EXCEPTIONS UNDER THE SPOTLIGHT

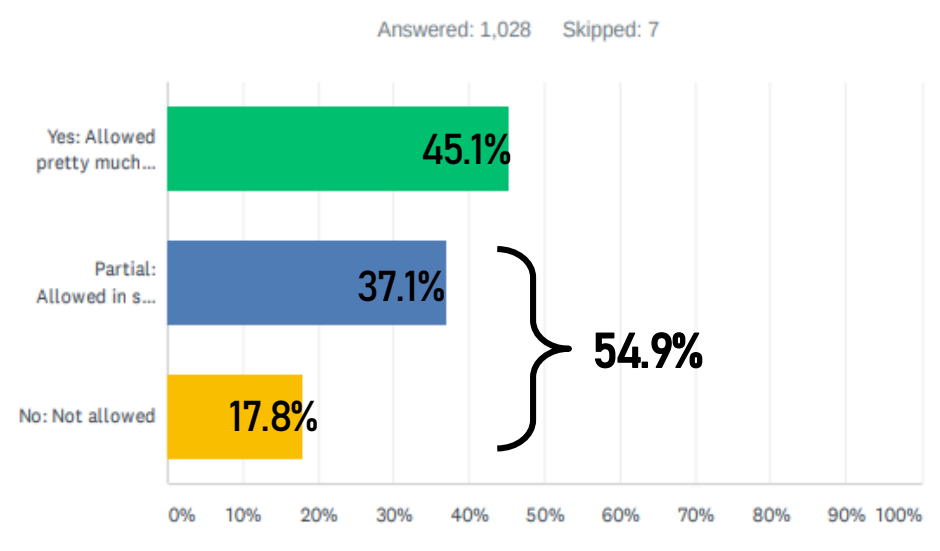
INBAL LEVI

WHO AM I?

- A C++ enthusiast.
- An embedded software engineer at Solar Edge working on smart home.
- One of the organizers of CoreCpp conference and user group.
- A Member of WG21.
- One of the founders of the Israeli NB.
- I've studied physics, I also love math. ☺

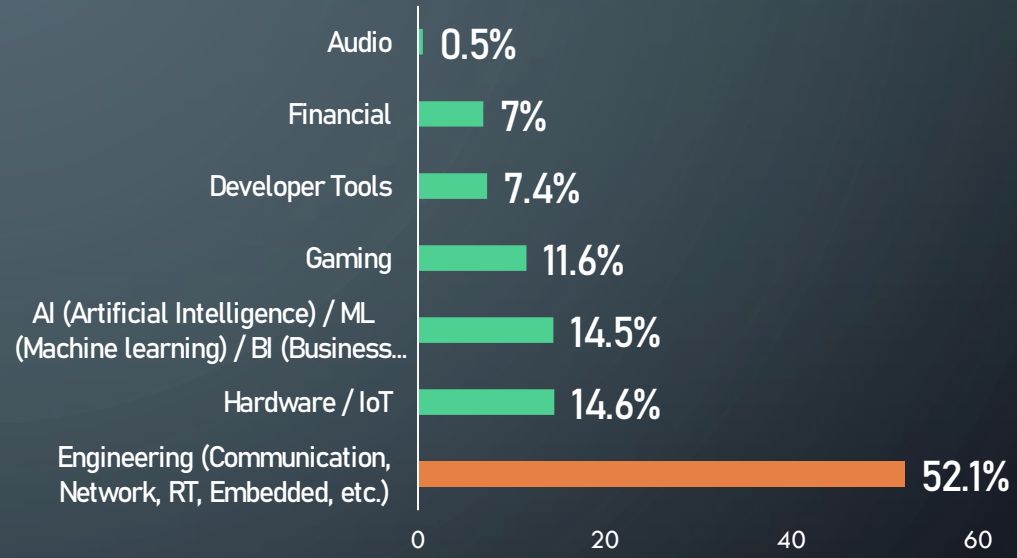
MOTIVATION

Q14 Is throwing exceptions to report errors allowed throughout your current project?



ISO Survey 2020

Q2 What is your company's main domain? (answers may overlap)



C++ Usage In Work Environment

How many of you are using Exceptions in your projects?

MOTIVATION

- Many industries and C++ projects (Such as Embedded, GPUs and Gaming) avoid using exceptions for performance reasons.
- The error handling mechanism was addressed by C++'s Direction group around 2016 (contracts) and again in 2018.
- The exception mechanism was addressed by Herb Sutter (Chair) in 2019:
  P0709: Zero-overhead deterministic exceptions: Throwing values
- The exception mechanism was debated in Prague (last WG21 meeting) with a paper by James Renwick, Tom Spink, Björn Franke:
  Low-cost Deterministic C++ Exceptions for Embedded Systems
- I expect a major change in this mechanism or alternatives in the near future.

OUTLINE

- Part 0: Prologue: What are Exceptions?
- Part I: History and Domain
- Part II: Exceptions Overhead
- Part III: Design Overview
- Part IV: Alternatives
- Part V: What's Next?

DISCLAIMERS

- This is not a talk about “Best practices of exceptions usage”.

  (Though it does refer to some guidelines)

- This is not a talk presenting a production level exceptions extension or library.

  The code presented is a POC (was run on Linux only).

- Concurrency is outside the scope of this talk.

- This talk aims to include a historical perspective. However, all of the decision making described is alleged, I wasn’t there. ☺

PART 0: WHAT ARE EXCEPTIONS

PART 0: WHAT ARE EXCEPTIONS

- A fail-handling mechanism

```
void foo() {

    // do stuff that can fail

    if (fail) {
        // throw
    }
}
```

- You can define your own exception type.
- It is recommended to derive your type from std::exception.
- It is recommended to throw by value, catch by const ref, and re-throw using throw.

PART 0: WHAT ARE EXCEPTIONS

- A fail-handling mechanism

```
void bar() {

    try {
        foo();
    }
    catch (error_type) {

        // do stuff, or re-throw;
    }
    catch (…) {
        // do stuff, or re-throw;
    }
}
```

Identification by type

Re-delegate responsibility

PART 0: WHAT ARE EXCEPTIONS

- A fail-handling mechanism

```
int main()
{
    try {
        bar();
    }
    catch (error_type) {
        // do stuff
    }

    // don't catch;
}
```

"Re-delegate responsibility"

PART I: HISTORY

PART I: HISTORY

- In the beginning, there was c...
- Then came “C with classes”.
- Then came “C++”with exceptions”.

