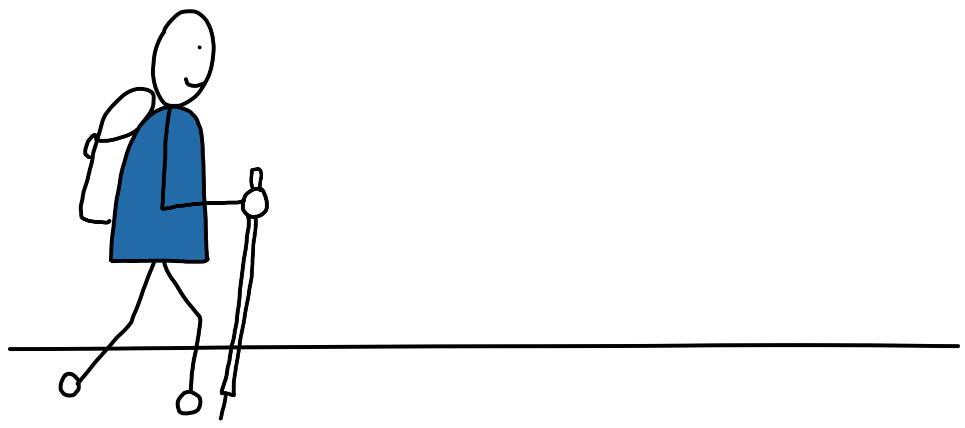
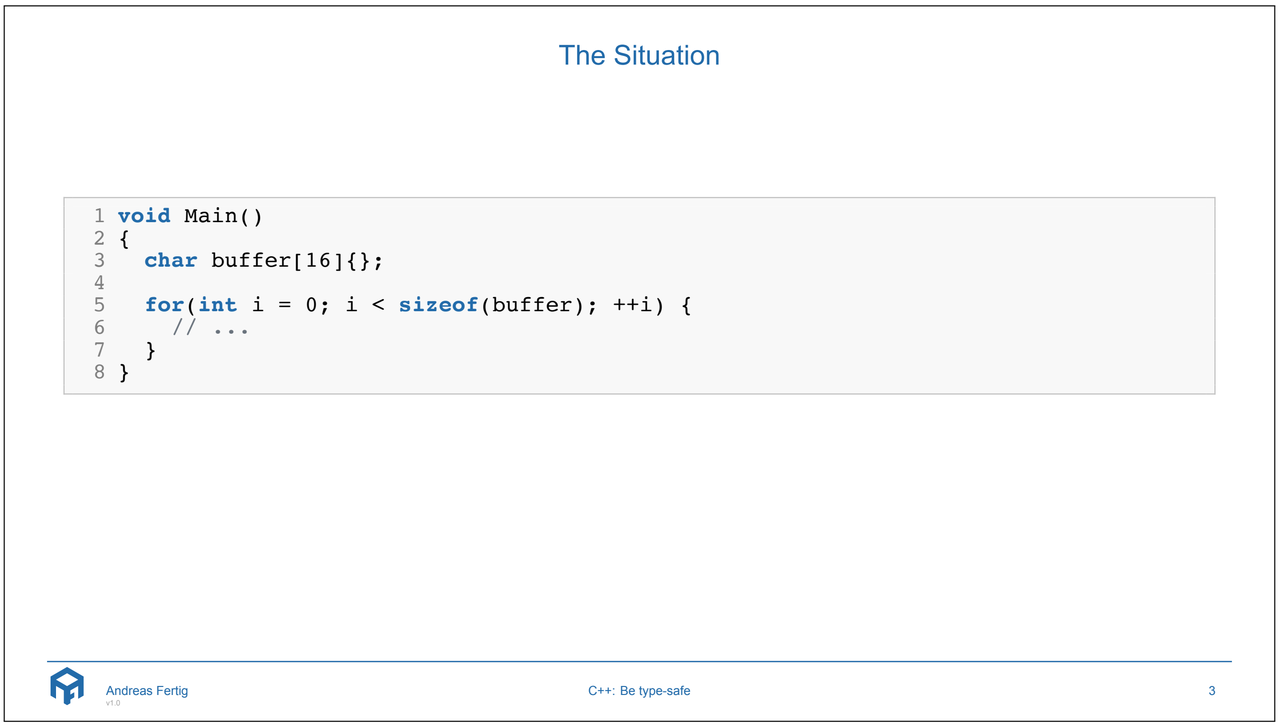
C++: Be type-safe

The journey of determining the number of elements in an array

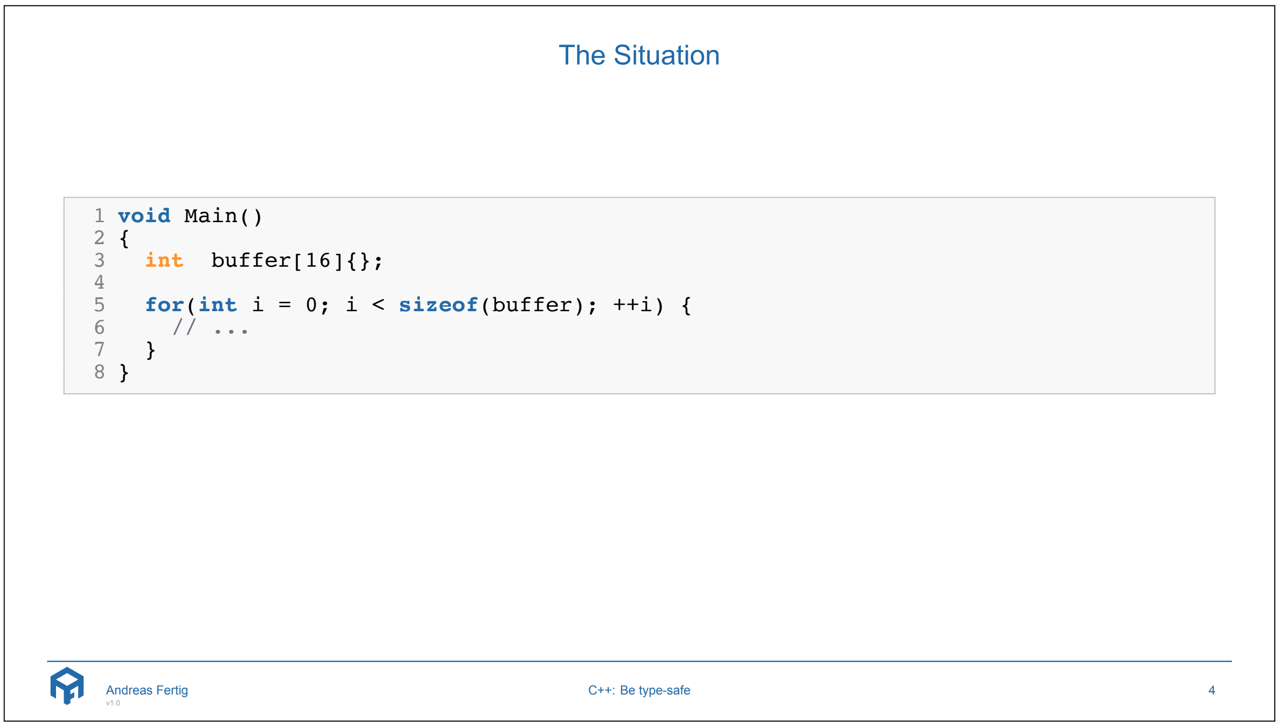
Andreas Fertig
https://www.AndreasFertig.Info
post@AndreasFertig.Info
@Andreas__Fertig





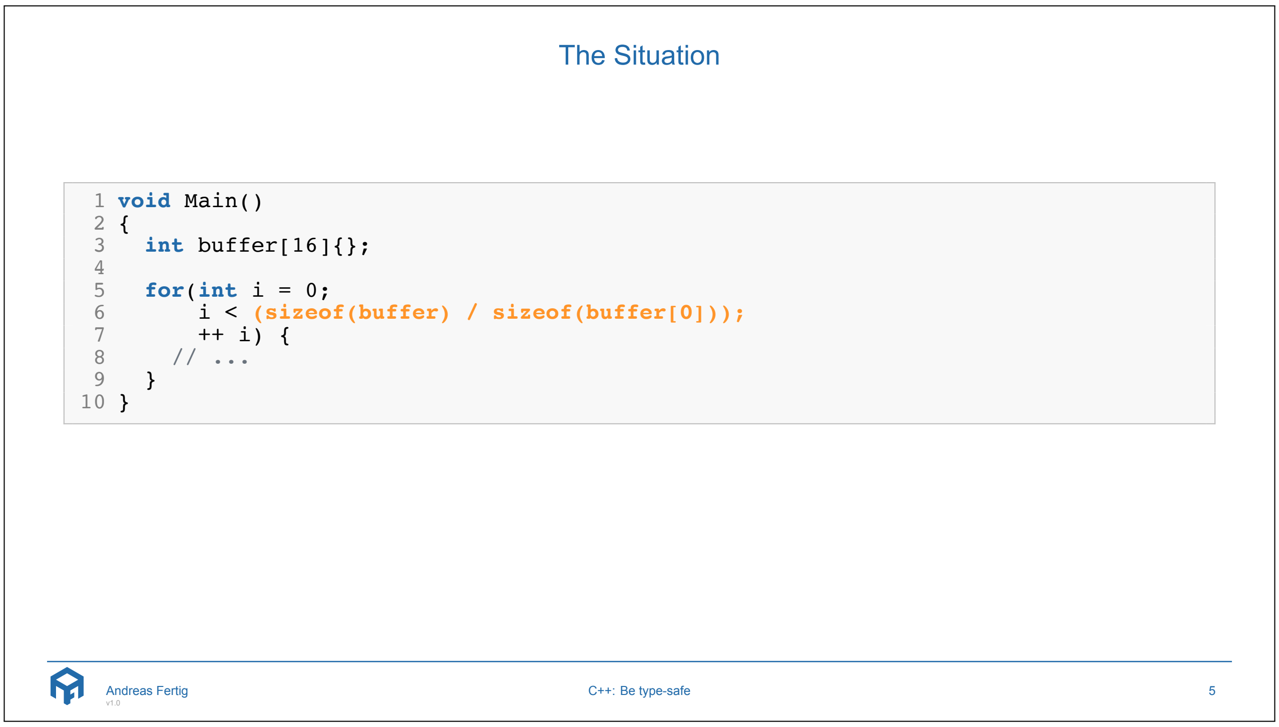
The Situation

```cpp
void Main()
{
  char buffer[16]{};

  for(int i = 0; i < sizeof(buffer); ++i) {
    // ...
  }
}
```



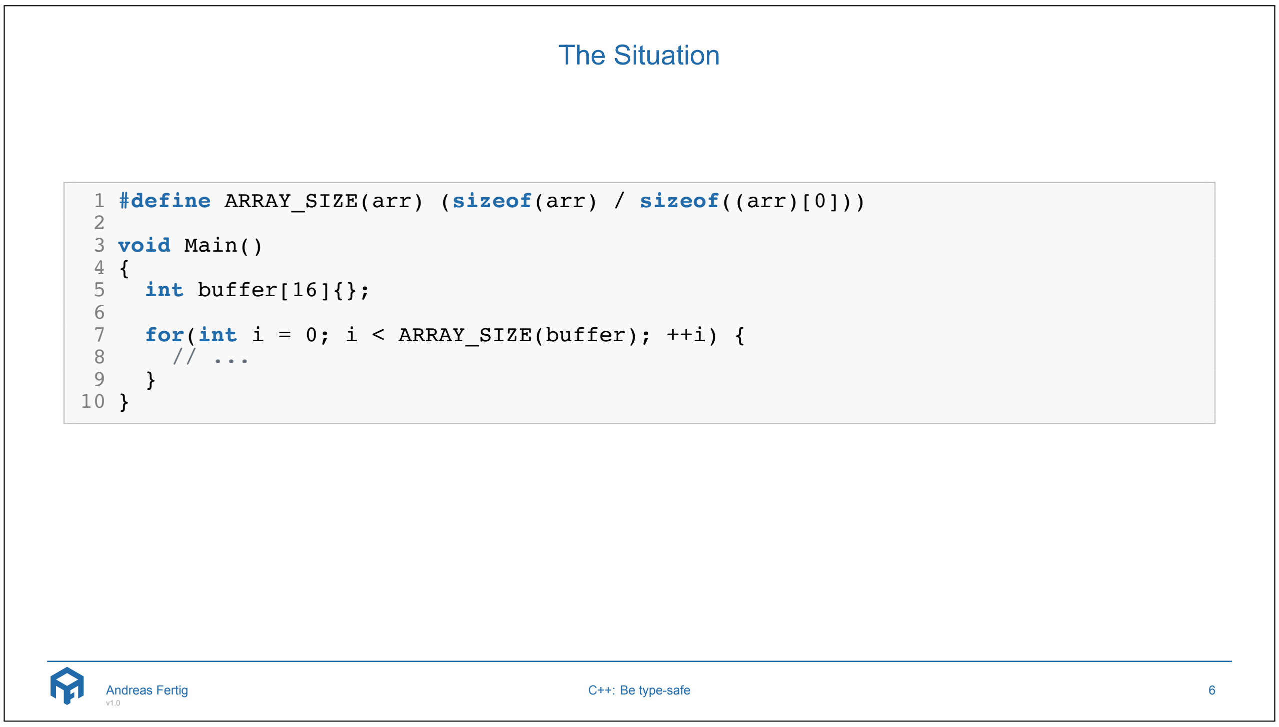
The Situation

```cpp
void Main()
{
  int  buffer[16]{};

  for(int i = 0; i < sizeof(buffer); ++i) {
    // ...
  }
}
```



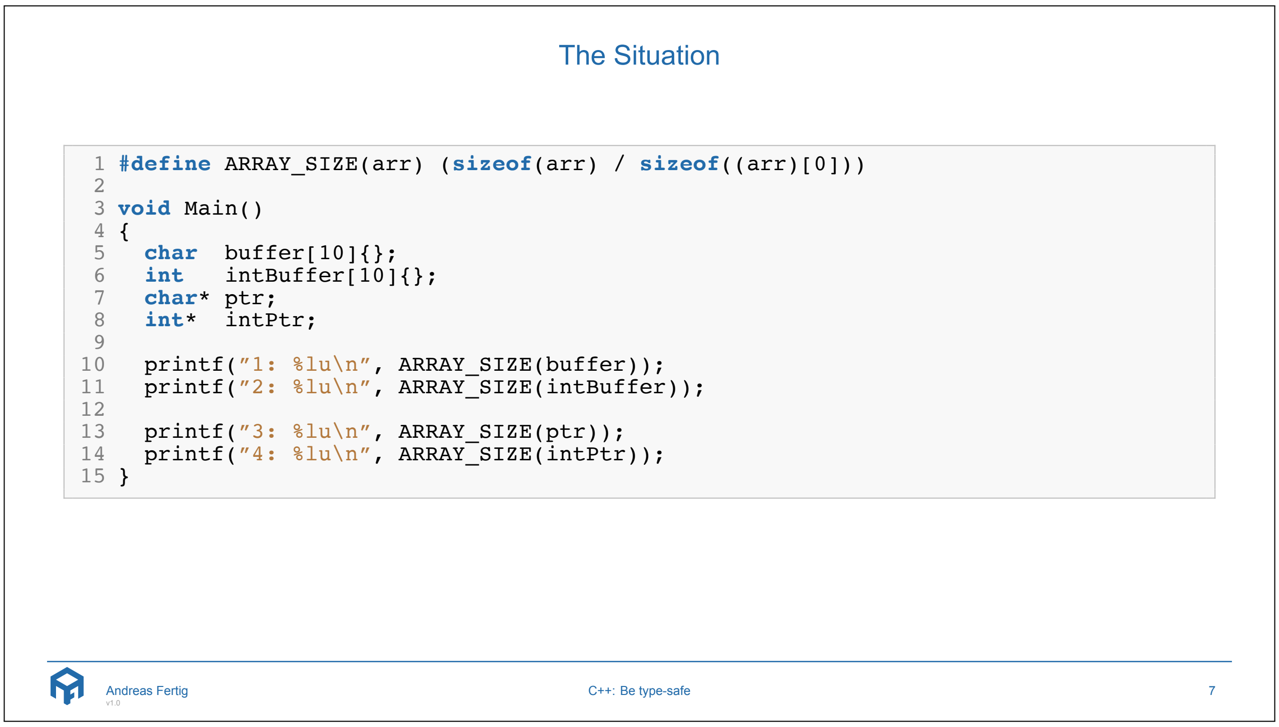
The Situation

```cpp
void Main()
{
  int buffer[16]{};

  for(int i = 0;
      i < (sizeof(buffer) / sizeof(buffer[0]));
      ++ i) {
    // ...
  }
}
```



