Improving C++ Enums: Adding Serialization, Inheritance, and Iteration

Hugo González Castro, 16 Apr 2009 CPOL

4.05 (13 votes)

Rate: ✗ ✗ ✗ ✗ ✗ ✗

A different approach to C++ enums: enum to string, enum extensions, and enum iterations.

- **Download source and examples - 8.52 KB**

Introduction

I was looking for an enumerated type which is easy to serialize (to write to and read from a file, for example). I didn't want to care about the internal integer numbers associated with the `enum`, or the overlapping problems with these numbers. I just wanted to care about the names of the enumerated items. I also wanted an easy way to extend the `enum` definitions, to work with polymorphic classes. I wanted them to make a list of errors without using numbers, and I also wanted to make a list of `string` IDs without using numbers. That was the reason I didn't care at the beginning about iterations or assigning specific values to each item. At the end, I also decided to implement these possibilities, as maybe someone could find them useful.

Background

There are several interesting articles about `enum`s that I have read before deciding to implement my own solution:

- C++ Tip: Simplify your coding with user-friendly enumerations - Thanks for the idea of using namespaces with enumerations.
- Converting C++ enums to strings - This article gave me some ideas to implement my own solution, but I used a different approach.

- Inheriting a C++ enum type - How to extend an enum definition. See how my approach below solves this problem without overlapping problems.
- Enum iteration and string conversion - One of the best solutions I've seen. I didn't use it because it doesn't support enum extensions/inheritance.

Using the Code

The key idea of my approach is to separate the list of enumerated items from the definition of the enumeration type itself. So, the first thing you have to do is to create a file with the list of items you want to enumerate. For example, declare "*DayEnum.h*":

```
// In DayEnum.h
ENUMITEM(Sunday)
ENUMITEM(Monday)
ENUMITEM(Tuesday)
ENUMITEM(Wednesday)
ENUMITEM(Thursday)
ENUMITEM(Friday)
ENUMITEM(Saturday)
```

The macro `ENUMITEM()` is implemented in several different ways to do the hard job. It is also possible to use`ENUMITEM_VALUE(,)` to assign specific values only to the items you want, but it is not recommended as this could create conflicts with `enum` extensions/inheritance. I recommend using only `ENUMITEM()` whenever possible.

The following example code will declare the `enum` type and the functions to convert the `enum` to a `string`, and to convert a `string` to an `enum`, to iterate, and to count items. All these will be defined (and encapsulated) in a namespace (as recommended in the first background article) or in a `static` class (from version 5.0 of this code, which allows a subclass definition), so we avoid all conflicts with other `enum` definitions.

```
// Declaration of the enumeration:
///////////////////////////////////////////////////
// Input parameters:
// IMPROVED_ENUM_NAME - the name of the enumeration
// IMPROVED_ENUM_FILE - the file with the enum items
// DefineImprovedEnum:
// Batch file of preprocessing commands (uses the input parameters above)
///////////////////////////////////////////////////
#define IMPROVED_ENUM_NAME  Day
#define IMPROVED_ENUM_FILE "DayEnum.h"
#include "DefineImprovedEnum.h"
```

Some authors would prefer to write `??=include "DefineImprovedEnum.h"` to make clear the different use of the include file. "The ??= token is a trigraph for #", but trigraphs are not recognized by all compilers.

As a new feature (from version 4.0 of this code) all the code above can also be defined in the same place (without the need of a separated file):

```
// Inline declaration of the enumeration:
#define IMPROVED_ENUM_NAME  Day
#define IMPROVED_ENUM_LIST  ENUMITEM(Sunday)    \
                          ENUMITEM(Monday)    \
                          ENUMITEM(Tuesday)   \
                          ENUMITEM(Wednesday) \
                          ENUMITEM(Thursday)  \
                          ENUMITEM(Friday)    \
                          ENUMITEM(Saturday)
#include "DefineImprovedEnum.h"
```

Using the enumeration is as simple as this:

```
void Test()
{
    // Conversion to string and conversion to enum:
    Day::EnumType t = Day::Monday;
    std::string text = Day::Enum2String(t);
    t = Day::String2Enum("Friday");

    // Iteration:
    t = Day::FirstEnumItem();
    t = Day::NextEnumItem(t);
    t = Day::LastEnumItem();
    t = Day::PreviousEnumItem(t);

    // Count:
    int n = Day::NumberOfValidEnumItem();
}
```

At the end of `Test()`, the value of `t` is `Friday`, and the value of `text` is "`Monday`".

