Delivering safe C++

Bjarne Stroustrup
Columbia University
www.stroustrup.com

Overview

- The challenges of safety
  - What is “safety”?
- C++ Evolution
  - with a focus on safety
- C++ Core Guidelines
  - How to write good contemporary C++
- Safety Profiles
  - How to guarantee safety

A cause for concern (not panic)

- The overarching software community across the private sector, academia, and the U.S. Government have begun initiatives to drive the culture of software development towards utilizing memory safe languages.
- ...
- 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®.
  - NSA: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2739r0.pdf

To contrast (not a cause for complacency)

- **February 2023 Headline: C++ still unstoppable**
- Last month, C++ won the TIOBE programming language of the year award for 2022. C++ is continuing its success in 2023 so far. Its current year-over-year increase is 5.93%. This is far ahead of all other programming languages, of which the most popular ones only gain about 1%.

Feb 2023	Feb 2022	Change	Programming Language	Ratings	Change
1	1		Python	15.49%	+0.16%
2	2		C	15.39%	+1.31%
3	4	^	C++	13.94%	+5.93%
4	3	v	Java	13.21%	+1.07%
5	5		C#	6.38%	+1.01%
6	6		Visual Basic	4.14%	-1.09%

- But what does Tiobe measure?
- But this implies that what we do matters to billions of people – for good and bad

We must address the "safety" issue

- There is a real, serious problem for many uses and users
  - Bugs and security violations from bad code in various language (incl. C++) and other causes
  - Diversion of resources from C++ to other languages
  - Discouraging people from learning C++
- Massive improvements are possible in many areas
- C++ has a massive image problem ("C/C++")
  - And it is getting worse
- Governments and large corporations can coerce developers
- Ignoring the safety issues now would hurt large sections of the C++ community and undermine much of the other work we are doing to improve C++.
  - So would focusing exclusively on safety
- Offering guaranteed safety will be in the best tradition of C++

We == WG21 + community

There is no C/C++ language.
Write contemporary C++

An opportunity

Complete type-and-resource safety

- Is an ideal (aim) of C++
  - From very early on (1979)
  - But “being careful” doesn’t scale
- Requires judicious programming techniques
  - Supported by libraries
  - Enforced by language rules and static analysis
  - The basic model for achieving that can be found in A brief introduction to C++'s model for type- and resource-safety (2015) and Type-and-resource safety in modern C++ (2021).
- No limitations of what can be expressed
  - Compared to traditional C and C++ programming techniques
- No added run-time overhead
  - Except necessary range checking

Type-and-resource safety

- Every object is accessed according to the type with which it was defined (type safety)
- Every object is properly constructed and destroyed (resource safety)
- Every pointer either points to a valid object or is the **nullptr** (memory safety)
- Every reference through a pointer is not through the **nullptr** (often a run-time check)
- Every access through a subscripted pointer is in-range (often a run-time check)

- That
  - Implies range checking and elimination of dangling pointers (“memory safety”)
  - Is just what C++ requires
  - Is what most programmers have tried to ensure since the dawn of time
- The rules are more deduced than invented

Enforcement rules are mutually dependent.
Don’t judge individual rules in isolation

Constraints on a solution

- C++ must serve wide variety of users/areas
  - One size doesn’t fit all
  - C++ is (also) a systems programming language – we can’t “outsource” dangerous operations to some other language
- We can’t just break billions of lines of existing code
  - Even if we wanted to – major users would insist on compatibility (probably compatibility by default)
  - We can change the use of C++
- We can’t just “upgrade” millions of developers
  - And teaching material, courses, videos, books, articles
- If you want a shiny new language, please go ahead
  - But it won’t be C++ or the job of WG21

Stability is a feature

- But we ***must*** improve

Challenges

- Describe a type-safe C++ use
  - No violations of the static type system
  - No resource leaks
- Convince developers to use that safe (or just safer) styles of use
  - Except where it is not appropriate
    - Direct use of system and hardware resources
    - Need for ultimate efficiency
    - Implementation of code that cannot be proven safe (e.g., some linked structures)
- Get this to work at scale
  - Not just "academic" examples
- Note: there is lots of great C++ "out there"
  - For any definition of "great" including "reliable over decades"

An opportunity
to finally accomplish
one of C++'s original aims

Safety is not just type safety

- **Logic errors**: perfectly legal constructs that don’t reflect the programmer’s intent, such as using < where a <= or a > was intended.
- **Resource leaks**: failing to delete resources (e.g., memory, file handles, and locks) potentially leading to the program grinding to a halt because of lack of available resources.
- **Concurrency errors**: failing to correctly take current activities into account leading to (typically) obscure problems (such as data races and deadlocks).
- **Memory corruption**: for example, through the result of a range error or by accessing and memory through a pointer to an object that no longer exists thereby changing a different object.
- **Type errors**: for example, using the result of an inappropriate cast or accessing a union through a member different from the one through which it was written.
- **Overflows and unanticipated conversions**: For example, an unanticipated wraparound of an unsigned integer loop variable or a narrowing conversion.
- **Timing errors**: for example, delivering a result in 1.2ms to a device supposedly responding to an external event in 1ms.
- **Allocation unpredictability**: for example, ban on free store allocation “after the engine starts.”
- **Termination errors**: a library that terminates in case of “unanticipated conditions” being part of a program that is not allowed to unconditionally terminate.

Security is not just memory safety

- Physical break-ins
- Spies (insider attacks)
- Spear phishing
- Door rattling
- Denial of service attacks
- SQL injection
- Corrupted input/Data

- Rule of thumb: always first attack the weakest link

Languages are not safe – uses can be

- All “safe” general-purpose languages have “escape clauses”
  - To access system and hardware resources
    - E.g., the operating system
  - To improve efficiency of key abstractions
    - E.g., linked data structures
  - “Trusted” code segments to work with unsafe code segments
    - Libraries
    - Code written under les stringent rules (e.g., old code)
    - Code written in other languages
- Often the ``escape clause’’ is C++
  - So, C++ needs to be able to do unsafe things
  - Do so efficiently
  - While being safe where it matters
    - guaranteed
    - Preferably by default.

This is ***not*** an excuse to ignore safety issues

We ***must*** improve where needed

C++ evolution – with a focus on safety

- It all started decades ago
  - 1979
- C++ was designed to be an evolving language
  - Relying on feedback is just good engineering
- Static type safety was an ideal from day #1
  - It still is
  - But an elusive ideal under real-world constraints
  - In general, “perfection” is an elusive concept

Being the world’s best
at just one or two things
isn’t enough

The earliest aims – Day #1 (1979)

- Efficient use of hardware – based on C
  - Direct access to hardware and system resources
  - No elaborate run-time support (e.g., no GC)
- Manage complexity – based on Simula
  - Classes
  - Strong static type checking

Overarching aims

I needed a tool for building a system
No language could do both

Only a few years earlier it was "well known" that you couldn't write good systems software except in assembler

Argument type checking – 1980

- C in 1979 aka “Classic C”

```
double sqrt();
double x = sqrt(2);           /* crash */
```

- C with classes in 1980 (C and C++ today)

```
double sqrt(double);          // argument type required
double x = sqrt(2);
```

Strong static type checking was and is the ideal

Argument type checking enforced – 1983

- Somewhat controversial
  - “I have to look at a declaration to figure out what a call means?”
- Definitely incompatible
  - A major point against C++ for people who didn’t like something or other about C++
  - But upgrade was easy: “I convert 10,000 lines of C to C++ per day”
- Essential for
  - type checking
  - overloading
  - adding user-defined types
  - consistent linking
- Type-safe linking
  - Ensuring much better consistency across separate compilations
  - Not perfect, but now we can use C++20 modules

