The CPP paradox

Jean-Marie Favre[1]
IMAG Institute
Université Grenoble (France)

Introduction

The CPP paradox

CPP is the *C p*re*p*rocessor. For most researchers this is a tool *of* the past. For most practitioners it is a tool *from* the past but one which is currently used. The heavy use of CPP can lead to unreadable programs; nevertheless large amounts of code are written using it. CPP makes maintenance difficult, but CPP is largely used for maintenance... CPP can convert good programs on which many program understanding tools apply, to incomprehensible programs with no tool assistance... The presence of CPP constructs in programs is a headache for maintainers and for tools builders, and CPP is still here... Most software engineering researchers consider CPP as an uninteresting tool of the past but, when implementing their research prototypes, they turn into practitioners and use it.

This introduction is controversial. The aim of this paper is to present what lies behind these paradoxes and assess what should be done. Such paradox are representative of other distinctions: Industry vs Research, State-of-the-practice vs State-of-the-art, Maintenance vs Development, Reengineering vs Engineering.

Programming-in-the-large vs Programming-in-the-small

The distinction introduced by De Remer and Kron in "*Programming-in-the-large vs Programming-in-the-small*" [Der76] is also relevant here. They argue that programming languages are well suited to describe algorithms but not to describe the structure of complex versioned software products. Since then, the separation of concerns between Programming-in-the-large (PITL) and programming-in-the-small (PITS) has been considerably reinforced and the corresponding research lines have evolved in parallel. This distinction usually implies a granularity step. For instance, PITL tools typically see files as elementary values without considering their contents. Conversely, programming languages deal with instructions, integer and character values and usually do not provide support for versioning.

As shown in this paper, the C preprocessor is a pragmatic tool which bridges the gap between PITS and some aspects of PITL. On the one hand, CPP has been designed by PITS practitioners and is itself a (degenerated) programming language. On the other hand CPP tries to partially solve some of the PITL problems, like modularity, versioning and configuration issues.

Content of this paper

Section I presents CPP from the State-of-the-practice point of view and gives answers to such questions as: *Why to talk about CPP? What mechanisms are provided by CPP? How, when and why are they still used in practice? What is wrong with CPP?*

CPP is then considered from a reengineering point of view. Section II answers questions like: *What is Reengineering-in-the-large? What techniques can be applied?*

---

1. *ADELE Team*
Laboratoire LSR, BP 53, 38041 Grenoble Cedex 9. e-mail: Jean-Marie.Favre@imag.fr,
Voice (33)76514964, Fax (33)76446675, http: //www-lsr.imag.fr/users/Jean-Marie.Favre

I • CPP & the State-Of-The-Practice

In the 70's practitioners designed the C language and CPP. Rationales were flexibility, efficiency, simplicity and portability. Faced with the heavy machinery of Algol68 and all the concepts it provides, they preferred to provide a fast compiler ensuring only a few checks, along with a debugger... De Remer and Kron claimed that new concepts and tools must be introduced to cope with large program production. Far from this idea, CPP has been designed to provide features coping with some of the problems.

Two decades on why talk about CPP in a software engineering research context? It makes sense in a maintenance and reengineering perspective. The paradox mentioned above should also be of interest. Software engineering should not just consist of the elaboration of new methods and tools for future development; it must also keep an eye on old software produced by old tools.

I.1) CPP significance

Why choose CPP as a case study?

- CPP, with the C language, is widely used both in industry and research. They are fundamentals in the UNIX operating system. Usually interfaces to system libraries are based on file inclusion and make an extensive use of conditional compilation. The same is true for large components like the X Window system.
- CPP is mainly used with the C language, but it is not limited to it. It is also used with other programming languages like Pascal dialects and C++ or even with text file or LaTeX documents.
- Since it provides features like macro substitution, textual inclusion and conditional compilation, CPP is relatively representative of other textual processors, and a large proportion of what is said in this paper should be applicable to other text processors.
- Versioning and configuration aspects are usually considered at high granularity level. On the other hand, thanks to conditional compilation, CPP is one of the few tools which enables us allows to deal with these problems at a fine grained level. Indeed, cpp files constitute a portable representation for program families[1]. Parametrized software products like GNU public domain tools make extensive use of CPP.
- CPP is based on textual representation of programs: a simple but effective technology. In spite of its evident drawbacks this representation is still widely used, and tools like emacs, sccs, diff and some textual mergers are representative of the state-of-the-practice.
- With the C language, CPP is both an old and a modern tool. For new application developments the industry shows an increasing interest in C++. While C++ features decrease the need for CPP tricks, the transition to this programming paradigm will not be immediate and CPP might not disappear rapidly. Thus, if nothing is done, in the next decade(s) CPP might still give maintainers headaches.

Although CPP is a very low-level tool, its significance has been mentioned in workshops on software configuration management [SCM] [Win88] [Sch89][Gen89][Malh94] and on software reuse [Gro93]. Problems due to its use have also been reported for large software products [Spe92] [Til92]. Some tools have been developed [Lit93] [Vo92]. Recently, different, independent research have shown a growing interest in CPP, taking it as a case study or seeing it from a reengineering point of view [Spu92] [Mun93] [Sin93] [Fav93] [Kro94] [Liv94] [Zel94] [Fav94] [Sne95] [Fav95].

1. While PITS deals with programs, PITL deals with program families.

I.2) CPP Features

In this section the main features of mechanisms provided by CPP are presented. The reader is invited to consult C programming language manuals for further information (or [Sta92]).

