



Handling Conditional Compilation In CDT's Core

Chris Recoskie
Team Lead
IBM CDT Team

Problem Statement

- The C Preprocessor (Cpp) allows one to add conditional compilation directives which cause the code to potentially be different depending on the state of defined macros
- CDT currently only parses and indexes one compilation path
- Searches and refactorings can miss elements in the inactive code
- How to solve this?

```
#ifdef FOO  
    foo( );  
#else  
    bar( );  
#endif
```

Problems With The Preprocessor

- Conditional directives can appear at any arbitrary point within a code fragment, provided that the conditional directives are the only thing that appears on a given line of source text
- Branches can contain arbitrary sized fragments of code that are not syntactically correct in isolation
 - They do not have to appear on the well formed boundaries of elements of the language grammar
 - They can “break” constructs
- In the example to the right, what is the type of *y*? It depends.
- Variables might be macros!

```
/* define a 32-bit int on
various platforms */
#ifdef HAS_32_BIT_INT
    int
#else
#ifdef HAS_16_BIT_INT
    long
#else

```

y;

```
#ifdef USE_MACRO_CONSTANTS
    #define x 42
#else
    int x = 42;
#endif
```

y = *x*;

Problems With The Preprocessor

- Parameterized macros can concatenate text
- In the right hand example, the variables `completeStatus` and `errorStatus` get set depending on runtime conditions
- What if such macros are defined differently in different branches conditional directives? The referenced variables can be completely different.

```
#ifndef USE_STATUS
#define ST(VAR) VAR##Status
#else
#define ST(VAR) VAR
#endif
```

```
#define ST(VAR)VAR##Status

int x;
switch(x) {
    case 0:
        ST(complete) = 1;
        break;
    case 1:
        ST(err) = 1;
        break;
}
```



Problems With The Preprocessor

- Conditional directives can have binary expressions

```
#ifndef __ASSEMBLY__
#if __GNUC__ > 3
    # include <linux/compiler-
gcc+.h>
#elif __GNUC__ == 3
    # include <linux/compiler-
gcc3.h>
#elif __GNUC__ == 2
    # include <linux/compiler-
gcc2.h>
#else
    # error Sorry, your compiler is
too old/not recognized.
#endif
#endif
```

Solution #1: Parse All Configurations

- Could allow the user to enumerate all configurations that they care about.
- Parse the entire code with each configuration separately to build up separate ASTs and indices, then apply operations to all of them
- Pros:
 - Simple to implement
- Cons:
 - Operations get slow and memory intensive the more configs you have
 - Combinatorics
 - 5 binary macros means 32 possible configs
 - What if they care about all configurations?
 - Non-binary macros means unbounded number of configs

Solution #2 – Support Conditional Directives in C/C++ Grammar

- Put rules in the grammar for matching conditional directives
- Place nodes in the AST corresponding to the directives
 - Node has sub-trees for each branch of the conditional
- Pros:
 - Represents all possible program configurations
- Cons:
 - Directives can appear between any arbitrary tokens
 - Makes the grammar exceedingly complex
 - How to handle directives that break syntactic constructs?
 - Would need nodes with smaller granularity than current AST
 - More like a parse tree
 - Don't really want visitors having to do their own parsing

Solution #3: Garrido-Style Pseudo-Preprocessing and Conditional AST

- AST contains all alternatives that appear as a result of conditional directives
- Nodes are marked with conditions which indicate under what circumstances the node applies
 - Nodes are not children of nodes for directives
- Pros:
 - Complete (but compact) representation of the code
 - Can always safely cache headers
- Cons:
 - Requires a lot of rework of CDT parsers and APIs

Step #1: Partially Preprocess

- Tokenize the Cpp directives along with the regular code
- Generate include dependency graph (with conditional edges)
- Macro calls are tokenized with a special macro call token, which is later resolved by indexing into a macro table
- Macro defined within conditional compilation directives have their macro table entries marked with their guarding conditions
- Text that is not a part of a Cpp directive is just parsed as a block of text

