SoftBench SDK: CodeAdvisor and Static Programmer's Guide

HP Part No. B6454-90005
Printed in USA February 1998

E0298

Notices

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty. A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Hewlett-Packard Sales and Service Office.

Printing History

New editions of this manual incorporate all material updated since the previous edition.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates incorporated at reprint do not cause this date to change.) The manual part number changes when extensive technical changes are incorporated.

January 1996 Edition 1 (B5073-90004)

February 1998 Edition 1 (B6454-90005)

Preface

This manual describes how to write new rules for the SoftBench CodeAdvisor product. It also documents the Static Database Application Programmer's Interace (API) for programmers who need to access the API for other purposes.

The following reference pages are available online via the `man` command:

softbench(5)	A high-level general description of SoftBench with a listing of generic command line options for all SoftBench tools, including C and C++ encapsulations
softcheck(1)	A detailed description of the `softcheck` command, which implements the SoftBench CodeAdvisor rule engine

On-line help is also available by pressing the Help key (usually F1 or Help) on any SoftBench tool.

Typeface Conventions

Convention	Description
italic font	Information you supply, either in syntax examples or in text descriptions. For example, if told to type: *filename*, you supply an actual file name like `sample`. Italics are also used for *emphasis*, and for *Titles of Books*.
`typewriter font`	Computer commands or other information that must be typed exactly as shown. For example, if told to type: `sample`, you type exactly the word in typewriter font, `sample`. Menu selections are in typewriter font separated by colons. See "Menu Conventions" in this chapter.
boldface font	A term that may need further clarification or definition, especially a familiar word (such as **menu**) used with a computer-specific meaning. These terms are clarified in the glossary.
[. . .]	Optional parameters in syntax examples are enclosed in brackets.
KeyCap	Represents a key on your keyboard that you must press, or an on-screen button that you must select, as part of the operation. For example, Return is the "Carriage Return" key, which completes a command input. This key may be labelled "`RETURN`", "`Return`", or "`Enter`".
Key1-Key2	A hyphen between keys indicates that two or more keys must be pressed at the same time. For example, "Control-C" means to press and hold the Control key while pressing and releasing the C key. The Control key may be labelled "`CTRL`", "`Ctrl`", or "`Control`".

Contents

1. User Defined CodeAdvisor Rules

2. Modifying Table-Driven Rules

Modification Process 2-1
Table Formats 2-1
Specifying Scope of Changes 2-2
The `NameConventions` Rule Family 2-3
Rule Format 2-3
Examples of Use 2-5
Extending `NameConventions` 2-5
The `ProhibIdent` Rule Family 2-6
Rule Format 2-6
Examples of Use 2-7
Extending `ProhibIdent` 2-7
The `ProhibDefines` Rule Family 2-8
Rule Format 2-8
Examples of Use 2-8
Extending `ProhibIdent` 2-9
The `DtorMatchCtor` Rule Family 2-10
Rule Format 2-10
Examples of Use 2-11

3. Understanding the Programming Model

The Rule Engine 3-2
The Rule Base Class 3-3
Example Rule 3-7
The RuleWithTable Base Class 3-10
Example Table-Driven Rule 3-14

4. Understanding the Static Database

Database Objects 4-1
Capabilities of the Database 4-2
Learning the Database API 4-3
Database Objects 4-3
Incomplete Objects 4-6
Database Types 4-6
Type Qualifiers 4-7
Accessing the Database 4-8
Opening and Closing the Database 4-8
Delimiting Transactions 4-9
Iterators 4-10
Attribute Iterators 4-11
Object Interfaces 4-12
Block Object 4-13
Class Object 4-14
Example 4-17
ClassTemplate Object 4-18
DataMember Object 4-19
Enum Object 4-20
EnumMember Object 4-21
File Object 4-22
Function Object 4-24
FunctionMember Object 4-26
FunctionTemplate Object 4-27
Label Object 4-28
Macro Object 4-29
Parameter Object 4-30
The PerBase Base Class 4-31
RefList Object 4-32
Example 4-34
Scalar Object 4-35
Struct Object 4-36
The Symbol Base Class 4-38
The SymbolTable Class 4-41
Tag Object 4-45
TemplateArgument Object 4-46
Typedef Object 4-47

The TypedSymbol Base Class 4-48
Variable Object 4-49
Using the Database API 4-50
The Example Rule 4-50
Understanding the Example Rule 4-50
The shadow Function 4-50
kindMask and langMask 4-51
The check Function 4-51
Final Definitions 4-52
Example Files 4-53
The UserRulesLocalHides Rule 4-55

5. Implementing Your Rule
Design Guidelines 5-1
Implementing the Rule 5-3
Decide What to Implement 5-3
Designing the Rule 5-4
Compiling the Rule 5-5
Testing the Rule 5-5
Adding Your Rule to a Rule Group 5-7
Classifying Your Rule 5-7
Rulegroup File Locations 5-7
Rulegroup File Format 5-8
Creating a New Rule Group 5-8
Updating the Group Index 5-9
Debugging Your Rule 5-10
Running softcheck Under SoftBench Debugger 5-10
Setting Breakpoints In Your Rule 5-12
Tracing Rule Execution 5-13
Documenting Your Rule 5-14
Writing the On-Line Help 5-14
Referring to Other Help Volumes 5-15
Associating Your Rule With the On-Line Help 5-15
Installing the On-Line Help Volume 5-15

A. Detailed Database Type Descriptions
Object Kind A-2
Attributes A-3
Scalar Types A-5
Language Types A-6
References A-7
Error Codes A-8

B. Iterators
Standard Iterators B-2
Attribute Iterators B-4

Index

Figures

4-1. Object Hierarchy 4-5
4-2. RefList Organization 4-33

1

User Defined CodeAdvisor Rules

SoftBench CodeAdvisor offers you a powerful tool for improving the reliability and maintainability of your C and C++ code. Many predefined rules come with the SoftBench CodeAdvisor product, allowing you to benefit from the product "right out of the box."

You can also extend the SoftBench CodeAdvisor functionality to meet your local needs.

- **The simplest way to customize the SoftBench CodeAdvisor product is to modify the ASCII files that are read by existing table-driven rules.** See Chapter 2 for information on this process.
- You can also create your own rules using the rule library interface. SoftBench CodeAdvisor uses the SoftBench Static Analyzer database as its "view" on your program. You must understand the Static database before you can begin to write rules.

  Adding a rule to the SoftBench CodeAdvisor rule set requires several steps:

  1. Understand the SoftBench CodeAdvisor programming model. Study the sample classes and examples in this manual.
  2. Understand the Static database. Learn its capabilities and limitations. Learn the API (application programmer interface) used to interact with the database.
  3. Design your rule, using the features of the database.
  4. Write the rule and link it into the rule-checking environment. Add the rule to a rule group. Test and debug the rule.
  5. Document the new rule.

  These steps are documented starting in Chapter 3. That section assumes you have some experience with C++ programming.

2

Modifying Table-Driven Rules

Several rules shipped with the SoftBench CodeAdvisor product read their definitions from ASCII files. By modifying the files, you can modify the rules' behavior. You can add new rule cases, delete current rules, or change a rule's definition. You need not do *any* programming; you simply edit a text file.

Modification Process

To modify table-driven rules, you edit or replace the table in the ASCII file to meet your needs. The exact changes you make will depend on the desired scope of the changes (how many users are affected) and the rule being changed. (See "Specifying Scope of Changes".)

Table Formats

Each rule table actually encompasses a *family* of related rules. Many rule ID's can be defined in a single table. This allows you to specify different filtering and different online help for each rule ID in the table.

Each non-comment line defines a separate rule ID. The rule ID is the first field in the line, and subsequent fields specify the exact values used by the rule. Each family of rules specifies the format of its file, and the meaning of the fields in the file. Lines starting with **#** are interpreted as a comment.

See the sections later in this chapter for details on each rule family.

Specifying Scope of Changes

Your changes and additions can affect different scopes, depending on where you make the change or addition. SoftBench CodeAdvisor checks several locations for rule table information:

`/opt/softbench/config/ruletables/$LANG/`*rule-family*

> Standard pre-configured rule tables, as defined by Hewlett-Packard. Ordinarily you should not change these files, but you may copy them to create your own rule file.

`/etc/opt/softbench/config/ruletables/$LANG/`*rule-family*

> Local changes and customizations. All users on the system are affected by these changes. If any rule table is present under the `/etc/opt/softbench` hierarchy, it totally *replaces* the corresponding rule table under `/opt/softbench`.

`$HOME/.softbench/ruletables/`*rule-family*

> Personal changes. Visible within all projects for that user.

`$PROJECTROOT/Projects/`*project-name*`/ruletables/`*rule-family*

> Personal changes. Visible only within the specified project.

The locations are checked in the order above. Later information overrides previous information; for example, personal customizations under `$HOME/.softbench` are merged in with the system-wide customization in `/etc/opt/softbench/config`, and override it on a rule-by-rule basis.

rule-family is the name of the rule table to be modified. In most cases it actually contains a collection of rules, since each rule table can define many rule ID's.

`$PROJECTROOT` points to the user's specified project information root. By default, this root is `$HOME/.softbench`. *project-name* is the name of the specific project to customize.

The NameConventions Rule Family

NameConventions allows you to specify almost any kind of required or prohibited condition in an identifier name. For example, you can create a rule that requires all class names to be capitalized, or that flags the use of certain prohibited characters.

Rule Format

Each line contains the following space-delimited fields:

Rule ID	Name of the rule. The same Rule ID cannot appear on multiple lines.
Help Volume	Name of the help volume that contains online help for the rule. Within that help volume, the rule ID is used as the help node name. To specify a different node (for example, to have several rules share the same help node), use the format *helpvolume_nodename*.
Kinds to Check	The "kind" of object that the rule applies to. (See "Object Kind" in Appendix A for a list of all "kinds" understood by SoftBench CodeAdvisor.) Within the NameConventions rule file, the "kind" must be one or more of: CLASS, CLASSTEMPLATE, DATAMEMBER, ENUM, ENUMMEMBER, FUNCTION, FUNCTIONMEMBER, FUNCTIONTEMPLATE, IDENTIFIER, LABEL, MACRO, PARAMETER, SCALAR, SOURCEFILE, STRUCT, TAG, TEMPLATEARGUMENT, TYPEDEF, UNION, VARIABLE. Multiple "kinds" are separated by a vertical bar (|).
Regular Expressions	An Extended Regular Expressions, as documented in *regexp(5)*. ERE's can contain multiple Regular Expressions separated by a vertical bar (|).
Match	Specifies whether the rule fires on identifiers that *match* or *do not match* the Regular Expression. MATCH indicates that the rule signals a violation on identifiers that *match* the Regular Expression; NOMATCH indicates that the rule signals a violation on identifiers that *do not match* the Regular Expression.

Required Attributes	Specifies all attributes that must be set on the identifier. (See "Attributes" in Appendix A for a list of all attributes understood by SoftBench CodeAdvisor.) This field can contain the keyword "`ANY`", meaning that there are no required attributes, or one or more of the following attributes separated by ampersands (`&`): `ABSTRACT`, `ANONMEM`, `COMPILE_ERRORS`, `CONST`, `DECLARED_STRUCT`, `DECLARED_UNION`, `GLOBAL`, `INLINED`, `INSTANTIATED`, `MERGE_MEMBERS`, `PRIVATE`, `PROTECTED`, `PROTECTED`, `PUBLIC`, `PURE`, `SPECIALIZATION`, `STATIC`, `SYNTHETIC`, `VOLATILE`.
Prohibited Attributes	Specifies all attributes that must not be set on the identifier. This field can contain the keyword "`ANY`", meaning that there are no prohibited attributes, or one or more of the attributes specified above separated by vertical bars.
Error Message	A message describing the condition that has been violated. The `printf(3)` format specifier `%s` should be included in the message. It will be replaced by the erroneous identifier name.

For example, the following line can be found in the `NameConventions` rule file:

```
NoUnderscoreOnExtern CommonCxx VARIABLE|FUNCTION ^_
  MATCH GLOBAL STATIC Identifier '%s' with external
  linkage beginning with underscore is in C language
  implementation reserved namespace
```

(This example has been broken into several lines for readability. It must appear on one line in the rule table.)

This line defines the `NoUnderscoreOnExtern` rule, which specifies that no externally-linked (global) identifiers may begin with an underscore. The online help for this rule is found in help volume `CommonCxx`, node `NoUnderscoreOnExtern`. The rule applies to `VARIABLE` and `FUNCTION` identifiers, and fires on identifiers that *match* the regular expression "`^_`" (underscore at the beginning of the identifier). The rule applies only to `GLOBAL` identifiers that are *not* `STATIC`.

Examples of Use

The rules shipped with SoftBench CodeAdvisor use `NameConventions` to detect problems such as:

- Illegal identifiers, such as global IDs beginning with underscore
- Stylistic conventions, such as non-capitalized class names

You can create additional rules like these to support your local conventions.

Extending `NameConventions`

The source for the `NameConventions` rule family can be found in `/opt/softbench/examples/CodeAdvisor/Rules/ruleNameConventions.C`. You can use this source to extend `NameConventions` for your local needs. See Chapter 3 and later chapters for information on writing rules.

The `ProhibIdent` Rule Family

`ProhibIdent` checks for prohibited identifier names. This includes calls to unsafe functions, uses of obsolete functions and variables, and other similar situations.

Rule Format

Each line contains the following space-delimited fields:

Rule ID	Name of the rule. Usually the name of the prohibited identifier is also used as the name of the rule.
Help Volume	Name of the help volume that contains online help for the rule. Within that help volume, the rule ID is used as the help node name. To specify a different node (for example, to have several rules share the same help node), use the format *helpvolume_nodename*.
Identifier	The prohibited identifier.
Kinds to Check	Identifier types to check. See the `NameConventions` rule for the list of accepted "kinds." Multiple "kinds" are separated by a vertical bar (`|`).
Error Message	A message describing the condition that has been violated. The message should identify the invalid identifier name, and explain why it is prohibited.

For example, the following line can be found in the `ProhibIdent` rule file:

```
gets CommonCxx gets FUNCTION gets can overflow buffer if input exceeds buffer size
```

This line defines the `gets` rule, which prohibits the use of the `gets` function. Note that the rule does not flag the use of local variables or parameters named `gets`, since it only applies to `FUNCTION` identifiers. The online help for this rule is found in help volume `CommonCxx`, node `gets`. The error message is a simple text string.

Examples of Use

The rules shipped with SoftBench CodeAdvisor use `ProhibIdent` to detect the use of unsafe, obsolete, and non-portable identifiers. You may add your own rules to prohibit the use of other identifiers.

Extending `ProhibIdent`

The source for the `ProhibIdent` rule family can be found in `/opt/softbench/examples/CodeAdvisor/Rules/ruleProhibIdent.C`. You can use this source to extend `ProhibIdent` for your local needs. See Chapter 3 and later chapters for information on writing rules.

The `ProhibDefines` Rule Family

`ProhibDefines` checks for prohibited identifier names, but it uses a more specialized algorithm than the `ProhibIdent` rule. `ProhibDefines` looks for identifiers that are not allowed in `#define` macros or in `-D` definitions on the compiler command line.

Rule Format

Each line contains the following space-delimited fields:

Rule ID	Name of the rule. Usually the name of the prohibited identifier is also used as the name of the rule.
Help Volume	Name of the help volume that contains online help for the rule. Within that help volume, the rule ID is used as the help node name. To specify a different node (for example, to have several rules share the same help node), use the format *helpvolume_nodename*.
Identifier	The prohibited identifier.
Error Message	A message describing the condition that has been violated. The message should identify the invalid identifier name, and explain why it is prohibited.

For example, the following line can be found in the `ProhibDefines` rule file:

```
_FILE64 CommonCxx_Port419 _FILE64 _FILE64 - may not be portable
```

This line defines the `_FILE64` rule, which prohibits the use of the `_FILE64` define. The online help for this rule is found in help volume `CommonCxx`, node `ProhibDefines`. The error message is a simple text string.

Examples of Use

The rules shipped with SoftBench CodeAdvisor use `ProhibDefines` to detect unsafe, obsolete, and non-portable define identifiers. You may add your own rules to prohibit the use of other identifiers.

Extending `ProhibIdent`

The source for the `ProhibIdent` rule family can be found in `/opt/softbench/examples/CodeAdvisor/Rules/ruleProhibIdent.C`. You can use this source to extend `ProhibIdent` for your local needs. See Chapter 3 and later chapters for information on writing rules.

The `DtorMatchCtor` Rule Family

`DtorMatchCtor` verifies that resources allocated in a class's constructor are deallocated in the destructor. Furthermore, the deallocator must match the allocator. For example, you cannot allocate memory using `new` and deallocate it using `free`.

Rule Format

Each line contains the following fields. Fields in this rule table are separated by vertical bars (`|`), since some of the field values (such as "`operator new`") have embedded spaces.

Rule ID	Name of the rule. The same Rule ID cannot appear on multiple lines.
Help Volume	Name of the help volume that contains online help for the rule. Within that help volume, the rule ID is used as the help node name. To specify a different node (for example, to have several rules share the same help node), use the format *helpvolume_nodename*.
Deallocators	One or more deallocator functions, separated by commas. Any of the specified Deallocators can be used to release resources allocated by any of the Allocators.
Allocators	One or more allocator functions, separated by commas. Any of the specified Allocators can be used to allocate resources that are later released by any of the Deallocators.
Error Message	A message describing the condition that has been violated. The following `printf(3)` format specifiers should be included in the message, in order, and will be replaced by the corresponding information: `%d` Number of calls to the allocators `%s` List of allocators `%s` Name of the constructor `%s` File name where the constructor is located `%d` Line number in the file where the constructor is located

For example, the following line can be found in the `DtorMatchCtor` rule file:

```
DtorMCtorXDeviceList|CommonCxx_DtorMatchCtor|
  XHPFreeDeviceList,XFreeDeviceMotionEvents|
  XHPListInputDevices,XGetDeviceMotionEvents|
  %d call(s) to (one of) %s in %s (file %s, line %d)
  not deallocated
```

(This example has been broken into several lines for readability. It must appear on one line in the rule table.)

