Towards a safer C++

Embo++
Bochum, Germany

J. Daniel Garcia

University Carlos III of Madrid
Spain

March 2023

Creative Commons License

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
  - University Carlos III of Madrid (since 2001).

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
  - University Carlos III of Madrid (since 2001).
- An ISO C++ language standards committee member.
  - UNE: Spanish Standards National Body.

Who am I?

- A C++ programmer.
  - Started writing C++ code in 1989.
- A university professor in Computer Architecture.
  - University Carlos III of Madrid (since 2001).
- An ISO C++ language standards committee member.
  - UNE: Spanish Standards National Body.
- **My goal**: Improve applications programming.
  - **Performance** → **faster** applications.
  - **Energy efficiency** → **better** performance per Watt.
  - **Maintainability** → **easier** to modify.
  - **Reliability** → **safer** components.

May 2022: OSS Security Mobilization Plan

Stream 4: Eliminate root causes of many vulnerabilities through replacement of non-memory-safe languages.

Some programming languages, like C and C++, make memory safety challenging and can lead to difficult to detect and eliminate software defects. In contrast, most programming languages, like Go and Rust, handle memory management and other kinds of security-sensitive tasks safely by default, making it easier for developers to avoid entire categories of vulnerabilities. Much of the modern Internet software infrastructure is built on software written in C, and that leads to a large number of vulnerabilities each year. About 70% of Microsoft's vulnerabilities in 2006-2018 were due to memory safety issues[5], and in 2020 Google

November 2022: NSA

November 2022: NSA

National Security Agency | Cybersecurity Information Sheet

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory

The path forward

Memory issues in software comprise a large portion of the exploitable vulnerabilities in existence. NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®. Memory safe languages provide

January 2023: Consumer Reports

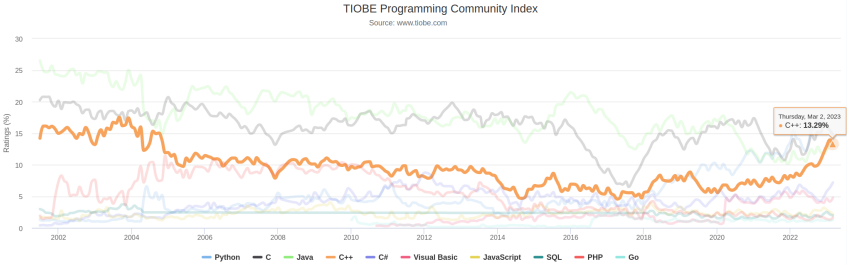
Why Memory Safety

Roughly 60 to 70 percent of browser and kernel vulnerabilities—and security bugs found in C/C++ code bases—are due to memory unsafety, many of which can be solved by using

January 2023: Consumer Reports

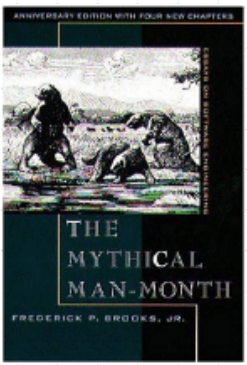
As much as possible, companies, government organizations, and other entities should commit to using memory-safe languages for new products and tools and newly developed custom components. They should also support the development of open source memory-safe code.

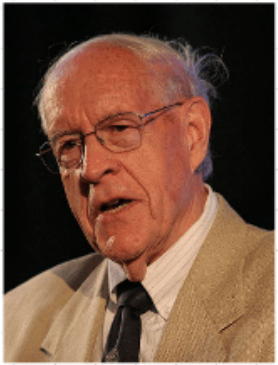
Tiobe Index

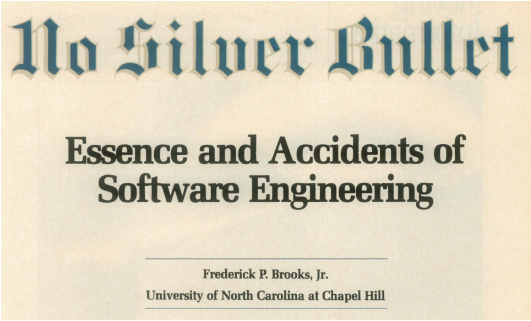


Year	Winner
2022	C++
2021	Python
2020	Python
2019	C
2018	Python
2017	C

Is there a silver bullet?







