Pre-Prague Highlight Papers

Inbal Levi

P1641R2: Freestanding Library: Rewording the Status Quo / Ben Craig

- Adding macro for separating freestanding implementation:

  #define __cpp_lib_freestanding 202001L *// freestanding only*

- Add: **//freestanding only** comment to the following:

- **__cpp_lib_atomic_flag_test**
- **__cpp_lib_atomic_float**
- **__cpp_lib_atomic_is_always_lock_free**
- **__cpp_lib_atomic_ref**
- **__cpp_lib_atomic_value_initialization**
- **__cpp_lib_atomic_wait**
- **__cpp_lib_bit_cast**
- **__cpp_lib_bitops**
- **__cpp_lib_bool_constant**
- **__cpp_lib_bounded_array_traits**
- **__cpp_lib_byte**
- **__cpp_lib_char8_t**
- **__cpp_lib_concepts**
- **__cpp_lib_destroying_delete**
- **__cpp_lib_endian**
- **__cpp_lib_hardware_interference_size**
- **__cpp_lib_has_unique_object_representations**
- **__cpp_lib_int_pow2**
- **__cpp_lib_integral_constant_callable**
- **__cpp_lib_is_aggregate**
- **__cpp_lib_is_constant_evaluated**
- **__cpp_lib_is_final**
- **__cpp_lib_is_invocable**
- **__cpp_lib_is_layout_compatible**
- **__cpp_lib_is_null_pointer**
- **__cpp_lib_is_pointer_interconvertible**
- **__cpp_lib_is_swappable**
- **__cpp_lib_launder**
- **__cpp_lib_logical_traits**
- **__cpp_lib_nothrow_convertible**
- **__cpp_lib_remove_cvref**
- **__cpp_lib_result_of_sfinae**
- **__cpp_lib_source_location**
- **__cpp_lib_three_way_comparison**
- **__cpp_lib_transformation_trait_aliases**
- **__cpp_lib_type_identity**
- **__cpp_lib_type_trait_variable_templates**
- **__cpp_lib_uncaught_exceptions**
- **__cpp_lib_void_t**

P1642R2: Freestanding Library: Easy **[utilities]**, **[ranges]**, and **[iterators]** / Ben Craig

- Adding feature test macro: **__cpp_lib_freestanding_iterator**
- Add to freestanding: **<utility>, <tuple>, <ratio>**
- Add the following parts from **<memory>**: pointer_traits, to_address, align, assume_aligned, allocator_arg_t, allocator_arg, uses_allocator, uses_allocator_v, uses_allocator_construction_args, allocator_traits, [specialized.algorithms] (This includes the algorithms in the ranges namespace), default_delete, unique_ptr, unique_ptr overload of swap, relational operators (including three-way / spaceship) involving unique_ptr, hash, unique_ptr specialization of hash, atomic
- Add **<functional> Except**: bad_function_call, function, function overloads of swap, function overloads of operator==, boyer_moore_searcher, boyer_moore_horspool_searcher
- Add **<iterator> Except**: istream_iterator and associated comparison operators, ostream_iterator, istreambuf_iterator and associated comparison operators, ostreambuf_iterator
- Add **<ranges> except**: basic_istream_view, istream_view
- From **<version>**, make freestanding:
  - __cpp_lib_addressof_constexpr
  - __cpp_lib_allocator_traits_is_always_equal
  - __cpp_lib_apply
  - __cpp_lib_as_const
  - __cpp_lib_assume_aligned
  - __cpp_lib_atomic_value_initialization
  - __cpp_lib_bind_front
  - __cpp_lib_constexpr_functional
  - __cpp_lib_constexpr_iterator
  - __cpp_lib_constexpr_memory
  - __cpp_lib_constexpr_tuple
  - __cpp_lib_constexpr_utility
  - __cpp_lib_exchange_function
  - __cpp_lib_integer_sequence
  - __cpp_lib_invoke
  - __cpp_lib_make_from_tuple
  - __cpp_lib_make_reverse_iterator
  - __cpp_lib_nonmember_container_access
  - __cpp_lib_not_fn
  - __cpp_lib_null_iterators
  - __cpp_lib_ssize
  - __cpp_lib_to_address
  - __cpp_lib_transparent_operators
  - __cpp_lib_tuple_element_t
  - __cpp_lib_tuples_by_type
  - __cpp_lib_unwrap_ref

