Back to Basics: Modern C++ Style

loops
pointers & references
smart pointers
variable declarations
parameter passing

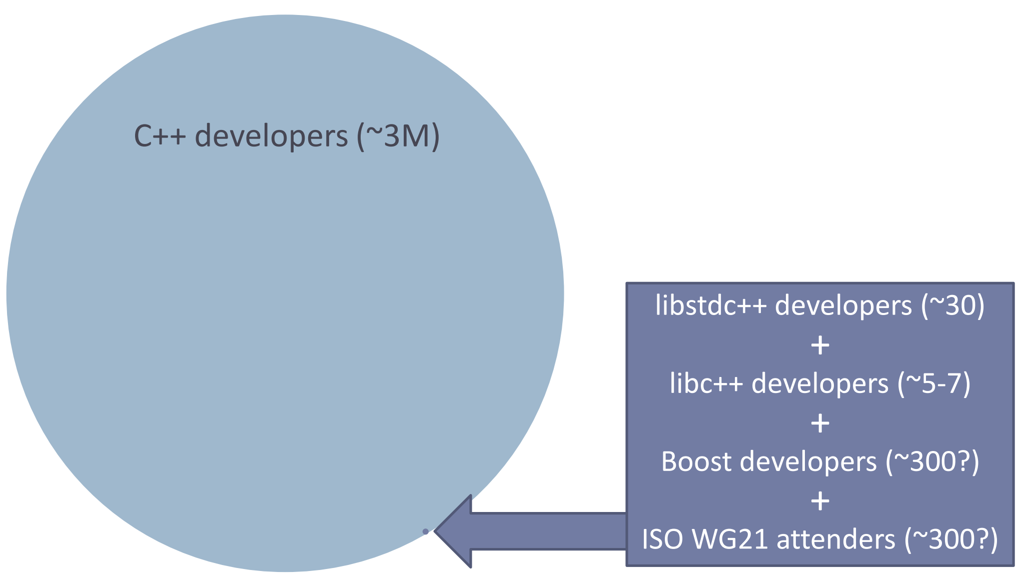
Herb Sutter

C A

Complexity Anonymous

A 12-step program
for **good people** attempting to
recover from **complexity addiction**

It's hard to ~~be~~ remember you're an expert



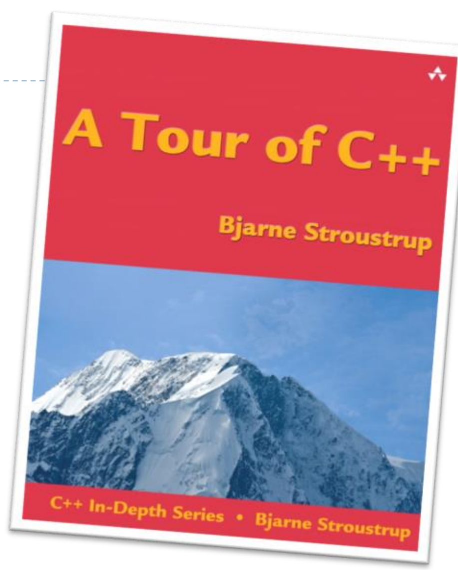
Reality Check

Occurrences of “&&” in Bjarne’s 90-min Tue keynote? **0**

Value of modern C++’s simple usable defaults? ***Priceless***

Most Important C++ Book

- “What should every C++ programmer be expected to know?”
  - For years, there has not been a single source to point to.
- Now there is. In 180 pages you can read on a long plane flight.
  - **Recommend it heavily!**
  - Also a demonstration that modern C++ is simpler to teach and explain.



This Talk

- This talk focuses on **defaults**, basic styles and idioms in modern C++.
  - ✗ ~~"Default" != "don't think."~~
  - ✓ "Default" == "don't **overthink**." Esp. don't **optimize prematurely**.
- These reinforce (not compete with) the "fundamentals."
  - **"Write for clarity and correctness first."**
  - **"Avoid premature optimization."** By default, prefer *clear* over *optimal*.
  - **"Avoid premature pessimization."** Prefer *faster* when *equally* clear.

Prefer range-for

why do this

```
for( auto i = begin(c); i != end(c); ++i ) { … use(*i); … }
```

when you can do this

```
for( auto& e : c ) { … use(e); … }
```

and soon this

```
for( e : c ) { … use(e); … }
```

Use smart pointers effectively…

… but still **use lots of raw * and &**, they're great!

wait, what?

Don't Use Owning *, *new*, or *delete*

- C++98:

```
widget* factory();
void caller() {
   widget* w = factory();
   gadget* g = new gadget();
   use( *w, *g );
   delete g;
   delete w;
}
```

  red ⇒ now "mostly wrong" ☺

- Don't use owning *, *new* or *delete*.
  - Except: Encapsulated inside the implementation of low-level data structures.

- Modern C++:

```
unique_ptr<widget> factory();
void caller() {
   auto w = factory();
   auto g = make_unique<gadget>();
   use( *w, *g );
}
```

- For "new", use *make_unique* by default, *make_shared* if it will be shared.
- For "delete", write nothing.

NB: important qualifier

Don’t Use **Owning** *, *new*, or *delete*

- C++98:

```
widget* factory();
void caller() {
   widget* w = factory();
   gadget* g = new gadget();
   use( *w, *g );
   delete g;
   delete w;
}
```

red ⇒ now “mostly wrong” ☺

- C++14:

```
unique_ptr<widget> factory();
void caller() {
   auto w = factory();
   auto g = make_unique<gadget>();
   use( *w, *g );
}
```

- Don’t use owning *, *new* or *delete*.
  - Except: Encapsulated inside the implementation of low-level data structures.
- For “new”, use *make_unique* by default, *make_shared* if it will be shared .
- For “delete”, write nothing.