This line defines the `DtorMCtorXDeviceList` rule, which specifies that X DeviceList objects allocated by `XHPListInputDevices` or `XGetDeviceMotionEvents` must be deallocated by `XHPFreeDeviceList` or `XFreeDeviceMotionEvents`. The online help for this rule is found in help volume `CommonCxx`, node `DtorMatchCtor`.

Examples of Use

The rules shipped with SoftBench CodeAdvisor use `DtorMatchCtor` to check most allocator/deallocator pairs in standard HP-UX libraries. You may add your own allocator/deallocator pairs for locally-used resource managers.

Note, however, that this rule *only* checks allocators in constructors, and deallocators in destructors. It does not check allocators or deallocators used in the normal flow of other code. Some of the Flow Analysis rules attempt to detect errors of this type.

3

Understanding the Programming Model

The SoftBench CodeAdvisor architecture implements rules in shared libraries. When the rule engine initializes itself, it reads in all the rule libraries it can find and invokes these rules as appropriate.

You can add your own rules by creating libraries for the rule engine to read. Your libraries will contain C++ code that define classes to implement the rules. SoftBench CodeAdvisor defines the `Rule` and `RuleWithTable` base classes, and the interface through which your rule is invoked.

Note that you do not need to write a `main()` procedure for your rules. Your rules exist in a shared library, and are not intended to be run by themselves. The library is loaded and called by the rule engine in the `softcheck` command.

The Rule Engine

SoftBench CodeAdvisor loads in all the rule libraries it finds in `/opt/softbench/lib/rulelibs`, `/etc/opt/softbench/lib/rulelibs`, and any directories specified by the `-l` option to `softcheck`. For each `Rule` or `RuleWithTable` in a rule library, exactly one instance of the rule must be created. The C++ code that defines the rule instance should be of the form:

```
static NewRuleClass instance;
```

All global data members are initialized when the shared library is loaded. Since the instance of *NewRuleClass* is global, this code forces a call to the `Rule` base class constructor in the main rule engine. The constructor notifies the rule engine of the existence of the new rule. The rule engine then calls each rule for all symbols that meet the rule's criteria (or only once, if that is what the rule specifies).

Once all rule libraries are loaded, and all rules are initialized, the rule engine scans through the Static database. For each symbol found in the database, the rule engine determines if any rules have expressed an interest in that symbol type, using the `kindMask()` and `langMask()` member functions documented in the next section.

If one or more rules are interested in that symbol, the rule engine calls the `check()` or `check_with_table()` function for each appropriate rule. The rule functions do whatever testing is required by the rule's definition. If the rule detects a violation, it signals the violation by calling the `violation()` function.

The Rule Base Class

Non-table-driven rules are written as a class derived from the `Rule` base class. `Rule` defines the interface functions required of all rules.

A `Rule` can define a single rule, or it can define a "multi-rule" that can issue violations on any of several closely related rules. This can provide significant performance benefits, since you can iterate through interesting objects (such as the base classes of a class) only once and check for several conditions.

The public interface to `Rule` is defined as follows:

```
class Rule {
public:
   Rule();
   virtual ~Rule() {};    // to ensure derived class objects destructors are
                          // called even when it's deleted through a Rule ptr

   // Returns a mask of the kind of symbols this rule checks.
   virtual int kindMask() const = 0;

   // Returns a mask of the language(s) this rule applies to.
   virtual Language langMask() const = 0;

   // The member function check() is called when the engine has found a
   // symbol of interest to the rule and the rule should be checked.
   virtual void check(SymbolTable *symtab, const Symbol &sym)=0;

   // Returns a one-line summary of the violation with no
   // instance-specific information.
   virtual const char *errorMess() const = 0;

   // Returns the name of this rule, if only one rule defined.
   virtual const char *name() const;

   // Returns a NULL-terminated list of rule names, if multiple rules defined.
   // Should be persistent and should not require deallocation.
   virtual const char *const *names();

   /////// End of functions to be defined by each rule ///////

   // When check() find a rule violation, it calls violation() with the
   // violating symbol and possibly, reference site.  Do not override these
   // functions.
   void violation(const Symbol &sym, const char *err,
                  const char *help_volume,
                  const char *rule_name = NULL);
   void violation(const char *file, const int line, const char *err,
```

```
                    const char *help_volume,
                    const char *rule_name = NULL);

    // When defining a multi-rule, call report() to determine if
    // a particular sub-rule should be checked.
    DBboolean report(const char *name) const;
};
```

You should not access the other public members, the data members, or the friend functions of the `Rule` class. They are used by the rule engine.

You must provide your own versions of all the pure virtual functions: `kindMask()`, `langMask()`, `check()`, and `errorMess()`. You must also define `name()` (if defining a single rule) or `names()` (if defining a multi-rule).

The `Rule` public interface functions are:

`Rule()` — Class constructor. As in any C++ class definition, you should implement a constructor to do any initialization (allocating memory, initializing data structures, and so on) required by your rules. Normally, however, you do not need to define your own constructor.

`~Rule()` — `Rule` objects are not currently deleted. The `Rule` class defines a destructor as a placeholder. If your rule does some operation that should be cleaned up (for example, if you allocate memory), you should define a destructor to do the appropriate cleanup action.

`kindMask()` — Returns a bitmask that tells what kinds of `Symbols` are checked by the rule. Symbol kinds are defined by the enumeration `PerKind` in the header file `DB_Common.h`. (See "Object Kind" in Appendix A for a listing.) The bitmask values are created by using the `PerKind` enumeration values to shift a bit into the appropriate field. A rule can handle several kinds of `Symbol`s by OR-ing the values together. For example, a rule that checks macros and functions should return a `kindMask` of "`1 << KIND_MACRO | 1 << KIND_FUNCTION`".

Note that you cannot specify `KIND_CLASS`, `KIND_CLASSTEMPLATE`, `KIND_STRUCT`, or `KIND_ENUM`, since these object types do not inherit from `Symbol`. Specify

KIND_TAG to receive all objects of these types. See the next section ("Example Rule") for an example.

As a special case, a value of 0 indicates the rule should be called only once for all symbols. You are then responsible for handling any iteration required by your rule. See /opt/softbench/examples/CodeAdvisor/Rules/ruleMixedIO.C for an example.

`langMask()` Returns a bitmask that tells which languages the rule applies to. Languages are defined in DB_Common.h. Language values do not need to be shifted, but can be used as they are defined. As an example, a rule that applies to C and C++ should return a langMask of "LANGUAGE_C | LANGUAGE_CPP". Return LANGUAGE_UNKNOWN if the rule applies to all languages.

`check()` The main rule-check function. check() is called for every symbol in the database matching the types described by kindMask(). The rule engine passes check() the SymbolTable of the Static database and the Symbol that matches the kindMask() and appears in a file matching langMask(). check() accesses all elements of the program through the database. See Chapter 4.

`errorMess()` Returns a string that gives a generic one-line summary of the rule.

`name()`, `names()` Return the name or names of the rule(s). If the rule defines only one rule, name() should return the ID of that rule. If the rule defines multiple rules, names() should return a NULL-terminated list of all rule ID's defined by the rule.

`violation()` If your rule's check() function finds a violation, it should call violation() to report the violation.

There are two variations of violation(). The first is used when a problem is found in a symbol definition or declaration. This form locates the definition of the symbol (or the declaration if no definition is found) for browsing purposes. The second form is used when a specific usage problem is detected, and specifies the location (file and line) of the violation.

Both forms have three additional parameters: an `err` parameter, which is a string describing the specific violation; `help_volume`, the name of the help volume containing the on-line help for this rule; and an optional `rule_name` parameter. `help_volume` can specify a help node using the format *helpvolume_helpnode*. `rule_name` is required only when issuing a violation from a multi-rule, and indicates which rule has fired.

You do not define your own `violation()`, but merely call it from `check()`.

`report()` When implementing a multi-rule, some of the sub-rules in your rule may not be active. (Some sub-rules may belong to inactive rule groups.) You should call `report()`, passing it the name of a sub-rule, to determine if you should execute the code that tests for that sub-rule. You should not call `violation()` if `report()` returns a `FALSE` value.

See Chapter 4 for explanations of the `Symbol`, `TypedSymbol`, and `SymbolTable` classes.

Example Rule

The following code defines a very simple rule that enforces a common coding convention: every class name should be capitalized. (You could use the Class Editor in Static Analyzer to find and fix every occurrence of noncapitalized classes with one simple operation, by selecting the class and choosing "`Edit: Modify ...`".)

This rule uses several data structures and functions from the Static API, which you don't need to understand yet. You can use this example to understand how rules are structured and linked into the rule engine. Since the rule uses the general XPG4 `regexp(5)` expression-matching library, it can easily be extended to implement other stylistic rules. (In fact, the `NameConventions` table rule is an example of such an extended rule.)

Source for this rule can be found in `/opt/softbench/examples/CodeAdvisor/Rules/ruleCapClass.C`.
To test the rule, make the example rule library, as explained in the `Makefile`, and install the new library in `/etc/opt/softbench/lib/rulelibs`.

```
// This is an example of a hypothetical design rule that
// could be implemented in an organization whose coding
// standards require that all Class names begin with a
// capital letter.

#include <Rule/Rule.H>
#include <ctype.h>
#include <stdio.h>
#include <regex.h>

// Define the rule interface

class UserRulesCapClass : public Rule
{
public:
   virtual int kindMask() const;
   virtual Language langMask() const;
   void check(SymbolTable *, const Symbol &);
   virtual const char *errorMess() const;
   virtual const char *name() const;
};

// This rule is invoked for all "Tag" objects.  Tags include all
// compound objects, such as Classes, Templates, Structs, Unions, and Enums.
// Can't specify a kindMask() of KIND_CLASS, since check() is invoked
```

```
// only on Symbol objects.  Class is not a Symbol; Tag is.

int UserRulesCapClass::kindMask() const
{ return 1 << KIND_TAG; }

// This rule applies only to C++ code.

Language UserRulesCapClass::langMask() const
{ return LANGUAGE_CPP; }


// Find all non-capitalized class names

void UserRulesCapClass::check(SymbolTable *, const Symbol &sym)
{
   Tag tag;
   Class cl;

   // Don't want to check instances; only the class name must be capitalized.
   // This code is a common idiom to reject instances.
   // The !tag.ClassType(cl) call also rejects enums.

   if (!sym.SymbolToTag(tag) || !tag.ClassType(cl) || IS_INSTANTIATED(cl.Attrib()))
      return;

   // Reject structs and unions, which are also represented as Classes.

   if (WAS_STRUCT(cl.Attrib()) || WAS_UNION(cl.Attrib()))
      return;

   // Pattern specifying that:
   //    First character is capital letter
   //    If second character exists, it is not uppercase
   static char *capitalized_pattern = "^[[:upper:]]($|[^[:upper:]])";
   static regex_t capitalized_compiled_reg;
   static DBboolean initialized = false;

   // only build regular expression once.
   if (!initialized) {
      // Note that if the regular expression is rebuilt for each rule run,
      // then regfree(&capitalized_compiled_reg) should be called once
      // regexec will no longer be called with the expression to avoid
      // a memory leak.
      if (regcomp(&capitalized_compiled_reg, capitalized_pattern,
                  REG_EXTENDED | REG_NOSUB)!=0)
         return;
      initialized = true;
   }
```

```
   if (regexec(&capitalized_compiled_reg, tag.Name(), 0, NULL, 0)!=0) {
      // doesn't match regular capitalized expression
      char buf[1024];
      sprintf(buf, "Class or class template name '%s' not capitalized", tag.Name());
      violation(tag, buf, "UserRules");
   }
}

// Generic one-line description of the rule

const char *UserRulesCapClass::errorMess() const
{
 return("Class name not capitalized.");
}

// Rule name, should match name of C++ Class

const char *UserRulesCapClass::name() const
{
 return("UserRulesCapClass");
}

// Force a call to base class constructor in the main program

static UserRulesCapClass instance;
```

The RuleWithTable Base Class

Table-driven rules are written as a class derived from the `RuleWithTable` base class. `RuleWithTable` inherits most of its interface from `Rule`, and adds components to work with rule tables.

Each `RuleWithTable` rule defines the family of rules included in its rule table, so every `RuleWithTable` rule is effectively a multi-rule. See the `name()` and `names()` function descriptions for specific information on how `RuleWithTable` uses them.

The public interface to `RuleWithTable` is defined as follows:

```
class RuleTableRecord {
public:
        const char * const name;
        const char * const help_location;
        const char * const message; // printf format string
        const char * const * const data; // array of table data fields
        void *client_data; // available to cache anything associated with entry
        RuleTableRecord() : name(NULL), help_location(NULL), message(NULL),
                            data(NULL) {};
        RuleTableRecord(const char * const rule_name,
                        const char * const help,
                        const char * const msg,
                        const char * const * const fields) :
                                        name(rule_name),
                                        help_location(help),
                                        message(msg),
                                        data(fields),
                                        client_data(NULL) {}
        ~RuleTableRecord();
        // only test "key" for equality, so entry found
        unsigned int operator==(const RuleTableRecord &rec) const
          { return strcmp(rec.name,name)==0; }
        unsigned int operator<(const RuleTableRecord &rec) const;
};

class RuleWithTable : public Rule {
public:
        RuleWithTable(unsigned int number_of_fields, // not including name or
                                                     // help location
                      const char *field_separator = " \t",
                      const char record_separator = '\n') :
                num_data_fields(number_of_fields),
                field_sep(field_separator),
                record_sep(record_separator),
                ruletable(NULL), ruletable_loaded(false) {}
```

```
        // User-Defined table-based rules should define check_table_entry
        // instead of check().
        virtual void check_table_entry(const SymbolTable &symtab,
                                       const Symbol &sym,
                                       RuleTableRecord &entry) = 0;
        virtual void check(SymbolTable *symtab, const Symbol &sym);
};
```

You should not access the private members of the `RuleWithTable` class. They are used by the rule engine.

The `RuleWithTable` interface is a combination of `Rule` and `RuleWithTable`. See the `Rule` description (earlier in this chapter) for an explanation of `kindMask()`, `langMask()`, `errorMess()`, and `violation()`. You will use them in table-driven rules just as you use them in normal `Rule`-derived rules.

You must provide your own version of the `check_table_entry()` function, as well as the functions required by `Rule`: `kindMask()`, `langMask()`, `errorMess()`, and `name()`.

`RuleWithTable::check()` invokes `check_table_entry()` for each entry in the rule table. You should not provide a `check()` function for table-driven rules unless you need different behavior than that provided by the default definition of `RuleWithTable::check()`. You should not provide a `names()` function, since `RuleWithTable` overloads the `Rule::names()` definition.

In addition to the inherited `Rule` interface functions, the following member functions are defined for `RuleWithTable`:

`RuleWith Table()` — Class constructor. Accepts arguments to specify the format of the table: number of data fields in the table (not counting the rule name and the help volume, which are always in fields 1 and 2, and the violation error message, which follows the last field), field separator characters (defaults to space and tab), and record (line) separator character (defaults to newline). Your table can use any field and record separator characters, allowing you to use fields with embedded spaces or newlines if necessary. The default format uses one record per line, with space/tab-separated fields. This is usually the best and most easily readable format.

Your rule constructor initializer list should invoke the `RuleWithTable()` constructor, passing in the appropriate arguments for your table format.

`name()`, `names()`

The rule IDs in a table-driven rule are defined in the rule table, not in the `RuleWithTable` definition. Define a `name()` function that returns the name of your rule *family*. This must match the basename of your rule table. The rule engine searches the locations described in "Specifying Scope of Changes" in Chapter 2 to find a rule table with that name.

Do not define `names()`. `RuleWithTable` defines a `names()` member function that returns the names of all rule IDs defined in the rule table.

`check_table_entry()`

The main rule-check function. Like the `check()` function for `Rule()`, this function is called for every symbol in the database that matches the `kindMask()`. Like `check()`, `check_table_entry()` receives two arguments pointing to the database `SymbolTable` and the `Symbol` to be checked.

In addition, `check_table_entry()` receives a `RuleTableRecord` argument. The rule engine reads each entry in your rule table, invokes the rule on the appropriate symbols, and passes the appropriate data to your `check_table_entry()` function in this argument.

For example, the `NoUnderscoreOnExtern` rule defined in the `NameConventions` rule table (see Chapter 2) specifies that the rule applies only to `FUNCTION` and `VARIABLE` symbols. The rule engine scans through all symbols, and invokes the `NameConventions::check_table_entry()` function on all `FUNCTION` and `VARIABLE` symbols with information from the `NoUnderscoreOnExtern` entry from the rule table. `NameConventions::check_table_entry()` then checks the symbol to see if it violates the conditions specified in the `NoUnderscoreOnExtern` table entry.

Note that you do not need to create any code specifically for the `NoUnderscoreOnExtern` rule. All rule ID's in the

`NameConventions` family are handled by the `NameConventions` rule code.

