C++ Proposal:
Extensions of Scoped Enumerations

Philip Ginsbach
philip.ginsbach@ed.ac.uk

December 2, 2018

Introduction

Scoped enumerations ("enum classes") provide a type safe alternative to C-style enums. It can be desirable to have enumerations that share a subset of their enumerators and there is a natural correspondence between subtyping and adding enumerators to enums. This proposal is for a simple, sound and practically useful extension to the core language standard to provide a mechanism for subtyping scoped enumerations.

Motivation

Consider the task of parsing simple arithmetic expressions of the following form.

```
term ::= <term><op><term>|<integer>
```

Implementing a lexer for such a language would typically involve an enum definition.

```
enum class TokenType { constant, binary_op };

struct Token {
    TokenType type;
    // ...
};
```

It is then quite natural to describe a simple syntax tree structure as below. The leaves of the syntax tree correspond to individual tokens and `TokenType` is naturally a subtype of `SyntaxType`. There is currently no way to express this relationship in C++.

```
enum class SyntaxType { constant, binary_op, binary_exp };

struct Syntax {
    SyntaxType     type;
    vector<Syntax> children;
    // ...
};
```

This causes real problems: In a parser, each token will be converted to a syntax tree leaf. The only safe way (without relying on the underlying integer values) to do this is with a switch statement or equivalent if-else branches.

```
switch(token.type) {
    case TokenType::constant:  stack.emplace_back(SyntaxType::constant);
    case TokenType::binary_op: stack.emplace_back(SyntaxType::binary_op);
}
```

Notice how this problem is only introduced by the use of scoped enumerations and is not present when using, for example, `typedef`s.

Proposed Solution

I propose the use of already existing syntax for the extension of scoped enumerations.

```
enum struct|class name : name { enumerator, ... };
```

The standard currently allows this syntax to specify the underlying integer type, instead it should also be allowed to use an enum class (the "base enumeration") to the right of the colon. This would have the following effect:

1. The underlying integer type is adopted from the base enumeration.
2. All enumerators from the base enumeration are available in the new enumeration, the underlying integer values are guaranteed to be identical.
3. Values of the base enumeration can be safely and implicitly cast to values of the extended enumeration type.

Note that the last property is essentially inverse to the effects of class inheritance.

Extending a scoped enumeration therefore results in a new scoped enumeration that is constructed as if all the enumerator definitions of the base enumeration are textually inserted before the first specified enumerator. Implicitly assigned enumerator values start with the increment of the assigned value of the last enumerator of the base enumeration. It furthermore means that enumerator values can not be renamed and enumerator names can not be assigned a new value.

Implementation

The feature is simple to implement, as it uses no new syntax and only simple semantics. An implementation of the proposed feature in clang is available on github at `https://github.com/ginsbach/CppProposal`. It requires the `release_70` branch of llvm to build, you can obtain the source code with the commands below. For build instructions please consult the llvm project documentation `https://llvm.org/docs/CMake.html`.

```
git clone https://github.com/ginsbach/CppProposal clang
git clone https://github.com/llvm-mirror/llvm --branch release_70
git -C llvm checkout 65ce2e56889af84e8be8e311f484a4dfe4b62d7a
ln -s clang llvm/tools/clang
```

Discussion

Implicit Base Types

The base types of scoped enumerations in C++ can be left unspecified, allowing the compiler to choose an integral type. This clashes with extensions, as some enumerators in the extended enum might not be representable in the chosen integral type.

I propose to only allow scoped enumerations with explicit integral type specifications to be extensible.

Multiple Base Enumerations

It is often useful to group enumerators together, in effect partitioning the enum into several smaller enums. This could be represented with something equivalent to multiple inheritance for classes, however, I do not think this should be done.

Multiple base enumerations could only work if they use the same integral type and it would fail if any of their enumerators have the same value or name. This means that the base enumerations would have to be carefully constructed with each other in mind, effectively introducing implicit dependencies between things that should be independent from a conceptual standpoint.