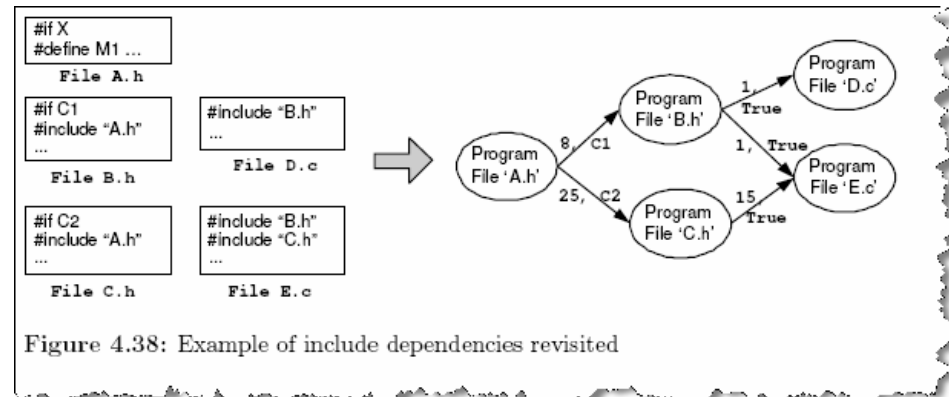


Figure 4.38: Example of include dependencies revisited

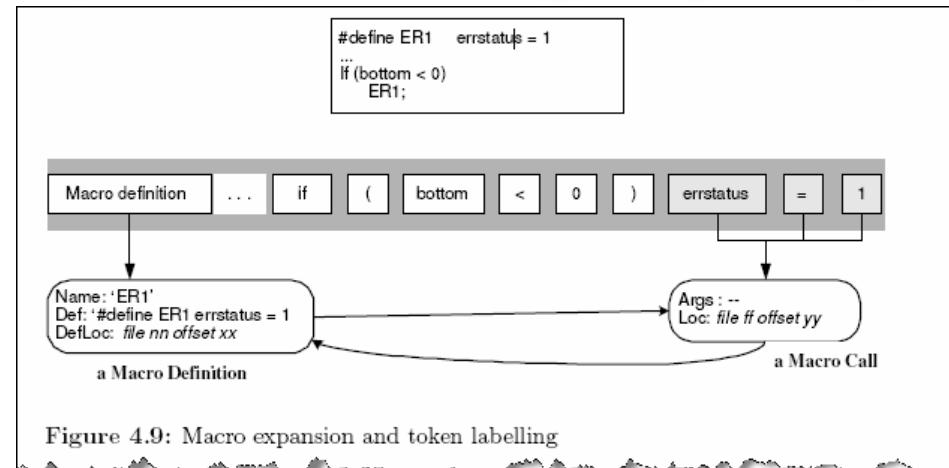


Figure 4.9: Macro expansion and token labelling

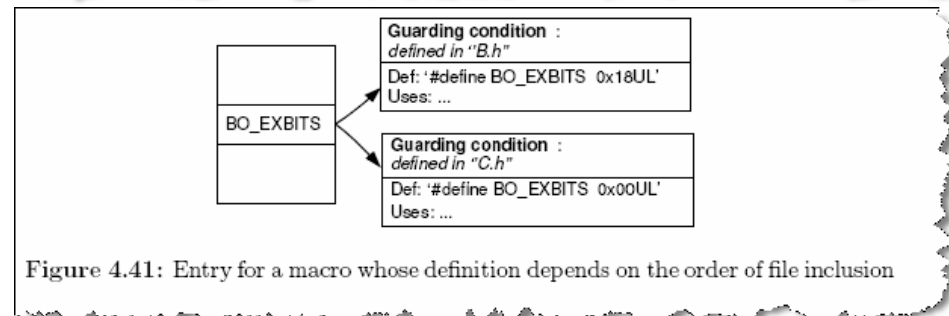


Figure 4.41: Entry for a macro whose definition depends on the order of file inclusion

