

The New C++ Standard

Roger Orr – OR/2 Limited
<http://www.howzatt.demon.co.uk/>

C+++ ?

The New C++ Standard

Roger Orr – OR/2 Limited

<http://www.howzatt.demon.co.uk/>

Why is a new standard needed?
What is in the new C++ standard?
When will it be ready?

Overview

- C++ was created by Bjarne Stroustrup just over 20 years ago
 - First 'standard' was the ARM (Annotated Reference Manual, 1990)
 - ISO standard was ratified 1998
 - ... and modified in 2003 (mostly 'bug fixes')
 - Library Technical Report 2005 (enhancements)
-
-

ISO standardisation for C++

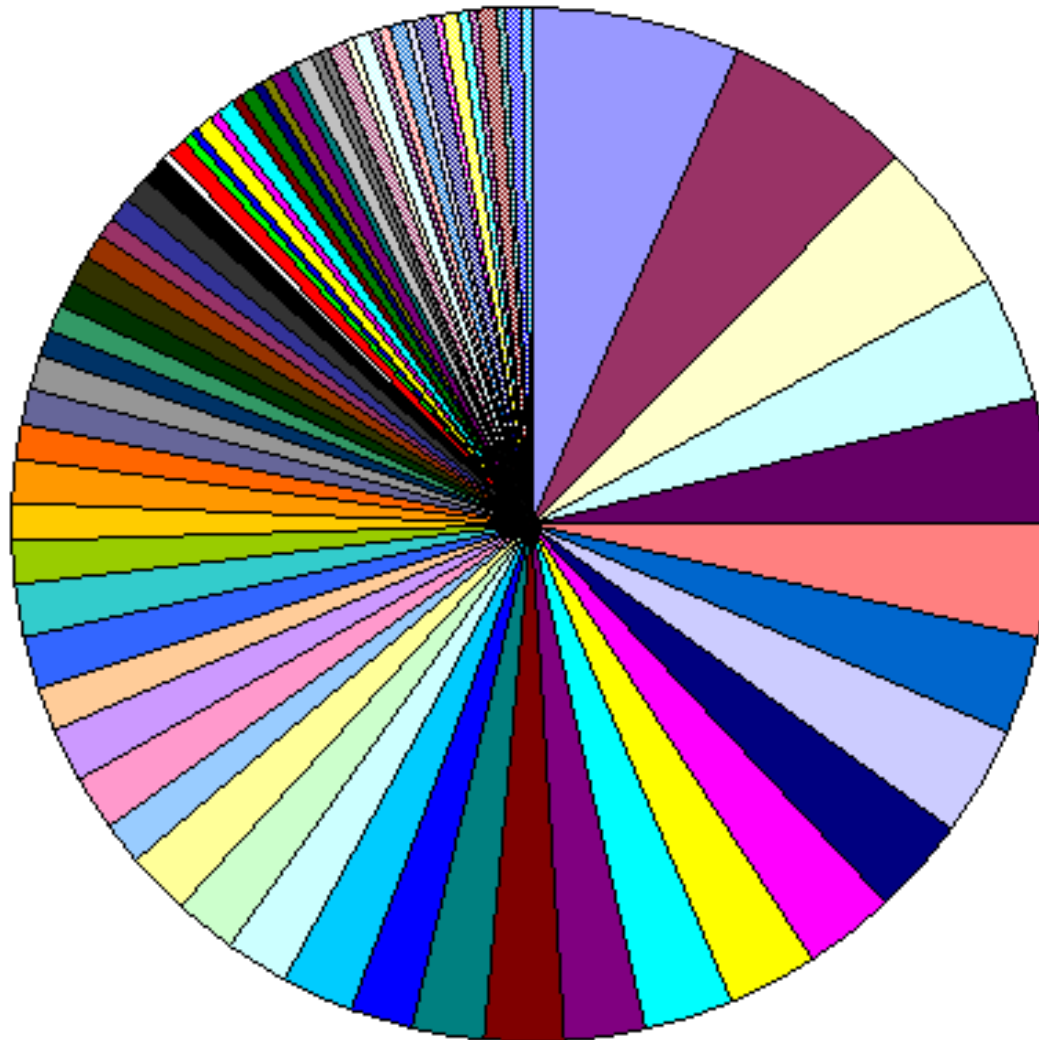
- Membership is over 20 nations (up to 12 represented at each meeting)
 - Technical meetings co-hosted with ANSI
 - Other nations have further technical meetings, for example the UK has the BSI C++ panel
 - About 100 active members (50+ at each meeting)
 - About 250 members in all
-
-