Why Brooks and Silver Bullets?

- Two kinds of complexity:
  - **Essential complexity**: Inherent to the problem to be solved.
    - Nothing can remove it.
  - **Accidental complexity**: Problems that can be fixed.
    - It can be removed over time.

Why Brooks and Silver Bullets?

- Two kinds of complexity:
  - **Essential complexity**: Inherent to the problem to be solved.
    - Nothing can remove it.
  - **Accidental complexity**: Problems that can be fixed.
    - It can be removed over time.
- How do we remove **accidental complexity**?
  - Growing software organically through **incremental development**.
  - We tried multiple times starting over and it did not work.

Why Brooks and Silver Bullets?

- Two kinds of complexity:
  - **Essential complexity**: Inherent to the problem to be solved.
    - Nothing can remove it.
  - **Accidental complexity**: Problems that can be fixed.
    - It can be removed over time.
- How do we remove **accidental complexity**?
  - Growing software organically through **incremental development**.
  - We tried multiple times starting over and it did not work.
- **Questions?**
  - Is memory safety **accidental** or **essential**?
  - Are programming languages **different** from *other* software?

The many dimensions of safety

- What do we mean by safety?
  - **Logic safety**
  - **Resource safety**
  - **Concurrency safety**
  - **Memory safety**
  - **Type safety**
  - **Conversion safety**
  - **Timing safety**
  - **Termination safety**

Example: Memory safety

- Achieving **memory safety**:
  - A pointer points either to **nullptr** or to a **valid object**.
  - An access through a pointer never refers to **nullptr**.
  - An access through a subscript is always within bounds.

Example: Memory safety

- Achieving **memory safety**:
  - A pointer points either to **nullptr** or to a **valid object**.
  - An access through a pointer never refers to **nullptr**.
  - An access through a subscript is always within bounds.

- Safety dimensions are not independent:
  - An object is accessed with the type it was defined
    - **Type safety**.
  - An object is properly constructed and destroyed.
    - **Resource safety**.

Solving the safety problem

- **Constraints**:

Solving the safety problem

- **Constraints**:
  - C++ is used in many different domains.
    - Different sets of priorities (e.g. performace vs safety).

Solving the safety problem

- **Constraints**:
  - C++ is used in many different domains.
    - Different sets of priorities (e.g. performace vs safety).
  - Radical changes are not acceptable.
    - Don't break my code.

Solving the safety problem

- **Constraints**:
  - C++ is used in many different domains.
    - Different sets of priorities (e.g. performace vs safety).
  - Radical changes are not acceptable.
    - Don't break my code.
  - New languages are tried out constantly.
    - And they usually fail.

Solving the safety problem

- **Constraints**:
  - C++ is used in many different domains.
    - Different sets of priorities (e.g. performace vs safety).
  - Radical changes are not acceptable.
    - Don't break my code.
  - New languages are tried out constantly.
    - And they usually fail.

- **What can we do then?**
  - Take an evolutionary approach.

Safety profiles

- A **profile** combines multiple approaches:

Safety profiles

- A **profile** combines multiple approaches:
  - A set of **coding rules**.
    - Isolated rules do not solve the problem.
    - Combination of rules make real impact.
    - **Goal**: Simplify code.

Safety profiles

- A **profile** combines multiple approaches:
  - A set of **coding rules**.
    - Isolated rules do not solve the problem.
    - Combination of rules make real impact.
    - **Goal**: Simplify code.
  - **Support libaries**:
    - Express safe code.
    - Avoid need for unsafe construct in general code.
    - Include run-time checks where needed.

