# Towards Object Orienting Linux Kernel Structures

**M. Tech. Second Stage Report**

Submitted in partial fulfillment of the requirements
for the degree of

**Master of Technology**

by

**Nishit Desai**
**Roll No: 04305802**

under the guidance of

**Prof. R. K. Joshi**

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

# Acknowledgment

I hereby express my sincere thanks and gratitude towards my guide *Prof. R. K. Joshi* for his constant help, encouragement and inspiration. Meeting with him have been a constant source of ideas, and gave me motivation for this project.

*Nishit Desai*
*IIT Bombay*

# Abstract

Object oriented programming has many advantages over conventional procedural programming languages for constructing highly flexible, adaptable, and extensible systems. Given many benefits of object oriented systems over conventional procedural systems and the rapidly escalating costs of maintenance of systems written in conventional languages, the migration of billions of lines of procedural code into object-oriented languages is an attractive option. This report discusses the problems which occur when linux kernel is compiled by g++. Two different approaches, complete conversion approach and gradual iterative conversion approach, to solve these problems are also discussed. It is also explained why iterative approach is chosen instead of complete conversion approach for our transformation process. The implementation details of conversion tool, which has been built to automatically remove some of the incompatibility problems is also described.

# Contents

# Chapter 1

# Introduction

Prior to the advent of object-oriented paradigm, code was typically organized as a collection of modules, each consisting of a collection of functions. Modules in larger applications were broken into submodules, etc. These in turn were decomposed into functions. Maintenance of these systems became more and more difficult and costly with the passage of time. Re-engineering of the old system with modern software development methodology such as object oriented paradigm was essential.

Object-oriented systems are more adaptable to changes since these systems have their components largely decoupled. Modifying one component does not affect other components of the system. Less time is required to maintain and implement change in a system thereby reducing costs.

In procedural programs it is rather difficult to define the artifacts and their relationships. Re-usability of software is also seriously affected by those features of procedural programs. On the other hand the object oriented paradigm offers some useful characteristics, such as well defined means of abstraction, the concept of encapsulation, or the combination of data and behavior that effectively support the software maintenance process.

Ivar Jacobson and Fredrik Lindstorm [10] describe re-engineering as *" the process of creating an abstract description of a system, reason about a change at the higher abstraction level, and then re-implement the system."*

**Re-engineering = Reverse engineering + Change + Forward engineering**

This project is an aim to re-engineer linux kernel with a complete change in implementation technique but no change in its functionality. Linux kernel is currently implemented in procedural language C. The Change in implementation technique means converting it into Object oriented language C++. Automatic object identification techniques have been proposed in stage I, but we can't directly make the entire kernel object oriented. We have to do this module by module, so that we can check whether the converted module is working correctly or not. For this module by module conversion, either the entire kernel has to be compiled by g++ or there has been some other way by which we can link g++ compiled code with the gcc compiled code. In any of the two approach, we have to make part of the kernel g++ compilable. When we compile kernel by g++, lots of incompatibility problems [8] occur. So, we have to first identify all those incompatibility problems. Because of the enormous size of the linux kernel, we manually can't make entire linux kernel g++ compatible. To remove this problem conversion tool has been made, which automatically removes some of the incompatibility problems. But some part of the tool operates on only preprocessed code, so we can't follow the approach of making entire kernel g++ compatible. This problem can be removed by choosing iterative conversion approach in which at a time only one module has been made g++ compatible, and this g++ compiled code can be linked with other gcc compiled code. After this module has been converted to object oriented form, other module has been taken, made g++ compatible and this process repeats. Part of the conversion tool can also be used in iterative conversion approach to automatically remove some of the incompatibility problems.

## 1.1 Summary of Work Done in Stage I

Existing techniques [7, 12] to identify objects from procedural code were studied in detail. We have also studied [9] the techniques to identify relationships between the objects. Improvements [5] on existing techniques were proposed for identification of objects using access graph. Rule based approach to identify constructor and to associate functions to classes [4] was also proposed. Effectiveness of the proposed method w.r.t. manual object extraction was also discussed.

## 1.2 Organization of Report

The next chapter discusses the problems in getting linux kernel compilable by g++. All incompatibility problems with an examples have been discussed in next chapter. Two different approaches, complete conversion approach and gradual iterative conversion approach, to solve these incompatibility problems are discussed in Chapter 3. Problems with complete conversion approach have also been discussed in that chapter. Chapter 4 explains how iterative conversion approach can be used to solve incompatibility problems. Conversion tool has been built, which automatically removes some of the incompatibility problems. Implementation details of this tool has been given in Chapter 5 with small example. Problems with this conversion tool have also been discussed in Chapter 5.

# Chapter 2

# C/C++ Incompatibilities in Linux Kernel

C++ being a superset language of C, any C program which has been compiled by a C compiler should be readily compiled by a C++ compiler. This is true but for a few exceptions. For example, in C++, typecasting needs to be done explicitly which otherwise would be done implicitly by a C compiler. Other problems that arise are due to the limited feature support of the GNU's g++ compiler as opposed to its gcc compiler. In this section we discuss various problems [14] encountered in getting the linux kernel source compiled by g++. Experiments have been carried out on current stable version of linux kernel, 2.6.14 available at [1].