CPP is a textual preprocessor. It means that, before the compilation phase, a textual transformation is applied to the program sources. These sources contain a mixture of cpp directives and pieces of text. The output of CPP ought to be a well-formed C program. Compilers for the C programming language can be considered as pure PITS tools. They do not deal with modularity or versioning problems. Their unique function is to transform a monolithic flow of characters into an object code. CPP, on the other hand, is responsible for the selection and composition of multiple (textual) components. This can be viewed as a configuration problem and, in this sense, CPP has a PTIL flavour[1].

CPP directives are lines beginning with a hash character '#'. Each directives and piece of text is processed sequentialy.

File inclusion

The file inclusion directive (#include) allows the composition of multiple files. It takes a filename as argument and temporarily changes the input to process the specified file. These directives can be nested but no cycle is allowed. Actually, a file name is specified, not the content of the file. So the cpp output heavily depends on the relation file name -> file content, i.e. on the file system state before the cpp invocation. CPP does not modify this relation which acts as a cpp parameter. In figure 1 an example is provided. Since the input contains a #include directive, the file system is also important. Furthermore, "directory path access" can also be specified[2] and change the interpretation of the relation. File inclusion directives deal with coarse grained information kept in the file store and indirectly identified with filenames.

Macro definition and macro substitution

The macro substitution mechanism provide roughly similar services but at a lower level of granularity. It deals with strings indirectly identified by macro names. The relation macro name -> macro value is not static: the #define *<macroname> <macrovalue>* directives allow sequential modification as the cpp processing goes on. Note that a macro value can be an empty string. The #undef *<macroname>* directive can be thought as a special case. It associates a special macro value, let's say "*undefined*", to the specified macro name. At the CPP invocation an initial value is given for this relation: some macro name assignments can be specified as parameters[3] and the *undefined* value is virtually assigned to other macro names[4]. Note that at definition time, the macro value

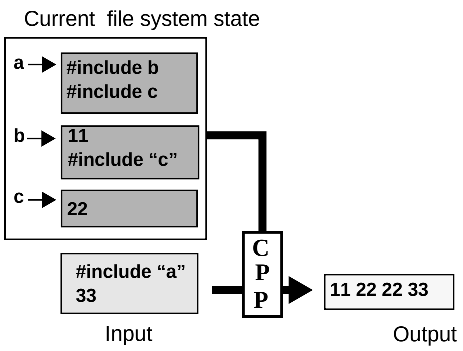


Figure 1

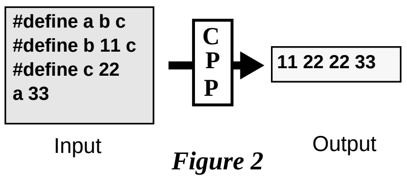


Figure 2

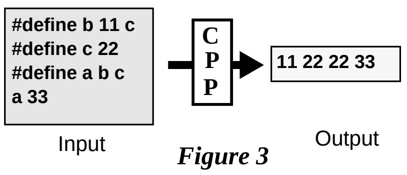


Figure 3

1. The same is true for link editors since they assemble object codes and choose proper libraries.
2. Via a sequence of options-I<directory> on command line. Note also that compiler may use different built in path access for system libraries.

is *not* processed by CPP. It is considered as a string and it is not interpreted (the same occurs with cpp files: their content are not interpreted when stored in the file system). This property implies that the relative positions of #define directives is not relevant if they refer to different macro names. For instance, the inputs shown in figure 2 and 3 are equivalent.

During the preprocessing, #include occurrences are substituted by the processed file contents. The same occurs for macros. An occurrence of a macro name is replaced by the corresponding macro value. Since macro names can be nested (macro values can contains macro names), the result is then processed for subsequent substitutions (see figure 2).

Macro definitions can be slightly more elaborate: they allow formal parameters which are instantiated with actual string parameters given for each macro name occurrence.

Conditional compilation

Conditional compilation allows pieces of text to be included or exclude for further processing. Since CPP is not necessarily used with a compiler, this feature should be called conditional inclusion. Conditional directives take the form of a #if *<expression>* *<cpptext1>* #else *<cpptext2>* #endif sequence. Here *<expression>* is a C restricted integer expression. It deals only with integer values. Basic arithmetic and relational operators are available. A result equal to 0 leads to the processing of the “else” part.

Conditional expressions depend on the state of the mapping macro name -> macro value. If a #include directive is present in the cpp file it may also depend on the mapping filename -> file content since these files can define alternative macro values for a same macro name. Indeed file inclusion, conditional inclusion and macro substitution are closely related and are often largely interleaved. The first one deals with files (usually large sets of lines, i.e. a very long string), the second one deals with file portions (usually a few lines, e.g. a medium string), and the third deals with (usually small) strings.

I.3) CPP uses and abuses

One of the major qualities of CPP is that it provides a great flexibility. At the same time this feature constitutes its main drawback[1]. CPP low-level mechanisms allow a great variety of usage and it is sometimes very difficult to recognize the purpose of a sequence of directives. Nevertheless, different classes can be identified.

CPP and textual abstraction

The possibility of giving a name to a string (a file or a macro value) is a basic support for textual abstraction. For instance, since the C language does not provide constant declarations, the #define directive is widely used for this purpose.

CPP and modularity

Thanks to the separate compilation mechanism, software can be broken down into various compilation units. This is a useful step in implementing a first level of modularity. However, the separation between interface and realisation is not made and standard link editor technology does not allow any consistency check. To cope with this problem, the file inclusion mechanism is

3. Trough the options -D <macroname> <macrovalue> on command line which is equivalent to #define <macroname> <macrovalue> in cpp source text. Note that built in assignments are usually done by compiler front end implementations in order to define macro names according to the language, operation system, etc.
4. It is also possible to see this relation as a partial function from macro name to strings. In this case the #undef directive correspond to a domain restriction operator, the #define directive to the overlap operator.
1. In a spaghetti plate each spaghetti is very flexible... The resulting structure of the spaghetti plate is also flexible... but incomprehensible!

widely used to implement the interface notion. Shared definitions can be clustered in an include file (typically macro and type definitions, procedure headings, etc.).

