GNU C Compiler Internals
On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. A URI to this license is given in the list of figures on page 63. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 61. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 67, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 63. This PDF was generated by the \LaTeX typesetting software. The \LaTeX source code is included as an attachment (source.7z.txt) in this PDF file. To extract the source from the PDF file, you can use the pdfdetach tool including in the poppler suite, or the http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/ utility. Some PDF viewers may also let you save the attachment to a file. After extracting it from the PDF file you have to rename it to source.7z. To uncompress the resulting archive we recommend the use of http://www.7-zip.org/. The \LaTeX source itself was generated by a program written by Dirk Hünniger, which is freely available under an open source license from http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf.
Contents

1 GNU C Compiler Internals/Print version 3

2 Introduction 5
  2.1 What is GCC? .................................................. 5
  2.2 History of GCC ................................................ 5
  2.3 Purpose of this book ......................................... 6

3 GNU C Compiler Architecture. Compiling an Expression 7

4 Compilation of a function 21
  4.1 Function Prolog and Epilog ............................... 21
  4.2 Local Control Flow Analysis .............................. 22

5 Function calls 23
  5.1 Global Control-Flow Analysis ............................. 23
  5.2 Parameter Passing ........................................... 23

6 Stackguard 27

7 GEM Framework 29

8 C Function Overloading 35
  8.1 Function overloading in C ................................. 35

9 Return Address Defense 37
  9.1 Return Address Defense ..................................... 37

10 Adding Syntactic Sugar 39
  10.1 toString() method for each structure as in Java .... 39
  10.2 Invoke a block of code from a function as in Ruby .. 39
  10.3 Dereference function results when a structure is returned 39
  10.4 Use functions to initialize a variable .................. 40
  10.5 Default values of function arguments as in C++ .... 40
  10.6 Reference parameters as in C++ ......................... 40
  10.7 GCC switches in object file .............................. 40
  10.8 Type information at run-time ........................... 40

11 Improving Code Style 43
  11.1 Break up enormous source files ......................... 43
  11.2 Break up enormous functions ............................ 43
  11.3 Break up enormous conditionals ....................... 44
### 11.4 Delete garbage

11.5 Use predicates for RTL objects

### 12 Security Enhancements

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return address protection (RAD)</td>
<td>45</td>
</tr>
<tr>
<td>Repair of control-hijacking attacks (DIRA)</td>
<td>45</td>
</tr>
<tr>
<td>Array bounds checking using segmentation hardware (CASH)</td>
<td>45</td>
</tr>
<tr>
<td>Detecting integer overflows (DIVINE)</td>
<td>45</td>
</tr>
<tr>
<td>Dynamic Information Flow Tracking (GDIF)</td>
<td>45</td>
</tr>
</tbody>
</table>

### 13 Links

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
</tr>
</tbody>
</table>

### 14 Contributors

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
</tr>
</tbody>
</table>

### 15 GNU Free Documentation License

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. PREAMBLE</td>
<td>51</td>
</tr>
<tr>
<td>1. APPLICABILITY AND DEFINITIONS</td>
<td>51</td>
</tr>
<tr>
<td>2. VERBATIM COPYING</td>
<td>53</td>
</tr>
<tr>
<td>3. COPYING IN QUANTITY</td>
<td>53</td>
</tr>
<tr>
<td>4. MODIFICATIONS</td>
<td>53</td>
</tr>
<tr>
<td>5. COMBINING DOCUMENTS</td>
<td>55</td>
</tr>
<tr>
<td>6. COLLECTIONS OF DOCUMENTS</td>
<td>55</td>
</tr>
<tr>
<td>7. AGGREGATION WITH INDEPENDENT WORKS</td>
<td>56</td>
</tr>
<tr>
<td>8. TRANSLATION</td>
<td>56</td>
</tr>
<tr>
<td>9. TERMINATION</td>
<td>56</td>
</tr>
<tr>
<td>10. FUTURE REVISIONS OF THIS LICENSE</td>
<td>57</td>
</tr>
<tr>
<td>11. RELICENSING</td>
<td>57</td>
</tr>
</tbody>
</table>

### 16 How to use this License for your documents

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>59</td>
</tr>
</tbody>
</table>

### 17 Contributors

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
</tr>
</tbody>
</table>

### List of Figures

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
</tr>
</tbody>
</table>

### 18 Licenses

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU GENERAL PUBLIC LICENSE</td>
<td>67</td>
</tr>
<tr>
<td>GNU Free Documentation License</td>
<td>68</td>
</tr>
<tr>
<td>GNU Lesser General Public License</td>
<td>69</td>
</tr>
</tbody>
</table>
1 GNU C Compiler Internals/Print version

1. Introduction
2. GNU C Compiler Architecture. Compilation of an expression
3. Compilation of a function
4. Function calls
5. Stackguard
6. GEM Framework
7. Function Overloading in C
8. Return Address Defense
9. Adding Syntactic Sugar
10. Improving Code Style
11. Security Enhancements
12. Links
13. Contributors

---

2 Introduction

2.1 What is GCC?

GNU Compiler Collection\(^1\) (GCC) is a free software project that includes compilers for Ada\(^2\), C\(^3\), C++\(^4\), Fortran\(^5\), Java\(^6\), and Objective-C\(^7\), as well as libraries for these languages. It is capable of generating executables for a variety of platforms\(^8\) including x86\(^9\), ARM\(^10\), MIPS\(^11\), PowerPC\(^12\), etc.

2.2 History of GCC

The homepage of GCC is \(^{13}\). The modern history of GCC begins with the GCC 2.95 release. This version was released on July 31, 1999. GCC 3.0 which is considered modern history for the C++ compiler was released on June 18, 2001. Additional branches were created later on. As of now, the active development branches are GCC 3.4 with a latest release on November 30, 2005, and GCC 4.0 released last time on September 28, 2005. GCC 4.1 was released on Feb 28 2006. GCC 4.2 is the development branch of GCC. The source code repository is available online\(^{14}\). GCC 4.2 (the trunk) is the only place where real development happens and new features can go. GCC has become popular in industry and academia. The availability of its source code allows one to add new features to the compiler. GCC is used in several source-code based security projects, that is, in the tools that instrument the source code of the program to make it more secure. However, very few documents describing the GCC internals have been published so far. When new functionality is implemented, the source code of GCC is modified directly. However, these compiler extensions are difficult to distribute because GCC is a really big program. A framework for creating modular extensions would greatly simplify the development of compiler extensions. GCC 4.0 and above includes SSA optimizers and should be used for further high level

\(^{1}\) http://en.wikibooks.org//en.wikipedia.org/wiki/GNU_Compiler_Collection
\(^{5}\) http://en.wikibooks.org//en.wikipedia.org/wiki/Fortran
\(^{8}\) http://gcc.gnu.org/install/specific.html
\(^{9}\) http://en.wikibooks.org//en.wikipedia.org/wiki/x86_architecture
\(^{10}\) http://en.wikibooks.org//en.wikipedia.org/wiki/ARM_architecture
\(^{13}\) http://gcc.gnu.org
\(^{14}\) http://gcc.gnu.org/viewcvs/branches/
optimizations and transformations. The RTL level should only be used for target specific optimizations and optimizations which are low level such as scheduling. Any GCC before 4.0 is in some minds an old dated compiler which was showing its age.

2.3 Purpose of this book

The purpose of this book is to address the demands of GCC hackers. We start with a description of GCC 3.4.1 architecture focusing on the source code parser. We chose this version of GCC because we used this version mostly. Then we address the problem of extension development. We present the GCC Extensibility Modules (GEM) project in the next chapter. GEM provides a number of hooks throughout GCC source code. It is implemented as a patch to GCC. A GEM-based compiler extension is developed as a stand-alone program. When the extension is completed, only its source code is distributed compared with distributing the source code of the GCC if GEM is not used. We give examples that demonstrate GEM programming at the end of the book.


---

15 #An_Overview_of_GCC_Architecture
16 http://research.alexeysmirnov.name/gem
17 #GEM_Framework
18 #Creating_a_Compiler_Extension
3 GNU C Compiler Architecture. Compiling an Expression

3.0.1 An Overview of GCC Architecture. Compilation of an expression.

This section is based on a Red Hat magazine article [1]. The GNU Compiler Collection (GCC) comprises a number of compilers for different programming languages. The main GCC executable gcc processes source files written in C, C++, Objective-C, Objective-C++, Java, Fortran, or Ada and produces an assembly file for each source file. It is a driver program that invokes the appropriate compilation programs depending on the language of the source file. For a C source file they are the preprocessor and compiler cc1, the assembler as, and the linker collect2. The first and the third programs come with a GCC distribution; the assembler is a part of the GNU binutils package. This book describes the internals of the preprocessor and compiler cc1. Each compiler includes three components: a front end, a middle end, and a back end. GCC compiles one file at a time. A source file goes through all three components one after another. Its representation is modified when it goes from a component to the next. Figure 1² illustrates the components and the source file representations associated with each of them. The abstract syntax tree (AST), register transfer language (RTL), and object are the main representations.

![Figure 2](http://en.wikibooks.org/wiki/File:GCC_ast_rtl_obj.jpg)

**Figure 2** GCC front end, middle end, and back end with source file representations.

Main Representations

The purpose of the front end is to read the source file, parse it, and convert it into the standard abstract syntax tree (AST) representation. The AST is a dual-type representation: it is a tree where a node can have children and a list of statements where nodes are chained one after another. There is one front end for each programming language. The AST is

² [Link to image](http://en.wikibooks.org/wiki/File:GCC_ast_rtl_obj.jpg)
then used to generate a register-transfer language (RTL) tree. RTL is a hardware-based representation that corresponds to an abstract target architecture with an infinite number of registers. An RTL optimization pass optimizes the tree in the RTL form. Finally, a GCC back end generates the assembly code for the target architecture using the RTL representation. Examples of back ends are the x86 back end and the MIPS back end. In the next sections we describe the internals of the C front end and the x86 back end. The compiler starts with its initialization and command line options processing. After that the C front end preprocesses the source file, parses it and performs a number of optimizations. The back end then generates the assembly code for the target platform and saves it to a file.

