Fixing freestanding

document id: N2814=09-0004

date: 2009-01-22

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

contributor: Jan van Bergen (jan.vanbergen@symbian.com)

context: baselined on C++0x CD, N2800 (WG21) / N4411 (SC22), October 2008

Acknowledgements: thanks to reviewers for comments and shepherding advice – BSI C++ panel, Alisdair Meredith, Roger Orr, Clark Nelson.

1. Introduction

This paper proposes several small, targetted, fixes to the C++0x CD which:

- enable the intent of the freestanding specification (17.6.2.4) to be preserved, by adding only a single header to it relative to its C++98/C++03 contents (containing the declaration of `std::initializer_list`)
- 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

As a side benefit, which was not originally a motivator, this proposal makes range-based `for` more usable and teachable for new C++ users.

1.1. Why?

The need for this paper arises because the C++0x CD 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 (5.3.4), which as proposed bring in `std::string` as a language dependency
- usability of `typeinfo` in containers (18.6), which as proposed brings STL containers into `<typeinfo>` and thus into freestanding use of type information
- implementation of lambda (5.1.1), which as proposed requires `<functional>` unnecessarily
- range-based `for` (6.5.4), which as proposed requires `<iterator_concepts>` even when iterating over C arrays and initializer lists
- initializer list (18.8), which as proposed requires the `Range` concept even though an initializer list is just an array with a length

The C++0x CD is inconsistent in that, while these new features are proposed with the impacts stated above, the headers required by the freestanding definition in 17.6.2.4 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 useful and competitive features, in particular lambda, range-based `for` and initializer-lists. Such an approach is really a non-starter.

1.3. What?

In summary the changes proposed are:

- array new (5.3.4 para 7): change the proposed `std::length_error` (an exception type requiring a `std::string`) to `std::bad_alloc` (an exception type requiring only a `char*`)
- `<typeinfo>` (18.6): 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>`
- lambda (5.1.1): in para 2, clarify that the requirement to "behave as a function object" as defined in `<functional>` doesn't actually mean a lambda must be *implemented* as a function object, so that lambdas can be used without a need for `<functional>`; in para 12, tweak the text referring to `std::reference_closure<R(P)>` to require an ABI dependency but not a header file dependency
- range-based `for` (6.5.4): make it explicit that, where the range is a C array whose length is known at compile time, and therefore the iterator is a C pointer, there is no need for the user to include `<iterator_concepts>`, and no need for the compiler to depend on the library
- initializer lists (18.8): 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>`

1.4. So what?

These changes remove no features of C++0x, and yet they:

- enable the current small set of headers required for the freestanding library to be preserved, with the addition only of a lightweight `<initializer_list>`
- enable lambdas, range-based `for` and initializer lists to be used in freestanding implementations
- enable lambdas and range-based for to be used without special header files (unless such special header files are needed for other reasons)
- reduce the dependency of language on library, in that lambda and range-based `for` can be used without library support

Even with these changes, it remains that the C++0x language places an increased dependency on the library compared with C++03. Lambdas, range-based `for` and list initialization all have special relationships with the library, which have to be supported in compiler front-ends, even if they are not exploited/supported in any library delivered along with the compiler.

Range-based `for` now has two special cases: one for C arrays, one for `initializer_lists`. These special cases are due to the special place of C arrays and `initializer_lists` in C++: they are effectively built-in types, unlike the generalized containers over which the more general range-based `for` iterates.

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.

3. Feature by feature

The following outlines the impact of the N4411 changes, feature by feature, on the freestanding definition in 17.6.2.4, which in the C++0x CD is still the same as it was in C++98 (sic).

For each feature, there is

- background on the new feature as defined in the CD
- an impact assessment on the freestanding definition
- a proposal to change to the current specification of the feature
- a statement of the resulting impact of the changed feature on the freestanding definition
- a statement of the impact of the change on the intent of the C++0x feature

The intent is to make changes which preserve the intent and usefulness of the C++0x feature, while also preserving the intent and usefulness of the freestanding definition – or, looked at another way, continuing to provide for minimal dependency of language on library.

3.1. Array new

Background

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

However, a *new-expression* (5.3.4) may throw a `std::length_error` (para 7), if the length is too long.

Strangely, a *new-expression* is *not* required to throw `std::length_error` if the given length is negative! – the behaviour then is simply undefined.

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.

Proposal

Replace the `std::length_error` required in 5.3.4 para 7 with `std::bad_alloc`.

Resulting Impact on freestanding implementation

The net effect of this proposal is no 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.