Example Table-Driven Rule

The following code defines `ProhibDefines`, a simple rule that is identical (except for its name) to the `ProhibDefines` rule that is shipped with SoftBench CodeAdvisor. See `/opt/softbench/config/ruletables/$LANG/ProhibDefines` for the table format used by this rule. (Since this rule is named `UserRulesProhibDefines`, it would normally search for a rule table with that same name. However, for demonstration purposes, this rule uses the `ProhibDefines` table.) The source for this rule can be found in `/opt/softbench/examples/CodeAdvisor/Rules/ruleProhibDefines.C`.

```
#include <Rule/RuleWithTable.H>
#include <iostream.h>
#include <stdio.h>
#include <regex.h>

class UserRulesProhibDefines : public RuleWithTable
{
public:
   UserRulesProhibDefines() : RuleWithTable(1) {} // 1 data field,
                                                  // default separators
   virtual int kindMask() const;
   virtual Language langMask() const;
   void check_table_entry(const SymbolTable &symtab, const Symbol &sym,
                          RuleTableRecord &entry);
   virtual const char *errorMess() const;
   virtual const char *name() const;
};

int UserRulesProhibDefines::kindMask() const
{ return (1<<KIND_SOURCEFILE | 1<<KIND_MACRO); }

Language UserRulesProhibDefines::langMask() const
{ return LANGUAGE_C | LANGUAGE_CPP; }

struct UserRulesProhibDefinesClientDataRecord {
   regex_t compiled_regex;
};


// Table Driven Rule:  Detect prohibited defines in #define and -D
void UserRulesProhibDefines::check_table_entry(const SymbolTable &,
                                               const Symbol &sym,
                                               RuleTableRecord &entry)
{
   char buf[1024];
```

```
File file;
UserRulesProhibDefinesClientDataRecord *cache;

if (entry.client_data)
   cache = (UserRulesProhibDefinesClientDataRecord *) entry.client_data;
else {
   cache = new UserRulesProhibDefinesClientDataRecord;
   entry.client_data = cache;
   const char *regexp = "([[:space:]]|^)-D";
   char *pattern = new char[strlen(regexp) + strlen(entry.data[0]) + 2];
   if (pattern)
   {
      sprintf(pattern, "%s%s", regexp, entry.data[0]);
      if (regcomp(&cache->compiled_regex,
                  pattern,
                  REG_EXTENDED | REG_NOSUB)) {
            delete cache;
            delete pattern;
            entry.client_data = NULL;
            cache = NULL;
            return;
      }
      delete pattern;
    }
}

if (sym.SymbolToFile(file)) {
   if (cache && file.CompileOptions())
   {
         if (regexec(&cache->compiled_regex,
                     file.CompileOptions(),
                     0,
                     NULL,
                     0)==0) {
            char buf2[1024];
            sprintf(buf, entry.message, entry.data[0]);
            sprintf(buf2, "File compiled with -D%s : %s", entry.data[0], buf);
            violation(file.Name(), 0, buf2,
                      entry.help_location, entry.name);
         }
   }
}
else { // not a file, must be a macro
   if (strcmp(sym.Name(), entry.data[0])!=0)
      return;

   SourcePosition defn;
   if (sym.DefinitionSite(defn)) {
      sprintf(buf, entry.message, entry.data[0]);
```

```
         violation(defn.file, defn.position.line, buf,
                   entry.help_location, entry.name);
      }
   }
}

const char *UserRulesProhibDefines::errorMess() const
{
 return("Identifier prohibited for specified reason.");
}


// Rule name -- also used as name of rule table.  By default, this rule
// uses the ProhibDefines rule shipped with CodeAdvisor.

const char *UserRulesProhibDefines::name() const
{
 return("ProhibDefines");
 // return("UserRulesProhibDefines"); // Use this line to use your own table
}

// forces a call to base class constructor in the main program
static UserRulesProhibDefines instance;
```

4

Understanding the Static Database

Rules use the Static database as their view on the program being checked.

The Static database is represented as a set of persistent objects. That is, the objects are stored in the file system of your computer so they are remembered from one session to another. Each time you build your program and regenerate the Static database, a new set of objects is created in the database for future use.

Database Objects

The objects in the database represent programming constructs such as functions and variables. Each object contains attributes that define the object, and associations with other objects to reflect semantics such as references, scope, and binding. For example, a variable object has an attribute of being either global or local to some scope, and a function object is associated with its parameter list.

The API (application programmer interface) for the database allows you to open the database, examine the contents of the database, and close the database. The database API notifies you of any changes made to the database (by another process rebuilding the database) while you are accessing it.

See the files under `/opt/softbench/examples/DbApi/Query` for some simple example database queries.

Capabilities of the Database

Since the Static database contains attributes and associations for each object, it is best matched to certain kinds of rule algorithms.

For example, the database is an ideal match for a rule that examines the member functions defined in a class. The class object lists its member functions on its association list, and each member function object gives full details on its type and declaration information. You can determine inheritance information on the class, allowing you to see if any member function shadows a function inherited from a parent class.

On the other hand, the database does not contain complete information on the structure of your code. For example, the database might indicate that your code references the variable `Count` in the function `Compute_it`. Using the information in the database, you could determine if `Count` is modified or merely used. The database would not, however, indicate exactly what kind of statement the reference was in; you could not tell from the database whether the variable was referenced in an `if` test, or as a parameter in a function call.

If you consider the information provided by SoftBench Static Analyzer (function and variable references, pointers to locations in the code, and so on), you will have a good idea of the information available to you in the Static database.

If your rule requires more understanding of the program than the database can provide, you may be able to get it by directly examining the source of your program. The Static database indicates on what line of what file the variable reference occurs; from this you can open the source file and examine the code as carefully as necessary for your rule.

Even in cases that aren't a perfect match for the database's capabilities, the information in the database is a tremendous aid in finding the information you need.

Learning the Database API

You access the Static database through an Application Programmer Interface (API). The API gives you an object-oriented view onto the contents of the database, through which you can access information on your program files.

Database Objects

The database is implemented as a collection of objects. The interface to the database consists of functions to open the database and examine those objects.

In order to understand the functions in the API, you must first understand the objects manipulated by the API.

This is a brief overview of the important objects in the database. Each object type has attributes to describe its name, its type, the other objects associated with it, and other important information. See "Object Interfaces" for detailed definitions.

`SymbolTable`	Contains object references that lead to all other objects in the database, much like the root directory of a file system "contains" all files below it. All navigation through the database begins at the Symbol Table. There is only one Symbol Table in the database.
`PerBase`	Represents the concept of persistent database object. All objects inherit from `PerBase`.
`Symbol`	Contains information required by named objects. Examples of named objects include files, `#define` macros, and built-in types (`Scalar`). All named objects inherit from `Symbol`. Note: aggregate objects (classes, class templates, structs, and enums) do *not* contain name information, and do not inherit from `Symbol`. Instead, the `Tag` object inherits name information from `Symbol`, and points to the `Class`, `ClassTemplate`, `Struct`, or `Enum`.
`TypedSymbol`	Contains information required by typed objects. Examples of typed objects include variables,

	parameters, and functions. All typed objects inherit from `TypedSymbol`.
`Block`	Represents blocks within functions.
`Class`	Represents C++ classes, and `struct`s and `union`s in C++ code.
`ClassTemplate`, `FunctionTemplate`	Represent class templates and function templates (both global template functions and member function templates).
`Enum`, `EnumMember`	Represent enumerations and enumeration constants.
`RefList`	Contains all references to a specific named object in the database. Each `RefList` contains references to an object in a specific file. For example, each variable object has `RefList`s associated with it to describe every reference (definition, declaration, use, modification) to the variable, one `RefList` per file.
`Struct`	Represents `struct`s and `union`s in C code. Note that `struct`s that are included in both C and C++ code appear as `Class` objects.
`File`	Contains all objects defined in a specific source file. Also contains attributes that indicate "includes" and "included by" relationships.
`Label`	Represents `switch` and `goto` labels.
`Macro`	Represents a `#define` macro.
`Scalar`	Represents built-in types, such as `int` and `char`.
`Tag`	Represents aggregate types: enum, struct, class, and class templates. Each aggregate types is represented by a `Tag` (to hold the `Symbol` information) *and* an `Enum`, `Struct`, `Class`, or `ClassTemplate` object.
`DataMember`, `FunctionMember`	Represent C++ class data members and member functions.
`Function`, `Parameter`	Represent program functions and their arguments.

`TemplateArgument` Represents class template and function template arguments.

\`Typedef`\ Represents named user-defined types.

\`Variable`\ Represents program variables.

See Figure 4-1 for a graphical representation of the database objects. Notice that `Class`, `Enum`, and `Struct` *do not* inherit from `Symbol`. The `Tag` object inherits from `Symbol`, holds the name information, and refers to the aggregate type.

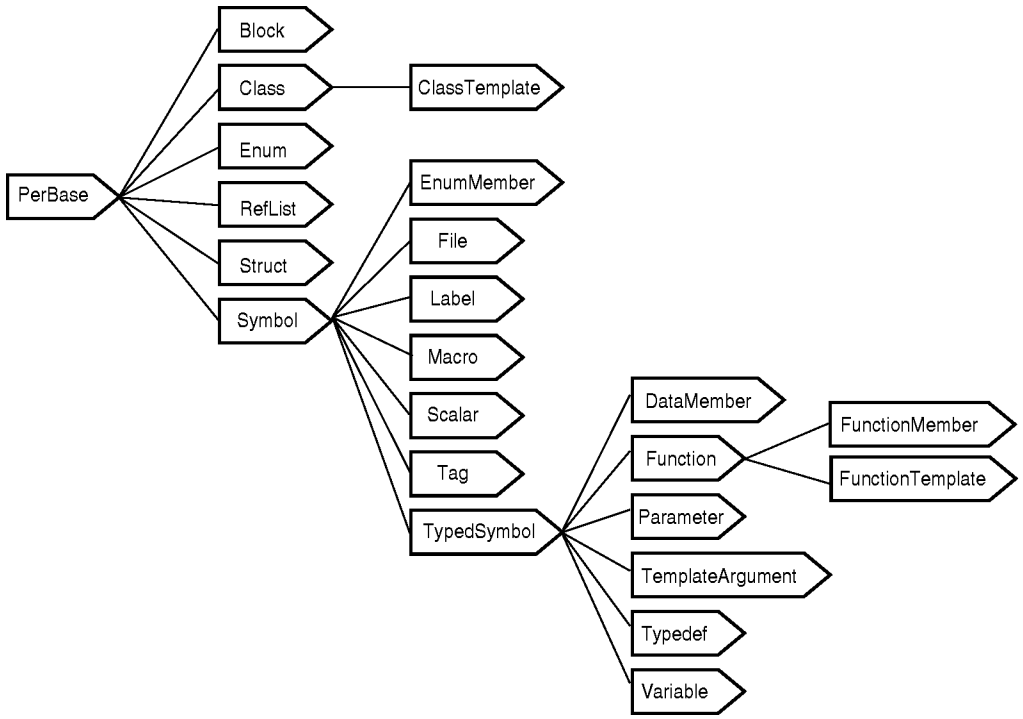

Figure 4-1. Object Hierarchy

Incomplete Objects

Some object types can be "complete" or "incomplete." An incomplete object is one for which complete information is not available; in particular, no definition is available for the symbol. This is most often encountered with externally defined objects. For example, a program might include the declaration "`class Myclass;`", but no definition of the class. The database knows `Myclass` is a class, but knows no more about it. `Myclass` will be incomplete in the database.

Incomplete objects behave differently than complete objects. For example, `Function::Parameters()` returns `FALSE` for an incomplete function. (See each object description for details.) In general, many methods return `FALSE` for incomplete objects. You must test the return value of appropriate methods to ensure an object is complete before using the values returned by the object's methods. Use `Symbol::DefinitionSite()` to test for completeness. If no definition is available, the object is incomplete.

Note that aggregate objects, such as `Class` and `Enum`, do not inherit from `Symbol`. You must retrieve the `Tag` associated with the aggregate to test `DefinitionSite()`.

Database Types

A number of utility types are used by the database to describe objects. The most important types include:

`PerHandle`	A "handle" referring to a persistent object in the database. Handles are managed by the `PerBase` base class.
`PerKind`	The type of a persistent object, as described in the previous section (`KIND_SOURCEFILE`, `KIND_VARIABLE`).
`Attribute`	Attributes of an object, such as `ATTR_GLOBAL`, `ATTR_STATIC`, `ATTR_PRIVATE`, or `ATTR_VIRTUAL`. Inline functions, such as `IS_GLOBAL()` and `IS_STATIC()`, are defined to simplify testing attribute values.
`ScalarType`	The basic type of a variable, such as `SCALAR_CHAR` or `SCALAR_FLOAT`.

`Language`	The language (such as `LANGUAGE_C` or `LANGUAGE_CPP`) associated with a file or symbol.
`Usage`	The type of reference to a symbol, such as `REF_DEFINITION`, `REF_MODIFICATION`, or `REF_CALL`.
`Reference`	A reference to a symbol, including the `Usage` type and the line and column where the reference occurs.
`SourcePosition`	A `Reference` within a specific file.

These types are defined in the header file `DB_Common.h`. See Appendix A for a complete listing.

Type Qualifiers

Typed objects, such as `Variable`s and `Parameter`s, inherit "type" and "type qualifier" information from the `TypedSymbol` base class. The `Type()` member returns a `Symbol` object referencing the derived object that describes the type. The `TypeQualifier()` member returns all modifying information on the base type, such as `*`, `&`, `[]`, or `const`.

For example, a `Variable` defined by "`int count`" has a `Type()` of "`int`". That is, the `Symbol` returned by `Type()` represents a `Scalar` object with a `ScalarType()` of `SCALAR_INT`. The `TypeQualifier()` string is null.

A `Variable` defined by "`class C &Cref const`" has a `Type()` that returns the `Tag` for "`class C`". The `TypeQualifier()` member returns "`& const`".

The possible values that may be found in a `TypeQualifier()` string are:

`*`	Qualifies the type as an indirection (pointer).
`[`*number*`]`	Qualifies the type as an array of dimension *number*.
`&`	Qualifies the type as a reference.
`const`	Qualifies the type as a constant.
`volatile`	Qualifies the type as volatile.

Use the type-safe conversion routines `Symbol::SymbolTo`*type*`()` to test the value returned by `Type()` and to convert it to the appropriate type for use.

Accessing the Database

The basic interface to the database is quite simple. You open the database, specifying what language(s) you are interested in, and the open call returns the database's global symbol table.

You then bracket each request to the database in a "transaction," so that no other process can change the database while you are reading it.

Remember to close the database when you are finished.

Note: rule writers do not need to open or close the database or manage transactions. The rule engine handles all transaction processing.

Opening and Closing the Database

The `SymbolTable` class (described later) defines two `friend` functions, `OpenDatabase` and `CloseDatabase`. As their names imply, these functions are used to open and close a specified database. The database manages the accesses to it, and can safely handle a writer (such as a compiler updating the database) at the same time a reader (such as your rule) has the database open.

```
DBboolean OpenDatabase(const char *filename,
                       SymbolTable &globalsymboltable,
                       Language language,
                       DBModifiedCallback callback);

void CloseDatabase(SymbolTable &globalsymboltable);
```

Given a `filename`, `OpenDatabase` opens the database in that file and returns a pointer to the `SymbolTable` in the database. `Language` is used to specify the language(s) you are interested in; normally you will pass in `LANGUAGE_CPP` or `LANGUAGE_CPP | LANGUAGE_C`. Use `LANGUAGE_UNKNOWN` if you are interested in all languages. The function `callback` is called if the database is modified by a writer while you have it open. You can use this to indicate that the database is now out of date, just like SoftBench Static Analyzer does. The `callback` function must be defined as "`void callback(void) {` *body* `}`".

It is possible to open and manipulate multiple databases at once. This is useful if there are multiple databases representing your program. For example, if you compiled a library separately from the main program, in another directory, the library would have its own Static database.

`CloseDatabase` simply closes the database and clears the `globalsymboltable` pointer.

Delimiting Transactions

The database can be accessed by a writer while readers have the database open, but not while readers are actively accessing the database. You must enclose each request to the database in a "transaction." This prevents a writer from changing the database in the middle of your access.

Call the method *GST*`.StartTransaction()` immediately before your database access, and call *GST*`.EndTransaction()` as soon as you have finished. (*GST* is the Global Symbol Table object returned by your call to `OpenDatabase`.)

Iterators

Since an object may have an arbitrary number of items associated with it (for example, a variable may be accessed in arbitrarily many locations), the database provides a mechanism to successively select and operate on each item in a list.

The Iterators mechanism manages the iteration through a collection of items. Using Iterators, it is easy to iterate through all objects in a list, without needing to understand the underlying iteration mechanism.

Iterators are accessed by calling an "iterator definition function" defined in certain database objects. As an example, the `SymbolTable` class defines "`ITERATOR(File) Files() const;`". This function returns a C++ object of type `ITERATOR(File)`. You use this iterator by saving the value returned by `Files()` and looping using the macros `ITERATE_BEGIN` and `ITERATE_END`. Within the body of the iteration loop, the current value of the iterator variable points to the current object to be operated upon.

```
ITERATOR(File) filei;      // Declare the iterator pointer
filei = GST.Files();       // Get the File iterator
ITERATE_BEGIN(filei)       // Iterate on all Files
{
  // Access information in each file, using filei as File pointer.
  // For example, to list all source files found in the database:
  cout << "File name is " << filei.Name() << endl;
}
ITERATE_END(filei);
```

See Appendix B for a more complete explanation of iterators.

Attribute Iterators

The Static API also defines a subset of iterators, called Attribute Iterators, that define a set of attributes along with each object in the iteration list. Attributes, as defined in the Static database, specify characteristics of a symbol such as Global, Static, Public, Private, and Virtual. See "Database Types". Attribute Iterators are identical to normal iterators, with the addition of two member functions (`GetIteratorAttribute()` and `SetIteratorAttribute()`) to access the attributes.

Attribute Iterators are primarily used for class inheritance links. Since a base class may be referenced by many derived classes, and each derived class may define a different inheritance characteristic (public, private, virtual), the inheritance information cannot be stored in either base or inherited class. The Attribute Iterator associates the inheritance information with each inheritance link.

See Appendix B for more information.

Object Interfaces

The class interfaces for the database objects define the bulk of the Static API. Each object defines the methods (functions) that are used to access the object. In addition, many objects also inherit from other, more generic objects (usually `Symbol`), which in turn define additional function interfaces.

The following sections describe the class definition interface to each object type.

Note that many object methods accept complex objects as parameters and, using C++ references, modify the parameters. These routines return the `DBBoolean` value `FALSE` if any error occurs.

Each object defines constructors and destructors. You should not use these functions, since objects are created and destroyed automatically as needed by your code. You are only interested in the additional `public` methods defined by each object type.

These object classes are defined in the header file `DB_Read.h`.

The descriptions in this chapter are arranged alphabetically for ease of reference. In addition to the actual objects you will encounter in the database, this chapter also includes descriptions for the `SymbolTable` class and the three object "base" classes: the `PerBase` class, which all database objects inherit from, the `Symbol` class, which all named objects inherit from, and the `TypedSymbol` class, which all typed `Symbol` objects inherit from.

The `SymbolTable` object is returned when you open a database, and is the source of navigation for all further database operations. You do not normally need to use the `SymbolTable` object when writing rules, since the rule engine handles the database manipulations for you. The exception occurs when you specify a `kindMask()` of `0`. In that case you must handle all iteration through the `SymbolTable`.

Block Object

`Block` represents the entire code block within a function.