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

(More on parameter passing coming later…)

- 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() );
```

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,*w2,w,*w,whatever);
} // ?!?!?!?!
```

Example (thanks Andrei): In late 2013, Facebook RocksDB changed from pass-by-value *shared_ptr* to pass-*/&.
QPS improved 4× (100K to 400K) in one benchmark.

http://tinyurl.com/gotw91-example

No Copy No Cry

FAQ: Smart Pointer Parameters — See GotW #91 (*tinyurl.com/gotw91*)

Refcounted smart pointers are about managing the **owned object's lifetime**.
Copy/assign one only when you intend to manipulate the **owned object's lifetime**.

Any "smart pointers (or std::vectors) are slow" performance claims based on code that copies/assigns smart pointers (or std::vectors) – including passing by value or copying/assigning in loops – when copies are not needed are fundamentally flawed.

Yes, this applies to your refcounted smart pointer:

- shared_ptr (Boost, TR1, std::)
- retain/release (Objective-C ARC, Clang 3.5)
- AddRef/Release (COM and WinRT, C++/CX ^)
- any other refcounting strategy you will ever see

Passing Smart Pointers (GotW #91)

&&? wait for it…

```
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 thinko(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>& );               // "will" or "might" reseat ptr
void may_share( const shared_ptr<widget>& );      // "might" retain refcount
```

How to "Do It Right" (Partial)

1. **Never pass … pointers (by value or by reference) unless you actually want to m… …tore, change, or let go of a reference.**
   - Prefer p…
   - **Else** if y… …slide.
2. **Express c…** …g when you don'…
   - It's fre…
   - It's sa…
   - It's declarative = express… …emantics.
   - **It removes many (often most) objects out of the …ulation.**
3. **Else use make_shared up front wherever possible**, if object will be shared.

Not quite done: One guideline missing…

…and it applies to any RC pointer type, in almost any language / library

Guideline: Dereference *Unaliased+Local* RC Ptrs

- The reentrancy pitfall (simplified):

```
// global (static or heap), or aliased local
... shared_ptr<widget> g_p ...

void f( widget& w ) {
   g();
   use(w);
}
void g() {
   g_p = ... ;
}

void my_code() {

   f( *g_p );        // passing *nonlocal

}       // should not pass code review
```

- "Pin" using unaliased local copy.

```
// global (static or heap), or aliased local
... shared_ptr<widget> g_p ...

void f( widget& w ) {
   g();
   use(w);
}
void g() {
   g_p = ... ;
}

void my_code() {
   auto pin = g_p;   // 1 ++ for whole tree
   f( *pin );        // ok, *local

}
```

Guideline: Dereference *Unaliased+Local* RC Ptrs

- The reentrancy pitfall (simplified):

```
// global (static or heap), or aliased local
... shared_ptr<widget> g_p ...

void f( widget& w ) {
   g();
   use(w);
}
void g() {
   g_p = ... ;
}

void my_code() {

   f( *g_p );        // passing *nonlocal
   g_p->foo();       // (or nonlocal->)
}       // should not pass code review
```

- "Pin" using unaliased local copy.

```
// global (static or heap), or aliased local
... shared_ptr<widget> g_p ...

void f( widget& w ) {
   g();
   use(w);
}
void g() {
   g_p = ... ;
}

void my_code() {
   auto pin = g_p;   // 1 ++ for whole tree
   f( *pin );        // ok, *local
   pin->foo();       // ok, local->
}
```

Summary: How to “Do It Right”

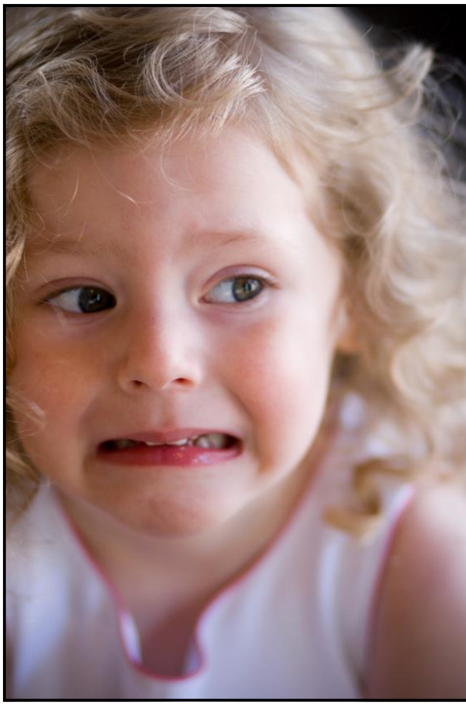
1. **Never pass smart pointers (by value or by reference) unless you actually want to manipulate the pointer ⇒ store, change, or let go of a reference.**
   - Prefer passing objects by * or & as usual – just like always.
     Remember: Take *unaliased+local* copy at the top of a call tree, don’t pass f(*g_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.

Write **make_unique** (by default)
or **make_shared** (when needed)
instead of *new* and *delete*.

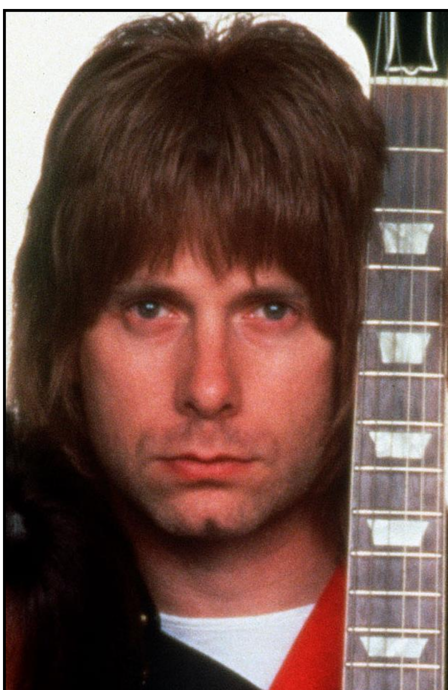
Don’t use <u>owning</u> raw *, *new*, or *delete* any more, except rarely inside the implementation details of low-level data structures.