Safety profiles

- A **profile** combines multiple approaches:
  - A set of **coding rules**.
    - Isolated rules do not solve the problem.
    - Combination of rules make real impact.
    - **Goal**: Simplify code.
  - **Support libaries**:
    - Express safe code.
    - Avoid need for unsafe construct in general code.
    - Include run-time checks where needed.
  - **Static analysis**:
    - Make sure that no unsafe code is executed.

Object access

- **Guiding principle**: Every object is accessed with the type it was originally defined.

Object access

- **Guiding principle**: Every object is accessed with the type it was originally defined.

- **Consequences**:
  - Allways intialize an object (CG ES.20).
  - **Avoid unsafe casts** (CG ES.48).
  - **Avoid unsafe unions** (CG C.181).
  - **Avoid unsafe use of pointers**.

Object initialization

Bad

```
int f(int x) {
  int z;        // z not init
  if (x>0) {
    z = x;
  }
  return z;
}
```

Object initialization

Bad

```
int f(int x) {
  int z;        // z not init
  if (x>0) {
    z = x;
  }
  return z;
}
```

Good

```
int f(int x) {
  int z = -1;       // z init
  if (x>0) {
    z = x;
  }
  return z;
}
```

Object initialization

Bad

```
int f(int x) {
  int z;        // z not init
  if (x>0) {
    z = x;
  }
  return z;
}
```

Good

```
int f(int x) {
  int z = -1;       // z init
  if (x>0) {
    z = x;
  }
  return z;
}
```

Better

```
int f(int x) {
  int z = [] {
    if (x>0) return x;
    return -1;
  }();

  return z;
}
```

Explictly uninitialized objects

- It rarely makes sense not to initialize an object.

Avoid useless initialization

```
message read_message() {
  [[ uninitialized ]] std::byte buffer[buf_size];
  int num_byes = read_buffer(buffer, buf_size);
  message m{buffer, num_bytes};
  return m;
}
```

- **[[uninitialized]]** is an hypothetical new attribute.
- Alternatively, locally ignore the initialization rule.

Avoid dangling pointers

- **Guiding principle**: Every **pointer** points either to an **object** or to the **null pointer**.

Avoid dangling pointers

- **Guiding principle**: Every **pointer** points either to an **object** or to the **null pointer**.
- What is a pointer?

Avoid dangling pointers

- **Guiding principle**: Every **pointer** points either to an **object** or to the **null pointer**.

- What is a pointer?
  - A raw pointer.
  - A reference.
  - A container of pointers.
  - A smart pointer.
  - A lambda capture of a pointer.
  - ...

The problem with dangling pointers

- Dangling pointers are a huge problem.
  - **Hard** to detect locally.

The problem with dangling pointers

- Dangling pointers are a huge problem.
  - **Hard** to detect locally.

An good-looking function?

```
void f(element * p) {
  do_something(p);
  delete p;
}
```

The problem with dangling pointers

- Dangling pointers are a huge problem.
  - **Hard** to detect locally.

An good-looking function?

```
void f(element * p) {
  do_something(p);
  delete p;
}
```

Another good-looking function?

```
void g() {
  element * pe = new element;
  f(pe);
  // More things
  pe-> process();
  delete pe;
}
```

Why should we eliminate dangling pointers?

- What if we allow dangling pointers?
  - No type safety.
  - No memory safety.
  - No resource safety.

Why should we eliminate dangling pointers?

- What if we allow dangling pointers?
  - No type safety.
  - No memory safety.
  - No resource safety.
- **Rules**:
  - Differentiate owners from not-owners.
    - Smart pointers are owners.
    - **gsl::owner<T*>** is an owner.
    - Raw pointers are not-owners.
    - Might consider a **not_owner<T*>**?
  - Identify pointers escaping outside their owner's scope.
    - Returning, throwing, out parameters, captures, ...