`Block` inherits from `PerBase`, and has no type or name properties.

```
class Block : public PerBase {
  public:
    Block();
    ~Block();

    unsigned int BeginLine() const;
    unsigned int EndLine() const;
    Attributes Attrib() const;

    File BlockFile() const;

    ITERATOR(Variable) BlockVariables() const;
    ITERATOR(Tag) BlockTags() const;
    ITERATOR(Typedef) BlockTypedefs() const;
    ITERATOR(Function) BlockFunctions() const;
    ITERATOR(Label) BlockLabels() const;
};
```

Method Definitions

`BeginLine()`, `EndLine()`, `BlockFile()`	Indicate the block's location.
`BlockVariables()`, `BlockTags()`, `BlockTypedefs()`, `BlockFunctions()`, `BlockLabels()`	Return iterators over the variables, tags (classes, structs, enums, and templates), typedefs, functions, and labels defined in the block.

Class Object

`Class` objects represent C++ classes. Structs and unions in C++ code are also represented by `Class`, since C++ makes little distinction between classes and the other aggregate types. (You can determine if the `Class` was declared as a struct or union by testing the `Attrib()` value using the attribute-testing functions `WAS_STRUCT()` and `WAS_UNION()`.) Structs parsed *only* by a C compiler are represented as `Structs`.

Each `Class` has a corresponding `Tag`.

`Class` objects contain `DataMember` and `FunctionMember` objects to represent the data fields and methods defined by the class. `Class` objects also contain iterators to list the parent (base) classes and children (derived) classes of the class.

`Class` objects contain only the members that are defined *by that class*. You must seek back through parent classes to find all members inherited from base classes. See below for an example.

For an incomplete struct, only `ClassTag()` and `Attrib()` return meaningful results. All other methods return `FALSE` or null values.

Note that class *instances* are represented as incomplete `Classes`. In this specific case, when the attribute `IS_INSTANTIATED` is set, `ExpandedFrom()` returns the template from which the class is instantiated. To derive information about instances, you must access the instantiating template.

`Class` inherits from `PerBase`, and has no type or name properties. The corresponding `Tag` contains the name information.

```
class Class : public PerBase {
  public:
    Class();
    ~Class();

    DBboolean ClassTag(Tag &tag) const;
    Attributes Attrib() const;

    int MemberCount() const;
    DBboolean FindDataMember(const char *name, DataMember &member) const;
    DBboolean FindFunctionMember(const char *name, FunctionMember &member) const;
    ITERATOR(DataMember) DataMembers() const;
    ITERATOR(FunctionMember) FunctionMembers() const;
    ITERATOR(Function) AllFunctions() const;
```

```
    ATTRIBUTE_ITERATOR(Tag) BaseClasses() const;
    ATTRIBUTE_ITERATOR(Tag) DerivedClasses() const;

    ITERATOR(Tag) NestedClasses() const;
    ITERATOR(Tag) NestedEnums() const;
    ITERATOR(Typedef) NestedTypedefs() const;

    ITERATOR(Symbol) Friends() const;
    // Class Template this class is an instance of.
    DBboolean ExpandedFrom(Tag &tag) const;

    friend class Tag;
};
```

Method Definitions

`ClassTag()`	Returns the `Tag` object associated with the `Class`.
`Attrib()`	Returns the attributes (such as `ATTR_GLOBAL`) of the class.
`MemberCount()`	Returns a count of the data and function members in the class.
`FindDataMember()`, `FindFunctionMember()`	Returns the `DataMember` or `FunctionMember` in this `Class` with the specified `name`.
`DataMembers()`, `FunctionMembers()`	Return iterators over all data members or function members in the class.
`AllFunctions()`	Like `FunctionMembers()`, but also returns `FunctionTemplates` in a `ClassTemplate`. `AllFunctions()` should be used if you want your rule to apply to both classes and class templates.
`BaseClasses()`, `DerivedClasses()`	Return *attribute* iterators over the immediate parent or immediate derived classes of this class. (Notice that these iterators return `Tag`s, not `Class`es.) For example, suppose class `Z` inherits from classes `Y1` and `Y2`, and `Y1` inherits from `X`. The `BaseClasses()` iterator on class `Z` returns *only* `Y1` and `Y2`. To find the deeper ancestors of `Z`, you must use the `BaseClasses()` iterators on `Y1` and `Y2`. Similarly, `DerivedClasses()` on `X` returns *only* `Y1` and any

	other classes that inherit directly from `X`. See below for an example. Note that `BaseClasses()` is guaranteed to return all base classes of a class, but `DerivedClasses()` cannot be guaranteed to return all derived classes. It is possible that code not included in the database derives from this class. `GetIteratorAttribute()` returns the attributes of the inheritance relationship: virtual, public, private, or protected.
`NestedClasses()`, `NestedEnums()`, `NestedTypedefs()`	Return iterators over the classes and enums nested within the class.
`Friends()`	Returns an iterator over the `friend` functions of the class.
`ExpandedFrom()`	If the class is an instance of a class template, `ExpandedFrom()` returns the tag of the class template of which it is an instance.

Example

This function prints all function members in the class referred to by a specified `Class`, including all inherited function members.

```
void function_members(Class cls) {

  Tag tag;
  cls.ClassTag(tag);
  printf("Function Members defined in class %s:\n", tag.Name());
  ITERATOR(FunctionMember) fmi = cls.FunctionMembers();
  ITERATE_BEGIN(fmi)
  {
     printf("  %s:\n", fmi.Name());
  }
  ITERATE_END(fmi)

  // Iterate over immediate parent classes of this class
  // and recursively print their function members

  ATTRIBUTE_ITERATOR(Tag) bci = cls.BaseClasses();
  ITERATE_BEGIN(bci)
  {
    Class cls2;
    bci.ClassType(cls2);        // BaseClasses returns Tags;
    function_members(cls2);     //  convert to Class & recurse
  }
  ITERATE_END(bci)
}
```

ClassTemplate Object

`ClassTemplate` objects represent C++ parametric classes. Each `ClassTemplate` has a corresponding `Tag`.

Like `Class` objects, `ClassTemplate` objects contain the data members and member functions defined by that class template.

For an incomplete object, only `ClassTag()` and `Attrib()` return meaningful results. All other methods return `FALSE` or null values.

`ClassTemplate` inherits from `PerBase`, and has no type or name properties. The corresponding `Tag` contains the name information.

```
class ClassTemplate : public Class {
  public:
    ClassTemplate();
    ~ClassTemplate();

    int ArgumentCount() const;
    ITERATOR(TemplateArgument) TemplateArguments() const;

    DBboolean FindFunctionTemplate(const char *name,
       FunctionTemplate &ftemplate) const;
    ITERATOR(FunctionTemplate) FunctionTemplateMembers() const;

    ITERATOR(Tag) Instantiations() const;
};
```

Method Definitions

`ArgumentCount()`, `TemplateArguments()`	Return a count of, and an iterator over, the template's arguments.
`FindFunction-Template()`	Returns the function template member with the specified `name`.
`FunctionTemplate-Members()`	Returns an iterator over all function templates in the template.
`Instantiations()`	Returns an iterator over all classes instantiated from this template.

DataMember Object

`DataMember` objects represent the data members of structures, classes, and class templates.

`DataMember` inherits type and name information from `TypedSymbol`.

```
class DataMember : public TypedSymbol {
  public:
    DataMember();
    ~DataMember();

    DBboolean MemberOf(Struct &parentstruct) const;
    DBboolean MemberOf(Class &parentclass) const;
};
```

Method Definitions

`MemberOf()` — The overloaded functions `MemberOf` return the aggregate structure (struct, class, or class template) of which this object is a member. If you do not know what type of object contains the `DataMember`, you can call each of the overloaded `MemberOf` functions until one returns `TRUE`.

Enum Object

`Enum` objects represent enumerated types. Each `Enum` has a corresponding `Tag`. `Enums` objects contain `EnumMember` objects representing each value defined by the enum.

For an incomplete enum, only `EnumTag()` and `Attributes()` return meaningful results. All other methods return `FALSE` or null values.

`Enum` inherits from `PerBase`, and has no type or name properties. The corresponding `Tag` contains the name information.

```
class Enum : public PerBase {
  public:
    Enum();
    ~Enum();

    DBboolean EnumTag(Tag &tag) const;
    Attributes Attrib() const;

    int MemberCount() const;
    DBboolean FindEnumMember(const char *name, EnumMember &member) const;
    ITERATOR(EnumMember) EnumMembers() const;

    friend class Tag;
};
```

Method Definitions

`EnumTag()`	Returns the `Tag` object associated with the `Enum`.
`Attrib()`	Returns the attributes (such as `ATTR_GLOBAL`) of the enum.
`MemberCount()`	Returns the number of members (constants) in the enum.
`FindEnumMember()`	Returns the `EnumMember` with the specified `name`.
`EnumMembers()`	Returns an iterator over all members in the enum.

EnumMember Object

`EnumMember` objects represent the constant values of an `Enum`.

`EnumMember` inherits name information from `Symbol`.

```
class EnumMember : public Symbol {
  public:
    EnumMember();
    ~EnumMember();

    Enum MemberOf() const;
    int Value() const;
};
```

Method Definitions

`MemberOf()`	Returns the enum of which this object is a member.
`Value()`	Returns the ordinal (numeric) value of this member.

File Object

`File` objects contain all the `Symbol`s and `RefList`s defined within a file.

`File` inherits name information from `Symbol`.

```
class File : public Symbol {
  public:
    File();
    ~File();

    Language FileType() const;
    const char *CompileName() const;
    const char *CompileOptions() const;
    const char *CompileHost() const;
    const char *CompileDir() const;
    time_t  ModifiedTime() const;

    ITERATOR(RefList) RefLists() const;

    ITERATOR(File) Includes() const;
    ITERATOR(File) IncludedBy() const;

    ITERATOR(Module) Modules() const;
    ITERATOR(Macro) Macros() const;
    ITERATOR(Variable) Variables() const;
    ITERATOR(Function) Functions() const;
    ITERATOR(Tag) Tags() const;
    ITERATOR(Typedef) Typedefs() const;
    ITERATOR(FunctionTemplate) FunctionTemplates() const;

    DBboolean EnclosingFunction(Symbol &symbol, int line) const;
};
```

Method Definitions

`FileType()`	Returns the `Language` type of the file.
`CompileName()`, `CompileOptions()`, `CompileHost()`, `CompileDir()`	If the `File` is a source file, these function return the name of the compiler used to compile the file, the compile options used to compile the file, and the host (system) and working directory on which the file was compiled.
`ModifiedTime()`	Returns time of last file modification.
`RefLists()`	Returns an iterator over all `RefList`s contained in the file.

`Includes()`, `IncludedBy()`	Return iterators over all files that this file includes, and all files that include this file.
`Modules()`, `Macros()`, `Variables()`, `Functions()`, `Tags()`, `Typedefs()`, `FunctionTemplates()`	Return iterators for all types of symbols defined within the file.
`EnclosingFunction()`	Returns the function that encloses the line `line` in the file. Notice that `EnclosingFunction` returns a `Symbol`, not a `Function`. The enclosing function may be a `FunctionMember` or a `FunctionTemplate`.

Function Object

`Function` represents complete and incomplete functions. An "incomplete" function is a function that is known only by its signature. It may be defined by an extern reference, or by a forward reference that is never completed.

Many incomplete function references are created by `#include` files, since they declare a function without defining it.

For incomplete functions, only the base `Symbol` methods are valid. All other methods return FALSE and/or null results.

`Function` inherits type and name information from `TypedSymbol`.

```
class Function : public TypedSymbol {
  public:
    Function();
    ~Function();

    DBboolean ParameterCount(int &count) const;
    ITERATOR(Parameter) Parameters() const;
    DBboolean ParameterTypeInfo(int N, Symbol &type, char *&qual) const;

    DBboolean DefinitionSite(SourcePosition &position) const;

    DBboolean FunctionBlock(Block &funblock) const;
    DBboolean MemberFunction(FunctionMember &funmem) const;
    DBboolean ExpandedFrom(FunctionTemplate &funtempl) const;
};
```

Method Definitions

`ParameterCount()`, `Parameters()` — Return the count of parameters, and an iterator over all parameters.

`ParameterTypeInfo()` — Return the type symbol and qualifier string (such as "`*`" or "`&`") of the Nth parameter. See "Type Qualifiers" earlier in this chapter for a more complete explanation.

`ParameterTypeInfo()` accepts a "`char *`" argument for the type qualifier string. On return the "`char *`" contains a pointer to the qualifier information. The memory allocated for the qualifier string is managed by the database. You should not release it.

	`ParameterTypeInfo()` works on incomplete functions that have full function signatures. Note that K&R C code has no signatures, and thus `ParameterTypeInfo()` does not work on this code.
`DefinitionSite()`	This function shadows the `DefinitionSite()` method in `Symbol`. It is specialized to handle multiple functions of the same name, such as if your database includes multiple `main()` functions.
`FunctionBlock()`	Returns the block containing the function's code.
`MemberFunction()`	Converts a `Function` object to a `FunctionMember` object.
`ExpandedFrom()`	If the function is a function instance, `ExpandedFrom()` returns the function template from which it was derived.

FunctionMember Object

`FunctionMember` objects represent function members of C++ classes. `FunctionMember` inherits from the `Function` class.

`FunctionMember` inherits type and name information from `TypedSymbol`.

```
class FunctionMember : public Function {
  public:
    FunctionMember();
    ~FunctionMember();

    Class MemberOf() const;
};
```

Method Definitions

`MemberOf()`	Returns the class of which this function is a member.

FunctionTemplate Object

`FunctionTemplate` objects represent C++ parametric functions. `FunctionTemplate` inherits from the `Function` class.

```
class FunctionTemplate : public Function {
  public:
    FunctionTemplate();
    ~FunctionTemplate();

    int ArgumentCount() const;
    ITERATOR(TemplateArgument) TemplateArguments() const;
    DBboolean MemberOf(ClassTemplate &parenttemplate) const;
    ITERATOR(Function) FunctionInstantiations() const;
    ITERATOR(FunctionMember) FunctionMemberInstantiations() const;
};
```

Method Definitions

`ArgumentCount()`, `TemplateArguments()`	Return a count of, and an iterator over, the function template's arguments.
`MemberOf()`	If the `FunctionTemplate` is a member of a `ClassTemplate`, returns the class template of which this function template is a member.
`Function-Instantiations()`	If the `FunctionTemplate` is a pure function template, `FunctionInstantiations()` returns a `Function` iterator over all instances of the template.
`FunctionMember-Instantiations()`	If the `FunctionTemplate` is a member function of a class template, `FunctionMemberInstantiations()` returns a `FunctionMember` iterator over the member function in all instances of the class template.

Label Object

`Label` represents the target of `switch` or `goto` commands. The `RefLists()` defined for a `Label` refer to the statements that branch to the `Label`.

`Label` inherits name information from `Symbol`.

```
class Label : public Symbol {
  public:
    Label();
    ~Label();

    // Label container; Block, Module or File.
    DBboolean Scope(Block &block) const;
    DBboolean Scope(Module &module) const;
    DBboolean Scope(File &file) const;
};
```

Method Definitions

`Scope()` — These overloaded functions return the `Block`, `Module`, or `File` that contains the `Label`.

Macro Object

The `Macro` object represents C preprocessor macros (`#define`). It is *not* used for C++ inline functions.

`Macro` inherits name information from `Symbol`.

```
class Macro : public Symbol {
  public:
    Macro();
    ~Macro();
};
```

`Macro` defines no interface methods of its own. All `Symbol` methods are available; in particular, `EnclosingFile()` and `EnclosingBlock()` can be used to find the definition scope for global and local macros, respectively.

Parameter Object

`Parameter` represents function parameters.

`Parameter` inherits type and name information from `TypedSymbol`.

```
class Parameter : public TypedSymbol {
  public:
    Parameter();
    ~Parameter();
};
```

Method Definitions

`Parameter` defines no interface methods of its own. All `TypedSymbol` methods are available.

`Parameter` objects are empty for incomplete functions. Use `ParameterTypeInfo()` for information on incomplete functions.

The PerBase Base Class

`PerBase` is the foundation class that defines the concepts of persistent database objects. In particular, `PerBase` defines object "handles" and conversion to higher-level objects. All database objects inherit directly or indirectly from `PerBase`. All `PerBase` methods are available to all database objects.

You will not encounter `PerBase` objects in the database. It is used only as a parent class for constructing objects.

```
class PerBase {
  public:
    PerBase();
    ~PerBase();

    // Handle, null test, and kind of this object.
    PerHandle Handle() const;
    DBboolean IsHandleNull() const;
    PerKind Kind() const;

    // Type-safe conversions to derived classes.
    DBboolean BaseToSymbol(Symbol &symbol) const;
    DBboolean BaseToBlock(Block &block) const;
    DBboolean BaseToEnum(Enum &enumeration) const;
    DBboolean BaseToStruct(Struct &structure) const;
    DBboolean BaseToClass(Class &cppclass) const;
    DBboolean BaseToRefList(RefList &reflist) const;
};
```

`Handle()`, `IsHandleNull()` — `Handle()` returns the "handle" of the object. The handle is the object's identifier in the database. `IsHandleNull()` tests to see if the symbol's `Handle` is null. Certain access methods, such as `Find` in the `SymbolTable` object, can return a null object handle. Normally, however, you will not encounter null handles.

`Kind()` — Returns an enumerated type that tells what kind of object (such as `KIND_VARIABLE` or `KIND_FUNCTION`) the `PerBase` represents. See "Database Types".

`BaseTo`*type*`()` — These functions provide a type-safe conversion from a base object to the appropriate higher-level object class. Use `Kind()` to determine which converter to use.

RefList Object

`RefList` represents an array of references. Each `RefList` lists all references to a symbol within one file. The `Symbol` object contains an iterator of `Reflists`, one for each file containing a reference to the `Symbol`.

`RefList` inherits from `PerBase`, and has no type or name properties.

```
class RefList : public PerBase {
  public:
    RefList();
    ~RefList();

    Symbol SymbolFor() const;
    File FileIn() const;

    int ReferenceCount() const;
    Reference operator[] (int index) const;
};
```

Method Definitions

`SymbolFor()`	Returns the `Symbol` referenced in the `RefList`.
`FileIn()`	Returns the `File` in which the `RefList` references occur.
`ReferenceCount()`	Returns the number of references in the array.
`operator[]`	The `[]` operator is overloaded to give access to the array of references. The `RefLists()` iterator defined by `Symbol` inherits from the `RefList` object, and hence inherits the `[]` operator. See below for an example.