Step #2: Parse and Complete Conditional Directives

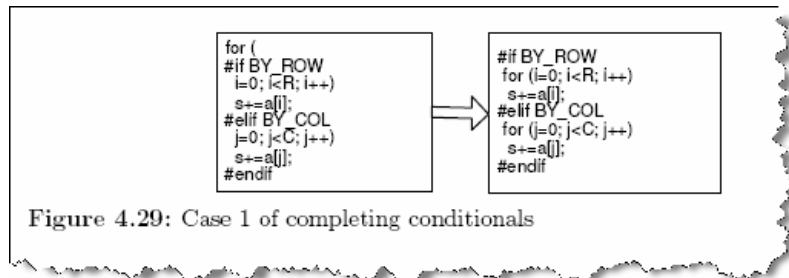


Figure 4.29: Case 1 of completing conditionals

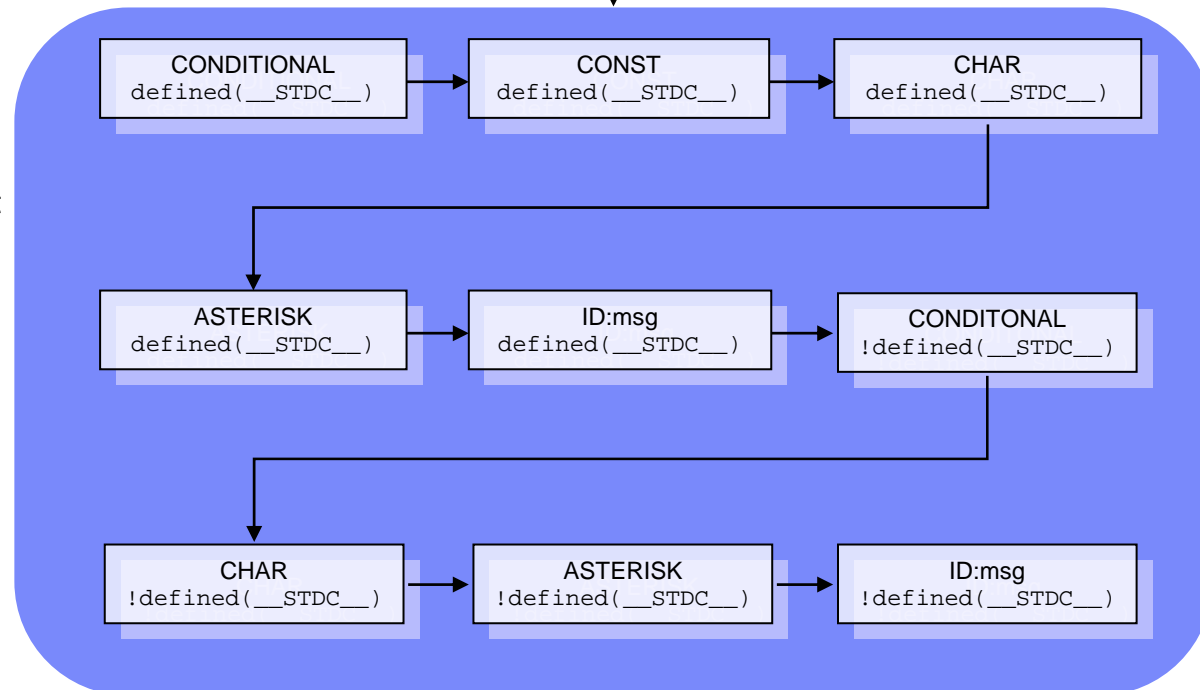
- Another parser that works on the output from Step #1
- Run Garrido's Conditional Completion Algorithm on directives
 - "Completes" all alternatives of a conditional so they are syntactically valid C/C++ constructs.
 - Requires detection of certain keywords and punctuation
 - For C:
 - Composite statements
 - For loops
 - Enums
 - Semi-colons
 - For now all other text is parsed as just blocks of text