A few things are so important
that you must introduce incompatibilities

But you always pay dearly for that:
developers and teachers can be very conservative

Key idea: “Represent concepts in code”

- Direct representation of ideas in code
  - Focus on classes
- Make code more declarative
- Make more information available to compilers
- Early examples
  - Vector
  - String
  - File handle
  - Concurrent task
  - Message queue
  - Coroutine
  - Hash table
  - Graphical shape
  - Complex number
  - Infinite integer

Some are still not standard

immutability

- Constants
  - **const int x = 7;**
  - **const string s = “Immutable”;**
- Interfaces
  - **const strcpy(char*, const char*);** **//** *today: don’t use strcpy() ; unsafe (potential range error)*
- Historical factoid
  - My original design was for **readonly** and **writeonly**

RAII (1979 and later)

- From my 1979 lab book:
  - A “new function” creates the run-time environment for member functions
  - A “delete function” reverses that
- Later (1983)
  - “new function” -> constructor
  - “delete function” -> destructor
  - Because constructors and destructors were/are not just for the new and delete operators.
- A slightly later formulation (1980s)
  - A constructor establishes a class invariant (if any) for an object
  - A destructor releases all resources owned by the object
- And (1988)
  - “Resource Acquisition Is Initialization”
    - Apologies for that name

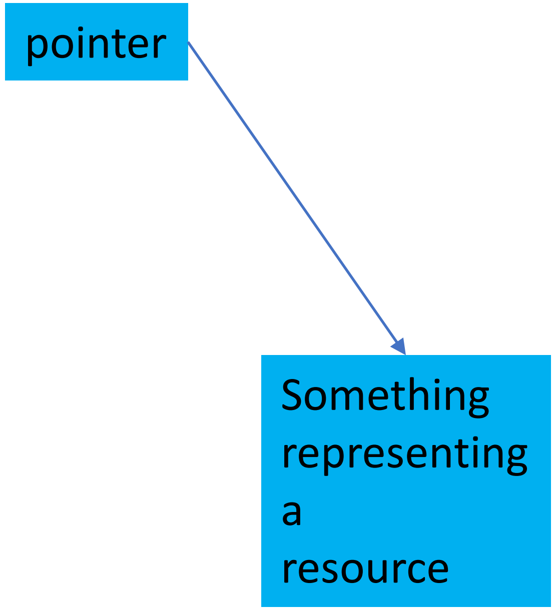
Memory isn’t the only critical resource

- File handles
- Locks
- Sockets
- Shaders
- …

Resources and Errors

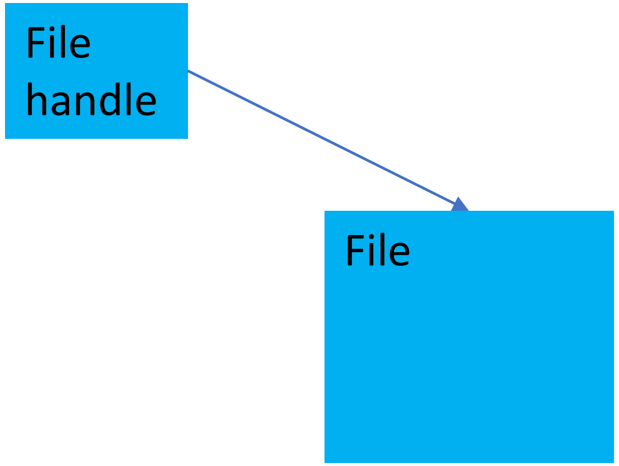
- A resource is something that must be acquired and released after use
  - E.g., files, memory, locks, database transactions, communication channels, GUI connections, threads
  - Explicit release is error-prone
  - Resource exhaustion can render a system inoperative
  - I wouldn’t call a system relying on explicit release safe

```
void f(const char* p)              // unsafe, naïve use
{
    FILE* f = fopen(p,"r");        // acquire
    // use f
    fclose(f);                     // release
}
```



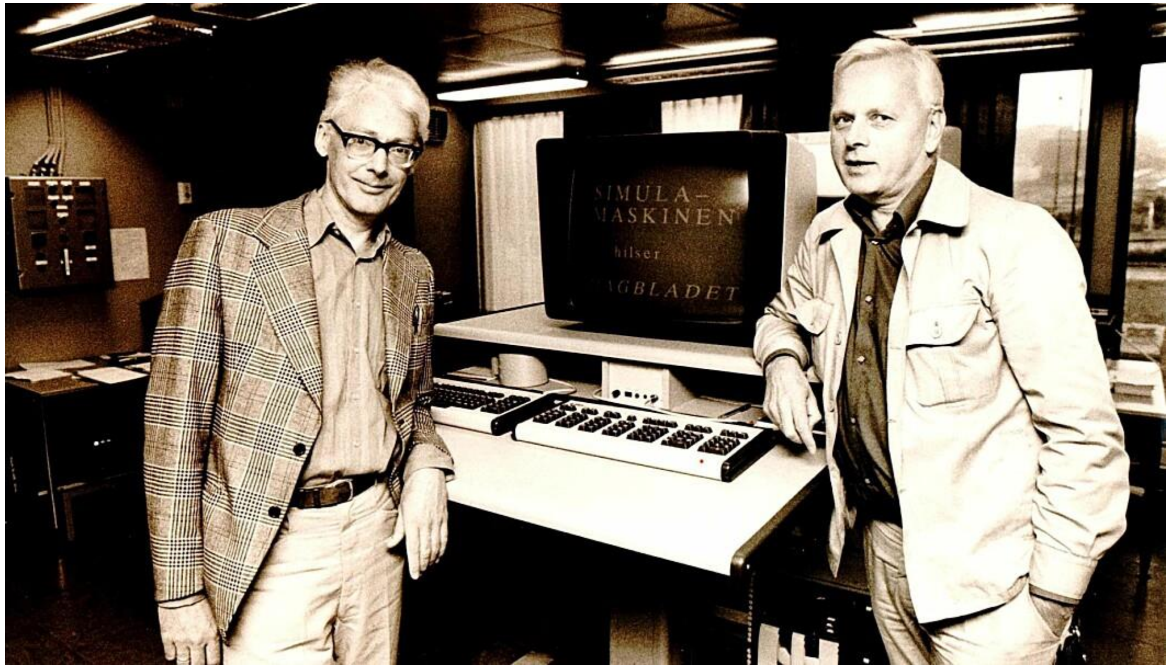
RAII (Resource Acquisition Is Initialization)

```
// use an object to represent a resource
class File_handle {          // belongs in some support library
        FILE* p;
public:
        File_handle(const char* pp, const char* r)
                { p = fopen(pp,r); if (p==0) throw File_error(pp,r); }
        File_handle(const string& s, const char* r)
                { p = fopen(s.c_str(),r); if (p==0) throw File_error(s,r); }
        ~File_handle() { fclose(p); } // destructor
        // copy operations
        // access functions
};

void f(string s)
{
        File_handle fh {s, "r“};        // now: ifstream fh{s}
        // use fh
}
```



OOP (1981 – 1984)