Exceptions were considered at a very early stage of the language.

PART I: HISTORY

- Let's go back a few years, to the era of C. There were two main alternatives:

1. Error codes (Return value)

```
int foo()
{
    // do something, failure returns != 0
}
int bar()
{
    // do something foo related
    if (!foo)
        return (err);
}


int main()
{
    if (!bar())
    {
        // do something
    }
}
```

2. Global (errno / user defined global)

```
int foo()
{
    // do something, failure init errno
}
int bar()
{
    // do something foo related
    if (!errno)           // optional
        return (err);
}

int main()
{
    bar();
    if (!errno)
    {
        // do something
    }
}
```

PART I: HISTORY

- Why exceptions?

(…) The exception handling mechanism
[1] Makes it easier to adhere to the best practices.
[2] Gives error handling a more regular style.
[3] Makes error handling code more readable.
[4] Makes error handling code more amenable to tools.

- The design took approximately 5 years, finished at 1989.

1989

Exception Handling for C++

Andrew Koenig
Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
ark@europa.att.com
bs@research.att.com

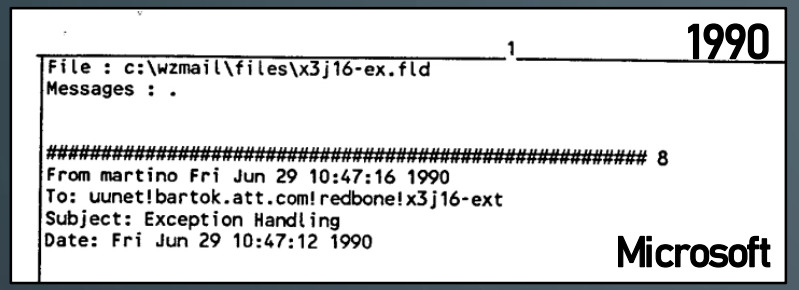
PART I: HISTORY

"In the original design of C++, exceptions were considered, but postponed, because there wasn't time to do a thorough job of exploring the design and implementation issues, and because of fear of the complexity they might add to an implementation.

In particular, it was understood that a poor design could cause run-time overhead and a significant increase in porting times."

B. Stroustrup, The Design and Evolution

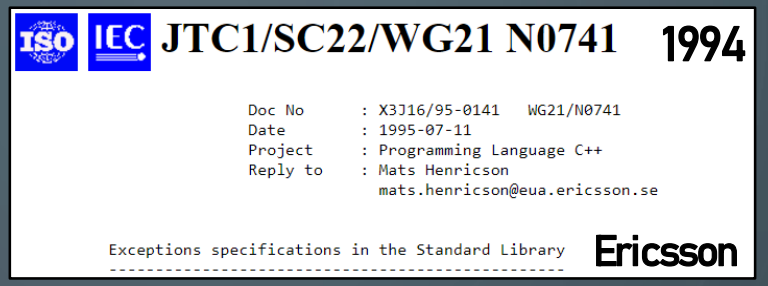
PART I: HISTORY


1990

```
File : c:\wzmail\files\x3j16-ex.fld
Messages : .

                                                  1
################################################## 8
From martino Fri Jun 29 10:47:16 1990
To: uunet!bartok.att.com!redbone!x3j16-ext
Subject: Exception Handling
Date: Fri Jun 29 10:47:12 1990
```

Microsoft


1994

ISO IEC JTC1/SC22/WG21 N0741

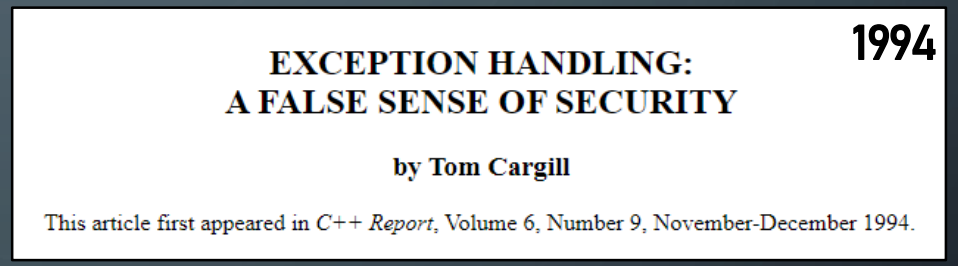
Doc No : X3J16/95-0141 WG21/N0741
Date : 1995-07-11
Project : Programming Language C++
Reply to : Mats Henricson
mats.henricson@eua.ericsson.se

Exceptions specifications in the Standard Library

Ericsson

Apple
IBM
Sun
And others...

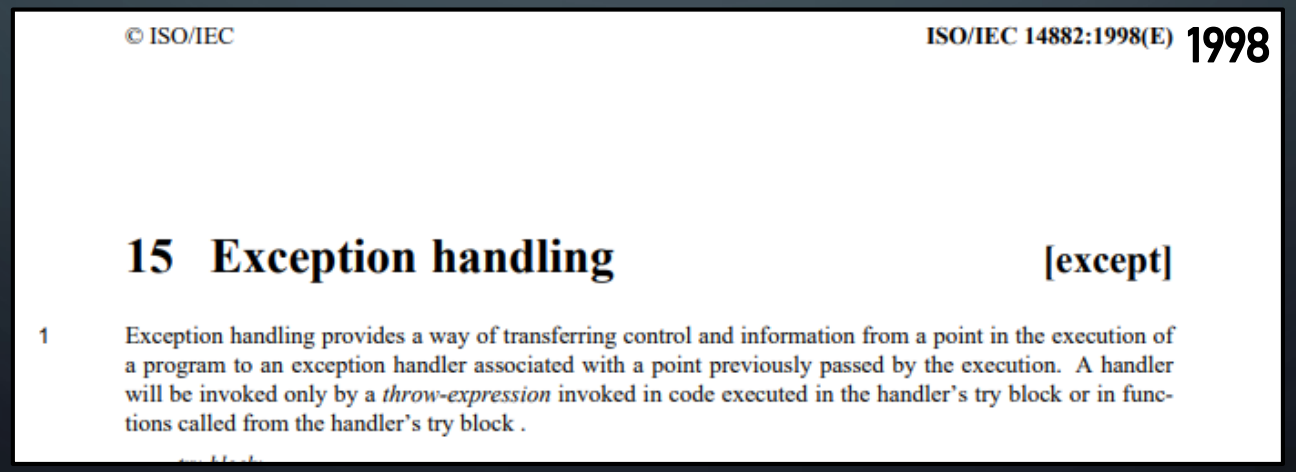
- Some voices have claimed that exceptions may lead to "bad quality" program.