3.2. `<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 18.6, 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.

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>`

Resulting impact on freestanding implementation

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.

3.3. Lambda

Background

5.1.1 lambda expressions, para 2, states that a lambda evaluates to a closure object, which "behaves as a function object", which is defined in 20.6.

20.6 defines `<functional>`, which (a) makes a simple statement that function objects have an `operator()()` which returns a result, and (b) include a lot of C++ template-speak to define such objects (most of which dates from C++98, though there are additions from 0x eg in the domain of variadic templates and conceptization).

Furthermore, 5.1.1 para 12 requires non-mutable lambdas whose capture-list consists exclusively of references, to be implemented in terms of – in fact, to be publicly derived from – a `std::reference_closure<R(P)>`, defined in 20.6.18, where `R` is the return type of the lambda expression, and `P` the parameter type list.

Current impact on freestanding definition

Both 5.1.1 para 2 and para 12 appear to require `<functional>` to use lambda. This header, if required in freestanding implementations, would introduce significant bloat.

Proposal

5.1.1 para 2 doesn't actually require `<functional>`: it's just referring to it for clarity of exposition. Therefore change the wording of 5.1.1 para 2 to state explicitly that "behaves as" doesn't require "must be implemented as", and in fact that lambdas can be used without placing any dependency on `<functional>`.

The requirement in 5.1.1 para 12 to map onto reference closures can be seen as a straightforward ABI requirement on both lambda and `reference_closure`. Therefore take this approach, and change the "publicly derived from" wording to "indistinguishable from".

Resulting impact on freestanding implementation

Lambda syntax can be used without needing `<functional>`, and without needing a radical expansion of the library.

Impact on C++0x

Lambda syntax can be used in a translation unit without needing to include `<functional>`.

3.4. Range-based for

Background

In 6.5.4, the syntax

`for (` *for-range-declaration* `:` *expression* `)` *statement* `;`

is defined as being equivalent to

```
{
auto && __range = ( expression ); // a range, of type RangeT
for ( auto __begin = std::Range<RangeT>::begin(__range),
                __end = std::Range<RangeT>::end(__range);
            __begin != __end;
            ++__begin) {
        for-range-declaration = *__begin;
        statement;
        }
}
```

where `__range`, `__begin` and `__end` are defined for exposition only.

The specification requires any program using range-based `for` to include `<iterator_concepts>` (the header in which the `Range` concept is defined, along with a `concept_map` of `Range` to arrays), or be ill-formed.

Current impact on freestanding definition

As currently specified, range-based `for` can't be used unless `<iterator_concepts>` is also available, which makes `<iterator_concepts>` a necessary part of the freestanding definition.

Furthermore, even the trivial example given in 6.5.4, namely

```
int array[5]={1, 2, 3, 4, 5};
for (int& x : array)
      x *= 2;
```

apparently requires `<iterator_concepts>`, even though the only thing being iterated through is a classic C array. This will appear silly to ordinary users and learners of the language.

Proposal

Amend the specification to permit range-based `for` on arrays whose length is known at compile time, to be implemented in terms of classic C pointers rather than iterators, and without `<iterator_concepts>`.

Resulting impact on freestanding implementation

The required headers for a frestanding implementation are not affected by range-based `for` itself – only by uses of range-based for which actually require the C++ `Range` concept.

Impact on C++0x

This does not compromise the expressive power of range-based `for`.

In fact it makes range-based `for` easier to use, in that when no C++ `Range` is required, no special header file needs to be included in order to use the language feature.

3.5. 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 listed in 17.6.2.4.

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 17.6.2.4.

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 18.8 (initializer lists) to 24.1.8 (ranges).

Add another special case to range-based `for` syntax, to make it possible to use this syntax on `std::initializer_lists` without language bloat.

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).

Resulting impact on freestanding implementation

Clause 17.6.2.4 must be amended to include `<initializer_list>`.

Impact on C++0x

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

This proposal recognizes that `std::initializer_list` is all but a fundamental type, in the same sense that C arrays are. It's for that reason that – along with C arrays – initializer lists have a special case in "range-based" for syntax.

4. Proposed changes to the text

In this section changes are proposed to the text of the C++0x CD which would implement the proposals made above.

The changes are grouped into

- language changes, affecting clauses 5 and 6, addressing array new, lambda, and range-based for
- library changes, affecting clauses 17, 18, 20 and 24, addressing `<typeinfo>`, lambda and list initialization

Changes are grouped into language changes and library changes, but are not otherwise presented in the sequence with which they impact the standard (which might make them fractionally easier to apply), but rather in a natural order of exposition (which should make them much easier to review).

