Switching on Boolean Conditions and Flags

Michael E. Gibson, 3 Jan 2011 BSD

★★★★★ 4.67 (33 votes)

Rate this: vote 1vote 2vote 3vote 4vote 5

Presents a technique and code for writing switch-like syntax for dealing with multiple conditions or flags.

- **Download source - 1.67 KB**

Introduction

If you've ever written or looked at code with several levels of nested `if` statements, you know how difficult it can be to maintain. In this article, I will present code that eliminates the need for the deep nesting in some cases. The nested `if` statements are changed to look more like the familiar, and much more maintainable `switch` statement.

Background

When writing code whose behavior depends on a number of conditions or flags, the normal method for determining that behavior is to check those conditions as efficiently as possible so that the code doesn't end up becoming too complex. When there are several conditions to be checked, the nesting of `if` statements can become very deep and hard to maintain.

In some cases, a balance has to be struck between complexity of the nesting and the code that executes when conditions are matched. Code can be interspersed within the nesting itself, but when complex nesting is used, the true flow of execution can become difficult to follow. Duplicating code can make flow easier to follow, but at the expense of the maintenance issues associated with code duplication. In either case, the nesting itself with the associated brackets doesn't help the code to be concise.

Hide Copy Code

```
// An example of complex nested if statements
if (foo > 1)
{
  if (bar < 12)
  {
    if (foo < bar+10)
    {
      // rare case code here looks more important than it is
    }
    else
    {
      // common case code
    }
  }
  else if (eof())
  {
    // same rare case code copied from above
  }
}
```

One technique to simplify this code is to store the result of the conditions in boolean values above the `if`statement nesting. The boolean values are then the ones used in the `if` statement conditions. In this case, conditions are only evaluated once, which may be an important consideration. While this simplifies the conditions, it does nothing to address the nesting problem. Another technique is to carefully eliminate the use of curly brackets when not required. However, this can cause the logic of the code to be unclear when `else` clauses are present. This technique is generally not recommended and often specifically forbidden in coding standards.

One solution to a particular version of this problem is the `switch` statement. It simplifies the structure of repeated`else if` blocks. For example, this:

Hide   Copy Code

```
if (a == 1)
  ;
else if (a == 2)
  ;
else if (a == 3)
  ;
else
  ;
```

Can be turned into this:

Hide   Copy Code

```
switch (a)
{
case 1:
  break;
case 2:
  break;
case 3:
  break;
default:
  break;
}
```

Now, a is only evaluated once and the cases are broken down nicely without a lot of extra syntactic noise. The problem with switch statements is that they don't address cases where multiple conditions need to be tested. They also only work with a particular type of condition: equality.

Switch Flags Library

In an effort to solve some of these issues, I have written a library in the form of a single header file (switch_flags.h) that allows switch statements to be more flexible. The Switch Flags library allows switchstatements to use multiple conditions. It consists of two sets of macros, switch_flags_x and flags_x, where xis a integer from 1 to 8. The switch_flags_x macros begin the switch flags block and take the conditions as parameters, in a similar way that switch statements themselves are used. Then, the flags_x macros are used in conjunction with case labels to express the truth values. They take as parameters the tokens T, F, or X meaningtrue, false, or either respectively. Each parameter in the flags_x macros correspond to the matching condition in the controlling switch_flags_x macro.

A Simple Example

This simple example demonstrates the basic usage:

Hide Copy Code

```
#include <switch_flags.h>

switch_flags_2(a > b, c != d)
{
case flags_2(T,T):
  // only execute when "a > b" and "c != d"
  break;
case flags_2(F,X):
  // execute when "a > b" is false
  break;
}
```

Let's go through this line by line.

Hide Copy Code

```
#include <switch_flags.h>
```

Include the header file that makes up the library. Since the library is header only, no link changes are required. Simply copy *switch_flags.h* in your project and `#include` it where needed.

Hide Copy Code

```
switch_flags_2(a > b, c != d)
```

Start the `switch` flags block and present the conditions. Due to the limitations of the C preprocessor, there is a different version of the macro for different numbers of conditions, each one with a suffix which is the number of parameters. Removing this requirement is a possible improvement to the library (see section on How It Works below).

The parameter positions of each condition are important. They will need to match the parameter positions of later`flags_x` macros. The conditions will be evaluated here and only once. Later `flags_x` macros will use the result evaluated here to test against.

Hide Copy Code

```
{
```

Open bracket to start the `switch` block in the same way that a normal `switch` block begins.

Hide Copy Code

```
case flags_2(T,T):
  // only execute when "a > b" and "c != d"
  break;
```

The first `case` label and associated block. The `flags_2` macro presents a particular case to check. Here, it checks for both conditions to be true using the `T` token for both parameters.