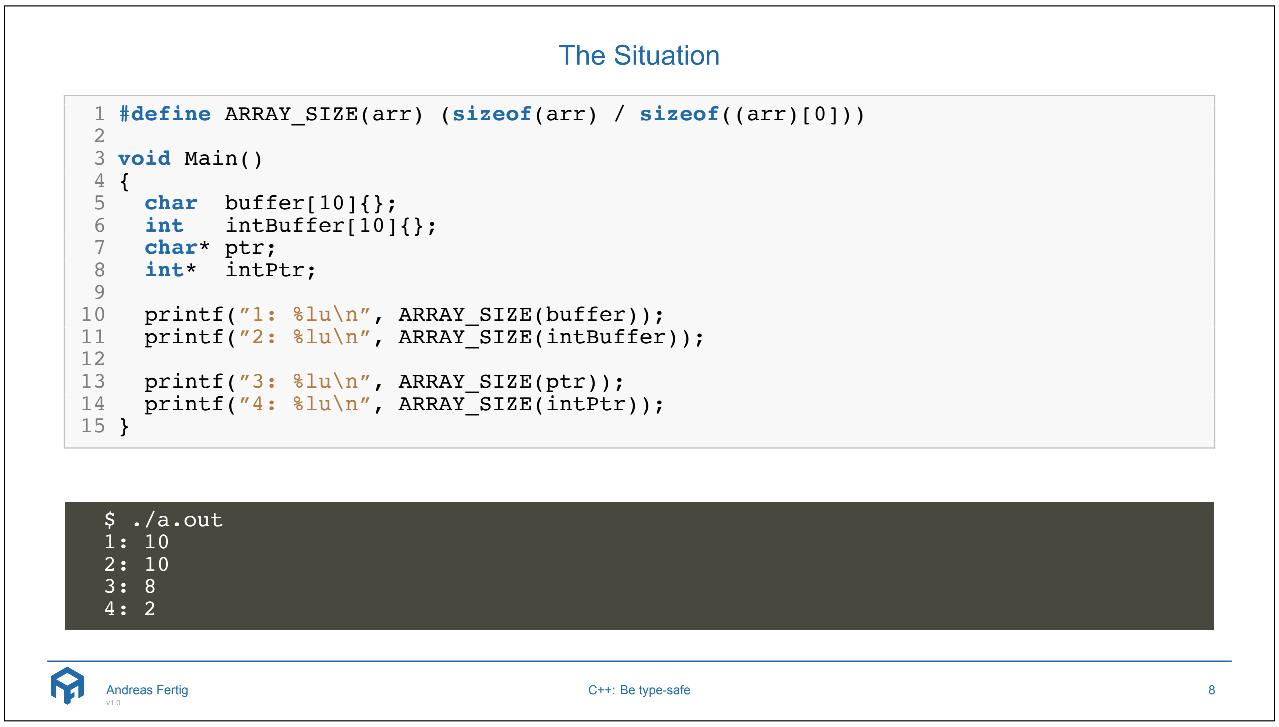
The Situation

```cpp
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

void Main()
{
  int buffer[16]{};

  for(int i = 0; i < ARRAY_SIZE(buffer); ++i) {
    // ...
  }
}
```



The Situation

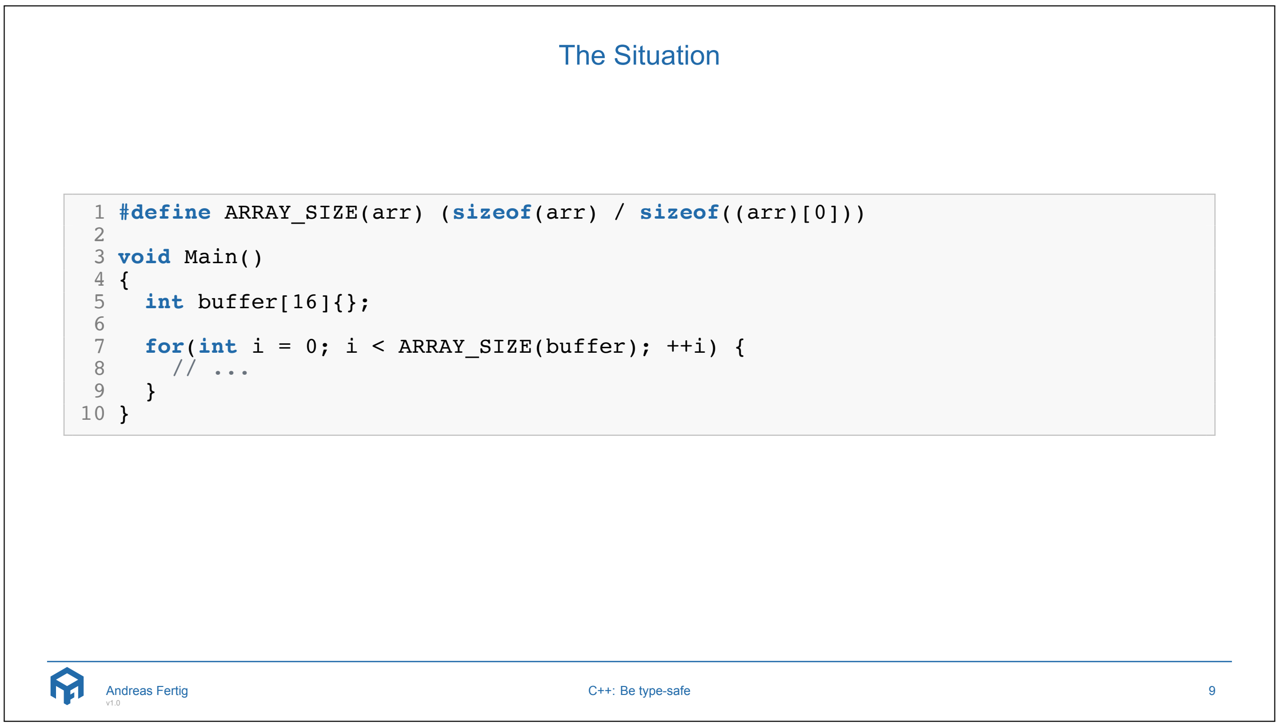
```cpp
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

void Main()
{
  char  buffer[10]{};
  int   intBuffer[10]{};
  char* ptr;
  int*  intPtr;

  printf("1: %lu\n", ARRAY_SIZE(buffer));
  printf("2: %lu\n", ARRAY_SIZE(intBuffer));

  printf("3: %lu\n", ARRAY_SIZE(ptr));
  printf("4: %lu\n", ARRAY_SIZE(intPtr));
}
```



The Situation

```cpp
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

void Main()
{
  char  buffer[10]{};
  int   intBuffer[10]{};
  char* ptr;
  int*  intPtr;

  printf("1: %lu\n", ARRAY_SIZE(buffer));
  printf("2: %lu\n", ARRAY_SIZE(intBuffer));

  printf("3: %lu\n", ARRAY_SIZE(ptr));
  printf("4: %lu\n", ARRAY_SIZE(intPtr));
}
```

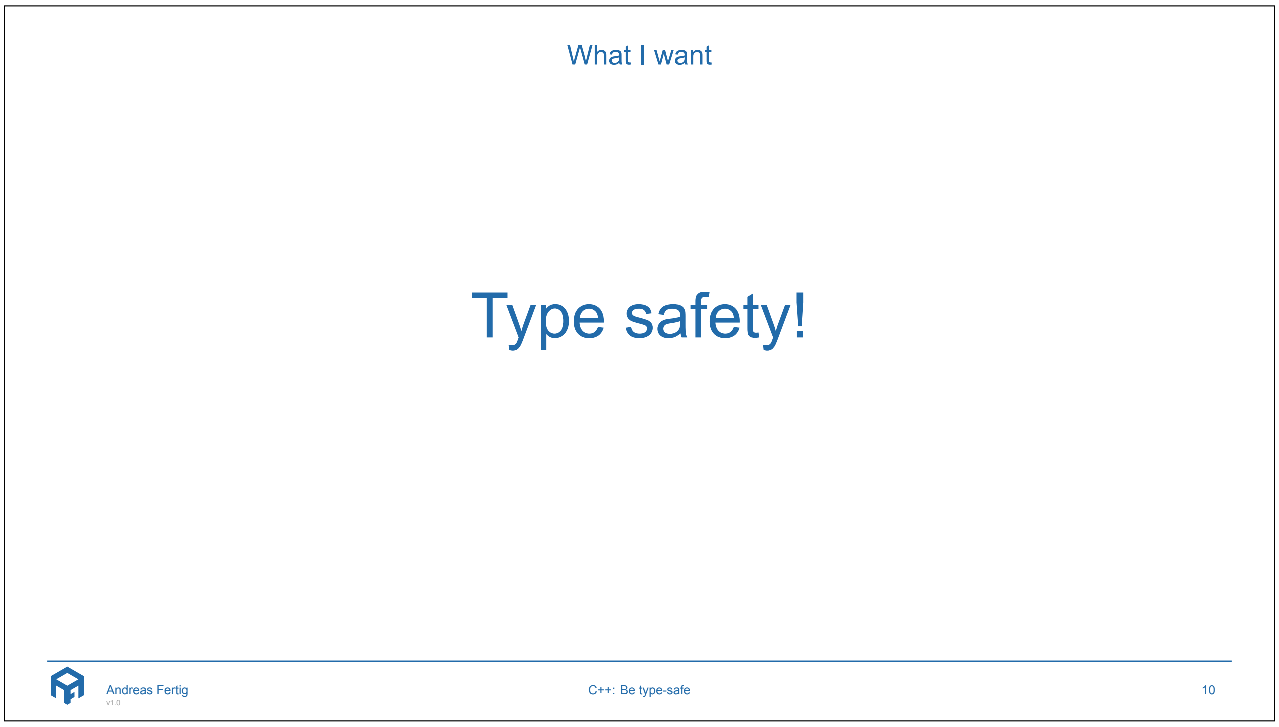
```
$ ./a.out
1: 10
2: 10
3: 8
4: 2
```



The Situation

```cpp
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))

void Main()
{
  int buffer[16]{};

  for(int i = 0; i < ARRAY_SIZE(buffer); ++i) {
    // ...
  }
}
```

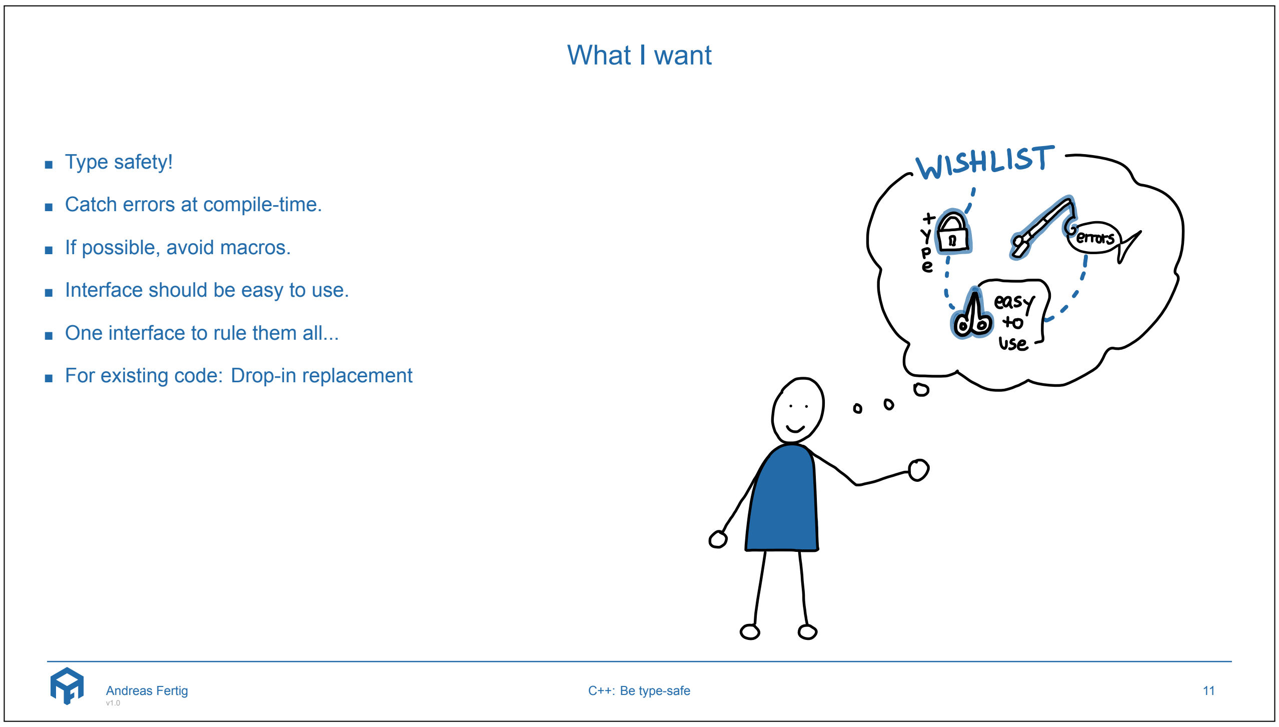
Andreas Fertig v1.0 C++: Be type-safe 9



What I want

Type safety!

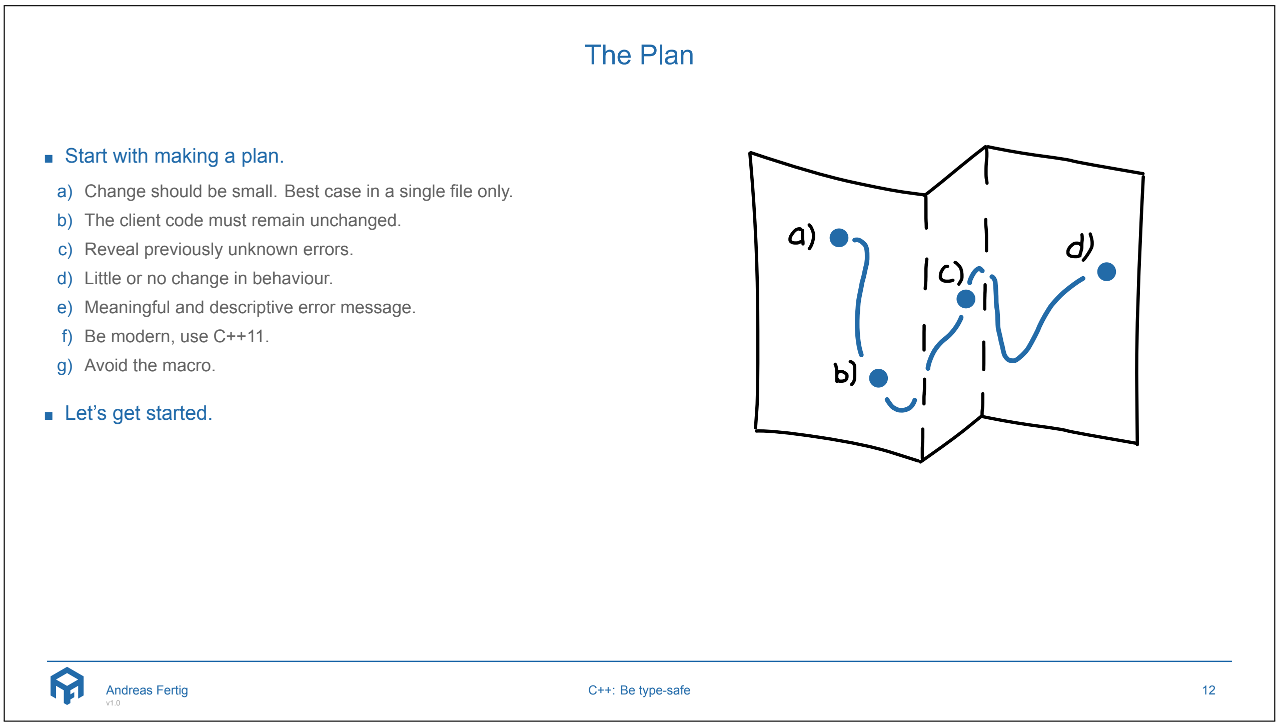
Andreas Fertig v1.0 C++: Be type-safe 10



What I want

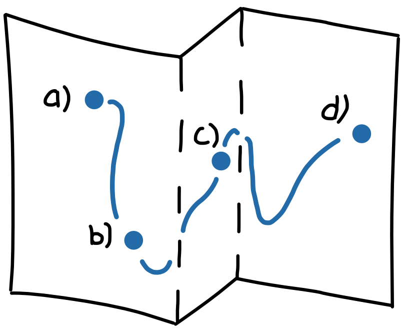
- Type safety!
- Catch errors at compile-time.
- If possible, avoid macros.
- Interface should be easy to use.
- One interface to rule them all...
- For existing code: Drop-in replacement