- Encapsulation
- Well-defined interfaces
  - Classes
  - Abstract base classes

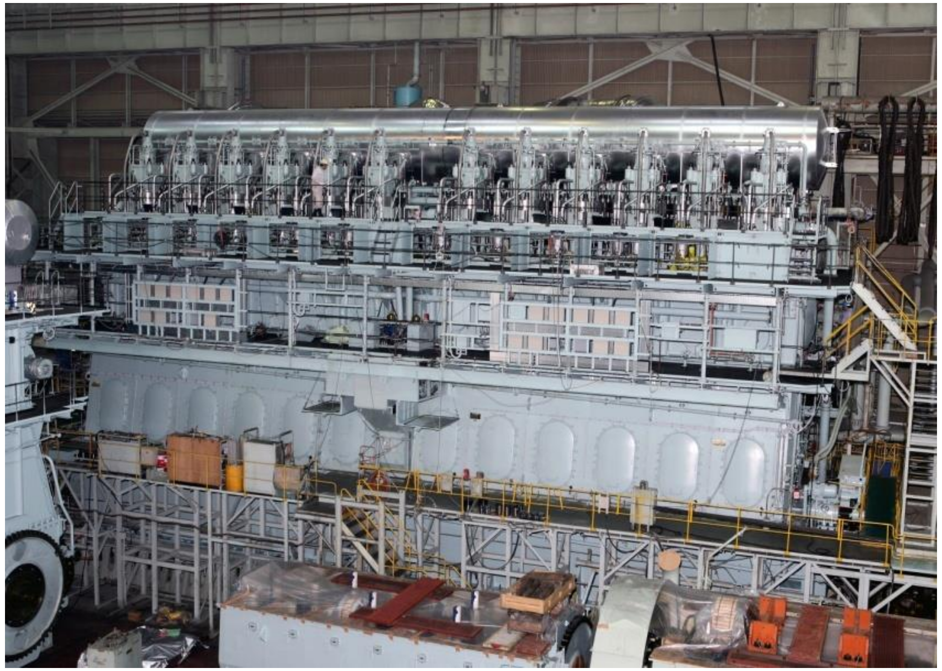


Problem: Too many pointers in interfaces

- Overloading
  - Built-in and user-defined types
  - Operator overloading (initially for resource management)
  - References: simpler and safer argument passing

Later (1988 – 2011)

- Templates
  - Compile-time selection of implementations
  - C++20 finally got concepts – precisely defined interfaces
- Exceptions
  - Guaranteed error-handling – or termination
  - Proper interaction with resource management (RAII)
- Containers
  - No need to fiddle with arrays (and pointers)
  - Enable range checking
- Algorithms
  - C++98: **sort(begin(v),end(v))**
  - C++20: **sort(v)**
- “smart pointers” (resource management pointers)
  - **unique_ptr**, **shared_ptr**



Range-for and span (2011, 2017)

- Enable efficient range checking

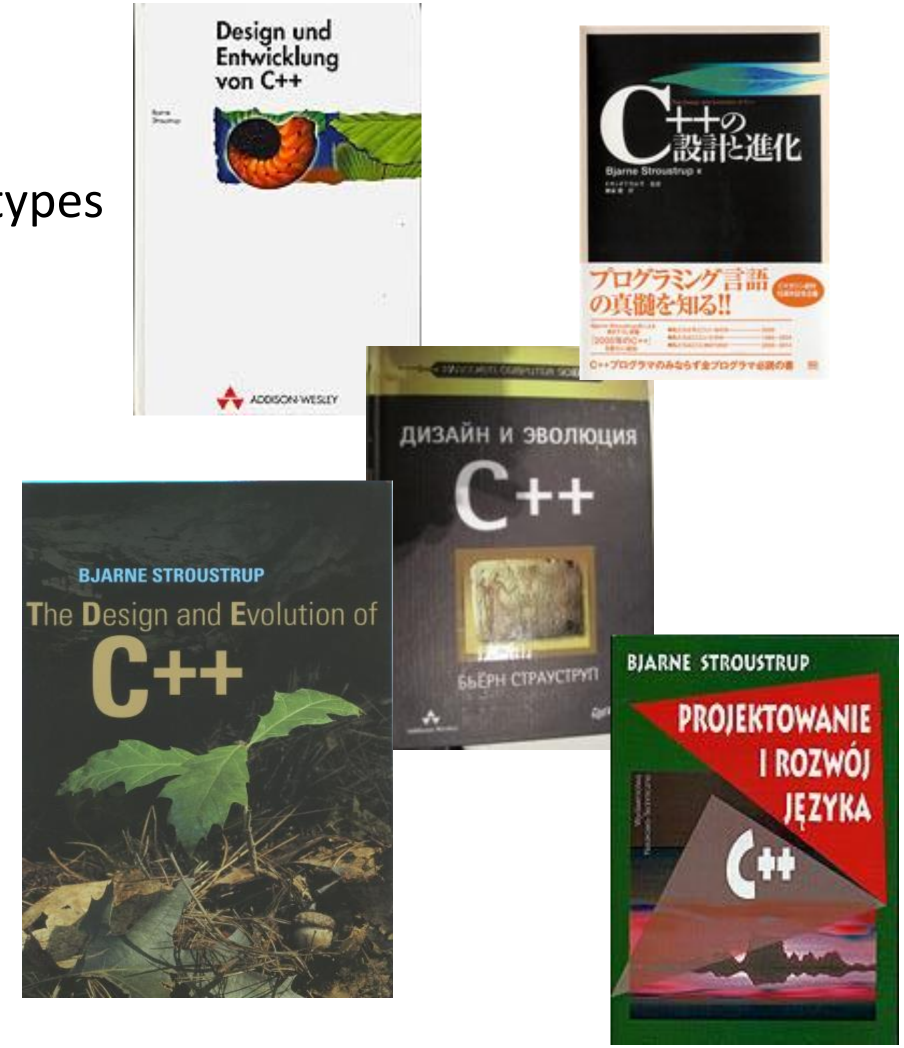
```
void f(int* p, int n)     // what if p==nullptr?  What if n is not the number of elements pointed to?
{
        for (int i =0; i<n; ++n)                    // what if someone "messes with" the control variable?
                cout << p[i] << ' ';
}

void g(span<int> s)
{
        for (const auto& x : s)                     // no loop variable to get wrong
                cout << x << ' ';
}
```

D&E (1994) – select language-technical rules

- No implicit violations of the static type system
- Provide as good support for user-defined types as for built-in types
- Say what you mean
  - Emphasizes declarative styles and abstraction
- Syntax matters (often in perverse ways)
  - In general, verbosity is to be avoided
- Leave no room for a lower-level language (except assembler)
- Preprocessor usage should be eliminated

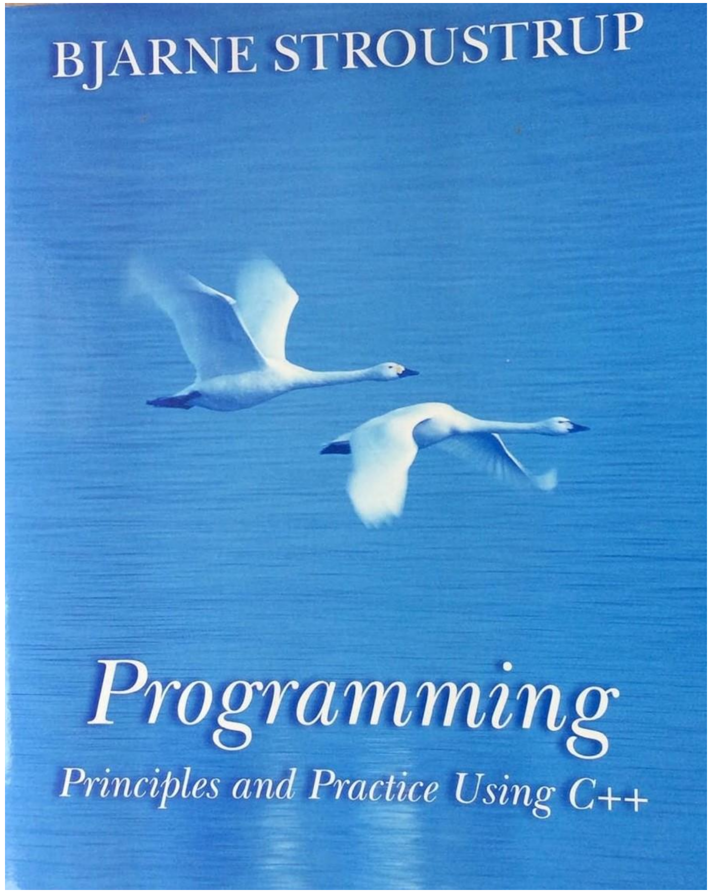
- Missing
  - Make simple tasks simple
  - Make error handling regular