How to Extend an Enum Definition (Inherit from Another Enum)

Another approach to solve the problem of Inheriting a C++ enum type with this code is to do the following:

```
// In Fruit.h
ENUMITEM(Orange)
ENUMITEM(Mango)
ENUMITEM(Banana)
```

and:

```
// In NewFruit.h
```

```
ENUMITEM(Apple)
ENUMITEM(Pear)
```

Then, as we have separated the lists of items, we can create a new list with all the items:

```
// In MyFruit.h
#include "Fruit.h"
#include "NewFruit.h"
```

And, declare the new `enum` type:

```
// Declaration of the extended enumeration:
#define IMPROVED_ENUM_NAME  MyFruit
#define IMPROVED_ENUM_FILE "MyFruit.h"
#include "DefineImprovedEnum.h"
```

That approach works well, but we don't have an easy way to convert from the base `enum` type to the extended`enum` type. So, I decided to directly implement some inheritance functions to extend the functionality. The following example code will declare the base `enum` and the extended `enum`, with the extended functionality:

```
// Declaration of the base enum (Fruit):
#define IMPROVED_ENUM_NAME  Fruit
#define IMPROVED_ENUM_FILE "Fruit.h"
#include "DefineImprovedEnum.h"

// Declaration of the extended enum
// (MyFruit inherits from Fruit, extended with NewFruit):
#define IMPROVED_ENUM_NAME  MyFruit
#define IMPROVED_ENUM_FILE "NewFruit.h"
#define IMPROVED_ENUM_INHERITED_NAME  Fruit
#define IMPROVED_ENUM_INHERITED_FILE "Fruit.h"
#include "DefineImprovedEnum.h"
```

Each `enum` definition has its own namespace and there are no overlapping problems.
The `DefineImprovedEnum`batch file defines the functions to convert items from one namespace to items of another namespace. Using the extended enumeration is as simple as this:

```
// Accepts only the base type
void eat(Fruit::EnumType fruit) {};
// Accepts only the extended type
void consume(MyFruit::EnumType myfruit) {};

void ExtendedTest()
{
    // Declarations:
    Fruit::EnumType fruitAux, fruit;
    MyFruit::EnumType myfruitAux, myfruit, newfruit;
    // Direct assignments:
    fruit    = Fruit::Orange; // OK
    myfruit  = MyFruit::Orange; // OK
    newfruit = MyFruit::Apple; // OK

    // Conversions to extended enum:
    myfruitAux = MyFruit::Inherited2Enum(fruit); // OK
    myfruitAux = MyFruit::Inherited2Enum(myfruit); // OK
    myfruitAux = MyFruit::Inherited2Enum(newfruit); // OK

    // Conversions to base enum:
    fruitAux = MyFruit::Enum2Inherited(fruit); // OK
    fruitAux = MyFruit::Enum2Inherited(myfruit); // OK
    fruitAux = MyFruit::Enum2Inherited(newfruit); // returns NotValidEnumItem

    // The following compiles:
    eat(fruit); // OK
    eat(MyFruit::Enum2Inherited(myfruitAux)); // Possible NotValidEnumItem
    consume(myfruit); // OK
    consume(MyFruit::Inherited2Enum(fruit)); // OK

    // Partial iteration:
    myfruitAux = MyFruit::FirstExtendedEnumItem();
    myfruitAux = MyFruit::NextInheritedEnumItem(newfruit);
    myfruitAux = MyFruit::LastInheritedEnumItem();
    myfruitAux = MyFruit::PreviousExtendedEnumItem(newfruit);

    // Partial count:
    int n = MyFruit::NumberOfInheritedValidEnumItem();
    int m = MyFruit::NumberOfExtendedValidEnumItem();
}
```