Andreas Fertig v1.0 C++: Be type-safe 11

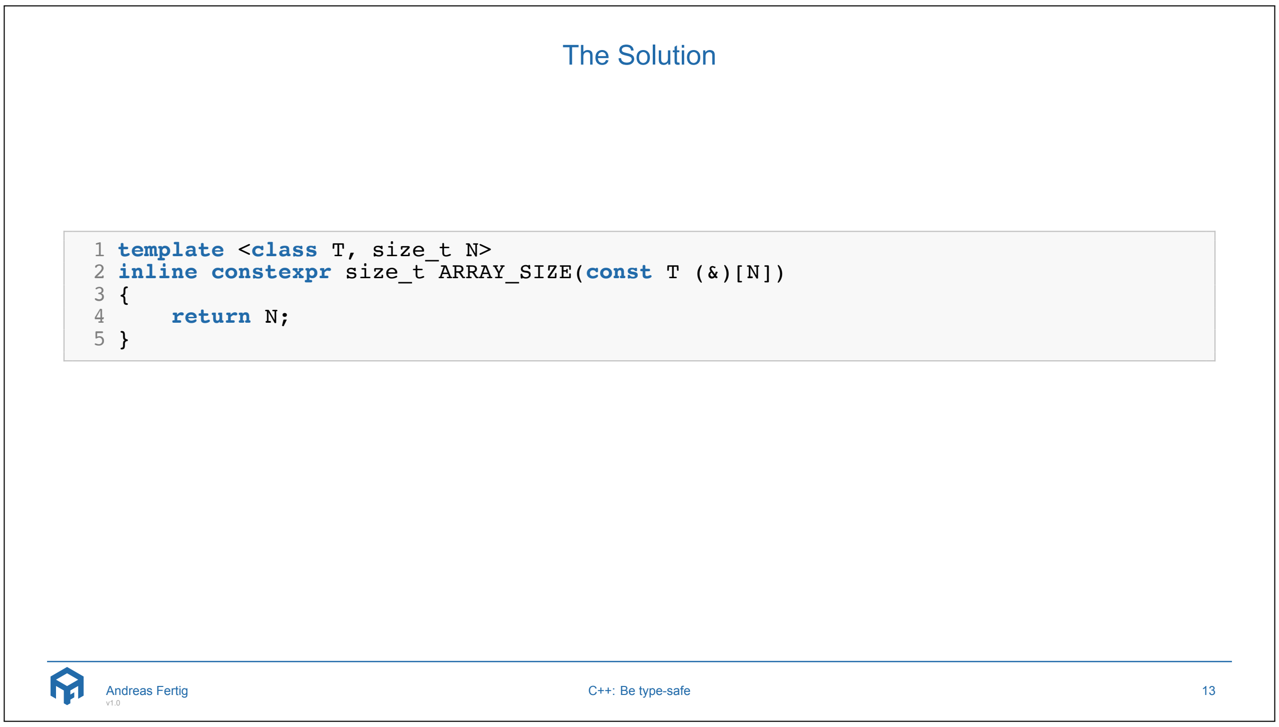


The Plan

- Start with making a plan.
  a) Change should be small. Best case in a single file only.
  b) The client code must remain unchanged.
  c) Reveal previously unknown errors.
  d) Little or no change in behaviour.
  e) Meaningful and descriptive error message.
  f) Be modern, use C++11.
  g) Avoid the macro.
- Let's get started.

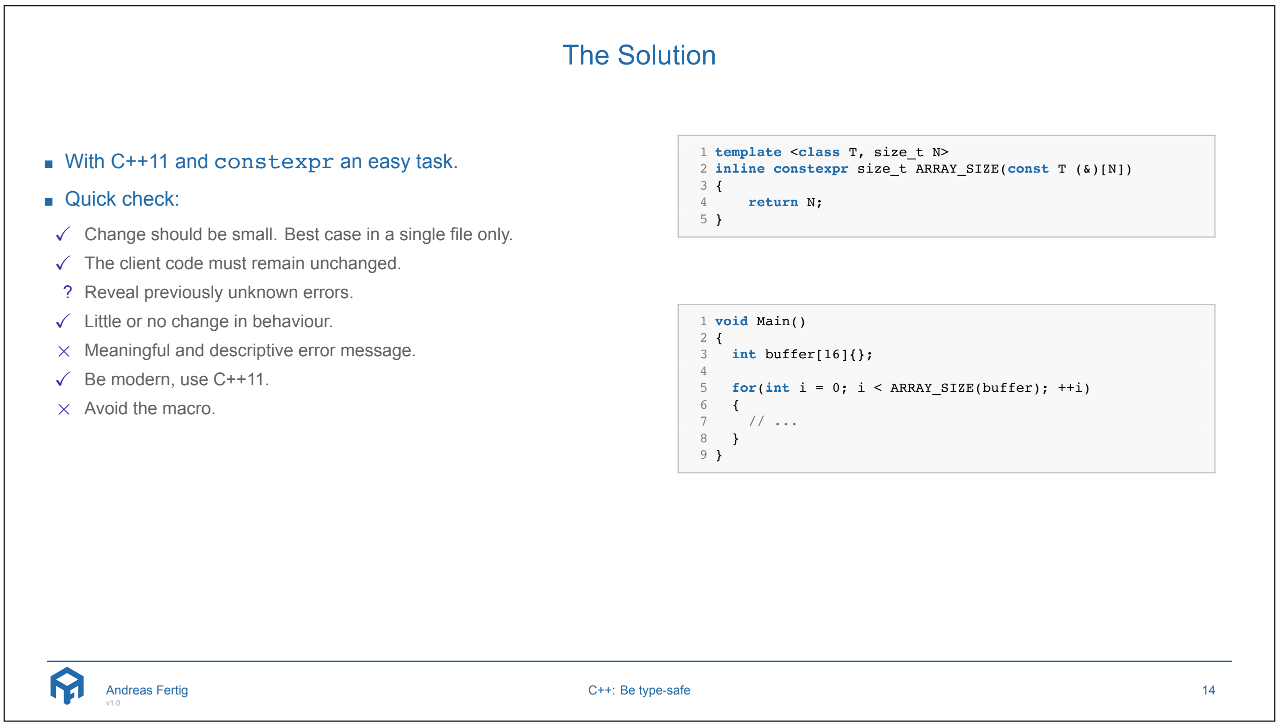


Andreas Fertig v1.0 C++: Be type-safe 12



The Solution

```cpp
template <class T, size_t N>
inline constexpr size_t ARRAY_SIZE(const T (&)[N])
{
    return N;
}
```



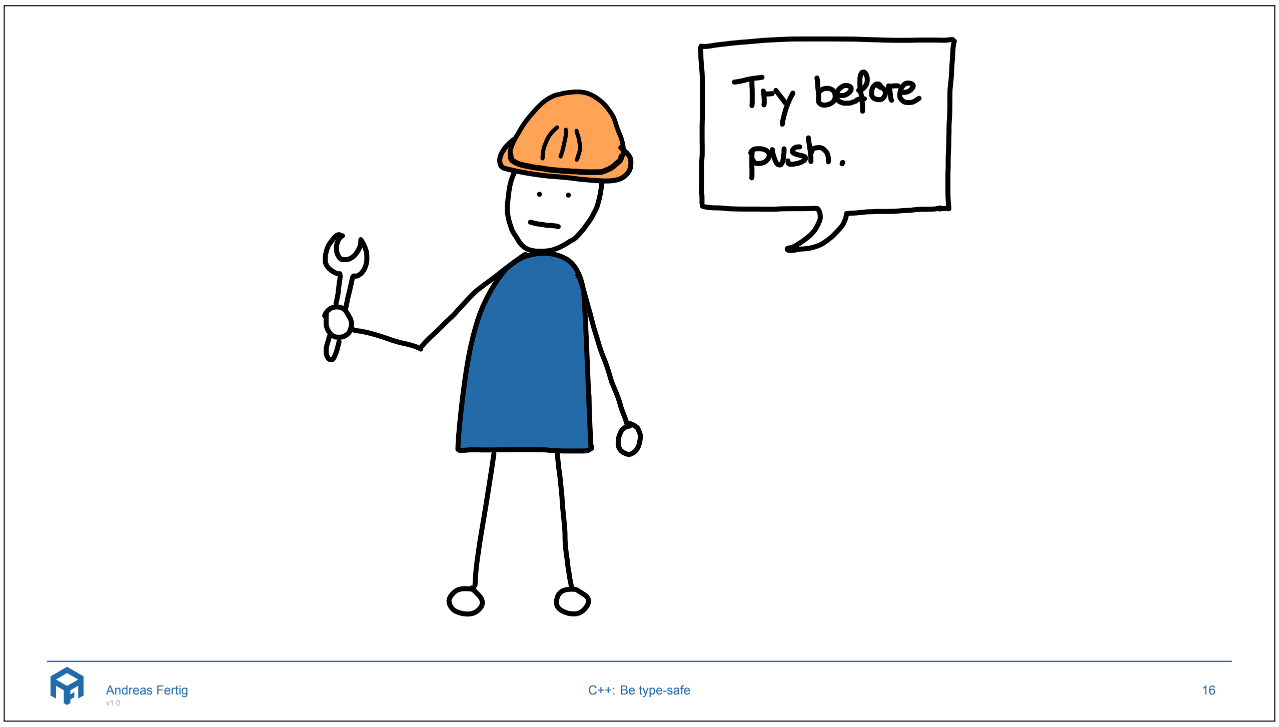
The Solution

- With C++11 and `constexpr` an easy task.
- Quick check:
  - ✓ Change should be small. Best case in a single file only.
  - ✓ The client code must remain unchanged.
  - ? Reveal previously unknown errors.
  - ✓ Little or no change in behaviour.
  - × Meaningful and descriptive error message.
  - ✓ Be modern, use C++11.
  - × Avoid the macro.

```cpp
template <class T, size_t N>
inline constexpr size_t ARRAY_SIZE(const T (&)[N])
{
    return N;
}
```

```cpp
void Main()
{
  int buffer[16]{};

  for(int i = 0; i < ARRAY_SIZE(buffer); ++i)
  {
    // ...
  }
}
```

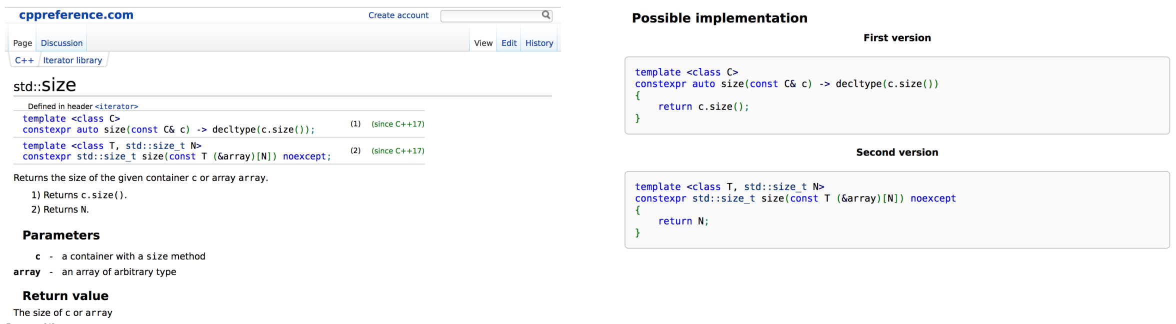
Andreas Fertig
v1.0
C++: Be type-safe
15



Andreas Fertig
v1.0
C++: Be type-safe
16

```cpp
template <class T, size_t N>
inline constexpr size_t ARRAY_SIZE(const T (&)[N])
{
    return N;
}
```

std::size



std::size

Defined in header `<iterator>`

```cpp
template <class C>
constexpr auto size(const C& c) -> decltype(c.size());              (1)  (since C++17)
template <class T, std::size_t N>
constexpr std::size_t size(const T (&array)[N]) noexcept;           (2)  (since C++17)
```

Returns the size of the given container c or array `array`.

1) Returns `c.size()`.
2) Returns N.

Parameters

c	-	a container with a `size` method
array	-	an array of arbitrary type

Return value

The size of c or `array`

Possible implementation

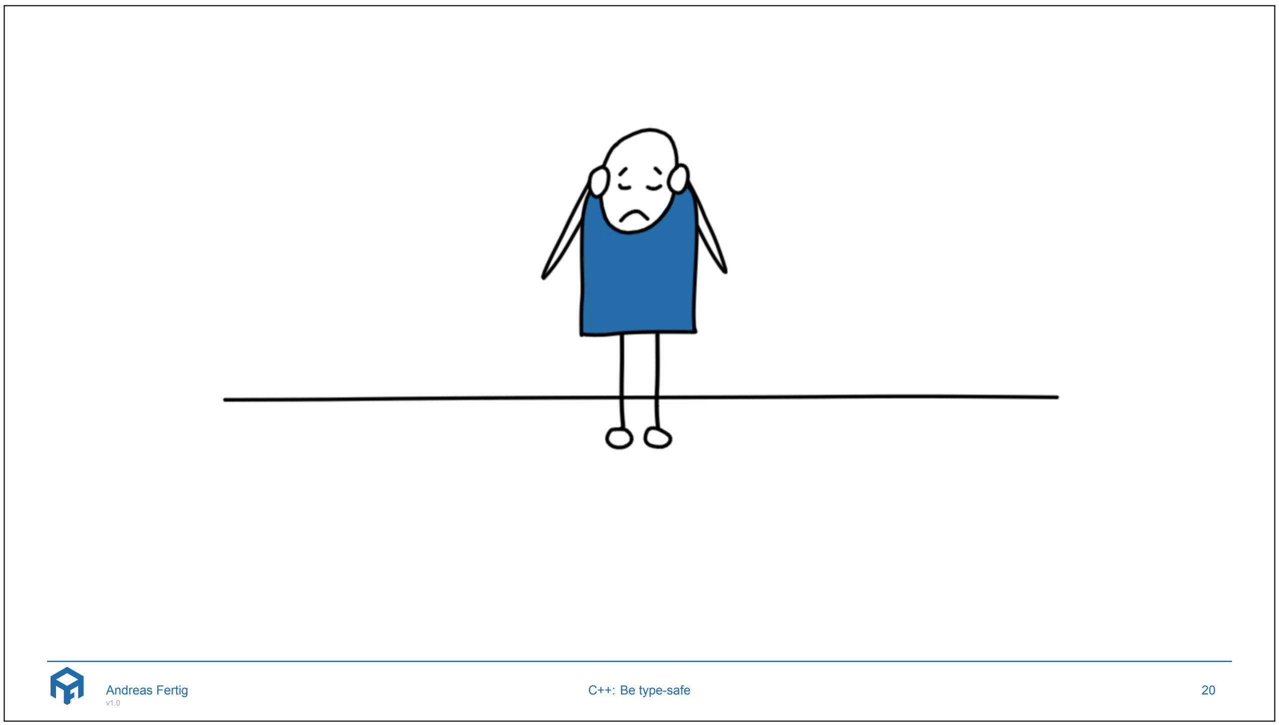
First version

```cpp
template <class C>
constexpr auto size(const C& c) -> decltype(c.size())
{
    return c.size();
}
```

Second version

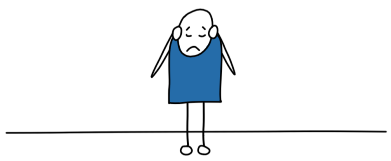
```cpp
template <class T, std::size_t N>
constexpr std::size_t size(const T (&array)[N]) noexcept
{
    return N;
}
```

Source: [1]



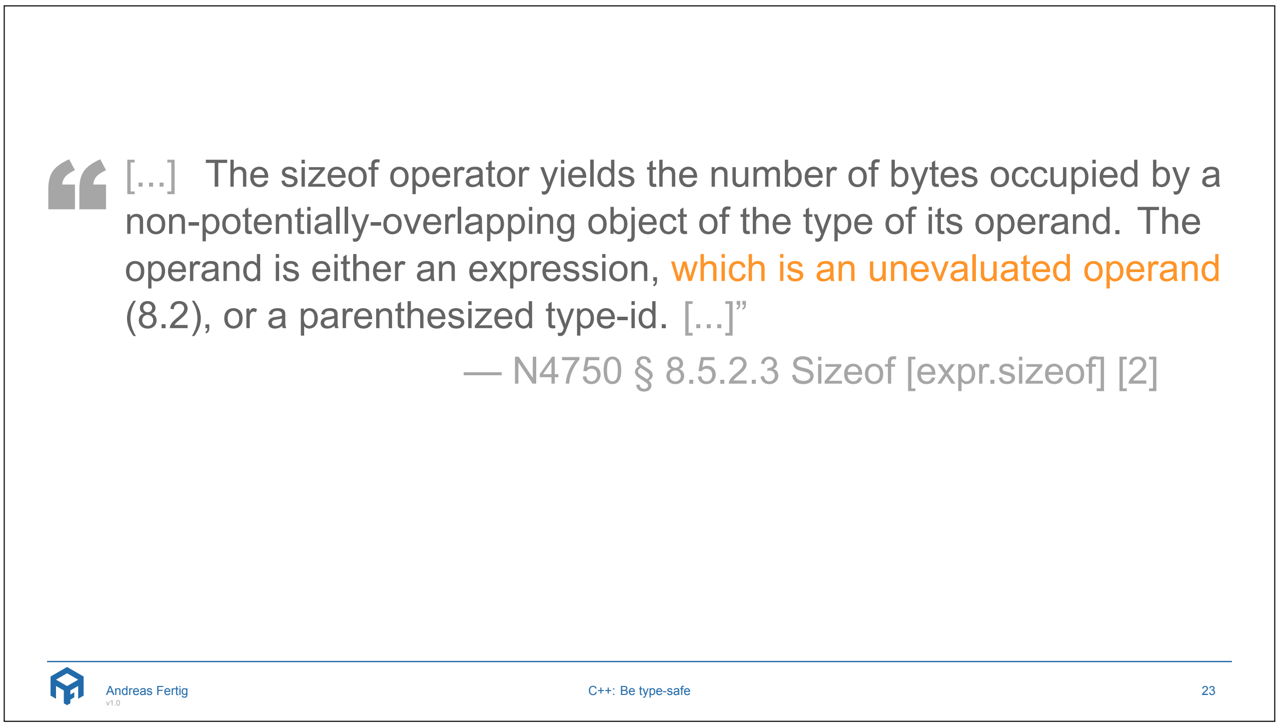
Arg

```
x.cpp:16:4: error: cannot form pointer to deduced class template specialization /
    type
  X* x;
   ^
x.cpp:16:6: error: declaration of variable 'x' with deduced type 'X *' requires /
    an initializer
  X* x;
     ^
2 errors generated.
```