## 2.1 C++ Reserved Keywords

Many of the C++ reserved keywords have been used in the existing C source of the linux kernel. These reserved keywords are used as normal identifiers in the C code, but on being compiled by the C++ compiler they are granted special meaning and are treated differently. The reserved keywords of C++ extended over C are listed out in 2.1. C code is free to use these keywords as identifiers and macro names. This will cause problems when C code containing these tokens is compiled as C++.

| | | | | |
|---|---|---|---|---|
| throw | bool | catch | class | delete |
| false | friend | virtual | namespace | new |
| operator | private | protected | public | template |
| true | try | using | typename | this |

Table 2.1: Additional Keywords provided in C++ over C

```
static inline void list_add_tail(struct list_head *new,
                                struct list_head *head)
{
       __list_add(new, head->prev, head);
}

a) kernel C source - linux-2.6.14/include/list.h

-----------------------------------------------------------------------


static inline void list_add_tail(struct list_head *new_changed,
                                 struct list_head *head)
{
       __list_add(new_changed, head->prev, head);
}

b) Equivalent C++ Source
```

Figure 2.1: Replacing keywords

The workaround is to rename such conflicting identifiers with ones which are not listed as C++ reserved keywords. Care needs to be taken that the new identifier introduced is not existent in the present scope. Further, we need to examine the impact of the replacement. If the identifier considered for replacement is used within a restricted local scope, the changes to be done are limited. However, if the identifier to be replaced exists globally or if it is part of a definition of a derived type like a struct or enum, then changes need to be propagated to all places where their instances are employed which includes several other source files in the kernel.

## 2.2 Typecasting

### Implicit Typecasting

In C, typecasts may be done implicitly by the compiler. The C++ standard has deprecated this feature and requires typecasts to be done explicitly, which produces better type-safe code. The linux kernel source liberally uses void pointers to handle data and relies on the C compiler to do the appropriate typecasts. Figure 2.2 shows a code fragment of C source and its equivalent source compilable by g++.

```
struct ctl_table_header *tmp;
tmp = kmalloc(sizeof(struct ctl_table_header), GFP_KERNEL);

a) kernel C source - linux-2.6.14/kernel/sysctl.c

-----------------------------------------------------------------


struct ctl_table_header *tmp;
tmp = (struct ctl_table_header *)
                kmalloc(sizeof(struct ctl_table_header), GFP_KERNEL);

b) Equivalent C++ Source
```

Figure 2.2: Explicit Typecasting required in C++

kmalloc allocates required memory and returns a void pointer which is implicitly typecasted by the C compiler to the appropriate type. In order to get the code compiled by g++ we need to explicitly state the type to which the result is to be casted.

#### Enumerated Types and Integers

In C, integer values can be freely used instead of enumerated values for assignment to variables of enumerated type. This is illegal in C++ and requires an explicit typecast. Modifications carried out to get the code compiled under g++ are shown in Figure 2.3.

#### Void pointers in Arithmetic Expressions

C++ does not allow void pointers to participate in arithmetic expressions. In C, arithmetic operations done with void pointers are carried out by implicitly considering them to be of type byte. To have same effect in C++, void pointers are explicitly typecasted to char pointers when used in arithmetic expressions.

## 2.3 Conflicting Names

#### Multiple Declarations

In C, a global data item may be declared several times as long as there is a maximum of one such declaration having an initialization. In C++, global data can be defined exactly

```
enum idle_type
{
      SCHED_IDLE,
      NOT_IDLE,
      ....
};
.....
enum idle_type itype;
itype = 0;

a) kernel C source - linux-2.6.14/kernel/sched.c

-----------------------------------------------------------------------


enum idle_type itype;
itype = (enum idle_type) 0;

b) Equivalent C++ Source
```

Figure 2.3: Typecast for Enumerated type

once. Consider the code fragment in Figure 2.4. C treats the second occurrence of tv1 as a prototype declaration. C++ treats the second occurrence as a new declaration in itself and finds this redeclaration illegal. The conflicting declaration should be removed in order to get the code compiled by g++.

#### Type Name Conflicts

In C, the name of an instance can be the same as the name associated with a type using typedef, provided both do not lie in the same scope. In C++, the instance name cannot be the same as the name assigned to a type with typedef. We require to rename the type or the instance name. Such code fragment is shown in Figure 2.5.

## 2.4 Function Definition Style

In C, an alternate style of function definition may be used, wherein, the argument types are specified after the argument list. This style of function definition is no longer sup-

```
typedef struct tvec_s {                typedef struct tvec_s {
        int i;                                 int i;
} tvec_t;                              } tvec_t;

tvec_t tv1;                            tvec_t tv1;
tvec_t tv1;

a) C source                            b) Equivalent C++ Source
```

Figure 2.4: Multiple Declarations of Global Data

```
typedef int (Func)(int,int);        typedef int (Func_type)(int,int);

struct S {                          struct S {
     Func *Func;                         Func_type *Func;
};                                  };

(a) C Source                        (b) Equivalent C++ Code
```

Figure 2.5: Function Typedef in Name Conflicts

ported in C++. The code can be compiled by adopting the standard style for function definition as shown in Figure 2.6.

## 2.5 Unlocatable Definitions

In C, nested or embedded definitions are visible in the scope where the enclosing data type is defined. In C++, these embedded definitions are not visible. Operations like accessing user-defined enumerated values for such embedded structures fail when applied outside. To solve the problem the embedded definition is to be moved outside the enclosing data structure. As shown in Figure 2.7, it is not possible to carry out sizeof on struct X.

```
int Func(a,b,c)            int Func(int a, float b, char c){
int a;                              ...
float b;                   }
char c;
{
  ...
}

(a) C Source                (b) Equivalent C++ Source
```

Figure 2.6: Old style of Function definition

## 2.6 Non-Trivial Initializers

These are some extensions provided by gcc compiler in addition to the default ways of initializing instances and array of elements. These extensions are not part of ANSI C. Such forms of initializations are provided to ease programming. However, such forms of non-trivial initializations are yet to be featured with g++ compiler. Hence all code using these advanced extended features of gcc will fail to compile under g++ and need to be appropriately recoded.

