
ORACLE®

ORACLE®

The new C++ Standard and Library (C++0x)

Paolo Carlini

ORACLE®
CONSULTING

The recent history

- After the 1998 Standard, the ISO C++ Committee remained essentially in “bug-fix mode” until about 2003, when Technical Corrigendum 1 (TC1) has been issued
  - C++98 + TC1 is informally known as C++03
- Afterwards, as you can also read in some of Bjarne Stroustrup papers back then, the plan was essentially working on new *pure library* facilities, which had suffered in the first Standard, with many last minute changes and rejections
  - eg, no hashed containers
  - nothing about threads, concurrency

The recent history (2)

- That attitude resulted first in a Technical Report, known as TR1 (paper #: N1836), issued in 2005, including many useful additions
  - Almost fully implemented in GCC
- After that, however, the plans changed (I was not there to report why and how, sorry...) and new core language features entered the discussions of the “evolution” subcommittee.
- The idea of a completely new standard became known as C++0x, meaning that people wanted to have it within the first decade of the 21th century, about 10 years after the first one...

The recent history (3)

- Unfortunately, no serious milestones set, no feature freezes, lots and lots of new reasonable (and much less reasonable ;) core language proposals over the years, until 2007!
- People realized about a year ago that, given all the bureaucracy needed for the last steps, there were no real hopes anymore to have the new standard ready by 2009: in fact, it will be C++1x.
  - Potentially interesting facilities recently dropped: garbage collection, modules, filesystem library, ...
  - Some of those already scheduled for TR2
  - (More or less) official statement of feature completeness

The final steps toward the new standard

- … but now finally we are almost there! March, 26th, on the iso-all mailing list:

  "This morning, the FCD text was completed by our tireless project editor Pete Becker, approved by the review committee of Steve Adamczyk and Howard Hinnant, and sent to SC22 for FCD ballot. The CD1 record of response was also delivered by Barry Hedquist on Wednesday. The SC22 secretariat has confirmed receipt of both required documents, and has informed us that the FCD ballot will begin today and close on July 26."

The final steps toward the new standard (2)

- Next step after the FCD (Final Committee Draft) will be the FDIS (Final Draft International Standard)
- Reasonably, the actual C++1x Standard will be published in 2011, but the technical work towards it will end in 2010.

- Some useful references:
  - http://www.research.att.com/~bs/what-is-2009.pdf
  - http://www2.research.att.com/~bs/C++0xFAQ.html
  - http://www.research.att.com/~bs/hopl-almost-final.pdf

The most important new features?

- Many, even counting all the ideas lately dropped or delayed to TR2 and in particular the recent removal of the famous “Concepts”:
  - Arguably, the language is growing very big, huge, too huge
  - The FCD is about 1300 pages vs 783 pages for the C++03 Standard. The library sections alone are almost three times as large.
  - As happens, big companies with representatives in the Committee pushed for idiosyncratic requirements ;)
  - Luckily, people like Bjarne Stroustrup still care *a lot* about teachability and comprehensibility and try to keep the situation under control

The most important new features? (2)

- A subset of changes, in the core language and in the library, just standardize existing and well established practice
- Typical examples in the core language:
  - Extern template
  - decltype (GCC's typeof, improved, preserves references)
  - long long
  - namespace association (also called “strong using”)
    - first implemented in GCC and heavily used by libstdc++-v3 for its debug-mode and parallel-mode
  - C99 preprocessor
  - __func__
  - general attributes, *thread local storage ...*

The most important new features? (3)

- Typical examples in the library:
  - unordered (ie, hashed) containers
  - C99 compatibility
  - singly linked list (very close to the legacy HP / SGI slist)
  - additional algorithms (also already in the “STL”)
  - enable_if

- These changes are generally considered all very welcome and uncontroversial

The most important new features? (4)

- Another rather uncontroversial class tries to avoid unnecessary inconveniences and limitations. Some examples for the core language:
  - default template arguments for function templates
  - variadic templates
  - right angle brackets
  - forward declaration of enumerations
  - local and unnamed types as template arguments
- And for the library:
  - specify header dependencies
  - simple numeric access (ie, beefed up atoi, strtol, & co)
  - improved const-correctness everywhere
  - generalized constant expressions

The most important new features? (5)

- Performance is still on the forefront today as it was 10 years ago, and another subset of changes has strictly (or largely) to do with it. Eminent examples:
  - rvalue references and “move semantics”
  - less restrictive characterization of POD-ness
  - placement insert for containers
  - Improved, so called “scoped”, allocator model
- With minor reservations for placement insert, IMHO all great and uncontroversial improvements
  - but rvalue references are conceptually *highly* non-trivial (more later in this presentation)

The most important new features? (6)

- A separate class for some “fancy” new features in the core language:
  - Lambda expressions and closures
    - Apparently an often requested improvement, directly inspired from functional programming
    - Delivered in GCC 4.5.0 (and MSVC 10)
  - auto (ie, deducing the type of variable from its initializer expression)
    - especially convenient with iterators and containers
  - initializer lists
  - template aliases (“template typedefs”)
  - for-loop
  - delegating / inheriting constructors

The most important new features? (7)

- … and another for useful additions / improvements to the runtime library
  - Everything coming straightly from TR1
    - with important improvements too, see the case of <random>, coming directly from Fermilab' in the field experience with huge “Monte Carlo” computations
  - unique_ptr (replacement for auto_ptr)
  - minimal unicode support / new character types
  - iostream / locale improvements and fixes of long standing issues
    - eg, parsing of integer and floating point types, satisfactory diagnosis of overflow situations

... and the GCC effort

- A few active committee members are implementing the new features in GCC (eg, for some time Doug Gregor, Jonathan Wakely, me, Jason Merrill, more).
- Detailed web pages track the evolution of the so-called C++0x mode of GCC (will become an alias for C++1x mode, of course), the reference one being, for core compiler and library features, respectively:

  http://gcc.gnu.org/projects/cxx0x.html

  http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.200x

... and the GCC effort (2)

- Microsoft people liked the GCC way of presenting the implementation status ;)

  (http://blogs.msdn.com/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx)

- For the time being all the C++1x core language and library features exist in parallel with the default C++98 ones, no ABI breakages, that of course means some code duplication and “dirty” preprocessor tricks, but that's life
  - Note that the Committee only recently realized that eventually most implementations will be necessarily forced to break the C++98 ABI, because, eg, std::list::size now is constant time, or std::string cannot be reference-counted anymore.