1994

EXCEPTION HANDLING:
A FALSE SENSE OF SECURITY

by Tom Cargill

This article first appeared in *C++ Report*, Volume 6, Number 9, November-December 1994.


1998

© ISO/IEC ISO/IEC 14882:1998(E)

15 Exception handling [except]

1 Exception handling provides a way of transferring control and information from a point in the execution of a program to an exception handler associated with a point previously passed by the execution. A handler will be invoked only by a *throw-expression* invoked in code executed in the handler's try block or in functions called from the handler's try block .

PART I: HISTORY

The following assumptions were made regarding exceptions usage:

1. Exceptions are used primarily for error handling.
2. Exception handlers are rare compared to function definitions.
3. Exceptions occur infrequently compared to function calls.
4. Exceptions are a language-level concept.

Exception Handling for C++ / Andrew Koenig Bjarne Stroustrup 1989

PART I: HISTORY

Original design	Standard / formal ISO	Existing practice
Exceptions are used primarily for error handling.	Control flow management is discouraged.	• Rarely used for control flow management (expensive, un-deterministic mechanism) • C++ does not return to the throwing code (but to the catch block).
Exception handlers are rare compared to function definitions.	The formal usage recommendation says exceptions are costly.	• If thrown, we have reached a point where cost is "the least of our problems".
Exceptions occur infrequently compared to function calls.	standard library does use exceptions extensively.	• It is common to use alternative error handling mechanisms.
Exceptions are a language-level concept.	Exceptions are platform-independent.	• C++ Exceptions follow ABI rules.

PART II: EXCEPTIONS OVERHEAD

PART II: EXCEPTIONS OVERHEAD

- Main difference between C and C++ is that the cleanup process is now part of the language.
- By using “setjmp”, “longjmp” we break the contract between the user and the language. (*)

- Terms:
  - “Happy path” is the default scenario with no exceptions or errors.
  - “Sad path” is the path which includes errors.

- (*) MSVC supports different behavior – e.g. calling DTORs for objects as part of stack unwinding

PART II: EXCEPTIONS OVERHEAD

- A few words about the happy path overhead:
  - Return values:
    - Occupy the return value.
    - Adds a need to “manually propagate” the error.
    - Adds an “if” statement, which translates into instructions.
  - Globals:
    - Unsafe for multiple threads
    - Limited.
  - Exceptions:
    - Size: RTTI information, catch tables, landing pad assembly, etc.
    - Runtime: Load cold code from memory, dynamic allocation.

PART II: EXCEPTIONS OVERHEAD

- A (mini) Benchmark: Exceptions vs. Error code (sad path):

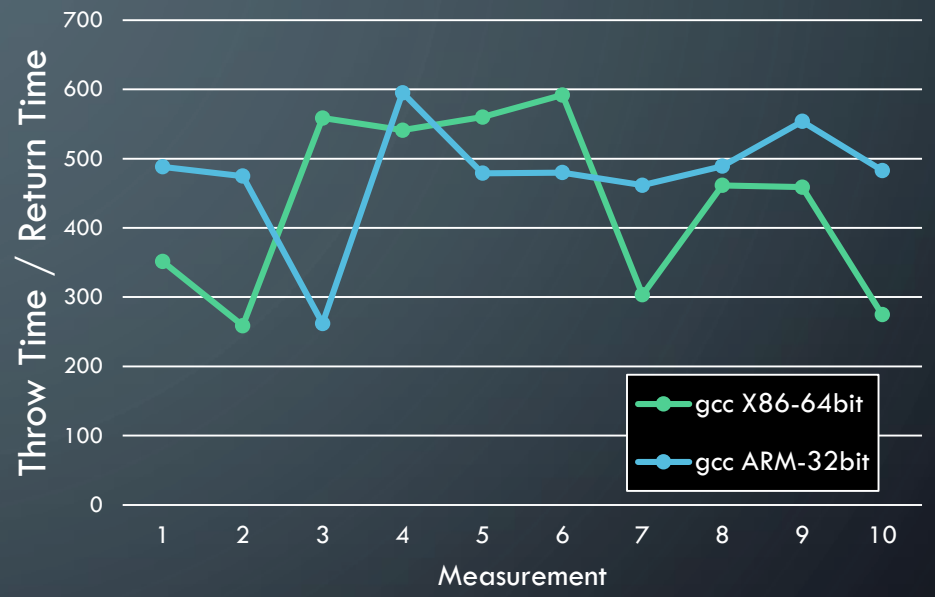
```
int main()
{
    for (int i = 0 ; i < ITERS ; i++) {
        try {
            ThrowingFunc();
        } catch (const length_error &e) {
        } catch (const domain_error &e) {
        } catch (const overflow_error &e) {
        }
    }

    for (int i = 0 ; i < ITERS ; i++) {
        int ret = ErrorCodeFunc();
        if (ret == -1) {
        }
        if (ret == -2) {
        }
        if (ret == -3) {
        }
    }
}
```

PART II: EXCEPTIONS OVERHEAD

- A (mini) Benchmark: Exceptions vs. Error code (sad path):

	Runtime (ms)			Size (B)		
	Throw3	Return	Ratio	Throw3	Return	Ratio
X86	2199	5	435	15536	15316	1.01
ARM	59104	128	477	14152	10356	1.4



- A clear observation that exceptions are slower on both platform.
- Exceptions perform worse for size and speed on ARM (single core).

PART II: EXCEPTIONS OVERHEAD

- Two main implementations for exceptions exist:
  1. Table based: gcc, clang (better happy path results).
  2. Frame based ("code approach"): MSVC (better sad path results).