Hide Copy Code

```
case flags_2(F,X):
  // execute when "a > b" is false
  break;
```

The second `case` statement and associated block. The `flags_2` macro again presents the case to check. However, it now checks for the first condition `a > b` to be `false` using the F token and doesn't check the second condition `c != d` at all by using the `X` token.

Hide Copy Code

```
}
```

Since there are no more possible cases, we close out the `switch_flags_2` block.

A More Complex Example

More complex usage is possible, including up to eight total conditions. For example, the following code replicates the example presented above (note the change in the number of parameters and the corresponding change in the macro suffix):

Hide Copy Code

```
#include <switch_flags.h>

switch_flags_4(foo > 1, bar < 12, foo < bar+10, eof())
{
case flags_4(T,T,F,X):
  // common code
  break;
case flags_4(T,T,T,X):
case flags_4(T,F,X,T):
  // rare case code
  break;
}
```

The `default` keyword can also be used to cover any cases not otherwise specified just as in a `switch` statement.

Caveats

There are of course some caveats to the use of this library. When using nested `if` statements, you can control the evaluation of the conditions themselves so that they are only evaluated in certain conditions. This may be important when the evaluation of a condition itself may cause a performance problem or error condition. This library however always evaluates all of its conditions every time. There's no conditional evaluation. Of course, conditional evaluation is supported within single conditions, so conditions like ptr && ptr->foo will work fine.

Also, because of how the library is implemented using case labels, it shares the requirement that cases not overlap. That is, there cannot be more than one case label that satisfies a particular set of truth values for the given conditions. With normal case labels, this isn't really a problem as overlapping cases are obvious. With this library, the X token can cause overlaps may not be entirely obvious. For example:

Hide Copy Code

```
case flags_3(T,X,T): // this case ...
case flags_3(T,T,X): // overlaps this case since they both cover the (T,T,T) case
```

A more liberal compiler could solve this problem by allowing overlapping cases. In fact, it seems that this is an unfortunate and unneeded restriction in C/C++.

How It Works

The library is made up entirely of macros. The switch_flags_x macros simply take the conditions and compute an integer value where each bit corresponds to a condition. If you're not familiar, this is a very common technique for storing multiple flags in a single variable. Each bit in the integer is either on or off depending on its corresponding condition either true or false. In C/C++, switch statements are allowed to evaluate their expressions. This is used in this library to evaluate the conditions at runtime and generate an integer value that represents the combined state of all the conditions.

Case statements are different however in that they do not allow their operands to be evaluated at runtime. They must be constants evaluated at compile time. However, they can include multiple constants in the same block by simply adding another case. This is how the library handles the X token, which doubles the number of matching integer values. Each time an X token appears in the flags_x macro, the macro must "split" and determine both sets of values that the swtich_flags_x macro creates. A new case label is then generated.

Here is what the simple (two parameter) case expands to after preprocessing:

Hide Copy Code

```
switch (((a > b) ? 1 : 0) | (((c != d) ? 1 : 0) << 1))
{
case ((((0<<1)|1)<<1)|1):
  break;
case ((((0<<1)|0)<<1)|0):
case ((((0<<1)|1)<<1)|0):
  break;
}
```

With a bit of simplification, the above reduces down to this:

Hide Copy Code

```
switch ((a > b ? 1 : 0) | ((c != d ? 1 : 0) << 1))
{
case 3:
  break;
case 0:
case 1:
  break;
}
```

Performance

Performance of this library was a key concern. I did not want it to take any more time or memory than more traditional methods. The implementation scheme accomplishes this well by taking advantage of the compiler to optimize away constants when possible. Since it is made up completely of macros, it won't add any size to binaries. The only performance trade-off to be considered is that the library does not support conditional evaluation (see Caveats above).

Suffix Problem

The library requires that the user specify the number of arguments redundantly by using a suffix on the name of the macro itself. A possible solution to this problem is to use variadic macros as specified in C99. However, this would limit the audience of the library to those with conforming preprocessors. The library as it stands now is usable with almost any decent C preprocessor, which makes the library applicable to very wide audience. The addition of variadic forms of the macros in the library would be a possible improvement to the library.

Wrap-up

I created this library to solve a particularly ugly piece of code I was writing and hope that it can be used by others to spruce up their own code. Please let me know if you use this code and any improvements that you feel would be helpful. Some possible improvements would be variadic versions of the macros, detection and automatic elimination of overlapping cases and a higher limit on the number of arguments.

History

- 3rd January, 2011: Initial post

License

About the Author

Michael E. Gibson

Software Developer (Senior) StorageCraft
United States