Do use <u>non-owning</u> raw * and &, especially for parameters.

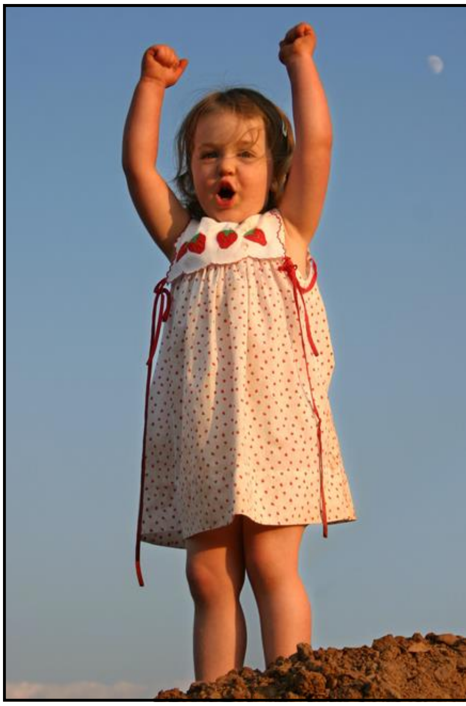
Don’t copy/assign refcounted smart pointers, including pass-by-value or in loops, unless you really want the semantics they express: altering object lifetime.



Let's talk about *auto*…



It's okay, it's *really simple*…



Spoiler

To make type track, deduce:
auto *var = init;*

To make type stick, commit:
auto var = ***type****{ init };*
or ***type*** *var{ init };*

Consider This Code

- **Guru Meditation Q:** What does this code do?

```
template<class Container, class Value>
void [redacted]( Container& c, Value v )
{
   if( find(begin(c), end(c), v) == end(c) )
      c.push_back( move(v) );
   assert( !c.empty() );
}
```

Why Not "Just Deduce the Type"?

- **Counterarguments: "Oi, but it's unreadable!" "What's my type?"**
- This is a weak argument for three reasons:
  - (Minor) It doesn't matter to anyone who uses an IDE.
  - (Major) It reflects bias to code against **implementations**, not **interfaces**.
  - (Major) We already ignore actual types with **templates** and **temporaries**.

```
template<class Container, class Value>    // what type is Container? Value?
void append_unique( Container& c, Value v ) // anything usable like this…
{
   if( find(begin(c), end(c), v) == end(c) )  // what type does find return?
      c.push_back( move(v) );                 // anything comparable to end(cont)…
   assert( !c.empty() );  // what type does .empty return?
}                         //  anything testable like a bool…
```

  - We also ignore actual types with virtual functions, *function*<>, etc.

Why Deduce: (1) Correctness

- With deduction you always get right type. *Repetition $\propto$ P(lying)*
  - Example:

```
void f( const vector<int>& v ) {
   vector<int>::iterator i = v.begin();             // ?
}
```

  - Options:

```
void f( const vector<int>& v ) {
   vector<int>::iterator i = v.begin();             // error
   vector<int>::const_iterator i = v.begin();       // ok + extra thinking
   auto i = v.begin();                              // ok, default
}
```

Why Deduce: (2) Correctness + Maintainability

- Using deduction makes your code more robust in the face of change.
  - Deduction tracks the correct type when an expression's type changes.
  - Committing to explicit type = silent conversions, needless build breaks.
- Examples:

```
int i = f(1,2,3) * 42;                          // before: ok enough
int i = f(1,2,3) * 42.0;                        // after: silent narrowing conversion
auto i = f(1,2,3) * 42.0;                       // after: still ok, tracks type

widget w = factory();                           // before: ok enough, returns a widget
widget w = factory();                           // after: silent conversion, returns a gadget
auto w = factory();                             // after: still ok, tracks type

map<string,string>::iterator i = begin(dict);   // before: ok enough
map<string,string>::iterator i = begin(dict);   // after: error, unordered_map
auto i = begin(dict);                           // after: still ok, tracks type
```

Why Deduce: (3) Performance

- Deduction guarantees no implicit conversion will happen.
  - A.k.a. "guarantees better performance by default."
  - Committing to an explicit type that requires a conversion means silently getting a conversion whether you expected it or not.

Why Deduce: (4) Usability

- Using deduction is your only good (usable and efficient) option for hard-to-spell and unutterable types like:
  - lambdas,
  - binders,
  - *detail::* helpers,
  - template helpers, such as expression templates (when they should stay unevaluated for performance), and
  - template parameter types, which are anonymized anyway,
- … short of resorting to:
  - repetitive *decltype* expressions, and
  - more-expensive indirections like *std::function*.

Why Deduce: (5) Convenience

- And, yes, "basic deduction" *auto x = expr;* syntax is almost always less typing.
  - Mentioned last for completeness because it's a common reason to like it, but it's not the biggest reason to use it.

Why Deduce: Wrapup

- **Prefer *auto x = expr;* by default on variable declarations.**
  - It offers so much correctness, clarity, maintainability, performance and simplicity goodness that you're only hurting yourself (and your code's future maintainers) if you don't.
  - Prefer to habitually **program against interfaces, not implementations**. We do this all the time in temporaries and templates anyway and nobody bats an eye.
- **But: Do commit to an explicit type when you really mean it,** which nearly always means you want an explicit conversion.
  - **Q:** But even then, does "commit to an explicit type" mean "don't use *auto*"?

