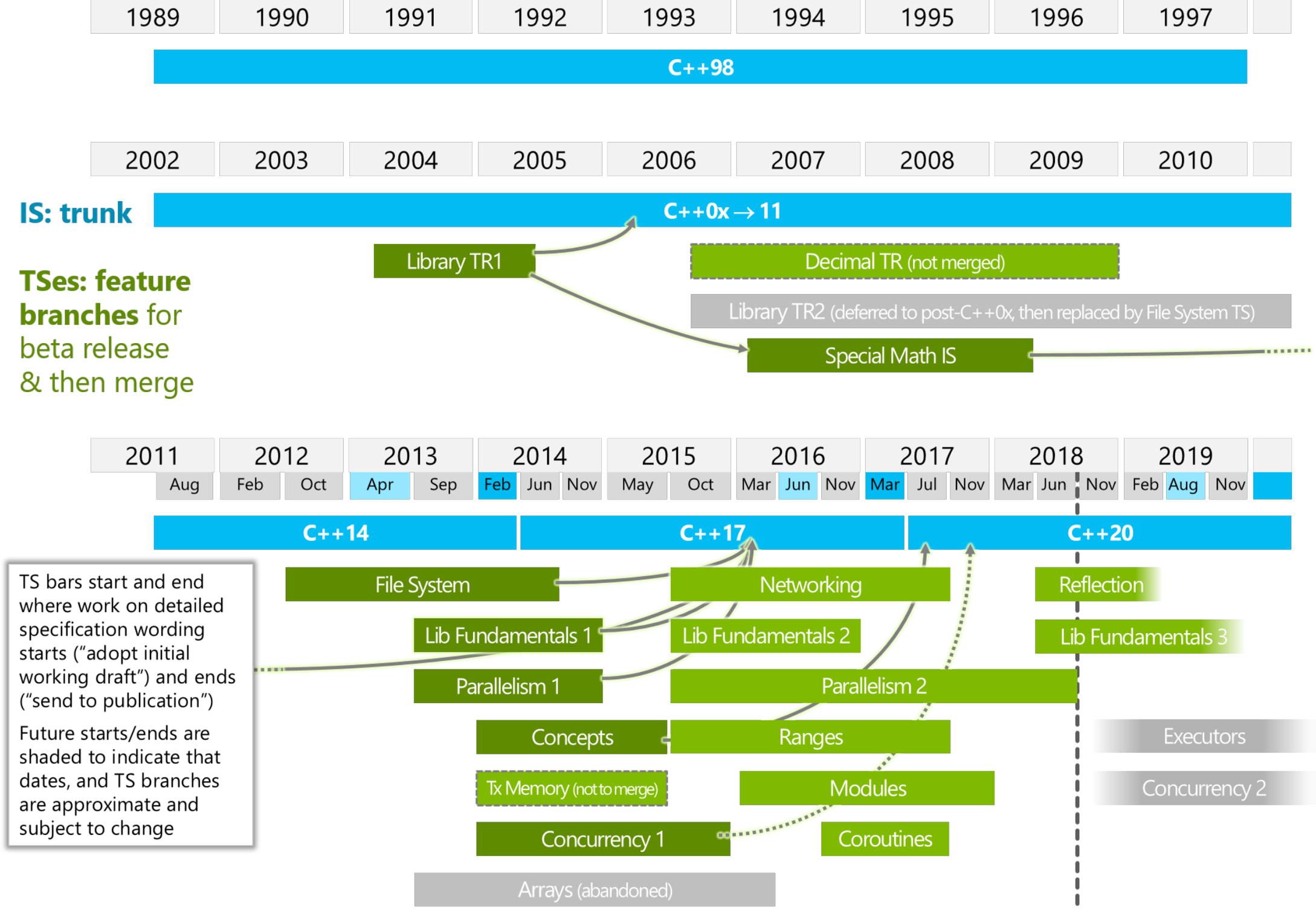
C++14, 17, and beyond for Geant4

Ben Morgan, Gabriele Cosmo

WARWICK
THE UNIVERSITY OF WARWICK



ISO C++ Standard

Core Language + Standard Library. C++17 current, C++20 in development.