- Table based implementation of exceptions will add the following functionality to the program:
  1. RTTI will be generated in order to identify the exception type.
  2. Exception handlers will be defined.
  3. Each catching function will contain additional information of list of catchable exceptions, as well as a cleanup table.

PART II: EXCEPTIONS OVERHEAD

- Functionality is implemented in two main libraries (focusing on gcc):
  - libgcc: functionality of stack unwinding, frame management. (Language-independent)
  - libstdc++: functionality of exceptions handling. (itanium-cxx)

```
_Unwind_DeleteException
_Unwind_Find_FDE _Unwind_ForcedUnwind
_Unwind_GetGR _Unwind_GetIP
_Unwind_GetLanguageSpecificData
_Unwind_GetRegionStart
_Unwind_GetTextRelBase
_Unwind_GetDataRelBase
_Unwind_RaiseException _Unwind_Resume
_Unwind_SetGR _Unwind_SetIP
_Unwind_FindEnclosingFunction
_Unwind_SjLj_Register
_Unwind_SjLj_Unregister
_Unwind_SjLj_RaiseException
_Unwind_SjLj_ForcedUnwind
_Unwind_SjLj_Resume
_Unwind_Exception
__deregister_frame
__deregister_frame_info
__deregister_frame_info_bases
__register_frame __register_frame_info
__register_frame_info_bases
__register_frame_info_table
__register_frame_info_table_bases
__register_frame_table
```

```
__cxa_allocate_dependent_exception
__cxa_allocate_exception
__cxa_exception
__cxa_rethrow
__cxa_atexit
__cxa_bad_cast
__cxa_bad_typeid
__cxa_begin_catch
__cxa_call_unexpected
__cxa_current_exception_type
__cxa_end_catch
__cxa_throw
__cxa_get_exception_ptr
__cxa_begin_catch
```

PART II: EXCEPTIONS OVERHEAD

- Stages of exception raising process:
  1. Allocate exception (*): __cxa_allocate_exception
  2. Throw: __cxa_throw
  3. Lookup phase.
  4. Cleanup phase.

- (*) In case of a **failure**, "emergency" Exception will be allocated only if:
  1. The exception object size, including headers, is under 1KB.
  2. The current thread does not already hold four buffers.
  3. There are fewer than 16 other threads holding buffers, or this thread will wait until one of the others releases its buffers before acquiring one.

PART II: EXCEPTIONS OVERHEAD

- Stages of the exception raising process:
  - Lookup phase:

    Trigger stack unwinding by calling: "_Unwind_RaiseException()" with: "_UA_SEARCH_PHASE"
    - In case of failure: call terminate()
    - In case of success: return "_URC_HANDLER_FOUND"

  - Cleanup phase:

    Trigger stack unwinding by calling: "_Unwind_RaiseException()" with:

    "_UA_CLEANUP_PHASE"

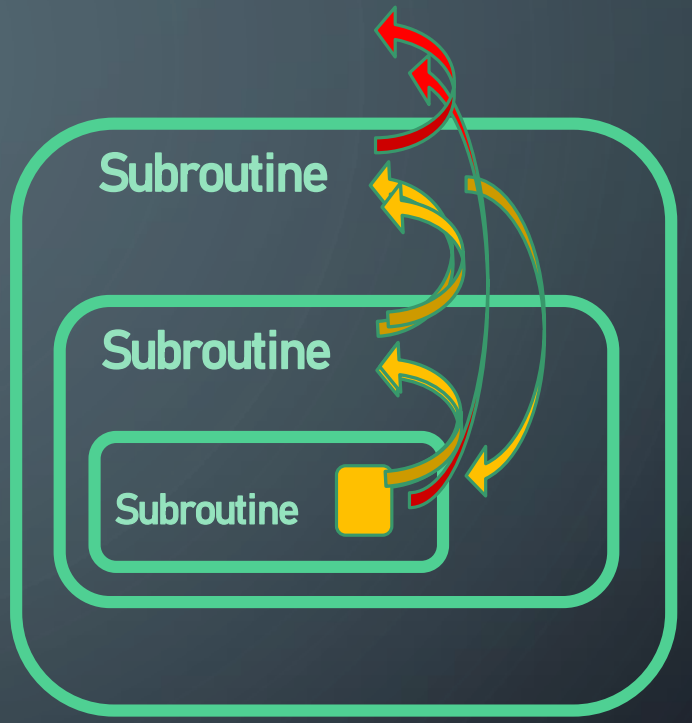
PART II: EXCEPTIONS OVERHEAD

- Stages of the exception cleaning process:
  1. Personality routine cleaning the stack frame takes over: __gxx_personality_v0
  2. _Unwind_Resume
  3. Upon reaching the stack frame with the proper catch statement:

     Catch is called:
     1. __cxa_begin_catch: Treat the exception, modify relevant data.
     2. __cxa_end_catch: Clean up and handle results - termination / re-throw
  4. If a re-throw is needed, _Unwind_Resume is called.

PART II: EXCEPTIONS OVERHEAD

- Complete progress:
  1. Allocate exception ☒ ☑
  2. Throw
  3. Lookup phase ☒ ☑
  4. Cleanup phase
     1. Stack unwind, look for catch
     2. Catch ☑
  5. Resume execution (on catch frame):
     1. If a 'finally' block exists, execute it.
     2. Execution resumes in the catch block.



