REJUVENATING C++ PROGRAMS THROUGH DEMACROFICATION

A Thesis

by

ADITYA KUMAR

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Bjarne Stroustrup
Committee Members,	Gabriel Dos Reis
	Guergana Petrova
Department Head,	Duncan M. (Hank) Walker

December 2012

Major Subject: Computer Science

ABSTRACT

As we migrate software to new versions of programming languages, we would like to improve the style of its design and implementation by replacing brittle idioms and abstractions with the more robust features of the language and its libraries. This process is called source code rejuvenation. In this context, we are interested in replacing C preprocessor macros in C++ programs with C++11 declarations.

The kinds of problems engendered by the C preprocessor are many and well known. Because the C preprocessor operates on the token stream independently from the host language's syntax, its extensive use can lead to hard-to-debug semantic errors. In C++11, the use of generalized constant expressions, type deduction, perfect forwarding, lambda expressions, and alias templates eliminate the need for many previous preprocessor-based idioms and solutions. Additionally, these features can be used to replace macros from legacy code providing better type safety and reducing software-maintenance efforts.

In order to remove the macros, we have established a correspondence between different kinds of macros and the C++11 declarations to which they could be transformed. We have also developed a set of tools to automate the task of demacrofying C++ programs. One of the tools suggest a one-to-one mapping between a macro and its corresponding C++11 declaration. Other tools assist in carrying out iterative application of refactorings into a software build and generating rejuvenated programs. We have applied the tools to seven C++ libraries to assess the extent to which these libraries might be improved by demacrofication. Results indicate that between 52% and 98% of potentially refactorable macros could be transformed into C++11 declarations.

ACKNOWLEDGEMENTS

First of all I would like to thank my advisor Dr. Bjarne Stroustrup for giving me this opportunity to do research under his guidance. He introduced to me the techniques of replacing macros with C++ declarations and provided suitable guidance to help me pursue this line of research. Of significant importance is the financial support by "Fujitsu Laboratories of America", that helped me devote sufficient time to complete this research in time. I would like to thank rest of my committee members, Dr. Gabriel Dos Reis and Dr. Guergana Petrova, for providing invaluable feedback and supporting me through out my research work.

Right from the day one, I got the help of Andrew Sutton. This thesis work would not have been possible without his support. My style of programming and technical writing continues to be influenced by his. He also proofread and helped me organize this thesis work. I would like to continue to collaborate with him in the future.

I owe special thanks to Dr. Lawrence Rauchwerger for teaching the course on Compiler Design which has significant impact on the design of tools during my research work. I would like to thank other faculty and staff members of Computer Science department for making it a great place to pursue research. I would also like to thank Abe, Michael and Yuriy for their help.

Most importantly, I would like to thank my family members – Mummy, Papa, Suryansh, Lala and Nandini – who are so close no matter how far, for their unconditional love and support.

Finally, I would like to thank God for making me fortunate enough to work with great people around me.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

As we migrate software to new versions of programming languages, we would like to improve the style of its design and implementation by replacing brittle idioms and abstractions with the more robust features of the language and its libraries. Using new language features and libraries we can improve readability (hence maintainability), reliability, and performance. We refer to this kind of program modification as *source code rejuvenation* [1]: a one-time modification to source code that replaces deprecated language and library features, and programming idioms with modern code.

In this context, we are interested in replacing C preprocessor macros with new features and idioms in the C++11 programming language. The presence of C preprocessor macros presents many obstacles for C/C++ programmers and software maintenance tools [2–4]. Because the C preprocessor operates on the token stream independently from the host language's syntax, its extensive use can result in a number of unintended consequences. Bugs resulting from these conflicts can lead to hard-to-debug semantic errors. Program analysis tools (e.g., debuggers, profilers and other cross-reference tools) are faced with the non-trivial task of maintaining mappings between abstractions in non-preprocessed code with those in the eventual translation unit [5].

In the C-family of languages there has been an effort to limit the use of unstructured preprocessor constructs. Java has none, C# limits preprocessing to conditional configuration, D replaces common preprocessor uses with different kinds of declarations (e.g., `version`), and most modern C++ coding standards ban "clever" or ad hoc usage of the C preprocessor [2, 6–8]. In C++, the number of reasonable uses

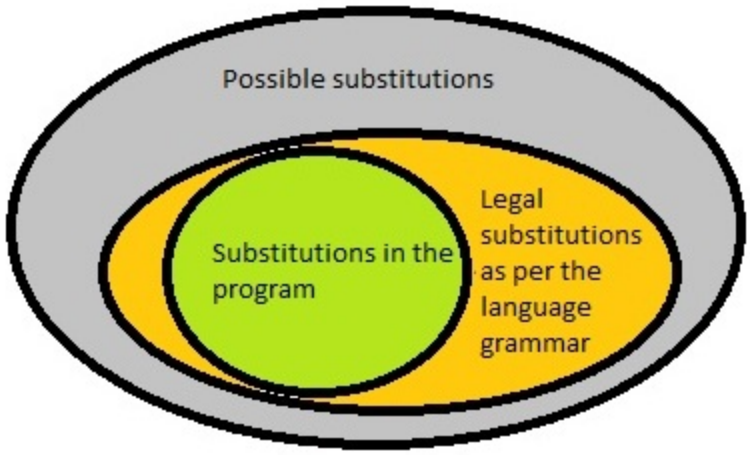

Figure 1.1: A comparative view of possible macro substitutions vs. legal substitutions.

of the C preprocessor decreases with every major revision of the language. By using C++11 features like generalized constant expressions, type deduction, perfect forwarding, lambda expressions, and alias templates [9], we can avoid using many previous macro-based idioms and solutions. Additionally, these new features can be used to replace macros from legacy code providing better type safety and reducing software-maintenance efforts. We develop techniques and tools to replace macros with C++11 declarations using these features.

1.1 Motivation

Macros are sources of errors because their substitutions are not subject to the execution model of C/C++ [3, 10]. The substitution of macros takes place at the token level, meaning the compiler does not have any knowledge of the abstraction they provide within the program. Even if macros have been correctly used in the program, there is always a chance of their misuse in future because their substitution cannot be precisely explained with the help of language grammar (Figure-1.1).

In the case of software where the macro usage and its caveats are undocumented

is specially error prone. In such a case bugs may arise in the future if programmers with little experience with the software try to modify the source code affected by macros. For example:

```
#define MAX(X,Y) (((X)>(Y))?(X):(Y))
```

This archetypal macro usage works perfectly fine until it is invoked with arguments having side-effects; it will then duplicate the side-effect and could cause unexpected program behavior. For example:

```
int i = 0, j=1;
int k = MAX(i++,j++);
//compiler sees int k = (((i++)>(j++))?(i++):(j++));
```

There are several other macro pitfalls which is why most coding standards ban "clever" or ad hoc usage of the C preprocessor [2, 6–8]. Spuler et al. and Ernst et al. present several macro pitfalls in their study. Spuler et al. consider a macro usage as erroneous if it is neither used to imitate function calls nor used as symbolic constants [11]. Ernst et al. provide a broader definition of erroneous macros. They consider a macro usage as erroneous "... if its functionality could be achieved by a C function, but, in some contexts, the macro behaves differently than that function would." [3].

After studying the use cases of more than thirty five thousand macros in programs spanning over 1.5 million lines of code (see Table-3.1 on p. 31), and their pitfalls we conclude that it is best to avoid using them unless it is absolutely required. Stroustrup says, "The first rule about macros is: Don't use them if you don't have to. Almost every macro demonstrates a flaw in the programming language, in the program or in the programmer." [12].

The presence of C preprocessor macros in C++ programs also affects the capa-

bilities of program analysis tools [4]. There are three different ways by which the tools deal with the C preprocessor. They may ignore the preprocessor, operate on preprocessed code, or transmit the C preprocessor information in the parse tree [3].

Tools that ignore the preprocessor can only be used to obtain approximate information about the program. For example: pattern matching tools in the text-editors like, vim and emacs are preprocessor agnostic. Those operating on preprocessed program are also enfeebled by the fact that they cannot interpret preprocessor abstractions in the final program; compilers and debuggers come under this category. They can only reason about one particular preprocessed version of the program with specific settings (with respect to macro definitions and conditional compilation).

Lastly, the tools which permit source code analysis in presence of macros are useful for maintaining and evolving C++ programs. These tools maintain a mapping between non-preprocessed code and preprocessed code in order to reason about the abstractions provided by macros in the program. Some tools in this category provide refactoring on preprocessor directives [13–15], others detect potential errors due to macro usage [5, 11, 16]. Although these tools provide facilities that make reasoning about the program easier in the presence of macros, they do not remove macros; this keeps the programs with macro usage susceptible to bugs in future.

Effectively, program analysis tools which could assist programmers in removing or suggesting alternative idioms to macros, are lacking. In fact, the techniques to minimize macro usage exists mostly in text-books and style guides [2, 6–8]. This is because it is not possible, in principle, to reason about macros with guarantees. Using syntax based macros, in principle, can provide facilities similar to C preprocessor along with syntactic safety [17,18]. However, this approach is only academic because the migration from C preprocessor to a syntax-based macro language is impractical for overly popular programming languages like C and C++.

Therefore, it is important that macro usage be minimized from C++ programs. By replacing macros with C++11 declarations, the behavior of the program improves because language abstractions make information explicit—to programmers, compilers, and other program analysis tools—that remains obscure with macro usage. Further, program analysis tools are able to assist users in carrying out software maintenance tasks with better guarantees and additional transformations because macro usage limits the capabilities of these tools infer certain properties from programs [4, 15, 19–21].

It should be noted that replacing macros with language declarations improves the maintainability of programs. It helps in ***adaptive maintenance*** [22]; since the future uses of macros (defined in the program) do not have a well defined scope (Figure-1.1), thus, removing them prevents future bugs due to the misuse of defined macros. It also helps in ***preventive maintenance*** [22] because removing the macros with language constructs increases the type-information which simplifies program comprehension for programmers and tools.

In the next section we describe our approach to cope with the C preprocessor macros and to what extent we were able to remove macros from C++ programs.

1.2 Research overview and contributions

This research work is an initiative to develop tools to replace a large number of macros from C++ programs in an automatic/assisted way. We attempt to cope with the macros in C++ programs by elevating macro definitions to equivalent C++ declarations. We define the term ***demacrofication*** as a process of removing macros from C++ programs by replacing them with corresponding C++11 declarations and leaving the program in a valid state (Figure-1.2).

A high level overview of demacrofication is shown in Figure-1.2. The process

begins with an initial version of the source code, which contains macros, and ends with a final version in which macros have been replaced by C++11 declarations.

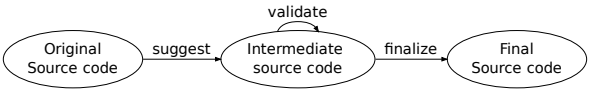

Figure 1.2: The demacrofication process.

There are three phases of demacrofication:

1. Identify the complete set of macros that can feasibly be replaced with C++11 declarations.

2. Refine that set to only those transformations that produce valid builds.

3. Produce a final, working version of the program.

The process is similar to how a programmer might manually perform this task: find a macro, replace it with a declaration, and rebuild the program to ensure that the change does not break the build.

In order to remove the macros, we have established a correspondence between different kinds of macros and the C++11 declarations to which they could be transformed. We have also developed a set of tools to automate the task of demacrofying C++ programs. One of the tools suggest a one-to-one mapping between a macro and its corresponding C++11 declaration. Other tools assist in carrying out iterative application of refactorings into a software build and generating rejuvenated programs. We have applied the tools to seven C++ libraries to assess the extent to which these libraries might be improved by demacrofication. Results indicate that between 68% and 98% of potentially refactorable macros could be transformed into C++11 declarations.

The following are major contributions of this research:

- A novel categorization of macros based on completeness and dependence of replacement text, and how they relate to C++11 expressions.

- An automated decision process to identify macros which could potentially be replaced, and a one-to-one mapping between potentially replaceable macros and corresponding C++ declarations.

- Tools to suggest possible transformations and assist in automating the source code rejuvenation process. The ***cpp2cxx*** framework is a working prototype that assists in carrying out the task of demacrofication [23].

- Experimental results to validate and justify the effectiveness of our approach.

1.3 Summary

In this chapter we discussed why macros are not as necessary in C++ programs as they are in C programs. We described how extensive macro usage deteriorates the type safety and limits the software maintainability of programs, and motivated the need to remove the macros from C++ programs. We pointed out why it is not feasible to adopt other macro based languages and hence, justified the worthiness of our work. We gave an overview of the process of demacrofication and the tools we developed to assist programmers demacrofy legacy C++ programs. We highlighted major contributions of our work and the usefulness of demacrofication tools in evolving legacy C++ programs. We explained that demacrofication of programs is also helpful from the perspective of software engineering.

In the next Chapter-2, we digress a little to get an idea of the approaches proposed to cope with the preprocessor directives in the past. The literature survey establishes the fact that coping with C preprocessor macros has been a non-trivial task. We explain why previous approaches are insufficient to provide a feasible alternative to

macro usage on a case by case basis. In Chapter-3 we describe the categorization of macros, and in Chapter-4 we present a couple of analyses which helps refine the selection of potentially demacrofiable macros. The concepts developed in Chapters 3 and 4 are used in formulating mappings between macros and C++11 declarations in Chapter-5. After that we introduce the ***cpp2cxx*** framework that we developed to carry out the task of demacrofication in Chapter-6. The extent to which macros were removed as a result of applying our demacrofication tools is presented in Chapter-7 and we conclude in Chapter-8.

2. RELATED WORK

There have been many studies of the use of C preprocessor in C and C++ programs. In C++, the number of reasonable uses of the C preprocessor decreases with every major revision of the language. In C++11, the introduction of generalized constant expressions, type deduction, perfect forwarding, lambda expressions, and alias templates eliminate the need for many previous macro-based idioms and solutions. However Sutton et al. speculated that it would not be possible to completely eliminate macro usage [24]. The literatures that address the issues related to macros and C preprocessor in general, can be classified into three broad categories.

1. Surveys of C preprocessor usage: In this category we include the research works which focus on studying the practical usage of macros in C/C++ programs. The studies include finding patterns in the usage of macros and conditionals, erroneous macros, different alternatives that can be used instead of macros. Relevant papers are [3, 24, 25].

2. Refactoring Macros: Some research works focus on building refactoring browsers for macro based languages. The refactoring browsers emulate the C preprocessor in order to facilitate the desired refactorings. They study problems encountered, during various refactorings, due to the presence of macros e.g., variable renaming etc. Relevant papers are [14–16, 26].

3. Macro Languages: In order to address macro pitfalls, some researches advocate the usage of syntax macros in C/C++. This would make the macro language more powerful and would make the representation of macros in the Abstract Syntax Tree (AST) possible. Relevant papers are [17, 18, 27]. The idea of

replacing the C preprocessor with a syntax macro based language has a major drawback that it requires the modifications in the language grammar to represent macros in the AST.

We now study several research works in each category and associate their work in the current context. We specifically focus on different approaches taken to remove macro pitfalls.

2.1 Surveys of preprocessor usage

Ernst et al. analyzed the preprocessor usage among a large number of contemporary software [3]. They were one of the first to examine practical programs to determine the obstacles to program understanding in the presence of C preprocessor. They identified common macro pitfalls. Based on their analysis they found patterns in the usage of macro bodies and characterized macros into 28 different categories. The taxonomies of macro bodies were based on three broad parameters.

1. Structure of macros in terms of the C language grammar.

2. Frequency of macro usage.

3. Context of macro usage.

The results in paper suggests that 42% of all the macros had replacement texts which were constants, 33% were expressions, 5.1% were statements and 2.1% were types. If we can find a one to one correspondence between the macro bodies and C++ declarations then it would be possible to translate around 80% of macros into valid C++ constructs. This is an approximate figure because macros bodies may reference free variables, create dynamic scopes, have out of order dependencies. However, these results were very important to motivate our effort to eliminate macro usage.

Mennie and Clarke specified that the problems encountered while refactoring C/C++ programs are due to the intermixing of two independent languages (C/C++ and C preprocessor) in a program [25]. The idea presented in the paper was to replace macros with equivalent language constructs. Their work was similar in spirit, although we have a much higher conversion ratio. They classified the macros into about two dozen categories based on the styles of macro usage. Our approach towards the classification is different from theirs, that during the classification process, we are only interested in the kinds of macros that could eventually be replaced.

The paper argued that even when macros are mostly used for simple tasks it is non trivial to get rid of them because of different language scoping rules and lack of strong typing for the macros. It seems there are more reasons, such as pass by name semantics for function-like macros, facility of lexical manipulation of tokens, and lack of rules for linear ordering between definition and usage.

