

# C++ Club Meeting Notes

Gleb Dolgich

2018-08-02

## Follow-up: Four Habit-Forming Tips to Faster C++ (1/4)

Post by KDAB

“Cache member-variables and reference-parameters”

The original version in the article doesn't even work without aliasing:

```
1 template<typename T>
2 class complex {
3     complex& operator*=(const complex<T>& a) {
4         real = real * a.real - imag * a.imag;
5         // BUG: real is changed, then used in subsequent calculation
6         imag = real * a.imag + imag * a.real;
7         return *this;
8     }
9 };
```

## Follow-up: Four Habit-Forming Tips to Faster C++ (2/4)

Post by KDAB

“Cache member-variables and reference-parameters”

The recommended version from the article:

```
1 template<typename T>
2 class complex {
3     complex& operator*=(const complex<T>& a) {
4         T a_real = a.real, a_imag = a.imag;
5         T t_real = real, t_imag = imag;
6         real = t_real * a_real - t_imag * a_imag;
7         imag = t_real * a_imag + t_imag * a_real;
8         return *this;
9     }
10};
```

## Follow-up: Four Habit-Forming Tips to Faster C++ (3/4)

Post by KDAB

“Cache member-variables and reference-parameters”

Proposed “economy” version:

```
1 template<typename T>
2 class complex {
3     complex& operator*=(const complex<T>& a) {
4         const auto real_this{real}; // modified by our code
5         const auto real_a{a.real}; // may be modified if aliased
6         real = real * a.real - imag * a.imag;
7         imag = real_this * a.imag + imag * real_a;
8         return *this;
9     }
10 };
```

## Follow-up: Four Habit-Forming Tips to Faster C++ (4/4)

Post by KDAB

“Cache member-variables and reference-parameters”

Proposed “easy” version:

```
1 template<typename T>
2 class complex {
3     complex& operator*=(complex<T> a) {
4         const auto self{*this};
5         real = self.real * a.real - self.imag * a.imag;
6         imag = self.real * a.imag + self.imag * a.real;
7         return *this;
8     }
9 };
```

## C++ Quiz by Shafik Yaghmour via Twitter (1/2)

[Tweet](#)

C:

```
1 | malloc(0)
```

C++:

```
1 | new int[0]
```

What's the behavior of these expressions?

1. Both return NULL/nullptr
2. Both return an address that's invalid to dereference
3. Behavior can vary
4. Undefined behavior

## C++ Quiz by Shafik Yaghmour via Twitter (2/2)

[Tweet](#)

In C99 and C11 `malloc(0)` is allowed to return either NULL or pointer that is not valid to dereference:

- ▶ [C99 N1256 7.20.3](#)
- ▶ [C11 N1570 7.22.3](#)

In C++ we get back an array with zero elements:

- ▶ [expr.new](#)
- ▶ [dynamic.allocation](#)
- ▶ [dynamic.allocation](#)

# Allocating 0 bytes

## How much memory does malloc(0) allocate?

On most systems, this little C program will soak up all available memory:

```
while (1) {
    malloc(0);
}
```

so the answer is not the obvious "zero." But before getting into `malloc(0)`, let's look at the simpler case of `malloc(1)`.

There's an interesting new C programmer question about `malloc`: "Given a pointer to dynamically allocated memory, how can I determine how many bytes it points to?" The answer, rather frustratingly, is "you can't." But when you call `free` on that same pointer, the memory allocator knows how big the block is, so it's stored *somewhere*. That somewhere is commonly adjacent to the allocated memory, along with any other implementation-specific data needed for the allocator.

Now we can approach the case of allocating zero bytes. It turns out there's a silly debate about the right thing to do, and it hasn't been resolved, so technically allocating zero bytes is implementation-specific behavior. One side thinks that `malloc(0)` should return a null pointer and be done with it. It works, if you don't mind a null return value serving double duty. It can either mean "out of memory" or "you didn't request any memory."

The more common scheme is that `malloc(0)` returns a unique pointer. You shouldn't dereference that pointer because it's conceptually pointing to zero bytes, but we know from our adventures above that at least `dimalloc` is always going to allocate a 32 byte block on a 64-bit system, so that's the final answer: it takes 32 bytes to fulfill your request for no memory.