Left-to-right auto style

- Deduce to track if you don't need to commit to a type:
  - const char* s = "Hello"; — **auto s** = "Hello";
  - widget w = get_widget(); — **auto w** = get_widget();
- Commit to stick to a specific type. Try it **on the right** (same syntax order):
  - employee e{ empid }; — **auto e** = employee{ empid };
  - widget w{ 12, 34 }; — **auto w** = widget{ 12, 34 };
- With heap allocation, type is **on the right** naturally anyway:
  - C++98 style: — **auto w** = new widget{};
  - C++14 style: — **auto w** = make_unique<widget>();
- Teaser: Does this remind you of anything else in C++11? and C++14?
  - int f(double); — **auto f**(double) -> int; // C++11
  - **auto f**(double) { … } // C++14

The Elephant

But what about

int x = 42;

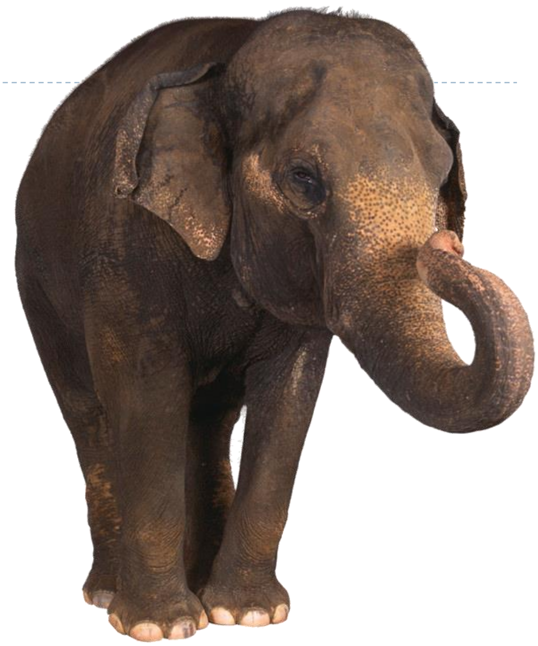
vs.

auto x = 42;

?

"OBVIOUSLY int x = 42;
is the tersest and clearest style."

Right?



Left-to-right auto style

```
employee e{ empid };              auto e = employee{ empid };
widget w = get_widget();          auto w = get_widget();
```

- Now consider literal suffixes:

```
int x = 42;                       auto x = 42;
float x = 42.;                    auto x = 42.f;      // no narrowing
unsigned long x = 42;             auto x = 42ul;
string x = "42";                  auto x = "42"s;     // C++14
chrono::nanoseconds x{ 42 };      auto x = 42ns;      // C++14
```

- Remember functions, lambdas, and aliases:

```
int f( double );                  auto f ( double ) -> int;
                                  auto f = [=]( double ) { /*…*/ };
typedef set<string> dict;         using dict = set<string>;
template<class T> struct myvec    template<class T>
{ typedef vector<T,myalloc> type; };   using myvec = vector<T,myalloc>;
```

Left-to-right modern C++ style

- The C++ world is moving to left-to-right everywhere:

	category name = type *and/or* initializer ;
Auto variables:	**auto e** = employee{ empid }; **auto w** = get_widget();
Literals:	**auto x** = 42; **auto x** = 42.f; **auto x** = 42ul;
User-defined literals:	**auto x** = “42”s; **auto x** = 1.2ns;
Function declarations:	**auto func** (double) -> int;
Named lambdas:	**auto func** = [=](double) { /*…*/ };
Aliases *(no more typedefs)*:	**using dict** = set<string>;
Template aliases:	template<class T> **using myvec** = vector<T,myalloc>;

I Know Some of You Have Been Wondering

- Consider:

```
auto x = value;
```

  - Q: Does this “=” create a temporary object plus a move/copy?
    - Standard says “No.” The code *T x = a;* has exactly the same meaning as *T x(a);* when *a* has type *T* (or derived from *T*)… and *auto x = a;* guarantees the types are the same (yay *auto*) so it always means exactly the same as *auto x(a)*.

I Know Some of You Have Been Wondering

- Consider:

```
auto x = type{value};
```

  - Q: Does this "=" create a temporary object plus a move/copy?
    - Standard says "Yes, but": The compiler may elide the temporary.
    - In practice, compilers do (and in the future routinely will) elide this temporary+move. However, the type must still be movable (which includes copyable as a fallback).

(The) Case Where You Can't Use "*auto* Style"

- **Case:** (1) Explicit "*type{}*" + (2) non-(cheaply-)moveable type.

```
auto lock = lock_guard<mutex>{ m };    // error, not movable
auto ai = atomic<int>{};               // error, not movable
auto a = array<int,50>{};              // compiles, but needlessly expensive
```

- **Non-cases:** Naked init list, proxy type, multi-word name.

```
auto x = { 1 };                         // initializer_list
auto x = 1;                             // int

auto a = matrix{…}, b = matrix{…};      // some lazily evaluated type
auto ab = a * b;                        // capture proxy (efficient by default)
auto c = matrix{ a * b };               // resolve computation

auto x = (long long){ 42 };             // use  int64_t{42}  or  42LL
auto y = class X{1,2,3};                // use  X{1,2,3};
```

Cases Where You Can't … Are Few

- A recent time I resisted using *auto*, I was wrong.
  - It came up when changing this legacy code:

    base* pb = new **derived**();

    to this modern code, where I and others kept not noticing the different types:

    unique_ptr<**base**> pb = make_unique<**derived**>();
    // too subtle: people keep not seeing it

    and now I actually do prefer the consistent and nearly-as-terse spelling:

    auto pb = unique_ptr<**base**>{ make_unique<**derived**>() };
    // explicit and clear: hard to miss it

    which makes what's going on nice and explicit – the conversion is more obvious because we're explicitly asking for it.

Prefer declaring local variables using ***auto***, whether the type should (1) track or (2) stick.

