If Else Then Else Bridgeable Locality

By: Anthony Sasso

March 21, 2022

Revision History

Revision	Date	Author(s)	Description
1.7	2022/03/21	Anthony Sasso	mainly just updated formatting, to make it fit better...
1.5	22/03/18	Anthony Sasso	depreciated sub-functions, added lambda & function pointer sections, since sub-functions are no longer applicable with knowledge on these solutions...
1	22/03/17	Anthony Sasso	added most of the code snippets, most of the writing as well
0.5	22/03/16	Anthony Sasso	created, don't recall how much was accomplished...

Contents

Introduction / Concept

The initial concept is that by having an if-else with duplicated functionalities we sometimes have to "break" the if-else into separate sections having an if-else . . . some combined / shared code . . . another if-else to continue the branching conditions.

I think that this is an issue due to the variables local to the initial if-else now needing to become (albeit still localized), higher in view and defined before in a parent function [1].

First General Case

```
int main(void){
    int x;
    int y;
    User exampleUser();
    i = exampleUser.getEnvCode();

    int initaliI = i; //needed for second if statement...

    if(i < 10){
        x = somefunc();
        i = x;

    }
    else{
        y = somefunc();
        i = y;
        ~exampleUser(); //stops us from being able to use exampleUser given this context...
    }

    // some code that is common between them... that is sufficiently long to not want to be repeated
        but not sufficiently isolated to be put into a function...

    if(initailI < 10){
        exampleUser.someClassFunc(x);

    }
    else{
        //some other funcs assuming exampleUser no longer exists...
        someOtherfunc(y);
    }
    return 0;
}
```

As you can see, the face we needed ***x***, ***y*** and an ***initialI*** cache ahead of the if-else is not ideal, and may end up increasing the "scope" of many variables that otherwise should just exist between both if statements.

The ideal would be to bridge them across in such a way that local variables in the initial if carry over into the second if, and similar for the else state; But that these are only parametrized inside those scopes, never "entering" the main parent scope.

[1] which may not be wanted & can cause issues as shown in the listed examples

A Minimal Solution to a Minor Problem?

```
int main(void){
    User exampleUser();
    i = exampleUser.getEnvCode();

    if(i < 10): CASE_A{
        int x = somefunc();
        i = x;

    }
    //No CASE needed, implicitly will be given another scope anyways...
        //else is also required to be UN_MACRO, so that all branchings are accounted for
    else{
        i = y;
        ~exampleUser(); //stops us from being able to use exampleUser given this context...
    }

    // some code that is common between them... that is sufficiently long to not want to be repeated
         but not sufficiently isolated to be put into a function...

    then: CASE_A{
        exampleUser.someClassFunc(x);
    }
    else{
        /some other funcs assuming exampleUser no longer exists...
        someOtherfunc(y);
    }
    return 0;
}
```

Here is my proposed solution to this issue that will minimize "weight" on the compiler, and maximize readability for the user:

Essentially we define a cache for if statements where they are intended to bridge, then the compiler goes through merging them into a final "master" if-else so that we don't need to actively repeat code. Some explicit requirements will be:

1. The initial ***"if"*** must have a CASE macro if one or more others do [2]
2. The final ***"else"*** must not have a case macro, this is to account for non-case outputs if only the first has a macro specification. . .
3. Any options that meet the "macro-level" of another option will be put together into the final master-case. But given a nested if statement to differentiate (this is due to them sharing a locality level, while still having different case calls / parameters).
4. This should be done through an if-else style syntax, but (as discussed later) could also be done through localized "sub-functions" that are completely dependent to their defined function and cannot be defined elsewhere.

[2] in the instance there is only one macro it would be recommended to make said initial if, with all subsequent being non-macro types

Implementation

In this scenario the easiest solution would be for the compiler to allocate n buffers upon the realization of the if(): *MACRO* { } . . . ELSE{ } definition. This should work for n cases where n is the number of if, else, and if else statements with macros. [3]

First the compiler will select the leading ***MACRO*** case and allocate it to a ***MACRO_NAME*** buffer. The pre-compiler will put a reference to that section in the codebase in order to return after allocation and skip over the if clauses until it exits, copying between the end of the if clause and the first then clause [4]. The pre-compiler will then allocate the "in-between" area to a buffer labelled either ***BETWEEN*** or ***INTER_CASE*** [5], appending the data inside the previous macro. The pre-compiler will then append the "then" clause with the same ***MACRO***.