PART III: DESIGN OVERVIEW

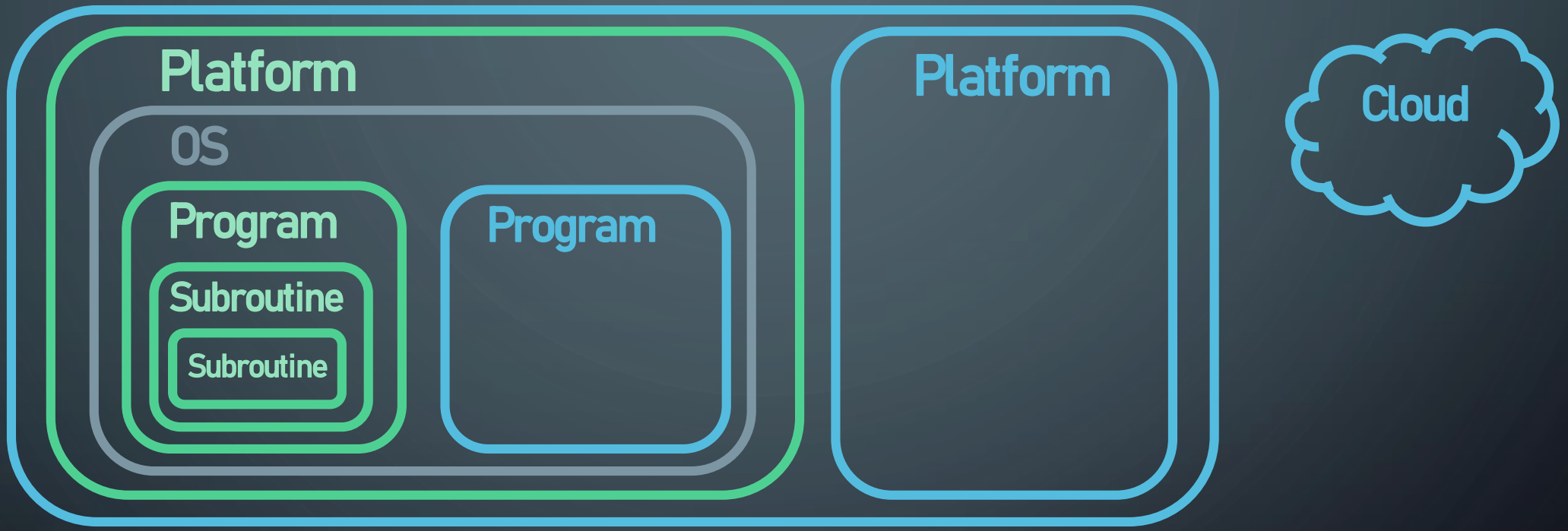
PART III: DESIGN OVERVIEW

- The assumptions of the original design from embedded perspective:

1. Type-safe transmission (...)
2. No added cost (in time or space) to code that does not throw an exception.
3. (...) handlers can be written to catch groups of exceptions as well as individual ones.
4. (...) allows cooperation with other languages, especially with C.
5. Exceptions occur infrequently compared to function calls.
6. Type-safe transmission of arbitrary amounts of information.
7. Exceptions are used primarily for error handling.
8. A mechanism that by default will work correctly in a multi-threaded program.

PART III: DESIGN OVERVIEW

- Errors collection: Let's take a step back, and have a more general overview



"No single unit of a system can recover from every error that might happen in it, and every bit of violence that might be done to it from "the outside." In extreme cases, power will fail or a memory location will change its value, for no apparent reason."

B. Stroustrup, The Design and Evolution

PART III: DESIGN OVERVIEW

There are roughly three (and a half) types of errors:

1. Program bugs (logic_error).
2. Recoverable errors.
3. Terminal errors (overflow_error, etc.):
    1. Errors which invalidate the program.
    2. Errors which exhaust resources.

PART IV: ALTERNATIVES

PART IV: ALTERNATIVES

- Some techniques for exceptions mechanism with (potentially) better performance in your code:
  1. Use current mechanism and library, optimize (for example minimize exception types).
  2. Overwrite function calls for libstdc++ (or equivalent).
  3. Re-implement parts of libstdc++ (or equivalent).
  4. Use alternative mechanisms (e.g. MSVC's Structured Exception Handling compiler extension).

- I've tried (1) and (2)

PART IV: ALTERNATIVES

- Let's look at our previous case:
- Optimized using: a single error type / a single return value

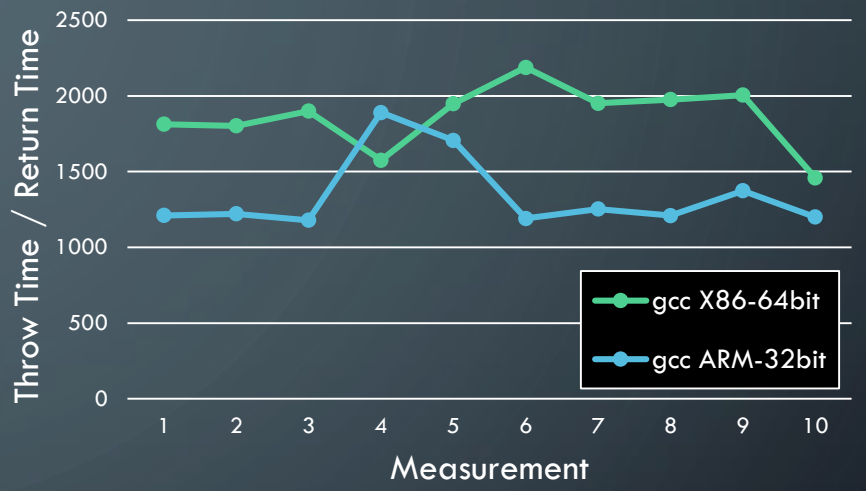
```
int main()
{
    for (int i = 0 ; i < ITERS ; i++) {
        try {
            ThrowingFunc();
        } catch (const logic_error &e) {
        }
    }
    for (int i = 0 ; i < ITERS ; i++) {
        int ret = ErrorCodeFunc();
        if (ret == -1) {
        }
    }
}
```

PART IV: ALTERNATIVES

- Single Type: Exceptions vs. Error code (Sad Path)

	Runtime (ms)			Size (B)		
	Throw1	Return	Ratio	Throw1	Return	Ratio
X86	2362	1.2	1861	14936	14056	1.1
ARM	53747	40	1343	14540	10276	1.4

	Runtime (ms)			Size (B)		
	Throw3	Throw1	Ratio	Throw3	Throw1	Ratio
X86	2199	2262	0.97	15536	14936	1.04
ARM	59104	53747	1.1	14152	14540	0.97



- Ratio when adding one exception is worse – just adding exceptions has significant overhead.
- Some gain on runtime on ARM, results are not consistent.