Arg

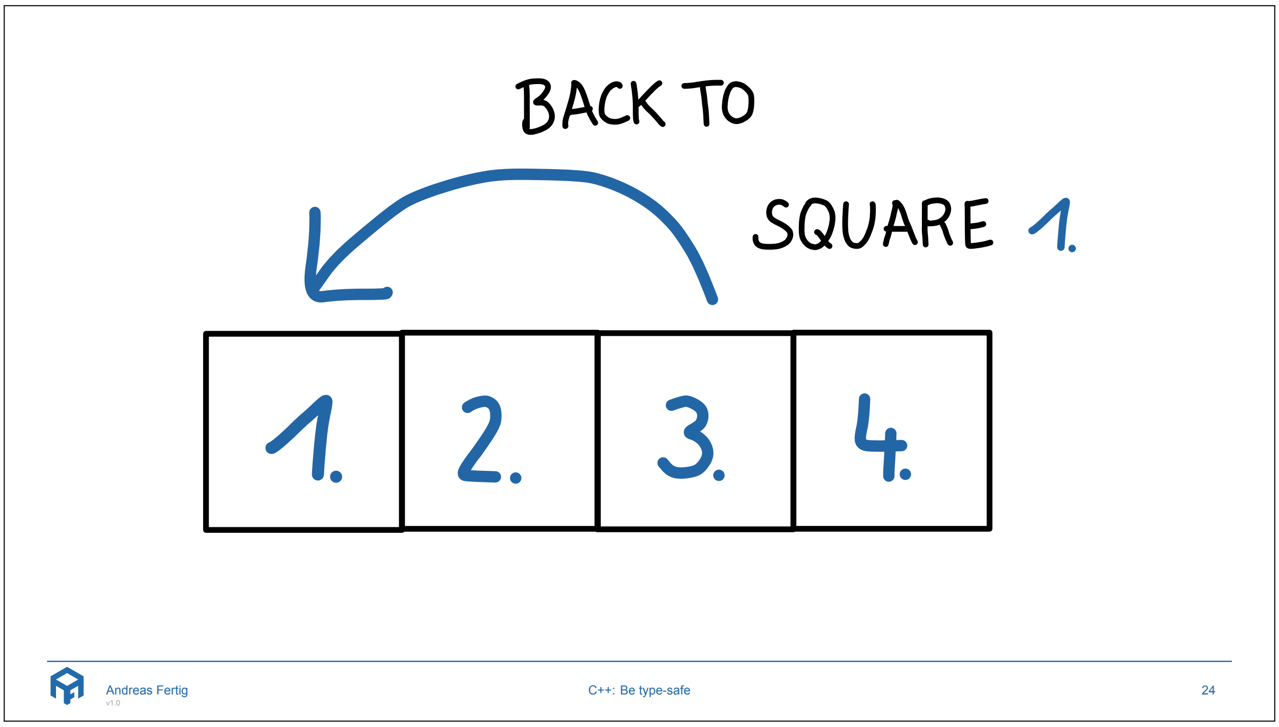
```cpp
struct X
{
  char v[12];
};

void Main()
{
  X* x;
  static_assert(ARRAY_SIZE(x->v) == 12, "Wrong array length");
}
```



“[...] The sizeof operator yields the number of bytes occupied by a non-potentially-overlapping object of the type of its operand. The operand is either an expression, which is an unevaluated operand (8.2), or a parenthesized type-id. [...]”

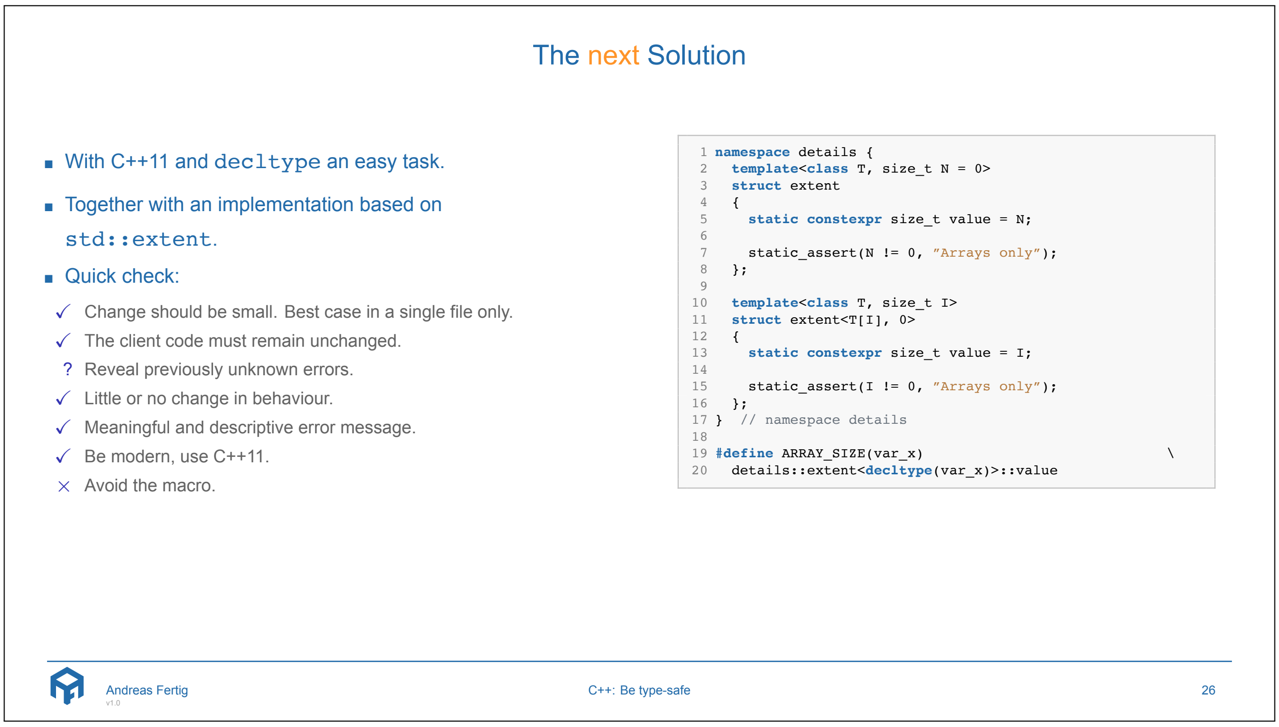
— N4750 § 8.5.2.3 Sizeof [expr.sizeof] [2]

Andreas Fertig v1.0 C++: Be type-safe 23



Andreas Fertig v1.0 C++: Be type-safe 24

The next Solution

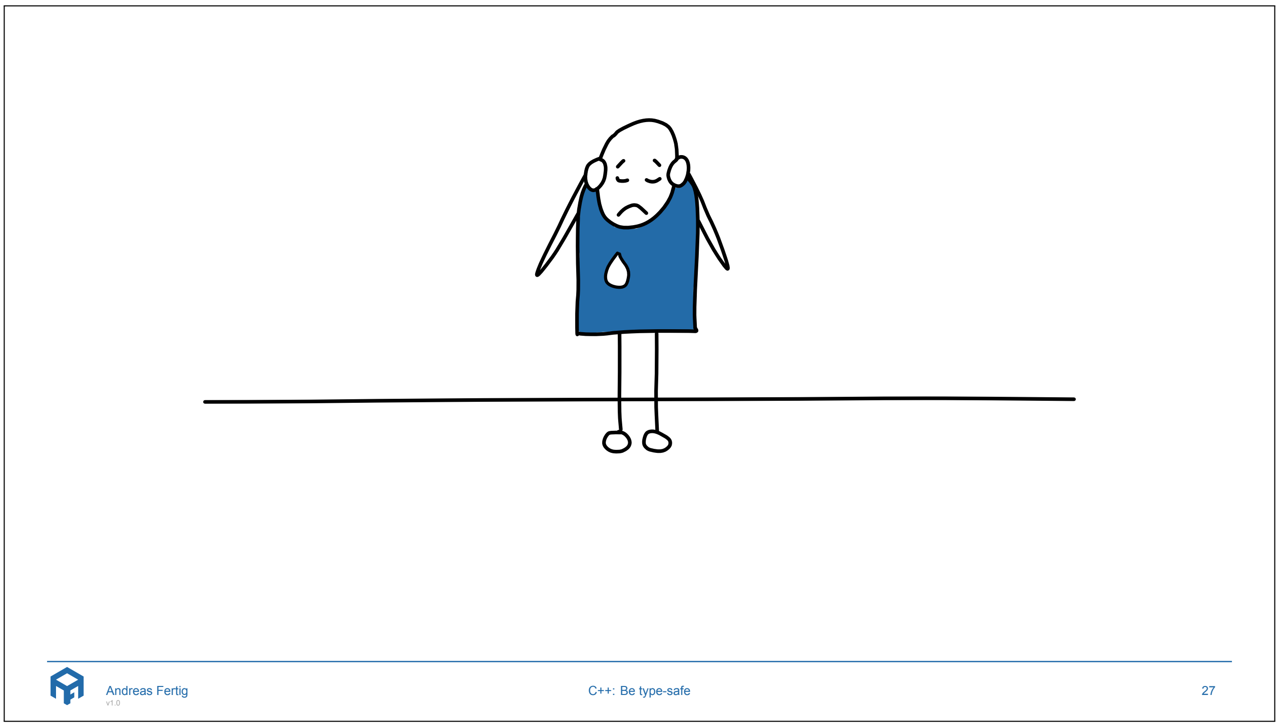


The next Solution

- With C++11 and `decltype` an easy task.
- Together with an implementation based on `std::extent`.
- Quick check:

✓ Change should be small. Best case in a single file only.
✓ The client code must remain unchanged.
? Reveal previously unknown errors.
✓ Little or no change in behaviour.
✓ Meaningful and descriptive error message.
✓ Be modern, use C++11.
× Avoid the macro.

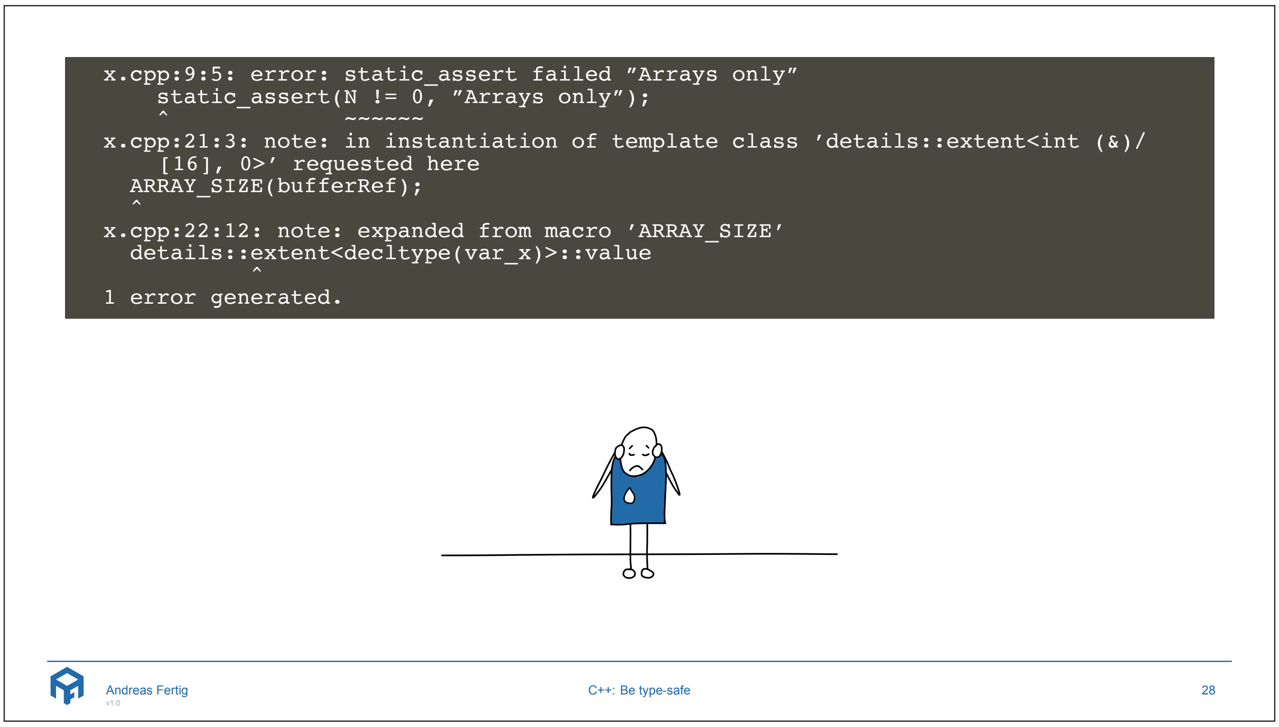
```cpp
namespace details {
  template<class T, size_t N = 0>
  struct extent
  {
    static constexpr size_t value = N;

    static_assert(N != 0, "Arrays only");
  };

  template<class T, size_t I>
  struct extent<T[I], 0>
  {
    static constexpr size_t value = I;

    static_assert(I != 0, "Arrays only");
  };
}  // namespace details

#define ARRAY_SIZE(var_x)                           \
  details::extent<decltype(var_x)>::value
```



Andreas Fertig
v1.0

C++: Be type-safe

27

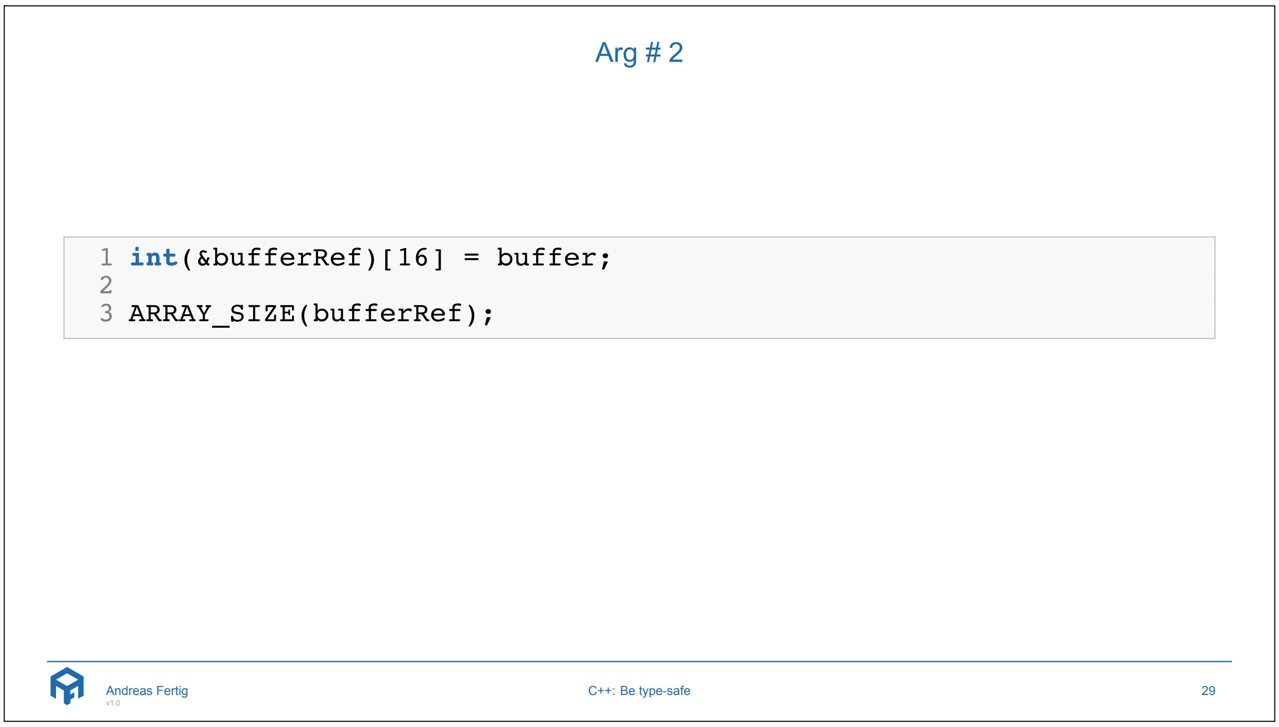


```
x.cpp:9:5: error: static_assert failed ”Arrays only”
    static_assert(N != 0, ”Arrays only”);
    ^             ~~~~~~
x.cpp:21:3: note: in instantiation of template class ’details::extent<int (&)/
    [16], 0>’ requested here
  ARRAY_SIZE(bufferRef);
  ^
x.cpp:22:12: note: expanded from macro ’ARRAY_SIZE’
  details::extent<decltype(var_x)>::value
           ^
1 error generated.
```

