The world is evolving and the C++ standard must evolve with it. **Kevlin Henney** explains what the standards working group is doing to make it happen – and what we're likely to see in the future

Standardising on C++

Kevlin Henney is an independent software development consultant and trainer. He can be reached at **www.curbralan.com**

THE C++ STANDARD[1] IS COMING UP TO ITS third birthday. The language itself is, of course, much older, dating back in prototypical form over 20 years and exposed to the public eye for more than 15. The language has matured in this time, effectively evolving through three distinct languages, each supporting different styles: early C++, classic C++ and modern C++[2]. It's modern C++ that embraces the standard, and vice versa. It was the move from classic to modern that set off the greatest growth in the language's features, as well as a dip in its portability.

The standard now presents a stationary and stable point for consumers and producers of C++ implementations – the regular developer, plus the writers of standard library implementations and compilers. There are signs that the compatibility gap is closing[3], and over the next year some imminent compiler releases look set to close it further. But stationary isn't the same as stagnant or staid. The past is often the best predictor of the future, and C++ is going to change.

The root of equality

Where do standards come from? Perhaps this is too broad a question – there are many different kinds of standard. The de facto standards that are declared by monopoly or hegemony in the name of marketecture are not the focus here. So, more precisely, where do *de jure* international standards come from?

ISO[4] is a federation of national standards bodies from over 140 different countries, such as BSI in the UK and DIN in Germany. ISO is a proper name, not an acronym or an abbreviation. It is based on the Greek word for "equal" that appears as the prefix in words such as isobar and isosceles. The long form of ISO is officially International Organisation for Standardisation.

There are many routes to standardisation through ISO. Some are developed as national standards and then become international standards, others are developed explicitly as international by many national bodies, and others still are taken from de facto standards in industry and then rubber-stamped. C initially took the first route, becoming an ANSI standard in 1989. It became an ISO standard in 1990 and has evolved that way ever since. This means that, technically, there is no longer such a thing as ANSI C (a good interview question for those in the know). With C++, the barn door was closed before the horse had a chance to bolt, so the language was standardised under ISO almost from the word go.

C++ standardisation work is carried out by the rather anonymously named SC22/WG21 (Subcommittee 22, Working Group 21), which now comprises 11 national standards bodies (NBs). The individuals involved, however, are far from anonymous, and the rogues' gallery that has made up the last couple of UK delegations comprises myself, Francis Glassborow, Andrew Sawyer, Mark Radford and Lois Goldthwaite, the convenor of the UK panel. If you are interested in becoming involved with the C++ (or C) standard at either the national or international level, contact Lois through standards@accu.org. The ACCU[5] supports the standardisation process and has a number of standards folk as members.

Since standardisation, WG21 has met twice a year. With one exception, the NBs arrange their own meetings according to their needs. For instance, the BSI panel has meetings a few weeks before and after each WG21 meeting, establishing positions on various issues beforehand and debriefing afterwards. The one exception I mentioned is the US NB, ANSI. ANSI – or, to be precise, NCITS (the National Committee for Information Technology Standards) – is by far the largest NB involved. The official arrangement is that ANSI meetings are collocated with the ISO meetings (or vice versa), an arrangement that gets the most heads around the table and the most bang per meeting buck. In practice, this means there is effectively one week-long meeting that occasionally forks to accommodate different organisational requirements, such as votes or, in the case of ISO, selection of an official language in which to hold the meeting.

FACTS AT A GLANCE

- **The C++ standard is nearly three years old and thoughts are now turning to the next standard.**
- **Humans are responsible for standardisation, which means that it is a political activity as well as a technical one.**
- **The focus for extensions is the library rather than the core language.**
- **Extensions should focus on ease of use, embracing common cross-platform utilities such as threading, and filling out obvious gaps in the existing standard.**

A practical world

In a perfect world, standards would be entirely correct, and arrived at objectively and instantaneously without any discord. However, C++ inhabits the real world and such wishful thinking doesn't apply. Like any other specification, the C++ standard has bugs. In standardese, bugs are known less affectionately as defects. It's also the result of push and pull between different stakeholders who are human rather than Vulcan: compiler writers, library authors, educators, regular developers and so on. Standardisation by committee has liabilities as well as benefits. The long and twisted tale of auto_ptr is a notable failure, but the adoption of the STL (not designed by a committee) into the standard library is a marked success. In practice, this means some desirable features are missing from the standard for technical and political reasons.

Since the standardisation of C++, WG21 has been in maintenance mode, attending to defects. The first of the set of fixes, TC1 (Technical Corrigendum 1), is soon to become official. But it's not just about bug fixing, or defect resolution. ISO requires standards to be reviewed every few years, to see if new concepts need to be incorporated or old ones retired, although pruning rarely occurs in practice.

It's against this background that the library working group last year proposed a new work item to explore possible future extensions to the standard library. A strong, almost conservative emphasis on prior art rather than novelty has informed the brief for this work. In part, this is a reaction to criticism that at times the C++ committee indulged in too much invention, without the appropriate proof of concept and existing practice that acted as the backbone, for instance, of the first C standard. Extensions to the language aren't a prime consideration, except where they would simplify the development of new library features, such as even better support for generic programming.