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

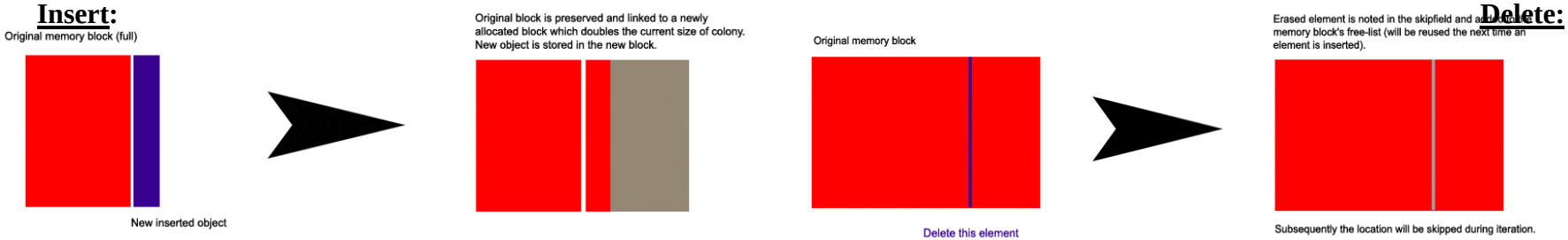
- Motivation: On freestanding systems, including linux kernel, there **is no right default way to have heap allocations**, therefore, we should define using default as **ill formed**.

- Suggestion: on freestanding systems 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.

- Note: The C++20 freestanding library does not include allocators. [P1642R1] proposes adding allocator machinery to freestanding, but doesn't add std::allocator itself.

P0447R10: Introduction of std::colony to the standard library \ Matt Bentley

- Smart large-memory-class vector-like type:
  a. Insert to full: by linking to a new block, which keeps locality and therefore cache (unlike std::list) and evoid large-scale copy
  b. Delete: by marking as free, managed using free-list



- **Performance:**
  - Insert (single): O(1) amortised
  - Insert (multiple): O(N) amortised
  - Erase (single): O(1) amortised
  - Erase (multiple):
    i. non-trivially-destructible types: O(N) amortised
    ii. trivially-destructible types: O(1) - O(N) amortised ~O(logN) average
  - std::find: O(N)
  - splice: O(1)
  - Iterator operators ++ and --: O(1) amortised
  - begin()/end(): O(1)
  - advance/next/prev: O(1) - O(N) ~O(log N) average

P0447R10: Introduction of std::colony to the standard library \ Matt Bentley

- Implementation highlights:
  - Meets: Container, AllocatorAwareContainer, ReversibleContainer
  - Iteration class
  - Pointer stability
  - Thread safe guarantees :

colony	Insertion	Erasure	Iteration	Read
Insertion	No	No	No	Yes
Erasure	No	No	No	Mostly*
Iteration	No	No	Yes	Yes
Read	Yes	Mostly*	Yes	Yes

P0447R10: Introduction of std::colony to the standard library \ Matt Bentley

- Insert:

single element	iterator **insert** (value_type &val)
fill	iterator **insert** (size_type n, value_type &val)
range	template <class InputIterator> iterator **insert** (InputIterator first, InputIterator last)
move	iterator **insert** (value_type&& val)
initializer list	iterator **insert** (std::initializer_list<value_type> il)

- Erase

single element	iterator **erase**(const_iterator it)
range	void **erase**(const_iterator first, const_iterator last)

