Remedial C++ 11

Now that we can actually use it…

Pete Williamson
(petewil00@hotmail.com)

Overview (1)

- auto
- lambdas
- nullptr
- = default, = delete
- shared_ptr
- Range based for loops

Overview (2)

- Uniform initialization = { , , };
- Moving objects (the move ctor) - see my other presentation on the NWCPP website.
- override
- using
- constexpr

Overview (3) - Stuff I haven’t used

- Decltype
- Threading library
- Variadic templates
- Strongly typed enums
- Delegating c’tors
- static_assert

auto

A Challenger for long type names

auto

What it does

Figure out what type to use for a variable

Example

```
for (auto& item : items) { … }
```

auto - why does it help?

```
std::vector<NetworkPacket> packets;

const auto& bar = packets.rbegin();
```

-vs-

```
const reverse_iterator<
    std::vector<NetworkPacket>>& bar =
  packets.begin();
```

auto

When to use it

- You need to know what type it makes
- To make the code easier to read, instead of easier to type

You can use it with const and ‘&’

lambda

I rode through the desert on a function with no name...

lambdas

What they are

A way to define a function with no name.

Example (no capture)

```
auto glambda = [ ](auto a, auto&& b) {
    return a < b;
};
bool b = glambda(3, 3.14);
```

Lambda example (capture)

```
int i = 3; int j = 5;
function<int (void)> f = [i, &j] {
  return i + j;
};
cout << f() << endl;
```

Lambdas

When to use

- Writing async code inline
- Passing comparators / find criteria
- Link to Herb's presentation from 2011.
  https://vimeo.com/23975522

Lambdas - comparator example

```
std::sort(page_infos->begin(),
          page_infos->end(),
          [](const PageInfo& a, const
             PageInfo& b) -> bool {
              return a.last_access_time >
                     b.last_access_time;
            });
```

Lambda - async completion ex

```
void RequestQueueTaskTestBase::InitializeStore() {
  store_.Initialize(base::BindOnce([](bool success) {
    ASSERT_TRUE(success); }));
  PumpLoop();
}
```

Lambdas

When not to use them

- Careful, can make code harder to read instead of easier if overused.
- Personally, I think that if it needs a function, a name helps. You may disagree.

Lambda bad example (from java)

```
public static boolean saveAndSharePage(final Activity activity, Tab tab, final Callback<ShareParams> shareCallback) {
    ...
     offlinePageBridge.getPagesByNamespace(OfflinePageBridge.LIVE_PAGE_SHARING_NAMESPACE,
            new Callback<List<OfflinePageItem>>() {
                @Override public void onResult(List<OfflinePageItem> items) {
                    offlinePageBridge.savePage(webContents,
                            new ClientId(OfflinePageBridge.LIVE_PAGE_SHARING_NAMESPACE, Integer.toString(tab.getId())),
                            new OfflinePageBridge.SavePageCallback() {
                                @Override public void onSavePageDone(int savePageResult, String url, long offlineId) {
                                    offlinePageBridge.getPageByOfflineId(
                                            offlineId, new Callback<OfflinePageItem>() {
                                                @Override public void onResult(OfflinePageItem page) {
                                                    sharePublishedPage(page, activity, shareCallback);
                                                }
                                            });
                                }
                            });
                }
            });
```

nullptr

< no picture here, nothing to see, move along now >

nullptr

What it does

Replaces NULL with a typed pointer.

Example

```
vector<Item>* items = nullptr;
```

When to use it

Always, everywhere you would have used NULL. It's just better.

= default, = delete

What it does

- Explicitly state that you rely on the default constructor, destructor, or assignment (with default)
- explicitly remove the default ctor, dtor, or assignment operator (with ‘delete’).

=default, = delete

Example

```
- MyUncopyableClass(
      MyUncopyableClass& other) =
          delete;
- MyNormalClass() = default;
```

=default, =disabled

When to use it

We always use it whenever we want default behavior.