PART IV: ALTERNATIVES

- The “thinnest” throw:

```
#include "mylib.hpp"
#include <cstdio>

int main()
{
    printf("Calling ThrowingFunction.\n");
    ThrowingFunction();
    printf("After calling ThrowingFunction.\n");

    return 0;
}
```

```
#include "mylib.hpp"
struct MyException() { /* CTOR, DTOR… */ };

void ThrowingFunction() {
      throw MyException(“SomeExc”);
}
```

```
#include <cstdlib>

extern "C" {

void* __cxa_allocate_exception(size_t thrown_size);
void __cxa_throw(void* thrown_exception,
     struct std::type_info *tinfo, void (*dest)(void*));
```

```
Calling ThrowingFunction
__cxa_allocate_exception called
Allocating Exceptions: 64B
MyException: SomeExc CTOR
__cxa_throw called
Exiting...
```

PART IV: ALTERNATIVES

- The “thinnest” throw:

```
#include "mylib.hpp"
#include <cstdio>

int main()
{
    printf("Calling ThrowingFunction.\n");
    ThrowingFunction();
    printf("After calling ThrowingFunction.\n");

    return 0;
}
```

```
#include <cstdlib>

extern "C" {

void* __cxa_allocate_exception(size_t thrown_size) {

    if (fail)
        std::terminate();
    return &buff;
}
void __cxa_throw(void* thrown_exception,
    struct std::type_info *tinfo, void (*dest)(void*){

    exit(0);
}
```

```
Calling ThrowingFunction
__cxa_allocate_exception called
Allocating Exceptions: 64B                  // Returning static &buf to the caller
MyException: SomeExc CTOR                   // Exception constructed
__cxa_throw called
Exiting...                                  // just exits!
```

PART IV: ALTERNATIVES

- The “thinnest” throw:

	Runtime (ms)			Size (B)		
	Throw1	thinnest	Ratio	Throw1	thinnest	Ratio
X86	2362	4	590	14936	9440	1.58

	Runtime (ms)			Size (B)		
	Throw3	Return	Ratio	Throw3	Return	Ratio
X86	2199	5	435	15536	15316	1.01

- We got performance gain which is very similar to the one with Return.

PART IV: ALTERNATIVES

- Customizable “throw”:

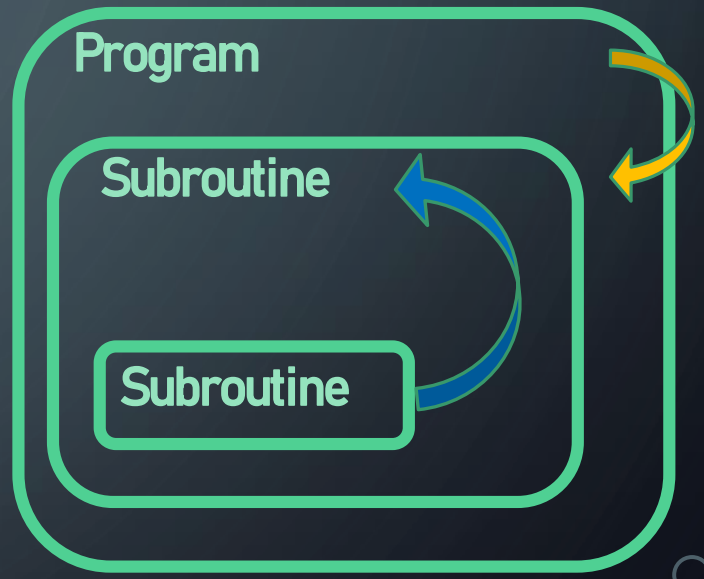
```
void foo() {
    // do stuff that can fail

    if (fail) {
        throw (unwind = false, callback = doSomething);
    }
}
```

```
using throw = throw(unwind = false, callback = doSomething);

void foo() {
    // do stuff that can fail

    if (fail) {
        throw;
    }
}
```



PART IV: ALTERNATIVES

Herb Sutter's P0709R4: Zero-overhead deterministic exceptions: Throwing values

From the Abstract (partial quote):

> (4.1) Today's dynamic exception types violate the zero overhead principle (…)
>
> (4.2) Program bugs shouldn't be runtime exceptions / error codes.
>
> > (…) Moving std away from throwing exceptions and toward preconditioning.
>
> (4.3) Allocation failure isn't like other recoverable run-time errors (…)

Define error as a recoverable error only ("fix usage"):

- Replace more "errors" with "status codes".
- Replace more "errors" with std::expected.
- "Deprecate" logic_error (replace it with precondition)

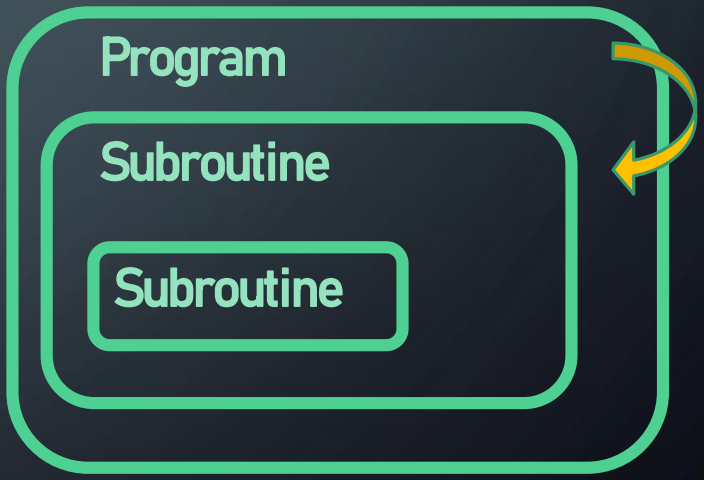
PART IV: ALTERNATIVES

Non-breaking Optimizations:

- Statically typed error.
- Return an equivalent of: union { Success; Error; } + bool
- Error code is returned on the stack.

Future directions:

- std::allocator can be pre-defined to report an error or perform fast exit.
- Options:
  1. Pre define default std::allocator, operator new to fast exit.
  2. Let the programmer decide (per program?).