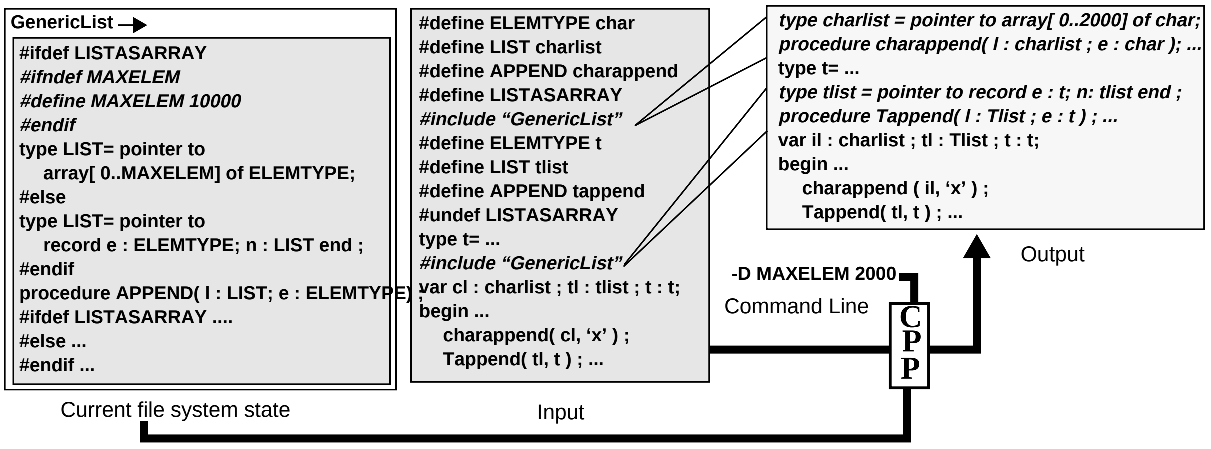
An include file, let's say X, may "depend" on another include file. X might assume that it will be included in a context in which these include files have been included. This solution is not reliable and X should itself include these files by means of #include directives. In this case an include file can be processed more than once. Giving such output to a compiler can lead to significant performance decreases [Vo92] [Lit93] and to compilation errors if multiple definitions are not allowed by the language. To cope with this problem a common practice consist to use a sequence #ifndef *<aname>* #define *<aname>* *<filecontent>* #endif [Spe92]. The file content will appear only once in the output.

CPP, parametrisation and genericity

The output resulting from the processing of a piece of cpp text depends heavily on the context in what it appears. This context depends on the file system state and parameters given in the command line. Since parameters and file contents are arbitrary values, there is potentially an infinity of interpretations for a cpp piece of text. CPP can implement some form of parametrisation and genericity [Win88][Til92][Gro93]. In figure 4 such an example of genericity is presented. In real situations the maintainers should understand this without explanations or graphical help... The file named GenericList contains some generic definitions and list implementations. The input of cpp is a client module which relies on two list definitions: a list of char implemented as an array and a list of t-value implemented as linked list[1]. At CPP invocation time, without changing any sources, the programmer has assumed that, for this particular instantiation, a maximum number of 2000 elements is sufficient. Note the construction #ifndef maxelem #define maxelem 10000 #endif in the include file GenericList. Such constructions are commonplaces [Spe92]. They allow default values to be to assigned to parameters.

CPP, portability and versioning

Through the conditional compilation mechanism different "versions" can be embedded in a single file. This is especially useful for software portability [Bro74][Til92][Mah94]. In fact, constraints associated with porting activities are very stringent. Delivery delays are usually very short. Furthermore, for embedded systems it is not uncommon for the target platform to be unavailable at the maintenance site. In this case, programmers can be temporarily remotely



```
#ifdef LISTASARRAY
#ifndef MAXELEM
#define MAXELEM 10000
#endif
type LIST= pointer to
    array[ 0..MAXELEM] of ELEMTYPE;
#else
type LIST= pointer to
    record e : ELEMTYPE; n : LIST end ;
#endif
procedure APPEND( l : LIST; e : ELEMTYPE) ;
#ifdef LISTASARRAY ....
#else ...
#endif ...
```

```
#define ELEMTYPE char
#define LIST charlist
#define APPEND charappend
#define LISTASARRAY
#include "GenericList"
#define ELEMTYPE t
#define LIST tlist
#define APPEND tappend
#undef LISTASARRAY
type t= ...
#include "GenericList"
var cl : charlist ; tl : tlist ; t : t;
begin ...
    charappend( cl, 'x' ) ;
    Tappend( tl, t ) ; ...
```

```
type charlist = pointer to array[ 0..2000] of char;
procedure charappend( l : charlist ; e : char ); ...
type t= ...
type tlist = pointer to record e : t; n: tlist end ;
procedure Tappend( l : Tlist ; e : t ) ; ...
var il : charlist ; tl : Tlist ; t : t;
begin ...
    charappend ( il, 'x' ) ;
    Tappend( tl, t ) ; ...
```

Figure 4

1. The use of the concat operator (##) provided by recent CPP versions can considerably simplify this example. This operator allows to prefix an identifier by a macro. For example provided that elemtype is set to char, the strings elemtype##list and elemtype##append are respectively substituted to charlist and charappend if append is elemtype is set to char.

located or cross development can be done [Gen89]. Far from the research horizons, these industrial realities explain why porting software usually consists of making patches in the text sources rather that restructuring it if needed. Here the flexibility of the conditional compilation mechanisms is appreciable and constructions like #ifdef sun ... #else ... #endif appear rapidly. Note that when porting, software products must often be adapted to new and foreign components. When development time comes round it is not always possible to anticipate these evolutions. Clearly, the appearance of standards like POSIX helps to limit variations between systems, but, as pointed out by Tilbrook and Crook, adherence to standards[1] only partially resolves porting problems [Til92]. With the increasing complexity of software developed, the trend is to reuse whenever possible large foreign components. Nowadays, the porting problems do not only consist in dealing with system calls and file systems but also with window systems, networking components, object managers, etc.