## Metashell GUI

- ▶ [Code](#) (MIT)
  - ▶ [Reddit](#)
- ▶ [Metashell](#) “The goal of this project is to provide an interactive template metaprogramming shell.” (GPL-3.0)

# Ericsson CodeCompass

CodeCompass is a software comprehension tool for large scale software written in C/C++ and Java

- ▶ [Code](#) (GPL-3.0)

## Expect the Expected, by Andrei Alexandrescu

- ▶ [Video](#)
- ▶ [Reddit](#)
- ▶ Howard Hinnant on how to initialize a class using delegating constructors

# CppCast: Parallel Ranges with Chris DiBella

- ▶ Podcast
- ▶ P0836R1 Introduce Parallelism to the Ranges TS

# Pacific++ 2017: Toby Allsopp “An Introduction to the Proposed Coroutine Support for C++” (1/2)

## ▶ Video

pacific++

### Coroutine body transformation

```
lazy<int> calculate()
{
    the promise is
    actually stored
    in the coroutine
    state

    int result = hard_work();
    co_return result;
}

local variables
may actually be
stored in the
coroutine state

using promise_type =
coroutine_traits<lazy<int>>::promise_type;
{
    promise_type p;
    auto r = p.get_return_object();
    co_await p.initial_suspend();
    try {
        int result = hard_work();
        p.return_value(result);
        goto final_suspend;
    } catch (...) {
        p.unhandled_exception();
    }
    final_suspend:
    co_await p.final_suspend();
    destroy;
}
```

Toby Allsopp—Pacific++ 2017

25



TOBY ALLSOPP

An Introduction to the  
Proposed Coroutine Support  
for C++

# Pacific++ 2017: Toby Allsopp “An Introduction to the Proposed Coroutine Support for C++” (2/2)

## Coroutine state

stores promise, parameter copies, local variables  
and any information needed to resume the  
coroutine at the correct point

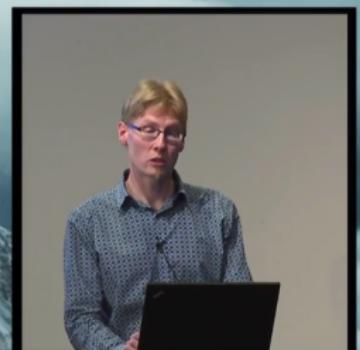
dynamically allocated (can be elided in certain  
circumstances)

created when the coroutine is called

destroyed when control flows off the end of the  
transformed coroutine body

promise
parameter copies
local variables
resume point

pacific++



TOBY ALLSOPP

An Introduction to the  
Proposed Coroutine Support  
for C++

# Qt and the Coroutines TS

- ▶ [Post by Jeff Trull](#)
- ▶ [Post by Jesús Fernández](#)
- ▶ [Boost.Asio coroutine support](#)
- ▶ [Coroutine Theory Series, by Lewis Baker](#)
- ▶ [cppcoro: Coroutine Utilities, by Lewis Baker](#)
- ▶ [How Coroutines Work, by Kirit Sælensminde](#)

# Exploiting Coroutines to Attack the “Killer Nanoseconds”



In this work, we compare and contrast the state-of-the-art approaches to reduce CPU stalls due to cache misses for pointer-intensive data structures. We present an in-depth experimental evaluation and a detailed analysis using four popular data structures: hash table, binary search, Masstree, and Bw-tree. Our focus is on understanding the practicality of using coroutines to improve throughput of such data structures.

# Herb Sutter: How to Adopt Modern C++17 into Your C++ Code (1/7)

- ▶ [Video](#)
- ▶ [Reddit](#)

# Herb Sutter: How to Adopt Modern C++17 into Your C++ Code (2/7)



## NB: Non-Owning \*/& Are Still Great

### ▶ C++98 "Classic":

```
void f( widget& w ) { // if required
    use(w);
}
void g( widget* w ) { // if optional
    if(w) use(*w);
}
```



\* and & FTW

### ▶ Modern C++ "Still Classic":

```
void f( widget& w ) { // if required
    use(w);
}
void g( widget* w ) { // if optional
    if(w) use(*w);
}
```

```
auto upw = make_unique<widget>();
...
f( *upw );
auto spw = make_shared<widget>();
...
g( spw.get() );
```

Microsoft Build

May 7-9, 2018 // Seattle, WA



## Did You Know: Smart Pointers are Pretty Smart

- ▶ Derived-to-base just works:

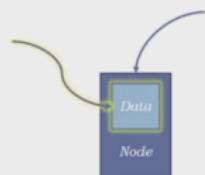
```
// void f(const shared_ptr<Base>&);  
f( make_shared<Derived>() );           // ok
```

- ▶ Non-const-to-const just works:

```
// void f(const shared_ptr<const Node>&);  
f( make_shared<Node>() );             // ok
```

- ▶ Bonus geek cred if you know the aliasing ctor:

```
struct Node { Data data; };  
shared_ptr<Data> get_data(const shared_ptr<Node>& pn) {  
    return { pn, &(pn->data) };          // ok  
}
```



Microsoft Build

May 7-9, 2018 // Seattle, WA

# Herb Sutter: How to Adopt Modern C++17 into Your C++ Code (4/7)



## Antipatterns Hurt Pain Pain

- ▶ Antipattern #1: Parameters  
(Note: Any refcounted pointer type.)

```
void f( refcnt_ptr<widget>& w ) {  
    use(*w);  
} // ?  
  
void f( refcnt_ptr<widget> w ) {  
    use(*w);  
} // ?!?
```

- ▶ Antipattern #2: Loops  
(Note: Any refcounted pointer type.)

```
refcnt_ptr<widget> w = ...;  
for( auto& e: baz ) {  
    auto w2 = w;  
    use(w2, *w, *w2, whatever);  
} // ?!?!?!
```

Example (HT: Andrei Alexandrescu): In late 2013, Facebook RocksDB changed pass-by-value *shared\_ptr* to pass-\*&.