Move semantics in some detail

"Move semantics is mostly about performance optimization: the ability to move an expensive object from one address in memory to another, while pilfering resources of the source in order to construct the target with minimum expense"

(From N1377, Hinnant, Dimov, and Abrahams)

Move semantics in some detail (2)

- Move semantics ideas already exists in the current C++03 language and library, to a *certain* extent:
  - Copy constructor elision in some contexts
    - Aka, NRVO, Named Return Value Optimization
  - auto_ptr “copy”
    - Special non-const reference constructor – i.e., auto_ptr(auto_ptr& a) - which takes ownership
  - list::splice
    - “Copy” of elements from list to list in O(1) via simple pointer adjustments
  - Swap on containers
    - Swap specialization able to deal with the whole container by swapping the pointers to the underlying data structure

Move semantics in some detail (3)

- In order to support a general use of such ideas, a *new* kind of reference is needed, able to bind to temporaries, an rvalue reference (vs lvalue reference):

```
void foo(A& t);          // Cannot bind to a temporary
void foo(const A& t);    // Can't change it


void foo(A&& t);   // Yes! foo can steal the resources
                   // owned by the temporary t
```

Move semantics in some detail (4)

- Then rvalue references can be used to *implement* move semantics, e.g., by *adding* a move constructor and a move assignment operator to a class:

```
class A
{
   // …

   A(A&& a);
   A& operator=(A&& a);
};
```

Move semantics: (toy) string example

```
class string
{
   char* data; size_t size;
public:
   // …

   string(string&& s)
   : data(s.data), size(s.size) // Pilfering!
   { s.data = 0; s.size = 0; }  // NB: s left
                                // consistent
                                // for sane
                                // destruction
   string& operator=(string&& s)
   { swap(s); return *this; }   // Pilfering!
```

Move semantics: (toy) string example (2)

- The extremely efficient move constructor and move assignment operator are *automatically* called, instead of the normal copying ones, when a temporary is involved.
- Actually, *much more* is possible. Consider:

```
string s0("1234567890");
string s1 = ((s0 + "a") + "b") + "cd";
```

  - Normally, each operator+() allocates memory (NRVO helps only for the copy constructor itself)...

Move semantics: (toy) string example (3)

- Overloads of operator+() for rvalue references can be added, *appending to temporaries*, e.g.:

```
string&&
operator+(string&& x, const string& y)
{ return x += y; }
```

- In the example, if the temporary created for `s0 + "a"` has sufficient capacity, memory is allocated only *once*!

Move semantics: std::vector example

- All the non-node-based containers can also exploit the move-ability (i.e., availability of move constructor and move assignment operator) of a type in the internal *implementation* details of operations such as *insert* and *erase*.

- In that case, the internal memory management operations can just *copy pointers* to data, instead of copying the actual raw data

Move semantics: std::vector example (2)

- *The core idea enabling such optimizations is that when it's safe to pilfer from a data source the implementation can explicitly cast it to its rvalue reference type - std::move is available for this purpose - and automatically activate move constructor and move assignment operator on it thereafter*

- Then, just as an example, user code like:

```
string s(1000, ' ');
vector<string> v(1000, s):
v.insert(v.begin(), s); // ~100 times faster!
```

Move semantics: std::vector example (3)

- More generally, types like `vector<list<int> >`, that is a vector (a non-node-based container) of lists (node-based containers), can also exploit the move-ability of the inner type during the internal memory management operations.
- Further improvements are possible (push_back operations, etc...)
- Much more...

- *By the way, all of this is already implemented in the GCC runtime library, libstdc++-v3*

Move semantics: algorithms

- Several of the std algorithms can also take advantage from move semantics
  - Either because temporary buffers can benefit from it (e.g., stable_partition, stable_sort, etc.)
  - Or, trivially, because the semantics really calls for moves not copies (e.g., remove, unique, sort)
    - For instance, unique can be changed to exploit move semantics by simply adjusting one line to use std::move. The GCC implementation looks like the below (slightly simplified for expositional purposes):

Move semantics: std::unique example

```
template<typename _ForwardIter>
  _ForwardIter
  unique(_ForwardIter __first, _ForwardIter __last)
   {
      // Skip the beginning, if already unique
      // ...
      _ForwardIter __dest = __first;
      ++__first;
      while (++__first != __last)
        if (!(*__dest == *__first))
           *++__dest = std::move(*__first);
      return ++__dest;
   }
```

Conclusions

- Let's stop here.
- Remember: your feedback is important, please take some time to read the papers or the FCD, get in touch with the authors. Constructive feedback is always welcome, nobody in the committee wants to deliver a defective standard!
- In libstdc++-v3 many features are early available for practical experimentation, send your observations to

  libstdc++@gcc.gnu.org
- ... or simply to me ;)

  paolo.carlini@oracle.com

Thanks!