One of the important speculations of this paper was the case when a macro is used for more than one type. They speculated that some transformation could be done using C++ templates. Although they might not have anticipated C++11 [9] features like generalized constant expressions and rvalue-references. Our refactoring for function-like macros is template based and uses type inference facility of C++11 (i.e., `auto` and `decltype`).

Since the macros obey different scoping rules it might be possible that, after demacrofication, the macro invocation points are no longer in the scope of the transformed macro. Mennie et al. discussed about where to place the translated constructs [25]. Should it be placed at the original place of definition? should it be placed near to the first usage, or should it be placed at the point of least common ancestor (LCA)? Each approach has its advantages and drawbacks. They do not try to migrate the transformed macro unless it is required, for example:

```
float circumference()
{
#define R 10;
#define PI 3.14
  return 2*PI*R;
}

float area()
{ return PI*R*R; }
```

To figure out the placement of declaration to be migrated, they constructed a block dependency graph. Briefly, the idea was to look at the first usage and trace backwards in the file where the migrated declaration can be *legally* placed. In the results that they present for the ***Vim***, their approach required significant number of movements but very few in ***Nethack***. After the macro has been moved to a preceding location with respect to the original definition, it has to be checked whether there are any name conflicts. One drawback of this approach is that, if there are significant number of movements, the organization the code will differ considerable from the original.

Our approach is slightly different from theirs. First, we do not move macros to a preceding location in the case of overlapping scope. However, there are situations where it is required to migrate the declaration to a different location. For example, when there is a function-like macro which is defined inside a function, our method of demacrofication requires the transformed construct to be placed at a place where the scope of all the variables inside its body intersects. Another situation would be when a macro accesses a member variable of class and is defined outside the scope of the class. Here also it is not possible to get the type information and access privileges

to a member variable of class if the transformed construct is kept outside the scope of a class.

Mennie et al. also present issues related to the migration of transformed declaration to appropriate locations. One of the issues relates to the placement of comments when a macro is moved. We try to address the issue to some extent. During the parsing process the comments which are within the body of macro are kept as part of the replacement text and are written as a part of the demacrofied construct. This is a general observation that comments related a macro tend to be within the body of the macro, so even the movement of demacrofied constructs has little effect on the maintainability aspects of a software.

One of the most important usages of C preprocessor is to achieve portability across various platforms. Sutton et al. describe techniques for building portable software by examining three heavily ported and widely used C++ libraries. QtGUI Toolkit, ACE, and Boost C++ libraries [24]. It identifies techniques that are used to achieve portability like: use of logical namespaces, replaceable and parameterized inclusions, macro-level compiler abstractions. They predicted that C preprocessor usage would become increasingly important as new language features are added to C and C++ programming languages. That means issues related to macro-pitfalls will continue to increase in future. Our work aims to achieve the opposite. We want to remove as many macros from source code as possible. However, to remove macros that are used for portability and configuration remains an open problem.

2.2 Refactoring macros

Padioleau [14] represents another school of thought to get rid of preprocessor pitfalls. He advocated the idea that preprocessor directives be directly represented in the Abstract Syntax Tree of C/C++ programs. But since a macro body can, in

principle, represent anything from an incomplete C/C++ expression to a full C/C++ program, he tried to address the issue by extending the grammar of the language by introducing fresh tokens. He implemented a front end (Yacfe) which represented preprocessor directives along with other C/C++ language constructs in one parse tree.

His approach has significant advantages from the programmer's point of view. First, it does not add new language features or operators. That means a programmer does not have to learn any new feature to write the program. The second advantage is that representing conditionals in the parse tree facilitates variability analysis by the refactoring browsers. This would help cross platform refactoring support for refactoring browsers using Yacfe. He analyzed and presented the results of parsing of around 16 large open source projects with a very high success rate.

The paper of Saebjornsen et al. focused on detecting errors in the usage of preprocessors by analyzing consistency in their usage [16]. They hypothesized that inconsistent macros usage is a strong indicator of macro errors. And the results that they present seem to support their claims. They described two general approaches to counter the errors related to preprocessors. First was developing an alternative language to C preprocessor. However, this is not feasible as there is a large amount of C/C++ legacy code that uses the C preprocessor. The second approach was to design analysis tools to detect incorrect preprocessor usage. Their second approach has three alternatives which are similar to the approaches given by [3] which are (a) ignoring the preprocessor, (b) analyzing the preprocessed code and (c) attempting to emulate the preprocessor. They worked on the last alternative of the second approach i.e., emulating the C preprocessor.

To detect inconsistencies in the preprocessor usage their tool generated expression trees of the arguments of all the macro invocations of a function-like macro [16].

And then the expression trees were normalized. Thereafter, any inconsistency in the structure of expression trees was a determinant of erroneous macro usage. It seems intuitive that the approach should indeed work. The only limitation was that it focused on a very specialized source of error of a macro usage. Even though it detected errors with a large success rate, this is a small percentage of errors that are possible when macros are used to their full potential [3]. Also, the paper talks about function-like macros to a great length which is a very small percentage of total macro usage in general. Nevertheless this concept can be implemented in refactoring browsers to determine error related to function-like macros.

Kastner et al. presented a variability aware framework to parse code with conditional compilation [28]. The framework used SAT solvers during lexing and parsing to effectively reason about features and their relationship. The variability aware lexer was used to construct parsers that produce AST with variability for un-preprocessed code. This helped in detecting syntax errors as well as variability aware type checking.

Spinellis presented algorithms to automatically analyze and remove identifier instances in text-macro based languages [26]. One of the important contributions of this paper was that it addressed the operation of renaming an identifier even when the identifier is generated as a result of token-concatenation by the preprocessor. He used the concept of token equivalence class to determine lexical equivalence and partial lexical equivalence of tokens to facilitate renaming identifiers.

To efficiently determine the impact of a change in a specific macro definition is difficult. Vidács et al. presented a technique to compute this [29]. They used *slicing* to compute macro slices and construct a macro dependency graph (MDG). They provided an effective way to perform impact analysis of macros but did not give experimental results to validate their concepts.

This thesis of Vidács addressed the problems related to refactoring of preprocessor based language particularly C/C++ [30]. It provided a model of preprocessor language elements and their relationships. The idea was to build a meta-model Columbus Schema (a C++ schema for various re-engineering and reverse engineering tasks such as creating UML class diagrams and calculating metrics [31]) which played a key role in fact extraction and representation process. He treated C preprocessor as a separate language. Refactoring was carried out at model level which was followed by the validation stage where each macro invocation point could be examined to validate the refactoring. The model-driven approach allowed one to carry out refactoring at the model level, including verifying the preconditions. The refactoring framework could be used to add a parameter to the macro identifier of object-like and function-like (including variadic) macros.

The work of Gravley et al. explained one aspect of a broader problem of object recovery by finding set of macros used to define constants and combined them into groups of `enum` types [32]. They tried to re-engineer legacy code into object oriented language by getting rid of macros. Their work intersects with our work on the issue of eliminating macro usage. However, our approach is completely different and is able to eliminate a high percentage of macro usage.

Garrido et al. studied the challenges that come during the refactoring of C programs because of the presence of C preprocessor directives [10]. They presented a set of execution rules that would preserve correctness after refactoring. They also addressed the computational complexity of program analysis, in the presence of conditionals, by representing the them as nodes in the AST.

Marian Vittek designed XRefactory which provides renaming-refactorings for C/C++ programs even in the presence of macros which exploits linguistic capabilities of C preprocessor [15]. However, not all refactorings were possible in the presence

of macros. Even though the tool provides facilities to make reasoning about the program easier, it does not *replace* the macros. That simply means that all bad things that may happen due to the presence of macros in a program will happen (Murphy's law).

2.3 Macro languages

Weise and Crew [17] presented one of the earlier works to address the issues related to macro pitfalls. They derived inspiration from lisp programming language [33] which has powerful syntax macros since early 1960's. Their work is important from the historical perspective since it brought to light the researches – carried out in 1960's and 1970's – supporting the use of syntactic macros to solve the software portability problems. In addition to supporting the same idea this paper proposed an extended version of syntax macros to be implemented in C. The idea was to make the C preprocessor more powerful and make it a part of the C language itself; meaning thereby to enforce parser to do type analysis on the macros as well. This would ensure the correctness of macros according to the grammar of the language. They introduced a set of operators, meta-declaration for syntax macros, and back-quote operator for declaring code-template which returned an AST. Further, they demonstrated how dynamic binding and exception handling could be done using syntax macros. It seems this paper went a little too far in addressing issues related to C preprocessor without considering if the solution was practically feasible or not.

The work of Willink and Muchnick [18] can be said to be an extension of the work done by Weise and Crew [17]. It emphasized the fact that having a powerful preprocessor would mitigate the sources of errors due to the usage of text based macro language like C preprocessor. They introduced a Flexible Object Generator (FOG) as an alternative to the C preprocessor. The FOG had powerful syntax macros

which provided an extensible meta-level facilities. They introduced new syntax, two extra lexical operators in a framework where meta-variables replaced object-like macros, meta-function replaced function-like macros, and meta-statements replaced conditionals. They preserved the concatenation and stringification operations. They identified the fact that `auto` keyword was almost obsolete in C++ so they tried to reuse the `auto` keyword in numerous ways to declare meta-functionality like: using outside of a function before the return type, before meta-level `if` statement to support conditional compilation. For example:

```
auto double WEIRD_NUMBER = 70;


//expression is a meta-type
auto expression subtract(expression a, expression b)
{
  //the $ symbol is used for substituting
  //the value of meta-variables
  $a - $b;
}
```

Willink and Muchnick also presented a meta-compilation model for C++ programs. The model had a meta-compilation stage before the compilation and it integrated well with the compilation stage. They presented a comparative advantage of their approaches as compared to the Weise's [17]. One significant advantage is that it required fewer modifications of the C++ programming language as compared to Weise's work. It would be desirable for C++ to have such features because but with minimum number of changes required to achieve them.

The work of McCloskey and Brewer advocated usage of syntactic macros as an alternative to the existing C preprocessor and presented an AST based macro language

called as ***ASTEC*** [27]. ***ASTEC*** provided all the functionalities that C preprocessor provides but lacked support for C++. Their work is better than the previous two proposals advocating the usage of syntactic macros because they kept in view following three important features:

1. Backward compatibility with C programs
2. Expressibility of all C preprocessor idioms
3. Non-expressibility of potential errors involving side-effects and precedence.

They supported variability analysis [28] by preserving both the branches of conditionals in the syntax tree. The ideas presented in the paper bolstered the claims of [17] and [18] that syntax macros are necessary to get rid of preprocessor errors. They even provided *one time* semi-automatic translation to ***ASTEC*** which eliminated macro pitfalls. The drawback is that the migration from C preprocessor to ASTEC would be impractical for a popular language like C.

Favre describe a new language ***APP*** which had the same semantics as C preprocessor [34]. He gave a formal definition of preprocessor to help tool builders reason about preprocessors by providing a neat framework. The main limitation of this paper is that the grammar is not well formed, thus limited in scope.

2.4 Summary

The surveys papers (Section-2.1) presented problems related to macros and C preprocessor in general and motivated further research to cope with them. As a result, several style guides and literatures suggest using language constructs, instead of macros, to provide abstractions in C++ [2, 6–8].

To assist programmers and maintainers in coping with macros various refactoring browsers provide useful refactorings for macro-based languages (Section-2.2). How-

ever, it does not solve the underlying problem. Our approach replaces macros with C++11 declarations which have precise meaning with respect to the language specifications unlike macros which are set of lexical substitutions.

Finally, the idea of having a syntax based macro language seems promising because it provides a nearly complete and type-safe alternative to C preprocessor (Section-2.3). But the approaches appear to be academic because it would be impossible to replace C preprocessor in general for overly popular programming languages like C and C++.

After analyzing various approaches taken to cope with the macros, our idea of replacing macros with language construct—automatically and assisted—provides the most feasible approach to cope with the macros in programs written in C++ and the programs to be migrated from C to C++.

3. CLASSIFYING MACROS

In this chapter we first give a brief description of the terms related to C preprocessor directives which are relevant in the context of this thesis. Further details related to the terminologies associated with preprocessor directives can be found in [9, 35, 36]. Then we provide a novel categorization of macros to help select the ones which could potentially be replaced by C++ declarations.

3.1 Basic terms related to preprocessor directives

We use the term ***identifier*** for a macro name and ***replacement text*** or ***replacement list*** or ***macro body*** for the fragment of code that replaces the macro during preprocessing. As a result a macro directive can be represented as:

```
#define identifier replacement-text
```

1. ***Function-like*** macros (parametric macros):

   The macros which are used like a function call are called function like macros. They take the arguments much like C/C++ function do. For example:

   ```
   #define ABS(x) ((x) >= 0 ? (x):-(x))
   ```

   One significant difference between a function-like macro and C/C++ style function call is that a function like macro without the pair of parenthesis are left unprocessed by the preprocessor while the function name in C/C++ style function are treated as function pointers [35].

2. ***Object-like*** macros:

   It is an identifier which gets replaced by a set of tokens known as replacement text/list. The set of tokens can be empty (see cat-7) in which case the macro

substitution has no effect. Commonly, they are used to give symbolic name to constants. For example:

```
#define MAX 100000
```

In this example, `MAX` is a object-like macro which gets replaced with `100000` at every use site where it is invoked.

3. ***Conditionals***:

   A conditional preprocessor directive behaves similar to an `if-else` statement. Altogether there are six conditional directives:

   ```
   #if
   #elif
   #else
   #ifdef
   #ifndef
   #endif
   ```

   The conditional directive directs the preprocessor to evaluate the condition for truth value and, depending upon the condition is true or false, a piece of code can be included/excluded from the program. One of the common uses is their use as *include guards* to avoid multiple inclusions of the header files in a translation unit. For example:

   ```
   #ifndef MYHEADERFILE_H
   #define MYHEADERFILE_H

   // headerfile contents

   #endif // MYHEADERFILE_H
   ```

The conditionals are also used in various ways to manage portability and configuration of software [24].

4. `undef`:

   This is used to *un*-define a macro. It means that the scope of macro ends when it is *un*-defined and hence no longer *visible* in the code occurring (lexically) after the `undef` directive. For example:

```
#define X 10
 int i = X;
#undef X


int j = X; //error!
```

   In this example the last line would result in a compilation error stating that the variable `X` is not declared in the scope of the declaration of the last line.

5. Predefined macros:

   These are object-like macros predefined as environment variables etc., in the sense that they can be directly used without *defining* them in the program. They are of three kinds (standard, common and system specific). For example:

```
__FILE__, __LINE__, __cplusplus
```

   For a more detailed explanation of predefined macros see [9, 35, 36].

6. Multiline macros:

   The syntax of C preprocessor macro allows it to be written in a single line [9]. But sometimes, the replacement text is too long or consists of multiple statements; writing it in a single line would deteriorate its readability. So

the replacement text can be split in multiple lines with the help of a line-continuation (\) token. Such macros for which the replacement text spans multiple lines with the help of line-continuation token are called ***multiline***. They can be object-like or function-like. For example:

```
#define OCTAL_CASES '0': case '1': case '2': \
  case '3': case '4': case '5': case '6': case '7'


#define decompose(a0, a1) {\
  int b0, c;  \
  c = a0 ^ a1;  \
  b0 = (a0 << 24);  \
  a0 ^= c ^ b0; }
```

7. Empty Macros:

   A macro with no replacement text is called an empty macro. Ernst Et al. [3] termed such macro as ***null-defined***. Since `null` has a specified meaning in C++, it would be more appropriate to use another name and hence the term ***empty*** macro. These kinds of macros are mainly used for configuration or used as boolean constant in conditional directives [24]. For example:

   ```
   #define MY_HEADERFILE_H
   ```

3.2 Classification of macros

There are two types of macros: ***object-like*** and ***function-like***. But when we study macro usage in programs, we find that it is not sufficient for program analysis tools to stick to just these two classifications. Since the macros get substituted with the replacement text, it is imperative for any program analysis tool to analyze the

replacement text and collect useful information about the context of macro usage and reason about program in a better way.

The downside is that, the C preprocessor performs lexical substitutions with no knowledge of the programming language due to which it is not possible, in principle, to reason about macros with guarantees. However, the programs are written with respect to some context, and hence, *most* of the time the replacement text reflects the language semantics. Obviously, there are some cases where it is not at all possible to make any reason about the replacement text in isolation. Therefore, it is reasonable to say that the replacement text of most of the macros have relevant semantic information. We justify this hypothesis when we present the results of our classification of more than 35,000 macros defined across seven different C++ libraries (Table-3.1 on p. 31).

