Not quite what you are looking for? You may want to try:

- Typesafe C++ Enum with ToString() and FromString()
- Converting C++ enums to strings

10,479,484 members (70,325 online)   ░░Workspaces   1   Member 8285579 ▼   380   Sign out ⊗

CODE PROJECT ®
For those who code

home   **articles**   **quick answers**   **discussions**   **features**   **community**   **help**

enum c++   🔍

Articles » Languages » C / C++ Language » General   Next →

Article
Browse Code
Bugs / Suggestions
Stats
Revisions (4)
Alternatives

Comments & Discussions (18)

Add your own alternative version

Improving **C++ Enum**s: Adding Serialization, Inheritance, and Iteration

By **Hugo González Castro**, 15 Apr 2009

★★★★☆   4.05 (13 votes)

Rate:

Fork me on ░░Workspaces

⚠️ **Is your email address OK?** You are signed up for our newsletters but your email address is either unconfirmed, or has not been reconfirmed in a long time. Please click **here** to have a confirmation email sent so we can confirm your email address and start sending you newsletters again. Alternatively, you can **update your subscriptions**.

Download source and examples - 8.52 KB

Introduction

I was looking for an **enum**erated 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 **enum**erated 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 **enum**erations - Thanks for the idea of using namespaces with **enum**erations.
- Converting **C++ enum**s 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.

Top News

Steve Ballmer's temper tantrum over Nokia buyout led to his firing, says report

Get the Insider News free each morning.

Related Videos

Android Development for Beginners
learntoprogram .tv

Using the Code

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

Collapse | Copy Code

```
// 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.

Collapse | Copy Code

```
// 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):

Collapse | Copy Code

```
// 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 **enum**eration is as simple as this:

Collapse | Copy Code

```
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);
```

Related Articles

Typesafe C++ Enum with ToString() and FromString()

XML Serialization of Generic Dictionary, Multidimensional Array, and Inherited Type, with sharpSerializer .NET

QxOrm - C++ ORM (Object Relational Mapping) Library

Add Custom Properties to a PropertyGrid

XML Serialization of a Class Inherited from Generic List of type Interface

Optimizing Serialization in .NET

.NET XML and SOAP Serialization Samples, Tips

Enums and Structs in C#

Yet Another XML Serialization Library for the .NET Framework

NetSerializer - A Fast, Simple Serializer for .NET

Exposing Dynamic Events in the WinForms Designer

Binding ASP.NET DropDownList to data structures

ASP.NET Advanced Generic Handler ASHX

Enumeration Types do not Enumerate! Working around .NET and Language Limitations

Automatically generate classes and enums from SQL Server datatables

Complete Comparison for VB.NET and C#

ASP.NET controls to display enum values

Attributes in C#

Generic Finite State Machine (FSM)

DevGlobalCache – A way to Cache and Share data between processes

Related Research

7 Tips for iOS 7

```
    // 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:

⊟ Collapse | Copy Code
```
// In Fruit.h
ENUMITEM(Orange)
ENUMITEM(Mango)
ENUMITEM(Banana)
```

and:

⊟ Collapse | Copy Code
```
// 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:

⊟ Collapse | Copy Code
```
// In MyFruit.h
#include "Fruit.h"
#include "NewFruit.h"
```

And, declare the new **enum** type:

⊟ Collapse | Copy Code
```
// 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:

⊟ Collapse | Copy Code
```
// 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 **enum**eration is as simple as this:

⊟ Collapse | Copy Code
```
// 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*":

□ Collapse | Copy Code

```
// 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
```

```cpp
// #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; }
```

```cpp
//! 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
```

```cpp
                    #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
```

```cpp
                    //////////////////////////////////////
                    return tprev;
        }

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

```cpp
            // 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 **enum**erations 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 `ImprovedEnum`s inside a class. The examples have also been updated.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

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

Article Top

Comments and Discussions

```
Add a Comment or Question    ?
                                          Search this forum  [              ]  Go

  ☑ Profile popups   Spacing  [Relaxed ▼]   Noise  [Medium ▼]   Layout  [Open All ▼]   Per page  [25 ▼]
                                                                                            Update

                                                                            First   Prev   Next
```

❓ A limited miracle? new   🧑 tbop   2-Jul-11 16:17

Wow. I first have to say that your solution is amazing.

But I'm unfortunately here in order to report what could likely be a bug.