It's against this background of change according to need that Bjarne Stroustrup, who has continued his involvement from the language's invention through to its standardisation, has made a proposal for considering future directions for the next standard, dubbed C++0X. In common with the library working group's proposed direction, the extensions he calls for should be mainly in the area of the library rather than the core language. His suggestions are in some places more detailed and go further towards meeting the demands of common users, while potentially bucking the interests of some of the stakeholders in standardisation.

The stage is set for a technical report in the next couple of years on directions for the next C++ standard.

Goals and penalties

There's a shared sense that extensions should make both the language and the library more consistent within themselves, while avoiding major extensions in the core language. Adding common algorithms and data structures that are "obviously" missing from the library is one example. Instances of these can be found in the SGI STL implementation[6] and the Boost library[7]. For example, single-linked lists, hash table associative containers and a fixed-size array class template in the library and in the language improve type safety, add template typedefs and introduce a typeof feature.

Another aim that Stroustrup stressed was teachability – features should be introduced that better support the expectations of language novices. In other words, for every new feature – or indeed for every existing feature – what are the challenges in teaching and using it correctly? This demand further reinforces the need for improved uniformity and the role of the principle of least astonishment in language design, plus avoiding nifty features for their own sake. The teachability requirement is slightly offset by the complementary desire that C++ remain strong, and become a stronger language for systems programming and library building.

There is also a desire that the relationship between C++ and other systems improve, whether languages or APIs. For instance, standard bindings should exist in other standards to C++, such as SQL, and standard bindings should be based on the fuller feature set of ISO C++ and not an antiquated dialect such as Corba.

The existing Corba-C++ mapping is based on a C++ not very different from a subset of the version described in the *Annotated C++ Reference Manual*[8]. A generous description is that it is very C-like, carefully sidestepping many of the recognised idioms of good C++ and ensuring as much reinvention of the standard library as possible, such as string and sequence types. This has improved

In a perfect world, standards would be entirely correct, and arrived at objectively and instantaneously without any discord. However, C++ inhabits the real world

a little in recent revisions, but not to the point where you could mistake the C++ used as being anywhere near modern C++. To put this into context, the ARM is more than 10 years out of date and has been superseded by a proper standard[1]. Java is heavily used in distributed systems development and did not even exist in this time frame, while XML is now seen as the lingua franca of distributed communication and didn't exist when Java hit the scene. The conservatism of the Corba–C++ mapping seems a little hard to justify in this light.

Another obvious standard with which C++ should have good relations is the C standard. C had a revision in 1999 and some of its extensions should be adopted for C/C++ compatibility. Unfortunately, some of its new features are unnecessary in C++. C's attempt to embrace numeric programming isn't entirely successful, with C++ offering more efficient and expressive means to the same ends. It's worth noting that C++'s own numerics library isn't entirely exemplary either, and its success in this field can be attributed more to the work of non-standard libraries such as Blitz++[9] than to either its own or C99's nouveau-Fortran approach.

However, it's one thing to have better liaison with the C language and the C committee, and quite another to merge the C and C++ committees – and potentially also the languages – as proposed by Bjarne. There are so many political and ideological obstacles to such a merger that it seems safe to say the most we can hope for is better liaison. In effect, the result of such a merger would have to be a takeover, with C++ being accepted as the new C. For some people – notably those who have moved from C++ to C – this won't be a problem, but for those who program in C and can't abide C++, it's what is known in the standards committees as an OMDB issue (Over My Dead Body).

Another area where idealism could jump the tracks is the laudable desire to remove embarrassments. Not everything in the language is perfect, but some things are buggier than others. On more than one occasion, experts have contemplated calling for the withdrawal of auto_ptr in its current, semantically troubled form, or of the failed specialisation of vector<bool> as a bit vector. The idea of dropping existing features raises compatibility

questions, and views of the committee members vary wildly when it comes to changes – perhaps more on this issue than any other. Unless resolved, this issue will result in the addition rather than the removal of features. Compatibility's good, but sometimes there can be too much of a good thing.

General utilities

Moving away from the politics and into the more comfortable realm of technology, what are some of the general utility features touted for inclusion?

The typeof feature has enjoyed support in the GNU gcc compiler for over a decade. It's used to deduce a type from an expression. typeof is not the same as typeid, which is part of the existing runtime-type information (RTTI) framework. typeid returns a type_info object that describes the type of its operand, but the result isn't itself a type. The problem that typeof resolves is addressed only in part through the use of template partial specialisation and traits. Consider the following problem: given a template parameter of a type that supports dereferencing through the unary * operator, how can you deduce and declare the dereferenced type? The following trivial function template demonstrates this problem, with ??? representing the type declaration in question:

```
template<typename indirection_type>
??? dereference(indirection_type arg)
{
  return *arg;
}
```