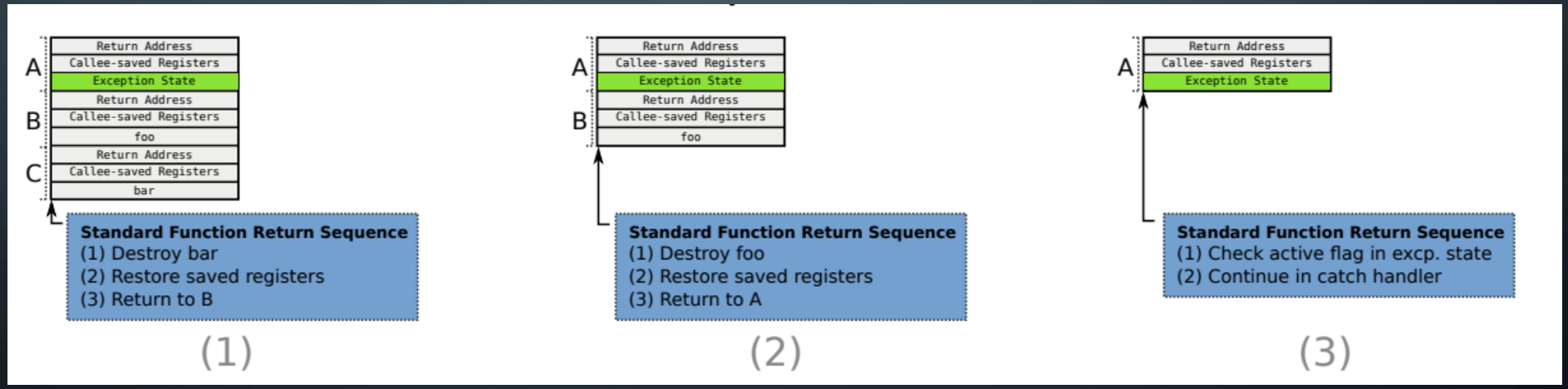


PART IV: ALTERNATIVES

James Renwick, Tom Spink, Bjorn Franke: Low-cost Deterministic C++ Exceptions for Embedded Systems

The proposal focuses on minimizing the cost:

1. Reducing data needed for stack unwinding.
2. Replacing allocated exception with exception state, kept on the stack.
3. Using Global in case of a re-throw to pass data between frames.



PART IV: ALTERNATIVES

You can create your own alternatives:

1. Avoid stack unwinding process, replace with alternatives. (can be defined with a flag)
    I. Signal to a different process.
    II. Report to cloud.
    III. Write to file.
2. Use pre-allocated space for exceptions.
3. Save minimal data needed.
4. Uniform exception (single type) or use static type: minimizing catch data + RTTI.

PART IV: ALTERNATIVES

Other suggestions for error handling are on the table as well:

1. Contracts, and pre-conditions.
2. Vicente Botet and JF Bastien's P0323R9: std::expected.
3. Ben Craig is collecting data for a (future) paper: Shared understanding of errors and exceptions.

Other holistic approaches:

1. Rostislav Khlebnikov and John Lakos' P2053: Defensive Checking Versus Input Validation.
2. Staffan Tjernstrom and Derek Haines are considering adding flexibility to "who determines the error".
3. And others...

PART V: WHAT'S NEXT?

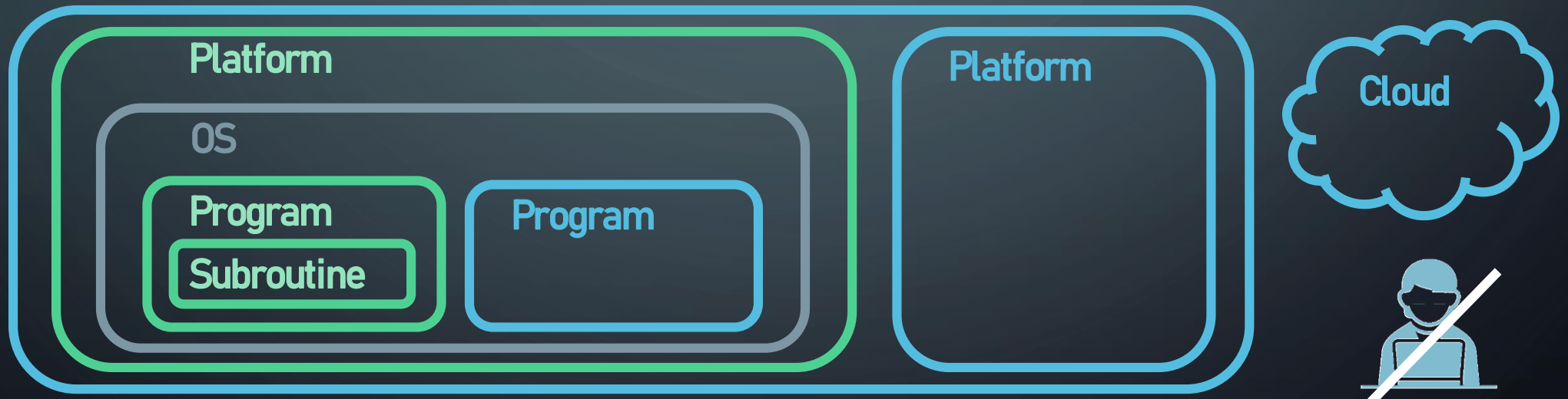
PART V: WHAT'S NEXT?

- We need to separate the semantics from the implementation.
- We need to allow behavior suitable for performance sensitive applications, as well as hosted environments.
- The overhead already exists, so adding a customizability to the framework (in the current state of things) can only increase the number of users.

PART V: WHAT'S NEXT?

- Last year, the keynotes by Bjarne Stroustrup was about C++ turning 30.
- A lot has happened in computing in the last 30 years!
- People are using C++ for:
  - Tight control over hardware resources.
  - Performance sensitive applications.

