Requires Clause

```
template <typename T>
        requires StrongType<T>
void SerializeStrongType(T const& value)
```

- Terse Syntax

```
void SerializeStrongType(StrongType auto const& value)
```

| Predefined concepts

Comparison concepts

`boolean` (C++20)	specifies that a type can be used in Boolean contexts (concept)
`equality_comparable` `equality_comparable_with` (C++20)	specifies that operator `==` is an equivalence relation (concept)
`totally_ordered` `totally_ordered_with` (C++20)	specifies that the comparison operators on the type yield a total order (concept)

Object concepts

`movable` (C++20)	specifies that an object of a type can be moved and swapped (concept)
`copyable` (C++20)	specifies that an object of a type can be copied, moved, and swapped (concept)
`semiregular` (C++20)	specifies that an object of a type can be copied, moved, swapped, and default constructed (concept)
`regular` (C++20)	specifies that a type is regular, that is, it is both `semiregular` and `equality_comparable` (concept)

Callable concepts

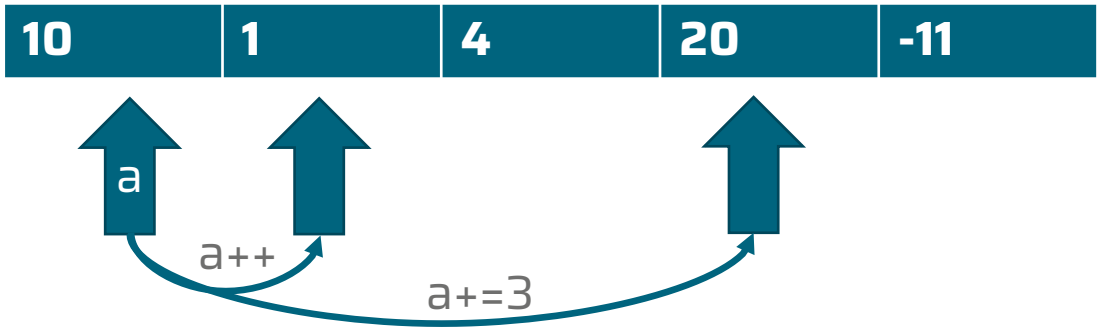
`invocable` `regular_invocable` (C++20)	specifies that a callable type can be invoked with a given set of argument types (concept)
`predicate` (C++20)	specifies that a callable type is a Boolean predicate (concept)
`relation` (C++20)	specifies that a callable type is a binary relation (concept)
`strict_weak_order` (C++20)	specifies that a `relation` imposes a strict weak ordering (concept)

https://en.cppreference.com/w/cpp/header/concepts

| A concept for any_of

```
template <typename InputIterator, typename Predicate>
bool any_of(
    InputIterator first, InputIterator last, Predicate pred) {
    for (; first != last; ++first) {
        if (pred(*first))
            return true;
    }
    return false;
}
```

Iterators



int arr[5];
std::array<int,5>

std::list<int>

Iterators are the glue between data & algorithm.

| A concept for any_of

```
template <typename I, typename P>
bool any_of(I first, I last, P pred) {
    for (; first != last; ++first) {
        if (pred(*first))
            return true;
    }
    return false;
}
```

```
template <class I>
concept InputIt = requires(I i1, I i2) {
    { ++i1 } -> I;
    { i1 != i2 } -> bool;
    { *i1 };
};
```

```
template <class Iterator, class Predicate>
concept PredicateForIterator =
requires(Iterator p, Predicate pred) {
    { pred(*p) } -> bool;
};
```


DYNAMIC
OFF

| Algorithms

*An **algorithm** is a sequence of instructions, typically to solve a class of problems or perform a computation.*

- Algorithms are everywhere.
- Explicit naming improves understanding.
- Explicit naming makes separate testing possible.

| Embedded domain

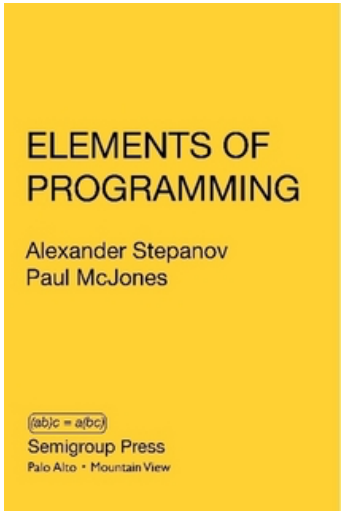
> *"We don't solve problems, we approximate solutions."*
>
> Sean Parent, A possible future of software development, 2007

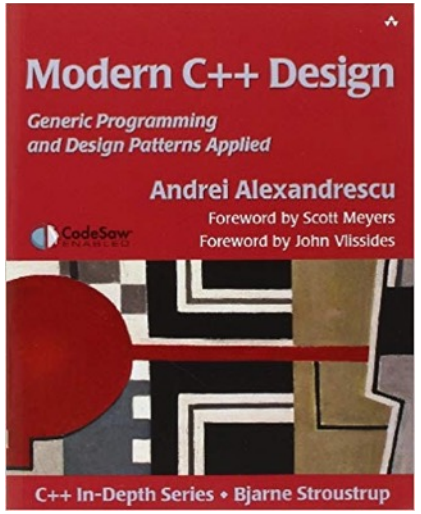
Generic Programming

- reliable solutions
- efficient solutions
- flexible solutions

| C++20 concepts

- Many applications in the embedded domain.
- Make C++ Templates user friendly.