Not articulating those
caused problems that still linger



Benefits come from using C++ well

- Use
    - RAII
    - **const**
    - Containers
    - Resource management pointers
    - Algorithms
    - Range-**for**
    - **span**
    - …
- Avoid
    - Owning raw pointers
    - Subscripting raw pointers
    - Unchecked pointer dereferencing
    - Uninitialized variables
    - …



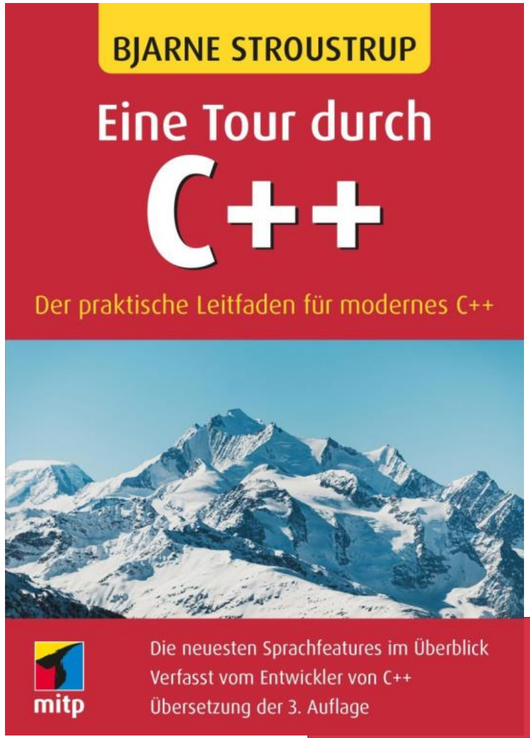
Published 2002

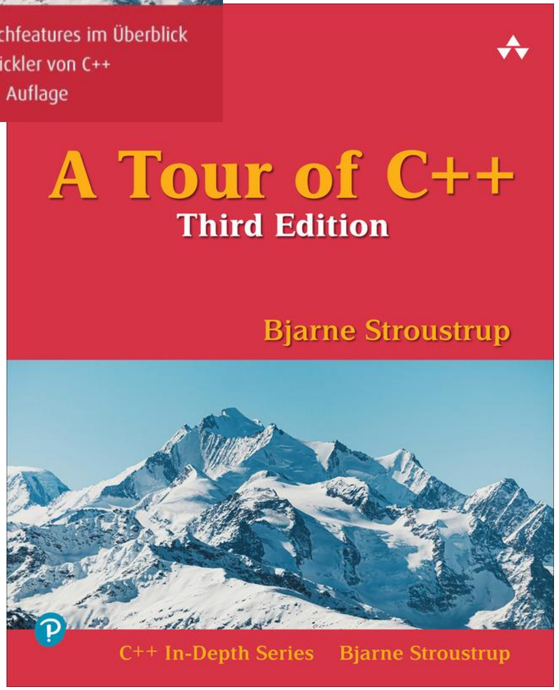
No pointers or arrays until chapter 17.
Then, with warnings

If you want “safe C++”

- Don’t write “C/C++”
  - There is no such language
  - But there is such use
  - Most documented safety and security violations are in such code

- Evolve your style towards what’s provably safe
  - Encapsulate “messy low-level” code





“Being careful” doesn’t scale

- We must
  - Formulate rules for safe use
    - For a variety of safe uses
  - Provide ways of verifying that the rules are adhered to
- Articulate guidelines
  - A start: C++ Core Guidelines
- Enforce guidelines where needed
  - Profiles
    - Enforcing a variety of guidelines
    - Serving a variety of needs

Do you use the safer features?

Consistently?

If not, why not?

State of affairs

- The parts of what I am describing have been tried
  - Many “at scale” (e.g., range-checked **string**, **vector**, and **span**)
  - But nowhere has all been integrated into a single system and systematically enforced
- Much is influenced by work on the C++ core guidelines
  - But this is not just about guidelines
  - We need enforced rules
- The aim is guaranteed type-and-resource safe C++
  - And more, e.g., safe arithmetic
  - Paths to gradual adoption
  - Major improvements can be achieved today, e.g., consistent range checking)
- This is not just about safety
  - better use of the type system improves productivity
  - and often also improves performance

C++ Core Guidelines

General strategy

- Rely on static analysis to eliminate potential errors
  - Static analysis is impossible for arbitrary code
  - Global static analysis is very expensive
- Rely on rules to simplify the language used
  - to the point where local static analysis is possible
- Provide libraries to make relying on the rules practical
  - Pleasant to use
  - Efficient to run

High-level rules – "Philosophy"

- Provide a conceptual framework
  - Primarily for humans
- Many can't be checked completely or consistently
  - *P.1: Express ideas directly in code*
  - *P.2: Write in ISO Standard C++*
  - *P.3: Express intent*
  - *P.4: Ideally, a program should be statically type safe*
  - *P.5: Prefer compile-time checking to run-time checking*
  - *P.6: What cannot be checked at compile time should be checkable at run time*
  - *P.7: Catch run-time errors early*
  - *P.8: Don't leak any resource*
  - *P.9: Don't waste time or space*
  - *P.10: Prefer immutable data to mutable data*
  - *P.11: Encapsulate messy constructs, rather than spreading through the code*
  - *P.12: Use supporting tools as appropriate*
  - *P.13: Use support libraries as appropriate*

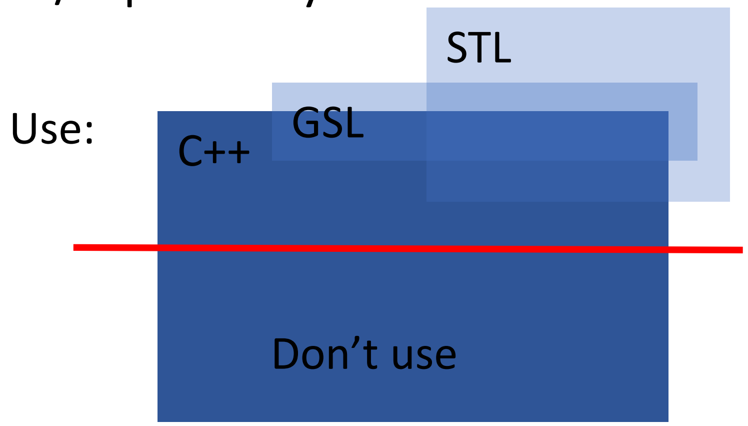
Lower-level rules

- Provide enforcement
  - Some complete
  - Some heuristics
  - Many rely on static analysis
  - Some beyond our current tools
  - Often easy to check “mechanically”
- Primarily for tools (static analysis)
  - To allow specific feedback to programmer
- Help to unify style
- Not minimal or orthogonal

- *F.16: Use **T*** or **owner<T*>** to designate a single object*
- *R.2*: In interfaces, use raw pointers to denote individual objects (only)
- *ES.20: Always initialize an object*