Software text sources also depend on programming languages dialects and sometimes on CASE tools. Thanks to conditional compilation some sources can be compiled with various implementations of C, Ansi C, and C++ compilers. In practice these variations leads to sequences like #ifdef _cplusplus... #else #ifdef STDC ... #else ... #endif.

While porting is an important aspect, other goals lead to variations in the source text of programs. Typical examples are optional code for debugging purposes (the traditional #ifdef DEBUG ... #endif sequence [Aba89]), adaptation to natural languages, time/space trade-off, etc. [Mun93] [Sin93] [Zel94].

I.4) CPP drawbacks

While widely used, it is well known that CPP is a problematic tool [Til92] [Gen89] [Sin93] [Kro94]. Some incomprehensible cpp pieces of code are reproduced in [Spe92]. To get a better feeling for what can dive maintainers headaches the reader is invited to have a look at some UNIX "portable interfaces"[2]. Figure 5 shows some pieces of code extracted from the include files of an operating system. The comments show typical situations that arise in practice. Unfortunately in much software this kind of informations are not present, the maintainer must rely only on the code. An interface from public domain code is reproduced in appendix A.

Rather than simply noting that cpp use can lead to the worst case, it is fundamental to know precisely what makes CPP so bad. Major flaws are identified below. Firstly, we present cpp intrinsic problems and then problems which arise when it is used with a programming language.

1 • ***Understanding a cpp piece of text may involve the examination of various files.*** The

```
/*
 * @(#)keyboard.h 1.11 88/09/28
 * Copyright (c) 1986 by Sun Microsystems, Inc.
...
 * Since this file is included by <mon/eeprom.h>, which is
 * included by <machine/eeprom.h>, kernel files (specifically
 * kbd.c) include it, too. The kernel file <sundev/kbd.h>
 * contains parallel definitions for the constants in this file.
 * All this should be cleaned up, but in the meantime, the
 * following #ifndef is a kludgey solution.
 */
#ifndef IDLEKEY
...
```

```
/*  @(#)buserr.h 2.2 92/05/27 SMI*/

/* This file is not needed for sun4m builds.
 * For historical reasons it was copied into this directory.
 * Since Some files in the non-sun4m directories include this file,
 * I will offer an empty file here
 */
```

Figure 5

1. Some people say that standards are fine because there is a great possibilities to choose...
2. In some old unix kernel appears "you are not expected to understand this".

relative order of text pieces may be essential. Invocation command lines are also of interest since they usually contain samples of parameter assignments. Unfortunately, they are usually not available to the maintainer[1].

2 • Sometimes the maintainer wants to focus his/her attention on a specific version of the source code. For instance he or she may choose to concentrate on the Sun version and try to understand why the French and English variants have unexpected behaviours. For that, ***the maintainer should reproduce mentally a partial cpp execution***. Without tools assistance it is cumbersome and error-prone.

3 • There is no lexical difference between regular identifiers and macro names. As a consequence, ***it is very difficult to know which are the parameters of a piece of code*** (see, for example, the GenericList file in figure 4). An explicit macro substitution syntax would have been a better solution[2]. In [Til92] a preprocessor which deals with this problem is proposed. With, CPP the common usage generally consists of using upper cases for macro names. Unfortunately, this rule is not always respected.

4 • ***Reading only a macro definition does not allow its aim to be determined***. The macro can be used as a version selector (e.g. debug, sun) or as a parameter (e.g. sizemax 100). In the first case it will appear (elsewhere) in some conditional expressions, in the latter case in some piece of code. Note that these possibilities are not exclusive. Trying to capture the meaning of a set of #define directives often requires browsing all cpp files to observe usages of these definitions!

5 • ***Since CPP does not interpret macro values at definition time the mental interpretation should also be deferred. This makes understanding even more complex***. See, for instance, the #define directives in the input of figure 4. The reader cannot capture the purpose of the #define elemtype T before reading the GenericList file and the definition of the T type. Consider also the construction #define A 1 #define B A ... #undef A ... #define A 10 ... B. When reading the B definition, the maintainer will probably infer that it is a short-cut for the value 1, but in fact the B occurrence is replaced by 10. Such constructions should always be considered with suspicion. Without tool assistance it may be difficult to detect such cases since they are likely to appear precisely in the worst cases: when CPP preprocessing involves complex file inclusion graphs.

6 • By default, *undefined* values are initially assigned to macro names. Thus ***for a macro there is no difference between an intentional lack of definition and an erroneously omission***. Indeed, relying on this value as a version selector is a bad (but generalized) practice. The #ifndef *<macro>* #define *<macro>* *<defaultvalue>* #endif sequences are exceptions: they provide great flexibility while ensuring that the macro will be defined with a correct value. Furthermore, such a construction usually constitutes a clue to the presence of a parameter. As stated above the *undefined* value is replaced by 0 in conditional expression. In an arithmetic term this is probably not the expected behaviour and it usually corresponds to an error.