### 3.0.2 Auxiliary Data Structures

GCC has a number of additional data structures that facilitate code development, for example vector and heap. The macros defined in vec.h implement a set of templated vector types and associated interfaces. These templates are implemented with macros, as we’re not in C++ land. The interface functions are typesafe and use static inline functions, sometimes backed by out-of-line generic functions. The vectors are designed to interoperate with the GTY\(^3\) machinery. Because of the different behavior of structure objects, scalar objects and of pointers, there are three flavors, one for each of these variants. Both the pointer and structure object variants pass pointers to objects around -- in the former case the pointers are stored into the vector and in the latter case the pointers are dereferenced and the objects copied into the vector. The scalar object variant is suitable for int-like objects, and the vector elements are returned by value. There are both 'index' and 'iterate' accessors. The iterator returns a boolean iteration condition and updates the iteration variable passed by reference. Because the iterator will be inlined, the address-of can be optimized away. The vectors are implemented using the trailing array idiom, thus they are not resizeable without changing the address of the vector object itself. This means you cannot have variables or fields of vector type -- always use a pointer to a vector. The one exception is the final field of a structure, which could be a vector type. You will have to use the embedded_size & embedded_init calls to create such objects, and they will probably not be resizeable (so don’t use the 'safe' allocation variants). The trailing array idiom is used (rather than a pointer to an array of data), because, if we allow NULL to also represent an empty vector, empty vectors occupy minimal space in the structure containing them. Each operation that increases the number of active elements is available in 'quick' and 'safe' variants. The former presumes that there is sufficient allocated space for the operation to succeed (it dies if there is not). The latter will reallocate the vector, if needed. Reallocation causes an exponential increase in vector size. If you know you will be adding N elements, it would be more efficient to use the reserve operation before adding the elements with the 'quick' operation. This will ensure there are at least as many elements as you ask for, it will exponentially increase if there are too few spare slots. If you want to reserve a specific number of slots, but do not want the exponential increase (for instance, you know this is the last allocation), use a negative number for reservation. You can also create a vector of a specific size from the get go. You should prefer the push and pop operations, as they append and remove from the end of the vector. If you need to remove several items in one go, use the truncate

\(^3\) [http://gcc.gnu.org/onlinedocs/gcc-4.3.4/gccint/GTY-Options.html](http://gcc.gnu.org/onlinedocs/gcc-4.3.4/gccint/GTY-Options.html)
operation. The insert and remove operations allow you to change elements in the middle of the vector. There are two remove operations, one which preserves the element ordering 'ordered_remove', and one which does not 'unordered_remove'. The latter function copies the end element into the removed slot, rather than invoke a memmove operation. The 'lower_bound' function will determine where to place an item in the array using insert that will maintain sorted order. When a vector type is defined, first a non-memory managed version is created. You can then define either or both garbage collected and heap allocated versions. The allocation mechanism is specified when the type is defined, and is therefore part of the type. If you need both gc'd and heap allocated versions, you still must have exactly one definition of the common non-memory managed base vector. If you need to directly manipulate a vector, then the 'address' accessor will return the address of the start of the vector. Also the 'space' predicate will tell you whether there is spare capacity in the vector. You will not normally need to use these two functions. Vector types are defined using a DEF_VEC_{O,P,I}(TYPEDEF) macro, to get the non-memory allocation version, and then a DEF_VEC_ALLOC_{O,P,I}(TYPEDEF,ALLOC) macro to get memory managed vectors. Variables of vector type are declared using a VEC(TYPEDEF,ALLOC) macro. The ALLOC argument specifies the allocation strategy, and can be either 'gc' or 'heap' for garbage collected and heap allocated respectively. It can be 'none' to get a vector that must be explicitly allocated (for instance as a trailing array of another structure). The characters O, P and I indicate whether TYPEDEF is a pointer (P), object (O) or integral (I) type. Be careful to pick the correct one, as you'll get an awkward and inefficient API if you use the wrong one. There is a check, which results in a compile-time warning, for the P and I versions, but there is no check for the O versions, as that is not possible in plain C. Due to the way GTY works, you must annotate any structures you wish to insert or reference from a vector with a GTY(() tag. You need to do this even if you never declare the GC allocated variants. An example of their use would be,

```
DEF_VEC_P(tree); // non-managed tree vector.
DEF_VEC_ALLOC_P(tree,gc); // gc'd vector of tree pointers. This must
                           // appear at file scope.

struct my_struct {
  VEC(tree,gc) *v; // A (pointer to) a vector of tree pointers.
};

struct my_struct *s;

if (VEC_length(tree,s->v)) { we have some contents }
VEC_safe_push(tree,gc,s->v,decl); // append some decl onto the end
for (ix = 0; VEC_iterate(tree,s->v,ix,elt); ix++)
  { do something with elt }
```


Additional Representations

![Diagram of GCC compilation process](http://en.wikibooks.org/wiki/File:Gcc_repr.jpg)

**Figure 3** Additional representations of GCC 4.1.

Figure 2\(^4\) shows additional representations of GCC 4.1. Because of the differences in languages, the format of the generated ASTs is slightly different for each language. The next step after AST generation is the unification step in which the AST tree is converted into a unified form called generic. After this, the middle end part of the compiler takes control. First, the tree is converted into another representation called GIMPLE. In this form, each expression contains no more than three operands, all control flow constructs are represented as combinations of conditional statements and goto operators, arguments of a function call can only be variables, etc. Figure 2\(^5\) illustrates the differences between a tree in generic form and a tree in GIMPLE form. GIMPLE is a convenient representation for optimizing the source code.

After GIMPLE, the source code is converted into the static single assignment (SSA) representation. The central idea of this form is the fact that each variable is assigned to only once, but can be used at the right hand side of an expression many times. Every time the same variable of a tree in the GIMPLE form is reassigned, the compiler creates a new version of that variable and stores the new value into it. When the same variable is assigned to in both branches of a conditional expression, one needs to merge the two possible values of the variable into a single variable. This operation is denoted as PHI function in the SSA form.

The SSA form is also used for optimizations. GCC performs more than 20 different optimizations on SSA trees. After the SSA optimization pass, the tree is converted back to the GIMPLE form.

**Take home:** GCC is a compiler collection that consists of a front end for each programming language, a middle end, and a back end for each architecture. The main representations that each source file goes through are AST in the front end, RTL in the middle end, and the assembly representation in the back end. GCC compiles one file at a time.

3.0.3 GCC Initialization

The C frontend includes the C/C++ preprocessor and the C compiler. Program `cc1` includes both the preprocessor and C compiler. It compiles a C source file and generates an assembly (.S) file. The compiler frontend and backend interact with each other using callback functions called language hooks. All hooks are included into a global variable struct `lang_hooks` that is defined in file `langhooks.h`. There are the following types of hooks: hooks for tree inlining, hooks for call graph, hooks for functions, hooks for tree dump, hooks for types, hooks for declarations, and language-specific hooks. The default values for the hooks are defined in file `langhooks-def.h`. GCC initialization consists of command line option parsing, initializing the back end, creating the global scope, and initializing the built-in data types and functions. Each declaration is associated with a scope. For example, a local variable is associated with its function's scope. A global declaration is associated with the global scope. File `toplev.c` contains the main `cc1` function `toplev_main()` and global variables that define the state of the compiler. Variable `current_function_decl` is the declaration of the function being compiled or `NULL` if between functions. Function `toplev_main()` is the function that processes the command-line options, initializes the compiler, compiles a file, and frees up the allocated resources. Function `decode_options()` processes the command-line options and sets the corresponding variables within the compiler. Following the command line option parsing function `do_compile()` is called. It performs the back-end initialization by calling function `backend_init()`. Back-end initialization includes a number of steps. Function `init_emit_once()` generates RTL expressions for a number of registers: `pc_rtx` for the program counter, `cc0` for the condition, `stack_pointer_rtx`, `frame_pointer_rtx`, etc. They are saved in array `global_rtl`. After that, function `lang_dependent_init()` performs language-dependent initialization which includes the initialization of a front-end and a back-end. The C initialization function

![Figure 4  GCC initialization.](image-url)
GCC compiler architecture. Compiling an Expression

`c_objc_common_init()` creates built-in data types, initializes the global scope and performs other initialization tasks. Function `c_common_nodes_and_builtins()` creates pre-defined types described in file `builtin-types.def`. The standard C types are created at the initialization time. The following table presents several types:

### GCC built-in types

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_type_node</td>
<td>char</td>
</tr>
<tr>
<td>integer_type_node</td>
<td>int</td>
</tr>
<tr>
<td>unsigned_type_node</td>
<td>unsigned int</td>
</tr>
<tr>
<td>void_type_node</td>
<td>void</td>
</tr>
<tr>
<td>ptr_type_node</td>
<td>void*</td>
</tr>
</tbody>
</table>

GCC built-in functions are the functions that are evaluated at compile time. For example, if the size argument of a `strcpy()` function is a constant then GCC replaces the function call with the required number of assignments. The compiler replaces standard library calls with built-in functions and then evaluates them once the function's AST is constructed. In case of `strcpy()`, the compiler checks the size argument and uses the optimized built-in version of `strcpy()` if the argument is constant. `builtin_constant_p()` allows to find out if the value of its argument is known at compile time. GCC builtins are used beyond GCC. For example, the string processing library of Linux kernel uses `builtin_constant_p()` to invoke the optimized version of a string processing function if the string size is known at compile time. GCC evaluates each builtin function using a corresponding `expand_builtin()` function. For example, `builtin_strcmp()` is evaluated using `expand_builtin_strcmp()`. The following table gives examples of GCC builtins:

### GCC builtin functions

<table>
<thead>
<tr>
<th>Builtin Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtin_constant_p</td>
<td>returns true if the argument is a constant</td>
</tr>
<tr>
<td>builtin_memcpy</td>
<td>equivalent to <code>memcpy()</code></td>
</tr>
<tr>
<td>builtin_strlen</td>
<td>equivalent to <code>strlen()</code></td>
</tr>
</tbody>
</table>

**Take home:** GCC initialization consists of command line option parsing, initializing the back end, creating the global scope, and initializing the built-in data types and functions.