There have been attempts to classify different use cases of macros which helps reason about their properties with respect to the program. Ernst et al. categorized the replacement text of macros into 28 different categories. They analyzed 26 different packages comprising of around 1.4 million lines of code [3]. Their classification was based on three parameters. First, structures the macros represent in terms of the C language grammar; Second, frequency of macro usage; and Third, the context of macro usage. Mennie et al. classified macros into about two dozen categories [25]. Their taxonomy was inspired by the styles of macro usage in the software they used for their case study. In both the classifications, many of the macro categories have no direct correspondence with C or C++ declarations. For example, there is no way Ernst et al.'s "syntactic" and "not C code" categories, and Mennie et al.'s "token pasting" and "literal expressions" could be expressed in terms of C/C++ language grammar.

Since we aim to replace macros with C++ declarations, we have a somewhat

restricted view of these taxonomies. In particular, we are only interested in the kinds of macros that could eventually be replaced. To achieve this, the following parameters serve as the classification criteria for the macros. Based on these parameters we can decide a couple of things like: whether to transform a macro or not, if the transformation could be carried out directly or it would require further analysis.

1. Syntactic nature of the macro bodies.

2. Presence of free variables in the macro bodies.

3. Contents of conditional directives.

We illustrate each classification criteria in detail in the next section.

3.3 Classification criteria

3.3.1 Syntactic nature of the macro bodies

The syntactic nature of a macro's replacement text is helpful in finding out whether it can be expressed as an abstraction in the C++ programming language. Since macros obey lexical semantics, it is possible to express a macro as a set of tokens which do not have any meaning in isolation as far as the grammar of programming language is concerned. Based on this criteria the replacement text of a macro can be classified into two categories.

If the replacement text of a macro cannot be expressed as abstraction in C then we call that macro as ***partial***. These types of macros are sometimes used to make the code look more readable, concatenate or stringify tokens etc. For example:

```
//specifying the linkage type
#define C_MODE_START extern "C" {
#define C_MODE_END }
```

```
//modifying the style of coding
#define IF if (
#define THEN )

//concatenation operator
#define CONCAT(a,b) a##b
```

Conversely, if the replacement text of the macro can be expressed as abstraction in C; then we call that macro as ***complete***. For example:

```
//expression
#define SUM(A, B) ((A)+(B))

//compound expression
#define TWO_CALLS do { First(); Second(); }while(0)

//cast expression
#define TYPE_CHAR (char*)
```

Note that in the case of macro like `TWO_CALLS` which emulates a compound statement, we relax the requirement for a terminating semicolon. This definition of a ***complete*** macro involves an assumption that macro-dependencies are not taken into account. Taking macro-dependencies into consideration would make it impossible to make such a reasoning because C preprocessor allows token pasting etc. Therefore, based on the definition, a ***complete*** macro could be a *type-expression, value-expression, declaration, or statement*. If we consider the way the Pivot [37] represents expressions, ***complete*** macros correspond to complete C++ AST fragments.

3.3.2 Presence of free variables in the macro bodies

It becomes easy to reason about a macro when its replacement text does not refer to free variables. When a macro has no dependencies it is not affected by changes to program outside its definition. This classification establishes a guarantee that helps in deciding whether dependency analysis should be performed or not. There are two kinds of macros based on this criteria. A ***dependent*** macro contains previously (system/user) defined macros or C++ keywords or any other identifier not in its scope. In other words, ***dependent*** macros contain unbounded variables. For example:

```
// __GNUC__ and __cplusplus are predefined macros
#define __GNUG__ (__GNUC__&&__cplusplus)

//MAX and MIN defined outside the scope of DIFF
#define DIFF((A),(B)) (MAX((A),(B)) - MIN((A),(B)))

//contains char - a keyword
#define TYPE_CHAR (char*)
```

One can argue that why C++ keywords are determinants of dependency. It should be noted that since preprocessing is the first stage of compilation, so a programmer can redefine a keyword to something else and, as a result, it would change the complete meaning of the *re-defined* keyword (with respect to the programming language under consideration). One of the classifications of Mennie et al. i.e., keyword redefinition, supports this argument [25].

On the other hand, a ***closed*** macro does not contains previously (system/user) defined macros or C++ keywords or any other identifier not in its scope. In other

words, ***closed*** macros only contain bounded variables. Obviously, this is the opposite of ***dependent*** macros. For example:

```
//integer literal
#define ARCHITECTURE 32

//string literal
#define DEVICE "COMPUTER"

//empty macros
#define MYFILE_H

//A and B are bounded
#define MIN(A,B) (((A)<(B))?(A):(B))

//a and b are in the scope of CONCAT
#define CONCAT(A,B) A##B
```

3.3.3 Contents of conditional directives

Macros are also used to control the conditional compilation of program text. Ernst et al. claim that 6.5% of all macros are used in conditional directives [3], although the numbers reported by Sutton for C++ Libraries appear to be much higher (although not specifically reported) [24]. We say that any macro appearing in a conditional or include directive is ***configurational***. For example:

```
#ifndef MY_HEADERFILE_H
#define MY_HEADERFILE_H
...
#endif
```

```
#if defined(MACRO1) && defined(MACRO2)
//...
#endif
```

The macros `MY_HEADERFILE_H`, `MACRO1` and `MACRO2` are categorized as configurational. Configurational macros are never transformed. Replacing a configurational macro with a C++ declaration would make it invisible to the preprocessor, thus guaranteeing a broken build.

A macro that is not configurational is ***non-configurational***. ***configurational*** macros may or may not be ***empty*** (Section-3.1).

3.4 Classification results

The three classification criteria described are orthogonal to each other. By combining the first two classification criteria one can have macros that are ***complete-dependent, complete-closed, partial-dependent*** and, ***partial-closed***. The ***complete-closed*** macros are easier to replace than those of other categories. In many cases they can be replaced even without looking at their use cases. The macros which are ***complete-dependent***, calls for further analysis (dependency, scope etc.). Based on the analysis results, some of these can also be transformed as described later. The classification of macros based on the classification criteria is presented in Table-3.1

In all the libraries except Cryptopp, majority of macros are complete-closed, meaning it is possible to replace a large percentage of macros from the program. The results in this table also support our claim made earlier in this chapter that majority of the macros have relevant semantic information.

Table 3.1: Classification of macros

			Preserved		Complete	
Package Name	KSLOC (NCNB) [1]	Total Macros	Empty Macros	Partial Macros	Closed	Dependent
Cryotopp-5.6.1	55	1020	164	240	142	474
p7zip-9.20.1	96	1098	284	119	660	35
scintilla-3.0.4	66	2694	52	15	2588	39
poco-1.4.3p1	144	2564	889	305	857	513
facebook-hiphop[2]	687	8159	1099	3092	2822	1146
wxWidgets-2.9.3	741	19262	2483	1923	11223	3633
ACE-6.0.6	151	5969	3227	613	1587	542

[1] NCNB = Non-Comment, Non-Blank (C/C++ files only) [38];
[2] git-5325f9a25ed380f687864f6e1f6d7b88185d108e (Aug-17, 2012)

3.5 Macro classification

In order to provide a one-to-one mapping between macros and C++11 declarations we categorize macros by leveraging classification criteria described above and additionally how those macros might be represented as elements of a C++ program.

To facilitate the mapping of macros to C++11 declarations, we consider the programming elements of C++ abstractly, in terms of IPR [37]. IPR is a complete, efficient, and hierarchical representation of the C++ language. In contrast to a typical compiler's AST, IPR represents only the internal elements of the language, not its external syntax. In IPR, nearly every kind program element is an expression; names, literals, types, and statements are different kinds of expressions. A declaration is a kind of statement, examples of which include classes, functions, and variables.

The reason for using IPR as a reference model in this context is that it provides a framework for classifying different kinds of expressions that might be found in the replacement text of macros. We do not actually use IPR as a physical artifact in our implementation. Our classification of macros with program elements is simply motivated by IPR's design.

For example, the `SUM` macro given as an example previously, (see-3.3.1), can easily be represented as a binary '`+`' expression whose arguments are parenthesis expressions, each of which contains name (also an expression in IPR).

Obviously, there are macros that map to syntax that cannot be represented as a C++ node in IPR; these are the ***partial*** macros described before. Macros whose replacement text includes token pasting, stringification, code snippets, and built-in macros (e.g., `__FILE__`) are not represented in our classification because they are not C++.

We classify the macros into following five different categories.

3.5.1 Empty macro

An empty macro is one whose replacement text is an empty token sequence. For example:

```
#define ASSERT(expr)
```

This `ASSERT` macro expands to an empty sequence of tokens, effectively removing `expr` from the translation. We consider empty macros to be complete, since they can be represented as an empty statement. Obviously, the macros is also closed. While empty macros are generally used for configuration, they are occasionally used to "no-op" expressions for some configurations. Removing assertions for release builds is a common practice.

3.5.2 Expressions alias

An expression alias is a macro whose replacement text can be recognized as a C++ expression (but not a declaration or a type). The macro may be ***dependent*** or ***closed***. The macros with no free variables (i.e., closed) will expand to literals. The notion of literal types is new in C++11 and includes scalar types, reference

types, and classes that can be `constexpr` initialized. For example:

```
//complete-closed
#define PI 3.14
#define SEVEN 3 + 4
#define FILENAME "header.h"

//complete-dependent
#define PRINT printf
#define SUM a + b
#define self this
```

Note that the string `"header.h"` has literal type because its type, `const char*` is considered a scalar type.

The replacement text of `PRINT`, `SUM` and `self` contain free variables, or identifiers that are not declared as macro parameters. Also, note that other classification schemes would have labeled the `self` macro as something other than an expression because it provides a keyword alias [3,25]. However, `this` is also expression, referring to an implicit argument of member functions. The fact that the expression is named by a keyword is immaterial.

One can easily determine if a macro's replacement text has literal type when it is *closed*. To determine whether a ***dependent*** macro expands to literals is non-trivial because of following reasons.

1. It macro may refer to a macro which *generates* tokens.
2. Its dependencies may span across multiple files.
3. It may have cyclic dependencies.

4. The dependencies may refer to program variables in which case it requires traversing the parse tree of the program.

There are often cases, where identifiers referenced in expression aliases are defined by other macros. For example:

```
#define PI 3.14
#define RADIUS 10
#define AREA_CIRCLE PI * RADIUS * RADIUS
```

In this example the macro `AREA_CIRCLE` depends upon macros `PI` and `RADIUS`. This special case is interesting because it is possible to transform these macros with additional information obtained through simple analysis (see Section-4.1).

3.5.3 Type alias

A type alias is an object-like macro whose replacement text can be recognized as a C++ type expressions. For example:

```
#define INT_VEC vector<int>
#define UINT unsigned int
#define UINT_PTR UINT*
```

According to the classification criteria (see-3.3.2), type aliases are ***dependent*** and each of these declarations represents a valid type expression, which can easily be modeled as a C++11 alias declaration.

3.5.4 Parameterized expression

A parameterized expression is a function-like macro that expands to an expression or a statement. The replacement text can be closed or dependent. For example:

```
//complete-closed
#define MIN(A, B) ((A) < (B) ? (A) : (B))
```

```
#define ASSIGN(A, B) { B = A; }


//complete-dependent
#define SUM(A, B, I) ((A)+(B)) + c[I]
#define DIFF(A, B) (MAX((A),(B))-MIN((A),(B)))
```

The `MIN` macro is typical of inline functions written using the C preprocessor. This classification of macros is used by both Ernst et al. [3] and Mennie et al. [25].

The `ASSIGN` macro is somewhat different. Although it is a parameterized compound statement, it does not compute a value. It performs an operation (assignment), resulting in a side effect. This is replaced by an inline function, but one needs to be sure that any such argument `B` is passed by reference which is difficult to implement without extensive front end support from compiler or similar tools.

The `SUM` macro has `c` as a dependent name, which might refer to a non-local variable. The `DIFF` macro has `MIN` and `MAX` as dependent names.

3.5.5 Parameterized type alias

A function-like macro whose replacement text can be recognized as type expression is a parameterized type alias. For example:

```
#define PTR_TYPE(T) T*
#define ValueType(I) typename value_type<I>::type;
```

The last example is taken from *Elements of Programming* where such macros are used to implement type functions related to concepts [39]. This is a case where macros are being used to implement "cutting edge" ideas about principled generic programming. Fortunately, there is an equivalent C++11 declaration (alias templates) that allows us to avoid macros with this style of programming (see Chapter-5).

As with non-parameterized type aliases, dependency is not generally an issue because macros of this sort can be transformed to alias templates, a new feature in C++11.

3.6 Summary

The classification of macros based on the properties of replacement text is important in the current perspective. It gives an insight into what kind of abstraction a macro is meant to provide. This insight helps in identifying appropriate C++ declaration for each macro (that correspond to *complete* C++ program fragments).

We introduced some terminologies related to C preprocessor directives which are be useful in understanding the concepts illustrated. We used two important concepts to arrive at the five categories described above. First, a macro can be object-like or function like. Second, in C++ an expression can be a value-expression or a type-expression. Making ***empty macro*** as a separate category encompasses the trivial case since empty tokens do not construct a valid expression (value or type). The three properties of completeness, dependence and configuration helps in deciding whether a transformation could be possible for a macro definition. We describe the application of our classifications in Chapter-5, the one after the next in which we explain different analyses that should be performed on macros.

4. ANALYSIS OF MACROS

In addition to classification of the macros there are other informations required in case of macros which have dependencies and in case of macros which are defined within conditionals. Both the situations produce a sort of context dependency among macros. Performing a full fledged context sensitive (preprocessor) analysis is non-trivial because it requires tracking all the macro definitions inside the program, in all the included header files, those introduced by the compilers and, those introduced by the build system. However, simple heuristics implemented in our tools give useful results. To serve the purpose we perform following two analysis:

4.1 Dependency analysis

The C preprocessor does not follow the usual "declare-before-use" program structure of C and C++. A macro can be referenced (by another macro) before it is defined. Consider the following program. The `CEL` macro converts a Fahrenheit value to Celsius value.

```
#define SLOPE (5.0 / 9.0)
#define CEL(T) SLOPE * (T - THRESH)
#define THRESH 32.0
```

The macro `THRESH` is referenced before it is defined. The macro is only looked up when an expansion of `CEL` is requested, so it will become a valid use. Contrast that with a corresponding C++ program:

```
double SLOPE = 5.0 / 9.0;
double CEL(double T) {
  return SLOPE * (T - THRESH); //Error!
}
```

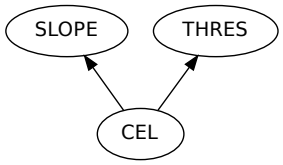

Figure 4.1: Macro dependency graph of the CEL macro.

```
double THRESH = 32.0;
```

In the equivalent C++ program, `THRESH` is not in scope when it is referenced in the body of the `CEL` function.

Solving this refactoring problem turns out to be a straightforward application of dependency analysis [40]. We can construct a directed macro dependency graph in which each vertex represents a defined macro, and an edge (u, v) represents the use of v by the definition of u. The macro dependency graph corresponding to the program above is shown in Figure-4.1. A correct ordering of definitions can be generated by topologically sorting the graph (Algorithm-1).

This is not a hypothetical problem. For example, in the ***wxWidgets-2.9.1*** library, we find instances of macros which are not defined in topological order [41]. For example, in file `include/wx/defs.h` (assuming the current directory is the root directory of ***wxWidgets-2.9.1***)

```
// file: include/wx/defs.h:960
#define wxINT64_MIN (wxLL(-9223372036854775807)-1)

// file: include/wx/defs.h:1054:
#define wxLL(x) wxCONCAT(x, wxLongLongSuffix)
```

To facilitate dependency analysis, we associate each macro identifier with a dependency list that contains all the free variables (Section-3.3.2) present in its replace-

ment text. According to our classification of macros these macros are ***dependent*** (Section-3.3.2). In the previous example the dependency list of `SLOPE` and `THRESH` will be empty while that of `CEL(T)` will have `SLOPE` and `THRESH`. Note that the argument `T` of `CEL(T)` will not be in the dependency list as it is not a free variable. After the dependency list is built, the tool topologically sorts the dependency lists and compare them with the lexical order. If there is a mismatch in the topological order and lexical order between two macro identifiers, it means the macros are defined out of order. In that case they are not considered as potentially refactorable candidates and appropriate warning message is reported.

Continuing the example of the previous set of macros (i.e., `SLOPE`, `CEL`, and `THRESH`), the lexical and topological orders are:

```
//lexical order
SLOPE, CEL, THRESH


//topological orders
CEL, SLOPE, THRESH
CEL, THRESH, SLOPE
```

We index the lexical order and the reverse topological order of the macros so as to compare their relative positions in both the sequences.

```
//indexed lexical order
0 SLOPE
1 CEL
2 THRESH


//indexed reverse topological order
0 SLOPE
```

```
1 THRESH
2 CEL
```

We compare relative positions of macros in both the sequences by comparing their indices. If the lexical order is after the (reverse) topological order i.e, if lexical index of a macro is greater than its reverse topological index then the macro is declared out of order. As we can see, the index of the lexical order macro `THRESH` i.e, 2 is *greater* than its index in the reverse topological order i.e., 1.