unique_ptr

unique_ptr

What it does

RAII (not RIAA). Pointer scoping to prevent memory leaks

When to use it

All heap allocations (with some exceptions)

unique_ptr example

Example 1:

```
{
  std::unique_ptr<Foo> foo =
    std::make_unique<Foo>;
    // use foo
    ...
}  // <- foo gets deleted here.
```

unique_ptr example

Example 2:

```
std::unique_ptr<Foo> foo_ptr(new Foo());


int MyFunction(std::unique_ptr<Foo>) { …
}
// takes ownership of foo, called as:
int bar = MyFunction(std::move(foo_ptr));
```

shared_ptr

shared_ptr

What it does

- Reference counted smart pointer

shared_ptr example 1 (use)

```
void AsyncCompileJob::AsyncCompileFailed(Handle<Object> error_reason)
{
  // {job} keeps the {this} pointer alive.
  std::shared_ptr<AsyncCompileJob> job =
      isolate_->wasm_engine()->RemoveCompileJob(this);
  resolver_->OnCompilationFailed(error_reason);
}
```

shared_ptr example 2 (decl)

```
 // Abstraction over the storage of the wire bytes. Held in a
shared_ptr so
  // that background compilation jobs can keep the storage alive while
  // compiling.
  std::shared_ptr<WireBytesStorage> wire_bytes_storage_;
```

shared_ptr example 3 (accessors)

```
// Using shared pointers with functions

 void SetWireBytesStorage(
      std::shared_ptr<WireBytesStorage> wire_bytes_storage) {
    base::MutexGuard guard(&mutex_);
    wire_bytes_storage_ = wire_bytes_storage;
  }

  std::shared_ptr<WireBytesStorage> GetWireBytesStorage() const {
    base::MutexGuard guard(&mutex_);
    return wire_bytes_storage_;
  }
```

From https://cs.chromium.org/chromium/src/v8/src/wasm/module-compiler.cc

Shared_ptr example 4 (use)

```
// Start the code section.
bool AsyncStreamingProcessor::ProcessCodeSectionHeader(
    size_t functions_count, uint32_t offset,
    std::shared_ptr<WireBytesStorage> wire_bytes_storage) {
...
  job_->native_module_->compilation_state()->SetWireBytesStorage(
      std::move(wire_bytes_storage) ); …
}
```

shared_ptr example 5 (pointed at)

```
// What is being pointed at
class WireBytesStorage {
 public:
  virtual ~WireBytesStorage() = default;
  virtual Vector<const uint8_t> GetCode(WireBytesRef) const = 0;
};
```

When to use shared_ptr

When to use it

We avoid it whenever possible, easier to reason about unique pointers.

Multithreading: you can give each thread or callback a shared pointer to keep object alive (assuming you do proper thread mutexes on refcount changes).

Range based for loops

For people who are home on the range

Range based for loops

What it does

An easier to read syntax for for loops

Example

```
for (auto& item : container) {…}
```

When to use it

This is good, use it always.

Uniform initialization

Yes, Captain Tyler Sir, my uniform is properly initialized.

Uniform Initialization - with { }

What it does

A way to initialize variables, objects, and arrays.

Uniform Initialization - with { }

Why we need it: C++ before it is inconsistent

```
// ok: initialize array variable
string a[] = { "foo", " bar" };
// error: initializer list for non-aggregate vector
vector<string> v = { "foo", " bar" };

void f(string a[]);
// syntax error: block as argument
f( { "foo", " bar" } );
```

Uniform Initialization - examples

```
int* pi = new int[5]{ 1, 2, 3, 4, 5 };  // Dynamic.
int arr[]           { 1,2,3,4,5 };
std::vector<int> v { 1,2,3,4,5 };
std::set<int> s    { 1,2,3,4,5 };
std::map<int,std::string> m { {0,"zero"}, {1,"one"},
{2,"two"} };

int total = totalElementsInVector({10,20,30,40});
```

Uniform Initialization - examples

```
X x1 = X{1,2};
X x2 = {1,2};  // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
     D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
     int a[3];
     S(int x, int y, int z) :a{x,y,z} { /* ... */ }; // solution to old problem
};
```

Uniform Initialization - with { }

When to use it

Array elements:

```
std::vector<int> v{ 10, 20, 30 };
```

Objects:

```
X x1 { “string”, 4.0 };
```

Uniform Initialization - with { }

When not to use it

Sometimes it doesn't do what you expect.

A template may not know whether the argument inside curlies refers to a value or a number of elements.

https://probablydance.com/2013/02/02/the-problems-with-uniform-initialization/

Uniform Initialization - surprises

```
template<typename T>
std::vector<T> create_ten_elements()
{
    return std::vector<T>{10};
}

int main()
{
    create_ten_elements<std::string>(); // create ten elements
    create_ten_elements<int>(); // create one element
    create_ten_elements<Widget>(); // Creates one element (for some Widget defn)
    create_ten_elements<char>(); // create one element
    create_ten_elements<std::vector<int>>(); // create ten elements
}
```

Uniform Initialization - when

Google style guide allows it.

Bad:

Be careful when using a braced initialization list {...} on a type with an std::initializer_list constructor.