ISO standardisation for C++

- Membership is over 20 nations (up to 12 represented at each meeting)
 - Technical meetings co-hosted with ANSI
 - Other nations have further technical meetings, for example the UK has the BSI C++ panel
 - About 100 active members (50+ at each meeting)
 - About 250 members in all

 - Bjarne describes the process as:
 - formal, slow, bureaucratic, and democratic
 - “the worst way, except for all the rest”
-
-

Core changes by author



- ★ Crawl, Lawrence
- ▶ Stroustrup, Bjarne
- Gregor, Doug
- ★ Boehm, Hans-J
- ★ Reis, Gabriel Dos
- ▶ Meredith, Alisdair
- Dimov, Peter
- ▶ Glassborow, Francis
- Jarvi, J.
- Brown, Walter E.
- McKenney, Paul E.
- ★ Sutter, Herb
- ★ Vandevoorde, Daveed
- ★ Dawes, Beman
- Maurer, Jens
- ▶ Goldthwaite, Lois
- Hinnant, Howard
- Siek, Jeremy
- Spertus, Michael
- Wong, Michael
- ★ Abrahams, David

ACCU member ▶ ACCU speaker ★

Why is a new standard needed?

- C++ is a very popular language
 - Lost some ground to simpler languages (e.g. C#, Java – which are getting more complex) and to dynamic languages (e.g. Python, Ruby)
 - Some historic rough edges
 - Restrictions using newer techniques (e.g. TMP)
 - The world is going parallel (“The free lunch is over” - Herb Sutter at the ACCU conference)
-
-

The Spirit of C++

- C++, like every language, has its own rationale. It's not explicit, but includes:



The Spirit of C++

- C++, like every language, has its own rationale. It's not explicit, but includes:
 - Multi-paradigm (~ TMTOWTDI)

The Spirit of C++

- C++, like every language, has its own rationale. It's not explicit, but includes:
 - Multi-paradigm (~ TMTOWTDI)
 - Trust the programmer

The Spirit of C++

- C++, like every language, has its own rationale. It's not explicit, but includes:
 - Multi-paradigm (~ TMTOWTDI)
 - Trust the programmer
 - You don't pay for what you don't use

The Spirit of C++

- C++, like every language, has its own rationale. It's not explicit, but includes:
 - Multi-paradigm (~ TMTOWTDI)
 - Trust the programmer
 - You don't pay for what you don't use
 - Don't break working code

The Spirit of C++

- C++, like every language, has its own rationale. It's not explicit, but includes:
 - Multi-paradigm (~ TMTOWTDI)
 - Trust the programmer
 - You don't pay for what you don't use
 - Don't break working code
 - Checks at compile time are better than runtime

Principles for the new standard

- From Bjarne's web site:
http://www.research.att.com/~bs/bs_faq.html#When-next-standard
 - “My personal view is that the key principles should be:
 - no major changes to the language itself
 - major extensions to the standard library
 - The changes and extensions should be chosen to make C++ a **better** platform for systems programming and library building, and to make C++ **easier** to teach and learn.”
-
-

So what is in the new standard?

- Major language changes
 - Almost all are extensions
 - Affects the ABI
- Minor language changes – smooth rough edges, support new features
- New library components – many already available in TR1.