We described one possible implementation of finding out the out of order definitions based on the dependency list. It would vary if the dependency list is constructed in a way such that the edges are in opposite direction. A special case where macros are self referenced should be taken care of separately.

The ***cpp2cxx-suggest*** tool performs dependency analysis of macros by constructing the dependency graph where edges point in the direction opposite to the one illustrated here. That approach is efficient because there is no need to reverse the topologically sorted graph. Appropriate warning message is displayed for out-of-order definitions and *all* the inter-dependent macros are preserved. Reordering would not be difficult within the tool, but experience with other tools, such as Rose [42] has been that many programmers strongly object to reordering, so we want to experiment further before fully automating that step. For example, if macro dependencies span multiple files or are lexically distant in the same file, then automated re-orderings make such comparisons difficult to make or reason about.

4.2 Using C preprocessor conditionals to isolate program text

Since conditionals can be used to enclose/isolate a piece of code, having a unified representation of conditionals (and macros) with the language declarations enables analysis of configuration and porting. It is not possible to perform such analysis in

isolation. It also help users isolate program-text within each conditional, one at a time, for analysis. We construct an abstract tree (e.g., Figure-4.2) where its depth is equal to the nesting depth of the conditionals and the breadth at any level is equal to the number of conditionals at that depth. This representation can depict complete translation unit from the perspective of conditionals and help in selectively transforming macros. For example, using this representation, the user can disable demacrofication for all macros within a particular conditional block.

If the following conditionals to be analyzed, our representation of preprocessor conditionals can be depicted as in Figure-4.2.

```
/*...*/
#if defined(MACRO11)
//...
# ifdef MACRO21
//...
# endif
//...
#elif defined(MACRO12)
//...
#endif
//...
#ifndef MACRO13
//...
#endif
/*...*/
```

As we can see in Figure-4.2, the nesting depth is two which is the height of the tree. Also, the breadth of the tree is five at the first level which is equal to the

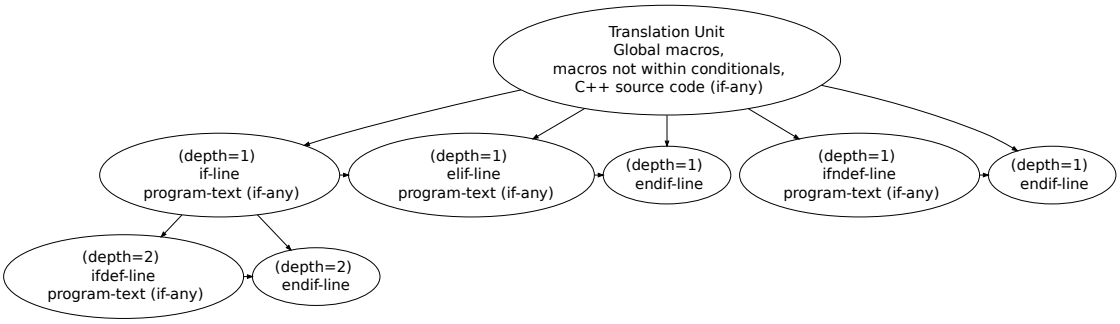

Figure 4.2: Abstract representation of preprocessor conditionals.

number of singly nested conditional directives.

The representation reflects the grammar of preprocessor conditionals [9]. A similar representation was presented by Garrido et al. [10, 13] although theirs serve a different purpose. Our representation is different from theirs in the sense that we assign nodes only to the conditional directives other than the root node. Their representation assigns separate nodes to language declarations. On the other hand, we put the declarations in the nearest preceding (if-block) node. The logic of putting declarations within conditionals in the nearest preceding conditional block is that if the conditional evaluates to true then the code to be analyzed is immediately present, and if the conditional evaluates to false then skipping the node skips the enclosed code as well. In their representation if the conditional evaluates to false, would require traversing an extra node in the tree.

Our representation has a couple of advantages during preprocessor analysis and translation:

1. During traversal of this tree, macros which are configurational can be easily determined and appropriate flag can be set to disqualify them from being transformed in subsequent stages.

2. Suppose there are macros within conditionals used for porting the software

and the user does not want to refactor them or suppose the user is performing source code rejuvenation [1] and wants to incorporate macro translations incrementally (e.g., for separation of concerns); then traversing the tree provides appropriate controls to achieve all these goals.

4.3 Summary

In this chapter we discussed the dependency analysis of dependent macros and the representation of program from the perspective of C preprocessor conditionals. The dependency analysis helps in finding out if a dependent macro could be transformed correctly into corresponding C++ declaration. Its application is made clearer in the next chapter. The abstract tree depicting the preprocessor conditionals are implemented in our framework and can be used to exercise available controls as illustrated before.

5. MAPPING MACROS TO C++ DECLARATIONS

In Chapter-3 we discussed the categorization of macros which was designed to discover how macro-bodies relate to C++ program fragments. We saw how ***complete*** macros relate to complete program fragments (e.g., parameterized/non-parameterized value/type expressions). The results of classification as shown in Table-3.1 also suggest that majority of the macros have relevant semantic information. In Chapter-4, we discussed the technique of identifying out-of-order definitions of dependent macros. Based on the classifications and the results of dependency analysis we can decide whether a macro could be transformed or not. The basic process by which this decision is made is shown in Figure-5.1 [1].

As illustrated in the Figure-5.1, first of all it is checked if a macro is referenced in any conditional directive, in any of the program files of the software. Any macro referenced in conditionals (***configurational*** macros) cannot be transformed as explained previously (Section-3.3.3). Then the replacement text is parsed to check if the macro body is a ***complete*** expression or not. The (expression) parser also collects several information based on a set of heuristics (such as whether a replacement text has constructs that would modify the control flow of the program, whether there are dependencies etc.). Based on the information collected by the parser, the macro is categorized. The macros which do not have a viable categorization according to our classification scheme are preserved. Others are subjected to dependency analysis if they are dependent. The macros which pass the dependency analysis (macros for which dependencies are *in-order*) and the ***closed*** macros are subsequently designated as candidates to be transformed into C++11 declaration.

[1] A step by step algorithm for this decision process is given in Algorithm-2

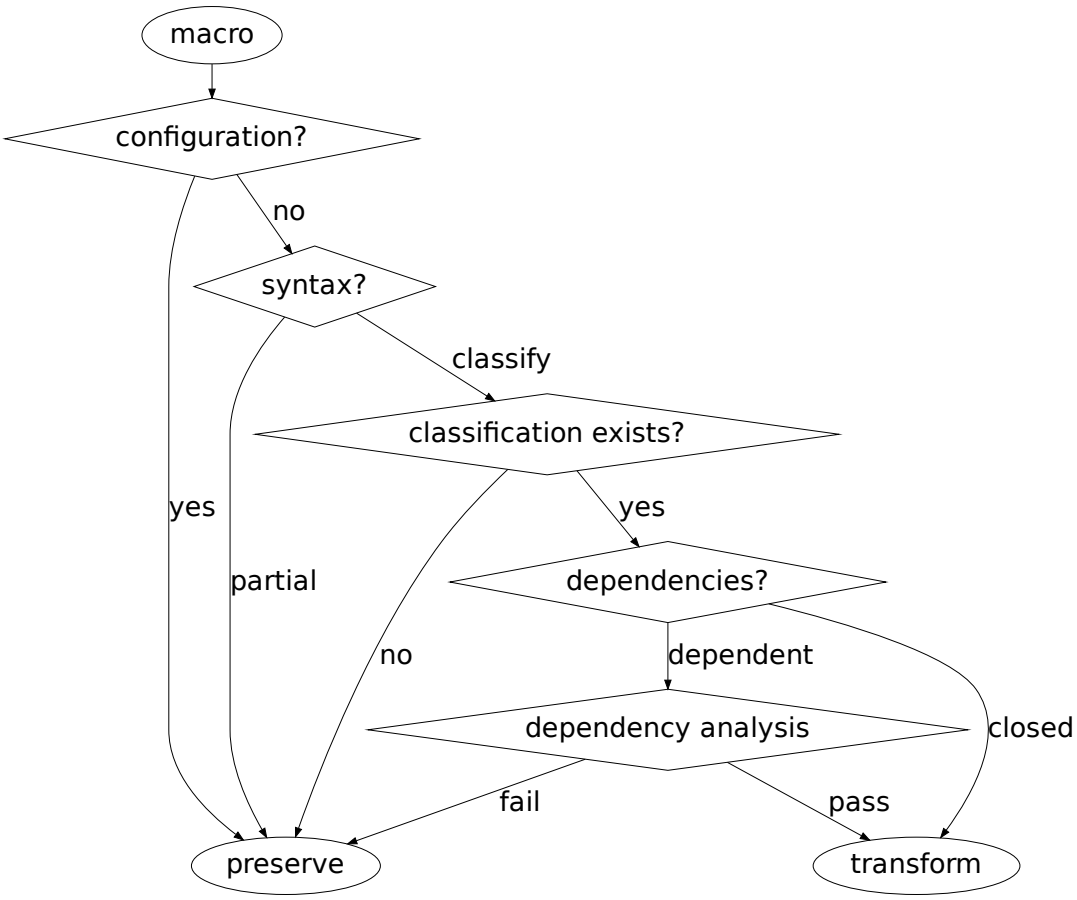

Figure 5.1: The decision process to determine if a macro should be replaced with a C++11 declaration.

From a broad perspective, the replacement text of a macro (which are transformable) will expand to an expression, a statement, or a declaration. Corresponding to each program element the transformations ought to change. Table-5.1 shows possible replacements for each program element. When the replacement text is an expression, we should elevate the macro definition to a variable or a function declaration. Since the macros with replacement text as statement(s) are meant to behave like inline function, we should replace such macros with a (inlined) function declaration. Lastly, if the macro is used for code-generation (i.e., the macro expands to a declaration), a straightforward transformation is non-trivial. Possibly, templates can be of help in some cases (Appendix-A) although this last transformation has not been implemented in the ***cpp2cxx*** framework.

Table 5.1: Possible transformations for different program elements

Macro replacement text	Possible transformation of macro
`//Expression` `#define PI 3.14`	`//Declaration` `constexpr double PI = 3.14;`

`//Statement`
`#define F(A,B) \`
`  if(A > B) A = B`
	`//Declaration`
	`template<typename T>`
	`void F(T& A, const T& B)`
	`{`
	`  if (A>B) A=B;`
	`}`

`//Declaration`
`#define DECL(T) class Node##T { \`
`  T* ptr; \`
`  };`
	`//Templates`
	`template<typename T>`
	`class Node {`
	`  T* ptr;`
	`};`

In this section, we present how different kinds of macros can be replaced with corresponding C++11 declarations.

5.1 Object-like macros

Consider a general object-like macro:

```
#define A X
```

where `X` is a value expression with a type (say `T`). Here, if we can find out the type of the replacement text (i.e., the type of value expression `X` in this case) at compile time, this macro can be directly replaced by an assignment expression in the following manner.

```
T A = X;
```

If the replacement text does not have any dependencies, this transformation would work without making any modifications at the use site. For example, an object-like macro which is ***complete-closed***:

```
#define MAX 10000
```

An equivalent C++ statement that represents similar semantics would be:

```
T MAX = 10000;
```

Where, `T` is the type of literal (`10000`). `T` can be determined in a couple of ways. First, using a program analysis tool we can find the type of literal (as `unsigned int`) and substitute it for `T`. This approach is complicated and requires analyses of use cases as well (if the macro has dependencies). The second approach would be to use the type inference facility of C++11. In C++11, we can deduce the type of a declared variable (lvalue) from its initializer at compile time. In this case the type of the `MAX` will be determined by the compiler based on the type of literal `10000`, if we specify the type of `MAX` as `auto` [9]. So the equivalent C++11 declaration for the macro `MAX` is:

```
auto MAX = 10000;
```

The second approach is simpler to implement than the first one because the compiler deduces the type and we just need to specify the declaration. In this

case one might argue that after this transformation the *variable* `MAX` may become mutable. In order to guarantee that the variable is a compile time constant, we can make the declaration a generalized constant expression by prefixing the `auto` with `constexpr` keyword [9] - another feature introduced in C++11. So the declaration finally becomes:

```
constexpr auto MAX = 10000;
```

It is possible to translate ***complete-closed*** object-like macros, without manual analysis in most of the cases. Transforming ***complete-dependent*** object-like macros requires further analysis. For example:

```
#define KR word64(K)<<32
```

Here, unless and until we deduce the type of `word64(K)`, and `K`, we cannot successfully transform `KR`. Moreover, we have to perform dependency analysis to determine whether the declaration of `word64(K)`, and `K` precede the definition of `KR` or not. If `word64(K)`, and `K` are not macros, then integrating our tool with a compiler can help gain valuable information about them and aid in translation. In a special case where the free variables inside a macro are *also* macros we can proceed with the transformation just after doing dependency analysis (Algorithm-2). For example:

```
#define PI 3.14
//dependent identifier PI is also a macro
#define TWO_PI 2*PI
```

As the macro dependency (`PI`) of `TWO_PI` can be transformed, it means the type of the macro can be determined. And that would be a sufficient condition to proceed with the transformation of `TWO_PI`.

If the replacement text is a statement then the macro behaves like an inlined function and can be replaced with one. However, there are a couple of issues. First,

if the macro is defined inside a function then replacing it with a function declaration would be illegal according to the grammar of the language. Second, if the macro is used to create dynamic scope (the macro references a variable which is defined in a future scope [3], means only finding an equivalent transformation will not solve the complete problem. We must figure out where to place the translated declaration such that the (dependent) variables referenced (if any) by the macro become visible [25]. As a result we have to make appropriate transformations depending upon the use case as described later.

5.2 Function-like macros

To express a function-like macro as a C++ abstraction we have to find if there is an equivalent function declaration for it. The basic approach to transform function-like macros remains similar i.e., how to infer the type of replacement text and then express the function-like macro as a declaration. Consider a general function-like macro:

```
#define F(a1, a2) X
```

where `X` is a *value* expression with a type `T`, and the arguments to the function i.e., `a1` and `a2`, have types say `T1` and `T2` respectively. Then we can write an equivalent C++ function-like this:

```
T F(T1 a1,T2 a2){
  return X;
}
```

If we consider this function declaration as a viable transformation, all we need to do is find out `T`, `T1`, and `T2`. Similar to the approach taken in the transformation of object-like macros, we use C++11 facilities to generate a function declaration such that the unknowns are determined by the compiler itself. Using a function template

can help infer the types of `T1` and `T2`. Because macro arguments are untyped, we must allow (conceivably) expressions of any type to be used as arguments to the function. Templates are capable of providing this level of flexibility.

To determine `T`, we have to use a late-binding syntax of function declaration. Combining both the concepts we have a function declaration with correct semantics.

```
template <typename T1, typename T2>
  inline auto F(T1&& a1, T2&& a2) -> decltype(X) {
    return X;
  }
```

The result of the function uses the new late-binding syntax for function return types. The result type, given as `decltype(X)`, is the deduced type of the expression `X`. Without the ability to deduce the type of the resulting expression, an automated, generic replacement would not be possible. We would have to investigate all possible uses of the macro in order to determine an appropriate result.

The function arguments `a1` and `a2` are passed by *forwarding* into the function body. Forwarding means that the actual type of the instantiated function parameter will be determined by the type of the function argument. Consider a possible use of the replaced function `F`:

```
int x = 0;
F(2, x);
```

Because `a1` and `a2` are passed by forwarding, results of deduced template argument types are modified by the function arguments. The argument type deduced for `T1` will be `int` because `2` is a literal having type `int`. The argument type deduced for `T2` will be `int&` because `x` is an lvalue (it refers to a non-const variable). In other words, the following version of `F` is called:

```
auto F(int a1, int& a2) -> decltype(X)
```

There are two reasons why perfect forwarding is so well-suited to this task. First, we cannot detect the presence of side effects in the replacement text of a macro. If an argument is modified, then it needs to be passed by lvalue reference. Second, we cannot know how macros are used in the program without substantial cooperation from a compiler front end. We do not know if arguments are intended to be passed by value, reference, or constant expression. Perfect forwarding allows us to defer decisions about parameter passing by adapting to the usage at the call site. Consider a simplified use-case of a macro defined in facebook-hiphop library in the file `$PROJECT_HOME/src/runtime/base/macros.h` [43], where `PROJECT_HOME` is the root directory of the library.

```
#include<string>
//macro
//#define NAMSTR(nam, str) nam

//C++11 declaration
template<typename T1, typename T2>
inline auto NAMSTR(T1&& nam, T2&& str) -> decltype(nam)
{ return nam; }

int main()
{
  std::string nam = "name";
  std::string* ptr = &NAMSTR(nam, "SKIP");
  return 0;
}
```

Here, the compiler reports error when perfect forwarding is not used, because taking the address of temporary (the returned variable) is not allowed.