1. Deduced and exact, when you want tracking: **auto x = *init*;**
2. With explicit type name, when you want to commit: **auto x = *Type* { *init* };**

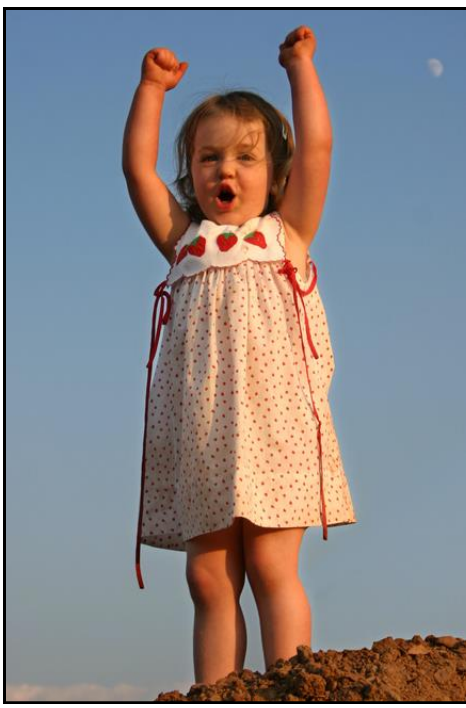
Note: Guarantees zero implicit conversions/temporaries, zero narrowing conversions, and zero uninitialized variables!

Consider having some functions in headers (e.g., templates, inlines), return ***auto*** (only):
One-liners, and wrappers that should track type

They're in headers anyway. (Insert *de rigueur* modules note here.)

C++14 makes it it convenient to not to not repeat yourself.

Remember: *auto* only ⇒ exact type, no conversions;
explicit return type ⇒ stable type, committed.



Remember, it's *really simple*

To make type <u>track</u>, <u>deduce</u>:

auto *var = init;*
auto *f() { … }*

To make type <u>stick</u>, <u>commit</u>:

auto var = ***type****{ init };*
auto f() -> ***type****;*
or
type *var{ init };*
type *f();*

Use return-by-value way more often.

BUT: Don't overuse pass-by-value.

Complete "how to pass params" details follow, but the summary fits on a slide…

… one slide for "default," one slide for "optimal"

Observation

"New features get overused." – B. Stroustrup

or

"It's about the lvalues, after all!" – S. Meyers

Just as exception safety isn't all about writing try and catch, using move semantics isn't all about writing move and &&

Up Front: Acknowledgments & Hat Tips

- The following is the result of recent discussions with many people, including but not limited to the following:
  - Gabriel Dos Reis
  - **Matthew Fiovarante (&& param ≡ move from)**
  - **Howard Hinnant (distinguish copy ctor/op= costs vs. move)**
  - **Stephan T. Lavavej (low cost of value return even in C++98)**
  - **Scott Meyers (reduce #objects, be aware of costs)**
  - Eric Niebler
  - Sean Parent
  - **Bjarne Stroustrup (practicality, judgment, design sense)**
  - VC++ MVP discussion list
  - *& many more*

C++98: Reasonable Default Advice

	Cheap to copy (e.g., int)	**Moderate cost to copy** (e.g., string, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to copy** (e.g., vector, BigPOD**[]**)
Out	X f()		f(X&) *
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy			

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

** or return X* at the cost of a dynamic allocation*

Modern C++: Reasonable Default Advice

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		f(X&) *
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

Summary of what's new in C++1x:

✓ Defaults work better

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

Using the Advanced Knobs, Optimal

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move	
In & move from	f(X&&)		

+1 consistency: same optimization guidance as overloaded copy+move construction and assignment

Summary of what's new in C++1x:

✓ Defaults work better

✓ + More optimization opportunities

Using the Advanced Knobs, Optimal

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		f(X&) *
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

Summary of what's new in C++1x:

✓ Defaults work better

✓ + More optimization opportunities

When do I write rvalue &&? Only to optimize rvalues

Just as exception safety isn't all about writing try and catch,
using move semantics isn't all about writing move and &&

Modern C++: A Narrowly Useful Option

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		f(X&)
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		**f(X)** & move ? *	
In & move from			

* *GOOD: this* ***can be faster than C++98*** *– can move from rvalues;*

BUT: also ***can be much slower than C++98*** *– always incurs a full copy, prevents reusing buffers/state (e.g., for vectors & long strings, incurs memory allocation 100% of the time)*

BUT: also problematic for noexcept

Journeyman Example: set_name

- Consider:

```
class employee {
   std::string name_;
public:
   void set_name( /*… ?? …*/ );  // change name_ to new value
};
```

- Q: What should we tell people to write here?
  - Hint: There has been a lot of **overthinking** going on about this. (I include myself.)

Option #1: Default (same as C++98)

- Default: **const string&**

```
class employee {
   std::string name_;
public:
   void set_name( const std::string& name ) { name_ = name; }
};
```

- Always 1 copy assignment – *but usually <<50% will alloc*
  - If small (SSO), ~5 int copies, no mem alloc – often dominant
  - If large, still performs mem alloc <50% of the time

Option #2: Optimized (new for C++11)

- If optimization justified: Add overload for **string&&** + move

```
class employee {
   std::string name_;
public:
   void set_name( const std::string& name ) { name_ = name; }
   void set_name( std::string&& name ) noexcept
                                          { name_ = std::move(name); }
};
```

- Optimized to steal from rvalues:
  - Pass a named object: 1 copy assignment (<<50% alloc), as before
  - Pass a temporary: 1 move assignment (~5 ints, no alloc → **noexcept**)
  - Note: Combinatorial if multiple “in + retain copy” parameters.

Option #3: Pass by Value?