- Member functions:

bool empty()
size_type size()
size_type max_size()
size_type capacity()
void clear()
void **change_group_sizes**(Skipfield_type min_group_size, Skipfield_type max_group_size)
void change_minimum_group_size(Skipfield_type min_group_size)
void change_maximum_group_size(Skipfield_type min_group_size)
void reinitialize(Skipfield_type min_group_size, Skipfield_type max_group_size)
void **swap**(colony &source)
void **sort**();
void **splice**(colony &source)

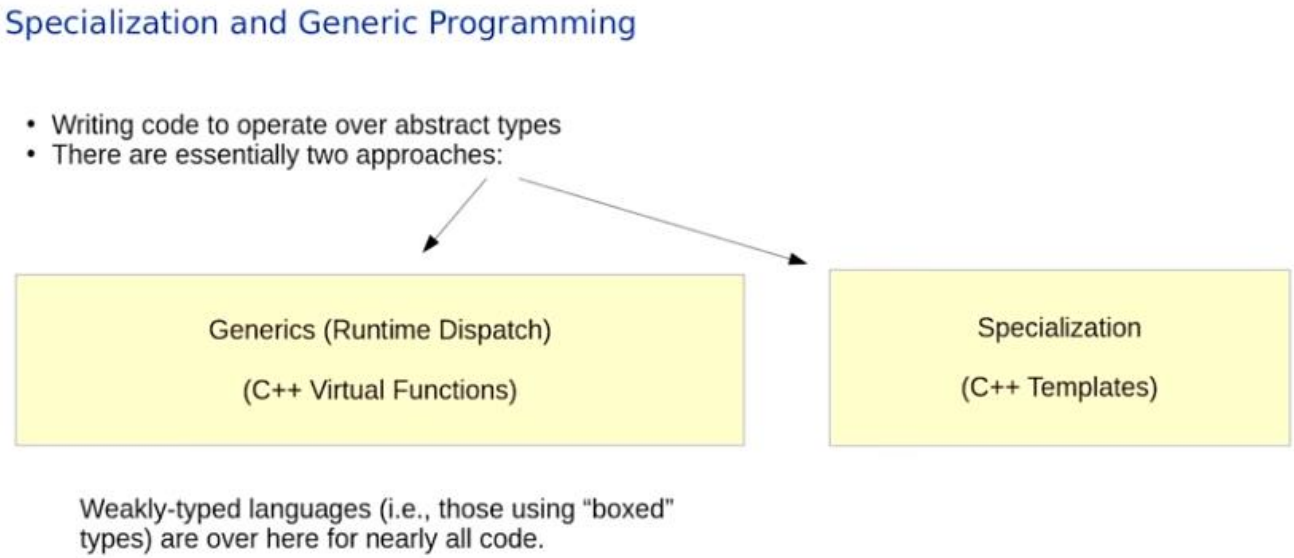
- Interface functions:

iterator begin(), iterator end(), const_iterator cbegin(), const_iterator cend()
reverse_iterator rbegin(), reverse_iterator rend(), const_reverse_iterator crbegin(), const_reverse_iterator crend()
iterator get_iterator_from_pointer(element_pointer_type the_pointer)
size_type get_index_from_iterator(iterator/const_iterator &the_iterator) (slow)
size_type get_index_from_reverse_iterator(reverse_iterator/const_reverse_iterator &the_iterator) (slow)
iterator get_iterator_from_index(size_type index) (slow)
allocator_type **get_allocator**()

- Free functions:

void **swap**(colony &A, source &B)
template <iterator_type> void advance(iterator_type iterator, distance_type distance)
template <iterator_type> iterator_type next(iterator_type &iterator, distance_type distance)
template <iterator_type> iterator_type prev(iterator_type &iterator, distance_type distance)
template <iterator_type> difference_type distance(iterator_type &first, iterator_type &last)

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel



P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

JIT Downsides:

- Compiling the specialization
- Memory overhead
- Runtime dispatch (optional)
- No Pre-runtime optimizations

JIT Upsides:

- Shorter compilation time
- Compile only what you use

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

The solution: [[clang::jit]]:

- Marked templated functions will be instantiated JIT (triggered by call / address taking)
- Doesn’t access file system during the program (for portability & performance)
- Doesn’t access external code during program runtime (doesn’t run compiler)
- Use only information stored in the binary file
- Compile original version AoT, and additional types, if needed, on runtime
- Type information for the JIT types is unavailable (decltype(auto)

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

```
template <int x>
[[clang::jit]] void run() {
  std::cout << "Hello, World, I was compiled at runtime, x = " << x << "\n";
}

int main(int argc, char *argv[]) {
  int a = std::atoi(argv[1]);
  run<a>();
}
```

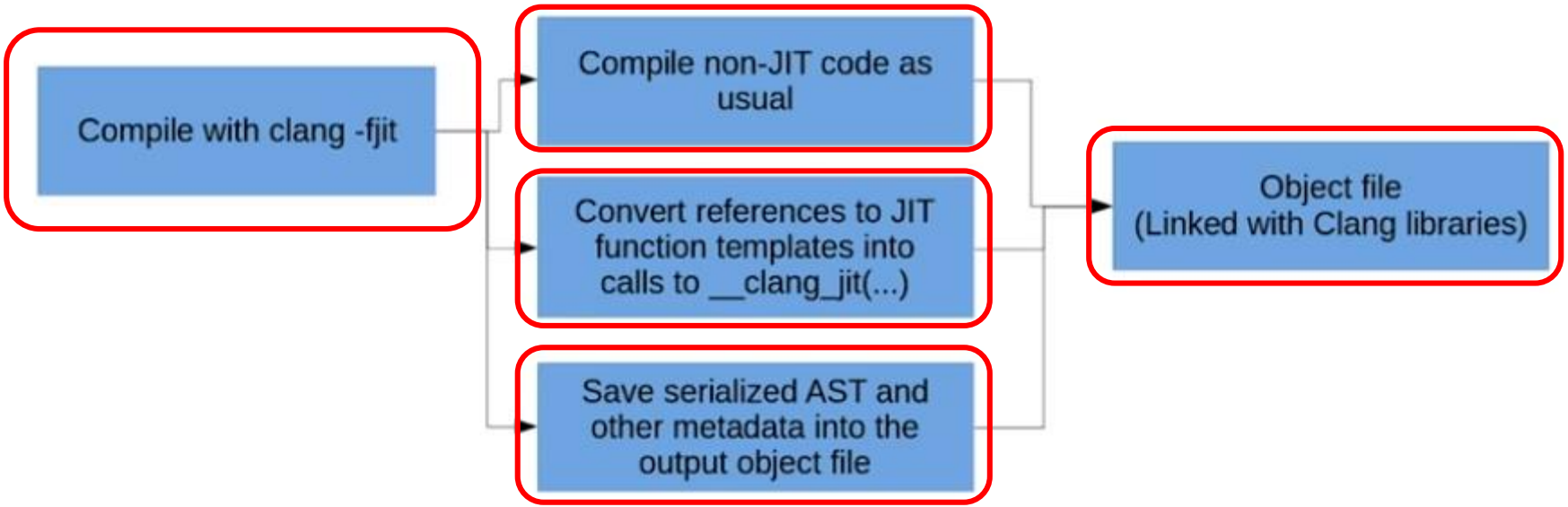
```
struct F {
  int i;
  double d;
};

template <typename T>
[[clang::jit]] void run() {
  std::cout << "I was compiled at runtime, sizeof(T) = " << sizeof(T) << "\n";
}

int main(int argc, char *argv[]) {
  std::string t(argv[1]);
  run<t>();
}
```

From (April 2019): https://arxiv.org/pdf/1904.08555.pdf

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel



P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

- Suggests to add infrastructure for JIT ([[clang::jit]] style), by adding:
  - Header: **<dynamic_instantiation>**:

```
#include <source_location>
#include <string>
namespace std {
        struct diagnostic;
}
```