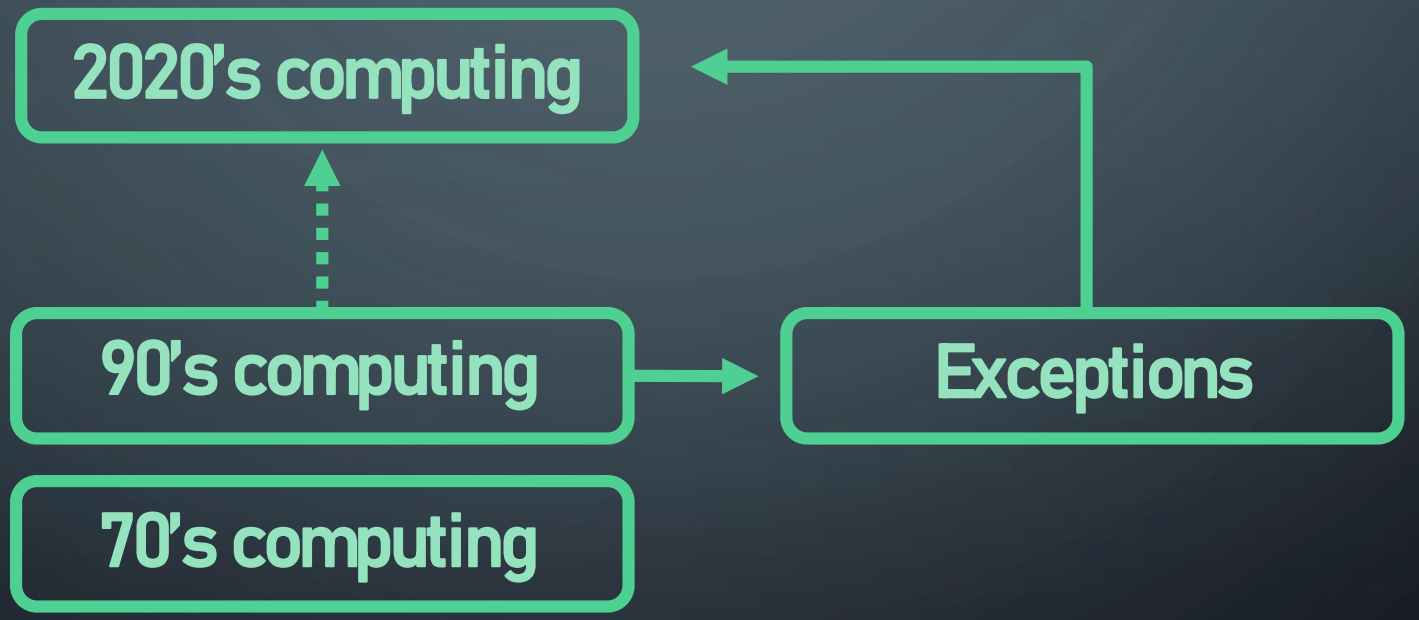
} Over 50% of Embedded / RT



- **Let's consider the big picture!**

PART V: WHAT'S NEXT?

- I invite you all to share with me your "ideal exception usage" (Email me!)
- I invite you all to experiment with the exceptions mechanism, and suggest alternatives. (*)
- Let's Rebase!



(*) An ABI break will effect the possible implementations.

LINKS

Historical references:

- Exception Handling for C++ / Andrew Koenig Bjarne Stroustrup: https://www.stroustrup.com/except89.pdf
- Exception handling: A false sense of security / Tom Cargill: https://ptgmedia.pearsoncmg.com/images/020163371x/supplements/Exception_Handling_Article.html

Technical references:

- Technical Report on C++ Performance (2006) / Various authors: http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf
- Exception handling on Itanium ABI / Various authors: http://refspecs.linux-foundation.org/abi-eh-1.21.html
- MSVC's SEH https://docs.microsoft.com/en-us/cpp/cpp/structured-exception-handling-c-cpp?view=vs-2019

  CppCon2018: James McNellis "Unwinding the Stack: Exploring How C++ Exceptions Work on Windows: https://www.youtube.com/watch?v=COEv2kq_H-8
- CppCon 2017: Dave Watson C++ Exceptions and Stack Unwinding: https://www.youtube.com/watch?v=_Ivd3qzgT7U
- P1640R0: Error size benchmarking / Ben Craig: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1640r0.html
- P1886R0: Error speed benchmarking / Ben Craig: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1886r0.html
- LCDA / Renwick, J, Spink, T, Francke, B: https://www.research.ed.ac.uk/portal/files/78829292/low_cost_deterministic_C_exceptions_for_embedded_systems.pdf
- ARM ABI: https://developer.arm.com/documentation/ihi0038/b

On-going proposals:

- P0323R9: std::expected: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0323r9.html
- P2053: Defensive Checking Versus Input Validation / Rostislav Khlebnikov, John Lakos: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2053r1.pdf
- P2018: Shared understanding of errors and exceptions / Ben Craig (Future paper, get updated)

Surveys:

- isocpp.org: Cpp Dev Survey / Herb Sutter, Various authors: https://isocpp.org/files/papers/CppDevSurvey-2020-04-summary.pdf
- CoreCpp: C++ Usage In Work Environment: Survey results / Inbal Levi: https://www.youtube.com/watch?v=cZF7JWLVfzc&feature=youtu.be

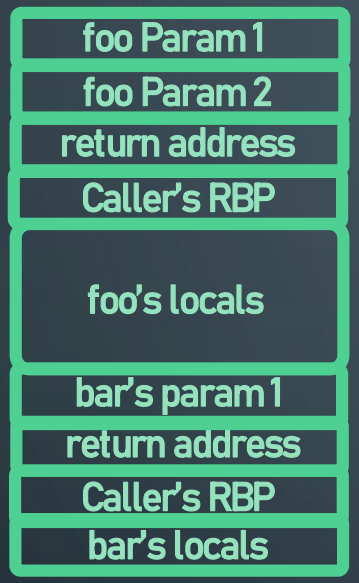
THANK YOU!

I hope you're now inspired to go and explore your system's error handling. ☺

Inbal Levi

Mail: sinbal2l@gmail.com
Twitter: @Inbal_l
Linkdin: /inballevi
Github: /inbal2l

APPENDIX A: MSVC'S IMPLEMENTATION




```
int bar(int i) {
    return i;
}

int foo(int i, int j) {
    return bar(i);
}

int main()
{
    foo(1, 1);
}
```


```
int bar(int i) {
    // throw()
    return i;
}

int foo(int i, int j) {
    // throw()
    return bar(i);
}

int main()
{
    foo(1, 1);
}
```

RBP – Frame Pointer
RSP – Stack Pointer (top of stack)