Rejuvenating C++ Programs through Demacrofication

Aditya Kumar
Computer Science Engineering
Texas A&M University
College Station,Texas-77840
Email: hiraditya@neo.tamu.edu

Andrew Sutton
Computer Science Engineering
Texas A&M University
College Station,Texas-77840
Email: asutton@cse.tamu.edu

Bjarne Stroustrup
Computer Science Engineering
Texas A&M University
College Station,Texas-77840
Email: bs@cs.tamu.edu

***Abstract*—We describe how legacy C++ programs can be rejuvenated using C++11 features such as generalized constant expressions, perfect forwarding, and lambda expressions. In general, this work develops a correspondence between different kinds of macros and the C++ declarations to which they should be transformed. We have created a set of *demacrofication* tools to assist a developer in the rejuvenation of C++ programs. To evaluate the work, we have applied the rejuvenation tools to a number of 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. Additional experiments demonstrate why these numbers are not readily achieved using fully automated rejuvenation tools. We also discuss some techniques to further assist in automating rejuvenation process.**

***Index Terms*—source code rejuvenation; macros; C++11; refactoring; demacrofication**

I. INTRODUCTION

As software developers make the decision to migrate C++98 software to C++11 it becomes advantageous to embrace new libraries, idioms, and language features offered by the new language. Using C++11 programming styles 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 work, we are interested in replacing C preprocessor macros with new features and idioms in the C++11 programming language. The kinds of problems engendered by the C preprocessor are many and well known [2], [3]. Because the 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.

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 most "clever" or ad hoc usage of the C preprocessor [2], [4]–[6]. In C++11, the number of reasonable uses of the C preprocessor has been reduced. 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.

Older C++ programs can be rejuvenated by replacing error-prone uses of the C preprocessor with type safe C++11 declarations. In this paper, we present methods and tools to *support* the task of *demacrofying* C++ programs during rejuvenation. This task is intended to be completed with the help of a programmer; it is not a fully automated process.

In particular, we have developed a classification of macros defined in terms of their properties of completeness and dependence and how they correspond to C++11 expressions, statements, and declarations. We have built a set of tools (`cpp2cxx` that assists in macro refactoring (the transformation from macro to declaration), and another tool assists in the iterative application of those refactorings to a software build.

Note that without these new C++11 features, there would be no cause to replace the previous use of macros. The transformations we propose in this paper cannot be achieved using C++98/03. The approach is intended to be applied as developers migrate to the new programming language.

In order to evaluate our approach, we have applied it to the demacrofication of 7 different C++ libraries with over 1.5 million lines of non-comment non- blank code. The evaluation comprised three different studies. We first applied the refactoring tool to each of the libraries in order to identify the kinds of macros used and the extent to which automated transformations could remove replaceable macros. We then conducted a case study applying the process manually as we fully migrated one of those libraries to C++11. Finally, we selected two of the remaining 6 libraries and experimented with fully automating the rejuvenation using our iterative, automated refactoring tools.

II. RELATED WORK

Pirkelbauer et al. describe source code rejuvenation as a "source-to-source transformation that replaces deprecated language features and idioms with modern code" [1]. Unlike refactorings, which are recurring maintenance tasks aimed to improve software design, source code rejuvenation is a one-time transformation caused by a major evolutionary change in the host language or underlying platform. Our goal in this work is to (partially) automate the rejuvenation of legacy C++ programs by replacing some macros with new C++11 declarations.

Ernst et al. studied preprocessor usage in a number of C programs [3]. Most relevant to our work, is their categorization of macro bodies (the replacement text) into 28 different categories, several of which directly correlate to C (and C++11) syntax trees. The results of their study suggest that 42% of macros had replacement text which were constants, 33% were expressions, 5.1% were statements, and 2.1% were types. If one could directly associate each replacement text with a C++11 declaration, then we could expect to deprecate roughly 80% of all macros used in those programs. Obviously, this will not be the case since some macros may reference free variables, contain recursive instantiations, be defined after their first use, etc. Our classification is, in part, informed by the work of Ernst *et al*. However, we do not use their classification outright, ours is primarily based on the relationship between a macro and its corresponding C++11 declaration.

