Prague Trip Report

PRAGUE
CONGRESS CENTRE

The following were voted into the standard

- **Modules** are now in the language.
- **Concepts** are now in the language (**P1851R0**: Guidelines For snake_case Concept Naming was suggested)
- **Contracts were removed** from C++20 and will be discussed for C++23.

- **P1999R0**: Process proposal: double-check evolutionary material via a Tentatively Ready status - was accepted.
- A decision was made that from now on, with every paper proposed, there will be a description of the **UB added in it**. (documenting core undefined or unspecified behavior)

P0592R4: To boldly suggest an overall plan for C++23

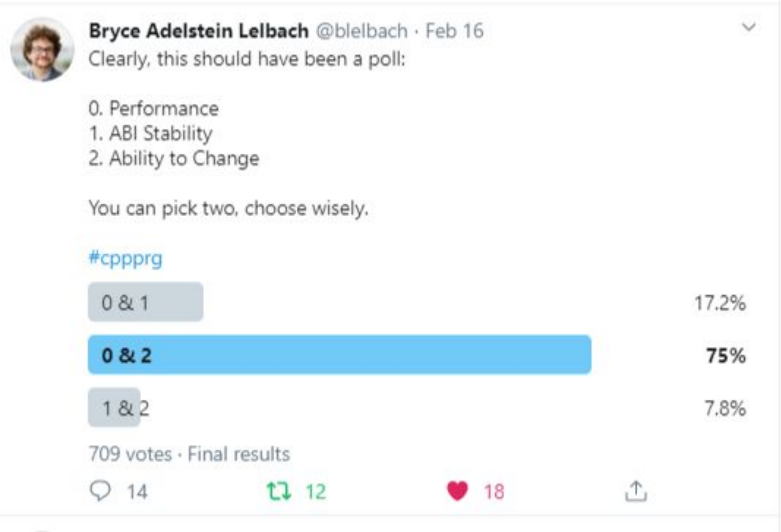
For C++23 Insert the following things to the standard:

- Library support for **coroutines**
- A **modular standard library**
- **Executors**
- **Networking**

Without a particular version, we should also make progress on:

- **Reflection**
- **Pattern matching**
- Contracts

P1863R1: ABI - Now or Never
P2028R0: What is ABI, and What Should WG21 Do About It?



Bryce Adelstein Lelbach @blelbach · Feb 16

Clearly, this should have been a poll:

0. Performance
1. ABI Stability
2. Ability to Change

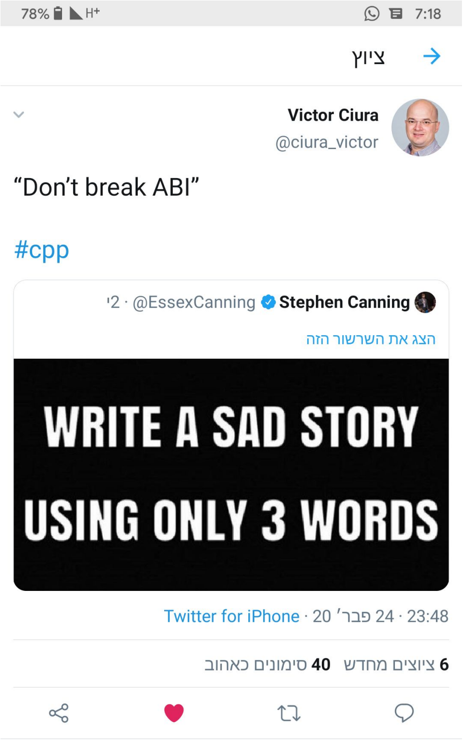
You can pick two, choose wisely.

#cppprg

0 & 1	17.2%
0 & 2	**75%**
1 & 2	7.8%

709 votes · Final results

14 12 18



78% H+ 7:18

ציוץ

Victor Ciura
@ciura_victor

“Don’t break ABI”

#cpp

Stephen Canning @EssexCanning · 2 י'

הצג את השרשור הזה

WRITE A SAD STORY
USING ONLY 3 WORDS

23:48 · 24 בפבר׳ 20 · Twitter for iPhone

6 ציוצים מחדש **40** סימונים כאהוב

P1863R1: ABI - Now or Never
P2028R0: What is ABI, and What Should WG21 Do About It?