The final if clause for the ***MACRO_NAME*** case is:

```
MACRO_BUFFER:
    if(case){
        INITIAL_IF_BUFFER
        BETWEEN_BUFFER
        FINAL_THEN_BUFFER
    }
```

This will then get added by the pre-compiler to the ***FINAL*** buffer of the pre-compiled post-sorted "master-statement" that will reflect the logic.

For ***NON_MACRO*** cases the pre-compiler will only get the initial if case assigning it to a ***NON_MACRO*** buffer [6], appending the ***BETWEEN*** code to it then adding that ***NON_MACRO*** buffer to the ***FINAL*** buffer.

The final if clause for the ***NON_MACRO*** case is:

```
//also always applies to else
NON_MACRO_BUFFER:
    else if(second_case){
        INITIAL_IF_BUFFER
        BETWEEN_BUFFER
    }
```

After this it will return to the next case and continue taking the initial into a buffer appending ***BETWEEN*** and the associated "then" for all of them (except for the ***NON_MACRO*** case described earlier). The resultant source code will function as if the developer had copied ***BETWEEN***, and associated then into each case, but without them having duplicate code or too many "helper-functions" that may reduce code legibility.

[3] Statements with macro labels will be considered ***MACRO_NAME*** where _NAME is the label and cases without macros will be assumed to ignore the if switch and will be considered ***NON_MACRO*** for the following definition.

[4] Requires a second then with the equivalent MACRO name, otherwise if there is no macro thereby fulfilling that clause it will throw a pre-compiler error due to no existing then closing case

[5] We will use the term ***BETWEEN*** for the purposes of this document, to not become too verbose. . .

[6] this one can be reused between all ***NON_MACRO*** cases, as with ***MACRO***

A more full example is as follows:

```
int main(void){
   if(caseOne): INITIALCASE{
      var initalVar;
      //some code here
   }
   else if(caseTwo): SECONDCASE{
      var anotherVar;
      //some other code here
   }

   //below is an example of a case with shared MACRO branching that has been prior defined
   else if(caseThree): INITIALCASE{
      var initalVar;
      //some code here, that will use the same "then" as caseOne...
   }

   //below is to show nesting for different cases with shared NONMACRO branching
   else if(caseFour){
      //some code here that is also non macro but does have a requirement
   }
   else{
      //some code assuming no cases & no macros
   }

   //Shared code that shouldn't / wouldn't be reasonable including into a function

   then: INITIALCASE{
      initalVar++;
      //more code assuming state from INITIALCASE was carried over
   }
   else then: SECONDCASE{
      anotherVar++;
      //other code assuming state from SECONDCASE was carried over
   }
   else{
      //code here assuming the non-macro else if OR final else were true (since both are using the
          implicit NON_MACRO branch)
   }
   return 0;
}
```

This is then translated for the compiler to:

```
int main(void){
   //required to logical OR so that either caseOne / caseThree enter, then are split a second time
       through caseOne & !caseOne (since if it is not then it would be the other)
   //for n shared branches where n > 2 then
      //if(caseOne){} ... else if (caseTwo){} ... else{} "caseN" will be used until the final else
          is left for the rightmost / last declared for the MACRO_NAME label

   if(caseOne || caseThree){
      var initalVar;

      if(caseOne){
         //some code here
      }
      else{
         //some code here, that will use the same "then" as caseone...
      }

      //Shared code that shouldn't / wouldn't be reasonable including into a function

      initalVar++;
      //more code assuming state from INITIALCASE was carried over
   }
   else if(caseTwo){
      var anotherVar;
      //some other code here

      //Shared code that shouldn't / wouldn't be reasonable including into a function

      anotherVar++;
      //other code assuming state from SECONDCASE was carried over
   }

   //below is to show nesting for different cases with shared branching
   else{
      if(caseFour){
      //some code here that is also non macro but does have a requirement
      }
      else{
         //some code assuming no cases & no macros
      }

      //Shared code that shouldn't / wouldn't be reasonable including into a function
   }
   return 0;
}
```

The benefit of this is that to the developer the top option only has the middle code written once, and without the need of any functions which could offload the functionality [7]. If that "***BETWEEN***" area is fundamental to the codebase this can cause issues in overly dispersing the code across too many functions; arguably causing an OOP equivalent to the problem stated above. . .