Implementation

The implementation is based on a batch file for preprocessing commands, called "*DefineImprovedEnum.h*":

```
// In DefineImprovedEnum.h

////////////////////////////////////////////////////////////////////////
// IMPORTANT NOTE:
// This is a "batch file of preprocessing directives"
// (because this cannot be done with a macro).
// Each time you include this file you are calling a batch file,
// it doesn't work as a macro include.
// If you want to declare several different enum types,
// you have to include this file several times.
// Do not use "#pragma once" directive, because it would have
// unexpected behaviour and results.
// Do not use directives like:
// #ifndef _IMPROVED_ENUM_H_ ; #define _IMPROVED_ENUM_H_ (same reason).
///////////////////////////////////////////////////////////////////////
// AUTHOR:      Hugo González Castro
// TITLE:       Improving C++ Enum: Adding Serialization,
//                                 Inheritance and Iteration.
// DESCRIPTION: A different approach to C++ enums: enum to string,
//             enum extension and enum iteration.
// VERSION:     v5.0 - 2009/04/13
// LICENSE:     CPOL (Code Project Open License).
//             Please, do not remove nor modify this header.
// URL:         ImprovedEnum.aspx
///////////////////////////////////////////////////////////////////////
// INPUT PARAMETERS:
// This file needs the following input parameters to be defined
// before including it:
// Input parameter: the name of the enumeration
// #define IMPROVED_ENUM_NAME [NameOfYourEnum]
// Input parameter: the file with the enum items
// #define IMPROVED_ENUM_FILE ["EnumItemFile"]
///////////////////////////////////////////////////////////////////////
// ENUMITEM FILE:
// The EnumItemFile is a list (one per line) of:
// ENUMITEM(EnumItem) or ENUMITEM_VALUE(EnumItem, Value)
///////////////////////////////////////////////////////////////////////
// ALTERNATIVE TO ENUMITEM FILE:
// IMPROVED_ENUM_LIST instead of IMPROVED_ENUM_FILE
// #define IMPROVED_ENUM_LIST  ENUMITEM(Item1) ... ENUMITEM(LastItem)
// #define IMPROVED_ENUM_LIST  ENUMITEM(Item1) \
//                           ENUMITEM(Item2) \
//                           ...
//                           ENUMITEM(LastItem)
///////////////////////////////////////////////////////////////////////
// OPTIONAL INPUT PARAMETERS:
// If you want to define a subclass instead of a namespace, you can
// #define IMPROVED_ENUM_SUBCLASS, or
// #define IMPROVED_ENUM_SUBCLASS_PARENT [ParentClass]
// to make subclass inherit from a ParentClass.
// If you want to extend an already defined ImprovedEnum, you have to
// define which type do you want to extend with
// IMPROVED_ENUM_INHERITED_NAME and IMPROVED_ENUM_INHERITED_FILE
// input parameters.
///////////////////////////////////////////////////////////////////////

// Checking ENUMITEM and ENUMITEM_VALUE macros are not already defined
#if defined(ENUMITEM)
#error ENUMITEM macro cannot be already defined
```

```
#elif defined(ENUMITEM_VALUE)
#error ENUMITEM_VALUE macro cannot be already defined
#endif

// Standard string class
#include <string>


#if defined(IMPROVED_ENUM_SUBCLASS_PARENT)

//! We define the IMPROVED_ENUM_NAME subclass (that
//! inherits from the specified parent class) which contains
//! the enum type and the static conversion methods from the
//! enum type to the string type and vice versa.
///////////////////////////////////////////////////////////
#define STATIC_METHOD static
class IMPROVED_ENUM_NAME : public IMPROVED_ENUM_SUBCLASS_PARENT
{
public:

#elif defined(IMPROVED_ENUM_SUBCLASS)

//! We define the IMPROVED_ENUM_NAME subclass, which contains
//! the enum type and the static conversion methods from the
//! enum type to the string type and vice versa.
///////////////////////////////////////////////////////////
#define STATIC_METHOD static
class IMPROVED_ENUM_NAME
{
public:

#else // IMPROVED_ENUM_SUBCLASS || IMPROVED_ENUM_SUBCLASS_PARENT

//! We define the IMPROVED_ENUM_NAME namespace, which contains
//! the enum type and the conversion functions from the
//! enum type to the string type and vice versa.
///////////////////////////////////////////////////////////
#define STATIC_METHOD
namespace IMPROVED_ENUM_NAME
{