Step #3: Preprocess/Re-tokenize for C/C++

- Conditionals are tokenized as a single terminal, with a condition
- When a conditional is encountered, assume it is true and push its condition onto the current condition stack
- On the contained code:
 - Run the preprocessor to expand macros assuming that the conditions in the condition stack are true
 - Mark tokens with conditions from condition stack
 - Nested conditions are conjoined
 - Compatible conditions are merged.

```

#ifdef __STDC__
    const char * msg;
#else
    char * msg;
#endif
    
```



Step #4: Parse For C/C++

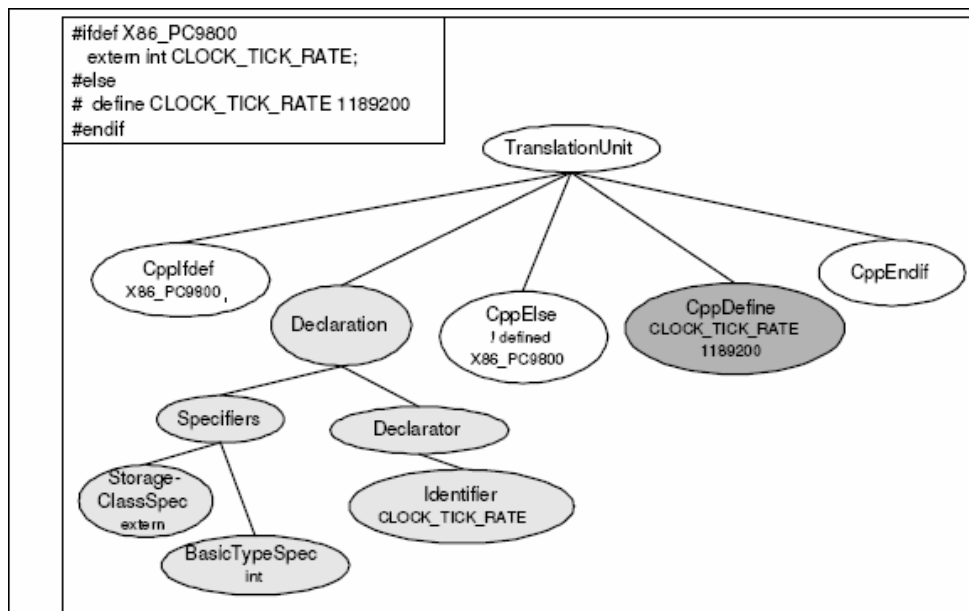
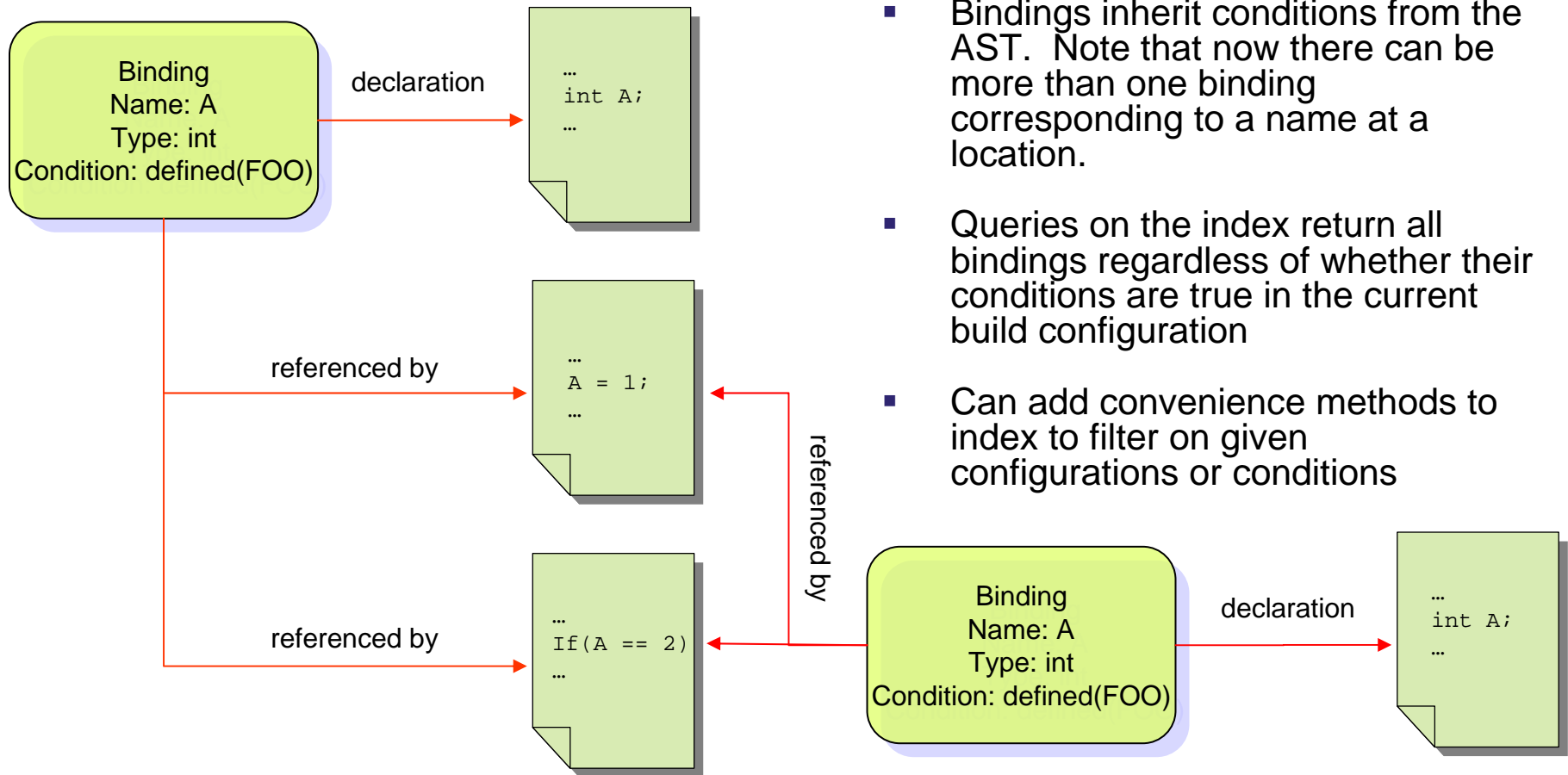