**Labeled Initializers:**

The label indicates to which element of the structure the immediate value be assigned. The labels and initializing values may be placed in any order and only the elements that need to be initialized are to be specified. If the instance is global the uninitialized values will be set to their default values. Under g++, it is not possible to use the label-initializers in any order. The order must be sane as that in the structure definition. Trailing elements which need not be initialized can be left out, but all elements to be initialized must be ordered. For proper ordering of the initializers, we may have to add some preceding initializing elements with their default values. Figure 2.8 shows a simple instance of this problem.

**Range Index initializers:**

gcc provides range index initializers to initialize a range of elements in an array to a particular value. The values after this specific form of initialization are then assigned to successive indexes. To have the code compiled by g++, we rewrite the code fragment by explicitly specifying the value repeatedly for the entire range and add default values for

```
union U {                          struct X {
    struct X {                         ...
       ...                         };
    } x;                           union U {
    ....                                struct X x;
};                                      ...
                                   };
size_X = sizeof(struct X);         size_X = sizeof(struct X);


(a) C Source                       (b) Equivalent C++ Source
```

Figure 2.7: Inaccessible Embedded definition

element s preceding the range. For example, int a[20]={[1...5]=1,5} is to be recoded as int a[20]={0,1,1,1,1,1,5,}.

**Index Initializers:**

Another form of array initialization provided by gcc is for assigning a value to a specific element by indicating its index. The compiler calculates the size of the array from the highest index specified. To code without using this extension, we need to calculate the size of the array, assign the correct value from the mentioned index and for the values unspecified default values are to be assigned. For example, int A[]={[2]2,[4]10} is to be recoded as int A[5]={0.0.2.0.10}.

## 2.7 Inbuilt Macros

gcc defines macros which can be used for debugging and verbose printing. These macros are used in print and other string functions. For example, __FUNCTION__, __LINE__, etc. The macro __FUNCTION__ evaluates to a string which indicates the name of the current function. In Figure 2.9, the string evaluates to *"this is funcA now"*. In g++, these is inbuilt macros are not defined and left as unresolved symbols.

A workaround is to define these inbuilt macros in places where they are used. We can define __FUNCTION__ as a null string with the loss of debugging information. The other option is to replace each occurrence of __FUNCTION__ with the function name in which it is embedded. However, this may not be possible cause several times __FUNCTION__ macro itself is part of a user defined macro.

```
struct A {                        struct A {
    float f;                          float f;
    char c;                           char c;
    int i;                            int i;
};                                };

struct A alpha = {c:'a'};         struct A alpha = {f:0.0,c:'a'};


(a) C Source                     (b) Equivalent C++ Source
```

Figure 2.8: Labeled Initializers

```
int funcA()
{
    printf("this is" __FUNCTION__"now");
}
```

Figure 2.9: Inbuilt Macro supported by gcc

## 2.8 Jump Crosses Initialization

In C++, a jump cannot be made crossing initializing statement. To circumvent this problem we have to split initializing statement into declaration and initialization. This scenario is depicted in Figure 2.10.

```
goto xlabel;                      goto xlabel;
if(condition)                     if(condition)
    int y = 0;                        int y; y =0;
    xlabel:                           xlabel:
    ...                               ...
};                                };

(a) C Source                     (b) Equivalent C++ Source
```

Figure 2.10: Jump Crosses Initialization

# Chapter 3

# Approaches to Solve Incompatibility Problem

Our aim is to convert linux kernel, which in its present form written in C language, to an object oriented form using C++. In practice, for small applications written purely in C code, it is possible to get the entire application running under a C++ compiler without many changes. However, the same does not hold for the linux kernel, which is a large system software comprising of several source files, designed to run on various computer architectures. The kernel, although mainly written in C, has assembly code written for platform specific functionalities. C++ is a superset of the C language with a few exceptions. A few features of C have been deprecated in C++ to ensure safer programs. Also many of the extensions provided by gcc are not supported by g++. Due to these issues, it is difficult to get the linux kernel built by a C++ compiler.

Work has been carried out on current stable version of linux kernel, 2.6.14. The kernel has been compiled [11] for Pentium IV architecture. In this chapter, we assess the different approaches [8] that could be used to inject C++ code into existing linux kernel. Presently the entire linux kernel, comprising of C source and assembly language (architecture dependent) code, is built using gcc with C style linkage. Two different approach, which is tried out for getting the linux kernel compiled by g++, are discussed.

## 3.1 Complete Conversion Approach

In this approach, we directly compile the entire linux kernel with g++ and link the object files in the default C++ style and then employ object identification technique to object orient the parts of the kernel. This process requires recoding various code fragments in almost all source files to circumvent the problems discusses in previous chapter. Manually removing these problems is very time consuming, and whenever some new software comes, we have to remove all these problems by hand. The enormous size of the kernel source leaves this approach undesirable to follow. So, tool has been made which automatically removes some of problems such that converted code has minimal incompatibility with g++ compiler. Detailed discussion on the tool has been carried out in Chapter 5. It has been found that applying this tool on linux kernel is also undesirable, because any tool like this operates on preprocessed code. Experiment shows that after preprocessing, kernel code become so large that applying this tool will also take lots of time. So, we can't apply this approach on linux kernel. Adding to this, there is one another linking problem with this approach. This problem is described in next subsection.

### 3.1.1 Linking Problem

Another problem with this approach is that we have to handle linking issues between object code of c++ source and object code of the assembly source. It is tedious to link between c++ code and assemble code due to name mangling done by the c++ compiler. Name mangling is done to support features like overloading of functions and may differ from compiler to compiler. Name mangling as done by the GNU C++ compiler is shown in Table 3.1. The mangled function name carries information about the parameter list of the function. For example, *funcB(char c, int i)* is represented as *funcB__Fci*, 'F' meaning that funcB is a function, which takes arguments of type char encoded as 'c' and of type int encoded as 'i'.

| Program | Unmangled (C style) | Mangled (C++ style) |
|---|---|---|
| int funcA() { } | funcA | funcA__Fv |
| int funcB(char c, int i) { } | funcB | funcB__Fci |
| int funcC(void *vp) { } | funcC | funcC__FPv |
| main() { } | main | main |

Table 3.1: Name Mangling done by GNU C++ compiler

If C++ style linking is to be used, the assembly language code must refer to the mangled form of symbols defined in C++ source. Also, for the C++ source to refer symbols in the assembler code, it must be ensured that the reference is done using the unmangled form.