#endif // IMPROVED_ENUM_SUBCLASS || IMPROVED_ENUM_SUBCLASS_PARENT

    //! Some stuff to get the string of the IMPROVED_ENUM_NAME
    ///////////////////////////////////////////////////////////
    #define GET_MACRO_STRING_EXPANDED(Macro)  #Macro
    #define GET_MACRO_STRING(Macro)  GET_MACRO_STRING_EXPANDED(Macro)
    #define ENUM_SEPARATOR  "::"
    #define ENUM_TYPE_NAME  GET_MACRO_STRING(IMPROVED_ENUM_NAME)
    STATIC_METHOD inline const std::string EnumSeparator() { return ENUM_SEPARATOR; }
    STATIC_METHOD inline const std::string EnumTypeName() { return ENUM_TYPE_NAME; }
    #ifdef  IMPROVED_ENUM_INHERITED_NAME
    #define PARENT_ENUM_TYPE_NAME  GET_MACRO_STRING(IMPROVED_ENUM_INHERITED_NAME)
    #define FULL_ENUM_TYPE_NAME    PARENT_ENUM_TYPE_NAME  ENUM_SEPARATOR  ENUM_TYPE_NAME
    #else //IMPROVED_ENUM_INHERITED_NAME
    #define PARENT_ENUM_TYPE_NAME  ""
    #define FULL_ENUM_TYPE_NAME    ENUM_TYPE_NAME
    #endif//IMPROVED_ENUM_INHERITED_NAME
    STATIC_METHOD inline const std::string ParentEnumTypeName()
                                    { return PARENT_ENUM_TYPE_NAME; }
    STATIC_METHOD inline const std::string FullEnumTypeName()
                                    { return FULL_ENUM_TYPE_NAME; }


    //! This defines the enumerated type:
    //////////////////////////////////////////
    typedef enum EnumTypeTag
    {
        //////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // a list of items separated by commas:
        #define  ENUMITEM(EnumItem) EnumItem,
        #define  ENUMITEM_VALUE(EnumItem, Value) EnumItem = Value,
        #ifdef   IMPROVED_ENUM_INHERITED_FILE
        #include IMPROVED_ENUM_INHERITED_FILE
        #endif// IMPROVED_ENUM_INHERITED_FILE
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        //////////////////////////////////////////
        NotValidEnumItem // We add this item to all enums
    } EnumType, Type;

    //! Conversion from enum to string:
    //////////////////////////////////////////
    STATIC_METHOD inline const std::string Enum2String(const EnumType& t)
    {
        switch (t)
        {
        //////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // a CASE list which returns the stringized value:
        #define  ENUMITEM(EnumItem) case EnumItem : return #EnumItem;
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
    #ifdef   IMPROVED_ENUM_INHERITED_FILE
    #include IMPROVED_ENUM_INHERITED_FILE
    #endif// IMPROVED_ENUM_INHERITED_FILE
    #ifdef   IMPROVED_ENUM_FILE
    #include IMPROVED_ENUM_FILE
    #else // IMPROVED_ENUM_LIST
            IMPROVED_ENUM_LIST
    #endif// IMPROVED_ENUM_FILE
    #undef   ENUMITEM_VALUE
    #undef   ENUMITEM
    ////////////////////////////////////////////
    }
    return ""; // NotValidEnumItem
}

//! Conversion from enum to full string (namespace::string):
/////////////////////////////////////////////////////////////////
STATIC_METHOD inline const std::string Enum2FullString(const EnumType& t)
{
    switch (t)
    {
    ////////////////////////////////////////////
    // With this mini-macro we make ENUMITEM file/s
    // a CASE list which returns the stringized value:
    #define  ENUMITEM(EnumItem) \
    case EnumItem : return  FULL_ENUM_TYPE_NAME  ENUM_SEPARATOR  #EnumItem;
    #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
    #ifdef   IMPROVED_ENUM_INHERITED_FILE
    #include IMPROVED_ENUM_INHERITED_FILE
    #endif// IMPROVED_ENUM_INHERITED_FILE
    #ifdef   IMPROVED_ENUM_FILE
    #include IMPROVED_ENUM_FILE
    #else // IMPROVED_ENUM_LIST
            IMPROVED_ENUM_LIST
    #endif// IMPROVED_ENUM_FILE
    #undef   ENUMITEM_VALUE
    #undef   ENUMITEM
    ////////////////////////////////////////////
    }
    return ""; // NotValidEnumItem
}

//! Conversion from string to enum:
////////////////////////////////////////////
STATIC_METHOD inline const EnumType String2Enum(const std::string& s)
{
    if (s == "") return NotValidEnumItem;
    ////////////////////////////////////////////
    // With this mini-macro we make ENUMITEM file/s
```

```
        // an IF list which returns the enum item:
        #define  ENUMITEM(EnumItem) if (s == #EnumItem) return EnumItem;
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #ifdef   IMPROVED_ENUM_INHERITED_FILE