Subset of superset

- Simple sub-setting doesn't work
  - We need the low-level/tricky/close-to-the-hardware/error-prone/expert-only features
    - For implementing higher-level facilities efficiently
    - Many low-level features can be used well
  - We need the standard library
- Extend language with a few abstractions
  - ***Use*** the STL
  - ***Add*** a small library (the GSL)
    - ***No*** new language features
    - Messy/dangerous/low-level features can be used to implement the GSL
  - ***Then*** subset
- What we want is "***C++ on steroids***"
  - Simple, safe, flexible, and fast
  - Not a neutered subset



No change of meaning:
The resulting code is ISO C++

We can

- Eliminate
  - Uninitialized variables
  - Range errors
  - Nullptr dereferencing
  - Resource leaks
  - Dangling references
  - Unions (use variants)
  - Casts
  - Underflow and overflow
  - Data races

Not discussed in this talk

- “Dangerous features” can be used in explicitly unverified code
- “Just test everywhere at run time” is ***not*** an acceptable answer
  - Hygiene rules + Static analysis + Run-time checks

Uninitialized variables

- Initialize every variable
  - Simple and effective
  - Implicit initialization of types with default constructors is OK

```
void f()
{
        int x;              // not OK
        int y = 7;          // OK
        std::string s;      // OK
        char* p;            // not OK
}
```

  - Mark exceptional cases **[[uninitialized]]**
    - E.g., I/O buffers

Range checking

- Don’t subscript raw pointers
  - **void f(int* p, int x) { p[x] = 7; }** *// not OK*
  - Except in the implementation of abstractions
- Use abstraction with sufficient data to range check
  - E.g., **vector** and **span**
  - **void f(span<int> s, int x) { s[x] = 7; }** *// OK for a checking span*
- Run-time range check every subscript operation
  - E.g., **vector** and **span** must be range checked
- Range-**for**
  - Prefer over C-style for loops
- Algorithms
  - Prefer range algorithms

Many more details
in the references

Null pointer dereferencing

- Don't dereference an unchecked pointer
  - **void f0(int* p) { *p = 7; }** *// not OK*
  - **void f1(int* p) { if (p) *p = 7; }** *// OK*
  - **void f2(not_null<int*> p) { *p = 7; }** *// OK (not_null constructor checks)*
  - **void f3(span<int> s) { s[2] = 2; }** *// OK (for checked span)*
- Except in the implementation of abstractions

No resource leaks

- We know how
  - Root every object in a scope
    - **vector<T>**
    - **string**
    - **ifstream**
    - **unique_ptr<T>**
    - **shared_ptr<T>**
  - RAII
    - "No naked **new**"
    - "No naked **delete**"

A resource leak is potentially damaging

Pointer Misuse

- Many (most?) uses of owning pointers in local scope are not resource safe

```
void f(int n, int x)        // not OK
{
        Gadget* p = new Gadget{n};          // look I'm a java programmer! ☺
        // …
        if (x<100) throw std::runtime_error{"Weird!"};        // leak
        if (x<200) return;                                    // leak
        // …
        delete p;                           // I want my garbage collector! ☹
}
```

- But
  - garbage collection would not release non-memory resources
  - why use a "naked" pointer?

Resource Handles and Pointers

- Use a cheap, safe “smart” pointer
  - A **std::unique_ptr** releases its object at when it goes out of scope

```
void f(int n, int x)        // OK
{
        auto p = make_unique<Gadget>(n);                    // manage that pointer!
        // …
        if (x<100) throw std::runtime_error{"Weird!"};      // no leak
        if (x<200) return;                                  // no leak
        // …
}
```

- This is simple and cheap
  - No more expensive than a “plain old pointer” used correctly

Resource Handles and Pointers

- But why use a pointer at all?
  - If you can, just use a scoped variable

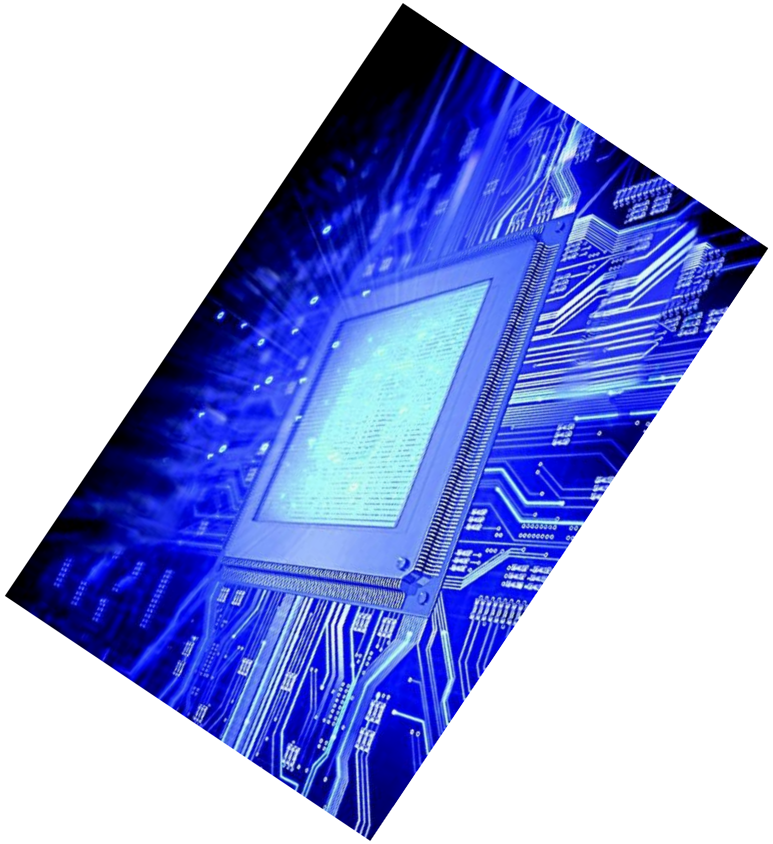
```
void f(int n, int x)        // OK, and better
{
        Gadget g {n};
        // …
        if (x<100) throw std::runtime_error{"Weird!"};        // no leak
        if (x<200) return;                                     // no leak
        // …
}
```

A “naked new” is a code smell.
Encapsulate!

- No explicit resource management
  - Code not littered with (easy to forget) **try-catch**
  - No spurious allocations/deallocations
  - No pointers

Dangling pointers

- We ***must*** eliminate dangling pointers
  - Or type safety is compromised
  - Or memory safety is compromised
  - Or resource safety is compromised
  - By “pointer” I mean anything that directly refers to an object
- Eliminated by a combination of rules
  - Distinguish owners from non-owners
  - Assume raw pointers to be non-owners
  - Catch all attempts for a pointer to “escape” into a scope enclosing its owner’s scope
    - **return**, **throw**, out-parameters, long-lived containers, …
- Statically verified