Finally, the use of inlining helps ensure that the performance will be no worse than using the original macro. As a concrete example of the refactoring, consider the archetypal inline function macro:

```
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
```

Using the concepts just described, this macro can be replaced by the following function template.

```
template <typename T1, typename T2>
inline auto MIN(T1&& X, T2&& Y)->decltype(((X) < (Y) ? (X) : (Y)))
{
        return ((X) < (Y) ? (X) : (Y));
}
```

Although an automated replacement can be used exactly like the original, we have introduced some additional complexity into the definition that we would like to remove: the extra parentheses, the deduced result type, the use of perfect forwarding. The ***demacrofier*** working in tandem with a compiler should be capable of fully replacing the macro with a function declared in the traditional style.

There are several things to note in this transformation:

1. The function has been inlined to get the same efficiency as with a macro substitution.

2. The argument of `decltype` is the same as the expression following the `return` keyword. This way it is very simple to carry out the transformation *automatically.*

3. There is a little overhead in terms of lines of code. But that can also be minimized once the tool works in tandem with a compiler or other program analysis front end tools. The compiler, can analyze the transformed declaration to provide the type of the function which can be collected and stored. Then, in the subsequent pass of source code generation, the trailing-return-type style of function definition can be replaced with a traditional-style of function definition.

Clearly, this transformation eliminates one of the macro-pitfalls i.e. duplication of side-effects [35] that has been attributed to these kinds of macros. This transformation has the advantage that we can do this with a tool and, more importantly, in a lot of cases this doesn't require any change in the code at the use site. Consider a different example where the function-like macro has no argument.

```
#define fun() FXY(1,200,3)
```

In this example there are no arguments to the parameterized macro. As a result, the equivalent function declaration will not be templated. The only thing required to be declared is the return type which is evaluated by the compiler while parsing a function declaration with trailing-return type. So the suggested transformation in this case is:

```
inline auto fun()->decltype(FXY(1,200,3))
{
       return FXY(1,200,3);
}
```

Another variation would be when the return type is known and the argument types need to be determined. For a ***complete-closed*** function-like macro which has multiple statements, we have a transformation as a function returning `void`.

```
#define decompose(a0, a1) {\
  int b0, c;   \
  c = a0 ^ a1; \
  b0 = (a0 << 24);  \
  a0 ^= c ^ b0; }
```

Here the parameter `a0` may be passed by reference while `a1` may be passed by value. During transformation we pass both the parameters by rvalue-reference [9]. Passing by rvalue-reference has the advantage that it can capture the temporaries as well. So we transform the above macro into the following function declaration:

```
template<typename T1, typename T2>
void decompose(T1 && a0, T2&& a1) { \
  int b0, c; \
  c = a0 ^ a1; \
  b0 = (a0 << 24); \
  a0 ^= c ^ b0; }
```

This method of transformation is same as the previous one except that now the return type of function definition is known i.e., `void`. Again, perfect forwarding is used to ensure that the parameter passing style matches the one used at the call site.

If the replacement text is dependent, then we must consider the source of the non-local identifiers. For example, consider the `DIFF` macro:

```
#define DIFF(A,B) (MAX((A),(B))-MIN((A),(B)))
```

Here, `MIN` and `MAX` are dependent names. If we know that those names are automatically replaceable macros, then `DIFF` will also be automatically replaceable.

However, if the macro has dependent names that are not macros, then automatic transformation might not be possible depending upon the nature of the dependence.

For example:

```
#define Acc(a, b)
{ a += f(b); }
```

In this example. `a` and `b` are passed as arguments whereas `f` is a dependency. Before doing any modification we will have to ascertain certain facts about `f` and the context in which `Acc` is called. To do so, we would need information from the compiler.

If `f` is a function or class type (i.e., `f(b)` invokes a constructor), then we can proceed in the usual fashion.

```
template <typename T1, typename T2>
void Acc(T1&& a, T2&& b)
{
    a += f(b);
}
```

The signature could be improved by examining the arguments accepted by `f`. For example, if `f` only takes its argument by value, then the function argument `b` could also be passed by value. If `f` is polymorphic, taking `const` and non-`const` arguments, then the forwarding approach is more appropriate.

If `f` is a local function object or lambda expression in the context in which `Acc` is called, then we would have to transform `Acc` into a lambda function or function object.

We can even transform macros where the replacement text is a C++ statement. Such macros behave just like inline functions with return type as `void`. In general, if we assume that the replacement text is a statement, in that case the transformation would be:

```
//object-like
#define A X

//equivalent function declaration
void A()
{ X }

//function-like
#define F(a1, a2) X

//equivalent function declaration
template<typename T1, typename T2>
void F(T1 && a1, T2 && a2)
{ X }
```

Of course, in the case of object-like macros (e.g., `A`) the macro invocation at the use site has to be modified as a function call. The templated function declaration uses the same perfect forwarding as in the previous examples. Since `X` is a statement, we have to parse the replacement text to find out if the statement is already enclosed within braces or not and place extra braces if required.

If the macro is closed, or the macro references global variables (defined before it), this transformation would be correct. Otherwise, if the macro references a variable defined in a different scope, we have to resort to a another transformation described as a special case in the next section.

5.3 Special cases

5.3.1 Function within a function

A possible variation in the declaration of a function-like macro is when it is defined within a function body. For example:

```
template<typename T>
void f(T const& t)
{
  int S[] = {1, 2, 3, 3, 4, 5};
#define S0(X) S[X]
#define SHIFT(X) X<<t
  int x=0, c=1;
  c = S0(x);
  x = SHIFT(c);
}
```

Using the previous technique to transform macros `S0` and `SHIFT` will fail because C++ does not allow a function definition inside of a function. C++11 provides lambda function declaration that can be defined inside the body of a function [9]. However, there are a couple of analyses which should be performed in order to do this transformation.

First, the lambda function declaration should have complete information about the parameters (arguments) as well as the variables used inside its body. The type of the parameters can be determined by using `decltype` (provides automatic type deduction of an expression at compile time [9]) whereas the list of free-variables used inside its body is captured in the closure. However, it is not correct to capture all the parameters by *reference* or by *value* as clarified in the following example. Incorrect

analysis in this stage could result in undefined program behavior.

Second, it is possible that macros are defined *before* the variables they reference in their replacement text. In that case the *placement* of the new declaration has to be decided. One of the methods to determine the location of the transformed declaration has been discussed by Mennie et al. based on finding the least common ancestor of each macro invocation [25]. Here, the placement must respect both the declaration of captured local variables and the arguments used to deduce the type of `X`. Additionally, that if macros are invoked multiple times in the same scope but with different variables, then deducing the type of parameters becomes difficult. After the analysis following transformations are done:

```
template<typename T>
void f(T const& t)
{
//#define S0(X) S[X]
//#define SHIFT(X) X<<t
  int S[] = {1, 2, 3, 3, 4, 5};
  int x=0, c=1;

  auto S0 = [&S](decltype(x) X) { return S[X]; };

  c = S0(x);

  auto SHIFT = [t](decltype(c) X) { return X<<t; };

  x = SHIFT(c);
}
```

The lambda function declaration of macro `S0` has `S` in the closure. The variable `S` should be captured by reference if it is modified inside the body of the lambda function or if we do not want to pass a copy to the lambda function. The declaration deduces the type of parameter `X` by inferring the type of value `x` passed as an argument at the use-site. Similarly, the declaration of `SHIFT` captures `t` by value (since `t` is passed by const-ref, it cannot be captured by reference). As in the previous case the type of parameter `X` is determined by inferring the type of value (`c`) used to invoke the declaration.

The current version of the ***demacrofier*** can only refactor simple use cases like that of macro `S0`. It cannot determine whether to capture the variables by reference or by value. By default, it generates a lambda function which captures all the dependencies by reference. Further, if an argument passed to the lambda function is used as *lvalue* then the argument has to be passed by reference. This has not been implemented and manual assistance is required to generate correct code in this case. It also cannot perform placement analysis and places the generated declaration just before the first macro-invocation.

5.3.2 Macros used to create dynamic scoping

Consider a situation when a ***dependent*** macro is invoked multiple times in a function and the variables referenced in the replacement text of macro are mutated in between points of invocation. This is typical for macros used to create dynamic scope. Consider a simple example to illustrate the idea:

```
int f()
{
  #define TWICE 2*a
  int a = 1;
```

```
  int first = TWICE;
  a++;
  int second = TWICE;
  return 0;
}
```

One possible solution to transmit the changes inside of the transformed declaration is to capture the free variables, in the replacement text, by reference (either as closure or as an argument of function declaration).

```
int f()
{
  //#define TWICE 2*a
  int a = 1;
  auto TWICE = [&a](){ return 2*a; };
  int first = TWICE();
  a++;
  int second = TWICE();
  return 0;
}
```

As we can see, the dependency (i.e., `a`) is captured by reference in the closure. The approach is similar to the previous example except that at the use-site, the (macro) invocation has to be modified as a function call.

5.3.3 Macro referencing variables in a different scope

There are instances of ***dependent*** macros where the free variables used inside the body of macros are member variables of a `class` or `struct`. Consider a simple example to illustrate the idea:

```
class temp{
  public:
  //
    int get_temp_kel();
    int get_temp_fah();
  private:
    int temp_cel;
};

#define TEMP_KEL 273+temp_cel

int temp::get_temp_kel()
{ return TEMP_KEL; }
```

In this case, the transformation will result in a compilation error because the macro `TEMP_KEL` accesses the member variable `temp_cel` of class `temp`. For object-like macros it is easier to put the transformed declaration inside the function body just before the first use site.

```
int temp::get_temp_kel()
{
  //#define TEMP_KEL 273+temp_cel
  constexpr auto TEMP_KEL = 273+temp_cel;
  return TEMP_KEL;
}
```

If the macro is invoked inside multiple functions, using this approach, we will have to put the translated declaration in *all* the functions.

In a slightly different situation when replacement text of *function-like* macro accesses the member variable of a class, we can have two possible solutions. Extending the previous example for a function-like macro, we may have the following situation:

```
#define TEMP_FAH() temp_cel*9/5 + 32


int temp::get_temp_fah()
{ return TEMP_FAH(); }
```

One alternative is to translate the function-like macro as a function-declaration and then make that a member function of the class whose variable(s) it accesses. For example:

```
class temp
{
  public:
  //...
    int get_temp_kel();
    int get_temp_fah();
  private:
    int temp_cel;


  //put this after the member variable declarations
  public:


  auto TEMP_FAH()-> decltype(temp_cel*9/5 + 32)
  { return temp_cel*9/5 + 32; }


};
```

```
//#define TEMP_FAH() temp_cel*9/5 + 32


int temp::get_temp_fah()
{ return TEMP_FAH(); }
```

This approach is elegant but requires processing of multiple files because, generally, C++ programs have declarations in a header (dot H) file and the implementations are in a source (dot C) file. It should be noted that the new function has to be placed lexically *after* all the member variables of the class have been defined, otherwise if a member variable is referenced in the `decltype`, it will result in a compilation error.

Another approach is to translate the function-like macro as a lambda function declaration and then place it inside *all* the functions which invoke it. For example:

```
int temp::get_temp_fah()
{
  auto TEMP_FAH = [&temp_cel](){ return temp_cel*9/5 + 32; };
  return TEMP_FAH();
}
```

In general, this approach would require changes to a single file if all the function declarations which invoke the macro are in one file.

5.3.4 Problems with the const-ness of function

There is a case when a function-like macro is invoked from a const-function [12]. According to our transformation methods, the function-like macro gets transformed into a non-const function. Subsequently, during the compilation we get an error message because the transformed function is not a const-function but is called from within a const-function. If the function-like macro actually could behave like a const-

function then there is no problem and we can (manually) make the transformed declaration a const-function declaration. Otherwise we have to revert back and preserve the macro.

5.4 Examples

Here we will take up a few examples from each kind of macro as per our classification to clarify the approach taken to translate macros into equivalent C++11 declarations. The transformation based on our classification provides a one-to-one mapping between a macro and its corresponding C++11 declaration. This way it helps in automating the task of generating equivalent declarations.

1. Empty Macros: The replacement text of an empty macro expands to empty token. If we know that the macro is non-configurational, we could replace an empty macro with a `void`, nullary function that has no function body.

   ```
   //original
   #define EMPTY
   //replacement
   void EMPTY() { }
   ```

   Unfortunately, this approach requires modification of the use site —the expansion of the macro would need to be modified from `EMPTY` to `EMPTY()`. We could find no situations where applying such a transformation would result in an improvement in the program, so we did not implement it. We only include it here to be thorough.

2. Expression Alias: When the replacement text does not depend upon free variables, i.e., the macro is ***complete-closed*** object-like.

   ```
   //original
   ```

```
#define SEVEN 3+4
//replacement
constexpr auto SEVEN = 3+4;
```

The `constexpr` specifier tells the compiler that the variable `MAX` is required to be evaluated at compile time. In other words, constants replaced by constexpr objects will be "folded" at compile time, just as they would have been if they were integer constant expressions. This is true of any `constexpr`-declared variable; C++11 has a much expanded definition of literal types; it includes floating point values (as shown here), classes that can be `constexpr` initialized, and arrays of such objects.

In many cases, the type of declared variable is obvious, and can be written explicitly; here, an ideal refactoring should use `unsigned int` instead of `auto`. This assignment of concrete type information improves the readers understanding of the program. Because our parser does not currently type-check the expressions that it recognizes, we default to using `auto` as the type specifier for the resulting definitions.

If the replacement text is dependent, then we must analyze the dependencies in order to determine if there is a viable replacement (Algorithm-2).

```
#define R 10
#define PI 3.14
#define AREA_CIRCLE PI * R * R
```

The definition of `AREA_CIRCLE` depends on the identifiers `PI` and `R`. Here, we have seen that both `PI` and `R` are previously defined macros, and we know that both can be replaced by `constexpr` variables. That is a sufficient condition

for automatically replacing `AREA_CIRCLE` with a `constexpr` declaration. For example:

```
constexpr auto R = 10;
constexpr auto PI = 3.14;
constexpr auto AREA_CIRCLE = PI * R * R;
```

Here, the use of `auto` in the definition of `AREA_CIRCE` is warranted. `PI` and `R` have different types, and the result of the expression involving those values is determined by the promotion rules of the C++ language (which it is clearly `double`) in this case.

Note that if the order of definitions was such that the replacement would produce a compiler error, we choose to preserve the original macros. Definition reordering is best done in an assisted manner.

A more difficult problem arises when the dependent identifiers are declared variables and not macros. For example:

```
#define SUM a + b


void summer()
{
  int a = 1, b=2;
  int c = SUM;
  a++;b++;
  int d = SUM;
}
```

This kind of usage of macro to create dynamic scope [3] complicates the analysis, but once we get the information about the context of usage it becomes

easier to transform the macros. There is a viable transformation that can be applied to remove the macro, but its automated application would require information from a C++ front end about the variables referenced by the expanded macro and the relative locations of the variable declarations, the macro definition, and the use of the macro. Knowing all of this, we could replace `SUM` with a lambda function.

```
void summer()
{
  int a = 1, b=2;
  auto SUM = [&a, &b]() { return a + b; };
  int c = SUM();
  a++;b++;
  int d = SUM();
}
```

Here, `SUM` is declared as a lambda function that captures its non-local arguments by reference. Applying the transformation also requires modification at the call site. We need to invoke `SUM` as a function call since it no longer *expands* to a sequence of tokens.

If `SUM` is used in multiple functions, we may not choose this particular replacement since it would lead to unnecessary duplication of code (i.e., clones). We should then choose to refactor the generated lambda expressions as a single function or function object. The ***demacrofier*** is capable of generating the replacement declaration, but it cannot calculate where the declaration should be placed in the resulting code; the tool emits a note giving the transformed macro, and suggesting the context in which it might be placed.

The placement of *lambda function* declaration is crucial in this process. In the above case deciding the location was rather easy because the complete function body is a single scope. In the situations when the macro invocation occurs within nested scope for example, if the macro is referenced in a compound statement (e.g., `if`, `switch`), we will have to apply the concepts suggested by Mennie et al. [25]. As of now the ***demacrofier*** does not support such analysis. So this transformation was done manually.

Consider the following example where transformation is possible, but should be guided by the programmer. Consider the macro definition that refers to a function.

```
#define PRINT printf
```

This macro can be rewritten as a declaration.

```
auto PRINT = printf;
```

An automated transformation is possible only if `printf` is known to name a function declaration or some other global declaration. In other words, the transformation relies on compiler knowledge. The type of the declarator depends on the declaration. While the declaration above is suitable for functions, an alias to a global variable would most likely need a reference. Declarations of this sort are not declared `constexpr` because we do not want them to be evaluated at compile time.

