A Quick Peek at C++11 & 14

Rex Kerr

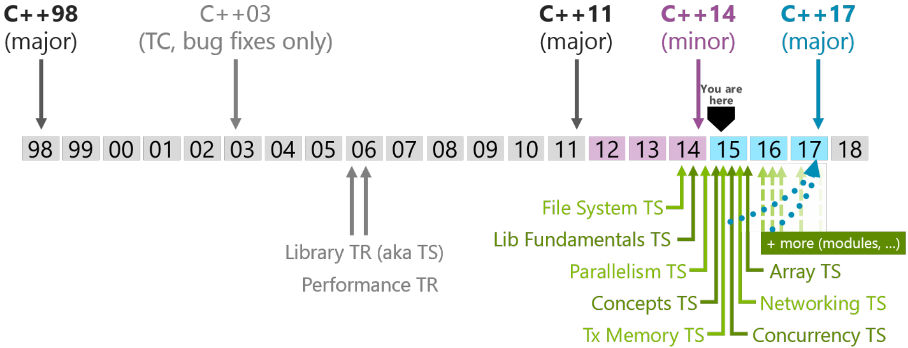
rk-logix, inc.

October 3, 2015

C++11 feels like a new language.

– Bjarne Stroustrup

Some History



https://isocpp.org/std/status

A Quick Preview

Before:

```
std::map<std::string,
    std::vector<std::auto_ptr<std::pair<int, float> > > > m;
/*...*/
for( std::map<std::string,
    std::vector<std::auto_ptr<std::pair<int, float> > >
    >::iterator it = m.begin(); it != m.end(); ++it )
{
   // use *it
}
```

After:

```
std::map<std::string,
    std::vector<std::unique_ptr<std::pair<int, float>>>> m;
/*...*/
for( const auto& v : m )
{
   // use v
}
```

Another Preview

Before:

```
std::vector<std::string> vs;
vs.push_back("Hello, ");
vs.push_back("my name ");
vs.push_back("is Rex.");

std::cout << std::accumulate(vs.cbegin(), vs.cend(),
    std::string("CPP03: ")) << std::endl;
```

After:

```
auto strings = { "Hello, ", "my ", "name ", "is ", "Rex." };

using namespace std::literals;
std::cout << std::accumulate(cbegin(strings), cend(strings),
    "CPP14: "s) << std::endl;
```

RAII

Question: Are we all familiar with the RAII idiom?

> A modern C++ programmer should (*almost*) never use
> `operator new` nor `operator delete`.

Is this controversial or surprising?

Smart Pointers

C++11 deprecated `std::auto_ptr` in favor of new smart pointers.

- `std::shared_ptr`
- `std::weak_ptr`
- `std::unique_ptr`

std::auto_ptr

Question: What was wrong with `std::auto_ptr`? Why was it deprecated and replaced?

- std::shared_ptr is intended to be used when there is ***shared*** ownership of an object.
- std::enable_shared_from_this mixin is useful for providing pointers to self

Tip: Think in terms of ownership and lifetime semantics. Don't think of std::shared_ptr as C++'s garbage collection. std::shared_ptr is not the "big hammer" for use on all pointer screws.

Creating `std::shared_ptr` owned objects

There is a factory function for creating `std::shared_ptr` objects:

```
// preferred
auto ptr = std::make_shared<foo>(1,2,3);

// avoid
std::shared_ptr<foo> ptr(new foo(1,2,3));
```

Using the factory has multiple benefits over raw new:

- Exception Safety
- Performance
  - WKWYL optimization
    - one allocation *vs.* two
    - cache locality

Smart Pointers: std::weak_ptr

std::weak_ptr is a non-owning 'weak' reference to an object owned by a std::shared_ptr.

```
std::weak_ptr<int> wp;
{
    auto sp = std::make_shared<int>(42);

    wp = sp;
    auto inner_sp = wp.lock();
    assert( !wp.expired() && inner_sp &&
                               "both wp & inner_sp are valid" );
}
auto outer_sp = wp.lock();
assert( wp.expired() && !outer_sp &&
                               "both wp & outer_sp are invalid" );
```

- std::weak_ptr is useful for tracking the lifetime of an object owned by a std::shared_ptr without affecting its lifetime
- std::weak_ptr helps to break cycles

Good, it's not expired!

```
    if(!wp_foo.expired())
    {
        auto sp_foo = wp_foo.lock();

        sp_foo->do_something()
    }
```

Comments?

Good, it's not expired!

```
if(!wp_foo.expired())
{
    auto sp_foo = wp_foo.lock();

    sp_foo->do_something()
}
```

Comments?