Rules for escaping pointers

- A pointer can be returned if:
  - It was passed to the scope.
    - As an argument.
    - Retrieved from from an external object.
  - It points to an object external to the scope.
    - Obtained by **new**.
    - Outlives the scope.

Rules for escaping pointers

- A pointer can be returned if:
  - It was passed to the scope.
    - As an argument.
    - Retrieved from from an external object.
  - It points to an object external to the scope.
    - Obtained by **new**.
    - Outlives the scope.
- Otherwise:
  - The pointer cannot be returned.
  - Imposes limitations on complexity.

3 Rules

- Initialization
  - Object initialization
  - Unsafe pointers
- Resource safety
- Run-time checks
  - Null pointer errors
  - Bounds checking
- Low level code

Ownership abstractions

- Use ownership abstractions for managing resources.
  - RAII pattern.
  - Acquire the resource in constructor or later.
  - Release the resource in the destructor or before.
  - Implement move semantics.

Ownership abstractions

- Use ownership abstractions for managing resources.
  - RAII pattern.
  - Acquire the resource in constructor or later.
  - Release the resource in the destructor or before.
  - Implement move semantics.
- Existing ownership abstractions:
  - **vector**, **map**, **unique_ptr**, **fstream**, **jthread**, ...

Ownership abstractions

- Use ownership abstractions for managing resources.
  - RAII pattern.
  - Acquire the resource in constructor or later.
  - Release the resource in the destructor or before.
  - Implement move semantics.
- Existing ownership abstractions:
  - **vector**, **map**, **unique_ptr**, **fstream**, **jthread**, ...
- For low level owners use **gsl::owner** or define yours,

```
template <typename T>
using owner = T;
```

Low-level ownership

- **gsl::owner<T*>** is equivalent to **T***.
  - Simplifies code analysis.
  - Makes easier code reviews.
  - Has no impact on ABIs.

Low-level ownership

- **gsl::owner<T*>** is equivalent to **T***.
  - Simplifies code analysis.
  - Makes easier code reviews.
  - Has no impact on ABIs.
- **owner** is **good** for:
  - Implementation of ownership abstractions
  - Avoiding ABI breaks.

Low-level ownership

- **gsl::owner<T*>** is equivalent to **T***.
  - Simplifies code analysis.
  - Makes easier code reviews.
  - Has no impact on ABIs.
- **owner** is **good** for:
  - Implementation of ownership abstractions
  - Avoiding ABI breaks.
- **owner** is **bad** for:
  - Wide use in application code.
  - Implies C-like interfaces.

Ownership rules

- A pointer returned by **new** is an owner.
  - It must be returned or deleted.
  - Unless stored in static storage.

Ownership rules

- A pointer returned by **new** is an owner.
  - It must be returned or deleted.
  - Unless stored in static storage.
- Only **owner** pointers can be deleted.

Ownership rules

- A pointer returned by **new** is an owner.
  - It must be returned or deleted.
  - Unless stored in static storage.
- Only **owner** pointers can be deleted.
- An owner passed to a scope must be deleted or passed to another scope as an owner.

Ownership rules

- A pointer returned by **new** is an owner.
  - It must be returned or deleted.
  - Unless stored in static storage.
- Only **owner** pointers can be deleted.
- An owner passed to a scope must be deleted or passed to another scope as an owner.
- A pointer passed to a scope as an owner is invalidated.
  - Invalidated pointers cannot be used any more.

Owners and dangling pointers

- Dangling pointers are a huge problem.
  - Difficult to detect locally.

An good-looking function?

```
void f(element * p) {
  do_something(p);
  delete p;
}
```

Another good-looking function?

```
void g() {
  element * pe = new element;
  f(pe);
  // More things
  pe-> process();
  delete pe;
}
```

Owners and dangling pointers

- Owners simplify finding dangling pointer bugs.
  - Easier to detect locally.