These problems are mainly with linux kernel. Because of these problems we couldn't continue with this approach on linux kernel. This approach might be applied on some other software, which doesn't contain assembly language code.

## 3.2 Gradual Iterative Conversion Approach

With this approach, it is possible to select a small portion of the kernel, compile it using g++ and link it with the existing gcc compiled kernel. The code to be compiled with g++ should use C style linking instead of the default C++ style. This approach restricts number of source files those need recoding in order to be g++ compilable. So, currently this approach is adopted for our transformation process. Figure 3.1 visualizes the conversion process for a selected C source file of linux kernel. Only the selected source files and the header files used by it are to be made compatible for g++. Since C style linking is used, the object code of newly compiled C++ code can be linked with the existing object files of both C source and assembly source without many changes. C style linking is specified by explicitly enclosing the C++ source within *extern "C"*. Detailed discussion on *extern "C"* is done in next chapter.

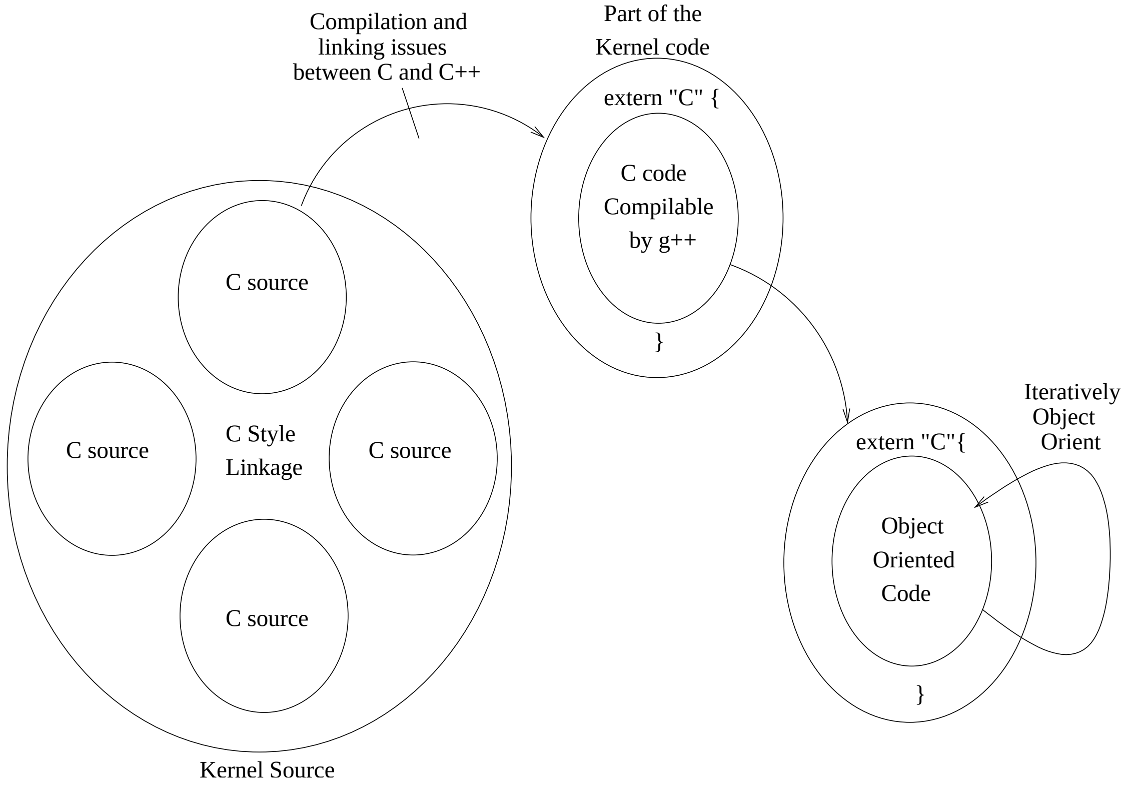

Figure 3.1: Gradual Iterative Conversion

Code compiled by g++ can now be object oriented using the object identification techniques. This approach is applied iteratively, gradually, converting the entire kernel to object oriented code. The C style wrapper shell around each c++ code source can then be removed in the final phase as all sections of the kernel can refer mangled names. The C style wrapper shell will have to retained between C++ source and assembly code.

# Chapter 4

# Iterative Conversion Approach

With this approach, small portion of the kernel is selected at any time. This is the portion which we currently want to make object oriented. Now, only this portion is made to be g++ compatible and compiled by g++ compiler, and it is linked to the rest of the kernel which is compiled by gcc compiler. The code to be compiled with g++ compiler should use C style linking instead of the default C++ style. This is required because if we can't specify C style linking to g++ compiler, then it mangles the function name and in that case we can't link g++ compiled code with the gcc compiled code. In C style linking, function names are not mangled by g++ compiler and retains in same form as by gcc compiler. *extern "C"* is used for specifying C style linking to g++ compiler.

This approach restricts number of source files those need recoding in order to be g++ compilable. So, currently this approach is adopted for our transformation process. Code compiled by g++ can now be object oriented using the object identification techniques. Using this approach, gradually entire kernel can be made object oriented by changing only one module at a time and without touching other part of kernel. Figure 3.1 shows the conversion process by this approach. In this chapter, we will discuss how extern "C" [6] can be used to specify C style linking. It has been also discussed that how final linking is done between g++ compiled code and gcc compiled code.

