Posted on **April 1, 2015**

Pattern matching can be a very powerful tool when it comes to preprocessor metaprogramming. It lets us do anything from boolean logic to addition/subtractions. These macros can be combined to help write macro loops which can help unroll loops!

Pattern matching on the preprocessor arises from using token concatenation. The general form of this is we have our macro call concatenated version of our macro were our parameters are the values we want to match. So to begin, lets look implementing AND via truth table

```
#define CAT(x, y) x ## y
#define PRIMITIVE_CAT(x, y) CAT(x, y)
/*
We will basically pattern match the truth table
AND(0, 0) = 0 -> AND_00 = 0
AND(0, 1) = 0 -> AND_01 = 0
AND(1, 0) = 0 -> AND_10 = 0
AND(1, 1) = 1 -> AND_11 = 1

In general macro format this would be
#define AND_xy [value of and(x, y)]
*/
#define AND_00 0
#define AND_10 0
#define AND_01 0
#define AND_11 1
#define AND(x, y) CAT(CAT(AND_, x), y)
// The resulting calls expand to correct values
AND(0, 0) // 0
AND(0, 1) // 0
AND(1, 0) // 0
AND(1, 1) // 1
```

We're not limited to returns values only. We can also have it return functions. Lets see how we can use this to implement if statements.

If statements are an example of how to use pattern matching to return functions. Let 1 and 0 represent true and false. Our IF macro will then simply return the appropriate IF function.

```
/*
 Lets also look at the general if form.
 IF(condition) IF_CONDITION return IF_TRUE else return IF_FALSE
 Lets say we represent 1 and true and 0 as false
 IF(condition) IF_CONDITION return IF_1 else return IF_0

 We want our IF macro to basically return the appropriately
 matched IF_ macro.
*/
// Note we used variadic parameters for false so that its optional
#define IF_1(true, ...) true
#define IF_0(true, ...) __VA_ARGS__
#define IF(value) CAT(IF_, value)

IF(1) ( a, b ) // Expands to IF_1(a, b) -> a
IF(0) ( a, b ) // Expands to 1F_0(a, b) -> b

IF(1) ( a ) // Expands to a
IF(0) ( a ) // Expands to nothing
```

Theres 2 problems with the current pattern matching scheme right now. The first what happens when we don't have a valid pattern match? Because we're concatenating the parameters and calling the appropriate macro we will simply have an unresolved macro.

```
AND (1, 2) // Expands AND_12
```

The second problem, is really visible in the AND implementation. We need to represent all possible outcomes to be pattern matched. We know AND(x, y) is only 1 if x and y = 1 and so the other 3 pattern matches are redundant and useless.

In order to solve these problem, we need to find a way to check if a pattern match exists. If we have a pattern match, we can return its value and if it doesn't then we return some default value. (Note, an alternative approach to following section is the probe/check idiom that Paul talks about).

To achieve this, first we need our pattern matches to return their values enclosed in some name that we can pattern match. For this purpose, lets simply pack all of our return values in an EXISTS container. A simple code example should clarify things [Note, this is to show how exists should be used]

```
#define AND_00 EXISTS(0)
#define AND_10 EXISTS(0)
#define AND_01 EXISTS(0)
#define AND_11 EXISTS(1)
#define AND(x, y) CAT(CAT(AND_, x), y)

AND(1, 0) // Expands to EXISTS(0)
AND(1, 2) // Expands to AND_12
```

Now lets solve the first problem. We know that if we have a pattern match then we return an EXISTS object. The problem is if we don't have a match, it returns an unevaluated value.

As a result, we need to make a macro that tests if it was passed an EXISTS object. If it was, then it returns the EXIST value. If it isn't passed one, then return DOESNT_EXIST. Since we can only match the EXISTS object, by default we return DOESNT_EXIST. If we do get passed an EXISTS, we update our result and return the EXISTS value.