Notice that there are `RefLists()` iterators defined on `Symbol` *and* `File` objects. The two-dimensional organization of `RefLists` (below) allows you to access references by symbol (stepping through the accesses in each file) or by file (stepping through accesses to all the symbols defined in that file).

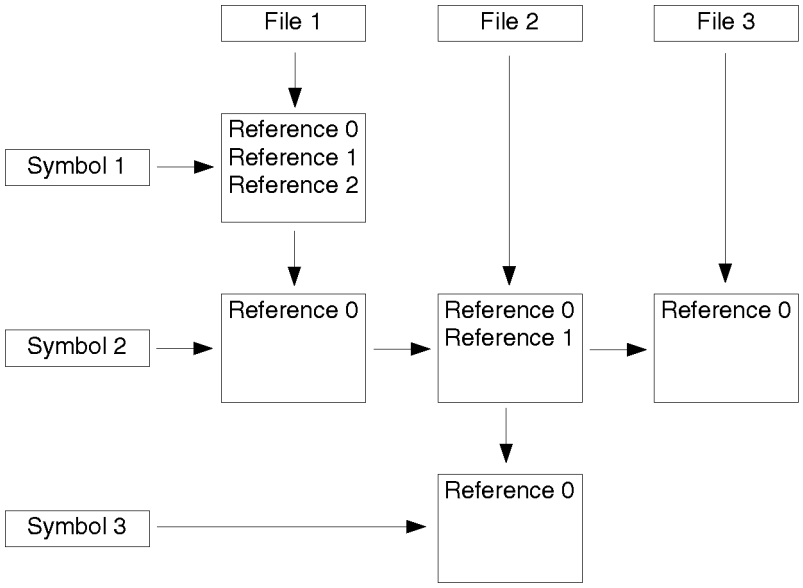

Figure 4-2. RefList Organization

In this illustration, the boxes containing "References" are `RefLists`. In this example, `Symbol1` is a local symbol referenced only in `File1`. `Symbol2` is global, and is referenced in all three files. `File1` contains references to `Symbol1` and `Symbol2`.

The `RefLists()` iterator on `Symbol1` returns one element: a `RefList` containing three references in `File1`. The `RefLists()` iterator on `File1` returns two elements: a `RefList` containing three references (the same `RefList` that was returned for `Symbol1`, since the references are references for `Symbol1`) and a `RefList` containing one reference to `Symbol2`.

See below for example code that illustrates the two-dimensional organization.

Example

These code fragments illustrate the use of `RefLists`. Notice the use of the overloaded `[]` operator.

This code is equivalent to choosing a "Symbol" in Figure 4-2 and following the arrows to the right:

```
// Print location of all references for the variable "var".

ITERATOR(RefList) rli = var.RefLists();
ITERATE_BEGIN(rli)
{
   printf("References in file %s:\n", rli.FileIn().Name());
   int i;
   for (i=0; i<rli.ReferenceCount(); i++) {
     printf("Line = %d, column = %d\n",
        rli[i].line, rli[i].column);
     }
}
ITERATE_END(rli)
```

This code is equivalent to choosing a "File" in Figure 4-2 and following the arrows downward:

```
// Print location of references to all symbols in the file "file".

ITERATOR(RefList) rli = file.RefLists();
ITERATE_BEGIN(rli)
{
   printf("References to symbol %s:\n", rli.SymbolFor().Name());
   int i;
   for (i=0; i<rli.ReferenceCount(); i++) {
     printf("Line = %d, column = %d\n",
        rli[i].line, rli[i].column);
     }
}
ITERATE_END(rli)
```

Scalar Object

`Scalar` objects represent built-in intrinsic types, such as `int` or `char`.

Notice that `Scalar` *does not* inherit from `TypedSymbol`, since a type has no `TypeQualifier` information. Instead, `Scalar` inherits name information from `Symbol`, and provides a `ScalarType` function to describe the type of the scalar.

```
class Scalar : public Symbol {
  public:
    Scalar();
    ~Scalar();

    ScalarType Type() const;
};
```

Method Definitions

`Type()`	Returns the type of the `Scalar`. `ScalarType` is defined in `DB_Common.h`. See Appendix A for a listing.

Struct Object

`Struct` objects represent structures and unions in C code. Structures and unions are represented as `Class` objects in C++ code, since C++ makes no real distinction between structs, unions, and classes.

Note: if a header file is included by both C and C++ files, any structs defined in the header file are promoted to `Class` objects even when they are used in C code.

Each `Struct` has a corresponding `Tag`.

`Struct` objects contain `DataMember` objects to represent the data fields in the struct.

For an incomplete struct, only `StructTag()` and `Attributes()` return meaningful results. All other methods return `FALSE` or null values.

`Struct` inherits from `PerBase`, and has no type or name properties. The corresponding `Tag` contains the name information.

```
class Struct : public PerBase {
  public:
    Struct();
    ~Struct();

    DBboolean StructTag(Tag &tag) const;
    Attributes Attrib() const;

    int MemberCount() const;
    DBboolean FindDataMember(const char *name,
                             DataMember &datamember) const;
    ITERATOR(DataMember) DataMembers() const;

    friend class Tag;
};
```

Method Definitions

`StructTag()`	Returns the `Tag` object associated with the `Struct`.
`Attrib()`	Returns the attributes (such as `ATTR_GLOBAL`) of the struct.
`MemberCount()`	Returns a count of data members in the struct.

`FindDataMember()`	Returns the `DataMember` in this `Struct` with the specified `name`.
`DataMembers()`	Returns an iterator over all data members in the struct.

The Symbol Base Class

`Symbol` is the base class through which all named objects (`Macro`, `Variable`, `Parameter`, `Function`, `File`, `Scalar`, `Tag`, `Typedef`, `EnumMember`, `DataMember`, and `TemplateArgument`) are derived (directly, or indirectly through `TypedSymbol`) from the `Symbol` class.

You will not encounter `Symbol` objects in the database; the class is used only as a parent class for other objects. All properties that make sense for named objects, such as the object's name, definition location, and so on, are defined in `Symbol` and are available to all derived types. Notice that `Symbol` inherits its concept of persistent database objects from `PerBase`.

Typed objects inherit from `TypedSymbol`.

```
class Symbol : public PerBase {
  public:
    Symbol();
    ~Symbol();

    char *Name() const;
    Attributes Attrib() const;

    // Enclosures of the symbol.
    DBboolean EnclosingBlock(Block &eblock) const;
    DBboolean EnclosingClass(Class &eclass) const;
    DBboolean EnclosingFile(File &efile) const;

    DBboolean DefinitionSite(SourcePosition &position) const;
    DBboolean DeclarationSite(SourcePosition &position) const;
    DBboolean ReferenceSite(SourcePosition &position,
            int mask = REF_MODIFICATION | REF_USE | REF_CALL | REF_DEREF |
                       REF_ADDROF | REF_VIRTUALCALL) const;

    ITERATOR(RefList) RefLists() const;

    // Conversion of this symbol to the derived kind.
    DBboolean SymbolToTypedSymbol(TypedSymbol &typedsymbol) const;
    DBboolean SymbolToVariable(Variable &variable) const;
    DBboolean SymbolToParameter(Parameter &parameter) const;
    DBboolean SymbolToFunction(Function &function) const;
    DBboolean SymbolToDataMember(DataMember &datamember) const;
    DBboolean SymbolToEnumMember(EnumMember &enummember) const;
    DBboolean SymbolToFunctionMember(FunctionMember &functionmember) const;
    DBboolean SymbolToScalar(Scalar &scalar) const;
    DBboolean SymbolToTag(Tag &tag) const;
    DBboolean SymbolToTypedef(Typedef &tdef) const;
```

```
    DBboolean SymbolToTemplateArgument(TemplateArgument &templatearg) const;
    DBboolean SymbolToFunctionTemplate(FunctionTemplate &functiontempl) const;
    DBboolean SymbolToModule(Module &module) const;
    DBboolean SymbolToFile(File &file) const;
};
```

Method Definitions

`Name()`	Returns the name of the object.
`Attrib()`	Lists attributes of the symbol, such as `ATTR_GLOBAL` or `ATTR_STATIC`. See "Database Types".
`EnclosingFile()`, `EnclosingBlock()`, `EnclosingClass()`	These functions return the handle of the file and block that contain the definition of the symbol. If the symbol is not enclosed by any block (as is the case with global variables), `EnclosingBlock` returns `FALSE`. `EnclosingClass` returns the parent class of the symbol, or `FALSE` if the symbol is not contained by a class.
`RefLists()`	Returns an iterator to all `RefLists` for the symbol. Each `RefList` contains all references to the symbol in a particular file.
`DefinitionSite()`, `DeclarationSite()`, `ReferenceSite()`	`DefinitionSite()` returns the location of the *definition* of the symbol (such as the code body for a function), if any. If more than one definite site exists, `DefinitionSite()` returns the first one it finds. `DeclarationSite()` and `ReferenceSite()` return the *first* declaration or reference found in the database. This can be useful as a simple test to determine if the symbol is ever declared or referenced. `ReferenceSite()` accepts a mask to specify the types of reference you want. If you need all occurrences of a declaration or reference, you must iterate through the `RefLists()` for the symbol.
`SymbolTo`*type*`()`	These functions convert a `Symbol` to its derived type. For example, if you had a `Symbol` that you determined (by checking its `Kind()`) was actually a `Variable`, you

could use `SymbolToVariable` to create a `Variable` object. For example, "`sym.SymbolToVariable(var)`" converts the `Symbol sym` into the `Variable var`. If the `Symbol` is not actually of (or derived from) type *type*, the function returns `FALSE`.

The SymbolTable Class

The `SymbolTable` class defines the global symbol table for a database. A database contains exactly one `SymbolTable`, which acts as the "root" of the database just as "`/`" acts as the "root" of a filesystem. The `SymbolTable` contains all `Files` and all globally-scoped objects in the database.

```
class SymbolTable {
  public:
    SymbolTable();
    ~SymbolTable();

    PerHandle Handle() const;

    // Time stamp of database and transaction management.
    const char *FileName() const;
    time_t ModifiedTime() const;
    DBboolean StartTransaction() const;
    DBboolean EndTransaction() const;

    DBboolean Contains(Symbol) const;

    ITERATOR(Macro) Macros() const;
    DBboolean Find(const char *name, Macro &macro) const;

    ITERATOR(Variable) GlobalVariables() const;
    DBboolean Find(const char *name, Variable &variable) const;

    ITERATOR(Function) GlobalFunctions() const;
    DBboolean Find(const char *name, Function &function) const;

    ITERATOR(Tag) GlobalTags() const;
    ITERATOR(Tag) LocalTags() const;
    DBboolean Find(const char *name, Tag &tag) const;

    ITERATOR(Typedef) GlobalTypedefs() const;
    DBboolean Find(const char *name, Typedef &tdef) const;

    ITERATOR(Module) GlobalModules() const;
    DBboolean Find(const char *name, Module &module) const;

    ITERATOR(File) Files() const;
    DBboolean Find(const char *name, File &file) const;
    void ActivateFiles(int count, char **filename) const;
    void ActivateFiles(int count, char **filename, Language lang) const;

    ITERATOR(FunctionTemplate) FunctionTemplates() const;
    DBboolean Find(const char *name, FunctionTemplate &funtempl) const;
```

```
    ITERATOR(Symbol) GlobalSymbols() const;
    ATTRIBUTE_ITERATOR(Symbol)
                    GlobalSymbols(const char *name, PerKind kind) const;

    DBboolean Find(const char *name, FunctionMember &funmember) const;
    DBboolean Find(const char *name, DataMember &datamember) const;

    ATTRIBUTE_ITERATOR(Symbol) SymbolsAtLocation(
        const char *name,
        const char *filename,
        long line,
        long column,
        DBboolean ignorecase,
        DBboolean useregexp,
        SymbolsAtLocationScoping& scoping,
        DBboolean allowFuzzyMatch = true) const;

    DBboolean EnclosingFunction(Symbol& fun,
                                const char *fileName,
                                int line);
  private:
    PerHandle SymbolTableHandle;

  friend DBboolean OpenDatabase(const char *filename,
                                SymbolTable &globalsymboltable,
                                Language language,
                                DBModifiedCallback callback);
  friend void CloseDatabase(SymbolTable &globalsymboltable);
};
```

Method Definitions

`Handle()`	Returns the "handle" (internal identifier) for the database. This is not usually useful for rule writers.
`FileName()`	Returns the file containing the database.
`ModifiedTime()`	Returns the last time of modification.
`StartTransaction()`, `EndTransaction()`	Delimit a "transaction." Call these routines around each series of database queries to prevent a writer from changing the database. `StartTransaction()` triggers a call of the callback routine provided in the `OpenDatabase` call if the database has been modified since the last transaction.

	Transaction management is handled by the rule engine, so rule writers need not be concerned about it.
`Contains()`	Tests whether a `Symbol` is found in the database. This can be useful if you have multiple databases open.
`Macros()`, `GlobalVariables()`, `GlobalFunctions()`, `GlobalTags()`, `LocalTags()`, `GlobalTypedefs()`, `GlobalModules()`, `Files()`, `FunctionTemplates()`	Return iterators to scan through all objects of the specified type. Both global and local iterators are provided for `Tag`s. The combination of the two iterators returns *all* `Tag`s in the database.
`GlobalSymbols()`	Two overloaded iterators return either all global symbols, or all global symbols of a specified name and `PerKind`.
`Find()`	A `Find()` method, for finding a global object by name, is defined for each object type. The desired object is returned in the second parameter. If your database contains more than one instance of the object, only the first instance is returned. Use the corresponding iterator to find all instances.
`ActivateFiles()`	Limits searches to the files specified in the `filename` array, and optionally limited to the languages specified by the `lang` mask. Each `filename` must be a full canonical filename of the form *host*`:`*fullpath*. Run the command `/opt/softbench/bin/path_to_canon` *filename* to see a sample canonical name. By default, all files are active. Pass a `count` value of 0 to resume searching all files.
`SymbolsAtLocation()`	Returns an *attribute* iterator listing all symbols at a specified location. Null values of `filename`, `line`, or `column` mean to return symbols with the

specified `name` in any `File`, or line or column in a `File`. `ignorecase` specifies a case-insensitive search, and `useregexp` specifies that `name` is a `regexp(5)`-style regular expression. If `useregexp` is true, `name` can contain any normal non-extended regular expression. The RE can also use `+` (preceding RE must appear 1 or more times) and `?` (preceding RE must appear 0 or 1 times). `scoping` specifies the type of "scoping" to use when searching for the symbols. See the Static Analyzer chapter of the *C and C++ User's Guide* for an explanation of scoping.

`EnclosingFunction()` The function whose scope includes the specified file and line. Returns `FALSE` if no function includes the location.

Tag Object

`Tag` objects represent all aggregate types, such as classes and enums. The two-part representation of aggregates (the `Tag` and the `Enum`, `Struct`, `Class`, or `ClassTemplate`) allows the database to handle self-referential objects.

Each tag can be mapped onto its corresponding aggregate, and vice versa.

The `Tag` inherits from `Symbol`, and therefore contains all information about the aggregate's name.

```
class Tag : public Symbol {
  public:
    Tag();
    ~Tag();

    PerKind TypeKind() const;
    DBboolean EnumType(Enum &enumeration) const;
    DBboolean StructType(Struct &structure) const;
    DBboolean ClassType(Class &cppclass) const;
    DBboolean ClassTemplateType(ClassTemplate &classtempl) const;
};
```

Method Definitions

`TypeKind()`	Returns the type (class, enum, struct, or template) of the tag.
`EnumType()`, `StructType()`, `ClassType()`, `ClassTemplateType()`	These functions convert a `Tag` into the corresponding object.

TemplateArgument Object

`TemplateArgument` objects represent C++ parametric type arguments. They are used for class template and template function arguments.

`TemplateArgument` inherits type and name information from `TypedSymbol`.

```
class TemplateArgument : public TypedSymbol {
  public:
    TemplateArgument();
    ~TemplateArgument();

    DBboolean ArgumentOf(ClassTemplate &classtempl) const;
    DBboolean ArgumentOf(FunctionTemplate &funtempl) const;
};
```

Method Definitions

`ArgumentOf()` — Returns the class or function template of which this is an argument. If you do not know what type of object contains the `TemplateArgument`, you can call each of the overloaded `ArgumentOf()` functions until one returns `TRUE`.

Typedef Object

`Typedef` objects represent named types.

`Typedef` inherits type and name information from `TypedSymbol`.

```
class Typedef : public TypedSymbol {
  public:
    Typedef();
    ~Typedef();
};
```

Method Definitions

`Typedef` defines no methods of its own, but inherits all typing and symbol information from `TypedSymbol`.

The TypedSymbol Base Class

`TypedSymbol` is the base class through which all typed objects (`Variable`, `Parameter`, `Function`, `Typedef`, `DataMember`, and `TemplateArgument`) inherit their type and name information. `TypedSymbol` inherits its name information from `Symbol`.

As with `Symbol`, you will not encounter `TypedSymbol` objects in the database. The class is used only as a parent class for other objects. The attributes that describe an object's type (`Type` and `TypeQualifiers`) are inherited from `TypedSymbol`. All attributes that make sense for named objects, such as the object's name, definition location, and so on, are defined in `Symbol` and are available to all derived types.

Notice that `Symbol` and `TypedSymbol` inherit their concept of persistent database objects from `PerBase`.

```
class TypedSymbol : public Symbol {
  public:
    TypedSymbol();
    ~TypedSymbol();

    // Type and type qualifiers of this symbol.
    Symbol Type() const;
    const char *TypeQualifiers() const;
};
```

`Type()`, `TypeQualifiers()` — Return the type symbol and qualifier string (such as "`*`" for a pointer) of the variable. See "Type Qualifiers" for a more complete explanation.

Variable Object

`Variable` represents complete and incomplete program variables.

For incomplete variables, only the base `Symbol` methods are valid. All other methods return `FALSE` and/or null results.

`Variable` inherits type and name information from `TypedSymbol`.

```
class Variable : public TypedSymbol {
  public:
    Variable();
    ~Variable();

    DBboolean Scope(Block &block) const;
};
```

Method Definitions

`Scope()` Returns the enclosing block within which the variable is defined, or `FALSE` if the variable is global.