- Another new option in C++11: **string** + move

```
class employee {
   std::string name_;
public:
   void set_name( std::string name ) noexcept
                                       { name_ = std::move(name); }
};
```

- Optimized to steal from rvalues, without overloading:
  - Pass named object: 1 copy *construction* (100% alloc *if long*) + move op=
  - Pass a temporary: 1 move assignment (~5 ints, no alloc → **noexcept**-ish)
  - This "noexcept" is… *problematic*

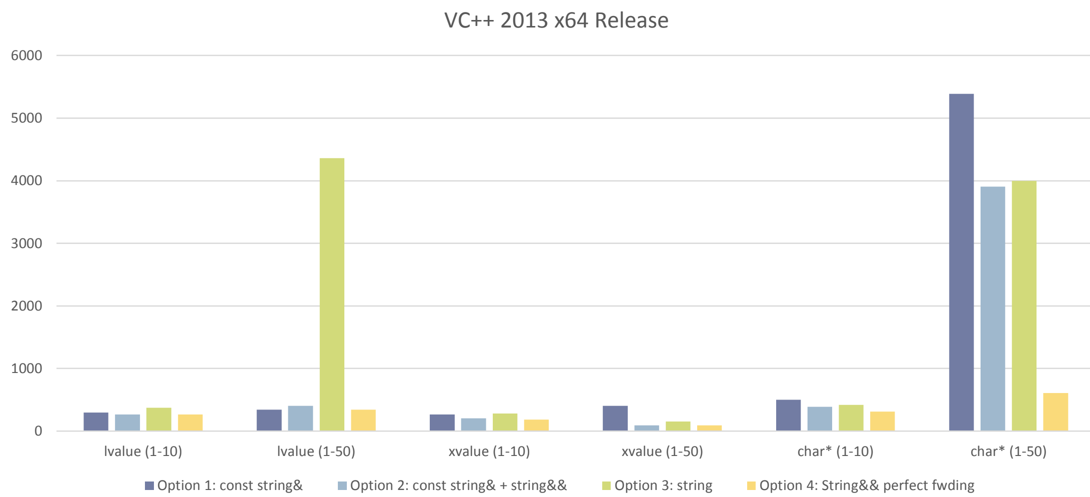
Option #4: Perfect Forwarding Idiom

- Still another new option in C++11: **Templated T&&** "perfect forwarding"

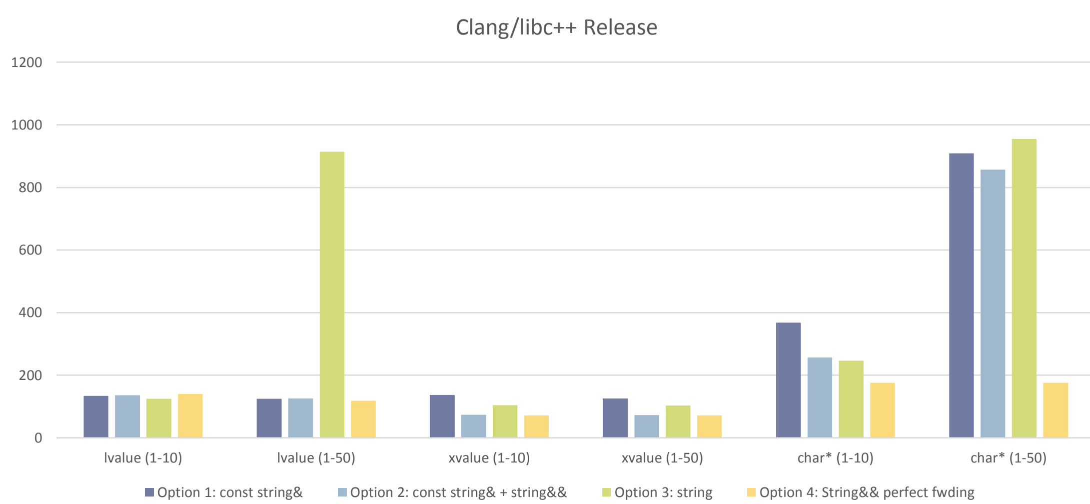
```
class employee {
   std::string name_;
public:
   template<class String, class = std::enable_if_t<!std::is_same<std::decay_t<String>,
                                                        std::string>::value>>
   void set_name( String&& name )
                      noexcept(std::is_nothrow_assignable<std::string&, String>::value)
         { name_ = std::forward<String>(name); }
};
```

- Optimized to steal from rvalues (and more), sort of without overloading:
  - Pass a named object: 1 copy assignment (<<50% alloc), as before
  - Pass a temporary: 1 move assignment (~5 ints, no alloc → **noexcept**)
  - "Unteachable!" Generates many funcs. Must be in a header. Can't be virtual.