Major language changes

- rvalue reference ('move' semantics)
 - decltype/auto/deduced return types
 - variadic templates
 - new style 'for' loop
 - initialisation
 - concepts
 - threads/memory model
 - atomics
 - lambda
 - garbage collection
-
-

rvalue reference/move semantics

- **Key motivation:** remove unnecessary creation of temporary objects by *moving* contents *
- **Language concept:** add "`type &&`" reference type and overload functions with "`type &`"
- Can be **used** with no code change:

```
string wrap( const string &tag, const string &contents ) {  
    string s = "<" + tag + ">" + contents + "</" + tag + ">";  
    return s;  
}
```

* http://home.twcny.rr.com/hinnant/cpp_extensions/STL_benchmarks.html
187.3 s reduced to 13.6 s using move semantics

decltype/auto/deduced return types

- Generic code needs to know about related types
- The C++98 solution was traits, but this doesn't solve all problems and adds complexity
- Biggest problems were reference types and const

```
int i;  
vector<decltype(i)> vec;  
auto beg = vec.begin();
```

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);  
(syntactic note: auto here is basically a place-holder)
```

variadic templates

- Some templates can take a variable number of arguments, and must currently be specialised explicitly for each case up to a predefined limit
- Add new syntax using “...” to provide a placeholder for additional arguments: a type-safe version of variable argument lists.
- Eg: Type-safe printf(), tuples, perfect forwarding

```
template<typename... Types> struct Tuple { };  
Tuple<int, float> t2; // two arguments: int and float
```

New 'for' loop syntax

- Many bugs caused by bad iteration
- Provide syntax, like many other languages, to automatically generate the iteration code
- Strong resistance to 'magic'

```
#include <iterator_concept> // header is required!  
#include <vector>
```

```
int main() {  
    std::vector<int> vec = { 1,2,3,4,5 };  
    for ( auto & x : vec )  
        x *= 2;  
}
```

initialisation

- Initialising in C++
 - inherits from C
 - and adds constructors
- Several proposals to make the syntax more consistent.
- Problems include ambiguity and breaking code

```
story = vector<string>{"once", "upon", "a", "time"};
```

```
for ( int i(0); i != max; ++i ) ...
```

concepts

- Hard to express template constraints and so
 - error messages are unclear
 - some errors are only detected at runtime
 - Concepts express **intent**
 - to the reader
 - to the compiler
 - Template declarations are augmented with a set of constraints (requirements)
 - Likely to be *written* by a small percent of programmers but *used* by nearly everyone
 - (See library example later)
-
-

threads/memory model

- Motivation is obvious; CPU speeds on the desktop have stopped increasing in favour of more cores
 - Core language changes needed to define an unambiguous memory model using terms like 'data-race', 'referenced-before' so behaviour of more aggressive hardware optimisations is defined
 - Compiler support is needed for some low level mechanics such as thread local storage and exception transfer, or code is non-portable and brittle
-
-

atomics

- Growing interest in lock-free programming
- Technically hard – a new lock free algorithm is a publishable result
- Minimum building block is a range of 'atomic' data types (together with memory model above)
- Problem: sequential consistency or weak C4?

```
struct event_counter
{
    atomic_ulong au;
    void inc() { au.fetch_add( 1, memory_order_relaxed ); }
    unsigned long get() { au.load( memory_order_relaxed ); }
};
```

lambda

- 'treat code as objects'
- Common in some other languages
- Harder in C++
 - Less abstract architecture
 - Object lifetime: reference or copy semantics?

```
double sum = 0; int factor = 2;  
for_each(array, array + 4, [&](n) { sum += factor * n });
```

Garbage collection

- C++ has 'deterministic finalisation' (destructors)
- May be avoided for memory management
 - Move semantics reduce this need for temporaries
 - Smart pointers and array perform well
 - Some data structures hard to manage explicitly
- Controversial!
 - Litter collection looks good
 - “Full-on” garbage collection conflicts with C++ idioms
- Still needed for resource management - *not* tackling finalisation – experience with Java and .Net is that this is very hard to use correctly.

Minor language changes

- strongly typed enums
 - constexpr
 - static_assert
 - some C99 compatibility
 - Unicode characters
 - delegating/inheriting ctors
 - right angle brackets
 - propagating exceptions
 - default/deleted methods
 - nullptr
 - attributes
 - local classes as template arguments
-
-

strongly typed enums

- “C enumerations constitute a curiously half-baked concept.”
 - Stroustrup *The Design and Evolution of C++*, p. 253
- An enumeration is a poor encapsulation
 - Easy confusion with int
 - Names 'leak' into surrounding scope
- Use an idea first introduced in C++/CLI

```
enum class E { E1, E2, E3 = 100, E4 /* = 101 */ };  
void f( E e ) {  
    if( e >= 100 ) // error: no E to int conversion  
        ;  
}  
int i = E::E2; // error: no E to int conversion
```

constexpr

- Forced to choose between type safety (“elegance”) and compile-time evaluation (“efficiency”) of non-trivial constants
- Templates *require* compile time values
- New function decorator, `constexpr`, produces a compile time constant if input arguments are known at compile time.