C++17 features

C++17 feature	Paper(s)	Version	GCC	Clang	MSVC	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)	HP aCC	Digital Mars C++	[Collapse]
New auto rules for direct-list-initialization	N3922	c++17-lang	5	3.8	19.0*	4.10.1	17.0					17.7			
static_assert with no message	N3928	c++17-lang	6	2.5	19.10*	4.12	18.0					17.7			
typename in a template template parameter	N4051	c++17-lang	5	3.5	19.0*	4.10.1	17.0					17.7			
Removing trigraphs	N4086	c++17-lang	5	3.5	16.0*	5.0									
Nested namespace definition	N4230	c++17-lang	6	3.6	19.0*	4.12	17.0					17.7			
Attributes for namespaces and enumerators	N4266	c++17-lang	4.9 (namespaces) / 6 (enumerators)	3.6	19.0*	4.11	17.0					17.7			
u8 character literals	N4267	c++17-lang	6	3.6	19.0*	4.11	17.0					17.7			
Allow constant evaluation for all non-type template	N4268	c++17-lang	6	3.6	19.12*	5.0									

Compiler Support

Shorter timescales c.f. C++11.

“Supports C++XY” may not mean “all language/library features”

Benefits of using C++14/17..

- ~~All the cool kids are using it~~
- As with C++11, easier, cleaner, more maintainable code
  - *Both from core language and classes/algorithms in Standard Library*
- Performance+Portability as well?
  - *Growing builtin support for concurrency, possibly SIMD*
  - *"Batteries Included" Standard Library can reduce reliance on third party libraries or own cross-platform adaptions*

Notable Features of C++14

- Extended use of `auto`
- Extended `constexpr`
- Added `std::make_unique`
- `[[deprecated]]` attribute
- Others at: https://isocpp.org/wiki/faq/cpp14-language

```cpp
// C++11: explicitly named return type
some_type f()          { return foo() * 42; }
auto f() -> some_type { return foo() * 42; }

// C++14: now deduces "-> some_type"
auto f()               { return foo() * 42; }

auto g() {
  if (some_condition) {
    return foo();
  }
  // O.k. as long as bar() deduces "-> some_type"
  return bar();
}
```

`auto` return type deduction

Use carefully! Explicit types are better! Best for complex derived return types of function templates

```cpp
// C++11:
// Limited: no variable declaration, no loops single return
constexpr int my_charcmp( char c1, char c2 ) {
    return (c1 == c2) ? 0 : (c1 < c2) ? : -1 : 1;
}

// C++14:
// Can now declare local variables, use loop/conditionals
constexpr int my_strcmp( const char* str1, const char* str2 ) {
    int i = 0;
    for( ; str1[i] && str2[i] && str1[i] == str2[i]; ++i )
        { }
    if( str1[i] == str2[i] ) return 0;
    if( str1[i] < str2[i] ) return -1;
    return 1;
}
```

Relaxed `constexpr`

Cannot have static/thread-local variables, no try blocks.

```
// C++11: have to state the parameter type
sort( begin(w), end(w), [](const shared_ptr<some_type>& a,
                           const shared_ptr<some_type>& b) { return *a<*b; } );

auto size = [](const unordered_map<wstring, vector<string>>& m) {
  return m.size();
};

// C++14: use type deduction
sort( begin(w), end(w), [](const auto& a, const auto& b) { return *a<*b; } );

auto size = [](const auto& m) {
  return m.size();
};
```

Generic Lambdas

Like auto return type, best used in generic code and inside functions.

```cpp
#include <memory>

auto myVector = std::make_unique<G4ThreeVector>(1.0, 2.0, 3.0);

myVector->getX();
```