Using the Database API

The following example is one of the actual rules delivered with the SoftBench CodeAdvisor product. This real-life example will help you to understand how the database API is used in rules.

The Example Rule

This rule, `UserRulesLocalHides`, detects local identifiers with the same name as a local or inherited data member or member function. You can read a description of the rule in the SoftBench CodeAdvisor online help for the `UserRulesLocalHides` rule. The source of the rule is included in the next section, and can also be found in `/opt/softbench/examples/CodeAdvisor/Rules/ruleLocalHides.C`. To test the rule, make the example rule library, as explained in the `Makefile`, and install the new library in `/opt/softbench/lib/rulelibs`.

The algorithm used is straightforward: for each class, scan through all member functions. In each function, check all parameters and all local variables to see if there is a conflict.

The majority of this processing happens in two functions: `UserRulesLocalHides::check()` and `shadow()`. `shadow()` is a utility routine that does the actual checking for conflicts.

Understanding the Example Rule

As with the simple `UserRulesCapClass` rule in Chapter 3, `UserRulesLocalHides` starts out by defining the `UserRulesLocalHides` class interface. Every rule you write should define the class interface like this. The only difference will be the actual name of the rule class.

The shadow Function

After defining a short utility function to extract the simple name of a class member (the part after the "`::`"), the code defines the `shadow` utility function. This function returns `TRUE` if it finds any visible symbol in the class `cl` or any base (inherited) classes with the same name as `sym`. If so, the hidden symbol is returned in `hidden_sym`.

After getting the name of the symbol, `shadow()` begins by iterating through all local functions in `cl`. (`AllFunctions()` returns all member functions in a class, and all function templates in a template.) Next it iterates through all local data members. The test is the same for both types of symbols: if the symbol is visible (if it is in this class, or is a non-`private` member of a base class), and has the same name as `sym`, return the `hidden_sym`.

If there are no collisions in the current class, `shadow` iterates through all base classes of the current class and calls itself recursively to check the base classes. Note that `BaseClasses()` returns only the *immediate* parent or parents of a class, not all ancestor classes. The recursive call takes care of moving up the inheritance chain.

If the current class has no base classes, `BaseClasses()` returns no items, so the iterator loop is never entered. Execution continues after the `ITERATE_END(tagi)` and the function returns `FALSE`, indicating it has not yet found any collisions.

kindMask and langMask

`kindMask()` returns the bitmask `1 << KIND_TAG`, indicating that `check()` should be called on all `Tag` objects. As with `UserRulesCapClass`, this rule applies only to `Class` objects, but `check()` is invoked only on `Symbol` objects. `Class` does not inherit from `Symbol`, so `check()` cannot be invoked on `Class`. The rule must accept all `Tag`s and filter out the non-`Class` objects.

`langMask()` indicates that the rule applies only to C++ code.

The check Function

Other than `shadow()`, the `check()` function handles the majority of the rule processing. `check()` is called for each `Tag` object. The first test in `check()`, as with `UserRulesCapClass`, screens out all class or template instances. (Since the instances have the same member names as the classes and class templates, there is no need to check them.) Note that `UserRulesLocalHides` does not filter out structs and unions like `UserRulesCapClass` did. Since structs and classes are essentially identical in C++, it is possible to have name conflicts in structs just as in classes.

`check()` then iterates through all functions in the class. Remember that `AllFunctions()` returns all member functions of a class, as well as all function templates in a template, so the same code can handle both cases.

The loop first rejects "synthetic" compiler-generated functions and "incomplete" functions. (Incomplete functions have a declaration but no definition, and therefore no `FunctionBlock`. Ordinarily you should check for incompleteness by testing the `DefinitionSite()`, as explained in "Incomplete Objects". However, since `check()` needs the `FunctionBlock` later, testing for `FunctionBlock` is a handy alternate way to check for incompleteness.) Since "incomplete" functions have no code definition and therefore no parameter definitions, they should not be checked.

Finally, `check()` iterates through all the member function parameters and all variables defined within the function, calling `shadow()` on each one. If `shadow()` detects any conflicts, `check()` calls `violation()` to signal a problem.

Notice that `violation()` passes `UserRules` as the "help location." Since no help node is specified (using the *helpvolume_helpnode* convention), the help system uses the rule name (`UserRulesLocalHides`) as the name of the help node.

Final Definitions

`errorMess()` defines a generic one-line description of the rule.

`name()` returns the name of the rule. For simplicity, `name()` should be the same as the rule class name.

Once the rule class is completely defined, a `static` definition forces a call to the `Rule` constructor. This links the rule into the rule engine and enables it for use.

Example Files

Source files for the example rules are available on-line in `/opt/softbench/examples/CodeAdvisor/Rules`. The files in this directory include:

`Makefile`	A `make` control file to build all the example files.
`ruleCapClass.C`, `ruleLocalHides.C`, `ruleNameConventions.C`, `ruleProhibDefines.C`, `ruleProhibIdent.C`	Sources for the example rules. Notice that the `ProhibDefines` and `ProhibIdent` rules are identical to the corresponding rules that are shipped with CodeAdvisor, except they are named `UserRulesProhib`*rule* instead of `Prohib`*rule*. However, for demonstration purposes they use the standard CodeAdvisor rule tables under `/opt/softbench/config/ruletables/$LANG`.
`UserRules.htg`	On-line help file for the example rules.
`Testcase`	A directory containing a simple test case for `UserRulesCapClass`.
`ruleTemplate.C`	A "template" file to use as a starting point when writing rules.
`debugPoints.C`	Source for the debuggable `main()` in `softcheck`—used only by SoftBench Debugger.
`RuleIndex.htg`	The main index into the on-line rule help. This file is organized according to the definitions in the rule group configuration file. If you change that file, you may want to change the index file accordingly.

You can examine the sources for any of the rules and support files. You can also build and test the rules using the `Makefile`. "`make`" builds the rule library. "`make all`" builds the rule library, test case, and help volume. See the comments in the `Makefile` for more information.

You can test the rule library using SoftBench CodeAdvisor, or using `softcheck`. Install the library in `/opt/softbench/lib/rulelibs` to test using

SoftBench CodeAdvisor, or specify the library location using the `-l` flag to `softcheck`.

"`make install`" installs the rule library and help volume in the standard locations. Note that you must do the install as "root" in order to install, since the required directories under `/opt/softbench` are not typically writeable by ordinary users.

The UserRulesLocalHides Rule

```
#include <Rule/Rule.H>
#include <string.h>
#include <assert.h>
#include <stdio.h>      // Note, only sprintf is used; no stdio/iostream mix

class UserRulesLocalHides : public Rule
{
public:
   virtual int kindMask() const;
   virtual Language langMask() const;
   void check(SymbolTable *, const Symbol &);
   virtual const char *errorMess() const;
   virtual const char *name() const;
};


// Return a pointer to the simple name of member or namespace qualified obj.
// Note that member names can be as complicated as
// Outer::Inner::Inner(const Outer::Inner &)
// so between last '::' (before '(') and first '(' is what is needed
//
void simpleName(char const *x, char *buf)
{

    char const *currcolon, *nextcolon, *start, *end = strchr(x, '(');

    if (!end)
        end = x + strlen(x);

    currcolon = strstr(x, "::");

    while (currcolon &&
           ((nextcolon = strstr(currcolon+1, "::")) < end) &&
           nextcolon)
        currcolon = nextcolon;

    if (currcolon)
        start = currcolon + 2;
    else
        start = x;

    strncpy(buf, start, end-start);
    buf[end-start] = '\0';
}
```

```
// Test to see if a symbol hides (or has the same name but doesn't hide)
// some member of a class, or some inherited member.
static DBboolean shadow(const Symbol &sym,      // symbol that may be shadowed
                        const Class &cl,        // class to check members of
                        Symbol &hidden_sym,     // symbol that sym collides with
                        DBboolean baseclassp = false    // is this a baseclass
                                                // of one where sym defined?
                        )
{
   char name[1024];
   simpleName(sym.Name(), name);

   // test sym name against local member functions
   ITERATOR(Function) fmi=cl.AllFunctions();
   ITERATE_BEGIN(fmi)
   {
      char buf[1024];
      simpleName(fmi.Name(), buf);
      if ((!(baseclassp && IS_PRIVATE(fmi.Attrib()))) && // visible
          strcmp(name, buf)==0) {                       // name matches
         hidden_sym = fmi;
         return true;
      }
   }
   ITERATE_END(fmi)

   // test sym name against local member data
   ITERATOR(DataMember) dmi=cl.DataMembers();
   ITERATE_BEGIN(dmi)
   {
      char buf[1024];
      simpleName(dmi.Name(), buf);
      if ((!(baseclassp && IS_PRIVATE(dmi.Attrib()))) && // visible
          strcmp(name, buf)==0) {                       // name matches
         hidden_sym = dmi;
         return true;
      }
   }
   ITERATE_END(dmi)

   // test base classes of this class
   ATTRIBUTE_ITERATOR(Tag) tagi=cl.BaseClasses();
   ITERATE_BEGIN(tagi)
   {
      Class baseclass;
      if (!tagi.ClassType(baseclass)) {
         // Can't put ClassType call in assert, since assert does not
         // invoke its argument in non-debugging environments.
         // We always need the side-effect of setting "baseclass".
```

```
         assert(tagi.ClassType(baseclass));
      }

      if (shadow(sym, baseclass, hidden_sym, true))
         return true; // as soon as you find one, it's safe to return
   }
   ITERATE_END(tagi)

   return false;
}

int UserRulesLocalHides::kindMask() const
{ return 1<<KIND_TAG; }

Language UserRulesLocalHides::langMask() const
{ return LANGUAGE_CPP; }

// For all member functions of all classes,
//   Find all local variables defined in all blocks within function
//   Also find all parameters of the member function
//      See if any local variable/parameter duplicates the name of any local or
//              inherited *visible* member.
//      If so, report a violation.
//
void UserRulesLocalHides::check(SymbolTable *, const Symbol &sym)
{  Tag tag, templ;
   Class cl;
   Symbol hidden_sym;

   if (!sym.SymbolToTag(tag) || !tag.ClassType(cl) || IS_INSTANTIATED(cl.Attrib()))
      return; // look at classes and templates, skip instances

   // Find all member functions
   ITERATOR(Function) fmi=cl.AllFunctions();
   ITERATE_BEGIN(fmi)
   {
      if (IS_SYNTHETIC(fmi.Attrib()))
         continue;  // skip compiler generated functions

      // locate function's main block
      Block fblock;
      if (!fmi.FunctionBlock(fblock))        // Incomplete function?
         continue;

      char buf[1024];
      // check member function's parameters
      ITERATOR(Parameter) parami=fmi.Parameters();
      ITERATE_BEGIN(parami)
      {
```

```
         if (shadow(parami, cl, hidden_sym)) {
            sprintf(buf,
                   "Parameter '%s' of '%s' hiding member '%s' with same name",
                   parami.Name(), fmi.Name(), hidden_sym.Name());
            violation(parami, buf, "UserRules");
         }
      }
      ITERATE_END(parami)

      // check variables defined in any block within function
      ITERATOR(Variable) vari=fblock.BlockVariables();
      ITERATE_BEGIN(vari)
      {
         if (shadow(vari, cl, hidden_sym)) {
            sprintf(buf,
                   "Local variable '%s' in '%s' hiding member '%s' with same name",
                   vari.Name(), fmi.Name(), hidden_sym.Name());
            violation(vari, buf, "UserRules");
         }
      }
      ITERATE_END(vari)
   }
   ITERATE_END(fmi)
}

const char *UserRulesLocalHides::errorMess() const
{
 return("Local variable or parameter hiding member (function or data) with same name");
}

const char *UserRulesLocalHides::name() const
{
 return("UserRulesLocalHides");
}

// Force a call to base class constructor in the main program
static UserRulesLocalHides instance;
```

5

Implementing Your Rule

Now that you understand the building blocks you can work with, you can decide how to implement your rule. You must decide what approach will work best within the SoftBench CodeAdvisor framework.

Design Guidelines

The following are suggested guidelines for your rule designs.

- *Do not generate excessive violations.*
  It's usually better to miss flagging a few errors than to flag incorrect violations. If you generate incorrect violations, or too many violations, the user will tend to discount the warnings.

  Rules that flag "possible" errors should be placed in a "possible error" rule group, so the user can enable them only if desired.

  Note that "noisy" rules may be useful to flag possible problems for code-reading sessions.

- *Choose breadth over depth.*
  Do not try to cover every possible case when writing a rule. There is often a nearly-infinite supply of odd corner cases. Your time is better spent covering the major cases, and then moving on to cover the major cases of another rule.

- *Check templates and classes, not instances.*
  Almost all rules that test classes check the class structure. For example, a rule might check the safety of the constructor, or the member functions and data members in the class. Since class instances have the same structure as their parent template, you should not test instances. If you did, it would result in duplicate error messages for the template and for all its instances.

- *Write your code to work for both classes and templates.*
  Most class rules apply equally well to classes and templates, so it makes sense to check both. Convert `Symbol` objects to `Tag`s using `Symbol::SymbolToTag()`, then verify the `Tag` refers to a `Class` using `Tag::ClassType()`. This test succeeds for both classes and class templates. Be aware that the test also succeeds for structs and unions in C++ code, since C++ treats them almost identically. Use the functions `WAS_STRUCT` and `WAS_UNION` to test the object's `Attrib()` value in the rare case (such as `UserRulesCapClass`) when it's important to distinguish between classes, structs, and unions. See the `UserRulesCapClass` rule in Chapter 3 for an example.

  Use `AllFunctions()` to iterate through member functions, since this iterator also returns function templates.

- *Test for incomplete objects.*
  Design your rules so they properly handle incomplete objects. Certain objects (`Variable`s, `Function`s, `Enum`s, `Struct`s, `Class`es, and `ClassTemplate`s) can exist in an "incomplete" state. This happens when only a forward or external reference is found, so no definitional information is available. (Class template instances are also represented as incomplete classes.) See "Incomplete Objects" in Chapter 4 for more information on detecting incomplete objects.

- *Implement a test case before implementing a rule.*
  It's possible that the C++ compiler already detects the rule you are considering. Make sure the job hasn't already been done for you. Try higher "verbosity" levels on your compiler, such as "`+w`" on HP-UX.

- *For table-driven rules, implement a general solution.*
  When writing a table-driven rule, consider similar problems that could be solved by your rule. Would the addition of another field make the rule more generally useful, without making the table unwieldy?

Implementing the Rule

Once you understand the rule model, the Static API, and the design guidelines, you can begin implementing your rules.

The example files provided with the system can be very helpful when learning the rule programming environment. If you have not studied the examples described in "Example Files" in Chapter 4, please do so before proceeding.

The following sections outline a recommended procedure for developing rules.

Decide What to Implement

First, you should consider what kind of rules you want to implement. There are several possible classes of rules:

- Rules that detect subtle C++ usage errors, such as the rules shipped with SoftBench CodeAdvisor. You may be aware of other C++ areas that can cause problems. If so, you may want to implement your own rules to check for them. Be aware that Hewlett Packard intends to continue to expand the set of rules shipped with SoftBench CodeAdvisor, and it is possible that your rule may be superseded by a new rule in a future release. However, in the meantime you will benefit from the rule.
- Rules that detect common coding errors, such as the SoftBench CodeAdvisor rule that detects accidental use of "`/n`" instead of "`\n`".
- Rules that enforce adherence to a standard such as XPG4.
- Stylistic rules that help to enforce local coding conventions. This can be done most efficiently with table-driven rules.

Develop a list of rule candidates. You may want to examine the current literature to get ideas for good rules. Your local coding conventions may provide a good source of ideas.

When you have drawn up a list of potential rules, you should prioritize them. Which rules are the most practical to implement? You don't want to spend time on a rule that turns out to be nearly impossible to implement. Your time might be better spent writing several less-challenging rules.

Keep the Static API capabilities in mind when assigning difficulty scores. An apparently simple rule may be difficult to implement if it requires program knowledge that the database does not provide.

Next, which rules would provide the most benefit? Which rules detect severe errors, and which detect minor problems? A rule might be simple to implement, but will it actually help to prevent coding problems? Will it catch only minor problems that could be ignored without penalty?

Once you understand these properties, you can use a simple "biggest bang for the buck" measure to decide which rules to implement.

Designing the Rule

Write a clear description of the rule. Write example code that illustrates the problem. This example will provide useful test cases.

Examine the logic of the rule. Can it be implemented as a table-driven rule? How can you implement it using the database? For example, does the rule apply to all functions? If so, you should use the API's built-in iteration to select all functions for you. Define a `kindMask()` that causes the rule engine to invoke your rule on every `Function` object.

Does the rule apply to all classes? If so, you must implement a "two-step" approach. Your rule's `check()` function can only be called on objects derived from `Symbol`, but `Class` does not inherit from `Symbol`. You must instead specify a `kindMask()` of "`1 << KIND_TAG`", and filter out the non-`Class` objects. See the `UserRulesLocalHides` rule in "Using the Database API" in Chapter 4 for an example.

It's a good idea to add a prefix to the beginning of your rule to avoid possible name collisions with future HP-supplied rules. For example, you could name your rule `MyruleTestThis` instead of just `TestThis`. Because your rule name is often used as the name of the corresponding online help node, you should only use alphanumeric characters in your rule name.

Once you have decided how the API should call your `check()` function, you can determine what `check()` should do when it is called. What does the rule test for? If it checks the parameters of a function, you would want to iterate using `Function::Parameters()`. If the rule checks for inheritance problems in classes, you would iterate over `Class::BaseClasses()` and recursively test

each of the base classes. The exact procedure you use will depend on your rule's semantics.

Compiling the Rule

The `Makefile` in `/opt/softbench/examples/CodeAdvisor/Rules` correctly compiles and links rule libraries, and can be invoked from the SoftBench Project builder. If you create your own `Makefile` or compile from the command line, you must remember the following points:

Rule code must be compiled with the `aCC` compiler, using the following options:

```
aCC -I/opt/softbench/include +z -c rulefile.C
```

The `Rule` class and other important types are defined in files under `/opt/softbench/include`. The `+z` option instructs the compiler to generate relocatable (position-independent) code.

