15,784,223 members

Sign in

articles quick answers discussions features community help

Search for articles, questions, tips

Articles / Programming Languages / C

C++ C

X Macros in C

FredBienvenu

9 Aug 2016 CPOL 6 min read 26.3K 24 13

Rate me: ★★★★★ 4.92/5 (27 votes)

How to enhance C/C++ language with macro to auto generate some code

Introduction

In this article, I want to explain what are **X macros** and how they work.

I know that this is **not a new idea**, but I have not found a lot of documentation over the web. You can find below a list of articles talking or using **X Macros**:

- http://www.codeproject.com/Articles/25541/C-C-macros-programming
- http://stackoverflow.com/questions/147267/easy-way-to-use-variables-of-enum-types-as-string-in-c
- http://www.drdobbs.com/the-new-c-x-macros/184401387

The idea here is to describe this technique **as simply as possible**. I will not tell you how to use them, but simply describe them with simple examples and I am sure you will find utilities for your projects.

I have discovered **X Macros** a few years ago and I found them extremely powerful and useful.

Description

Basics Behind

The concept behind **X Macros** is based on the possibility to pass macro **as parameter** to another macro.

Let's look at the below macro declaration:

C++

```
#define APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAM_MACRO, parameter) PARAM_MACRO(parameter)
```

As it is declared, the macro `APPLY_A_MACRO_ON_SECOND_PARAMETER` will expand as:

C++

```
PARAM_MACRO(parameter)
```

Nothing complicated here. But now, you can ask **what is PARAM_MACRO?**

A First Example

At this time, `PARAM_MACRO` is **nothing** because it is **not declared yet**. And that is what makes this technique very powerful.

Now, we can declare **different kind of macros** that apply on only one parameter and pass them to `APPLY_A_MACRO_ON_SECOND_PARAMETER`.

C++

```
// Let's declare two different macros
// first one will declare one integer with parameterized name
#define PARAMETER_TO_INT_DECL(parameter) int parameter;

// second one will create a getter function that will return the variable
#define PARAMETER_TO_GETTER_FUNC(parameter) int get_##parameter() { return parameter; }
```

Now, we can **call** `APPLY_A_MACRO_ON_SECOND_PARAMETER` with those two macros.

Starting with `PARAMETER_TO_INT_DECL`:

C++

```
APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAMETER_TO_INT_DECL, width)
```

1. `APPLY_A_MACRO_ON_SECOND_PARAMETER` expands like this:

   C++

   ```
   PARAM_MACRO(parameter)
   ```

2. Let's now replace with the **parameters values**:

   C++

   ```
   PARAMETER_TO_INT_DECL(width)
   ```

3. `PARAMETER_TO_INT_DECL` expands like this:

   C++

   ```
   int parameter;
   ```

4. Let's now replace with the **parameter value** for the last expansion:

   C++

   ```
   int width;
   ```

Let's look at the expansion with `PARAMETER_TO_GETTER_FUNC` now:

C++

```
APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAMETER_TO_GETTER_FUNC, width)
```

1. `APPLY_A_MACRO_ON_SECOND_PARAMETER` expands like this:

   C++

   ```
   PARAM_MACRO(parameter)
   ```

2. Let's now replace with the **parameters values**:

   C++

   ```
   PARAMETER_TO_GETTER_FUNC(width)
   ```

3. `PARAMETER_TO_GETTER_FUNC` expands like this:

   C++

   ```
   int get_##parameter() { return parameter; }
   ```

4. Let's now replace with the **parameter value** for the last expansion:

   C++

   ```
   int get_width() { return width; }
   ```

To sum up:

C++

```
APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAMETER_TO_INT_DECL, width)
APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAMETER_TO_GETTER_FUNC, width)

// will turn into
int width;
int get_width() { return width; }
```

This example may not impress you much. I stayed basic but here is what we can say: we have created a way to **automatically standardize** the getters function.

It remains a bit complicated to use, so we can declare a new macro like this one:

C++

```
#define INT_VAR_PLUS_GETTER(variable_name)\
    APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAMETER_TO_INT_DECL, variable_name)\
    APPLY_A_MACRO_ON_SECOND_PARAMETER(PARAMETER_TO_GETTER_FUNC, variable_name)
```

Now, let's call `INT_VAR_PLUS_GETTER`:

C++

```
INT_VAR_PLUS_GETTER(width)

// will turn into
int width;
int get_width() { return width; }
```

Now, we have a macro library that permits in one call to **declare both variable and getter**. Everything is standardized.

