Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P0830R0**
Date: 2017-10-15
Reply to: Nicolai Josuttis (nico@josuttis.de)
Audience: LEWG, LWG?
Prev. Version:

Using Concepts and `requires` in the C++ Standard Library

Because concepts were now adopted (again) as a C++ language feature, this raises the question of how to deal with it in the C++ Standard Library. This paper has the intent to come up with corresponding guidelines.

1. Concept Name Conventions

As done in the Ranges TS, standard concept names should be defined with CamelCase.

For example:

```
template <class T> concept bool SignedIntegral() {
  …;
}
```

2. ABI Compatibility

Although the C++ standard does not guarantee ABI compatibility, it is always a goal to be able to have it. The reason is that this allows to link together code compiled with different compiler versions. With ABI breaks, customers/programmers might otherwise be required to recompile the whole code base.

In principle, `requires` clauses have to be part of the signature to be able to overload functions having different `requires` clauses only.
However, in practice this is not a problem with the following constraints:

- The `requires` clauses applies to template code only (this implied non-template member functions of class templates).
- The concepts and directly formulated `requires` clauses are local and stateless.
  - There are no side effects changes in behavior, for example, due to static variables.
- There is no need/support for full specializations.

To be able to change `requires` clauses, `requires` clauses should confirm to the guidelines above.

Let me explain the reason for these constraints.

Reasoning for Constraints for ABI Compatibility

First let's look at the current situation regarding possible changes on requirement with `enable_if` clauses by example.

Assume in one translation unit (TU) we have a function template foo() with an `enable_if` clause:

tu1.cpp:

```
emplate<typename T>
std::enable_if_t<(sizeof(T) >= 4), int>
foo(T) {
  std::cout << "foo() for 'std::enable_if_t<(sizeof(T) >= 4)>'\n";
  return 42;
}
void tu2();
```

```
int main()
{
    foo(42ll);  // __Z3fooIxENSt9enable_ifIXgestT_Li4EEiE4typeES1_
    tu2();
}
```

The symbol for foo() in tu1.o will contain the condition as part of the return type.
For example, if you compile with ``g++80 -O0 -c tu1.cpp'' and check with nm:

```
__Z3fooIxENSt9enable_ifIXgestT_Li4EEiE4typeES1_
```

The generic return type “XgestT_Li4EE” (which includes the condition) is part of the signature, and therefore part of the ABI.

Now assume we later compile foo() in a different module with a slightly modified clause (now comparing the size of T with 8 instead of 4):

tu2.cpp:
```
========
  template<typename T>
  std::enable_if_t<(sizeof(T) >= 8), int>
  foo(T) {
    std::cout << "foo() for 'std::enable_if_t<(sizeof(T) >= 8)>'\n";
    return 42;
  }

  void tu2()
  {
      foo(42ll);  // __Z3fooIxENSt9enable_ifIXgestT_Li8EEiE4typeES1_
  }
```

Now, we we see the modified symbol for foo() in tu2.o:

```
__Z3fooIxENSt9enable_ifIXgestT_Li8EEiE4typeES1_
```

Thus, the change breaks the ABI, because the condition being part the symbol changed from “XgestT_Li4EE” to “XgestT_Li8EE”.

Nevertheless, we can link the executable using both object files.
As a result the executable has both symbols:

```
__Z3fooIxENSt9enable_ifIXgestT_Li4EEiE4typeES1_
__Z3fooIxENSt9enable_ifIXgestT_Li8EEiE4typeES1_
```

Thus, usually **having different requirements with `enable_if` usually is no problem at all**, because we use templates, where the call result in inline instantiations of the templates. Any call of foo() is located in the same object file as the instantiation and the resulting executable just has both versions of foo() and works just fine. The program above outputs:

```
foo() for 'std::enable_if_t<(sizeof(T) >= 4)>'
foo() for 'std::enable_if_t<(sizeof(T) >= 8)>'
```

However, as Daveed Vandevourde comments: “*All kinds of things could go wrong, of course, but for typical situations they don’t happen.*”