        #include IMPROVED_ENUM_INHERITED_FILE
        #endif// IMPROVED_ENUM_INHERITED_FILE
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        ///////////////////////////////////////////
        return NotValidEnumItem;
    }

    //! Conversion from full string (namespace::string) to enum:
    ////////////////////////////////////////////////////////////
    STATIC_METHOD inline const EnumType FullString2Enum(const std::string& s)
    {
        if (s == "") return NotValidEnumItem;
        ///////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // an IF list which returns the enum item:
        #define  ENUMITEM(EnumItem) \
        if (s ==  FULL_ENUM_TYPE_NAME  ENUM_SEPARATOR  #EnumItem) return EnumItem;
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #ifdef   IMPROVED_ENUM_INHERITED_FILE
        #include IMPROVED_ENUM_INHERITED_FILE
        #endif// IMPROVED_ENUM_INHERITED_FILE
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        ///////////////////////////////////////////
        return NotValidEnumItem;
    }

    //! Enum iteration to next:
    ////////////////////////////////////////
    STATIC_METHOD inline const EnumType NextEnumItem(const EnumType& t)
    {
        switch (t)
        {
        case NotValidEnumItem :
```

```
    ///////////////////////////////////////////
    // With this mini-macro we make ENUMITEM file/s
    // a CASE list which returns the next item:
    #define  ENUMITEM(EnumItem) return EnumItem; case EnumItem :
    #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
    #ifdef   IMPROVED_ENUM_INHERITED_FILE
    #include IMPROVED_ENUM_INHERITED_FILE
    #endif// IMPROVED_ENUM_INHERITED_FILE
    #ifdef   IMPROVED_ENUM_FILE
    #include IMPROVED_ENUM_FILE
    #else // IMPROVED_ENUM_LIST
            IMPROVED_ENUM_LIST
    #endif// IMPROVED_ENUM_FILE
    #undef   ENUMITEM_VALUE
    #undef   ENUMITEM
    ///////////////////////////////////////////
               return NotValidEnumItem; // (This indentation is intentional)
    }
    return NotValidEnumItem; // (This line is intentional too, do not remove)
}

//! Enum iteration to previous:
///////////////////////////////////////////
STATIC_METHOD inline const EnumType PreviousEnumItem(const EnumType& t)
{
    EnumType tprev = NotValidEnumItem;
    ///////////////////////////////////////////
    // With this mini-macro we make ENUMITEM file/s
    // an IF list which returns the previous item:
    #define  ENUMITEM(EnumItem) \
    if (t == EnumItem) return tprev; else tprev = EnumItem;
    #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
    #ifdef   IMPROVED_ENUM_INHERITED_FILE
    #include IMPROVED_ENUM_INHERITED_FILE
    #endif// IMPROVED_ENUM_INHERITED_FILE
    #ifdef   IMPROVED_ENUM_FILE
    #include IMPROVED_ENUM_FILE
    #else // IMPROVED_ENUM_LIST
            IMPROVED_ENUM_LIST
    #endif// IMPROVED_ENUM_FILE
    #undef   ENUMITEM_VALUE
    #undef   ENUMITEM
    ///////////////////////////////////////////
    return tprev;
}

//! The first and the last Enums:
///////////////////////////////////////////
STATIC_METHOD inline const EnumType FirstEnumItem()
```

```cpp
                                { return NextEnumItem(NotValidEnumItem); }
    STATIC_METHOD inline const EnumType LastEnumItem()
                                { return PreviousEnumItem(NotValidEnumItem); }

    //! Number of enum items:
    ///////////////////////////////////////////
    STATIC_METHOD inline const int NumberOfValidEnumItem()
    {
        return 0
        ///////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // a counter list:
        #define  ENUMITEM(EnumItem) +1
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #ifdef   IMPROVED_ENUM_INHERITED_FILE
        #include IMPROVED_ENUM_INHERITED_FILE
        #endif// IMPROVED_ENUM_INHERITED_FILE
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        ///////////////////////////////////////////
        ;
    }

    // This is only needed when working with inherited/extended enums:
    ////////////////////////////////////////////////////////////////////
    #ifdef IMPROVED_ENUM_INHERITED_NAME
        //! Conversion from inherited enums:
        //! The same class items are returned without change, but