[7] Although helper functions in the case where they are truly non-fundamental & "off-loadable" are fine

Sub-Functioning [DEPRECIATED]

*** NOTE : This section had been written prior to learning about lambda functions, and looking at it now is essentially the same thing. For the sake of consistency, and possibly if this turns out to not be drop in replaceable with a similar lambda based Implementation it will remain. . . ***

An alternative to the listed above would be to define "code snippet" style functions before a branch occurs and have that inserted as you would a helper function. The only meaningful difference is that they are private to the parent function, defined similar to a case definition, and will always be destroyed upon exiting the scope [8]

A demonstration (using the if case from above), through sub-functioning is as follows [9]:

```
int main(void){
    // Note that sub-functions DO NOT have inputs, due to them inheriting the scope of the called
        function implicitly...
    sub subFunc{
        //Shared code that shouldn't / wouldn't be reasonable including into a function
    }

    //this replaces the INITALCASE re-usability from before, technically also removing the nesting
        in the final if-else-then-else pre-compilation as well...
    sub initalCase{
        initalVar++;
        //more code assuming state from INITIALCASE was carried over
    }

    // Since we cannot / wouldn't want to use nested if's we can just give inital cases' then
        result to another sub-function and use that only for these two cases...
    if(caseOne){
        var initalVar;
        //some code here

        subFunc();
        initalCase():
    }
    else if(caseTwo){
        var anotherVar;
        //some other code here

        subFunc();

        anotherVar++;
        //other code assuming state from SECONDCASE was carried over
    }
    else if(caseThree){
        var initalVar;
        //some code here, that will use the same "then" as caseone...

        subFunc();
        initalCase():
    }
    else if(caseFour){
        //some code here that is also non macro but does have a requirement

        subfunction();
    }
    else{
        //some code assuming no cases & no macros

        subfunction();
    }
    return 0;
}
```

---

[8] This includes forbidding that function from being passed out of scope, this can be justified due to that use-case necessitating the creation of a public function already defined of scope NOT a sub-function within THIS scope. . .

[9] interestingly enough this "should" compile exactly like the above example with the sub-function acting as an alias for the compiler to swap the logic with, the exception being that shared then branches will ALSO need sub-functions that can be input into those localities, which removes the nesting seen in the above pre-compilation solution

Lambda Functions & Localized Function Pointers

Using the above example we can essentially implement the sub-function use case using function pointers or lambda functions. . . the only major alteration being syntax and the inclusion of variables, here is the revised code example:

```
int main(void){

   //unlike sub-functions lambda functions DO require inputs, or at least the declaration of pass by
       reference... (will use pass by reference because that is what I had originally had in mind
       with the sub-function solution)
   void lambdaFunc [&](){
      //Shared code that shouldn't / wouldn't be reasonable including into a function
      };

   //this replaces the INITALCASE re-usability from before, technically also removing the nesting in
       the final if-else-then-else pre-compilation as well...
   void initalCase [&]() {
      initalVar++;
      //more code assuming state from INITIALCASE was carried over
      };

   // Since we cannot / wouldn't want to use nested if's we can just give inital cases' then result
       to another sub-function and use that only for these two cases...
   if(caseOne){
      var initalVar;
      //some code here

      lambdaFunc();
      initalCase():
   }
   else if(caseTwo){
      var anotherVar;
      //some other code here

      lambdaFunc();

      anotherVar++;
      //other code assuming state from SECONDCASE was carried over
   }
   else if(caseThree){
      var initalVar;
      //some code here, that will use the same "then" as caseone...

      lambdaFunc();
      initalCase():
   }
   else if(caseFour){
      //some code here that is also non macro but does have a requirement

      lambdaFunc();
   }
   else{
      //some code assuming no cases & no macros

      lambdaFunc();
   }
   return 0;
}
```

Final Comments & Notes

I think overall a combination of both options would be applicable with the sub-functioning being essentially obsolete with lambdas already having this functionality, and them being the most code-clean solution as compared to the “if-else-then-else”; but the latter being more applicable for imperative programming or non-functional languages where lambda functions or functions of any kind cannot be utilized. . .

It is also meaningful to mention the added complexity for equally branched cases (ex caseOne & caseThree, caseFour & else) that the lambda cases could give. One reason is due to the function call themselves being likely readable but still having a level of obfuscation the case’d alternative can mitigate this by being more explicitly “hard-coded” in how it shows the correlation between equally branched paths.

Not to mention refactoring this may require more mental load to remember which case was which implicit branch and change to the correlated lambdas. Comparatively, the “if-else-then-else” option gives a more explicit static branch identity that can be seen closer to how the code will eventually be compiled, run, and debugged. In the case where a branch must be altered it could also increase the legibility through said explicit nature; providing a quicker refactorization by just changing the ***MACRO_NAME*** label in order to alter its then case potentially decreasing debug times for situations where one initial case needs to be altered later on.