Mennie and Clarke aim to replace a number of macros with equivalent C++ declarations [7]. In his thesis, Mennie describes how simple constants and constant expressions, enumerated constants, and parameterless functions can be migrated to C++ declarations [8]. The approach taken here analyzes macro definitions and their uses within a program, while ours focuses solely on macro definitions. We also benefit from new C++11 language features such as perfect forwarding, which make it possible to generate correct definitions of correct inline functions.

Gravley and Lakhotia also describe a method for replacing sequences of constants (including #define directives) with `enums` [9]. The approach relies on heuristics as "visual cues" to facilitate the grouping of constant expressions into enumerations. A similar application is used to transform sets of static final fields into enumerations in Java but is based on interprocedural type inference instead of heuristics [10]. These approaches focus on specific transformations of macros into declarations, whereas we try to accommodate a range of transformations. The techniques presented in these works also come with an added degree of difficulty since the macros they are transforming attempt to map multiple macro definitions into a single enumeration. We describe a 1-to-1 mapping between macro definitions and their corresponding C++11 declarations.

Most refactoring tools involving the C preprocessor involve its use for source code configuration and portability [11]. For example, Baxter and Mehlich describe automated methods for removing dead preprocessor configurations [12]. Garrido and Johnson study issues related to more general refactorings of C macros and conditionals [13]. The thesis of Vidács addresses similar refactorings at the model level [14] after abstracting from the source code through reverse engineering [15]. These works vary from ours in the domain to which they apply. We do not directly address issues related to macros used to conditionally compile source code since there is no way (in any version of C++) to represent such variations directly in the syntax of the language.

III. REJUVENATING C++ PROGRAMS

The process by which we refactor macros during rejuvenation is not a single- pass operation. The general process of demacrofication is shown in Figure 1. The reason for this is that we have not found any particular set of criteria that would allow us to apply a transformation to all macros in a C++98 program and produce a fully correct, fully functioning C++11 program. There does not seem to be a silver bullet. We have, however, found criteria that allow some transformations to be fully automated and others that require user intervention.

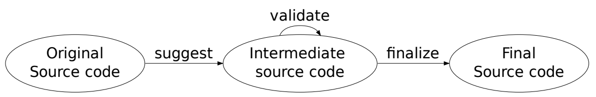

Fig. 1: Demacrofication is a process involving multiple transformations of the source code.

The process begins by suggesting a set of initial refactorings. These initial refactorings are based on a classification of macros in the initial source text and their relation to C++11 declarations (if any). This suggestions might be applied automatically or incrementally by a developer (this is a decision made by the developer). We note that not all automatic refactorings will result in correct programs. The conditions resulting in breakage are discussed throughout this section. Eventually, through iterative analysis and fixing of any broken transformations, the programmer can produce a final, functioning version of the source code.

IV. CLASSIFYING MACROS

The classifications of macros described by Ernst et al. was based on the structures the macros represent in terms of the C language grammar, frequency of usage and the context of usage [3]. Mennie et al. classified the macros based on the styles of macro usage in the software that they used for their case study [7]. As a result, many of the macros in those different classifications have no direct correspondence with C or C++ declarations. For example, there is no way to join two tokens into a single identifier except to use the C Preprocessor.

Because our goal is 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, we found the following three parameters that could serve as the classification criteria for the macros:

A. syntactic structure of the macro bodies,
B. the presence (or absence) of free variables in the replacement text, and
C. the presence of macros within conditionally compiled sections of text.

This classification is defined solely for the purpose of informing our tools about the which macros can be automatically transformed and which might require user intervention.

A. Objects and Functions

A C Preprocessor macro is essentially a named sequence of tokens, called its *replacement text*. Macros come in two flavors: object-like macros and function-like macros [16]. Their definitions have this form:

```
// object-like
#define PI 3.14

// function-like
#define SUM(A, B) ((A)+(B))
```

Here, `PI` is an object-like macro with the replacement text `3.14`, and `SUM` is a function-like macro that will expand to the expression `((A)+(B))`. Whether a macro is object-like or function-like will affect the transformation into a C++ declaration (if any transformations are viable).