When building Enum subclass it seems that you cannot add more than 16 EnumItem, otherwise they are no longer generated. Here is the piece of code that I cannot make compile because of this. Is that intended? Explainable? Why?

```cpp
public:
                #define IMPROVED_ENUM_SUBCLASS
                #define IMPROVED_ENUM_NAME   IntAttributeEnum
                #define IMPROVED_ENUM_LIST   ENUMITEM(lochan)\ //1
                        ENUMITEM(hichan)                  \ //2
                        ENUMITEM(lokey)                   \ //3
                        ENUMITEM(hikey)                   \ //4
                        ENUMITEM(key)                     \ //5
                        ENUMITEM(lovel)                   \ //6
                        ENUMITEM(hivel)                   \ //7
                        ENUMITEM(lobend)                  \ //8
                        ENUMITEM(hibend)                  \ //9
                        ENUMITEM(lochanaft)               \ //10
                        ENUMITEM(hichanaft)               \ //11
                        ENUMITEM(lopolyaft)               \ //12
                        ENUMITEM(hipolyaft)               \ //13
                        ENUMITEM(seq_length)              \ //14
                        ENUMITEM(seq_position)            \ //15
```

```
                         ENUMITEM(sw_lokey)                    \ //16
                         ENUMIEM(sw_hikey)\ //After that one (included)
nothing will be registered
                         ENUMIEM(sw_last)                      \
                         ENUMITEM(sw_down)                     \
                         ENUMITEM(sw_up)                       \
                         ENUMITEM(sw_previous)                 \
                         ENUMITEM(group)                       \
                         ENUMITEM(off_by)                      \
                         ENUMITEM(on_loccN)                    \
                         ENUMITEM(on_hiccN)                    \

              #include "EnumToString.hpp"
```

Thank you once again

Reply · Email · View Thread · Permalink · Bookmark

📋 Re: A limited miracle? new   🧑 Hugo González Castro   2-Jul-11 20:47

Thank you very much for your comments 🙂 . There shouldn't be a limitation, the code is a precompiled generation of a class with all the items… the only explanation I can imagine is that maybe there is a limitation on the compiler you are using. I have tested that code under Windows XP, Vista and 7, in 32 and 64-bit versions, with MS Visual Studio 2005 (but should work also with higher versions). May I ask which compiler/precompiler are you using?

Another details you may check are:
- After the backslash of each precompiled line you cannot put any comments (eg: //16) because the compiler may fail.
- The last ENUMITEM shouldn't end with backslash.
Those things are related to the precompiler, but do not explain why it is failing for you after the item 16. You may activate the /EP /P compiler options (for MS compiler) to generate and see the preprocessed file with the real code that the macros generate.

Another problem that could appear is if any of your items has the same name of a reserver word or conflicts with another global variable/constant/define… in that case, you may check the first failing item. Regards,

Hugo González

Reply · Email · View Thread · Permalink · Bookmark

📋 Re: A limited miracle? new   🧑 Hugo González Castro   2-Dec-11 12:30

I have tested it with an enum of 256 items and it worked! 🙂

Reply · Email · View Thread · Permalink · Bookmark

📄 My vote of 5 new   🧑 rauvega   8-Apr-11 5:01

Very useful. Thank you! 🙂

Reply · Email · View Thread · Permalink · Bookmark

📄 Re: My vote of 5 new   🧑 Hugo González Castro   21-Apr-11 18:28

Thanks for your support!

Reply · Email · View Thread · Permalink · Bookmark

❓ Multiple inheritance new   🧑 pete221   12-Jun-09 4:31

Really useful code! ... but is it possible to have multiple layers of inheritance of enum definitions?

Reply · View Thread · Permalink · Bookmark

📝 Re: Multiple inheritance new   🧑 Hugo González Castro   17-Nov-09 9:02

Thank you. It is not possible to have multiple layers of inheritance with this version. I have go beyond and I'm about to publish a new version... still don't know when it will be finished...

Reply · Email · View Thread · Permalink · Bookmark

📄 Generalisation new   🧑 **Triantafillos Paradas**   **21-Apr-09 4:03**

Hello,

Is it possible to make this very useful piece of code to accept also wide strings?

Reply · Email · View Thread · Permalink · Bookmark   5.00/5 (1 vote)

📄 Re: Generalisation new   🧑 Hugo González Castro   21-Apr-09 12:39

Thank you very much for your improvements of the code you sent me 😊 .