One solution to this problem isn't really a solution at all: require an extra template parameter to require the caller to specify the return type, or use a reference argument to deduce it instead of returning a value. On the other hand, if we could guarantee that indirection_type would always be an iterator or pointer, we could use the standard iterator_traits:

```
template<typename indirection_type>
std::iterator_traits<indirection_type>::reference
dereference(indirection_type arg)
{
  return *arg;
}
```

This approach fails when arg is a non-iterator smart pointer, which leads to something like a trait per function approach, so the function caller must now ensure that an appropriate specialisation exists, in addition to performing the actual call:

```
template<typename indirection_type>
struct dereference_traits
{  // assume iterator or pointer by default
  typedef std::iterator_traits<
    indirection_type>::reference reference;
};
typename<typename element_type>
struct dereference_traits< std::auto_ptr<element_type> >
{
  typedef element_type &reference;
};
... // etc
template<typename indirection_type>
dereference_traits<indirection_type>::reference
dereference(indirection_type arg)
{
  return *arg;
}
```

What this approach lacks in elegance it makes up for in verbosity. typeof would simplify the problem to the point of it being trivial:

```
template<typename indirection_type>
typeof(*indirection_type())
dereference(indirection_type arg)
{
  return *arg;
}
```

You can currently templatise classes and functions, but not typedefs. Consider the following problem. You want to provide users of your code with the name of a templated sequence type, so they can write something like the following:

```
nonstd::sequence<int> ints;
nonstd::sequence<callback *> callbacks;
```

What is sequence? It could be a custom array or list template, or it could be any other class template that satisfies the sequence requirements in the library – except that it couldn't be. Although, in your initial version, you might want to write the following:

```
namespace nonstd
{
  template<typename value_type>
  typedef std::vector<value_type> sequence;
}
```

you can't. You have to resort to faking the template typedef as follows:

```
namespace nonstd
{
  template<typename value_type>
  struct sequence
  {
    typedef std::vector<value_type> type;
  };
}
```

Which also affects the user's previously transparent view of things:

```
nonstd::sequence<int>::type ints;
nonstd::sequence<callback *>::type callbacks;
```

One of the most common questions from novices coming to C++ from scripting and dynamically typed languages is how to convert between an int and a string. The C programmer's answer is to resort to the atoi and sprintf functions. This is fine as far as it goes... except for the irregularity, lack of safety and lack of extensibility to embrace other, similar type conversions. The C++ librarian's solution is to use stringstream, which performs in-memory I/O formatting but takes a few lines to do it. A solution currently in the Boost library uses – but encapsulates – stringstream, and presents a familiar cast-like interface to the user:

```
int i = lexical_cast<int>("42");
std::string s = lexical_cast<std::string>(42);
```

A more advanced platform

In addition to the common requests for simple extension such as the typeof and template typedef, the obvious omissions like hash tables, and the novice FAQs such as lexical_cast, consideration is also being given to adding more advanced platform features to the library.

Bjarne has proposed, among other features, that threads, remote procedure calls (RPCs) and extended type information (XTI) should be considered. The aim isn't to create a native C++ threading or RPC mechanism, but to provide a standard interface to facilities that commonly exist – Win32 threads or Pthreads, Corba or COM(+). Certainly, threading is one of the most commonly discussed for standardisation. The main challenge isn't whether it can or should be done, but which one of the many C++ threading libraries represents the most appropriate model for standardisation. XTI would provide C++ with a more comprehensive reflection model than its current RTTI system, and perhaps one that could be used in conjunction with any proposed RPC model.

Beman Dawes, one of the leading figures in Boost, conducted a simple survey at ACCU's Spring Conference this year to gauge interest in library extensions. Not surprisingly, multithreading scored highly, as did better date-time facilities, regular-expression matching, network programming, a proper numerics library, better string facilities, directory access and simple serialisation. Some of these features can be considered common requirements that are increasingly in demand and found in recent languages and platforms, such as Java and .Net, as well as in raw API form, such as in C, whereas other features are more particular to C++. What is interesting is that a common GUI API wasn't found to be high on developers' wish lists.

Another place where idealism could jump the tracks is the laudable desire to remove embarrassments. Not everything in the language is perfect, but some things are buggier than others

C++'s evolution will continue to be affected by a cross-platform outlook that ensures that it doesn't become any more involved with fashion than is strictly necessary. Some jobs are better done by proprietary or open source libraries and do not belong genuinely in the standard for a language. Others, however, are common and general enough that they can be considered standard utilities. The good news for the standard is that such utilities form a domain with a lot of existing practice to standardise. ■

References

1. *International Standard: Programming Language – C++*, ISO/IEC 14882:1998(E), 1998
2. Kevlin Henney, "The Miseducation of C++", *Application Development Advisor*, April 2001
3. Herb Sutter, "C++ Conformance Roundup", *C/C++ Users Journal, April 2001*, www.cuj.com/roundup
4. The International Organisation for Standardisation, www.iso.ch
5. The Association of C & C++ Users, www.accu.org
6. Standard Template Library Programmer's Guide, www.sgi.com/tech/stl
7. The Boost libraries, www.boost.org
8. Margaret A Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990
9. The Blitz++ library, www.oonumerics.org/blitz