Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Eric Christopher, *Red Hat, Inc.*
David Edelsohn, *IBM*
Richard Henderson, *Red Hat, Inc.*
Andrew J. Hutton, *Steamballoon, Inc.*
Janis Johnson, *IBM*
Toshi Morita
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Al Stone, *HP*
Zack Weinberg, *Codesourcery*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

“The C++ library is being enhanced”

Paolo Carlini
SUSE Labs / Novell
pcarlini@suse.de

Abstract

This contribution summarizes the contents of TR1 (Technical Report 1) and the rationale behind the most relevant additions to the standard in force: components for creating and manipulating function objects, support for metaprogramming techniques and unordered containers, are some of the items described in better detail. In particular, it is discussed how those facilities have been added to the C++ run-time library delivered as part of GCC 4.0: the project experimentally provides a large subset of the new features, both to the benefit of the users and contributing to the ongoing standardization process. Moreover, the present work reviews the most important open issues and a few topics at the interface with core language and GNU C++ front-end development, such as alignment attributes in templates, prospective techniques dealing with macro-related troubles and variadic templates, which surfaced, often unexpectedly, during the first months of the effort.

1 Introduction

To date, the home page of ISO WG21[ISO] still shows the headline that “The C++ library is being enhanced.” The news dates back to May 2001, when, at the Copenhagen meeting, the international standardization working groups J16 and WG21 passed a motion to request a new work item: a technical report of type 2 on standard library extensions. Four years later, the document—known as TR1—is being finalized, its latest incarnation being draft N1745.

In practice, this means that almost certainly the described items will be part of the next C++ Standard—C++0x—with possible minor modifications dictated by consistency with core language changes: most of the proposals in TR1 stem by and large from existing practice [Boost], relying only on features provided by the current language. Other libraries will be considered for C++0x inclusion besides those already present in TR1: October 2005 is the cutoff date for major proposals. A few additional proposals in the area of system programming have a very good chance to be accepted [Stroustrup05], and the same is true for the library part of the exciting “move semantics” novelties (see N1771, N1770, and N1690/N1377 for background).

This contribution, however, concentrates on library additions already voted as part of TR1, with particular attention devoted to the components part of the C++ runtime library delivered with GCC 4.0: smart pointers, facilities for the creation and manipulation of function objects, support for metaprogramming techniques and unordered containers are some examples. Moreover, the work reviews the most important open issues and touches upon a few topics at

the interface with the core language and GNU C++ front-end developments, such as alignment attributes in templates, prospective techniques dealing with macro-related troubles and variadic templates, which came to the fore, often unexpectedly, during the first months of the implementation effort.

2 What's in TR1?

Let's start by summarizing the content of each chapter of TR1, together with the number of the original proposals, which often discuss enlightening background material and provide the rationale for each addition[1]:

- Chapter 2 – General utilities
  - **Reference wrappers (N1453)**
  - **Smart pointers (N1450)**
- Chapter 3 – Function objects
  - **Function return types (N1454)**
  - **Member pointer adaptors (N1432)**
  - **Function object binders (N1455)**
  - **Polymorphic function wrappers (N1402)**
- **Chapter 4 – Metaprogramming and type traits (N1424)**
- Chapter 5 – Numerical facilities
  - Random number generation (N1452)
  - Mathematical special functions (N1422)
- Chapter 6 – Containers
  - **Tuple types (N1403)**
  - **Fixed size array (N1479)**
  - **Unordered associative containers (N1456)**
- Chapter 7 – Regular expressions (N1429)
- Chapter 8 – C compatibility (N1568)

Items in boldface are provided in GCC 4.0, with various levels of completeness and solidity[2] and are further discussed in the remaining part of this section, with particular attention to the problems encountered during the implementation.

2.1 Smart pointers

A shared-ownership semantics smart pointer is probably one of the most requested additions, and, among many other interesting uses, is the canonical solution for storing polymorphic objects in containers (Figure 1).

```
class B
{ /* ... */ };
class D : public B
{ /* ... */ };
...
typedef shared_ptr<B> Spt;
vector<Spt> vp;
Spt b(new B);
Spt d(new D);
vp.push_back(b);
vp.push_back(d);
```

Figure 1: Vector of polymorphic objects

[1] Beware, however, that only the most recent draft, N1745 to date, constitutes the reference for all the technical details. The same remark applies to documentation coming with *Boost* libraries, often very useful but not necessarily consistent with the facilities in TR1.

