C++ Proposal:
Extensions of Scoped Enumerations

Philip Ginsbach
philip.ginsbach@ed.ac.uk

December 3, 2018

abstract

I propose an extension mechanism for scoped enumerations, similar in spirit to class inheritance. This allows a scoped enumeration to contain another scoped enumeration as a subset. This crucially allows implicit typesafe casting from enumerators in the base enumeration to the extension.

Contents

1 Introduction

Scoped enumerations ("enum classes") provide a type safe alternative to C-style enums.

It can be desirable to have a subset of an enumeration to be another self-contained enumeration. For example, there might be an enumeration for types of leaves in a tree and another enumeration for types of nodes in a tree. In this case, the choices for types of generic nodes is an extension of the choices for types of leaves.

There is currently no way to express this relationship properly in C++ but it is desirable from a standpoint of modularity. This proposal is for a simple, sound and practically useful extension to the core language standard to provide a mechanism for extending scoped enumerations. This retains the type safe properties of scoped enumerations, introduces no runtime overhead and uses existing syntax.

2 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); break;
    case TokenType::binary_op:
        stack.emplace_back(SyntaxType::binary_op); break; }
```

This problem can be avoided by discarding the definition of TokenType and by using SyntaxType everywhere. However, this breaks modularity: why should the lexer have to be aware of the existence of a parser when generating a stream of tokens?

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

3 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. This is natural and well understood in type theory. It arises from that fact that enumerations in C++ are an instance of *sum* types, whereas structs are an instance of *product* types. The terms *sum* and *product* here are to be understood in their category theory sense. The inversion of the subtype/supertype relationship is then a direct reflection of the fact that product and direct sum (i.e. coproduct) are duals.

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. Enumerator values can not be renamed and already existing enumerator names can not be assigned a new value.

3.1 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
```

4 Discussion

4.1 std::underlying_type

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

4.2 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.

4.3 Extensions vs Restrictions

There are other important reasons to have enums overlap that are not covered by this proposal. Consider the following examples of the days of the week.

```
enum class WeekDays : int {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
enum class WorkDays : int {Mon, Tue, Wed, Thu, Fri};
enum class WeekendDays: : int {Sat, Sun};
enum class ProductiveDays : int {Tue, Wed, Thu};
```

I argue that this is an entirely different problem. There is a clear complete set and parts of it are predicated with different additional properties, as can be expressed with boolean predicate functions. It might be desirable to embed this into the type system but it is quite unrelated to what this proposal is about.

```
bool isWorkDay(WeekDay day) { return day==WeekDay::Mon
             || day==WeekDay::Tue || day==WeekDay::Wed
             || day==WeekDay::Thu || day==WeekDay::Fri };
bool isWeekendDay(WeekDay day) { return day==WeekDay::Sat
                                     || day==WeekDay::Sun };
bool isProductiveDay(WeekDay day) { return day==WeekDay::Tue
                   || day==WeekDay::Wed || day==WeekDay::Thu };
```

This does not necessitate what this proposal provides, which is a form of modularity. In the motivating example, tokens should not be thought of as a subset of syntactical constructs, they are on their own a complete thing. There is no reason why a lexer should even be aware of the existence of a parser and this is reflected in the fact that the definition of TokenType stands on its own.

On the other hand, the definition of SyntaxType incorporates TokenType but it doesn't need to be aware of its contents but can treat it as an opaque thing, just like a derived class doesn't need to be aware of the internals of its base class.

4.4 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.

4.5 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.

4.6 Syntax

With this proposal, there are three uses of the same syntax that mean different things.

"`class Derive : Base {`"
: This is class inheritance. `Derived` is now a subtype of `Base`, i.e. a `Derive` can be used as a `Base`.

"`enum class Extended : Base {`"
: This extends a scoped enumeration. `Extended` is now a supertype of `Base`, i.e. a `Base` can be used as an `Extended`.

"`enum class Enum : int {`"
: This specifies the integral type of a scoped enum.

This situation is not perfect but a compromise in order not to introduce new syntax or keywords. There have been several other suggestions such as the following.

"`enum class Extended >: Base {`"
: This is more in line wih type theory notation.

"`enum class Extended : extend Base {`"
: This is very clear but requires a new keyword.

"`enum class Extended : continue Base {`"
: This requires no new keyword but is arguably less clear than "extends".

5 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;
}
```