For example, if we have a static member in foo(), we suddenly have two different static members, not (necessarily) shared. If inside foo() we'd have

```
static int numCalls = 0;
```

we'd get two different symbols:

```
00402040 D __ZZ3fooIxENSt9enable_ifIXgestT_Li4EEiE4typeES1_E8numCalls
00402044 D __ZZ3fooIxENSt9enable_ifIXgestT_Li8EEiE4typeES1_E8numCalls
```

If we replace the `enable_if` clause by `requires` clauses, for template code we have pretty much the same behavior:
If

```
template<typename T> requires (sizeof(T) >= 4) int foo(T);
```

becomes

```
template<typename T> requires (sizeof(T) >= 8) int foo(T);
```

and even if we replace

```
template<typename T> std::enable_if_t<(sizeof(T) >= 4), int> foo(T);
```

by

```
template<typename T> requires (sizeof(T) >= 8) int foo(T)
```

this usually works in practice, because both the call and the instantiation are in the same object file provided we don't have side effects and/or stateful behavior.

But as Jonathan Wakely comment:

> *In addition to the local static issue there's an issue with non-template member functions of class templates in explicit instantiations. Those can't have `enable_if` constraints though, so they are affected as described below for ordinary functions.*

So, we also have the ability for full specializations into account.

Note1: When trying these examples out with compiler supporting concepts out that gcc you might not see yet that the `requires` clause is part of the name mangling at all, which is a bug and has to get fixed (in coordination with all who care for a standard mangling).

Requires with Non-Template Code

Originally, require clauses should also be able to be used with ordinary functions:

```
int foo() requires (sizeof(T) >= 4);
```

In this case, the call and the definition might easily occur in different translation units. And when we change the requirement to

```
int foo() requires (sizeof(T) >= 8);
```

and only compile the call without compiling the definition again, the linker will search for a function using a different name mangling and will fail.

However, in Toronto meeting 2017, CWG seems to have decided that
CWG decided in Toronto that:

- Consensus: requires-clauses on non-templated functions are ill-formed

So, this doesn't seem to be an issue, yet.

3. Semantic Guidelines

As a third group of guideline let me come up with some proposals regarding the question of when and how to define concepts.

To some extent, this is similar to dealing with patterns. For that reason, let me introduce two new terms here:

- Concept System
  - A set of concepts, which play together to cover a couple of requirements by a consistent way and style.
- Concept Language
  - A concept system, where the concepts depend on and compose each other to form a consistent set of concepts being able to help in various contexts.

The goal for a C++ standard library should be to establish at least a concept system; ideally a concept language.

As a general guideline, Eric Niebler and Casey Carter pointed to an important aspect:

Concepts should care about semantics, not syntax. Or in other words: concepts should reason about what your code means and not whether whether it compiles

For this reason

- Not each trait is a useful concept. For example, a trait is_copyable checks whether we can call the copy constructor, while a concept IsCopyable means that after a copy (there might be different ways to perform it) two objects have the same value.
- For the same reason, not every STL requirement is necessarily a concept.

That means that **we still need traits and we still need `requires` clauses** to fix unintended behavior ("shall not participate in overload resolution unless…") **without introducing concepts for them.**

In fact:

- Traits yielding types are never concepts.
- Traits and `enable_if` clauses without semantics should not be concepts.
- Concepts are requirements for humans
  - Concepts can't be generated.
  - There are no high order concepts

For this reason, a better categorization for requirements, as Walter Brown discusses in P0788R0.

4. Summary

Here some recommendations I see from the discussion we had so far.

- Concept names use CamelCase.
- Concepts and `requires` clauses should always be predicates (be stateless and return bool).
- Don't convert all traits or `enable_if` clauses into concepts.
- Use concepts and `requires` clauses only for template code.
- Concepts evolve
  - May be: "Three strikes and you conceptify" (if a concept proved to be useful three times in three contexts, we should standardize it).

The good now is that with these rules ABI compatibility is not an issue. Thus, we can fix mistakes.

5. Acknowledgements

Thanks to Daveed Vandevoorde, Ville Voutilainen, Alisdair Meredith, Walter Brown, Eric Niebler, Bob Steagull, Casey Carter, Beman Dawes, and several others who helped me to come up with this analysis and recommendation. Any error is probably due to my lack of understanding.