  - Class: **diagnostics**:

```
                                        namespace std {
                struct diagnostic {
                const std::string &message() const;
                const std::source_location &location() const;
        };
}
```

    Return lvalue which meets: Cpp17CopyConstructible, Cpp17CopyAssignable, and Cpp17Destructible
  - Operator: **dynamic_function_template_instantiation < id-expression > (expression-list_opt)**
    Return lvalue which meets: Cpp17MoveConstructible, Cpp17MoveAssignable, and Cpp17Destructible
  - Operator: **dynamic_template_argument < template-argument ..._opt >**

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

```
template <typename T, int I>
void foo(int a) {
  std::cout << "I was compiled at runtime, I = " << I << "\n";
  std::cout << "I was compiled at runtime, sizeof(T) = " << sizeof(T) << "\n";
}
...
template <int J>
struct A { };
...
auto A_tid = dynamic_template_argument<A>;
auto A5 = A_tid.compose(5);

int i = ..., j = ...;
auto foo_TI = dynamic_function_template_instantiation<foo>(A5, i);
```

```
for (auto &W : foo_TI.warnings()) {
        std::cerr << "warning: " << W.message()<<'\n';
}

if (foo_TI) {
        foo_TI(j);
} else {
        std::cerr << "compilation failed!\n";

for (auto &W : foo_TI.errors()) {
        std::cerr << "error: " << W.message() << "\n";
}
}
```

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

Eigen library instantiation of metrixes:

Compile time:

	Time over Base
JIT Only	0.92s
(AoT) Single Specialization (double, size = 16)	2.37s
(AoT) Single Specialization (double, size = 7)	0.72s
(AoT) Single Specialization (double, size = 3)	0.62s
(AoT) Single Specialization (double, size = 1)	0.37s
(AoT) Two Specializations (double, size = 16) and (double, 7)	3.12s
Generic AoT Only (three floating-point types with dispatch)	7.12s
Generic AoT Only (double only)	2.72s
Nothing (just the includes and a main function)	-

double

size = 3

JIT	1.0s
Single Specialization	1.01s
AoT	8.05s

double

size = 7

JIT	8.34s
Single Specialization	8.45s
AoT	20s

double

size = 16

JIT	35.3s
Single Specialization	35.1s
AoT	36.2s

P1609R3: C++ Should Support Just-in-Time Compilation \ Hal Finkel

- Issues from the paper:

1. How to indicate that the **feature isn't available** (feature macro, constexpr function, both, something else)?

1. Restriction on **overloaded function templates** - how to relax (optional parenthesized type list after the id?)

1. **Statefulness** of the instantiation process - the result of friend injection **persist across different evaluations**?

1. How to provide additional **compilation implementation information** (e.g., compiler-optimization flags)?

1. How to provide ability to save/restore the **compilation state** of an instantiations (into / out of a **stream**)?

1. How to expose **result types** of the operators named **types provided**? Header file?

1. Should **dynamic_template_argument** and **typeid** be unified in some way?

P1708R2: Simple Statistical Functions \ Michael Wong, Micheal Chiu, Richard Dosselmann, Eric Niebler, Phillip Ratzlof, Vincent Reverdy

Suggests adding to **<numerics>** the following, as part of a **stats class holds information** on the container:

- Mean: μ or ¯x: (on **linear run-time)** $\frac{1}{n}\sum_{i=1}^{n} x_i.$
- Median: (without sorting, in linear time using the quickselect algorithm)
- Mode: value having the highest frequency (can be performed in linear time by counting consecutive (repeated) values)
- Standard Deviation: (computed in a a single pass) Populatio $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu^2).}$ $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.}$
  sample (s):
- Variance: (computed in a a single pass) Population: σ^2 sample: s^2