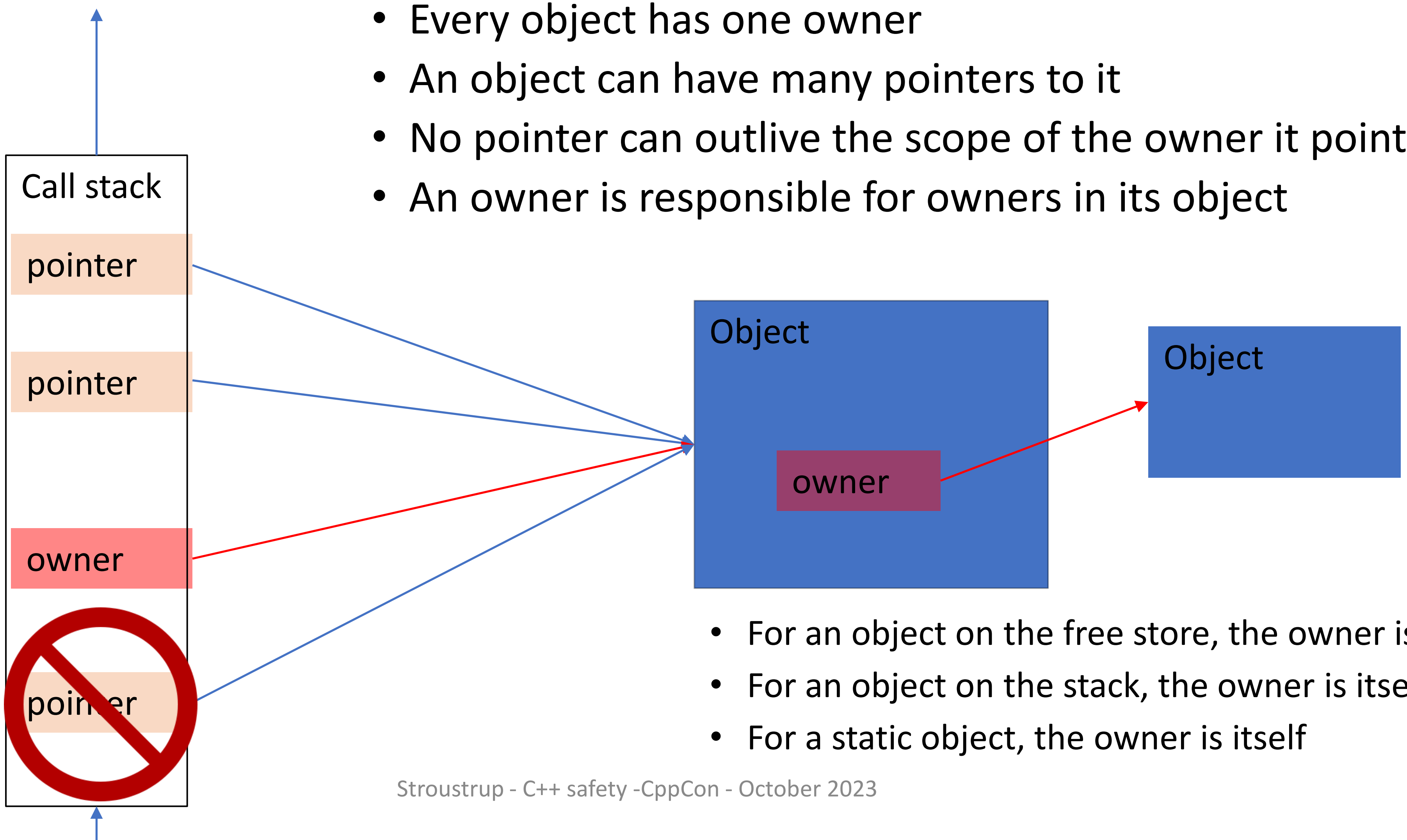
Dangling pointers

- Ensure that no pointer outlives the object it points to (statically)

```
void f(X* p)
{
      // …
      delete p;            // not OK: delete non-owner
}

void g()
{
      X* q = new X;        // not OK: assign object to non-owner
      f(q);
      // … do a lot of work here …
      q->use();            // Make sure we never get here
}
```

Owners and pointers

- Every object has one owner
- An object can have many pointers to it
- No pointer can outlive the scope of the owner it points to
- An owner is responsible for owners in its object



- For an object on the free store, the owner is a pointer
- For an object on the stack, the owner is itself
- For a static object, the owner is itself

How to avoid/catch dangling pointers

- Any pointer can be passed as an argument, but not stored into a surrounding scope

```
int* glob = nullptr;

void g(int* p)
{
        glob = p;               // not OK: trying to store away the pointer
}

void f(int* p)
{
        int x = 4;
        g(&x);                  // OK
        g(p);                   // OK
}
```

- Enforcement is difficult if returned pointers are calculated in complex code
  - Don’t write such code
  - Get a good static flow analyzer (perfect analysis is impossible in general)

How to avoid/catch dangling pointers

- Remember containers

```
void g(vector<int*>& v)
{
        // …
}
vector<int*> f(int* p)
{
        int x = 4;
        vector<int*> res = {p, &x, new int{7}};        // OK (except for the ownership violation)
        g(res);                                        // OK
        return res;                                    // not OK
}
```

Invalidation

- This must be prevented:

```
void f(vector<int>& vi)
{
	vi.push_back(9);  // may relocate vi's elements
}

void g()
{
	vector<int> vi { 1,2 };
	auto p = vi.begin();		// point to first element of vi
	f(vi);
	*p = 7;			// UB, may appear to work correctly
}
```

- Can be done statically
  - has been done

Invalidation

- "invalidated" == "when a pointer points to an element of a container that may have reallocated its elements"
- "container" == "any object that directly or indirectly could contain a pointer"
  - Classes with pointer members
    - E.g., **vector** and **thread**
  - Smart pointers (they are classes, e.g., **unique_ptr** and **shared_ptr**)
  - Threads (they are classes)
  - Lambdas (they are classes, and remember capture-by-reference)
  - Pointers to pointers
  - References to pointers
  - Arrays of pointers
- "pointer" == "anything that directly refers to an object"

Invalidation assumptions

- No **const** member function invalidates (easily validated)
- Any non-**const** member function invalidates
- Add **[[not_invalidating]]** to non-**const** member functions that do not invalidate
  - E.g., **vector::operator[]()**
  - Easily validated

Not yet Core Guideline

How do we represent ownership?

- High-level: Use an ownership abstraction
- Low-level: Primarily for communicating with C-style interfaces
  - mark owning pointers **owner**
  - Currently, **owner<T*>** is just an alias for **T***
  - An **owner** must be **delete**d or passed to another **owner**
  - A non-**owner** may not be **delete**d

Stay high level
except to implement
necessary abstractions
and interfaces

- Note
  - I talk about pointers
  - What I say applies to anything that refers to an object
    - References
    - Containers of pointers
    - Smart pointers
    - Pointers to pointers
    - Lambda captures
    - ..

Efficient communication
with C is essential

You can copy the pointer of an **owner** but not the ownership

- **owner** is low level, prefer **unique_ptr** or other ownership abstractions

```
void f1(owner<int*> p)          // f1() used_to_take_a_raw_pointer
{
    // ...
    delete p;    // required or f() must transfer ownership before returning
}

void f2(int* p)
{
    // ...
    delete p;    // error: p is not an owner
}

void f3(int* p1, owner<int*> p2)
{
    f1(p1);    // error: p1 is not an owner
    f2(p1);    // call OK, as ever, but f2()'s definition is not
    f1(p2);    // transfers ownership; p2 in now invalid in f3() because f1() will delete it
    f2(p2);    // call OK: a non-owning copy of p2 is passed
}
```

Enforced by static analyzer or compiler

Where do we go from here?