[2] Maybe this is the proper place to remind again, as done in the release notes, that this is experimental material and no guarantees are provided.

At the same time is one of the hardest to implement correctly: exception safety, thread safety are very difficult to get right. In short, there are very good reasons to have it standardized in the library. In addition to class `shared_ptr`, `tr1/memory` provides also `weak_ptr`, which stores a "weak reference" to an object already managed by `shared_ptr`. The latter allows functions or classes to keep references to objects without affecting their lifetime. As convincingly argued in the original paper, it's important to have available both, to cover all important shared-ownership use cases and avoid memory leaks with cyclical data structures, inescapable with `shared_ptr` alone due to its reference-counted nature.

The latter issue brings naturally to delicate design choices and to GNU's TR1. When Jonathan Wakely ported the reference Boost implementation (due to Peter Dimov, Beman Dawes and Greg Colvin) to libstdc++-v3, it became quickly clear that parts of the project were not ready for the "concurrency revolution" [Sutter05]: around March-April of this year, when the Boost project was already moving to very efficient "lock free" solutions (for an introduction, see [Alexandrescu04]), embarrassing disuniformities surfaced in the quality of the atomicity primitives across the various targets: e.g., missing memory barriers, etc. Many of these troubles can be obviously avoided if a large set of compiler builtins for atomic memory operations is available, easily usable in the library. Indeed, this is now the case in mainline (i.e., for GCC 4.1): Richard Henderson added a complete set of builtins modeled after the operations defined by Intel for IA64 and target bits are already done for x86, ia64, alpha, powerpc. This means that very soon any concurrency paradigm will be easily at reach; in the meanwhile, in the 4.0.x time frame, the open coded atomicity primitives in the library are audited and opportunistically fixed.

2.2 Reference wrappers and function objects

Over the years, many founded complaints have been advanced about facilities in `<functional>`, see, f.i., [Meyers01]: TR1 both adds powerful generalizations of existing facilities and also offer new ones [Becker05], summarized here as follows:

reference_wrapper A CopyConstructible and Assignable wrapper around a reference to an object. Implicit conversion to a reference is provided, and two functions, `ref` and `cref`, return instances of `reference_wrapper` (Figure 2); it can be used where ordinary references cannot, e.g., in containers.

```
void f(int& r) { ++r; }

template<class F, class T>
  void g(F f, T t) { f(t); }

int i = 0;
g(f, i);
cout << i; // 0
g(f, ref(i));
cout << i; // 1
```

Figure 2: `tr1::ref` by example (from N1453).

function A powerful generalization of the notion of function pointer, subsuming function pointers, member function pointers and arbitrary function objects [Sutter03.1]. In Figure 3 are shown a few simple examples, a glimpse at its flexibility and facility of use.

mem_fn Can be considered a generalization of `std::mem_fun` and `std::mem_`

```
int add(int x, int y)
{ return x + y; };

function<int (int, int)> f;
f = &add;
cout << f(2, 3); // 5
f = minus<int>();
cout << f(2, 3); // -1

function<bool (int, int)> g;
g = equal_to<long>();
// argument conversion ok
```

Figure 3: `tr1::function` in action (from N1402).

`fun_ref`, supporting pointers to member functions with an arbitrary number of arguments (subject to implementation-imposed constraints[3]) and pointers to data members. Returns a function object accepting a reference, a pointer, or an iterator as its first argument: this means that, for example, all the cases discussed in [Meyers01] can be elegantly covered with something as simple as Figure 4.

```
template<class It, class R,
         class T>
  void
  for_each(It first, It last,
           R T::* pm)
  {
    for_each(first, last,
             mem_fn(pm));
  }
```

Figure 4: `tr1::mem_fn` solving Meyers problem (from N1432).

bind In turn, can be seen as a generalization of `std::bind1st` and `std::bind2nd`, again supporting arbitrary function objects, functions, member function pointers, etc: see Figure 5 for some introductory examples, also involving the interesting _X syntax for the placeholder arguments.

```
int f(int a, int b)
{ return a - b; }

int four = 4;
int six = 6;

cout <<
  bind(f, 3, _1)(four); // -1

cout <<
  bind(f, _2, _1)(four, six); // 2
```

Figure 5: `tr1::bind` and its placeholders (adapted from our testsuite).