To achieve this lets have some fun with macros (Okay, this is most probably abuse but who doesnt like to hack stuff around! :)). The trick to updating parameters is that we need to splice out the old parameters and add in new parameters BEFORE the macro call is evaluated. Lets look at how we can do this:

```
// To splice out old parameters we need an EAT macro
// This macro simply "eats" all of its parameters
#define EAT(...)

/*
 The next is we need some sort of update rule
 The general form of this is:
    NEW_PARAMS ) EAT (
*/
#define REPLACE_B NEW_B) EAT (
(a, REPLACE_B, OLD_B) // This expands to (a, NEW_B)
/*
 The expansion chain goes as follows
 (a, REPLACE_B, OLD_B)
 -> (a, REPLACE_B, OLD_B)
 -> (a, NEW_B) EAT (OLD_B)
 -> (a, NEW_B)
*/
```

The one thing to remember is that these expansions need to happen BEFORE they are used as parameters in a macro. Lets see how this applies

```
#define FOO(x, y) x + y

// This is an error because we didnt update
// the parameters before calling FOO
FOO(A, REPLACE_B, OLD_B)

// If we use the DEFER/EVAL idiom, we can evaluate it properly
EVAL(DEFER(FOO) (A, REPLACE_B, OLD_B)) // Becomes A + NEW_B
```

Now using this idiom, we can implement our EXIST testing macro

```
/*
 We want a LOOKUP_PATTERN(x) to return
 EXISTS(returnValues) if x returns an EXISTS
 DOESNT_EXIST if it doesnt

 We will store our result in the form
 (Value to Test, EXIST result)

 By default, this will be
 (Value to Test, DOESNT_EXIST)

 If we encounter an EXISTS, we want to update parameters to
 (EXPANDED, EXIST Value Tested)
*/

#define EAT(...)
#define EXPAND_TEST_EXISTS(...) EXPANDED, EXISTS(__VA_ARGS__) ) EAT (
#define GET_TEST_EXISTS_RESULT(x) ( CAT(EXPAND_TEST_, x),  DOESNT_EXIST )

GET_TEST_EXISTS_RESULT( EXISTS(FOO) ) // ( EXPANDED, EXISTS(foo) )
GET_TEST_EXISTS_RESULT( FOO )         // ( TEST_foo, DOESNT_EXIST )

/*
 Once we have our test macros, the next step is to extract the values out.
 Remember though, we need to make sure the evaluation of
 GET_TEST_EXISTS_RESULT to happen before we extract the value.

 In the following code, GET_TEST_EXIST_VALUE calls
 GET_TEST_EXIST_VALUE_ which allows for one deferal and proper execution
*/
#define GET_TEST_EXIST_VALUE_(expansion, existValue) existValue
#define GET_TEST_EXIST_VALUE(x) GET_TEST_EXIST_VALUE_  x

#define TEST_EXISTS(x) GET_TEST_EXIST_VALUE (  GET_TEST_EXISTS_RESULT(x) )
// An alternative way to write #define TEST_EXISTS(x) is as
// EVAL(DEFER(GET_TEST_EXIST_VALUE_) (  GET_TEST_EXISTS_RESULT(x) ))

TEST_EXISTS( EXISTS(FOO) ) // Expands to EXISTS(FOO)
TEST_EXISTS( FOO )         // Expands to DOESNT_EXIST
```

We can now implement an an else case by testing if we get an EXISTS value. If we don't we can return a default value.

```
/*
  We can use TEST_EXISTS to get an appropriate EXISTS or
  DOESNT_EXIST value. By pattern matching we can return the EXISTS value
  if it exists and if not we can return some default value
*/
#define DOES_VALUE_EXIST_EXISTS(...) 1
#define DOES_VALUE_EXIST_DOESNT_EXIST 0
#define DOES_VALUE_EXIST(x) CAT(DOES_VALUE_EXIST_, x)

DOES_VALUE_EXIST(EXISTS()) // 1
DOES_VALUE_EXIST(DOESNT_EXIST) // 0

#define EXTRACT_VALUE_EXISTS(...) __VA_ARGS__
#define EXTRACT_VALUE(value) CAT(EXTRACT_VALUE_, value)

#define TRY_EXTRACT_EXISTS(value, ...) \
  IF ( DOES_VALUE_EXIST(TEST_EXISTS(value)) )\
  ( EXTRACT_VALUE(value), __VA_ARGS__ )

TRY_EXTRACT_EXISTS( EXISTS(FOO), DEFAULT ) // FOO
TRY_EXTRACT_EXISTS( FOO, DEFAULT )         // DEFAULT
```