B. Completeness

The syntax of a macro's replacement text is helpful in determining whether it can be expressed as declaration in the C++ programming language. Based on this criteria, macros can be classified into two categories. If it is not possible to represent the replacement text of a macro as a C++ expression 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:

```
#define C_MODE_START extern "C" {
#define CONCAT(a,b) a##b
```

Conversely, if the replacement text of a macro can be expressed as a C++ expression we call the macro as ***complete***. For example:

```
#define Z ((X)+(Y))
#define FUN_CALL do { f(); } while(0)
#define TYPE_CHAR (char*)
```

Note that in the case of macro like `FUN_CALL` which emulates a compound statement, we have relaxed 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 [17] represents expressions, ***complete*** macros correspond to complete AST fragments.

C. Dependency

In a macro, we consider an identifier to represent a "free variable" when it is not declared as a parameter of that macro. We say that a macro with any free variables is ***dependent*** since the final valuation of that macro depends on the context in which it is expanded. A macro that contains previously defined macros or even C++ keywords is considered to be dependent. We include keywords because preprocessing may actually redefine keywords. Examples of dependent macros include:

```
#define __GNUG__ (__GNUC__&&__cplusplus)
#define DIFF(A,B) (MAX((A),(B))-MIN((A),(B)))
#define TYPE_CHAR (char*)
```

In the future, we plan to make built-in type names not free variables, meaning that `TYPE_CHAR` would be classified as closed. This will eventually permit us to reason bout about type expressions such as `char*`.

In contrast, a ***closed*** macro does not contain previously (system/user) defined macros or C++ keywords or any other identifier not in its scope. For example, the following macros are considered to be closed.

```
#define ARCHITECTURE 32
#define CONCAT(a,b) a##b
```

The reason that `CONCAT` is closed is that `a` and `b` are local to the scope of the macro: they are parameters.

D. Configuration

Many macros are used to control the conditional compilation (and occasionally inclusion) of C++ files. 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) [11]. We say that any macro appearing in a conditional or include directive is ***configurational***. For example:

```
#ifdef BOOST_NO_NOEXCEPT
...
#endif
```

The macro `BOOST_NOEXCEPT` is 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 the build.

A macro that is not configurational is ***non-configurational***.

E. Definition Order

The C Preprocessor does not follow the usual "declare-before-use" program structure of C and C++. A macro can be referenced any time before it is defined, and expansions of undefined macros simply result in an empty sequence of replacement tokens. 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!
}
double THRESH = 32.0;
```

Obviously, `THRESH` is not in scope when it is referenced in the body of the `CEL` function. This is not a hypothetical

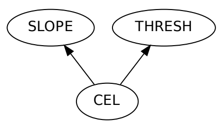

Fig. 2: The CEL macro depends on SLOPE and THRESH.

problem; we found instances of macro ordering problems when conducting our experiments (e.g., wxWidgets-2.9.3 has several occurrences of this exact problem).

Solving this refactoring problem turns out to be a straightforward application of dependency analysis. 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 2. A correct ordering of definitions can be generated by topologically sorting the graph.

However, one does not simply reorder definitions in a program. Experience shows that programmers frequently reject nonlocal transformations of their source code; they want to compare the original source with the improved source on a line-for-line basis. If macro dependencies span multiple files or are lexically distant in the same file, then automated reorderings make such comparisons difficult to make or reason about.

V. MAPPING MACROS TO C++

We classified the macros across three different categories based upon the properties described above. Having done that, we can now define a partial mapping from macros to C++11 declarations. *This mapping is undefined for any macro that is **partial** or **configurational**.*

To construct the mapping of macros to C++11 declarations, we consider the programming elements of C++ abstractly, in terms of IPR [17]. 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.

In the following sections, we describe how complete, non-configurational macros are mapped to C++11 declarations. The dependency of macros, whether or not they contain free variables, plays a significant role in this mapping. In some cases, dependent macros may not have a defined mapping to a C++11 declaration.

The basis for this the mapping is derived from the function-like or object-like nature of the macro and the classification of its replacement text as an expression, statement, declaration, or type. In this discussion we assume that complete statements can be fully parsed, resulting in a hypothetical AST and allowing us to reason about the kind of program fragments represented by replacement text. Our implementation of this assumption partially parses the expression, and makes the judgments heuristically Essentially, it is a best-effort attempt to characterize a sequence of tokens as a C++ expression.

In subsection, we present examples of macros that can be transformed and give the general mapping for those macros. We discuss issues related dependent macros and when mappings can be defined. We also present mappings that require "perfect knowledge" from a compiler front-end and how that information can be used to resolve some cases.

A. Expression Alias

An expression alias is a macro whose replacement text can be recognized as a C++ expression (but not a statement or a type). The macro may be dependent or not.

If the expression parsed from the replacement text has *literal type*, then the macro can be replaced with a constant expression declaration. By By literal type we mean that the type of the expression is the same as that of a literal values (e.g., 3 is an `int` literal, `3.14` is a double literal, etc.). For example:

```
#define PI 3.14
#define SEVEN 3 + 4
#define FILENAME "header.h"
```

Note that the string `"header.h"` has literal type because its type, `const char*`, is considered as such when referring to a string literal. The rule for transforming for these macros whose replacement text is an expression with literal type is:

```
// macro
#define A X

// C++11 declaration
constexpr auto A = X;
```

The type of the declared variable `A` is `auto`; it is deduced from the type of the initializer expression `X`. The examples above would be mapped to:

```
constexpr auto PI = 3.14;
constexpr auto SEVEN = 3 + 4;
constexpr auto FILENAME = "header.h";
```

A more advanced transformation, one that can fully type-check a C++ expression, could automatically assign a concrete type to the declaration of `A`. While our implementation does not perform this transformation (we do not do full type checking), we do want to highlight the fact that concrete types can be deduced from the type of literal expressions.

It is not always possible to guarantee a correct transformation when the macro is dependent since the scope of the free variable may differ from that of the macro's definition. However, if we determine that the free variables are previously defined macros that are mapped to constant expression declarations, then the transformation can proceed in the manner given above. Consider:

```
#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 variable declarations. That is a sufficient condition for automatically replacing AREA_CIRCLE with a constexpr declaration. The resulting program is:

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

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.

In all other cases, transformations must be considered carefully. Here, we give some examples where transformations are 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 to a reference. Declarations of this sort are not declared constexpr because we do not want them to be evaluated at compile time.

Consider a the following use of an expression alias macro.

```
#define SUM a + b

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

The dependent macro refers to identifiers that are actually variable declarations in a future scope. There is a possible 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, the corresponding transformation is to replace SUM with a lambda function declared in the same scope as the local variables:

```
void summer()
{
 int a = 1, b=2;
 auto SUM = [&a, &b]() { return a + b; };
 int c = 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 since it no longer expands to a sequence of tokens.

Obviously, this is a complex transformation, and one that we have not fully implemented. Our demacrofication tool is capable of generating the replacement declaration, but we have not yet calculated where the it 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. This allows the user to perform the transformation manually with assistance from our tools.

B. 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*
```

We generally consider type aliases to be dependent even though type expressions like unsigned int obviously refer to a built-in types and could be considered closed. The distinction matters little for this set of macros. Dependent or not, each of these declarations represents a valid type expression, which can easily be modeled as a C++11 alias declaration. The general transformation for a type alias is:

```
// macro
#define A T

// C++11 declaration
using A = T;
```

Here, A is declared as an alias to the type expression T. We prefer to using declarations to typedefs 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 [18], a fact that we use to transform function-like macros that define parameterized type expressions (Section V-D).

This, the declarations corresponding to the macros above are:

```
using INT_VEC = vector<int>;
using UINT = unsigned int;
using UINT_PTR = UINT*;
```

Note that we have not implemented this transformation since recognizing type expressions is not straightforward. We include the mapping for completeness and reserve the implementation for future work.

C. Parameterized Expression

A parameterized expression is a function-like macro that expands to an expression or a statement. For example:

```
#define MIN(A, B) ((A) < (B) ? (A) : (B))
#define ASSIGN(A, B) { B = A; }
```

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 and Clarke [7].

The `ASSIGN` macro is different. Although it is a parameterized compound statement, it does not compute a value. It performs an operation resulting in a side effect. This can be still be implemented as an inline function, but we need to be sure that any such argument `B` is passed by reference.

The mapping for the *closed* macros like `MIN` whose replacement text is an expression (but not a statement) is:

```
// macro
#define F(A1, ..., An) X

// C++11 declaration
template <typename T1, ..., typename Tn>
auto F(T1&& A1, ..., Tn&& An)
  -> decltype(X)
{
  return decltype(X);
}
```

Here, `F` is a function-like macro with n arguments and replacement text `X`. Because macro arguments are untyped, we must allow expressions of any conceivable type to be used as arguments to the function. Templates are capable of providing this level of flexibility. As such, the resulting C++11 declarations is a function template with n distinct template (type) parameters ($\mathtt{T}_i$) corresponding to each of the n function parameters ($\mathtt{A}_i$).

The result type of the function uses the new late-binding syntax for function return types. The result type, written 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, if one existed.

The function parameters 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. Of all new C++11 features, perfect forwarding is the least obvious and least well understood. However, it suits this application perfectly. Consider a possible use of the replaced function `F` (assuming `F` takes only 2 arguments).

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

Because the arguments are forwarded, the deduced template argument types will preserve qualifiers on the type arguments. The templat argument type deduced for `T1` will be `int` because `2` is a literal having type `int` (there are no qualifiers). The type deduced for `T2` will be `int&` because `x` is an lvalue (it refers to a non-const variable). In other words, the specialization of `F` that is ultimately called after instantiation will be:

```
auto F<int, int&>(int A1, int& A2)
  -> decltype(X)
```

The exact mapping for the `MIN` macro is:

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

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.

The mapping for *closed* macros like `ASSIGN` whose replacement text is classified as a statement is:

```
// macro
#define F(A1, ..., An) S

// C++11 declaration
template <typename T1, ..., typename Tn>
void F(T1&& A1, ..., Tn&& An)
{
  S;
}
```

Because a statement has no result type we can omit the deduced result type specification and the `return` statement. We simply construct a `void` function with the same pattern of template and function parameters above. The final mapping of the `ASSIGN` algorithm is:

```
template <typename T1, typename T2>
inline void ASSIGN(T1&& A, T2&& B) {
  B = A;
}
```

This illustrates another reason 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 do not know how macros are used in the program without substantial cooperation from the 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.

Finally, the use of inlining in both mappings helps ensure that the performance will be no worse than using the original macro.

A parameterized expression can be *dependent* (having free variables). In these cases, as with expression aliases, a transformation must consider the properties of the free variables in the expression. Consider the following:

```
#define OP(A, B) (X +
    MAX((A),(B))-MIN((A),(B)))
```

Here, `X`, `MIN` and `MAX` are free variables. If we know that `X` is an expression alias mapped to a declaration, and we know that `MIN` and `MAX` are parameterized expressions mapped to function templates, then the mapping is straightforward:

```
template <typename T1, typename T2>
inline auto OP(T1&& A, T1&& B)
 -> decltype((X + MAX((A),(B))-MIN((A),(B))))
{
 return (X + MAX((A),(B))-MIN((A),(B)));
}
```

This is similar to the analysis applied to dependent expression aliases. When free variables are not macros, then we must rely on information from the compiler. Consider a 2nd example:

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

Before doing any modification we will have to ascertain certain facts about `f` and the context in which `Acc` is called. 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>
inline void Acc(T1&& a, T2&& b) {
 a += f(b);
}
```

We note that 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. The mechanism for doing so, and the problems inherent in the transformation, are similar to those involved in the transformation of expression aliases involving lambda function transformations.

In either case, the requirement for compiler knowledge places these transformations out of the scope of a straightforward, fully automated transformation. We defer to the user in these cases.