Andreas Fertig
v1.0

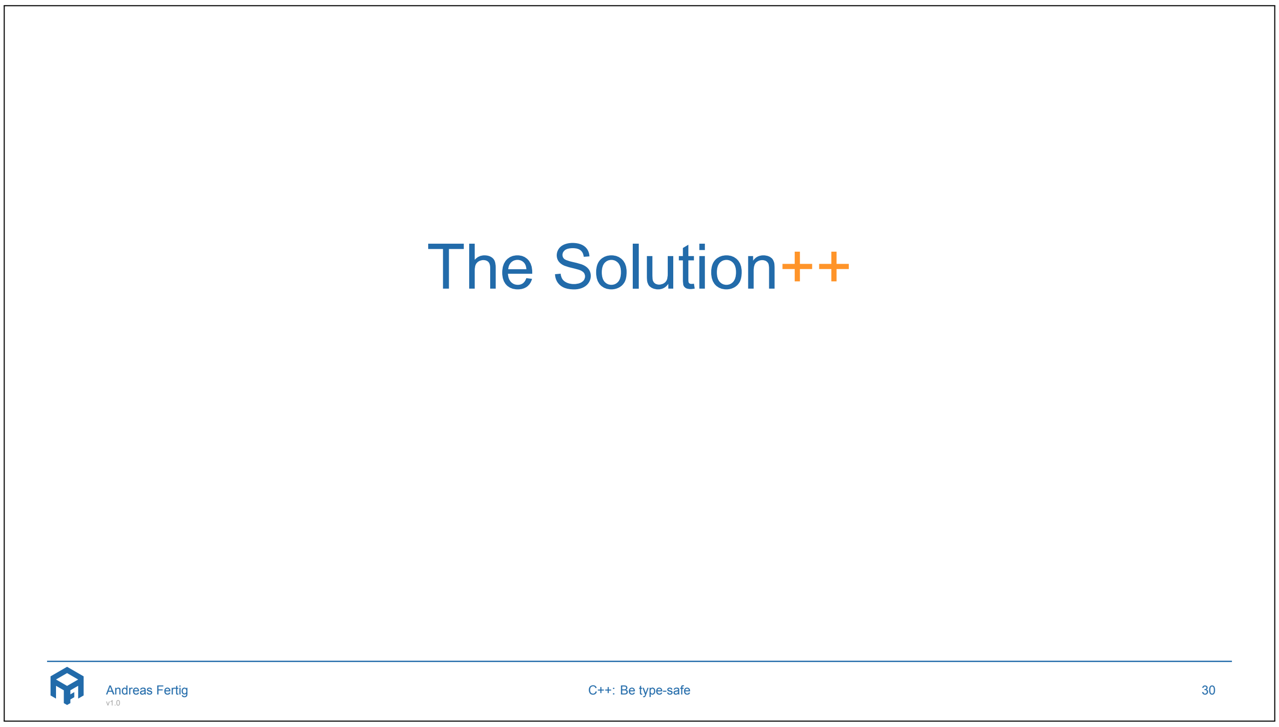
C++: Be type-safe

28

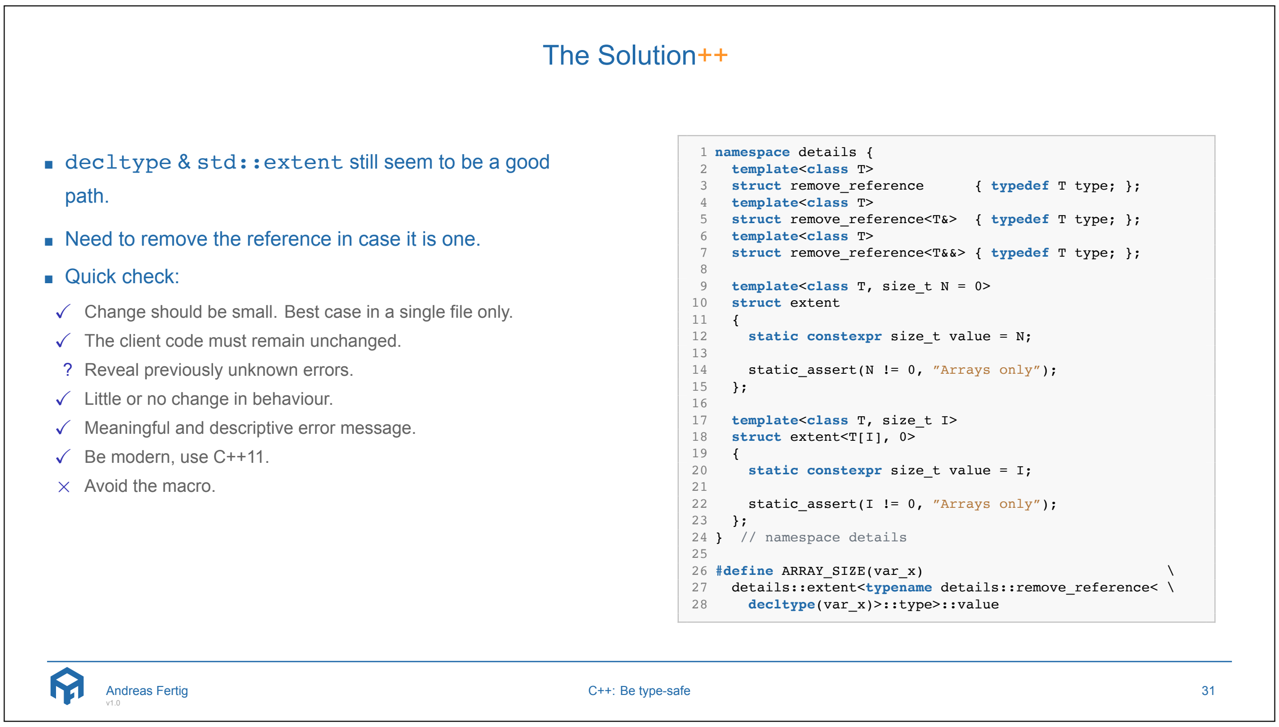


Arg # 2

```cpp
int(&bufferRef)[16] = buffer;

ARRAY_SIZE(bufferRef);
```



The Solution++

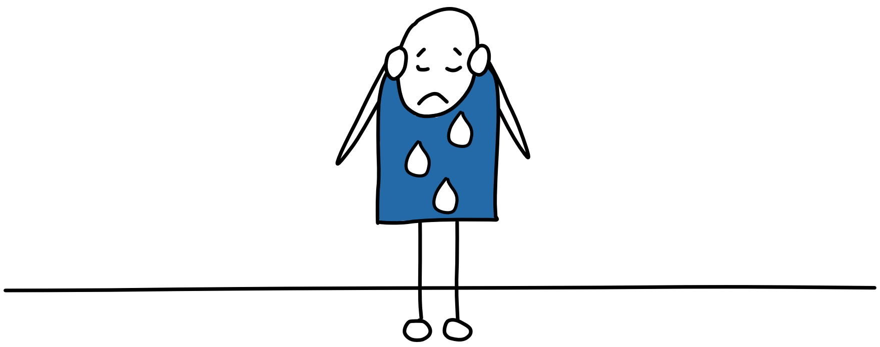


The Solution++

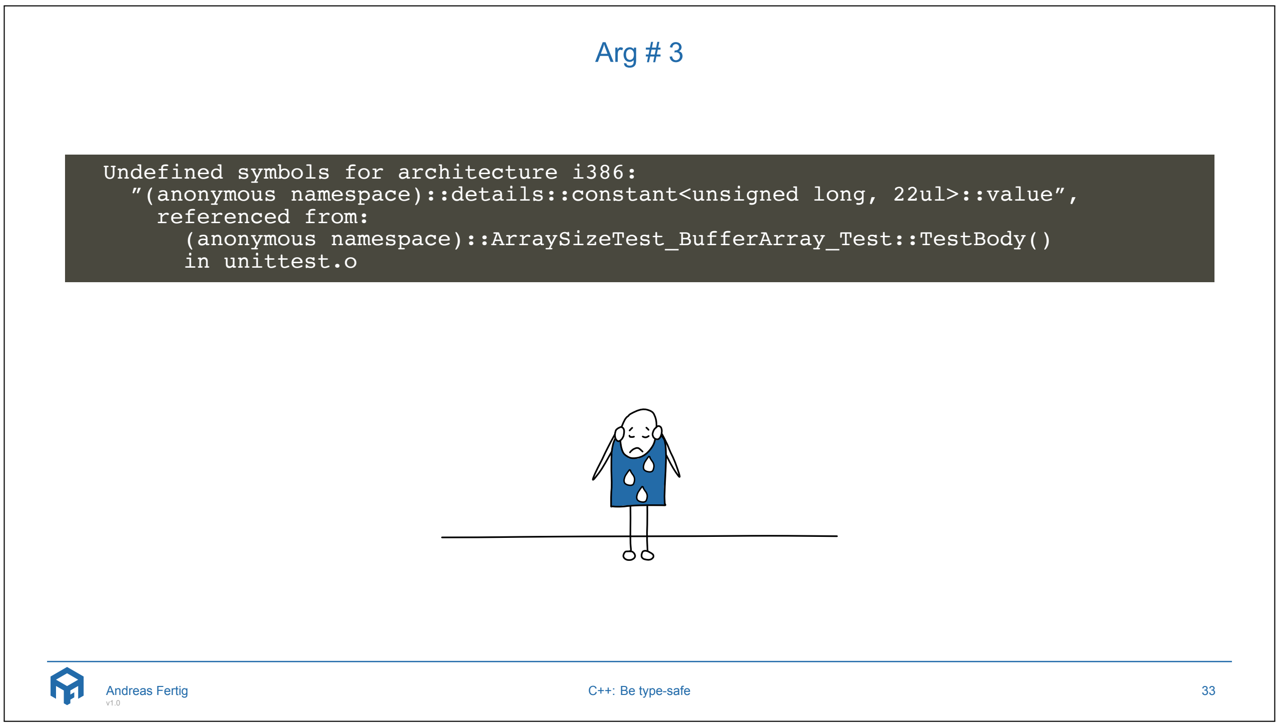
- `decltype` & `std::extent` still seem to be a good path.
- Need to remove the reference in case it is one.
- Quick check:
  - ✓ Change should be small. Best case in a single file only.
  - ✓ The client code must remain unchanged.
  - ? Reveal previously unknown errors.
  - ✓ Little or no change in behaviour.
  - ✓ Meaningful and descriptive error message.
  - ✓ Be modern, use C++11.
  - × Avoid the macro.

```cpp
namespace details {
  template<class T>
  struct remove_reference      { typedef T type; };
  template<class T>
  struct remove_reference<T&>  { typedef T type; };
  template<class T>
  struct remove_reference<T&&> { typedef T type; };

  template<class T, size_t N = 0>
  struct extent
  {
    static constexpr size_t value = N;

    static_assert(N != 0, "Arrays only");
  };

  template<class T, size_t I>
  struct extent<T[I], 0>
  {
    static constexpr size_t value = I;

    static_assert(I != 0, "Arrays only");
  };
}  // namespace details

#define ARRAY_SIZE(var_x)                                  \
  details::extent<typename details::remove_reference< \
    decltype(var_x)>::type>::value
```

Andreas Fertig v1.0 C++: Be type-safe 31

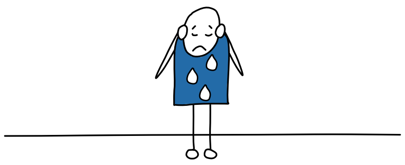


Andreas Fertig v1.0 C++: Be type-safe 32

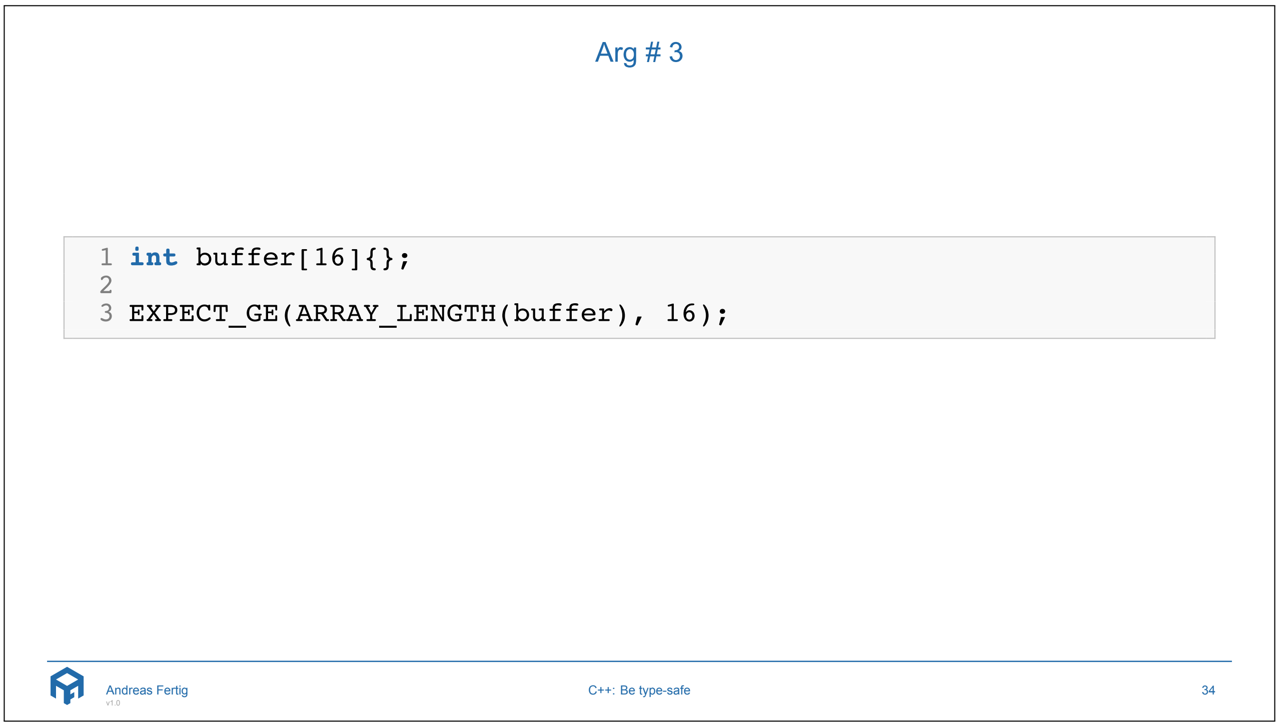


Arg # 3

```
Undefined symbols for architecture i386:
  "(anonymous namespace)::details::constant<unsigned long, 22ul>::value",
    referenced from:
      (anonymous namespace)::ArraySizeTest_BufferArray_Test::TestBody()
      in unittest.o
```