7 • The usage of the #else directive is also problematical since the universe of variation is not closed. For instance, a program originally defined for a Sun and then ported to Hp would probably have some constructions like #ifdef hp ... #else ... #end. This hides the fact that the else part has been tested only on Sun and when ported to a new platform it will be taken by default. A construction like #ifdef hp ... #endif #ifdef sun ... #endif is the worst because no code will be produced for another platform! A better construction (very rare in practice) will signal an error when needed: #ifdef hp... #else #if sun ... #else #error #endif. More generally, ***preconditions***

1. Is is worthwhile to look at makefiles when used in conjunction to CPP.

2. For instance an explicit $*<macroname>* notation (as used in shell scripts) would have simplified the comprehension of the GenericList include file.

on parameters and contexts are not made explicit in cpp programs. In [Zel94] it is suggested that a sequence checking proper precondition on macro value be inserted at the beginning of each cpp file. For instance, #if ! defined(sun) || ! defined(hp) && !defined(GUNC) #error #endif.

The flaws presented above are intrinsic to cpp features and will remain even when preprocessing plain text or other documents[1]. As shown below, when CPP is used with programs more problems crop up!

8 • ***While maintainers consider cpp input, compilers or CASE tools consider cpp output***. ***This difference can lead to misunderstanding*** and difficulties in interpreting compiler errors or use tools like debuggers. The reader may have noticed that in the figure 2 a great help is provided by the lines between cpp input and output. Technically this kind of informations is transmitted to the compiler via the insertion of #file *<filename>* and #line *<linenumber>* annotations in cpp output. But no #macro *<macroname>* construction is provided[2]. Without tool assistance to debug cpp execution, maintainers try to decipher cpp output. To understand the reason for an error they must reproduce mentally the multiple nested substitution chains. Since the cpp output contains no trace of macro definition and expansion this task is very painful!

9 • ***CPP works on textual representation. A syntactic basis would have been better*** [Win88] [Wei93]. Sometimes such directives split syntactic entities; the reader must mentally recombine them. Usually, this does not lead to major problems since compilers will check syntax. Nevertheless, if the language provides very similar syntactic constructions for different semantics, serious problems are likely to arise. Let us consider the following example: #define A 1 ... #define B A+4 .... c := B*2 ; the value assigned to C is ... 9! In fact the expression B*2 is replaced by 1+4*2. This substitution “cuts” the corresponding syntactic tree.

10 • ***No semantic checking can be ensured on a cpp include file***: (1) CPP deals with strings not with semantic entities, (2) the output of cpp depends on the inclusion context, which is not restricted. For example, no semantic checking can be done for the include file GenericList alone. All the compiler can do is to ensure that a specific instantiation is correct.

11 • ***Test coverage problems encountered at program level arise at a program family level***. With no tool assistance the maintainers may have difficulties to determine which pieces of text have been exercised and under which conditions.

12 • It was stated in the introduction that CCP also represents headaches for tools builders. ***The presence of cpp directives in programs impose serious limitation on the applicability of worthwhile techniques***, especially those which are based on program source transformations. This is due to the fact that analysis techniques are available for programs, not for program families. For instance, it would be useful to apply slicing techniques on real industrial C programs, but currently the presence of cpp instructions (along with other factors) makes this impossible. Different tools manage versioned information similar to cpp files, but, since they do not take into account all cpp tricks, they are not applicable to real C programs [Kru83] [Sne93] [Mun83] [Zel94].

1. Nevertheless the complexity is much lower because pieces of text are less interconnected (or at least the semantic connection between these part are not taken into account).
2. This pragmatic choice can be explained by the fact that include files are expected to contain long strings included few times whereas macro value are usually short strings included many times. Usually #file and #line roughly allow the programmer to trace the execution of cpp. When macro values are large, as for “inline” function, maintainers are lost in cpp output.

I.5) So what?

Faced with the problem presented here, different reactions are possible. Maintainers may take an aspirin and try to patch patches before deadlines expire. Practitioners may impose a discipline on the use of CPP [Spe92] [Aba89], set up a portability framework and implement a better preprocessor [Til92]. Some academics may say “with the ‘alpha’ method this may not occur” and 20 years later when millions of lines of code have been developed with ‘alpha’ and (new) maintenance problems arise they may say “with the ‘beta’ method this may not occur”. Some researchers may ask “What can research do to make maintenance of real programs easier? What can we do with cpp files?”

Of course, all these reactions are complementary and correspond to different points of view. In the next section the last option is developed.

II • CPP and Reengineering

The need to deal with maintenance of a growing amount of large and old software leads to the emergence of *Software Reengineering* [Arn93]. In the remainder of this paper the taxonomy proposed by Arnold is used because it presents “Reengineering” as a general term which encompasses a large set of activities defined by goals. According to this definition a reengineering activity may or may not modify the software. Thus program understanding is also a reengineering activity.

II.1) Reengineering-In-The-Large vs Reengineering-In-The-Small

As suggested in [Fav94] the distinction between Programming-in-the-large and Programming-in-the-small applies to all domains of software engineering and thus can be specialized to reengineering:

> *Reengineering-in-the-small activities are reengineering activities which focus on the detail of small portions of the software.*

For example, the aim of the European Esprit project REDO is to generate Z specifications from source code [Bow91]. Since a detailed knowledge of source code is necessary, it is essentially a Reengineering-In-The-Small (RITS) approach.

> *Reengineering-in-the-large activities are reengineering activities which focus on the structure of software and/or their versioning dimensions.*

The C++ Information Abstractor from AT&T is a typical example of a Reengineering-in-the-large (RITL) system [Gra92].