`std::make_unique`

Simply an oversight this didn't make C++11!

```cpp
// deprecated.cc
[[deprecated("removed in the next release")]]
void function()
{}

int main()
{
  function();
}
```

```
bmorgan — -bash — 113×11
$ clang++ -std=c++14 deprecated.cc
deprecated.cc:8:3: warning: 'function' is deprecated: removed in the next release [-Wdeprecated-declarations]
  function();
  ^
deprecated.cc:2:3: note: 'function' has been explicitly marked deprecated here
[[deprecated("removed in the next release")]]
  ^
1 warning generated.
$
```

[[deprecated]]

Useful to provide early warning of entities that will change in upcoming releases

Notable Features of C++17

- Structured bindings
- Init statements for `if` and `switch`
- `if constexpr`
- Template argument deduction for class templates
- Standard library enhancements
  - ***string_view***, ***optional***, *filesystem, parallel algorithms*
- Others listed at: https://isocpp.org/files/papers/p0636r0.html

```cpp
// C++11: Need to declare variables, use std::tie
std::set<S> mySet;
S value{42, "Test", 3.14};

std::set<S>::iterator iter;
bool inserted;
std::tie(iter, inserted) = mySet.insert(value);

if (inserted) std::cout << "Value was inserted\n";

// C++17: Structured bindings
std::set<S> mySet;
S value{42, "Test", 3.14};

auto [iter, inserted] = mySet.insert(value);

if (inserted) std::cout << "Value was inserted\n";
```

Structured Bindings

Works with Tuples, Arrays, (public) Structs, any type having `tuple_size`, `get`

```cpp
// C++11: Separate initialization and conditional
// Have to use a scope if we want to hide val
{
  auto val = GetValue();
  if ( condition(val) )
    // on true
  else
    // on false
}

// C++17: Can combine initialization and conditional
// Now val is only visible in the scope of the if/else
if ( auto val = GetValue(); condition(val) )
  // on true
else
  // on false
```

Init statement for `if`, `switch`

Most useful for scoping values used only in conditionals. Works very well with Structured Bindings.

```cpp
// C++17: Discards branches of if statement at compile-time based on a
constexpr condition
if constexpr (cond)
  onTrue(); // discarded if condition is false
else
  onFalse(); // discarded if condition is true

// Example: tuple get<N> interface for a struct:
struct S {
  int n;
  std::string s;
};
```

```cpp
// C++11: Would need two functions, get<0>, get<1>
// C++17: Single function and if constexpr:
template <size_t I>
auto& get(S& s) {
  if constexpr (I == 0) return s.n;
  else if constexpr (I == 1) return s.s;
}
```

if constexpr

As a compile-time if, useful to unify code that might otherwise be split into different specialisations

```cpp
// C++11: function taking a template class argument
void foo(std::pair<int, char>);

// This is o.k.
foo(std::make_pair(42, 'z'));
foo(std::pair<int, char>(42, 'z'));

// This fails...
foo(std::pair(42, 'z'));

// C++17: Now valid to call
foo(std::pair(42, 'z'));

// Can also use for local variables:
std::pair p(10, 'x');
std::tuple t(1, 3.14, true, 't');
```

Template argument deduction for class templates

Useful for clarity when template arguments are complex/long

```cpp
// C++11:
// Many possible representations of "sequence of characters"
bool compare(const std::string& s1, const std::string& s2);
bool compare(const char* s1, const char* s2);
bool compare(const char* s1, const std::string& s2);
bool compare(const MyStringType& s1, const MyStringType& s2);
...

// C++17:
#include <string_view>
// A non-owning, read-only view of a character sequence, in pseudo-code:
class string_view {
  size_t len;
  const CharT* str;
};

bool compare(std::string_view s1, std::string_view s2);

class MyStringType {
 public:
  operator std::string_view() const;
};
```