Uniform Initialization - when

Good:

The brace form prevents narrowing of integral types. This can prevent some types of programming errors.

```
int pi(3.14);  // OK -- pi == 3.
int pi{3.14};  // Compile error:
narrowing conversion.
```

Moving instead of copying

Moving instead of copying

What it does

Allow you to prevent copying as you move through layers - make something only once, and pass it up through the layers.

For example, the windows networking stack wants to prevent copying at every interface.

Moving instead of copying

Example

```
crunch_data(std::move(my_data));
```

When to use it

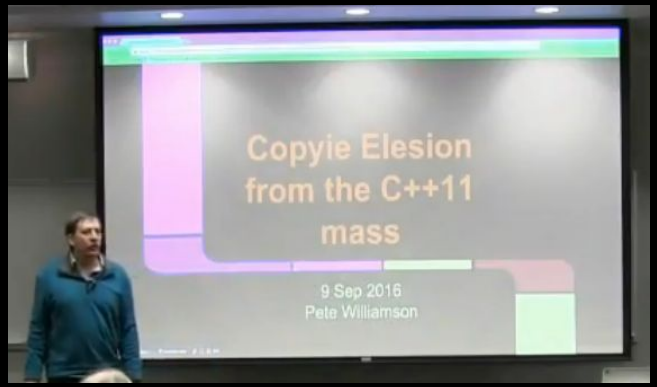
We try to use it as much as possible.

Also good for functional programming.

Moving instead of copying

Presentation:

https://www.youtube.com/watch?v=KcBmR05DU7o&feature=youtu.be



override

override keyword

What it does

Make explicit that we intend to override a virtual function, so we can get compile warnings if we messed up somewhere.

override example

In the base class .h file:

```
class Task
{  public: virtual void Run() = 0; }
```

In the derived class .h file:

```
class TestTask : public Task {
 public:
  void Run() override;
}
```

override keyword

When to use it

Good idea, we always use it on the chromium project.

constexpr

What it does

- Implies const
- Lets compiler know we can evaluate this function or variable at compile time

constexpr

Examples

```
static constexpr size_t kVendorNameSize =
    3 * sizeof(cpu_info[1]);
V8_INLINE static constexpr int  SizeFor(
     int length) {
         return kHeaderSize + length *
                kTaggedSize;
}
```

constexpr

When to use it

Use constexpr to specify true constants and the functions that support their definitions. Avoid complexifying function definitions to enable their use with constexpr. Do not use constexpr to force inlining.

using

What it does

Sets up an alias for a more complicated typename, similar to typedef.

using - example

```
namespace mynamespace {
// Used to store field measurements. DataPoint may
// change from Bar* to some internal type.
// Client code should treat it as an opaque pointer.
using DataPoint = foo::Bar*;

// A set of measurements. Just an alias for user
// convenience.
using TimeSeries = std::unordered_set<DataPoint,
std::hash<DataPoint>, DataPointComparator>;
}  // namespace mynamespace
```

using

When to use it

Use with classes, not with entire namespaces.
std::string is OK, std:: is not.

using vs typedef:

In new code, using is preferable to typedef, because it provides a more consistent syntax with the rest of C++ and works with templates.

Warning - stuff I haven’t used yet.

Although I can tell you how it works, I don’t have personally based wisdom on how and when to use them.

decltype

What it is

An expression to return the type of its argument without evaluating it (like sizeof).

Decltype

Example (with typedef/using)

```
typedef decltype(&nvmlInit) INITPROC;


using Unwrapped =
decltype(Unwrap(std::declval<ForwardType>
()));
```

Decltype

Example

```
template <class T>
  const decltype(T::list_.get())
      list(const T& iter) {
    return iter.list_.get();
  }
```

decltype

When to use it

Sometimes when writing templates, we need to know the type of something, and can’t just use “T”.

Thread Support Library

Some nice threads...

Thread Support Library

What it does

Supports writing multithreaded and async programs at the C++ level. There's enough here for a lecture on its own.

Thread class, Mutexes, futures, etc.

Thread Support Library - example

```
#include <iostream>
#include <thread>
void call_from_thread() {
    std::cout << "Hello, World" << std::endl;
}

int main() {
    //Launch a thread
    std::thread t1(call_from_thread);
    //Join the thread with the main thread
    t1.join();
    return 0;
}
```

Thread Support Library

When to use it

We don’t use it in Chromium because we have an older alternative that is already everywhere in the code base.

It is more a bit simpler than Posix, and can be portable.

New containers: unordered

New containers: unordered

What it does

Like a hash map but with no ordering, good for sets.