- We can write good C++
  - Too many developers don’t
  - Guidelines are not enough, we need guarantees
  - We can (and do) use static analyzers but we need standards
- Alternatives: We can
  1. Change C++
  2. Start using another language
  3. Enforce a variety of guidelines: Profiles

Alternative 1: Fix C++

- How?
  - There are many incompatible ideas
    - => years of delay and chaos
  - A single “cleaned up” language cannot support the wide variety of safety notions
  - A “cleaned up C++” would have to interoperate with “classical C++” code “forever”
    - Serious design constraint
- There are billions of lines of C++
  - Much critical
  - Much high quality
  - Gradual adoption is essential
  - Partial adoption is essential (“safety critical code only”)
- Compatibility
  - C++ was meant to evolve
  - But we can’t break massive amounts of existing code

Alternative 2: Use another language

- The popular alternative with supporters of other languages
  - “safety” is used as an argument
    - Often mixing C and C++ in arguments
    - Often ignoring C++’s strengths
    - Often ignoring the weaknesses of alternatives
    - Often the safety mentioned is just memory safety
    - Often the need for unsafe constructs is unmentioned
    - Often the need to inter-operate with other languages is unmentioned
    - Often the cost of conversion is underestimated
- This is natural
  - Human nature
- Which other language?
  - Must interoperate with C and C++ code “forever”
    - Serious design constraint

Alternative 2: Use another language

- There are many “newer languages”
  - All different
- Each new tool chain require resources and expert users
  - Code from each language needs to communicate with code in all(?) other languages in a system
  - Each language has its own update schedule
- Every new language claims to be simpler, cleaner., safer, and more productive than C++
  - Languages grow significantly in size and complexity over time
- The old languages and systems don’t go away
- Often the claimed superiority of a new language over C++ is in a limited domain
  - Often compared to C/C++, rather that to C++
- Enthusiasts deliver better than average developers
  - Once the number of users of a language increases, their quality and enthusiasm converge towards the industry average

Consider converting a 10-million-line system

- Needing high reliability and high performance
  - i.e., the kind of system that's critical in some way
- A good developer completes N lines of tested production-quality code a day
  - What is N? 5?, 10?, 100?
  - Say – optimistically – 2,000 lines/year
  - Say – optimistically – that a reimplementation (without feature creep) could be 5 million lines
  - Then it would take 500 developers 5 years to complete the new system
  - The old system would have to be maintained for those 5 years say by 50 developers
- What is the loaded salary of a good developer?
  - Employee's compensation plus employer's other cost (e.g., buildings, computers, heating)
  - Say, $500,000 in the US
- So, the cost would be 550*5*$500,000 == ~$1,400,000,000
  - Vs. ~$125,000,000 for normal maintenance and development
  - Roughly $1B added cost

For a 1-million line system divide by 10

For a 100-million line system multiply by 10

Consider converting a 10-million-line system

- Assumptions
  - US developers
    - developers with relevant experience in the domain and the new language can be found
    - maybe outsourcing could cut cost, but has its own problems and costs
  - The new system would be half the size of the old one (unlikely, but)
    - Better understanding from the start
    - Better language
    - Better tooling
    - The new language/languages can cope with the messy parts of the system
    - No feature creep
  - The new system would work and be delivered on time
    - Large projects often have time and cost overruns

Consider converting a 10-million-line system

- There are people for whom $1B or $100M are not scary numbers
- There are people who consider 10M line systems medium sized

- I consider these numbers an argument for an incremental and evolutionary approach.
  - Obviously, that could be in C++ or in a combination of C++ and other languages
  - Either way, C++ will play a major role for decades to come
  - We can and must improve C++ – as ever

- ***A complex system that works is invariably found to have evolved from a simple system that worked.*** – Gall's law

Alternative 3: C++ Profiles

- How to ***guarantee*** safety?
  - For a variety of definitions of safety
  - For a variety of users
  - Mostly statically
- Guidelines are not enough
  - “being careful” is not verifiable
  - The source code doesn’t directly express what’s to be guaranteed
  - Tool chains are complex and vary among systems
- How can we gradually improve safety?

Profiles summary

- The meaning of all constructs is defined by the ISO C++ standard
- The most fundamental guarantee offered is complete type-and-resource safety

- Ownership (that is, the obligation to delete/destroy) constitutes a DAG
- A pointer is the **nullptr** pointer, or is valid (i.e., points to an objects)
- A pointer (outside the implementation of abstractions) points to a single object
- There is a way to enforce **nullptr** checking and range checking
- Subscripting is done on abstractions such as **span** and **vector**, not on pointers

- Gradual conversion from older code to modern code offering guarantees is supported
- The set of guarantees is open
- A set of fundamental guarantees are standard
- There are rules for composing code fragments supporting different guarantees

- The set of guarantees assumed by and provided by a unit of code is stated in the code

Many notions of safety

- **Logic errors**: perfectly legal constructs that don’t reflect the programmer’s intent, such as using < where a <= or a > was intended.
- **Resource leaks**: failing to delete resources (e.g., memory, file handles, and locks) potentially leading to the program grinding to a halt because of lack of available resources.
- **Concurrency errors**: failing to correctly take current activities into account leading to (typically) obscure problems (such as data races and deadlocks).
- **Memory corruption**: for example, through the result of a range error or by accessing and memory through a pointer to an object that no longer exists thereby changing a different object.
- **Type errors**: for example, using the result of an inappropriate cast or accessing a union through a member different from the one through which it was written.
- **Overflows and unanticipated conversions**: For example, an unanticipated wraparound of an unsigned integer loop variable or a narrowing conversion.
- **Timing errors**: for example, delivering a result in 1.2ms to a device supposedly responding to an external event in 1ms.
- **Allocation unpredictability**: for example, ban on free store allocation “after the engine starts.”
- **Termination errors**: a library that terminates in case of “unanticipated conditions” being part of a program that is not allowed to unconditionally terminate.

Type-and-resource safety

- Every object is accessed according to the type with which it was defined (type safety)
- Every object is properly constructed and destroyed (resource safety)
- Every pointer either points to a valid object or is the **nullptr** (memory safety)
- Every reference through a pointer is not through the **nullptr** (often a run-time check)
- Every access through a subscripted pointer is in-range (often a run-time check)

- That
  - Implies range checking and elimination of dangling pointers (“memory safety”)
  - Is just what C++ requires
  - Is what most programmers have tried to ensure since the dawn of time
- The rules are more deduced than invented

Enforcement rules are mutually dependent.
Don’t judge individual rules in isolation

What about C/C++?

- Arbitrary C or C++ code is too complex for static analysis
  - Halting problem
  - Dynamic linking
  - Cost of global analysis
  - Direct access to hardware
- Arbitrary C or C++ forces us to deal with too low an abstraction level
  - Hides complexity in messy old-style code
  - Pushes complexity into the applications code => bugs
  - Regression into the 1980s
  - “C/C++” will never be safe – for any definition of “safe”
- We care about performance as well as type-and-resource safety
  - Eventually much higher productivity

Strategy: Safety Profiles

- Our approach is “a cocktail of techniques” not a single neat miracle cure

- Static analysis (maybe in compilers)
  - to verify that no unsafe code is executed
- Coding rules
  - to simplify the code to make industrial-scale static analysis feasible
- Libraries
  - to make such simplified code reasonably easy to write
  - to guarantee run-time checks where needed

Safety Profiles

- Profiles: A coherent set of guarantees
  - Not just a lot of unrelated tests
  - Specified as a set of guarantees, not as a set of specific tests
- A profile: a coherent sets of rules yielding a guarantee
  - Current: bounds, type, memory
  - E.g., type-and-resource-safe, safe-embedded, safe-automotive, safe-medical, performance-games, performance-HPC, EU-government-regulation
  - Must be visible in code
    - To indicate intent
    - To trigger analysis
    - To trigger run-time checks (where needed)

Is this strategy “too novel”?