Figure 5.2: Abstract syntax tree with Cpp directives as nodes

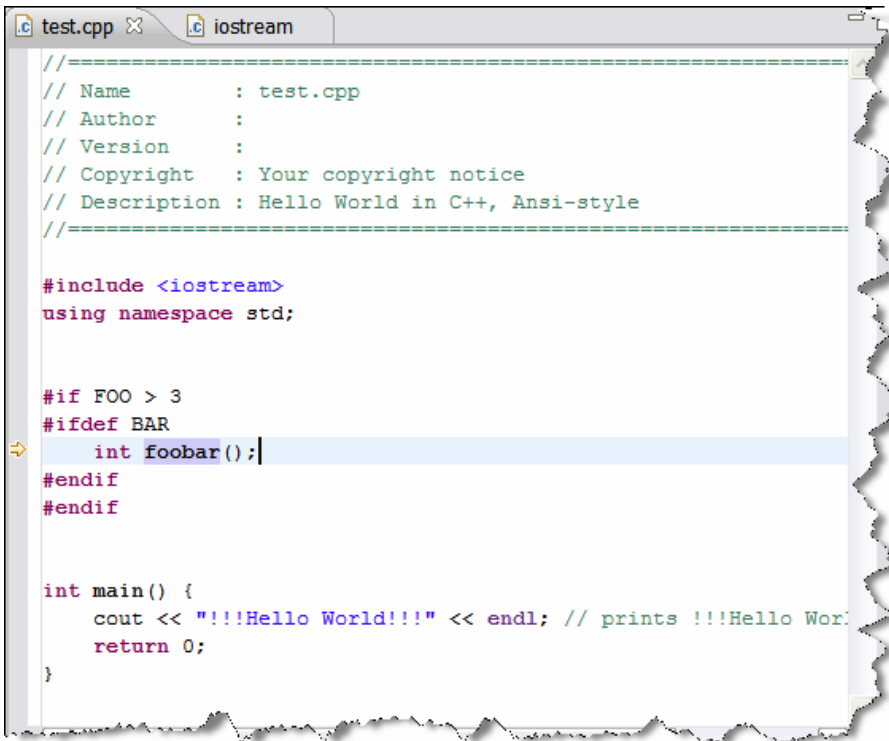
- C/C++ parser grammar includes terminals for conditional directives
- Parse and build an AST.
 - AST nodes and bindings have conditions on them based on the conditionals that guard them