        //! other items are converted from one namespace to the other:
        ///////////////////////////////////////////
        STATIC_METHOD inline const EnumType Inherited2Enum(const EnumType& t)
                                                { return t; }
        STATIC_METHOD inline const EnumType Inherited2Enum(
                            const IMPROVED_ENUM_INHERITED_NAME::EnumType& t)
        {
            switch (t)
            {
            ///////////////////////////////////////////
            // With this mini-macro we make ENUMITEM file
            // a CASE list which returns the converted value
            // from one namespace to the other:
            #define  ENUMITEM(EnumItem) \
            case IMPROVED_ENUM_INHERITED_NAME::EnumItem : return EnumItem;
            #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
            #ifdef   IMPROVED_ENUM_INHERITED_FILE
            #include IMPROVED_ENUM_INHERITED_FILE
            #endif// IMPROVED_ENUM_INHERITED_FILE
            #undef   ENUMITEM_VALUE
            #undef   ENUMITEM
            ////////////////////////////////////////////
            }
            return NotValidEnumItem;
        }

        //! Conversion to inherited enums:
        //! The same class items are returned without change, but
        //! other items are converted from one namespace to the other:
        ////////////////////////////////////////////
        STATIC_METHOD inline const IMPROVED_ENUM_INHERITED_NAME::EnumType Enum2Inherited(
                                   const IMPROVED_ENUM_INHERITED_NAME::EnumType& t)
                                                                    { return t; }
        STATIC_METHOD inline const IMPROVED_ENUM_INHERITED_NAME::EnumType Enum2Inherited(
                                                                const EnumType& t)
        {
            switch (t)
            {
            ////////////////////////////////////////////
            // With this mini-macro we make ENUMITEM file
            // a CASE list which returns the converted value
            // from one namespace to the other:
            #define  ENUMITEM(EnumItem) \
            case EnumItem : return IMPROVED_ENUM_INHERITED_NAME::EnumItem;
            #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
            #ifdef   IMPROVED_ENUM_INHERITED_FILE
            #include IMPROVED_ENUM_INHERITED_FILE
            #endif// IMPROVED_ENUM_INHERITED_FILE
            #undef   ENUMITEM_VALUE
            #undef   ENUMITEM
            ////////////////////////////////////////////
            }
            return IMPROVED_ENUM_INHERITED_NAME::NotValidEnumItem;
        }

        //! Enum iteration to next extended (not inherited):
        ////////////////////////////////////////////
        STATIC_METHOD inline const EnumType NextExtendedEnumItem(
                                             const EnumType& t)
        {
            switch (t)
            {
            case NotValidEnumItem :
            ////////////////////////////////////////////
            // With this mini-macro we make ENUMITEM file/s
```

```
        // a CASE list which returns the next item:
        #define  ENUMITEM(EnumItem) return EnumItem; case EnumItem :
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        ////////////////////////////////////////////
                              return NotValidEnumItem;
        }
        return NotValidEnumItem;
    }

    //! Enum iteration to previous extended (not inherited):
    ////////////////////////////////////////////
    STATIC_METHOD inline const EnumType PreviousExtendedEnumItem(
                                        const EnumType& t)
    {
        EnumType tprev = NotValidEnumItem;
        ////////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // an IF list which returns the previous item:
        #define  ENUMITEM(EnumItem) \
        if (t == EnumItem) return tprev; else tprev = EnumItem;
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        ////////////////////////////////////////////
        return tprev;
    }

    //! The first and the last extended Enums:
    ////////////////////////////////////////////
    STATIC_METHOD inline const EnumType FirstExtendedEnumItem()