std::string_view

Helps to minimise allocations (but check performance!), and function specializations

```cpp
#include <optional>

// A function that might not produce a result
std::optional<int> GetAnswer(int r) {
  if ( some_condition_on(r) )
    return 42;

  return {}; // or std::nullopt
}

// Handle an optional result
auto result = GetAnswer(someInteger);

if (result)
  PrintResult(*result);
else
  Report("no result obtained");
```

std::optional

More expressive way to deal with empty values than exceptions, output params etc

Other Standard Library Features

- Parallel Algorithms
  - *Provide execution policy: sequential, parallel, parallel_unsequenced*
  - `std::algorithm_name(policy, /* args for algorithm_name */)`
  - *Likely needs care to integrate with overall concurrency (Geant4/Exp)*
- Filesystem
  - *Classes and functions for path/directory/file operations and queries*
  - *Especially useful for platform-independent path manipulation*

Features removed in C++17

- Full list of removals and deprecations available through
  - *https://isocpp.org/files/papers/p0636r0.html*
- Geant4 is using some of these features
  - *Should start to review and replace if possible with C++11-compatible workarounds*
- See also https://en.cppreference.com/w/cpp for additional help/info

```
// Removed in C++17
// <memory>
class auto_ptr;

// <algorithm>
template< class RandomIt >
void random_shuffle( RandomIt first, RandomIt last );
// ... and other signatures

// <functional>
template< class Res, class T >
std::mem_fun_t<Res,T> mem_fun( Res (T::*f)() );
// ... and other signatures

template< class Res, class T >
std::mem_fun_ref_t<Res,T> mem_fun_ref( Res (T::*f)() );
// ... and other signatures

template< class F, class T >
std::binder1st<F> bind1st( const F& f, const T& x );

template< class F, class T >
std::binder2nd<F> bind2nd( const F& f, const T& x );

// ... and others in <functional>
```

Using C++14/17.. in Geant4

- Can we start use of newer standards more rapidly?

- Easy to add builds **against** new standards **as soon as Compilers/CMake support them**

  - *Early warning of upcoming issues from deprecated/removed features*

  - *Implies testing resource, even if only a single OS/Compiler*

- **Using C++14/17/.. features in Geant4 impacts our user support**

Costs of using C++14/17.. in Geant4

- Choice of Standard+Features sets minimum GCC/Clang/Xcode/VS required to build and use Geant4
  - *Limits Linux/macOS/Windows versions we can support with **system compiler(s)***
  - ***Mitigated in CentOS/Ubuntu by Software Collections/Toolchain PPA***
- Related to this is API/ABI compatibility with externals (Xerces, Qt)
  - *Conservatively, all C++ externals linked to should be **compiled with same standard***
- **Requires pragmatism to balance developers needing modernity vs users wanting stability/ease of use (or vice versa, and various shades of grey in between…)**

Balancing Benefits and Costs

- Balance old/new with a **backward compatibility policy, e.g.**
  - *Define oldest standard to support as C++11*
  - *Use features from C++14/17* ***if enabled and available, provided that…***
    - ***…a workaround is available when compiling against*** *C++11*
- Can review oldest supported standard each year
  - *Decision to increase oldest supported standard should be driven by overall user experience/requirements of ease-of-use vs performance*
  - *Should have a clear and published removal timetable*

Challenges for C++ Backward Compatibility

- Workarounds practically implemented using `#ifdef/#elif/#else` blocks
  - *Needs care to keep code clear, and does increase SLOC/testing space*
  - *Mitigate by centralising any workarounds in Global category*
- However, some features from newer standards **might not be implementable in oldest supported**, or might require external like Boost (e.g. optional/filesystem…)
  - *Another aspect to pragmatism, e.g. requiring a newer compiler may be easier than supporting an external library*

Summary

- C++ Standard updating steadily, along with compilers
- Many new features in C++14, 17, and 2a that could be useful in Geant4
- To support and use new standards as quickly as possible, need to consider overall experience for users and developers
- **Consider using a backward compatibility policy to allow faster rollout whilst supporting older standards**