The PITL/PITS distinction can also be applied to reengineering sub domains. For example, *Restructuring-In-The-Large* can be distinguished from *Restructuring-In-The-Small*. The Arch system [Sch91] is representative of the former case while traditional techniques for goto instructions elimination are related to the latter case. In what concerns *Program Understanding*, it is proposed in [Fav94b] that the term *Program Family Understanding* be used when the focus is on structural or versioning aspects of software.

The structural aspect of software is important in PITL, and RITL tools take it into account. But versioning has been forgotten by the reengineering discipline [Fav94a]. Almost all the existing tools only consider a non-versioned world. Gulla emphasizes the lack of views taking version aspects into account [Gul93]. To understand a program family, a maintainer might apply classical tools successively to each version and then try to make the synthesis by himself... If he wants to use a Restructuring-in-the-large tool, he might do it on a specific version, but nothing ensures that this restructuration is compatible with other ones. ***Large software products are versioned and reengineering must deal with it***.

II.2) CPP and reengineering.

Preprocessors are isolated tools in the sense that they are based on rather specific concepts. Their behaviour is not well defined. Furthermore different preprocessors are used in legacy systems. Building reengineering tools in this context can be rather difficult.

To cope with this problem, in [Fav95] CPP is studied from a non-conventional point of view: this preprocessor is seen as a basic imperative programming language working on strings. So we have defined APP (Abstract PreProcessor), a small language which is compatible with CPP, but which uses Programming-in-the-small concepts.

Roughly speaking, macros are variables; the relation macro name -> macro value acts as the memory; #define and #undef directives are assignments; control structures are very poor: there are only conditional instructions, no loop, no goto statement; files act as procedures and each fragment of text is interpreted as an output statement (i.e. A CPP execution modifies the memory (of macro) and generates an output).

The abstraction process from CPP to APP is shown in the table below. APP is an abstract language and it is defined for reengineering purposes only. It appears here with a concrete syntax only to facilitate the comprehension. In fact, APP programs are CPP abstract trees.

```
procedure GenericList is
begin
   if [defined $LISTASARRAY ] then
       if [not defined $MAXELEM] then
           MAXELEM := “[10000]”
       else
           output [type $LIST = pointer to] ;
           output [array [0.. $MAXELEM ] of $ELEMTYPE ; ] ;
   else
       output [type $LIST = pointer to] ;
       output [record e $ELEMTYPE n “:” LIST end “;” ] ;
   output [ procedure $APPEND ( l : $LIST ; e : $ELEMTYPE ) ] ;
   if [defined $LISTASARRAY] then
       ...
   else
       ...
end
```

```
procedure p is
begin
   ELEMTYPE := “[ char ]” ;
   LIST := “[ charlist ]” ;
   APPEND := “[ charappend ]” ;
   LISTASARRAY := “[]” ;
   call GenericList ;
   ELEMTYPE := “[ t ]” ;
   LIST := “[ tlist ]” ;
   APPEND := “[ tappend ]” ;
   LISTASARRAY := “[]” ;
   output [ type t = ... ] ;
   call GenericList ;
   output [ var cl : charlist ; tl : tlist ... ] ;
   ...
end
```

TABLEAU 1 Abstraction: from CPP to APP (cpp file = app procedure)

concrete terminology	CPP	APP	abstract terminology
cpp file	<File> ::= <FileName> <FileContent>	<Proc> := **procedure** <ProcName> **is** **begin** <ProcBody> **end**	app procedure
file name	<FileName>	<ProcName>	procedure name
file content	<Filecontent> ::= <CppDirective>	<ProcBody> ::= <Stmt>	procedure body
cpp directive	<CppDirective> ::= <SequenceDirective> | <TextualInclusion> | <Conditionals> | <TextFragment> | <MacroDef> | <MacroUndef>	<Stmt> ::= <SequenceStmt> | <CallStmt> | <IfStmt> | <OutStmt> | <AssignStmt>	statement
directive sequence	<SequenceDirective> ::= <CppDirective> <CppDirective>	<SequenceStmt> ::= <Stmt> **;** <Stmt>	statement sequence
textual inclusion	<TextualInclusion> ::= **#include “**<FileName>**“**	<CallStmt> ::= **call** <ProcName>	procedure call

TABLEAU 1 Abstraction: from CPP to APP (cpp file = app procedure)

concrete terminology	**CPP**	**APP**	**abstract terminology**

conditional compilation	<Conditionals> ::=
	#if <CondTokenList>$_1$
	<CppDirective>$_1$
	#elsif <CondTokenlList>$_2$
	<CppDirective>$_2$
	#elsif ...
	...
	#else
	<CppDirective>$_n$
	#endif
		<IfStmt> ::=
		if <CondTerm>$_1$
		then <Stmt>$_1$
		else if<CondTerm>$_2$
		then <Stmt>$_2$
		else ...
			conditional instruction

condition	<CondTokenList> ::= <TokenList>	<CondTerm> ::= <Term>	condition term
text fragment	<TextFragment> ::= <TokenList>	<OutStmt> ::= **output** <Term>	output instruction
token list	<TokenList> ::= [<Token>]	<Term> ::= [<Factor>]	term
token	<Token> ::= <MacroNameOcc> | <StdToken>	<Factor> ::= <Const> | <VariOcc>	factor
regular token	<StdToken>	<Const>	constant
macro occurence	<MacroNameOcc> ::= <MacroName>	<VarOcc> ::= **$**<Var>	variable occurence
macro name	<MacroName>	<Var>	variable
macro definition	<MacroDef> ::= **#define** <MacroName> <MacroVal>	<AssignStmt> ::= <Var> **:=** <Value>	variable assignment (dynamic binding)
macro value	<MacroVal> ::= <TokenList> | <ParameterizedMacro>	<Value> ::= <Term> | <Function>	value
parametrized macro	<ParameterizedMacro> ::= **(** [<Var>] **)** <TokenList>	<Function> ::= function **“fun** [<Var>] . <Term>**“**	function

The denotational semantics of APP have been defined [Fav95] so it is possible to think about CPP files on a formal basis. The benefit of this abstraction is that it allows us to use concepts available in the PITS domain. For example, it is possible to apply program analysis techniques like slicing, partial evaluation or data flow analysis. Slicing is useful for maintainers wishing to read a CPP piece of code. Partial evaluation allows them to remove unused versions. Computing reaching definitions and live variables allows them to know which macros are used in a given piece of code. Inter-procedural data flow analysis is necessary in the cases of multiple files.