## 4.1 Accessing C Code From C++ Source

In this section, it is discussed that how C code can be accessed from C++ code. This is possible because C++ language provides a linkage specification with which we can declare that a function or object follows the linkage conventions for a some other supported language. The default linkage for objects and functions is C++. G++ compiler also support C linkage, for gcc C compiler.

To access C code from C++ code, we have to specify C style linkage for the C code which is accessed from C++ code. For example, to access a function compiled by the C compiler, function has to be declared to have C linkage. Following notations can be used

```
extern "language_name" declaration;
extern "language_name" {declaration; declaration;}
```

Figure 4.1: Declaring Linkage Specifications

to declare linkage specifications.

First notation indicates that declaration has linkage of *language_name*, while second notations indicates that everything between the curly braces has linkage of *language_name*. To be more clear about how to access C code from C++ code consider one example. Consider that the file *kernel.c* is going to be compiled by gcc compiler, and file *new.cpp* will be compiled by g++ compiler. File *new.cpp* is also accessing some function defined in *kernel.c*

```
int temp(int i){
    ...
}

int main(){
   ...
}

a) C source - kernel.c

extern "C" {
     int temp();
}
int C_access(){
       temp(10);
       any C++ code
}

b) C++ source - new.cpp
```

Figure 4.2: Accessing C code from C++ code

Here, in this example temp() is defined in C file and it is compiled by gcc compiler. Now, C++ code is accessing this temp() function, and C style linkage has been specified for temp function. So, when g++ compiler compiles *new.cpp* file, it will not mangle temp.

If C style linkage is not specified for temp function in *new.cpp*, then g++ compiler uses mangled name for calling temp function, and gcc hasn't mangled the temp function, while compiling temp definition. So, error is generated if we don't specify C style linkage for temp function. Commands to run above program are as follows:

```
gcc -c kernel.c
g++ -c new.cpp
g++ kernel.o new.o -o out
```

Here final linking has to be done by g++ compiler. If gcc is used for final linking than error is generated. In reverse case when we are accessing C++ code from C code this final linking can also be done by gcc compiler.

## 4.2 Accessing C++ Code From C Source

Same method which was discussed in last section can be used to access C++ code from C code. In this case C style linkage specification is provided for function which is defined in C++ code, but access by C code. Consider the same example, which was discussed in last section, with minor changes. In this case consider that temp is defined in *new.cpp* and, main function, which is defined in kernel.c, is using temp function. So, we have to specify C style linkage for temp function in *new.cpp*.

```
int main(){
   temp(10);
   ...
}

a) C source - kernel.c

extern "C" {
     int temp();
}
int temp (int i){
     any C++ or C code
}
b) C++ source - new.cpp
```

Figure 4.3: Accessing C++ code from C code

If we don't specify C style linkage, then error is generated. Same kind of reasoning which is given in last section can also be applied here. Same set of three commands can be used to compile and link the above program. In this case final linking can also be done by gcc instead of g++. Using same method, we can also write the code in which C code is accessing some C++ function, and C++ code is accessing some C functions.

If there is some need to use extern "C" in header file and if this header file is used in both C code and C++ code, then we have to use __cplusplus macro to eliminate this problem because "extern C" can not be used in C code. Consider following header file

```
#ifdef __cplusplus
extern "C" {
#endif
... /* body of header */
#ifdef __cplusplus
} /* closing brace for extern "C" */
#endif

a) header file - c_cplusplus.h
```

Figure 4.4: Mixed Language Header

If this header file is included in C code then extern "C" will not come in that file, but if it is included in C++ file then extern "C" will be included in that file because __cplusplus macro is by default defined for c++ file.

We can make analogy of linux kernel with this kernel.c and new.cpp. Consider that kernel.c is the portion of linux kernel which is compiled by gcc compiler, and new.cpp is the portion of the kernel which we have made g++ compilable.

Experiments has been done by making the file *msgutil.c* of the ipc kernel code to g++ compatible. Using this iterative approach, only *msgutil.c* and header files, which are included in *msgutil.c*, need to be changed. So, this portion becomes our new.cpp and rest is kernel.c. Conversion tool, which is described in next chapter, can be used to make these files g++ compatible.

# Chapter 5

# Design and Implementation of Conversion Tool

Whether we use complete conversion approach or iterative conversion approach, manually removing all incompatibility problems is undesirable and time consuming. To avoid this manual conversion, tool has been made which automatically removes some of the incompatibility problems. Some problems are easily removed without parsing the source code, simple python scripts are used to remove those problems. Some other problems are removed by parsing and changing the the part of the code, which contains incompatibility and other part of the code is outputted as it is. For remaining problems entire code needs to be parsed. For this entire gcc C grammar is needed. ANTLR (ANother Tool for Language Recognition) is used to for this purpose. In this chapter, first ANTLR [13] is explained and then implementation details of tool is explained. Problems with tool have also been discussed.

## 5.1 ANTLR

ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. ANTLR is nearly same as Lex (lexer) and Yacc (parser). It accepts grammatical language descriptions and generates programs that recognize sentences in those languages. ANTLR knows how to build recognizers that apply grammatical structure to three different kinds of input:

- character streams
- token streams
- trees structures

These correspond to lexers, parsers, and tree walkers. The syntax for specifying these grammars, the meta-language, is nearly identical in all cases.