It is very useful to support both, `string` and `wstring`. I will add the support of `wstring` in version 6.0, but I need the code to be multiplatform so I cannot include `"windows.h"` to use the `TEXT` macro. I will solve that with:

```
    //! Enable WideStrings when using UNICODE (Multiplatform)
    //////////////////////////////////////////////////////////
    #ifdef USE_UNICODE
    #define TXT_EXPANDED(Txt) L##Txt
    typedef std::wstring String;
    #else
    #define TXT_EXPANDED(Txt) Txt
    typedef std::string String;
    #endif
    #define TXT(Txt) TXT_EXPANDED(Txt)
```

and

```
    //! Some stuff to get the string of the IMPROVED_ENUM_NAME
    //////////////////////////////////////////////////////////
    #define GET_MACRO_STRING_EXPANDED(Macro)  #Macro
    #define GET_MACRO_STRING(Macro)
TXT(GET_MACRO_STRING_EXPANDED(Macro))
    #define ENUM_SEPARATOR  TXT("::")
```

Regards,

Hugo

modified on Wednesday, April 22, 2009 6:05 AM

Reply · Email · View Thread · Permalink · Bookmark

📄 OOP new   🧑 **TheSebaster**   **20-Apr-09 14:44**

Never heard about object oriented programming??

Reply · Email · View Thread · Permalink · Bookmark   1.00/5 (4 votes)

📄 Re: OOP new   🧑 Hugo González Castro   20-Apr-09 19:52

Never heard about constructive comments?? 😊

My code can generate a class with static methods to encapsulate an `enum` type with some improvements. That's OOP. I'm also working in a version with non-static methods to generate something better than a `.net enum class`, but in a multiplatform environment.

If you have any constructive ideas, I mean a better way to do those things, I'd like to know them. 👋

Reply · Email · View Thread · Permalink · Bookmark

📄 **Re: OOP** new　　　　　🗿 Harrison H　　　　　18-Mar-11 10:37

Don't listen to him. He's obviously a newb. Great article!

Reply · Email · View Thread · Permalink · Bookmark

📄 **Re: OOP** new　　　　　🗿 Hugo González Castro　　　　　18-Mar-11 20:36

Thank you 😊
I have evolved this a lot since I published this article, but my code now is property of my company, so I cannot publish it.
I have planned to remake this from scratch in a different and better way, but this time the code will be mine and I will publish it. It will take me a few months, but I will publish here the news 😊

Reply · Email · View Thread · Permalink · Bookmark

📄 **Re: OOP** new　　　　　🗿 Harrison H　　　　　18-Mar-11 10:37

You're a tool. This couldn't be solved by OOP because enums aren't a higher type in C++.

Reply · Email · View Thread · Permalink · Bookmark

📄 **Re: OOP** new　　　　　🗿 Hugo González Castro　　　　　18-Mar-11 20:47

My new work is about extending this basic type with a higher type in C++, and I got reflexion and generic enums. Hope I can finish this soon...

Reply · Email · View Thread · Permalink · Bookmark

📄 **Some remark** new　　　　　🗿 xvit　　　　　25-Dec-08 5:18

Hi!
you do not check, no there already declared ENUMITEM and ENUMITEM_VALUE

Reply · Email · View Thread · Permalink · Bookmark　　　　　2.00/5 (2 votes)

📄 **Re: Some remark** new　　　　　🗿 Hugo González Castro　　　　　26-Dec-08 21:53

Hi. That's the neverending problem with the MACRO definitions: there cannot be more that one definition at the same block of code. As I call `#undef` after using `ENUMITEM()` and after using `ENUMITEM_VALUE()`, I do not leave my macros defined below my block of code.
But if you leave a macro -with the same name of mine- defined after your block of code... that cannot work even if I check the existence of your macros.
Anyway, if you want to make that check, just add (at the beginning of `DefineImprovedEnum.h`"):

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

Regards,

Hugo

Reply · Email · View Thread · Permalink · Bookmark

📧 **Re: Some remark** new   🖼 Hugo González Castro   6-Apr-09 2:40

I have added that check to the last update (version 4.0) of the code.

Reply · Email · View Thread · Permalink · Bookmark   1.00/5 (1 vote)

Last Visit: 31-Dec-99 23:00   Last Update: 20-Mar-14 15:23   Refresh   **1**

📄 General   📧 News   💡 Suggestion   ❓ Question   🐛 Bug   ✅ Answer   😄 Joke   😠 Rant   ⓘ Admin