### 3.0.4 Parser and Preprocessor

Following the initialization, function `do_compile()` calls function `compile_file()`. This function invokes `parse_file()` front-end language hook which is set to function `c_common_parse_file()` for C language. The latter function invokes function `finish_options()` which initializes the preprocessor and handles `-D`, `-U`, and `-A` command
line options (which are equivalent to \#define, \#undef, and \#assert respectively). The C preprocessor handles the preprocessor directives such as \#define, \#include in the source code.

**Parser**

Parser is implemented manually in file `c_parser.c`. Compared to earlier releases of GCC, the new parser generates a lower level AST. There were special tree codes for loops, for example `FOR_STMT` denoted a for() loop. In this release the loops are represented as conditional statements and gotos, that is trees of codes `COND_EXPR`, `LABEL_EXPR`, and `GOTO_EXPR`. This probably means that it is impossible to raise AST representation to original source code.

**Preprocessor**

The preprocessor is implemented as a library. The C language lexer function `c_lex()` calls the `libc++` function `cpp_get_token()` which handles the preprocessor keywords. The state of the preprocessor is defined by variable `cpp_reader *parse_in`. Type `struct cpp_reader` contains most importantly the list of text buffers being processed. Each buffer corresponds to a source file (.c or .h). Function `cpp_get_token()` calls appropriate functions for legitimate preprocessor keywords. For example, when `#include` is encountered, function `do_include_common()` is invoked. It allocates a new buffer and places it on top of the buffer stack making it the current buffer. When a new file is compiled the buffer is popped off the stack and the compilation of the old file continues. Function `do_define()` is called whenever a new macro is defined using `#define` keyword.

**Take home:** The preprocessor takes care of the preprocessor directives, for example `#include` and `#ifdef`.

**3.0.5 From Source Code to AST**

After running the preprocessor GCC constructs an abstract syntax tree (AST) for each function of the source file. An AST is a number of connected nodes of type `struct tree`. Each node has a `tree code` that defines the type of the tree. Macro `TREE_CODE()` is used to refer to the code. Tree codes are defined in files `tree.def` and `c-common.def`. Trees with different tree codes are grouped into `tree classes`. The following tree classes are defined among others in GCC:

**GCC tree classes**

<table>
<thead>
<tr>
<th>Tree Class</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'d'</td>
<td>declarations</td>
</tr>
<tr>
<td>'&lt;'</td>
<td>comparison</td>
</tr>
<tr>
<td>'2'</td>
<td>binary arithmetic</td>
</tr>
</tbody>
</table>
There are two types of trees: statements and expressions. Statements correspond to the C constructs, for example an expression followed by a ';', a conditional statement, a loop statement, etc. Expressions are what the statements are built up from. Examples of expressions are an assignment expression, an arithmetic expression, a function call, etc. Examples of tree codes are given in the following table:

**GCC tree codes**

<table>
<thead>
<tr>
<th>Tree Code</th>
<th>Tree Class</th>
<th>Explanation</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODIFY_EXPR</td>
<td>'e'</td>
<td>assignment expression</td>
<td>TREE_OPERAND(t,0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TREE_OPERAND(t,1)</td>
</tr>
<tr>
<td>CALL_EXPR</td>
<td>'e'</td>
<td>function call</td>
<td>TREE_OPERAND(t,0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TREE_OPERAND(t,1)</td>
</tr>
<tr>
<td>FUNCTION_DECL</td>
<td>'d'</td>
<td>variable declaration</td>
<td>DECL_SOURCE_FILE(t)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DECL_NAME(t)</td>
</tr>
<tr>
<td>ARRAY_TYPE</td>
<td>'t'</td>
<td>array type</td>
<td>TREE_TYPE(t)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TYPE_DOMAIN(t)</td>
</tr>
<tr>
<td>DECL_STMT</td>
<td>'e'</td>
<td>variable declaration</td>
<td>TREE_OPERAND(t,0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DECL_INITIAL(TREE_OPERAND(t,1))</td>
</tr>
</tbody>
</table>

In addition to the tree code that defines the type of the tree, a number of operands different for each tree type are available. For example, an assignment expression has two operands which correspond to the left and right sides of the expression. Macro `TREE_OPERAND` is used to refer to the operands. Macro `IDENTIFIER_POINTER` is used to find the name of an identifier that an `IDENTIFIER_NODE` tree represents. The following table presents several tree nodes, their purpose, and their operands. Each tree has a type that corresponds to the type of C expression that it represents For example, the type of the `MODIFY_EXPR` node is the type of the left-hand side operand. `NOP_EXPR` and `CONVERT_EXPR` trees are used for type casting. Tree `NULL_TREE` is equivalent to `NULL`. Function `debug_tree()` prints out the tree to `stderr`. When a new identifier is lexed, it is added to the pool of strings that GCC maintains. The tree code of an identifier is `IDENTIFIER_NODE`. When the
same identifier is lexed again, the same tree node is returned. Function \textit{get\_identifier()} returns the tree node for an identifier.

**Figure 5** Parsing variable declaration.

A new variable declaration is processed in a number of function calls. First, function \textit{start\_decl()} is called with the name of the declaration, its type as returned by the lexer, and its attributes. The function calls \textit{grok\_declarator()} that checks the type and argument nodes and returns a tree with the code appropriate for the declaration: \textit{VAR\_DECL} for variables, \textit{TYPE\_DECL} for types, etc. The declaration is then added to the scope. A scope contains all declarations created within a function, but does not contain global declarations. There is also the global scope that contains global declarations. When a function is parsed, its declarations are attached to its body as a \textit{BLOCK} node. When a declaration is created, the identi-
fier node is associated with the declaration node using `IDENTIFIER_SYMBOL_VALUE`. Function `lookup_name()` returns the declaration for a given identifier. When a declaration leaves the scope the tree attributes `C_DECL_INVISIBLE` is asserted. A symbol table is not maintained in GCC. Instead, the compiler uses the pool of identifiers and `C_DECL_INVISIBLE` attribute. Language hook `lang_hooks.decls.getdecls()` returns the variables in the scope chained together. Functions `start_init()` and `finish_init()` are called for an initialized declaration. Function `finish_decl()` finalizes the declaration. For an array declaration it computes the size of an initialized array. Then function `layout_decl()` is called. It computes the size and alignment of the declaration. Parsing a function depends on the presence of its body. A function declaration is parsed using the same functions as those used for variable declarations. For a function definition, function `start_function()` is called. Then the compiler parses the body of the function. When the function ends function `finish_function()` is called. Functions `start_decl()` and `start_function()` take declaration's attributes parameter as one of their arguments. The attributes are described in the GCC Manual. Attributes are extensions of GNU implementation of C. The following table presents a few of them and explains their purpose:

**Function attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor</td>
<td>call function automatically before main()</td>
</tr>
<tr>
<td>destructor</td>
<td>call function automatically after main()</td>
</tr>
<tr>
<td>alias</td>
<td>alias to another function</td>
</tr>
</tbody>
</table>

For each type of C statement there is a function that constructs a tree node of the corresponding type. For example, function `build_function_call()` builds a `CALL_EXPR` node for a function call. It takes the identifier of the function name and the arguments as its parameters. The function finds the function declaration using `lookup_name()` and type casts the arguments using `default_conversion()`. After a function has been parsed, use macro `DECL_SAVED_TREE` to access its body. It is represented with a `BIND_EXPR` tree which binds local variables to statements. `BIND_EXPR_VARS` gives a chain of declared variables. `BIND_EXPR_BODY` returns a tree of `STATEMENT_LIST` type. The following API allows one to traverse a statement list and to manipulate it:

**Tree construction API**

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsi_start(stmt_list)</td>
<td>get an iterator pointing at list head</td>
</tr>
<tr>
<td>tsi_last(stmt_list)</td>
<td>get an iterator pointing at list tail</td>
</tr>
<tr>
<td>tsi_end_p(iter)</td>
<td>is end of list?</td>
</tr>
<tr>
<td>tsi_stmt(iter)</td>
<td>get current element</td>
</tr>
<tr>
<td>tsi_split_statement_list_before(&amp;iter)</td>
<td>split elements at iter</td>
</tr>
<tr>
<td>tsi_link_after(&amp;iter, stmt, mode)</td>
<td>link chain after iter</td>
</tr>
<tr>
<td>tsi_next(&amp;iter)</td>
<td>next element of the list</td>
</tr>
<tr>
<td>append_to_statement_list(tree, &amp;stmt_list)</td>
<td>append tree to stmt_list</td>
</tr>
</tbody>
</table>

It is possible to instrument function prolog/epilog at this level as demonstrated in `simplify_function_tree()`. To add a statement to function epilog, use `TRY_FINALLY_EXPR`
tree. Its first operand are the old statements, the second argument is the epilog statements. This type of tree instructs the following passes to execute the statements when a common exit basic block of a function is created. To instrument function prolog, prepend the tree with the desired statements. Therefore, `BIND_EXPR_BODY` will have the prolog and the `TRY_FINALLY_EXPR` tree. The ASTs are then converted into the SSA and eventually to the RTL representations. Whether the conversion occurs after parsing each function or when the whole file is parsed is controlled by the compiler option `-funit-at-a-time`. It is false by default.

**Take home:** GCC parser builds the AST representation of the source file. ASTs are built up of tree nodes. Each node has a code. Tree nodes correspond to the statements and expressions of C. Function `debug_tree()` prints out the tree.