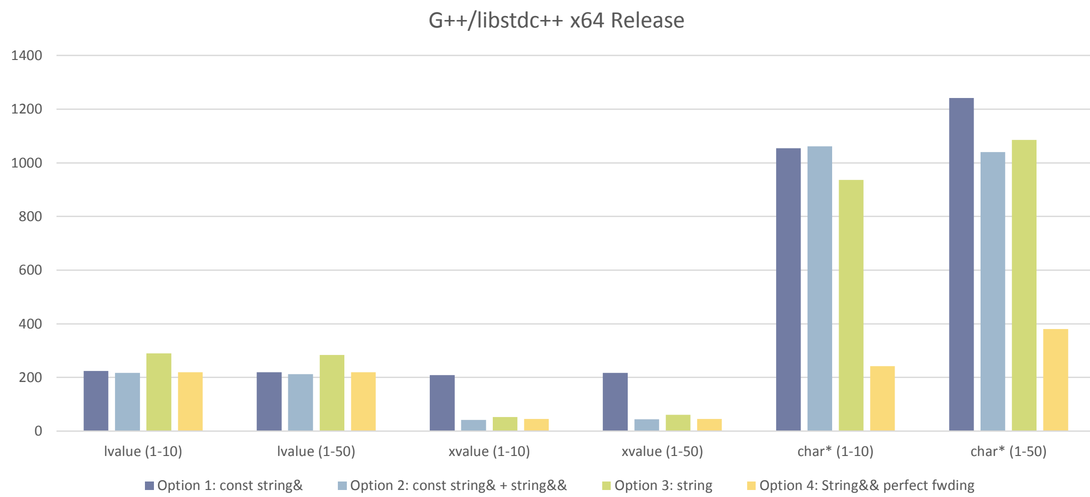
Visual C++



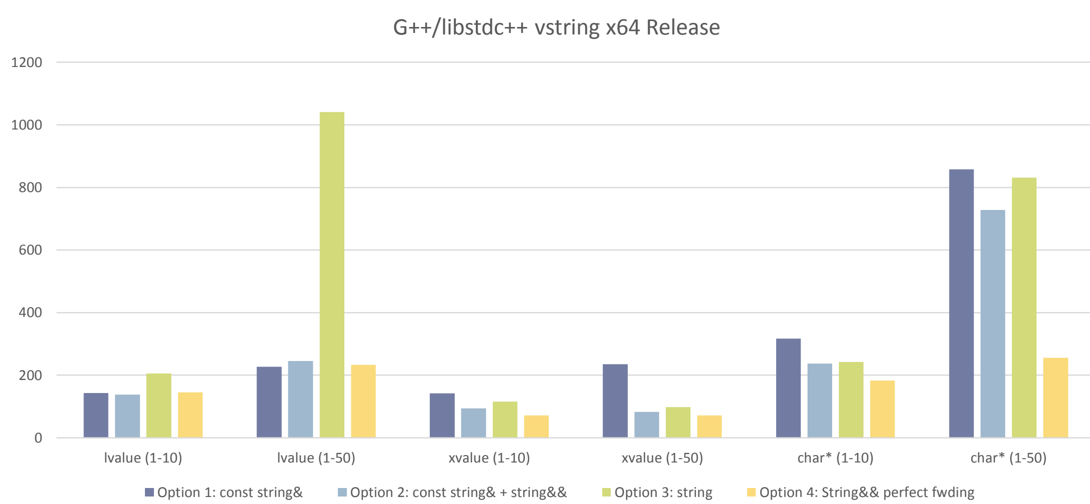
Clang libc++



gcc libstdc++



gcc libstdc++ (coming soon)

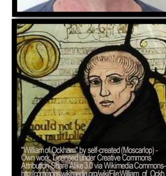


vector & large string SMFs: Rough Costs

	Constructor	operator=
Default	$$	
Move	$$	$
Copy	$$$$	$$$

(More) Geek Heroes

- Howard Hinnant: **"Don't blindly assume that the cost of construction is the same as assignment."**
  - For strings and vectors, "Capacity plays a large role in their performance. Copy construction always allocates (except for short). Copy assignment (except for short) allocates/deallocates 50% of the time with random capacities on the lhs and rhs. To keep an eye on performance, one must count allocations and deallocations."
- William of Occam: **'Do not multiply entities needlessly.'**
  - Attributed. Talking about hypotheses; applies to 'entities.'
- Andrei Alexandrescu: **"No work is less work than some work."**
- Scott Meyers: **'It's a bad habit to just create extra objects.'**
  - "Just create 'em because they're cheap to move from" is thoughtcrime.



This Talk

REPRISE

- This talk focuses on **defaults**, basic styles and idioms in modern C++.
  - ✗ ~~"Default" != "don't think."~~
  - ✓ "Default" == "don't **overthink**." Esp. don't **optimize prematurely**.

- These reinforce (not compete with) the "fundamentals."
  - **"Write for clarity and correctness first."**
  - **"Avoid premature optimization."** By default, prefer *clear* over *optimal*.
  - **"Avoid premature pessimization."** Prefer *faster* when *equally* *clear*.

Option #3: Pass by Value?

- Another new option in C++11: **string** + move

```
class employee {
   std::string name_;
public:
   void set_name( std::string name ) noexcept
                                    { name_ = std::move(name); }
```

…overloading:

…0% alloc *if long*) + move op=

…ints, no alloc → **noexcept**-ish)

An interesting attempt that temporarily drew in a number of experts!
But: at least "too cute" & probably just an antipattern… except for one case…

Option #3: Pass by Value *for Constructors*

- There is one place where this is a good idea: **Constructors**.

```
class employee {
   std::string name_;
   std::string addr_;
   std::string city_;
public:
   void employee( std::string name, std::string addr, std::string city )
      : name_{std::move(name)}, addr_{std::move(addr)}, city_{std::move(city)} { }
};
```

- Constructors are the primary case of **multiple** “in + retain copy” params, where overloading const&/&& is combinatorial.
- Constructors always construct, so no worries about **reusing** existing capacity.
- Note: Probably prefer not to write the misleading “noexpect”…

Option #1: **Default** (same as C++98)

- Default: **const string&**

```
class employee {
   std::string name_;
public:
   void set_name( const std::string& name ) { name_ = name; }
};
```

- Always 1 copy assignment – *but usually <<50% will alloc*
  - If small (SSO), ~5 int copies, no mem alloc – often dominant
  - If large, still performs mem alloc <50% of the time

Option #2: **Optimized** (new for C++11)

- **If optimization justified:** Add overload for **string&&** + move

```
class employee {
   std::string name_;
public:
   void set_name( const std::string& name ) { name_ = name; }
   void set_name( std::string&& name ) noexcept
                                          { name_ = std::move(name); }
};
```

- Optimized to steal from rvalues:
  - Pass a named object: 1 copy assignment (<<50% alloc), as before
  - Pass a temporary: 1 move assignment (~5 ints, no alloc → **noexcept**)
  - Note: Combinatorial if multiple "in + retain copy" parameters.

