Fixing freestanding: iteration 2.2

document id: N2932=09-0122

date: 2009-07-17

author: Martin Tasker (martin.tasker@nokia.com)

context: baselined on pre-Frankfurt C++0x WD, N2914 2009-06-22, on pre-Frankfurt LWG issues D2894 2009-06-16, and on post-Summit CWG issues N2860 2009-03-23.

Acknowledgements: thanks to Beman Dawes; Alisdair Meredith, Chris Jefferson, LWG and CWG for comments and suggestions.

1. Introduction

This paper is a third iteration of N2814, *Fixing Freestanding*, submitted to Summit.

This proposal aims to:

- enable the intent of the freestanding specification [compliance] to be preserved, by adding only two headers to it relative to its C++98/C++03 contents, namely `<initializer_list>` and `<type_traits>`
- maintain the historically low dependency of the C++ language, and of C++ compilers, on a corresponding library

The benefits of this proposal are:

- it enables current users of the freestanding specification to continue using that specification
- it enables compiler implementers to work fairly independently of library supply
- it enables library implementers to work fairly independently of compiler supply

Compared with N2814, the second revision, N2886:

- re-baselines on the post-Summit WD, recent issues documents, and NB comments
- omits consideration of range `for`, since this is handled adequately in Beman Dawes' N2872, *Ensuring Certain C++ Features "just work"*, issued in the 2009-05 mid-term mailing.
- omits issues with lambda, since these were addressed in the post-Summit redrafting of lambda in N2859, *New Wording for C++0x Lambdas*
- includes consideration of `<type_traits>`, as suggested at Summit

Compared with N2886, this third revision:

- re-baselines on pre-Frankfurt WD
- omits references to Concepts
- changes the recommendation in relation to fixing array `new` behaviour

Related NB comments, language and library issues are flagged in the text.

This paper considers, but omits, changes motivated by the addition of other headers of interest in embedded contexts, eg `<array>` and `<ratio>`, as suggested at Summit. This is not addressed here, because it's a different issue from the one the paper is addressing (essentially about minimalism), and is provided for anyway in the definition of freestanding in [compliance], which says that a freestanding implementation should deliver "*at least*" the listed headers. If further work is required on this topic it could be done in a separate paper.

1.1. Why?

The C++0x WD specifies new C++0x features which impact the freestanding specification and increase the dependency of the language on the library. Such features include:

- array `new` expressions [expr.new], which as proposed bring in `std::string` as a language dependency.

- usability of `typeinfo` in containers [support.rtti], which as proposed brings STL containers into `<typeinfo>` and thus into freestanding use of type information
- initializer lists [support.initlist], which one the one hand are fundamental to C++ and yet are not reflected in the freestanding specification; and which on the other hand as proposed require the `Range` concept even though an initializer list is just an array with a length
- type traits, originally from TR1, a language feature which is offered to programmers either vai an unspecified compiler interface, or the specification in `<type_traits>` which is not reflected in the freestanding specification

The C++0x WD is inconsistent in that, while these new features are proposed with the impacts stated above, the headers required by the freestanding definition in [compliance] are unchanged since C++98 (sic).

1.2. How?

This paper proposes small changes, which preserve the intent on the one hand of the new features listed above, and on the other hand of the freestanding definition and its corollary, a low dependency of language on library.

This is a middle course between two obvious alternatives:

- amend the freestanding specification, and the language/library dependencies, to fit the new C++0x features as specified. This would add very significantly to language/library dependencies, and would severely dilute the value of the freestanding specification for any party currently interested in it.
- delete those new features in C++0x which create the increased dependency of the language on the library. This would rob C++0x of some nice new features. Such an approach is really a non-starter.

1.3. What?

In summary the changes proposed are:

- array `new` [expr.new para 7]: change the proposed `std::length_error` (an exception type requiring a `std::string`) to `std::bad_array_new_length`, derived trivially from `std::bad_alloc` (an exception type requiring only a `char*`)
- `<typeinfo>` [support.rtti]: take the `type_index` and `hash<type_index>` definitions, which require `<functional>`, out of `<typeinfo>` and put them into another header `<typeindex>`, so that `<typeinfo>` (and therefore use of RTTI) doesn't depend on `<functional>`
- initializer lists [support.initlist]: add `<initializer_list>` to required freestanding headers, but have it define `std::initializer_list` only – reflecting the fact that `std::initializer_list` is fundamental in C++0x; accordingly, move the `concept_map` of `std::initializer_list` to `Range` into `<iterator_concepts>`; and have range-based `for` handle `initializer_lists` as a special case which doesn't require `<iterator_concepts>`
- type traits [meta]: add `<type_traits>` to required freestanding headers

2. Value of maintaining the freestanding definition

The freestanding definition, dating from C++98, originally enabled the C++ language to be used in:

- small systems, with a space-motivated reason to eliminate libraries as far as possible
- realtime systems, with functionally-motivated reasons to replace the standard (non-real-time) libraries
- proprietary systems, with other motivations (sometimes as prosaic as pre-1998 legacy) for doing their own thing in libraries rather than using the standard library

Things have changed since 1998, in particular the product categories which would count as "small systems", given more than a decade of exponentially plummeting memory costs. Nonetheless, the freestanding definition remains valuable. All the above reasons still apply in one segment or another of the software industry addressed by C++.