### 3.0.6 From AST to GIMPLE

An AST is gimplified when `gimplify_function_tree()` is eventually called from `finish_function()`. GIMPLE representation is based on SIMPLE described in [2]. According to this paper, the goal is to represent the tree as basic statements:

**GIMPLE trees**

<table>
<thead>
<tr>
<th>Code</th>
<th>GIMPLE tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x=a binop b</code></td>
<td><code>x=a</code></td>
</tr>
<tr>
<td><code>*p=a binop b</code></td>
<td><code>*p=a</code></td>
</tr>
<tr>
<td><code>x=unop a</code></td>
<td><code>x=*q</code></td>
</tr>
<tr>
<td><code>*p=unop a</code></td>
<td><code>*p=*q</code></td>
</tr>
<tr>
<td><code>x=cast b</code></td>
<td><code>x=cast b</code></td>
</tr>
<tr>
<td><code>*p=cast b</code></td>
<td><code>*p=cast b</code></td>
</tr>
<tr>
<td><code>f(args)</code></td>
<td><code>f(args)</code></td>
</tr>
<tr>
<td><code>-</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td><code>x=f(args)</code></td>
<td><code>x=f(args)</code></td>
</tr>
</tbody>
</table>

Temp variables are created as necessary in functions `create_tmp_var()` and `declare_tmp_vars()`. At this stage, GCC optimizes of complex conditional expressions, that is

```c
if (a || b) stmt;
```

is translated into

```c
if (a) goto L1;
if (b) goto L1; else goto L2;
L1:
stmt;
L2:
```

Also, each branch of a conditional expression is wrapped into `STATEMENT_LIST` tree.

### 3.0.7 From GIMPLE to RTL

*Register Transfer Language* represents an abstract machine with an infinite number of registers. Type `rtx` describes an instruction. Each RTL expression has a code and machine
mode. Similarly to ASTs, codes are grouped in a number of classes. They are defined in mode-classes.def.

**Classes of RTL expressions**

<table>
<thead>
<tr>
<th>Classes</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTX_CONST_OBJ</td>
<td>represent a constant object (e.g., CONST_INT)</td>
</tr>
<tr>
<td>RTX_OBJ</td>
<td>represent an object (e.g., REG, MEM)</td>
</tr>
<tr>
<td>RTX_COMPARE</td>
<td>comparison (e.g., LT, GT)</td>
</tr>
<tr>
<td>RTX_COMM_COMPARE</td>
<td>commutative comparison (e.g., EQ, NE, ORDERED)</td>
</tr>
<tr>
<td>RTX_UNARY</td>
<td>unary arithmetic expression (e.g., NEG, NOT)</td>
</tr>
<tr>
<td>RTX_COMM_ARITH</td>
<td>commutative binary operation (e.g., PLUS, MULT)</td>
</tr>
<tr>
<td>RTX_TERNARY</td>
<td>non-bitfield three input operation (IF_THEN_ELSE)</td>
</tr>
<tr>
<td>RTX_BIN_ARITH</td>
<td>non-commutative binary operation (e.g., MINUS, DIV)</td>
</tr>
<tr>
<td>RTX_BITFIELD_OPS</td>
<td>bit-field operation (ZERO_EXTRACT, SIGN_EXTRACT)</td>
</tr>
<tr>
<td>RTXInsn</td>
<td>machine insn (INSN, JUMP_INSN, CALL_INSN)</td>
</tr>
<tr>
<td>RTX_MATCH</td>
<td>something that matches in insns (e.g., MATCH_DUP)</td>
</tr>
<tr>
<td>RTX_AUTOINC</td>
<td>autoincrement addressing modes (e.g., POST_DEC)</td>
</tr>
<tr>
<td>RTX_EXTRA</td>
<td>everything else</td>
</tr>
</tbody>
</table>

Machine modes listed in file machmode.def specify a size and format of data at the machine level. At the syntax tree level, each ..._TYPE and each ..._DECL node has a machine mode which describes data of that type or the data of the variable declared. A list of instructions is built when function is compiled. Function emit_insn() adds an instruction to the list. A variable declaration AST has its RTL already generated. Use DECL_RTL to access it. Function emit_cmp_and_jump_insn() outputs a conditional statement. emit_label() prints out a label. These functions chain instructions one after another. Macros PREV_INSN and NEXT_INSN are used to traverse the list. It is possible to access the first and last instructions using first_insn and last_insn. get_insns() gives the first insn of the current sequence or current function. Use debug_rtx() to print out an RTL instruction on the screen and function print_rtl() to print a list of RTL expressions. A number of functions create the nodes. For example, gen_label_rtx() build a label. The most generic functions are located in the target-specific directory. For example, x86 architecture rtl generation files genrtl.c and genrtl.h are in ./host-i686-pc-linux-gnu.

### 3.0.8 From RTL to Object

Each target architecture has its description represented as struct gcc_target targetm. Default initializers are available in file targhooks.c The backend generates the assembly code for the specified target platform. Function output_asm_insn() is called for each instruction written to the assembly file. Function final_start_function() generates the prolog of the function before it is saved to the assembly file.
3.0.9 Lowering Passes

The processing of a function includes its lowering when a number of optimization passes are applied to it in function tree_lowering_passes(). As a result of lowering a function its control-flow graph is generated. A subsequent call to function cgraph_create_edges() uses basic block information to augment edges of the callgraph with invocations that the current function performs. References to functions yet undefined are saved in function record_references().

All lowering passes

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove_useless_stmts</td>
<td>N/A</td>
</tr>
<tr>
<td>mudflap_1</td>
<td>narrow-pointer bounds-checking by tree rewriting</td>
</tr>
<tr>
<td>lower_cf</td>
<td>Lowers GIMPLE into unstructured form</td>
</tr>
<tr>
<td>pass_lower_eh</td>
<td>Exception handling semantics and decomposition for trees</td>
</tr>
<tr>
<td>pass_build_cfg</td>
<td>Create basic blocks</td>
</tr>
<tr>
<td>pass_lower_complex_O0</td>
<td>complex operation lowering without optimization</td>
</tr>
<tr>
<td>pass_lower_vector</td>
<td>Lower vector operations to scalar operations</td>
</tr>
<tr>
<td>pass_warn_function_return</td>
<td>Emit return warnings</td>
</tr>
<tr>
<td>pass_early_tree_profile</td>
<td>set up hooks for tree-based profiling</td>
</tr>
</tbody>
</table>

3.0.10 Switch Statement

Let us consider how a switch statement is translated from the source code to GIMPLE to RTL. Function c_parser_switch_statement() is called when the statement is encountered in the source code. A typical switch statement has a number of case statements that might have breaks. Therefore, the parser has c_break_label tree that marks the place where switch ends. The function parses the body of the statement and generates LABEL_EXPR tree for the break label if at least one break was found. Function c_finish_case() attaches the body to the SWITCH_EXPR tree as one of its operands. In addition, this tree has two other operands: switch condition and switch labels. The operands are accessed using macro SWITCH_COND(), SWITCH_BODY(), and SWITCH_LABELS(). The labels are not filled in at the parse time. A switch statement is simplified in function simplify_switch_expr(). The idea is to separate the body from the decision part and generate switch labels to make it possible to redirect execution to the appropriate case after verifying the condition. We will consider the case when default label exists. This function has two pointers to the statement list: pre_p which is the list of side effects and expr_p which is the statement itself. The body of the switch is simplified in simplify_to_stmt_list(). Case labels are saved in field case_labels of variable struct simplify_ctx() and simplify_ctxp(). Then the function creates a TREE_VEC of labels and initializes them with the corresponding case label. The TREE_VEC is assigned to SWITCH_LABELS operand of switch statement which is then appended to the pre_p list. The original statement is then overwritten with the SWITCH_BODY using expr_p pointer. Finally, the SWITCH_BODY operand of the switch statement in the side effects list is wiped out so that it contains labels only. From this point on it is clear that compiler tries to represent the original statement using a jump table
which maps each possible index value to the address of the corresponding case. Function expand_case() implements this idea. It generates a table_label at which jump instructions for each possible index value are generated. Then function try_tablejump() is called which expands index tree into index rtl and invokes do_tablejump(). This function generates absolute index rtl which combines the base address table_label and index offset. It emits jump instruction to the proper entry of the jump table afterwards. The execution continues in function expand_case(). The jump table is generated using SWITCH_LABELS:

```
labelvec[i] = gen_rtx_LABEL_REF (Pmode, label_rtx (n->code_label));
```

A number of jump instructions are emitted finally:

```
if (CASE_VECTOR_PC_RELATIVE || flag_pic)
  emit_jump_insn (gen_rtx_ADDR_DIFF_VEC (CASE_VECTOR_MODE,
                                             gen_rtx_LABEL_REF (Pmode, table_label),
                                             gen_rtvec_v (ncases, labelvec),
                                             const0_rtx, const0_rtx));
```

**Take home**: The backend generates the assembly code for the specified target platform.

1.  

---

7 #ref_rh_magazine
8 http://www.redhat.com/magazine/002dec04/features/gcc/
9 #ref_simple_paper
4 Compilation of a function

4.1 Function Prolog and Epilog

A prolog is used to initialize the function, for example set up its stack frame. A function epilog is used to finalize execution. At each level of compilation, there are common actions performed for each function. Therefore, a number of prolog/epilog generation functions exist in GCC. At the GIMPLE level prolog is generated in function gimplify_function_tree(). It adds profiling hooks if instructed so using -finstrument-functions. At the RTL level, function expand_function_start is used. It starts the RTL for a new function, and set variables used for emitting RTL. It also makes the label for return statements to jump to and decides whether to return the value in memory or in a register. After this, it calls assign_parms() which maps the formal-ins to the actual-ins. Machine-level prolog and epilog are added in pass flow2, function thread_prologue_and_epilogue_insns() which relies upon machine...
Compilation of a function

description file, for example i386.md. The description has a number of entries. The pro-
log and epilog entries is used in this case. They are set to ix86_expand_prologue() and
ix86_expand_epilogue() respectively that generate RTL prologue and epilog for i386 archi-
tecture. The machine-specific code takes care of registered used in the function so that they
are preserved between function calls. The push instructions are added as the very first in-
structions of a function. The machine-specific code is generated as an RTL expression rather
than assembly because this is the representation used at this pass of the compiler. The RTL
representation is the most general across various targets. Therefore, a push instruction of
i386 corresponds to a number of RTL expressions:

```c
static rtx
gen_push (rtx arg)
{
  return gen_rtx_SET (VOIDmode,
                    gen_rtx_MEM (Pmode,
                    gen_rtx_PRE_DEC (Pmode,
                    stack_pointer_rtx)));
}
```

At the assembly output level, prolog is generated in functions assemble_start_function() and
final_start_function(). Their output is mostly related to debugging. One should note
that the prolog functions are executed from the higher-level representation to the lower-
level ones. At the run-time, the execution order of the added code is opposite. That is,
the machine-level prolog is executed first. In case of i386, it saves the registers used in the
function and sets up a new stack frame. Then the parameters are received and the profiling
hooks are called.