Step #4: Index



- Build an index from the AST.
- Bindings inherit conditions from the AST. Note that now there can be more than one binding corresponding to a name at a location.
- Queries on the index return all bindings regardless of whether their conditions are true in the current build configuration
- Can add convenience methods to index to filter on given configurations or conditions

UI Behaviour In Presence of Conditionals



```

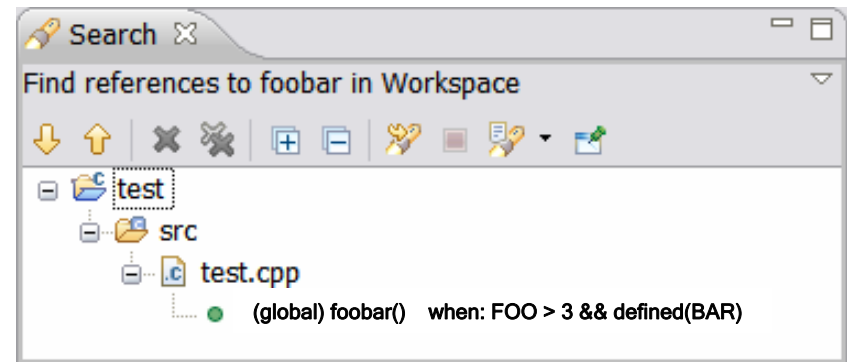
test.cpp x iostream
//=====
// Name      : test.cpp
// Author    :
// Version   :
// Copyright : Your copyright notice
// Description: Hello World in C++, Ansi-style
//=====

#include <iostream>
using namespace std;

#if FOO > 3
#ifdef BAR
    int foobar();
#endif
#endif

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello Wor
    return 0;
}
    
```

- Search matches, etc. show hits in all compilation paths
- Hits shown with their conditional next to them
- Could specify “condition working sets” to restrict scope (tie in to build configurations too)
- Could have an option to only report matches from the active configuration
 - Could maybe parse “the old way” in this case to make things faster.





Effort Required

- Quick and dirty effort estimates and work breakdown structure have been created.
- Over 8 person-months of effort required.
 - Not enough resources at IBM to do this for Ganymede, but this is a priority for IBM for the future, so we can commit to working on this post-Ganymede.

- Conditional Compilation	177 days
- Pseudo Preprocessor	25 days
Condition data structures/API	5 days
Conditional token class(es)	2 days
Grammar for lexer	2 days
Include Dependency Graph data structure	5 days
IDG build actions	5 days
Macro Table data structure	3 days
Macro Table build actions	3 days
- Conditional Completion	20 days
Conditional parser	10 days
Conditional Completion Algorithm	10 days
Preprocessor	3 days
- Parser	25 days
Grammar changes for conditionals	2 days
Changes to grammar actions to track conditions	3 days
Changes to IASTNodes	10 days
Changes to IASTBindings	5 days
Changes to binding resolution algorithm	5 days
- Indexer	17 days
Changes to indexing framework	2 days
Changes to PDOM to allow storing conditionals w/ nodes	10 days
Changes to IIndexBindings	5 days
- Cmodel	7 days
ICElement changes for conditions	5 days
Cmodel builder	2 days
- UI	80 days
Search	10 days
Content Assist	10 days
Navigation	10 days
Call Hierarchy	10 days
Type Hierarchy	10 days
Class Browser	10 days
Include Browser	10 days
Cmodel UI representation	10 days



References

- Garrido, Alejandra, Ph.D. *Program Refactoring In The Presence of Preprocessor Directives*