3. Type Alias: We can also substitute type alias macros by equivalent `using` declarations. For example:

```
//original
#define BYTE char
```

```
//replacement
using BYTE = char;
```

We prefer to `using` declarations to `typedef`s because a) the syntax more clearly delineates the new type name from the aliased type expression, and b) `using` declarations can be parameterized over type arguments [9]. We can use that fact to support replace parameterized type aliases and be consistent in the use of declarative style.

Determining whether a macro is a type alias requires a type expression parser which is not yet implemented in the tool.

4. Parametrized Expression: When there are no dependencies the transformation is straightforward for function-like macros.

```
//original
#define DIFF(A,B) ((A)-(B))
//replacement
template<typename T1, typename T2>
auto decltype DIFF(T1 A, T2 B)->decltype(((A)-(B)))
{
  return ((A)-(B));
}
```

When there are dependencies, first dependency analysis is performed. After that, we can replace the function-like macro with a function declaration like we did in the previous example. Performing dependency analysis on free-variables which are not macros is non-trivial. Currently the ***demacrofier*** relies on the validation phase (Chapter-6) to testify the correctness of the following transformation.

```
//original
#define TX(A,B) { B = transform(A); }
//replacement
template<typename T1, typename T2>
void TX(T1 && A, T2 && B)
{
  B = transform(A);
}
```

As explained earlier in special cases (Section-5.3), if a parameterized macro is defined inside of a function then we can replace it with a lambda function.

5. Parametrized Type Alias: The transformation of parameterized type aliases, like their non-parameterized versions, is straightforward.

```
//original
#define Ptr(T) T*


//replacement
template <typename T>
  using Ptr = T*;
```

Here, the macro argument `T` is taken as a template argument in the resulting alias template. The transformation also requires refactoring at the use site of such macros, replacing expressions like `Ptr(int)` with `Ptr<int>`. It is conceivable, that macro arguments in such declarations are used in ways other than type parameters. However, our experiments found no such occurrences. The transformation for parameterized type aliases has not been implemented yet.

5.5 Summary

In this chapter, we illustrated possible transformations of macros. We described how concepts developed in the previous chapters (related to classification and analyses of macros) can be useful in automating the process of demacrofication by providing a one-to-one mapping between different categories of macros and their corresponding C++11 declarations.

In the following chapter we describe the implementation of these concepts and complexity of the demacrofication process.

6. IMPLEMENTATION

The high level process of demacrofication is shown in Figure-1.2. The process begins with an initial version of the source code, which contains macros, and ends with a final version in which macros have been replaced by C++11 declarations. There are three phases of translation:

1. identify the complete set of macros that can feasibly be replaced with C++11 declarations,
2. refine that set to only those transformations that produce valid builds, and
3. produce a final, working version of the program.

The ***cpp2cxx*** framework helps to automate each phase of the demacrofication process (Figure-6.1). First we illustrate the design of the ***cpp2cxx*** framework and then we analyze the overall complexity of the demacrofication process using the tools.

6.1 Design of the cpp2cxx framework

The first step of the demacrofication process i.e., identifying the set of viable replacements, is carried out by the ***cpp2cxx-suggest*** tool. The tool identifies macros that can be replaced and generates a corresponding C++11 declaration.

The second phase is implemented in the ***cpp2cxx-validate*** tool. This program takes the complete set of configured transformations and attempts to determine which subset result in valid transformations. The program searches for a valid configuration by iteratively re-building the software for each computed candidate suggested by the ***cpp2cxx-suggest*** tool (Figure-6.2).

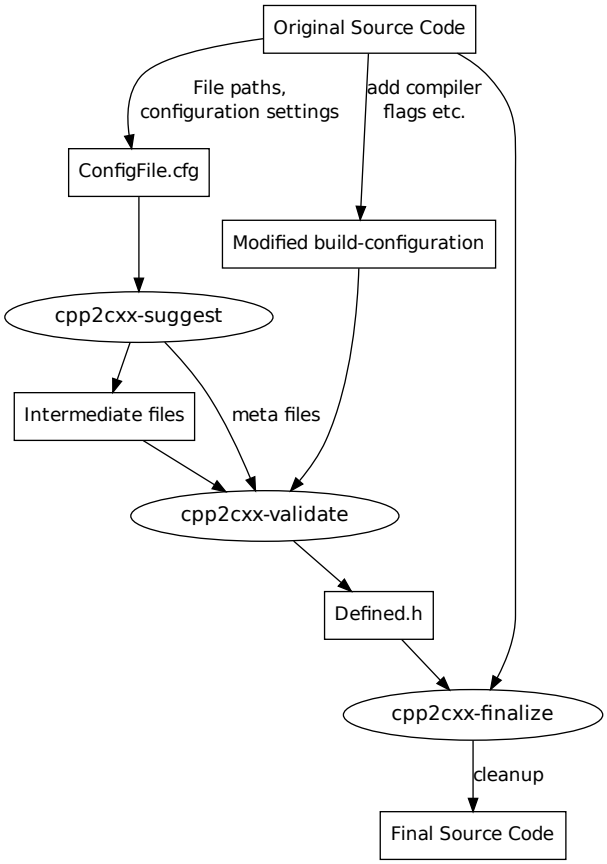

Figure 6.1: The cpp2cxx framework.

The third and final phase is carried out by the ***cpp2cxx-finalize*** program. The program generates the final version of the source code by removing the conditional directives from each transformation, selecting a C++ declaration if the transformation succeeded in the second phase or the original macro if it failed.

Now we illustrate the implementation and functionalities of each tool in the ***cpp2cxx*** framework.

6.1.1 cpp2cxx-suggest

This tool (Figure-6.2) is written in C++ using Boost libraries and [44], the Clang front end [45] and the GNU C++11 compiler [46]. It has been designed to be configurable and can be customized for each software just by modifying the parameters in a configuration file `ConfigFile.cfg`. It contains the names and paths of all the files to be processed, the names and paths of all the files to be generated, and several configuration settings to suit the user's requirements.

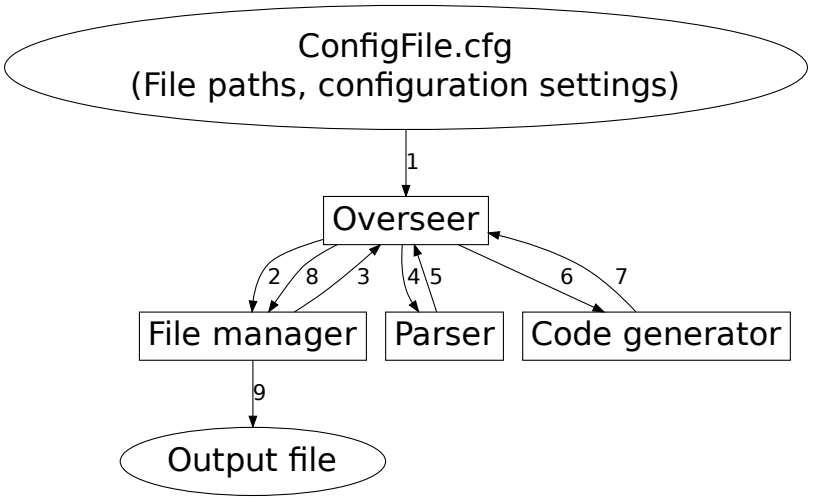

Figure 6.2: The cpp2cxx-suggest framework. The numbers show the sequence in which each module gets involved during the translation of a file.

The output of the ***cpp2cxx-suggest*** tool is an intermediate version of the source

code in which each transformation is guarded by a conditional directive with the help of a "control-switch". For example, if the tool found a macro defining the value of `PI`, the resulting intermediate would contain:

```
//file name : file.cpp
//USE_PI_filecpp_3_9 (unique macro switch for PI)

#if defined(__cplusplus) && defined(__GXX_EXPERIMENTAL_CXX0X__) && \
    defined(USE_PI_filecpp_3_9)

 //C++11 declaration
 auto PI = 3.14;

 #else

 //original macro
 #define PI 3.14

 #endif
```

The flags `__cplusplus` and `__GXX_EXPERIMENTAL_CXX0X__` are used to guarantee that the switch is enabled only when a C++ compiler (g++ in this case) is compiling with support for C++11. This is necessary because we are using the C++11 features.

Note that the set of transformations formulated to carry out the translation works for most macros. However, the method cannot be applied uniformly to all the macros because of three factors. First, The mapping from (untyped) macros to (typed) language declarations cannot be, in principle, guaranteed to be unique.

Second, the limited front-end support from the compilers, and the fact that our tool analyses one file at a time causes some of the translations to fail. Finally, a macro may always have future uses not captured by the tool and not validated by the available set of uses in the program (Figure-1.1).

It generates statistical information about the macros as per our classification criteria 3. It also produces useful warning messages like: macro identifiers with lower case, or with leading underscores, list of macros which uses concatenation/stringification operators. Independently of our transformation tools and plans for automatic code rejuvenation, the messages produced are useful for discovering and manually correcting undesirable behaviors and eliminating maintenance problems. Macro use in large programs is usually ill understood and maintainers therefore often dare not touch macros. By using this tool, *all* uses can be examined.

The tool is composed of four modules (Figure-6.2):

6.1.1.1 Overseer

It is the central point of communication for other modules. It collects configuration settings, and transfers specific settings, and controls to each module of the tool.

6.1.1.2 File manager

It is responsible for input/output (I/O) operations. It associates standard I/O stream (std::istream, std::ostream) objects with source files. These I/O streams are used by the rest of the modules to perform I/O operations.

6.1.1.3 Parser

The Parser module reads a source file, parses it for the C preprocessor directives, analyzes dependencies and collects relevant information for subsequent analyses.

6.1.1.4 Code generator

The Code generator reads the source file and decides whether a macro should be refactored based on the information collected by the Parser. It rewrites the contents of the source file with the macro translations included. In order to facilitate iterative refactoring of macros (by the ***cpp2cxx-validate*** tool), each generated translation is wrapped in a conditional that would allow it to be enabled or disabled.

6.1.2 cpp2cxx-validate

In order to evaluate the correctness of the source to source translation, we designed the ***cpp2cxx-validate*** (Figure-6.3) tool. It is written in Python [47]. It rebuilds the library for each macro, that was translated by the ***cpp2cxx-suggest*** tool. If the build fails for any macro then it removes that macro from the list of macros to be refactored by the ***cpp2cxx-finalize*** tool. The reason for building incrementally is that some macros may impact definitions in many different files. A special header file, `Defined.h` contains the current list of refactorings successfully introduced for each build.

When this tool completes its execution —it can take several hours depending on the number of macros, size of the library, and the computing resource available for this iterative process— the header file `Defined.h` contains the list of all the macros that can be replaced by C++11 declarations.

Before using the tool, the user needs to understand the build system of the software at hand. After that, the compiler flags in the build system must be modified to ensure that the C++ compiler compiles in C++11 mode and it defines the macros in the header file `Defined.h` while compiling. If we take the example of gcc [46], the modification would be like:

```
CXX_FLAGS += -std=c++0x -imacros Defined.h
```

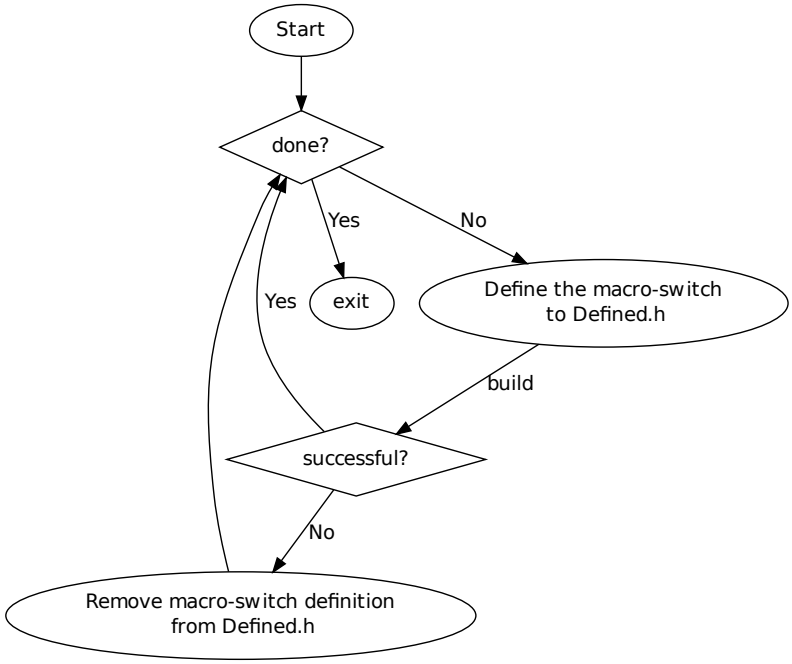

Figure 6.3: The cpp2cxx-validate framework.

By default, the tool issues a single build command (i.e., `make`) to the console; this should be modified depending upon the software and the platform.

6.1.3 cpp2cxx-finalize

Once a set of macros that would preserve the build is generated by the ***cpp2cxx-validate*** tool, changes can be made into the original source code to get rejuvenated program. The***cpp2cxx-finalize*** tool takes original files and the list of macro-switches (from file `Defined.h`), and generates rejuvenated C++11 programs. Currently this tool uses the same back-end as the ***cpp2cxx-suggest*** tool as well as the config-file `ConfigFile.cfg`. More functionalities will be added to this tool by integrating it with the Pivot (a source-code analysis infrastructure [37]).

6.2 Performance measures

Before demacrofying a program, it is important to have an understanding of the total resources that would be required to generate the finally rejuvenated program. A general overview – of the amount of time and space that would be required to complete the demacrofication process – is presented here.

6.2.1 Time complexity

The time complexity of demacrofication depends upon the total number of macros and how macros are referenced in the program.

Each tool in the ***cpp2cxx*** framework handles one macro at a time so, for a given program, the time complexity is directly proportional to the total number of macros.

The way macros are referenced in the program also affects the total time to complete the demacrofication process. As an example consider a macro defined in a header (dot `H`) file and the header file is included by some source (dot `C`) files. Since the validation algorithm (Figure-6.3) iteratively introduces one macro at a time into the program, the build system will compile every source file that includes that header file.

If a macro is included in multiple files then all the files would be build (compile + link) and the time taken will vary accordingly.

If the program is written in such a way that the files have less inter-dependencies (w.r.t. macro definitions and invocations), then for each iteration few files would be compiled and hence, less average time would be taken for translation and validation.

The total time taken while running the ***cpp2cxx*** tools is shown in Table-6.1. Since only four of the seven libraries were finally rejuvenated, the last column does not have entries for the remaining three. From the table we can see that validation is essentially the bottleneck. The time taken to run the ***cpp2cxx-suggest*** and

cpp2cxx-finalize tool increases with the size of the source code. Similar trend is more or less followed while running the ***cpp2cxx-validate*** tool, except for scintilla library where the time taken is same as that of p7zip library even though p7zip is 1.5 times the size of scintilla. We attribute this to a couple of factors. First is the total number of macros. As we can see that scintilla library has around 2.5 times the macros in p7zip library. The second factor is interdependency among macros and program files. In scintilla library there is a file `Scintilla.h` which has more than 900 macros; and that file is included by 111 (out of 128) files. That means each time a macro-switch (for the `Scintilla.h` header) was introduced for validation, *all* the 111 files would be recompiled. As a result, it took around 6 hours to complete the validation.

Table 6.1: Time taken to run tools
(Single thread of execution within Ubuntu-12.04 operating system on Lenovo X220)

Package Name	KSLOC (NCNB) [1]	Total Macros	cpp2cxx-suggest/ cpp2cxx-finalize (approx-minutes)	cpp2cxx-validate[2] (approx-hours)
Cryotopp-5.6.1	55	1020	15	2
p7zip-9.20.1	96	1098	15	6
scintilla-3.0.4	66	2694	15	6
poco-1.4.3p1	144	2564	30	-
facebook-hiphop-php-git	544	4951	60	9*24
wxWidgets-2.9.3	741	19262	120	-
ACE-6.0.6	151	5969	30	-

[1] NCNB = Non-Comment, Non-Blank; [2] Excluding the time to configure the build system

Since the validation requires recompilation of the package (at least of the dependent files) for each transformation, ***cpp2cxx-validate*** tool takes maximum time during the demacrofication. We would like to improve the efficiency of validation phase. This calls for developing better algorithms to introduce suggestions into the system. There are a couple of suggestions that we would like to discuss here.

First, we can make informed decisions to introduce suggestions into the programs.

If we assume that a large number of macros can successfully be transformed, then we can introduce the suggestions in a 'greedy' manner and gradually accumulate the set of successful transformations. So we can introduce all the suggestions, if that fails we introduce half of them leaving the other half, and continue this until we find a set of successful transformations. After that we repeat this algorithm to all the remaining suggestions. Using this approach, the best case (where large number of macros are transformable) will have much faster (logarithmic) convergence. However, the worst case (where all the suggestions are incorrect) will have $2 * n$ (where n = total number of macros) time complexity. As we shall see in the experimental results (Table-7.2 on p. 89), that we can safely assume that most suggestions are correct.