Andreas Fertig v1.0 C++: Be type-safe 33



Arg # 3

```cpp
int buffer[16]{};

EXPECT_GE(ARRAY_LENGTH(buffer), 16);
```

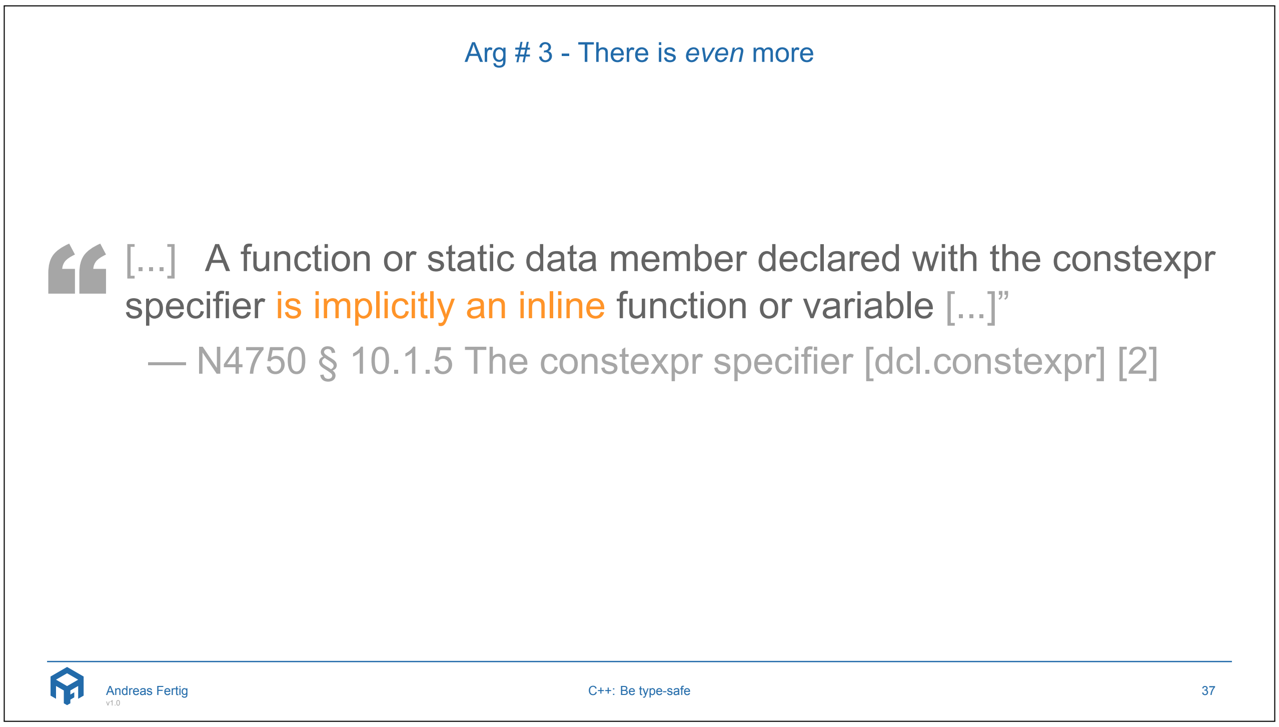
Andreas Fertig v1.0 C++: Be type-safe 34

Arg # 3

```
int buffer[16]{};

const auto& ref = ARRAY_LENGTH(buffer);
```

Arg # 3 - There is more

`-pedantic`

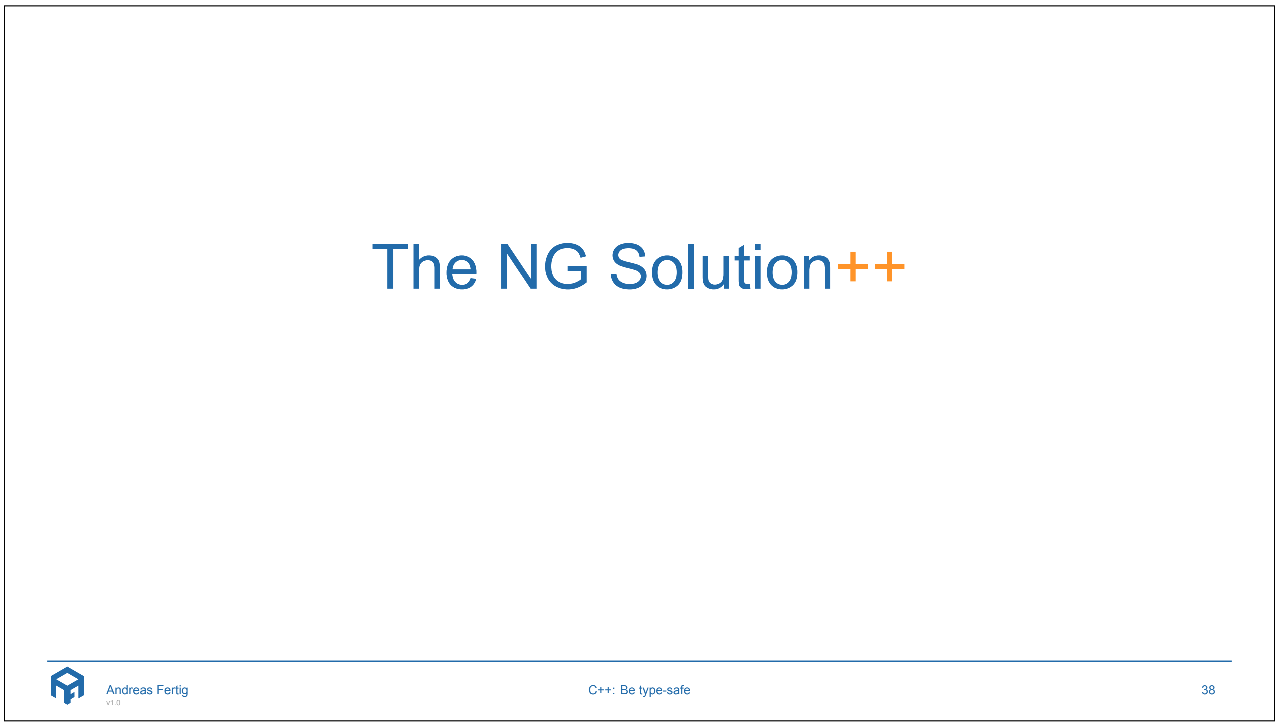


Arg # 3 - There is *even* more

“[...] A function or static data member declared with the constexpr specifier is implicitly an inline function or variable [...]”

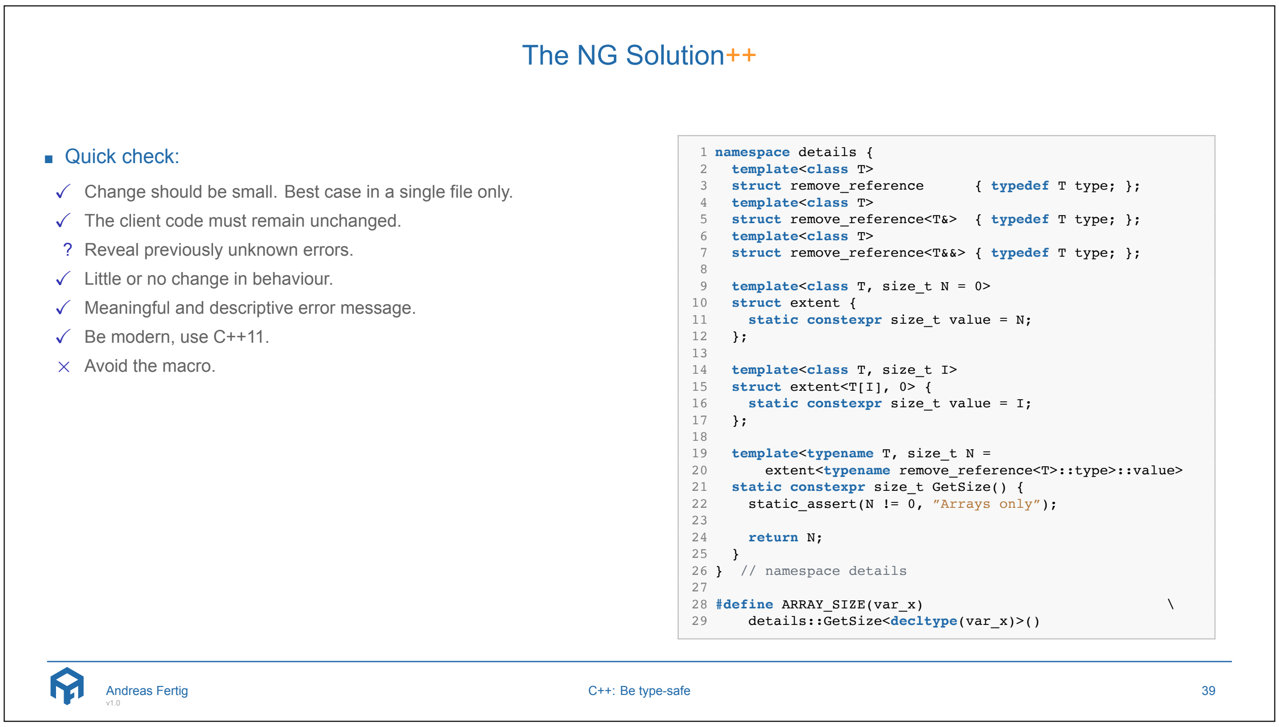
— N4750 § 10.1.5 The constexpr specifier [dcl.constexpr] [2]

Andreas Fertig v1.0 C++: Be type-safe 37



The NG Solution++

Andreas Fertig v1.0 C++: Be type-safe 38

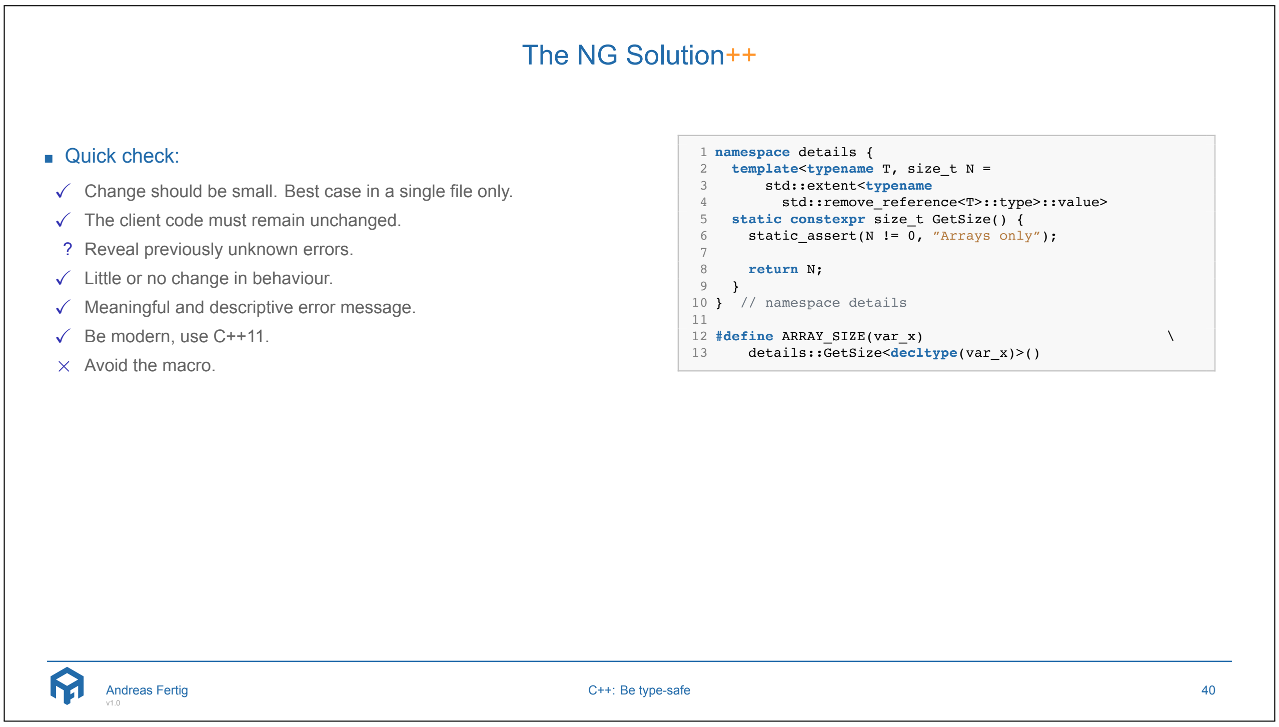


The NG Solution++

- Quick check:
  - ✓ Change should be small. Best case in a single file only.
  - ✓ The client code must remain unchanged.
  - ? Reveal previously unknown errors.
  - ✓ Little or no change in behaviour.
  - ✓ Meaningful and descriptive error message.
  - ✓ Be modern, use C++11.
  - × Avoid the macro.

```cpp
namespace details {
  template<class T>
  struct remove_reference      { typedef T type; };
  template<class T>
  struct remove_reference<T&>  { typedef T type; };
  template<class T>
  struct remove_reference<T&&> { typedef T type; };

  template<class T, size_t N = 0>
  struct extent {
    static constexpr size_t value = N;
  };

  template<class T, size_t I>
  struct extent<T[I], 0> {
    static constexpr size_t value = I;
  };

  template<typename T, size_t N =
      extent<typename remove_reference<T>::type>::value>
  static constexpr size_t GetSize() {
    static_assert(N != 0, "Arrays only");

    return N;
  }
}  // namespace details

#define ARRAY_SIZE(var_x)                                       \
    details::GetSize<decltype(var_x)>()
```

Andreas Fertig v1.0 C++: Be type-safe 39

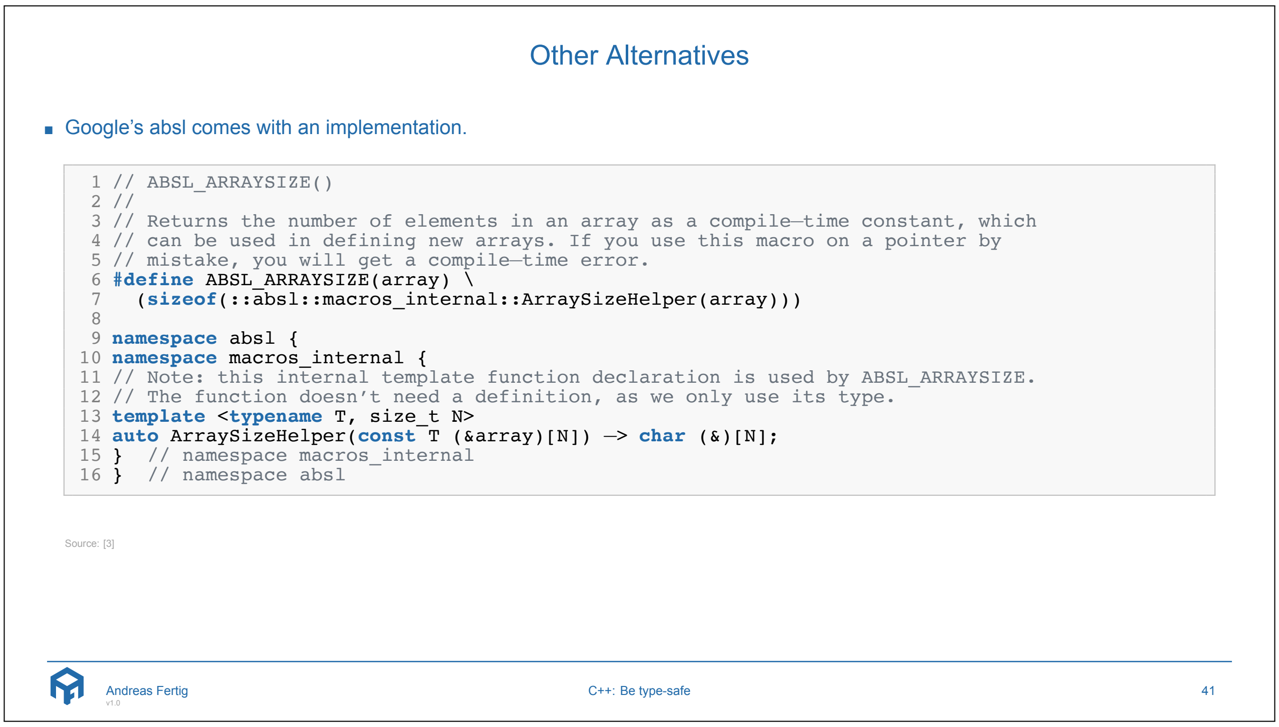


The NG Solution++

- Quick check:
  - ✓ Change should be small. Best case in a single file only.
  - ✓ The client code must remain unchanged.
  - ? Reveal previously unknown errors.
  - ✓ Little or no change in behaviour.
  - ✓ Meaningful and descriptive error message.
  - ✓ Be modern, use C++11.
  - × Avoid the macro.

```cpp
namespace details {
  template<typename T, size_t N =
      std::extent<typename
        std::remove_reference<T>::type>::value>
  static constexpr size_t GetSize() {
    static_assert(N != 0, "Arrays only");

    return N;
  }
}  // namespace details

#define ARRAY_SIZE(var_x)                                       \
    details::GetSize<decltype(var_x)>()
```