All ANTLR grammars are subclasses of Lexer, Parser and TreeParser. For better understanding consider the simple calculator program. Suppose we have to write parser for calculator and we want to use java as our language for building parser, then following is the code for parser and lexer. Grammar for any parser is written in class which is extended

```
class ExprParser extends Parser;

expr:   mexpr ((PLUS|MINUS) mexpr)*
    ;
mexpr
    :   atom (STAR atom)*
    ;
atom:   INT
    |   LPAREN expr RPAREN
    ;

class ExprLexer extends Lexer;

options {
        ...
}

LPAREN: '(' ;
RPAREN: ')' ;
PLUS  : '+' ;
....
```

Figure 5.1: Sample example for parser and lexer in ANTLR

from Parser class. In above example, grammar is written in ExprParser which is extended from Parser. In the same way, grammar for lexer is written in class which is extended from Lexer class. Now, in main function we will create the object L of ExprLexer, and then object P of ExprParser is created. L is passed as argument to P because parser needs tokens from lexer.

To evaluate the expression we can specify action corresponding to each rule in ExprParser class. Instead of specifying actions which evaluate input expression, we can also specify actions which generate syntax tree from expression. Syntax tree is an intermediate representation that holds all or most of the tokens and relationships between those tokens. If syntax tree has been built in parser, then we have to write treeparser which parses tree and does some actions, in our case actions are evaluation of expression. Grammar for treeparser can be written in class extended from TreeParser. Following is the skeleton class for ExprTreeParser which parses the tree generated by ExprParser and evaluates the expression embedded in tree.

```
class ExprTreeParser extends TreeParser;

options {
....
}

rules to parse the tree and corresponding actions
```

Figure 5.2: Sample example for treeparser

Options section specifies the bunch of command line arguments to parser generator. An options section may be specified on a per-file, per-grammar, per-rule, and per-subrule basis.

This tool has been chosen for our purpose instead of lex and yacc because it contains gcc C source to source translation [15] framework . This framework contains gcc C parser and tree emitter. Gcc C parser builds syntax tree, and gcc tree emitter prints source code by parsing the tree generated by parser. For our purpose, we have to change gcc emitter part such that output generated by it should not contain g++ incompatibility problems.

## 5.2 Implementation Details of Conversion Tool

This section describes implementation details of tool, which removes some of the g++ incompatibility problems. For some problems simple python script is used, while for some other problems small grammar is written in lex and Yacc, and remaining problems are solved using grammar written in ANTLR. Not all problems described in chapter 2 are removed using this tool, but most of the problems are resolved. For each problem we will describe how that problem is sovled. All parts of the tool are not merged into one part. For each problem, corresponding program has to be run to remove that problem.

### Reserved Keywords

To remove this problem, python script has been written. This script replaces keyword by some other name, if that keyword is found as an identifier and if it exists in Table

2.1. For example, if keyword new is found as an identifier, it is replaced by new_changed. This program recursively goes into each directory, replaces the keyword in input files and generate new files. We can give regular expression for file names which needs to be changed. For example, "*.c" is given if we wants to modify only C files.

#### Function Definition Style

For this problem also python scripts [14] has been written which changes old function definition style to a new function definition style. This program parses the source code. All parts other than the old function definitions are outputted as it is to a new file. When old function definitions are found, it parses the source code until first left curly brace is found. After the left curly brace is found, it generates new function definition using its arguments and outputs it to a new file.

#### Unlocatable Definitions

For this problem, lex and yacc is used. Simple grammar is written in Yacc to parse the struct, union or enum structure. When one of these construct is found, some action has been taken. Other code is outputted as it is to a new file. Stack is maintained to determine where currently we are. If stack contains no element, then we are outside the structure, else we are inside some structure. Length of stack determines the level up to which source code is parsed. When structure definition ends, and if it is the inner structure, then there is a need to bring this inner structure from inside to outside. So, it is outputted to a new file and it is followed by a semicolon. If it is top level structure then it is not followed by semicolon, because semicolon for this top level structure comes from input file.

#### Range Index Initializers

This problem is also solved using lex and yacc. Grammar, which parses the array initialization containing range index or index, is written in yacc. Other part of the C source code is outputted as it is to a new file. Here also stack is used as a data structure. Each entry of stack contains structure containing lower index, upper index and value for this index range. Each element of initialization is pushed on to a stack. After initialization completes, stack is used to find the highest index which is initialized. Value corresponding to each index is also found using stack elements. After doing this, values for zero to highest index are outputted to a new file.

### 5.2.1 Problems solved using ANTLR

For all the following problems, ANTLR gcc C source to source translator is used. In which, first the parser generates the syntax tree from C source code, then emitter generates same source code (with different spacing) from the tree. So, for changing particular part of the source code, we have to first identify the rules, which match to that particular part. In this translator, we can also print the entire tree generated by parser. This tree contains tokens, corresponding code part and rules which match to that part. So, for any part of the source code we can easily find the rules. Only actions of those rules need to be changed to change that part.

### Jump Crosses Initialization

To solve this problem, rule which matches to *goto* statement has been found out using syntax tree. Then some flag is turned on in actions of this rule. After this, whenever some initialization statement occurs, we will first see that whether that flag is on or off. If it is on then we split that initialization statement, otherwise nothing has to be done. When label is found we will turned off the flag in the action of that rule.

### Implicit Typecasting

For solving this problem, we required the types of each identifiers. So, when some declaration/definition occurs we store variable name and the corresponding type on stack. When new scope starts, some marker has been pushed on stack, and when that scope completes, all the items can be popped from the stack up to that marker. Rule corresponding to binary expression (a = b) is found, and stack is searched for identifier which occurs on left side of expression. Suppose type T is found for a, then actions are changed such that statement (a = b) becomes (a = ((T)b). Not all typecasting errors are removed by this method. Most of the typecasting errors occur because of malloc and kalloc functions which return void pointer. That kind of errors are removed by this method. For complete solution, all kind of statements in which typecasting errors may occur have to be identified and then, actions of all rules which match to those statements have to be changed to remove typecasting errors. In practice, most of the typecasting errors occur because of assignment problems. So, only that problem is solved automatically, but it can be extended to remove other kind of typecasting errors.