4.1. Language changes

Array new

Amend 5.3.4 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_alloc` (~~19.1.4~~18.5.2.1).

Lambda

Amend 5.1.1 para 2 as follows:

> The evaluation of a lambda-expression results in a closure object, which is an rvalue. Invoking the closure object executes the statements specified in the lambda-expression's compound-statement. Each lambda expression has a unique type. Except as specified below, the type of the closure object is unspecified. [Note: A closure object behaves as if it were a function object (20.6) whose function call operator, constructors, and data members are defined by the lambda-expression and its context.

However, lambdas may be used without inclusion of `<functional>` or any other library definition. —end note]

Amend 5.1.1 para 12 as follows:

> If every name in the effective capture set is preceded by & and the lambda expression is not mutable, `F` shall be implemented in a manner indistinguishable ~~is publicly derived~~ from `std::reference_closure<R(P)>` (20.6.18), where `R` is the return type and `P` is the *parameter-type-list* of the lambda expression. Converting an object of type `F` to type `std::reference_closure<R(P)>` and invoking its function call operator shall have the same effect as invoking the function call operator of `F`. [*Note:* This requirement effectively means that such `F`'s must be implemented using a pair of a function pointer and a static scope pointer. —*end note*][*Note:* This requirement avoids a dependency of the compiler's support for lambda on the library, but it does place an ABI constraint on the library implementation of `std::reference_closure<R(P)>`. —*end note*]

Range-based for

Amend 6.5.4, the range-based for statement, as follows:

The range-base for statement

```
for ( for-range-declaration : expression ) statement
```

is equivalent, if *expression* is an array of N elements, N known at compile time, to

```
{
      for ( auto __begin = ( expression ),
                 __end = __begin + N;
                 __begin != __end;
                 ++__begin ) {
            for-range-declaration = *__begin;
            statement
      }
}
```

That is, iteration ranges over the array.

If *expression* is of type `std::initializer_list`, then the statement is equivalent to:

```
{
      for ( auto __begin = ( expression ).begin(),
                 __end = ( expression ).end();
                 __begin != __end;
                 ++__begin ) {
            for-range-declaration = *__begin;
            statement
      }
}
```

That is, iteration ranges over the initializer list.

If expression is a range of type `_RangeT`, then the statement is equivalent to:

```
{
      auto && __range = ( expression );
      for ( auto __begin = std::Range<_RangeT>::begin(__range),
                 __end = std::Range<_RangeT>::end(__range);
                 __begin != __end;
                 ++__begin ) {
            for-range-declaration = *__begin;
            statement
      }
}
```

In the above, ~~where~~ `__range`, `__begin`, and `__end` are variables defined for exposition only~~, and `_RangeT` is the type of the expression~~.

[Example:

```
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
      x *= 2;
```

```
        int total=0;
        for (int x : { 1, 2, 3, 4, 5 })
                total += x;

        vector<int> v = { 1, 2, 3, 4, 5 };
        for (int& i : v)
                i *= 2;
```

—end example]

If the header `<iterator_concepts>` (24.1) is not included prior to a use of the range-based for statement, and the range-based for is not iterating over an array of known length or an initializer list, the program is ill-formed.

{ *Note:* The special-case treatment of arrays and initializer lists reflects the fact that they are fundamental in C++. – *end note*]

4.2. Library changes

<typeinfo>

In 18.6, 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;
        }
```

Delete entire section 18.6.2 class `type_index`, including its subordinate headings and their content. This text should instead be moved to a new section 20.10.

Create a new section 20.10 Type Indexes. Begin the 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;
                }
        }
```

then include all headings and content previously in 18.6.2.

In 17.6.2.3 headers, table 13, add `<typeindex>` to the list.

In 20 general utilities library, table 20, add

20.10 type indexes `<typeindex>`

to the end of the table.

Lambda

In 20.6.18 class template reference_closure, amend the first body paragraph as follows:

The `reference_closure` class template represents reference-only closures (5.1.1). The implementation of `reference_closure` in any given library is constrained by the compiler's ABI requirements on reference closures.

List initialization

To the list of required headers in 17.6.2.4, table 15, add

18.8 Initializer Lists `<initializer_list>`

In 18.8, initializer lists, 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;~~
```

Delete section 18.8.3, initializer_list concept maps, and insert its text instead into 24.1.8.1, ie a subsection of 24.1.8, Ranges.

Insert new section 24.1.8.1 initializer_list concept maps, with text taken from current 18.8.3.