Rules should be linked using a command like this:

```
aCC -L /opt/softbench/lib/SB6.0 -b rulefile.o -o libRules.sl
```

Notice that the rule library name *must* start with `lib` and end with `.sl`.

The rule libraries are defined in `/opt/softbench/lib/SB6.0`. `-b` instructs the linker to create a shared library.

Testing the Rule

After you write the initial code for your rule, you will enter an iterative development process. Run the rule on your example code. Does it catch all the cases it should?

You should also run the rule on some large code samples. Verify that any violations are valid. Does the rule detect the appropriate error cases? Does it flag code that should not be flagged? Does it trigger so many violations that users will get overwhelmed and tend to ignore the rule's warnings?

Modify your implementation to refine the error cases detected by the rule, and test again. Continue in this process until your rule meets your requirements.

You may find `softcheck` very useful in testing your rules. You can easily invoke SoftBench CodeAdvisor on your rule with the command

`softcheck -l` *YourRuleLibDir* `-r` *RuleToCheck*

See *softcheck(1)* for more information on `softcheck`. See "Debugging Your Rule" for a full explanation of running and debugging rules.

Adding Your Rule to a Rule Group

The SoftBench CodeAdvisor product includes over 1000 rules. Many of them flag potential problems, so they can generate violations in cases where there is currently no error. Since it would be difficult to use the output of all rules at once, rules are organized into *rule groups*.

Each rule group contains rules that are related in some way. You can select any set of rule groups, and run the analysis using only those rules.

For example, one of the most useful rule groups is the `DefiniteDefects` group. Any violations generated by these rules almost certainly indicate a serious problem in your program.

You must add your rule to a rule group in order to use it from the CodeAdvisor user interface.

Classifying Your Rule

You should examine the set of rule groups and determine which group is the best match for your rule.

Rulegroup File Locations

To add your rule to an existing rule group, copy the file `/opt/softbench/config/rulegroups/$LANG` (where `$LANG` is `C` for English installations) into one of the following locations, depending on how widely the group should be made visible. You can copy only the groups that you want to change, rather than copying the entire rulegroups file.

`/etc/opt/softbench/config/rulegroups/$LANG`

> Local changes and customizations. All users on the system are affected by these changes. A rulegroups file in the `/etc/opt/softbench` hierarchy totally *replaces* the rulegroups file under `/opt/softbench` on the same machine.

`$HOME/.softbench/rulegroups`

Personal changes. Visible within all projects for that user.

`$PROJECTROOT/Projects/`*project-name*`/rulegroups`

Personal changes. Visible only within the specified project.

The locations are checked in the order above. Later information overrides previous information; for example, personal customizations under `$HOME/.softbench` are merged in with the system-wide customizations in `/etc/opt/softbench/config`, and override it on a group-by-group basis.

`$PROJECTROOT` points to the user's specified project information root. By default, this root is `$HOME/.softbench`. *project-name* is the name of the specific project to customize.

Rulegroup File Format

Each non-comment line in the rulegroups file contains three fields, separated by commas:

- The rulegroup name. Note that some groups are sub-groups that are included in other groups. These group names start with a period (".").
- A comment field.
- A list of rule names, separated by colons (":").

Edit your copy of the file and add the name of your rule to the appropriate rule group. (Note that you cannot edit this file using the standard `vi` or Softvi editors, since those editors can't handle the extremely long lines in the file. SoftXEmacs, the default editor, can edit the file.)

Creating a New Rule Group

If your rule does not fit into any of the existing rule groups, or if you want to create a group that contains only your locally-written rules, you can add a new group to the rulegroup file. Simply add a new line, using the format described above. If you add your rule to one of the local customization locations, you can create a file containing only your new group. The new file will be merged into the existing rulegroup file.

The name of your group can contain *only* alphanumeric characters. The first character must be alphabetic.

After adding your new group to the rulegroup file, start SoftBench and display the CodeAdvisor page. You should see your group in the "Rule Groups" area.

Updating the Group Index

Under the rule group selection area on the CodeAdvisor page, the Rule Group Help... button displays an index of all rules sorted by rule group.

After adding a new rule or rulegroup, or after moving a rule from one group to another, you should update the rule group index.

The source for the group index help volume is located in `/opt/softbench/examples/CodeAdvisor/Rules/RuleIndex.htg`. Edit this file to reflect your changes, compile the help volume using "`$PATH=/usr/dt/bin:$PATH dthelptag RuleIndex.htg`", and install the `RuleIndex.sdl` file in `/etc/opt/softbench/dt/appconfig/help/$LANG`.

Debugging Your Rule

After you have implemented your rule, you can test it by running it under SoftBench CodeAdvisor or by using the `softcheck` command.

SoftBench CodeAdvisor provides the complete user interface that your users will see, and also allows you to test the linkage to your on-line help. Install the new library in `/opt/softbench/lib/rulelibs`.

`softcheck` provides a very simple and "light-weight" interface to the rule engine. You *must* use `softcheck` if you need to use SoftBench Debugger to find subtle problems. See the *softcheck(1)* reference page for an explanation of `softcheck`.

Be aware that the rule engine holds open a transaction during the time that you debug your rule. Other processes will be unable to update the database (the `Static.sadb` file) while you are debugging.

Running softcheck Under SoftBench Debugger

The example `Makefile` handles debug and static flags (`-g` and `-y`) from Builder. If you invoke `make` directly from the command line, you must specify the debug flags using "`make CXXOPTS="-g -y"`".

The `softcheck` executable shipped with SoftBench CodeAdvisor is not debuggable. User defined rules can be debugged by building the rules into a debuggable shared library and then running the debuggable SDK version of `softcheck`, `/opt/softbench/examples/CodeAdvisor/softcheck`, under SoftBench Debugger.

1. Run Debugger and load the debuggable `softcheck` program (above).
2. If the debugger is running in "stand-alone mode," so that SoftBench has not conveyed project information to it, you may need to choose "`File: Add Source Directories ...` " to tell the debugger where your source files are located.
3. Once Debugger has started, choose "`File: Rerun ...` ".
4. In the "`Program Arguments`" Input Box, enter "`-p` *project*" to specify the project you are working on. If you do not specify a project, enter "`-d` *dir*`/Static.sadb`" to tell `softcheck` where to find your program's

Static database. Enter "`-l` *library-dir*", where *library-dir* is the directory containing your debuggable rule library. (This is not necessary if your library is in the standard location, `/opt/softbench/lib/rulelibs`.)

5. If you want to run only a few rules in your test library, specify them by entering "`-r` *rule-name*" or "`-r` *rule-group*'"" in the "`Program Arguments`" Input Box. Multiple rules or rule groups can be specified by separating them by colons. Individual rules can be excluded using the syntax "`-r ~`*rule-name*".

6. If desired, you can set the environment variable `RULE_DEBUG` by entering the appropriate values in the "`Program Environment Variables`" section of the dialog box. See "Tracing Rule Execution".

7. You may want to select (Save As Defaults) so the preceding setup information is stored for future debugging sessions. If you save your settings, you can retrieve them in future sessions by selecting (Load Defaults).

8. Select (OK) to run `softcheck`.

9. SoftBench Debugger now starts `softcheck`, and pauses in `main()` in the file `debugPoints.C`. Set a breakpoint at the procedure `libsLoaded()` and select (Continue). See the next section for detailed instructions on setting breakpoints at `libsLoaded()` and in your rule library.

10. Your program pauses at `libsLoaded()`. Your rule library is now loaded into the program. Choose "`File: Enable Images/Libraries ...` " and load debug information about your rule library, as described in the next section.

11. You can now display your rule's source or set breakpoints in your rule by referring to the fully-qualified procedure names (such as `UserRulesCapClass::check`).

Setting Breakpoints In Your Rule

Since rules are stored in dynamically-loaded shared libraries, you must know how to debug these libraries within SoftBench Debugger. You cannot set breakpoints in your rule library immediately after running your program, since the library has not been loaded. You must load your rule libraries first.

1. Enter "`libsLoaded`" in the SoftBench Debugger "`( ):`" Input Box and choose "`Break: Set At ( )`". (Or, since `libsLoaded()` is directly after `main()` in `debugPoints.C`, you can simply scroll the window down to `libsLoaded` and set the breakpoint by clicking in the Annotation Margin to the left of the Source File Edit Area.) This sets a breakpoint on an empty function that is called after the rule libraries are loaded, but before any rules are checked.

2. Select (Continue) as needed until you reach the breakpoint at `libsLoaded`. Depending on how the libraries were built and a few timing issues, you may encounter `SIGCHLD` and/or `SIGALRM` signals, or you may stop as your shared library is loaded.

3. Choose "`Execution: Images ...`".

   Your rules library will be one of the last libraries listed in the "Dynamic Images" dialog box. Select the toggle button next to your library to load its debug information.

   The debug information can also be loaded by entering "`property libraries -add`" (or "`pro lib -add`") followed by the basename of your rule library in the "`Debugger Input`" Input Box.

4. Set any break points needed in your rules.

5. Select (Continue). When you encounter the breakpoints in your rule, you can debug as you normally would.

Tracing Rule Execution

You can cause `softcheck` to generate some extra output that may be useful in your debugging. The environment variable `RULE_DEBUG` accepts several values:

`RULE_DEBUG=1` Displays a message just before calling each rule. The message indicates the object on which the rule is being invoked. This can be very useful if, for example, you encounter a core dump in your rules. By turning on this message, you can immediately see what rule caused the core dump, and what object triggered the problem.

`RULE_DEBUG=4` Displays a message when the `Static.sadb` file is loaded into the database.

These values are elements in a bitmask, and can be combined. For example, "`RULE_DEBUG=5`" displays messages when rules are called *and* when `Static.sadb` files are loaded.

You can also set the `SA_SHLIB_TEST` variable (to any value) to display a message when a rule library is loaded. This allows you to ensure that your library is being loaded properly. If not, you may need to change the `-l` arguments to `softcheck`, or ensure your library is installed in the standard location searched by `softcheck`. Note that this diagnostic message can sometimes appear *after* violations have been displayed.

You can set `RULE_DEBUG` and `SA_SHLIB_TEST` at a shell prompt before calling the `softcheck` command, or in the "`Program Environment Variables`" Input Box in the "`File: Load New Executable ...` " dialog box in SoftBench Debugger.

Documenting Your Rule

In addition to the normal documentation that is recommended for any program, you should provide on-line help for your rule.

When your rule detects and reports a violation, the user has the option of displaying an on-line summary and explanation of the rule. In SoftBench Program Builder, this is done by selecting the Help button after selecting the violation display.

When the user selects Help, a message is sent to the SoftBench On-Line Help server to display the help text. The help server searches for and displays the help associated with your rule.

Writing the On-Line Help

SoftBench uses the CDE-standard `dthelptag` on-line help tool. Use `dthelptag` to compile your help volume, and `dthelpview` to view it. See *Common Desktop Environment: Help System Author's and Programmer's Guide* for a description of the HelpTag language and compilation tools. A compressed PostScript copy of this manual is available in `/opt/softbench/dt/doc/Help_Pgrmer_Guide.ps.Z`.

Your help text should conform to the format used by the standard SoftBench CodeAdvisor help. Each node should include the following:

- A node title, named after the rule
- An italicized one-line summary of the rule
- A more in-depth explanation of the rule and its rationale
- "What Triggers Rule": A clear description of the conditions that cause the rule to fire
- "Corrective Action": Recommended steps to resolve the problem
- "Exceptions": Cases in which a rule violation may be ignored

If desired, you may also mention the rule's origin.

See `/opt/softbench/examples/CodeAdvisor/Rules/UserRules.htg` for a sample help volume. Notice the two entity declarations at the start of the example file. These entities define the character set to be used, and are *required*

in all help volumes. If you are writing help for languages other than English, refer to the CDE Help System Author's Guide for additional instructions.

Referring to Other Help Volumes

The basic HelpTag tools allow you to refer to other nodes within your help volume. The SoftBench help server has been extended to allow you to refer to a node in another volume, using the `EXTERNREF` hyperlink keyword:

```
<link hyperlink="EXTERNREF helpvolume helpnode" type="AppDefined">
Hyperlink text
<\link>
```

Associating Your Rule With the On-Line Help

Your rule specifies a help volume name (and, optionally, a help node) in the `help_volume` argument to `violation()`. The help volume (the *helpvol*`.sdl` file generated by the `dthelptag` command) must use the same name as the `help_volume` argument.

Each rule help node within the help volume must have an `id=` entry that specifies the node name. This may be the same as the name of the rule it describes, or it may be any arbitrary name if the name is specified in `help_volume`.

For example, the help node for the `UserRulesCapClass` rule is in the `UserRules.htg` help volume, and is named `UserRulesCapClass`. The `help_volume` parameter specifies `UserRules` as the name of the help volume. By default, the name of the rule (`UserRulesCapClass`) is used as the name of the associated help node. To specify a different help node name, pass a value like "`UserRules_AnotherNode`" in `help_volume`.

Help node names can contain *only* alphanumeric characters, and must begin with a letter. For this reason, it is best if you use only alphanumeric characters in your rule name.

Installing the On-Line Help Volume

As root, install the *rulelib*`.sdl` file in the `/etc/opt/softbench/dt/appconfig/help/$LANG` directory. For non-localized installations, `$LANG` has a value of `C`.

A

Detailed Database Type Descriptions

The Static database interface provides two header files to declare the constants, types, and functions used to access the database. These files are:

`DB_Common.h` Common types and constants used by the database. The contents of this file are described in this Appendix.

`DB_Read.h` The "read" interface to the database. The contents of this file are described in "Object Interfaces" in Chapter 4.

The database header files are found under *install_dir*`/include/DB_Access`.

Object Kind

Database objects are represented by a "handle" of type `PerHandle`. They are typed by the enum `PerKind`.

The first four `PerKind` values (`KIND_BADSYMBOL`, `KIND_SYMBOLENTRY`, `KIND_FILEENTRY`, `KIND_RELATION`) are only used internally. You will not encounter them.

The remaining `PerKind` values correspond to the different object types defined in the database:

> `KIND_REFLIST`, `KIND_ENUM`, `KIND_STRUCT`, `KIND_CLASS`, `KIND_CLASSTEMPLATE`, `KIND_SOURCEFILE`, `KIND_SCALAR`, `KIND_MODULE`, `KIND_MACRO`, `KIND_IDENTIFIER`, `KIND_LABEL`, `KIND_TAG`, `KIND_TYPEDEF`, `KIND_VARIABLE`, `KIND_PARAMETER`, `KIND_BLOCK`, `KIND_FUNCTION`, `KIND_ENUMMEMBER`, `KIND_DATAMEMBER`, `KIND_FUNCTIONMEMBER`, `KIND_FUNCTIONTEMPLATE`, and `KIND_TEMPLATEARGUMENT`.

See "Object Interfaces" in Chapter 4 for a description of each object type.

Each database object is tagged with a `PerKind`, allowing you to determine what type of object it represents. Various functions (such as the `SymbolTable::GlobalSymbols` iterator) allow you to "filter" their results by specifying the `PerKind` you are interested in.

Rules specify a `kindMask` that limits the `PerKind`s for which they are designed.

Attributes

Each object has an attribute field that describes the attributes pertinent to that object. The `Attribute` type is defined as a bit vector:

```
typedef unsigned long Attribute;
```

Attributes are combined as necessary for a given object.

The interface also defines inline functions in `DB_Common.h` to test the associated `Attribute` values. These predicate functions generally start with `IS_`, `WAS_`, or `HAS_`, such as `IS_GLOBAL()`, `WAS_STRUCT()`, and `HAS_DEFAULT()`. The predicate associated with each `Attribute` value is listed below.

The attributes are:

`ATTR_GLOBAL`	Must be set on all symbols in the global `SymbolTable`. (`IS_GLOBAL()`)
`ATTR_CONST`	Applies to constant class members and constant `Variable`s. (`IS_CONST()`) Note that `ATTR_CONST` applies *only* to the element that is actually constant. For example, `i` in "`const int i;`" has its `ATTR_CONST` bit set. However, `p` in "`char * const p;`" is *not* constant! `p` can be changed, but what `p` points to is constant. Therefore `p` does not have its `ATTR_CONST` bit set. `p2` in "`const char *p2;`" *is* constant, and `ATTR_CONST` is set.
`ATTR_STATIC`	Applies to static class members and static `Variable`s. (`IS_STATIC()`)
`ATTR_VOLATILE`	Applies to local `Variable`s. (`IS_VOLATILE()`)
`ATTR_PUBLIC`	Applies to class members and inheritance relationships. (`IS_PUBLIC()`)
`ATTR_PRIVATE`	Applies to class members and inheritance relationships. (`IS_PRIVATE()`)
`ATTR_PROTECTED`	Applies to class members and inheritance relationships. (`IS_PROTECTED()`)

ATTR_VIRTUAL	Applies to class members and inheritance relationships. (IS_VIRTUAL())
ATTR_PURE	Applies only to virtual class member functions. (IS_PURE())
ATTR_ABSTRACT	Applies to classes that contain a pure virtual function. (IS_ABSTRACT())
ATTR_DECLARED_STRUCT	Applies to C++ classes that were declared as a C struct. (WAS_STRUCT())
ATTR_DECLARED_UNION	Applies to C++ classes that were declared as a C union. (WAS_UNION())
ATTR_DEFAULT	Applies to function parameters that have a default initializer. (HAS_DEFAULT())
ATTR_SPECIALIZATION	Applies to specialized class and function template instances. (IS_SPECIALIZATION())
ATTR_INLINED	Applies to functions and member functions that are declared inline. (IS_INLINED())
ATTR_COMPILE_ERRORS	Applies to Files that compiled with errors. (HAS_COMPILE_ERRORS())
ATTR_INSTANTIATED	Applies to functions and classes that are instances of a template. (IS_INSTANTIATED())
ATTR_SYNTHETIC	Applies to compiler generated functions, class members (such as automatically created class constructors or destructors), and variables. (IS_SYNTHETIC())

Scalar Types

`Scalar` types are described by members of the `ScalarType` enum. Legal `ScalarType` values are:

`SCALAR_CHAR`	Signed character type.
`SCALAR_UNSIGNED_CHAR`	Unsigned character type.
`SCALAR_WIDE_CHAR`	The NLS wide character type.
`SCALAR_SHORT`	Signed short integer type.
`SCALAR_UNSIGNED_SHORT`	Unsigned short integer type.
`SCALAR_INT`	Signed integer type.
`SCALAR_UNSIGNED_INT`	Unsigned integer type.
`SCALAR_FLOAT`	Floating point type.
`SCALAR_DOUBLE`	Double precision floating point type.
`SCALAR_LONGDOUBLE`	Long double precision floating point type.
`SCALAR_TEMPLARG`	Class type variable of a template.
`SCALAR_FUNCTYPE`	Type is a function.
`SCALAR_LOGICAL`	Fortran logical type.
`SCALAR_STRING`	Pascal string type.
`SCALAR_TEXT`	Pascal file type.
`SCALAR_LABEL`	Type code label.
`SCALAR_POINTER`	Fortran pointer type.
`SCALAR_VOID`	C and C++ void type.
`SCALAR_LONG`	Signed long integer type.
`SCALAR_UNSIGNED_LONG`	Unsigned long integer type.