### Void pointers in Arithmetic Expressions

This problem is solved by finding rules for arithmetic expression. Then in action of that rule, when some identifier I occurs, symbol table is searched for I. If void * is found as type of I, then action is changed such that identifier I is replaced by ((char *)I) in output.

Problem for inbuilt macros can be easily solved by putting all macros in one header file and including this header file in all other header files. Multiple declaration problem is also easily solved by searching the symbol table for each new global declaration and definitions. Other problems has been not solved automatically, because those problems occur rarely in system. So, those kind of problems have to be removed by hand.

## 5.3 Example: Test Case

This tool has been applied on many sample example codes. For now, consider one simple example given in Figure 5.3. This example is not taken from kernel code, but it is built such that it contains lots of code which is not allowed by g++ compiler. If we apply our tool on this code, then tool removes all the incompatibility present in the code. We have not merged the set of tools into one tool, so one by one we have to apply all the tools. Output of one tool is given as input to next tool. If we apply following set of tools in the given sequence then code given in Figure 5.4 is generated.

- Apply script which changes name of the identifier, whose name is same as one of the reserved keywords given in Table 2.1.
- Run program which converts old function definition style to a new function definition style.
- Apply tool which solves embedded structure definition problem. This program brings out inner structure definitions from inside to outside of outermost structure.
- Apply tool which solves range index initialization and index initialization problems.
- Apply tool which has been built using ANTLR. This tools removes problems like implicit typecasting, jump crosses initialization and void pointers in arithmetic expressions. This ANTLR based tool can only be applicable on preprocessed code. This problem is discussed in detail in next section.

This generated code doesn't contain any incompatibility. It can be compiled by g++ compiler directly without more changes.

```
#include<stdlib.h>
struct Outer{
       int i;
       struct A{
               int j;
       }cd;
};
enum class { off , on };
enum class bulb;
typedef struct A * AP;
static void test (a)
int a;
{
       bulb = a;
}
int main () {
        int i=0;
        int y=10;
        void *v;
        goto label;
        if (i<10) {
                float y = 0,ft=0;
                char *a [10] = {[1]"ab","de",[4]"cd"},d = 'b';
                struct A ***ab,*ef;
                AP fk;
                struct A *cd = malloc(sizeof(struct A));
                v = &y;
                fk = malloc(sizeof(struct A));
                v = v+5;
                label:ef = malloc(sizeof(struct A));
                **ab = ef;
        }
}
```

Figure 5.3: Sample code containing g++ incompatibility problems

## 5.4 Open Issues

There are some problems with this tool. Tool like ANTLR can only be applied on preprocessed code. This conversion tool is also using ANTLR to remove some of the problems.

So, part of tool which is using ANTLR can only be applied on only preprocessed code. If we apply this conversion tool on normal code, then ANTLR part of this tool generates parsing error. Detail discussion on this problem is given in following subsection. Because of enormous size of kernel, we can't apply this part on preprocessed kernel code. But for other softwares, which are not as big as linux kernel, this tool can be applied on preprocessed code.

### 5.4.1 Preprocessing problem

Tool like ANTLR, which converts one form of the source code to another form, has to be applied on preprocessed form of the source code. If it doesn't applied on preprocessed code, then when some macro occurs in the code, no rule is matched to part of the code and ANTLR gives parse error. To understand this consider the following example.

```
#define int X;
X a;
```

In this example, no rule of C grammar will match to statement *X a*, because X is neither a built in data type nor it is a type define by typedef. So, ANTLR generates parse errors for this statement. Two different approaches to solve this problem have been tried.

#### Search a Macro table when Parse error occurs

First approach is to build a macro table from macro statements. Now, when ANTLR generates some parse errors, search macro table for that error generating token and find a new token. Then, give this new token to ANTLR and start parsing from the previous state at which error occurred. If above example is considered, then error is generated for token corresponding to X. If macro table is searched for X, then int is found as new token. If int is given to ANTLR and ANTLR starts parsing from previous state with int as new token, it correctly parse the above code. It is tried to change the ANTLR code to implement this solution. But it is concluded that it is always not possible to find a exact token which has generated the parsing error.

#### Prepossess, Do changes and Go back to Original Form

Second approach is to prepossess the entire code, do changes on this prepossess code and then go back to original form. For doing this we have to remember all the places where the preprocessor has done replacement. For this, code of the preprocessor needs to be changed. In this approach, conversion tool has to decide that where to make changes. Because it is also possible that macro statement itself contains g++ incompatibility. In that case, changes in macro statement has to be done. In some other cases we can't change the macro statement, we have to change portion of the code where the macro replacement has been done by preprocessor. Implementation of this issues itself is a big problem and it is not our current focused problem.

We couldn't follow these two approaches because of the problem discussed above. So, the approach is to apply conversion tool on preprocessed code without worrying about the original form, because at last we want the kernel executable file.

For application of this conversion tool on linux kernel, ANTLR gcc c grammar has been changed. ANTLR gcc C grammar [15] was written before five years, and in this time gcc [2] might have added some syntax. So, ANTLR grammar is changed to meet this requirements. Currently this grammar is able to parse kernel preprocessed code.

Experiments have been done on linux kernel and result shows that after preprocessing, kernel code became 10 to 15 times larger than the original code. So, application of conversion tool on preprocessed kernel code was taking lots of time, and because of this we couldn't apply this ANTLR based tool on preprocessed kernel code. But part of the tool which doesn't use ANTLR has been applied on ipc part of linux kernel (msgutil.c) to make it g++ compatible. All parts of this conversion tool can be used on small scale, where software is not big as linux kernel after preprocessing.