D. 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;
```

We note that this last example is taken from *Elements of Programming* where such macros are used to implement type functions related to concepts [19]. This is one 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.

As with non-parameterized type aliases, the general transformation is achieved using C++11 alias declarations.

```
// macro
#define A(T1, ..., Tn) X

// C++11 declaration
template <typename T1, ..., typename Tn>
using A = X;
```

The macro parameters T_i are mapped to corresponding template parameters of an alias declaration. No additional substitution is needed for `X` since the replacement text is written as the resulting type. The declarations corresponding to macros presented initially are:

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

template <typename T>
using ValueType =
 typename value_type<T>::type;
```

As with type aliases, dependent names in parameterized type aliases are to be expected. However, because their resolution requires knowledge from the C++ compiler, we have not implemented their automated transformation. We include them here because we believe them to be an important feature of demacrofication and intend to support their transformation fully in the future.

VI. IMPLEMENTATION

Our implementation is composed of three separate tools that support the different stages of demacrofication shown in Figure 1. The `cpp2cxx-suggest` implements a straightforward translation the macros into equivalent C++ according to the criteria described in the previous sections. When automated transformations are not possible or obvious, suggestions to the user. The result of the application of this tool is a collection of intermediate source code and configuration files.

The `cpp2cxx-validate` program iteratively manipulates suggested transformations to find the largest possible set that results in a functioning build. The `cpp2cxx-finalize` program strips intermediate configuration and modifications to the source code and generates the final build. These two tools are discussed in more detail in Section VII-C.

The `cpp2cxx-suggest` program implements the initial phases of demacrofication. The basic process by which demacrofication happens is shown in Figure 3.

The goal of this process is to determine if there is a C++11 declaration that can be used to replace the definition of a macro. Ultimately, we must make a decision to preserve a macro in the original program or replace it.

The program iteratively considers macros in the order in which they appear in the source text. Configurational macros, and those whose replacement texts are partial syntax trees are preserved (i.e., no transformation is applied).

The classification step attempts to determine if the replacement text of the macro corresponds to a complete C++11

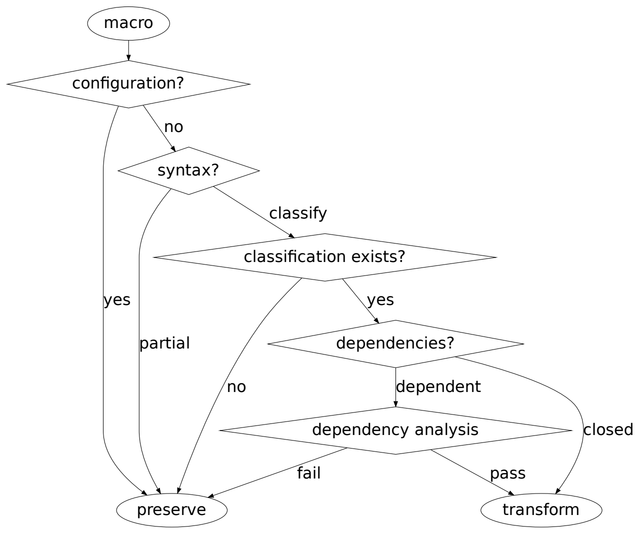

Fig. 3: The demacrofication process is a decision procedure that determines if a macro can be replaced with a C++11 declaration.

expression, statement, declaration, or type. The current implementation is a best-effort attempt to parse the sequence of tokens in the replacement text in the absence of any C++ information. Because it operates on partial information, it is not as accurate as we would like.

If the macro is dependent, the tool performs dependency analysis to determine if a mapping exists. If the dependencies can be resolved (i.e., finding previously defined macros that map to declarations), the the transformation can be applied. Otherwise, the macro is preserved, although the tool will emit a suggestion for the possible transformation.

VII. EVALUATION

Our evaluation consists of three distinct activities.

- First, we applied the `cpp2cxx-suggest` tool to a 7 different C++ libraries in order to ascertain the extent to which those libraries might be improved through demacrofication.
- Second, we selected one of those libraries and migrated it to C++11, acting as the user of our demacrofication tools. The goal of this study was to understand, in detail, the impact of our automated refactoring on a library and the amount of effort required to complete the process.
- Finally, we selected two of the remaining libraries as targets for our automated validation method. The goal of this experiment was to determine the extent to which we could reduce the amount of user intervention in the demacrofication process.