```
constexpr int abs(int x)
{ return x < 0 ? -x : x; }
const int m = ..., n = ...
char array[ abs( m-n ) ];
```

static_assert

- Better compile time checking
- Neither `assert` nor `#error` work well with templates
- Solves similar problem to `constexpr` concepts, but simpler

```
static_assert(sizeof(VMPage) == PAGESIZE,  
    "Struct VMPage must be the same size"  
    " as a system virtual memory page.");
```



some C99 compatibility

- C99 has had a mixed reception with little take up, but some changes are included, such as:
- `long long` data type
- `_Pragma` macro
- Variadic macros
- `__func__`

Unicode characters

- Internationalisation is becoming increasingly important and this drives a need for Unicode
- Based on C99 work but creates **distinct** types to support overloading rather than typedefs
- New types, `char16_t` and `char32_t`
- New literals, `u'c'`, `U'c'`, `u"str"`, `U"str"`

delegating/inheriting constructors

- When implementing a new type the constructors cannot easily share code with siblings or inherited constructors
- Similar features in other languages (C#, Java)

```
struct B1 {  
    B1( int ) {}  
};
```

```
struct D1 : B1 {  
    using default B1; // implicitly declares D1( int )  
    int x;  
};
```

right angle brackets

- For historical reasons nested templates need ' '
 - `std::vector<std::string<char>> vec; // error`
 - Unexpectedly hard to change as affects a key parsing rule
 - May break existing code – in some contrived examples even change the meaning.
 - (No issue for C# or Java as they do not support non-type generic arguments)
-
-

propagating exceptions

- Related to the work on threading, as a way is needed to pass an exception from one thread to another.
- Prefer an explicit solution to 'compiler magic'

```
typedef unspecified exception_ptr;
```

```
exception_ptr current_exception();
```

```
void rethrow_exception( exception_ptr p );
```

```
template< class E > exception_ptr copy_exception( E e );
```

default/deleted methods

- C++ defines a number of implicitly generated methods.
 - may need to suppress the generation
 - may want to explicitly restore a default function
- Replaces two common current tricks:
 - Declare `private` copy ctor/copy assignment operator
 - Inherit from `boost::noncopyable`

```
struct atomic_bool {  
    atomic_bool() = default; // otherwise suppressed!  
    constexpr atomic_bool( bool );  
    atomic_bool( const atomic_bool & ) = delete;  
    ...  
};
```

...

nullptr

- C++ uses literal 0 for both zero and null.
 - Can't use C 'trick' of `#define NULL ((void*)0)`
 - Can cause nasty overload problems
- `nullptr` is a zero pointer that can be converted to any pointer type.
- Implemented in C++/CLI (with similar wording)
- The alternative `null` was rejected as breaks code

attributes

- Need to annotate C++ entities with additional non-type information in a standard, yet extensible, way
 - `void fatal(void) [[noreturn]];`
- Attributes can be extended by compiler vendors
- Possible uses include:
 - override
 - dynamic libraries
 - thread local storage
 - alignment

local classes as template arguments

- A problem with using algorithms with function objects is the class must be non-local.
- Typically breaks localisation/encapsulation

```
void printByName(std::set<Payload> const &s) {  
    class ByName {  
    public:  
        bool operator()(Payload const &lhs, Payload const &rhs) {  
            return lhs.compareName(rhs);  
        }  
    };  
    std::set<Payload, ByName> s2(s.begin(),s.end());  
    ...  
}
```

New library components

- Items already in TR1 (paper N1836)
 - `shared_ptr/weak_ptr`
 - `function()`
 - random numbers/numerical facilities
 - `tuple`, `array`, `unordered_XXX`
 - `type_traits`
 - regular expressions
 - Atomic, locking and mutex types
 - Thread library - thread pools and futures
 - Placement insert
 - 'Eating our own dog food'
 - More to come in TR2
-
-

shared_ptr/weak_ptr

- Implementation experience in boost
- Provides a standard for reference counted smart pointers
- `weak_ptr` is for breaking dependency cycles
- Avoids the `auto_ptr` single-ownership problem
- Still likely to be a need for additional smart pointers for various use cases

function()

- Motivation: generalisation of the 'observer' pattern
- Treat functions and objects the same way