This is NOT thread-safe! Just lock it and check the pointer:

```
auto sp_foo = wp_foo.lock();

if(sp_foo)
{
    sp_foo->do_something()
}
```

Be careful with `std::weak_ptr`

WARNING

When using `std::make_shared`, long lived `std::weak_ptr` objects can prevent deallocation of the memory block.
(the destructor is still run deterministically when the last `std::shared_ptr` goes out of scope)

WARNING

`std::make_shared` can hurt performance by introducing *false sharing*.

The lesson here is that you should be aware of how `std::make_shared` works and aware of your usage patterns and choose appropriately.

`std::shared_ptr`, not just for pointers!

You can use `std::shared_ptr` even with non-pointer types that require a special function to destroy them.

```
{
    std::shared_ptr<lib::handle_t> ctx(lib::get_context(),
        &lib::release_context);

    // use ctx
    ctx->do_something();

    // lib::ReleaseContext(ctx) is called when exiting scope
}
```

std::shared_ptr, not just for pointers!

rk·logix quality software

You can use `std::shared_ptr` even with non-pointer types that require a special function to destroy them.

```
{
    std::shared_ptr<lib::handle_t> ctx(lib::get_context(),
        &lib::release_context);

    // use ctx
    ctx->do_something();

    // lib::ReleaseContext(ctx) is called when exiting scope
}
```

Unfortunately you cannot specify a custom deleter when using `std::make_shared`

You can use std::shared_ptr to ensure that something happens on scope exit.

```
{
   std::shared_ptr<void> at_exit(nullptr, [](auto)
      {
         std::cout << "Exiting scope..." << std::endl;
      });

   std::cout << "Running stuff in scope...\n";
}
```

You can use `std::shared_ptr` to ensure that something happens on scope exit.

```
{
   std::shared_ptr<void> at_exit(nullptr, [](auto)
      {
         std::cout << "Exiting scope..." << std::endl;
      });

   std::cout << "Running stuff in scope...\n";
}
```

Output:

```
Running stuff in scope...
Exiting scope...
```

Here Be NULL Dragons

What does the following code print?

```
void foo(long)  { std::cout << "long" << std::endl; }
void foo(long*) { std::cout << "ptr"  << std::endl; }

int main() {
   long l = 42;
   long* pl = &l;

   foo(l);
   foo(pl);
   foo(NULL);
}
```

Here Be NULL Dragons

What does the following code print?

```
void foo(long)  { std::cout << "long" << std::endl; }
void foo(long*) { std::cout << "ptr"  << std::endl; }

int main() {
   long l = 42;
   long* pl = &l;

   foo(l);
   foo(pl);
   foo(NULL);
}
```

Output:

```
long
ptr
long
```

Why?

NULL

NULL is defined as an *implementation-defined null pointer constant*, and is a macro. From sys/_types.h on my MacBook:

```
#ifdef __cplusplus
#ifdef __GNUG__
#define __DARWIN_NULL __null
#else /* ! __GNUG__ */
#ifdef __LP64__
#define __DARWIN_NULL (0L)
#else /* !__LP64__ */
#define __DARWIN_NULL 0
#endif /* __LP64__ */
#endif /* __GNUG__ */
#else /* ! __cplusplus */            // <--- !!!
#define __DARWIN_NULL ((void *)0)
#endif /* __cplusplus */
```

Use of NULL and 0 for null pointers leads to potential ambiguity, and was especially problematic for generic programming (templates).

nullptr and std::nullptr_t

C++11 provides a new `std::nullptr_t` type and `nullptr` keyword to avoid the above ambiguity.

```
    /*...*/
    foo(l);
    foo(pl);
    foo(nullptr);
}
```

Output:

```
long
ptr
ptr
```

`nullptr` must always correspond with a pointer type.

Leaks?

Question: Is it possible to leak memory when using a `std::shared_ptr`?

It is possible to create cycles that permanantly tie up resources and lead to 'leaked' memory. Consider the following:

```
struct A;
struct B;

struct A : std::enable_shared_from_this<A>
{
   A(std::shared_ptr<B> b) : b_(b) { }
   ~A() { std::cout << "...destroying A..." << std::endl; }
   std::shared_ptr<B> b_;
};

struct B : std::enable_shared_from_this<B>
{
   ~B() { std::cout << "...destroying B..." << std::endl; }
   std::shared_ptr<A> a_;
};
```

rk·logix quality software

What happens?

```
int main()
{
    {
        auto a = std::make_shared<A>( std::make_shared<B>() );

        a->b_->a_ = a->shared_from_this();

        std::cout << "...created pointers..." << std::endl;
    } // ...note the artificial scope...

    std::cout << "...left scope..." << std::endl;
}
```