- The three options presented at Titus's paper:
  a. Break ABI on C++23
  b. Prioritise not breaking ABI
  c. Continue as is - consider ABI for each paper separately
- Polls:
  a. The discussion of the claim that we should consider ABI break did not reach a consensus.
  b. The need to consider ABI break for every release was agreed on.
  c. The strong notion was not to promise backward compatible ABI for all versions in the future.
- **P1881R1:** Epochs: a backward-compatible language evolution mechanism - The Epochs feature will currently be held back, will be discussed in the future.

The following were voted to a positive directions, for C++23 or beyond

(LEWG)

- **P0443R12:** A Unified Executors Proposal for C++
- **P2003R0:** Fixing Internal and External Linkage Entities in Header Units
- **P1678R2**: Callbacks and Composition

(EWG)

- **P1847R2**: Make declaration order layout mandated
- **P1468R3**: Fixed-layout floating-point type aliases
- **P1371R2:** Pattern matching - was reviewed and changes where discussed.
- **P1040R5:** std::embed and #depend
- **P1967R1**: #embed - a simple, scannable preprocessor-based resource acquisition method

The following were voted to a positive directions, for C++23 or beyond

(SG14)

- **P2013R0:** Freestanding Language: Optional::operator new
- **P0709R1**: Zero-overhead deterministic exceptions: Throwing values took the next step, with presenting: low_cost_deterministic_C_exceptions_for_embedded_systems

(SG21)

- **P1774R2**: Portable Assumptions
- **P2064R0**: Assumptions

Vznik Pražské křižovatky podpořili
Prague Crossroads was sponsored by
Havlová
Havel
Národní Divadlo / National Theatre
DaimlerChrysler Automotive Bohemia
Český Telecom / Czech Telecom
sterstvo Kultury ČR / Ministry of Culture CR
sche Bank
el Praha
bel Coatings CZ
onuments Fund
ovna / Czech Mint
edom Foundation
Republic
oup Czech Republic
Credit Bank Czech Republic and Slovakia
Unipetrol RPA
Děkujeme / Thank you
Vize 97
Taste of Israel

Taste of Israel
Core C++
2019

TASTE
HOLY LAND
YAMAHA

P0443R12: A Unified Executors Proposal for C++

```
static_thread_pool pool(4);
auto exec = pool.executor();

auto future = std::async(exec , [] {
    std::cout << "Hello world, from a new execution agent!" << std::endl;
});

std::for_each(std::execution::par.on(exec), data.begin(), data.end(), func);
```

P2013R0: Freestanding Language: Optional ::operator new \ Ben Craig

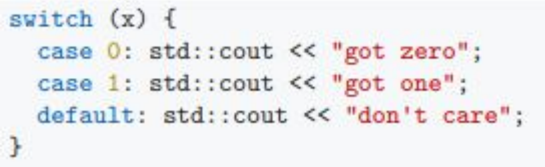
- Motivation: On freestanding systems there **is no right default way to have heap allocations**, therefore, we should define using default as **ill formed**.

- Suggestion: Without **default heap storage**, the presence of the replaceable allocation functions (i.e. allocating ::operator new, including the nothrow_t and align_val_t overloads, single and array forms) will be implementation defined.

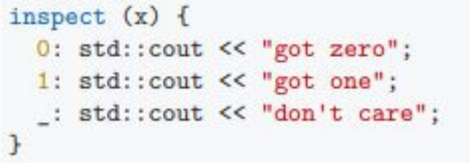
P1371R2: Pattern Matching /Sergei Murzin, Michael Park, David Sankel, Dan Sarginson

- Combination of papers: P1260R0, P1308R0
  - Inspect rather than switch
  - First Match rather than Best Match
  - Statement rather than Expression
  - Language rather than Library

- Implements:
  a. Matching Integrals:

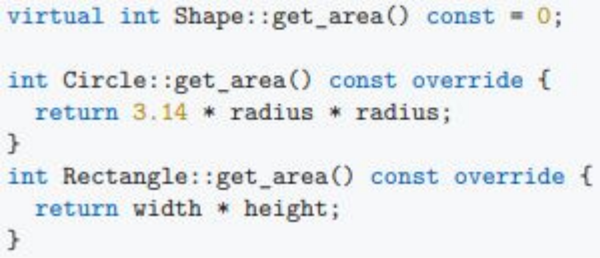


```
switch (x) {
  case 0: std::cout << "got zero";
  case 1: std::cout << "got one";
  default: std::cout << "don't care";
}
```