```
class Subject {  
    virtual void Attach( function<void (Subject*)> o );  
};  
class Timer : public Subject { /* ... */ } timer;  
  
void tick( Subject * sender ); // Classic observer style  
class Tock { // Function object style  
public: void operator()( Timer * timer ) { /* ... */ }  
} tock;  
  
timer.attach( &tick );  
timer.attach( tock );
```

random numbers/ numerical facilities

- `rand()` is not a very good random number source – (but good for Code Critique examples!)
 - TR1 provided a variety of types for providing random numbers, and ways to combine them:
 - `mersenne_twister`
 - `random_device`
 - `xor_combine`
 - `binomial_distribution`
 - `variate_generator` (combines generator + distribution)
 - However the special maths functions in TR1 did **not** make it into the next C++ standard.
-
-

tuple, array, unordered_XXX

- Tuple shown earlier for type safe list of objects
`typedef tuple<double, currency> price;`
 - Array – more efficient than vector for compile-time fixed lengths as avoids allocation
`array<int, 10> arr;`
 - `unordered_XXX` for hashed collections
 - `set`, `multiset`, `map` and `multimap` supported
 - Can provide your own hash function, or use the system default template for standard types
-
-

type_traits

- Properties of a type

Eg: `is_integral<T>::value`

- Relationship between types

Eg: `is_convertible<T,U>::value`

- Transformation of types

Eg: `add_pointer<T>::type`

regular expressions

- Regular expressions available in many languages
 - there are varieties of syntax and so a variety of RE options are configurable

```
void demo( string const & patt, string const & str )
{
    regex rgx(patt);
    cregex_iterator first(str.begin(), str.end(), rgx), last;
    ostream_iterator <cmatch> out(cout, "\n");
    copy(first, last, out);
}
```

Atomic, locking and mutex types

- Atomics (see above) for lock free algorithms
- Locking ensures serialised access to data
- Condition variables are a clean abstraction (but surprisingly hard to provide on some platforms)

```
{ // Thread A
  mtx::scoped_lock lock( mutex );
  conditionVar.wait( lock );
}
{ // Thread B
  mtx::scoped_lock lock( mutex );
  ...
  conditionVar.notify();
}
```

Thread library – pools and futures

- Standard way to create and refer to a pool of worker threads. Tasks are submitted to the pool and will be done when the next thread is free
- The result from a task is a 'future' – a potential value or an exception

```
auto future_value = launch_in_pool( function_object );
```

```
future_value.ready();
```

```
future_value.get(); // wait, then return value or throw exception
```

Placement insert

- Standard library collections such as `list`, `set` and `map` never move objects – but they have to be copied in initially
- Placement insert avoids the copy – uses other facilities like `std::forward` and move semantics

```
alist.push_front( args );
```

```
themap.emplace( iter, args );
```

'Eating our own dog food'

- The library can use new approved core changes for new features, and also retrofit to older ones
 - Examples include
 - rvalue references
 - constexpr
 - default/deleted
 - Unicode
 - long long
 - variadic templates
 - static_assert
 - concepts (a lot to do *and* need to avoid breaking code)
-
-

When will it be ready?

- Nearing a draft standard
 - No new proposals accepted
 - Final word-smithing
 - Integrating similar proposals
 - Couple of problem areas still under resolution
- Latest draft of standard is N2461.pdf (22 Oct 07)
- Keep an eye on the WG21 web site:
<http://www.open-std.org/jtc1/sc22/wg21/>

What can we do in the meantime?

- Many parts of the new standard are implemented in versions (or branches) of g++
 - EDG have implemented some of the new features, but it depends on the front-end what is visible
 - Borland have plans for a C++0x beta program
 - A few parts of the new standard are in MSVC
 - TR1 support beta just released for VS 2008
 - Lots of library components are derived from boost
-
-