- “People are afraid of new things.
  You should have taken an existing product and put a clock in it.”
  – Homer Simpson

- Each individual technique has been tried many times before
  - Succeeded for specific tasks
    - E.g., smart pointers, libraries, static analyzers
  - Failed as general solutions
    - Static analysis – doesn’t scale to complete safety
    - Guidelines/rules – aren’t followed without enforcement
    - Foundation libraries – doesn’t give full access to the machine and system
    - Language subsetting – the most dangerous language features are essential (e.g., subscripting of pointers)
- A combined and coherent approach is necessary
  - Similar to Ada’s safety profiles: https://docs.adacore.com/gnathie_ug-docs/html/gnathie_ug/gnathie_ug/the_predefined_profiles.html#the-predefined-profiles

Profiles: beyond guidelines

- A type-and-resource safety profile can insert range checks
  - E.g., bind to a range-checked **string**, **vector**, **span**, etc.
- A performance profile can eliminate safety checks (after proving them redundant)
  - Allowing us to make safer defaults
- In-code annotations
  - To state the programmer's intention so that they can be verified/enforced
  - To detect mixing of profiles
    - We cannot upgrade billions of lines of code at once
    - We want to use several different profiles in larger programs
  - Mixing profiles is a generalization of the safe/unsafe oversimplification
    - "safe code" calling C code (e.g., the OS), assembler, etc.
    - "unsafe code" passing information (e.g., pointers) to "safe code"

Mixing profiles

- We can use subset/superset properties of profiles
  - The profile of an imported module cannot be stated (it might change, but the build-system knows)
- We can’t express the relationships among arbitrary profiles
  - That would be a whole complex language
- Disjoint profiles combine cleanly
  - E.g., type-and-resource and arithmetic
- We mustn’t require users to litter their code with many opt-outs of a stated profile
  - Needing/wanting so would be a sign of a bad program design (like the use of many casts)
  - We mustn’t require users to mark every call to a different profile
- A single “foreign call” annotation is too coarse
  - We need to know which profile or aspect of a profile that’s violated

Mixing profiles

Work in progress

- Still under development
  - Suggestions welcome
- Module-based controls
  - **export My_module [[provide(memory_safety)]];** *// enforce memory_safety for My_module*
  - **import std [[enable(memory_safety)]];** *// enforce memory_safety for uses of std*
  - **Import Mod [[suppress(type_safety)]];** *// don't check type_safety for uses of Mod*
- In-code controls
  - **[[suppress(type_safety)]] X** *// suppress type_safety check for declaration or scope X*
  - **[[enforce(type_safety)]] X** *// enforce type_safety check for declaration or scope X*
- Suggested initial standard profiles
  - **type_safety**: no type-or-resource violations.
  - **range**: no pointer arithmetic; no **nullptr** dereference, **span** and **vector** range **throw** or terminate on violations.
  - **arithmetic**: no overflow, no narrowing conversions, no implicit signed/unsigned conversions.

Summary

Work in progress

- **[[enforce(P)]]**
- **[[provide(P)]]**
- **[[enable(P)]]**
- **[[suppress(P)]]**
- **[[profile(P) = enforce(P1,P2)]]**
- Standard profiles
  - Type-safety
  - Arithmetic
  - Range
- Control of response to run-time violations
- **owner**
- **not_null**
- **not_end()**
- **[[not-invalidating]]**
- **[[uninitialized]]**
- **dynarray** – a vector where the size s fixed at construction
- Constraints on UB

Design Alternatives for
Type-and-Resource Safe C++
P2687

Safety Profiles:
Type-and-resource Safe
programming in ISO Standard C++
P2816

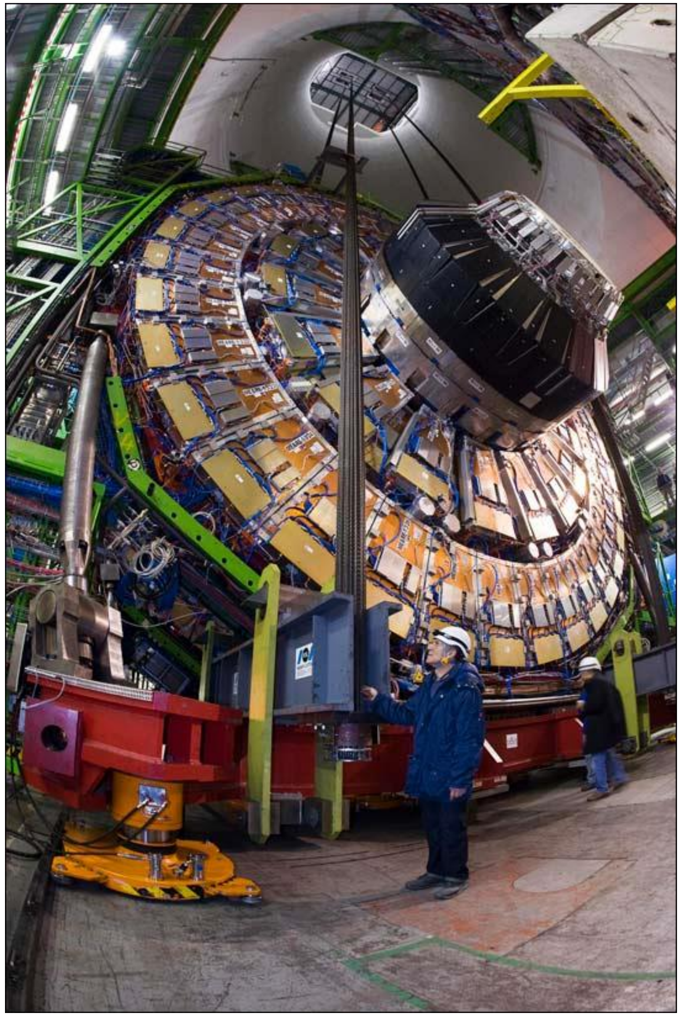
``Are we there yet?''

- We have come a long way
  - From “classic C”
  - From “C with Classes”
  - From C++11
- We can write type-and-resource safe C++
  - Use contemporary C++
  - Avoid C-style and 1980s style C++
- C++ Core Guidelines
  - Directions, rules, and some enforcement
  - But not standardized
  - Uneven enforcement across implementations
- We need to standardize “Profiles”
  - And get implementations deployed
  - Finally reach type-and-resource safe C++!

How can you help

- We need help to refine "profiles"
  - "memory safety" is not sufficient
- Which profiles do we need?
  - Which needs to be standardized?
  - Which first?
- How do we best formalize a profile?
  - By guarantees, not lists of rules.
  - We need work on key examples
- What can we get ready soon?
  - Faster than WG21 can move
  - With current compilers and tool chains
  - Subsets of "ideal profiles"
  - Industry?
- What library components do we need to simplify use?

- See https://github.com/BjarneStroustrup/profiles



References

- M. Wong, H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde: DG Opinion on Safety for ISO C++ . P2759R1. 2023-01-22.
- B. Stroustrup: A call to action: Think seriously about "safety"; then do something sensible about it. P2739R0. 2022-12-6.
- B. Stroustrup and G. Dos Reis: Safety Profiles: Type-and-resource Safe programming in ISO Standard C++. P2816
- B. Stroustrup and G. Dos Reis: Design Alternatives for Type-and-Resource Safe C++. P2687R0. 2022-20-15
- B. Stroustrup: Type-and-resource safety in modern C++. P2410r0. 2021-07-12.
- B. Stroustrup, H. Sutter, and G. Dos Reis: A brief introduction to C++'s model for type- and resource-safety. Isocpp.org. October 2015. Revised December 2015.
- B. Stroustrup: Writing Good C++14 CppCon 2015.
- The C++ Core Guidelines
- The Core Guidelines Support Library (GSL)
- H. Sutter: Lifetime safety: Preventing common dangling. P1179R1. 2019-11-22.
- A Microsoft guide to using the Core Guidelines static analyzer in Visual Studio.
- T. Ramananandro, G. Dos Reis, and X. Leroy: A mechanized semantics for C++ object construction and destruction, with applications to resource management. ACM/SIGPLAN Notices 2012/01/18.
- B. Stroustrup: Thriving in a crowded and changing world: C++ 2006-2020. ACM/SIGPLAN History of Programming Languages conference, HOPL-IV. London. June 2020.
- B. Stroustrup: C++ -- an Invisible Foundation of Everything. ACCU Overload No 161. Feb'21.

We didn't start yesterday