### 4.2 Local Control Flow Analysis

A control-flow graph is built up of basic blocks which is a sequence of instructions with only
one entry and one exit. Type struct basic_block describes it. There is a pointer to the state-
ment list that defines the basic block. Each basic block also has a pointer to the previous and
next BB. Therefore, it is possible to link them in a list. Fields preds and succ give access to
control/data flow edges into and out of the block. Function create_basic_block() takes the
first and the last statements and inserts the new BB after a given one. make_edge() links the
two BBs. Use macro FOR_EACH_BB to traverse a CFG. Each function flow graph has an
entry and an exit block accessed using ENTRY_BLOCK_PTR and EXIT_BLOCK_PTR
respectively. Function build_tree_cfg() takes a GIMPLE tree and generates the CFG.
First, it initializes the CFG with two BBs: ENTRY_BLOCK and EXIT_BLOCK. It calls
make_blocks(), then make_edges().
5 Function calls

5.1 Global Control-Flow Analysis

The functions of a file are used to generate a callgraph in file cgraphunit.c. The two relevant functions are cgraph_finalize_compilation_unit() which is called from function pop_file_scope() after the file has been parsed and cgraph_finalize_function() which is called from finish_function(). The effect of each function depends on the compilation mode. The unit-at-a-time mode instructs the compiler to build the callgraph only after each function has been parsed. When this option is not present a function is converted as soon as it is parsed. cgraph_finalize_function() calls cgraph_analyze_function() that converts it to RTL. Otherwise, the function is queued in cgraph_nodes_queue. At the end cgraph_finalize_compilation_unit() takes care of the queue. cgraph_nodes is the global variable representing the callgraph. Function dump_cgraph allows one to print out the callgraph.

5.2 Parameter Passing

In this chapter we will find out how functions call each other. Typically, a function passes a number of parameters when it makes a call. A stack frame is created when the function begins. It is possible, however, that the stack frame of the previous function gets reused. This type of function call is called a sibling call. When the function body is not large enough the run-time overhead of setting up a stack frame is too high. In this case the callee function gets inlined into the parent function. Function expand_call() takes a CALL_EXPR tree and generates RTL expression. It has to decide argument passing mode. struct arg_data contains necessary information for each argument.

```
struct arg_data

Field Name          Explanation
tree tree_value     Tree node for this argument
enum machine_mode   Mode for value
rtx value           Current RTL value for argument, or 0 if it isn't precomputed
rtx initial_value   Initially-compute RTL value for argument; only for const functions.
rtx reg             Register to pass this argument in, 0 if passed on stack
rtx tail_call_reg   Register to pass this argument in when generating tail call sequence
```

23
### struct arg_data

- **rtx parallel_value**
  - If REG is a PARALLEL, this is a copy of VALUE pulled into the correct form for emit_group_move.

- **int unsignedp**
  - If REG was promoted from the actual mode of the argument expression, indicates whether the promotion is sign- or zero-extended

- **int partial**
  - Number of bytes to put in registers. 0 means put the whole are in registers or not passed in registers.

- **int pass_on_stack**
  - Nonzero if argument must be passed on stack.
  - Note that some arguments may be passed on the stack even though pass_on_stack is zero, just because FUNCTION_ARG says so

- **struct locate_and_pad_arg_data**
  - Some fields packaged up for locate_and_pad_parm

- **rtx stack**
  - Location on the stack at which parameter should be stored

- **rtx stack_slot**
  - Location on the stack of the start of this argument slot

- **rtx save_area**
  - Place that this stack area has been saved, if needed

- **rtx *aligned_regs**
  - If an argument's alignment does not permit direct copying into registers, copy in smaller-sized pieces into pseudos. These are stored in a block pointed to by this field.

- **int n_aligned_regs**
  - Says how many word-sized pseudos we made

Generating a function call is impossible without certain machine-specific information, for example the number of hardware registers of different types. A number of macros defined in each architecture .h file take care of connecting the middle-end with the back-end:

### Argument macros

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT_CUMULATIVE_ARGS</td>
<td>Initialize CUMULATIVE_ARGS data structure for a call to a function whose data type is FNTYPE.</td>
</tr>
<tr>
<td>FUNCTION_ARG</td>
<td>Define where to put the arguments to a function. Value is zero to push the argument on the stack, or a hard register in which to store the argument.</td>
</tr>
<tr>
<td>FUNCTION_ARG_ADVANCE</td>
<td>Update the data in CUM to advance over an argument.</td>
</tr>
</tbody>
</table>

Function init_cumulative_args() in file i386.c takes care of this in case of x86 architecture. It takes into account function attributes regparm andfastcall that a user might specify in which case the number of available registers is set accordingly. However, if the function takes a variable number of arguments then all parameters are passed in the stack. Parame-
ters' location is decided in function `initialize_argument_information()`. A machine-specific function `arg()` returns the rtl for an argument if it goes into a register:

```
ret = gen_rtx_REG (mode, regno);
```

It is possible that a parameter is passed both on the stack and in a register, for example if the parameter's type is addressable. Depending on certain conditions, sibling call instruction chain is generated in addition to the normal chain. Let us consider the case when only normal chain is generated. Variable `rtx argblock` is the address of space preallocated for stack parameters (on machines that lack push insns), or 0 if space not preallocated. A number of machine-specific variables shape up the stack. `ACCUMULATE_OUTGOING_ARGS` instructs the compiler to preallocate the sufficient number of bytes for all arguments of any function call in the function prolog. After that, function arguments are saved in that area without modifying stack frame size. `ACCUMULATE_OUTGOING_ARGS` depends on variable `target_flags`. It depends on machine configuration and command-line options. In case of `ACCUMULATE_OUTGOING_ARGS`, i386-specific variable

```
const int x86_accumulate_outgoing_args = m_ATHLON_K8 | m_PENT4 | m_NOCONA | m_PPRO;
```

and a command-line option `-maccumulate-outgoing-args` enable this feature. This means that it is enabled on a Pentium4 and that push/pop instructions are not used to pass function's parameters. If we preallocated stack space, compute the address of each argument and store it into the ARGS array. Precompute parameters as needed for a function call. this routine fills in the INITIAL_VALUE and VALUE fields for each precomputed argument. precompute_arguments() Given a FNDECL and EXP, return an rtx suitable for use as a target address in a call instruction

```
funexp = rtx_for_function_call (fndecl, addr);
```

Precompute all register parameters. It isn't safe to compute anything once we have started filling any specific hard regs. precompute_register_parameters (num_actuals, args, &reg_parm_seen); Now store (and compute if necessary) all non-register parms. These come before register parms, since they can require block-moves, which could clobber the registers used for register parms.Parms which have partial registers are not stored here, but we do preallocate space here if they want that.

```
store_one_arg (&args[i], argblock, flags,
adjusted_args_size.var != 0,
reg_parm_stack_space)
```

Do the register loads required for any wholly-register parms or any parms which are passed both on the stack and in a register. Their expressions were already evaluated.

```
load_register_parameters (args, num_actuals, &call_fusage, flags,
pass == 0, &sibcall_failure);
```
Finally, emit_call_1() generates instructions to call function FUNEXP, and optionally pop the results. The CALL_INSN is the first insn generated. When the placement of the arguments is decided variable struct args_size saves the total size of stack arguments. It records the size of a sequence of arguments as the sum of a tree-expression and a constant. The tree part is necessary to handle arguments of variable size, for example arrays arguments which size is not known at compile-time. The C language does not allow variable-sized arguments. One might wonder how the callee function finds out where the arguments arrive. It also uses machine-specific information that the caller used. Rerunning INIT_CUMULATIVE_ARGS, FUNCTION_ARG, and FUNCTION_ARG_ADVANCE in the callee decides whether an argument should arrive in a register or on the stack identically to expand_call.
6 Stackguard

Stackguard is implemented in the following files. These files translate GIMPLE to RTL. They take advantage of the CFG to instrument function prolog and epilog. 1) cfg-expand.c Stackguard is created in function expand_used_vars(). Depending on stackguard flag either all arrays or character arrays only are allocated first. tree_expand_cfg() takes care of prolog when it calls stack_protect_prologue(). The corresponding function stack_protect_epilogue() is called from calls.c to take care of tail calls and from tree_expand_cfg(); 2) function.c stack_protect_prologue(), stack_protect_epilogue() The functions use machine definition targetm. 3) targhooks.c Contains default initializers of target architecture hooks. default_stack_protect_guard() builds a VAR_DECL tree node that represents variable __stack_chk_guard. default_stack_protector_fail() builds a call to function __stack_chk_fail(). These ASTs are converted into RTL trees using expand_expr_stmt(). The prolog and epilog functions add RTL expressions directly as well. For example, to detect a compromise function epilog generates a conditional statement that compares the canary word on the stack with its initial value:

```c
emit_cmp_and_jump_insns (x, y, EQ, NULL_RTX, ptr_mode, 1, label);
```

where x and y are corresponding declaration RTLS:

```c
x = validize_mem (DECL_RTL (cfun->stack_protect_guard));
y = validize_mem (DECL_RTL (guard_decl));
```

Stackguard implements a certain amount of static analysis to make sure that suspicious functions with arguments of fixed length do not have vulnerabilities. The underlying mechanism is that of builtins. Stackguard has file strcpy.h that defines functions of interest, for example strcpy(). The new definition uses builtins __ssp_bos() to find out object size and __builtin_strcmp_chk():