I propose that multiple base enumerations should not be allowed.

Empty Extension

There is no fundamental reason to disallow an extension to a scoped enumeration that doesn't actually add any new enumerators. It would simply create two scoped enumerations that are identical but can only be implicitly cast in one direction.

I propose to allow this.

std::underlying_type

The underlying type of an extension of a scoped enumeration is the underlying type of the base enumeration.

Example

Consider the example below based on the motivation section. It can be readily compiled with the previously mentioned, extended fork of clang.

```
// Tokens can either be constants or binary operators. The numeric values are
// intentionally arbitrary.
enum class TokenType : int { constant=211, binary_op=223 };

struct Token {
    TokenType type;
    string    value;
};

// A node in the parse tree can either be a leaf (i.e. a token) or a binary
// expression. We can elegantly express this with Extended Scoped Enumerations.
// Without this feature, we'd have to duplicate the entries in TokenType and
// we'd have no guaratees when casting from TokenType to SyntaxType, which can
// be done implicitly and safely with Extended Scoped Enumerations.
#ifdef ALLOW_SCOPED_EXTENSIONS
enum class SyntaxType : TokenType { binary_exp /* = 224 */ };
#else
enum class SyntaxType : int { constant=211, binary_op=223,
                              binary_exp /* = 224 */ };
#endif

struct Syntax {
    SyntaxType     type;
    vector<Syntax> children;
    string         value;
};

Syntax parse(vector<Token> tokens)
{
    vector<Syntax> stack;

    for(Token token : tokens)
    {
        // We can construct an instance of Syntax with token.type, although it
        // is actually of type TokenType, because it is implicitly cast.
        // Without Extended Scoped Enumerations, we'd either need a big switch
        // statement or we'd have to use an unsafe cast, relying on the
        // underlying numeric values for elements in TokenType to be the same as
        // in SyntaxType. That would make obsolete the use of enum class,
        // as the advantage over C-style enums is supposedly type safety.
#ifdef ALLOW_SCOPED_EXTENSIONS
        stack.push_back({token.type, {}, token.value});
#else
        if(token.type == TokenType::constant)
            stack.push_back({SyntaxType::constant, {}, token.value});
        if(token.type == TokenType::binary_op)
            stack.push_back({SyntaxType::binary_op, {}, token.value});
#endif
```

```
        while(stack.size() >=3 &&
              stack[stack.size()-1].type != SyntaxType::binary_op &&
              stack[stack.size()-2].type == SyntaxType::binary_op &&
              stack[stack.size()-3].type != SyntaxType::binary_op)
        {
            stack[stack.size()-3] = Syntax{SyntaxType::binary_exp,
                                           {stack[stack.size()-3],
                                            stack[stack.size()-2],
                                            stack[stack.size()-1]}};
            stack.resize(stack.size() - 2);
        }
    }

    if(stack.size() == 1)
        return stack[0];
    throw;
}

double evaluate(Syntax syntax)
{
    switch(syntax.type) {
    case SyntaxType::constant:
        return atof(syntax.value.c_str());
    case SyntaxType::binary_exp:
        double left  = evaluate(syntax.children[0]);
        double right = evaluate(syntax.children[2]);
        if(syntax.children[1].value == "+") return left+right;
        if(syntax.children[1].value == "-") return left-right;
        if(syntax.children[1].value == "*") return left*right;
        if(syntax.children[1].value == "/") return left/right;
    }
        throw;
}

int main()
{
    vector<Token> tokens{{TokenType::constant, "4.0"},
                         {TokenType::binary_op, "*"},
                         {TokenType::constant, "2"},
                         {TokenType::binary_op, "+"},
                         {TokenType::constant, "1.5"}};

    Syntax syntax = parse(tokens);

    cout<<"4.0 * 2 + 1.5 = "<<evaluate(syntax)<<"\n";

    return 0;
}
```