What happens?

```
int main()
{
    {
        auto a = std::make_shared<A>( std::make_shared<B>() );

        a->b_->a_ = a->shared_from_this();

        std::cout << "...created pointers..." << std::endl;
    } // ...note the artificial scope...

    std::cout << "...left scope..." << std::endl;
}
```

Output:

```
...created pointers...
...left scope...
```

What happens?

```
int main()
{
    {
        auto a = std::make_shared<A>( std::make_shared<B>() );

        a->b_->a_ = a->shared_from_this();

        std::cout << "...created pointers..." << std::endl;
    } // ...note the artificial scope...

    std::cout << "...left scope..." << std::endl;
}
```

Output:

```
...created pointers...
...left scope...
```

Notice that it never said '. . . destroying A. . . ' nor '. . . destroying B. . . '

`std::unique_ptr`

`std::unique_ptr` is a non-reference counting smart pointer for use when there is no shared ownership of the data.

`std::unique_ptr` should be your goto smart pointer when possible.

- semantic correctness (say what you mean)
- no reference counting overhead

There is also a `std::make_unique` factory, but it was not added until C++14.

lambda expressions

C++11 added lambda expressions, sometimes called 'anonymous functions'. The general form is as follows:

```
[ capture-list ] ( params ) mutable exception attribute -> ret
    { body }
```

Many of the items are optional:

- capture list can be empty, but must be present
- params list can be left out in some cases
- mutable keyword if it is not mutable
- function attributes are optional
- return type can be auto-deduced in some cases

rk·logix quality software

What happens?

```cpp
auto x = 1;

auto xref_plus_y = [&](int y) { return x + y; };
auto xval_plus_y = [=](int y) { return x + y; };

std::cout << xref_plus_y( 4 ) << ' ';
std::cout << xval_plus_y( 4 ) << ' ';
x = 2;
std::cout << xref_plus_y( 4 ) << ' ';
std::cout << xval_plus_y( 4 ) << std::endl;
```

lambda example

rk·logix quality software

What happens?

```
auto x = 1;

auto xref_plus_y = [&](int y) { return x + y; };
auto xval_plus_y = [=](int y) { return x + y; };

std::cout << xref_plus_y( 4 ) << ' ';
std::cout << xval_plus_y( 4 ) << ' ';
x = 2;
std::cout << xref_plus_y( 4 ) << ' ';
std::cout << xval_plus_y( 4 ) << std::endl;
```

Output:

```
5 5 6 5
```

lamb-do & lamb-don’t

- Lambda expressions make it MUCH easier to use standard algorithms
- A local lambda is great for reducing code duplication
- Too much of a good thing can be bad

You've been a very naught lambda!

rk·logix quality software

A bad lambda example:

```
class foo {
    foo()
    {
        some_signal.connect( []( /* signal data */ )
        {  /* 46 line signal handler */
        });

        some_other_signal.connect( []( /* signal data */ )
        {  /* 18 line signal handler */
        });

        yet_another_signal.connect( []( /* signal data */ )
        {  /* 32 line signal handler */
        });
    }
    /* ... */
};
```

lambda improvements in C++14

C++14 made lambda expressions even easier to use.

- generic lambdas (auto parameter type deduction)

```
for_each(begin(v), end(v), [](auto i) { cout << i; });
```

- loosened return type deduction rules
  - C++11 : return type automatically deduced iff the body consisted of nothing but a single return statement with an expression, otherwise void.
  - C++14 : return type is deduced from return statements as if for a function whose return type is declared auto.

Note that lambda expresssions don't add any new functionality, it's merely *'syntactic sugar'* that eases the process of creating function objects.

keyword `auto`

The keyword `auto` has undergone a lot of changes:

- no longer legal as a storage class specifier (C++11)

```
void foo(auto int); // no longer legal
```

- can now be used for automatic type deduction (C++11)

```
auto        sp = std::make_shared<foo>();
auto&       tr = get_thingref();
auto const  v  = get_value();
```

- can now be used to specify trailing return types (C++11)

```
auto foo() -> bool;  // equivalent to bool foo();
```

- automatic return type deduction, even for non-lambda (C++14)

```
auto foo() { return 3+2; } // returns decltype(3+2), or int
```

- can be used for generic lambdas (C++14)

range-based `for` loops

New range-based for loop syntax makes it easier to perform an operation on each item in a collection.

```
for(range_declaration:range_expression)
    loop_statement
```

It can be used with standard containers…

```
std:vector<int> v = {0,1,2,3,4};
for( auto const i : v) { std::cout << i << ' '; }
```

…arrays

```
int a[] = {0,1,2,3,4};
for( auto const i : a) { std::cout << i << ' '; }
```

…and initializer lists *(not covered yet)*

```
for( auto const i : {0,1,2,3,4} ) { std::cout << i << ' '; }
```

non-member `std::begin/std::end`

rk·logix quality software

New non-member `std::begin` & `std::end` functions make it easier to write generic and maintainable code that doesn't care about the container type:

C++98

```
std::for_each(v.begin(), v.end(), &foo);
```

C++11

```
std::for_each(begin(v), end(v), &foo);
       // note the lack of std:: -- using ADL
```

C++14 also adds non-member `cbegin` and `cend`, which were not available in C++11.