1. define __ssp_bos(ptr) __builtin_object_size (ptr, __SSP_FORTIFY_LEVEL > 1)
2. define strcpy(dest, src)  

```c
((__ssp_bos (dest) != (size_t) -1)  
 ? __builtin_strcmp_chk (dest, src, __ssp_bos (dest))  
 : __strcmp_ichk (dest, src))
```

The builtins are evaluated at compile time. Function warning() called in maybe_emit_chk_warning() prints out a message if the destination buffer is overflown.
In case when Stackguard is unable to find out statically if the buffer is overrun function __strcpy_chk() is called at run-time. Library libssp provides the function. However, the protected program does not require linking with this library explicitly. Instead, GCC instructs the loader to use a shared library.
7 GEM Framework

7.0.1 Hooks

GEM framework is designed to facilitate development of compiler extensions. The idea of GEM is similar to the idea of Linux Security Modules\(^1\) (LSM), a project that defines hooks throughout Linux kernel that allow one to enforce a security policy. GEM defines a number of hooks throughout GCC’s source code. It is implemented as a patch to GCC. With GEM, a compiler extension is developed as a stand-alone program. It is compiled into a dynamically-linked module which is specified as the command line argument when GCC is invoked. GCC loads the module and calls its initialization function. The module then registers its hooks that are call-back functions in GCC. In addition to the compiler hooks, GEM provides macros and functions that simplify extension development. In this chapter we will first introduce the hooks that GEM framework adds to GCC. Then we describe the typical issues in extension programming. The project home page is at \(^2\) GEM adds several hooks throughout GCC source code. New hooks are added to GEM as necessary.

- Hook **gem_handle_option** to function `handle_option()` which processes each command line option. The hook takes the current option as its argument. If the hook returns value **GEM_RETURN** then GCC ignores the option.

- Hook **gem_c_common_nodes_and_builtins** which is called after all standard types are created. The GCC extension can create additional types.

- Hook **gem_macro_name** allows one to save the name of the macro being defined. Another GEM hook **gem_macro_def** is called when the macro definition is parsed. Using the macro name of the new macro definition it is possible to re-define the macro. This hook is added to function `create_iso_definition()`.

- Hooks **gem_start_decl** and **gem_start_function** are called when a function or variable declaration/definition starts.

- Hook **gem_build_function_call** allows one to modify the name and the arguments of a function call.

- Hook **gem_finish_function** is inserted to `finish_function()` which is called from from grammar file. The compiler extension receives the function body of the function before it is translated into RTL.

- Hooks **gem_output_asm_insn** and **gem_final_start_function** are added to function `output_asm_insn()` which is called for each instruction of the assembly code and

---

1 http://lsm.immunix.org/
2 http://research.alexeysmirnov.name/gem
function `final_start_function()` called when the assembly code is written to the file, respectively. The former hook receives the text that is written to the file which allows it to modify the output. The latter hook can modify function's prolog.

**Take home:** GEM hooks are defined mostly at the AST level. A few hooks are defined at the assembly level. The new hooks are added as necessary.

### 7.0.2 Traversing an AST

When the function's AST is constructed one can instrument it. GEM's `gem_finish_function` hook receives the AST of a function. The idea is to traverse the AST and instrument the AST nodes as necessary. Function `walk_tree()` takes the AST, the callback function, the optional data, `NULL` by default, and the `walk_subtrees` parameter, `NULL` by default. The callback function is called for each node of the AST before the operands are traversed. If the callback function modifies the `walk_subtree()` variable then the operands are not processed. The following code demonstrates the idea:

```c
static tree walk_tree_callback(tree *tp, int *walk_subtrees, void *data) {
  tree t=*tp;
  enum tree_code code = TREE_CODE(t);
  switch (code) {
    case CALL_EXPR:
      instrument_call_expr(t);
      break;
    case MODIFY_EXPR:
      instrument_modify_expr(t);
      break;
  }
}
walk_tree(&t_body, walk_tree_callback, NULL, NULL);
```

**Take home:** Function `walk_tree()` traverses an AST applying user-defined callback function to each tree node.

### 7.0.3 Instrumenting an AST

In this section we describe functions that create new tree nodes and how to add the new nodes to an AST.

#### Lookup of a Declaration in the Symbol Table

```c
void gem_find_symtab(tree *t_var, char *name) {
  tree t_ident = get_identifier(name);
  if (t_ident) *t_var = lookup_name(t_ident); else *t_var=NULL_TREE;
}
```
Building Tree Nodes

The `walk_tree` callback function can instrument the AST. Functions `build1()` and `build()` construct new tree nodes. The former function takes one operand, the latter one takes more than one operand. The following code computes the address of the operand, same as `&` C operator:

```c
    t = build1(ADDR_EXPR, TREE_TYPE(t), t);
```

The following example refers to an array element `arr[0]`:

```c
    t = build(ARRAY_REF, integer_type_node, arr, integer_zero_node);
```

The following example builds an integer constant:

```c
    t = build_int_cst(NULL_TREE, 123);
```

Building a string constant is more difficult. The following example demonstrates the idea:

```c
tree gem_build_string_literal(int len, const char *str) {
    tree t, elem, index, type;
    t = build_string(len, str);
    elem = build_type_variant(char_type_node, 1, 0);
    index = build_index_type(build_int_2(len-1, 0));
    type = build_array_type(elem, index);
    T_T(t) = type;
    TREE_READONLY(t)=1;
    TREE_STATIC(t)=1;
    TREE_CONSTANT(t)=1;
    type=build_pointer_type(type);
    t = build1 (ADDR_EXPR, type, t);
    t = build1 (NOP_EXPR, build_pointer_type(char_type_node), t);
    return t;
}
```

To build a function call one needs to find the function's declaration and build the list of arguments. Then the CALL_EXPR is constructed:

```c
gem_find_symtab(&t_func_decl, "func");
t_arg1 = build_tree_list(NULL_TREE, arg1);
t_arg2 = build_tree_list(NULL_TREE, arg2);
...
    TREE_CHAIN(t_arg1)=t_arg2;
...
    TREE_CHAIN(t_args)=NULL_TREE;
    t_call = build_function_call(t_func_decl, t_arg1);
```

If you want to build a list of statements `{ stmt1; stmt2; ... }`, then you need to use function `append_to_statement_list()`:

```c
tree list=NULL_TREE;
for (i=0; i<num_stmt; i++) {
    BUILD_FUNC_CALL1(t_call, t_send, t_arr[i], NULL_TREE);
    append_to_statement_list(t_call, &list);
}
Adding Nodes to a Tree

GCC 4.1 has an interface that allows one to add a chain of nodes into another chain of nodes implemented in file `tree-iterator.c`. Functions `tsi_start()` and `tsi_last()` create a tree statement iterator and assigns it to the first or the last tree in the list, respectively. Functions `tsi_link_before()` and `tsi_link_after()` link a statement using the iterator either before or after the current statement. There is also function `append_to_statement_list()` that adds a node to a list. If the specified list argument is `NULL_TREE` then a new statement list is allocated.

Building Function and Variable Declarations

A global declaration is added in hook `gem_c_common_nodes_and_builtins()`. In this following example we build a structure type and create a global variable of this type. The structure has a field of type unsigned int and a function pointer field.

```c
    t_log = make_node(RECORD_TYPE);
    decl_chain = NULL_TREE;
    field_decl = build_decl(FIELD_DECL, get_identifier("addr"),
                             unsigned_type_node);
    TREE_CHAIN(field_decl)=decl_chain;
    decl_chain=field_decl;
    DECL_FIELD_CONTEXT(decl_chain) = t_log;
    ...
    t_func_type = build_function_type_list(void_type_node, unsigned_type_node,
                                           NULL_TREE);
    field_decl = build_decl(FIELD_DECL, get_identifier("add_addr"),
                             build_pointer_type(t_func_type);
    TREE_CHAIN(field_decl)=decl_chain;
    decl_chain=field_decl;
    DECL_FIELD_CONTEXT(decl_chain) = t_log;
    ...
    TYPE_FIELDS(t_log) = nreverse(decl_chain);
    layout_type(t_log);
    pushdecl(buildDecl(TYPE_DECL, get_identifier("log_t"), t_log));
    decl = build_decl(VAR_DECL, get_identifier("log"), build_pointer_type(t_log));
    DECL_EXTERNAL(decl)=1;
    pushdecl(decl);
```

7.0.4 When to Instrument

In this section we will describe when each of GEM hooks is used.

- Add new function and type declarations in hook `gem_c_common_nodes_and_builtins`.
- Instrument an AST after it is parsed in hook `gem_finish_function`.
- Modify attributes of a declaration in hooks `gem_start_decl` and `gem_finish_decl`.

Let us say we would like to replace local array declarations `char arr[10]` with a heap array `char *arr=(char*)malloc(10);`
Parameter Passing

```c
void l2h_start_decl(void *p_decl, void *p_declspecs, init initialized, void *p_attr) {
    struct c_declarator *decl = *((struct c_declarator**)p_decl);
    if (current_function_decl == NULL_TREE) return;
    if (decl->kind == cdk_array) {
        decl->kind = cdk_pointer;
        decl->u.pointer_quals = 0;
    }
}

void l2h_finish_decl(tree decl, tree *init, tree spec) {
    ...
    BUILD_FUNC_CALL1(t_call, t_malloc, build_int_cst(NULL_TREE, size),
                     NULL_TREE);
    *init = build1(NOP_EXPR, build_pointer_type(char_type_node), t_call);
    DECL(decl) = build_int_cst(NULL_TREE, 0); // if this field is NULL the init is ignored
}
```

- Replace function call with a proxy function

### 7.0.5 Function Prolog/Epilog

The assembly instructions are written to the assembly file:

```c
#define OUTPUT_ASM_INST(inst) \
 p=inst; \ 
 putc('	', asm_out_file); \ 
 while (*p++) putc(p, asm_out_file); \ 
 putc('
', asm_out_file);
OUTPUT_ASM_INST("pushl %eax");
OUTPUT_ASM_INST("popl %eax");
```

**Take home:** Assembly instructions are added to function prolog and epilog using hooks `gem_output_asm_insn` and `gem_final_start_function`. 

33
8 C Function Overloading

8.1 Function overloading in C

The C function overloading extension aims at including a C++ feature to C language which allows use functions with the same name but different argument types. The cfo/test.c example of GEM demonstrates this feature:

```c
void ec_aa_add(int from, char *to);
void ec_aa_add(int from, int to);
...
```

are used to add an element to a pool data structure. The idea behind the implementation of the extension is to rewrite each function declaration so that the new name includes the type information of function's arguments. In the above case, the modified names are `ec_aa_add_int_char_ptr` and `ec_aa_add_int_int`. The compilation continues normally with the updated names. Because of the above modification, the call names need modification as well. The renaming takes into account the types of the arguments so that the appropriate function is called. For example, the compiler will modify

```c
ec_aa_add(1,2);
```

to

```c
ec_aa_add_int_int(1,2);
```

Three hooks are used in this extension. Functions `cfo_start_decl()` and `cfo_start_function()` intercept declarations. They call `cfo_alias_decl()` that replaces the name using argument types. To preserve the library code for the programs that do not use CFO extension, the following technique is used. If the function name is encountered first time, an alias is created to its old name so that one can invoke the function using either of them. Thus, the legacy code will use the old name, while the CFO-compiled code will use the type-augmented name. Finally, the declaration name is updated to include type information. Hook `cfo_build_function_call()` is invoked from the parser when a new function call is found. It replaces the name of callee function with the name that takes into account the types of actual-in arguments.
9 Return Address Defense

9.1 Return Address Defense

A prerequisite to understanding the proposed extension is the following paper: Tzi-Cker Chiueh, Fu-Hau Hsu, "RAD: A Compile-time Solution to Buffer Overflow Attacks," 21st IEEE International Conference on Distributed Computing Systems (ICDCS), Phoenix, Arizona, USA, April 2001. 1 The paper proposed to instrument function prolog and epilog. The return address is saved in a well-protected buffer so that overwriting stack frame does not hurt it. The copy of the address is cross-checked with the one on the stack to detect a possible attack. Implementing the proposed method has a number of difficulties. Different architectures have different stack frame layout. For example, the x86 prolog has the code to save clobbered registers which is generated in the machine-specific component of GCC. Therefore, instrumenting prolog at the RTL level will not take into account those registers because their number is unknown at that time. There are two approaches to this problem. The first is to instrument prolog at the machine level which is not portable across different architectures. The second approach is to instrument the call-site instead. The idea of the proposed implementation is to find the approximation of the return address using the address of a code label immediately following the call instruction. It is also possible to find the address on the stack where the return address is written using the number of actual arguments that are pushed on the stack. In case of x86 frame layout, the space is reserved at the frame creation time, therefore the stack pointer not modified when a function is called. The return address approximation and the stack pointer are saved in a well-protected buffer, similarly to the original RAD approach. The following code snippet illustrates the idea:

```c
func1() {
    ...
    jmp L2
L1: // the return address is on stack
    push $esp // we also need stack pointer
    rad_prolog() // save the two arguments in a safe buffer
    add $esp, 8 // make sure no traces left
    func2(); // the original call-site
    jmp L3
L2: call L1
L3: ....
```

When the instrumented code is executed, the control flow is directed to a label after the original call instruction. At that point, a call instruction saves the current EIP value and directs the control flow back to label L1. Therefore, the return address approximation argument is already on stack. We also need to push the ESP as the second argument. Function rad_prolog adds the pair to a safe buffer. The arguments are popped and the

1 http://www.ecs1.cs.sunysb.edu/tr/TR96.pdf
original function is called. The added call instruction is jumped over upon return. We also need to instrument function epilog. GIMPLE level works in this case because there is a TRY_FINALLY_EXPR code. The most recent pair of return address and its location in the well-protected buffer are used to validate the return address on the stack. An alarm is generated in case the two values mismatch. This extension does not use any machine-specific information. Therefore, it is portable across different architectures, even though we were unable to test it under anything except x86.
10 Adding Syntactic Sugar

10.1 toString() method for each structure as in Java

10.2 Invoke a block of code from a function as in Ruby

Linux implementation of lists allows to invoke a block of code on each element of the list: pagelist.c:\footnote{http://lxr.linux.no/source/fs/nfs/pagelist.c?v=2.4.20#L199}:

```c
list_for_each_prev(pos, head) {
    struct nfs_page *p = nfs_list_entry(pos);
    if (page_index(p->wb_page) < pg_idx)
        break;
}
```

list_for_each_prev takes the code in the brackets as a parameter. The trick is to use a macro that rolls out to a for() loop whose body becomes the code in the brackets. The goal of this project is to allow programmers to use code blocks in function calls.

10.3 Dereference function results when a structure is returned

C allows one to dereference the return value of a function if it is a pointer to a structure:

```
get_struct()->field=0;
```

If the function returns the structure, not a pointer to it, then a compile-time error is generated:

```
get_struct().field=0;
> request for member 'field' in something not a structure or union
```

This extension addresses the problem of dereferencing structures that are return values.
10.4 Use functions to initialize a variable

When a variable is defined and initialized the initializer is constant. You will get an error if you try to use a function, no matter what this function is:

```c
int getint() { return 1; }
int i=getint();
> initializer element is not constant
```

When a variable is used the function it was initialized with is called.

10.5 Default values of function arguments as in C++

```c
void func(int a=0) {
    printf("a=%d\n", a);
}
int main() {
    func();
}  
> syntax error before '=' token
```

10.6 Reference parameters as in C++

```c
void test(int &a, int &b);
int x,y;
test(x,y);
```

10.7 GCC switches in object file

GCC switches in object file\(^2\)

10.8 Type information at run-time

There is no type information in C language available at run-time. The idea is to allow program to get the names and offsets of a structure's field at run time, the symbolic name of an enum declaration, etc. For example, instead of writing

```c
enum tree_code code;
```

\(^2\) [http://gcc.gnu.org/wiki/RecordGCC%20command%20line%20 switches%20in%20object%20 files](http://gcc.gnu.org/wiki/RecordGCC%20command%20line%20 switches%20in%20object%20 files)
... switch (code) {
    case VAR_DECL:
        printf("VAR_DECL\n");
        break;
    case BLOCK:
        printf("BLOCK\n");
        break;
}
...

one could write

printf("%s\n", type_info(code).name);
11 Improving Code Style

These exercises are from ¹

11.1 Break up enormous source files

Not terribly hard. Watch out for file-scope globals. Suggested targets:

**Big Files**

<table>
<thead>
<tr>
<th>File Size</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>494K</td>
<td>java/parse.y</td>
</tr>
<tr>
<td>413K</td>
<td>combine.c</td>
</tr>
<tr>
<td>408K</td>
<td>dwarf2out.c</td>
</tr>
<tr>
<td>375K</td>
<td>cp/pt.c</td>
</tr>
<tr>
<td>367K</td>
<td>fold-const.c</td>
</tr>
<tr>
<td>356K</td>
<td>loop.c</td>
</tr>
</tbody>
</table>

There are several other files in this size range, which I have left out because touching them at all is unwise (reload, the Fortran front end). You can try, but I am not responsible for any damage to your sanity which may result.

11.2 Break up enormous functions

This is in the same vein as the above, but significantly harder, because you must take care not to change any semantics. The general idea is to extract independent chunks of code to their own functions. Any inner block that has a half dozen local variable declarations at its head is a good candidate. However, watch out for places where those local variables communicate information between iterations of the outer loop! With even greater caution, you may be able to find places where entire blocks of code are duplicated between large functions (probably with slight differences) and factor them out.

11.3 Break up enormous conditionals

Harder still, because it's unlikely that you can tell what the conditional tests, and even less likely that you can tell if that's what it's supposed to test. It is definitely worth the effort if you can hack it, though. An example of the sort of thing we want changed:

```c
if (mode1 == VOIDmode
 || GET_CODE (op0) == REG || GET_CODE (op0) == SUBREG
 || (modifier != EXPAND_CONST_ADDRESS
 & & ! direct_load[(int) mode1]
 & & GET_MODE_CLASS (mode) != MODE_COMPLEX_INT
 & & GET_MODE_CLASS (mode) != MODE_COMPLEX_FLOAT)
/* If the field isn't aligned enough to fetch as a memref, fetch it as a bit field. */
 || (model != BLKmode
 & & SLOW_UNALIGNED_ACCESS (model, alignment)
 & & ((TYPE_ALIGN (TREE_TYPE (tem))
 < GET_MODE_ALIGNMENT (mode))
 || (bitsize > 0
 & & (TREE_CODE (TYPE_SIZE (TREE_TYPE (exp)))
 == INTEGER_CST)
 & & 0 != compare_tree_int (TYPE_SIZE (TREE_TYPE (exp)),
 bitsize))))
 || (modifier != EXPAND_CONST_ADDRESS
 & & node == BLKmode
 & & SLOW_UNALIGNED_ACCESS (mode, alignment)
 & & (TYPE_ALIGN (type) > alignment
 || bitpos % TYPE_ALIGN (type) != 0)))
{
```

11.4 Delete garbage

#If 0 blocks that have been there for years, unused functions, unused entire files, dead configurations, dead Makefile logic, dead RTL and tree forms, and on and on and on. Depending on what it is, it may not be obvious if it's garbage or not. Go for the easy ones first.

11.5 Use predicates for RTL objects

GCC has simple predicates to see if a given rtx is of some specific class. These predicates simply look at the rtx_code of the given RTL object and return nonzero if the predicate is true. For example, if an rtx represents a register, then REG_P (rtx) is nonzero. Unfortunately, lots of code in the middle end and in the back ends does not use these predicates and instead compare the rtx_code in place: (GET_CODE (rtx) == REG). Find all the places where such comparisons can be replaced with a predicate. Also, for many common comparisons there is no predicate yet. See which ones are worth having a predicate for, and add them. You can find a number of suggestions in the mailing list archives.
12 Security Enhancements

12.1 Return address protection (RAD)
Rad\textsuperscript{1}

12.2 Repair of control-hijacking attacks (DIRA)
Dira\textsuperscript{2}

12.3 Array bounds checking using segmentation hardware (CASH)
Cash\textsuperscript{3}

12.4 Detecting integer overflows (DIVINE)
Divine\textsuperscript{4}

12.5 Dynamic Information Flow Tracking (GDIF)
Gdif\textsuperscript{5}

\footnotesize
\textsuperscript{1} http://www.ecsl.cs.sunysb.edu/RAD
\textsuperscript{2} http://www.ecsl.cs.sunysb.edu/dira
\textsuperscript{3} http://www.ecsl.cs.sunysb.edu/cash
\textsuperscript{4} http://www.ecsl.cs.sunysb.edu/iop
\textsuperscript{5} http://www.ecsl.cs.sunysb.edu/~lclam/gdif/
## 13 Links

1. GCC website\(^1\)
2. #ref\_gcc\_summit\(^2\)
3. #ref\_gem\_url\(^4\)

\[
\begin{align*}
1 & \quad http://gcc.gnu.org \\
2 & \quad #ref\_gcc\_summit \\
3 & \quad http://www.gccsummit.org \\
4 & \quad #ref\_gem\_url \\
5 & \quad http://research.alexsmirnov.name/gem
\end{align*}
\]
14 Contributors
15 GNU Free Documentation License

As of July 15, 2009 Wikibooks has moved to a dual-licensing system that supercedes the previous GFDL only licensing. In short, this means that text licensed under the GFDL only can no longer be imported to Wikibooks, retroactive to 1 November 2008. Additionally, Wikibooks text might or might not now be exportable under the GFDL depending on whether or not any content was added and not removed since July 15.

Version 1.3, 3 November 2008 Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <\(^1\)> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

15.1 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others. This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software. We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

15.2 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as

\(^1\) http://fsf.org/
"you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law. A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language. A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them. The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none. The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words. A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque". Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only. The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text. The "publisher" means any person or entity that distributes copies of the Document to the public. A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition. The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only
as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

15.3 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3. You may also lend copies, under the same conditions stated above, and you may publicly display copies.

15.4 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects. If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages. If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public. It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

15.5 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely
this License, with the Modified Version filling the role of the Document, thus licensing
distribution and modification of the Modified Version to whoever possesses a copy of it. In
addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the
   Document, and from those of previous versions (which should, if there were any, be
   listed in the History section of the Document). You may use the same title as a
   previous version if the original publisher of that version gives permission.

2. List on the Title Page, as authors, one or more persons or entities responsible for
   authorship of the modifications in the Modified Version, together with at least five
   of the principal authors of the Document (all of its principal authors, if it has fewer
   than five), unless they release you from this requirement.

3. State on the Title page the name of the publisher of the Modified Version, as the
   publisher.

4. Preserve all the copyright notices of the Document.

5. Add an appropriate copyright notice for your modifications adjacent to the other
   copyright notices.

6. Include, immediately after the copyright notices, a license notice giving the public
   permission to use the Modified Version under the terms of this License, in the form
   shown in the Addendum below.

7. Preserve in that license notice the full lists of Invariant Sections and required Cover
   Texts given in the Document's license notice.

8. Include an unaltered copy of this License.

9. Preserve the section Entitled "History", Preserve its Title, and add to it an item
   stating at least the title, year, new authors, and publisher of the Modified Version as
   given on the Title Page. If there is no section Entitled "History" in the Document,
   create one stating the title, year, authors, and publisher of the Document as given
   on its Title Page, then add an item describing the Modified Version as stated in the
   previous sentence.

10. Preserve the network location, if any, given in the Document for public access to a
    Transparent copy of the Document, and likewise the network locations given in the
    Document for previous versions it was based on. These may be placed in the "History"
    section. You may omit a network location for a work that was published at least four
    years before the Document itself, or if the original publisher of the version it refers to
    gives permission.

11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title
    of the section, and preserve in the section all the substance and tone of each of the
    contributor acknowledgements and/or dedications given therein.

12. Preserve all the Invariant Sections of the Document, unaltered in their text and in
    their titles. Section numbers or the equivalent are not considered part of the section
    titles.

13. Delete any section Entitled "Endorsements". Such a section may not be included in
    the Modified version.

14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in
    title with any Invariant Section.

15. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as
Secondary Sections and contain no material copied from the Document, you may at your
option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles. You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard. You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one. The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

15.6 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers. The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work. In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

15.7 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects. You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.
15.8 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

15.9 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail. If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

15.10 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License. However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation. Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice. Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not
10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Doc-
umentation License from time to time. Such new versions will be similar in spirit to the
present version, but may differ in detail to address new problems or concerns. See 2. Each
version of the License is given a distinguishing version number. If the Document specifies
that a particular numbered version of this License "or any later version" applies to it, you
have the option of following the terms and conditions either of that specified version or of
any later version that has been published (not as a draft) by the Free Software Foundation.
If the Document does not specify a version number of this License, you may choose any
version ever published (not as a draft) by the Free Software Foundation. If the Document
specifies that a proxy can decide which future versions of this License can be used, that
proxy's public statement of acceptance of a version permanently authorizes you to choose
that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web
server that publishes copyrightable works and also provides prominent facilities for anybody
to edit those works. A public wiki that anybody can edit is an example of such a server.
A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set
of copyrightable works thus published on the MMC site. "CC-BY-SA" means the Creative
Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation,
a not-for-profit corporation with a principal place of business in San Francisco, California,
as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of
another Document. An MMC is "eligible for relicensing" if it is licensed under this License,
and if all works that were first published under this License somewhere other than this
MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover
texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008. The
operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA
on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

---

2 http://www.gnu.org/copyleft/
16 How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation. If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Category¹:

• GNU C Compiler Internals²

¹ http://en.wikibooks.org/wiki/Special:Categories
² http://en.wikibooks.org/wiki/Category:GNU_C_Compiler_Internals
# 17 Contributors

<table>
<thead>
<tr>
<th>Edits</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Adrignola$^1$</td>
</tr>
<tr>
<td>1</td>
<td>Darklama$^2$</td>
</tr>
<tr>
<td>1</td>
<td>DavidCary$^3$</td>
</tr>
<tr>
<td>1</td>
<td>Derbeth$^4$</td>
</tr>
<tr>
<td>1</td>
<td>Dirk Hünniger$^5$</td>
</tr>
<tr>
<td>1</td>
<td>Jomegat$^6$</td>
</tr>
</tbody>
</table>


List of Figures

- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- CFR: Copyright free use.

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses\textsuperscript{7}. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your web browser.
List of Figures

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The people from the Tango! project⁸</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Az1568, Webaware</td>
</tr>
</tbody>
</table>

⁸ http://tango.freedesktop.org/The_People
The GNU General Public License is a copying and distribution license that allows free reproduction and distribution of software code. It is designed to protect the freedom of users to change and distribute software in a way that best suits their needs. The license is based on the idea that software is a tool, not a commodity, and that users should have the right to control how they use and modify it. The GNU General Public License (GPL) is a copyleft license that requires that any derivative work of a GPL-licensed work also be released under the GPL.

The license is aimed at promoting the sharing of software, ensuring that it remains free and open to all. It allows anyone to copyleft a program, making it available for others to use, modify, and distribute, provided that any derivative work is also released under the GPL.

The license includes terms for the distribution of source code, which must be made available to users along with the binary code. It also includes provisions for the use of the GPL in conjunction with other licenses, allowing for flexibility in how software can be distributed.

The GNU General Public License is a critical component of the free software movement, ensuring that software remains free and open to all while allowing developers to earn a living from their work. It has been adopted by many software projects and is a cornerstone of the free software community.

18.1 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.
895 Massachusetts Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

The GNU General Public License (GPL) is a popular and effective free software license that ensures the freedom of users to run, study, copy, distribute, and modify free software. The GPL is a copyleft license, which means that any derivative work of a GPL-licensed work must also be released under the GPL. The primary goal of the GPL is to protect and ensure the freedom of users to continue using and changing the software. It is often used for GNU software projects, and its principles are also applied to other Free Software projects.

A 'covered work' under the GPL refers to anything that runs under the GPL, includes a work derived from any covered work, and includes any complete or incomplete work that contains a notice or copyright notice that is required to be included with the corresponding source code. The GPL applies to the software modules that are combined with the source code to form a complete work.

The GPL is designed to ensure that users have the freedom to use, study, modify, and distribute the software. It requires that the source code be made available along with the compiled code, allowing users to understand and modify the software as needed. This helps to promote the sharing and improvement of software, which is a key aspect of the free software movement.

By using software that is released under the GPL, you are agreeing to respect the terms and conditions of the license, which ensure the continued availability and freedom of the software for all users. This license is an important tool for supporting the free software movement and ensuring that software remains free and open to all.
18.2 GNU Free Documentation License

Version 1.3, 1 November 2008


Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and unaltered work, freed from copyright risk and suit, free for all time with respect to the License and in the various countries where copyrights are in effect.

To provide a well-understood medium of communication of software and other kinds of human knowledge, the Free Software Foundation, Inc. (FSF), and others (collectively "the Author(s)") have made available to the public domain a versatile way to distribute and modify the Free Software Foundation's software.
The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL. You may convey a combined work under sections 1 and 2 of this license without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reason engineer for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: 1) Convey the source code for the Combined Work under the terms of License, or 2) Convey an object code version of the Combined Work under the terms of your choice; if a modified version of the Linked Version is not available under the terms of your choice, the Library is deemed part of the Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. * e) Provide Installation Information, but only if you do so to the extent that such information is necessary to install and use a modified version of the Linked Version. 3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.


You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reason engineer for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: 1) Convey the source code for the Combined Work under the terms of License, or 2) Convey an object code version of the Combined Work under the terms of your choice; if a modified version of the Linked Version is not available under the terms of your choice, the Library is deemed part of the Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. * e) Provide Installation Information, but only if you do so to the extent that such information is necessary to install and use a modified version of the Linked Version. 6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version published by the Free Software Foundation. If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.