As current status, we are able to compile *msgutil.c* by g++ and this file is linked with some other gcc compiled kernel code, but it is not linked with entire remaining kernel. For this, some linking command has to be changed in kernel [3] makefiles. This problem occurs because kernel uses ld command for final linking, but we require final linking by g++ as described in previous chapter.

```
#include<stdlib.h>
struct A {
       int j ;
};
struct Outer {
       int i;
       struct A {
               int j;
       } cd;
};
enum class_changed { off , on };
enum class_changed bulb;
typedef struct A * AP;
static void test (int a)
{
       bulb = (enum class_changed)(a);
}
int main ( ) {
       int i = 0;
       int y = 10;
       void * v;
       goto label;
       if (i<10) {
          float y ; y = 0 ; float ft ; ft = 0;
          char * a[10] ; a[0] = 0 ; a[1] = "ab" ; a[2] = "de";
          a[3] = 0 ; a[4] = "cd" ; char d ; d = 'b';
          struct A ***ab , *ef ;
          AP fk ;
          struct A * cd ; cd = (struct A *)(malloc (sizeof (struct A)));
          v = (void *)(&y) ;
          fk = (struct A *)(malloc (sizeof(struct A)));
          v = (void *)(((char *)v) + 5);
          label : ef = (struct A *)(malloc (sizeof (struct A)));
          **ab = (struct A *)(ef);
       }
}
```

Figure 5.4: Sample code without g++ incompatibility problems

# Chapter 6

# Conclusion and Future Work

Problems which occur when linux kernel is compiled by g++ have been identified. Two different approaches, complete conversion approach and iterative conversion approach, were attempted to solve the incompatibility problems. The method for performing iterative conversion has been developed and explained in the report. It has also been explained why iterative approach is chosen instead of complete conversion approach for our conversion process.

In this stage, a conversion tool which automatically removes some of the incompatibility problems has been built. It has been discussed in the report why all parts of conversion tool cannot be applied on linux kernel. Experiments have been done on linux kernel and part of the ipc kernel code has been made g++ compatible.

The following table shows the work done in stage I and II, and it also shows work which need to be done in stage III.

| | |
|---|---|
| stage I | Existing techniques for automatic object identification from procedural code were studied, and improvements on these techniques were proposed |
| stageII | Conversion tool has been built which is useful for making linux kernel g++ compatible |
| stage III | Manually or by applying tool, methods proposed in stage I need to be applied on part of linux kernel after making it compatible through work done in stage II |

Table 6.1: Work distribution on all stages

The future work involves doing the following things:

**Tuning the tool for linux kernel:** A tool which removes some of the g++ incompatibility from C code was attempted, but it is not fully built. There are some bugs in the tool. So, we need to improve this tool to remove those bugs and add more functionalities to the tool. Currently we are able to compile small portion of linux kernel with g++ and this portion is linked with some other gcc compiled kernel code, but it is not linked with entire remaining kernel. For this, some linking command has to be changed in kernel makefiles. [3] This problem occurs only when C++ code calls a C function, and not vice versa. The problem occurs because kernel uses ld command for final linking, but we require final linking by g++.

**Objectifying part of linux kernel:** We need to improve the object identification techniques proposed in stage I and apply these techniques on parts of linux kernel. Currently we are considering ipc part of linux kernel for this purpose. We need to build a tool which proposes candidate objects and relationships between them, associates functions to classes and also gives coupling and cohesion metrics values for each class. These values are useful in proposed object identification techniques.

**Dynamic memory allocation for objects:** When we are objectifying any portion of procedural software, we have to do dynamic memory allocation for objects which are created at runtime. In normal software this can be done using *new*. But kernel doesn't use any library functions like malloc, printf, so we cannot use c++ *new* function directly. For memory allocation, kernel has its own *kalloc* function. So, for dynamic creation of objects we need to build our own *new* function using *kalloc*.

# References

[1] The Linux Kernel Archives. Website. `http://www.kernel.org/`.

[2] Using the GNU Compiler Collection. Website, 2003. `http://www.delorie.com/gnu/docs/gcc/gcc_55.html`.

[3] Kernel Makefile Documentation. Website, February 4, 2003. `http://lwn.net/Articles/21835/`.

[4] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 259–262, New York, NY, USA, 1995. ACM Press.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[6] Stephen Clamage. Mixing C and C++ Code in the Same Program. Website, 1998. `http://developers.sun.com/prodtech/cc/articles/mixing.html`.

[7] Jean-Francois Girard and Rainer Koschke. A comparison of abstract data types and objects recovery techniques. *Sci. Comput. Program.*, 36(2-3):149–181, 2000.

[8] Anil Gracias. Towards Re-engineering of Linux Kernel from Procedural to Object-Oriented. Master's thesis, Department of Computer Science, Indian Institute of Technology Bombay, 2002.

[9] Maarit Harsu. Identifying object-oriented features from procedural software. *Nordic J. of Computing*, 7(2):126–142, 2000.

[10] Ivar Jacobson and Fredrik Lindstrom. Reengineering of old systems to an object-oriented architecture. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 340–350, New York, NY, USA, 1991. ACM Press.

[11] R. Krishnakumar. Compiling the Linux Kernel. Website, 1998. `http://linuxgazette.net/111/krishnakumar.html`.

[12] Sying Syang Liu and Norman Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *ICSM '90: Proceedings of the IEEE International Conference on Software Maintenance*, pages 266–271, San Diego, CA, USA, 1990. IEEE Computer Society.

[13] Terence Parr. Antlr Reference Manual. Website, 2005. `http://www.antlr.org/doc/index.html`.

[14] Ashish Vanarse. Techniques for Re-engineering Procedural C code to Object Oriented C++ Code. Master's thesis, Department of Computer Science, Indian Institute of Technology Bombay, 2005.

[15] Monty Zukowski and John D. Mitchell. Grammar for GNU C Compiler. Website, April 28, 1998. `http://www.antlr.org/grammar/cgram/grammars/`.