anonymous constants: past, present and future

Why anonymous constants?

1. Naming everything is not the solution, otherwise we wouldn't have lambdas i.e. anonymous functions
2. Requires name
3. Excessively wordy
4. Moved far from point of use
5. Requires maintaining file, manual sorting sparse list

Why should anonymous constants be static?

- programmer expectation that constants are static like string literals, ex. "Hello World", and assembly inline/local constants
- temporaries can immediately dangle; dangling constants are embarrassing
- want `ROM` guarantee or at least next best thing `const` and `static`
- memory safe
- thread safe

embarrassing (Part 1)

Most if not all programming languages in use, except C++, don't immediately dangle something as simple and beginner as constants/literals.

embarrassing (Part 2)

Assembly is easier and safer

Guaranteed static and const

```
	.global a
	.section	.rodata
	.align 4
	.type	a, @object
	.size	a, 4
a:
	.long	5
```

Decomposed constant in assembly instruction

```
mov <register>,<constant>
mov <memory>,<constant>
```

embarrassing (Part 3)

C is easier and safer

"*6.5.2.5 Compound literals*" **¶ 5**

"*The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has* ***automatic storage duration associated with the enclosing block****.*"

embarrassing (Part 4)

C++ `WAS` easier and safer

CFront i.e. C with classes, because C++ was preprocessed to C code, which didn't immediately dangle their constants/literals.

Requirements

1. `const [&]`; will only be used in a constant fashion
2. `constexpr`; can be constructed at compile time
3. `consteval`; was constructed at compile time

Result

- `constinit`; constant initialization, sensible lifetime extension of a temporary to a global

Lambda and macros (present)

Current best solution

```cpp
const std::string& dangler(const std::string& s) { return s; }
#define CONSTANT ...// macro can conceal lambda usage
void h() {
    dangler([] /*consteval*/ -> auto const & {
            static constinit const std::string anonymous{"Hello Wo
            return anonymous;
        }());
    dangler(CONSTANT("Hello World"s));
}
```

CON(s) (explicit)

- How many times must we say const?
- Compilers must undue boilerplate

Static storage for braced initializers (C++26) (p2752r3)

```
void f(std::initializer_list<double> il);
void h() {
    f({1, 2, 3});
    //static constexpr double __b[3] = {double{1}, double{2}, doub
    //f(std::initializer_list<double>(__b, __b+3));
}
```

When size is 5000, 500, 50, 5, 1?

Superfluous bracelets when size is 1?

CON(s) (explicit)

- No size restrictions
- No static guarantee

std::span over an initializer list (C++26) (p2447r6)

```
const double& dangler(const double& d) { return d; }
const double& dangler(std::span<const double, 1> s)
    { return dangler(s.front()); }
void h() {
    dangler({1});// practically safe
    //static constexpr double __b[3] = {double{1}}; // backing arr
    //f(std::initializer_list<double>(__b, __b+1));
}
```

CON(s) (explicit)

- library user must wrap in initializer list
- library writer must duplicate functions
- No static guarantee

`std::constant_wrapper` (p2781r4)

```
std::cw<1>
std::cw<2uz>
std::cw<3.0>
std::cw<4.f>
std::cw<foo>
std::cw<my_complex(1.f, 1.f)>
```

CON(s) (explicit)

- Only works on structural types

C `constexpr static` (C23)

```
&(static constexpr struct foo) {1, 'a', 'b'}
```

PRO(s)

- Best explicit syntax; better if one keyword

CON(s) (explicit)

- Only works on C23
- `constexpr static` vs. constant or read_only
- "For the storage duration of the created objects we go with C++ for compatibility"

constexpr structured bindings and references to constexpr variables (p2686r3)

- **Allowing static and non-tuple constexpr structured binding**
- `Making constexpr implicitly static`
- ~~Always re-evaluate a call to get?~~
- **Symbolic addressing**

CON(s)

- **Wording:** ~~or temporary object~~

Better than Rust?

```
// rvalue_static_promotion
const X: &'static T = &<constexpr foo>;
```

"... the only keyword that most Rust programmers should need to know is const – I imagine static variables will be used quite rarely."

CON(s) (explicit)

- not really a automatic promotion as it must be requested
- like C++, a tool that must be used, not default

Breakage? (Part 1)

	with static	
code	**safer**	**simpler**
`auto [](const std::string& s) { return s; }("HW"s);`	✓	✓
`const std::string& s = "HW"s;`	~	✓
`const std::string s = "HW"s;`	~	✓

Breakage? (Part 2)

The point

- While the consistency is appreciated for simplicity
- Only temporaries passed to function calls need it due to immediate dangling. The other two would benefit for instance if & was returned.

Breakage? (Part 3)

The point

- Lifetime extending temporary constants passed to functions is not expected to cause problems as the function knows not whether the argument is global, local or temporary.

Breakage? (Part 4)

What about the other two?

Working Draft, Standard for Programming Language C++ [n4910]

6.9.3.2 Static initialization [basic.start.static]

3 "*An implementation is permitted to perform the initialization of a variable with static or thread storage duration as a static initialization even if such initialization is not required to be done statically …*"

Performance (Part 1)

https://en.cppreference.com/w/c/language/const

const type qualifier

"*Objects declared with const-qualified types may be placed in read-only memory by the compiler, and if the address of a const object is never taken in a program, it may not be stored at all.*"

Performance (Part 2)

- avoids repeated stack [and heap] allocations every time function called in one thread
- avoids repeated stack [and heap] allocations every time function called by multiple threads
- assembly inline constants / embedded in instruction
- potential deduplication
- potential referencing component of larger constant

Teachability (Part 1)

Requirements

1. const [&]; will only be used in a constant fashion
2. constexpr; can be constructed at compile time
3. consteval; was constructed at compile time

Teachability (Part 2)

Requirement #1 const [&]

C++ Core Guidelines

F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const [cppcgrf42]

Teachability (Part 3)

Requirement #2 constexpr

Make everything constexpr

- As the limit of constexpr approach 100%, the programmer only need to concern themselves with requirement #3

Teachability (Part 4)

Requirement #3 consteval

- Was the constant initiated with only constants?

Teachability (Part 5)

```
auto whatever = {
    {1, 2, 3},
    {4, 2, 5},
    {3, 2, 1},
}
```

- How easy is it for the programmer to tell that this was only initialized with constants? VERY EASY

Teachability (Part 6)

```
const auto i = 2;
auto whatever = {
    {1, i, 3},
    {4, i, 5},
    {3, i, 1},
}
```

- How easy is it for the programmer to tell that this was only initialized with constants? EASY

Teachability (Part 7)

```
auto whatever = {
    {1, i, 3},
    {4, i, 5},
    {3, i, 1},
}
```

- How easy is it for the programmer to tell that this was NOT initialized completely with constants? EASY. Even if `i` was far away, the variable is an indicator of it not likely being constant. After all, it is variable.

MAY vs MUST

May be static vs Must be static

- How does a programmer know whether the compiler made it static?
- How does a programmer know whether they even have a dangle that needs fixing?
- Pessimism = uglier, harder to maintain code.
- Have to look at assembly code to know.
- Expect beginners to look at assembly code.
- Varies among compilers
- Varies among a single compiler's flags