Document Number: P0027R0
Reply-to: Troy Korjuslommi, troykor@gmail.com

Named Types

The proposal is for a new mechanism to create named types with behavior similar to built-ins and classes.
The primary problem involves template instantiations, but should be extended to include non-template based types as well. Type creation in Ada, including the creation of generic types, is based on a similar mechanism.

E.g. instead of a typedef such as
typedef std::vector <std::string> stringvec_t;
we would have
newtype std::vector <std::string> stringvec_t;
OR the more Ada'esque
newtype stringvec_t is std::vector <std::string>;

One would then use stringvec_t just like a vector, except that a vector<string> could no longer be used where a stringvec_t is required. The compiler/debugger would also identify the type as stringvec_t, and hide the type's internals from the user (including in error messages and in the debugger).

This proposal suggests the keyword "newtype." The keyword is subject to debate, so please replace it with a word of your own liking for now.

Potential benefits would be:
- More readable error messages when templates are involved.
- A convenient means to specify the object file where a specific template instantiation should be stored.
- A means to communicate that a specific type/template instantiation should be used, disallowing any further type substitutions, template expansions and type deductions. This would prevent a problem with template expansions where the programmer accidentally uses the wrong type arguments in two or more locations, and the compiler converts the types or expands another template type.
- A means to define a limited scope for types (when a name is declared within a lexical scope, it cannot be used outside that scope).
- A means to define a new type whose behavior is identical to another type, but whose identities are distinct, and can be used to enforce strong type checking. Such use is handy when a type doesn't provide new facilities, but is only valid in certain contexts, and only that type is valid in such contexts. As an example, if we have "newtype width_t is uint32_t" and a corresponding height_t type, and use these in various functions dealing with widths and heights, any accidental swapping of the arguments would produce a compile time error.

Problems to solve:
- Verbose error messages (see a minimal example below).
- The "extern template" mechanism only allows one to specify that a template is defined elsewhere, it doesn't specify that a specific template instantiation

is the one which should be used throughout (or where to store the object code).
- The typedef system falls short of its potential, as the types "named" by it are not seen by the compiler (at least not in error messages), but expanded more like macros. struct/class can be used to create new types, but they work as a container, and are not able to assume the facilities of another type.
- Types created using typedef do not create a new type which can be used to enforce strong typing constraints. E.g. a function which accepts an argument type myint (typedef int myint), can be called with an int. A truly new type would mean that passing an int argument would fail.
- The lack of an ability to define new types (whether with typedef or templates) also means an inability to limit the scope of types. This ability is sometimes very useful in ensuring that a type is only used in a valid context. Ada makes extensive use of such type scoping, to great benefit. A scope limited type argument in a function means that it cannot be called from an outer scope where that argument type was not declared. typedefs and templates do not provide this type of checking, as their scope cannot be similarly restricted. One can achieve similar results using a locally declared class/struct, but then one has to provide a full implementation of it as well.

Furthermore, I see the ability to create new native looking types as a means to expand the language's ability to express more complex ideas.

A little more ambitious proposal would be to add the ability to define ranges for numeric values (long, int, short, double, float) when declaring a new type. This is straight from Ada's playbook. E.g. "newtype age is uint8_t(0..130);"
Range limiting functionality can currently be provided with templates, so strictly speaking, no new language facilities are needed, except in terms of user convenience.

There is an "opaque typedefs" proposal in the C++ working group, which seems similar. It doesn't quite seem to follow the same line of thought, though.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3957.html
52. N3741, N3515 Toward Opaque Typedefs for C++1Y

The example below demonstrates error message differences between a simple type (a struct) and a template based type.

```
#include <iostream>
#include <string>
#include <vector>

#ifdef TEMPLATE_TEST

typedef std::vector <std::string> stringvec_t;

// GENERATED ERROR (clang++):
```

```
// named_template.cpp:21:4: error: no member named 'call' in
'std::__1::vector<std::__1::basic_string<char>,
//       std::__1::allocator<std::__1::basic_string<char> > >'
//         t.call();

#else

struct stringvec_t {
};

// GENERATED ERROR (clang++):
// named_template.cpp:21:4: error: no member named 'call' in 'stringvec_t'
//         t.call();

#endif

void Print(const stringvec_t& t) {
          t.call();
}

int main() {
          stringvec_t names;
          Print(names);
          return 0;
}
```