The main difficulty lies in the fact that CPP uses dynamic binding. This is not a conventional problem in program analysis. We have designed and implemented an algorithm which builds a program dependence graph and takes dynamic binding into account. Slices are based on this result.

Currently we are implementing different tools. They are based on:
(1) a functional language (Camllight a dialect of ML),
(2) lex and yacc for the CPP to APP conversion,
(3) the daVinci graph editor to show different views of software.

III • Conclusions

CPP can be seen as a tool for representing program families. Recently there has been growing interest in studying this tool in a research context [Spu92] [Mun93] [Sin93] [Fav93] [Kro94] [Liv94] [Zel94] [Fav94] [Sne95] [Fav95]. Usually only specific techniques are applied. We propose to use a general approach with a formal basis in order to be able to apply the results obtained to other preprocessors.

Bibliography

[Aba89] A. Abacus; “Parameterizing C Code at Compile and Run Time”, in Structured Programming, Vol. 10, N. 4, Springer Verlag, 1989, pp. 209-214.

[Arn93] R.S. Arnold; “*Software Reengineering*”, IEEE Computer Society Press, ISBN 0-8186-3272-0, 1993, 675 pages.

[Bro74] P.J. Brown; “Macro processors and techniques for Portable Software”, Wiley series in computing, ISBN 0 471 11005 1, 1974, 244 pages.

[Fav93] J.M. Favre; “*Vers une representation multi-langages et multi-versions des programmes*” in 6th International Conference On Software engineering and its Applications, Paris, France, 1993, pp. 459-468, In French.

[Fav94] J.M. Favre; “*Reengineering-In-The-Large vs Reengineering-In-The-Small*”, first SEI Workshop on Software Reengineering, Software Engineering Institute, Carnegie Mellon University, May 1994.

[Fav95] J.M. Favre; “*Une approche pour la maintenance et la ré-ingéniérie globale des logiciels*”, PhD dissertation, Unsiversity of Grenoble, July 1995.

[Gen89] W. M. Gentleman, S.A. MacKay, D.A. Stewart, M. Wein; “*Commercial Realtime Software Needs Different Configuration Management*” , in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 152-161.

[Gra92] J.E. Grass; “*Cdiff: A Syntax directed Differencer for C++ Programs*” in USENIX, Computing Systems, Vol. 5, N. 1, 1992.

[Gro93] F.J. Grosch, G. Snelting; “Polymorphic Components for Monomorphic Languages” in Proc. of the 2nd Internation Workshop on Software Reusability, pages 47-55, Lucca, Italy, March 1993, pp. 47-55.

[Gul93] B. Gulla; “*The constraint diagram: an approach to visualizing the version space*” , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993, pp. 112-122.

[Kin81] J.C. King; “*Program Reduction using Symbolic Execution*” in ACM SIGSOFT Software Engineering Notes, Vol. 6, N. 1, January 1981, pp. 9-14.

[Kro94] M. Krone, G. Snelting; “*On the Inference of Configuration Structures from Source Code*”, Proc. 16th International Conference on Software Engineering, Sorrento, Italy, 1994.

[Kru83] V. Kruskal; “*Managing Multi-Version Programs with an Editor*” in Research Report, RC 10217 (#45003), IBM Thomas J. Watson Research Center, P.O. box 218, Yorktown Heights, New York 10598, August 83.

[Lit93] A. Litman; “*An Implementation of Precompiled Headers*” in Software - Practice and Experience, 23:3, March 1993, pp. 341-350.

[Liv94] P. E. Livadas, D.T. Small; “Understanding Code Containing Preprocessor Construct”, in IEEE Third Workshop on Program Comprehension, Washington, November 1994.

[Mah94] A. Mahler; “Variants: Keeping Things Together and Telling Them Apart” , Chapter 3, in W.F. Tichy, Editor; “Configuration Management”, Trends in Software 2, ISBN 0-471-94245-6, John Wiley & Sons, 1994.

[Mun93] B.P. Munch; “*Versioning in Software Engineering Database : the Change Oriented Way*” , PhD dissertation, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, 1993.

[Nar89] K. Narayanaswamy; “*A Text-Based Representation for Program Variants*” in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 30-33.

[Sch89] U. Schroeder; “*Incremental Variant Control*” in Proc. of the 2nd International Workshop on Software Configuration Management, Princeton, (New Jersey), October, 1989, pp. 145-148.

[Sch93] R.W. Schwanke, V.A. Strack; “*Configuration Management Problems and Architectural Integrity*” , in Proc. of the 4th International Workshop on Software Configuration Management, Baltimore, Maryland, USA, May 1993, pp. 225-228.

[SCM] Proc. of International Workshop on Sofware Configuration Management. 1st SVCC Grassau 1988, 2nd Princeton 1989, 3rd Trondheim 1991, 4th Baltimore 1993.

[Sin93] P. Singleton, O.P. Brereton; "*A Case for Declarative Programming-in-the-Large*", in Proc. 5th International Conference on Software Engineering and Knowledge Engineering, San Francisco, California, 1993.