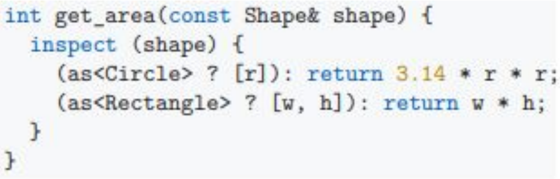
```
inspect (x) {
  0: std::cout << "got zero";
  1: std::cout << "got one";
  _: std::cout << "don't care";
}
```

  b. Matching Polymorphic Types:



```
virtual int Shape::get_area() const = 0;

int Circle::get_area() const override {
  return 3.14 * radius * radius;
}
int Rectangle::get_area() const override {
  return width * height;
}
```



```
int get_area(const Shape& shape) {
  inspect (shape) {
    (as<Circle> ? [r]): return 3.14 * r * r;
    (as<Rectangle> ? [w, h]): return w * h;
  }
}
```

P1371R2: Pattern Matching /Sergei Murzin, Michael Park, David Sankel, Dan Sarginson

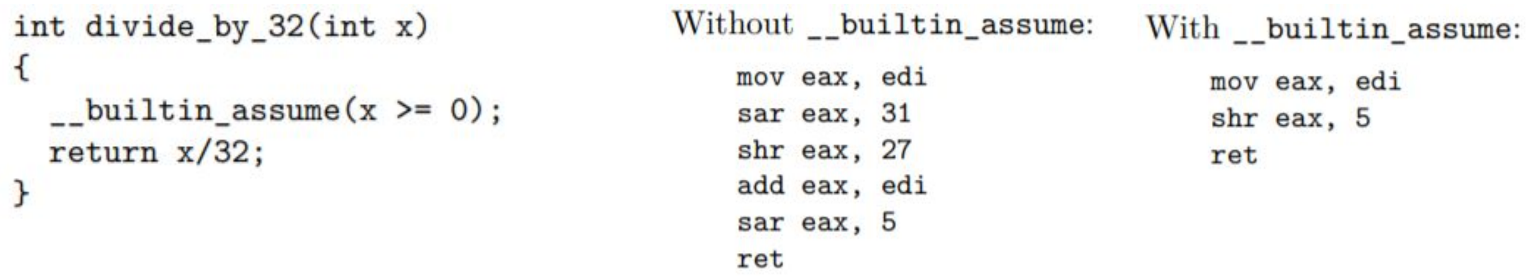
- Types of Patterns:
  - Primary Patterns:
    - Wildcard Pattern _: …
    - Identifier Pattern: x: …
    - Constant Pattern: 0: … (int zero = 0; … zero: …)
  - Compound Patterns:
    - Structured Binding Pattern: [0, y]: std::cout << "on y-axis";
    - Alternative Pattern: <C1> c1: strm << "got C1: " << c1; (C1 can be concept, type, constant, auto)
    - Binding Pattern: <point> p at [x, y]: strm << "got point" << p;
    - Extractor Pattern: (email ? [address, domain]): std::cout << "got an email";
    - As Pattern
- Pattern Guard: [x, y] if test(x, y): std::cout << x << ',' << y << " passed";
- inspect constexpr
- Exhaustiveness Checking
- Patterns in range-based for loop

P1774R2: Portable assumptions \ Timur Doumler

- Motivation:
  a. All major compilers offer built-ins that give the programmer a way to allow the compiler to assume that a given C++ expression is true (**on run-time)**, and to optimise based on this assumption.
  b. Assert is for **debug mode**, Assume is for **release mode** (and doesn't evaluate expression - no side effects)

- Options exists on compilers:
  VS: __assume(expression); Clang: __builtin_assume(expression); GCC: __builtin_unreachable();

- Improved assembly by using assumptions:



```
int divide_by_32(int x)
{
  __builtin_assume(x >= 0);
  return x/32;
}
```

Without __builtin_assume:

```
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

With __builtin_assume:

```
mov eax, edi
shr eax, 5
ret
```

- Proposed syntex: compiler attribute: **[[assume(expression)]]** (with alternatives - Macro, language extensions) (already in std: std::assume_aligned)

P2064R0: Assumptions \ Herb Sutter

- Defined the difference between: assert(expr) and assume(expr):
- Assert ⇏ Assume:
    a. Assert exists to be **checked for false**, whereas Assume must be guaranteed to **never be false.**
    b. Assert **evaluates** its expression, whereas Assume **never evaluates** it.
    c. Assert is a **safe debugging** aid provides **informed error messages**, and assume is **for release,** and if failed, **injects a run-time diagnostic** into the caller's local call site location.
    d. Assert should be used pervasively by **all programmers**, whereas Assume is a **dangerous power tool for experts** only, only in **function bodies**, and is in practice used ~1000 less frequently than Assert.
- Assert ⊥̸ Assume:
    a. Assuming on function declaration ([[pre assume: …]]) doesn't make sense, since it's not up to the writer.
- Assume ⇒ Assert:
    a. Assumes are used on function bodies only, since it's a call dependent.
    b. Use assert on **debug**, assume on **release** to cover same conditions:

```
#ifdef NDEBUG
        #define __unsafe_assume(b)   __compiler_magic(b)
#else
        #define __unsafe_assume(b)   assert(b) #endif
```

P2064R0: Assumptions \ Herb Sutter

- Assume should not be expressed as an **Attribute** (Referring to: P1774R2):

    a. Awkward to write in the one place they should appear, which is as a statement

    b. Allow be written outside of **function bodies** (on declarations), where **not meaningful** and actively harmful.

    c. Harder to express that it Asserts its parameter as a precondition for test time diagnostics if contracts (Asserts) are eventually also added as attributes, because we **can't write an attribute on an attribute**.

    d. (In contrast, unsafe_assume(bool b) [[pre: b]] is easy to write naturally, exactly documente the relationship)

    e. Novel invention not supported by any existing practice in the past >20 years of commercial compilers.

P2064R0: Assumptions \ Herb Sutter

Hierarchy of assumptions: **As-if < UB < Assume(false) < Assume(expr) ≤ Miscompile**

We can enable optimizations primarily via the as-if rule, which cannot change the observable behavior of a program. The following are possible emulation:

- **Undefined Behaviour:** *(volatile int*)0 = 0xDEAD

- **Assume(expr)**: __assume(0) , __builtin_unreachable()
  Emulations:
  ```
      #define __hand_rolled_assume(expr) if(expr){}else{ *(volatile int*)0 = 0xDEAD; }
  #define __hand_rolled_assume(expr) if(expr){}else{ const int i = 0; (int&)i=0xDEAD; }
  #define __hand_rolled_assume(expr) if(expr){}else{ __builtin_unreachable(); }
  ```

- **Assume(false):** __assume(expr) , __builtin_assume(expr)
  Emulations:
  ```
      #define __hand_rolled_assume_false() (*((volatile int*)0)=0xDEAD)
  ```

P2064R0: Assumptions \ Herb Sutter

Discussion:

1. Why not implement Assume(expr) in terms of Assume(false):
    - **__assum(0) is __builtin_unreachable() (and not assume data)** (claimed by Hal Finkel & Eric Burmer's)

```
int test(bool cnd1, bool cnd2)
{
          int x;
           if (cnd1)
                    x = 5;
          else if (cnd2)
                    x = 6;
          else
                    __assume(0)
          return x;                    // warning C4701: 'x' potentially uninitialized
}
```

2. Why not implement either Assume in terms of UB:
    - It's hard to know whether the programmer intended the UB to imply unreachability or fact injection, should have specific syntax

Related paper by Hal Finkel, Generalized Dynamic Assumptions, 2015: N4425

P2064R0: Assumptions \ Herb Sutter

- Surveying real-world compilers: Cases and insights:
    1. Sample survey: Actual branch elision on major compilers and -O levels
        - A list of compiler behaviour test cases
    2. Existing products' usability limitations on using facts via time travel: Violations of sequential consistency and causality in current practice
        - Consider the following example:

```
auto test(int x)
{
        int local = 0;
        local += x;
        f(local);                // f's argument is 'local'
        int local2 = local;      // return value is 'local'
        ASSUME(x==0);
        return local2;
}
```

        - return 0
        - f is called with x value

Related paper by Hal Finkel, Generalized Dynamic Assumptions, 2015: N4425