Easier to write maintainable code?

Question: How does non-member `std::begin` & `std::end` make it eaiser to write more generic and maintanable code?

Fail Early, Fail Fast!

Detecting errors at runtime is good. Detecting them at compile time is even better!

```
namespace hardcoded { constexpr auto x_dim() { return 800; } }
/*...*/

static_assert(hardcoded::x_dim() == 832,
    "This 3rd party library won't work if x_dim isn't 832!");
foo(hardcoded::x_dim(), hardcoded::y_dim());
```

Result:

```
% clang++ --std=c++14 static_assert.cpp
static_assert.cpp:12:1: error: static_assert failed "This 3rd party library won't work if x_dim is
static_assert(hardcoded::x_dim() == 832,
^             ~~~~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

The message string cannot be dynamically created (must be knowable at compile time), and will be optional in C++17.

That's Classy!

rk·logix quality software

```
struct foo {
   foo()                 = default;
   foo(foo&&) noexcept   = default;
   ~foo()     noexcept   = default;

   foo(const foo&)             = delete;
   foo& operator=(const foo&) = delete;
};

void fn(foo&&) {}

int main() {
   foo f;

   // foo f2 = f;      <-- won't compile
   //     error: call to deleted constructor of 'foo'

   // foo f3;  f3=f;  <-- won't compile
   //      error: overload resolution selected deleted operator '='

   fn(std::move(f));
}
```

Better?

Question: In what ways is `=delete` better than making the method private?

That's Super-Classy!

Any comments on this code?

```
struct B
{
   virtual void foo() const {}
};

struct D : B
{
   virtual void foo() {}
};
```

That's Super-Classy!

C++11 has some comments about it!

```
struct B
{
   virtual void foo() const {}
};

struct D : B
{
   virtual void foo() override {}
       // error: 'foo' marked 'override' but does not
       //        override any member functions
};
```

That's Super-Classy!

rk·logix quality software

'Sealing' a method or class with `final`

```
struct B {
   virtual void foo() const {};
};

struct D : B {
   virtual void foo() const override final { }
};

struct D2 final : D {
   virtual void foo() const override {}
         // error: declaration of 'foo' overrides a 'final'
             function
};

struct D3 : D2 {};
         // error: base 'D2' is marked 'final'
```

Literally!

rk·logix
quality software

New user defined literals, and some standard ones as well. Literals allow for cleaner syntax while avoiding errors:

```
using namespace std::literals;

std::chrono::seconds s1 = {30};
                auto s2 = 30s;
                auto s3 = s1 + s2;
             // auto s4 = s1 + 30;     <-- compilaton error...
                auto s5 = 1h + s1;  // <-- this is OK!
```

Standard Literals

rk·logix quality software

Standard library

The following literal operators are defined in the standard library

Defined in inline namespace `std::literals::complex_literals`	

`operator""if`
`operator""i` (C++14)
`operator""il`
	A `std::complex` literal representing pure imaginary number (function)

Defined in inline namespace `std::literals::chrono_literals`	
`operator""h` (C++14)	A `std::chrono::duration` literal representing hours (function)
`operator""min` (C++14)	A `std::chrono::duration` literal representing minutes (function)
`operator""s` (C++14)	A `std::chrono::duration` literal representing seconds (function)
`operator""ms` (C++14)	A `std::chrono::duration` literal representing milliseconds (function)
`operator""us` (C++14)	A `std::chrono::duration` literal representing microseconds (function)
`operator""ns` (C++14)	A `std::chrono::duration` literal representing nanoseconds (function)
Defined in inline namespace `std::literals::string_literals`	
`operator""s` (C++14)	Converts a character array literal to `basic_string` (function)

http://en.cppreference.com/w/cpp/language/user_literal

I want to do it too!

```cpp
struct gigawatts
{
    explicit gigawatts(long double gw) : gw_(gw) {}
    long double value() const { return gw_; }
private:
    long double gw_;
};

auto operator "" _GW(long double gw) { return gigawatts(gw); }

int main()
{
    auto flux_power = 1.21_GW;
    std::cout << flux_power.value() << u8" jigawatts\U0000203D"
        << std::endl;
    std::cout << R"("Great Scott!" --\Dr. Emmet Brown\)" <<
        std::endl;
}
```

Raw string literals mess up the LaTeX syntax highlighting!