The freestanding definition is closely related to the question of dependency of language (and compiler) on library. A key architectural principle of C++ (inherited from C) has been to minimize this dependency. This principle was key to the approachability and portability of C and C++, and also to the adoption of C and C++ in non-mainstream systems. These considerations – though changed in detail – still hold: minimizing the dependency of the C++ language on its library remains valuable.

Note that practical freestanding systems may use any C++ standard library header they wish, such as <array>, <ratio> etc. This is already permitted by the standard, with the phrasing “at least” in [compliance], para 2, which also dates from C++98.

3. Language changes

3.1. Array new

Background

`operator new[]` is required – as before – only to throw `std::bad_alloc`.

However, a *new-expression* [expr.new] may throw a `std::length_error` (para 7), if the length is too long.

Current impact on freestanding definition

While `std::bad_alloc` uses only `char*` for its diagnostic, `std::length_error` brings in `std::string`: this implies a material extension to the freestanding library.

Outline proposal

Replace the `std::length_error` required in [expr.new para 7] with `std::bad_array_new_length`, trivially derived from `std::bad_alloc`.

Wording

Amend [expr.new para 7] as follows:

> When the value of the expression in a *noptr-new-declarator* is zero, the allocation function is called to allocate an array with no elements. If the value of that expression is such that the size of the allocated object would exceed the implementation-defined limit, no storage is obtained and the *new-expression* terminates by throwing an exception of a type that would match a handler (15.3) of type `std::`~~`length_error`~~`bad_array_new_length` (~~19.1.4~~18.6.2.x).

In the header <new> synopsis in [support.dynamic], add class `bad_array_new_length`.

Create a new sub section of 18.6.2 with the following text:

> 18.6.2.x Class bad_array_new_length [bad.arraynewlength]
>
> ```
> namespace std {
>     class bad_array_new_length : public bad_alloc {
>     public:
>         bad_array_new_length() throw();
>     };
> }
> ```
>
> The class `bad_array_new_length` defines the type of objects thrown as exceptions by the implementation to report an attempt to allocate an array of size greater than an implementation-defined limit ([expr.new]).
>
> `bad_array_new_length() throw();`
>
> *Effects*: constructs an object of class `bad_array_new_length`.
>
> *Remarks*: the result of calling `what()` on the newly constructed object is implementation defined.

Resulting Impact on freestanding specification

The net effect of this proposal is no material change in the dependencies of array new expressions, so that there is no impact on the requirements of the freestanding library compared with C++98.

Impact on C++0x

This proposal maintains the intent of the new C++0x feature – namely to provide a diagnostic for over-long array new requests.

Related issues

The issue raised here is the same as CD comment UK-72 / CWG issue 805, and the resolution is aligned with the suggestion in CWG issue 805.

4. Library changes

4.1. <typeinfo>

Background

Paper N2530 contains a simple proposal to make it possible to use `type_info` as an index to an associative container.

This impacts `<typeinfo>`, defined in [support.rtti], by

- bringing in a function `hash_code()` into `struct type_info`
- introducing `type_index` and `hash<type_index>`, which together depend on the header `<functional>`

Current impact on freestanding definition

This brings in `<functional>` to the set of required freestanding headers. Transitive closure on `<functional>` would introduce significant library bloat and/or implementation complexity.

Outline proposal

Split the implementation so that the functionality required by N2530 is delivered, but the freestanding definition isn't compromised:

- maintain the `hash_code()` function in `type_info` – its obvious implementation is so simple (use the `type_info`'s address) that there's no strong reason not to do this
- split the `type_index` and `hash<type_index>` definitions out of `<typeinfo>` and put them in another header, `<typeindex>`

Wording

In [support.rtti], amend the `<typeinfo>` synopsis as follows:

```
namespace std {
      class type_info;
~~      class type_index;~~
~~      template <class T> struct hash;~~
~~      template<>~~
~~      struct hash<type_index> : public~~
~~      std::unary_function<type_index, size_t> {~~
~~            size_t operator()(type_index index) const;~~
~~      }~~
      class bad_cast;
      class bad_typeid;
}
```

Move [type.index] "class `type_index`" from its present location at 18.7.2 into a new location at the end of clause 20 (20.11 unless some other text gets there first). Begin the relocated section with

Header <typeindex> synopsis:

```
namespace std {
      class type_index;
      template <class T> struct hash;
      template<>
      struct hash<type_index> : public
      std::unary_function<type_index, size_t> {
            size_t operator()(type_index index) const;
      }
}
```

and then include all headings and content brought over from the previous location.