Language Types

The `Language` type is used to determine the programming language contained in a `File`. `Language` is a bit vector defined as:

```
typedef unsigned long Language;
```

The legal `Language` values are:

`LANGUAGE_C`	C source file.
`LANGUAGE_F77`	FORTRAN 77 source file.
`LANGUAGE_PASCAL`	HP Pascal source file.
`LANGUAGE_COBOL`	HP COBOL source file.
`LANGUAGE_BASIC`	BASIC source file.
`LANGUAGE_ADA`	Ada source file.
`LANGUAGE_CPP`	C++ source file.
`LANGUAGE_UNKNOWN`	Any source file kind.

References

A reference is a tuple of line, column, length, and usage information. The line, column and length describe the token position in the file; the `Usage` describes the context in which the reference occurs. `Reference` is defined as follows:

```
typedef struct  { unsigned long length : 8;
                  unsigned long line : 24;
                  unsigned short column;
                  Usage use; } Reference;
```

Note that the `line` field causes a type mismatch if you attempt to print it using `cout`. Cast it to an integer (`cout << (int) ref.line`) to avoid this problem.

`Usage` is defined as a bit vector containing any combination of the following values:

`REF_DEFINITION`	Site at which the object is defined and the storage of the construct is determined. An object usually has only one `REF_DEFINITION` site.
`REF_DECLARATION`	Site at which an object is introduced into scope. A `REF_DEFINITION` is also a `REF_DECLARATION` site.
`REF_MODIFICATION`	Site at which the memory associated with the object is written.
`REF_CALL`	Site at which a function or procedure is called.
`REF_DEREF`	Site at which a pointer value is used to read or write memory.
`REF_ADDROF`	Site at which the address of the object is determined.
`REF_USE`	Site at which the object is read or used.
`REF_VIRTUALCALL`	Site at which a virtual function is being called via the dynamic binding mechanism.

Error Codes

The database interface routines define a global variable `DBError` to allow the application to diagnose any problems. This variable is primarily set during database open/close operations and during write operations; therefore, it is not generally used in rules.

`DBError` has two fields: one to record any system error (`errno`) and the other to record any error condition detected by the database.

The definition of `DBError` is:

```
typedef struct { unsigned short database;
                 unsigned short system; } DBErrorCode;
extern DBErrorCode DBError;
```

The database error codes are:

`DBERR_INCORRECT_DB_VERSION`	The file being opened as a database file is not a database file or is an obsolete version.
`DBERR_DATABASE_NOT_OPEN`	A database operation was attempted without a database file open.
`DBERR_DATABASE_ALREADY_OPEN`	The process attempted to open a database that it already had open.
`DBERR_MAPPING`	There was an error in mapping the database file.
`DBERR_FILETABLE_EXCEEDED`	There was an attempt to open more than the maximum number of databases (512) that may be simultaneously opened.
`DBERR_DBSIZE_EXCEEDED`	There was an attempt to create a database larger than the configured maximum size.
`DBERR_DBFILE_OPEN`	There was a problem in opening the database file.
`DBERR_DBFILE_RESIZE`	There was a system failure in an attempt to resize the database file.
`DBERR_DBFILE_STAT`	There was a system failure in an attempt to `stat` the database file.

DBERR_DBFILE_READ	There was a system failure in an attempt to read from the database file.
DBERR_LOCKFILE_OPEN	There was a system failure in an attempt to open the lock file.
DBERR_LOCK	There was a system failure in an attempt to lock the lock file.
DBERR_BAD_NAME	A bad (non string) name value was passed to a routine in the write interface.
DBERR_BAD_ATTRIBUTES	A bad attribute value was passed to a routine in the write interface.
DBERR_BAD_SCALAR	A bad ScalarType was passed to a routine in the write interface.
DBERR_BAD_HANDLE	A bad handle was passed to a routine in the write interface.

B

Iterators

Iterators are the mechanism used to loop through an arbitrary number of objects in the Static database. Because of some limitations in the C++ template mechanism, it's not possible to define general iterators using templates. Instead, the Static database interface simulates the template functionality using `#define`s.

"Iterators" in Chapter 4 gives a simple explanation of the use of iterators. That explanation is sufficient for most users. This section explains the mechanism behind iterators.

Standard Iterators

Iterators are defined as follows:

```
class Iterator {
  public:
    Iterator(long count, PerHandle *handles);
    Iterator();
    Iterator(const Iterator &iterator);
    ~Iterator();
    Iterator &operator=(Iterator &iterator);
    void add(long count, PerHandle *handles) const;
    DBboolean Open(PerHandle &handle) const;
    DBboolean Next(PerHandle &handle) const;
    DBboolean Done() const;
  protected:
    PerHandle IteratorHandle;
};

#define ITERATOR(Base) Base##Iterator

#define ITERATOR_IMPLEMENT(Base, Handle)              \
class ITERATOR(Base) : public Base, public Iterator {  \
  public:                                              \
    ITERATOR(Base)(long count, PerHandle *PH) :        \
      Base(), Iterator(count, PH) { }                  \
    ITERATOR(Base)():Base(),Iterator() { }             \
    ~ITERATOR(Base)() { }                              \
    DBboolean Open() {return Iterator::Open(Handle);}  \
    DBboolean Next() {return Iterator::Next(Handle);}  \
}
```

`Iterator` defines a base class upon which all iterators are implemented. The iterator for each object type in the database is defined by invocations of `ITERATOR_IMPLEMENT` in `DB_Read.h`.

`ITERATOR_IMPLEMENT(`*object*`)` defines the class `ITERATOR(`*object*`)` (which expands to *object*`Iterator`). This new class inherits from both `Iterator` and the *object* base class.

Static database code can then declare functions of type `ITERATOR(`*object*`)`. These functions return an iterator on objects of type *object*. Since the iterator class inherits both from `Iterator` and from *object*, the new iterator can be used to access both `Iterator` operations (to step through objects in the iteration list) and *object* operations and data (to manipulate objects in the list).

The methods `Open()` and `Next()` allow navigation through the array of iterators. For readability, the following macros are defined:

```
// Macros for constructing iterator loops.
#define ITERATE_BEGIN(sym)   if ((sym).Open()) do
#define ITERATE_END(sym)     while ((sym).Next());
```

Thus, to access all `RefLists` on symbol `sym`, you could write:

```
ITERATOR(RefList) rli = sym.RefLists();
ITERATE_BEGIN(rli)
{
  // manipulate Symbol rli:
  printf("Referenced in file %s\n",
          rli.FileIn().Name());
}
ITERATE_END(rli)
```

Attribute Iterators

A few objects use a specialized form of `Iterator` called `AttributeIterator`. `AttributeIterator`s are identical to `Iterators` in every way, except that each object in the iteration list includes an `Attribute` field.

`Attribute`, as defined in `DB_Common.h`, specifies what kind of symbol is defined by the current object. As an example, a symbol may be `ATTR_PUBLIC` or `ATTR_PRIVATE`.

Attribute iterators are defined in only two situations: in the Global Symbol Table and in `Class` objects. The attribute iterators in the Global Symbol Table give access to the `Attribute` value of all global symbols and on all symbols matching a set of criteria. The attributes returned by these iterators can include any attribute that can apply to a symbol. The attribute iterators in classes describe the nature of the inheritance relationship with base and inherited classes: public, private, protected, or virtual.

`AttributeIterator`s define two additional member functions, `GetIteratorAttribute()` and `SetIteratorAttribute()`, to access the `Attribute` field of the symbol or inheritance. Ordinarily you should use only the `GetIteratorAttribute()` member.

Other than the occasional use of the `GetIteratorAttribute()` and `SetIteratorAttribute()` accessor functions, you use `AttributeIterators` exactly the same as ordinary `Iterator`s.

Attributes are defined as follows:

```
class AttributeIterator : public Iterator {
  public:
    AttributeIterator(long count, PerHandle *handles, Attributes *attr);
    AttributeIterator();
    ~AttributeIterator();
    void add(long count, PerHandle *handles, Attributes *attr) const;
    DBboolean SetIteratorAttribute(Attributes) const;
    DBboolean GetIteratorAttribute(Attributes &attr) const;
};

#define ATTRIBUTE_ITERATOR(Base) Base##AttributeIterator

#define ATTRIBUTE_ITERATOR_IMPLEMENT(Base, Handle)                        \
class ATTRIBUTE_ITERATOR(Base) : public Base, public AttributeIterator {  \
  public:                                                                 \
    ATTRIBUTE_ITERATOR(Base)(long count, PerHandle *PH, Attributes *ATT)  \
        : Base(), AttributeIterator(count,PH,ATT) { }                     \
    ATTRIBUTE_ITERATOR(Base)() : Base(), AttributeIterator() { }          \
    ~ATTRIBUTE_ITERATOR(Base)() { }                                       \
    DBboolean Open() {return Iterator::Open(Handle);}                     \
    DBboolean Next() {return Iterator::Next(Handle);}                     \
}
```

Index

A

Aggregate objects, 4-6
`AllFunctions()`, 4-15
API, 4-1
`ArgumentCount()`, 4-18, 4-27
`ArgumentOf()`, 4-46
`ATTR_`*attrtype*, A-3
`Attrib()`, 4-15, 4-20, 4-36, 4-39
`Attribute`, 4-6, A-3
Attribute iterators, 4-11, B-4

B

`BaseClasses()`, 4-15
`BaseTo`*type*`()`, 4-31
`BeginLine()`, 4-13
`Block`, 4-4, 4-13
`BlockFile()`, 4-13
`BlockFunctions()`, 4-13
`BlockLabels()`, 4-13
`BlockTags()`, 4-13
`BlockTypedefs()`, 4-13
`BlockVariables()`, 4-13
Boldface font, vi
Breakpoints, 5-12

C

`check()`, 3-4, 3-11
`check_table_entry()`, 3-11
`Class`, 4-4, 4-14
Class inheritance, B-4
`ClassTag()`, 4-15
`ClassTemplate`, 4-4, 4-18
`ClassTemplateType()`, 4-45
`ClassType()`, 4-45
`CompileDir()`, 4-22
`CompileHost()`, 4-22
`CompileName()`, 4-22
`CompileOptions()`, 4-22
Compiling rules, 5-5
Computer font, vi
`Contains()`, 4-42

D

Database, 4-1
  opening and closing, 4-8
  transactions, 4-9
`DataMember`, 4-4, 4-19
`DataMembers()`, 4-15, 4-36
`DBERR_`*errtype*, A-8
`DBError`, A-8
Debugging rules, 5-10, 5-13
`DeclarationSite()`, 4-39
`DefinitionSite()`, 4-24, 4-39
`DerivedClasses()`, 4-15
Design guidelines, 5-1
Designing rules, 5-4
Documenting, 5-14
`dthelptag`, 5-14
`DtorMatchCtor` rule, 2-10

E

Ellipses, vi
`EnclosingBlock()`, 4-39
`EnclosingClass()`, 4-39

`EnclosingFile()`, 4-39
`EnclosingFunction()`, 4-22, 4-42
`EndLine()`, 4-13
`EndTransaction()`, 4-42
`Enum`, 4-4, 4-20
`EnumMember`, 4-4, 4-21
`EnumMembers()`, 4-20
`EnumTag()`, 4-20
`EnumType()`, 4-45
`errorMess()`, 3-4, 3-11
Example rules, 3-7, 4-50–58
`ExpandedFrom()`, 4-15, 4-24
`EXTERNREF`, 5-15

F

`File`, 4-4, 4-22
`FileIn()`, 4-32
`FileName()`, 4-42
`Files()`, 4-42
`FileType()`, 4-22
`Find()`, 4-42
`FindDataMember()`, 4-15, 4-36
`FindEnumMember()`, 4-20
`FindFunctionMember()`, 4-15
`FindFunctionTemplate()`, 4-18
Font usage, vi
`Friends()`, 4-15
`Function`, 4-4, 4-24
`FunctionBlock()`, 4-24
`FunctionInstantiations()`, 4-27
`FunctionMember`, 4-4, 4-26
`FunctionMemberInstantiations()`, 4-27
`FunctionMembers()`, 4-15
`Functions()`, 4-22
`FunctionTemplate`, 4-4, 4-27
`FunctionTemplateMembers()`, 4-18
`FunctionTemplates()`, 4-22, 4-42

G

`GetIteratorAttribute()`, B-4
`GlobalFunctions()`, 4-42
`GlobalModules()`, 4-42
`GlobalSymbols()`, 4-42
`GlobalTags()`, 4-42
`GlobalTypedefs()`, 4-42
`GlobalVariables()`, 4-42
Groups, 3-6, 4-53, 5-1, 5-7, 5-11
  index, 5-9
Guidelines, 5-1

H

`Handle()`, 4-31, 4-42
`HAS_`*attrtype*, A-3

I

Implementing rules, 5-3
`IncludedBy()`, 4-22
`Includes()`, 4-22
Incomplete objects, 4-6
Index of rules, 5-9
`Instantiations()`, 4-18
`IS_`*attrtype*, A-3
`IsHandleNull()`, 4-31
Italic font, vi
`ITERATE_BEGIN`, 4-10
`ITERATE_END`, 4-10
Iterators, 4-10, B-1

K

Keycaps, vi
`Kind()`, 4-31
`KIND_`*kind*, A-2
`kindMask()`, 3-4, 3-11

L

`Label`, 4-4, 4-28
`LANG_`*langtype*, A-6
`langMask()`, 3-4, 3-11
`Language`, 4-6, A-6

`LocalTags()`, 4-42

M

`Macro`, 4-4, 4-29
`Macros()`, 4-22, 4-42
`Makefile`, 4-53
`MemberCount()`, 4-15, 4-20, 4-36
`MemberFunction()`, 4-24
`MemberOf()`, 4-19, 4-21, 4-26, 4-27
`ModifiedTime()`, 4-22, 4-42
`Modules()`, 4-22

N

`name()`, 3-4, 3-11
`Name()`, 4-39
`NameConventions` rule, 2-3
`names()`, 3-4, 3-11
`NestedClasses()`, 4-15
`NestedEnums()`, 4-15
`NestedTypedefs()`, 4-15

O

Object types, 4-3
Online examples, 4-53
On-line help, 5-14
  external links, 5-15
`operator[]`, 4-32

P

`Parameter`, 4-4, 4-30
`ParameterCount()`, 4-24
`Parameters()`, 4-24
`ParameterTypeInfo()`, 4-24, 4-30
`PerBase`, 4-3, 4-6, 4-31
`PerHandle`, 4-6
`PerKind`, 4-6, A-2
Personal rule tables, 2-2
`ProhibDefines` rule, 2-8
`ProhibIdent` rule, 2-6

Q

Qualifiers, 4-7

R

`Reference`, 4-7, A-7
`ReferenceCount()`, 4-32
`ReferenceSite()`, 4-39
`RefList`, 4-4, 4-32
`RefLists()`, 4-22, 4-39
`REF_`*reftype*, A-7
Regular expression, 4-44
`report()`, 3-4
`Rule()`, 3-4
`Rule` class, 3-3
`RULE_DEBUG`, 5-13
Rule engine, 3-2
Rule groups, 3-6, 4-53, 5-1, 5-7, 5-11
  index, 5-9
Rules
  compiling, 5-5
  debugging, 5-10, 5-13
  designing, 5-4
  implementing, 5-3
  index, 5-9
  testing, 5-5
Rule tables, 2-1
  `DtorMatchCtor`, 2-10
  `NameConventions`, 2-3
  `ProhibDefines`, 2-8
  `ProhibIdent`, 2-6
  scope, 2-2
`RuleWithTable()`, 3-11
`RuleWithTable` class, 3-10

S

`SA_SHLIB_TEST`, 5-13
`Scalar`, 4-4, 4-35, A-5
`SCALAR_`*scalartype*, A-5
`ScalarType`, 4-6
Scope, 2-2
`Scope()`, 4-28, 4-49

`SetIteratorAttribute()`, B-4
Setting breakpoints, 5-12
`softcheck`, 5-10
`SourcePosition`, 4-7
`StartTransaction()`, 4-42
Static API, 4-1
Static database, 4-1
`Struct`, 4-4, 4-36
`StructTag()`, 4-36
`StructType()`, 4-45
`Symbol`, 4-3, 4-38
`SymbolFor()`, 4-32
`SymbolsAtLocation()`, 4-42
`SymbolTable`, 4-3, 4-41
`SymbolTo`*type*`()`, 4-39

T

Table rules, 2-1
  `DtorMatchCtor`, 2-10
  `NameConventions`, 2-3
  `ProhibDefines`, 2-8
  `ProhibIdent`, 2-6
`Tag`, 4-4, 4-45
`Tags()`, 4-22
`TemplateArgument`, 4-4, 4-46
`TemplateArguments()`, 4-18, 4-27
Testing rules, 5-5
Tracing rules, 5-13
Transactions, 4-9
`Type()`, 4-35
`Typedef`, 4-5, 4-47
`Typedefs()`, 4-22
`TypedSymbol`, 4-3, 4-48
`TypeKind()`, 4-45
Type qualifiers, 4-7
Typewriter font, vi

U

`Usage`, 4-7

V

`Value()`, 4-21
`Variable`, 4-5, 4-49
`Variables()`, 4-22
`violation()`, 3-4, 3-11, 5-15

W

`WAS_`*attrtype*, A-3