[Sne95] G. Snelting; "Reengineering of Configurations Based on Mathematical Concept Analysis", Computer science report 94-02,Technical University of Braunschweig,
Germany, January 1995, 28 pages.

[Spe92] H. Spencer, G. Collyer; "*#ifdef Considered Harmful, or Portability Experience With C News*" in USENIX, Summer 1992 Tecnical Conference, San Antonio (Texas), June, 1992, pp. 185-197.

[Spu92] D. Spuler, A.S.M. Sajeev; "Static Detection of Preprocessor Macro Errors in C",
Technical report 92-7, James Crook University, 1992, 18 pages.

[Sta92] R. Stallman; "The C Preprocessor", GNU Project, Free software foundation, July 1992, 52 pages.

[Til92] D. Tilbrook, R. Crook; "*Large Scale Porting through Parametrization*" in USENIX, Summer 1992 Tecnical Conference, San Antonio (Texas), June, 1992, pp. 209-216.

[Vo92] K.P. Vo, Y.F. Chen; "*Incl: A Tool to Analyse Include Files*" in USENIX, Summer 1992 Tecnical Conference, San Antonio (Texas), June, 1992, pp. 199-208.

[Wei93] D. Weise, R. Crew; "Programmable Syntax Macros", in ACM SIGPLAN '93
Conference on Programming Language Design and Implementation, 1993, pp. 156-165.

[Win88] J.F.H. Winkler, C. Stoffel; "*Program-Variations-in-the-Small*", in Proc. International Workshop on Software Version and Configuration Control, Grassau (Germany), January, 1988, pp. 175-196.

[Zel94] A. Zeller; "*Configuration Management with Feature Logics*", Technical report TR-94-01, Technische Universitat Braunschweig, Germany, March 1994.

Appendix A

```
#ifndef P2C_H
#define P2C_H
#include <stdio.h>
/* If the following heuristic fails, compile -DBSD=0 for non-BSD sys-
tems or -DBSD=1 for BSD systems. */
#ifdef M_XENIX
# define BSD 0
#endif
#ifdef vms
# define BSD 0
# ifndef __STDC__
# define __STDC__ 1
# endif
#endif
#ifdef __TURBOC__
# define MSDOS 1
#endif
#ifdef MSDOS
# define BSD 0
#endif
#ifdef FILE /* a #define in BSD, a typedef in SYSV (hp-ux, at least) */
# ifndef BSD /* (a convenient, but horrible kludge!) */
# define BSD 1
# endif
#endif
#ifdef BSD
# if !BSD
# undef BSD
# endif
#endif
#if (defined(__STDC__) && !defined(M_XENIX)) || de-
fined(__TURBOC__)
# include <stddef.h>
# include <stdlib.h>
# define HAS_STDLIB
# if defined(vms) || defined(__TURBOC__)
# define __ID__(a)a
# endif
#else
# ifndef BSD
# ifndef __TURBOC__
# include <memory.h>
# endif
# endif
# ifdef hpux
# ifdef _INCLUDE__STDC__
# include <stddef.h>
# include <stdlib.h>
# endif
# endif
# include <sys/types.h>
# if !defined(MSDOS) || defined(__TURBOC__)
# define __ID__(a)a
# endif
#endif
#ifdef __ID__
# define __CAT__(a,b)__ID__(a)b
#else
# define __CAT__(a,b)a##b
#endif
#include <ctype.h>
#include <math.h>
# define EXIT_SUCCESS 0
# define EXIT_FAILURE 1
#define Enum
#define SETBITS 32
#define Inline inline
#if defined(hpux) || defined(A2_RS6_AIX32_STD) || defined
(__CLCC__) && !defined (__GNUC__)
#define Signed
#define Volatile
#ifndef hp9000s300
#define Enum
#else
#define Enum enum
#endif
#else
#define Have_Full_Inline
#define Signed signed
#define Volatile volatile
#define Enum enum
#endif
#define Void void /* Void f() = procedure */
#define Const const
typedef char *Anyptr;
#define Register register /* Register variables */
#define Char char /* Characters (not bytes) */
#define Static static /* Private global funcs and vars */
# define Local static /* Nested functions */
typedef Signed char schar;
typedef unsigned char uchar;
typedef int boolean;
#ifndef true
# define true 1
# define false 0
#endif
#ifndef TRUE
# define TRUE 1
# define FALSE 0
#endif
extern int P_ioresult;
#define PP(x) x
#define PV() (void)
#define FileNotFound 10
#define FileNotOpen 13
#define FileWriteError 38
#define BadInputFormat 14
#define EndOfFile 30
#define BUFEOF(f) (__CAT__(f,_BFLAGS) != 2 && P_eof(f))
#define RESETBUF(f,type) (__CAT__(f,_BFLAGS) = 1)
#define SETUPBUF(f,type) (__CAT__(f,_BFLAGS) = 0)
#define Free(p) (free((Anyptr)(p)), (p)=NULL)
#define Malloc(n) (malloc(n) ?: (Anyptr)_OutMem())
extern int _OutMem PV();
extern int _CaseCheck PV();
extern int_EscIO PP( (int) );
extern int P_peek PP( (FILE *) );
extern int P_eof PP( (FILE *) );
extern int P_eoln PP( (FILE *) );
#define FILEBUFNC(f,type) int __CAT__(f,_BFLAGS); \
        type __CAT__(f,_BUFFER)
/* Memory allocation */
#include "adlstr.h"
typedef struct _TEXT{
 FILE *f;
 FILEBUFNC(f,Char);
} _TEXT;
#endif /* P2C_H */
/* End. */
```