An good-looking function?

```
void f(gsl::owner<element*> p) {
  do_something(p);
  delete p;  // Must delete an owner
}
```

Another good-looking function?

```
void g() {
  gsl::owner<element*> pe = new element; // OK
        owners init from new
  f(pe);  // OK pass an owner to function takin an
        owner
  // More things
  pe->process(); // Bad: Invalidated owner
  delete pe;       // Bad: Deleting invalidated owner
}
```

Using owners to implement abstractions

Implementing vector

```
template <semiregular T>
class vector {
public:
  // ...
private:
  gsl::owner<T*> buffer;
  T * end;
  T * reserve_end;
};
```

- Use **gsl::owner** for memory that needs to be released.

3 Rules

- Initialization
  - Object initialization
  - Unsafe pointers
- Resource safety
- Run-time checks
  - Null pointer errors
  - Bounds checking
- Low level code

Null-pointer problems

- A pointer may point to a valid object or to **nullptr**.

Null-pointer problems

- A pointer may point to a valid object or to **nullptr**.

Implementation side

```
void f(element * p) {
  if (p != nullptr) {
    p->do_something();
  }
  // ...
}

void g(element * p) {
  p->do_other_thing();
}
```

Null-pointer problems

- A pointer may point to a valid object or to **nullptr**.

Implementation side

```
void f(element * p) {
  if (p != nullptr) {
    p->do_something();
  }
  // ...
}

void g(element * p) {
  p->do_other_thing();
}
```

Call side

```
element * q = get_element();

f(q); // Does f() accept nullptr?
if (q != nullptr) f(q); // Needed check?

g(q); // Does f() accept nullptr?
if (q != nullptr) g(q); // Needed check?
```

- Compilers do not read documentation.

Non-null interfaces

- Constructing a **gsl::not_null<T*>** checks that a pointer is not-null.
  - The pointer can be used safely.

Implementation side

```
void f(gsl::not_null<element*> p) {
  if (p != nullptr) { // OK, but redundant
    p->do_something();
  }
  // ...
}

void g(gsl::not_null<element*> p) {
  p->do_other_thing(); // OK
}
```

Call side

```
element * q = get_element();

f(q); // OK
if (q != nullptr) f(q); // Redundant

g(q); // OK
if (q != nullptr) g(q); // Redundant
```

- Redundant checks can be warned

Accessing a range

- Use pointers only for single objects.

Bad

```
void f(element * v) {
  v[5] = create(); // Range?
  // ...
}
```

Better

```
void f(gsl::span<element> v) {
  v[5] = create(); // Range checked
}
```

Accessing a range

- Use pointers only for single objects.

Bad

```
void f(element * v) {
  v[5] = create(); // Range?
  // ...
}
```

Better

```
void f(gsl::span<element> v) {
  v[5] = create(); // Range checked
}
```

- **IMPORTANT**: There is a difference between **gsl::span** and **std::span**.

Traversal and bounds checking

- Traversing a primitive array is fast but unsafe.

Bad

```
double sum(double v[], int size) {
  double r = 0.0;
  for (int i=0; i<size; ++i) {
    r += v[i];
  }
  return r;
}
```

Traversal and bounds checking

- Traversing a primitive array is fast but unsafe.

Bad

```
double sum(double v[], int size) {
  double r = 0.0;
  for (int i=0; i<size; ++i) {
    r += v[i];
  }
  return r;
}
```

Better

```
double sum(gsl::span<double> v) {
  double r = 0.0;
  for (std::size_t i=0; i<v.size(); ++i) {
    r += v[i]; // Range checking
  }
  return r;
}
```

Traversal and bounds checking

- Traversing a primitive array is fast but unsafe.

Bad

```
double sum(double v[], int size) {
  double r = 0.0;
  for (int i=0; i<size; ++i) {
    r += v[i];
  }
  return r;
}
```

Better

```
double sum(gsl::span<double> v) {
  double r = 0.0;
  for (std::size_t i=0; i<v.size(); ++i) {
    r += v[i]; // Range checking
  }
  return r;
}
```

Best

```
double sum(gsl::span<double> v) {
  double r = 0.0;
  for (auto x : v) {
    r += x; // No checking
  }
  return r;
}
```

- Avoid checking with range-for loops.