Andreas Fertig v1.0 C++: Be type-safe 40



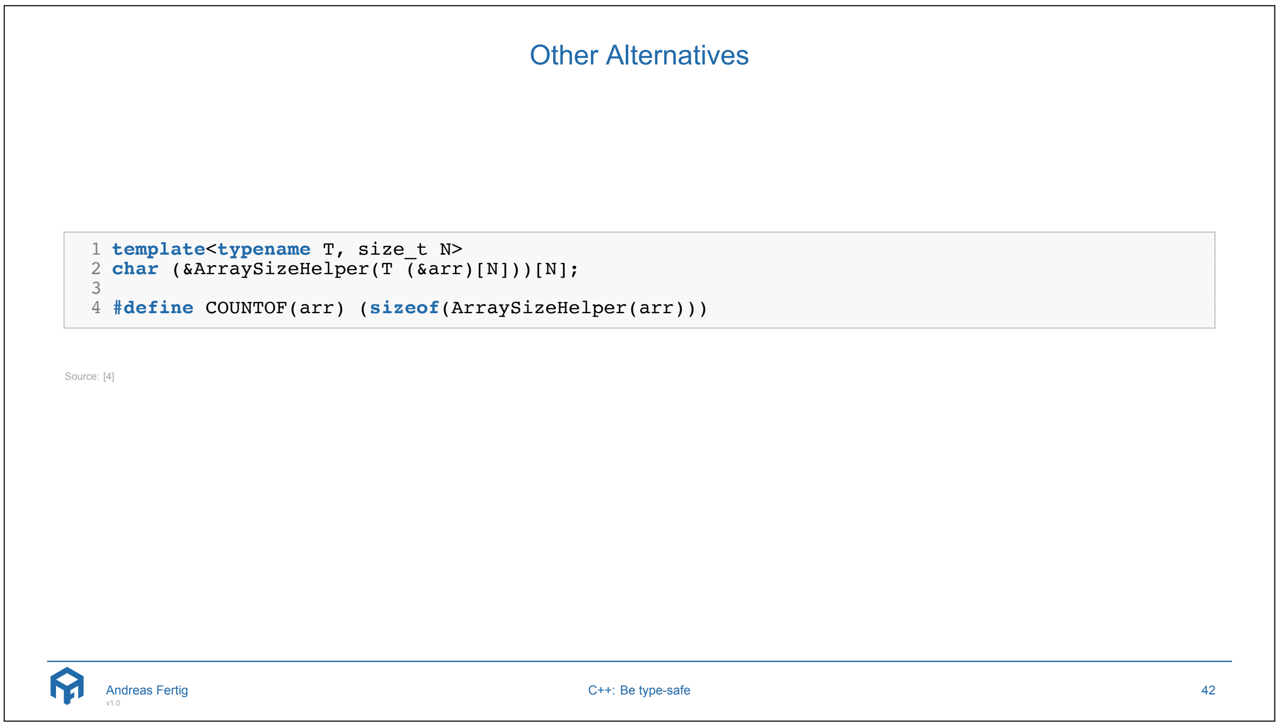
Other Alternatives

- Google's absl comes with an implementation.

```cpp
// ABSL_ARRAYSIZE()
//
// Returns the number of elements in an array as a compile–time constant, which
// can be used in defining new arrays. If you use this macro on a pointer by
// mistake, you will get a compile–time error.
#define ABSL_ARRAYSIZE(array) \
  (sizeof(::absl::macros_internal::ArraySizeHelper(array)))

namespace absl {
namespace macros_internal {
// Note: this internal template function declaration is used by ABSL_ARRAYSIZE.
// The function doesn't need a definition, as we only use its type.
template <typename T, size_t N>
auto ArraySizeHelper(const T (&array)[N]) –> char (&)[N];
}  // namespace macros_internal
}  // namespace absl
```

Source: [3]

Andreas Fertig
v1.0
C++: Be type-safe
41

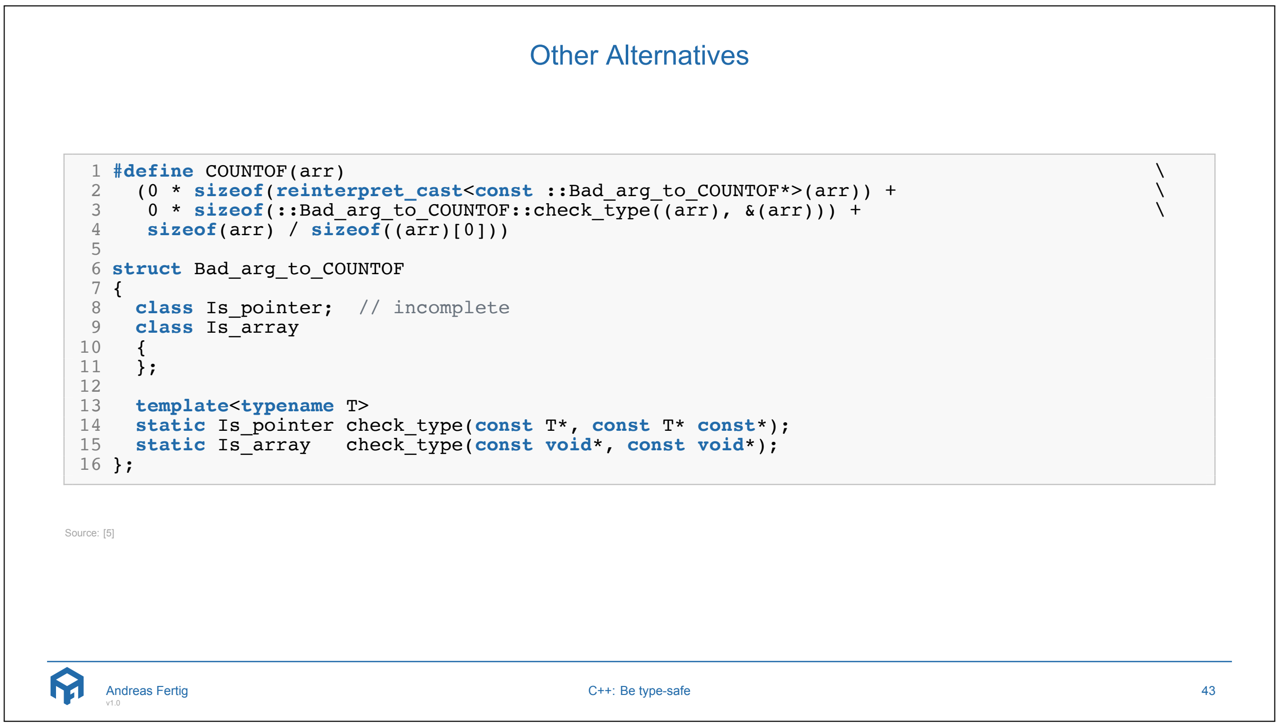


Other Alternatives

```cpp
template<typename T, size_t N>
char (&ArraySizeHelper(T (&arr)[N]))[N];

#define COUNTOF(arr) (sizeof(ArraySizeHelper(arr)))
```

Source: [4]

Andreas Fertig
v1.0
C++: Be type-safe
42



Other Alternatives

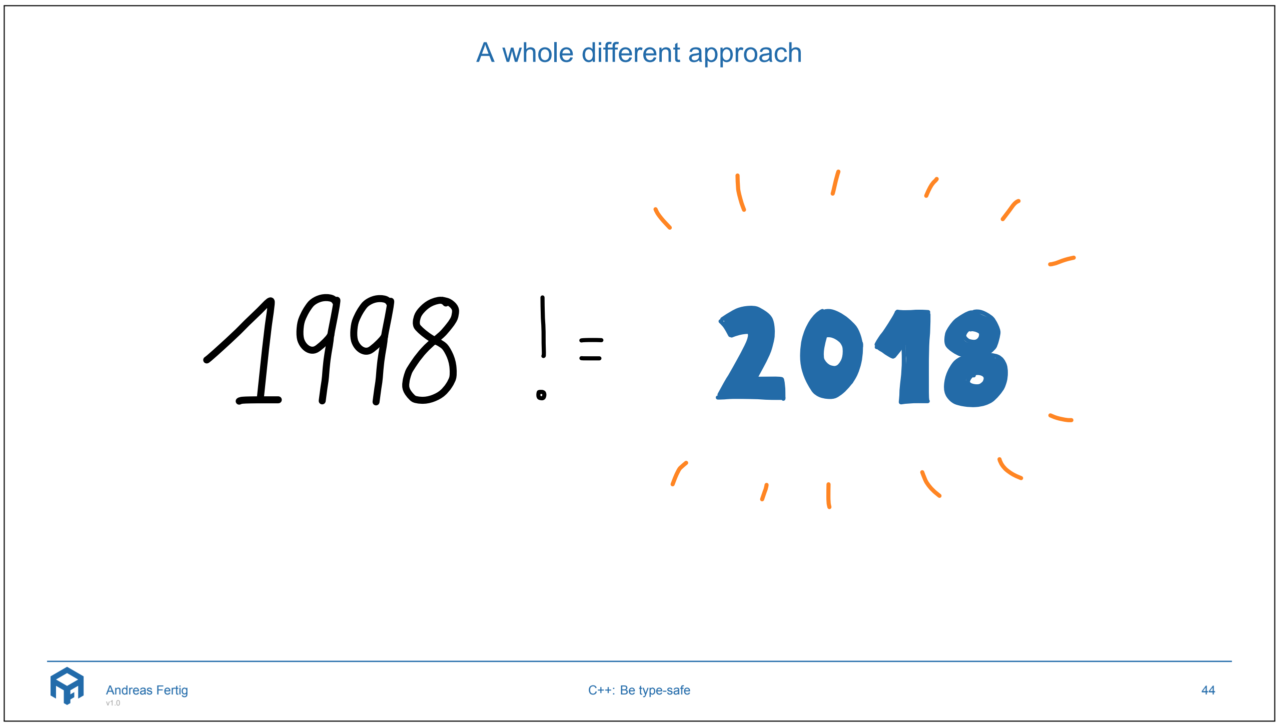
```cpp
#define COUNTOF(arr)                                                      \
  (0 * sizeof(reinterpret_cast<const ::Bad_arg_to_COUNTOF*>(arr)) +      \
   0 * sizeof(::Bad_arg_to_COUNTOF::check_type((arr), &(arr))) +         \
   sizeof(arr) / sizeof((arr)[0]))

struct Bad_arg_to_COUNTOF
{
  class Is_pointer;  // incomplete
  class Is_array
  {
  };

  template<typename T>
  static Is_pointer check_type(const T*, const T* const*);
  static Is_array   check_type(const void*, const void*);
};
```

Source: [5]

Andreas Fertig
v1.0

C++: Be type-safe

43



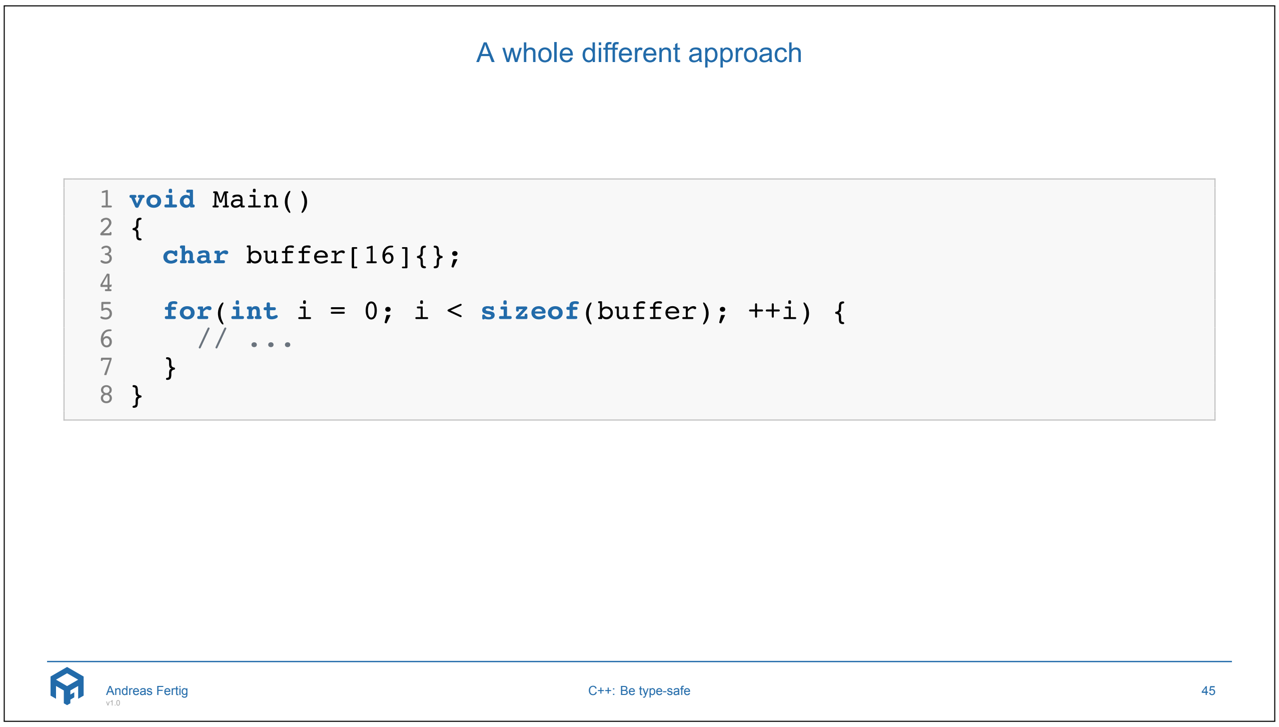
A whole different approach

1998 != 2018

Andreas Fertig
v1.0

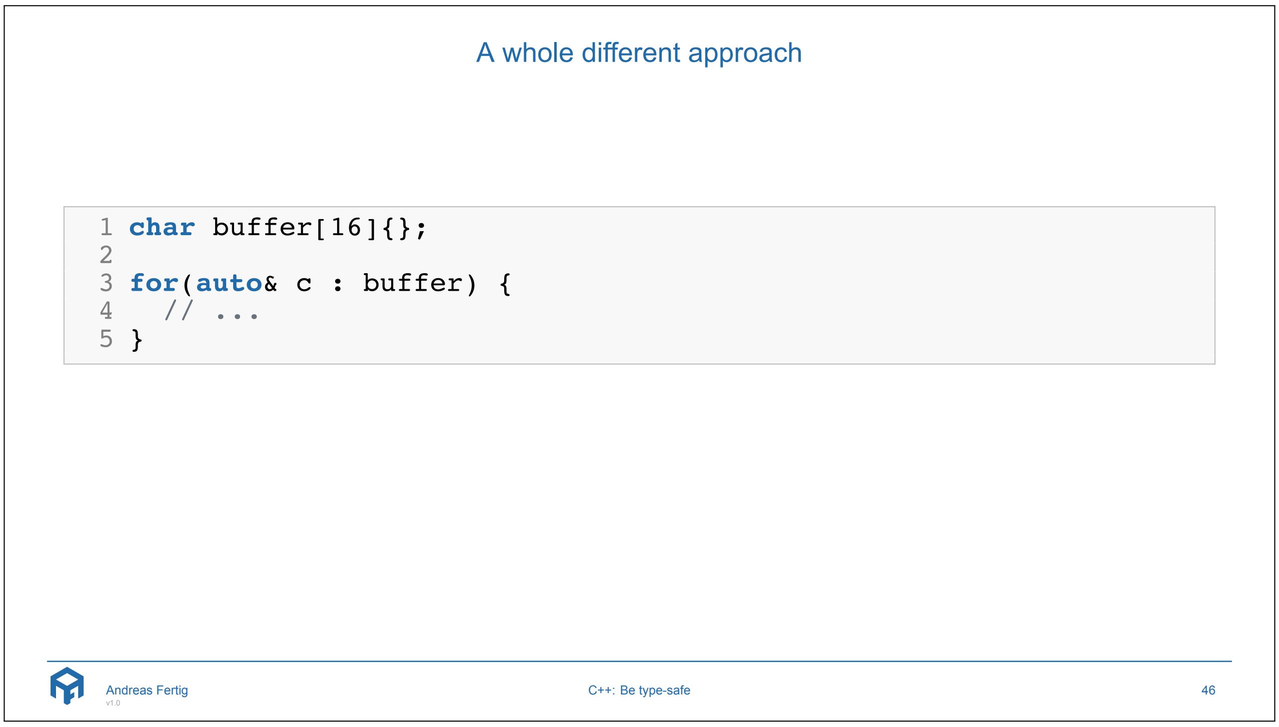
C++: Be type-safe

44



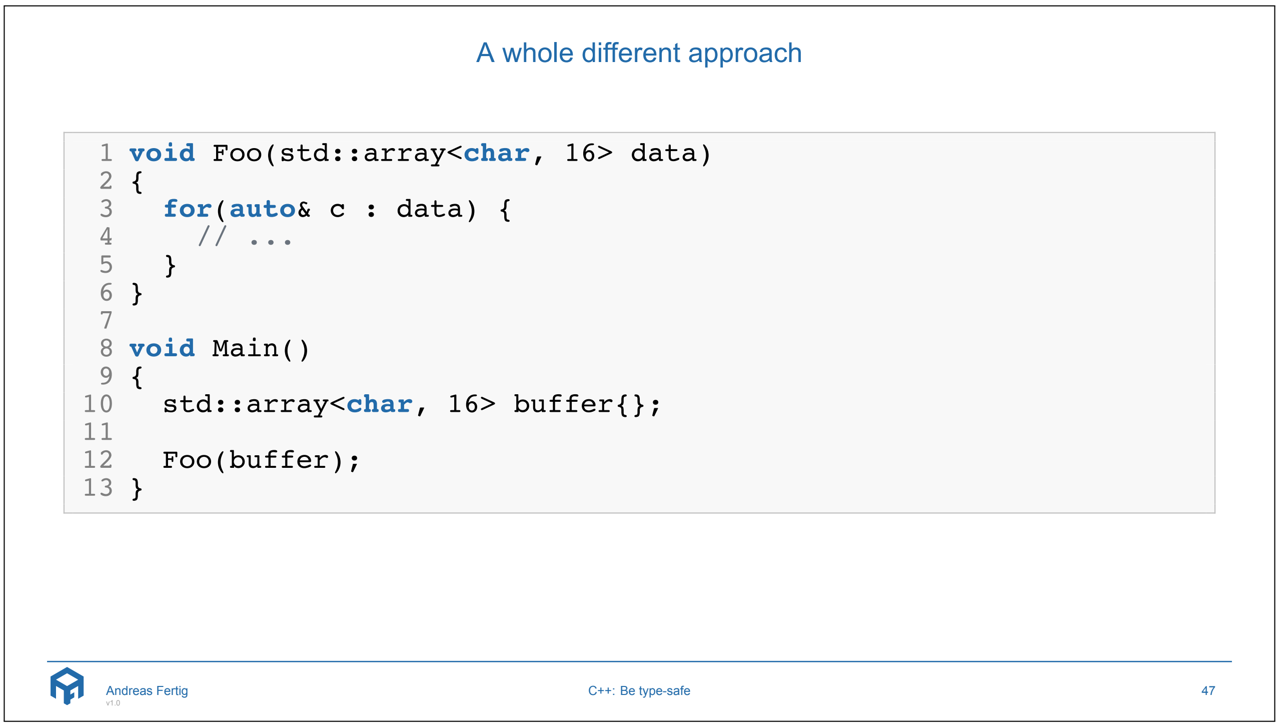
A whole different approach

```cpp
void Main()
{
  char buffer[16]{};

  for(int i = 0; i < sizeof(buffer); ++i) {
    // ...
  }
}
```