In [headers], table 13, add `<typeindex>` to the list.

In [utilities.general], table 30, add

20.11	type indexes	<typeindex>

to the end of the table.

Resulting impact on freestanding specification

A single extra function to implement, compared with C++03.

Impact on C++0x

The functionality required by N2530 is still delivered. An additional header, <typeindex> is needed.

Related issues

CD comment UK-194 (still open after Summit) duplicates this proposal. Comment DE-17/LWG 1078 (open) clashes with it.

4.2. List initialization

Background

In list initialization, objects have a constructor with a sequence initializer, ie a final initializer_list<E> parameter. Their constructor can then iterate through the initializer list and construct algorithmically.

The type initializer_list<E> is defined in <initializer_list>, along with concept_maps to define Range on initializer_lists.

The type initializer_list, and the correspondingly indicated notion of a sequence constructor, have a special role in list initialization, which is a major convenience feature of the C++0x specification, bringing initialization in C++ onto a par with initialization in other languages.

Current impact on freestanding definition

std::initializer_list is fundamental to list initialization, and yet is a very simple type. To include list initialization in C++ requires <initializer_list>, containing the std::initializer_list type, to be added to the headers required for freestanding implementations, listed in [compliance].

The concept_maps to Range, if implemented as currently specified in <initializer_list>, would bring in <iterator_concepts> and corresponding baggage, which would adversely affect the definition of freestanding.

Proposal

Clearly the ability to use list initialization, and therefore sequence constructors, are fundamental to C++0x. Therefore, std::initializer_list, defined in <initializer_list>, is a fundamental part of C++0x: therefore, add <initializer_list> to the list of headers specified in the freestanding definition in [compliance].

Remove the concept_maps from initializer_list to Range from <initializer_list>, and place them in <iterator_concepts>, since this is where Range is defined. In the Standard, move the text specifying this concept map from [support.initlist] to [iterator.concepts.range].

Note: the usefulness of this proposal depends also on the special-casing of initializer-lists in range-based for described in Beman Dawes' N2872 *Ensuring Certain C++ Features "just work"*.

This would enable such syntax as:

```
for (int x : { 1, 2, 3, 4, 5 } ) { runTestCase(x); }
```

to work in freestanding C++, or in hosted C++ without including any header files (other than any required for runTestCase()).

Wording

To the list of required headers in [compliance], table 15, add

18.9 Initializer Lists <initializer_list>

In [support.initlist], in the class definition of initializer_list, delete

~~template<typename T>~~
~~concept_map Range<initializer_list<T> > see below;~~
~~template<typename T>~~
~~concept_map Range<const initializer_list<T> > see below;~~

Move [support.initlist.concept] from its current location at 18.9.3 onto the end of [iterator.concepts.range], probably at 24.2.8.1.

Resulting impact on freestanding specification

[compliance] must be amended to include <initializer_list>.

Impact on C++0x

There's no impact at all to the proposed list-based initialization functionality which is a major syntactic enhancement to C++ -- and, yet, for many users, also a major simplification.

Related issues

CD comment UK-195 (still open after Summit) duplicates this proposal.

4.3. Type traits

Background

The facilities defined by header <type_traits>, specified in [meta], are "used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time."

Unlike many other headers in C++, <type_traits> features depend, as is rather strongly implied by the above, on private compiler interfaces which are not explicitly specified in the C++ language.

Many standard headers and application programs depend on <type_traits>.

Current impact on freestanding definition

<type_traits> is not currently in table 15 in [compliance].

It's possible to deliver a compiler without a dependency on <type_traits>, so this motivation for inclusion in the freestanding definition does not apply.

However, it's not possible to write (or build) a full library implementation, or anything else that depends on <type_traits>, without access either to undocumented compiler interfaces, or to a <type_traits> delivered with the compiler.

Therefore, based on a (hitherto unstated) requirement that the freestanding specification should form a completely documented specification on which to build a full implementation of this standard without further reliance on undocumented compiler interfaces, <type_traits> should be included in the freestanding definition.

Proposal

Include <type_traits> in the freestanding definition.

Move the subclause, [meta], from its current home in Clause 20 (general utilities library) to a new home in Clause 18 (language support library).

Wording

To the list of required headers in [compliance], table 15, add

18.nn	Metaprogramming and type traits	<type_traits>

Move [meta] from its current location at 20.6 somewhere into Clause 18.

Resulting impact on freestanding specification

Type traits, as defined in <type_traits>, must now be considered part of the freestanding specification.

Impact on C++0x

No impact.

Related issues

None