Second, we can improve the worst case of first approach by using the experimental results that (almost) all of the object-like macros will be transformed successfully. So we can apply the first approach on the set of object-like macros to identify the set of successful suggestions. And for remaining function-like macros (which are less than 15%), we can do the linear way because the suggestions of function-like macros have higher failure rate (Section-7.3).

There are other practical issues that should be considered during the process. It is required of the programmer to understand the build system properly; a small mistake in the settings, configuration etc., would generate a wrong code and then everything has to be redone. It happened the first time during the demacrofication of scintilla library, when the Makefile was not configured properly because the `-std=c++0x -imacros Defined.h` flag was not appended to the `CXX_FLAGS` in the Makefile during the validation. Due to this the validation result was 100%. After careful verification, the mistake was found and the experiment was repeated. In that way it took double the time.

6.2.2 Space complexity

The ***cpp2cxx-translate*** tool generates an intermediate file for each program file that is processed. It also generates a few temporary files having one entry per macro. As a result the translation process requires an extra space approximately equal to the total size of the program files (Section-6.1.1).

The validator tool ***cpp2cxx-validate***, copies the translated files into the project directory and starts off with the validation with the help of intermediate files. At the end of the validation, it copies the original files to the project directory (if configured), otherwise the user has to copy the original files to the project directory. So a backup of the program files is also required (Section-6.1.2).

Finally, the ***cpp2cxx-finalize*** tool generates rejuvenated program files corresponding to each file that is processed, in a separate directory. As a result it also requires an extra space approximately equal to the total size of the program files (Section-6.1.1).

Summing the total space requirements, the total space required is approximately equal to three times the size of the source code plus the total size of the object-files that is created during the build process.

6.3 Summary

In this chapter we gave an overview of the ***cpp2cxx*** framework. We have hosted the source files along with user guide online on Github [23]. We also analyzed the complexity involved in the process of demacrofication which is useful in gauging the resources required to carry out source code rejuvenation through demacrofication.

7. RESULTS AND EVALUATION

The framework was used to evaluate the demacrofication process in three different experiments. The first experiment assessed the extent to which a C++ program might be improved through the removal of macros using the transformations described previously. In the second experiment, each macro of an existing library (i.e., Cryotopp) was studied so as to get a better picture of performing source code rejuvenation through demacrofication. The aim of this study was to fully identify use cases for demacrofication and problems demacrofying real-world code. The last experiment was to perform fully automated rejuvenation of two C++ libraries in order to determine the viability and practicality of such a task.

7.1 Automatic demacrofication

In the first experiment the ***cpp2cxx-suggest*** tool was applied to seven C++ libraries comprising over 1.5 million lines of non-comment, non-blank code. The purpose of the experiment was to estimate the number of macros that might possibly be refactored, either automatically or assisted by the programmer. This experiment measures the extent to which the program might be improved (made more secure and maintainable) through demacrofication.

The demacrofier is applied to each file in the package. Files including non-C/non-C++ code (e.g., lex/yacc files) were excluded from the test. The results of the demacrofication *were not compiled* to test if the transformation would preserve the build. This experiment only estimates the extent of transformability.

From the Table-7.1 it is clear that we can remove a significant number of macros which are neither empty nor partial. Hypothetically, all macros in the ***closed*** and ***dependent*** category can be replaced using one of the transformations described in

Table 7.1: Results after running the cpp2cxx-suggest tool

		Preserved		Complete		
Package Name	Total Macros	Empty Macros	Partial Macros	Closed	Dependent	Actually Demacrofied [1]
Cryotopp-5.6.1	1020	164	240	142	474	322 (52.3%)
p7zip-9.20.1	1098	284	119	660	35	656 (94.3%)
scintilla-3.0.4	2694	52	15	2588	39	2595 (98.7%)
poco-1.4.3p1	2564	889	305	857	513	987 (72.0%)
facebook-hiphop	8159	1099	3092	2822	1146	3305 (83.2%)
wxWidgets-2.9.3	19262	2483	1923	11223	3633	12593 (84.7%)
ACE-6.0.6	5969	3227	613	1587	542	1455 (68.3%)

[1] As a percentage of closed + dependent

Chapter-5. The results, however, are not quite 100% for these cases because of a couple of issues. Definition ordering issues prevent automatic refactoring, as do macros that would be refactored as lambda functions. Also, there is no type-expression parser in the current version of the demacrofier which prevented the suggest tool to provide alternatives for ***type-aliases*** and ***parameterized type aliases***.

7.2 Single system case study

Although the classification criteria and demacrofication process are sufficient to identify potentially demacrofiable macros from the syntactical point of view, there are other factors which come to play due to different scoping rules of C preprocessor and C++. It became clear during this experiment when each macro of a library (Cryptopp-5.6.1) was studied in its original form as well as demacrofied form. This experiment helped find out different practical uses of macros. It also helped improve the rejuvenation tools by understanding the interaction between preprocessor and other C++ declarations.

During the study, two different kinds of use cases were found which the tool could not handle. These were later incorporated into the automated demacrofication tool (although with limited support).

7.2.1 Macros as local functions

As discussed during in the Section-5.3, when a function like macro is defined inside a function definition then replacing the macro with a function declaration would result in compiler error, instead a lambda function declaration would be suitable.

There were several instances where function-like macros were being defined inside a function.

7.2.2 Macros involving member variables

There were instances of dependent macros where free variables used inside the replacement text were member variables of a `class` (Section-5.3.3). For example:

```
class temp {
  public:
    int kelvin() const;
    int fahrenheit() const;
  private:
    int cel;
};

#define TEMP_KEL 273 + cel
#define TEMP_FAR cel * 9 / 5 + 32

int temp::kelvin() const
{ return TEMP_KEL; }

int temp:: fahrenheit() const
{ return TEMP_FAR; }
```

Here demacrofication will result in a compilation error because it is not possible to create a global lambda function that captures references to non-static member variables of a class. Even if we could, those members are private. Mennie et al. [25] suggest that the macro replacement be placed inside a function that used the macro:

```
int temp::get_temp_kel()
{
  constexpr auto TEMP_KEL = 273 + cel;
  return TEMP_KEL;
}
```

If the macro is invoked inside multiple functions, this would lead to a replication of the macro replacement. This would be harmful to maintenance, so a better approach would be to introduce a new, `inline` member function with the correct semantics.

Although we found and characterized only two problems of this nature, it is almost certain that there are more. We plan to continue identifying and addressing macro/code usage problems as we build and support our tools.

In addition to these problems, we also found a large number of macros that included control flow statements (`return`, `goto`, etc.), possibly nested to some level. Trying to transform these into functions would break the control flow of the original program.

All told, we were able to refactor about 48.6% of the non-empty macros in the library. This is about half of what was predicted from our initial application of the `cpp2cxx-suggest` tool. We only modified 15 macros; the remainder were not readily replaced with declarations. This took less than a day to complete.

Mileage will vary for each project since every project generally has its own style of macro use. The macros in Crypto++ were largely computational in nature and tended to include program fragments that could not be fully represented as C++

program elements.

7.3 Automatic demacrofication with validation

The third experiment was conducted to evaluate the correctness of the source to source translation. After the demacrofication the libraries cannot be assumed to compile. So the macros which were transformed incorrectly has to be preserved. To validate the transformation the ***cpp2cxx-validate*** tool was designed that would transform a library, one macro at a time, attempting to rebuild the entire system after each transformation. The build process was designed to be incremental because some macros might impact definitions in many different files. The working of the validator script is described in chapter on implementation (Chapter-6).

In order to iteratively refactor macros, each generated macro was wrapped in a conditional (this is done by the ***cpp2cxx-suggest*** tool) that would allow it to be enabled or disabled. For example:

```
// file name is "file.cpp"
#if defined(__cplusplus) && \
    defined(__GXX_EXPERIMENTAL_CXX0X__) && \
    defined(USE_PI_filecpp_3_8)
  auto PI = 3.14;
#else
#define PI 3.14
#endif
```

The unique macro switch for the macro `PI` is `USE_PI_filecpp_3_8`. A special header file, `Defined.h` contains the current list of macros introduced for each build.

Since the validation takes a significant amount of time, the validation was performed only for two libraries (i.e., p7zip and Scintilla). Table-7.2 lists the total

number of macros that were finally introduced into the respective libraries as compared to the total number of macros that were potentially refactorable as explained in Table-7.1

Table 7.2: Results after validation

Package Name	Total Macros	Potentially Refactorable	Finally Validated [1]
Cryptopp-5.6.1	1020	616	179 (29.0%)
p7zip-9.20.1	1098	695	606 (87.2%)
scintilla-3.0.4	2694	2627	2585 (98.4%)
facebook-hiphop	8159	3968	2588 (65.2%)

[1] Percentage listed w.r.t. potentially refactorable macros

From the results in Table-7.2 it is clear that some transformations could not preserve the build. This happened due to various reasons like: the tool works on a per-file basis so its knowledge is limited to single file—it is hard to gauge the impact of a local change to a larger build. Also, it is more likely for function-like macros to fail the transformation than object-like macros. The reason for this is simply due to the complexity involved in parsing the replacement text of function-like macros without extensive support from compiler front end.

Interestingly, the numbers here are much greater than those reported for the manual work on the Crypto++ library. We attribute this to the different styles of macros. Crypto++ has 566 function-like macros and a large number of them are used to generate code and higher order programming. For example, in file `serpentp.h`, 18 macros take function-name as argument like the one given below:

```
#define beforeS0(f) f(0,a,b,c,d,e)
```

Also, out of 566 function-like macros, 468 are dependent, which justifies why it has least conversion ratio.

7.4 Limitations

The process of mapping from (untyped) macros to (typed) language declarations cannot be, in principle, guaranteed to be unique. Although the results are impressive, there are several obstacles to demacrofy 100% macros. Some of them arise due to the difference in the execution model of C/C++ and substitutions of the C preprocessor, such as pass by name semantics for function-like macros, facility of lexical manipulation of tokens, lack of rules for linear ordering between definition and usage, different language scoping rules and lack of strong typing for the macros. Others arise because the tool does not have a type expression parser and freely available compiler front ends do not have a very good support for analyzing macros. Following is a list of known limitations of ***cpp2cxx-suggest*** tool.

7.4.1 Incorrect type inference

The use of `auto` and the `decltype` in ways illustrated might assign type to the macro which may not be what was desired. It might happen that the type inferred by `auto` or `decltype` be the *subtype/supertype* of the intended type. For example:

```
//#define MAX 100


auto MAX = 100;
```

Here the type of `MAX` inferred will be `int` but while using it as a macro the programmer might have intended its use as a unsigned integer `unsigned int`. For example:

```
unsigned int temp = 10;
//comparing unsigned and signed integers
if(temp > MAX)
        temp = MAX;
```

For these kinds of errors, further analysis of variables interacting with the macros at the point of invocation is needed.

7.4.2 Macros defining string literals

As we know that in C++ programs adjacent string literals are concatenated. When a macro is defined as a string literals and it is invoked adjacent to another string literal, both the string literals get concatenated. However, after translation of macro into a declaration, the 'variable' representing the string literal will be of type `const char*` and it won't concatenate with another string literal. For example:

```
//#define CRYPTOPP_BLOCKING_RNG_FILENAME "/dev/srandom"
constexpr auto CRYPTOPP_BLOCKING_RNG_FILENAME = "/dev/srandom";


//usage
m_fd = open(CRYPTOPP_BLOCKING_RNG_FILENAME, O_RDONLY);
if (m_fd == -1)
    throw OS_RNG_Err("open " CRYPTOPP_BLOCKING_RNG_FILENAME); //error!
```

7.4.3 Macros defined from the command line

The tool does not have any idea about the macros which are defined from the command line. Due to lack of information about those kinds of macro there is nothing that the tool can do.

7.4.4 Macros modifying control flow of a program

The macros having `return`, `goto` etc. which modifies the control flow of the program were not considered for transformation in the current implementation. It might be possible to do those transformations when our tool works in tandem with Pivot [37] or other refactoring browsers. For example:

```
#define VALIDATE(X, THRES) if(X>=THRES) return
#define COND_JUMP(X, THRES, LABEL) if(X>=THRES) goto LABEL
```

7.4.5 Replacing partial macros

The macros with operations which do not have any equivalent in C++ e.g., stringification, concatenation and variadic macros could not be replaced. Using template meta-programming methods and other workarounds are useful in some of the cases as illustrated in Appendix-A.

7.4.6 Parsing macro bodies

It is non-trivial to implement a robust type/value expression parser for an industrial strength language like C++. Another problem is that macro-replacement text is just a set of tokens and cannot be just *fetched* to existing compiler-front end like clang, even though it provides a lot of front end support. Due to these limitations a simple value-expression parser was implemented to parse through the macro body to make sure the replacement text is actually an expression. There is a lot of approximation along with several heuristics to collect useful information which would aid subsequent stages of ***cpp2cxx-suggest*** tool in providing better suggestions (Figure-6.2).

7.4.7 Moving the translated declaration in same file

Similar difficulties arise even when the translated declaration is to be moved at a location where all the free-variables referenced in the macro body is *visible*. It involves finding a least common ancestor (LCA) of all the macro invocations and placing the translated declaration at that point. Automating this process would be possible once the ***demacrofier*** is integrated with a compiler front end.

7.4.8 Moving the translated declaration in between files

When a macro body references a member variable of a `class` or a `struct` (Section-5.3.3) there is a possible transformation which requires the translated declaration to be moved to the file where the `class` is declared. Implementing this facility would require the knowledge of multiple files (or a translation unit) along with other details; in fact it would be like implementing a complete front end to support such transformations.

7.4.9 Analysis of dependent macros

Whenever the dependent tokens within the replacement texts are predefined macros, it becomes difficult to perform further analysis in an automatic way. Further, in the case of `partial` macros is is not possible to find any suitable method of demacrofication. For example:

```
#define SYMBOL_EXPORT __attribute__((dllexport))
#define MOVE_NAMESPACE boost::interprocess
```

There are macros which contains `__FILE__` and `__LINE__` macros, which has to be preserved because transforming them would change the abstraction they are meant to provide.

7.4.10 Dependent macros not in topological order

During the dependency analysis of macro bodies, if a set of dependent macros are not in topological order, then all the interdependent macros were preserved without modification. Reordering would not be difficult within the tool, but experience with other tools, such as Rose [42] has been that many programmers strongly object to reordering, that is why this facility has been deferred until later.

7.4.11 Formatting

Although the tool does not play with the formatting of program in general. The tokenizer, `BOOST.Wave`, that does not keep the line-continuation (i.e. backslash newline) at the iterator level. This modifies the formatting of source file at all the places where line-continuation tokens are used. Recently they added the facility to keep this token but the support is only for SLex lexer module [48]. For example, consider the following macro definition:

```
#define Funtion(x) X + \
                   Y
```

Tokenizing the macro definition, and reproducing the macro definition eats up the line-continuation token and results in the following output.

```
#define Funtion(x) X + Y
```

When the rejuvenated code is generated, a macro is replaced with equivalent C++ declaration, which does not have same amount of text (or tokens) as that of macro definition; due to this the line-numbers of the each program fragment would change.

7.4.12 Scope of macros

The scoping rules for macros are different from those of C++ declarations. After demacrofication, the scope of declaration will obey C++ rules and may not be what the user wanted.

Another issue comes when a macro has been `#undef`ined; the macro is not *visible* beyond that point so shouldn't the declaration. But after demacrofication, the scope of the demacrofied-variable may reach beyond the point where it was `#undef`ined. The scope may even end before the point where the macro was `#undef`ined. Programmer

intervention is required in this case.

7.4.13 Nested macros at the use site

Invocation of nested (function-like) macros is ignored due to involved complexity in parsing.

```
#define X 100
#define Y 200
#define F(A,B) ((A)+(B))
#define G(A,B) ((A)-(B))

int z = F(X,Y);
int x = F(10,15);

//the tool logs use case of F, X and Y
int y = F(X+Y,100);
//nested function-like macro invocations are not logged
int m = G(F(X,Y), X);
```

7.5 Summary

In this chapter we presented the experimental results collected as a result of applying ***cpp2cxx*** tools to different C++ libraries. We believe that the results are promising and justifies that this approach is correct and feasible. That said, there is a lot of improvement required in the implementation of ***cpp2cxx*** framework. We will continue to improve upon the concepts implemented in the tools and add new ones once we get proper supports from a compiler front end.

8. CONCLUSION AND FUTURE WORK

We presented several approaches that can be used to demacrofy C++ programs by replacing macro definitions with C++11 declarations. We evaluated aspects of our approach and implementation in a number of different ways. The results of the evaluation convince us that C++ programs can be effectively demacrofied, often using straightforward and simple mapping. The resulting program will have improved behavior and type information (Section-1.1). There are, however, cases where the user is required to make an informed decision about how a macro should be replaced. We believe that the efficacy of the automated approach would be greatly improved if our tools had full knowledge of the C++ program structure through better compiler front end support.