We describe each activity separately.

A. Automatic Demacrofication

In the first experiment we applied our demacrofication tool to seven C++ libraries comprising over 1.5 million lines of non-comment, non-blank code. The purpose of the experiment is to estimate the number of macros that might possibly be refactored, either automatically or assisted by the programmer. In other words, we are measuring the extent to which the program might be improved (made more maintainable) through demacrofication.

The `cpp2cxx-suggest` program is applied to each source code file in a library. Files including non-C/non-C++ code (e.g., lex/yacc files) were excluded from the test. The test was configured such that every transformation that could be identified was applied. The results of the demacrofication *were not compiled* to test if the automated transformation would preserve the build; we knew in advance that a naiv̈e application of the tool would most likely result in a broken build. We are only interested in the extent to which macros might be removed from a library.

From the Table-I 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 Section-VI. 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, we have not yet implemented support for demacrofying type aliases.

B. Case Study

In this experiment we took one library (Crypto++-5.6.1) and migrated it use the C++11 programming language. We used the `cpp2cxx-suggest` tool to suggest and apply an initial set of transformations. When the transformation was not obvious or would break the build, we intervened and applied a correct transformation by hand.

There were three primary goals of the study. First, we wanted to identify the causes of compilation errors resulting from automatic transformation. Second, we want to analyze those errors to determine how we might improve our demacrofication tools by understanding the interaction between preprocessor and source code declarations. Third, we wanted to gauge the amount of effort required to fully demacrofy a library.

As a result of the study, we found three situations where we were able to correct the demacrofication manually. These were later incorporated into the automated demacrofication tool and implemented to produce suggestions (not automatic transformations).

We found two instances where function-like macros were being defined inside a function.

```
void f() {
#define S0(X) S[X]
   int x = 10;
   x += S0(x);
}
```

If we simply generate a new function definition, we will produce an obviously incorrect program; C++ does not permit function declarations inside other functions. However, the macro can be refactored the macro as a lambda function. To

TABLE I: Results before validation

			Preserved		Complete		
Package Name	KSLOC (NCNB) [1]	Total Macros	Empty Macros	Partial Macros	Closed	Dependent	Actually Demacrofied [2]
Cryoto++-5.6.1	55	1020	164	373	300	183	457 (94.6%)
p7zip-9.20.1	96	1098	284	119	660	35	656 (94.3%)
scintilla-3.0.4	66	2694	52	15	2588	39	2595 (98.7%)
poco-1.4.3p1	144	2564	889	305	857	513	987 (72.0%)
facebook-hiphop-php-git	544	4951	1033	934	2538	446	2591 (86.8%)
wxWidgets-2.9.3	741	19262	2483	1923	11223	3633	12593 (84.7%)
ACE-6.0.6	151	5969	3227	613	1587	542	1455 (68.3%)

[1] NCNB = Non-Comment, Non-Blank
[2] As a percentage of closed + dependent

do so, we need to know the types of all arguments passed to the function, and where to place the resulting declaration. For example:

```
void f() {
  int x = 10;
  auto S0 = [&S](decltype(x) X)
    { return S[X]; };
  x += S0(x);
}
```

Here, the non-local `S` is captured by reference, and the type of the function argument is given as `decltype(x)`. This is effectively the same as forwarding `x` in a function template. Determining where the declaration should be placed is not straightforward. One method has been discussed by Mennie and Clarke [7] based on the least common ancestor of each macro use. Here, however, the placement must respect both the declaration of captured local variables and the arguments used to deduce the type of `X`. Additionally, that if `S0` is invoked multiple times in the same scope but with different variables, then deducing the argument type of `X` is made much harder, but is still possible. Our modified implementation will suggest the refactoring for the lambda function but not the placement of the declaration.