A whole different approach

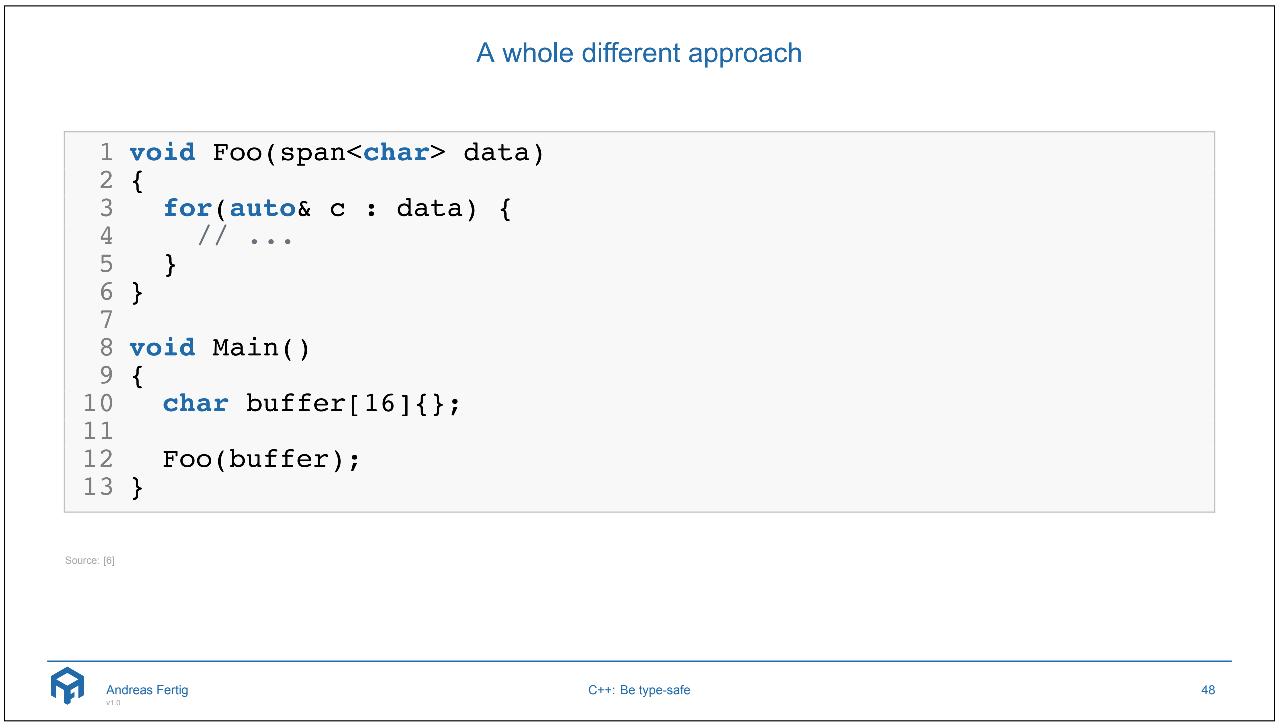
```cpp
char buffer[16]{};

for(auto& c : buffer) {
  // ...
}
```



A whole different approach

```cpp
void Foo(std::array<char, 16> data)
{
  for(auto& c : data) {
    // ...
  }
}

void Main()
{
  std::array<char, 16> buffer{};

  Foo(buffer);
}
```

Andreas Fertig v1.0 C++: Be type-safe 47



A whole different approach

```cpp
void Foo(span<char> data)
{
  for(auto& c : data) {
    // ...
  }
}

void Main()
{
  char buffer[16]{};

  Foo(buffer);
}
```

Source: [6]

Andreas Fertig v1.0 C++: Be type-safe 48

Be

MODERN!

}

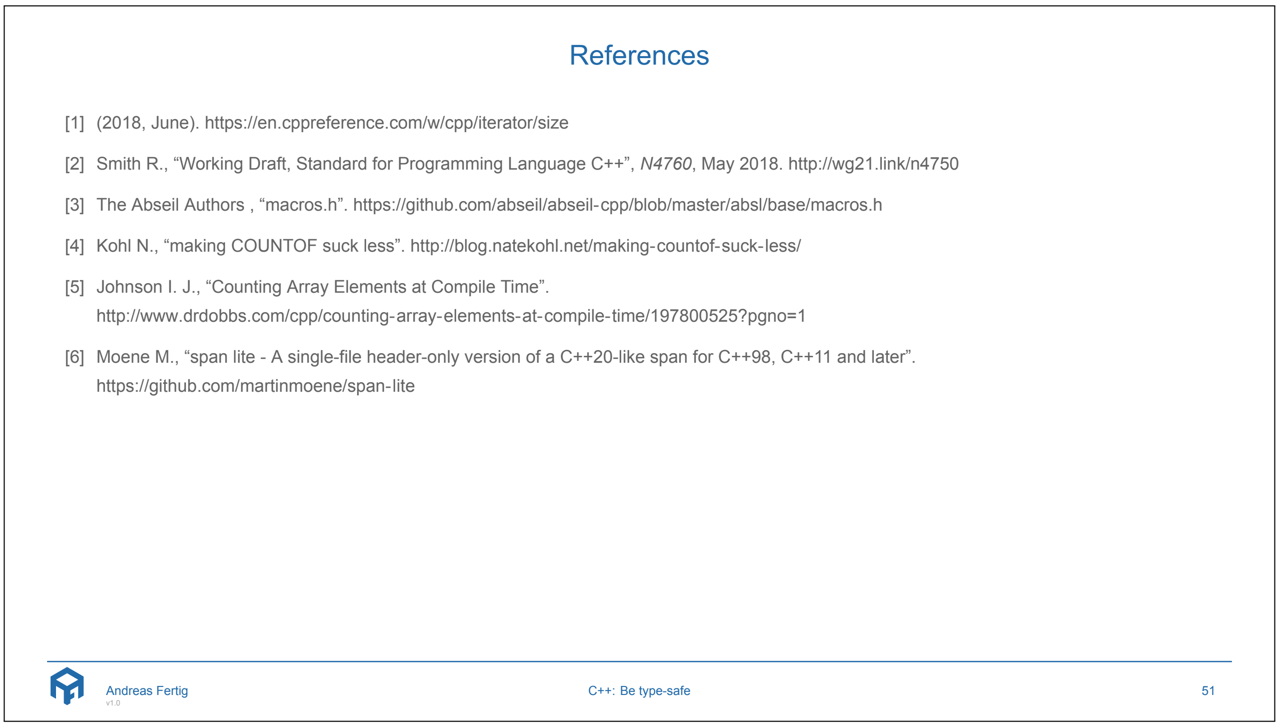
Ich bin Fertig.

Available online:

https://www.AndreasFertig.Info

Images by Franziska Panter:

https://panther-concepts.de



References

[1] (2018, June). https://en.cppreference.com/w/cpp/iterator/size

[2] Smith R., “Working Draft, Standard for Programming Language C++”, *N4760*, May 2018. http://wg21.link/n4750

[3] The Abseil Authors , “macros.h”. https://github.com/abseil/abseil-cpp/blob/master/absl/base/macros.h

[4] Kohl N., “making COUNTOF suck less”. http://blog.natekohl.net/making-countof-suck-less/

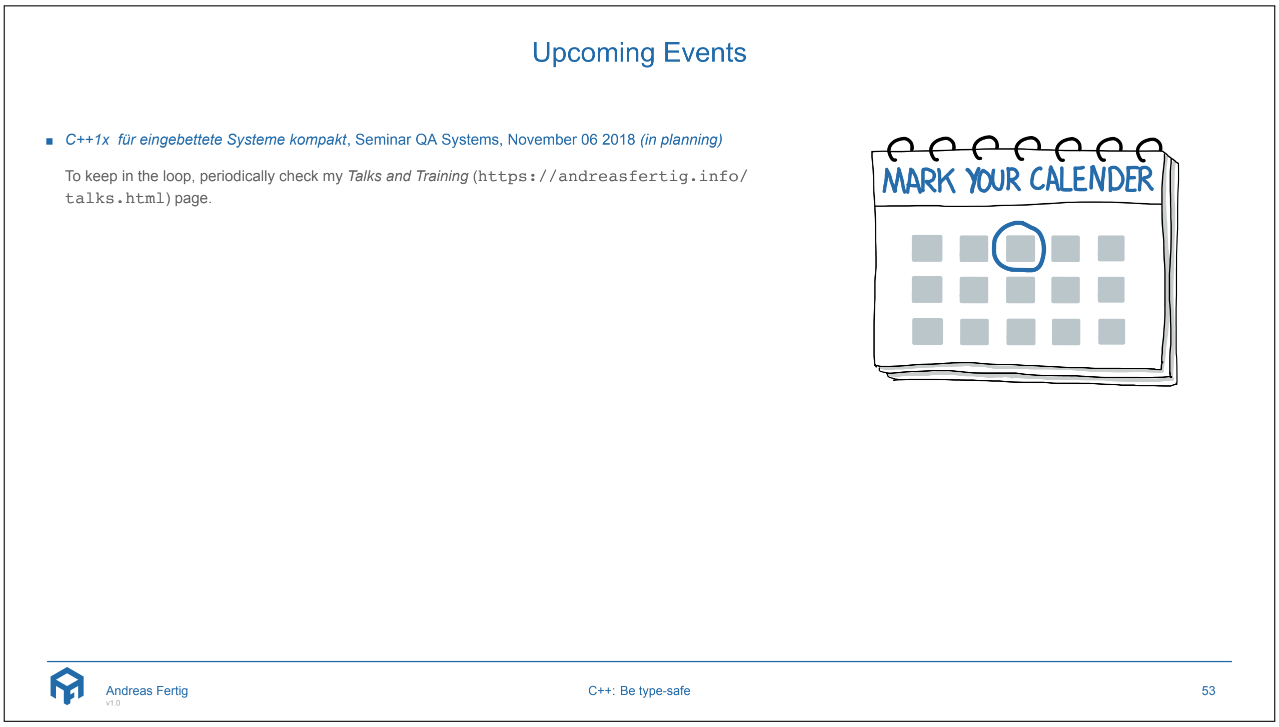
[5] Johnson I. J., “Counting Array Elements at Compile Time”. http://www.drdobbs.com/cpp/counting-array-elements-at-compile-time/197800525?pgno=1

[6] 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

References

Images:

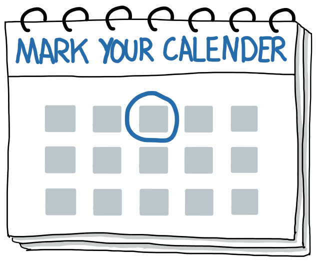
2: Franziska Panter
11: Franziska Panter
12: Franziska Panter
15: Franziska Panter
16: Franziska Panter
19: Franziska Panter
20: Franziska Panter
21: Franziska Panter
24: Franziska Panter
27: Franziska Panter
28: Franziska Panter
32: Franziska Panter
33: Franziska Panter
44: Franziska Panter
49: Franziska Panter
53: Franziska Panter

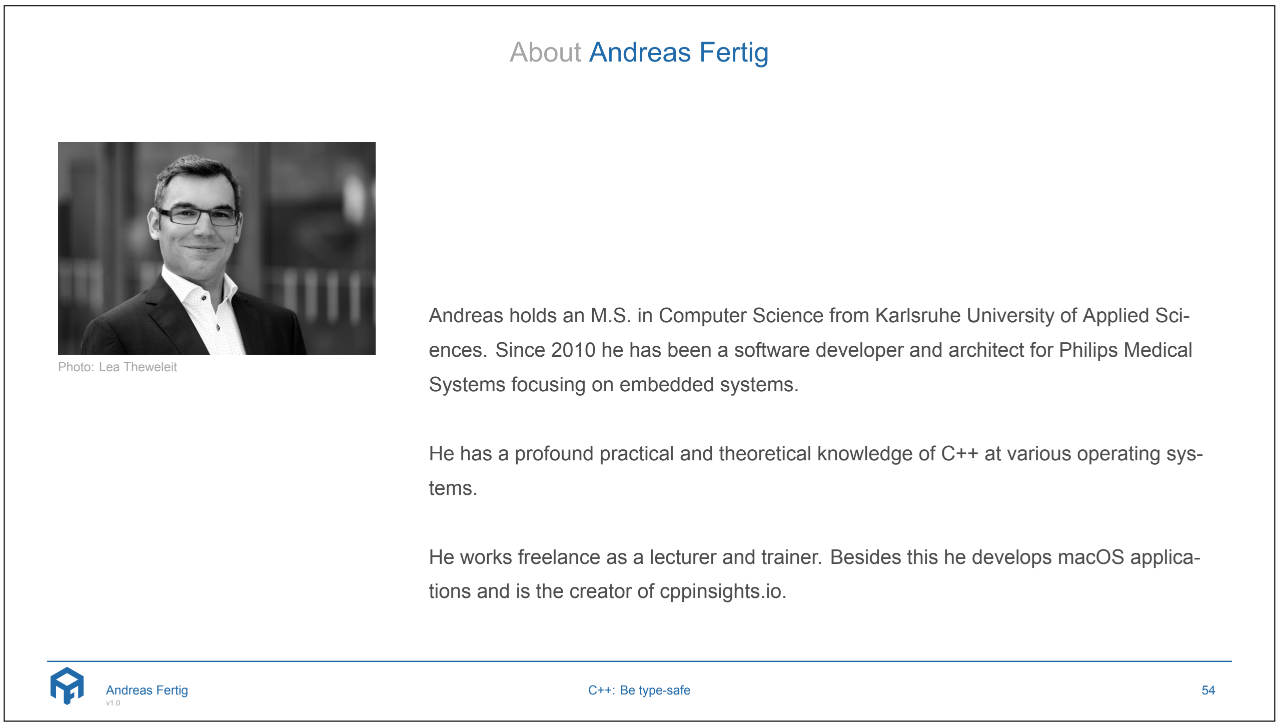


Upcoming Events

- *C++1x für eingebettete Systeme kompakt*, Seminar QA Systems, November 06 2018 *(in planning)*

To keep in the loop, periodically check my *Talks and Training* (`https://andreasfertig.info/talks.html`) page.





About Andreas Fertig

Photo: Lea Theweleit

Andreas holds an M.S. in Computer Science from Karlsruhe University of Applied Sciences. Since 2010 he has been a software developer and architect for Philips Medical Systems focusing on embedded systems.

He has a profound practical and theoretical knowledge of C++ at various operating systems.

He works freelance as a lecturer and trainer. Besides this he develops macOS applications and is the creator of cppinsights.io.