By now should be sufficiently obvious that fulfilling completely this kind of requirements (only partially sketched above) asks for a lot of ingenuity. Indeed, the GNU implementation has been contributed by Doug Gregor, one of the most active members of the *Boost* project and regular of the ISO C++ meetings. Still, the first draft of the `bind` function template stressed the compiler to the point that each testcase required *tens* of seconds to build! Compile-time performance bottlenecks not encountered before in the name lookup area[4] became important, and only with a concerted effort of a few days, partially touching the compiler too—much more work is needed—a decently fast version has been prepared in time for GCC 4.0.

Before moving to the next section, an hint to a vast topic, that cannot be really discussed here, but is very important for the programmer. These sophisticated abstractions have non trivial costs, both in terms of performance and

[3] At least 10, according to Annex A, but see below.

[4] See `http://tinyurl.com/ahno6`.

of space: for instance, in the GNU implementation a `function` object is typically the size of four pointers and may require two calls through function pointers. In the older *Boost* version different trade-offs have been preferred and things may further change in the future as the implementors receive more feedback from the actual users of the library.

2.3 Metaprogramming and type traits

The presence of support for metaprogramming techniques could be largely expected considering the growing importance of such topics during the last years [Abrahams05]. In fact, only the most basic facilities are included, under the name of `type_traits`. The set of categories, properties and relations is however very large and includes:

- A rather complete set of primary categories, e.g., `is_void`, `is_integer`, `is_pointer`, `is_function`, etc.
- Likewise for composite categories, for instance, `is_arithmetic`, `is_object`, `is_scalar`, etc.
- Type properties, e.g., `is_const`, `is_polymorphic`, `is_abstract`, `alignment_of`, `rank`, etc.
- Relationships between types, like `is_convertible` or `is_base_of`.
- Modifications of array types, reference types, const–volatile qualifiers, pointer types, and an aligned_storage class.

Figure 6 shows an example involving `align_of` and `aligned_storage`, maybe two among the less obvious facilities, that actually fulfill a very common need.

```
const size_t a =
  alignment_of<int>::value;

typedef aligned_storage<8, a>::type
  a_type;
```

Figure 6: Introducing `tr1::align_of` and `tr1::aligned_storage`: `a_type` is a type suitable for use as uninitialized storage for any object whose size is at most 8 and whose alignment is a divisor of `a`.

An interesting feature of `type_traits` is that its members are either a delight to implement—consider, for instance, a typical, recursive implementation of `rank` or the mildly sophisticated and instructive applications of "SFINAE" [Vandevoorde03] in other cases—or next to impossible to program in pure C++ without compiler support—consider `is_union` (vs `is_class`) or the various `has_*` properties. In the face of this, implementors are *temporarily* granted a latitude to implement some features only partially or in a weak form. Consider, f.i., `is_pod<T>`: it's only required that it evaluates to true for T irrespective of cv-qualification, irrespective of being T element type of an array, and always when `is_scalar<T>` is true or `is_void<T>` is true.

As far as the GNU effort is concerned, having clarified that we are indeed availing ourselves of the above mentioned latitude, a few other remarks are in order. First, once more, the C++ front-end has been stressed in rather uncommon directions: for example, to satisfactorily implement `is_member_function_pointer` and `is_member_object_pointer`, a rather obscure bug/missing feature surfaced[5] and had to be fixed anyway for the sake of `bind`. Likewise, `aligned_storage` emphasized

[5] One of our "bug masters" even considered closing it as "INVALID."

long standing weaknesses of attributes in templates in general, and alignment attributes in particular—see, for example, c++/17743 and c++/19163. Currently a workaround specializes `aligned_storage` for alignments in the range from 1 to 32, but a much cleaner and general implementation is certainly possible in principle, only delayed to better times.