P1708R2: Simple Statistical Functions \ Michael Wong, Micheal Chiu, Richard Dosselmann, Eric Niebler, Phillip Ratzlof, Vincent Reverdy

```
template<typename T = double, typename Allocator = allocator<T>>
// ... requires ...
class stats {
public:
constexpr stats() noexcept;                    /* construction/destruction */
constexpr stats(int m);
constexpr stats(const stats& other);
constexpr stats(stats&& other);
stats& operator=(const stats& other);
stats& operator=(stats&& other);
~stats() = default;

template<typename ForwardIt>                   /* calculation */
// ... requires ...

void calc(ForwardIt first, ForwardIt last);    /* non template */

template<typename ForwardIt, typename UnaryPredicate>
// ... requires ...
void calc(ForwardIt first, ForwardIt last, UnaryPredicate p);
void calc(range r);
void calc(range r, UnaryPredicate p);
```

```
/* UnaryPredicate - metrics */
static const int metric_mean = 0b0000001;
static const int metric_median = 0b0000010;
static const int metric_mode = 0b0000100;
static const int metric_population_stddev = 0b0001000;
static const int metric_sample_stddev = 0b0010000;
static const int metric_population_var = 0b0100000;
static const int metric_sample_var = 0b1000000;
static const int metric_all = 0b1111111;


void metrics(int m);
constexpr int metrics() const noexcept;
constexpr T mean() const noexcept;

tuple<bool,T,T> median() const noexcept;
constexpr std::list<T> mode() const noexcept;
constexpr T population_stddev() const noexcept;
constexpr T sample_stddev() const noexcept;
constexpr T population_var() const noexcept;
constexpr T sample_var() const noexcept;
};
```

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

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

    a. 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**.

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

    a. 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
}
```

1. 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
  1. 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

P1795R1: System topology discovery for heterogeneous & distributed computing \ Gordon Brown, Ruyman Reyes, Michael Wong, Mark Hoemmen, Jeff Hammond, Tom Scogland, Domagoj Šarić

- Separate proposals for high-level interface and one for the low-level interfaces. (this is the low level one)
- Define abstract properties of system architectures and topology that are not tied to any specific hardware, Including:
  - **a hierarchy depth-based view**
  - **a memory-centric view**
  - **network-centric view.**

- Provide interface for querying properties of an execution resource, including relative affinity (זיקה) properties. binding execution agents and initialization of data.

- Examples of resources:
  - many-core CPUs, GPUs, FPGAs and DSPs to specifically designed vision and machine learning processors
  - memory modules

P1795R1: System topology discovery for heterogeneous & distributed computing \ Gordon Brown, Ruyman Reyes, Michael Wong, Mark Hoemmen, Jeff Hammond, Tom Scogland, Domagoj Šarić

- Header: **<system>**:

```
namespace experimental {
class system_topology {
        system_topology() = delete;
};
class system_resource {
                /* to be defined */
};
template <class T>
to-be-decided<system_resource> traverse_topology(const system_topology &, const T &) noexcept;
        /* this_system::discover_topology */
namespace this_system {
        system_topology discover_topology();
                }  // namespace this_system
}  // experimental
```

- Class: <**system_topology**>
- Class: <**system_resource**>
- Free functions:
  - **this_system::discover_topology**
  - Template function: template <class T>

to-be-decided<system_resource> **traverse_topology**(const

tem_topology &, const T &) noexcept;

P2004R0: Numbers and their Scopes \ Antony Polukhin

- Suggest separating evolving SG6 work from previous papers:
    1. P0101 was providing the "Introduction on SG6" and "Design Principles".
    2. P1889 is for design. Proposed scopes are:
        3. New numeric types that are widely useful and were already discussed
        4. Basic building blocks for implementing new numeric types on top of build-in types
        5. Minimal and consistent functionality to make the introduced types and functions usable for basic use-cases
- Newer ideas will be on different papers (in order to separate working process from the already accepted ideas)

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

P2070R0: A case for optional and object_ptr \ Peter Sommerlad, Anthony Williams, Michael Wong, Jan Babst

- Suggests to add: optional<T&>
  object_ptr<T> const

  std::optional<reference_wrapper<T>>
  object_ptr<T>