At this point, I am sure most of you **are still not impressed**.

What is the point of all this complexity to such a simple (and maybe unuseful) result?

Let's move to a **more complex and useful example**.

Useful Example: enum to string

Motivations

As I said in the introduction, I found **X macros** technique a few years ago. At that time, I was working on maintaining the code of a big C project.

On this project, there were a **lot of enums** used and I wanted to add logs in order to trace what were the `enum` values when some bugs happened.

In the first place, I was logging like that:

C++

```
printf("enum value : %d\n", enum_value);
```

But the problem with this is when the **enum type** of `enum_value` has **more than a hundred entries**.

Let's say then that you have this log to interpret : "`enum value : 57`".

Then you have to find the **enum declaration**, and count the entries till you reach the 57th entry. This is painful. Furthermore, 57 is not an informative enough data regarding the `enum` type. We just want to have an `enum` that allows to **transform 57 into a readable string**.

Let's Do It Without Macros

First, we will implement such a function **without macro help** in order to see how **X macros** will help us in automatizing work and avoiding errors.

Let's start with a **simple enum declaration**:

C++

```
typedef enum IceCreamFlavors
{
    CHOCOLATE = 56,
    VANILLA = 27,
    PISTACHIO = 72,
}
IceCreamFlavors;
```

Now, let's define the function that will **turn the enum values into strings**:

C++

```
const char* IceCreamFlavors_toString(IceCreamFlavors flavor)
{
    switch(flavor)
    {
    case CHOCOLATE:
        return "CHOCOLATE";
    case VANILLA:
        return "VANILLA";
    case PISTACHIO:
        return "PISTACHIO";
    default:
        // the error handling might seem a bit too strict !
        return 0;
        // you can also return something like:
        return "## unknown IceCreamFlavors value ##";
    }
}
```

Creating and maintaining this `_toString` function is **a bit repetitive**. You can easily make **copy/paste errors** or **forget one entry**. Furthermore, when you **update the enum**, you **have to ensure** that the `_toString` function is **properly updated**. This is **potentially a source of errors**.

Now, Let's See How X Macros Will Help Us

What we want is a **single location** in the code where to store the `enum` values. And we want that the *_toString* function **exists** and **is updated automatically** according to `enum` values.

Storing the Enum Values

First, we create a macro that stores the `enum` values:

C++

```
#define SMART_ENUM_IceCreamFlavors(_)\
    _(CHOCOLATE, 56)\
    _(VANILLA, 27)\
    _(PISTACHIO, 72)
```

Remember the description chapter. What we have done here is to declare a macro `SMART_ENUM_IceCreamFlavors` that **takes another macro** (`_`) as parameters. This will allow us to **do what we want** with the parameters given to `_`.

Enum Declaration

First, we want to **declare the enum** the C way.

We first need a macro that will turn the `SMART_ENUM_IceCreamFlavors` entries (lines under) into **C enum entries**.

C++

```
#define SMART_ENUM_ENTRY(entry_name, entry_value) entry_name = entry_value,
```

Then, we need a macro to **build the entire C enum**:

C++

```
// the macro takes the enum macro definition as parameter
// (in our case we will pass SMART_ENUM_IceCreamFlavors)
#define SMART_ENUM_DECLARE_ENUM(MACRO_DEFINITION, enum_name)\
typedef enum enum_name\
{\
    MACRO_DEFINITION(SMART_ENUM_ENTRY)\
}\
enum_name;
```

Now, let's call this macro and see the expansion:

C++

```
// the first parameter is the macro definition of the enum,
// the second one is the name we want to give to the enum
SMART_ENUM_DECLARE_ENUM(SMART_ENUM_IceCreamFlavors, IceCreamFlavors)
```

1. `SMART_ENUM_DECLARE_ENUM` expands like this:

   C++

   ```
   typedef enum enum_name\
   {\
      MACRO_DEFINITION(SMART_ENUM_ENTRY)\
   }\
   enum_name;
   ```

2. Replacing with the macro parameters:

   C++

```
typedef enum IceCreamFlavors
{
    SMART_ENUM_IceCreamFlavors(SMART_ENUM_ENTRY)
}
IceCreamFlavors;
```