                      { return NextExtendedEnumItem(NotValidEnumItem); }
    STATIC_METHOD inline const EnumType LastExtendedEnumItem()
                      { return PreviousExtendedEnumItem(NotValidEnumItem); }

    //! Number of extended enum items:
    ////////////////////////////////////////////
    STATIC_METHOD inline const int NumberOfExtendedValidEnumItem()
    {
        return 0
        //////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file
        // a counter list:
        #define  ENUMITEM(EnumItem) +1
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #ifdef   IMPROVED_ENUM_FILE
        #include IMPROVED_ENUM_FILE
        #else // IMPROVED_ENUM_LIST
                IMPROVED_ENUM_LIST
        #endif// IMPROVED_ENUM_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        //////////////////////////////////////////
        ;
    }

    //! Enum iteration to next inherited:
    //////////////////////////////////////////
    STATIC_METHOD inline const EnumType NextInheritedEnumItem(
                                        const EnumType& t)
    {
        switch (t)
        {
        case NotValidEnumItem :
        //////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // a CASE list which returns the next item:
        #define  ENUMITEM(EnumItem) return EnumItem; case EnumItem :
        #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
        #include IMPROVED_ENUM_INHERITED_FILE
        #undef   ENUMITEM_VALUE
        #undef   ENUMITEM
        //////////////////////////////////////////
                              return NotValidEnumItem;
        }
        return NotValidEnumItem;
    }

    //! Enum iteration to previous inherited:
    //////////////////////////////////////////
    STATIC_METHOD inline const EnumType PreviousInheritedEnumItem(
                                            const EnumType& t)
    {
        EnumType tprev = NotValidEnumItem;
        //////////////////////////////////////////
        // With this mini-macro we make ENUMITEM file/s
        // an IF list which returns the previous item:
            #define  ENUMITEM(EnumItem) \
            if (t == EnumItem) return tprev; else tprev = EnumItem;
            #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
            #include IMPROVED_ENUM_INHERITED_FILE
            #undef   ENUMITEM_VALUE
            #undef   ENUMITEM
            ////////////////////////////////////////
            return tprev;
        }

        //! The first and the last inherited Enums:
        ////////////////////////////////////////
        STATIC_METHOD inline const EnumType FirstInheritedEnumItem()
                              { return NextInheritedEnumItem(NotValidEnumItem); }
        STATIC_METHOD inline const EnumType LastInheritedEnumItem()
                              { return PreviousInheritedEnumItem(NotValidEnumItem); }

        //! Number of inherited enum items:
        ////////////////////////////////////////
        STATIC_METHOD inline const int NumberOfInheritedValidEnumItem()
        {
            return 0
            ////////////////////////////////////////
            // With this mini-macro we make ENUMITEM file
            // a counter list:
            #define  ENUMITEM(EnumItem) +1
            #define  ENUMITEM_VALUE(EnumItem, Value) ENUMITEM(EnumItem)
            #include IMPROVED_ENUM_INHERITED_FILE
            #undef   ENUMITEM_VALUE
            #undef   ENUMITEM
            ////////////////////////////////////////
            ;
        }

    #endif // IMPROVED_ENUM_INHERITED_NAME

    // Free temporary macros:
    ///////////////////////////
    #undef STATIC_METHOD
    #undef ENUM_SEPARATOR
    #undef ENUM_TYPE_NAME
    #undef PARENT_ENUM_TYPE_NAME
    #undef FULL_ENUM_TYPE_NAME
    #undef GET_MACRO_STRING
    #undef GET_MACRO_STRING_EXPANDED
}
#if defined(IMPROVED_ENUM_SUBCLASS) || defined(IMPROVED_ENUM_SUBCLASS_PARENT)
;
#endif
```

```
// Free this file's parameters:
////////////////////////////////
#undef IMPROVED_ENUM_NAME
#undef IMPROVED_ENUM_FILE
#undef IMPROVED_ENUM_LIST
#undef IMPROVED_ENUM_SUBCLASS
#undef IMPROVED_ENUM_SUBCLASS_PARENT
#undef IMPROVED_ENUM_INHERITED_NAME
#undef IMPROVED_ENUM_INHERITED_FILE
// Do not use directives like: #endif (reason above)
```