⇒ 4x QPS (100K to 400K) in one benchmark

Example: C++/WinRT factory cache was slow.  
“Obvious” suspect: cache’s mutex lock

Actual culprit: >50% time spent in extra  
AddRef/Release on returned object

Microsoft Build

May 7-9, 2018 // Seattle, WA

# Herb Sutter: How to Adopt Modern C++17 into Your C++ Code (5/7)



## Guideline: Dereference *Unaliased+Local* RC Ptrs

### ► The reentrancy pitfall (simplified):

```
// global (static or heap), or aliased local  
... shared_ptr<widget> other_p ...  
  
void f( widget& w ) {  
    g();  
    use(w);  
}  
void g() {  
    other_p = ... ;  
}  
  
void my_code() {  
  
    f( *other_p );      // passing *nonlocal  
    other_p->foo();   // (or nonlocal->)  
} // should not pass code review
```

### ► "Pin" using unaliased local copy:

```
// global (static or heap), or aliased local  
... shared_ptr<widget> other_p ...  
  
void f( widget& w ) {  
    g();  
    use(w);  
}  
void g() {  
    other_p = ... ;  
}  
  
void my_code() {  
    auto pin = other_p; // 1 ++ for whole tree  
    f( *pin );          // ok, *local  
    pin->foo();        // ok, local->  
}
```

Microsoft Build

May 7-9, 2018 // Seattle, WA



## Passing Smart Pointers

```
unique_ptr<widget> factory();           // source – produces widget
void sink( unique_ptr<widget> );          // sink – consumes widget
void reseat( unique_ptr<widget>& );        // "will" or "might" reseat ptr
void thinks( const unique_ptr<widget>& ); // usually not what you want

shared_ptr<widget> factory();            // source + shared ownership
                                         // when you know it will be shared, perhaps by factory itself
void share( shared_ptr<widget> );         // share – "will" retain refcount
void reseat( shared_ptr<widget>& );        // "might" reseat ptr
void may_share( const shared_ptr<widget>& ); // "might" retain refcount
```

Microsoft Build

May 7-9, 2018 // Seattle, WA

# Herb Sutter: How to Adopt Modern C++17 into Your C++ Code (7/7)



## Summary: “Doing it right”

1. **Never pass smart pointers (by value or by reference) unless you actually want to manipulate the pointer ⇒ store, change, or let go of a reference.**
  - Prefer passing objects by \* or & as usual – just like always.
  - Remember: Take unaliased-local copy at the top of a call tree, don't pass f(\*other\_p).
  - Else if you do want to manipulate lifetime, great, do it as on previous slide.
2. **Express ownership using unique\_ptr wherever possible**, including when you don't know whether the object will actually ever be shared.
  - It's free = exactly the cost of a raw pointer, by design.
  - It's safe = better than a raw pointer, including exception-safe.
  - It's declarative = expresses intended uniqueness and source/sink semantics.
  - It removes many (often most) objects out of the ref counted population.
3. **Else use make\_shared up front wherever possible**, if object will be shared.

Microsoft Build

May 7-9, 2018 // Seattle, WA

# Unity (“jumbo”) builds

AKA: amalgamated, or Single Compilation Unit (SCU) builds.

- ▶ [A Guide to Unity Builds by Viktor Kirilov](#)
- ▶ [Support for Unity \(Jumbo\) Files in Visual Studio 2017 15.8 \(Experimental\)](#)
- ▶ [Chromium instructions](#) and [GoogleDoc](#)
- ▶ [Reddit](#)
- ▶ Other tools:
  - ▶ CMake [cotire](#) and [Unity Build macro](#)
  - ▶ [FASTBuild](#)
  - ▶ [Meson](#)
  - ▶ [waf](#)
  - ▶ [RudeBuild plugin for Visual Studio](#)

## `const auto*` versus `const auto` for Pointer Types

Article

# Twitter

 std::data::pup 🐶      [↑ 3 Replies, 5 Quotes](#)

@6b766e      

whys is called 'the rust community' and not  
'the cargo cult'

27/07/2018, 22:58 (Friday)  
Twitter for iPhone

Retweeted by @odinthenerd  
29/07/2018, 11:52

1225 Likes	196 Retweets	<a href="#">Thread &gt;</a>
------------	--------------	-----------------------------