3. Expanding `SMART_ENUM_IceCreamFlavors`:

   C++

   ```
   typedef enum IceCreamFlavors
   {
       _(CHOCOLATE, 56)\
       _(VANILLA, 27)\
       _(PISTACHIO, 72)
   }
   IceCreamFlavors;
   ```

4. and replacing with macro in parameter:

   C++

   ```
   typedef enum IceCreamFlavors
   {
       SMART_ENUM_ENTRY(CHOCOLATE, 56)
       SMART_ENUM_ENTRY(VANILLA, 27)
       SMART_ENUM_ENTRY(PISTACHIO, 72)
   }
   IceCreamFlavors;
   ```

5. `SMART_ENUM_ENTRY` expands like:

   C++

   ```
   entry_name = entry_value,
   ```

6. So the final result is:

   C++

   ```
   typedef enum IceCreamFlavors
   {
       CHOCOLATE = 56,
       VANILLA = 27,
       PISTACHIO = 72,
   }
   IceCreamFlavors;
   ```

And **here we are**!

_toString Function Creation

Now, we want to create such a similar macro in order to **create the _toString function** with the **same macro definition** of the `enum`. Thus, we will have **only one location** where the `enum` values are stored !

Let's start with the macro that will **turn each entry** of the macro definition **into a case statement** for the function:

C++

```
// the macro takes two parameters as the macro definition use macro that takes two
// so entry value is not use, but it is not a big deal
#define SMART_ENUM_TOSTRING_CASE(entry_name, entry_value) case entry_name: return #entry_name;
```

Now, let's write the macro that will **build the entire _toString function**:

C++

```
// the macro takes the enum macro definition as parameter
// (in our case we will pass SMART_ENUM_IceCreamFlavors)
#define SMART_ENUM_DEFINE_TOSTRING_FUNCTION(MACRO_DEFINITION, enum_name)\
const char* enum_name##_toString(enum_name enum_value)\
{\
    switch(enum_value)\
    {\
    MACRO_DEFINITION(SMART_ENUM_TOSTRING_CASE)\
    default:\
        // the error handling might seem a bit too strict !\
        return 0;\
        // you can also return something like:\
        return "## unknown enum_name value ##";\
    }\
}
```

Now, let's call this macro and see the expansion:

C++

```
// the first parameter is the macro definition of the enum,
// the second one is the name we want to give to the enum
SMART_ENUM_DEFINE_TOSTRING_FUNCTION(SMART_ENUM_IceCreamFlavors, IceCreamFlavors)
```

1. `SMART_ENUM_DEFINE_TOSTRING_FUNCTION` expands like this:

   C++

   ```
   const char* enum_name##_toString(enum_name enum_value)\
   {\
       switch(enum_value)\
       {\
       MACRO_DEFINITION(SMART_ENUM_TOSTRING_CASE)\
       default:\
           // the error handling might seem a bit too strict !\
           return 0;\
           // you can also return something like:\
           return "## unknown enum_name value ##";\
       }\
   }
   ```

2. Replacing with the macro parameters:

   C++

   ```
   const char* IceCreamFlavors_toString(IceCreamFlavors enum_value)
   {
       switch(enum_value)
       {
       SMART_ENUM_IceCreamFlavors(SMART_ENUM_TOSTRING_CASE)
       default:
           // the error handling might seem a bit too strict !
           return 0;
           // you can also return something like:
           return "## unknown IceCreamFlavors value ##";
       }
   }
   ```

3. Expanding `SMART_ENUM_IceCreamFlavors`:

   C++

   ```
   const char* IceCreamFlavors_toString(IceCreamFlavors enum_value)
   {
       switch(enum_value)
       {
       _(CHOCOLATE, 56)\
       _(VANILLA, 27)\
       _(PISTACHIO, 72)
       default:
           // the error handling might seem a bit too strict !
           return 0;
           // you can also return something like:
           return "## unknown IceCreamFlavors value ##";
       }
   }
   ```

4. and replacing with macro in parameter:

   C++

   ```
   const char* IceCreamFlavors_toString(IceCreamFlavors enum_value)
   {
       switch(enum_value)
       {
       SMART_ENUM_TOSTRING_CASE(CHOCOLATE, 56)
       SMART_ENUM_TOSTRING_CASE(VANILLA, 27)
       SMART_ENUM_TOSTRING_CASE(PISTACHIO, 72)
       default:
           // the error handling might seem a bit too strict !
           return 0;
           // you can also return something like:
           return "## unknown IceCreamFlavors value ##";
       }
   }
   ```