In the future, we plan to further investigate the integration of a compiler front end to supply the information needed to improve our mappings and decision-making capabilities. In particular, the ability to know about C++ declarations will make it possible to automatically refactor a larger class of macros and improve our ability to place the transformed results.

In the performance measures we saw that the bottleneck in the demacrofication process is the validation step (Section-6.1). We would like to improve upon the algorithm which introduces macros during the validation phase so that total time taken could be minimized.

During the transformation of function-like macros, the syntax of generated function declarations (i.e., with trailing return type) is not traditional. As a next step after the demacrofication, we would like to rewrite the syntax of generated declarations to make them idiomatic. This is possible since the declarations are now C++,

their definitions as well as uses can be analyzed with the help of static analysis tools. We plan to integrate the ***cpp2cxx*** framework with IPR to perform such source to source program analyses and transformations [37]. Integrating with IPR would also help in analyzing the macro invocations because, after demacrofication, they turn into function calls.

REFERENCES

[1] P. Pirkelbauer, D. Dechev, and B. Stroustrup, "Source Code Rejuvenation is not Refactoring," *SOFSEM 2010: Theory and Practice of Computer Science*, pp. 639–650, 2010.

[2] B. Stroustrup, *The design and evolution of C++*, vol. 1, Addison-Wesley Reading, Boston, MA, 1994.

[3] M.D. Ernst, G.J. Badros, and D. Notkin, "An empirical analysis of C preprocessor use," *Software Engineering, IEEE Transactions on*, vol. 28, no. 12, pp. 1146–1170, 2002.

[4] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[5] G.J. Badros and D. Notkin, "A framework for preprocessor-aware C source code analyses," *Software Practice and Experience*, vol. 30, no. 8, pp. 907–924, 2000.

[6] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landray, "Google C++ style guide," `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`, 2012.

[7] R. Seacord, *Secure Coding in C and C++*, Addison-Wesley Professional, Boston, MA, 2005.

[8] B. Stroustrup, K. Carroll, and L.M. Aero, "C++ in safety-critical applications: The JSF++ coding standard," `http://www.stroustrup.com/JSF-AV-rules.pdf`, 2005.

[9] C++ Standards Committee, P. Becker, et al., "Programming languages-c++ (final committee draft). C++ standards committee paper wg21/n3092= j16/10-0082," `http://www.open-std.org/jtc1/sc22/wg21/`, 2012.

[10] A. Garrido and R. Johnson, "Challenges of refactoring C programs," in *Proceedings of the international workshop on Principles of software evolution*. ACM, 2002, pp. 6–14.

[11] ASM Sajeev and DA Spuler, "Static detection of preprocessor macro errors in c," Tech. Rep., Technical Report JCU-CS-92/7, Department of Computer Science, James Cook University, 1992.

[12] B. Stroustrup and Safari Tech Books Online, *The C++ programming language*, vol. 3, Addison-Wesley Reading, Boston, MA, 1997.

[13] Alejandra Garrido, "Program refactoring in the presence of preprocessor directives," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2005, AAI3199001.

[14] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *Compiler Construction*. Springer, 2009, pp. 109–125.

[15] M. Vittek, "Refactoring browser with preprocessor," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003, pp. 101–110.

[16] A. Saebjoernsen, L. Jiang, D. Quinlan, and Z. Su, "Static Validation of C Preprocessor Macros," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 149–160.

[17] D. Weise and R. Crew, "Programmable syntax macros," in *ACM SIGPLAN Notices*. ACM, 1993, vol. 28, pp. 156–165.

[18] E.D. Willink and V.B. Muchnick, "Object-oriented preprocessor fit for C++," in *Software, IEE Proceedings-*. IET, 2000, vol. 147, pp. 49–58.

[19] CDT Eclipse, "Eclipse C/C++ Development Tooling-CDT," `http://www.eclipse.org/cdt/`, 2012.

[20] D. Waddington and B. Yao, "High-fidelity C/C++ code transformation," *Science of Computer Programming*, vol. 68, no. 2, pp. 64–78, 2007.

[21] Devexpress, "Refactor for C++," `http://documentation.devexpress.com/#RefactorCPP/CustomDocument2669`, 2012.

[22] ISO, "International standard - iso/iec 14764 ieee std 14764-2006 software engineering #2013; software life cycle processes #2013; maintenance," *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pp. 1–46, 2006.

[23] Aditya Kumar, "cpp2cxx framework," `https://github.com/hiraditya/cpp2cxx/`, 2012.

[24] A. Sutton and J.I. Maletic, "How we manage portability and configuration with the C preprocessor," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 275–284.

[25] C.A. Mennie and C.L.A. Clarke, "Giving meaning to macros," in *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*. IEEE, 2004, pp. 79–85.

[26] D. Spinellis, "Global analysis and transformations in preprocessed languages," *Software Engineering, IEEE Transactions on*, vol. 29, no. 11, pp. 1019–1030, 2003.

[27] B. McCloskey and E. Brewer, "ASTEC: a new approach to refactoring C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 21–30, 2005.

[28] C. Kästner, P.G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. ACM, 2011, pp. 805–824.

[29] L. Vidács, Á. Beszédes, and R. Ferenc, "Macro impact analysis using macro slicing," in *Proceedings of ICSOFT 2007, Second International Conference on Software and Data Technologies*, 2007, pp. 230–235.

[30] László Vidács, "Software maintenance methods for preprocessed languages," Ph.D. dissertation, University of Szeged, Szeged, Hungary, 2009.

[31] C. Riva, M. Przybilski, and K. Koskimies, "Environment for Software Assessment," in *Proceedings of the Workshop on Object-Oriented Technology*. Springer-Verlag, 1999, p. 74.

[32] J.M. Gravley and A. Lakhotia, "Identifying enumeration types modeled with symbolic constants," in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*. IEEE, 1996, pp. 227–236.

[33] G.L. Steele, *Common LISP: the language*, Digital Press, Burlington, MA, 1990.

[34] J.M. Favre, "CPP denotational semantics," in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 2003, pp. 22–31.

[35] R. Stallman and Z. Weinberg, "The C preprocessor. GNU Online documentation," `http://gcc.gnu.org/onlinedocs/gcc-4.6.2/cpp.pdf`, 2012.

[36] Microsoft, "C preprocessor online documentation," `http://msdn.microsoft.com/en-us/library/79yewefw.aspx`, 2012.

[37] B. Stroustrup and G. Dos Reis, "The Pivot: A brief overview," `https://parasol.tamu.edu/pivot`, 2012.

[38] Al Danial, "CLOC," `http://cloc.sourceforge.net`, 2012.

[39] A.A. Stepanov and P. McJones, *Elements of programming*, Addison-Wesley Professional, Boston, MA, 2009.

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, The MIT Press and McGraw-Hill Book Company, New York, 2001.

[41] J. Smart, R. Roebling, et al., "wxWidgets–cross-platform GUI library," `http://www.wxwidgets.org/`, 2012.

[42] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[43] Facebook, "HipHop for PHP," `https://github.com/facebook/hiphop-php/`, 2012.

[44] Boost, "Boost C++ Libraries," `http://www.boost.org/`, 2012.

[45] Chris Lattner, "clang: a C language family frontend for LLVM," http://clang.llvm.org, 2012.

[46] GCC, "GCC, the GNU Compiler Collection," http://gcc.gnu.org, 2012.

[47] J. Python, "Python Programming Language," *Python (programming language) 1 CPython 13 Python Software Foundation 15*, p. 1, 2009.

[48] Hartmut Kaiser, "Boost.Wave V2.3," http://www.boost.org/doc/libs/1_49_0/libs/wave/ChangeLog, 2012.

[49] Daveed Vandevoorde, "Modules in C++ (Revision 2). C++ standards committee paper N1778= 05-0038," http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1778.pdf, 2012.

[50] W. Bright, H. Sutter, and A. Alexandrescu, "Proposal: static if declaration. C++ standards committee paper N3329= 12-0019," http://www.open-std.org/jtc1/sc22/WG21/docs/papers/2012/n3329.pdf, 2012.

APPENDIX A. ALTERNATIVES TO COMMON C PREPROCESSOR IDIOMS

Although macros are deprecated by most coding standards and style guides [2, 6–8], they are used quite often by C++ programmers (see Table-3.1). The fact is that even when C++ provides numerous facilities to avoid macro usage [2], it is not sufficient enough to remove all use cases of C preprocessor macros. Ernst et al. [3] comment that a "disciplined use of the preprocessor can reduce programmer effort and improve portability, performance, or readability". Although performance is not a problem when we are talking about C++, which has several facilities to improve efficiency like function inlining, pass arguments to a function by reference etc., the following facilities provided by the C preprocessor either cannot be expressed in C++ or expressed with complicated workarounds.

A.1 Facility to organize source code in separate files

A C++ program can be organized across separate files using the `#include` directive. However, there is a proposal to have modules in C++ which would have similar facilities [49].

A.2 Code generation

The C preprocessor has a simple semantic model as it is just a textual substitution. Using text based substitution before the compilation phase helps in generating similar declarations. The argument expressions (in function like macros) are not evaluated which can help pass function name, type name etc., as arguments to function like macro. For example:

```
#define LIST_NODE(name, type)\
  struct name {\
```

```
    struct type *data;\
  }\

LIST_NODE(int_struct, int);
LIST_NODE(float_struct, float);
LIST_NODE(char_struct, char);
```

Similar abstractions can be provided using templates.

```
//alternative method
  template<typename type>
    struct list{
      struct type data;
    };

//use cases
list<int*> int_struct;
list<float*> float_struct;
list<char*> char_struct;
```

A.3 Ability to generate tokens

The stringification (#) and the concatenation (##) operators allow programmer to generate tokens which reduces programmers effort in writing repeated code. It also helps in writing good error/warning messages. For example:

```
#define STRUCT_NAME(name) name##2##link

#define ERROR_IF(EXP) \
         do { if (EXP) \
         fprintf (stderr, "Error: \%u," #EXP "\n", __LINE__); } \
```

```
        while (0)\
```

Since C++ does not have these operators, there is no straightforward method to express `STRUCT_NAME`. To replace `ERROR_IF` we can use other methods to handle errors such as `static_assert` for compile time errors and exception handling for run time errors.

A.4 Higher order functional-programming

One can exploit the lexical substitution mechanism of C preprocessor to provide higher-order functional programming abstractions. For example:

```
#define FUN(a,b) a((b)str)


int g(void* a)
{
  //...
  return -1;
}


int f()
{
  char* str = "abcde";
  FUN(g,void*);
}


#define MAKE_MAP(TYPE, RTYPE)\
RTYPE* map_##TYPE (TYPE[n] array, int n, RTYPE(*f)(TYPE)) {\
  RTYPE* result = (RTYPE*)malloc(sizeof(RTYPE)*n);\
  for(int i = 0; i < n; i++) {\
```

```
    result[i]=f(array[i]);\
  }\
  return result;}\
```

Similar abstractions can be provided using templates in the following way.

```
//alternative method
template<typename Ta, typename Tb, typename Tc>
void FUN(Ta a, Tc c)
{
  a((Tb)c);
}

int g(void* a)
{
  //...
  return -1;
}

int f()
{
  char* str = "abcde";
  typedef int(*fp)(void*);
  FUN<fp,void*,char*>(g, str);
}



//alternative method
template<typename TYPE, typename RTYPE, unsigned int n>
```

```
RTYPE* map(TYPE array[], RTYPE(*f)(TYPE))
{
    RTYPE* result = (RTYPE*)malloc(sizeof(RTYPE)*n);
    for(int i = 0; i < n; i++) {
          result[i]=f(array[i]);
    }
    return result;
}
```

A.5 Managing portability and configuration using conditionals

Conditional directives are used to incorporate variable configurations and settings in C++ programs. They are also used to manage portability of programs. Sutton et al. present a detailed analysis of how some of the widely ported C++ libraries manage portability and configuration [24]. There is a proposal to include ***static if*** in C++ [50] which might be used to eliminate this traditional use of C preprocessor.

Algorithm 1: Determine out-of-order macros

1: Generate dependency list for all the macros.
2: Topologically sort the dependency list of all macros.
3: Compare the position of each macro with the position of macros in its dependency list.
4: If the macro references another macro defined later than itself, then mark the macro definition as 'out-of-order'.

Algorithm 2: Decide if a C++11 declaration exists for a macro definition

```
if macro = configurational then
    Preserve;

else

    if macro.syntax = partial then
        Preserve;

    else

        if classification exists then

            if There are dependencies then
                Perform dependency analysis;

                if There are out-of-order dependencies then
                    Preserve;

                else
                    Suggest Transformation;

            else
                Suggest transformation;

        else
            Preserve;
```

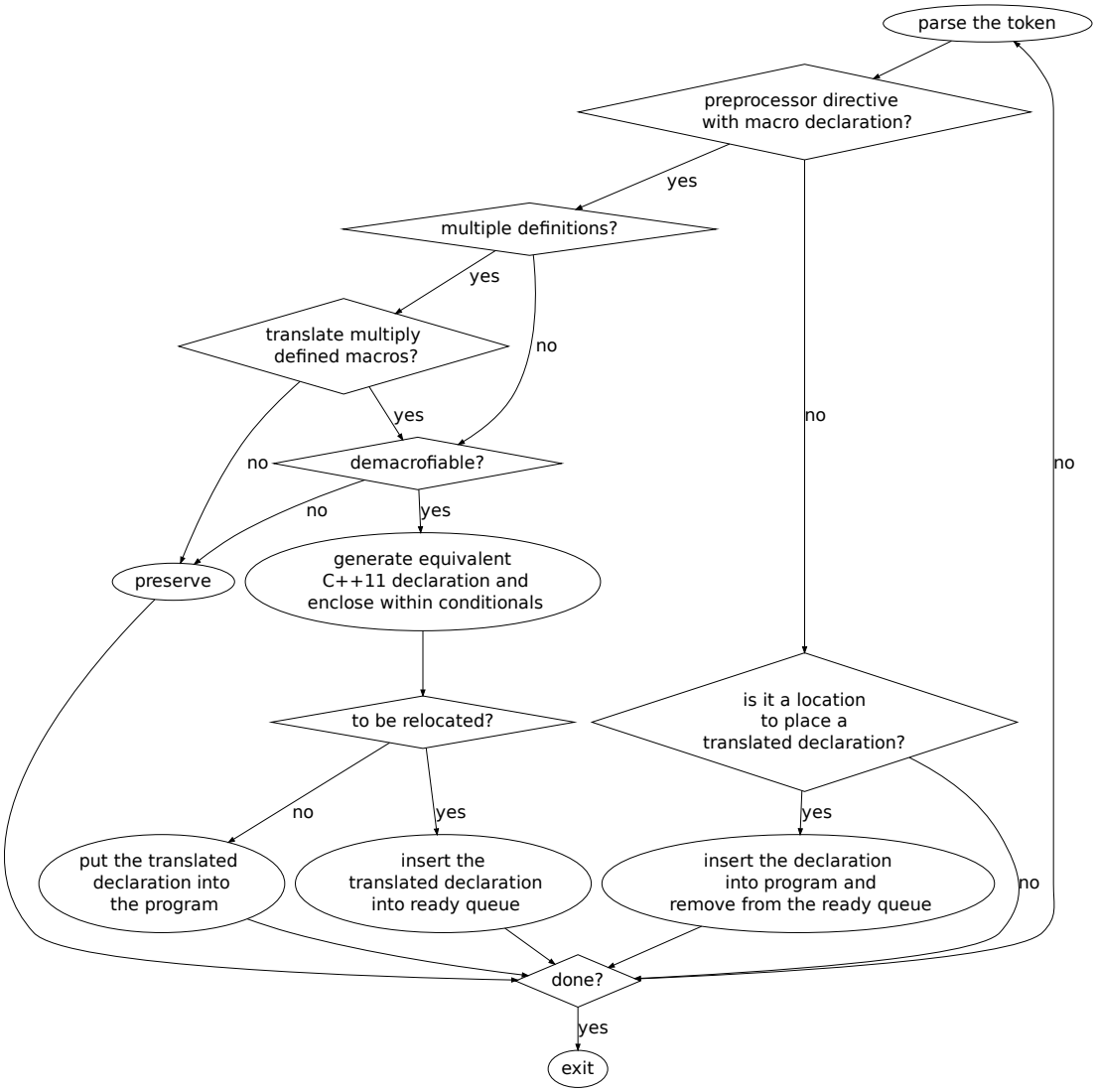

Figure B.1: Algorithm to carry out the translation of macro once relevant information is collected by the parser in the cpp2cxx-suggest tool.