A final note about the potential usefulness of `type_traits` for implementing other chapters of the standard library: after all, our `cpp_type_traits.h` provides a very small subset of those functionalities and is now arguably redundant. Unfortunately, section 1.3.3 of TR1 reminds us of annoying issues with user macros that would spoil the standard conformance of the implementation. Luckily, as will be briefly discussed below, there are hopes that these annoyances will be soon fixed at the core language level.

2.4 Tuple types

One the first two proposals accepted for TR1, in October 2002. Tuples are already present in several other languages (e.g., Python, Haskell, ML, etc.) and can considered a straightforward generalization of `std::pair`. Getting oneself acquainted with `tr1::tuple` is therefore easy [Sutter03.2] and, compared to pairs, it offers handy facilities, like `tie`, that allows to unbundle in a single statement all the elements to separate variables (Figure 7). Also, types can be accessed via `tuple_element`, overloads of the streaming operators `<<` and `>>` are provided, and more.

Interestingly, however, in N1403, Jaakko Järvi pointed out right at the beginning that, whereas `tuple` can be implemented without requiring any core language change, this requirement was part of the TR1 call, it would benefit from a *set* of changes, like variadic templates (N1704),

```
tuple<int, double> t(1, 1.0);
int    first;
double second;
tie(first, second) = t;
```

Figure 7: Unbundling a `tr1::tuple`

templated typedefs, default template arguments for function templates, and making a reference to a reference to some type T equal to a reference to T[6]. Once more, becomes clear that often implementing the powerful abstractions of TR1 is not at all straightforward and pushes the language to its limits. Chris Jefferson, rewrote at least two times our `tuple`, testing various trade-offs between compile-time performance, size of the headers and flexibility. In particular, without variadic templates, one is forced to duplicate a lot of template code for a number of arguments varying from one up to a maximum—at least 10, according to TR1: the issue can be hidden somehow but can only fundamentally solved by language changes.

2.5 Fixed size array

This is a very simple wrapper, a drop-in, efficient and safe replacement for a traditional array. A few basic observations, based on the original proposal:

- Constraints on the design: must be implemented as an aggregate type in order to support initializer syntax:

  ```
  array<int,4> a = {0, 1, 2, 4};
  ```

- Stack allocation is typically more efficient and safer than dynamic allocation.

[6] Actually, this may happen soon as part of the "move semantics" proposals, see in particular the section "Reference to References" of N1377.

We have a special affective attachment for this simple container, the very first bits of GNU's TR1, back in October, 2004.

2.6 Unordered associative containers

This is definitely a very welcome addition: hash tables were proposed for the C++ standard back in 1995 and the proposal rejected for reasons of timing. Finally TR1 includes hash-based counterparts of the `map` and `set` containers, to wit

- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

For most operations the worst-case complexity is linear but the average case is *much* faster. As part of the proposal, Matt Austern discussed the existing implementations (as extensions), in particular the rather popular GNU version, inherited from the original SGI code and currently provided in namespace `__gnu_cxx`. The following is a partial[7] list of interesting differences:

- Among the specialization of the `hash` function object, two are provided for `string` and `wstring`.
- A complete set of load factor[8] control facilities:
  - `float load_factor() const`
  - `float max_load_factor() const`
  - `void max_load_factor(float)`
- A so-called *bucket interface*, that exposes the bucket structure:
  - Overloading of `begin()` and `end()`: in `n` is an integer, `[begin(), end())` is a range of iterators pointing to the elements in the `n`th bucket.
  - `begin(n)` and `end(n)` return iterators of type `local_iterator` (or `const_local_iterator`).

The bucket interface was probably the most controversial part, not present in any of the existing implementations of hashed containers: the provided rationale is an interesting reading, in particular about its usefulness for investigating how well the hash function performs, which reminds the `use_count` member of `shared_ptr`, also meant for debugging and testing purposes.

In the GNU implementation, provided by Matt Austern himself, names are not uglified (i.e., are not prefixed by underscores), not deemed necessary as long as the classes belong experimentally to namespace TR1 and are not part of the actual C++ standard—some members of the project also consider that auspicious for the solution of the already mentioned annoying issues with macros. However, this caveat is necessary for users willing to try immediately the new features together with pre-existing code.

3 (More) open issues

This section returns to a few themes already touched upon in the first part of the paper, to provide additional details, missing references and, in some cases, solicitate appropriate actions.