5. `SMART_ENUM_TOSTRING_CASE` expands like:

   C++

   ```
   case entry_name: return #entry_name;
   ```

6. So finally, by expanding it, we got:

C++

```
const char* IceCreamFlavors_toString(IceCreamFlavors enum_value)
{
    switch(enum_value)
    {
    case CHOCOLATE: return "CHOCOLATE";
    case VANILLA: return "VANILLA";
    case PISTACHIO: return "PISTACHIO";
    default:
        // the error handling might seem a bit too strict !
        return 0;
        // you can also return something like:
        return "## unknown IceCreamFlavors value ##";
   }
}
```

And here we are again! We have **built the _toString function** with the **same macro definition** that served to create the **enum declaration**.

Final Macro

Now we have all the tools, let's create the last macro:

C++

```
#define DEFINE_SMART_ENUM(MACRO_DECLARATION, enum_name)\
    SMART_ENUM_DECLARE_ENUM(MACRO_DECLARATION, enum_name)\
    SMART_ENUM_DEFINE_TOSTRING_FUNCTION(MACRO_DECLARATION, enum_name)
```

Now, if we call the macro:

C++ Shrink ▲

```
// I rewrite the macro declaration here to remember
#define SMART_ENUM_IceCreamFlavors(_)\
    _(CHOCOLATE, 56)\
    _(VANILLA, 27)\
    _(PISTACHIO, 72)

// we call the builder
DEFINE_SMART_ENUM(SMART_ENUM_IceCreamFlavors, IceCreamFlavors)

// the result will be...

typedef enum IceCreamFlavors
{
    CHOCOLATE = 56,
    VANILLA = 27,
    PISTACHIO = 72,
}
IceCreamFlavors;

const char* IceCreamFlavors_toString(IceCreamFlavors enum_value)
{
    switch(enum_value)
    {
    case CHOCOLATE: return "CHOCOLATE";
    case VANILLA: return "VANILLA";
    case PISTACHIO: return "PISTACHIO";
    default:
        // the error handling might seem a bit too strict !
        return 0;
        // you can also return something like:
        return "## unknown IceCreamFlavors value ##";
    }
}
```

We now have a beautiful and powerful one line builder to:

1. **ensure a deep linkage** between the `enum` declaration and the `_toString` function
2. reduce the coding time
3. avoid stupid errors

Of course, **each update on the macro definition** will have an impact on both **enum declaration** and **_toString function**.

Conclusion

I hope this article is **clear enough** and **not too boring**. I tried to make it **as clear as possible**, so that explains its size. The macro expansions can be hard to follow. It happens frequently that I have to mentally re expand macros in order to understand what they do.

This said, I truly think that X macros is a **wonderful tool**.

I have read a lot of criticism against that technique because it uses macros that are considered **unsafe**.

I totally agree that using macros is touchy and I don't recommend to use them for everything.

But I see that as a **language extension**, that can't be achieved in another way.

If you look at the "`Enum` to `string`" example, the built code is always **pretty simple**. Macros are not hiding some **critical code** that we will need to be debugged. It should be used to automate **well known simple mechanisms** that **have been tested before** macro-izing them.

License

Written By

FredBienvenu

France

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.

Comments and Discussions

You must Sign In to use this message board.

Search Comments

Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

A minor nit about using "_" as a macro name	**sailnfool**	**8-Sep-20 9:36**
Enum to String	**A_Griffin**	**26-Aug-17 1:13**
Re: Enum to String	FredBienvenu	28-Aug-17 6:42
Re: Enum to String	A_Griffin	28-Aug-17 6:44
like it and will try it	**Midnight489**	**10-Aug-16 9:34**
Re: like it and will try it	FredBienvenu	10-Aug-16 11:32
Re: like it and will try it	FredBienvenu	10-Aug-16 11:32
Nice!	**Ben Ratzlaff**	**10-Aug-16 6:23**
Re: Nice!	FredBienvenu	10-Aug-16 11:24
:)	**Edward671**	**9-Aug-16 5:04**
Re: :)	FredBienvenu	9-Aug-16 5:34

Last Visit: 31-Dec-99 19:00 Last Update: 20-Nov-23 23:47 Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Posted 9 Aug 2016