3 Rules

- Initialization
  - Object initialization
  - Unsafe pointers
- Resource safety
- Run-time checks
  - Null pointer errors
  - Bounds checking
- Low level code

Can we bypass rules?

- C++ is used for **low-level use of resources** (including memory).
  - We **cannot forbid** all access to raw memory.
  - Other languages use C++ for those tasks, but we cannot.

Can we bypass rules?

- C++ is used for **low-level use of resources** (including memory).
  - We **cannot forbid** all access to raw memory.
  - Other languages use C++ for those tasks, but we cannot.

- **Alternatives**:
  - Enable selectively profiles.
  - A future annotation **[[unverified]]**.
    - For use in fundamental primitives.
  - Should allow to specify profile dimensions.
    - **[[unverified lifetime]]**
    - **[[unverified bounds]]**
    - ...

The sad history of contracts

- 2014: Multiple proposals on contracts programming.
- 2016: Joint proposal trying to consider trade-offs.
  - Gabriel Dos Reis, J. Daniel Garcia, John Lakos Alisdair Meredith, Nathan Myers, Bjarne Stroustrup
  - Targeting C++20.
- 2018: Added to draft ISO/IEC 14882:2020.
- 2019: Removed from the draft.
- Today: Still working on it.

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.
  - **Precondition**: What are the *expectations* of the function?

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.
  - **Precondition**: What are the *expectations* of the function?
  - **Postconditions**: What must the function *ensure* upon termination?

What is a contract?

- A contract is the set of **preconditions**, **postconditions** and **assertions** associated to a function.
  - **Precondition**: What are the *expectations* of the function?
  - **Postconditions**: What must the function *ensure* upon termination?
  - **Assertions**: What predicates must be satisfied in specific locations of a function body?

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by **pre**.

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by **pre**.

```
double sqrt(double x) [[pre: x>0]];
```

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by **pre**.

```
double sqrt(double x) [[pre: x>0]];

class queue {
  // ...
  void push(const T & x) [[pre: ! full () ]];
  // ...
};
```

Expectations

- **Precondition**
  - A predicate that should hold upon entry into a function.
  - It expresses a function's expectation on its arguments and/or the state of objects that may be used by the function.
  - Expressed by **pre**.

```
double sqrt(double x) [[pre: x>0]];

class queue {
  // ...
  void push(const T & x) [[pre: ! full () ]];
  // ...
};
```

- Preconditions use a modified attribute syntax.
- The expectation is part of the function declaration.

Assurances

- **Postcondition**
  - A predicate that should hold upon exit from a function.
  - It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function.
  - Postconditions are expressed by **post**.

Assurances

- **Postcondition**
  - A predicate that should hold upon exit from a function.
  - It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function.
  - Postconditions are expressed by **post**.

```
double sqrt(double x)
  [[ pre: x>=0]]
  [[ post result: result >=0]];
```

Assurances

- **Postcondition**
  - A predicate that should hold upon exit from a function.
  - It expresses the conditions that a function should ensure for the return value and/or the state of objects that may be used by the function.
  - Postconditions are expressed by **post**.

```
double sqrt(double x)
  [[ pre: x>=0]]
  [[ post result: result >=0]];
```

- Postconditions may introduce a name for the result of the function.

Assertions

- **Assertions**
  - A predicate that should hold at its point in a function body.
  - It expresses the conditions that must be satisfied, on objects that are accessible at its point in a body.
  - Assertions are expressed by **assert**.