[7] Also, pure "bug" fixes will not be further discussed, f.i., `insert(iterator, const value_type&)` missing from the SGI extension.

[8] Defined as the average number of elements per bucket.

3.1 `type_traits` vs reflection

As already observed in Section 2.3, parts of `type_traits` are impossible to implement without compiler support. Whereas these days there is a lot interest in "reflective" techniques and libraries (see, f.i., N1775 and N1751), there are no plans to provide such facilities as part of C++0x. Therefore, in order to completely implement the final form of `type_traits`—when the special latitude temporarily granted in Section 4.9 of TR1 will be removed—an effort will be needed in the GNU C++ front-end: very likely, it could take the form of additional builtins, similar to the current __align_of__, used by the library to ascertain properties of types, especially class types. More generally, during the recent years we are learning that in many other areas of the runtime library, builtins are maybe not strictly necessary but often *very* useful to improve the QoI from various points of view and increase the chances for optimizations: besides the atomic primitives, already mentioned in Section 2.1, mathematical functions are also an important example.

3.2 Macro scopes and modules

In two different circumstances, use of `type_traits` in other areas of the library and unordered containers, annoying issues with macros have been mentioned. Recently, Bjarne Stroustrup in person presented a proposal, N1614, for a new, simple scoping mechanism for the preprocessor, see Figure 8 for a practical explanation of its basic features. In a nutshell, a mechanism of "macro scopes" is introduced, useful to isolate code inside the scope from code outside.

Another concrete possibility is that *modules* will be part of the next standard: at the Lillehammer meeting, David Vandevoorde officially presented for the first time a scheme for

```
#define A 9
#define B 10

#scope
int A = 7;   // ok
#define B 7; // ok
#define C 99;
#endscope

int y = B; // 10
int z = C; // error: C undefined
```

Figure 8: `#scope` (aka `#nospam`) in action (from N1614)

```
namespace << std;
int main()
{
  std::cout << ''Hello!'';
}
```

Figure 9: An "Hello!" from the future (from N1778).

modules in C++, nicely summarized in paper N1778. Modules would be completely shielded from macros: in Figure 9 a module namespace named `std`, encapsulating the library, is imported and made available to the translation unit. Macros possibly defined in the latter cannot reach into `std`.

4 Conclusions

Exciting days for C++! Ahead of us there is a lot of hard work but also a better programming language and, especially, a better library. The GNU project is not badly positioned in the context of this effort but certainly help is immediately needed to complete the implementation of TR1—some of the missing chapters are very big—and keep on tracking the standardization effort. Beyond TR1, other large chunks

of work will impact the C++0x library. A cursory list would mention: an expected overhaul of iterators, allocators, the library half of "move semantics," and of course *concepts* (see, e.g., N1782 and N1799). But that is really another story...

Acknowledgments

To NIN for WITH_TEETH, the soundtrack of this paper, and to the contributors of GNU's TR1.

References

[ISO] All the papers cited by number in this work can be freely accessed from: `http://www.open-std.org/jtc1/sc22/wg21`.

[Boost] `http://www.boost.org`.

[Stroustrup05] B. Stroustrup, *The Design of C++0x*, C/C++ Users Journal, Volume 23, Number 5, 2005.

[Sutter05] H. Sutter, *A Fundamental Turn Toward Concurrency in Software*, Dr. Dobbs Journal, #370, March 2005.

[Alexandrescu04] A. Alexandrescu, *Lock-Free Data Structures*, C/C++ Users Journal, Volume 22, Number 10, 2004.

[Meyers01] S. Meyers, *Effective STL*, Item 41, Addison-Wesley, 2001.

[Becker05] P. Becker, *C++ Function Objects in TR1*, C/C++ Users Journal, Volume 23, Number 5, 2005.

[Sutter03.1] H. Sutter, *Generalized Function Pointers*, C/C++ Users Journal, C++ Experts Forum, August 2003.

[Abrahams05] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming*, Addison-Wesley, 2005.

[Vandevoorde03] D. Vandevoorde and N.M. Josuttis, *C++ Templates*, Addison-Wesley, 2003.

[Sutter03.2] H. Sutter, *Tuples*, C/C++ Users Journal, C++ Experts Forum, June 2003.