How do I resolve this <unresolved overloaded function type> error when using std::function?

In the following (working) code example, the templated register_enum() function is used to iterate over an enumeration and call a user provided callback to convert an enum value to a c-string. All enumerations are defined within a class, and enum to string conversion is done with a static to_cstring(enum) function. When a class (like the shader class below) has more than one enumeration and corresponding overloaded to_cstring(enum) function, the compiler can't decide which is the correct to_cstring() function to pass to register_enum(). I think the code explains better than I can...

```
#include <functional>
#include <iostream>

// Actual code uses Lua, but for simplification
// I'll hide it in this example.
typedef void lua_State;

class widget
{
    public:
        enum class TYPE
        {
            BEGIN = 0,
            WINDOW = BEGIN,
            BUTTON,
            SCROLL,
            PROGRESS,
            END
        };

        static const char *to_cstring( const TYPE value )
        {
            switch ( value )
            {
                case TYPE::WINDOW: return "window";
                case TYPE::BUTTON: return "button";
                case TYPE::SCROLL: return "scroll";
                case TYPE::PROGRESS: return "progress";
                default: break;
            }
            return nullptr;
        }
};

class shader
{
    public:
        enum class FUNC
        {
            BEGIN = 0,
            TRANSLATE = BEGIN,
            ROTATE,
            SCALE,
            COLOR,
            COORD,
            END
        };

        enum class WAVEFORM
        {
            BEGIN = 0,
            SINE = BEGIN,
            SQUARE,
            TRIANGLE,
            LINEAR,
            NOISE,
            END
        };

        static const char *to_cstring( const FUNC value )
        {
            switch ( value )
            {
                case FUNC::TRANSLATE: return "translate";
                case FUNC::ROTATE: return "rotate";
                case FUNC::SCALE: return "scale";
                case FUNC::COLOR: return "color";
                case FUNC::COORD: return "coord";
                default: break;
            }
            return nullptr;
        }

        static const char *to_cstring( const WAVEFORM value )
        {
            switch ( value )
            {
                case WAVEFORM::SINE: return "sine";
                case WAVEFORM::NOISE: return "noise";
                default: break;
            }
            return nullptr;
        }
};

// Increment an enum value.
// My compiler (g++ 4.6) doesn't support type_traits for enumerations, so
// here I just static_cast the enum value to int... Something to be fixed
// later...
template < class E >
E &enum_increment( E &value )
{
    return value = ( value == E::END ) ? E::BEGIN : E( static_cast<int>( value ) +
1 );
}

widget::TYPE &operator++( widget::TYPE &e )
{
    return enum_increment< widget::TYPE >( e );
}

shader::FUNC &operator++( shader::FUNC &e )
{
    return enum_increment< shader::FUNC >( e );
}

shader::WAVEFORM &operator++( shader::WAVEFORM &e )
{
    return enum_increment< shader::WAVEFORM >( e );
}


// Register the enumeration with Lua
template< class E >
void register_enum( lua_State *L, const char *table_name, std::function< const
char*( E ) > to_cstring )
{
    (void)L; // Not used in this example.
    // Actual code creates a table in Lua and sets table[ to_cstring( i ) ] = i
    for ( auto i = E::BEGIN; i < E::END; ++i )
    {
        // For now, assume to_cstring can't return nullptr...
        const char *key = to_cstring( i );
        const int value = static_cast<int>(i);
        std::cout << table_name << "." << key << " = " << value << std::endl;
    }
}

int main( int argc, char **argv )
{
    (void)argc; (void)argv;

    lua_State *L = nullptr;

    // Only one to_cstring function in widget class so this works...
    register_enum< widget::TYPE >( L, "widgets", widget::to_cstring );

    // ... but these don't know which to_cstring to use.
    register_enum< shader::FUNC >( L, "functions", shader::to_cstring );
    //register_enum< shader::WAVEFORM >( L, "waveforms", shader::to_cstring );

    return 0;
}
```

Compiler output:

```
$ g++ -std=c++0x -Wall -Wextra -pedantic test.cpp -o test && ./test
test.cpp: In function ‘int main(int, char**)’:
test.cpp:140:69: error: no matching function for call to
‘register_enum(lua_State*&, const char [10], <unresolved overloaded function
type>)’
test.cpp:140:69: note: candidate is:
test.cpp:117:7: note: template<class E> void register_enum(lua_State*, const char*,
std::function<const char*(E)>)
```

How do I pass the correct to_cstring function to register_enum()? I realise I could rename the individual to_cstring() functions but I'd like to avoid this if possible. Perhaps my design is smelly and you can recommend a better approach.

My question appears similar to Calling overloaded function using templates (unresolved overloaded function type compiler error) and How to get the address of an overloaded member function? but so far I am unable to apply that information to my specific issue.

c++ c++11

edited May 23 '17 at 12:31
Community♦
1 1

asked Apr 6 '12 at 1:48
x-x
3,412 7 34 66

2 Answers

The error tells you that there are two potential overloads that could be used, and the compiler cannot decide for you. On the other hand, you can determine which one to use by using a cast:

Or without the typedef (in a harder to read one-liner):

```
register_enum< shader::FUNC >( L, "functions",
                (const char *(*)( shader::FUNC ))shader::to_cstring );
```

*Note that in function signatures, the top-level `const` gets removed.

The next question is why did the compiler not find the appropriate overload by itself? The problem there is that in the call to `register_enum` you pass the type of the enum, and that determines the type of the `std::function` to be `std::function< const char* ( shader::FUNC ) >`, but `std::function` has a templated constructor, and before trying to infer the type of the argument to the constructor, the compiler must know which overload you want to use.

edited Apr 7 '12 at 0:18

answered Apr 6 '12 at 2:07
David Rodríguez - dribeas
165k 14 216 417

I would say that in this case, if the compiler cannot decide between the two overloads, you should rename one of them! This will prevent further ambiguous usage of the functions.

answered Jan 18 at 22:40
xcski
314 3 9

Why the downvote? this is a much simpler solution. – xcski Jan 18 at 23:19