So after all of that work, lets look at how it actually helps us make our AND cleaner.

```
// We only need to match one case now
// We can have all cases return 0!
#define AND_11 EXISTS(1)
#define AND(x, y) TRY_EXTRACT_EXISTS ( CAT(CAT(AND_, x), y), 0 )

AND(0, 1) // 0
AND(1, 1) // 1
AND(-1, 1) // Invalid
```

Note how we no longer need to specify the values anymore. We can extend this to add more features. Why we want to do these will become apparent in the next sections.

```
#define AND_11 EXISTS(1)
#define AND(x, y) TRY_EXTRACT_EXISTS ( CAT(CAT(AND_, x), y), 0 )

AND(0, 0) // 0
AND(0, 1) // 0
AND(1, 0) // 0
AND(1, 1) // 1

#define OR_00 EXISTS(0)
#define OR(x, y) TRY_EXTRACT_EXISTS ( CAT(CAT(OR_, x), y), 1 )

OR(0, 0) // 0
OR(0, 1) // 1
OR(1, 0) // 1
OR(1, 1) // 1

#define XOR_01 EXISTS(1)
#define XOR_10 EXISTS(1)
#define XOR(x, y) TRY_EXTRACT_EXISTS ( CAT(CAT(XOR_, x), y), 0 )

XOR(0, 0) // 0
XOR(0, 1) // 1
XOR(1, 0) // 1
XOR(1, 1) // 0

#define NOT_0 EXISTS(1)
#define NOT(x) TRY_EXTRACT_EXISTS ( CAT(NOT_, x), 0 )

NOT(0) // 1
NOT(1) // 0

#define IS_ZERO_0 EXISTS(1)
#define IS_ZERO(x) TRY_EXTRACT_EXISTS ( CAT(IS_ZERO_, x), 0 )

IS_ZERO(0)   // 1
IS_ZERO(1)   // 0
IS_ZERO(10)  // 0
IS_ZERO(010) // 0

// We can even chain logic now too!
#define IS_NOT_ZERO(x) NOT ( IS_ZERO(x) )

IS_NOT_ZERO(0)   // 1
IS_NOT_ZERO(1)   // 0
IS_NOT_ZERO(10)  // 0
IS_NOT_ZERO(010) // 0

/*
 This is useful for when we use tuples
 Our goal we have (FOO, Bar) vs FOO, Bar

 The way this works is simple, IS_ENCLOSED_TEST x
 If x is enclosed, we call IS_ENCLOSED_TEST(...)
 If it isnt, the macro remains unevaluated and we default to 0
*/
#define IS_ENCLOSED_TEST(...) EXISTS(1)
#define IS_ENCLOSED(x, ...) TRY_EXTRACT_EXISTS ( IS_ENCLOSED_TEST x, 0 )

IS_ENCLOSED(Foo, Bar)   // 0
IS_ENCLOSED((Foo, Bar)) // 1
```

One issue with TRY_EXTRACT_EXISTS is that if we get passed a value such as (x, y), CAT will fail since CAT(pattern, (x, y)) does not produce a valid token pasting. We can get around this by doing the same update parameters hack we used before.

```
// Find the result of testing whether a macros is enclosed or not
#define ENCLOSE_EXPAND(...) EXPANDED, ENCLOSED, (__VA_ARGS__) ) EAT (
#define GET_CAT_EXP(a, b) (a, ENCLOSE_EXPAND b, DEFAULT, b )

// Pattern match the result of testing if it is enclose or not
#define CAT_WITH_ENCLOSED(a, b) a b
#define CAT_WITH_DEFAULT(a, b) a ## b
#define CAT_WITH(a, _, f, b) CAT_WITH_ ## f (a, b)

// Defer the call to the CAT so that we get the updated parameters first
#define EVAL_CAT_WITH(...) CAT_WITH __VA_ARGS__
#define CAT(a, b) EVAL_CAT_WITH ( GET_CAT_EXP(a, b) )

CAT(a, (x, y)) // Expands to a(x, y)
```

This entry was posted in **All**, **Programming** and tagged **c**, **cpp**, **macro**, **preprocessor**, **programming** by **saad**. Bookmark the **permalink [http://saadahmad.ca/cc-preprocessor-metaprogramming-basic-pattern-matching-macros-and-conditionals/]** .