- unordered_map
- unordered_set
- unordered_multimap
- unordered_multiset

unordered_set - example (1)

```
#include <iostream>
#include <unordered_set>
#include <algorithm>

int main() {

    // Creating an Unoredered_set of type string
    std::unordered_set<std::string> setOfStrs;

    // Insert strings to the set
    setOfStrs.insert("first");
    setOfStrs.insert("second");
    setOfStrs.insert("third");
```

Unordered_set example (2)

```
    // Try to Insert a duplicate string in set
    setOfStrs.insert("second");

    // Iterate Over the Unordered Set and display it
    for (std::string s : setOfStrs)
        std::cout << s << std::endl;

}
```

Output:

```
third
second
first
```

Unordered

When to use it

These containers maintain average constant-time complexity for search, insert, and remove operations. In order to achieve constant-time complexity, they sacrifice order for speed by hashing elements into buckets.

Variadic templates - what it does

What it does

Lets you make templates with any number of arguments.

Variadic Templates - why we need

```
#define TYPELIST_1(T1) \
    ::Loki::Typelist<T1, ::Loki::NullType>

#define TYPELIST_2(T1, T2) \
    ::Loki::Typelist<T1, TYPELIST_1(T2) >

#define TYPELIST_3(T1, T2, T3) \
    ::Loki::Typelist<T1, TYPELIST_2(T2, T3) >

#define TYPELIST_4(T1, T2, T3, T4) \
    ::Loki::Typelist<T1, TYPELIST_3(T2, T3, T4) >
```

Variadic Templates - example

```
template<typename T, typename... Args>
   T adder(T first, Args... args) {
      return first + adder(args...);
}

long sum = adder(1, 2, 3, 8, 7);
```

Variadic templates - when to use

When to use it

Remember we used to have to do this by making one overload for every possible number of args? Modern C++ design's loki library had to do that.

Strongly typed enums

What it does

As it says, adding strong typing to enums

Enums - the old way

```
enum Selection
{
    None,
    Single,
    Multiple
};

Selection sel = Single;
```

Strongly typed enums - example

```
enum class Selection
{
    None,
    Single,
    Multiple,
};
Selection s = Selection::Multiple;
```

Strongly typed enums

When to use it

Using these is more type safe, and they don't get converted to int automatically. Chromium uses them.

Also, you can specify the underlying type now in both kinds:

```
enum class Selection : unsigned char
enum Selection : unsigned char
```

Delegating constructors

Delegating constructors

What it does

Lets one constructor call another ctor to do the work, then fixup the state.

Delegating Constructors - example

```
A() {
    x = 0;
    y = 0;
    z = 0;
}

// Constructor delegation
A(int z) : A()  {
    this->z = z; // Only update z
}
```

Delegating constructors

When to use them

For classes with const members, using an Init() method to init the members is impossible.

Even for non-const members, most classes' constructors need to do simple work "call[ing] virtual functions or attempt[ing] to raise non-fatal failures" -- and delegated constructors eliminate copies of that boilerplate.

static_assert

<static electricity in action - Imagine a photo of a balloon with hair here>

static_assert

- What it does
  - Compile time asserts. Break at compile time if something is wrong, always easier than catching at runtime

static_assert example

```
static_assert(
    param_is_forwardable ||
    !std::is_constructible<Param,
      std::decay_t<Unwrapped>&&>::value,
    "Bound argument |i| is move-only but”
    “will be forwarded by copy. "
    "Ensure |Arg| is bound using”
    “base::Passed(), not std::move().");
```

static_assert

When to use

Very helpful deep inside libraries to check assumptions and communicate to programmer what is wrong.

New library additions

Lots of additions to the std lib not covered here, we're concentrating on language features tonight.

https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP11.md#c11-library-features

Library additions:

std::move
std::forward
std::thread
std::to_string
std::tie
std::weak_ptr
std::chrono

std::array
std::async
Memory Model
Type traits
Tuples
Smart ptr
Unordered containers

Links (1)

The Biggest Changes in C++11 (and Why You Should Care)

https://smartbear.com/blog/develop/the-biggest-changes-in-c11-and-why-you-should-care/

The 15 C++11 features you must really use in your C++ projects.

http://cppdepend.com/blog/?p=319

Links (2)

Complete list of features:

https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP11.md

Features for other new C++ versions (stay tuned)

https://github.com/AnthonyCalandra/modern-cpp-features

Links (3)

Official C++ 11 website:

https://isocpp.org/wiki/faq/cpp11

Chromium style guide for C++ 11

https://chromium-cpp.appspot.com/

Links (4)

CPP Reference:

https://en.cppreference.com/w/

Thanks for Listening!

Let me know if you are interested in Remedial C++ 14 or 17, or the library additions to C++11.