There are instances of dependent macros where free variables used inside the replacement text are member variables of a `class`. 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. [7] suggest that the macro replacement be placed inside a function that used the macro:

```
int temp::get_temp_kel()
{
  constexpr auto TEMP_KEL = 273 + temp_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.

We modified our tool to recognize these cases and emit the different possible transformations as suggestions to the user.

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 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 macros in the library. This is about half of what was predicted from our initial application of the `cpp2cxx-suggest` program. 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. This will not be true for all such libraries.

C. Automatic Demacrofication with Validation

The third experiment was conducted to evaluate the extent to which we could automatically generate a functioning build. After initially demacrofying a library, it might not compile due to some of the problems discussed above. We want to revert the changes to the macros which were transformed incorrectly. To do so, we built a tool `cpp2cxx-validate` that would apply suggested transformations one macro at a time, attempting to rebuild and retest the entire system in each iteration. The reason for building incrementally is that some macros may impact definitions in many different files.

In order to iteratively refactor macros, we wrap each generated macro in a condition that would allow it to be enabled or disabled. For example:

```
// file name is "file.cpp"
#if 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`. Technically, the macro should only be defined for C++11, but we omit those conditions here. A special header file, `DefinedIncludeGuards.h` contains the current list of macros introduced for each build.

A build automation script that does the following:

1) Use the 'demacrofier' to transform the program, wrapping each new declaration in the style above.
2) Create an empty file, `DefinedIncludeGuards.h`.
3) For each suggested transformation,
   a) add its macro switch to the include guard file
   b) build the library and run the test suite
   c) if the build or testing fails, remove the switch from the guard file.

When the program finally terminates—it can take many hours depending on the number of macros and size of the library—the library will be fully transformed and compiled. The order in which macro-switches were included was the same as the order the macros were demacrofied. This simple approach was taken to avoid complexity in determining the best order in which demacrofied constructs should be introduced into the program.

We applied this process to two libraries: p7zip and Scintilla. Table-II 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-I

TABLE II: Results after validation

Package Name	Total Macros	Potentially Refactorable	Finally Validated [1]
p7zip-9.20.1	1098	695	606 (87.2%)
scintilla-3.0.4	2694	2627	2585 (98.4%)

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

From the results in Table-II it is clear that some transformations could not preserve the build. This happened due to various reasons, some of which have been described above. 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 macro use in the libraries. The style of macro use in p7zip and scintilla is more amenable to demacrofication.

VIII. Conclusions and Future Work

In this paper we presented an approach that can be used to demacrofy C++ programs by replacing macro definitions with C++11 declarations. We evaluated aspects of the approach and our tooling 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. 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 our automated approach would be greatly improved if our tools had full knowledge of the C++ program structure.

In the future, we plan to further investigate the integration of a compiler front end to supply the information needed to improve our mappings 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 perhaps improve our ability to place the transformed results.

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++," *Reading, MA: Addison-Wesley,— c1994*, vol. 1, 1994.

[3] M. Ernst, G. 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] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landray, "Google C++ style guide," http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml, 2008.

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

[6] B. Stroustrup, K. Carroll, and L. Aero, "C++ in safety-critical applications: The JSF++ coding standard," 2006.

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

[8] C. Mennie, "Giving meaning to macros," Waterloo, Ontario, Canada, 2004.

[9] J. 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.

[10] R. Khatchadourian, J. Sawin, and A. Rountev, "Automated refactoring of legacy java software to enumerated types," in *ICSM*. IEEE, 2007, pp. 224–233.

[11] A. Sutton and J. 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.

[12] I. D. Baxter and M. Mehlich, "Preprocessor conditional removal by simple partial evaluation," in *WCRE*, 2001, pp. 281–290.

[13] 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.

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

[15] L. Vidács, Á. Beszédes, and R. Ferenc, "Columbus schema for c/c++ preprocessing," in *CSMR*. IEEE Computer Society, 2004, pp. 75–84.

[16] R. Stallman and Z. Weinberg, "The C preprocessor. GNU Online documentation," 2001.

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

[18] C++ Standards Committee and Becker, P. and others, "Programming languages-c++(final committee draft). C++ standards committee paper wg21/n3092= j16/10-0082," 2010.

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