Assertions

- **Assertions**
  - A predicate that should hold at its point in a function body.
  - It expresses the conditions that must be satisfied, on objects that are accessible at its point in a body.
  - Assertions are expressed by **assert**.

```
double add_distances(const std::vector<double> & v)
   [[ post r:  r>=0.0]]
{
   double r = 0.0;
   for  (auto x :  v)  {
      [[ assert: x >= 0.0]];
      r  += x;
   }
   return r;
}
```

Effect of contracts

- A contract has no observable effect on a correct program (except performance).
  - The only semantic effect of a contract happens if it is violated.
  - A contract should not have side-effects.

Effect of contracts

- A contract has no observable effect on a correct program (except performance).
  - The only semantic effect of a contract happens if it is violated.
  - A contract should not have side-effects.

- Multiple semantics:
  - No evaluation.
  - Terminate on violation.
  - Invoke a handler?
  - Throw an exception?

What can we do today?

- Start thinking in terms of contracts.
  - OK. You probably do that already.
- Use some approximation of contracts.
  - GSL supports **Expects** and **Ensures** as macros.
    - **Expects** ⇒ **pre**.
    - **Ensures** ⇒ **post**.
  - They need to be in the body.

What can we do today?

- Start thinking in terms of contracts.
  - OK. You probably do that already.
- Use some approximation of contracts.
  - GSL supports **Expects** and **Ensures** as macros.
    - **Expects** ⇒ **pre**.
    - **Ensures** ⇒ **post**.
  - They need to be in the body.

```
double sqrt(double x) {
  Expects(x>=0);
  // Compute result
  Ensures(result>=0);
  return result;
}
```

Some thoughts

- **Memory safety** is a **real concern**.
  - **Reminder**: There is no silver bullet.

Some thoughts

- **Memory safety** is a **real concern**.
  - **Reminder**: There is no silver bullet.
- There **many dimensions** of **safety**.
  - Memory is just one of them.
  - Safety dimensions are not orthogonal.

Some thoughts

- **Memory safety** is a **real concern**.
  - **Reminder**: There is no silver bullet.
- There **many dimensions** of **safety**.
  - Memory is just one of them.
  - Safety dimensions are not orthogonal.
- Towards **profiles** combining **multiple approaches**:
  - Coding rules.
  - Support libraries.
  - Static analysis.

Some thoughts

- **Memory safety** is a **real concern**.
  - **Reminder**: There is no silver bullet.
- There **many dimensions** of **safety**.
  - Memory is just one of them.
  - Safety dimensions are not orthogonal.
- Towards **profiles** combining **multiple approaches**:
  - Coding rules.
  - Support libraries.
  - Static analysis.
- Prepare for a **future world** with **contracts**.

Some thoughts

- **Memory safety** is a **real concern**.
  - **Reminder**: There is no silver bullet.
- There **many dimensions** of **safety**.
  - Memory is just one of them.
  - Safety dimensions are not orthogonal.
- Towards **profiles** combining **multiple approaches**:
  - Coding rules.
  - Support libraries.
  - Static analysis.
- Prepare for a **future world** with **contracts**.
- **Evolutionary approaches** have advantages.

using std::cpp 2023 in Madrid (Spain)

think-cell
SCALIAN SPAIN
JFrog CONAN C/C++ package manager
BBVA
B/S/H/
BME a SIX company
GFT
JUNGHEINRICH

- **Main Conference**: April, 27-28.
  - **Last tickets**: Open March 27th at 9:00.
  - **Invited speakers**: Odin Holmes, Dietmar Kuhl, Mateusz Pusz, Timur Doumler.
- **Trainings**: April, 26th.
  - Coroutines with Mateusz Pusz.
  - SIMD wiht Mathias Kretz.

https://eventos.uc3m.es/go/usingstdcpp-2023
https://usingstdcpp.org/

References

- The C++ Core Guidelines:
  - **https://github.com/isocpp/CppCoreGuidelines**
- GSL Library:
  - **https://github.com/microsoft/GSL**
- Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. Bjarne Stroustrup.
  - **wg21.link/p2816**

Towards a safer C++

Embo++
Bochum, Germany

J. Daniel Garcia

University Carlos III of Madrid
Spain

March 2023