https://www.youtube.com/playlist?list=PLroqQtpVoIg3tbtucA8DNhUV2PxOd4YOI

Generic Programming*

David R. Musser†
Rensselaer Polytechnic Institute
Computer Science Department
Amos Eaton Hall
Troy, New York 12180

Alexander A. Stepanov
Hewlett–Packard Laboratories
Software Technology Laboratory
Post Office Box 10490
Palo Alto, California 94303–0969

Abstract

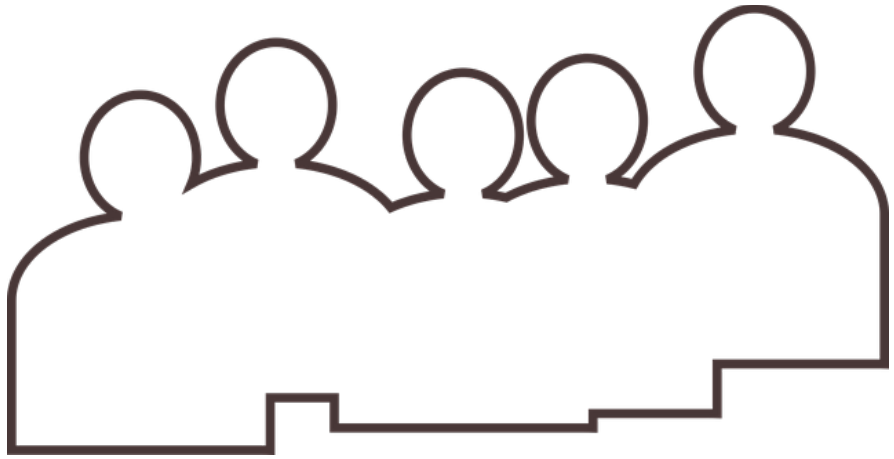
Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software. For example, a class of generic sorting algorithms can be defined which work with finite sequences but which can be instantiated in different ways to produce algorithms working on arrays or linked lists.

Four kinds of abstraction—data, algorithmic, structural, and representational—are discussed, with examples of their use in building an Ada library of software components. The main topic discussed is generic algorithms and an approach to their formal specification and verification, with illustration in terms of a partitioning algorithm such as is used in the quicksort algorithm. It is argued that generically programmed software component libraries offer important advantages for achieving software productivity and reliability.

*This paper was presented at the First International Joint Conference of ISSAC-88 and AAECC-6, Rome, Italy, July 4-8, 1988. (ISSAC stands for International Symposium on Symbolic and Algebraic Computation and AAECC for Applied Algebra, Algebraic Algorithms, and Error Correcting Codes). It was published in *Lecture Notes in Computer Science* 358, Springer-Verlag, 1989, pp. 13-25.

†The first author's work was sponsored in part through a subcontract from Computational Logic, Inc., which was sponsored in turn by the Defense Advanced Research Projects Agency, ARPA order 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic., Inc.

Thanks for listening!



embeff
BETTER EMBEDDED.

Making high quality embedded software commonplace.

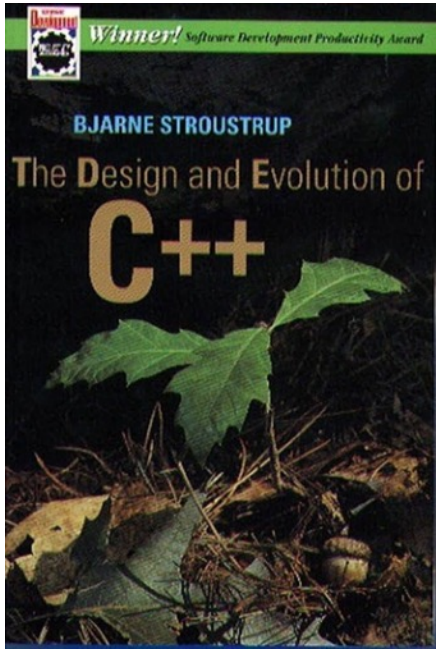
Embedded Software

- Real time
- Many target systems
- Low level code
- Quality standards

Classical Software

- Automation
- Fast feedback
- Reusable libraries
- Sophisticated tooling

| C++ user friendly?



It is more important to allow a useful feature than to prevent every misuse.

```
template <typename T,
          typename = typename
std::enable_if_t<std::is_arithmetic_v<T>>>
T product(T const t1, T const t2) {
    return t1 * t2;
}
```

```
template <numeric T>
T product(T const t1, T const t2) {
    return t1 * t2;
}
```

https://mariusbancila.ro/blog/2019/10/04/concepts-versus-sfinae-based-constraints/