What is a T&&? A forwarding reference

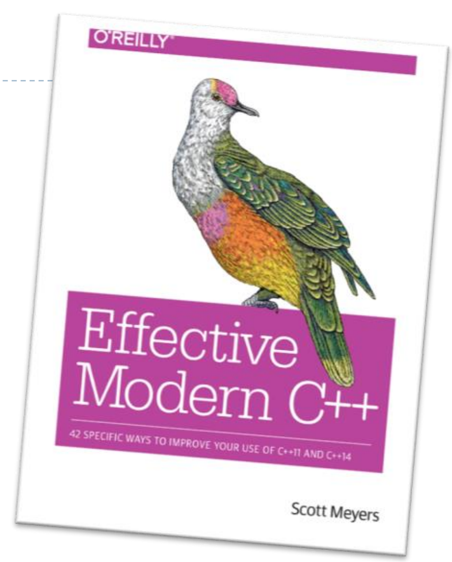
Quiz

```
void foo( X&& x );
template<class Y>
void bar( Y&& y );
```

- **Q: What are the types of the function parameters?
  What arguments to they accept or reject?
  What is the parameter for?**
- A: Fundamentally different.
  - *foo* takes **rvalue** reference to **non-const**.
    foo accepts **only rvalue** X objects.
    foo's parameter is to **capture temporaries** (and other rvalues).
  - *bar* takes *mumble* reference to **everything**: const, volatile, both, **and** neither.
    bar accepts **all** Y objects.
    bar's parameter is for **forwarding** its argument onward.

Forwarding References

- Scott Meyers pointed out that T&& is very different, and needs a name.
  - He coined "universal reference."
  - For his book whose final galleys are due, um, today.
- Here at CppCon, a few of us met and ultimately agreed that this does need a name. (Thanks, Scott.)
  - But we still disliked "universal." (Sorry, Scott.)
  - We think the right name is **"forwarding reference."**
  - The committee/community may disagree. Time will tell.
  - In the meantime, Scott will add a footnote and index entry for "forwarding reference," and switch to it in future printings if the community agrees. (Thanks, Scott!)



Uses and Abuses of &&

- Use && only for parameter/return types:
  - ***myclass&&* rvalue references to optimize rvalues**, usually overloading const& / && – note this covers the move SMFs!

```
void f( const string& );     // default way to express "in + retain a copy"
void f( string&& );          // what to add to additionally optimize for rvalues
```

  - ***T&&* forwarding references to write forwarders**, which are neutral code between unknown callers and callees and want to preserve rvalueness/cv-ness.
    - Note this includes the new proposed *for(e:c)*, which is… *drum roll*… a neutral forwarder between a collection/range and the calling code.
    - Also includes generic lambda auto&& parameters… use for forwarders only.
- Don't use auto&& for local variables.
  - You should know whether your variable is const/volatile or not!
  - (Except rarely if you're just handing it off… in the body of a forwarder.)

Dessert Slide:

Use ***tuple*** for multiple return values.

Yes, C++11 has multiple return values! (Who knew?)

Sweet Realization: We're Already Doing It

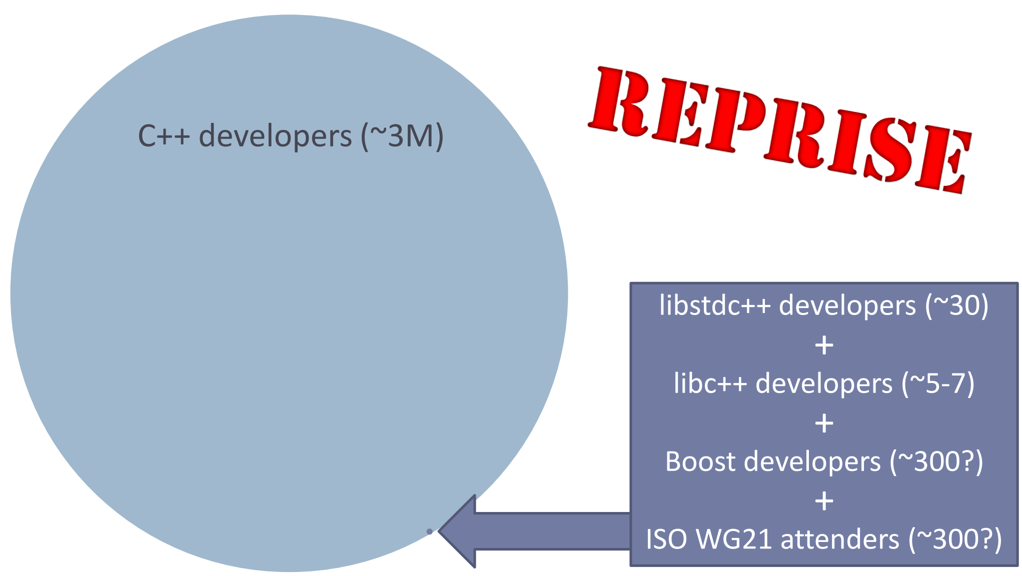
- Given a *set<string> myset*, consider:

```
// C++98
pair<set<string>::iterator,bool> result = myset.insert( "Hello" );
if (result.second) do_something_with( result.first );      // workaround

// C++11 – sweet backward compat
auto result = myset.insert( "Hello" );                      // nicer syntax, and the
if (result.second) do_something_with( result.first );      // workaround still works

// C++11 – sweet forward compat, can treat as multiple return values
tie( iter, success ) = myset.insert( "Hello" );             // normal return value
if (success) do_something_with( iter );
```

It's hard to ~~be~~ remember you're an expert



Back to Basics: Modern C++ Style

loops
pointers & references
smart pointers
variable declarations
parameter passing

Questions?