Points of Interest

I found no way to make the `#include` or `#define` directives inside a macro, so I think this cannot be done with a standard macro definition. I decided to make a file with all the preprocessing directives I needed, and then include it wherever I needed it in my code. The only problem I see with this approach is the way I have to pass the arguments/parameters to this file, because the code to define the `enum` type is not as clear as in the other solutions. Anyway, for my particular problem, it was the best solution I found. Any constructive comments and ideas are welcome.

Apart from that, I was exploring the concept of inheritance in enumerations and compared it with class inheritance. Derived classes add variables and methods to base classes, and derived/extended `enum` adds items to the base`enum`. I wanted to mix derived classes with respective derived `enum`s (in a polymorphic pattern) as I thought it was a nice idea, but that could be nonsense (as class inheritance is for specialization, but `enum` inheritance is for extension). As you can have a method that takes a pointer of the base class but can be called with derived classes (using polymorphism), I wanted a method that takes "a base `enum`" but can be called with "derived `enum`s". This can be done encapsulating each `enum` in a class, and that is the new approach with `IMPROVED_ENUM_SUBCLASS`and `IMPROVED_ENUM_SUBCLASS_PARENT` input parameters. Ideas are welcome...

History

- v1.0 - 2008/12/16
  - First version with `Enum2String` and `String2Enum`
- v2.0 - 2008/12/22
  - Added `Enum2Inherited` and `Inherited2Enum`
- v3.0 - 2008/12/23
  - Added `Count` and `Iteration` and published on The Code Project
- v4.0 - 2009/04/02
  - Added `IMPROVED_ENUM_LIST` for inline `enum` declaration, `Enum2FullString` and`FullString2Enum` to generate unambiguous `string`s when working with several `enum`s, all functions declared `inline` to enable the use on header files without problems, and a "vcproj" example.
- v5.0 - 2009/04/13
  - Added `IMPROVED_ENUM_SUBCLASS` and `IMPROVED_ENUM_SUBCLASS_PARENT` to encapsulate the`enum` in a `static` class or a `static` derived class instead of a namespace. The calling syntax is exactly the same as

the syntax in the namespace. You cannot declare a namespace inside a class, but with this option now you can declare `ImprovedEnums` inside a class. The examples have also been updated.

License

About the Author

Hugo González Castro

Software Developer (Senior)
Spain
B.Sc. Mathematics and Computer Science.
Programming in C++ since 2003.

Follow on Twitter Google